Homework 3b Writeup

Michael Tang

due May 21, 2019

1 Usage

Use "make" to compile and link the modules from utils and from the root directory. "make clean" is available to remove all .o files.
Execute the program as follows:
./pacver [-n *nthreads*] [-W *pkwk*][-M *timelim*][-T *npck*][-D *qdepth*][-u *uniflag*][-x *rngseed*][-p *pflag*][-S *opmode*][-L *lockmode*][-o *outpath*]

- -n specifies number of sources. In parallel mode this translates to number of worker threads (so the total number of threads will be n+1). Required.
- -W specifies average work per packet. Required.
- -M specifies time limit for threads, in milliseconds. All threads will run until the time limit is reached. Due to time difference measurement in-program being based in seconds, -M must be a multiple of 1000. Either this or -T but **not both** must be specified.
- -T specifies number of packets each worker thread is to process. This was used for correctness testing purposes only. This XOR -M must be specified.
- -D specifies queue depth, default 8 if not specified.
- -u is a flag that has default value 1. If 0, packets will follow exponential load distribution, if 1, packets will follow uniform load distribution.
- -x specifies the seed for the packet generator. For all tests, -x 23 was used. Required.
- -p is a flag with default value 0. If 0, the program will run in serial mode, if 1, the program will run in parallel mode.
- -S specifies the strategy to use in parallel mode, with 0-2 being LockFree, HomeQueue, and Awesome modes in sequence. Required if -p 1, ignored if -p 0.
- -L specifies lock type, 0-3 for TASLock, PThread_Mutex, ALock, and CLHLock in sequence. Required if -p 1 and -S ¿ 0 is specified.
- -o is the path for the file to which user wants to output. If not specified, program will default to stdout. **WARNING:** files open in append mode. If you want to start a clean version of existing output files make sure to remove them first.

2 Design Changes

2.1 Input, Initialization, and Output

Following clarification in design review of how we were supposed to use -T and -M, both arguments have been supported. -u and -p were added to provide additional operation detail.
Operation flow remains the same as specified in the design document, and as in HW2: from argument parsing in main() to thread preparation and launch in opcontrol(), with the dispatcher thread calling dispatcher() and the worker threads calling operator().
Relevant outputs - specifically total packets dispatched, packets checked per thread, total packets checked, and sum of checksums - have been moved to post-operation of all threads instead of in the middle of operation. I.e., instead of printing checksums as they process them, worker threads now add the checksum to a running thread subtotal, then write that subtotal to their private slot in an array that holds all thread subtotals. All those subtotals are added up to get the total sum of checksums, which is printed at the end of opcontrol(). This significantly saves critical section operation time and allows the program to have higher throughput.

2.2 Queue Attachment Strategy

The design is more fleshed out here than in the design document. The operator() function measures time from the beginning of actual operation (trying deq() for LockFree, trying to grab the queue's lock for HomeQueue, and trying to find a free queue for Awesome), then jumps to output information writing and termination after that time limit is reached. Difference from start time is measured with the ¡time.h¿ function difftime().
LockFree operation remains the same as in HW2. HomeQueue first grabs the home queue's lock with TryLock, then performs the same operation as LockFree. After deq() and checksum are finished on the packet, the thread releases the lock and starts from the top of the operation loop.
Awesome mode has each thread start with the queue that they would be attached to in HomeQueue/LockFree, and like in HomeQueue they grab that queue's lock. However, if either TryLock returns that the lock is busy or deq() returns that the queue is empty, the thread will begin iterating over all queues until it can find one that has an open lock AND is not empty. Importantly, after a packet is successfully dequeued the thread releases that queue's lock. This allows other threads to begin pulling other packets even while the first thread is still checksumming. Admittedly this placement of unlock() could also have been done for LockFree, but it makes no difference there because it is only one thread that will ever operate on that queue.

3 Testing Results

3.1 Correctness

I followed through with my decision in the design document to not test the locks or Lamport queue in-depth for any of their invariants, because they have already been tested and proven correct in Homeworks 3a and 2 respectively. However, just to be sure, I set up tests with a fixed number of packets per queue -T across all modes to test the non-dropping and correct processing invariants. These tests output number of packets dispatched as well as total number of packets processed, and the total sum

of checksums. For the program to be correct, the number of packets dispatched must equal the total number of packets processed each session (to ensure non-dropping) and the total sum of checksums must be the same as serial mode across all modes. For the sum of checksums comparison, uniform sessions were compared with serial-uniform, and exponential sessions were compared with serial-exponential. The output (3b_tests/actual_correctness.txt, outputted from the CSIL Linux machine with shell script /3b_tests/corrcommands) was then run through a Python 3 script (3b_tests/correctness_check.py), which outputs any incorrectnesses to 3b_tests/correctness_ver.txt.
There were no discrepancies discovered, so we can safely conclude that the program runs correctly.

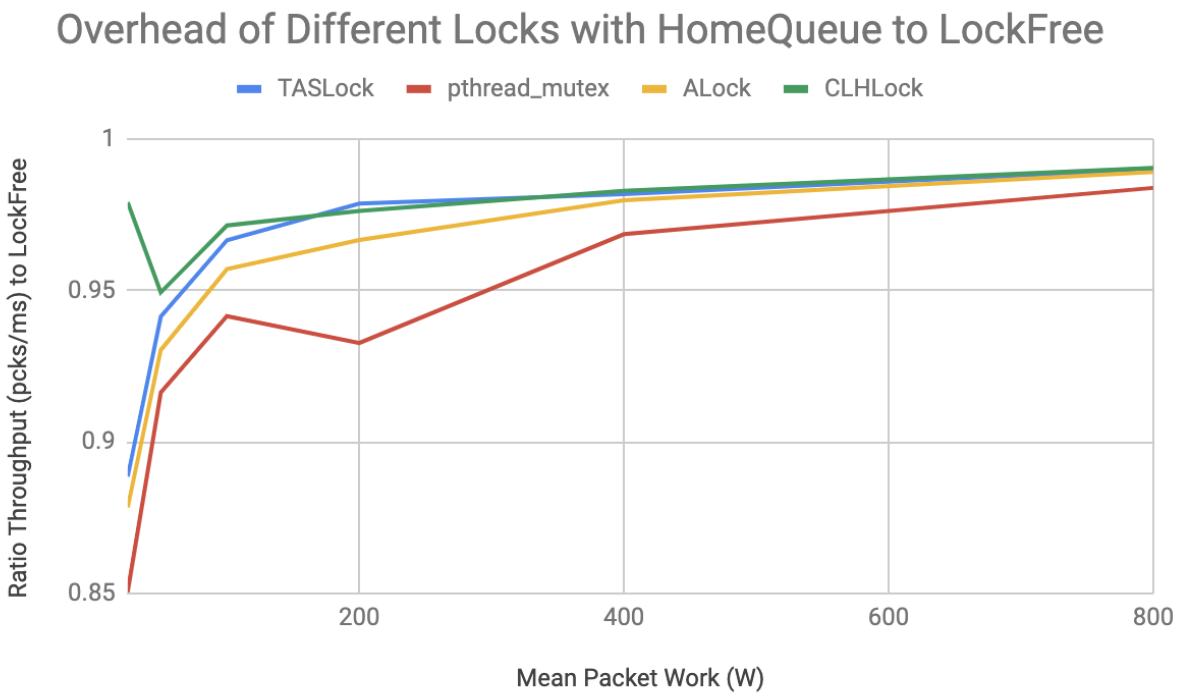
3.2 Performance

All performance tests were performed as directed in the instructions; Awesome Mode tests will be detailed in their section. All trials of all modes in every experiment was repeated 5 times and the statistic averaged across those 5 trials. -M 2000 was picked because trials were observed to have little variance while keeping the runtime of experiments from being too high.
All performance test outputs are available in hw3/3b_tests, and data was put through Jupyter notebooks using Python 3 and the Pandas package to output the needed data tables. The notebooks can also be found in 3b_tests and are labelled according to their experiment.

