CMSC 23010: Homework 1 Final Writeup and Theory

Will Thomas

October 28, 2022

Introduction

The goal of the following documentation is to discuss design changes, implementation challenges, correctness outcomes, and performance outcomes for a program that computes packet checksums in parallel.

Design

Shared Functions

time handler

The time handler is passed virtually every program input as well as a time value. It will write these to a specified file.

Array Sum

My array sum takes an array of integers and returns the sum of every value. As will be shown later, the dispatch thread will repeatedly make calls to it in the condition of a while loop such that it keeps running the loop while an array given to array sum returns a non-zero value.

enq

enq enqueues a packet to a lamport queue.

deq

deq dequeues a packet from a lamport queue.

get_packet

get_packet takes a packet source, a packet type, and the index of the required source and returns that type of packet.

q_init

q_init returns a lamport queue of depth d.

Serial Functions

serial_helper

serial_helper takes as input the number of packet generators n, the packet source, the packet type, and the number of packets per source and iterates through and computes a checksum on all of them.

Concurrent Functions

dispatch

The dispatch function takes as input a void pointer that, when called, is a pointer to a dispatch data struct. It contains the thread id, the n value, the t value, the queue depth value d, the packet type p, and the packet source pointer. It then creates every thread and sends them off (with similar inputs) to the worker function. From there, the dispatch function makes an int array of length n each cell containing the value T and enters a while loop that continually calls array_sum until array_sum returns 0. As implemented in the design document, within the while loop the dispatch thread will iterate through the worker threads, calling get_packet to give them one packet at a time. As discussed in class, if a worker's queue is full, the dispatch will simply wait until the worker's queue is not full to enqueue the packet. Each time it gives a worker a packet, it decrements the value in the array. Once the array is empty, the dispatch thread joins all the workers.

worker

worker takes as input a void pointer to a worker data struct. In the struct, there is the thread id, an initialized lamport queue pointer, and a file pointer (this was just used for correctness testing). The worker then spins in a while loop constantly trying to dequeue from its queue until something comes back (i.e., once the dispatcher gives it a packet). Then, the worker will compute the checksum and de-crement a t counter. Once t reaches zero, meaning that the worker has processed t packets, the function exits.

Serial Queue Functions

sq_dispatch

A thread is called in main and sent to compute sq_dispatch. The thread has the same arguments as the normal dispatch function. It then enters a while loop that is the same as the one in the normal dispatch function (checking to see if an array of n t's is empty), and then iterates through every packet source, enqueues one packet to the corresponding queue(it has initialized $n-1$ lamport queues in an array), and then enters the sq_worker function. After the worker has processed the checksum, it's value in the count array is decremented by one.

sq_worker

The sq_worker function is called from sq_dispatch. It is passed a queue pointer and the value t. It then dequeues one task and computes a checksum.

Main

The program itself takes the following arguments:

1. n, or num generators
2. t, num packets per generator
3. d, queue depth
4. w, amount of work to be done per packet
5. v, or version of code. 1 is serial, 2 is concurrent, 3 is serial-queue
6. p, or packet type. C, U, or E.
7. S, refers to trial number. Set as the seed in the packet generator.

8. output, or the output file to which the timing data needs to be written.

The main function parses the inputs and runs the correct functions accordingly. If v == 1, the serial version of the code is run, and so on and so forth. Before this, it creates the packet source with inputs w, n, and s. All calls to a timer as well as calls to the time handler (time output) function are made within if statements in main according to what version of the program must be run. Once everything terminates, the packet source is deleted.

Testing

Testing is done by a python script called fifo_test.py. Essentially, it runs a bunch of different inputs, small and large and testing a wide cross section of all possible inputs. The outputs (checksums) are recorded via the c code. Because all queue interactions are between one reader and one writer, there are never more than 2 threads in the tests done here. Testing for more than 2 threads would cause a whole host of problems with concurrent file writes, and doing it correctly would require locks, and the benefit would be small. As such, the python script, after running all of these inputs, iterates through the output files (there is one file for serial, one for concurrent, and one for serial-queue) and makes sure that, for inputs that are identical save for the method (serial, concurrent, serial-queue), that every checksum is the same and in the correct order. Once this test passes, we can say with reasonable certainty that the queues uphold first in first out behavior.

Performance

To test performance, I uploaded the makefile, all the needed util files, my c code, and output directory holding the following (empty) text files: concurrent.txt, which concurrent checksums are written to, serial.txt, which serial checksums are written to, squeue.txt, where serial queue checksums are written to, and then three more timing files constant_load.txt, uniform_load.txt, exponential_load.txt. In addition to this, I have experiments.py that runs the required experiments as laid out in the homework document. After running via the SLURM array, we have the following data.

