Warm UP, Question 1

1. Starts to differ at when the array size reaches 4096.
2. The absolute error grows, porportional to the size of the vector, relative error stays the same.
3. Abstract vector class is undefined in the context of the HW.
4. Yes. The build up of the error could introduce numerical instability for algorithm. This could be a concern when we are summing up large amount of smaller chunks of results for computations, each time there will be different. If the algorithm that is using the result has chaos or it's numerical unstable, then this error will get magnified significantly when doing computations.

5.1.1, Block Partition and Threads, Pnorm, Question 2

(1)

What was the data race?
Data race happens when there are multiple threads working on the same section of code, with a shared variable.
In this case, the "&partial" is shared among several thread, and each of them will read the value and add to it.
Of context switch happens between 2 or more thread between the "Read" and "Increment", the final value will be incorrect, and that is race condition.

(2)

What did you do to fix the data race? Explain why the race is actually eliminated (rather than, say, just made less likely).
The race codntion is eliminated by referring to an instance of "std::mutex" and the use of "std::lock_guard" on the mutex instance.
The thread that has the access to mutex will be able to work on reading and changing the variable at the same time with that memory, and no other threads will be able to work on that critical section while the function hasn't been terminated yet.
I am sure that fixes it, beause that is what is taught in the lecture.
I also first accumated the partial sum in a local variable instead of adding it directly to the shared variable, this allows multiple thread to work on their own part of the partial sum first, before adding their resulted partial sum to the shared variable.

(3)

5.1.2 Block Paritioning + Task, Fnorm, Question 3

```
   N  Sequential    1 thread   2 threads   4 threads   8 threads  16 threads      1 thread     2 threads     4 threads     8 threads    16 threads
 1048576     2.26136     2.01762     1.59185     1.65709     1.52233     1.11325             0   3.54027e-14   2.52052e-14   2.53976e-14   2.61672e-14
 2097152     1.44142     1.53665     2.67209     3.31201     2.75852     2.01856             0   1.29206e-14   6.80033e-15   1.04725e-14   8.84043e-15
 4194304     2.38313     2.28947     3.06601     3.61578     3.01315     2.69557             0   3.13487e-14   3.94262e-14   2.73099e-14   3.00024e-14
 8388608     2.45612     2.30067     3.03766     3.54065     3.23596     2.57806             0   6.48634e-14   5.98321e-14   6.21438e-14   7.95495e-14
16777216     1.84365     2.25414     3.14854     3.46432      3.5588     3.47457             0   2.28867e-14   1.92325e-14   1.23088e-14   7.30835e-15
33554432     2.20029     2.05855     3.07839     3.63734     3.73866      3.8022             0   3.39496e-13   3.29163e-13   3.22365e-13   3.27259e-13

       N  Sequential    1 thread   2 threads   4 threads   8 threads  16 threads      1 thread     2 threads     4 threads     8 threads    16 threads
 1048576     1.90722     2.20705     2.27664     2.24628     2.31841     2.17851             0   1.88503e-14    1.5965e-14   1.71191e-14   2.03891e-14
 2097152     2.33655     2.15002     2.34727     2.14551     2.15002     2.27424             0   7.75133e-15   2.21661e-14   1.79504e-14   1.50947e-14
 4194304     1.95266     2.38856     1.92753     1.97845     2.42165     2.36699             0   3.09629e-14   9.42348e-15   9.42348e-15   1.13466e-14
 8388608     2.01203     2.02323     2.41265     1.97201     1.94388     1.92671             0   4.35216e-15   6.80026e-16   5.71221e-15   5.71221e-15
16777216     2.06036     2.33945      2.1009     2.30276     2.02135     2.31182             0   7.69133e-15   4.23023e-15   3.26882e-15   2.88425e-15
33554432     2.33422     2.33422     2.10703     2.40534     2.41834     2.42708             0   1.61279e-13   1.41289e-13   1.23883e-13   1.28098e-13
```

(1)

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

As we can see from the data, there is not much of an increase in Gflop/sec for small value of N. And to compute the speed up, I wrote a python code that does it.
The total amount of flops for computing the norm is $2N$. The amount of time it takes to evaluate it will be $2N$ divide by the data above. And, the amount of speed up is computed by: $\frac{T(1,n)}{T(p,n)}$, the time it takes for one thread to compute over the time it takes for p thread to compute.

