Computer Science 133, Lab 1

Michael Wu
UID: 404751542

January 24th, 2019

1 Speedup Summary

The following table summarizes the results of my implementation.

Problem Size	1024^3	2048^3	4096^3
Sequential Speed (GFlops)	0.595169	0.283576	0.195333
Parallel Speed (GFlops)	19.3156	19.4528	17.8648
Parallel Speedup	32.45x	68.60x	91.46x
Parallel-Blocked Speed (GFlops)	15.3914	17.6912	21.0508
Parallel-Blocked Speedup	25.86x	62.39x	107.77x

The speedup differences are mainly due to the sequential version performing worse as the problem size increases, since there are more cache misses in the sequential version. The non-blocked parallel implementation began to drop off slightly as the problem size increased due to cache misses as well. The blocked version seemed to increase in performance as the problem size increased, since blocking works better when used on a larger array.

2 Parallel Blocked Optimizations

There seemed to be a wide range of block size values that could lead to good performance, as long as they were not too large or too small. The following table summarizes the performance of my blocked parallel implementation in relation to block size on the 4096^3 problem size. I chose to use a block size of 256 since it led to good performance in all three problem sizes of 1024^3, 2048^3, and 4096^3.

Block Size	Speed (GFlops)
32	16.6554
256	21.0924
512	20.5402
1024	21.0501
2048	18.2274

Additional optimizations I performed were loop unrolling and transposing the b matrix so that the innermost loop traversed the rows of b instead of the columns. Loop unrolling gave me approximately twice the speed, and transposing the matrix gave me approximately ten times the speed. Note that loop unrolling led to small rounding errors on the order of 10^{-7} as the array size increased, as the floating point addition might have been performed in a different order.

3 Parallel Blocked Thread Performance

The `m5.2xlarge` instance supports up to 8 threads, since there are 8 cores on the machine. The following summarizes the scalability of my blocked implementation with different thread numbers on the 4096^3 problem size.

Threads	Speed (GFlops)
1	4.70493
2	9.71357
4	19.1020
8	21.0384

4 Results

In my implementation, it seems that the blocked parallel implementation outscales the regular parallel implementation as the array size increases. For small arrays it is faster to use the regular parallel implementation. Note that as array size increases, the speedup over the sequential version increases as well because of cache efficiency. Some of this has no relation to parallelism, because even with a single thread the optimizations can lead to a speed of 4.70493 GFlops in the blocked implementation. With additional threads, the blocked implementation can become over five times faster.

5 Challenges

Two main challenges I faced were related to loop unrolling and testing my implementations. Loop unrolling caused small rounding errors that led me to believe my implementation was incorrect. I only was able to fix this after reading on Piazza that the TAs would raise the error threshold to the order of 10^{-5}, so these small rounding errors would be acceptable. Testing my implementation was also a challenge because for most problem sizes the parallel implementation performed better than the blocked parallel implementation. I was considering submitting my parallel implementation since I was confused on which problem size my implementation would be tested on. There were so many tuning parameters that I wasted a lot of time fiddling with. In the end I chose to optimize for the 4096^3 problem size, as that is what the TAs gave their benchmarks for.