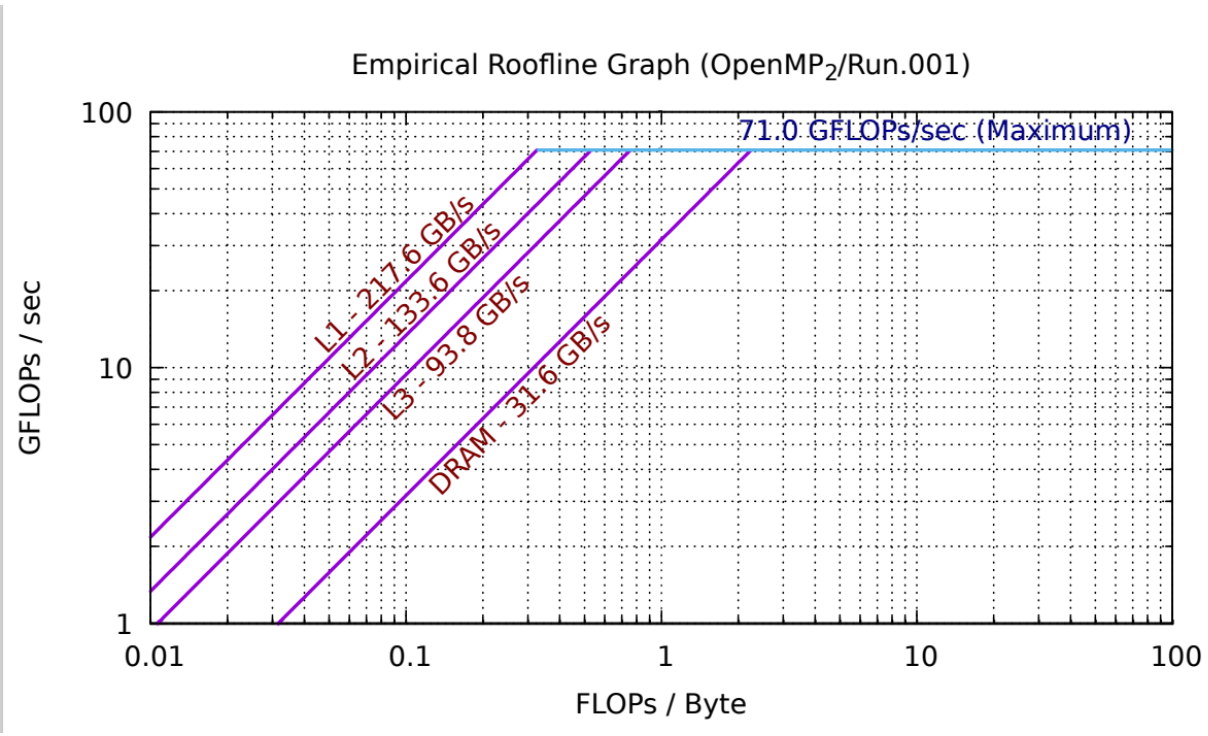
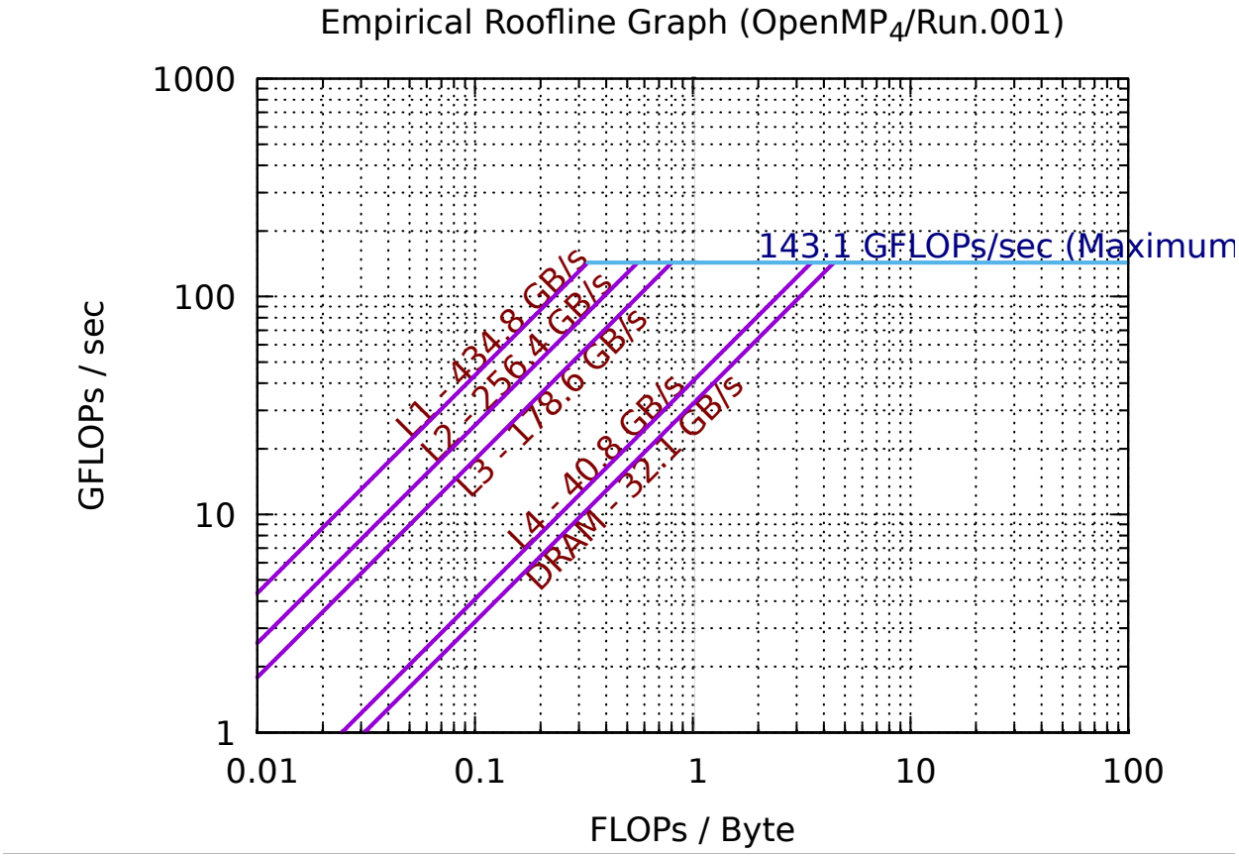
