Parallel Programming HW3b Writeup

Andrew McCarthy

Source files are located in the hw3b/ directory. My code is in clhlock.c/.h and packet.c. Instructions on how to compile the program are located under the "details" section.

- Overview

This program is designed to mimic a firewall and process threads for a set amount of time. The program can use three different strategies to do this. The first one is serial. In this version, a single thread cycles between each source processing each packet linearly. The second version is lock free. This is the same design as in project two, except the worker threads run for a set amount of time instead of processing a set amount of threads. The second version is homequeue. This is similar to the lock free version, except each thread must acquire a (uncontended) lock on the queue before dequeuing an element. The final version is my "awesome" implementation. In this version, each thread cycles between all the queues. Also, it must obtain a lock before dequeuing an element. This version is designed to mitigate load balancing issues that were present in project two. By not having each thread mapped to a single queue, queues with more expensive packets will not slow down the entire program.

- Design

The dispatcher for this program works similarly to the one from project two. A single thread loops through each queue and enqueues items. If the current queue is full, the dispatcher will wait until a spot opens up before moving on to the next queue. The dispatcher also keeps track of how much time has passed. Once the timer has reached the limit specified by the user, the dispatcher thread sets each worker thread's 'endFlag' to 1. The flags are stored in a global int array called *endFlags.

The serial, lock free, and homequeue functions work exactly as they did in homework 2, except that they end once their endflag = 1, and home queue has each worker acquire a lock before dequeuing.

The “awesome” worker thread design has each thread cycle through all the queues starting with the i’th queue. Each thread must obtain a lock on a queue before dequeuing an element. If the current queue has a lock on it, the thread will move on to the next one. Each worker thread dequeues packets until its endFlag is set to false by the dispatcher.

Each thread for all the strategies is passed an args struct that contains its thread number, the number of packets it has processed (used to calculate the total number of processed packets once the worker threads have joined), the lock type it should use, and the total number of threads.

I chose to use the mutex and clh lock for this assignment because they were my two best performing locks from hw3a. This would ensure that my awesome design could achieve the best possible performance.

For this assignment, I also changed my queue implementation. I ran into serious bugs with my old implementation that I was unable to fix due to its unnecessary complexity. I implemented new enqueue and dequeue functions that work by keeping track of head and tail variables for each queue and by storing the packets in an array for each queue. For enqueue, if tail - head equals the queue depth, the function returns one. If it doesn't, the packet is inserted into the appropriate position in the packet array and the tail value is incremented. For the dequeue function, if tail - head equals zero, the function returns NULL. If it doesn’t, the appropriate packet is removed and the head pointer is incremented. The head and tail values are set to zero at the start of the program.

The total processed count is calculated by having each thread maintain a local variable that is incremented each time a packet is successfully dequeued. Once they have finished running, they update the processed field of their parameters and the main thread adds them all up.

- Details

The program can be compiled by running make. The object files and the executable can be removed by running make clean. The executable is called packet. The program has different required parameters depending on which mode it is running in. To run the program in serial mode, pass in the following parameters: -milliseconds (run time) -source (number of sources) -w

(average work) -seed (seed for the packet generator). To run the program in parallel mode, include the -parallel flag and the following parameters: -milliseconds ‘value’ -source ‘value’ -w ‘value’ -seed ‘value’ -depth (depth of the queues) -type (lock type: 0 for mutex, 1 for clh) -strategy (worker strategy: 0 for lock free, 1 for home queue, 2 for awesome). You can also include the -exp flag if you want the program to use exponential packet generation. Not including it will make the program use uniform packet generation. The parameters can be provided in any order. Here is an example execution: ./packet -parallel -milliseconds 2000 -source 7 -depth 8 -w 5000 -seed 5 -type 0 -strategy 0 -exp. To run the program in test mode, use ./packet (seed) (test option). Here are the possible values for ‘test option’: 0 (correctness test) 1 (lock overhead test) 2 (speedup test) 3 (awesome test).

- Correctness Testing

For correctness testing, I changed the dispatcher code to enqueue a specified number of packets before terminating. If the workers processed this specific number of packets, I considered the test a success. This test was designed to ensure that the locks and the queues were working correctly.

- Performance Testing

For performance testing, I ran the tests outlined in the project description, as well as an ‘awesome’ test where I compared the throughput of my awesome design to the throughput of the lock free design with exponential packet generation. For this test, I used the same n and w values as the previous two speedup tests.

Posted below are the raw results from my tests:

Lock Overhead Test: row is thread count, column is lock type. Value is lockfree throughput / homequeue throughput

Lock Type	25	50	100	200	400	800
mutex	0.986939	0.948322	1.048343	1.022535	1.009728	1.003831
clh	1.202608	1.18836	1.208297	1.125792	1.078582	1.044881

Speedup Test with Uniform Generation: row is thread count, column is strategy and lock type. Value is parallel throughput / serial throughput. Each section is a different w value.

w=1000	1	2	3	7	13	27
lockfree	0.973227	1.928618	2.83379	6.396498	6.588963	0.018049
mutex-homequeue	0.967893	1.7826	2.464652	3.825107	3.988	1.261794
clh-homequeue	0.936064	1.743042	2.563284	4.409015	4.732263	0.000611
w=2000	1	2	3	7	13	27
lockfree	0.980992	1.94481	2.871509	6.40425	11.276366	0.03425
mutex-homequeue	0.981228	1.891509	2.674989	5.401093	5.212581	2.736523
clh-homequeue	0.961323	1.898344	2.77962	5.807614	8.811539	0.001145
w=4000	1	2	3	7	13	27
lockfree	0.980882	1.960095	2.870215	6.601675	11.446054	0.072263
mutex-homequeue	0.980992	1.89954	2.78558	6.105635	9.563977	5.202484
clh-homequeue	0.981816	1.874835	2.772605	6.282035	10.473703	0.002328
w=8000	1	2	3	7	13	27
lockfree	0.98377	1.950845	2.885111	6.380005	11.056647	0.155243
mutex-homequeue	0.995636	1.901756	2.815571	6.413881	11.110973	8.483271
clh-homequeue	0.976354	1.882176	2.883392	6.481195	11.674783	0.004438

