Disclaimer

Any code in this assignment is using all recommended optimizations in class including O3.

Introduction

As a precursor to this project, we are currently programming a firewall since homework 2. The object of the firewall is to compute checksums for a series of packets coming out of a set of packet sources. The data structures we are using to store these packets are called lamport queues. They are lock-free queues, that allow two threads to enqueue from one side and dequeue from another. As we saw in homework 2, however, we are seeing good speedup with our strategy when we have constant packets, but when we get uniform and exponential packets, what happens is that our worker threads have an uneven distribution of work. As such, it makes sense that worker threads should be able to pull work from other queues if they have nothing to do. To do that, we need to use locks, and that was what hw3a was about programming. Now we finally have our locks programmed and ready to go; we are going to load balance our firewall!

Modules

The modules I am using for this assignment are exactly the same as the plan. We have a serial module where I am programming the serial version of the assignment. We also have a parallel module where I am putting in the lock free implementation, both the lock implementations, as well as my awesome implementation. Lastly, we also have a main module that accepts command line arguments that I can pass into the different implementations. I was actually pretty pleased with my design document as far as the modules go, as I was able to adhere well to them.

Deviances

There were three deviances I made from my design document worth talking about. One was the choice of the two locks I will be using for this assignment, the second was my awesome load balancing strategy, and the last one was my strategy for measuring time.

My design document plan was to use the anderson array lock and the TTAS lock, as they were the most successful locks, as far as performance goes. I, however, changed my mind to use both the TTAS and the pthread_mutex lock. The TTAS lock was used because my implementation for it produced the best results, and the pthread_mutex was used because it is the library default, so I presumed developers have put some optimizations to make it more scalable. As such, I used both the TTAS lock and the pthread_mutex lock in this assignment.

My second deviation was my awesome strategy. I wanted to start off simple first, and my awesome strategy incurred too much overhead so I shifted it. Basically what this new approach does is follow a greedy approach where if a worker cannot pull a packet off its default queue, it shifts to another random queue, and tries to pull a packet of it. After this, it shifts back to its default queue, and the process starts again. This approach is not only simple to code, but incurs very little overhead, and as such, was very promising for giving me good overhead.

Lastly, I used a new strategy for measuring time. Basically the original idea was to use the stopwatch given to us in utilities, and periodically stop time to see if it was over the inputted time, and if it was over, I would return from the function. This approach, however, was a little frustrating to do when I actually wrote up all the code for initializing the locks and whatnot, as my code felt a little too cluttered. Basically now I used a strategy where I spawned a pthread, put it to sleep for the inputted time, and soon as it wakes up, it edits a shared flag and turns the value on, so in my main function, every time I loop, I just have to check whether the flag is on, and if it is, I can return from the function.

Running the Code

Here are detailed instructions regarding how to run the code. First navigate to the hw3b folder, and run the make command. Then, we have the flock executable, and we need to run it. My code is not written to have any tolerance to errors. If the instructions here are not followed, the code simply will not produce the desired outcome. Then run /flock -M integer specifying time -N integer specifying number of sources -W specifying expected work per packet -U specifying whether packets are uniform or exponential -E specifying experiment number, -L specifying whether its serial, TTAS lock, or pthread_lock and -S representing lock-free, homequeue, or awesome.

Correctness Hypotheses

My correctness hypotheses were that none of the invariants we have specified in the design document would be violated. To make sure this was the case, I used homework 2, as a standard for comparing. Every time I computed a checksum, I added its value to a long int of initial value 0, and compared the result if I put the same inputs in this assignment too. The idea is they won't perfectly match, as the time it takes for the timing thread to signal the flag change might result in more checksums being computed before the experiment is stopped. As such, the numbers will not be perfect, but if they are reasonably close, we can expect our experiment to be running correctly both as far as time goes, and as far as the computing of checksums goes.

Guaranteeing correctness would be a longer project, and would involve logging every checksum computed, and comparing each one rigorously with the same thing done for homework 2 to make sure that all the checksums are correctly computed. This extensive logging effort is a bit too much for this assignment, given the time, and hence we opted not to do it, but to acknowledge it as something that could be done in the future to guarantee correctness.

Experiment Methodology

Before we jump into the specific experiments, it is worth talking about general methodology. Since the experiments are done on computers, which are dynamic systems, each experiment is susceptible to noise. As such, the project specification advised us to repeat each trial around a certain number of times. After doing so, I took the median of the repeated trials, and used that value to eliminate any outliers. Nevertheless, there is no way to guarantee perfect data free of noise.

Experiment 1

Hypotheses

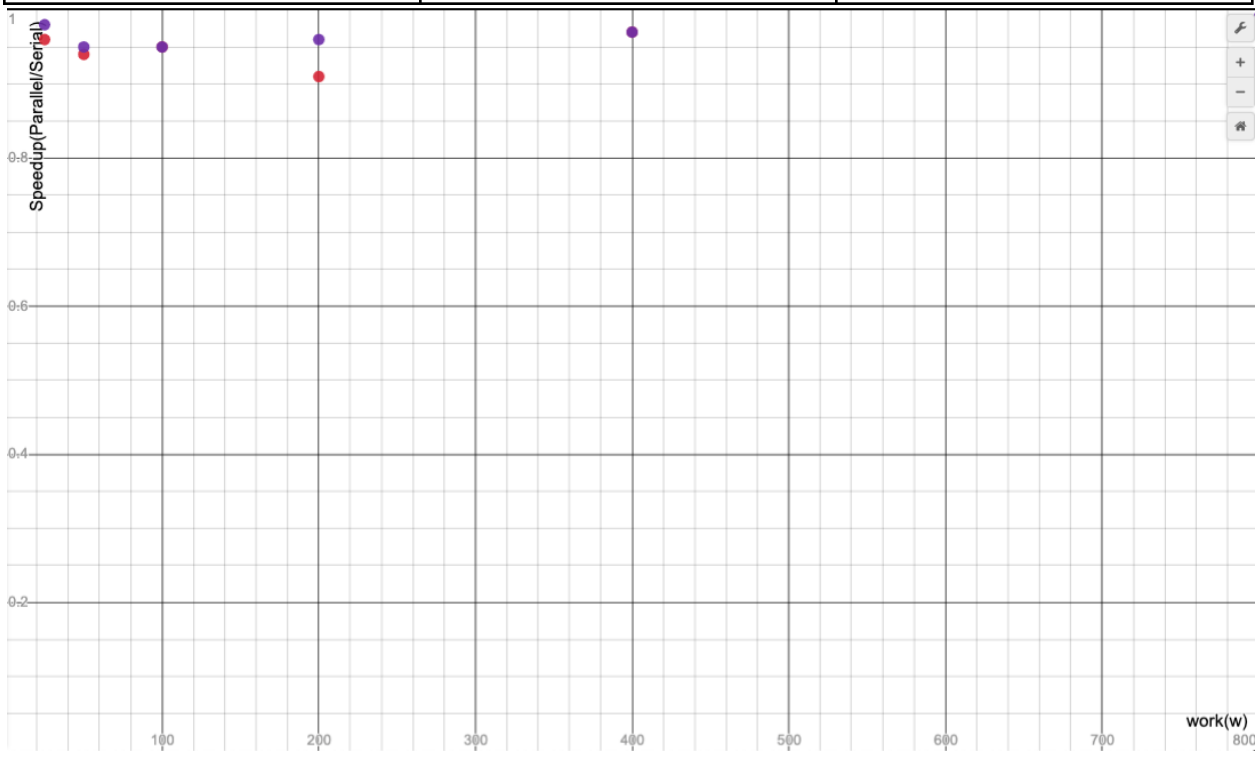
1. I expect the speedup for both locks to be very close to 1, but not exactly 1, as the locks do incur some overhead. As W increases, expecting speedup to approach 1
2. Between both locks, I am expecting to see the maximum deviation be 5% from 1 for all the speedups
3. I am expecting to see the speedup of TTAS perform about 5% better than pthread_mutex across the board
4. I am expecting the dispatcher rates to be about 10% slower for TTAS and 15-20% slower for pthread_mutex when compared to homework2 due to lock overhead

