Writeup Document

Alex Miller

May 29, 2021

Changes

Changes to the design

I have made the following addresses changes to my project's design, broken down by module:

- Top level modules (serial.c and parallel.c)
  - I modified serial.c and parallel.c to take in an optional flag denoting that the programs should be run in "correctness testing" mode. In this mode, $chksum\_serial()$ and $chksum\_parllalel()$ would not dispatch packets from n sources for M seconds, but $n * M$ packets total. The values these functions would return (with this flag specified) would not be throughput numbers but amalgamated checksums, as described in assignment 2.
- Algorithm implementation level (chksum.c)
  - I modified both $chksum\_serial()$ and $chksum\_parallel()$ to take in the additional flag denoting whether they should generate checksums in correctness testing mode or not.

    If so, the dispatchers for these functions would not rely on a timed approach to halting packet generation, but would instead halt themselves and their threads after generating $M * n$ packets. The long integer returned by these methods would not represent their respective throughputs, but amalgamated checksums.

    Threads launched by the parallel dispatcher would not tally throughput for queues but amalgamated checksums instead. This requires the definition of the following worker methods:
    - $void\ *L\_worker\_test(void\ *args)$
    - $void\ *H\_worker\_test(void\ *args)$
    - $void\ *A\_worker\_test(void\ *args)$

    all of which would implement the behaviors of their namesakes, but instead record amalgamated checksums and not mere throughput. These methods would deviate from their original versions in that they generate amalgamated checksums and only exit once all queues are empty and the closing of the system is signaled. However, the $A\_worker\_test()$ method has the problem that it can't necessarily tell when *all* queues are empty. As such, when indicated, $clear\_queue$ (discussed below) will be responsible for generating returning the amalgamated checksum of any packets left in a queue upon cleanup, and adding that checksum to $chksum\_parallel()$'s return value. This will insure that, if A_worker_test isn't able to reach all packets enqueued by the dispatcher, that it at least generated the correct checksums for all packets it was able to reach.

    $chksum\_parallel()$ would be responsible for setting $worker\_method$ to one of these functions when specified by the correctness testing flag.
  - My original design for both the serial and parallel dispatchers utilized a scheme in which each would repeatedly calculate and check whether they had surpassed the allotted time limit. Once written, however, this method seemed bulky and to incur no small amount of increased overhead and complexity within the packet dispatching loops.
    Now, rather than having the dispatching threads calculate time intervals themselves, they both call the new method $start\_timed\_flag()$, which takes in two arguments:

    - volatile bool *done : a volatile pointer to a malloced flag
    - int M : a integer specifying a number of milliseconds

  This method spawns a new thread that sleeps for M milliseconds before setting the value pointed to by *done* to *true*. The method than detaches the spawned thread and returns.
  Dispatching sections can then just operate so long as the value pointed to by *done* is $false$. In *chksum_serial*(), this value is allocated before the test is run. In *chksum_parallel*, this value is allocated in the *create_queue_pool*() method and pointed to by all *packet_queue_t* instances in the queue pool.
  This method does incur the extra overhead of managing an additional thread, which may complicate performance analysis. However, as that thread's primary task is sleeping, the effect should be negligible.
  - I also steamlined the behavior defined by the Awesome load balancing strategy:
    - Within an infinite while loop, the thread attempts to acquire a queue's associated lock with it's *trylock*() method.
    - If it succeeds, the thread waits to enter the critical section:
      - The thread attempts to dequeue from the queue and binds the result to a pointer called *packet*.
      - The thread then unlocks the queue's lock
      - If $packet \neq NULL$, then the thread generates a checksum for the packet and frees it.
      - At this point, if the thread has been signaled to close, it returns
      - Otherwise, it atomically increments the queue's *through_count* counter
    - Once it has either attempted to acquire the lock and failed or has exited the critical section, the thread checks if it has been signalled to close. If it has it returns. Otherwise, the thread moves onto the next queue.

- Queue data structure (queue.c)
  - My original declaration of my queue-pool initialization method was as follows:

    *packet_queue_t *create_queue_pool(int num_q, int D, char L) : allocate an array of num_q packet_queue_t structs, each of depth D. Each queue should have a lock of type L associated with it. If L does not specify a char code for a valid lock (if L is not t, p, m, or a), then no locks are allocated.*

    This method is now declared as:

    packet_queue_t *create_queue_pool(int num_q, int D, char L, int n) : allocate an array of *num_q* packet_queue_t structs, each of depth D. Each queue should have a lock of type L associated with it. If L does not specify a char code for a valid lock (if L is not t, p, m, or a), then no locks are allocated. Each lock in the queue pool will be initialized to work with n total threads.
  - My implementation also now includes the method:

    long clear_queue(packet_queue_t *Q, bool correct)

    This method frees any packets in a queue – it is used to cleanup allocated system resources. If $correct = true$, then it also generate an amalgamated checksum for the packets left in a queue and returns this value. Otherwise it merely returns 0
  - My original design incorporated a volatile pointer to a boolean that was used to synchronize the exit of worker threads operating off of a queue pool. However this approach proved undesirable – it seemed to detract from multi core performance. Therefore, my design is more similar to that for HW2 in that each queue has a volatile boolean which is used to signal a thread's closing. As such, *packet_queue_t* no longer contains the member $volatilebool * done$, but $volatilebooldone$.

- Lock data structure (lock.c)
  - No changes to report

- Testing script (test_script.sh)
    - I neglected to state that I would vary trial numbers based on the type of packet generator specified by an experiment. Trials using a uniform packet generator will still utilize 5 trials. Trials using an exponential packet generator, however, will utilize 11 trials.
    - Additionally, I will include another file, *correct_test_script.sh*, which utilizes the optional correctness testing scheme I described above. This script will verify that amalgamated checksums are consistent across *chksum_serial*() and *chksum_parallel*(), for all load balancing strategies.
    - Rather than the queue depth of 8 specified in the assignment writeup, I am opting to use D = 32. The purpose of this change is to make the performance of our load balancing strategies easier to reason about − a queue depth of 8 stands to pose a significant performance bottleneck, which I realized following testing. Using D = 8, systems were rarely able to run my LockFree implementation with faster than a 2.5x speedup, and consistently performed worse than serial performance. Using D = 32 alleviated the bottleneck posed by queue depth − my LockFree implementation was able to put up as much speedup as in HW2. Removing this bottleneck allows me to reason about the advantages and disadvantages of different load balancing techniques. I still include data from past trials where D = 8, for reference.
- These changes are consistent with my previously stated invariants

Changes to Test plan:

As mentioned above, I will not try to verify if my Awesome load balancing strategy is strictly consistent with my serial implementation. As the Awesome strategy doesn't map worker threads to queue, the possibility remains that the threads exit our correctness testing implementation prior to generating checksums for all generated packets. As such, worker threads themselves might not generate an amalgamated checksum that is consistent *chksum_serial*(). This discrepancy is mitigated by the use of *clear_queue*(). For now, I am deeming this a sufficient test of correctness for my Awesome load balancing strategy. To desire more would mean having to alter the structure of the strategy as to make it a non-meaningful test of correctness.

