Andrew McCarthy

Parallel Programming HW2 Problems

25) L2 says if method call m precedes method call n in H, then the same is true in S. H is a history & S is a legal sequential history that is equivalent to the complet extension of H. L2 is the property that helps linearizability be compositional, so by taking away the L2 statement from the linearizability definition it takes away the compositional property. Thus, it makes the resulting property the same as sequential consistency

29) The statement is NOT equivalent. A wait free object guarantees that every call finishes in a finite number of steps. In an infinite history where each thread completes an infinite number of method calls, there is no guarantee of the previous statement. Therefore, it is not equivalent

30) The statement is equivalent. Lock-free objects guarantee that infinitely often some method call finishes in a finite number of steps. In an infinite history of x, where an infinite # of method calls are completed there is always some method call being finished. therefore since it finished then it completes in a finite # of steps and there are an infinite number of calls which have this characteristic. Thus, infinitely often some method call finishes in a finite number of steps so object x is lock free.

31) A wait-free method guarantees that every call completes in a finite number of steps. A bounded wait-free method puts a bound on the # of steps a call can take. Here, the method has a bounded wait-free property because the call returns after 2^i steps. This means that there are a finite # of steps the method completes, and thus the method is wait-free. Also, the call does not exceed 2^i steps so therefore it has a bound.

∴ It is a bounded wait-free method

32) Thread 1 enqueues x onto the queue, & Concurrently thread 2 enqueues y. Then, thread 3 dequeues
- 1 executes tail.getAndIncrement which sets i to 0 & the tail goes to one.
- 2 also executes tail.getAndIncrement which sets i to 1 & tail goes to two.
- 2 stores y at index 1 of items
- 3 dequeues & finds the item at index 0 to be null
  ↳ it goes to index 1 & finds & returns y
- 1 stores x at 0th index
- 3 dequeues & finds item at 0 to be x & returns

∴ the code returns y before x even though thread 1 executed line 15 before thread 2. So it can't be a linearization point

- line 1b example:
  - thread 1 executes tail.getAndIncrement before 2
  - 2 then executes item[i].set(y) before 1 set item[i] to x
  - 3 dequeues 0^{th} index & returns x
  - 3 dequeues 1^{st} index & returns y

∴ line 1b is not a linearization point

- this doesn't mean enqueue is not linearizable. It means sometimes there is not a fixed statement which is a linearization point.

Parallel Programming HW2 Write Up

Andrew McCarthy

Source files are located in the hw2/src directory. My code is in main.c, lamport.c/.h, and atomic.h. Instructions on how to compile and run the code is under the 'Details' section

- Overview

This program is designed to mimic the functions of a firewall in both parallel and serial execution. The parallel version enqueues packets into a queue (each source gets its own queue) while worker threads concurrently dequeue and process the packets. The serial version of the code retrieves packet from each source and processes them serially using a double for loop. The program also includes a serial-queue version which enqueues and dequeues packets before they are processed serially (this is designed to test the overhead of the queue structure). The program takes in 7 arguments when run normally. These are the number of threads (including master), the number of packets from each source, the queue depth, the mean/expected work to process each packet, the seed, the packet generation type (constant, uniform, and exponential), and the running mode (parallel, serial, and serial-queue). The program can also run a series of performance tests by making the first input parameter test and the second the seed.

- Design

The project code is in the Lamport.c/.h, atomic.h and main.c files. The lamport files are where the code for the lamport/lock-free-queues are. Atomic is where the code to make use of the atomic gcc builtins is. Main.c is the where the dispatcher and worker code is as well as all the code for managing input arguments and running the program in serial, serial-queue or parallel mode.

The lamport queue has three functions: create_queue, enqueue_queue, and dequeue_queue. The depth of the queue is set during allocation in the create_queue function. Element_t is a structure used to hold the address of enqueued elements and an update count for the slot. Queue_t contains the tail and head information, the depth information, and a list of the elements stored in it.

