Rohan Tiwari
AMATH 583 PS5

Question 1:

1.1 At what problem size do the answers between the computed norms start to differ?

The answers for sorted norm start to differ from N=8:

Calling two_norm first time: 2.94053

Calling two_norm second time: 2.94053

Calling two_norm with reversed values: 2.94053

Calling two_norm with sorted values: 2.94053

Norms differ!

Absolute difference: 4.44089e-16

Relative difference: 1.51023e-16

The answers for reverse norm start to differ from N=32:

Calling two_norm first time: 6.35792

Calling two_norm second time: 6.35792

Calling two_norm with reversed values: 6.35792

Norms differ!

Absolute difference: 8.88178e-16

Relative difference: 1.39696e-16

Calling two_norm with sorted values: 6.35792

Norms differ!

Absolute difference: 8.88178e-16

Relative difference: 1.39696e-16

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

Relative errors seem to remain around the same but absolute error increases with problem size.

1.3 Does the ``Vector`` class behave strictly like an algebraic vector space?

It does not because it should override other methods as well and not just the operator.

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

As absolute errors become larger with increasing problem size it can be a problem for applications that are operating on very large input sizes. E.g., such applications might breakdown the problems into

smaller pieces to solve and each partial problem itself is large enough to have enough error that combining the results makes it even worse.

Question 2 (Block Partitioning + Threads):

2.1 What was the data race?

'partial' is passed by reference and is therefore a shared variable. The worker_a methods is executed in parallel by different threads which end up operating on 'partial' variable's value in parallel leading to unreliable outcomes as there is no safeguard like lock / mutex in place. This is a race condition.

2.2 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 condition is solved by having each thread operate on the worker function and modify a local variable local_partial with the sums it has calculated for its problem size i.e. from begin till end of the vector x. After this, under a guarded section the shared variable sum is modified to accumulate this partial sum. Instead of applying the mutex inside the for loop, the mutex is applied at one place outside the loop to increase efficiency. Lock_guard follows RAII style programming practice by limiting the scope of mutex and releasing it as soon as the control leaves that block.

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

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.42987	1.86844	2.40673	2.99061	2.70275	0	1.92406e-16	3.84812e-16	3.84812e-16
2097152	2.39115	1.63223	2.67908	2.59091	2.08434	0	1.49607e-15	1.49607e-15	1.90409e-15
4194304	2.12693	2.27457	3.03935	3.39345	2.9874	0	2.5002e-15	2.69252e-15	2.88485e-15
8388608	2.48977	2.29101	3.29462	3.62299	3.42931	0	4.07946e-16	4.07946e-16	8.15892e-16
16777216	2.41647	2.22847	3.157	3.61355	3.65858	0	2.3079e-15	2.69255e-15	1.92325e-15
33554432	2.30615	2.4994	3.51355	3.85683	3.6772	0	5.3025e-15	7.47788e-15	7.34192e-15

Overall, looks like 4 threads is performing the best. For the largest problem size N=33554432, 4 threads gives 1.54x speedup over 1 thread and 1.05x speedup over 8 thread case.

2 threads performs better than 1 thread case. For the largest problem size N=33554432, 2 threads give 1.40x speedup over 1 thread. For N=33554432, 8 threads gives 1.47x speedup over 1 thread.

Question 3 (Block Partitioning + Tasks):

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

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.42405	1.97042	2.42405	2.90468	2.5334	0	1.92406e-16	3.84812e-16	1.92406e-16
2097152	2.43669	2.14103	2.85869	2.81931	2.11886	0	1.49607e-15	1.49607e-15	1.90409e-15
4194304	1.75641	2.23101	2.78877	3.1115	2.80368	0	2.5002e-15	2.69252e-15	2.88485e-15
8388608	2.54794	2.0693	3.31465	3.58723	3.25528	0	4.07946e-16	4.07946e-16	6.7991e-16
16777216	2.53105	2.21169	3.14012	3.54805	3.61355	0	2.3079e-15	2.69255e-15	1.92325e-15
33554432	2.14063	2.10703	3.173	3.6772	3.53205	0	5.3025e-15	7.47788e-15	7.34192e-15

4 threads seem to perform the best overall.

Comparing N=33554432 performance with 1 thread, we observe a speedup of 1.51x for 2 thread, 1.75x for 4 thread and 1.67x for 8 thread.