Results

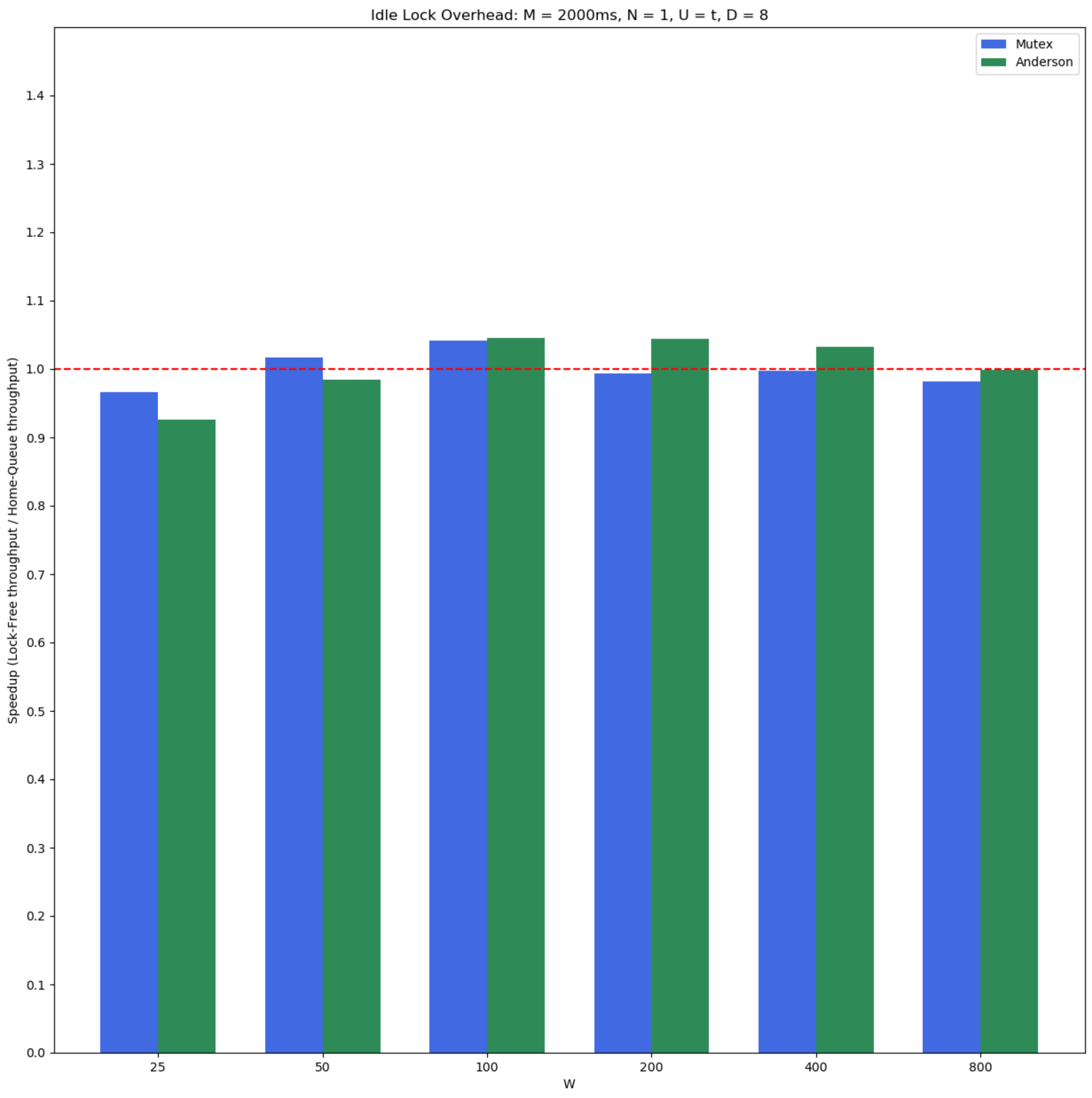
As of writing this, *correct_test_script.sh* confirms that my implementations are consistent. Moreover, testing revealed that while timing my implementation using the StopWatch_t type was inconsistent, my timed flag implementation was consistent; For M = 2000 ms, test_script.sh should complete after 80 minutes. When using stopwatch based dispatchers, the script did not complete its task within 90 minutes; when using my timed flag, the test script completed its run in at most 82 minutes.

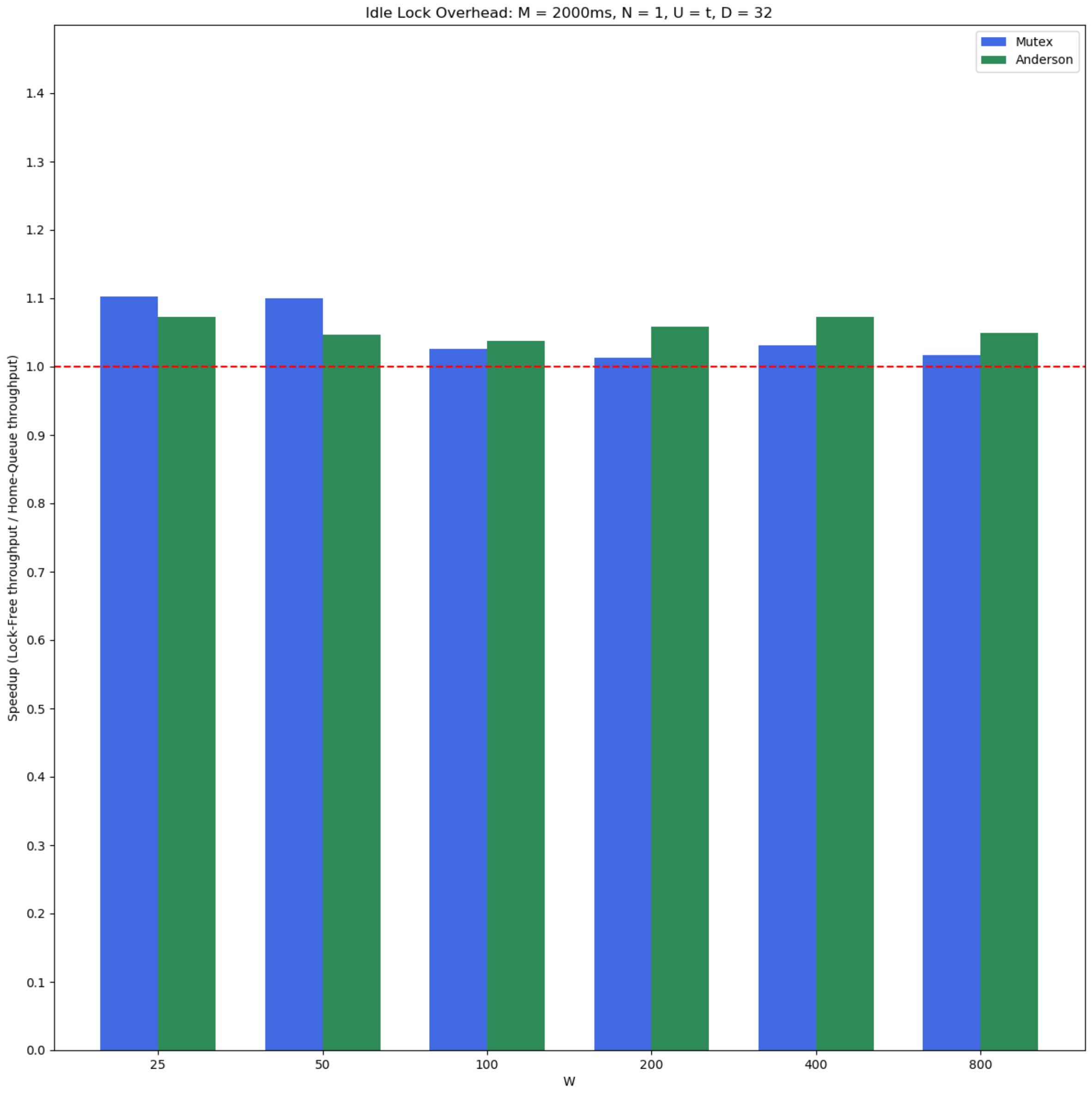
The raw performance data for the Idle Lock Overhead and Speedup tests are stored in two folders; performance data for trials where $D = 8$ are in the folder called $HW3b/hw3b/exp\_data\_8$; performance data for trials where $D = 32$ are in the folder called $HW3b/hw3b/exp\_data\_32$. overhead.csv describes the data from experiment 1. speedup.csv contains speedup data from experiments 2, 3, and 4.
This data was collected by running implementations for given number of trials, calculating measurements per trial, and taking median values. As such there is no other collected data other than median speedup and throughput. That data was not recorded as it did not serve the purpose of analysis and would've have complicated my data storage scheme.

As per how many trials I utilized; for experiments utilizing uniform packets, I used 5 trials; for experiments utilizing exponential packets, I used 11 trials.

