Collin Paiz

Professor Anastasiu

COEN 145

18 October 2022

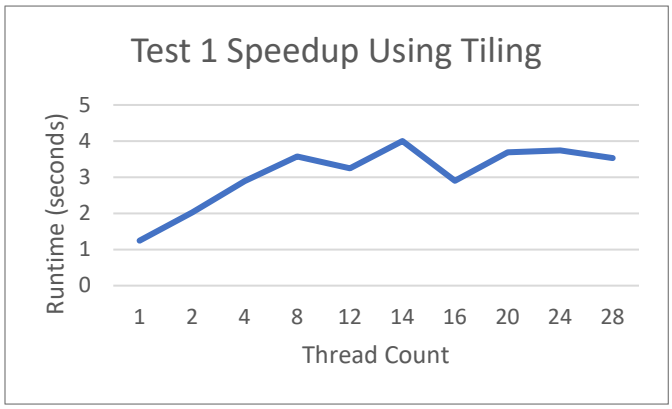
Assignment 1

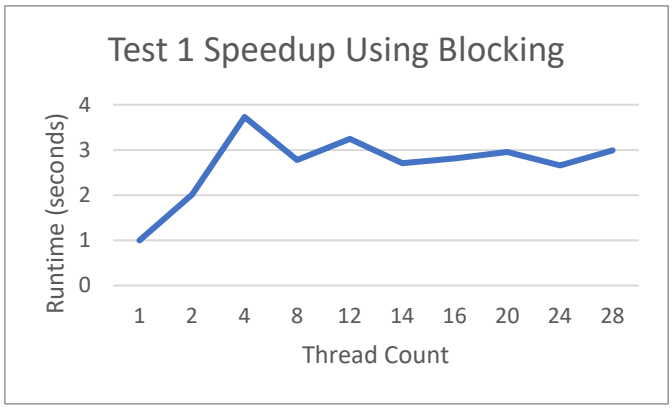
Parallelizing Dense Matrix Multiplication

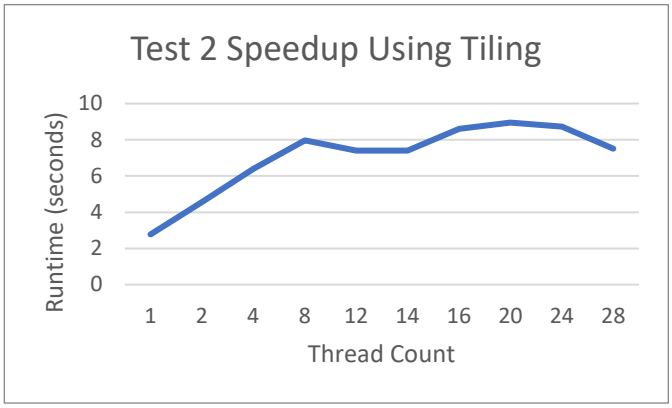
Description

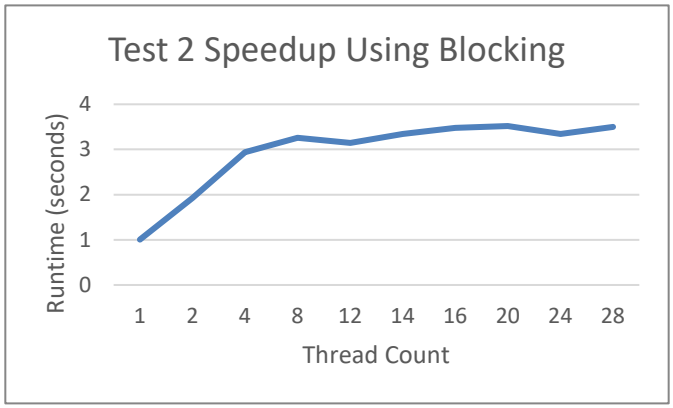
To parallelize my code, I first put a "#pragma omp parallel for" outside the outermost for-loop; this will run the execution of each tile or block in parallel with each other. I used "shared(C)" to share the result matrix between threads and "private(*loop iteration variables,* sum)" so that each thread has its own version of the variables necessary for iterating through its own tile or block, as well as its own sum variable for keeping track of the current computation value. I chose to parallelize my code this way because it makes sense for each thread to have its own designated block or tile. If each thread is given its own block or tile, then individual sections of the overall result matrix can be computed at the same time. Within each block or tile, its computations are performed by that parallel thread that was given that job.