Data

Speedup of Table (Parallel/Serial)

W(Work)	Speedup for Pthread_Mutex	Speedup for TTAS
25	0.96	0.98

50	0.94	0.95
100	0.95	0.95
200	0.91	0.96
400	0.97	0.97
800	0.97	0.99



Worker Rate Table (total # of packets/Time)

W (Work)	Worker Rate for Pthread_Mutex	Worker Rate for TTAS

25	1287	1439
50	948	966
100	872	913
200	540	571
400	236	249
800	70	93

Analysis

The reason for why I expect the speedup of both locks to be close to 1 is that only one worker thread is pulling packets off the lamport queue. Hence we are getting no parallelism, and as such, the only thing we can do as programmers is to minimize the overhead. Overhead comes from spawning the timer thread and communicating the information about the flag that the timer thread edits and setting up other data structures used when sources are greater than one. The smallest speedup is 0.91 and the largest is 0.99, so we can agree with the first part of the first hypothesis. The second part of the first hypothesis is that the speedup approaches one as w increases, as the overhead is becoming a smaller percentage of the overall work. As far as that trend goes, we cannot really justify it in this experiment, as all the speedups are very close to 1. While this trend would make sense, we can only prove its efficacy by doing the same experiment over a larger range of work values. Furthermore, some of the checksum helpers might be getting cached, which makes the speedup look higher than it should be from trial to trial. Hypothesis 2 says I expect the minimum speedup amongst both locks to be 0.95. This is not true, however, as

the pthread mutex has a speedup of 0.91 when work is 200 which is a 9% difference. This goes to show that when I made hypothesis 2, I clearly overestimated the overhead that goes into setting up the locks, the actual work it takes to lock and unlock said locks, and the overhead that goes into the timer thread. This particular data point might have also involved many cache lines being kicked out, which might be a little unlucky explaining why the deviance is about twice as much as it is for other data points. My third hypothesis was that I am expecting to see the speedup of TTAS perform about 5% better than pthread_mutex across the board. This is completely untrue, as the difference in speedup for the same work values is on average less than 2%. I made this hypothesis expecting lock type to matter a lot and the 5% I chose was a guess based on the hw3a assignment, but as we see here, the specific lock type adds only a little bit of overhead. Part of this is the strategy I used to write my code. I initialized my code to work for either lock, and then chose the one that the command line input asked for, so a lot of the difference in locktype is mitigated by this. My last hypothesis is that I am expecting the dispatcher rates to be about 10% slower for TTAS and 15-20% slower for pthread_mutex when compared to homework2 due to lock overhead. This comparison is very important, as we did the same thing in homework 2 without locks. My difference was accounting for the expected overhead the locks would give in addition to the timer thread. The pthread_mutex dispatcher rate was about 16 percent worse than that in homework 2, while the TTAS dispatcher rate was on average about 11% worse than that in homework 2. We can see that my hypothesis was very close, as I correctly made an educated guess as to the overhead of the locks and the timer thread. The slight difference comes from me not taking into account that I would be initializing both locks regardless of which one I ended up using, and as such, my code takes a little longer than it is supposed to. This part of the experiment demonstrates that the overhead of both locks is not as

much as I expected it to be, and as such, when I finally implement my awesome strategy, this overhead should be beatable, and I should see a positive speedup if I am clever.

Experiment 2

Hypotheses

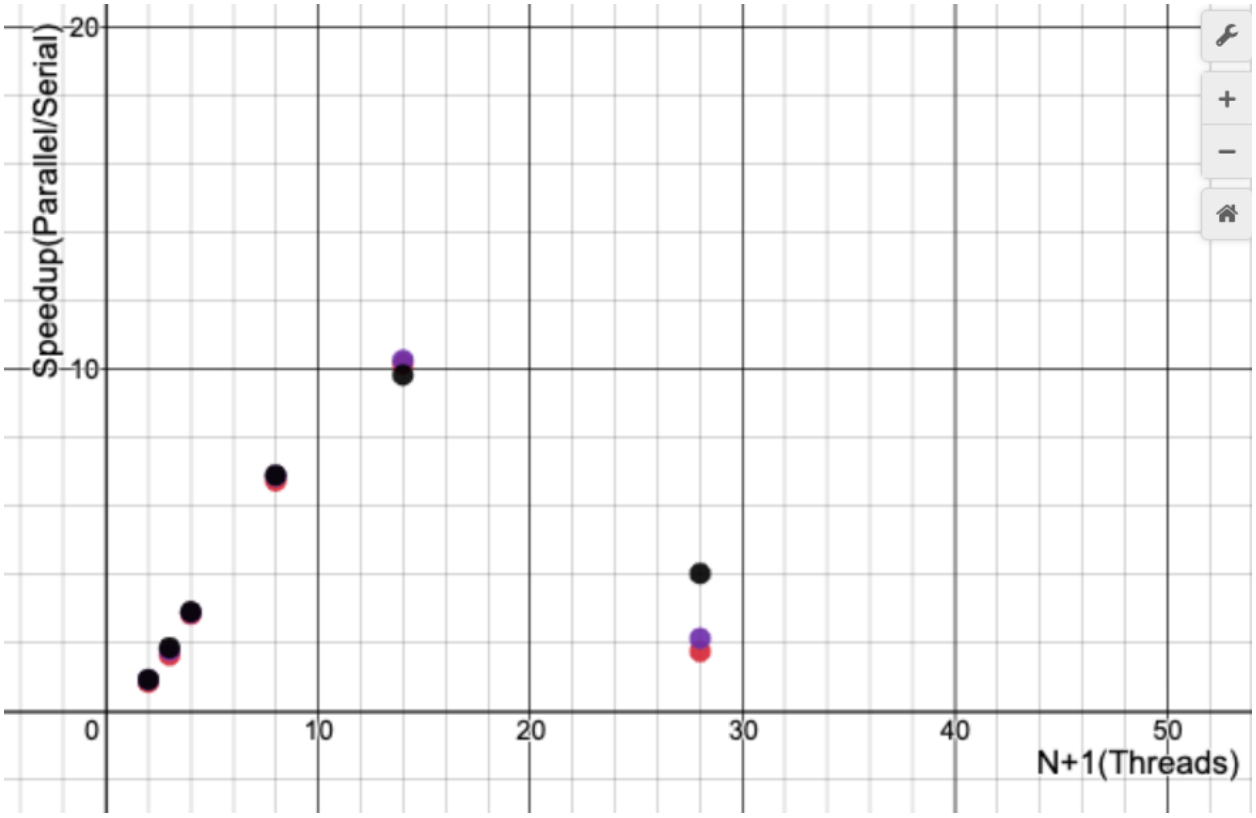
1. Within each W, I am expecting the speedup for each of the three categories to increase exponentially as W increases.
2. I am expecting to see the speedups quickly approaching N, as we are getting closer to perfect parallelism, as n increases
3. Between W's,I am expecting to see each speedup increase by about 5-10% each time W goes up
4. I am expecting to see TTAS perform ~15% better than pthread_mutex, and I am expecting to see Lockfree performing about 5% faster than TTAS for each data point

Data

Speedup for W = 1000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS	Speedup for Lockfree
2	0.85	0.92	0.93
3	1.64	1.79	1.86