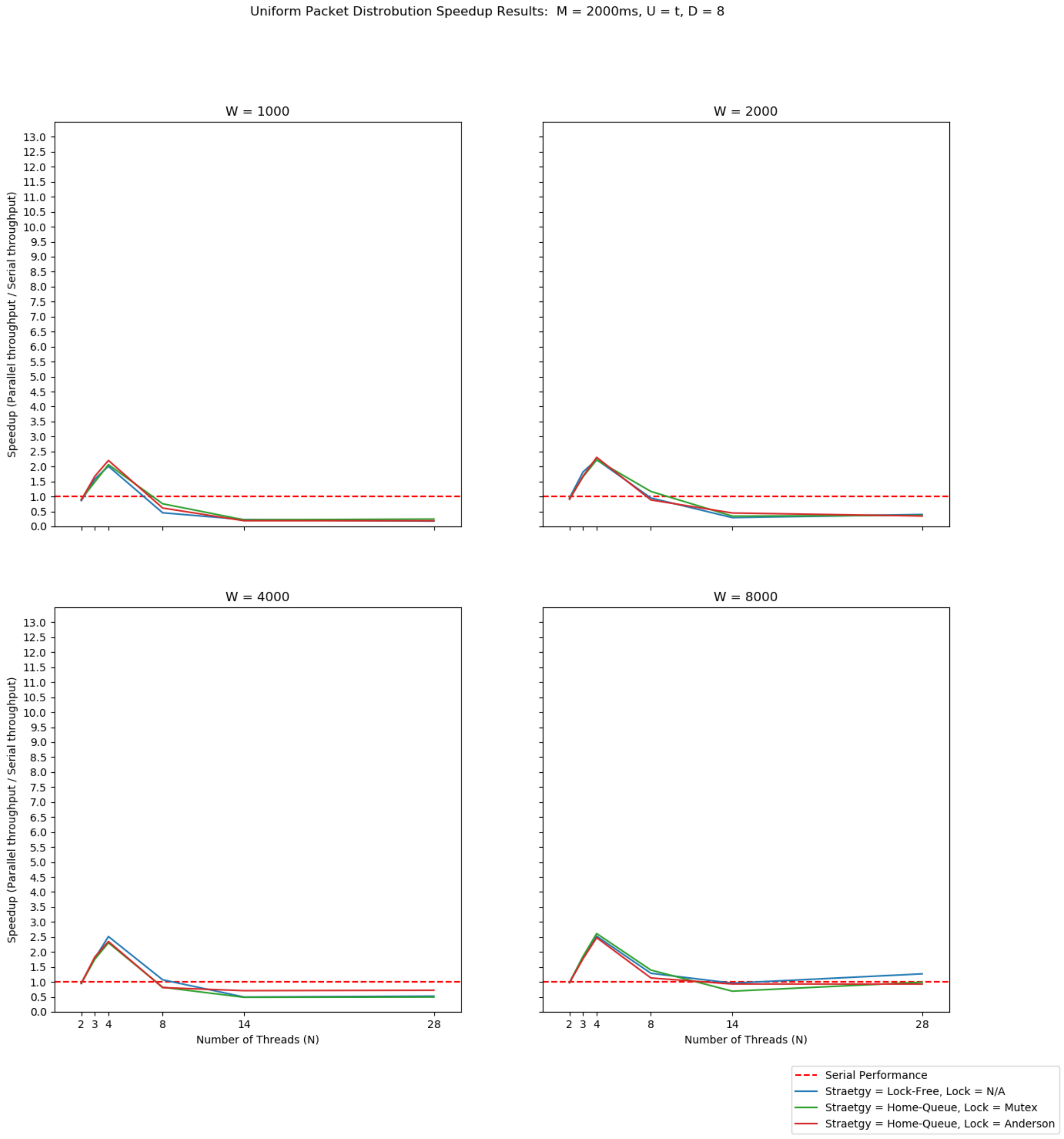
I took this data and plotted it using the script analyze.py. These graphs are stored in the folder $HW3b/Docs/graphs$. I will display the graphs pertinent to my analysis below. If the graph is labeled, each point is labeled accordingly: $< var >, < best >, < best\_l >$, where var indicates the variance of observed speedups across different load balancing strategies and lock types for set values of W and n − for Awesome tests, this value will be the variance between comparable Mutex and Anderson lock tests; *best* denotes what load balancing strategy exhibited the best speedup; *best* can be either $l, p,$or a, indicating that the best strategy was either LockFree, a comparable strategy using a Mutex lock type, and a comparable strategy using a Anderson lock type; and *best_l* can be either p or a, indicating whether, among comparable HomeQueue and Awesome tests, the Mutex or Anderson lock test performed best.

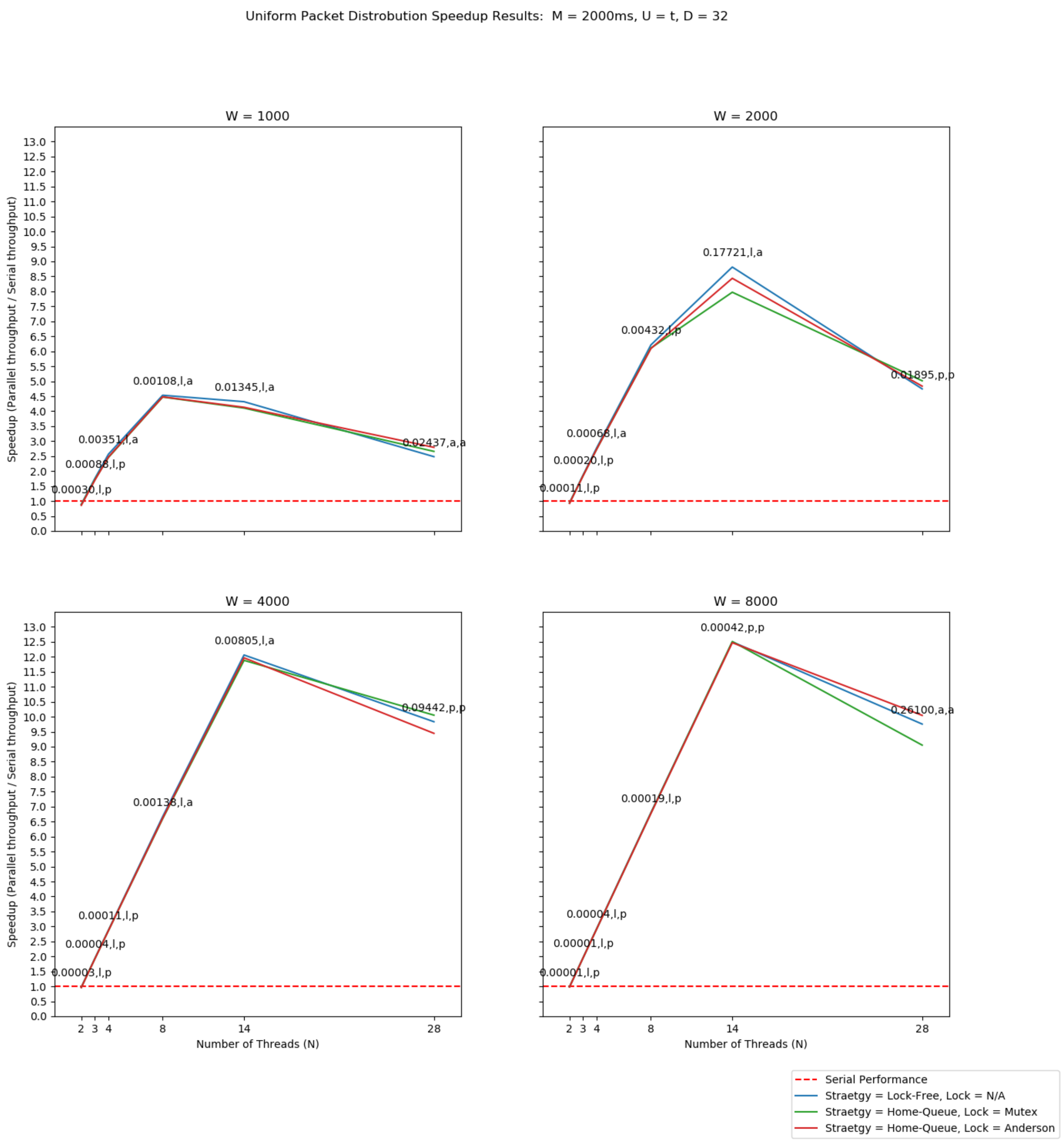
Experiment 1: Idle Lock Overhead



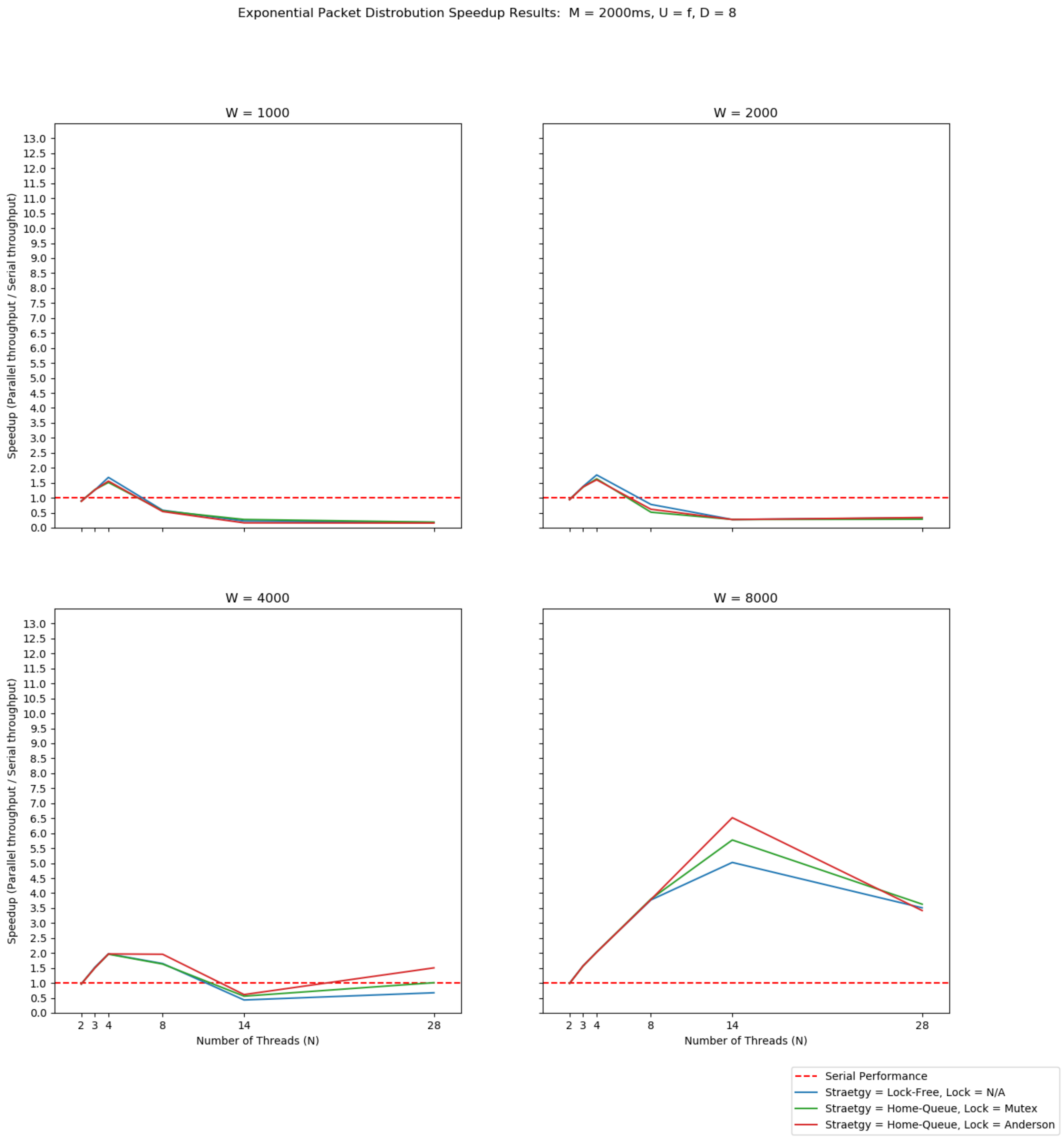

Idle Lock Overhead: M = 2000ms, N = 1, U = t, D = 32
Mutex
Anderson
Speedup (Lock-Free throughput / Home-Queue throughput)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1.1
1.2
1.3
1.4
25
50
100
200
400
800
W