The enqueue function starts by taking a copy of the tail pointer and the element at said pointer. Next, the function makes sure the element's pointer is correctly pointing to the tail of the queue. Next, the function checks if the queue is full; if it is, the function returns 0. Next, the function ensures the head pointer is not lagging behind. Next, the function checks if the current slot's pointer is non-null. This ensures that the tail pointer does not lag behind. Next, the data in the slot is copied and a DWCAS operation is performed. If it is successful, the tail pointer is incremented and 1 is returned. If it fails, the operations are restarted. The dequeue function is similar to the enqueue function except it removes an element from the queue.

The atomic.h file has two functions: CAS and DWCAS (compare-and-swap and double-compare-and-swap). These functions are used by the queue to synchronize and ensure that the tail and head pointers do not lag behind.

The dispatcher for the packets works by using an array representing the total number of packets still needed for each source, and an int representing the total number of packets that need to be generated (numQueues * packetsPerSource). The dispatcher continuously loops through each queue and generates and inserts a packet if the queue is not full. If a packet is successfully inserted, the corresponding value in the packets-needed array is decremented and the total-number-of-packets int is decremented. Once the packets-needed int is 0, the loop exits.

The worker threads continuously dequeue elements from the queue and calculate the fingerprint. Once the worker has processed the designated number of packets (number of packets that each source generates) it exits.

The serial and serial-queue versions of the code work by looping through all the sources then generating and processing the necessary packets. The serial-queue version enqueues and dequeues packets before processing them.

- Divergence From Design Document

There are a two big changes I made to my design after I began coding. First, I incorrectly stated how to properly use memory fences in my design document. Once I started coding I realized my mistake and changed my design plan. Second, in my design document I stated that I would load all the packets into the queues before starting the worker threads. After reading the project description again I realized this would be impossible as the queue depth could not hold all of the packets at once. I corrected this mistake once I began coding.

- Details

To create the executables for the program run make. To remove the executable and object files run make clean. To run the designated tests, make “test” the first input parameter and a seed value the second. To run in normal mode, you must provide the following parameters preceded by their corresponding flag: -n (number of threads including master), -t (packets from each source), -d (queue depth), -w (mean work per packet), -g (packet generation mode-> must be “constant”, “uniform”, “exponential”), -m (parallel or serial execution -> must be “parallel”, “serial”, “serial-queue”). A -s (seed) parameter can also be provided but it is not required. The parameters can be given in any order. When running in normal mode, the program prints out the run time to stdout. When running in test mode, the program prints the test results to stdout. Here is an example execution: ./lamport -n 14 -t 60000 -d 32 -w 3000 -s 50 -g constant -m parallel

- Issues

One issue I ran into was the -Wstrict-aliasing warning blocking me from compiling my code. Upon looking into this warning, I learned it wasn't an issue for my use case (converting pointers to my element_t struct in lamport.c) so I disabled it using compiler flags.

- Correctness testing

My worker threads can only finished once they process as many packets as is set by the T parameter. Due to this, if any errors caused a packet to not be properly enqueued/dequeued, the program would spin indefinitely. To test this, I ran the parallel version of my code with various input parameters and made sure it finished.

- Performance Testing

To test performance difference between serial and parallel executions, I ran the tests outlined in the project description with preset values and printed the results to stdout. The results of these tests are listed below:

--Output:

speedup queue overhead with w = 25 and n = 2 is 2.154754
speedup queue overhead with w = 25 and n = 9 is 1.071721
speedup queue overhead with w = 25 and n = 14 is 1.555466
speedup queue overhead with w = 50 and n = 2 is 1.347470
speedup queue overhead with w = 50 and n = 9 is 1.344514
speedup queue overhead with w = 50 and n = 14 is 1.363018
speedup queue overhead with w = 100 and n = 2 is 1.211009
speedup queue overhead with w = 100 and n = 9 is 1.223723
speedup queue overhead with w = 100 and n = 14 is 1.227577
speedup queue overhead with w = 200 and n = 2 is 1.127900
speedup queue overhead with w = 200 and n = 9 is 1.120298
speedup queue overhead with w = 200 and n = 14 is 1.124955
speedup queue overhead with w = 400 and n = 2 is 1.077529
speedup queue overhead with w = 400 and n = 9 is 1.068528
speedup queue overhead with w = 400 and n = 14 is 1.058897
speedup queue overhead with w = 800 and n = 2 is 1.006073
speedup queue overhead with w = 800 and n = 9 is 1.032700
speedup queue overhead with w = 800 and n = 14 is 1.039192

dispatcher rate time with w = 1 and n = 2 is 4559.582031
dispatcher rate time with w = 1 and n = 3 is 5008.192871

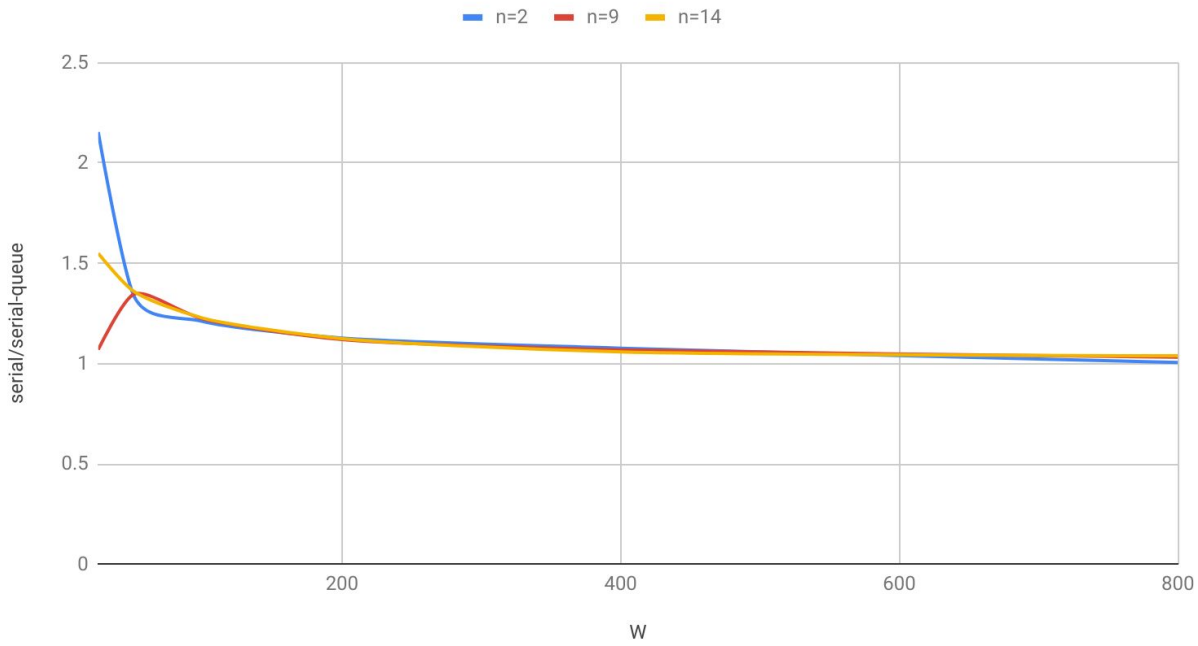
dispatcher rate time with w = 1 and n = 5 is 4600.687012
dispatcher rate time with w = 1 and n = 9 is 4278.637207
dispatcher rate time with w = 1 and n = 14 is 4129.667969
dispatcher rate time with w = 1 and n = 28 is 5822.081055
speedup constant packets with w = 1000 and n = 2 is 1.116477
speedup constant packets with w = 1000 and n = 3 is 1.153973
speedup constant packets with w = 1000 and n = 5 is 1.318144
speedup constant packets with w = 1000 and n = 9 is 1.613507
speedup constant packets with w = 1000 and n = 14 is 1.613503
speedup constant packets with w = 1000 and n = 28 is 1.123558
speedup constant packets with w = 2000 and n = 2 is 2.204270
speedup constant packets with w = 2000 and n = 3 is 2.223579
speedup constant packets with w = 2000 and n = 5 is 2.687395
speedup constant packets with w = 2000 and n = 9 is 3.012343
speedup constant packets with w = 2000 and n = 14 is 3.114397
speedup constant packets with w = 2000 and n = 28 is 2.656482
speedup constant packets with w = 4000 and n = 2 is 4.075189
speedup constant packets with w = 4000 and n = 3 is 4.466882
speedup constant packets with w = 4000 and n = 5 is 5.387151
speedup constant packets with w = 4000 and n = 9 is 5.929618
speedup constant packets with w = 4000 and n = 14 is 6.370973
speedup constant packets with w = 4000 and n = 28 is 4.855897
speedup constant packets with w = 8000 and n = 2 is 8.366705
speedup constant packets with w = 8000 and n = 3 is 9.396678
speedup constant packets with w = 8000 and n = 5 is 11.555341
speedup constant packets with w = 8000 and n = 9 is 11.704694
speedup constant packets with w = 8000 and n = 14 is 12.350133
speedup constant packets with w = 8000 and n = 28 is 7.593753