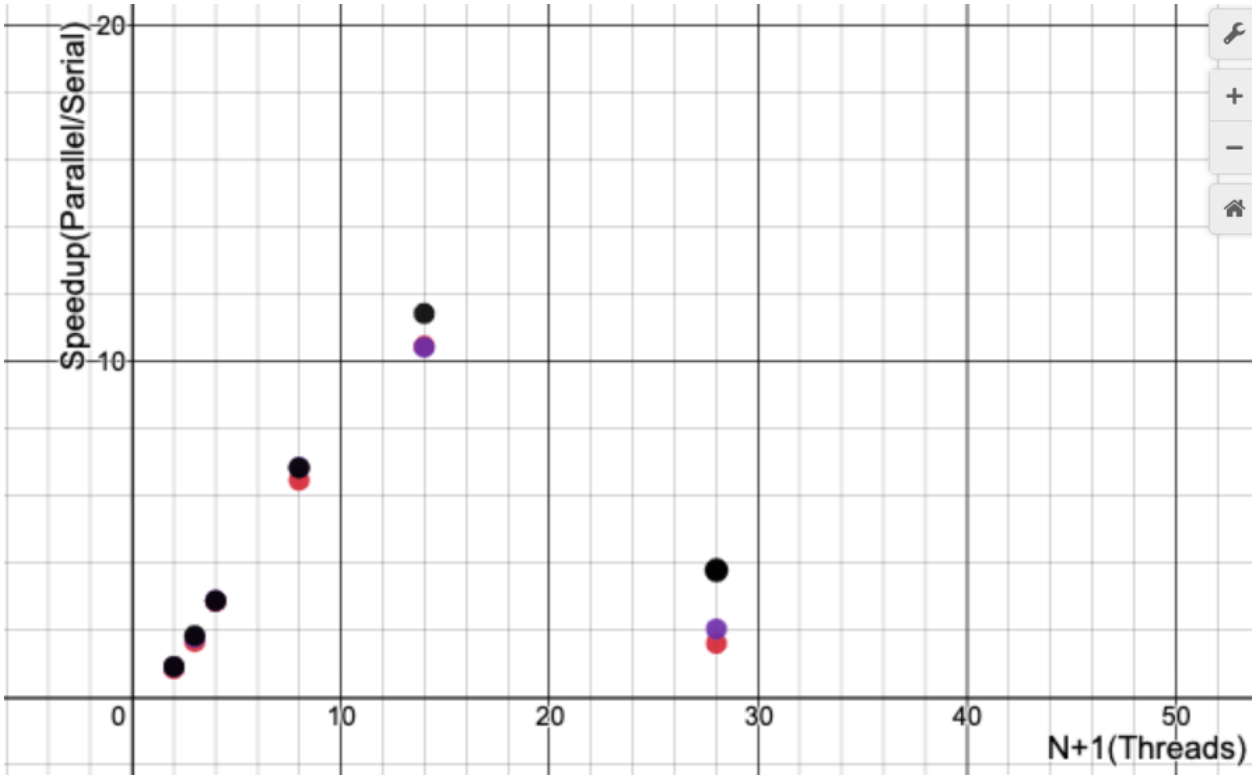
4	2.84	2.89	2.91
8	6.74	6.89	6.91
14	10.16	10.28	9.84
28	1.75	2.13	4.03



Speedup for W = 2000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS	Speedup for Lockfree
2	0.86	0.91	0.92

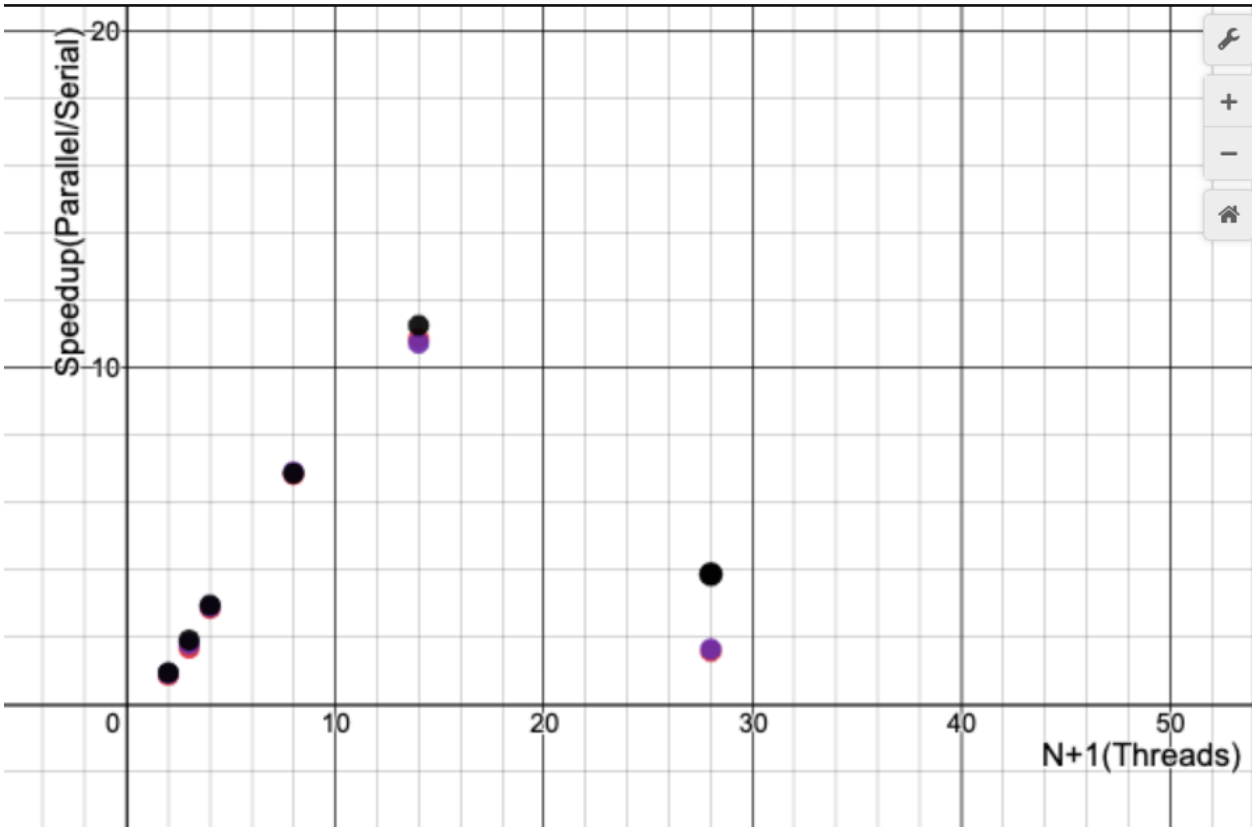
3	1.66	1.79	1.84
4	2.86	2.89	2.88
8	6.47	6.85	6.84
14	10.47	10.43	11.44
28	1.61	2.04	3.79



Speedup for W = 4000 (Parallel/Serial)