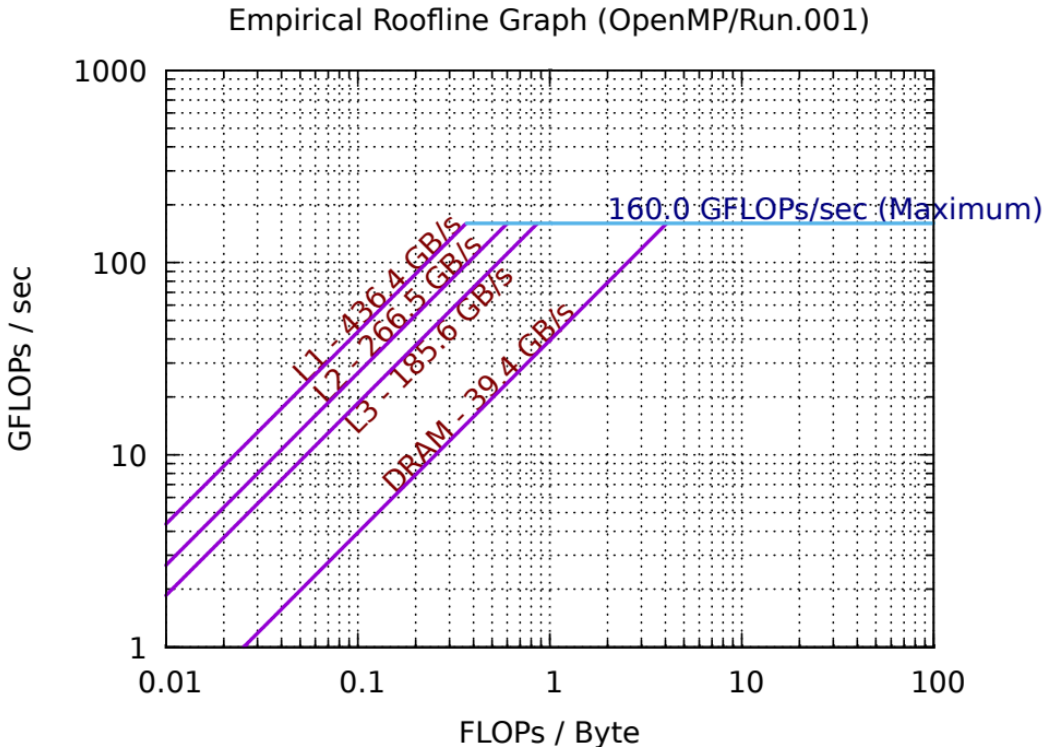