	N	Sequential	thread 1	thread 2	thread 4	thread 8	thread 16
0	1048576.0	1.120806	1.0	0.788974	0.821309	0.754518	0.551764
1	2097152.0	0.938028	1.0	1.738906	2.155344	1.795152	1.313611
2	4194304.0	1.040909	1.0	1.339179	1.579309	1.316091	1.177377
3	8388608.0	1.067567	1.0	1.320337	1.538965	1.406529	1.120569
4	16777216.0	0.817895	1.0	1.396781	1.536870	1.578784	1.541417
5	33554432.0	1.068854	1.0	1.495417	1.766943	1.816162	1.847028

(2)

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

	N	Sequential	Thread 1	Thread 2	Thread 4	Thread 8	Thread 16
6	1048576.0	0.864149	1.0	1.031531	1.017775	1.050456	0.987069
7	2097152.0	1.086757	1.0	1.091743	0.997902	1.000000	1.057776
8	4194304.0	0.817505	1.0	0.806984	0.828302	1.013854	0.990969
9	8388608.0	0.994464	1.0	1.192474	0.974684	0.960781	0.952294
10	16777216.0	0.880703	1.0	0.898032	0.984317	0.864028	0.988190
11	33554432.0	1.000000	1.0	0.902670	1.030468	1.036038	1.039782

(3)

Explain the differences you see between “partitioned_two_norm_a“ and “partitioned_two_norm_b“.

The speed up of using eager evaluation on the async object goes up as the problem size goes up and the thread count goes up. However, for smaller inputs, there is no speed up, there is speed down instead. Which could a mixed of lack of universal scheduler, private L1 Cache, and the overhead for threads and limited memory band width.
Due to the limited memory bandwidth and low numerical intensity of the algorithm the speed up cannot be gained by adding more thread, as it can be seem here.
There is no speed up when deferred launch is applied to all the threads. It’s easily observed in here.

5.1.4 Cyclic Partitioning, Cnorm, Question 4

This is the results by running the cnorm:

```
  N  Sequential    1 thread   2 threads   4 threads   8 threads  16 threads       1 thread      2 threads      4 threads      8 threads     16 threads
  1048576     2.37283      1.9439     2.10589     1.66254    0.875934    0.661536              0   3.15546e-14   2.11647e-14   2.82837e-14   2.52052e-14
  2097152     2.44251     2.11886     1.20828     1.08412    0.901683    0.765453              0   2.28491e-14   1.10165e-14   1.38727e-14   7.07234e-15
  4194304    0.974513     1.69947     1.98594     1.59116    0.873813    0.641723              0   4.11572e-14   3.59644e-14   3.07717e-14   3.00024e-14
  8388608     2.00095     1.97201     2.23467     2.19863    0.857326    0.657335              0     7.547e-14   8.68925e-14   7.99574e-14   7.51981e-14
 16777216     2.33017     2.22425     2.39675     2.23271    0.616486    0.781894              0   3.73111e-14   9.03928e-15    3.8465e-16   2.69255e-15
 33554432     2.22215       2.046     2.37554     1.81867    0.596258    0.602684              0   3.90617e-13   3.17198e-13   3.20461e-13   3.31066e-13
```

(1)

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

	Sequential	thread 1	thread 2	thread 4	thread 8	thread 16
0	1.220654	1.0	1.083332	0.855260	0.450607	0.340314
1	1.152747	1.0	0.570250	0.511652	0.425551	0.361257
2	0.573422	1.0	1.168564	0.936268	0.514168	0.377602
3	1.014675	1.0	1.133194	1.114918	0.434747	0.333332
4	1.047621	1.0	1.077554	1.003804	0.277166	0.351532
5	1.086095	1.0	1.161065	0.888891	0.291426	0.294567

The speed up is computed by a script I wrote, it's computed by $T(1, n)/T(p, n)$, where the function T is the amount of time takes to finish the task with first arg as thread count and the second arg as the size of the problem.

(2)

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

I made a post about it, about this weird patterns where, performance is decreased when multiple thread uses striding memory accesss patterns on the array.
The roofline model shows that, the memory scale up when there are 2 threads, but scale down again when it grows to 4, and the L1, L2, L3, all scales up as the number of threads goes up.
So a striding memory acess patterns applied on the cyclic norm should at least stays the same compare to the baseline, where there is one thread.
But here my hypothesis on why this happens regardless:

1. AMD Ryzen series doesn't have a unified scheduler across all cores, each of them will operate on their own. Sharing data between cores might be an issue?