speedup uniform packets with w = 1000 and n = 2 is 3.316725
speedup uniform packets with w = 1000 and n = 3 is 4.108906
speedup uniform packets with w = 1000 and n = 5 is 4.283566
speedup uniform packets with w = 1000 and n = 9 is 5.362582
speedup uniform packets with w = 1000 and n = 14 is 5.106752
speedup uniform packets with w = 1000 and n = 28 is 3.702444
speedup uniform packets with w = 2000 and n = 2 is 6.911411
speedup uniform packets with w = 2000 and n = 3 is 8.140733
speedup uniform packets with w = 2000 and n = 5 is 9.215037
speedup uniform packets with w = 2000 and n = 9 is 9.934791

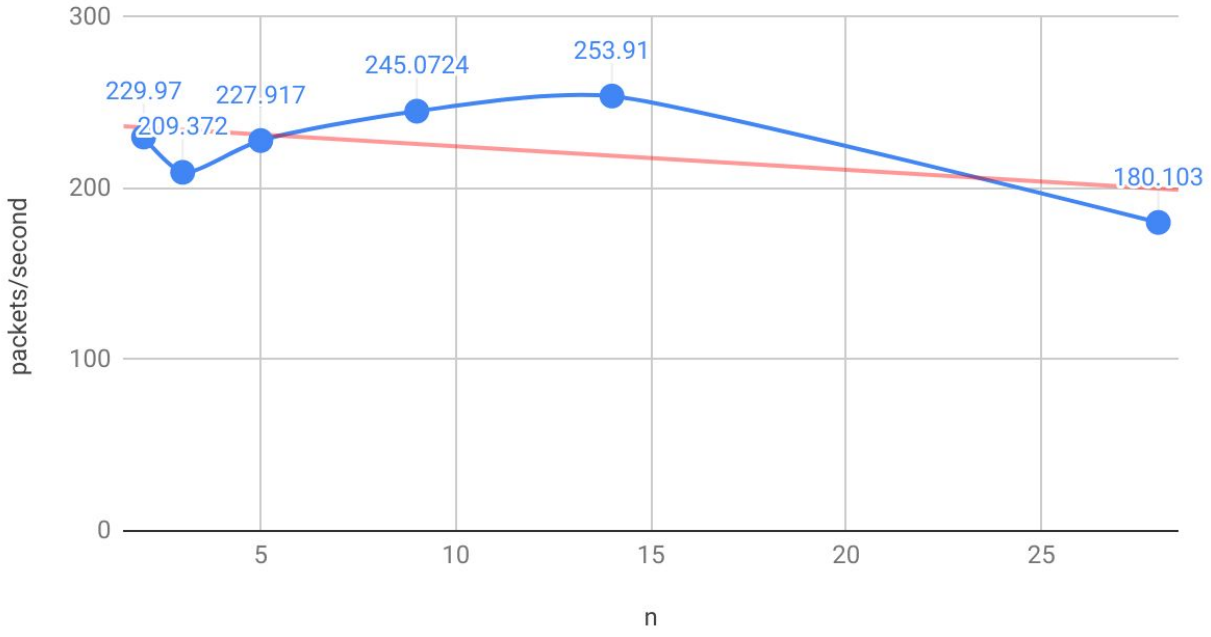
speedup uniform packets with w = 2000 and n = 14 is 10.678123
speedup uniform packets with w = 2000 and n = 28 is 7.932747
speedup uniform packets with w = 4000 and n = 2 is 13.528705
speedup uniform packets with w = 4000 and n = 3 is 15.826137
speedup uniform packets with w = 4000 and n = 5 is 18.915048
speedup uniform packets with w = 4000 and n = 9 is 20.077365
speedup uniform packets with w = 4000 and n = 14 is 20.721141
speedup uniform packets with w = 4000 and n = 28 is 13.243738
speedup uniform packets with w = 8000 and n = 2 is 30.575239
speedup uniform packets with w = 8000 and n = 3 is 32.185424
speedup uniform packets with w = 8000 and n = 5 is 39.510968
speedup uniform packets with w = 8000 and n = 9 is 40.513812
speedup uniform packets with w = 8000 and n = 14 is 41.673233
speedup uniform packets with w = 8000 and n = 28 is 32.894543

