Homework 3, Part 1

Cooper Faber

May 2021

1 Results

1.1 Table

Threads	Static	Global	Stealing	Stealing32
8	2987s	9922s	3407s	2978s
0	10324s	10337s	10372s	12280s

1.2 Graph

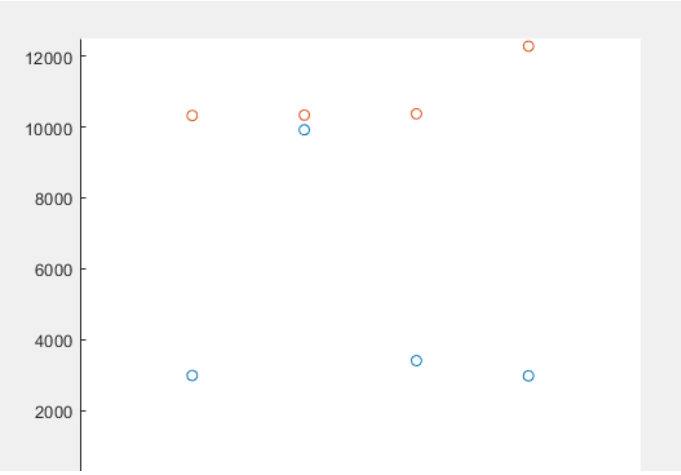

Figure 1: Results

1.3 Context

Results in the graph above are symmetrical with the table, i.e Static is first, Global is second, and so on. Blue dots are with 8 threads, and orange dots are with 1.

2 Implementation

My implementation of the static and global queue were, again, almost entirely done via the pseudocode given in lecture, and there's nothing special I added. For static (and stealing/stealing32) chunks are assigned via even chunk_sizes, which is an even division of size by num_threads. The stealing queue, however, leaves us to decide how to find a target on our own, so I chose a for loop that iterates from 0 to NUM_THREADS, checking each queue on the way up to see if there's any work left to do. My stealing32 finds a target in a similar way, however, unlike stealing, stealing32 attempts to dequeue 32 items, and moves forward if it cannot. The key here, obviously, is that SIZE is divisible by 32; this would be a more difficult optimization to perform otherwise - and going back, the chunk_size operation only works as well as it does because of that assumption. (Although that would be an easier fix).

3 Analysis

Going in, I didn't expect Static to be so fast - but, given the above assumptions of easy divisibility, it makes sense. One thing I was surprised by was the additional overhead of the atomic operations in Global/Stealing – Global should, theoretically, be passing out computations to threads at about the same pace via the global counter. However, apparently that single change is enough to decrease the speed by over 3x! On the same note, given our computations with widely variable computation time, Stealing should also have an edge over Static, but the required atomic operations significantly increase the overhead as well, to the point that even with optimization, Stealing32 barely passes Static. This, however, may also have something to do with my implementation - the times I received running it varied heavily by machine (local desktop had stealing32 at 1.4x slower than static; reported values are from my newer laptop). A data race is not involved, as output values on both machines were consistent, and Thread Sanitizer reported no errors.

Moving on to single-threaded results, they were consistent across the board, with the exemption of Stealing32. My assumption here is that the batched dequeues manage to slow down the performance (as that is the only difference between Stealing32 and Stealing) in a similar way that they improve the performance in the multi-threaded version – however, the decrease in relative atomic operations that provide the initial speedup should still apply here.