Yash Shah
405-565-567

Lab 1 Report

Sequential v. Parallel v. Parallel Blocked

Problem size: 1024 x 1024 x 1024

	Sequential	Parallel	Parallel Blocked
Time (s)	3.52563	0.031207	0.03953
Perf (GFlops)	0.609107	68.8142	54.3254

Speedup (Parallel vs. Sequential) = 3.52563 / 0.031207 = ~113.0x
Speedup (Parallel Blocked vs. Sequential) = 3.52563 / 0.03953 = ~89.1x

Problem size: 2048 x 2048 x 2048

	Sequential	Parallel	Parallel Blocked
Time (s)	60.9346	0.245426	0.265439
Perf (GFlops)	0.28194	70.0002	64.7225

Speedup (Parallel vs. Sequential) = 60.9346 / 0.245426 = ~248.1 times
Speedup (Parallel Blocked vs. Sequential) = 60.9346 / 0.265439 = ~229.5 times

Problem size: 4096 x 4096 x 4096

	Sequential	Parallel	Parallel Blocked
Time (s)	731.531	3.46415	2.08698
Perf (GFlops)	0.187879	68.8142	54.3254

Speedup (Parallel vs. Sequential) = 731.531 / 3.46415 = ~211.2 times
Speedup (Parallel Blocked vs. Sequential) = 731.531 / 2.08698 = ~350.3 times

Observations:

The speedup drastically improves as the problem size increases from 1024^3 to 4096^3. This is the case because as problem sizes grow, the number of independent operations that can be performed concurrently also grow. Additionally, the relative overhead of parallelism becomes almost negligible, adding almost nothing to computation size as the problem size grows.

For a parallel-blocked solution, a smaller problem size actually performs worse than sequentially. Only when data locality and efficient use of cache is required does a parallel-blocked solution perform better than a sequential or parallel solution. Henceforth, as the problem size grows, a parallel blocked solution yields better performance. Both parallel and parallel blocked solutions

Impact of optimizations in omp-blocked.cpp

No optimizations

Time: 13.9954 s
Perf: 9.82029 GFlops

With only zero-ing out matrix c

Time: 13.9104 s
Perf: 9.88031 GFlops

The time of execution decreases by .08s and GFlops increase by 0.06, with this added optimization.

With zero-ing out matrix c & with pragma omp

Run blocked parallel GEMM with OpenMP
Time: 2.17538 s
Perf: 65.1793 GFlops

The time of execution decreases by 11.74s and GFlops increase by 55.3, with these optimizations.

Scalability

1 thread

Time: 14.3174 s
Perf: 9.59943 GFlops

2 threads

Time: 6.22934 s
Perf: 22.0632 GFlops

3 threads

Run blocked parallel GEMM with OpenMP
Time: 4.09945 s
Perf: 33.5262 GFlops

4 threads

Run blocked parallel GEMM with OpenMP
Time: 2.5017 s
Perf: 54.9382 GFlops

Discussion

From 1 to 2 threads, the time taken is approximately halved, and the performance more than doubles. Similarly, as we increase the number of threads from 2 to 4, we see a consistent drop in time taken and a boost in performance. Given this, the scalability from 1 to 4 threads seems near-linear.

Given that 4 threads' performance is less than the optimal performance we saw before, we can infer that AWS m5.2xlarge uses greater than 4 threads. After researching, it is noticed that 8 threads are used in the m5.2xlarge server, yet the increase in performance decreases, henceforth after 4 threads, the increase in threads do not increase linearly as found between 1-4 threads.

Discussion of results and challenges

The results were pleasing, but I wasn't able to get to the A+ range most likely due to the fact that I couldn't figure out loop unrolling and other optimizations in enough time. Luckily, I had optimized the ordering of the loops well enough, in addition to parallelizing the matrix multiplication segment, which was able to yield results sufficient enough for submission.

Some notable challenges for me was dealing with the race condition in omp-blocked when attempting to solve the matrix multiplication. I kept running into issues where I had shared resources that were being hit at the same time, resulting in my answer being incorrect. Eventually, one of the campuswire posts displayed the algorithm that I eventually ended up using (in using 3 blocks, 2 for fast access, and 1 for slow access) to resolve this issue and solve the problem. After that, optimizing became easier, as I was able to change the loop orientations and add the pragma omp statement to parallelize the actual matrix multiplication segment of the code.