AMATH 583 - Problem Set #5
Written By: EJ Rainville, Spring 2021

Question 1

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

The norm computed with reversed values begins to differ from the other norms with a problem size of approximately 85. The norm computed with sorted values begins to differ from the other norm at a problem size of approximately 19000.

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

The relative difference stays approximately the same as the problem size increases while the absolute difference increases with increasing problem size.

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

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?

Yes, I am concerned that as we get to larger problems errors can become significant even though they are small relative to the problem. This can be important if you are computing a large problem with small values or many orders of magnitude difference between values.

Section 1 - pnorm

What was the data race?

The data race that was occurring is a low-level data race due to multiple threads trying to update a shared variable at almost the same time. The variable "partial" in the function worker_a is shared and therefore as multiple threads try to access it with no protection there is a chance that each thread could load the current value onto the thread stack then update the value and write it back out without "knowing" that the other thread had also operated on the original value thus the overall shared value would not be updated for each operation and the final value would be incorrect. The paper titled "High-level Data Races" written by Artho et al. was used as a supplement for understanding data races, article can be found at this link http://www.runtime-verification.org/course/resources/lecture9/hldataraces.pdf.

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

In order to fix this data race, I used a lock guard within the worker_a function. By using a lock_guard to solve the problem it guarantees that a data race will not occur because only one thread can access the shared value "partial" at a specific time. The thread that accesses the shared value will update the shared value prior to another thread being able to access it which guarantees that the data race problem is eliminated.

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

The values from the pnorm.exe test are shown in the following table.

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	5.80935	5.15728	8.15183	10.8691	9.71949	0	0	1.92318e-16	0
2097152	4.94401	4.78229	8.45794	11.2463	10.8873	0	5.43881e-16	4.07911e-16	9.51792e-16
4194304	4.78802	4.53929	8.06597	10.8101	10.5917	0	9.6159e-16	1.92318e-16	5.76954e-16
8388608	4.68034	4.58201	8.1382	11.1277	11.4791	0	1.3595e-16	6.79751e-16	0
16777216	3.65858	3.07436	5.26639	5.90153	7.07473	0	3.84567e-16	1.34599e-15	3.84567e-16
33554432	3.82387	3.73866	4.64421	4.70939	4.03056	0	7.20652e-15	7.61444e-15	6.79861e-15

Here, we can see that the data race is eliminated with the relative error being below machine precision for all problem sizes. We also see that there is significant speedup for the multiple threads; however, the speedup is not proportional for each thread since these threads are running concurrently rather than directly in parallel. We see that four threads in general gives the highest performance since it is most likely the "sweet spot" that allows for more concurrent computation but does not break the job into too many parts that there is time spent organizing the threads rather than computing on them which is most likelly the case for 8 threads since we see a reduction in performance. For a problem size of 1048576, we see a speed up for 1, 2, 4 and 8 threads of 0.8877, 1.403, 1.8709 and 1.673 respectively.

Section 2 - fnorm

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

The performance values from the 'partitioned_two_norm_a' function are shown in the following table.

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	3.85812	4.76805	7.95927	8.71403	8.21811	0	0	1.92318e-16	0
2097152	5.22148	4.94401	8.60009	11.499	11.499	0	5.43881e-16	4.07911e-16	9.51792e-16
4194304	4.89989	4.72332	8.2565	10.9227	11.275	0	9.6159e-16	1.92318e-16	5.76954e-16
8388608	4.76209	4.60135	8.26151	11.3596	11.8535	0	1.3595e-16	5.43801e-16	0
16777216	4.1943	4.12072	6.82794	7.67585	8.63533	0	3.84567e-16	1.34599e-15	3.84567e-16
33554432	4.18124	4.10452	5.02688	4.37191	3.64722	0	7.20652e-15	7.61444e-15	6.66264e-15

Here we see that 4 and 8 threads are very similar in their performance and generally have the best performance except for very large problems where 2 threads has higher performance. The speedup values for the largest problem size of 33554432 for 1, 2, 4 and 8 threads are 0.981, 1.202, 1.0545, and 0.8722 respectively so there was not much speedup from multithreading

this problem for large problems. For the smallest problem of 1048576 we see speedup values for 1, 2, 4 and 8 threads as 1.235, 2.063, 2.258, and 2.13 respectively. Which shows for these smaller problems, partitioning the computations into multiple threads and concurrently computing them leads to significant speedup.

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

The performance values for the 'partitioned_two_norm_b' function are shown in the table below.

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	5.07953	5.26473	5.26473	4.65819	4.97944	0	1.15258e-15	1.34467e-15	1.34467e-15
2097152	4.60996	4.71618	4.67311	4.50841	4.50841	0	2.71928e-15	2.03946e-15	2.03946e-15
4194304	4.35094	4.35094	4.36907	4.33296	4.2625	0	9.61294e-16	3.84518e-16	0
8388608	4.29338	4.2105	4.13075	4.13075	4.24326	0	1.49582e-15	1.08787e-15	5.43933e-16
16777216	4.14984	4.03576	4.04967	3.45413	2.46724	0	5.00099e-15	4.03926e-15	3.84691e-15
33554432	3.56962	3.8022	3.91305	3.89037	3.82387	0	3.67179e-15	4.62374e-15	5.57568e-15

Here we don't see nearly the same kind of performance that we see for the function above and we see very consistent performance values for each number of threads for any problem size. The speedup values for the smallest problem size for 1, 2, 4 and 8 threads are 1.03, 1.03, 0.918, and 0.98 which indicates that using any number of threads doesn't lead to any significant speedup for this algorithm. We see similar values for speedup for the largest problem as well.

