Questions

Norm

- Question 1: Look through the code for `run()` in `norm_utils.hpp`. How are we setting the number of threads for OpenMP to use?

Answer: In the `run()` function, we have a for loop which defines the number of threads we will iterate. Also, there is a if statement to check if _OPENMP is defined. If _OPENMP is defined, that means we are using OpenMP, then the program will call omp_set_num_threads(nthreads) to set the number of threads for OpenMP to use.

- Question 2: Which version of `norm` provides the best parallel performance? How do the results compare to the parallelized versions of `norm` from ps5?

Answer: Overall, my norm_block_reduction gives me the best parallel performance for most problem sizes and number of threads. To be more detailed, compared with the sequential performance, my norm_block_reduction with 1 thread is about 0.99 times speedup in the performance; with 2 threads is about 1.93 times speedup in the performance; with 4 threads is about 3.48 times speedup in the performance; with 8 threads it can even reach about 8.02 times speedup in the performance with N = 1048576 (sequential reading is 2.61191 GFlops/s, my reading is 20.9497 GFlops/s.). With 8 threads, on average it's about 4.07 times speedup in the performance compared with the sequential reading for my norm_block_reduction.

Compared with the several parallelized versions of `norm` from ps5: for ps5, my pnorm and fnorm both reaches about 2 times speedup in the performance for 8 threads. For my norm_block_reduction with 8 threads, on average we reach about 4 times speedup in the performance (with peak of about 8 times speedup). So, with OpenMP, we are getting some significant performance improvement especially when we are using more threads. For cnorm from ps5, we ended up getting lower performance compared with the sequential performance, which also holds here when we are using OpenMP. For my norm_cyclic_critical, I am getting lower performance even for multi-threads. For my norm_cyclic_reduction, I am getting some performance speedup but not that significant compared with my other functions. For rnorm from ps5, I am from Amath 483 so I did not do that one which I cannot make a comparsion.

- Question 3: Which version of `norm` provides the best parallel performance for larger problems (i.e., problems at the top end of the default sizes in the drivers or larger)? How do the results compare to the parallelized versions of `norm` from ps5?

Answer: For N = 33554432 my norm_payfor has a slighly better performance compared with norm_block_reduction. For norm_block_reduction, with 1 threads is about 0.99 times speedup in the performance; with 2 threads is about 1.93 times speedup in the performance; with 4 threads is about 3.48 times speedup in the performance; with 8 threads is about 4.07 times speedup in the performance. For norm_payfor, with 1 threads is about 0.99 times speedup in the performance; with 2 threads is about 1.95 times speedup in the performance; with 4 threads is about 3.54 times speedup in the performance; with 8 threads is about 4.13 times speedup in the performance. So, we see that my norm_payfor provides the best parallel performance foor larger problems.

Compared with ps5: For my pnorm and fnorm with N = 33554432, my performance for both are very close. For 1 thread, it's about 0.96 times speedup; for 2 threads, it's about 1.61 times speedup; for 4 threads, it's about 1.94 times speedup; for 8 threads, it's also about 1.97 times speedup. We see that compared with ps5 versions of `norm`, my norm_payfor with OpenMP gets a better performance improvement for multi-threads (e.g. 2, 4, 8 threads). Since cnorm gets lower performance, there is not much meaning to make a comparsion in here because the point of those assignnments are to find methods that can improve the performance, not to lower it. Again, I am from Amath 483 so I did not do rnorm.

- Question 4: Which version of `norm` provides the best parallel performance for small problems (i.e., problems smller than the low end of the default sizes in the drivers)? How do the results compare to the parallelized versions of `norm` from ps5?

Answer: For N = 1048576 and lower, my norm_block_reduction provides the best parallel performance for small problems. To be more detailed, with 1 thread is about 0.98 times speedup; with 2 threads is about 1.83 times speedup; with 4 threads is about 3.53 times speedup; with 8 threads is about 8.02 times speedup.

Compared to ps5: For my pnorm and fnorm with N = 1048576 and lower, the performance improvement are very similar for the two versions. With 1 thread is about 0.64 times speedup; with 2 threads is about 1.1 times speedup; with 4 threads is about 1.52 times speedup; with 8 threads is about 0.89 times speedup. We see that compared with ps5 versions of `norm`, my norm_block_reduction with OpenMP gets a significant performance improvement for 1, 2, 4 and 8 threads. Since cnorm gets lower performance, there is not much meaning to make a comparsion in here because the point of those assignnments are to find methods that can improve the performance, not to lower it. Again, I am from Amath 483 so I did not do rnorm.

Sparse Matrix-Vector Product

- Question 5: How does `pmatvec.cpp` set the number of OpenMP threads to use?

Answer: In the main function, we have a for loop which defines the number of threads we will iterate. Also, there is a if statement to check if _OPENMP is defined. If _OPENMP is defined, that means we are using OpenMP, then the program will call omp_set_num_threads(nthreads) to set the number of threads for OpenMP to use.

- Question 6: (For discussion on Piazza.)

What characteristics of a matrix would make it more or less likely to exhibit an error if improperly parallelized? Meaning, if, say, you parallelized `CSCMatrix::matvec` with just basic columnwise partitioning -- there would be potential races with the same locations in y being read and written by multiple threads. But what characteristics of the matrix give rise to that kind of problem? Are there ways to maybe work around / fix that if we knew some things in advance about the (sparse) matrix?

- Question 7: Which methods did you parallelize? What directives did you use? How much parallel speedup did you see for 1, 2, 4, and 8 threads?