Parallel Overhead

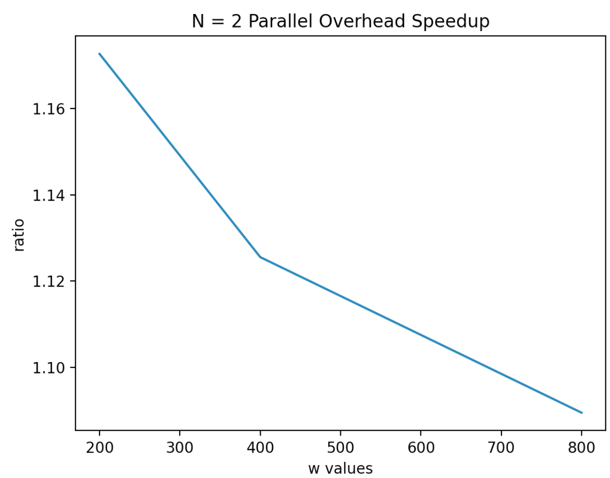
1. Tables:

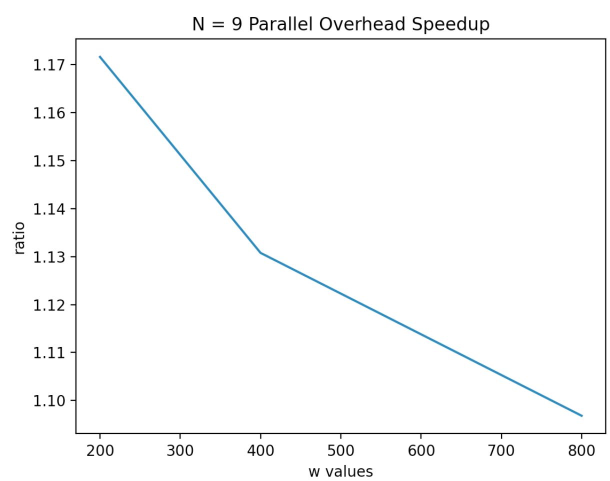
Parallel Overhead

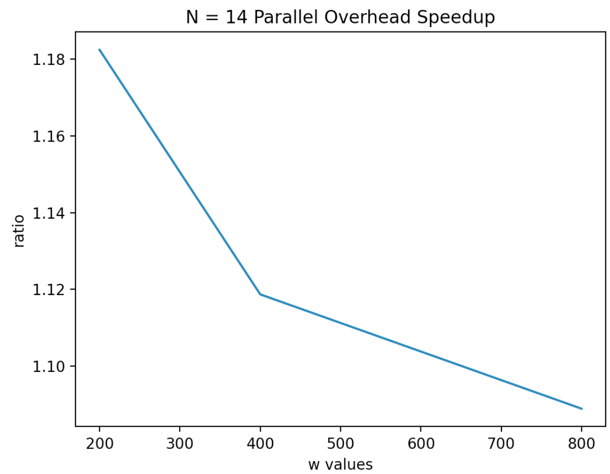
	n	t	w	v	p	time
0	1	2621	200	1	U	3.639
1	8	582	200	1	U	5.500
2	13	374	200	1	U	5.692
3	1	1310	400	1	U	2.915
4	8	291	400	1	U	4.565
5	13	187	400	1	U	4.692
6	1	655	800	1	U	2.635
7	8	145	800	1	U	3.832
8	13	93	800	1	U	4.002
9	1	2621	200	3	U	4.267
10	8	582	200	3	U	6.444
11	13	374	200	3	U	6.730
12	1	1310	400	3	U	3.281
13	8	291	400	3	U	5.162
14	13	187	400	3	U	5.249
15	1	655	800	3	U	2.871
16	8	145	800	3	U	4.203
17	13	93	800	3	U	4.358

	Parallel n	Overhead w	Speedup ratio
0	1.0	200.0	1.172575
1	8.0	200.0	1.171636
2	13.0	200.0	1.182361
3	1.0	400.0	1.125557
4	8.0	400.0	1.130778
5	13.0	400.0	1.118713
6	1.0	800.0	1.089564
7	8.0	800.0	1.096816
8	13.0	800.0	1.088956

2. Plots







3. Analysis

   In my design document, I hypothesized that for each value of $n \in 2, 9, 14$, we will see the speedup approach 1 as W grows large. This is because the overhead required to create and join each thread will become smaller compared to the overall amount of work that needs to be done to compute each checksum. As seen in the plots, this turned out to be correct. The parallel overhead is substantial for small values of w but once more work must be done its share of the over all runtime approaches 1. We can define worker rate for row 9 in the first plot as $\frac{t*1}{run_time} = 614.25$.