Speedup Test with Exponential Generation: row is thread count, column is strategy and lock type. Value is parallel throughput / serial throughput. Each section is a different w value.

w=1000	1	2	3	7	13	27
lockfree	0.985707	1.269779	1.693292	3.751735	5.925591	0.02058
mutex-homequeue	0.987342	0.995793	1.284308	2.515475	2.409631	1.328311
clh-homequeue	0.960942	1.147157	1.43108	2.651893	3.583798	0.000618

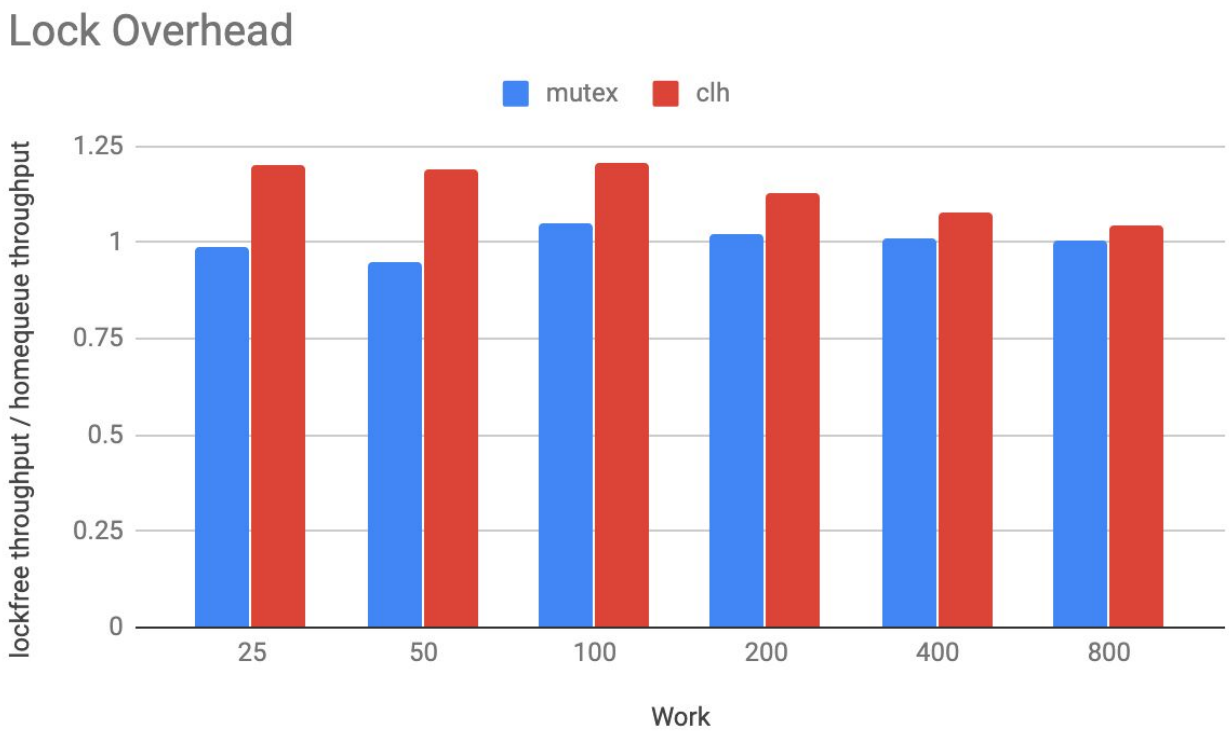
w=2000	1	2	3	7	13	27
lockfree	0.995509	1.249624	1.659325	3.550359	6.439097	0.042835
mutex-homequeue	0.975673	1.083884	1.400152	2.971466	3.841498	2.396715
clh-homequeue	0.978088	1.184056	1.525839	3.031197	5.15266	0.001103
w=4000	1	2	3	7	13	27
lockfree	1.002228	1.229602	1.656997	3.559904	6.070604	0.073307
mutex-homequeue	1.006514	1.178622	1.492807	3.25723	5.092897	3.346035
clh-homequeue	0.995782	1.230515	1.601167	3.484857	5.701194	0.002484
w=8000	1	2	3	7	13	27
lockfree	0.994769	1.291097	1.647215	3.895751	6.18946	0.146676
mutex-homequeue	0.992134	1.201327	1.534763	3.753241	6.186941	4.32058
clh-homequeue	0.985072	1.256349	1.572688	3.644254	6.217259	0.004536

Awesome Speedup with Exponential Generation: row is thread count, column is lock type. Value is awesome throughput / lock free throughput. Each section is a different w value.