3.2.1 Idle Lock Overhead

First is the measurement of lock overhead via comparison of HomeQueue throughputs to LockFree. The data and graph for this are below.

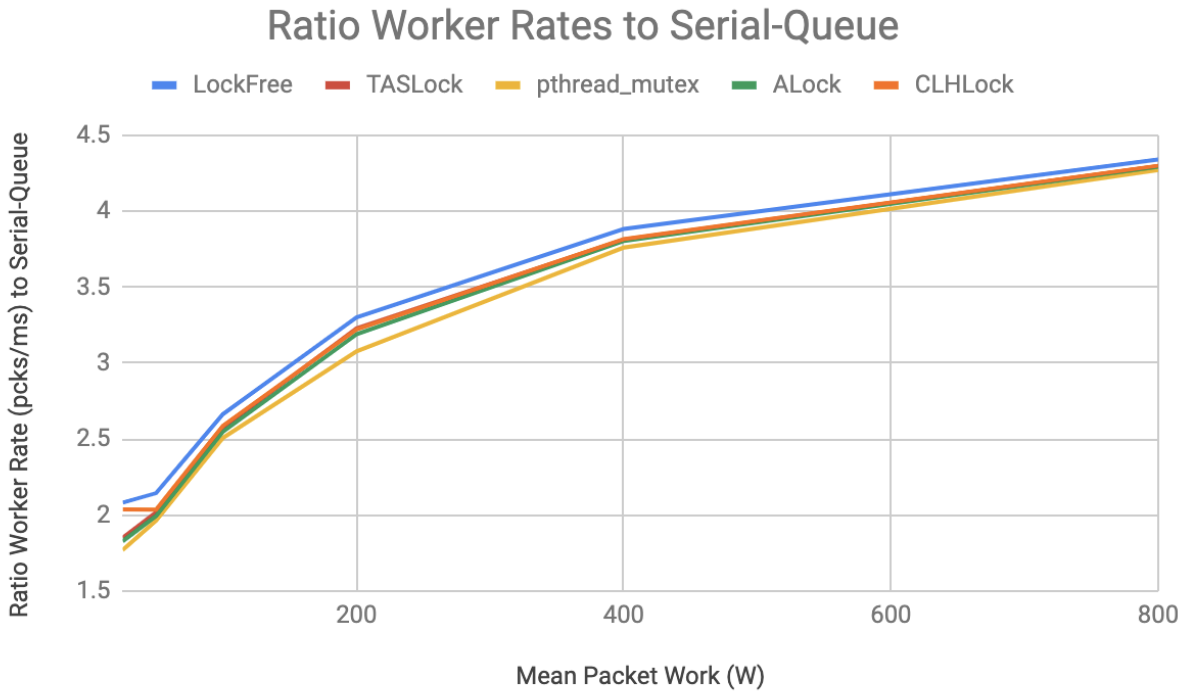
W	LockFree	TASLock	pthread_mutex	ALock	CLHLock
25	2382.9148	0.8885827978	0.8504486186	0.8784181037	0.9789774691
50	1852.2476	0.9413024479	0.9162819269	0.9303239481	0.949322596
100	1520.38	0.966500809	0.9414619371	0.9569474737	0.9713714992
200	1123.3679	0.9786009552	0.9325753389	0.9665848561	0.9761388055
400	736.7183	0.9817251723	0.968504651	0.9796555888	0.982758946
800	434.1277	0.9899184042	0.9837656063	0.9890119889	0.9903804802



Contrary to my hypothesis in the design document but in agreement with the results of the lock overhead testing for Part a, HomeQueue across all locks increased in speed as W increased, converging to a speedup ratio of about .99. At all W levels CLHLock, TASLock and ALock had higher throughput levels than pthread_mutex, with TASLock and CLHLock having neck and neck performance while ALock, initially lagging TASLock by .01 (in speedup ratio), caught up to only lag TASLock by .0009. This too is consistent with Part (a) overhead testing.
I hypothesize again that ALock is slower than CLHLock because both have the same queue creation overhead but the getAndIncrement() call that ALock uses to change its tail pointer is slower than the testAndSet() call that CLHLock uses for the same.

Below are the data and graph for comparison of worker rates to the serial-queue results of HW2.

W	LockFree	TASLock	pthread_mutex	ALock	CLHLock
25	2.081259374	1.849371278	1.77000416	1.828215913	2.037506035
50	2.143991392	2.018144345	1.964500564	1.994606536	2.035339474
100	2.663430234	2.574207476	2.507518187	2.548762833	2.587180219
200	3.299538023	3.228931061	3.07706779	3.189283485	3.220807104
400	3.882293941	3.811345688	3.760019738	3.803310957	3.815359101
800	4.339790387	4.296038375	4.269336522	4.292104722	4.298043688

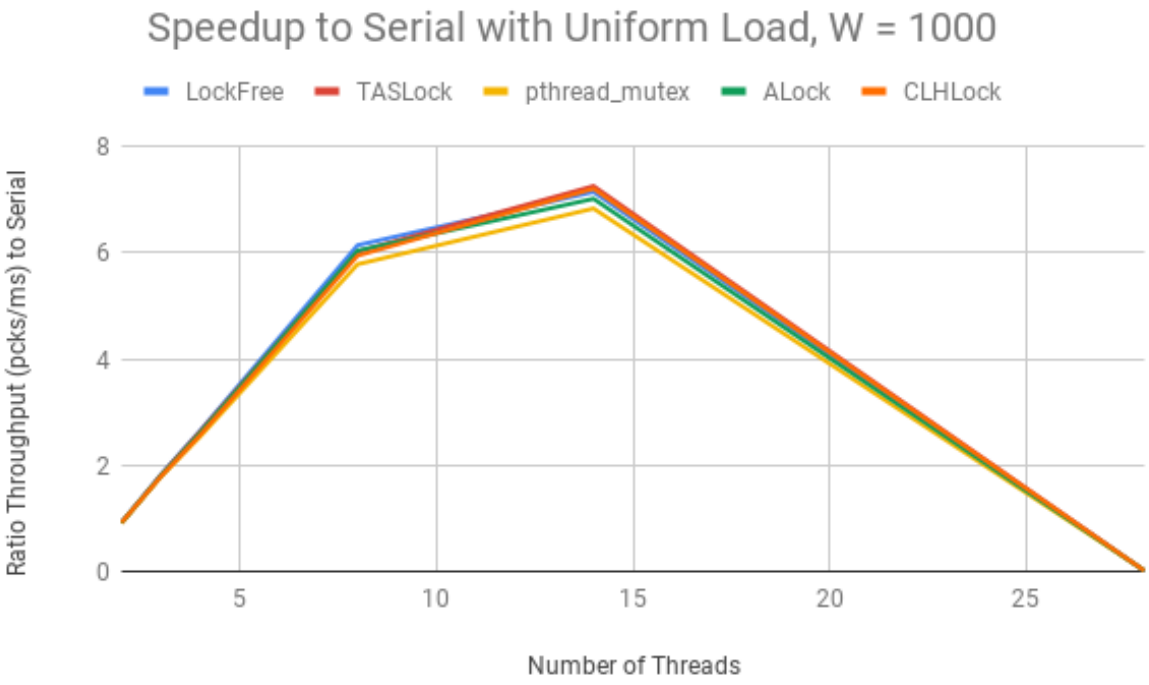


All modes experienced a higher worker rate than serial-queue, starting at around $2\times$ at $W = 25$ and ending as high as $4.33\times$ at $W = 800$. As is expected due to overhead, all HomeQueue modes had a smaller worker rate increase than LockFree. Between the HomeQueue modes worker rate increase was fairly equivalent, with TASLock maintaining the highest throughout except for $W = 800$, where pthread_mutex was highest. This is due to the fact that one lock is only ever accessed by one thread, making contention a non-issue and removing the worker rate advantages related to contention for ALock and CLHLock.