Dispatcher Rate

1. Tables

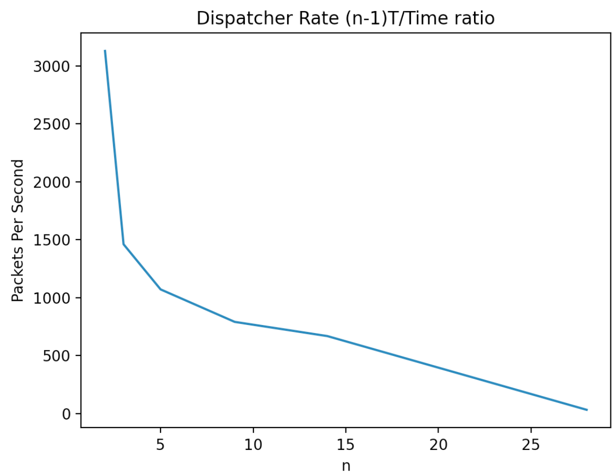
Dispatcher Rate

	n	t	w	v	p	time
0	1	1048576	1	2	U	670.359009
1	2	524288	1	2	U	1076.642944
2	4	262144	1	2	U	1223.426025
3	8	131072	1	2	U	1490.965942
4	13	80659	1	2	U	1688.828003
5	27	38836	1	2	U	33352.761719

Dispatch Rate Ratio

	n	ratio
0	2.0	3128.401307
1	3.0	1460.896585
2	5.0	1071.352066
3	9.0	791.197147
4	14.0	668.644763
5	28.0	32.603237

2. Plots



3. Analysis

   It appears that my hypothesis was incorrect. Packets per second plummets sharply and immediately once n begins to increase, but the decrease tapers off in steepness slightly after $n = 10$. I believe that the reason for this is all of the context switches involved in having one dispatch thread enqueueing to an increasing number of workers.