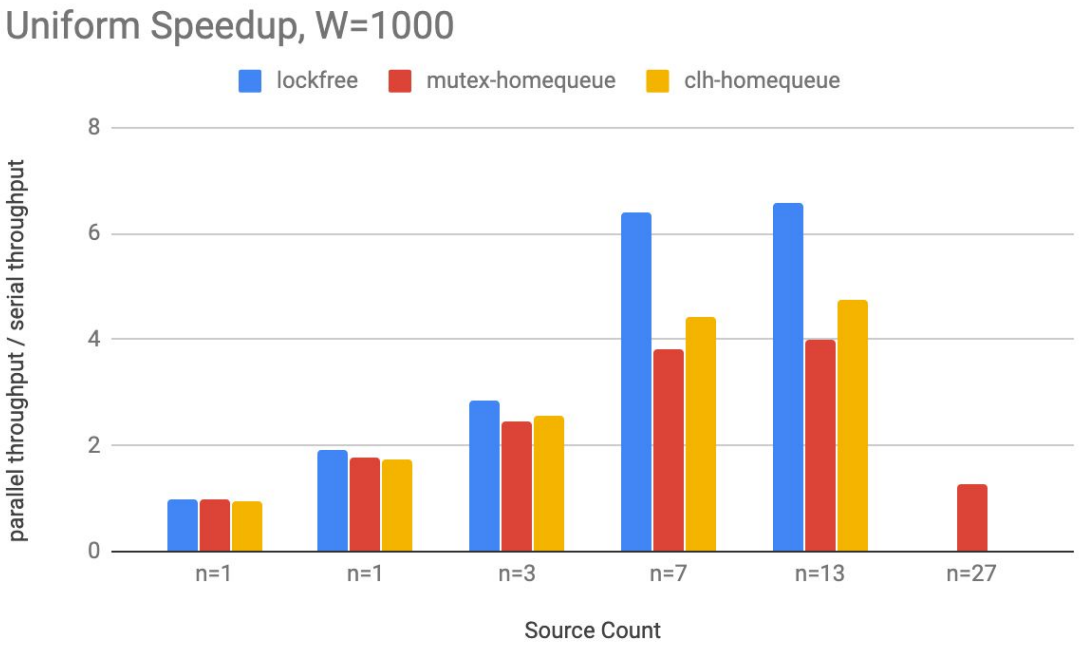
w=1000	1	2	3	7	13	27
mutex	0.999876	1.271605	1.637503	0.267938	0.150438	14.530242
clh	0.967423	1.40332	1.583926	1.058408	0.561504	14.780481
w=2000	1	2	3	7	13	27
Mutex	0.993605	1.522534	1.667461	1.154573	0.319142	10.248108
clh	0.991885	1.516445	1.751142	1.578328	1.040467	17.257869
w=4000	1	2	3	7	13	27
mutex	1.000349	1.53881	1.815671	1.595632	0.936471	19.909277
clh	0.977152	1.515217	1.799001	1.793513	1.613485	29.426282

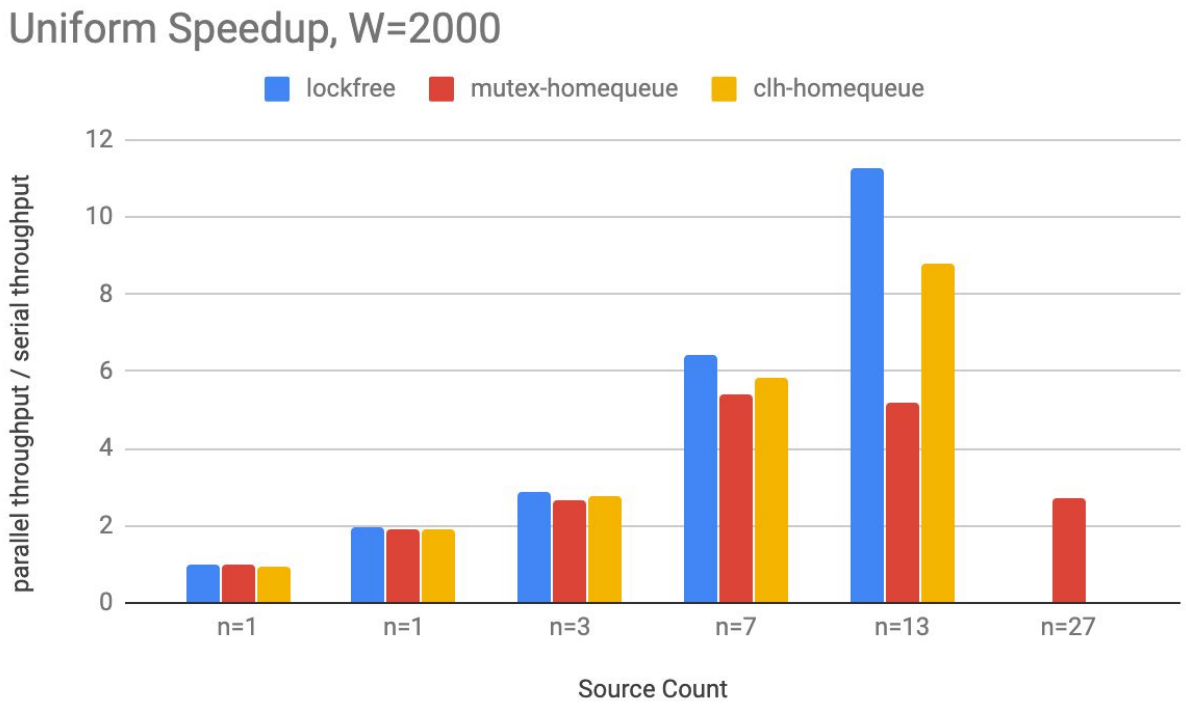
w=8000	1	2	3	7	13	27
mutex	1.004564	1.55927	1.79155	1.707454	1.540871	28.175998
clh	1.00624	1.560293	1.796778	1.865065	1.884253	29.355814