2. Stride memory patterns doesn't makes things slow because one fetch on the cache might be consumed by multiple threads, in different cores.

5.1.5 Divide and Conquer, Rnorm, Question 5

Here is the data created from running "rnorm.exe":

```
       N  Sequential    1 thread   2 threads   4 threads   8 threads      1 thread     2 threads     4 threads     8 threads
 1048576     1.63564     2.40101     2.24628     1.37528    0.705392   3.54027e-14   2.52052e-14   2.53976e-14   2.61672e-14
 2097152     2.38557     2.33655     2.18211     2.05092    0.990717   1.29206e-14   6.80033e-15   1.04725e-14   8.84043e-15
 4194304     1.94181      2.5827     3.29741     2.96208     2.14872   3.13487e-14   3.94262e-14   2.73099e-14   3.00024e-14
 8388608     2.47845     3.35544     3.78652     3.26503        2.74   6.48634e-14   5.99681e-14   6.21438e-14   7.95495e-14
16777216     1.96718     3.32693       3.813     4.03576     3.31753   2.28867e-14   1.92325e-14   1.23088e-14   7.30835e-15
33554432     2.04913     3.45922     4.30185     4.22068     3.95923   3.39496e-13   3.29163e-13   3.22365e-13   3.27259e-13
       N  Sequential    1 thread   2 threads   4 threads   8 threads      1 thread     2 threads     4 threads     8 threads
 1048576     2.33988     2.45346     2.40673     2.44752     2.33988   1.88503e-14    1.5965e-14   1.71191e-14   2.03891e-14
 2097152     2.44251     1.89872     1.93828     1.94195     2.33123   7.75133e-15   2.21661e-14   1.79504e-14   1.50947e-14
 4194304     2.37234     2.08051     2.46145     2.36699     2.46724   3.09629e-14   9.42348e-15   9.42348e-15   1.13466e-14
 8388608     2.37069     2.48977      2.4506     2.45612      2.4506   4.35216e-15   6.80026e-16   5.71221e-15   5.71221e-15
16777216     1.98045     2.08228      1.9221     1.98045       2.046   7.69133e-15   4.23023e-15   3.26882e-15   2.88425e-15
33554432     2.01831     1.52347     2.30219     1.78481     2.13044   1.61279e-13   1.41289e-13   1.23883e-13   1.28098e-13
```

(1)

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

This is the table with all the thread launched with eager evaluation:

	N	Sequential	Thread 1	Thread 2	Thread 4	Thread 8	Thread 8
0	1048576.0	0.760666	1.0	0.978660	0.549659	0.244107	0.118363
1	2097152.0	0.805165	1.0	1.082019	0.788508	0.433080	0.268809
2	4194304.0	0.770877	1.0	1.192055	0.774193	0.681819	0.453972
3	8388608.0	0.717147	1.0	1.183821	1.015770	0.811082	0.590824
4	16777216.0	0.703031	1.0	1.115384	1.104761	1.041914	0.908615
5	33554432.0	0.691792	1.0	1.207602	1.083992	1.200583	1.137743

And this is the speed up for the deferrend implementation.

	N	Sequential	Thread 1	Thread 2	Thread 4	Thread 8	Thread 8
6	1048576.0	1.011710	1.0	1.026125	0.979593	0.997688	0.933044
7	2097152.0	1.141858	1.0	1.152581	1.152581	1.158016	1.158016
8	4194304.0	1.025002	1.0	0.997789	1.006697	1.000000	1.015768
9	8388608.0	1.004377	1.0	1.000000	0.989225	0.995661	0.884391
10	16777216.0	1.026156	1.0	0.990290	1.017961	1.026156	1.019998
11	33554432.0	1.123783	1.0	1.167517	1.148089	1.134872	1.200002

No significant Speed up is observed under both cases. This is the case because to spawn 8 threads at the bottom of the recursion tree, we actually spawned: $1 + 2 + 4 + 8$ threads which is 15 in total, and that is a lot of threads.

(2)

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.

I used "deferred" and it still runs well, and didn't provide much of a speed up when the size of the problem is big. But it's much faster compare to when "Async" is used when the size of the problem is small.

General

(1)

For the different approaches to parallelization, were there any major differences in how much parallel speedup that you saw?
Yes, there are, it happened for the block partitioning case, I didn't happen for the other cases, at least not by a great margin.