3.2.2 Speedup with Uniform Load

First below are the data and graph for speedup of LockFree and HomeQueue with various locks to serial with a uniform load, at average packet work of 1000 ($W = 1000$).

nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	390.3327	0.9247213979	0.9167031099	0.9095891787	0.9196662232	0.9226454765
3	401.8437	1.82458752	1.809401267	1.784258656	1.800645873	1.789420862
4	410.28	2.65266769	2.638558058	2.54309106	2.615184752	2.57392878
8	410.2719	6.146463845	6.024283408	5.786854279	6.042499133	5.949567835
14	409.9971	7.150102769	7.261214287	6.838094708	7.021358444	7.211014419
28	409.9904	0.01662819422	0.0177260248	0.01868165694	0.01770260962	0.0173462598

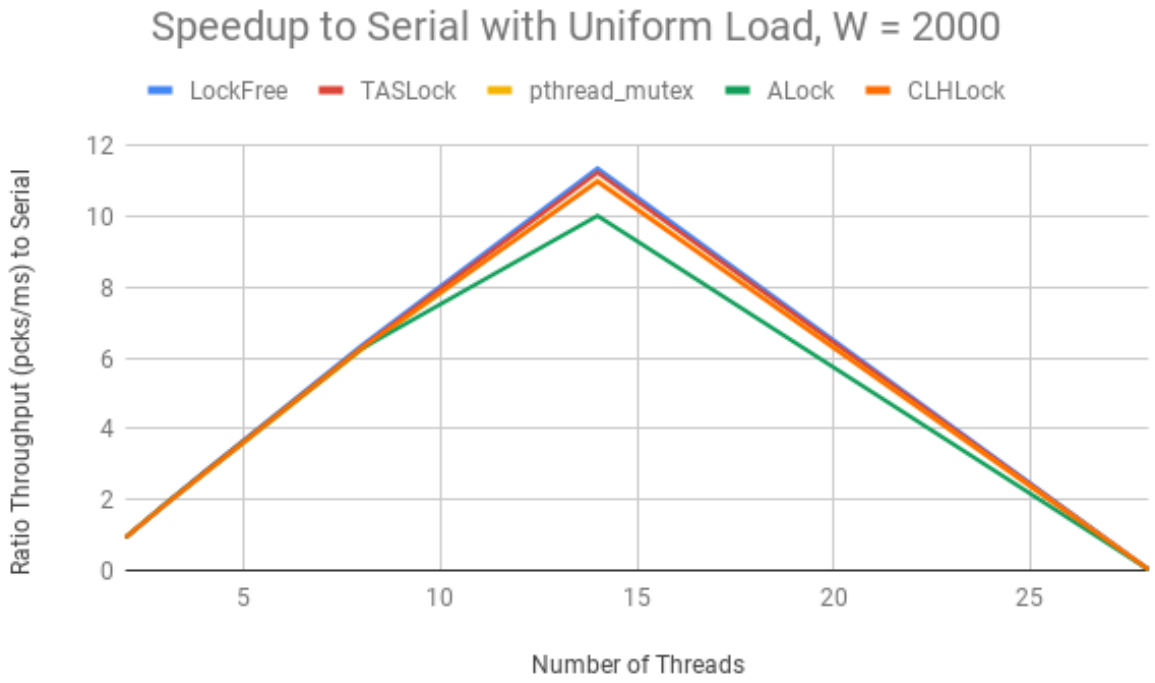


In line with expectations, LockFree performed the same as in Homework 2.
All parallel modes followed approximately the same throughput speedup, going from around 0.9× serial at 2 sources to 7× serial at 14 threads, back down to 0.02× serial at 28 threads. The collapse in speedup from 14 to 28 threads can be explained by the fact that SLURM only has 14 nodes usable by testers - beyond that, multiple threads can populate one core and context switching between threads of the same core ruins parallelism.

Mostly following expectations of lock overhead, LockFree performs better than all the lock modes except at 14 and 28 threads. At 14 threads, LockFree is outperformed by TASLock and CLHLock, and at 28 threads all versions of HomeQueue outperform LockFree.
At 28 threads, the difference between all the parallel modes is at its smallest, and there could likely have been larger relative variances in throughput between trials depending on how the SLURM cluster handled context switching during that session. So I venture to say that at that point the parallel modes can be regarded as approximately equal, with many more trials needed to even out variance if we want to more granularly distinguish them.
However, the better performance of TASLock and CLHLock at 14 threads is on a wider margin, and there is no inter-thread context switching at that many threads. This defies my expectations and also goes against the insights of the Idle Lock Overhead experiment. The only two explanations I can think of are that the additional delay added with lock handling reduced the necessary memory barrier synchronization delay, or that at that specific number of threads the variance was too high for 5 trials to even it out. Future experimentation would extend the -M runtime limit to figure out which of those two explanations are more valid.
What did follow my initial hypothesis, though, was that speedup rankings of the HomeQueue modes generally went in the order (from fastest to slowest) observed in Part (a) overhead testing and the Idle Lock Overhead test above: TASLock, CLHLock, ALock, pthread_mutex.

Next are the data and graph for the same speedup experiment but with $W = 2000$.

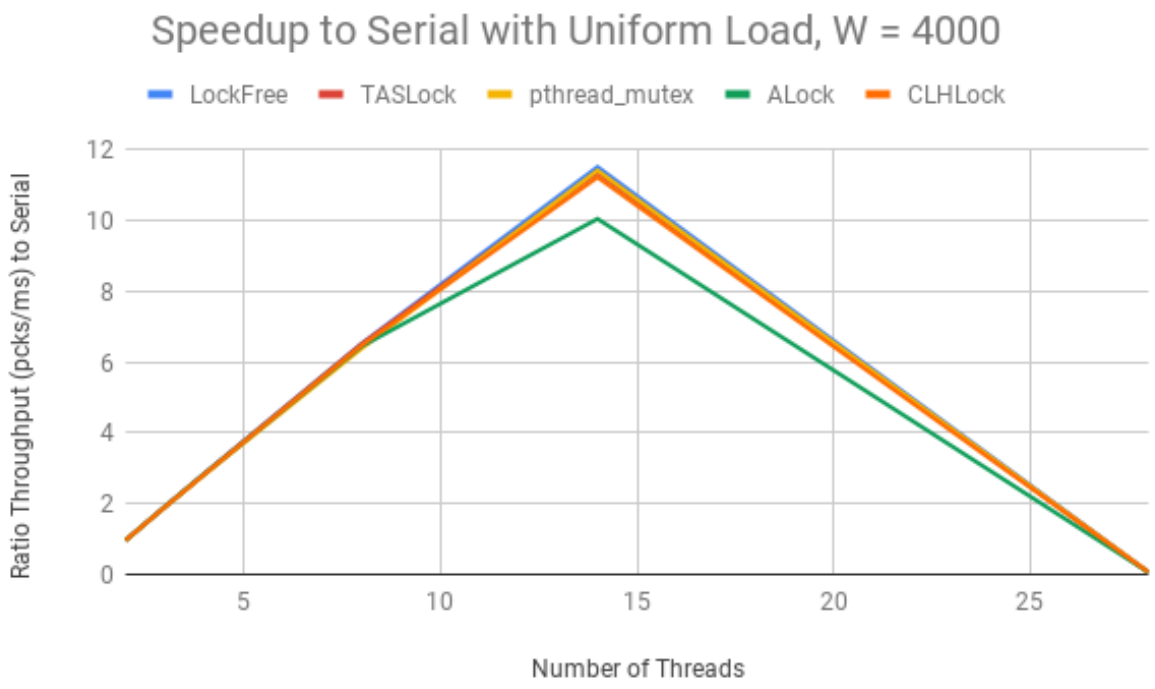
nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	208.597	0.9342061487	0.9306744584	0.9272098832	0.9297180688	0.9339961744
3	208.4249	1.875262984	1.865228675	1.850684347	1.859807537	1.847950509
4	208.1612	2.784192731	2.775221319	2.722181655	2.752418318	2.740020715
8	208.6868	6.355712484	6.294889279	6.235243437	6.270722921	6.281652218
14	208.745	11.36562121	11.24689933	11.01450765	10.02304055	10.98617117
28	208.8188	0.0358315439	0.03573097825	0.03454909232	0.03315984959	0.03565244126



