CMSC 23010: Homework 3b Final Writeup and Theory

Will Thomas

December 12, 2022

Introduction

The goal of the following documentation is to discuss design changes, implementation challenges, correctness outcomes, and performance outcomes for a program that attempts to simulate a firewall that processes incoming packets.

Design

Effectively, this program was just combining hw2 and hw3a, and then writing the code for the awesome protocol. As such, I will go over the big picture of what the code does, and then explain how I implemented awesome. More detailed explanations of my code are in homeworks 2 and 3a.

Big Overview of Code

The program takes nine inputs. They are as follows: t, or the number of packets per source; n, or the number of sources; w, or the mean work per packet; u/e, or a flag representing uniform (1) or exponential packets (0); exp_number, or the experiment number; parallel/serial, or a flag representing parallel (1) or serial (0); queue depth (always 8); locktype, which is 0 for no lock, 1 for TTAS, and 2 for mutex; and finally strategy, which, assuming the parallel/serial flag holds 1, is 0 for lockfree, 1 for homequeue, and 2 for awesome.

The main function handles all of the inputs and makes sure that nothing bad happens whenever the inputs are incorrect. It then handles the running of either a serial or concurrent version of the program. Seeing as serial is trivial, we can dive more into the concurrent branch. Most of the setup is done within the dispatch function. This is where the workers are initialized and launched from.

poly_locker and poly_unlocker

These two functions are the only additional functions for this assignment. Essentially, they allow me to use a union lock_t type consisting of mutex and ttas. Whenever I need to lock or unlock, they make it so that I do not need separate conditionals.

Awesome

As stated in my design document, the idea behind my Awesome protocol is to shorten the amount of time that the dispatcher will be stalled when trying to enqueue a package into a full queue. The big-picture idea was that there would be some sort of signal that would be set by the dispatch thread whenever it was stalled trying to enqueue a package that would tell worker threads with empty queues which other queue to try and get a package from.

When this was actually implemented, it became considerably more complicated. The worker thread struct has extra fields corresponding to auxiliary queue, auxiliary lock, a help needed flag, a currently helping flag, a decrement pointer, and a t_update value. Essentially, what happens is that whenever the dispatch thread

attempts to enqueue a package and is unable to do so, it sets these shared auxiliary lock and queue pointers to the lock and queue of the worker that has a full queue. Then, an atomic operation sets the help needed flag to 1. Worker threads who have empty queues will see this and set their own helping flags. Meanwhile on the dispatch side, once it is able to enqueue a package it will set the help needed flag back to 0, but it will wait until all worker threads that came to help have also set their helping flags to 0.

Testing

Correctness of everything except for awesome follows directly from homeworks 2 and 3a. I checked this quickly by creating two sink text files that I piped computed checksums into for serial, lockfree, and homequeue. Then, I wrote a quick python script to check that all checksums were present according to the output from serial. The same protocol worked for awesome; regardless of its purported efficiency, every checksum is still computed.

Performance

To test performance, I uploaded my Makefile, my main.c file, experiments.py which runs the required experiments, the needed utils, and two output directories needed for the experiments to my uchicago linux machine. Instead of choosing a time limit M, I decided to make it so that for every experiment 1000 checksums had to be computed by each worker.
NOTE: All UNIFORM experiments were run 6 times, and all EXPONENTIAL experiments were run 11 times.

Idle Lock Overhead

I predicted that HOMEQUEUE would run 17% as fast as LOCKFREE when using TTAS and about 14% as fast as LOCKFREE when using Mutex. This was clearly not the case. In fact, we had considerable speedup peaking for both TTAS and Mutex at $W = 100$. This is highly irregular, especially considering the initial (expected) slowdown to roughly 85% the speed of LockFree at $W = 50$. This leads me to believe

that there is some sort of error in this data, but I am not sure how this could have happened. Comfortingly, after $W = 200$ there is once again a slowdown, and at $W = 800$ both TTAS and Mutex are at roughly 90% the speed of LOCKFREE. For future reference, it may be wise to have a larger number of packets to be computed per thread, as I believe that may help deal with the unexplained spike in speedup. Beyond that, it is unclear how the HOMEQUEUE methods outperformed LOCKFREE under any conditions due to the extra overhead involved in initializing as well as repeatedly obtaining and releasing a lock. It is also intuitive that the worker rate would get slimmer as the amount of work per packet grows.