Speedup with Constant Load

1. Tables

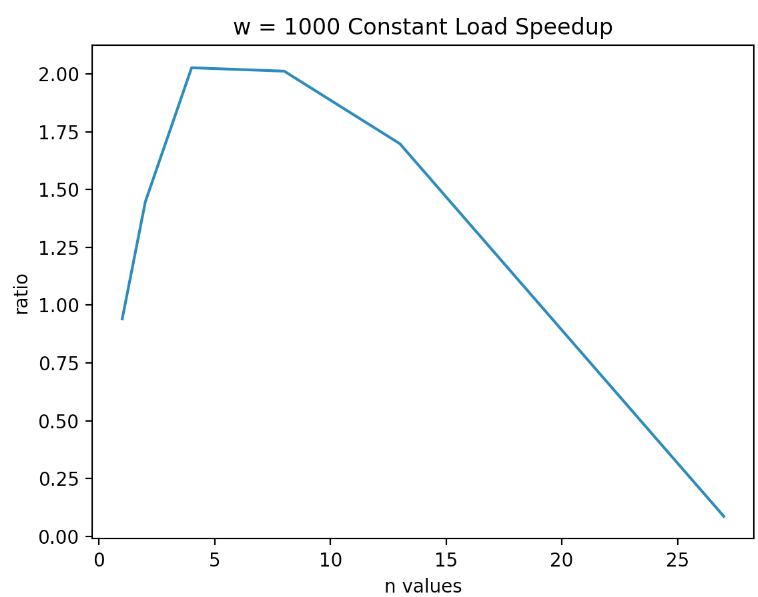
Constant Load

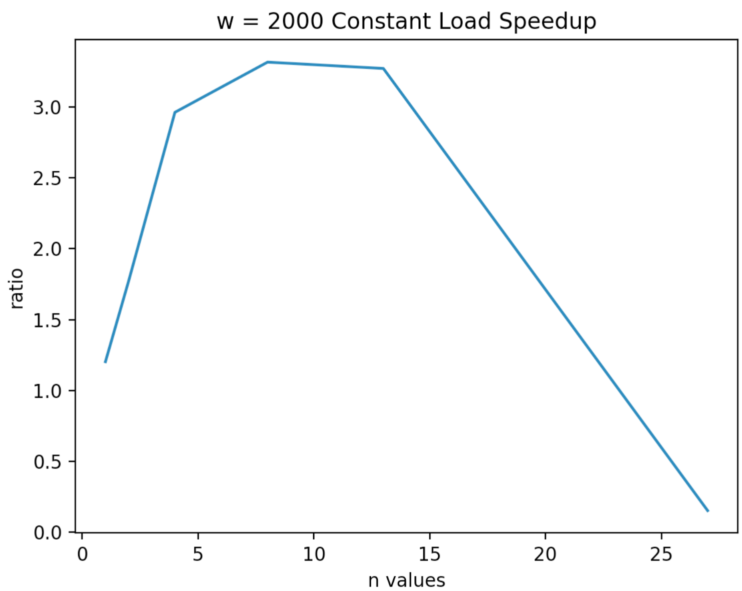
	n	t	w	v	p	time
0	1	32768	1000	1	C	96.744003
1	2	32768	1000	1	C	204.634995
2	4	32768	1000	1	C	371.614014
3	8	32768	1000	1	C	759.539978
4	13	32768	1000	1	C	1225.031006
5	27	32768	1000	1	C	2520.760986
6	1	32768	2000	1	C	221.998001
7	2	32768	2000	1	C	348.661987
8	4	32768	2000	1	C	694.750000
9	8	32768	2000	1	C	1382.979980
10	13	32768	2000	1	C	2267.410889
11	27	32768	2000	1	C	4684.289062
12	1	32768	4000	1	C	394.412994
13	2	32768	4000	1	C	668.049988
14	4	32768	4000	1	C	1356.021973
15	8	32768	4000	1	C	2701.454102
16	13	32768	4000	1	C	4364.558105
17	27	32768	4000	1	C	8967.533203
18	1	32768	8000	1	C	662.591003
19	2	32768	8000	1	C	1335.661011
20	4	32768	8000	1	C	2636.427979
21	8	32768	8000	1	C	5245.382812
22	13	32768	8000	1	C	8513.212891
23	27	32768	8000	1	C	17646.980469
24	1	32768	1000	2	C	102.857002
25	2	32768	1000	2	C	141.376007
26	4	32768	1000	2	C	183.347000
27	8	32768	1000	2	C	377.483002
28	13	32768	1000	2	C	721.348999
29	27	32768	1000	2	C	29096.060547
30	1	32768	2000	2	C	184.481003
31	2	32768	2000	2	C	196.837997
32	4	32768	2000	2	C	234.518005
33	8	32768	2000	2	C	416.994995
34	13	32768	2000	2	C	693.004028
35	27	32768	2000	2	C	30581.335938
36	1	32768	4000	2	C	346.024994
37	2	32768	4000	2	C	378.838013
38	4	32768	4000	2	C	407.683990