Following my expectations before testing, the speedup of all LockFree and HomeQueue modes over serial mode improved substantially with this jump in W. This can be attributed to dispatch throughput in serial mode being dependent on processing throughput, since serial mode alternates between dispatch and processing on one thread. Parallel modes, with their dispatch speed independent of processing speed, would not experience the full effects of this bottleneck. The numbers back this up - in this case, at 14 threads serial mode experienced a 49.1% decrease in throughput from $W = 1000$ to $W = 2000$ - approximately halving it, inversely proportional to the doubling of W. LockFree and HomeQueue modes, on the other hand, only experienced approximately a 20% average decrease at the same number of threads and with the same change of W.
What was not predicted beforehand was the slowdown of ALock and CLHLock relative to pthread_mutex, especially the substantial relative slowdown of ALock. The only thing that has changed regarding the locks are distance from lock acquisition to lock release in a single attempted dequeue. Based off of this, I extend the tenuous explanation that the address of a thread's slot in the queue, which is stored as part of the lock structure for my implementation of both queue-based locks, was cached during lock(), but for longer periods between lock operation (longer work time for packet from larger W) that cache line may have been more likely to be removed before unlock() was called, by another context on the same core. Thus, upon calling unlock() and not having the slot address readily available on cache, the thread would have to read from memory. The additional W may have made this time cost, which pthread_mutex does not have, more significant than the time benefit that ALock and CLHLock have of not using system calls.

Next are the data and graph for $W = 4000$.

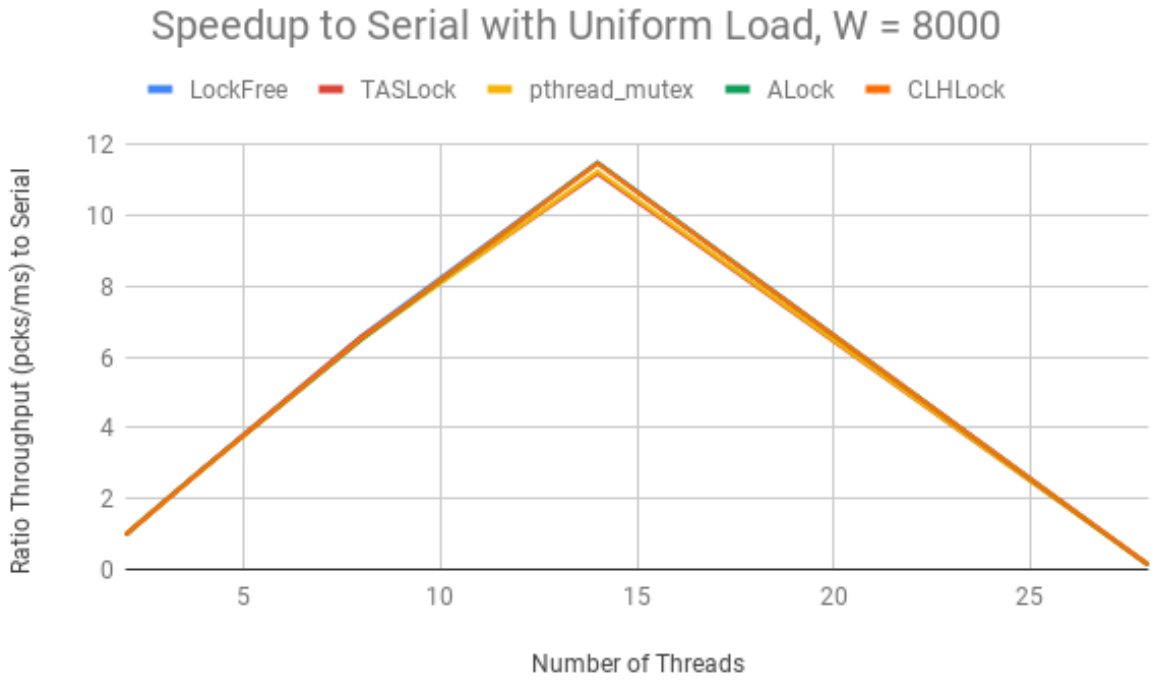
nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	104.8873	0.966261883	0.964523827	0.9637019925	0.9642740351	0.9650758481
3	105.0109	1.916690553	1.912911898	1.90503462	1.909006589	1.903229093
4	105.1216	2.848879774	2.841221975	2.818275217	2.834068355	2.825948235
8	104.9835	6.521356213	6.512883453	6.383635524	6.448484762	6.469741436
14	105.1338	11.51759282	11.34816491	11.4151272	10.04904988	11.22508556
28	105.134	0.07006296726	0.0689558087	0.06954648354	0.06619457074	0.06945326916



Here pretty much the same picture is given as for $W = 2000$, meaning the speedup ratios themselves are approximately the same and the relative ordering of speedup ratios is also the same. The lack of change in ratios to serial but the continued inversely proportional decrease of serial throughput means that this inverse proportion of throughput to work is also now applicable to LockFree and HomeQueue. This goes against my expectations prior to testing, and the best explanation for this happening is that the work bottleneck, while continuing to only exist on worker threads, is now large enough to become a direct bottleneck for the entire process.

Last are the data and graph for $W = 8000$.

nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	52.6994	0.9818707613	0.9800680843	0.9796411344	0.9808555695	0.9810775834
3	52.8352	1.937906926	1.936619905	1.932808052	1.933684362	1.927777315
4	52.7584	2.878967141	2.874996209	2.86462061	2.872782344	2.868191606
8	52.7361	6.590223775	6.579210446	6.502746695	6.503095602	6.53248913
14	52.8825	11.49575947	11.1900988	11.24790432	11.49017917	11.47761358
28	52.921	0.1312390166	0.1437860207	0.1264469681	0.1267209614	0.1379887001

