CSC 258 Project 3

Brandon Willett

April 1, 2018

Testing

All of these tests were run on `node2x14a`, since both of the other machines (checked with `ps`) had other class members' projects running and I wanted the results to be accurate. Like with the last project, each table contains number of threads on the y-axis and the testing sample on the x-axis. Each individual box contains two numbers, being the mean on the top and the standard deviation on the bottom (being over 5 trials). All times are in milliseconds.

Baseline (`histogram.cpp`)

Threads	**Hawkes**	**Earth**	**Flood**	**Noise**	**Phobos**	**Bear**	**Univ**
1	304.88 29.41	7616.09 148.93	374.44 5.98	1626.82 342.67	1398.10 20.60	1714.89 14.94	259.44 61.74

Exercise 1 (`histo-private.cpp`)

Threads	**Hawkes**	**Earth**	**Flood**	**Noise**	**Phobos**	**Bear**	**Univ**
1	429.43 14.20	14383 440.22	989.12 89.45	3642.26 10.92	3456.41 208.00	4192.49 317.16	700.45 13.43
2	1864.38 53.07	57044 5105.50	5499.71 448.22	25792 10284	17851 813.50	27073 1873.22	3467.33 383.90
4	3218.95 507.89	161842 14175	9820.26 1912.93	37836 5176.00	39225 8942.85	51007 14926	8743.43 1103.66
8	827.40 532.29	62360 13483	5162.08 682.93	15083 2811.90	17970 2035.45	17436 3111.69	1720.23 734.32

Exercise 2 (`histo-lockfree.cpp`)

Threads	Hawkes	Earth	Flood	Noise	Phobos	Bear	Univ
1	3191.49 329.76	87799 843.98	9403.61 2291.17	22904 461.11	21152 741.45	22386 1972.13	6473.95 486.52
2	3686.01 394.44	183538 21603	10856 1584.86	45021 12964	35569 1510.62	60564 4837.00	7119.77 247.13
4	4895.70 111.33	237139 15271	11400 437.93	38268 3885.41	48058 987.94	63844 2531.51	8388.58 149.81
8	5012.28 249.57	346773 24938	11087 146.89	27841 528.15	56883 2803.14	74258 11713	10700 296.57

Exercise 3 (`histo-lock1.cpp` and `histo-lock2.cpp`)

Note, in this exercise, `histo-lock1.cpp` uses a test-and-test-and-set lock implementation, and `histo-lock2.cpp` instead uses a simple ticket lock implementation. The book had written the ticket lock to have a small backoff when your ticket wasn't up, but in my tests that made performance worse across the board, so I commented that line out.

Threads	Hawkes	Earth	Flood	Noise	Phobos	Bear	Univ
1	6078.44 1455.92	288094 17587	19168 2138.81	58554 3458.94	52463 2409.19	57171 2573.38	11865 929.81
2	15979 909.13	961902 62948	45567 11176	139601 6245.23	118458 6609.32	127744 4368.74	25625 1312.17
4	34828 1416.68	2095337 278540	105301 1440.96	486225 11942	351477 27567	486927 15637	63407 2584.53
8	58346 1177.31	3703855 104262	185570 6941.64	775611 39178	663416 23926	804052 38491	119859 3457.35

Threads	Hawkes	Earth	Flood	Noise	Phobos	Bear	Univ
1	6152.37 1804.19	276564 280.11	17502 1592.61	56458 1739.23	50524 1500.47	57847 2230.25	12793 1627.33
2	16438 1542.63	1121514 80434	40007 852.65	139705 3792.12	121271 7770.04	237606 69103	26809 2052.15
4	38289 1219.87	2412147 110940	117930 4990.26	469225 8861.71	412130 30410	504072 32148	72978 3054.14
8	43231 1522.97	2841143 108619	137195 5156.76	556923 39609	462191 30677	555932 45609	81159 1868.90

Results

The results were surprisingly bad! Not a single one of the multithreaded versions of the program actually outperformed the serial version. This confused me for a while, but I've decided now that it shouldn't come as much of a surprise – if the task were heavily CPU bound, with only a little bit of reading/writing to shared data, we'd see great improvement in the threading. However, each loop iteration in this task just does two near-instant arithmetic operations, and then writes to shared data, that's it. Since nearly all of each iteration's time is spent writing to shared data, it makes sense that there'd be little to no speedup to be found in parallelizing the task. And in fact, that's what we see here – the overhead gotten by using threads and (even worse) locks for the shared data severely overcomes any benefit we'd get from sharing the "work", since there's actually barely any work to be done other than writing data.

By that logic, the version of the program that performs the best will be the one that writes to shared data the least (whether via atomics or locks), and that's indeed the case, as `histo_private` does significantly better than all the other versions, though still not matching the serial version (threads must be expensive)!

Note

One sad thing is that the `histo_private` code doesn't actually produce perfectly correct results when run on more than one thread. Looking at the diff, it seems that a few bins are off by a small amount (± 10), which indicates that there's an off by one error or something somewhere. This is weird to me, since the code is very straightforward and I spent hours looking through it with no luck, it looks perfect to me (but maybe that's just because I don't have much experience with C++). The point is, since this program is clearly doing all the work in the right sort of way, I ask that you don't take off too many points for the slight incorrectness of it, and if you do, please at least let me know what's causing it – because I can't find the problem for the life of me; all the other versions use nearly identical code and work fine. Thank you