Speedup with Uniform Load

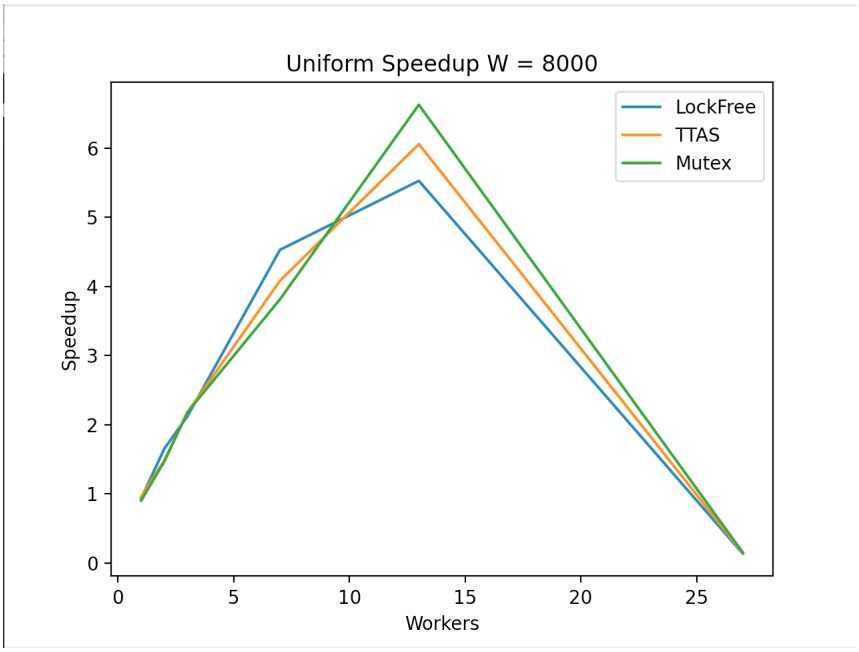
I hypothesized that as the value of W grows, the comparable speedup for a certain number of threads will increase monotonically for LockFree, TTAS, and Mutex. This clearly is the case, and it makes a lot of sense why it would be that way. If each packet takes longer to compute, there is simply more benefit to parallelism. We see in the $W = 8000$ example that the Mutex HOMEQUEUE implementation has a 6 time speedup over serial.

I also hypothesized that LockFree would remain the fastest. This, interestingly enough, is not the case. In fact, lockfree is often the slowest. When run with $W = 1000$ and $n = 15$, we see that lockfree is almost half as fast as Mutex. This is unexpected given the fact that the methods with locks should only add overhead.

Also, as is to be expected, once the number of threads becomes larger than the number of cores, performance plummets. We saw this in HW2.


Uniform Speedup W = 2000
LockFree
TTAS
Mutex
Speedup
Workers
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
5
10
15
20
25


Uniform Speedup W = 4000
LockFree
TTAS
Mutex
Speedup
Workers
0
1
2
3
4
0
5
10
15
20
25



