Assignment 3 Report: Posix threads

Serena kobeissi

CMPS 270

My experimental setup for this assignment: MacBook air m1 chip : 8-core GPU

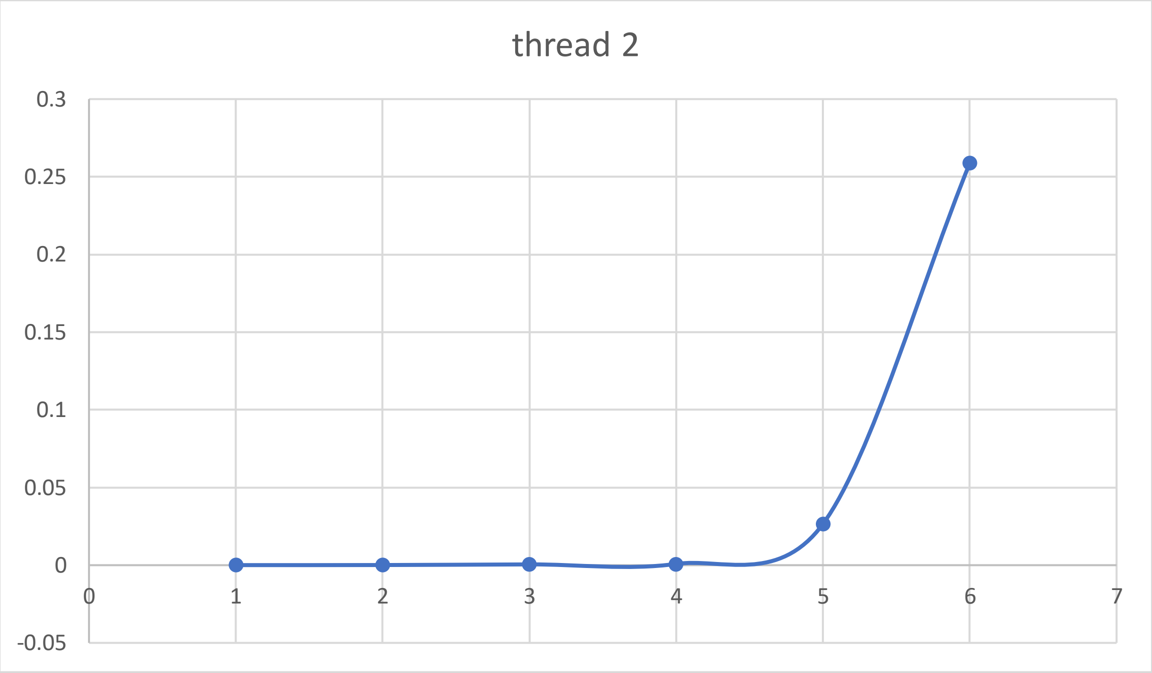
When implementing the counting one parallel program and testing it on different array sizes and different number of threads, I realized that the time my algorithm was taking was increasing and not decreasing as it should. After some research and after communicating with my TA, I understood that on my 8-core chip there is a threshold that my setup can handle when it comes to number of threads and seeing improvement. So, increasing the number of threads doesn't always mean decrease in runtime.
Moreover, my code wouldn’t run on a 1 000 000 000 size array and that could be due to many internal reasons.

In my algorithm, my main function included a for loop in which both sequential and parallel codes showed the time taken by each. This was used to show the difference between the two. This table shows the time taken by the parallel code.

Count1.c:

Threads	Array Dimension						
	100	10 000	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
	Time	Time	Time	Time	Time	Time	Time
1	0.000047	0.000096	0.000516	0.000843	0.026551	0.258804	Not compiling
2	0.000069	0.000094	0.000354	0.001595	0.013542	0.009292	Not compiling
4	0.0001258	0.000163	0.000352	0.001315	0.007546	0.072987	Not compiling
8	0.000222	0.0002201	0.0002501	0.001220	0.0087791	0.089402	Not compiling
16	0.000232	0.000260	0.000311	0.001034	0.009257	0.089679	Not compiling
32	0.0005836	0.000567	0.000439	0.001255	0.0093371	0.090661	Not compiling
64	0.000995	0.000593	0.000939	0.001354	0.009225	0.089418	Not compiling

This graph shows the increase in time for different array sizes using one thread. Here 1, 2,3,4,5 etc.. represent 100,10000,100000 etc…



To fix the race condition problem, I implemented a code using mutex.
These were the statistics.

Count_mutex.c:

Threads	Array Dimension						
	100	10 000	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
	Time	Time	Time	Time	Time	Time	Time
1	0.000050	0.000068	0.000063	0.000059	0.000037	0.000036	Not compiling
2	0.000072	0.000115	0.000051	0.000046	0.000065	0.000079	Not compiling
4	0.000142	0.0001738	0.000255	0.000133	0.000094	0.000089	Not compiling
8	0.000134	0.000173	0.000291	0.000114	0.000105	0.000126	Not compiling
16	0.000342	0.000229	0.000380	0.000285	0.000192	0.000218	Not compiling
32	0.000342	0.000606	0.000415	0.000525	0.000442	0.000318	Not compiling

64	0.000858	0.000945	0.000211	0.000594	0.000632	0.000783	Not compiling

As we can see, when using the mutex method, the runtime when I add threads is increasing but in small amounts. In my case, I did not see a lot of improvements since in the implementations without mutex the runtime also increased. If you have many threads and the access to the object happens often, then multiple locks would increase parallelism. At the cost of maintainability, since more locking means more debugging of the locking.

count_private.c:

In the third code, the count was not a global variable, but it was local to each thread. We added all the counts in the main. We can see that in this experiment, there is a decrease in runtime when adding threads.

Threads	Array Dimension						
	100	10 000	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
	Time	Time	Time	Time	Time	Time	Time
1	0.027144	0.000142	0.000554	0.002982	0.026506	0.263777	Not compiling
2	0.000242	0.000147	0.000391	0.001818	0.026284	0.135070	Not compiling
4	0.000168	0.000122	0.000266	0.000873	0.008927	0.069413	Not compiling
8	0.000220	0.000228	0.000176	0.001008	0.007874	0.053223	Not compiling
16	0.000307	0.000489	0.000294	0.001459	0.007448	0.042332	Not compiling
32	0.000408	0.000646	0.000348	0.000864	0.007252	0.041111	Not compiling
64	0.000838	0.000772	0.000623	0.001290	0.005373	0.038921	Not compiling