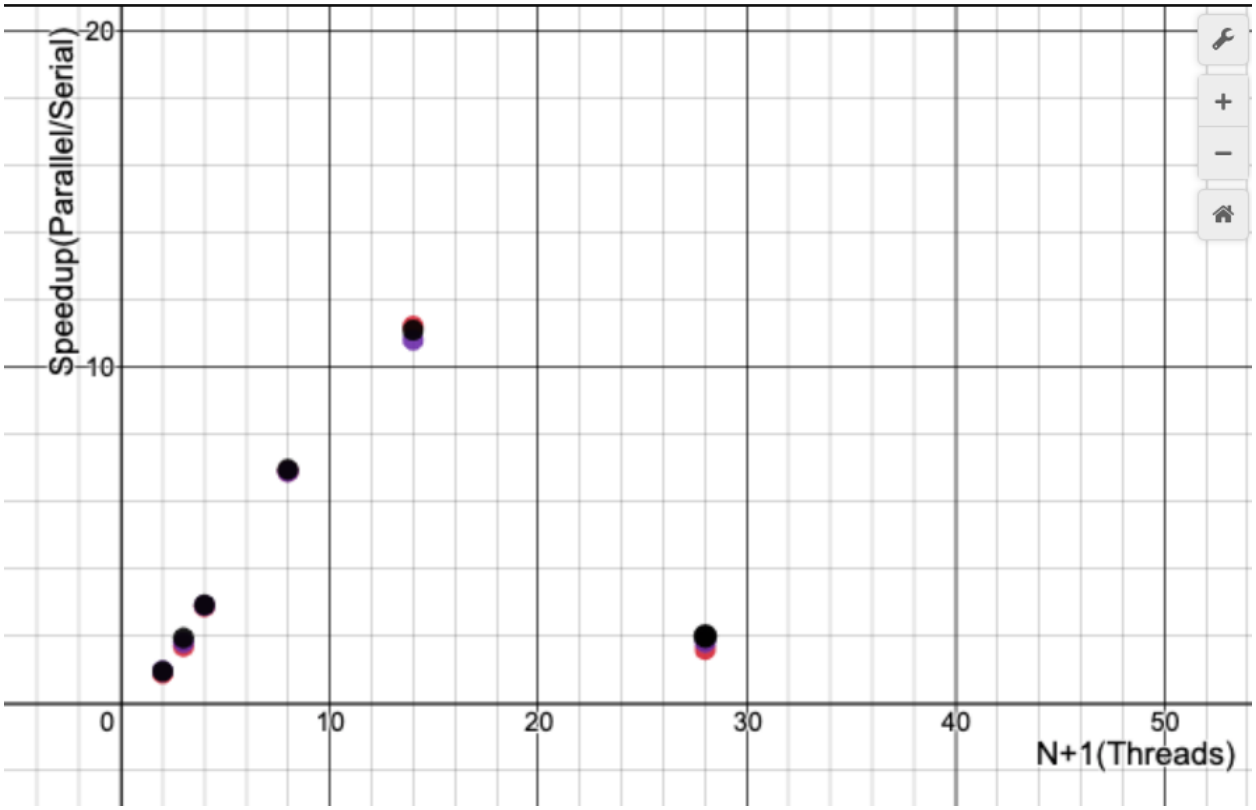
N+1(threads)	Speedup for	Speedup for TTAS	Speedup for Lockfree

	Pthread_Mutex		
2	0.87	0.93	0.94
3	1.67	1.80	1.91
4	2.85	2.91	2.95
8	6.84	6.91	6.87
14	10.87	10.74	11.27
28	1.58	1.65	3.87



Speedup for W = 8000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS	Speedup for Lockfree
2	0.89	0.96	0.95
3	1.69	1.81	1.94
4	2.88	2.91	2.93
8	6.91	6.90	6.96
14	11.23	10.82	11.11
28	1.60	1.82	2.00



Analysis

Overall this experiment was very cool, as we got to see how the current configuration works with both the lock designs, and the lock-free design without load balancing the project. Ultimately, our awesome design can be compared to the results in this experiment, and the next one. With regard to the first hypothesis, the first speedup increases by about 100%, and then increases to about 200%, and then by the time we get to N+1 = 28, we see like a 900% decrease in speed. While the shape might be exponential for the most part, we need to talk about the N+1 = 28 point. The reason I predicted the exponential behaviour in the first place is that I thought that the idle overhead should comprise a smaller and smaller percentage of the overall code, so the speedup in my experiments should be about the ideal speedup(#threads*serial), as we increase the value of N. The N+1 data point, however, is completely messed up, as slurm only

lets 13 threads run at a time, so at least half are just sitting there in an idle fashion taking up time, and space, and not contributing anything to the speed other than slowing everything down. Our second hypothesis that the speedups approach N, as N increases is found in the data, as we are getting closer and closer to perfect parallelism, as n increases. This is once again because the overhead comprises a smaller percentage of the overall work, so we are getting closer and closer to that ideal number. The only reason we see deviances when n is like 8(we see speedups around 7) is that our code is not at all load balanced well. For the third hypothesis, I predicted that between W's, I am expecting to see each speedup increase by about 5-10% each time, as W increases. The values are much closer than that (~2%) each time, reflecting that changing the value of W at this scale does not do as much as I predicted it to. Furthermore, the time spawning the threads and actually doing the computing of checksums far dwarfs the difference in W values between the tables. Last hypothesis is that I am expecting to see TTAS perform ~15% better than pthread_mutex, and I am expecting to see Lockfree performing about 5% faster than TTAS for each data point. The difference in speedups between the locks is between 5-10%, so I was a little bit incorrect in that area. I think the reason for this is that I overestimated what the difference in locks actually would do. The bottom line is that the pthread_mutex is not as bad (as far as performance goes) as I thought it was. The difference between lockfree and the TTAS however is ~1%, so my hypothesis was very incorrect there. The reason for this is that even in my lock free implementation, I initialized all the locks, and just did not use them, which consumes a lot of overhead. This theory is justified, as we see the results in homework 2 producing ~10% speedup more than my lock free implementation in this assignment. If we want to see greater speedups even closer to perfect parallelism, we have to deal with this load balancing issue.

Experiment 3

Hypotheses

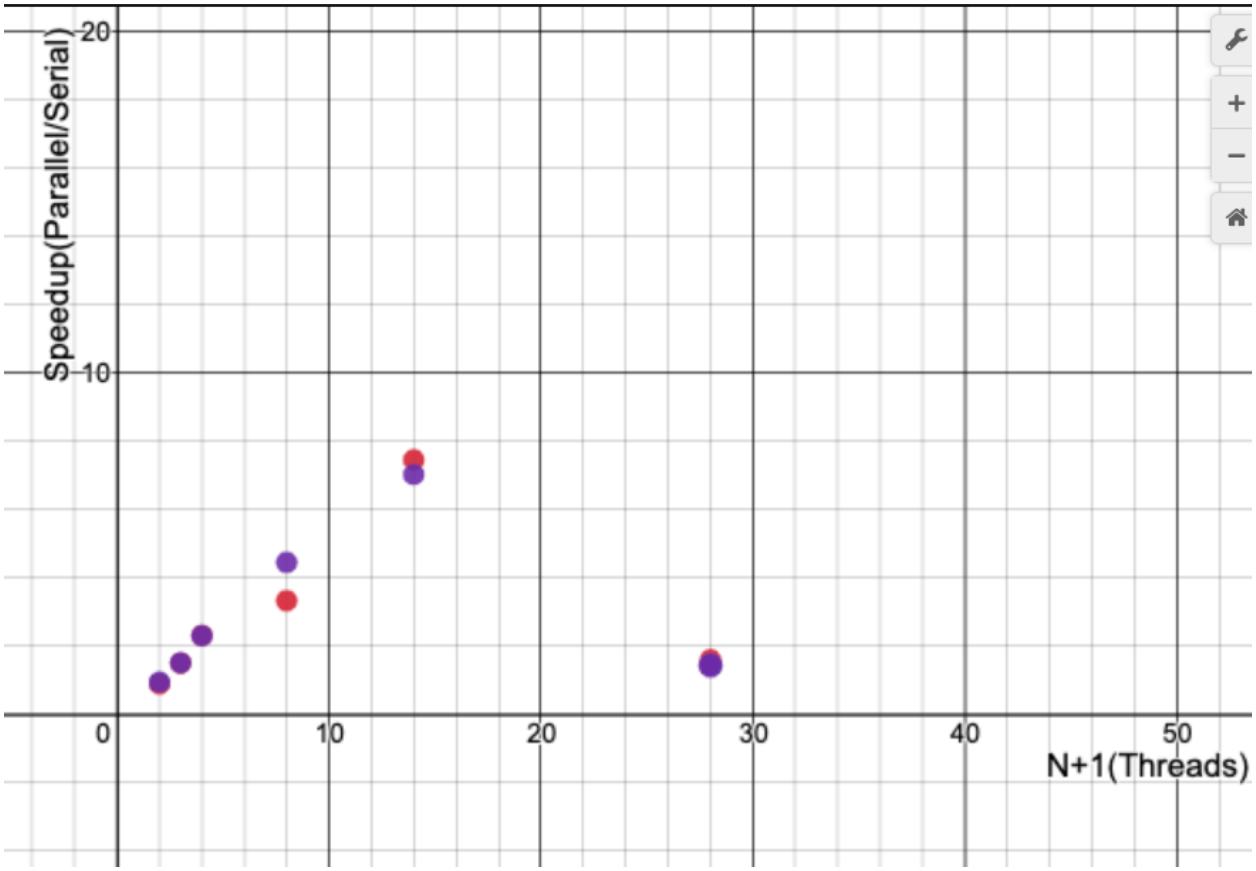
1. Within each W, I am expecting the speedup for each of the three categories to increase exponentially as W increases.
2. I am expecting to see the speedups quickly approaching N, as we are getting closer to perfect parallelism, as n increases. I am expecting to see these results perform about 10% worse than last experiment's results, as we haven't load balanced, and exponential packets are going to hurt bad load balancing more.
3. Between W's,I am expecting to see each speedup increase by about 5-10% each time W goes up
4. I am expecting to see TTAS perform ~15% better than pthread_mutex,

