Homework 3, Part 1

Cooper Faber

May 2021

1 Results

1.1 Table

Sync	Async	Enq8	Enq8API
2.566s	3.013s	2.283s	1.978s

1.2 Graph

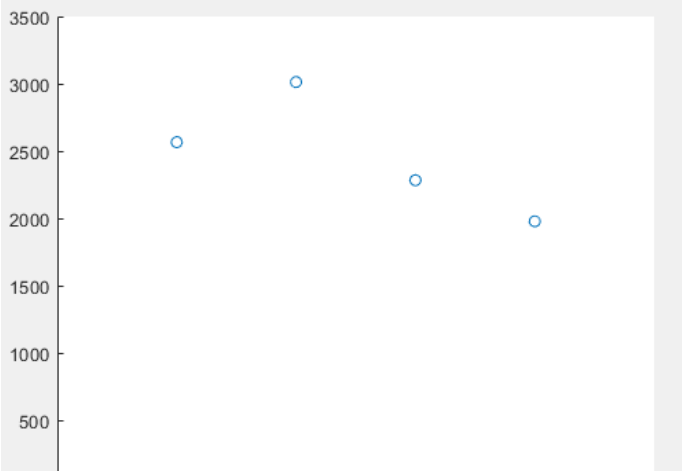

Figure 1: Results

1.3 Context

For part 1, we were given 4 different variations on a concurrent queue - a Synchronous one, with one slot; an Asynchronous one, with many slots; and two variations on the asynchronous queue, where we experimented on optimization in the main() function. Results in the graph above are symmetrical with the figure, i.e Sync is first, Async is second, and so on.

2 Implementation

My implementation for the sync and async queue are both more or less from the notes given in class - the pseudocode was very helpful. For the async queue, my basic enq() and deq() functions check both the tail and head values, to verify that the queue is empty, before proceeding with their operation. I originally attempted to use atomic_fetch_add to increment the head tail, but quickly realized that not only was it not performance efficient, it wasn't necessary; and worse, made it much more difficult to do modular arithmetic. My implementation instead uses x.store((x.load()+1)%SIZE) to keep the buffer circular. For queue8, my implementation was definitely not optimal, performance wise - however, it is still much faster than the synchronous version.
Queue8 has one outer loop, for the total amount of values, and 4(!) inner loops that load the array, enq the value, deq the corresponding result, and a final loop to store the results in the original array. An obvious improvement would be to combine the first and last two loops, directly en-queuing when you receive a value and sending it back when you dequeue. (Honestly, I'd probably implement this if I wasn't so close to the deadline. I'll put it on my list of regrets.)
Finally, my queue8API takes queue8 and makes it more efficient - sending 8 values every time it enqueues/deqeues, and bypassing a lot of atomic incrementing by doing so.

3 Analysis

The values received from timing the program line up with expectations; sync is slower than optimized versions (queue8 and queue8API), and is faster than async. As all of our operations have the same execution time (trig operations), there is no advantage gained from adding more space to the queue if we're still sending things singularly- and it's clearly slowed down by the necessary atomic operations (which, according to piazza, may actually be an issue on my local machine).
Queue8 manages to beat out Sync (though not by much) as it enqueues its data in batches of 8, taking advantage of the kind of chunking covered in previous assignments, and utilizing the extra space in the queue.
Queue8API improves upon queue8 by streamlining the CQueue object, guaranteeing that data will only ever be provided/taken in increments of 8. This allows us to skip 7 atomic operations for each dequeue, which gives the performance boost you see in the above graph.