Experiment 2: Speedup with Uniform Load

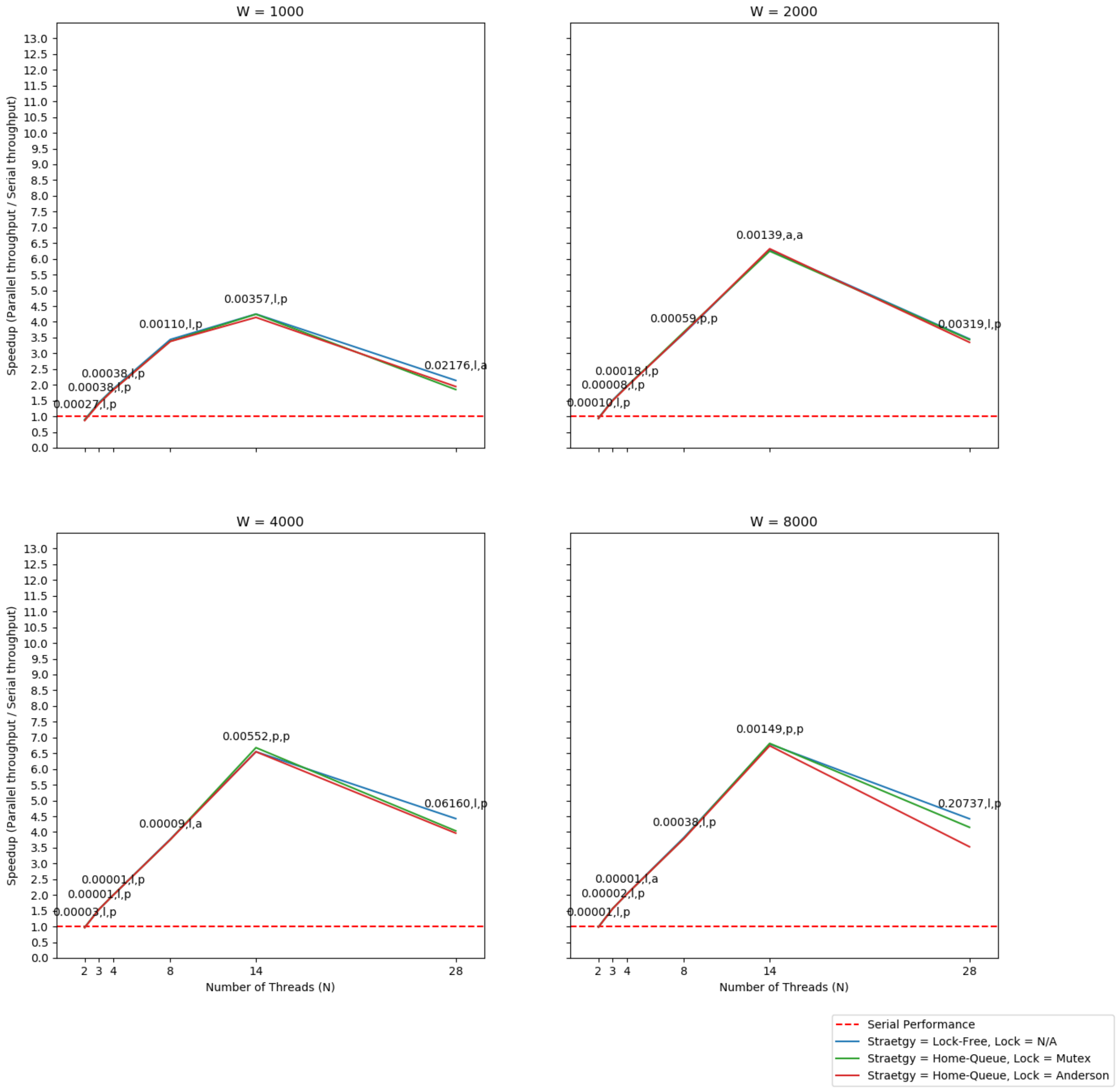



Uniform Packet Distrobution Speedup Results: M = 2000ms, U = t, D = 32
W = 1000
0.00030,l,p
0.00088,l,p
0.00351,l,a
0.00108,l,a
0.01345,l,a
0.02437,a,a
W = 2000
0.00011,l,p
0.00020,l,p
0.00068,l,a
0.00432,l,p
0.17721,l,a
0.01895,p,p
W = 4000
0.00003,l,p
0.00004,l,p
0.00011,l,p
0.00138,l,a
0.00805,l,a
0.09442,p,p
W = 8000
0.00001,l,p
0.00001,l,p
0.00004,l,p
0.00019,l,p
0.00042,p,p
0.26100,a,a
Speedup (Parallel throughput / Serial throughput)
Number of Threads (N)
Serial Performance
Straetgy = Lock-Free, Lock = N/A
Straetgy = Home-Queue, Lock = Mutex
Straetgy = Home-Queue, Lock = Anderson

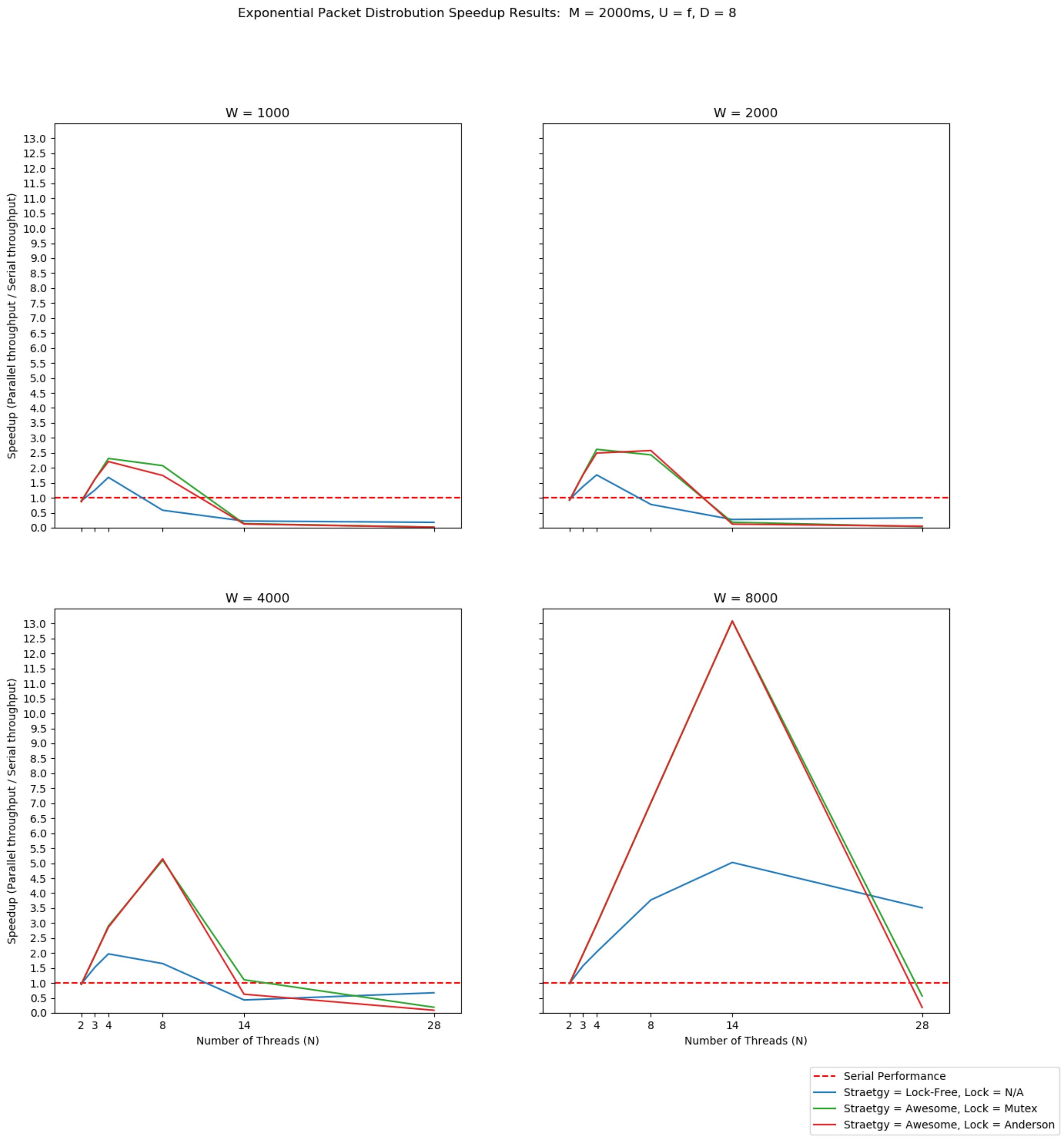
Experiment 3: Speedup with Exponential Load