Data

Speedup for W = 1000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS
2	0.89	0.94
3	1.50	1.51
4	2.31	2.30
8	3.33	4.45

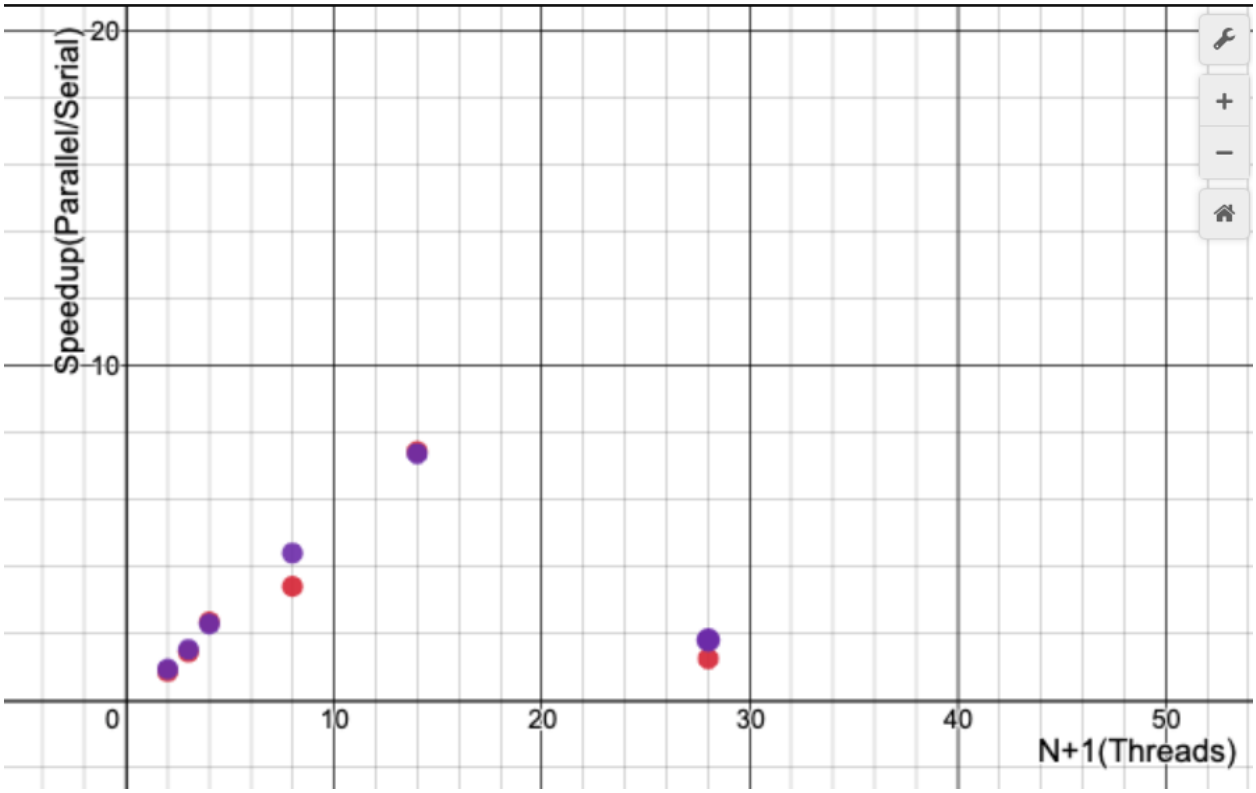
14	7.46	7.03
28	1.39	1.43



Speedup for W = 2000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS
2	0.87	0.94
3	1.45	1.53
4	2.35	2.29

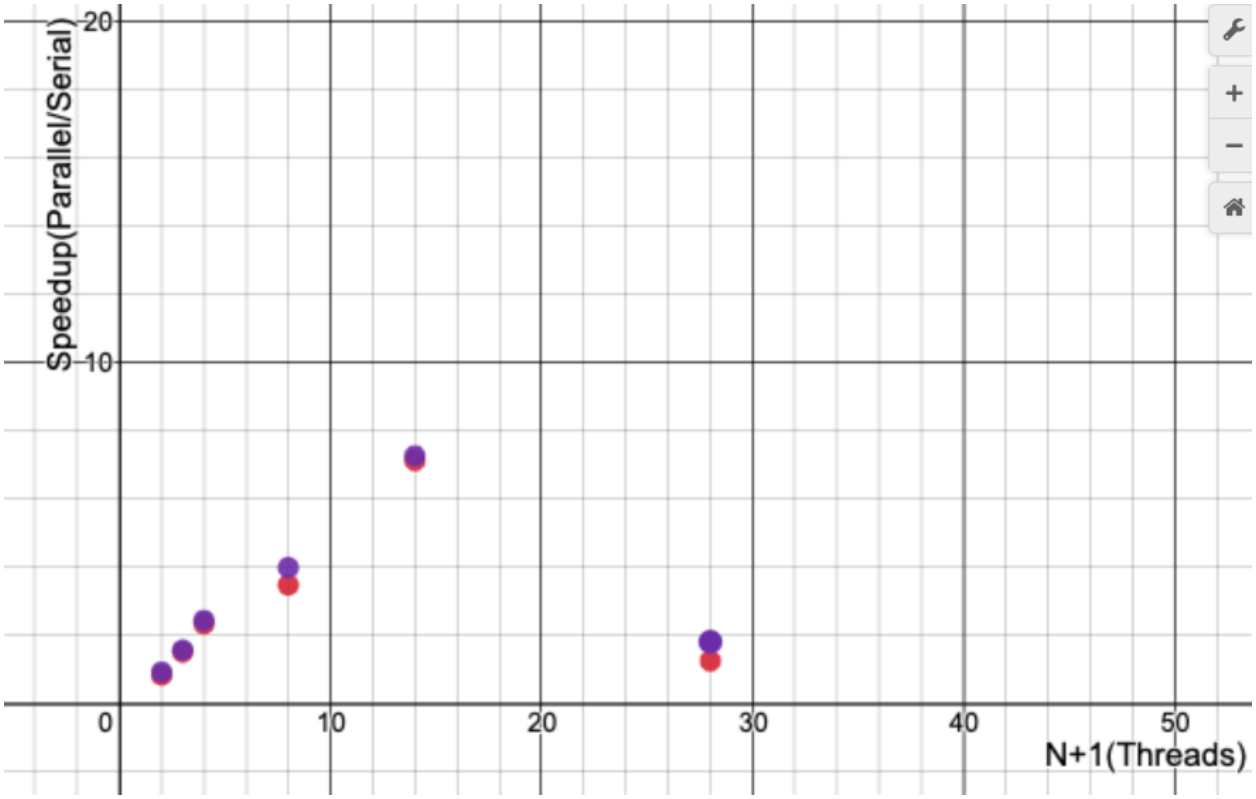
8	3.41	4.41
14	7.45	7.38
28	1.25	1.81



Speedup for W = 4000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS
2	0.83	0.92

