CMSC 23010 Homework 3a & 3b Design
Ruolin Zheng
May 4, 2019

3a Design

Modules

For correctness testing in both 3a and 3b, we need a LINKEDLIST object. Each node stores a `counter` (3a) or a `checksum` (3b). It supports only one operation `append` which we'll create unit tests for.

Correctness Testing Plan

We want to ensure that there are no skips, no duplicates, and no out-of-order entries as we increment the Counter. We test for correctness in a separate, modified program which outputs to file. All threads can access a shared LINKEDLIST object. Each thread acquires the lock, increments the Counter, writes the Counter value to the LINKEDLIST and then releases the lock (compared to the normal routine of `lock(), incr(), unlock()`.) We then output the data in the LINKEDLIST to a text file and compare with the standard text file containing entries 1 to *BIG*.

My Test for Fairness

This test investigates how unfair the TASLOCK can get and whether increased contention leads to more unfairness. We need a separate, modified program in which each thread keeps a local count of how many times it has incremented the counter. A thread will write the count to its corresponding slot in a shared array of length n before it exits. For $n \in \{2, 4, 8, 14\}$, for $L \in \{$TASLOCK, MUTEX, ALOCK$\}$ (ALOCK and CLHLOCK are both FCFS, so either will suffice,) we run PARALLEL and evaluate the fairness (or unfairness.) We measure the unfairness using the formula for variance, taking BIG/n as the expected mean, $\frac{\sum_{i=1}^{n}(count_i - BIG/n)^2}{n-1}$. The greater the value, the more unfair the lock is. We run 10 trials to obtain the median unfairness for each parameter combination. We plot the unfairness for each L on the Y-axis vs. n on the X-axis.

Performance Hypotheses

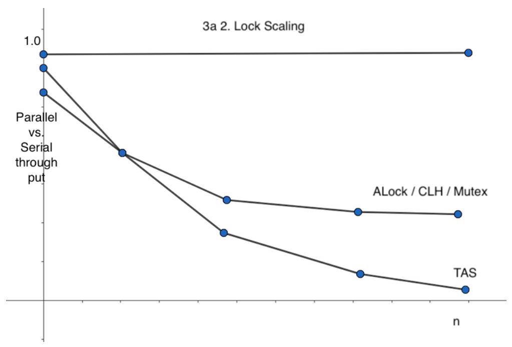
Note: We don't know the detailed implementation of MUTEX *and will assume it is generic and fair even in contention. In all tests below, the number of threads is smaller than the number of cores.*

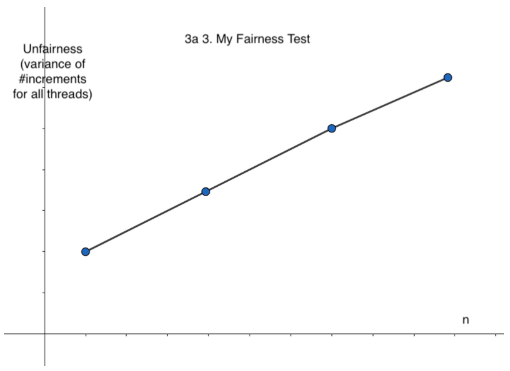
1. Idle Lock Overhead

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. Lock Scaling

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





3. My Test for Fairness
We expect ALOCK to have about 0 variance since it is FCFS; the same holds for MUTEX. I hypothesize that TASLOCK has consistently non-zero variance which increases as n and contention increase. This is possible because when the bus traffic is heavy, if a thread has good access to the bus, it does not have to invalidate its cache often, and may release and re-acquire the lock indefinitely often. However, I do recognize that fairness is heavily dependent on OS scheduling and will accept this if the actual data shows no apparent positive relationship between n and fairness.