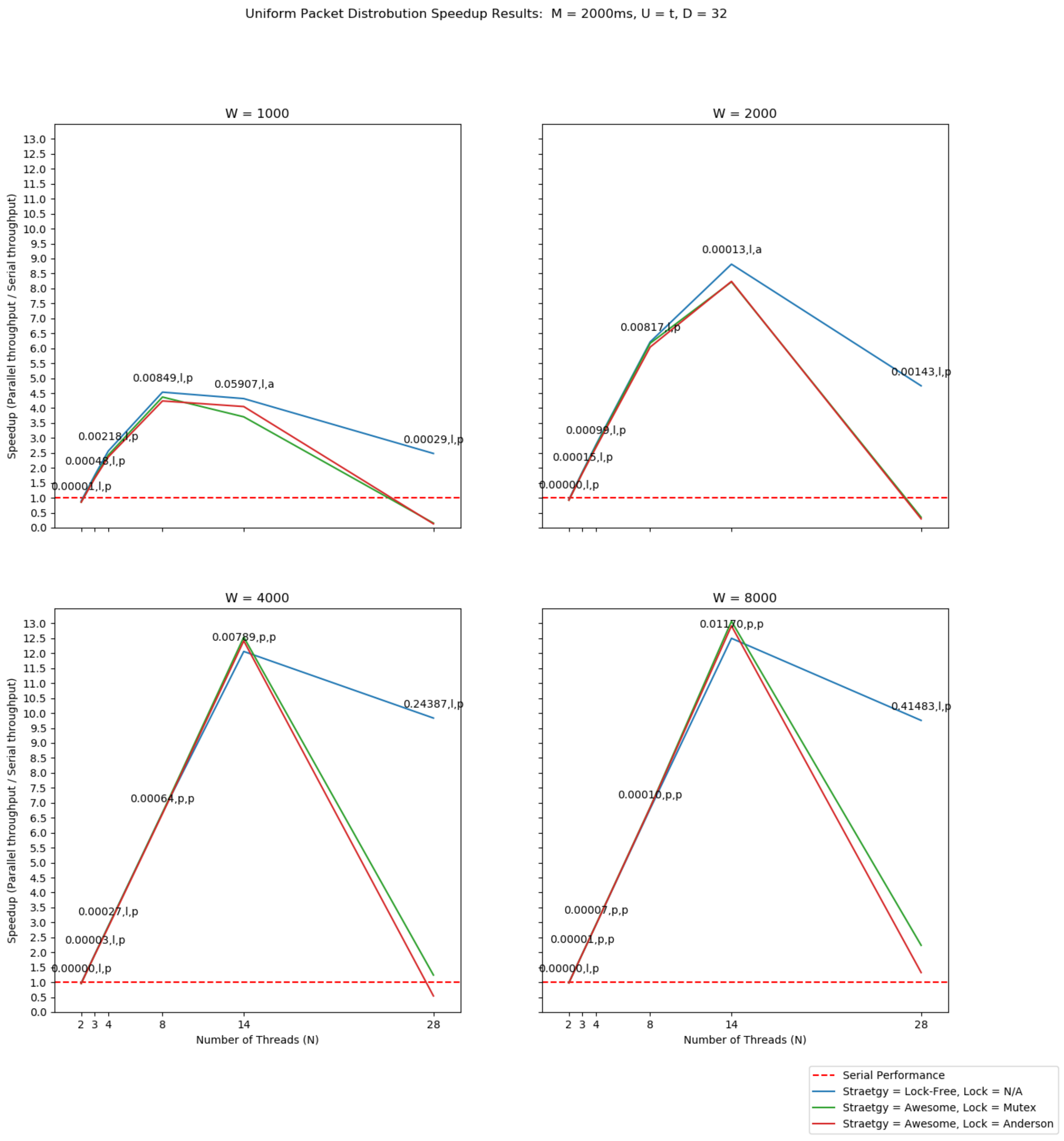


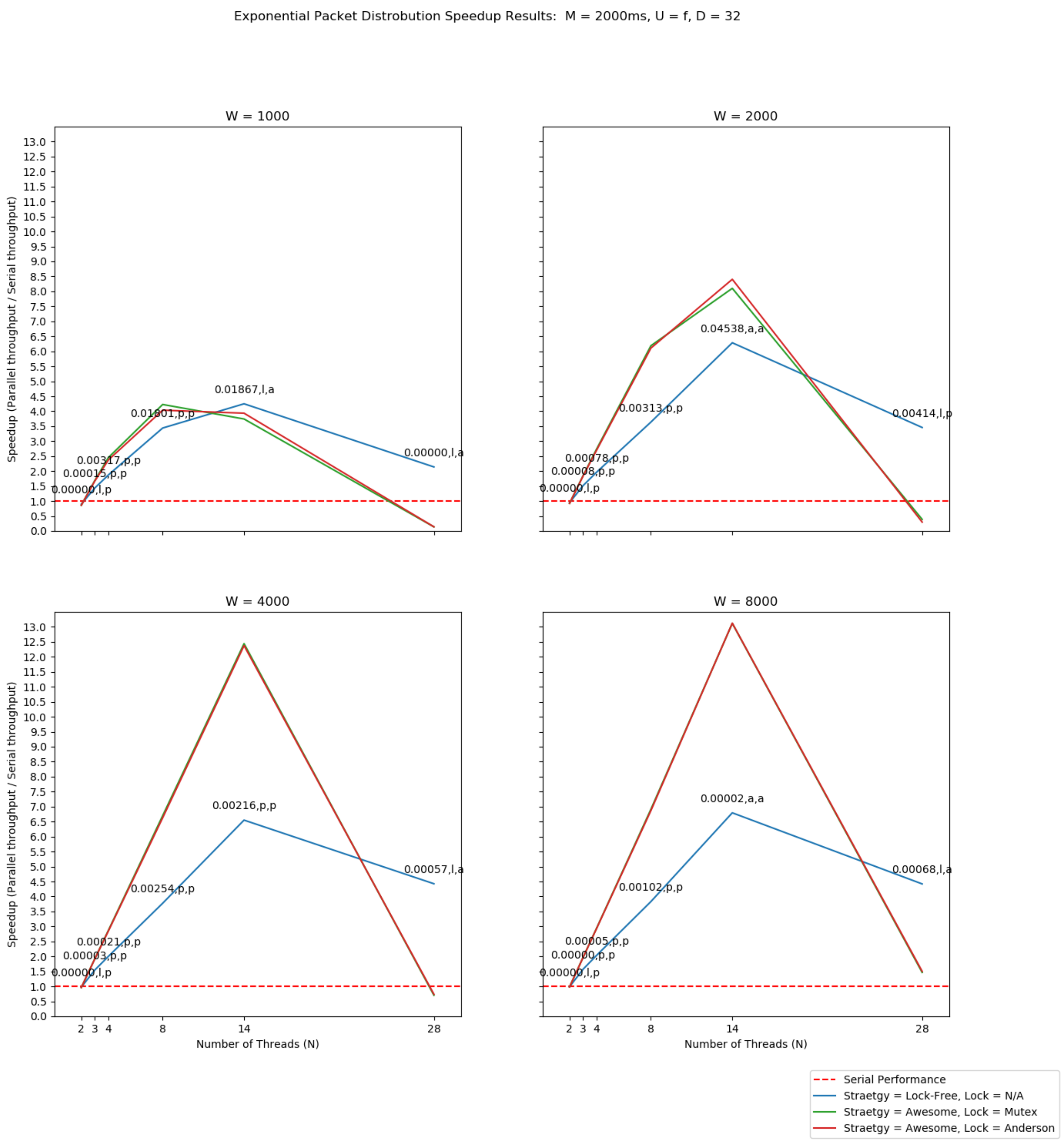
Exponential Packet Distrobution Speedup Results: M = 2000ms, U = f, D = 32



Experiment 4: Speedup with Exponential Load (Awesome Test)




Uniform Packet Distrobution Speedup Results: M = 2000ms, U = t, D = 32
W = 1000
W = 2000
W = 4000
W = 8000
Speedup (Parallel throughput / Serial throughput)
Number of Threads (N)
0.00001,l,p
0.00048,l,p
0.00218,l,p
0.00849,l,p
0.05907,l,a
0.00029,l,p
0.00000,l,p
0.00015,l,p
0.00099,l,p
0.00817,l,p
0.00013,l,a
0.00143,l,p
0.00000,l,p
0.00003,l,p
0.00027,l,p
0.00064,p,p
0.00789,p,p
0.24387,l,p
0.00000,l,p
0.00001,p,p
0.00007,p,p
0.00010,p,p
0.01170,p,p
0.41483,l,p
Serial Performance
Straetgy = Lock-Free, Lock = N/A
Straetgy = Awesome, Lock = Mutex
Straetgy = Awesome, Lock = Anderson



Analysis

Idle Lock Overhead:

Hypothesis: The purpose of this test is to measure the overhead of using locks in the HomeQueue strategy in comparison to the LockFree strategy. Between these two strategies, I expected the HomeQueue strategy to entail greater overhead in its use of a lock around the worker thread's invocation of *dequeue*(). This hypothesis followed from observations in HW3a – in counter tests, for $n = 1$, both the Mutex lock and Anderson's lock displayed 10x slowdown over ideal, LockFree performance. Therefore, I expected to see similar, observable, levels of overhead in these tests. At the very least, I ruled out the possibility of any observed speedup of HomeQueue over LockFree.