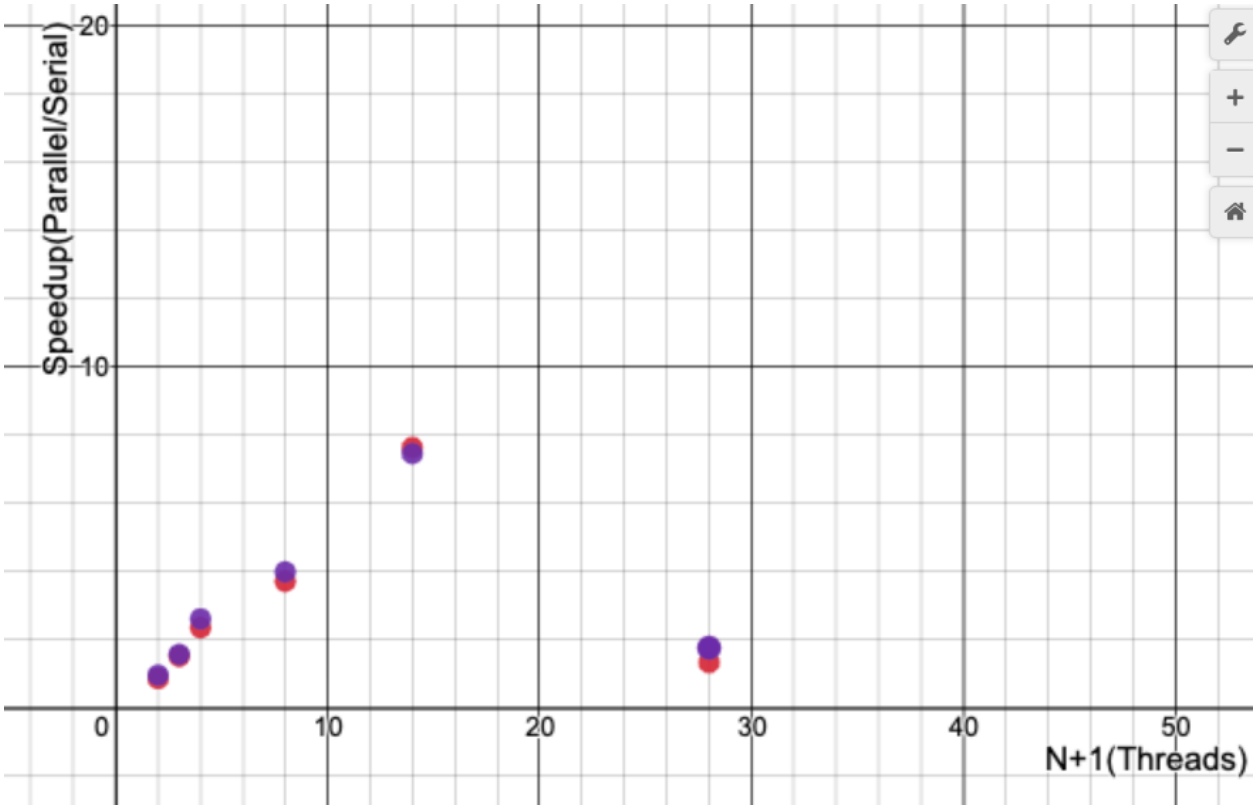
3	1.51	1.57
4	2.34	2.44
8	3.48	3.99
14	7.13	7.27
28	1.25	1.81



Speedup for W = 8000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS

2	0.86	0.96
3	1.51	1.57
4	2.35	2.61
8	3.71	3.99
14	7.64	7.46
28	1.33	1.76



Analysis

In experiment 3, we are finally using exponential packets so we can expect the speedups to be considerably worse than last experiment, as we are bearing the full brunt of not load balancing our poor firewall. As the first hypothesis says, we are expecting to see our speedups rapidly approaching N, as we are getting closer to perfect parallelism. While our speedups are indisputably exponentially increasing, it is definitely not keeping up with N (it's on average ~20% away from N). The reason behind this is we have not balanced our firewall, so there are many situations where threads are just sitting there idle, as nothing is in their associated lamport queue, while other queues are working over time. It is worth also addressing the N+1 = 28 point. The speedup is about 40% worse than last experiment, and not even remotely close to the 27 goal. This is the compounding effects of not load balancing the firewall, and slurm only running 13 threads at once, so most are just sitting there idle. My next hypothesis is the same as that for the last experiment, and is that I am expecting to see each speedup increase by about 5-10% each time W goes up. In reality, this increase is so small that sometimes it is not even increasing. This is due to how badly not load balancing hurts us, when we are using exponential packets. My last hypothesis is that I expect to see TTAS perform ~15% better than pthread_mutex across all trials in this experiment. The difference was usually between 10-15%, so I was actually pretty accurate this time, but honestly looking back at it, I was right for the wrong reasons. I wanted to attribute this to the difference in lock performance, but as we saw in prior experiments, the difference between both these locks are not as significant as we think. What I think was happening was honestly the luck of the draw with regard to how the exponential packets were distributed. I think the difference in performance we saw was almost completely due to load balancing, and not lock type. The evidence for this theory is the last two experiments where we see that lock type does not matter too much. This experiment does show the horror of not distributing work evenly

amongst our poor worker threads that do not have a chance of doing it themselves with exponential packets.

Experiment 4

Disclaimer

For my awesome strategy, I chose uniform packets instead of exponential. The reason for this is that my awesome algorithm is greedy in nature (I explained it properly in the deviances section), so I was worried that randomly choosing a queue to try to obtain lock would not work as well with exponential packets, as each thread doing work with a particular queue is holding on to that lock for so much time.