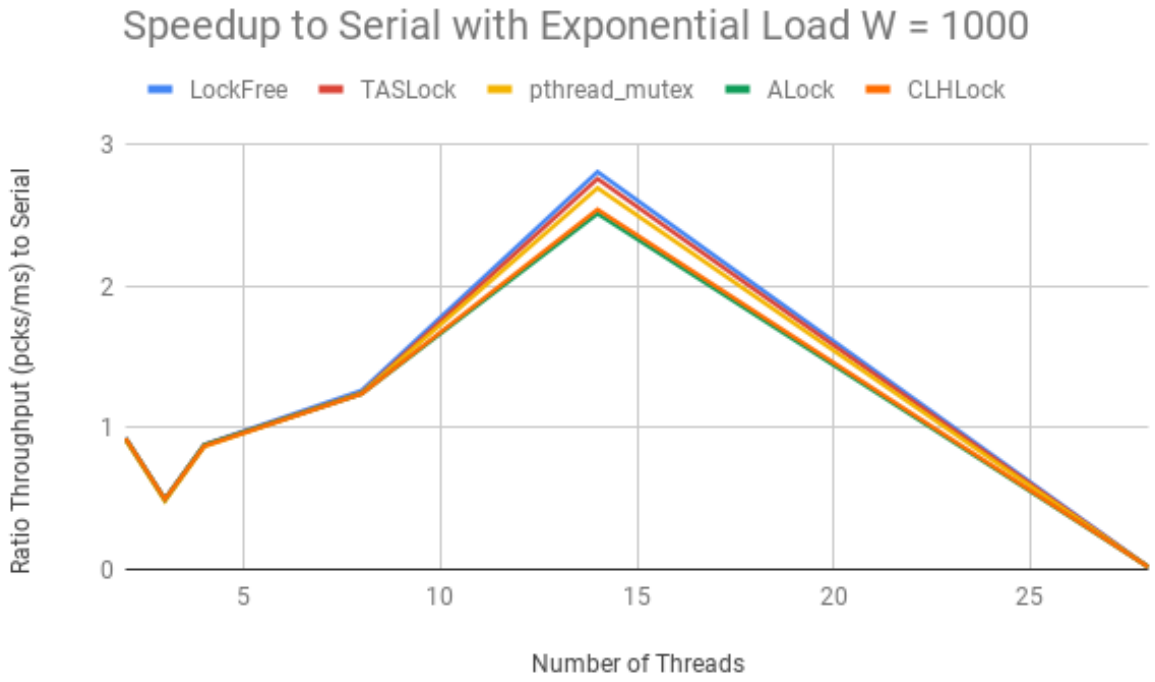


The ratios to serial remain approximately the same, following my explanation for the $W = 4000$ data of W becoming a direct bottleneck on the parallel modes at higher amounts.
However, ALock and CLHLock are now faster than pthread_mutex, and TASLock becomes the slowest at 14 threads. To keep in line with my explanations of why the queue-based locks were relatively slower than pthread_mutex at $W = (2000, 4000)$, I can only say that at $W = 8000$, these differences no longer matter as the time taken to work on the packet is long enough to dwarf any lock-based overhead.
To support this, note that LockFree, while still being relatively faster than the HomeQueue modes, is much less so at this W, and is even outpaced by TASLock and CLHLock at 28 threads.

3.2.3 Speedup with Exponential Load

First below are the data and graph for speedup of LockFree and HomeQueue with various locks to serial with an exponential load, at average packet work of 1000 ($W = 1000$).

nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	387.5892	0.930476133	0.9220798722	0.9160887352	0.9260776616	0.9272332666
3	290.5922	0.4971406665	0.4947871966	0.4823653216	0.4950057159	0.4955453037
4	322.0651	0.881385161	0.8788701415	0.8683259999	0.8785882109	0.8721699433
8	371.6076	1.263811612	1.246361485	1.244384399	1.241003951	1.239886375
14	384.4981	2.80904457	2.760090622	2.694967283	2.513844152	2.543086949
28	398.6686	0.01909405456	0.01779598393	0.01690802837	0.0181953131	0.01795877579

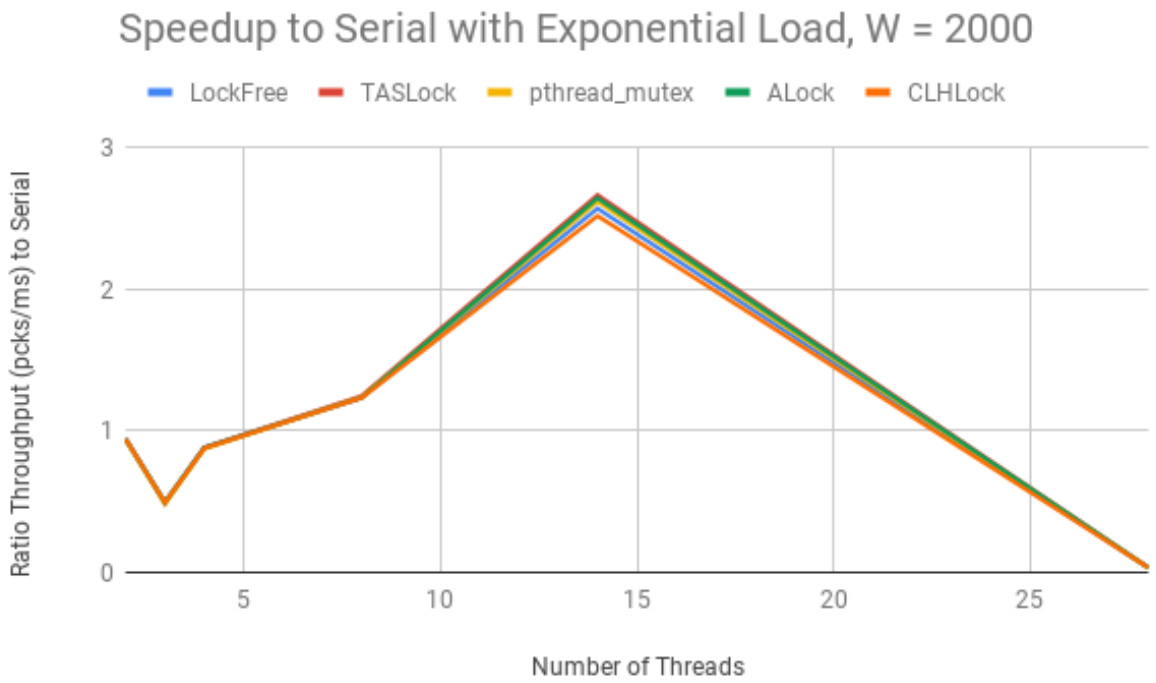


LockFree here performed the same as in Homework 2. Like in Homework 2, this speedup over serial is substantially lower than with uniform load - at 14 threads, LockFree is 60.7% slower at $W = 1000$ with exponential load than with uniform load. Also following expectations is that up until a certain number of threads (which here is 8), the parallel modes are actually slower than serial mode due to parallelization benefits being crippled by a bigger proportion of threads having unequal loads. Serial mode also takes a throughput dip at those lower source amounts due to a higher proportion of packets being larger load.

The HomeQueue modes were again slower than LockFree. However, pthread_mutex being faster than ALock and CLHLock at 8 threads and above was not hypothesized - this could be due to the same change of significances of different types of overheads that we observed with uniform load at $W = (2000, 4000)$.

Next are the data and graph for the same modes and exponential load distribution with $W = 2000$.

nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	205.7319	0.9471389707	0.9419389993	0.938922452	0.9417654724	0.9429650919
3	148.2632	0.4978780979	0.4858238592	0.484468162	0.4971179632	0.4963342218
4	164.8873	0.883096515	0.8820473135	0.8729150153	0.8804656271	0.8775472702
8	190.9406	1.235988051	1.246618582	1.237527273	1.233703571	1.232967216
14	202.8261	2.571520135	2.666893462	2.620299853	2.644970741	2.519311371
28	204.9781	0.0346012574	0.03501398442	0.03287082864	0.03491251017	0.03550818356