For larger values of W, I expected the relative overhead of HomeQueue over LockFree to decrease. This follows from our observations in HW2 – larger values of W entailed longer packet processing times, which gave more opportunities for parallelism and speedup. In this case, larger values of W will otherwise outweigh the increased overhead entailed in acquiring locks, reducing the relative overhead of HomeQueue over LockFree. Additionally, I expected worker rates to decrease for larger values of W. This should further effect the influence of overhead in our use of locks – that worker rates go down would also imply that worker threads dequeue less often and therefore call *lock*() and *unlock*() less frequently.

As to the relative performance of these tests across the use of our different locks: both the Mutex and our Anderson locks showed similar performance for use with a single acquiring thread – as demonstrated in HW3a, the Anderson Lock entailed a 10.33x slowdown when used with a counter, while the Mutex lock performed similarly with a 10.60x slowdown. Therefore I expected similar Idle Lock Overhead from our uses of these locks in this experiment.

Result analysis and discussion: As the data for $D = 8$ demonstrates, these expectations were not consistently accurate. For one, it is not entirely clear from the data that the use of locks in the HomeQueue strategy entailed any non-trivial amount of overhead – in some cases the data suggests that the HomeQueue strategy allowed for small levels of speedup. Notably, where I predicted to observe the most relative slowdown of HomeQueue over LockFree, namely $D = 8, W = 25$, both HomeQueue tests actually performed better relative to their corresponding LockFree test than any other data point. Both tests actually experienced speedup over LockFree, with the Mutex lock test processing 3.5% and the Anderson lcok test processing 7.9% more packets than their corresponding LockFree test. Overall, out of 12 total data points, 7 demonstrated relative speedup of our HomeQueue implementation over our LockFree implementation.

I thought that this result may be explained or otherwise better understood through a comparison of the contexts in which these locks were tested for overhead between HW3a and HW3b. In 3a, these locks were tested by setting them around a single volatile counter increment (with accompanying bounds check). However, in my HomeQueue implementation, they are set around a while loop based on my dequeue() method, which includes a two volatile reads, a modulo operation, as well as a volatile increment. Additionally, both my LockFree and HomeQueue implementations must perform the necessary packet processing.

This discrepancy may go far to explain we don't see something close to 10x slowdown of HomeQueue over LockFree. As the critical section around which the locks are placed is significantly larger here than in our counter tests, it makes sense that the effect of overhead in their use should be less significant or otherwise observable. Moreover, as threads under either load balancing strategy also have to process the packets they dequeue, the time spent locking and unlocking while using the HomeQueue strategy is further overshadowed by thread operations outside the critical section.

As such, we can understand our results from $D = 8$ trials as not demonstrating inherent speedup in our use of locks, but rather a principle that was somewhat gleaned in my results from HW3a – critical section size can otherwise mitigate the overhead experienced in using locks.

This deduction is further corroborated by experimental data from $D = 32$ trials. Notably in these trials, we don't see any speedup of HomeQueue over LockFree. This confirm my expectation that the use of the locks should add *overhead* not *speedup*. That we see this result is most likely due to the fact that queue size is no longer a bottleneck to implementation performance. With a queue depth of 8, the dispatcher and worker threads probably encounter more situations in which they yeild the processor – namely when the worker thread attempts to dequeue the empty queue or the dispatcher tries to enqueue into a full queue – increasing critical section time. However, when using a queue depth of 32, this effect of yielding the processor is less apparent as there are less chances that a worker thread or dispatcher encounter an entirely empty or full queue. As such the data we get from these trials give us a more reliable picture of lock overhead.

The data from trials using $D = 32$ more readily confirm some of my expectations about the effects of using locks and the value of W. As I mentioned above, the lack of HomeQueue speedup relative to LockFree supports my original contention that the use of locks incurs observable overhead – no trial for $D = 32$ had HomeQueue outperforming comparable LockFree tests. Therefore we can say the use of locks in our HomeQueue strategy does impart unavoidable overhead – however whether or not that overhead is observable is dependent on confounding performance impacts or bottlenecks. In this case queue depth was the concerned bottleneck.

It was wrong however to think that the observed the overhead of using a single uncontested lock would be of a similar level to the overhead in comparable counter tests in assignment 3a. As discussed above, the complexity of the application is deployed in will reduce the observable overhead of locks that are used in its implementation. Therefore, while the use of a single uncontested Mutex or Anderson lock around a counter caused 10x slowdown, I should not be surprised that the largest observed slowdown of any HomeQueue test over its comparable LockFree test is 10.2%, $L = p, W = 25$.

My contention that for larger values of W, the relative overhead of HomeQueue over LockFree should decrease is somewhat supported by data from trials where $D = 32$. Quickly I will mention that, the fact that this contention strictly isn't supported from data from $D = 8$ trials – for $D = 8$ tests, we see relative speedup of HomeQueue (using either lock) over LockFree for some smaller values of W, such as $W = 25$, while we see slowdown for larger values, such as $W = 100$ – is perhaps explained by reasoning similar to above; the bottleneck posed by queue depth complicates reasoning about lock performance. But with $D = 32$ trials, we see the effect somewhat. Notably for $D = 32, W = 25$, both HomeQueue tests observed greater slowdown than their comparable tests for $D = 32, W = 800$; The Mutex lock tests experienced a 8.55% drop in slowdown from $D = 32, W = 25$ to $D = 32, W = 800$; the Anderson lock tests experienced a 2.26% drop. Moreover the first four of the Mutex lock tests ($W \in 25, 50, 100, 200$) and the first three of the Anderson lock tests($W \in 25, 50, 100$) observed sequential drops in overhead from one test to the next. However, this sequential decrease in overhead from test to test is not observed in the whole data set; notably Mutex lock tests observed a 1.88% increase in overhead from tests $D = 32, W = 200$ to $D = 32, W = 400$. The Anderson lock tests experienced 2.06% increase from $D = 32, W = 100$ to $D = 32, W = 200$ and a 1.35% increase from $D = 32, W = if(l == best)200$ to $D = 32, W = 400$.

Overall, that a majority of HomeQueue tests (using either lock) experienced less overhead as the value of W increased generally supports my expectation that lock overhead should become less noticeable as the value of W increases.

Furthermore, as expected, for $D = 32$ tests, LockFree and HomeQueue worker rates generally decreased as the value of W increased (see csv files for corroboration). Only one tests experienced an increase in worker rate from as the value of W increased; from $D = 32, W = 25$ to $D = 32, W = 50$, HomeQueue tests using the Anderson lock were able to process 23.54% more packets / ms. All other tests exhibited sequential drops in worker rates.

As to the relative performance of the Mutex and Anderson locks: as expected both locks displayed similar performance.The difference in either locks' cumulative throughput is also negligible; $D = 8$ tests using the Anderson lock only processed 3.4% more packets than Mutex tests. In $D = 32$ tests the Anderson lock was within similar negligible bounds, only processing 4.92% more packets.

Speedup with Uniform Load:

Hypothesis: The purpose of this test is to asses the performance of our LockFree and HomeQueue implementations in relation to our serial implementation under a uniform packet distribution. As larger values of W imply more opportunity to parallelize the computation of checksums, I expected both parallel implementations to perform better as W increased. Therefore, while using both our LockFree and HomeQueue strategies, I expected increased speedup of parallel.c over serial.c for greater values of W.

The same might be said for larger values of n given a fixed value for W – for one, larger n implies that implementations use more sources and have to process more packets, leaving room for parallelization. However, larger values of n also imply larger amounts of overhead in managing threads and queues. There is also the concern that once $n >$ the number of cores in the system (14) should entail loss in performance – as observed in HW2.

Therefore, in comparing the observed speedup for different values of W, among all load balancing strategies, I expected to see increased speedup over our serial implementation for larger values of W. For a given value of W, I expected to see greater speedup for greater values of n – as observed in the results for HW2, I expected speedup to appear linear for large enough values of W and $n \leq 14$. However, I expected this to be accompanied by diminishing returns on speedup for larger values of n, especially $n > 13$.