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

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.47752	2.34531	2.40673	2.44752	2.49587	0	1.1541e-15	7.69399e-16	3.847e-16
2097152	2.57786	1.92009	1.80178	2.38002	2.42514	0	1.90383e-15	2.3118e-15	1.63186e-15
4194304	2.57635	2.51457	2.53892	2.52669	2.47306	0	1.1539e-15	3.84632e-16	1.92316e-16
8388608	2.46167	2.27191	2.59647	2.51272	2.54794	0	1.08804e-15	2.99211e-15	2.31209e-15
16777216	2.42145	2.5256	1.89115	2.34881	2.51479	0	6.15306e-15	7.49905e-15	8.26818e-15
33554432	2.00624	1.45415	2.29432	1.35573	1.41133	0	1.9038e-15	1.35985e-15	3.39963e-15

1 thread seem to perform the best overall.

Comparing N=33554432 performance with 1 thread we observe a speedup 0.93x for 4 thread and 0.97x for 8 thread basically indicating adding more threads is useless.

2 threads gave a slight speedup in general over 1 thread case for the same N. Comparing N=33554432 we see a speedup of 1.57 but this is not consistent.

3.3 Explain the differences you see between partitioned_two_norm_a and partitioned_two_norm_b.

When deferred is used, the execution is deferred until the get() is called in the second for loop which means that the task is run sequentially on the calling thread. When async is used, the execution of tasks begins right away on different threads and the results are combined in the second for loop where .get()waits for the task to complete.

Question 4 (Cyclic Partitioning + Your Choice) :

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

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.37842	1.0304	1.4931	0.976645	0.730895	2.52052e-14	6.3494e-15	4.04052e-15	3.07849e-15
2097152	2.10578	1.14604	1.45371	1.239	0.805835	6.93633e-15	1.57768e-14	3.94419e-15	6.80033e-15
4194304	2.49661	1.30745	1.21786	1.18617	0.860194	2.98101e-14	1.11547e-14	5.96202e-15	7.69293e-16
8388608	2.22102	1.17513	1.60844	1.44058	0.872415	7.53341e-14	2.71964e-16	1.16945e-14	4.75937e-15
16777216	2.46207	1.38983	1.85237	1.54731	0.918221	2.69255e-15	3.46185e-14	6.34673e-15	3.0772e-15
33554432	2.41399	1.38512	1.85897	1.47492	0.829016	3.3093e-13	5.96871e-14	1.38681e-14	1.03331e-14

2 threads implementation generally looks the fastest

Comparing for N=33554432, we see a speedup of 1.34x for 2 threads, 1.06 for 4 threads. 8 threads is consistently slow.

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

Cyclic partitioning generally performs poorly compared to blocked. In blocked implementation, each task (and thread) gets a contiguous block of memory to work with which is not the case with cyclic implementation. Contiguous block of memory can easily pull into closer caches increasing locality and efficiency. In cyclic implementation as the data is not contiguous, efficiency is lost in taking strides and accessing data which leads to more cache misses.

Question 5 (Divide and Conquer + Threads)[Gradescope has this but not the website]:

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

recursive_two_norm_a was implemented using threads to get the below results first and later this implementation using tasks :

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.39533	1.90363	2.56555	2.94702	2.91305	1.92406e-16	1.92406e-16	1.03699e-06	3.84812e-16
2097152	2.42514	2.11448	2.81157	3.01003	3.0733	1.49607e-15	1.90409e-15	1.06972e-06	2.0401e-15
4194304	2.54509	2.28448	3.1115	3.31828	3.39345	2.5002e-15	2.88485e-15	3.67101e-11	2.88485e-15
8388608	2.59647	2.3452	3.08929	3.08929	3.26503	4.07946e-16	8.15892e-16	3.29395e-08	1.08786e-15
16777216	2.43653	2.37254	3.46432	3.50569	3.50569	2.3079e-15	1.92325e-15	5.77292e-09	1.34628e-15
33554432	2.32613	2.30615	3.46816	3.48617	3.63734	5.3025e-15	7.34192e-15	3.6516e-08	6.66211e-15

8 threads implementation comes out toe best the fastest but there is gradual speed up going from threads 1 to 8 overall.

If we take the largest problem size, N=3355432, the speed up is 1.50x for 2 threads, 1.51x for 4 threads and 1.58x for 8 threads compared with the 1 thread approach.

Question 6 (Divide and Conquer + Tasks (AMATH 583 ONLY)):

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

Output for std::launch::async:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.30257	1.90005	2.20224	2.45943	2.73937	1.92406e-16	1.92406e-16	1.03699e-06	3.84812e-16
2097152	2.09715	1.69439	2.57786	2.69318	2.77347	1.49607e-15	1.90409e-15	1.06972e-06	1.90409e-15
4194304	2.5575	2.255	3.19688	3.29741	3.1775	2.5002e-15	2.88485e-15	3.67101e-11	2.69252e-15
8388608	2.59031	2.36555	3.32475	3.23596	3.39726	4.07946e-16	8.15892e-16	3.29395e-08	1.08786e-15
16777216	2.542	2.44159	2.99593	2.50406	3.40407	2.3079e-15	1.92325e-15	5.77292e-09	1.5386e-15
33554432	2.04913	2.05855	3.40654	3.56962	3.66715	5.3025e-15	7.34192e-15	3.6516e-08	6.39019e-15

8 threads implementation looks to be the most efficient.