speedup exponential packets with w = 1000 and n = 2 is 3.399936
speedup exponential packets with w = 1000 and n = 3 is 3.703270
speedup exponential packets with w = 1000 and n = 5 is 4.296386
speedup exponential packets with w = 1000 and n = 9 is 4.522426
speedup exponential packets with w = 1000 and n = 14 is 4.326938
speedup exponential packets with w = 1000 and n = 28 is 2.478234
speedup exponential packets with w = 2000 and n = 2 is 6.627845
speedup exponential packets with w = 2000 and n = 3 is 6.906067
speedup exponential packets with w = 2000 and n = 5 is 8.147613
speedup exponential packets with w = 2000 and n = 9 is 8.275903
speedup exponential packets with w = 2000 and n = 14 is 8.624423
speedup exponential packets with w = 2000 and n = 28 is 6.190574
speedup exponential packets with w = 4000 and n = 2 is 13.983295
speedup exponential packets with w = 4000 and n = 3 is 13.482806
speedup exponential packets with w = 4000 and n = 5 is 14.692791
speedup exponential packets with w = 4000 and n = 9 is 16.763318
speedup exponential packets with w = 4000 and n = 14 is 16.805151
speedup exponential packets with w = 4000 and n = 28 is 10.432623
speedup exponential packets with w = 8000 and n = 2 is 25.911302
speedup exponential packets with w = 8000 and n = 3 is 27.579255
speedup exponential packets with w = 8000 and n = 5 is 29.516639
speedup exponential packets with w = 8000 and n = 9 is 32.175600
speedup exponential packets with w = 8000 and n = 14 is 33.636067
speedup exponential packets with w = 8000 and n = 28 is 27.26255

