Final Write-up

Changes Made to The Original Design

Code Infrastructure

I tweaked my code so that in the end we have 3 binaries, **serial, parallel, serial_queue**, and so that it takes the same command line arguments as specified before but now also with a distribution argument for either constant, uniform or exponential packets.

Correctness Testing

Stress Testing

In my initial write-up, I suggested that calculating the sum of all packet checksums from a particular source, since this would catch packets being sent to the wrong queue. However, this has no guarantee that we maintain the FIFO ordering of queues. In particular, we'd like to have that if A is enqueued before B, then A is dequeued before B.

To test this, I will write a unit test for the queues themselves. This means that I will keep the checksum-per-source test to ensure the dispatcher dispatches correctly, but add another test for the queues. The way I will do this is to stress test the queue at their extremes. In particular, odd behaviour is most likely to happen when a thread spins: either when the queue is full, or when the queue is empty.

To get the queue into these conditions, I will vary the speeds of the dequeue and enqueue methods. When I want to stress test the queue during a full condition, I'll have the dequeue method include a sleep, while the enqueuer enqueues at a normal rate. This means that if you have the dispatcher thread enqueueing and a worker thread dequeuing (for this test, the worker will not compute a checksum), then the queue will fill up much more quickly. Conversely, to stress-test the queue when it's empty, I will add a sleep call to the enqueue method.

Concretely in code, the stress test will have the dispatcher grabbing constant packets. The way constant packets are generated in my program is that the PacketGenerator internally has a counter of which seed to put in next for the next constant packet. Thus, all I have to check is that the worker is receiving packets whose seeds are 0, 1, 2, etc. This will verify correctness.

Moreover, I will not be doing any checksum computation for this queue stress-test, as the purpose is to test that items go in and out FIFO and that no packets are lost. You can view the test with `make stress_test`, and running it with either "./stress_test D" or "./stress_test E", indicating whether you'd like the dequeue or enqueue operation to be slowed.

Trials Per Experiment

Also note that we will be doing 12 trials for exponential distributions, 5 for uniform, and 1 for constant. This follows from class discussion on getting representative data.

Where to Start Timing From

We include setup and teardown of the queues because we want to benchmark our serial-queue against our parallel implementation. To replicate this, the only difference should be that the parallel program has threads. Thus, we need to account for set up and teardown time of the queues. In the grand scheme of things, this should not play a large role because a firewall is usually set up and then runs for weeks on end, but we include this cost nonetheless for correctness.

Lamport Queue Implementation

Something I didn't cover last time was the queue API. In particular, if the queue is full, should the queue method spin, or should it return control back to the caller and the caller then spins? We will go with the latter, as it provides more flexibility.

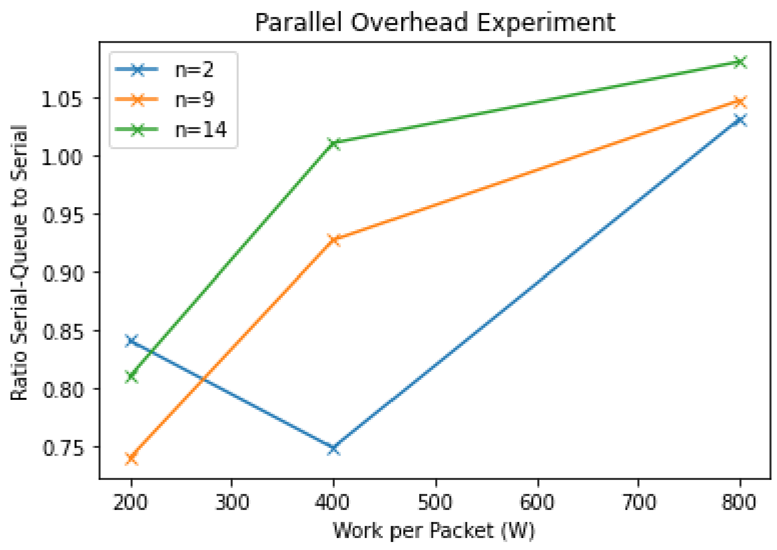
In terms of setting up the memory barrier, we use the `__sync_synchronize` barrier in our queue operations.

Source Code Changes

Also, I modified the packetSource so that getConstantPacket works. I changed the packetSource struct so that it has a variable that is set as the seed for each constant packet, which gets incremented every time a constant packet is generated.