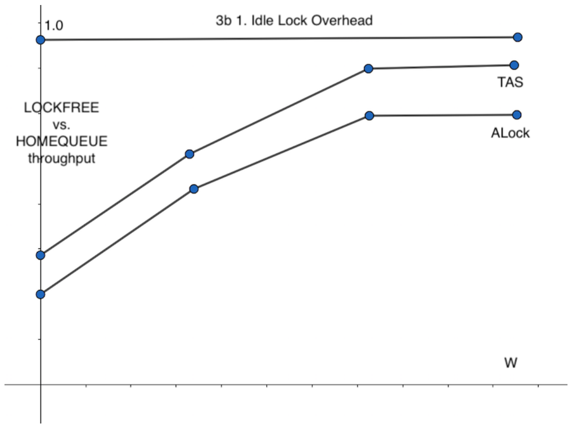
3b Design

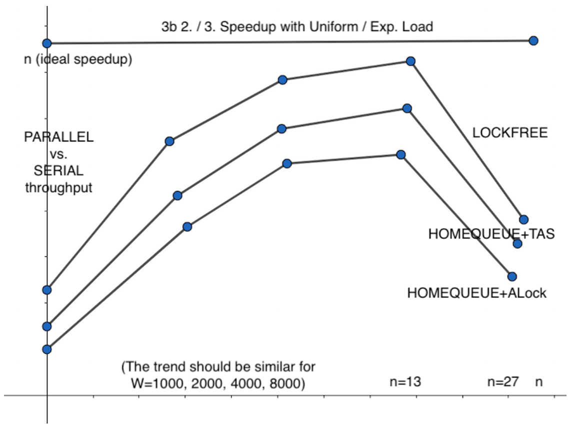
My Awesome Strategy: Select the Queue the Dispatcher is Spinning on
I hypothesize that the main reason UNIFORM and EXPONENTIAL are slower than CONSTANT is that some worker constantly gets heavy-load packets and always has a full queue. The dispatcher has to spin on this full queue and may starve the other workers with empty queues. Therefore, my strategy would be, when a dispatcher starts to spin on a full queue, it makes this queue accessible for all workers by broadcasting `full_queue_pointer`. The worker who is scheduled to run will try to grab the lock with `trylock` implemented in 3a, and work on that full queue. Once the queue is no longer full, the worker sets `full_queue_pointer` to `NULL`. As long as `full_queue_pointer` is `NULL`, each worker works on its designated queue.
- **A potential problem** with this implementation is that updates to `full_queue_pointer` may force each worker to invalidate their cache line. So we may need to make sure these calls do not happen too frequently.

Correctness Testing Plan for Modules
- We've already tested the correctness for the LAMPORTQUEUE and the DISPATCHER in HW2, and their behavior does not change in this assignment.
- For LOCK, we test for correctness as described above in 3a.
- For WORKER, we want to ensure that all checksums have been computed based on the SERIAL version which we assume to be correct. We test for correctness in a separate, modified program which accepts T, the number of packets per source, instead of M. The program will write the resultant checksums to a file. Because the AWESOME strategy allows a worker to pick a queue other than its designated one, we use a LINKEDLIST shared by all threads instead of the matrix in HW2 to store the checksums. For the same parameters $n, W, T, seed$, we run PARALLEL and SERIAL respectively, appending each checksum to the LINKEDLIST. Because the worker may process packets out of order, we first sort entries from the two output files and then compare.

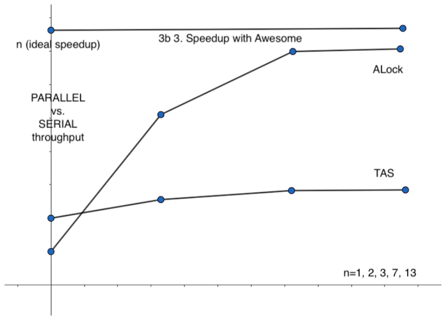
Performance Hypotheses
Note: I will use TASLOCK and ALOCK for these tests. I will exchange ALOCK for CLHLOCK if the latter proves more efficient in actual tests. The throughput ratio of HOMEQUEUE to LOCKFREE should be consistently lower than 1.0 because HOMEQUEUE incurs unnecessary overhead in acquiring the uncontended lock.
1. Idle Lock Overhead
- Because there is no contention, TASLOCK should have a higher speedup than ALOCK since the latter incurs overhead consistent to what we observe in the Counter Tests.
- The overhead of the locks decreases **Worker Rate** and thus can be expressed as **Worker Rate** in HW2

minus our measured overhead for TASLOCK and ALOCK. In accordance with the Counter Tests, the Worker Rate for TASLOCK should be greater than ALOCK because the former has less overhead when there is no contention.





2. Speedup with Uniform Load
- We have three curves for each W, LOCKFREE, HOMEQUEUE+TASLOCK, and HOMEQUEUE+ALOCK. LOCKFREE is the same as PARALLEL in HW2, except that we use throughput ratio instead of runtime ratio. Therefore, the LOCKFREE curve should slope upward for $n \in \{1, 2, 3, 7, 13\}$, but decreases sharply at $n = 27$ when the number of threads exceeds the number of cores. This convex shape should hold for all three curves.
- Our ideal throughput speedup ratio is n-fold, and we expect LOCKFREE to be the closest to ideal. Both HOMEQUEUE curves should lie below LOCKFREE. Because there is no contention, consistent with **1. Idle Lock Overhead**, TASLOCK should scale at least as well as ALOCK, and perhaps better.
- As in HW2, the expected values of a uniform distribution tend to those from a constant one. Therefore, there shouldn't be too much variance in trend across the four W graphs.
3. Speedup with Exponential Load
- As in HW2, the variability increases as W increases. Therefore, larger W values should have curves approaching the ideal n-fold speedup.
- For each W, consistent with **2. Uniform**, all curves are convex. LOCKFREE lies above HOMEQUEUE with either lock. We only use locks but not load balancing in this test, so in contention-free situation like HOMEQUEUE, TASLOCK scales at least as well as ALOCK, and perhaps better.
4. Speedup with Awesome
- We attempt to achieve load balancing with AWESOME, but introduce contention and thus render TASLOCK inefficient. We run PARALLEL and SERIAL on uniform packets for
$n \in \{1, 2, 3, 7, 13\}, W \in \{1000, 2000, 4000, 8000\}, L \in \mathit{Locks}$. For each W, we plot PARALLEL vs. SERIAL for each L on the Y-axis and $n + 1$ on the X-axis.
- Consistent with **2. Lock Scaling** in 3a: for smaller n, TASLOCK has a higher speedup than ALOCK. As n increases, ALOCK scales better. The TASLOCK curve starts at a higher speedup than ALOCK but does not increase much and may even decrease. ALOCK starts low, but increases steadily and eventually tends to our ideal ratio n. Because we use uniform packets, this trend should be approximately the same for all four W.


n (ideal speedup)
3b 3. Speedup with Awesome
ALock
PARALLEL vs. SERIAL throughput
TAS
n=1, 2, 3, 7, 13