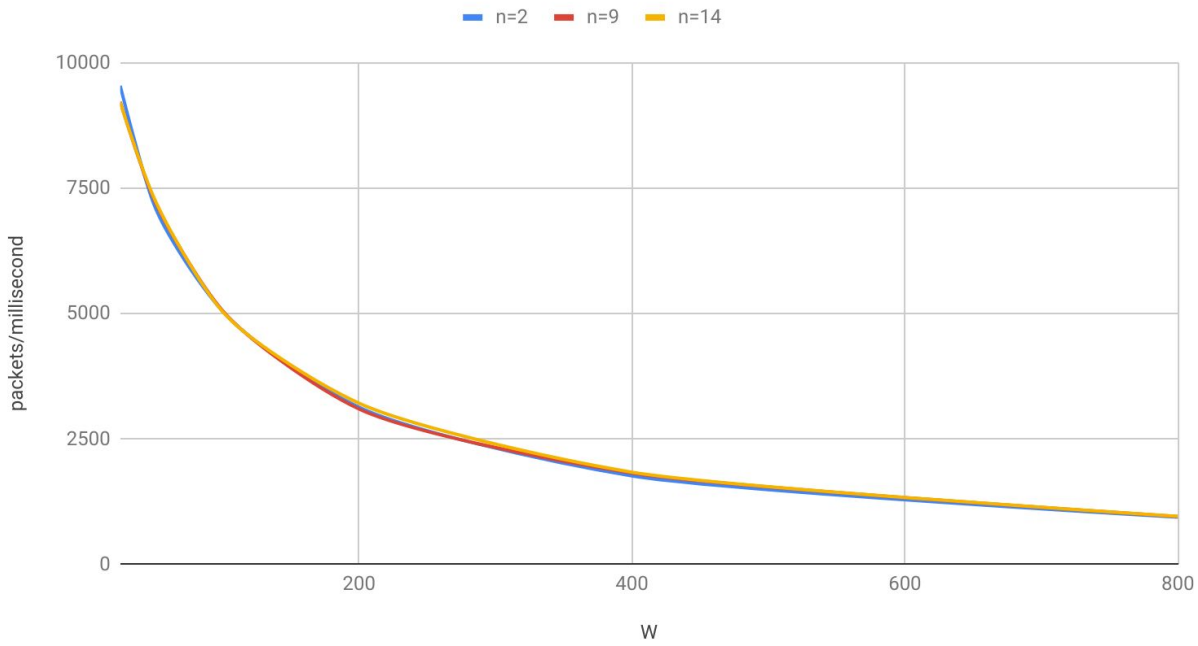
Queue Overhead Speedup



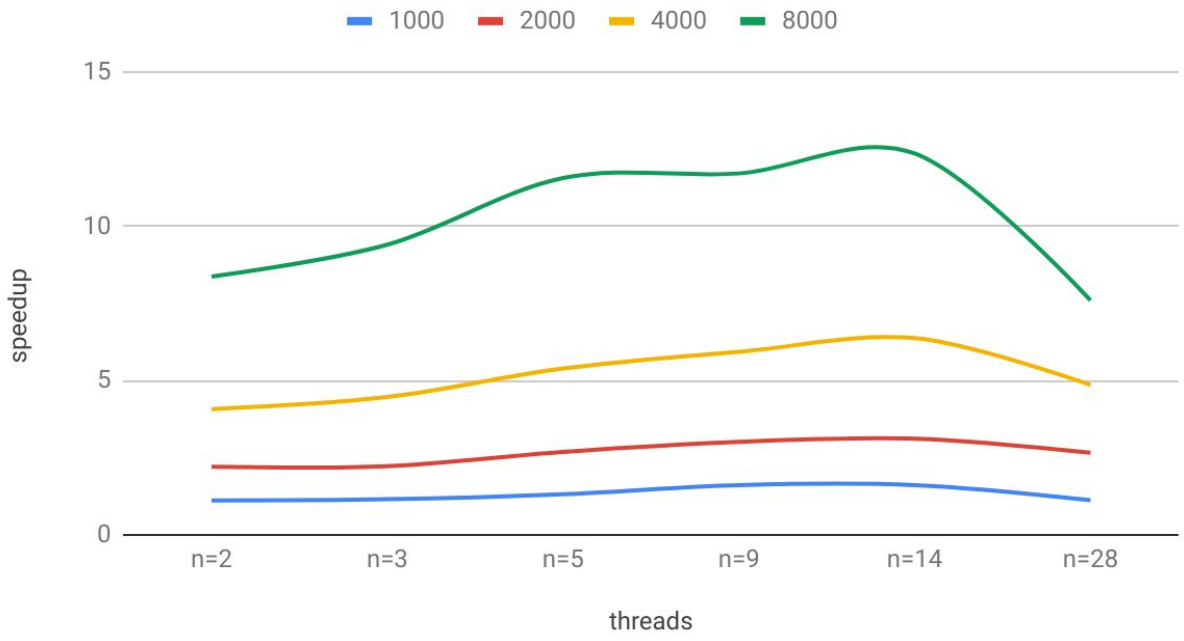
Dispatcher Rate



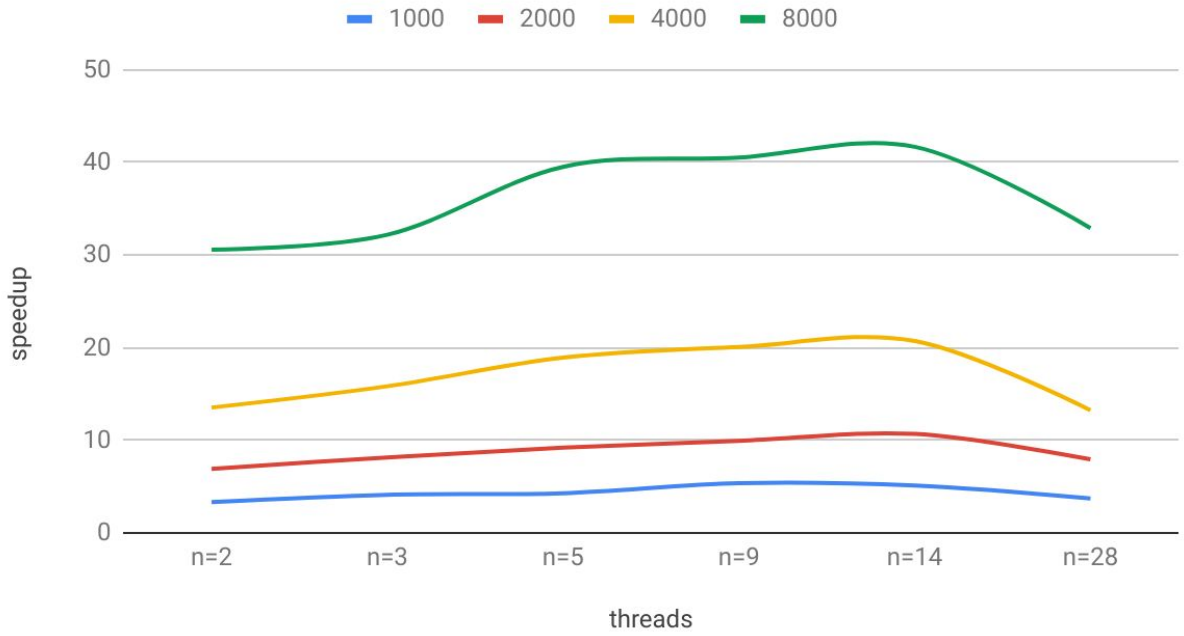
Worker Rate



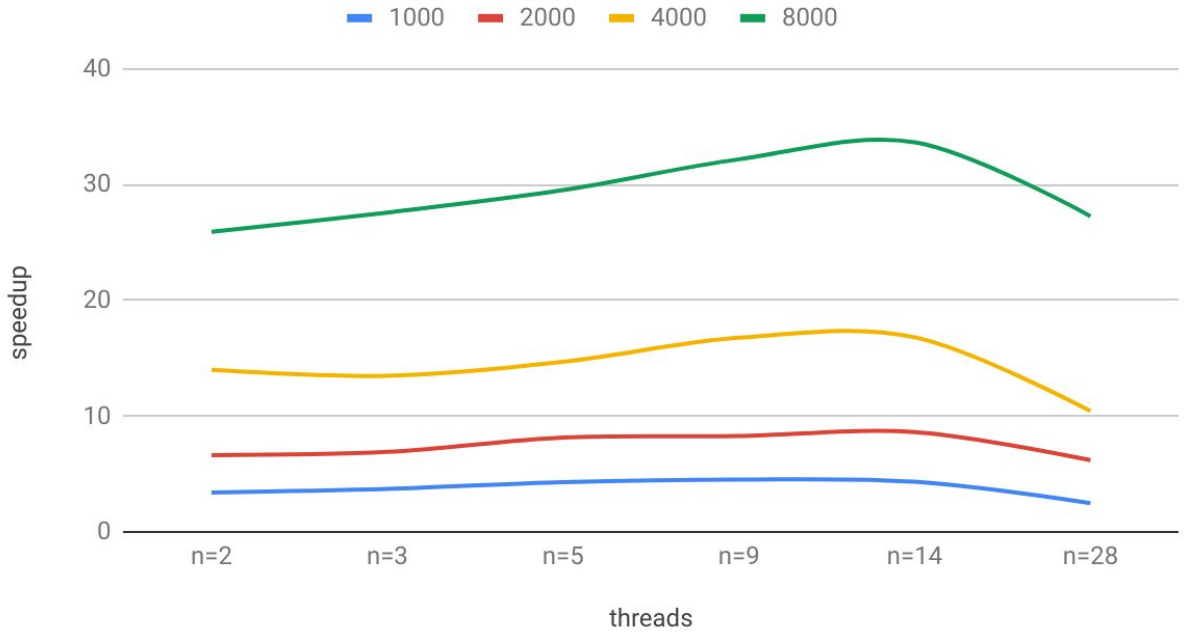
Constant Packet Speedup



Uniform Packet Speedup



Exponential Packet Speedup



- Performance Results

Overall, the results of my performance tests somewhat fall in line with my hypothesis. Starting with queue overhead, I predicted that the extra run time due to the queue would be most pronounced at higher thread counts. However, the overhead due to enqueuing and dequeuing did not change much as more threads were added (aside from some odd results in the beginning).

The dispatcher rate test fell in line with my hypothesis. I predicted that at higher thread counts, the extra overhead to allocate threads and enqueue/dequeue items would cause the dispatcher rate to decrease. The tests saw some immediate gains when the thread count was increased slightly, but when the thread count was 28, the dispatcher rate dropped