Between load balancing strategies, I expected to see consistently faster performance from LockFree, as opposed to HomeQueue, due to the overhead entailed in using locks. Between the use of Mutex and Anderson's locks, I expected to see similar levels of performance – in line with our hypothesis from Experiment 1. Moreover, I did not expect to see discrepancy in lock overhead for varying values of n, as all locks used in my Home Queue implementation are uncontested.

Result analysis and discussion: I include graphs for $D = 8$ tests for reference – the speedup results for these tests were certainly limited by the queue depth; no test was able to top a speedup level of 2.51x; that performance suffered as n increased past 8 is also explained by queue depth, as delays in the dispatcher due to insufficient queue space would mean delays in greater numbers of queues.

As to my results for $D = 32$ tests, LockFree performance was in line with my expectations, as well as the speedup I observed in assignment 2. For one, my results confirm that observed speedup should increase for larger values of W and larger values of n, so long as $n \leq 14$. $D = 32$ tests exhibited the following local maximums in their speedup curves:

- $W = 1000$: 4.5327x speedup at $n = 8$
- $W = 2000$: 8.81198x speedup at $n = 14$
- $W = 4000$: 12.06142x speedup at $n = 14$
- $W = 8000$: 12.50104x speedup at $n = 14$

As expected, these maxima confirm that potential speedup should increase with the value of W and n, up to $n = 14$. The $W = 1000$ test stands out here in that it's local maximum occurs at $n = 8$, though at $n = 14$ its speedup only falls by 4.85%, which is negligible. Moreover, as expected, performance did in fact fall as the number of threads exceeded the number of cores; $D = 32$ tests exhibited the following drops in performance as from $n = 14$ to $n = 28$, expressed in a percentage:

- $W = 1000$: 42.51357% drop in performance
- $W = 2000$: 46.15841% drop in performance
- $W = 4000$: 18.47162% drop in performance
- $W = 8000$: 21.96281% drop in performance

Therefore, all LockFree tests suffered a significant drop in performance as the number of threads exceeded the number of cores. However, up to that point, tests with sufficient load approached linear speedup that almost scaled with the number of worker threads; notably, for $W = 8000$, our LockFree speedup results were shy of ideal speedup by the following percentages:

- $n = 2$, 1 worker threads: 0.98601x speedup 1.39900% shy of ideal scaling
- $n = 3$, 2 worker threads: 1.95762x speedup 2.11900% shy of ideal scaling
- $n = 4$, 3 worker threads: 2.92583x speedup 2.47233% shy of ideal scaling
- $n = 8$, 7 worker threads: 6.79893x speedup 2.87243% shy of ideal scaling

- $n = 14$, 13 worker threads: 12.50104x speedup 3.83815% shy of ideal scaling
- $n = 28$, 27 worker threads: 9.75546x speedup 63.86867% shy of ideal scaling

Additionally, performing a linear regression on the first five data points reveals that the $W = 8000$ test experienced a performance to worker thread ratio of $0.9686 : 1$, with an R^2 values of 0.9999, further supporting that LockFree tests experienced linear speedup that scaled with the number of worker threads.

As to how the HomeQueue strategy compares in performance to LockFree, there is not much else to say. As demonstrated in the Idle Lock Overhead tests, the overhead entailed in using either the Anderson or Mutex locks with the HomeQueue strategy grows more negligible as the value of W increases. In those tests, for $W = 800$, neither lock type experienced more than a 5%. As such the effect of using locks was otherwise mitigated by the use of sufficiently large values of W, namely $W \in \{1000, 2000, 4000, 8000\}$.

Furthermore, This is also represented by the fact that all comparable tests across LockFree and both HomeQueue implementations exhibited variance less than 1, indicating consistently similar performance among LockFree and HomeQueue tests. That being said, LockFree did seem to outperform all other implementations in a majority ($N = 18$) of tests. This perhaps suggests that LockFree performance is something of a bound on HomeQueue performance, as suggested in the results of the Idle Lock Overhead test. In that case, perhaps instances of HomeQueue tests outperforming LockFree tests is a reflection of system runtime aberrations – this, again, is supported by the small levels of variance seen across comparable tests.

Between the two locks utilized in the HomeQueue tests, differences in performance were negligible, or otherwise insignificant. However tests utilizing Mutex locks did outperform comparable tests with Anderson locks in a majority ($N = 15$) of tests. This perhaps suggests that the Mutex lock is superior in this application but, again, any advantage is negligible. Finally we should say then is that these results support the contention that locks, at least used in the way they are in HomeQueue, do not effect the scalability of ParllelPacket

Speedup with Exponential Load:

Hypothesis: I adopt the same reasoning from the prior experiment. However, because this test employs a exponential packet generator, the amount of work each thread must perform is constantly increasing at an exponential rate. This may make it seem like we should experience even greater opportunities for speedup even as we continue to process packets. However, as different sources generate packets with different curves that describe the work needed to generate a checksum for the i-th packet, all of which merely share an average amount of work, this leaves the possibility of some queues becoming overburdened with large packets as the test goes on, creating performance bottlenecks. Therefore, while greater values of W should entail greater opportunities for speedup, without load balancing, this effect will not be as noticeable as when using an Uniform packet generator.

Like the previous experiment, in comparing the observed speedup of our parallel implementations over our serial implementation for different values of W, I expected increased speedup for larger values of W. For a given value of W, I expected to see greater speedup for greater values of n – as observed in the results for HW2, we can expect speedup to appear linear for large enough values of W and n (the number of worker threads) ≤ 13. However, this is accompanied by diminishing returns on speedup for larger values of n, especially $n > 13$.

Between load balancing strategies, I expected to see consistently faster performance from LockFree, as opposed to HomeQueue, due to the overhead entailed in using locks. Between the use of Mutex and Anderson's locks, I expected to see similar levels of performance – in line with our hypothesis from Experiment 1. Moreover, I did not expect to see discrepancy in lock overhead for varying values of n, as all locks used in my Home Queue implementation are uncontested.

Result analysis and discussion: As to my results for $D = 32$ tests, LockFree performance was in line with my expectations, as well as the speedup I observed in assignment 2. For one, my results confirm that observed speedup should increase for larger values of W and larger values of n, so long as $n \leq 14$. $D = 32$ tests exhibited the following local maximums in their speedup curves:

- $W = 1000$: 4.24745x speedup at $n = 14$
- $W = 2000$: 6.28448x speedup at $n = 14$
- $W = 4000$: 6.54938x speedup at $n = 14$
- $W = 8000$: 6.79314x speedup at $n = 14$

As expected, these maxima confirm that potential speedup should increase with the value of W and n, up to $n = 14$. Moreover, as expected, performance did in fact fall as the number of threads exceeded the number of cores; $D = 32$ tests exhibited the following drops in performance as from $n = 14$ to $n = 28$, expressed in a percentage:

- $W = 1000$: 49.64061% drop in performance
- $W = 2000$: 45.00404% drop in performance
- $W = 4000$: 32.43803% drop in performance
- $W = 8000$: 34.95114% drop in performance

Therefore, all LockFree tests suffered a significant drop in performance as the number of threads exceeded the number of cores. However, up to that point, tests with sufficient load approached linear speedup. That such speedup was linear is appreciable from the graph of $W = 8000$ tests; the first five LockFree tests in that group ($W = 8000, n \in \{2, 3, 4, 8, 14\}$) show increases in performance that scale linearly with the number of worker threads. Performing a linear regression on this set of points results in line with a slope of 0.4659 and an R^2 value of 0.9977. As such, the tests did not exhibit performance that perfectly scaled with the number of worker threads, and as such did not exhibit ideal speedup. For $W = 8000$, our LockFree speedup results were shy of ideal speedup by the following percentages:

- $n = 2$, 1 worker threads: 0.98601x speedup 1.39900% shy of ideal scaling
- $n = 3$, 2 worker threads: 1.95762x speedup 2.11900% shy of ideal scaling
- $n = 4$, 3 worker threads: 2.92583x speedup 2.47233% shy of ideal scaling
- $n = 8$, 7 worker threads: 6.79893x speedup 2.87243% shy of ideal scaling
- $n = 14$, 13 worker threads: 12.50104x speedup 3.83815% shy of ideal scaling
- $n = 28$, 27 worker threads: 9.75546x speedup 63.86867% shy of ideal scaling

Overall then, comparing LockFree performance under Uniform packet generation to that under Exponential Packet generation, I was right to contend that LockFree tests under Exponential load would be able to exhibit linear speedup, scaling with the number of worker threads, for large enough values of W. Furthermore, I was also correct to judge that this speedup would pale in comparison to that observed from Uniform load test. Unlike tests under Uniform loads, Exponential tests were unable to demonstrate ideal speedup; nor did any Exponential test exhibit greater maximum speedup than Uniform tests for comparable values of W.