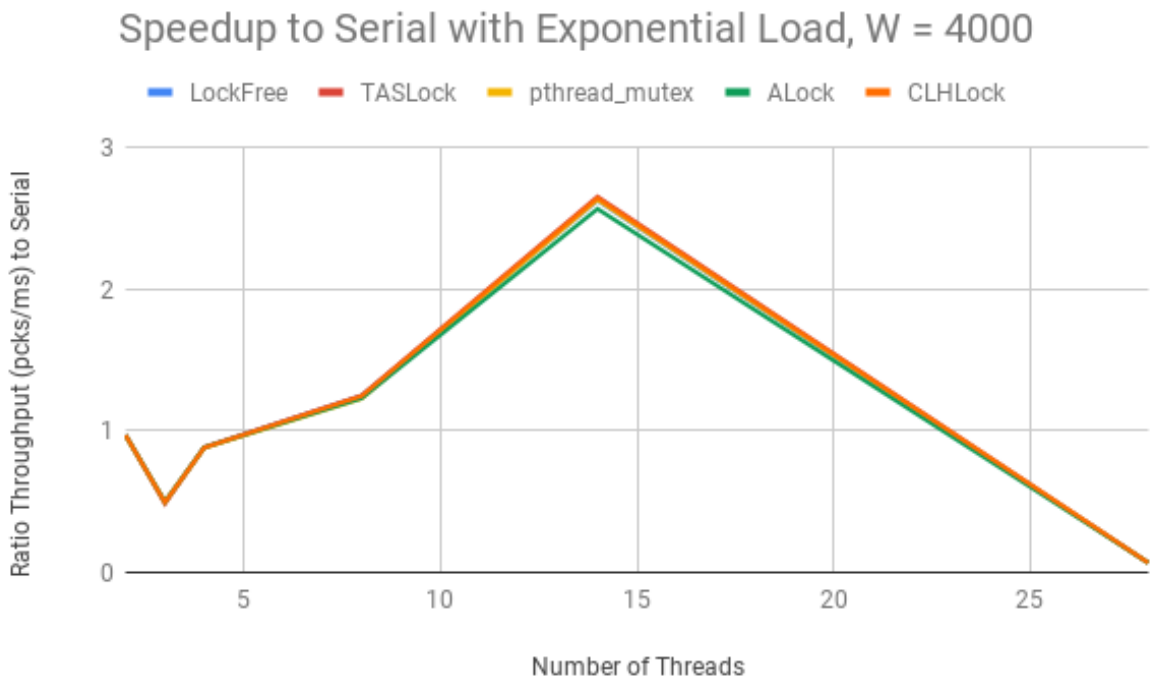


Pretty much the same ratio numbers remain, with the closing of the speedup deficiency of the queue-based locks relative to the other two locks supporting the idea of relative overhead changes that I posited above in exponential, $W = 1000$ and in uniform, $W = (2000, 4000)$. Absolute throughput of all modes also decreases in inverse proportion to mean work increase, which is as expected because the actual mean work

experienced in an exponential load session should be close to the mean work specified.

Next are the data and graph for $W = 4000$.

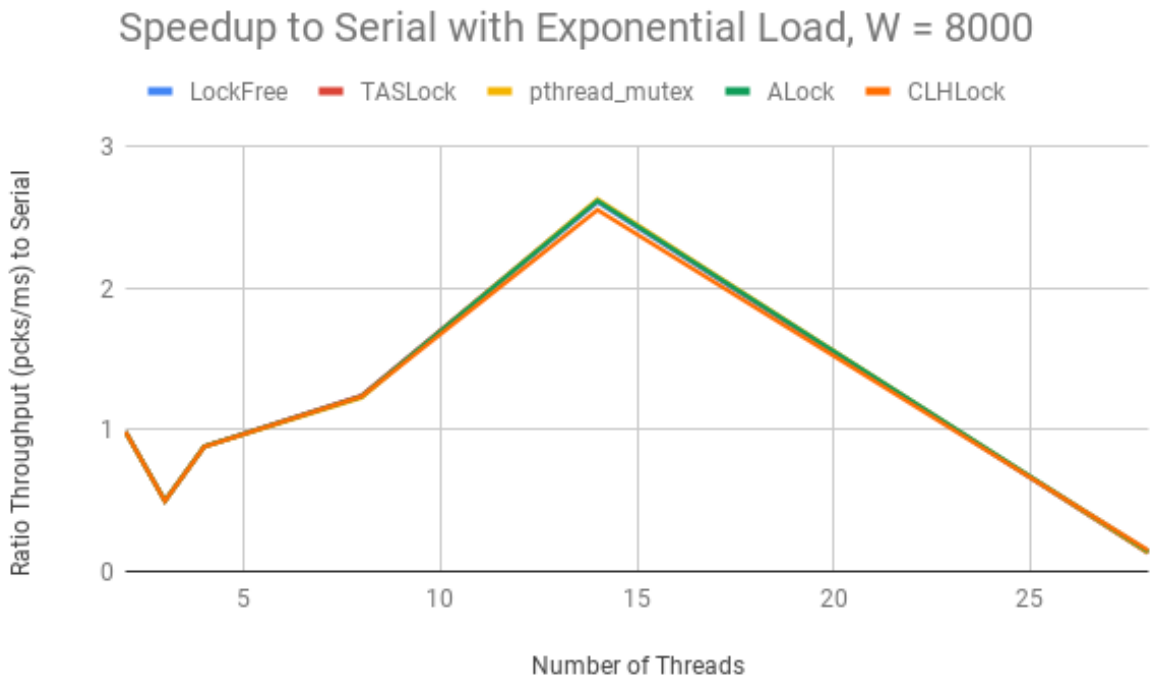
nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	104.4963	0.9711903675	0.968774014	0.9674763604	0.9704774236	0.9707204944
3	75.0794	0.4988212479	0.4855552921	0.4980274216	0.498055392	0.4969592192
4	83.487	0.8839412124	0.8842622205	0.8802663888	0.8832297244	0.8817899793
8	96.6151	1.248438391	1.249250893	1.228059589	1.230958722	1.245862189
14	102.909	2.628760361	2.654969925	2.632628827	2.568604301	2.645405164
28	104.0531	0.06806140326	0.06930403803	0.06812675451	0.0691464262	0.06904839933



The same ratios remain and the absolute throughput decreases in the same proportion to work increase as before, as expected. Mean work increase has closed the gaps between parallel locks even more, also as expected.

Last are the data and graph for $W = 8000$.

nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	52.6281	0.9876510837	0.9859105687	0.9840902484	0.9854754399	0.9858003614
3	37.5828	0.4987653927	0.4979857807	0.4991112956	0.4989942208	0.4979299041
4	41.9452	0.8822964249	0.8833287241	0.8812736618	0.8832071369	0.8808707552
8	48.505	1.24040202	1.242076075	1.227089991	1.235678796	1.235548912
14	51.6564	2.609152012	2.622362379	2.630297504	2.618411271	2.55428369
28	52.346	0.1334371299	0.1300863485	0.1326156726	0.1412887327	0.1498108738



Again the same ratios remain and the throughput decrease is as expected, and the gap between modes has closed so much that they are mostly indistinguishable on the line graph.

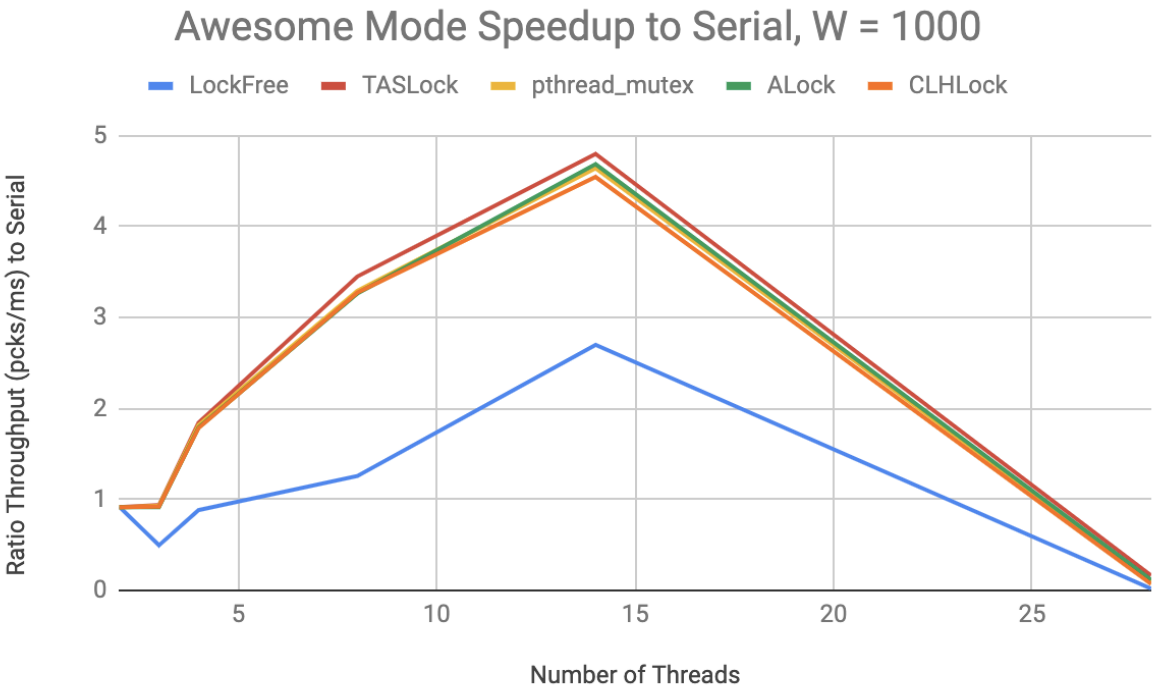
3.2.4 Speedup in Awesome Mode, with Exponential Load

My implementation of Awesome Mode has the most ideal improvement over the one-thread-one-queue modes when loads are unbalanced between queues. In this case it would be best for threads with less work to transition to processing queues with more work when their own queue is empty. So, I conducted the exponential speedup experiment again but this time with LockFree and Awesome Mode with all of the locks.