Answer: I parallelized two functions in CSRMatrix.hpp: matvec() and t_matvec(); and two functions in CSCMatrix.hpp: matvec() and t_matvec(). For the four functions, I have used the OpenMP directive "#pragma omp parallel for schedule(static)". For CSR matvec() and CSC t_matvec() I have placed this directive on the outer for loop, and for CSR t_matvec() and CSC matvec() I placed this direction on the inner for loop. I successfully passed all of the tests provided in here and in Gradescope. However, when I ran ./pmatvec.exe, the execution time was very long, and I only got performance improvement for CSR matvec() and CSC t_matvec(). My CSR t_matvec() and CSC matvec() have very low performances. So I commented out the OpenMP directive for my CSR t_matvec() and CSC matvec() and ran pmatvec.exe again but the result did not show much difference. My submission of the code portion to Gradescope contains the commented code, I just included it there for grading. Also, for CSR matvec() and CSC t_matvec(), we can also parallelize it with the directive "#pragma omp parallel for reduction(+:temp)" where the parameter "temp" is a double that I hoist for y(i), I also left this directive commented in the code just to show you for grading. I cannot parallelize my CSR t_matvec() and CSC matvec() with this directive since we cannot hoist the y variable out for those two functions. The two different approach give me very similar performance improvement.

To be detailed, for my CSR matvec() with performance improvement (compared with COO): with 1 thread is about 1.35 times speedup on average (peak is 1.41 times speedup at N = 128); with 2 threads is about 2.13 times speedup on average (peak is 2.69 times speedup at N = 128); with 4 threads is about 3.24 times speedup on average (peak is 5.46 times speedup at N = 256); with 8 threads is about 4.38 times speedup on average (peak is 9.06 times speedup at N = 256).

To be detailed, for my CSC t_matvec() with performance improvement (compared with COO^T): with 1 thread is about 1.36 times speedup on average (peak is 1.42 times speedup at N = 2048); with 2 threads is about 2.54 times speedup on average (peak is 3.17 times speedup at N = 256); with 4 threads is about

2.96 times speedup on average (peak is 3.28 times speedup at N = 128); with 8 threads is about 3.93 times speedup on average (peak is 7.47 times speedup at N = 256).

Sparse Matrix Dense Matrix Product (AMATH583 Only)

- Question 8: Which methods did you parallelize? What directives did you use? How much parallel speedup did you see for 1, 2, 4, and 8 threads? How does the parallel speedup compare to sparse matrix by vector product?

Answer: I am from Amath 483.

PageRank Reprise

- Question 9: Describe any changes you made to pagerank.cpp to get parallel speedup. How much parallel speedup did you get for 1, 2, 4, and 8 threads?

Answer: I changed the "CSRMatrix A = read_csrmatrix(input_file)" in pagerank.cpp into "CSCMatrix A = read_cscmatrix(input_file)" to let it use CSC matrix instead. To be detailed: with 1 thread is about 0.97 times speedup in the performance; with 2 threads is about 1.08 times speedup in the performance; with 4 threads is about 1.13 times speedup in the performance; with 8 threads is about 1.15 times speedup in the performance. We see that overall the performance improvement with our change from CSR matrix to CSC matrix was not very large which is reasonable since we have parallelized both our CSRMatrix.hpp and CSCMatrix.hpp with OpenMP for To-Do 1 and Question 7. In question 7, we see that the two parallelized functions for CSRMatrix.hpp and CSCMatrix.hpp achieved a very similar performance improvement, so the result we got in here is rational.

- Question 10: (EC) Which functions did you parallelize? How much additional speedup did you achieve?

Answer: In the last two for loop in pagerank.cpp, I added the OpenMP directive "#pragma omp parallel for schedule(static)" on them. Unfortunately, there is not much of difference by adding this directive to the two for loops. I did not get much additional speeedup.

Load Balanced Partitioning with OpenMP

- Question 11: What scheduling options did you experiment with? Are there any choices for scheduling that make an improvement in the parallel performance (most importantly, scalability) of pagerank?

Answer: I experiment with the options: static, dynamic, auto, and guide. Among those choices, I think the kind static works the best for the "schedule" clause. Since in our case, we want each thread does the same amount of work without interrupting each other (avoiding race conditions), so dynamic and guide is not the right choice because the work for each thread can be randomized. For auto, even that it can divide the problem into equal sized, it lacks the ability to define the chunk-size like the static does. With static and auto sometimes from my end, we are getting some performance improvement. I just want to point out that for the extra credit question, I also used the schedule clause with the static kind to improve performance in pagerank.cpp.

OpenMP SIMD

- Question 12: Which function did you vectorize with OpenMP? How much speedup were you able to obtain over the non-vectorized (sequential) version?

Answer: I vectorized my norm_block_reduction() function with OpenMP by commenting my previous OpenMP directive "#pragma omp parallel reduction (+:sum)", and adding the new OpenMP directive "#pragma omp simd" right before the for loop. I also tried this with my norm_cyclic_reduction(), but I only got performance improvement with my norm_block_reduction() function.

For my vectorized version of norm_block_reduction() function, compared with the sequential performance: with 1 thread is about 1.94 times speedup in the performance; with 2 threads is about 1.95 times speedup in the performance; with 4 threads is about 1.95 times speedup in the performance; with 8 threads is about 1.96 times speedup in the performance. We can see that overall, disregard how many thread we used, we got about 2 times speedup in the performance after vectorizing the norm_block_reduction() function with OpenMP. Compared to our previous OpenMP directive with 4 or 8 threads, this method does not over perform. But, a 2 times speedup is still a significant improvement in the performance.

To-Do 6

a. The most important thing I learned from this assignment was?

Answer: To be able to apply OpenMP in parallelizing the program without actually changing a large chunk of the original code.

b. One thing I am still not clear on is?

Answer: OpenMP is another form of thread that we used for ps5, but the performance improvement achieved by OpenMP is better than the thread we used for ps5. I am not clear why, are there any difference between the two?