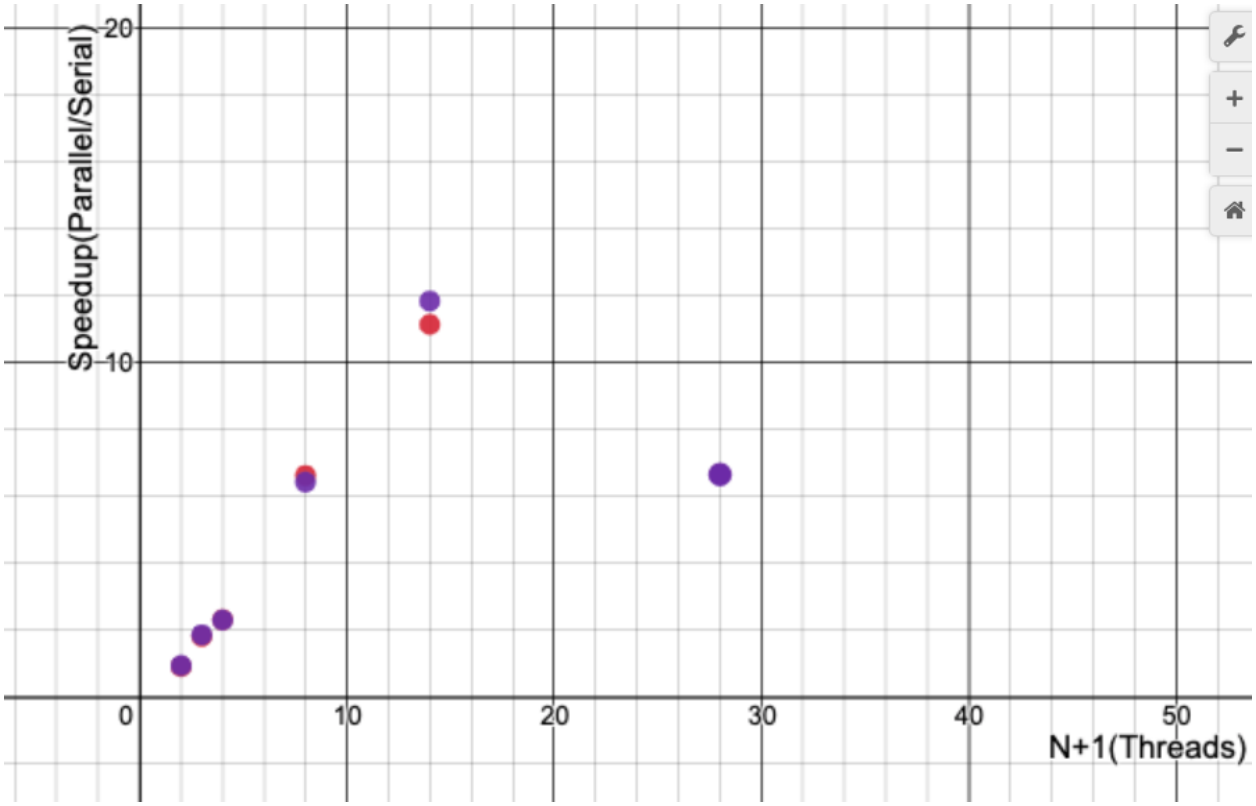
Hypotheses

1. Within each W, I am expecting the speedup for each of the two categories to still increase exponentially as W increases
2. I am expecting to see the speedups quickly approaching N, as we are getting closer to perfect parallelism, as n increases. I am expecting these results to be very close to N due to better load balancing, and as such, am expecting the speedup to get within 10% of perfect parallelism, so about a 20% increase from experiment 2
3. Between W's, I am still expecting to see the speedup increase by about 5-10% each time W goes up
4. I am expecting to see TTAS perform ~15% better than pthread_mutex for each data point

Data

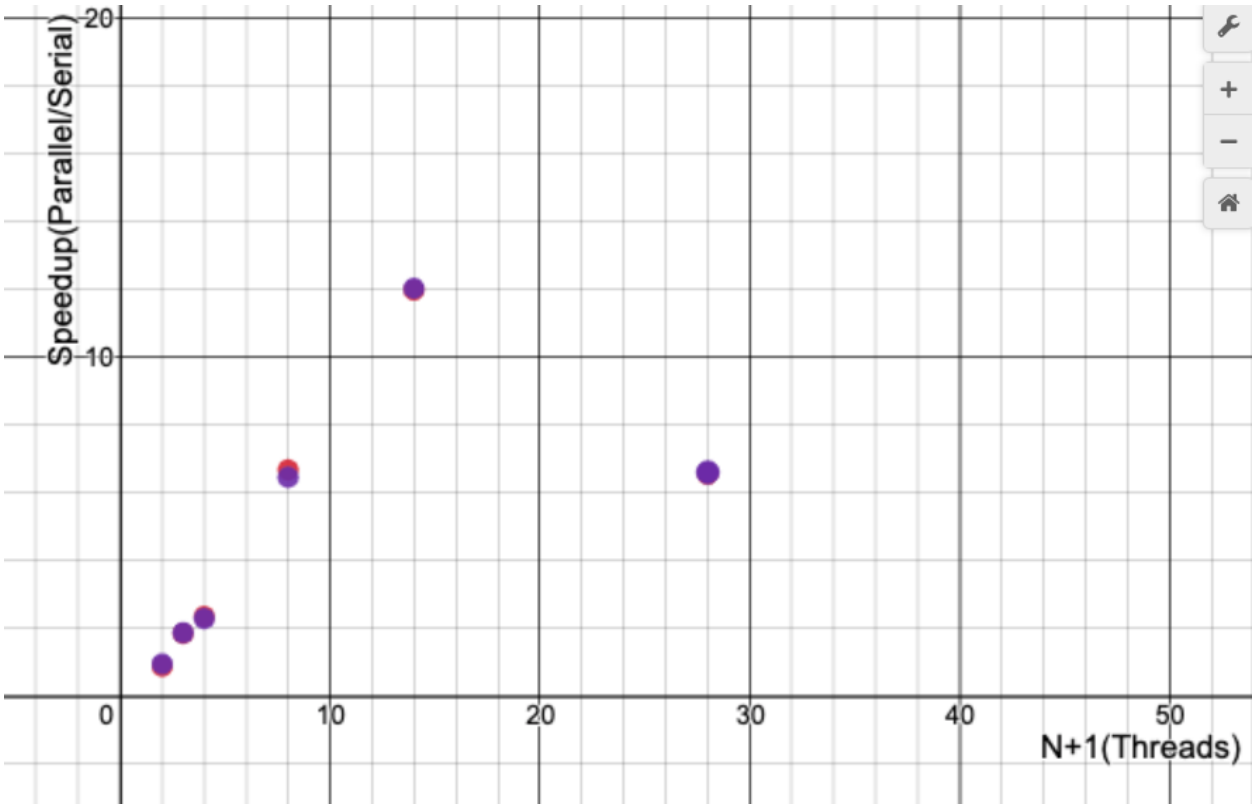
Speedup for W = 1000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS
2	0.91	0.94
3	1.81	1.86
4	2.31	2.30
8	6.63	6.43
14	11.15	11.85
28	6.67	6.66



Speedup for W = 2000 (Parallel/Serial)

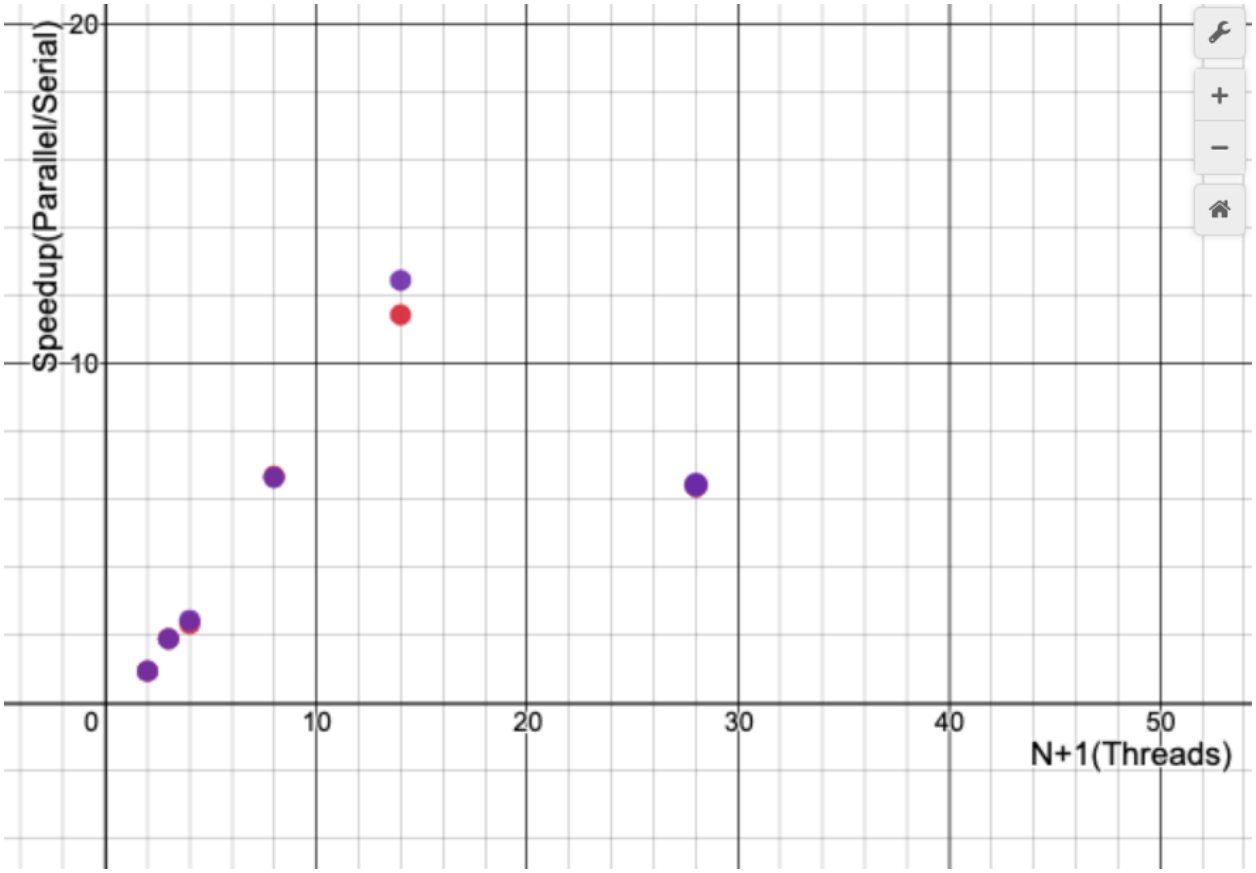
N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS
2	0.88	0.95
3	1.85	1.87
4	2.35	2.29
8	6.68	6.46
14	11.99	12.04
28	6.55	6.61