I expected that LockFree would be an adequate comparison measure instead of having to also add data for HomeQueue sessions, because throughput for LockFree and HomeQueue would be fairly close. The results of the HomeQueue with exponential load experiment vindicate this.

First below are the data and graph for speedup of LockFree and Awesome with various locks to serial with an exponential load, at $W = 1000$.

nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	391.941	0.9188678908	0.9143256255	0.9127508987	0.9141299328	0.9135854631
3	290.6204	0.4959548607	0.9357154556	0.9289361655	0.9142458685	0.9194020791
4	322.0268	0.8825426952	1.843084799	1.816756556	1.790949387	1.786578633
8	371.7496	1.257128185	3.451411649	3.293522844	3.265478968	3.272117307
14	394.4441	2.698371455	4.799224022	4.643832168	4.686505135	4.545673519
28	398.6529	0.01813984045	0.1619692218	0.09822805754	0.1154889379	0.0692053162

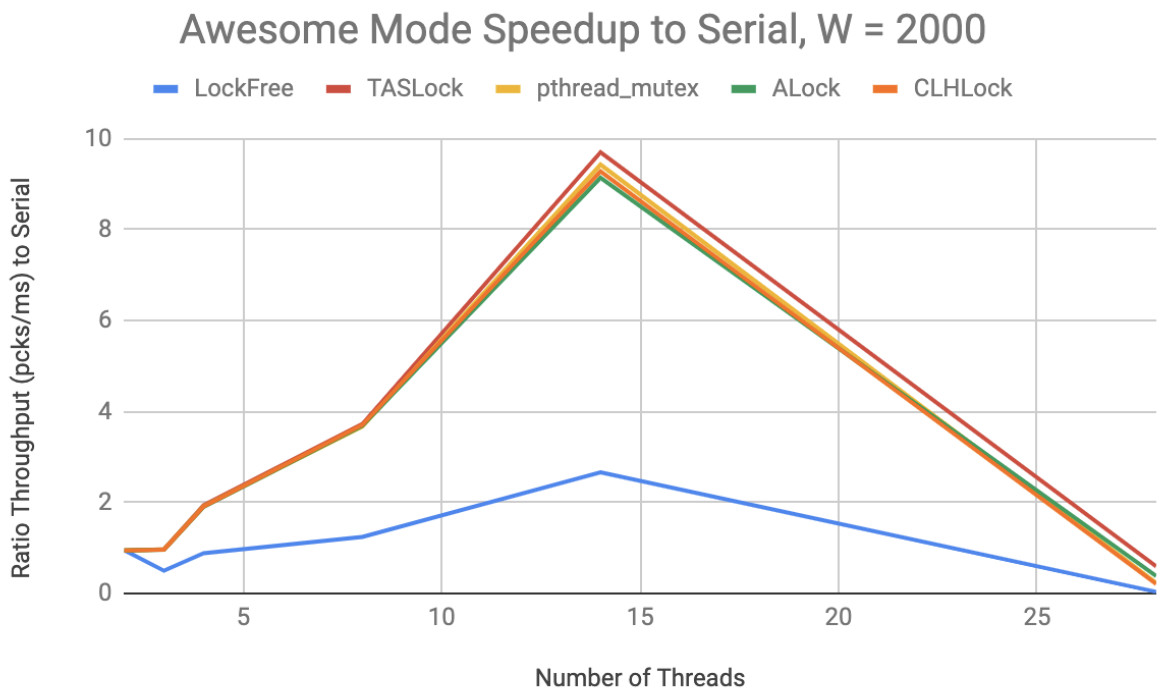


Immediately noticeable and as hypothesized is a significant jump in throughput from LockFree to Awesome. Specifically, at 14 threads, the best performing lock TASLock was 1.78× faster than LockFree. Also noteable, and something that I did not think about before testing, is that unlike LockFree and Home-

Queue, Awesome throughput for all locks is nondecreasing until after 14 threads (after which throughput decrease is due to lack of additional cores). This indicates that the dip in throughput caused by a higher proportion of threads with higher load that the other parallel modes experienced was done away with in Awesome Mode by the load balancing - there was a higher proportion of threads with lower load as well. TASLock performed better than the queue-based locks, which at first is surprising. I hypothesized that the opposite would happen, that TASLock would be slower than the queue-based locks due to the higher contention that comes with letting multiple threads grab a single queue's lock. But a very good reason for this would be that at this lower mean work rate, absolute load imbalance is lower as well, meaning that there is likely less wandering of threads to other queues and less contention. There is this lower contention overhead, and also the higher memory overhead of CLHLock and ALock having to create slots in the lock for every thread. Both combine to make TASLock faster at this lower mean packet work.

Next are the data and graph for $W = 2000$.

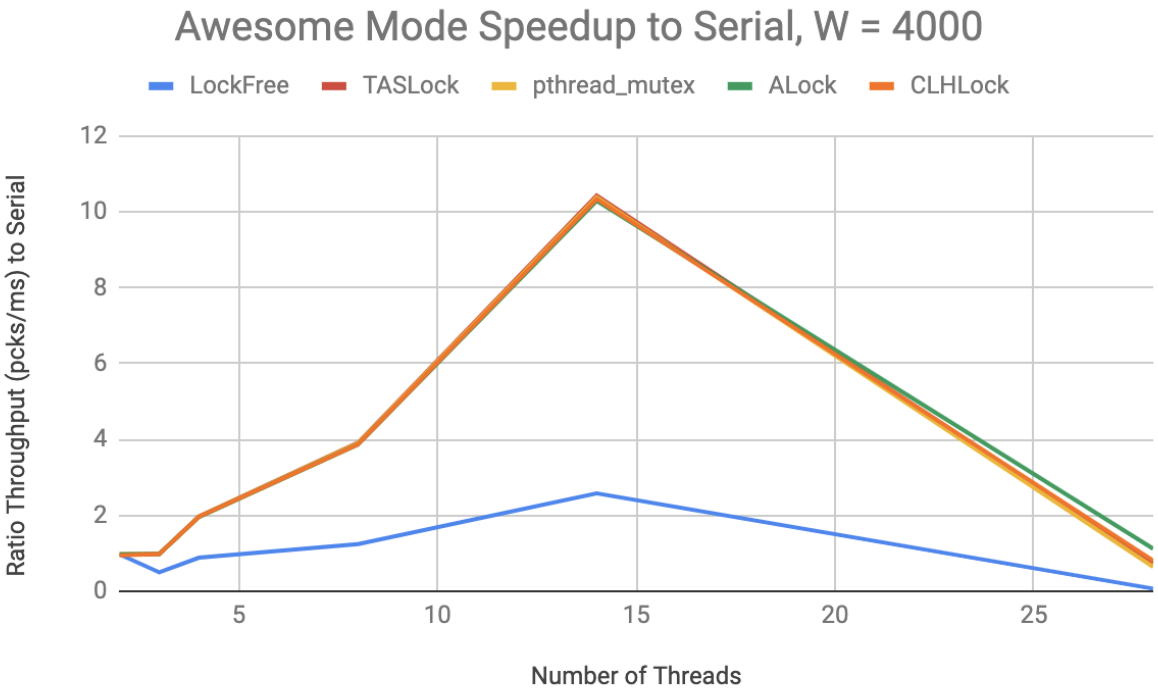
nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	205.305	0.948940357	0.9451727917	0.9446569738	0.9454075644	0.9454981613
3	148.3544	0.4983013648	0.9689109322	0.9657354281	0.9606469373	0.9625066732
4	164.9738	0.8833869378	1.936633575	1.915774505	1.903694405	1.908152082
8	190.7093	1.24334786	3.727887418	3.669452932	3.688360243	3.702389973
14	202.8389	2.664421371	9.701983199	9.428726443	9.143107165	9.281548559
28	205.0764	0.03359869785	0.5928780689	0.2289722269	0.3789826621	0.2087734132