Experimental Results

Parallel Overhead

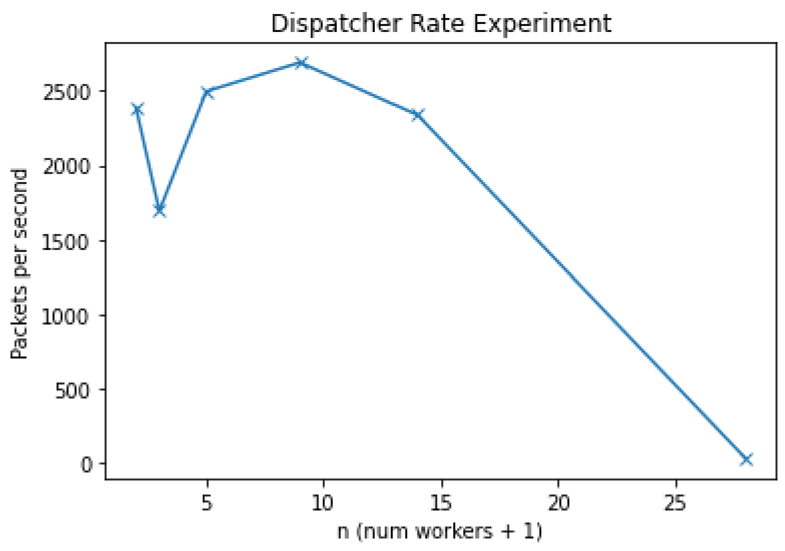


The above graph says that serial-queue speed approaches serial speeds as work per packet (W) increases. I am initially tempted to say that this lines up with what I'd expect. Initially, the serial-queue pays for an overhead of setting up queues, so as the total amount of work increases, then the overhead becomes a smaller cost of the total program compared to computation, thus serial_queue speed approaches parallel speed.

However, this does not take into account that the number of packets processed per thread, T, is based on W. Thus, the total amount of work between runs is actually the same. I might then explain the increased serial_queue to serial speed ratio with work as follows: As you increase the amount of work per packet (but maintain a constant amount of overall work), then each packet spends more time getting processed by both serial-queue and serial implementation. Thus, this means that there is less back-and-forth for the serial-queue implementation to go between writing packets to queue and then emptying the queues. This decrease in back and forth leads to a serial_queue speedup. By this logic, we could also say that if you increase the depth, then there is less back and forth and more speedup for the serial_queue approach.

The variation in n also makes sense now. As you increase n, you increase the amount of packets that the queues can store, thus the serial_queue implementation wastes less time going back and forth. This is why the speedup ratio is higher for higher n's than it is for lower n's.

Dispatcher Rate

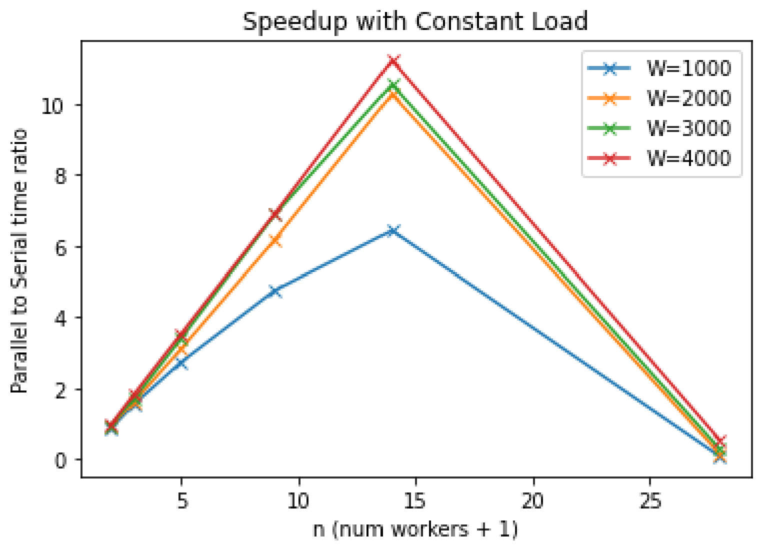


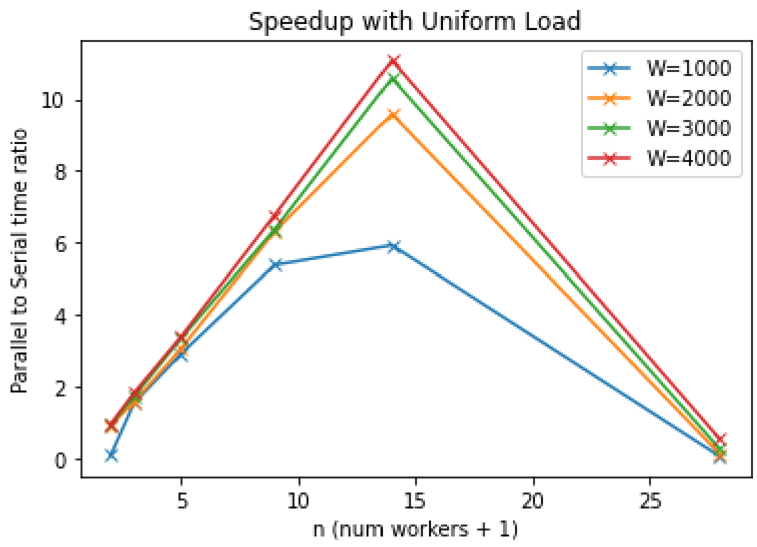
This graph somewhat lines up with my hypothesis– as you spawn more threads, you increase the overhead of managing more queues and threads so you pay for this in processing less packets per second.