39	8	32768	4000	2	C	480.928009
40	13	32768	4000	2	C	683.429993
41	27	32768	4000	2	C	31493.800781
42	1	32768	8000	2	C	671.129028
43	2	32768	8000	2	C	703.017029
44	4	32768	8000	2	C	719.408997
45	8	32768	8000	2	C	758.794006
46	13	32768	8000	2	C	826.955017
47	27	32768	8000	2	C	32974.414062

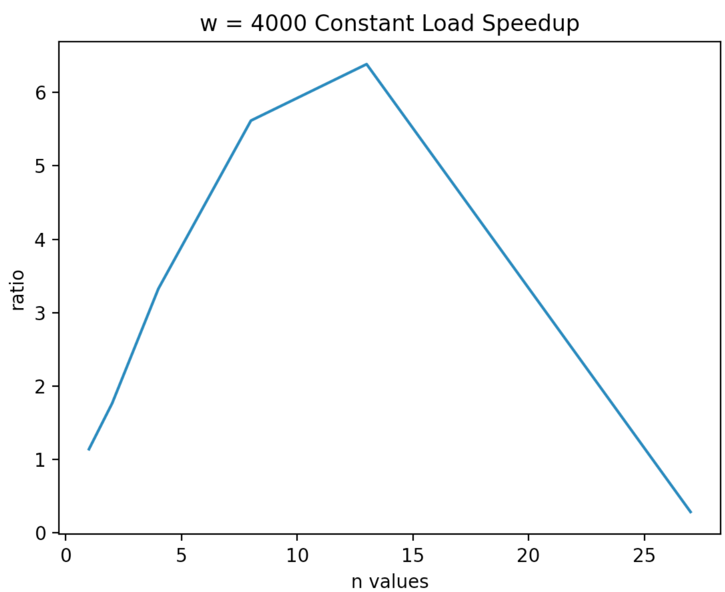
Constant Load Ratio

	n	w	ratio
0	1.0	1000.0	0.940568
1	2.0	1000.0	1.447452
2	4.0	1000.0	2.026834
3	8.0	1000.0	2.012117
4	13.0	1000.0	1.698250
5	27.0	1000.0	0.086636
6	1.0	2000.0	1.203365
7	2.0	2000.0	1.771314
8	4.0	2000.0	2.962459
9	8.0	2000.0	3.316539
10	13.0	2000.0	3.271858
11	27.0	2000.0	0.153175
12	1.0	4000.0	1.139840
13	2.0	4000.0	1.763419
14	4.0	4000.0	3.326159
15	8.0	4000.0	5.617169
16	13.0	4000.0	6.386255
17	27.0	4000.0	0.284740
18	1.0	8000.0	0.987278
19	2.0	8000.0	1.899899
20	4.0	8000.0	3.664714
21	8.0	8000.0	6.912789
22	13.0	8000.0	10.294651
23	27.0	8000.0	0.535172

2. Plots







3. Analysis

   I initially hypothesized that, as was the case in parallel overhead, when the packets are more work intensive the speedup will be more pronounced. Because each packet will be taking more time, the benefit from doing them concurrently will begin to outweigh the downside of parallel overhead. I believe that as the number of cores increases we will see a slowdown for smaller amounts of packet work but a more significant speedup for larger amounts of packet work.

