Homework 3 Design

Michael Tang

due May 7, 2019

1 Part a: Lock Implementations

1.1 Input and Initialization Design

The only required user inputs here are for the number of threads and the type of lock to use. Thus, main() will parse:

- -n for number of threads, which if 0 will run the tests in serial mode, while if ≥ 1 will run in parallel, and
- -L for type of lock, 0 to 3 for TAS, Mutex, Anderson, and CLH in order. -L will only be required if -n ¿ 0.
- -o for output path, default stdout if nothing is given.

Parameters will then be passed to an operator function that will either start running the counter for serial mode, or go through thread creation. Each thread will be running a worker function that keeps calling TryLock until a globally shared flag is set signalling the end of the counter.

1.2 Lock Design

TAS, Anderson, CLH will be implemented as specified in the textbook. Since objects the way the textbook implements them are not supported by C, for each lock the state will be held in a struct designed for that lock, and there will be specific lock() and unlock() calls for each lock that only accept that lock's struct.

The wrapper TryLock function will, given a lock object, check if the lock is already in use first. If the lock is not in use the function will call the corresponding lock() method to grab that lock. If the lock is already in use, the function will exit to allow the thread to do other things. In the scope of Part A, "other things" just entails another attempt at TryLock.

1.3 Testing

1.3.1 The Tests

Counter Test
The test will be run as specified. For now $BIG = 2323$, but the suitability of that number will be tested before using SLURM, and adjusted as needed. I would surmise that this is a big enough number for variances to be low, so if anything it would need to be adjusted downwards.

My Test: Fairness
I would like to examine how usage of each lock affects the distribution of work across threads. The current Counter Test can be slightly adjusted by having each thread mark when it has been able to access the lock and increment the counter. This seems best done by having an array of integer counters for every thread, where the thread increments the shared counter as well as its own counter when it accesses the critical section. This way at the end we can see exactly how many times each thread accessed the critical section while reducing the amount of work in the critical section itself, preserving high contention.

1.3.2 Correctness

TASLock
The mutex correctness of TASLock has already been reasoned about, and the design itself has no perceivable intermediate invariants. Therefore we can simply check if the implementation works by running the counter performance test and checking if the counter grabbed each time is unique. If it is not, then two threads have entered the critical section at the same time and there is something wrong with the code.
Anderson
Besides mutex, this design is also reasoned to be starvation-free and have first-come-first-served fairness. So we can check mutex correctness with the same examination of the counter test as before, while also examining to check that every thread got to increment the counter for starvation-freeness. For FCFS, a possible test is to print the state of the queue upon each lock() and unlock() to make sure that it is going in that order.
CLH
The same as Anderson, except an entire queue need not be printed and only the current node and its predecessor. A picture of the entire "queue"'s state will still be visible.

1.3.3 Performance Hypotheses

For these, I guess that pthread_mutex has a performance about equivalent to the TTAS lock, so equivalent to TAS in overhead and between TAS and ALock in terms of speedup and fairness.
1. Idle Lock Overhead
There would be some overhead time of TASLock over Serial, the amount of which seems to depend on the number we are counting to. Since here ALock would be performing fundamentally the same operation as TASLock, reading one memory space to find that it is True, the overhead should also be the same. CLHLock should have slightly more overhead because a new node is still actively created each time the lock is acquired.
2. Lock Scaling
At lower n, with probably less contention, the three lock types should have about the same below-1 speedup over serial counting. The higher n gets, TASLock's speedup should go down even further as it is most affected by the additional contention, while CLHLock and ALock should experience much less reduction. ALock should also have slightly higher speedup than CLHLock at all levels since it has no node creation overhead, but still an array creation overhead.
3. My Fairness Test
Like with Experiment 2, n will be the independent variable being adjusted. At these different levels the variance of individual thread work from the expected work BIG/n will be plotted. This is accurately able to portray how equitably work is being distributed. As n gets higher the work variance for TASLock should increase much faster than for ALock and CLHLock, with no significant difference between the

latter two. The latter two should still experience an increase in variance due to more division of work.

2 Part b: Load Balancing

2.1 Input and Initialization Design

Input options are:

- -n, nsources (workers).
- -T, number of packets per thread, **replacing experiment runtime** because I am not sure how setting the runtime of the experiment would allow us to compare performance times for different modes later on.
- -D, queue depth, default 8.
- -W, mean work per packet.
- -s, work distribution mode, between 0: uniform and 1: exponential.
- -X, seed number for packet generator.
- -L, lock algorithm type, like in Part A.
- -S, queue picking strategy, 0 for serial mode, 1-3 for LockFree, HomeQueue, and Awesome in order.

Then operation flow will be like in Homework 2, going into an "operator" function that creates threads and initializes dispatchers and workers. In the "worker" functions that the threads operate on, what happens will differ depending on queue-picking strategy, detailed below.

2.2 Queue-Picking Design

For LockFree, the worker function will be like in HW2. For HomeQueue, instead of directly calling deq() the worker will keep attempting to call TryLock.
The Awesome design will be a rotating worker design, where each worker will be assigned a queue initially but if their queue is empty, the worker will iterate over all the other queues to see if any other queue has packets and is not being worked on. This will more thoroughly distribute the work available and reducing idling by worker threads.

2.3 Testing

2.3.1 Correctness

The correctness of the Lamport Queue has already been tested in HW2, and is not needed even if multiple workers take turns on the same queue in Awesome mode because that mode utilizes a lock on the entire dequeueing process. The correctness of the lock is also tested in Part A. Therefore, there are no additional correctness tests needed as all invariants are satisfied.

2.3.2 Performance Hypotheses

1. Idle Lock Overhead

Consistent with Part A overhead testing, I expect HomeQueue to have speedup < 1 relative to LockFree, with TASLock and ALock being faster than CLHLock. As W increases, I expect TASLock and ALock to have about the same decrease in speedup while CLHLock decreases faster because of additional node generation overhead scaling with more work.

2. Speedup with Uniform Load

ParallelPacket should experience the same speedup under LockFree as it did in HW2. In HomeQueue there should be less speedup, the same proportion to its LockFree time as in Experiment 1, and the same orderings of proportions for the lock types.

3. Speedup with Exponential Load

It seems like the results will echo that of Experiment 2, with the parallel speedup over serial being like that for exponential distribution in HW2.

4. Speedup with Awesome

With the Awesome design, less thread idling should mean that speedup is higher than the HomeQueue and LockFree designs, and much closer to the ideal speedup $\frac{WT}{n}$, where T is the total number of packets processed.