Part2: Write Up

My implementation for the IO Queue is very similar to what was presented in the CSE113 lecture slides. For my private variables, I create two atomic integers (head and tail) and an integer pointer, which acts as a queue and holds all the enqueued values. I create five functions `init`, `enq`, `deq`, `size`, `deq_32`, which handle all the operations on the queue.

- `init` takes an int that specifies the size of the queue. Head and tail are initialized to zero.
- `enq` pushes a new element to the queue by uses the function `atomic_fetch_add` to increment the head.
- `deq` removes an item from the queue but uses the function `atomic_fetch_add` to increment the tail.
- `size` returns how many elements are in the queue.
- `deq_32` dequeues 32 elements at a time instead of 1 at a time.

The results of part2 demonstrated how doing the same calculation can be done in many ways with greatly differing performance. Based on the Part2 graph that I made, the stealing32 implementation outperforms all the other queues. This is expected because of how many elements it dequeues at a time rather than our other implementations which only dequeues one element at a time. The performance of the stealing binary is somewhat close to stealing32 with a 2.2 second runtime. I guess that is the difference between dequeuing one element and thirty-two at a time. The static binary has a runtime of 13.268 seconds and falls behind the other two. Lastly, the global binary is the slowest out of all. With a run time of 79.255 seconds, this is the most inefficient implementation. Even though I like the idea of an implicit worklist to speed things up, we end up with poor performance. I think this is because of too much contention between the threads.

The second part of this homework shows how doing the same computation can be speed up or slowed down depending on the threading that is going on behind the scenes.