   This is correct in a way, but not entirely. Obviously, increasing the number of cores will only provide benefit up to a certain point. Interestingly, while the speedup is not more pronounced for more work intensive packets, there is more room for speedup with adding more cores. That is, the w = 4000

Constant Load Speedup increases up until nearly 15 cores, while the w = 1000 Constant Load Speedup begins to decrease after about 5 cores.

Also, there should be a steeper plummet after 16 cores, but the graph is too smoothed to show this clearly.

Speedup with Uniform Load

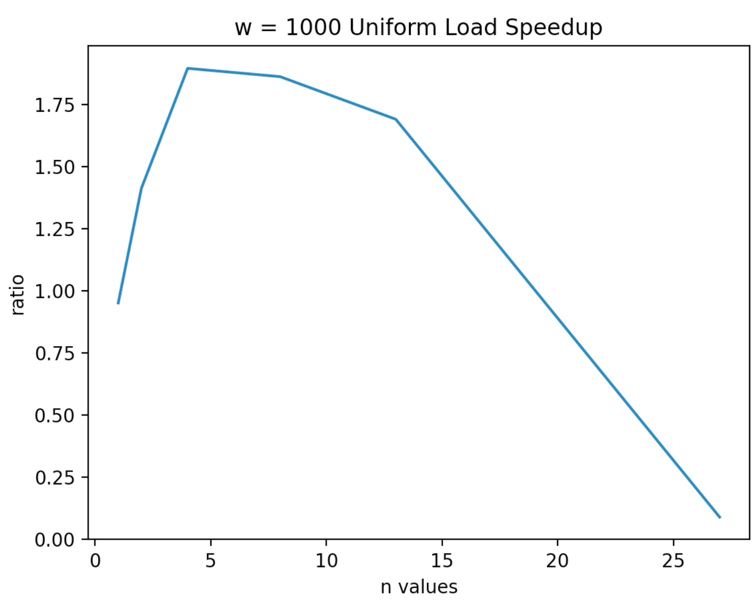
1. Tables

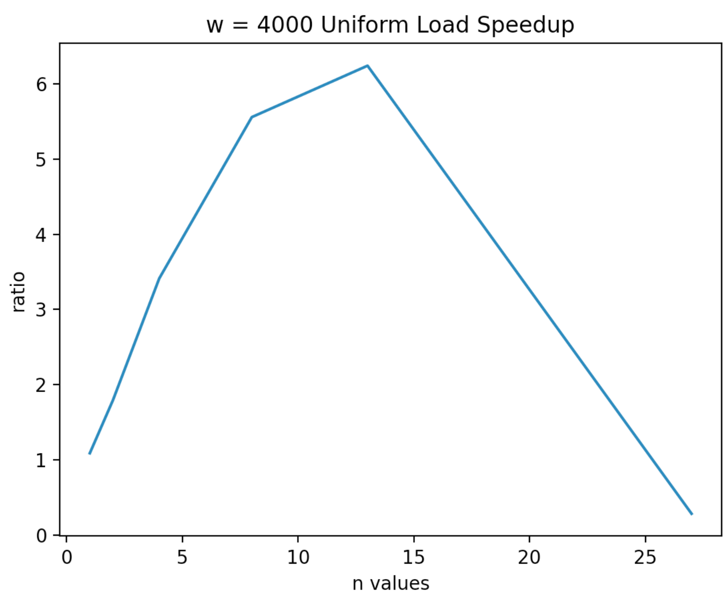
Uniform Load

	n	t	w	v	p	time
0	1	32768	1000	1	U	97.109001
1	2	32768	1000	1	U	188.123001
2	4	32768	1000	1	U	373.105011
3	8	32768	1000	1	U	762.161987
4	13	32768	1000	1	U	1212.668945
5	27	32768	1000	1	U	2547.112061
6	1	32768	2000	1	U	214.677994
7	2	32768	2000	1	U	348.790009
8	4	32768	2000	1	U	722.392029
9	8	32768	2000	1	U	1414.987061
10	13	32768	2000	1	U	2279.260010
11	27	32768	2000	1	U	4689.376953
12	1	32768	4000	1	U	377.338989
13	2	32768	4000	1	U	686.687988
14	4	32768	4000	1	U	1364.119019
15	8	32768	4000	1	U	2706.056885
16	13	32768	4000	1	U	4377.330078
17	27	32768	4000	1	U	8983.942383
18	1	32768	8000	1	U	662.661011
19	2	32768	8000	1	U	1314.911987
20	4	32768	8000	1	U	2650.045898
21	8	32768	8000	1	U	5278.749023
22	13	32768	8000	1	U	8558.045898
23	27	32768	8000	1	U	17661.812500
24	1	32768	1000	2	U	102.148003
25	2	32768	1000	2	U	133.244003
26	4	32768	1000	2	U	196.923004
27	8	32768	1000	2	U	409.496002
28	13	32768	1000	2	U	717.625000
29	27	32768	1000	2	U	28481.304688
30	1	32768	2000	2	U	184.251007
31	2	32768	2000	2	U	213.177002
32	4	32768	2000	2	U	249.451996
33	8	32768	2000	2	U	417.031006
34	13	32768	2000	2	U	707.187012
35	27	32768	2000	2	U	30815.359375
36	1	32768	4000	2	U	346.903015
37	2	32768	4000	2	U	382.822998
38	4	32768	4000	2	U	399.739990
39	8	32768	4000	2	U	486.825989
40	13	32768	4000	2	U	701.234985
41	27	32768	4000	2	U	31717.710938
42	1	32768	8000	2	U	671.500000
43	2	32768	8000	2	U	706.607971
44	4	32768	8000	2	U	729.161987
45	8	32768	8000	2	U	749.866028
46	13	32768	8000	2	U	812.281982
47	27	32768	8000	2	U	31604.693359

Uniform Load Ratio	n	w	ratio
0	1.0	1000.0	0.950670
1	2.0	1000.0	1.411868
2	4.0	1000.0	1.894675
3	8.0	1000.0	1.861220
4	13.0	1000.0	1.689837
5	27.0	1000.0	0.089431
6	1.0	2000.0	1.165139
7	2.0	2000.0	1.636152
8	4.0	2000.0	2.895916
9	8.0	2000.0	3.393002
10	13.0	2000.0	3.222995
11	27.0	2000.0	0.152177
12	1.0	4000.0	1.087736
13	2.0	4000.0	1.793748
14	4.0	4000.0	3.412516
15	8.0	4000.0	5.558571
16	13.0	4000.0	6.242316
17	27.0	4000.0	0.283247
18	1.0	8000.0	0.986837
19	2.0	8000.0	1.860879
20	4.0	8000.0	3.634372
21	8.0	8000.0	7.039590
22	13.0	8000.0	10.535807
23	27.0	8000.0	0.558835

2. Plots





3. Analysis

   I hypothesized similar performance for uniform speedup as I did for constant speedup. The performance is similar, but not int he way that I hypothesized. Regardless, we see a similar increasing speedup in n as w grows larger, which makes sense. As long as each worker is spending more time on each packet, there is more benefit to be gained from increased parallelism.

