Question 1

- At what problem size do the answer between the computed norms start to differ?

Answer: I found out that the two_norm forward and two_norm reverse start to differ around the problem size 1100. For the two_norm forward and two_norm sorted, the norm starts to differ around the problem size 1600. However, I found out that for smaller problem sizes, sometimes the computed norms are different from the two-norm forward, but I think that for very small problem sizes, the result maybe not very insightful. For example, from my end, I have the two-norm forward differs from the two-norm reversed at problem size of 2. I got Absolute difference: 1.11022e-16 and Relative difference: 1.16342e-16, since the differences are both on e-16, I think choose a larger problem size would be more useful.

- How do the absolute and relative errors change as a function of problem size?

Answer: For problem size 1100, for two-norm forward and two-norm reversed, I have Absolute difference: 7.10543e-15 and Relative difference: 1.89159e-16.

For problem size 11000, for two-norm forward and two-norm reversed, I have Absolute difference: 2.84217e-14 and Relative difference: 2.34869e-16.

For problem size 110000, for two-norm forward and two-norm reversed, I have Absolute difference: 5.68434e-14 and Relative difference: 1.48572e-16.

For problem size 1100000, for two-norm forward and two-norm reversed, I have Absolute difference: 2.50111e-12 and Relative difference: 2.06418e-15.

From the above examples, it is clear that as we increase the problem size, the absolute difference and relative difference also show a general increase trend. So, it seems like the absolute and relative errors are directly proportional to the problem size.

- Does the Vector class behave strictly like a member of an abstract vector class?

Answer: For an abstract vector class, the two-norm reversed and two-norm sorted should be exactly the same as the two-norm forward no matter how large the problem size is as long as all the elements in the vector are included for once. However, in our case, as problem size increases, we have two-norms implemented in the different ways different from each other. So, the Vector class does NOT behave strictly like a member of an abstract vector class.

- Do you have any concerns about this kind of behavior?

Answer: For large-scale and complex computations, the errors we get for each loop/function by such vector computation would accumulate very fast so that the final result might be different as expected to cause trouble.

Question 2: pnorm

- What was the data race?

Answer: We passing sum into the helper function worker_a so that every thread accesses and updates sum before the previous thread finishes its process. This causes the following thread accesses and updates from an incorrect partial sum caused by the previous threads.

- What did you do to fix the data race? Explain why the race is actually eliminated (rather than, say, just made less likely).

Answer: In the helper function worker_a, I created a double temp variable, set it equals to 0.0. Also, before the function worker_a I created a mutex called p_mutex. Inside the function worker_a, inside the for loop, instead of updating the partial variable, I have temp += x(i) * x(i); so that the partial variable is not changed yet. Then I used Lock Guard to bloock scope:

```
{ std::lock_guard<std::mutex> p_guard(p_mutex);
    partial += temp;
```

}

I am using the mutex Lock Guard to lock the updating process of the partial variable in the current thread to make sure the update is done properly and completely, then release the updated partial variable to the next thread. With those updates, I can ensure the correctness of my sum update after each thread.

- How much parallel speedup do you see for 1, 2, 4, and 8 threads?

Answer: For 1 thread, it's about 0.97 times speedup (no speedup); for 2 threads, it's about 1.63 times speedup; for 4 threads, it's about 1.96 times speedup; for 8 threads, it's also about 1.98 times speedup. We see that for 4 threads and 8 threads, the performance improvement is about the same, both showing about 2 times the performance compared with the sequential performance.

Question 3: fnorm

- How much parallel speedup do you see for 1, 2, 4, and 8 threads for `partitioned_two_norm_a`?

Answer: For 1 thread, it's about 0.96 times speedup (no 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. The result is very similar when we see the speedup for pnorm in the previous section. We see that for 4 threads and 8 threads, the performance improvement is about the same, both showing about 2 times the performance compared with the sequential performance.

- How much parallel speedup do you see for 1, 2, 4, and 8 threads for `partitioned_two_norm_b`?

Answer: For 1 thread, it's about 0.98 times speedup (no speedup); for 2 threads, it's about 0.99 times speedup (no speedup); for 4 threads, it's about 0.99 times speedup (no speedup); for 8 threads, it's also about 0.97 times speedup (no speedup). We can see that clearly in this case, for 1, 2, 4, 8 threads, there is no speedup compared with the sequential performance.

- Explain the differences you see between `partitioned_two_norm_a` and `partitioned_two_norm_b`.