For N=3355432, 8 threads give a 1.73x speedup, 4 threads give 1.65x and 2 threads is almost the same as 1 thread case.

6.2 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.

Output with std::launch::async:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.30257	1.90005	2.20224	2.45943	2.73937	1.92406e-16	1.92406e-16	1.03699e-06	3.84812e-16
2097152	2.09715	1.69439	2.57786	2.69318	2.77347	1.49607e-15	1.90409e-15	1.06972e-06	1.90409e-15
4194304	2.5575	2.255	3.19688	3.29741	3.1775	2.5002e-15	2.88485e-15	3.67101e-11	2.69252e-15
8388608	2.59031	2.36555	3.32475	3.23596	3.39726	4.07946e-16	8.15892e-16	3.29395e-08	1.08786e-15
16777216	2.542	2.44159	2.99593	2.50406	3.40407	2.3079e-15	1.92325e-15	5.77292e-09	1.5386e-15
33554432	2.04913	2.05855	3.40654	3.56962	3.66715	5.3025e-15	7.34192e-15	3.6516e-08	6.39019e-15

Output with std::launch::deferred:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.53977	2.41248	2.48972	2.42405	2.44161	1.1541e-15	5.77049e-16	3.67901e-08	5.77049e-16
2097152	2.46012	1.88821	1.73459	2.38002	2.45422	1.90383e-15	1.63186e-15	7.44777e-07	1.76785e-15
4194304	2.43855	2.54509	2.37772	2.33017	2.31986	1.1539e-15	1.92316e-16	2.26023e-08	1.92316e-16
8388608	2.46167	2.37586	2.4841	2.35533	2.3252	1.08804e-15	2.31209e-15	2.04331e-07	2.99211e-15
16777216	2.5256	2.5256	2.54752	2.49873	2.5256	6.15306e-15	8.26818e-15	1.03977e-08	8.26818e-15
33554432	2.52764	2.36716	2.5372	2.44923	2.46271	1.9038e-15	3.39963e-15	6.5245e-08	1.9038e-15

The deferred case does not offer much consistent speed up. The deferred threads are not started until the. get() is called at which point the program essentially becomes sequential. For the async case, threads are started right away and their results are collected by calling .get() later.

Question 7 (General):

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

Block partitioning with tasks seems to be the most efficient overall at 4 threads. Within each approach, the trend is the same with 4 threads being the most efficient. 8 threads performs the best for divide and conquer (recursive approach). Cyclic partitioning performs the worst.

7.2 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.)

Smaller problem sizes would fit in L1/L2 caches on the core, but larger ones would need to use the L3 cache or main memory which are both shared and slower than L1 and L2.

If we consider any of the approaches generally the 4 thread approach comes out to be better than 2 thread approach and then there is a slight drop in performance on the 8 thread approach. 2 thread is generally efficient than the single thread approach. This implies that always adding more threads does not help with performance and could become a bottleneck.

Question 8:

The performance of sequential runs is significantly faster than the multithreaded versions.

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
128	26.3158	0.00321434	0.00231139	0.00147203	0.000513349	0	0	1.36752e-16	1.36752e-16
256	22.7273	3.30948e-05	0.00411077	0.00275125	0.000856728	0	0	0	0

8.1 What is causing this behavior?

For smaller problems, the data easily fits into the nearby caches which is great from a locality point of view. Increasing the number of threads gives small pieces of an already small problem for each thread to operate upon. Each thread has a small amount of work to do. We need to look closely as to how the OS works with multiple threads. The OS uses a time sliced approach, runs each thread for a given time slice then persists its state and starts the next thread. Storing and reloading this state is overhead which will become significant if there is not much work for the many threads to do which is typically the case with smaller problem sizes. Therefore, it is advisable to consider the problem size when determining whether to do multithreading and how many threads to run.

8.2 How could this behavior be fixed?

Can implement multithreading only for problem sizes above a certain threshold.

8.3 Is there a simple implementation for this fix?

Include a check for problem size threshold. If the problem size is too small, default to sequential implementation.

(Answer not required in Questions.rst but you should think about this.) What does this situation look like if we are computing the transpose product? If we partition A by rows and want to take the transpose product against x can we partition x or y? Similarly, if we partition A by columns can we partition x or y?