Speedup with Exponential Load

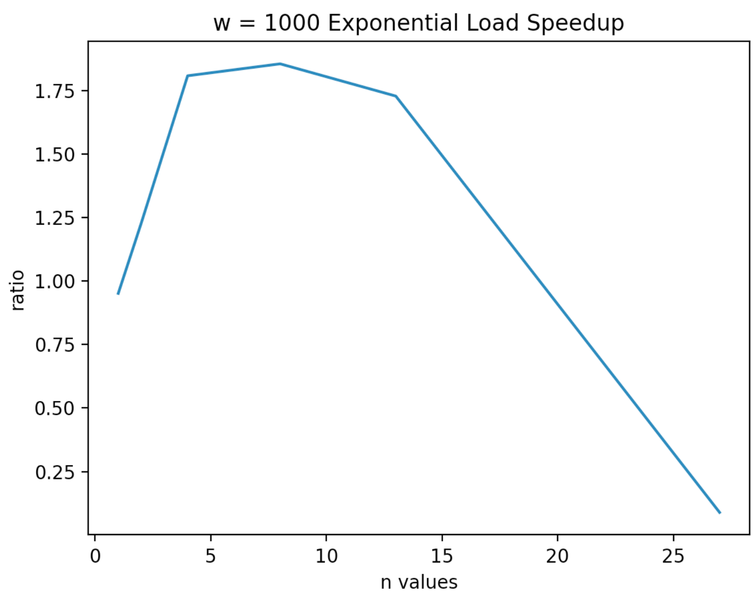
1. Tables

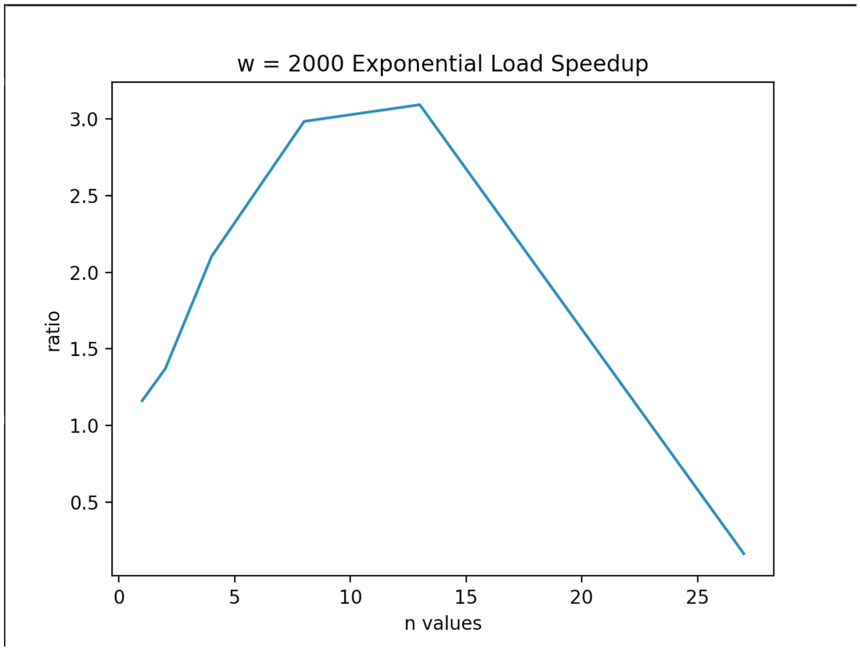
Exponential Load

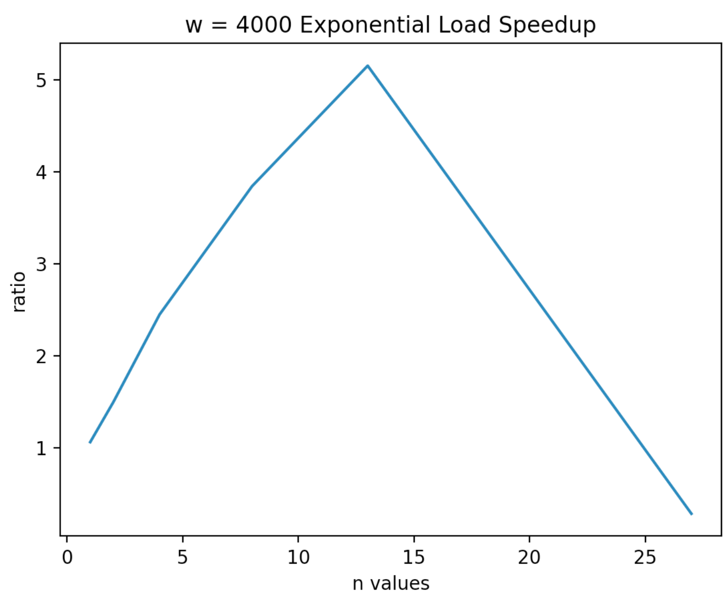
	n	t	w	v	p	time
0	1	32768	1000	1	E	97.744003
1	2	32768	1000	1	E	195.276001
2	4	32768	1000	1	E	378.114990
3	8	32768	1000	1	E	776.640991
4	13	32768	1000	1	E	1244.142944
5	27	32768	1000	1	E	2569.392090
6	1	32768	2000	1	E	215.617004
7	2	32768	2000	1	E	350.950012
8	4	32768	2000	1	E	701.567993
9	8	32768	2000	1	E	1399.058960
10	13	32768	2000	1	E	2276.466064
11	27	32768	2000	1	E	4754.986816
12	1	32768	4000	1	E	369.097992
13	2	32768	4000	1	E	693.989990
14	4	32768	4000	1	E	1362.447998
15	8	32768	4000	1	E	2690.397949
16	13	32768	4000	1	E	4383.090820
17	27	32768	4000	1	E	9019.209961
18	1	32768	8000	1	E	663.372009
19	2	32768	8000	1	E	1318.875977
20	4	32768	8000	1	E	2634.879883
21	8	32768	8000	1	E	5259.047852
22	13	32768	8000	1	E	8557.749023
23	27	32768	8000	1	E	17728.572266
24	1	32768	1000	2	E	102.754997
25	2	32768	1000	2	E	158.817993
26	4	32768	1000	2	E	209.128998
27	8	32768	1000	2	E	418.652008
28	13	32768	1000	2	E	719.937988
29	27	32768	1000	2	E	28877.468750
30	1	32768	2000	2	E	185.649994
31	2	32768	2000	2	E	256.148010
32	4	32768	2000	2	E	333.587006
33	8	32768	2000	2	E	468.613007
34	13	32768	2000	2	E	735.658997
35	27	32768	2000	2	E	29157.218750
36	1	32768	4000	2	E	346.871002
37	2	32768	4000	2	E	462.992004
38	4	32768	4000	2	E	555.992981
39	8	32768	4000	2	E	699.890015
40	13	32768	4000	2	E	850.616028
41	27	32768	4000	2	E	31520.480469
42	1	32768	8000	2	E	670.119995
43	2	32768	8000	2	E	867.862976
44	4	32768	8000	2	E	1036.131958
45	8	32768	8000	2	E	1203.732056
46	13	32768	8000	2	E	1366.125000
47	27	32768	8000	2	E	32419.478516