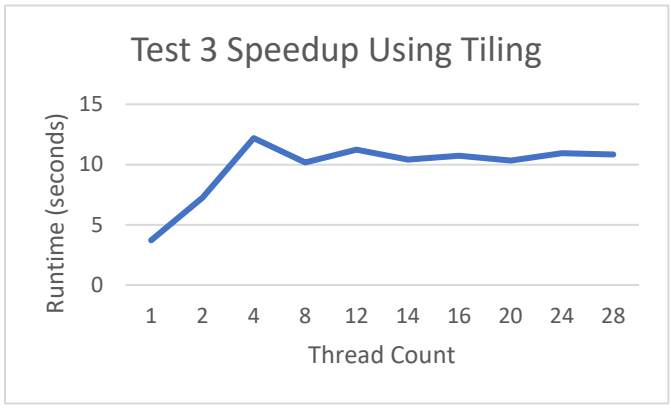
Results

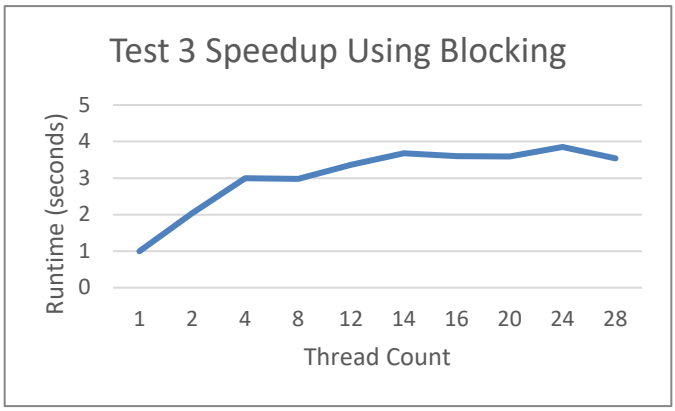












Raw Data

Serial

Test 1 Runtime (s)	Test 2 Runtime (s)	Test 3 Runtime (s)
1.23972	27.5961	318.315

Block

Thread Count	Block Size	Test 1 Runtime (s)	Test 2 Runtime (s)	Test 3 Runtime (s)
1	50	1.24394	27.4639	319.163
2	50	0.614987	14.3145	155.759

4	50	0.33236	9.37694	106.345
8	50	0.446025	8.47776	106.732
12	50	0.381842	8.76558	94.7139
14	50	0.457406	8.24649	86.6108
16	50	0.440261	7.93321	88.4655
20	50	0.419262	7.84223	88.5926
24	50	0.465945c	8.25204	82.6547
28	50	0.465945	7.89341	89.8752

Tiling

Thread Count	Block Size	Test 1 Runtime (s)	Test 2 Runtime (s)	Test 3 Runtime (s)
1	200	0.995612	9.95141	85.7866
2	200	0.612121	6.05035	43.8469
4	200	0.428677	4.31201	26.1027
8	200	0.346749	3.46524	31.2666
12	200	0.38221	3.72461	28.3285
14	200	0.309762	3.72781	30.5909
16	200	0.42768	3.20914	29.6415
20	200	0.336278	3.08389	30.7774
24	200	0.331138	3.16034	29.1098
28	200	0.350998	3.67482	29.3568

Analysis

As the results show (above), as the number of threads increases, in general, the speedup also increases (the runtime decreases). This is due to multiple threads computing parts of the result matrix at the same time, rather than one thread performing each computation one after another (as displayed by the serial run or original serial code provided). The more threads we have running at the same time, the more computations we can perform at once, thus reducing the runtime of the program. That being said, there are diminishing returns for some of the test runs, meaning as we added more threads, the runtime did not improve or at least not by a significant amount. For example, when using tiling for test 2, speedup over the serial execution peaked at 20 threads and then trended downwards when adding more threads. This indicates that using more than 20 threads for this test is not beneficial to the program's performance. In general, for each test run we reached a thread count where the speedup did not see any noticeable improvements and any deviations were likely within the margin of error.

As the size of the problem increases, so does the runtime. This is likely because of the larger number of computations that are required when the matrices are larger. As the dimensions of the matrices scale up linearly, the number of computations required to multiply them together scales up exponentially.

In general, as we increase the size of the program (resulting in a longer runtime), increasing the number of threads helps offset this increasing runtime. But as we can see with many of the tests, there are diminishing returns. Another example of this can be seen while using tiling in test 3. The greatest speedup was achieved when using four threads; utilizing more threads beyond four even slightly hurt performance. So while theoretically adding more threads should improve speedup, experimentally we can see that there is a limit on how much those extra threads can actually help.

Performance characteristics seemed to differ based on the shapes of the matrices. This is likely due to the number of iterations for each for-loop. If we have two square matrices like in test 1, then depending on our tile size or block size, we can properly parallelize each block or tile a number of times. But if we were to use the block strategy for a super long matrix of the same size (very low row count, high column count), for example, even though it has the same number of entries, it would be more difficult to parallelize because it wouldn't be easy to split up work between threads efficiently.