When we are taking transpose product, we need to access A_T column wise as each column is a row of original A. So if we partition A by columns, we can partition y but each thread would need entire x to operate on. If we partition A by rows, we can partition x but not partition y as we will need entire y.

(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.)

CSR should be partitioned on y. If we partition on x, we will end up with race conditions because entire y will be accessed at various indices by different threads leading to unpredictable outcomes and data races. Because of the same reason CSC^T is partitioned on y.

CSR^T and CSC should be partitioned on x, because if we partition on y entire y will be have to be shared across threads leading to race conditions.

Question 9:

9.1 Which methods did you implement?

CSR and CSC^T. Both are partitioned on y.

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

Given below is the output for a sample run. We look at CSR and CSC^T results as those have been optimized.

1 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	0.953032	0.962196	0.397886	1.02111	1.00068	0.40269
128	16384	81408	0.941734	0.941734	0.959757	1.01822	1.00799	0.937333
256	65536	326656	0.736461	0.570498	0.595667	0.836887	0.784987	0.739149
512	262144	1308672	0.697958	0.739885	0.805337	0.769807	0.82436	0.997083
1024	1048576	5238784	0.721509	0.717214	0.854553	0.743778	0.732474	0.83098
2048	4194304	20963328	0.719771	0.507433	1.01949	0.832291	0.683123	0.471086

2 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	0.840911	0.67843	0.016838	0.889496	0.758094	0.162978
128	16384	81408	0.80558	0.786625	0.808828	0.915933	0.849955	0.849955
256	65536	326656	0.559466	0.725902	0.969027	0.843861	0.806879	1.08303
512	262144	1308672	0.69564	0.693336	1.03657	0.775509	0.769807	1.03147
1024	1048576	5238784	0.688526	0.639215	1.0663	0.762608	0.774868	1.02112
2048	4194304	20963328	0.645025	0.727578	1.21088	0.822091	0.834361	1.14086

4 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	0.922289	0.953032	0.211785	0.990776	0.990776	0.203184
128	16384	81408	0.924375	0.937333	0.724149	0.997957	0.993016	0.713841
256	65536	326656	0.661852	0.649124	1.15072	0.810107	0.79112	1.17067
512	262144	1308672	0.69564	0.654336	1.02641	0.775509	0.778392	1.10204
1024	1048576	5238784	0.704632	0.499967	1.0041	0.593557	0.625933	0.688526
2048	4194304	20963328	0.592603	0.695878	1.1979	0.783676	0.824111	1.09612

8 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	0.926559	0.953032	0.0716053	0.981062	1.00068	0.068306
128	16384	81408	0.91177	0.895488	0.228462	0.993016	0.978484	0.228983
256	65536	326656	0.675089	0.67962	0.578648	0.800501	0.715642	0.540071
512	262144	1308672	0.688775	0.714633	0.778392	0.753193	0.761409	1.00185

1024	1048576	5238784	0.762608	0.719355	1.27505	0.78753	0.825288	1.14754
2048	4194304	20963328	0.733946	0.680351	1.16463	0.798603	0.824111	1.29503

CSR does poorly for smaller problem sizes compared to the 1 thread approach but the efficiency gets better with larger problem sizes 256 and beyond for 2,4 threads and 512 and beyond for 8 threads.

For largest problem size N=2048 and CSR, 1.18x speedup (2 threads), 1.17x speedup (4 threads) and 1.14x speedup (8 threads). Overall, 4 threads tends to perform well for problem sizes 256 and beyond.

CSC^T also does poorly for smaller problem sizes compared to the 1 thread approach but the efficient gets better with problem sizes greater than and equal to 256 for 2,4 threads and greater than and equal to 512 for 8 threads. For largest problem size N=2048 and CSC^T, 2.42x speedup (2 threads), 2.32x speedup (4 threads) and 2.75x speedup (8 threads). Overall, 2 threads tends to perform well for problem sizes 256 and beyond for CSC^T.

Question 10:

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

The data is read as CSRMatrix. The mult method computes y=x*P.

Y=x*P can be written as y=P_T*x. To avoid race conditions, it is easier to partition on y otherwise we would need to do some synchronization on y which will be overhead. We can use CSR^T (CSR t_matvec) but that will lead to race conditions on y requiring synchronization. If we use CSR, there won't be race conditions but order of inputs need to change to y=P*x.

We could use CSC^T (t_matvec) which involves reading the data as CSC. This won't give us race conditions if we partition on y.

10.2 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?

To use CSR (first option), the data would need to be transposed to calculate y=P_T*x.

10.3 How would we use the second?

Read data as CSCMatrix. This is implemented.