Homework 2, Part 1

Cooper Faber

May 2021

1 Experiments

I have 4 cores on my vaguely elderly machine, so tests were done at 4 threads and 128 threads.

1.1 Throughputs

Threads	CPP_MUTEX	FILTER_MUTEX	BAKERY_MUTEX
4	5937243	3352360	4609691
128	5133923	353	2865

1.2 Variances

Variance was calculated via coefficient of variation, from Wikipedia and Piazza. (Hopefully those hyperlinks work!)

Threads	CPP_MUTEX	FILTER_MUTEX	BAKERY_MUTEX
4	2.97%	0.23%	0.8%
128	14.49779%	41.8898%	373.73525%

2 Yield

2.1 Throughputs

Threads	CPP_MUTEX	FILTER_MUTEX	BAKERY_MUTEX
4	N/A	1689443	1723257
128	N/A	7438	31756

2.2 Variances

Threads	CPP_MUTEX	FILTER_MUTEX	BAKERY_MUTEX
4	N/A	0.28196%	0.31863%
128	N/A	19.33294%	10.80153%

3 Graphs

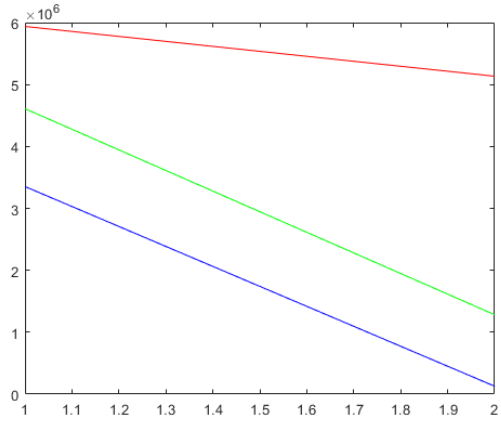

Figure 1: Throughput

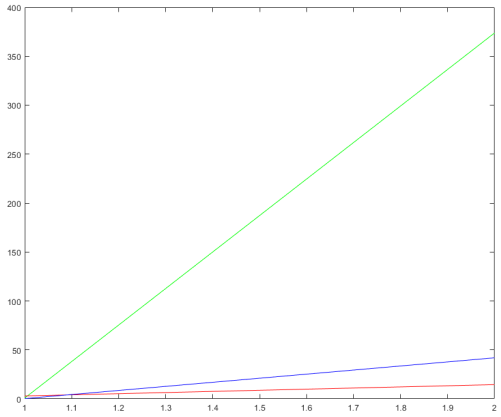

Figure 2: Variance

In the above figures, the red line is CPP, the green line is BAKERY, and the blue line is FILTER. Although the x-axis goes from 1 to 2, 1 represents 4 threads, whereas 2 represents 128. As the only data entered was for 4 threads and 128 threads, these actual lines are a rough (likely incorrect, as will be expanded upon) estimate.

4 Analysis

So, as seen, the filter and bakery mutexes perform well (although not as well as cpp_mutex) up until the amount of threads exceeds the amount of cores. The graphs above indicate a regular, gradual drop after passing 4 threads. This, in practice, is false. After passing 4 threads both filter and bakery drop significantly in throughput and variance, and continue to fall from there. There are different reasons for each algorithm, so let's start with Filter.

Filter works on guaranteeing a single "level" for each thread, such that every thread takes up a level up to the critical section while waiting. It's fairly evident that adding threads would decrease throughput, as each additional thread creates an additional level, which is another check that new threads must pass. This creates large overhead, and at high levels of threads, brutalizes the throughput. Variation also jumps quite a bit as threads increase, and that is likely due to the first few threads making it through significantly more than the later ones, which make it through very very few times, although this is mitigated somewhat by yield().

Bakery, on the other hand, is a fair mutex. However, the OS does enjoy playing favorites, which is why there is such a massive jump in variation. Regarding throughput first, though - as the amount of threads increase, more time is spent assigning labels and checking other labels, slowing down the entire process. After all, it's easier for a thread to check 3 other labels than 127. It's worth noting that although bakery slows down, it does not slow down to the speed of filter as threads increase, showing that it does still have a capability of handling a load. However, it doesn't have the capability of fairly balancing that load, as shown in the variance chart. When running bakery on >4 threads, the first 4 threads hog a majority of the time in the critical section, leaving later threads with measly portions of time. As I stated above, I believe the OS is responsible for this. To quoth the midterm: "It is possible that one of the threads, say t0, continually acquires the mutex, especially if the OS schedules t0 on the core instead of t1." Following this logic, it's likely that the threads scheduled to the cores will be more aggressive in obtaining the lock quickly, which will result in a faster time to the critical section. Adding yield() to the bakery lock will decrease variance, allowing threads further down the line more access to the lock. However, this is also a result of lowering throughput. As less threads can make it in within the time limit, the first 4 greedy threads will also make it in less, lowering the variance coefficient.