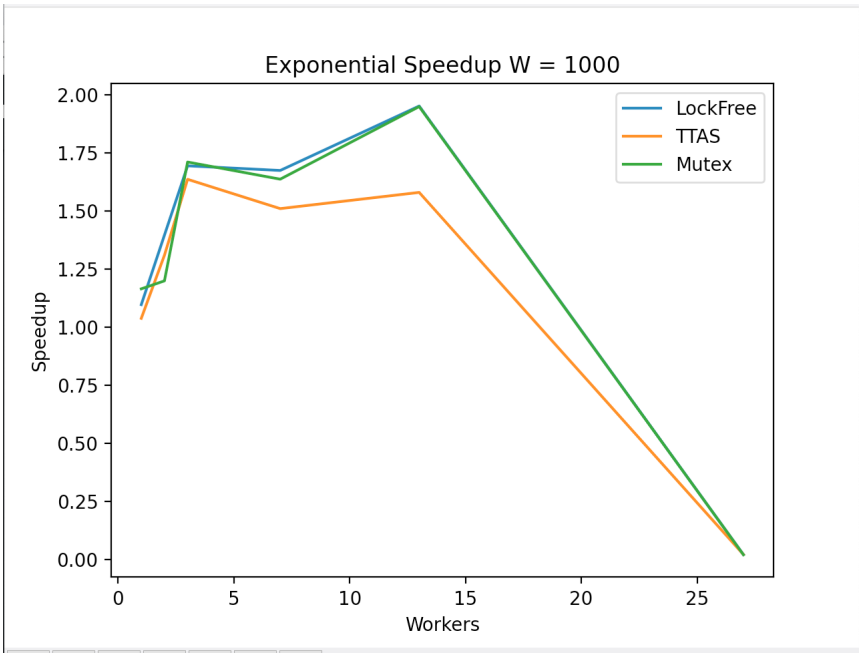
Speedup with Exponential Load

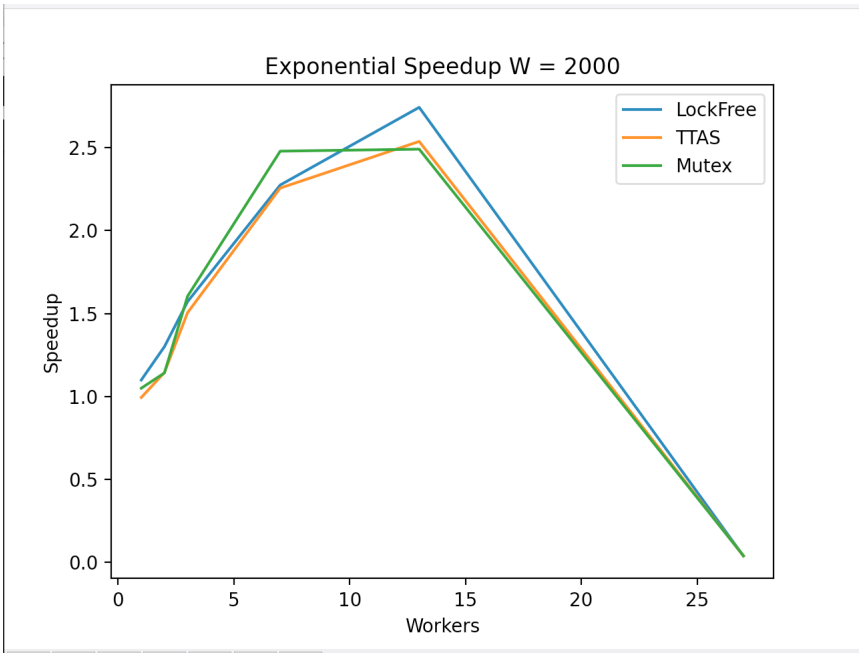
I hypothesized that exponential load speedup would yield similar results to uniform load speedup except with a smaller spread to the data. My reasoning was that the work for a single packet has the capability of being quite large, meaning that the parallel overhead in LOCKFREE and the lock overhead in HOMEQUEUE will become increasingly irrelevant in comparison.

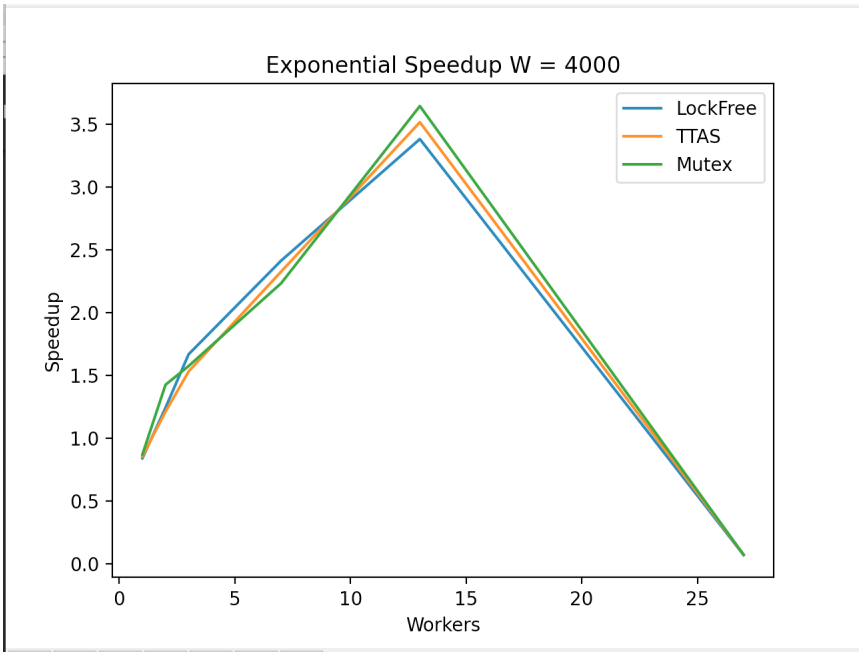
Looking at the data, this seems to be the case. In the graph of Exponential Speedup with $W = 4000$, we can see that the speedup of all three versions over serial are within .5 of eachother.