As to how the HomeQueue strategy compares in performance to LockFree, there is not much else to say. As demonstrated in the Idle Lock Overhead tests, the overhead entailed in using either the Anderson or Mutex locks with the HomeQueue strategy grows more negligible as the value of W increases. In those tests, for $W = 800$, neither lock type experienced more than a 5%. As such the effect of using locks was otherwise mitigated by the use of sufficiently large values of W, namely $W \in \{1000, 2000, 4000, 8000\}$.

Furthermore, This is also represented by the fact that all comparable Exponential load tests across LockFree and both HomeQueue implementations exhibited variance less than 1, indicating consistently similar performance among LockFree and HomeQueue tests. That being said, LockFree did seem to outperform all other implementations in a majority ($N = 20$) of tests. This perhaps suggests that LockFree performance is something of a bound on HomeQueue performance, as suggested in the results of the Idle Lock Overhead test. In that case, perhaps instances of HomeQueue tests outperforming LockFree tests is a reflection of system runtime aberrations – this, again, is supported by the small levels of variance seen across comparable tests.

Between the two locks utilized in the HomeQueue tests, differences in performance were negligible, or otherwise insignificant. However tests utilizing Mutex locks did outperform comparable tests with Anderson locks in a majority ($N = 20$) of tests. This perhaps suggests that the Mutex lock is superior in this application but, again, any advantage is negligible. Finally we should say then is that these results support the contention that locks, at least used in the way they are in HomeQueue, do not effect the scalability of ParllelPacket

Speedup with Exponential Load (Awesome Test)

For my final experiment, I proposed to redo the prior experiment but using $S \in \{LockFree, Awesome\}$.

Hypothesis: Like the previous experiment, in comparing the observed speedup of our parallel implementations for different values of W, I expected increased speedup for larger values of W. For a given value of W, I expected to see greater speedup for greater values of n – as observed in the results for HW2, we can expect speedup to appear linear for large enough values of W and $n \leq 13$. However, this is accompanied by diminishing returns on speedup for larger values of n, especially $n > 13$. I expected this estimation to be true for both our LockFree and Awesome load balancing strategies.

Unlike the previous experiment, while the speedup of our LockFree implementation will suffer due to the use of an exponential packet generator, I expected that the speedup of our Awesome implementation would not. Rather, because worker threads are able to search for work to do, I expected that they would alleviate uneven burdens put on the implementation's various queues due to the exponential distribution. This is the situation under which the Awesome strategy should succeed; otherwise, if there was an even distribution of work among queues, there would be no performance to be gained from making threads search for work. In this test then, I expected that the Awesome strategy would outperform the LockFree strategy for all values of n and W.

However, I also expected that this increase in performance from load balancing would somewhat be mitigated by greater values of n – under Awesome strategy, the queues' locks can be in contention among different worker threads, making them a greater possible source of slowdown than they were when used in the HomeQueue strategy, which only utilized uncontested locks. Performance between the two locks, however, should be similar, if not with the Mutex lock performing a bit better – this is confirmed by our curves from HW3a, Experiment 2. This experiment demonstrated that the Mutex lock consistently outperforms our Anderson lock in the presence of contention – which is bound to occur under the Awesome strategy.

Result analysis and discussion: Firstly I should mention that, between the two locks used in the Awesome load balancing strategy, the Mutex lock exhibited better performance. Tests utilizing Mutex locks outperformed comparable tests with Anderson locks in a majority ($N = 18$) of tests. That being said, as the variance measurements for these tests demonstrate, differences in performance between Awesome tests that used Mutex locks and those that used Anderson locks were negligible. No comparison revealed a variance greater than 0.05, indicating coherent and comparable results. Overall then, we can say that the data supports the idea that the Mutex lock generally outperformed the Anderson lock in Awesome tests, as expected from our results from HW3a. However, the difference in performance is sufficiently negligible. For this reason, in order to simplify analysis, discussion of Awesome test results will average comparable performance data points across both lock types – data concerning Awesome performance for given values of W and n will be the average Awesome performance across $L = p$ and $L = a$.

Briefly I will mention that LockFree data shown in graphs for this experiment replicate those shown in the previous experiment; as such no additional space will be given to their discsussion here.

Rather, moving on to a discussion on how the Awesome strategy performed under Exponential load: In $D = 32$ tests, Awesome performance results certainly seem to be in line with my expectations. For one, my results confirm that observed speedup should increase for larger values of W and larger values of n, so long as $n \leq 14$. Awesome tests exhibited the following (averaged) local maximums in their speedup curves:

- $W = 1000$: 4.1294x speedup at $n = 8$

- $W = 2000$: 8.25264x speedup at $n = 14$
- $W = 4000$: 12.40724x speedup at $n = 14$
- $W = 8000$: 13.121275x speedup at $n = 14$

As expected, these maxima confirm that potential speedup should increase with the value of W and n, up to $n = 14$. The $W = 1000$ test stands out here in that it's local maximum occurs at $n = 8$, and that at $n = 14$ its speedup only by 7.09%, which is not negligible. However, as expected, performance did in fact fall significantly once the number of threads exceeded the number of cores; $D = 32$ tests exhibited the following drops in performance as from $n = 14$ to $n = 28$, expressed in a percentage:

- $W = 1000$: 96.36501% drop in performance
- $W = 2000$: 95.84060% drop in performance
- $W = 4000$: 94.20250% drop in performance
- $W = 8000$: 88.70518% drop in performance

Therefore, all Awesome tests suffered a significant drop in performance as the number of threads exceeded the number of cores. However, up to that point, tests with sufficient load approached linear speedup. That such speedup was linear is appreciable from the graph of $W = 8000$ tests; the first five Awesome tests in that group ($W = 8000, n \in \{2, 3, 4, 8, 14\}$) show increases in performance that scale linearly with the number of worker threads. Performing a linear regression on this set of points results in line with a slope of 0.9850 and an R^2 value of 0.9999. As such, the Awesome tests exhibited performance that perfectly scaled with the number of worker threads – ideal speedup. For $W = 8000$, our Awesome speedup results were shy of ideal speedup by the following percentages:

- $n = 2$, 1 worker threads: 0.98039x speedup 1.96100% shy of ideal scaling
- $n = 3$, 2 worker threads: 1.96729x speedup 1.63550% shy of ideal scaling
- $n = 4$, 3 worker threads: 2.94234x speedup 1.92183% shy of ideal scaling
- $n = 8$, 7 worker threads: 6.89050x speedup 1.56429% shy of ideal scaling
- $n = 14$, 13 worker threads: 13.12128x speedup -0.93288% shy of ideal scaling
- $n = 28$, 27 worker threads: 1.48202x speedup 94.51102% shy of ideal scaling

Overall then, considering Awesome performance, I was right to contend that Awesome tests under Exponential load, like LockFree tests, would be able to exhibit linear speedup, scaling with the number of worker threads, for large enough values of W.

Comparing Awesome performance results to LockFree results, I can say that I was mostly correct in my contention that the Awesome load balancing strategy would outperform the LockFree strategy. albeit not got all combinations of W and n. Rather, LockFree tests outperformed Awesome tests in a not-insignificant number of combinations of W and n ($N = 9$). However, of those, such differences in performance are insignificant or otherwise within the margin of error for 5 of those tests. Notably, LockFree tests did significantly outperform comparable Awesome tests for tests where $n = 28$. Otherwise, Awesome tests outperformed comparable LockFree tests by margins that grew as W and n increased (barring $n = 28$). Furthermore, while LockFree tests were not able to demonstrate ideal speedup, even with sufficiently large values of W, Awesome tests were in able to put up such ideal speedup – no Awesome test with $W = 8000$, $n \in \{2, 3, 4, 8, 14\}$ was shy of ideal performance by more than 2%, and one test even exceeded the ideal.

Generally then, the Awesome strategy overcame load balancing limitations imposed on LockFree tests under Exponential load. That while doing so the Awesome strategy exhibited ideal speedup further supports the robustness of its design. Briefly, I will also mention that this improvement of Awesome over LockFree did in fact occur to due the nature of the Exponential load distribution, as testing the Awesome strategy on a Uniform distribution of packets did not yield any such improvement over LockFree tests on a Uniform distribution.

Lastly, I will discuss why LockFree tests did significantly outperform comparable Awesome tests for tests where $n = 28$. For one, that either strategy experienced a significant drop in performance as the number of threads exceeded the number of cores ($n > 14$) is explained by limitations of the system. However, that Awesome tests for $n = 28$ consistently performed worse than LockFree tests for $n = 28$ is perhaps further explained by the limitations of lock scalability when using the Awesome strategy. The Awesome strategy, unlike the HomeQueue strategy, allows for the contention of locks by allowing worker threads to cycle through queues. Though this contention is mitigated through the use of $trylock()$, increasing number of threads trying to acquire the same lock at the same time would increase the amount of threads that end up having to wait on others, noticeably increasing overhead. This would explain why Awesome tests experience drops in performance from $n = 14$ to $n = 28$ on the order of 90%. In addition to suffering the impacts of using more threads than cores, these tests' performance drops are further compounded by increased levels of lock contention. This perhaps suggest that lock scalability is a problem for all Awesome tests, for any value of n, but only truly becomes observable for $n = 28$, or when the number of threads exceeds the number of system cores.