(2)

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.)
According the roofline model, The memory bandwitth increases for each threads as we jumps from one threads to 2, but then it decreases because of limited memory bandwidth. In my case, there are 4 memory controllers on the Ryzen 4900HS (But I only have 2 physical RAM), hence it becomes that bottle neck as more and more threads are working with the same chunk of memory in the VRAM.
For small problem size, there is a slow down. The reason is unclear, as it defies the fact that cache hit should increase performance.

Conundrum # 1, Question 7

It runs forever on my laptop and I were not able to finish it and see what we meant by behaviors. So I will try get some larger value to run, and this is what the larger value gives:

```
   N  Sequential    1 thread    2 threads    4 threads    8 threads  16 threads        1 thread      2 threads      4 threads      8 threads     16 threads
  65536       2.843      1.3487     1.30135      1.04135     0.527257    0.136247              0    4.0467e-15      9.635e-16      1.927e-15     1.5416e-15
 131072     2.79718     1.42243     1.81411      1.60222     0.808224    0.279172              0   9.91532e-15   4.61809e-15   5.84053e-15   5.02557e-15
 262144     2.71071     1.79101     2.64634       2.5717      1.74126    0.564096              0   1.73336e-15   4.62228e-15   5.97045e-15   5.20007e-15
```

And we can see that, Gflops per second decreases as the number of threads increase. This could be the result from cache line hit when there are multi-thread. Or, the cost and over head of creating the thread outweight the performance incrase when the size of the problem is too small.
And this is the speed up table:

	N	Sequential	Thread 1	Thread 2	Thread 4	Thread 8	Thread 16
0	65536.0	2.107956	1.0	0.964892	0.772114	0.390937	0.101021
1	131072.0	1.966480	1.0	1.275360	1.126396	0.568199	0.196264
2	262144.0	1.513509	1.0	1.477569	1.435894	0.972222	0.314960

This is a tremendous slow down. Don't use several threads when the is really small.

Question 8

No we cannot partition both vector for a sparse matrix. If we partition by rows then we partitioned y, but not x, if we partition by columns then we partition x but not y. Transposing things doesn't help with partition both x and y.
The only way to achieve that is Block Sparse Matrix like this:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Where, each sub matrices are stored as sparse matrix. And this will be very efficient if they turns out to be the same matrix.
And in the case of dense matrix, we can do this recusively and combine the results as we goes from the root of the recursion trees to the top of the recrusion tree.

Question 9

(Answer not required in Questions.rst but you should think about this and discuss on Piazza.) Are there any potential problems with not being able to partition x? Are there any potential problems with not being able to partition y? (In parallel computing, "potential problem" usually means race condition.)

I would rather patition on y. This is the case because if y is not partition, then we will have multiple threads modifying the entries of the y vector. And this is potential problem for data races some tiny bit of overhead too.

Parallel Matvec

(1)

Which methods did you implement?

Firstly, for a CSR matrix, the most efficient ways of partitioning is by rows. We partition the A matrix by rows in to shorter sub-matrices with the same width as A.
For the Matrix vector multiplication of the CSR matrix, we don't want to break the inner forloop, nor we

want to share the same element in vector y with different thread, and the inner forloop accumulates on the row of the matrix, and that would mean we don't want to split the matrix x, therefore, the only option left is splitting the output vector y and sharing the matrix x on different thread. Vector x will be read only.
For the transpose however, we are takine the dot product with the vector x with each column of A; let's say the j th column, and then store it to the j th element of the y vector. However, since we have to lock on the row and then goes through the column, multiple threads will have to modify the content of the vector y at the same time when we partition the matrix by row. Therefore, it's better to let each thread withold each element and then we accumulate the results from each thread, and the parallelization stil works.

For the CSC matrix, the outer loop will lock on the column of the matrix and then iterate through all the rows while fixing a certain column of the matrix. And in this case, it will mean that multiple threads will be writing the results to the vector y. Potentially creates race conditions.
For transpose multiplication, we have the same problem of if we choose to split the matrix by columns, we will have to handle the race race condition when storing the results to the y vector.

(2)

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

Conundrum# 2, Question 11

Note: I ran out of time to finish the HW, I have to do some works about PDEs and Statistics now. Sorry C++, I might use late days but it depends, and if this is not removed then that mean I just don't have time to do it even with the late days.