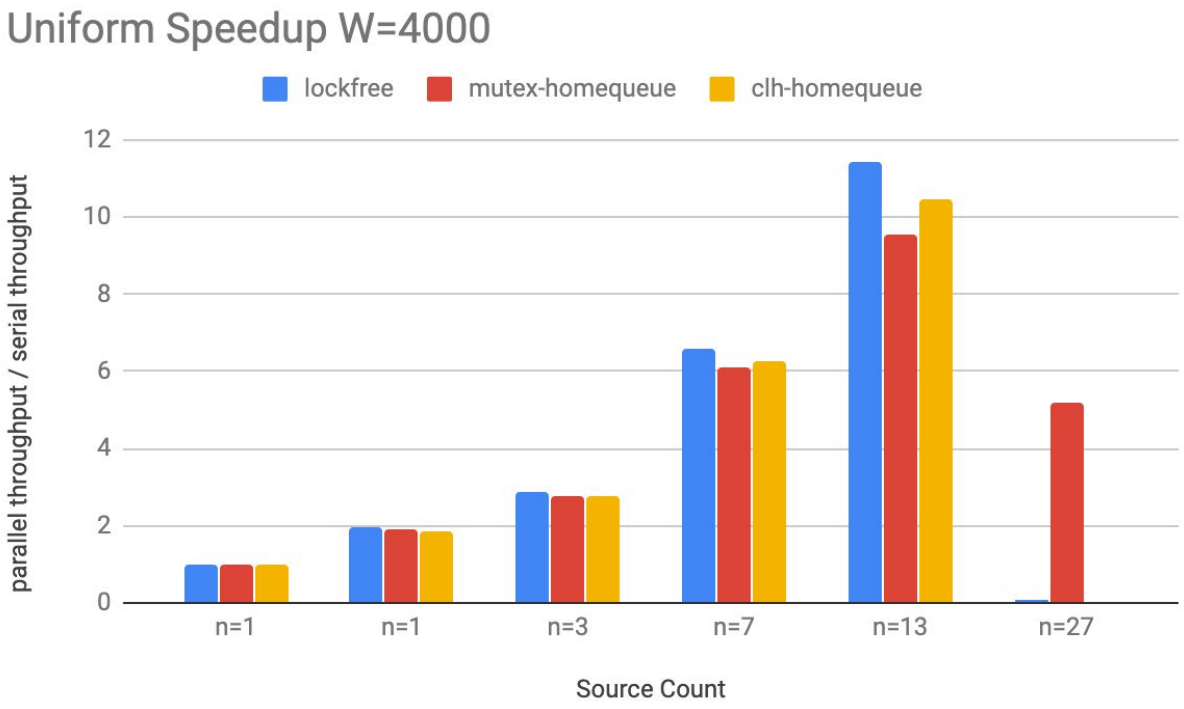
Posted below are graphs illustrating the results of my tests:

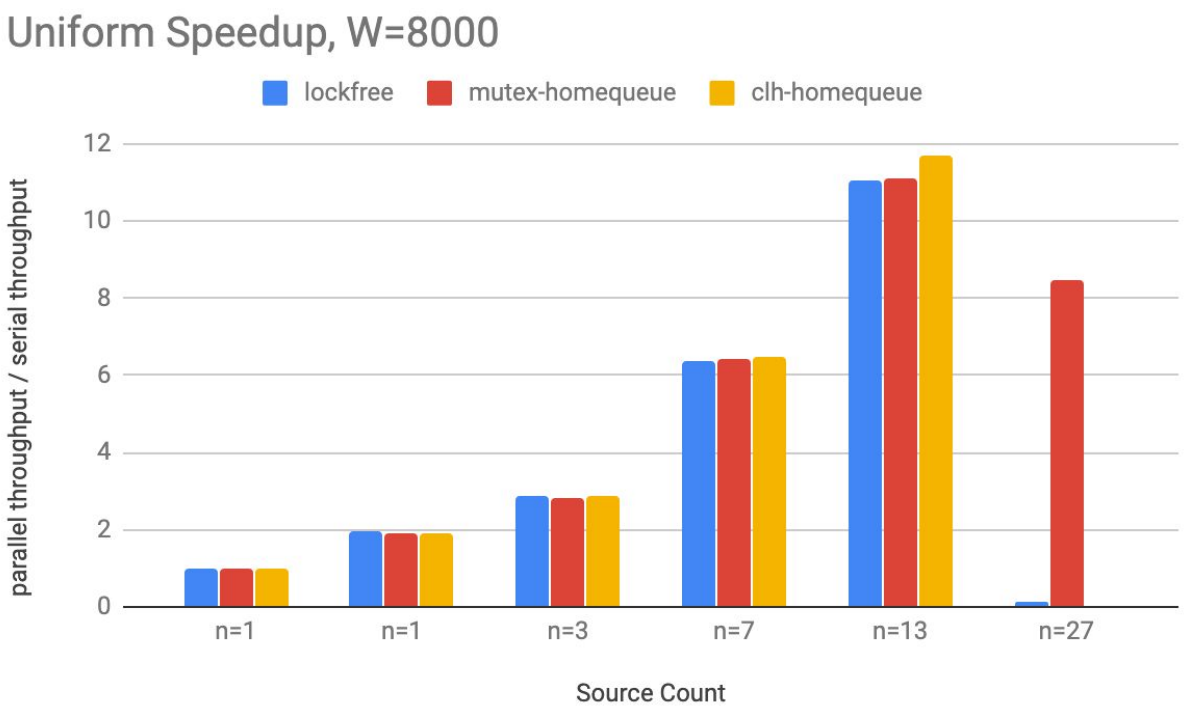


Parallel Speedup

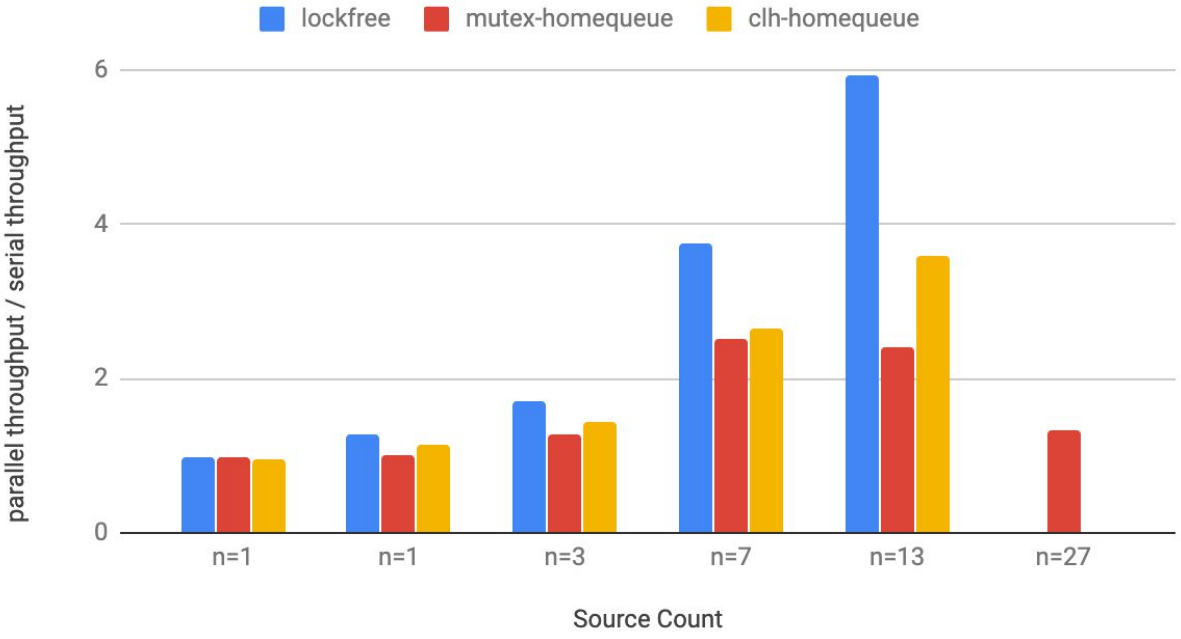




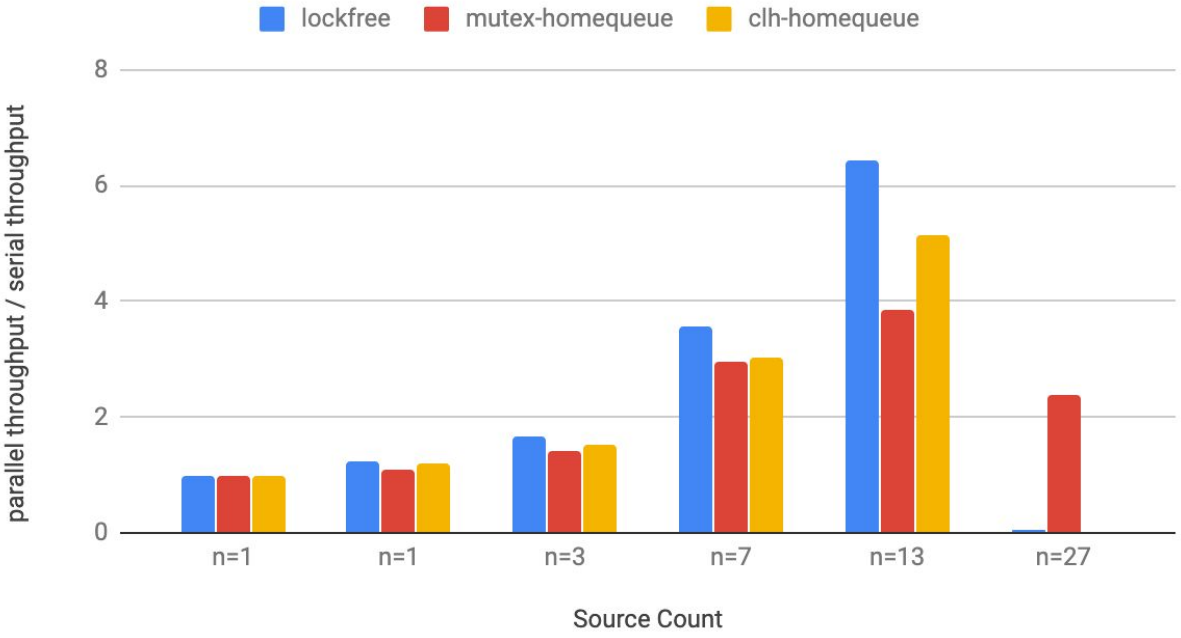

Uniform Speedup W=4000
lockfree
mutex-homequeue
clh-homequeue
parallel throughput / serial throughput
12
10
8
6
4
2
0
n=1
n=1
n=3
n=7
n=13
n=27
Source Count

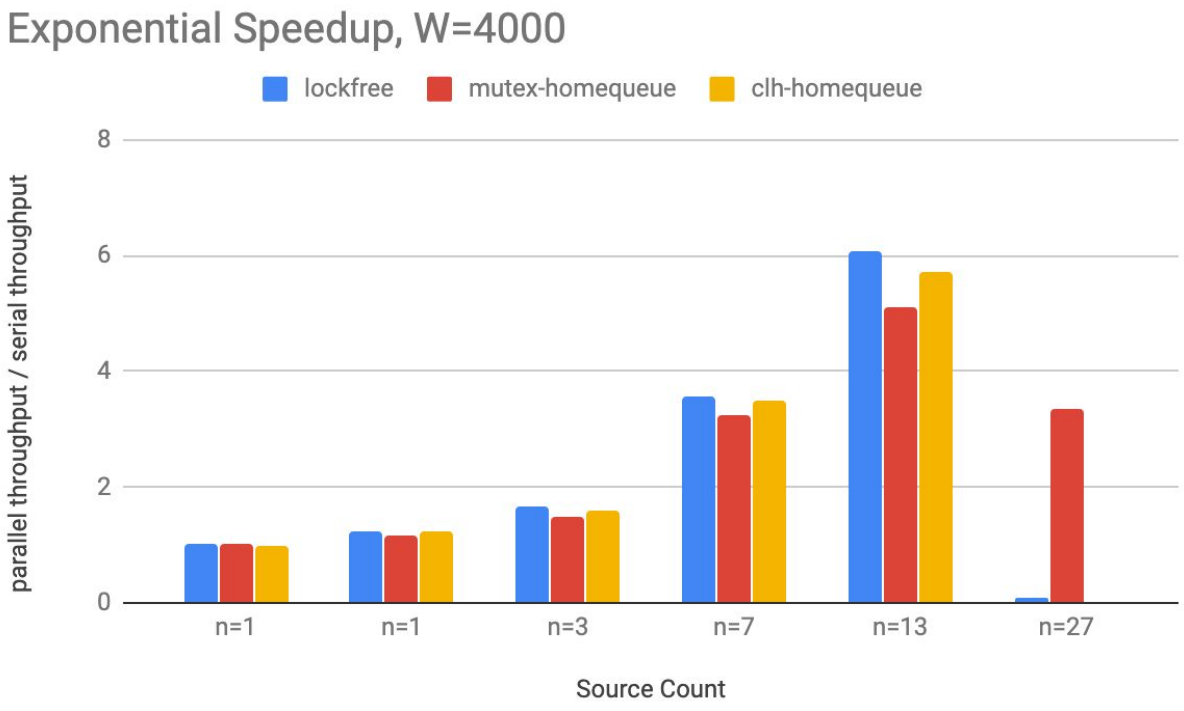

Uniform Speedup, W=8000
lockfree
mutex-homequeue
clh-homequeue
parallel throughput / serial throughput
12
10
8
6
4
2
0
n=1
n=1
n=3
n=7
n=13
n=27
Source Count