	Parallel n	Overhead w	Speedup ratio
0	1.0	200.0	1.172575
1	8.0	200.0	1.171636
2	13.0	200.0	1.182361
3	1.0	400.0	1.125557
4	8.0	400.0	1.130778
5	13.0	400.0	1.118713
6	1.0	800.0	1.089564
7	8.0	800.0	1.096816
8	13.0	800.0	1.088956

2. Plots







3. Analysis

   I hypothesized that the exponentially distributed packets will lead to serious imbalances in work, especially as W gets large. I also hypothesized that we will get the best performance from moderately small values of W and moderately small values of n, and that we will get very bad performance from large values of both n and W due to the compounded negative effects of large amounts of parallel overhead and extremely work intensive packets mixed in that may effect certain workers disproportionately.

   This again was not quite what happened in reality. The first plot showing the w = 1000 speedup looks similar to the other two w = 1000 speedup graphs for constant and uniform packets. The max speedup is slightly smaller than the max speedup for constant or uniform, but this is to be expected as a negative effect of outlier packets that take up a lot of work.

The exponential speedup graph shows a similar smaller overall speedup as well as an increasing but almost flat plateau after about $n = 8$. Still, more work intensive packets is allowing more benefit to be gained from increased parallelism. As expected, after 15 cores, the performance drops off.

Finally, looking at the w = 4000 speedup, we see a jagged increase going up to about 14 cores that quickly plummets. The speedup is smaller than the w = 4000 speedup for constant and uniform, which is once again probably an effect of outlier packets. It seems that, on the whole, outlier packets do not have nearly as much of a detrimental effect on performance as I expected, especially as the average amount of work gets large.

Theory Problems

■ Exercise 25:

If we drop condition L2 from the linearizability definition, is the resulting property the same as sequential consistency?

No; the first condition simply implies that there exists an extension such that its completion is equal to a legal sequential history. Note that legal sequential history simply means that things precede each other; although things may happen simultaneously it can still be interpreted as an order. However what we need to recognize now is that this does not necessarily mean that there only exists one sequential order that an extension can be equal to. Consider having overlapping events such that we have two valid histories where the only difference is that two events are switched in order. What we can recognize now is that we can interpret our history in two ways, one of which is not necessarily the program order.

■ Exercise 29:

Is the following property equivalent to saying that object x is wait-free?

For every infinite history H of x, every thread that takes an infinite number of steps in H completes an infinite number of method calls.

For a method to be wait-free it must be the case that all method calls finish in a finite amount of time, and an object is wait-free if all of its methods are wait free.

From here, we can prove the statement. Consider a wait-free object that has threads that take an infinite number of steps to execute. Suppose for contradiction that one of these threads has a finite number of method calls. Because of this, for the thread to still take infinite steps to execute it must be the case that one of these method calls takes infinite time, implying that x is not wait free.

In the other direction, suppose that in every infinite history H of the aforementioned object, infinite method calls must be made by any thread that runs infinitely. If we suppose for the sake of contradiction that the object is not wait-free, the implication is that there is some method of the object that is not wait-free, which means that this method could be called and not complete in infinite steps. This is a contradiction. As such, the property is equivalent to saying that the object x is wait-free.

■ Exercise 30:

Is the following property equivalent to saying that object x is lock-free?

For every infinite history H of x, an infinite number of method calls are completed.

We can define a method to be lock-free if infinitely often some method completes in finite time. Also, note that an object x is lock free if all of its methods are lock free. If we assume that lock-free object x has threads running infinitely, we can suppose for contradiction that there are only finite method calls that get completed.

It is important to realize that an infinite history implies at all points in time that there exists some kind of action after t. It is also important to notice that a finite number of method calls completed implies that there must be a most recent completed method call, meaning that no method call is completing in finite time, which is a direct contradiction, implying x is not lock free.

To prove the other direction, suppose that we have an object x that is not lock free but also such that every infinite history H of x has infinite method calls. This directly implies that there is a method of x that is not lock free. Therefore the problem statement and lock freedom are equivalent.

■ Exercise 31:

Consider the following rather unusual implementation of a method m. In every history, the ith time a thread calls m, the call returns after 2i steps. Is this method wait-free, bounded wait-free, or neither?

First we can recognize that for $n > 0$ where n is a number of steps, we know that the nth method call of m will take more than n steps. As a result of our being able to find a method call of m which can consistently supercede any bound on number of steps, we can conclude that m is not bounded wait-free.

■ Exercise 32:

This exercise examines a queue implementation (Fig. 3.17) whose enq() method does not have a linearization point.

There is no linearization point. Two threads can call getandincrement() at the same time. In fact, one thread can call getandincrement() and another can call items[i].set(X).