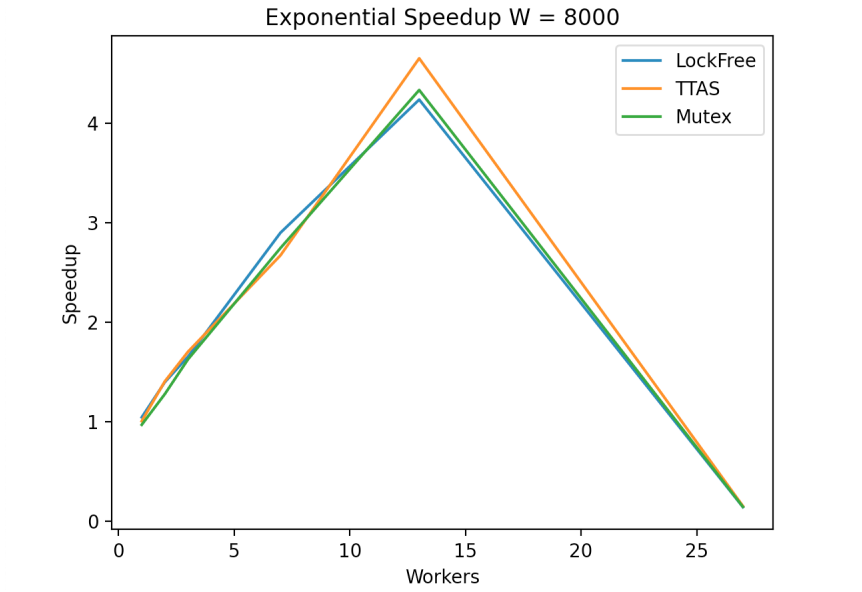
What is also interesting is that LockFree seems to be operating more as expected with Exponential Load than it did with Uniform Load. Seeing as it does not need to deal with the extra overhead from locks, it would make sense for it to have the most substantial speedup.

The common theme of increase in mean work's correlation with increase in benefit from parallelism remains, but because the exponential packets can be quite large, there is a depressing effect on the overall speedup. This can be seen by comparing $W = 8000$ for exponential and uniform packets. Here, we only get a 5x speedup in the best case (TTAS), but with uniform packets, we got a 7x speedup (with Mutex).


Exponential Speedup W = 1000
LockFree
TTAS
Mutex
Speedup
Workers
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
5
10
15
20
25

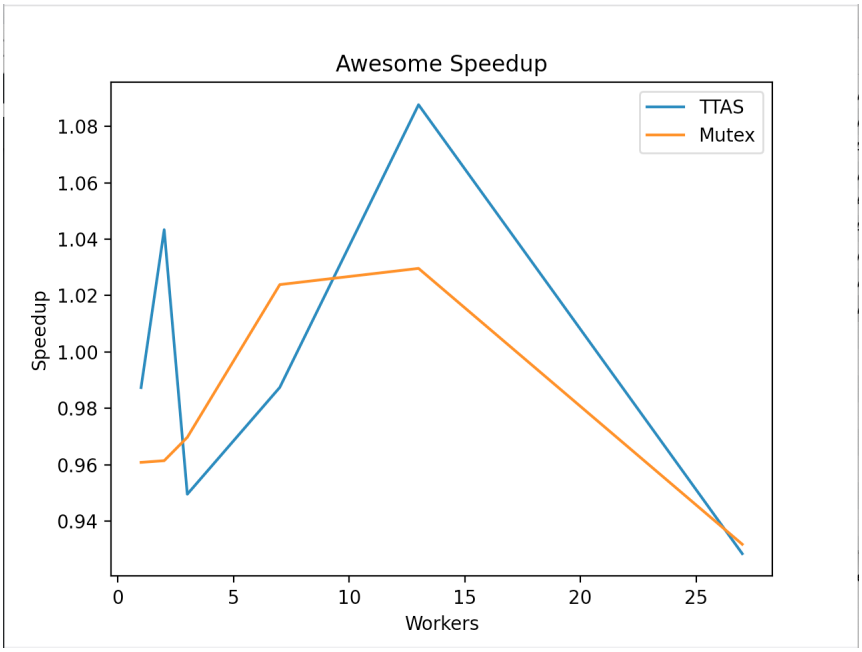

Exponential Speedup W = 2000
LockFree
TTAS
Mutex
Speedup
Workers
0.0
0.5
1.0
1.5
2.0
2.5
0
5
10
15
20
25





Speedup with Awesome

Truthfully, my Awesome method does not work as intended. I predicted that it would "handily outperform the other strategies in cases of Exponential Packets with a large mean work time," although it seems to just essentially be a graph of what Exponential Speedup would have looked like if it was in reference to LockFree and not the serial implementation. As expected, we have a large dropoff in performance once the number of threads gets too large. I think that there are numerous reasons for why my awesome method does not outperform the other methods. I think that a large amount of atomic operations has some effect, but I think that, rather unfortunately, the core of the issue is with the implementation.


Awesome Speedup
TTAS
Mutex
Speedup
1.08
1.06
1.04
1.02
1.00
0.98
0.96
0.94
0
5
10
15
20
25
Workers