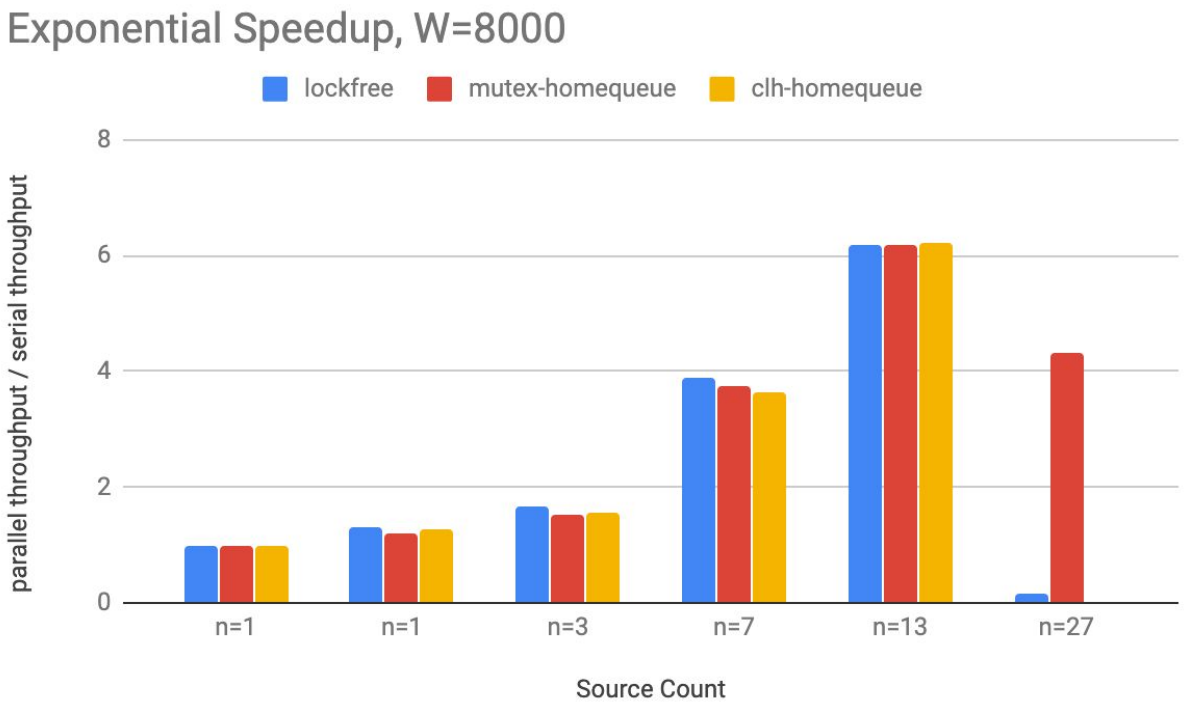
Exponential Speedup, W=1000



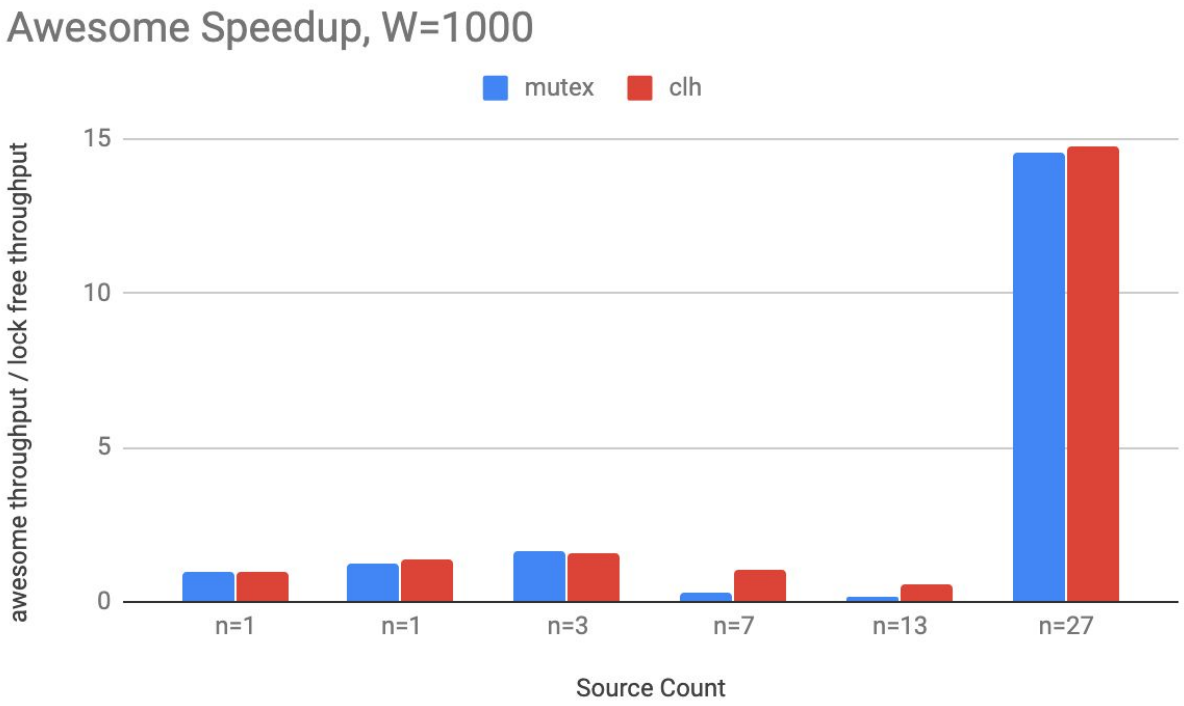
Exponential Speedup, W=2000

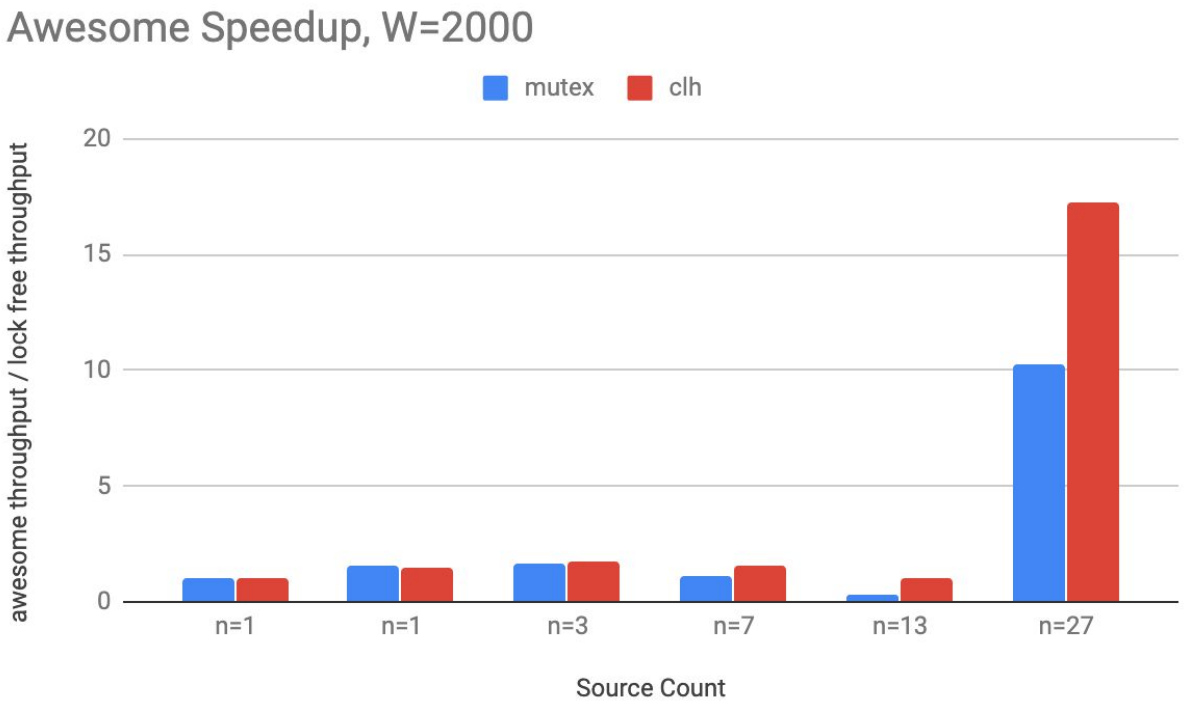