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

The difference between the two algorithms is the asynchronous launching policy where 'partitioned_two_norm_a' has a policy of 'std::launch::async' and 'partitioned_two_norm_b' has a policy of 'std::launch::deferred.' The policy for the 'partitioned_two_norm_a' suggests that the asynchronous tasks are launched as soon as they are created so there are tasks being evaluated and launched while the rest of the tasks are being created and this concurrent computation results in the speedup we see above. In 'partitioned_two_norm_b' we are using a deferred launch policy meaning that the tasks are not launched until the result is asked for which means that no concurrent computation is occurring while the tasks are being created but they are all concurrently computing once they are asked for which floods the CPU essentially the same as the sequential run which is why the performance is effectively the same as the sequential run.

Section 3 - cnorm

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

The performance values for thee cyclic norm are shown in the following table.

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	4.24717	1.59436	2.80785	3.7438	3.50982	1.65393e-14	1.00005e-14	9.6159e-16	2.30782e-15
2097152	5.30264	1.35193	2.85869	3.66814	3.21827	8.15822e-15	8.7021e-15	4.21508e-15	4.89493e-15
4194304	4.92289	1.31565	2.92082	3.86928	2.99593	3.82713e-14	3.07709e-15	2.69245e-15	2.30782e-15
8388608	4.78298	1.1336	3.02922	4.00926	2.78194	1.41388e-14	1.22355e-15	2.1752e-15	7.34131e-15
16777216	4.27056	1.68979	3.05835	4.0778	2.66305	4.74941e-14	1.17293e-14	2.26895e-14	3.84567e-15
33554432	4.18124	1.70327	3.08547	4.0672	2.34646	1.61807e-14	2.937e-14	1.31893e-14	7.47847e-15

Here we see reduced performance using multiple threads compared to the sequential run. The speedup values for the smallest problem size for 1, 2, 4, and 8 threads are 0.375, 0.6609, 0.8815 and 0.826 respectively. Therefore, we are not gaining any speedup and actually losing performance by using this cyclic multithreaded formulation.

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

The performance of cyclic partitioning is significantly reduced compared to blocked partitioning as well as sequential runs. In order to understand this difference, we must go back to our models of hierarchical memory and memory storage. The benefit of blocked partitioning is that within each block the data are stored next to each other therefore the data can be quickly pipelined into the CPU within each task. However, with cyclic partitioning the data is not stored directly next to each other and are separated by the stride therefore more time must be spent accessing the data which reduces the performance as we see in the performance measurements above.

Section 4 - General

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

Between all the different approaches to parallelization, we see that the tasks method using an asynchronous launch method and a recursive algorithm gives the highest performance for small problems. In general for very large problems we did not gain much speedup for multiple threads or tasks compared to the sequential runs which is most likely due to time being spent accessing data from memory since the large problems do not fit in cache and therefore reducing the locality and performance. The cyclic partitioning gave the overall lowest performance since the data in each partition was not stored next to it and therefore at each step more time needed to be taken to access the data which reduced the performance.

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.)

In order to better understand this problem we must think back to our roofline model and hierarchical memory model. The smaller problems are faster since all of the data for computation can fit within one of the caches which is located closest to the CPU and therefore can be accessed more quickly which improves performance. We see a reduction in performance in all of the algorithms for large problem sizes which is due to the data not fitting within cache and therefore the performance improvements from the multi-threading are lost to the time that it takes for the data to be accessed from DRAM rather than cache. The size of the caches on this machine are limiting the parallel speedup for two_norm and determine where we should see ideal speedup which would be problems smaller than the size of the L1 cache.

Section 6 - Conundrum #1

1. **What is causing this behavior?**

When running the block partitioned norm function on smaller problem sizes of 128 and 256 we see that the performance is drastically reduced for the multi-threaded algorithm but is much faster for the sequential run. The performance values are shown in the following table.

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
128	14.0845	0.0105298	0.00730616	0.00399032	0.00217016	0	0	0	0
256	26.3159	0.0211807	0.0153037	0.00869507	0.00434946	0	0	1.94081e-16	1.94081e-16

Here we see that the performance of the sequential run is significantly faster than the performance of the multithreaded runs for both sizes. Thinking back to the hierarchical memory model that we have, we can understand why the sequential run would have improved performance. Since these are relatively small problems, we expect that the data can fit into the cache and therefore data access time can be reduced thus improving the overall performance. The reduced performance in the threads however is most likely due to the very small threads having low priority to be submitted in the CPU. Since for small problems that are multithreaded each have very small threads therefore they will have low priority to be scheduled and so they sit idle while other tasks on the computer (background updates/ jobs) are submitted prior to the small threads. Information to answer this question was found in lecture notes on CPU scheduling from UC Davis at the following link https://web.cs.ucdavis.edu/~pandey/Teaching/ECS150/Lects/05scheduling.pdf.

2**. How could this behavior be fixed?**

This behavior can be fixed by setting multithreading only for problems of a sufficiently large size so that the threads will require enough computing power to need to be submitted quickly and therefore the problem can be solved with a high performance.

3. **Is there a simple implementation for this fix?**
This size limit could be implemented as a loop within the norm function so that multithreading will only occur for problems over a certain size otherwise they are computed sequentially.

Parallel matvec

Which methods did you implement?

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

Conundrum #2

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

The two matrix vector operations that we could use are CSR format with a x partitioned multiply or we could use a CSC format with a y partitioned multiply.

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?

3. How would we use the second?