Answer: For `partitioned_two_norm_a`, we getting speedup result very close to pnorm where we used `thread`, which makes sense. For `partitioned_two_norm_b`, as thread number goes up, we don't see a speedup. This is very intuitive since for `partitioned_two_norm_b` we "deferred" the run until get() is called. For `partitioned_two_norm_a`, we have the program run right away and when we call get(), we can simply get the results instead of just starting running the program.

Question 4: cnorm

- How much parallel speedup do you see for 1, 2, 4, and 8 threads?

Answer: I used Cyclic Partitioning + Tasks but from the result I got, there is no speedup for 1, 2, 4, and 8 threads compared with the sequential performance. I have passed all the test cases, but still all the numbers I am getting is lower than the sequential performance reading. So, I went ahead tried to implement the Cyclic Partitioning + Threads, but it seems like the outcome is very similar. For 1 thread, it's about 0.33 times the speedup; for 2 thread, it's about 0.54 times the speedup; for 4 thread, it's about 0.68 times the speedup; for 8 threads, it's about 0.48 times the speedup. We see that by implementing the Cyclic Partitioning, we actually getting slower performance.

- How does the performance of cyclic partitioning compare to blocked? Explain any significant differences, referring to, say, performance models or CPU architectural models.

Answer: The overall performance of cyclic partitioning is lower than blocked. It is worth to notice that the cyclic partitioning performance is even lower than the sequential performance. We are getting worse performance when use cyclic partitioning. We are getting very similar performance for block partitioning with threads and tasks with both getting about 2 times the performance compared with the sequential performance. This might because for block partitioning, we try to fetch the data next to each other, so that we can access the data we want directly from the cache. However, for cyclic partitioning, we try to fetch the data that are several steps away from each other, so that we lose the benefit of fetching them directly

from cache L1 or L2 since they may not be in there. We might need extra time to get the data we want from higher level caches such as L3, L4 or even from RAM, which cause more time.

Question 5: rnorm (Amath 583 Only)

- How much parallel speedup do you see for 1, 2, 4, and 8 threads?

Answer: This is for Amath 583 Only, I am an Amath 483 student.

- What will happen if you use `std:::launch::deferred` instead of `std:::launch::async` when launching tasks? When will the computations happen? Will you see any speedup? For your convenience, the driver program will also call `recursive_two_norm_b` -- which you can implement as a copy of `recursive_two_norm_a` but with the launch policy changed.

Answer: This is for Amath 583 only, I am an Amath 483 student.

Question 6: General

- For the different approaches to parallelization, were there any major differences in how much parallel speedup that you saw?

Answer: For my fnorm and pnorm, I got very close results. Both approaches achieved about 2 times the performance compared with the sequential performance. But, my cnorm's results are very different from my fnorm and pnorm results as for my cnorm the results are actually worse than the sequential performance. The reason for this have already been included in Question 4.

- You may have seen the speedup slowing down as the problem sizes got larger -- if you didn't keep trying larger problem sizes. What is limiting parallel speedup for two_norm (regardless of approach)? What would determine the problem sizes where you should see ideal speedup? (Hint: Roofline model.)

Answer: Since we need the data load into cache to let threads fetch the data and solve the problem, as the problem sizes increases, we would have more and more data that we need to load them into cache. When to the point where the problem size is too large to load the majority data into the cache, no matter how many thread we are using, the algorithm will not speed up by much since thread does not help us to load data into cache, it only solves the problem with the data that is already in the cache. If the problem size is so large that the cache if full, no matter how many thread we have, we cannot speed up. So, the cache is the key element to limit parallel speedup for two_norm, and it is also the key factor to determine the problem sizes where you should see ideal speedup. We can see this with a Roofline model provided with this assignment with instructor's computer: as when we have more threads, our RAM does not always scale with the number of threads we have.

Question 7: Conundrum #1

1. What is causing this behavior?

Answer: When the problem size is small, our computer is already capable to solve the problem fast without parallelization. The reason for parallelization is that when we have a very large problem size, we try to have each thread/task to do part of the work so that the overall performance and execution time can be optimized. However, in order to achieve parallelization, we need to add and modify the code a bit so it works that way. In this assignment, for example, I used Block + Threads with Lock Guard (mutex) in pnorm.hpp to achieve parallelization. For large problem size, this modification can enhance the performance. However, for smaller problem sizes, when we use parallelization, everytime when we lock and unlock the lock guard, and moves to the next thread would take a large part of the execution time. It appears that it's not worth the work for a smaller problem size to use parallelization because the time for those extra work might ended up with taking more time than just solve the problem without parallelization. This is why this command is running so slow. I waited for 10 minutes, and it's still not done. (./pnorm.exe 128 256)

2. How could this behavior be fixed?

Answer: When we try to solve a problem, we can write code (e.g. if loop) to check the problem size. If the problem size is large, we use the parallelized version of function to solve the problem; if the problem size is small, we can just solve the problem directly without parallelization.