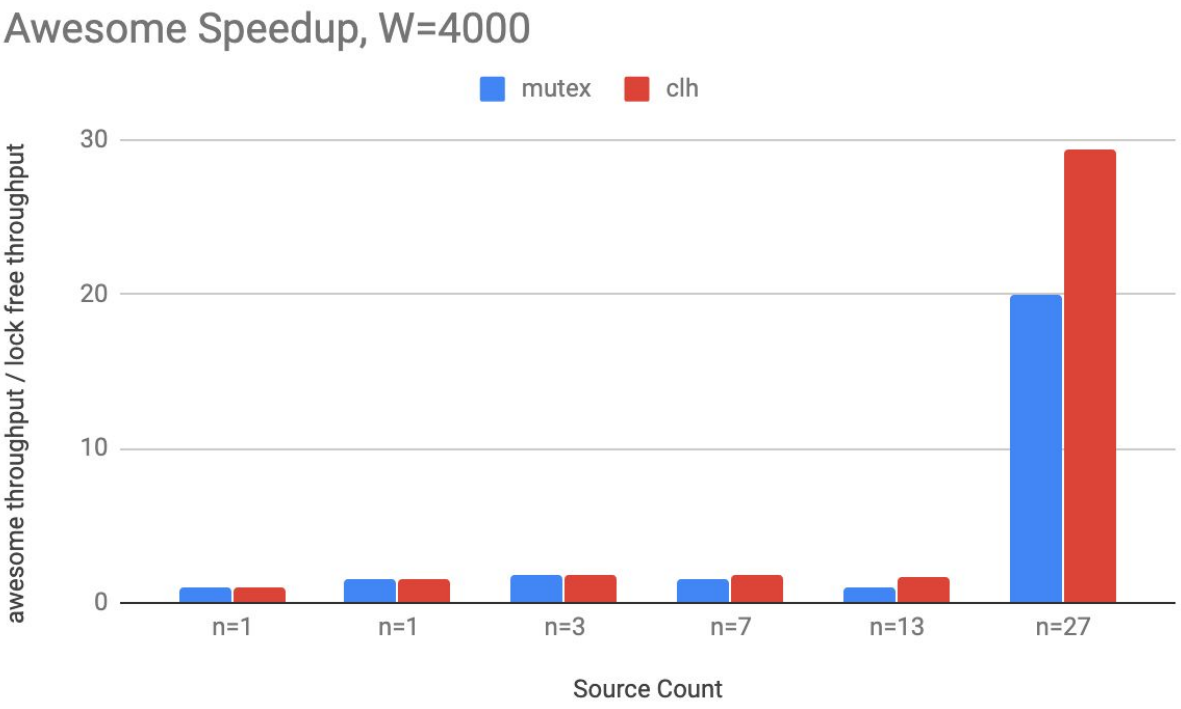

Exponential Speedup, W=4000
lockfree
mutex-homequeue
clh-homequeue
parallel throughput / serial throughput
8
6
4
2
0
n=1
n=1
n=3
n=7
n=13
n=27
Source Count