Speedup for W = 4000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS
2	0.94	0.94
3	1.90	1.89
4	2.34	2.44
8	6.69	6.65
14	11.45	12.47

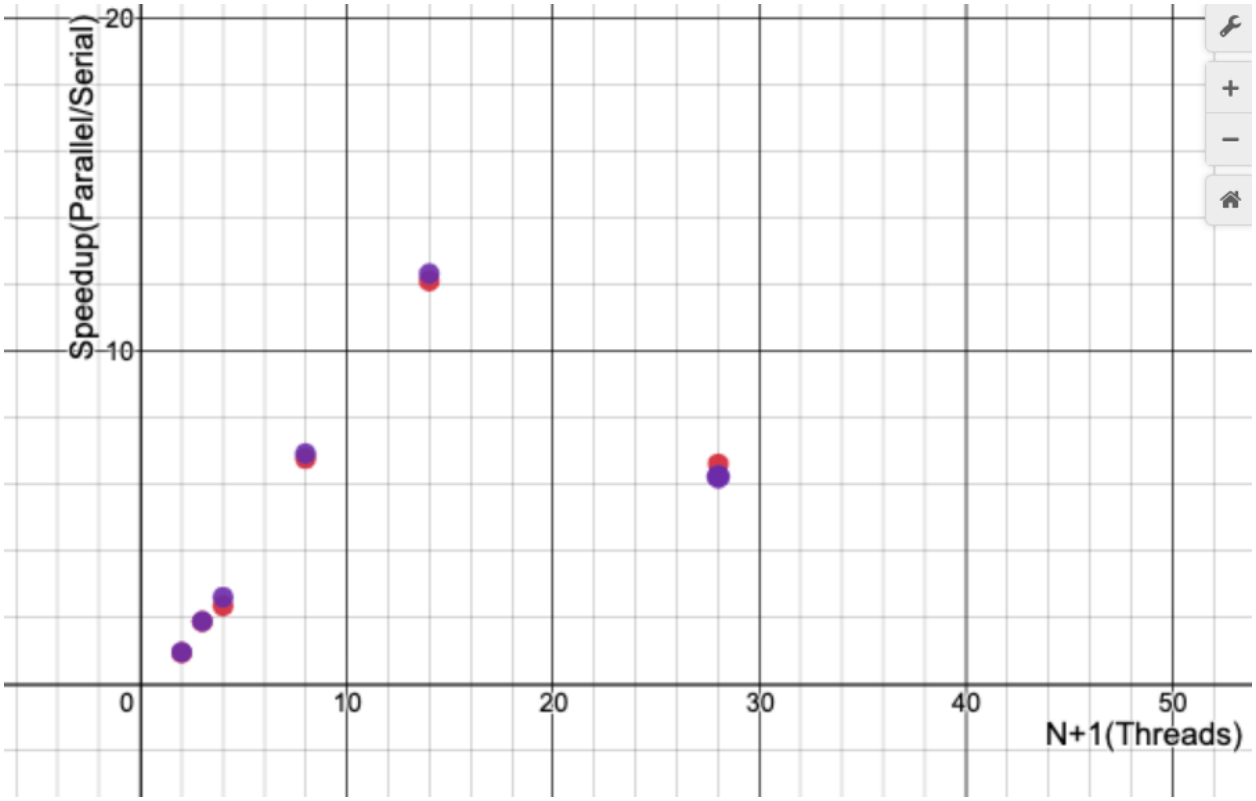
28	6.38	6.44



Speedup for W = 8000 (Parallel/Serial)

N+1(threads)	Speedup for Pthread_Mutex	Speedup for TTAS
2	0.94	0.96
3	1.88	1.89
4	2.35	2.61

8	6.78	6.93
14	12.12	12.34
28	6.62	6.24



Analysis

In many ways, this final experiment is the culmination of this class. We have learned all about how to cleverly parallelize things, and now we get to apply it to our firewall. As such, I was super excited to do this experiment. Once again, our first hypothesis is that within each W, I am expecting the speedup for each of the two categories to still increase exponentially as W increases. This is true for the most part except for n +1 = 14 and n+1 = 28, when the percent change is ~200% and -200% respectively. The reason for this is that our locks take up a smaller

and smaller percentage of the overall work. Compound that with the superior load balancing scheme that we have, and we can see that the speedup is a good exponential increase that is as close to perfect parallelism, as we have seen in our firewall so far. This brings us to hypothesis 2, as we are expecting to see the speedups quickly approaching N, as we are getting closer to perfect parallelism, as n increases. I am expecting these results to be very close to N due to better load balancing, and as such, am expecting the speedup to get within 10% of perfect parallelism, so about a 20% increase from experiment 2. This experiment went above and beyond my expectations, as I got comfortably within 5% of perfect parallelism, which means that the percent increases from experiment 2 varied between as low as 10 and as high as about 30%. Clearly, the greedy approach works very well for uniform packet handling. My third hypothesis is the same as last time and is that between W's, I am still expecting to see the speedup increase by about 5-10% each time W goes up. Once again, this hypothesis is wrong, as in the scale we are doing this experiment in, with these values of N and W, W going up beyond a certain threshold makes minute amounts of differences way under 5%. My last hypothesis was confirming that TTAS indeed barely outperforms pthread_mutex(~5%), and proving that my last hypothesis overestimated the difference between locks. Overall, this experiment was a clear success, as we see massive speedup increases across the board. It would be very interesting to try my code on the exponential packet setup. I am expecting the greedy approach to not work well, so it would be interesting to formalize that if we had more time.

Conclusion

Overall, these experiments clearly demonstrate that my awesome approach is a good way of load balancing uniform packets. Even then, to confirm that it really is, we need to do many more trials over a larger variety of n values, and a larger variety of w values. The awesome

approach will likely fall apart for exponential packets, as we have to be cleverer than just choosing random queues, as they likely already have their lock taken due to exponential packets taking a while to get their checksums computed. I am very curious to also try a dynamic programming approach where I keep an array of work in each queue, and assign threads to the entries with maximum work to see if that could load balance for both exponential and uniform packets effectively. I will try over the summer!!!!