significantly. This leads me to believe that the queue overhead, as well as the extra work required of the workers and dispatcher to handle the larger number of threads, leads to a significant decrease in performance at higher thread counts.

The worker rate graph shows exactly what I expected. As W increases, the packets per second goes down significantly. This decrease is not as pronounced in the parallel version however.

I predicted that the speedup between parallel and serial for all the different types of packets would be more or less linear. Also, I suspected that the slope of this linear line would decrease from constant to uniform to exponential. While the graphs showed a very slight linear trend, it was nowhere near what I expected. Interestingly, the speedup dropped significantly at 28 threads. As observed in the dispatcher rate test, very high thread counts introduce a lot of extra overhead. It seems that as thread counts increase, the extra overhead outweighs much of the benefit of the extra parallelization, especially at very high thread counts.

Another interesting result of my test was the increased speedup as the mean work increased. Speedup appears to double for each doubling of the mean work. This is obviously due to this extra work being split among the threads in parallel while being added linearly to the serial version. Also, This change was much more pronounced in the uniform and exponential version when compared to the constant version. This goes against my hypothesis as I suspected that the increased load imbalance would decrease the performance gains as the parallelization would be less effective. I believe this is because the constant packets function I created returned packets that were much less time consuming than the ones returned by the uniform and exponential functions. This would mean that performance gains would be less noticeable as both the parallel and serial versions would finish relatively quickly. Regardless, the exponential packets saw slightly less performance gains than the uniform packets which falls in line with my hypothesis.