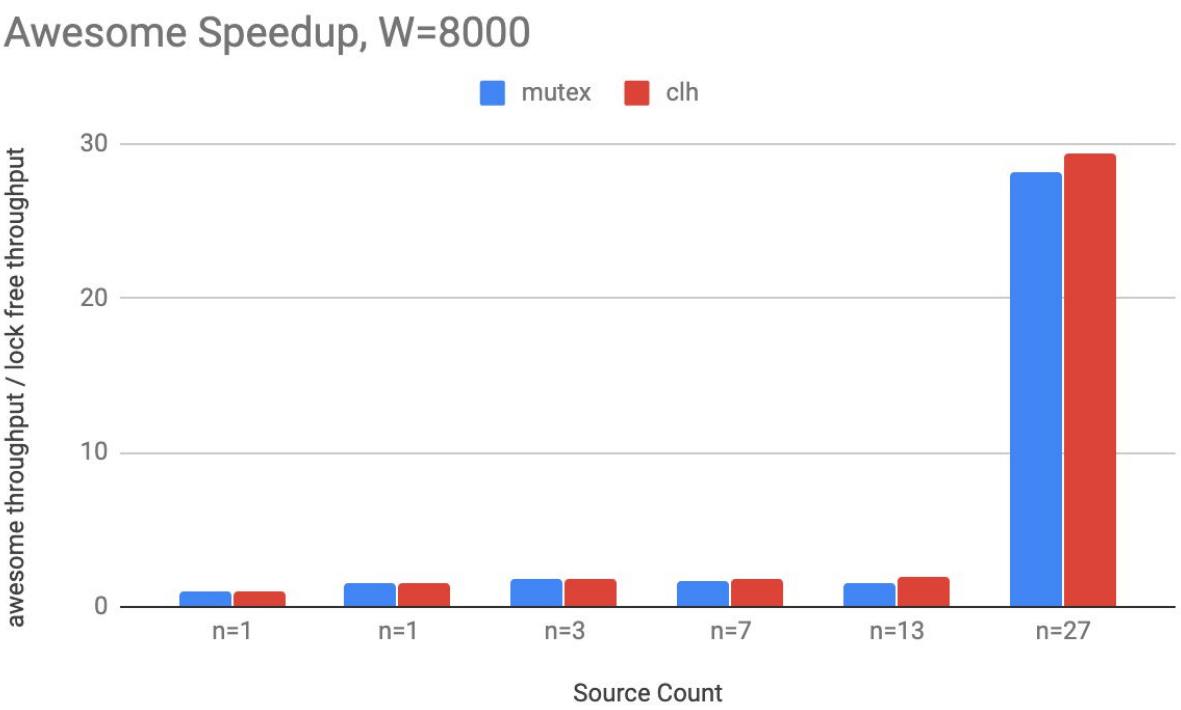

Exponential Speedup, W=8000
lockfree
mutex-homequeue
clh-homequeue
parallel throughput / serial throughput
8
6
4
2
0
n=1
n=1
n=3
n=7
n=13
n=27
Source Count

Awesome Test






Awesome Speedup, W=4000
mutex
clh
awesome throughput / lock free throughput
30
20
10
0
n=1
n=1
n=3
n=7
n=13
n=27
Source Count


Awesome Speedup, W=8000
mutex
clh
awesome throughput / lock free throughput
30
20
10
0
n=1
n=1
n=3
n=7
n=13
n=27
Source Count

- Performance Analysis

The results from the lock overhead test were exactly what I expected. The lock had a small overhead of around 20%, falling down to as low as 2-3% for higher average work values. The homequeue and lock free were more equal at higher work values as a result of packet processing taking up a greater percentage of the time. This means a smaller percentage of the time would be spent acquiring the locks.

My results from the speedup test were also what I expected. The lock free version achieved the highest speed up as there was no lock overhead. Interestingly, the clh lock achieved a higher speedup than the mutex lock. This means my clh lock was, on average, performing better than the mutex lock, contradicting the results from the counter test. I suspect this is because the clh lock is less complex than the mutex lock, and actual lock performance does not matter if the queues are uncontended. The performance increases were much more substantial for the tests using uniform packet generation, as was seen in my tests from hw2. These speedups reached as high as 1,100%. For exponential packet generation, the speedups only reached as high as around 600%. One strange observation from this test is the results for thread count = 27. Here, the lock free and clh lock versions were slower than the serial version. The mutex lock however, always achieved a higher throughput, but not as high as the previous observation. I suspect this is due to the thread count exceeding the core count.

My results from the awesome test were also what I expected. At a thread count of 1, both the lock free and awesome version achieved approximately the same input. At thread count = 2-7, the awesome version achieved a performance increase of around 40-60%. As with my previous test, the clh lock performed better than the mutex lock. For some reason I cannot explain, at a thread count of 13 the awesome version would achieve worse throughput than the lock free version. I again suspect this may have something to do with thread count exceeding core count. Also interestingly, at a thread count of 27, the awesome version achieved a much higher throughput than the lock free version (around 2700%). I also suspect this is in some way due to the core vs thread counts. Overall, my implementation of awesome was able to achieve around 40-70% better throughput compared to lock free.

Overall, my performance testing results lined up well with my hypotheses from the writeup.

One interesting observation I made while testing my awesome version was how it performed compared to lock free when I changed the dispatcher. I tested what would happen if I modified the dispatcher to not wait for empty slots if a queue were full and instead move on to the next queue. When I did this, I found that lock free would achieve a higher packets/second throughput than awesome. The lock free version has an equal worker rate between all the queues. Thus, the queues with the lower work packets will achieve a higher throughput than the queues with higher work packets. The awesome version on the other hand, spreads the worker rate out among all the queues. This means each queue will have a more equal throughput when compared to the lock free version. Overall, the awesome version will achieve higher throughput on the higher work queues than the lock free version while sacrificing throughput on the lower work queues. When my dispatcher was modified, if the higher work queues were taking a long time to dequeue elements, the dispatcher would unfairly prioritise loading packets into the lower work queues. This means the lock free version would have a higher packets/second throughput than the awesome version even though it was doing less work. When I used the original version of my dispatcher, packets would be loaded into the queues equally and the awesome version would achieve better performance.