I would have actually expected the curve to always decrease, as spawning more threads and queues only increases the overhead of managing extra data structures. The peak at n=10 doesn't make sense to me. Perhaps this means that the dispatcher is enqueueing at a rate which would warrant the extra capacity provided from having more queues at n =10, but not worth the overhead at n=14.

The drop off at n =28 makes sense. The slurm machines only have 14 usable cores, so spawning 28 threads means that there are 2 threads assigned per core. We pay for such an arrangement, as there is context switching eg. you're forced to switch registers and load in and out as one core has to switch between allowing thread one versus thread two.

Constant + Uniform





I will discuss the constant and uniform load runs together because the graphs are very similar.

This fact lines up with what we'd expect: the uniform distribution and constant distribution are very similar in that each thread very likely gets about the same amount of work. Even though uniform is more variable, we're only drawing from [0,2W], so the variance isn't huge. Moreover, we're running the uniform experiment multiple times, so we expect regression towards mean behaviour, which is essentially the behaviour of constant packets.

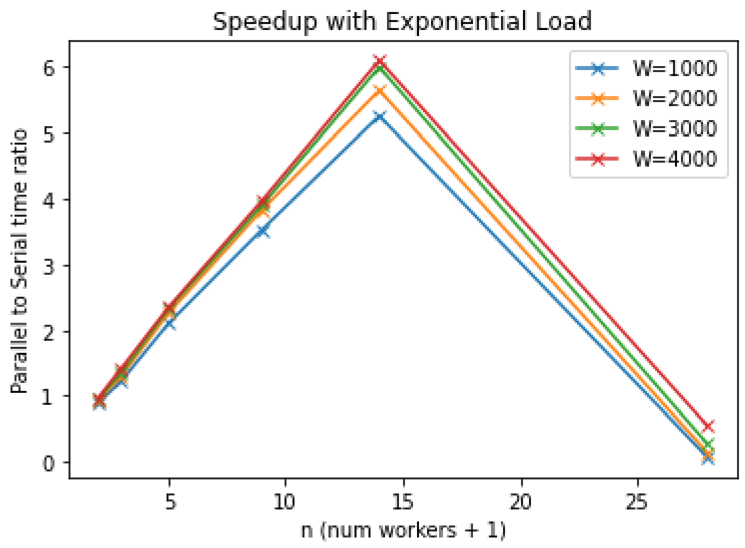
The curve also lines up in that as we add more threads (n increases), then the parallel speed up becomes higher compared to serial. This is because we literally have more threads doing the work, and these threads are using a single core to its maximum. This is unlike the n=28 dropoff case because that's when threads share a core, which leads to inefficiencies described previously, hence the steep decline.

Now, It also makes sense that the parallel speed-up ratio is higher for higher W's because increasing W increases the total amount of compute work, which makes thead-spawn overhead much lower when compared as a fraction to the total work. This is ever so slight, but still noticeable in the graphs above.

I'm honestly not sure why the blue curve (W=1000) doesn't rise with the other for n=14. My guess would be that since W is small, and so the total amount of work is small, spawning more threads at n=14 and the thread overhead takes up a larger fraction of the total work, so the speedup isn't as noticeable there.

But either way, 10x speedup is great!

Exponential



This graph follows the same shape as above for the same reasons: more threads means more workers to do tasks., but too many (at n=28) means that cores are overloaded with threads and become inefficient.

The smaller magnitude is also explained by how exponential loads have more variance in work among queues in the parallel program. Thus, the speed of our program devolves to the speed of the slowest queue, which here, is about twice as slow as the average queue in the constant or uniform case.

Theory HW

Ex 25
Yes, if you drop condition L2 of linearizability, you get the definition that a history H is (linearizable, or something new we're trying to define) if it has an extension H' and there is a legal sequential history S such that complete(H') is equivalent to S.

This just means that complete(H') i.e. H' where every method invocation also has a return is a valid sequential order where method call orders in a thread are preserved. This is the definition of sequential consistency.

Ex 29
Yes.
An object x being wait-free means that every call to x (i.e. a method modifying it) finishes its execution in a finite number of steps.

Here is what we were given: "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."

One can read this as: "for a given infinite history H of x, if you are a thread that takes an infinite number of steps in H, then you complete an infinite number of method calls."

The contrapositive is "if you are a thread that doesn't complete an infinite number of method calls, then you are a thread that does not complete an infinite number of steps."

I.e. if you are a thread that calls x, then your execution finishes in a finite number of steps, exactly the definition of wait-free.

Ex 30

No, the statement does not mean it's lock-free.

Lock-free means that "infinitely often, some method call finishes in a finite number of steps." This means that certain threads may starve, i.e. the method calls from those threads are made infinitely often and not served. This does not align with the statement we're given, because we have an infinite number of method calls that are NOT completed.

Ex 31
This method is wait-free at the least because every call finishes its execution in a finite number of steps, given that 2^i is some real non-infinite number for a real number i.

It is bounded wait-free by the bound of 2^i for each method call. It's fuzzy because you wouldn't say that the method is bounded wait-free (in general) for some bound B because you can always find a call to the method i such that 2^i > B.

However, the method is bounded wait free in the sense that each method call i is bounded.

Ex 32