The jump in throughput from LockFree to Awesome grows even wider, with TASLock being 3.64× faster than LockFree at 14 threads now. I expected this widening beforehand, as larger mean work leads to a large work variance between queues for exponential load and the other modes would cope much worse with this than Awesome.
TASLock has a higher absolute throughput ratio increase over CLHLock here than before - $(0.52 \times serial)$ pcks/ms at $W = 2000$ versus $(0.25 \times serial)$ at $W = 1000$. However, the ratios themselves have more than doubled from $W = 1000$ to $W = 2000$, so the relative throughput ratio increase of TASLock over CLHLock remains about the same.

Next are the data and graph for $W = 4000$.

nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	104.4512	0.9707260424	0.9701765035	0.9692918798	0.9701324638	0.9445339067
3	75.0616	0.498495902	0.983469577	0.980655888	0.9785456212	0.9796527119
4	83.4788	0.8830313804	1.968938221	1.963649454	1.956362573	1.95917167
8	96.7174	1.241052799	3.912133701	3.897512754	3.867503676	3.869428872
14	102.8734	2.578294292	10.42866961	10.38281519	10.28814251	10.33391431
28	104.0061	0.06805273921	0.737661541	0.6419383094	1.121052515	0.8067901787



The ratio of Awesome to LockFree and to Serial is now about the same as it was at $W = 4000$. This lack of additional relative speedup was not expected, but is likely because all threads are now being kept busy by the higher work, and there is no additional speed optimization via load balancing to be found. The gap between locks is, as expected, now minimal.

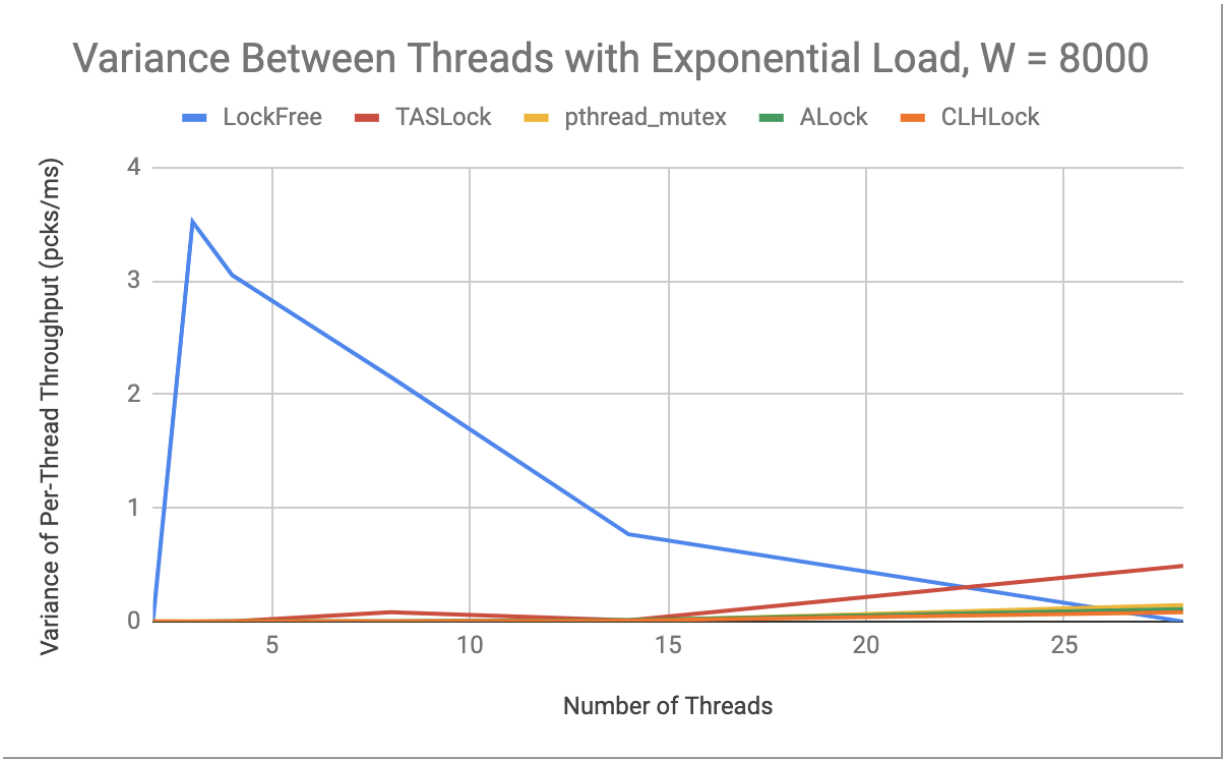
Last for speedup data are the data and graph for $W = 8000$.

nthreads	Serial	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	52.6603	0.9862439067	0.9851747901	0.9843867202	0.9852013756	0.9830688393
3	37.5802	0.4993826536	0.9955960852	0.9950798559	0.9941405315	0.9948110973
4	41.8885	0.8846246583	1.991685069	1.986955847	1.98399322	1.986793511
8	48.4992	1.224220193	3.876055688	3.969745893	3.95332088	3.958611688
14	51.6441	2.592284501	10.76898426	10.7788576	10.66282112	10.71710805
28	52.3712	0.1435483625	1.839759257	1.124942717	0.8146805878	0.7055366308

Following the observations from $W = 4000$, the ratios remain about the same and locks are fairly indistinguishable.
Through all of these work levels, however, TASLock has remained faster than the queue-based locks. I edit the explanation I made at $W = 1000$ to say that the locks most likely did not experience high enough contention at *any* work level for the contention downsides of TASLock to outweigh the memory overhead of the queue-based locks.

I also thought it interesting to observe variances of single thread throughput, to demonstrate that Awesome truly does load balance significantly better than the one-thread-one-queue modes. Below is the variance of single thread throughput between LockFree and Awesome with all locks at the same numbers of threads taken before, taken at $W = 8000$ to see maximal load balancing from Awesome.

nthreads	LockFree	TASLock	pthread_mutex	ALock	CLHLock
2	0.0000151416	0.0000026216	0.0002625256	0.000127292	0.0016480744
3	3.520465695	0.0001323281	0.0003513581	0.000102588	0.0001033916
4	3.05038483	0.0005631384889	0.001478068622	0.0008773062222	0.0003709358222
8	2.154230239	0.08212102028	0.001046500163	0.001505899135	0.001356944653
14	0.7680438293	0.01127460362	0.006742802802	0.01182608316	0.007468089808
28	0.00003064673032	0.4889533784	0.1456316588	0.1078392574	0.08286131014



As I expected, Awesome Mode with all locks maintained extremely low variance at all levels except 28 threads, which again can be forgiven due to context switching. The difference of variance between Lock-Free and Awesome is highest at the lowest amount of threads, and decreases as the number of threads increases. This is reasonable, because a higher amount of threads means proportionally less queues with high workload and proportionally less with low workload as well, giving LockFree some natural load balance that allows it to decrease variance.