3. Is there a simple implementation for this fix?

Answer: I think so, but we are not required to write it down from the assignment instruction. We are only required to do 1 and 2 based on the assignment instruction.

Question 8 & 9 are not required in Questions.rst

Question 10: Parallel matvec

- Which methods did you implement?

Answer: I used Block + Thread approach when implementing the matvec(), and t_matvec() for CSRMatrix.hpp and CSCMatrix.hpp. For the four functions implemented, I have one helper function for each of them. The four overloaded functions are implemented in a way such that parallelization can be achieved.

- How much parallel speedup do you see for the methods that you implemented for 1, 2, 4, and 8 threads?

Answer: If we take COO as the sequential performance for comparsion, then:

For CSR matvec (compared with COO): For 1 thread with grid size N <= 512, I am getting lower reading for CSR compared with COO, but for larger N >= 1024, it's about 1.25 times speedup in the performance. For 2 threads, for grid size N >= 256, it's about 1.77 times speedup in the performance. For 4 threads, for grid size N >= 256, it's about 1.95 times speedup in the performance. For 8 threads, for N >= 512, it's about 2.05 times speedup in the performance. So overall, for CSR matvec with parallelization, with 4 or more threads and larger grid size, it's about 2 times the speedup in the performance compared with COO matvec.

For CSR t_matvec (compared with COO^T): For 1 thread with grid size N >= 2048, it's about 1.01 times speedup in the performance. For 2 threads with N >= 256, it's about 1.73 times speedup in the performance. For 4 threads with N >= 256, it's about 2.29 times speedup in the performance. For 8 threads with N >= 512, it's about 2.13 times speedup in the performance. We see that the best performance speedup achieved by CSR t_matvec with parallelization is about 2.3 times speedup compared with COO t_matvec.

For CSC matvec (compared with COO): For 1 thread with grid size N <= 2048 (all grid size provided), I am getting lower reading for CSC compareed with COO. It's about 0.86 times the performance compared with COO (no speedup). For 2 threads with N >= 256, it's about 1.32 times speedup in the performance. For 4 threads with N >= 256, it's about 1.93 times speedup in the performance. For 8 threads with N >= 512, it's about 1.95 times speedup in the performance. So overall, for CSC matvec with parallelization, with 4 or more threads and larger grid size, it's about 2 times the speedup in the performance compared with COO matvec.

For CSC t_matvec (compared with COO^T): For 1 thread with grid size N >= 1024, it's about 1.31 times speedup in the performance. For 2 threads with grid size N >= 256, it's about 1.97 times speedup in the performance. For 4 threads with grid size N >= 256, it's about 2.31 times speedup in the performance. For 8 threads with grid size N >= 512, it's about 2.0 times speedup in the performance. We see that the best performance speedup achieved by CSC t_matvec with parallelization is about 2.3 times speedup compared with COO t_matvec.

Question 11: Conundrum #2

1. What are the two "matrix vector" operations that we could use?

Answer: First, we could use CSR::t_matvec. Second, we could use the CSC::matvec instead of CSR::matvec. In my pagerank.hpp, I used the second method, and the CSC::matvec I used is the one overloaded with the third parameter being the number of threads (partitions) to achieve parallelization.

2. How would we use the first in pagerank? I.e., what would we have to do differently in the rest of pagerank.cpp to use that first operation?

Answer: First is we use CSR::t_matvec. We can modify the CSRMatrix.hpp (which we already did for To-Do 8) to have a parallelized version of t_matvec. Then, we can change the pagerank.hpp to use CSR::t_matvec. We do not have to call the transpose matrix product to compute the transposed product, instead, We just need to send in the transposed matrix to the call. There is no need to change the pagerank.cpp and pagerank_test.cpp with this operation.

3. How would we use the second?

Answer: Second is we use CSC::matvec (this is the one I used and modified in pagerank.hpp). We can use the one with parallelization in CSCMatrix.hpp (we did this for To-Do 8). In pagerank.hpp, we change the CSR::matvec to CSC:matvec, and when we call mult(), we used the overloaded version wih third parameter being the num_threads. For this operation, we need to modify pagerank.cpp and pagerank_test.cpp in placed where CSRMatrix was used into CSCMatrix. (Please see detailed changes in the files that I submitted.) We have to change this or the testing files will try to keep using the CSR matrix which would cause error. (Since we need to change the cpp and test file, when submitted to Gradescope, I commented out this modification since it will fail the Make command since I assume the test file provided in Gradescope still try to read a CSR matrix instead of CSC matrix.)