Questions
=========

Norm
----

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

We're making a call to omp_set_num_threads(nthreads) to set the number of threads.

* Which version of ``norm`` above provides the best parallel performance? How do the results compare to one of the parallelized versions of ``norm`` from ps5? Justify why one is faster than the other. You can select any one of the parallelized versions of ``norm`` from ps5.

The 'norm_block_reduction' version performs the best. The GFlops for this version are listed below:

N	Sequential	1 thread	2 threads	4 threads	8 threads
1048576	2.28282	2.32505	4.13821	7.98083	17.1895
2097152	2.07349	2.11249	3.56553	7.32484	13.9424
4194304	2.17054	2.21757	3.81158	6.98573	10.0334
8388608	2.20289	2.07433	3.92725	6.70017	9.93911
16777216	2.17886	2.1876	4.05397	6.64951	9.65061
33554432	2.09528	2.26282	3.67002	7.65085	10.1242

The best performing version of norm from ps5 was 'fnorm' with async launch policy. The fnorm performance is listed below:

N	Sequential	1 thread	2 threads	4 threads	8 threads
1048576	8.28547	4.41409	5.02899	6.92347	3.84345
2097152	5.19498	3.84741	6.91493	8.8225	4.54849
4194304	3.22639	2.50856	4.1943	6.59482	6.99051
8388608	3.20741	2.76781	4.60135	8.07792	8.1382
16777216	3.23528	2.91416	5.06209	9.10392	9.03389
33554432	3.29773	2.64729	5.18215	9.79691	9.32068

The 'norm_block_reduction'version performs slightly better as we increase the number of threads to 8 while the performance is slightly worse for 'norm_block_reduction' at a lower number of threads. The performance gain for the omp version could be because omp allows us to explicitly assign the number of available threads to 8 while with the <futures> version we could not utilize 8 threads on a 6 core cpu.

* Which version of ``norm`` above 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 one of the

parallelized versions of ``norm`` from ps5? Justify why one is faster than the other. You can select any one of the parallelized versions of ``norm`` from ps5. It can be either a different version or the same version of the previous question.

'norm_block_reduction' and 'norm_parfor' both provided the best performance (and similar) for larger problems, with performance increasing as the number of threads are increased see above. The performance of 'norm_block_reduction' is shown above in Q1, and for 'norm_parfor', it is as follows:

N	Sequential	1 thread	2 threads	4 threads	8 threads
33554432	2.10467	2.12755	3.71647	7.07473	9.91059

We again see that with smaller number of threads fnorm performs slightly better than omp version, while omp does slightly better with 8 threads. The block_critical and cyclic critical perform the worst. Using critical ensures safety but effectively serializes different threads and parallelization benefits are lost. cyclic_reduction can still gain from parallelization but is less ideal than block_reduction and parfor due to non-contiguous memory operations.

* Which version of ``norm`` above 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 one of the parallelized versions of ``norm`` from ps5? Justify why one is faster than the other. You can select any one of the parallelized versions of ``norm`` from ps5. It can be either a different version or the same version of the previous two questions.

We tried problem sizes from 8196 to 1048576. Block_reduction performs the best and parfor has a slightly worse(<10% difference) performance. Block_critical and cyclic_critical have the worst performance while cyclic_reduction is somewhere in between. The reasons are again that using critical effectively serializes the code and cyclic uses non-contiguous memory operations.
The block_reduction and parfor are much faster for 8 threads than fnorm at small problem sizes, (especially close to 1048576, while roughly the same performance for 1, 2 and 4 threads. The reason could be that omp allows explicitly specifying the number of threads, which allows usage of 8 threads, while there no such mechanism in <futures>.

Sparse Matrix-Vector Product

* How does ``pmatvec.cpp`` set the number of OpenMP threads to use?

A call is made to omp_set_num_threads(nthreads) to set the number of threads.

* (For discussion on Piazza. You can discuss this question on Piazza but you have to answer this question independantly in your submission.) 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?

Consider the CSR matrix, if different rows of the matrix have an identical column index for a non-zero element e.g. rows r1, r2, r3 … rn each have a non-zero element for the column id ci, there would be a race condition at y(ci), causing error to be more likely at that location. The number of such rows- r1 to rn increases the odds of error would be higher. Having information about what rows share common column indices, and what column indices are shared, can be used to separately calculate such y(ci)(or placed under critical section), while rest of the y can be computed using omp parallel for.

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

We parallelized matvec and t_matvec in both CSR and CSC.
For matvec in CSR and t_matvec in CSC we used omp parallel for, for the outer loop. We saw about 1.5-2x speed up on each 2x increase in the number of threads. For t_matvec in CSR and matvec in CSC, we protected the race condition by using omp critical on the inner loop, this lead to decrease in performance on increasing the number of threads.

Sparse Matrix Dense Matrix Product (AMATH583 Only)

* 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?

We parallelized matmat for both the CSR and CSC matrices. We used omp parallel for directive for the outer loop in each case. We saw approximately 1.5-2x speed up each time on increasing the number of threads by a factor of 2, for both CSR and CSC.

PageRank Reprise
---

* 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?

We changed the matrix type to CSCMatric in pagerank.hpp and pagerank.cpp and used the read_cscmatrix function in pagerank.cpp.
We see significant speedup due to parallelization.
For example, with web-Standord.mmio dataset, the runtimes for pagerank are as follows: 1 Thread- 643 ms; 2 Threads- 553 ms; 4 Threads- 299 ms; 6 Threads- 263 ms and 8 Threads- 228 ms.

* (Extra Credit) Which functions did you parallelize? How much additional speedup did you achieve?

We parallelized one_norm and two_norm in amath583.cpp using parallel for reduction(+:sum). We only saw a marginal (<5%) speedup for web-Stanford.mmio dataset.

Load Balanced Partitioning with OpenMP
---

* 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?

We experimented with the following options-
omp_set_schedule(omp_sched_static,x); x = 0,4,8
omp_set_schedule(omp_sched_dynamic,x); x = 0,4,8
omp_set_schedule(omp_sched_guided);
omp_set_schedule(omp_sched_auto);
We tested the pagerank algorithm on web-Google.mmio for nthreads =4,8. The performance for all the schedule conditions varied between 700-900 ms over different iterations for 4 threads and between 550-650 ms for 8

threads. We also tested the algorithm for G_n_pin_pout.mtx, but no significant difference was seen.

OpenMP SIMD
---

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

We vectorized the 'norm_block_reduction' and 'norm_block_critical' functions. The performance gain was significant over the non-vectorized versions. 'norm_block_reduction' showed almost 3-4x improvement for small problem size N = 1048576, with gain decreasing as the number of threads increased. As the problem size increased the gain due to simd over the non-vectorized version reduced to approximately 1.5x gain.

About PS6
---

* The most important thing I learned from this assignment was ...

How to parallelize a serial code without much change in the structure of the code e.g. norm_parfor.

How to use loop vectorization within each thread to get further speed up.

* One thing I am still not clear on is ...

Adding 'omp_set_schedule(…)' throws the following warning: 'indentifier "omp_set_schedule" is undefined' in the script. But, the code compiles without any warning and there are no significant performance improvements for different scheduling options.