Empirical Roofline Graph (OpenMP2/Run.001)
71.0 GFLOPs/sec (Maximum)
L1 - 217.6 GB/s
L2 - 133.6 GB/s
L3 - 93.8 GB/s
DRAM - 31.6 GB/s
GFLOPs / sec
FLOPs / Byte
100
10
1
0.01
0.1
1
10
100


Empirical Roofline Graph (OpenMP4/Run.001)
143.1 GFLOPs/sec (Maximum
L1 - 434.8 GB/s
L2 - 256.4 GB/s
L3 - 178.6 GB/s
L4 - 40.8 GB/s
DRAM - 32.1 GB/s
GFLOPs / sec
FLOPs / Byte
1000
100
10
1
0.01
0.1
1
10
100



norm
---

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

We tried the following size- 2, 2^2, 2^3, 2^4.... ,2^20. The norms differed almost always except for
2^6, 2^7, 2^8, 2^9, 2^10. But the error usually increased with problem size.

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

Both absolute and relative error increase with problem size but absolute error grows at a faster rate.
Between 2 to 2^20, the absolute error goes from 1.11e-16 to 1.159e-12, while the relative error goes from
1.16e-16 to 1.34e-15.

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

No. The vector class does not strictly behave like algebraic vector space.
The order in which elements are added should not affect norm calculation.

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

Yes, with computation pertaining to very large vectors and matrices this behavior could lead to significant errors. For 1 Billion element vector, the abolute error is ~1.67e-10 and relative error is 4.58e-15.
Whether this error is serious would depend on the sensitivity of application.

pnorm
---

* What was the data race?

In the original code each thread modified the original variable 'partial' independently without any oversight. The race could be caused in the following way:
Suppose thread 1 writes to 'partial' as follows: 'partial = partial+partial_T1', then the next operation (say by thread 2) should be: 'partial = partial+partial_T1+partial_T2'. But if Thread 2 accessed 'partial' before Thread 1 could write to it, it would update the following: 'partial = partial+partial_T2', i.e. the information that Thread 1 had already updated 'partial' would be lost, which would result in error.

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

We used mutex to lock 'partial', i.e. it is accessed by a thread, no other thread can access it. Other threads can only access (and update) 'partial', after the previous thread that accessed it has finished the update step.
We also used a local variable 'partial_i' to store the sum before updating partial, so that a single thread cannot keep the lock for a lonf time. This helps in better performance.

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

The performance in GFLops/s is shown below. We see that as problem size increases, the sequential approach drops in performance, while

multi-threading helps in achieving and maintaining high performance.
There isn't much speed up going from 4 threads to 8 threads. This could also be attributed to there being 6 cores on the cpu.
Going from 1 thread to 2 threads to 4 threads, there is almost 2x speed up, especially for larger problem sizes.

N	Sequential	1 thread	2 threads	4 threads	8 threads
1048576	8.71403	4.05955	5.554	6.92347	3.91794
2097152	5.78198	3.45747	4.71618	8.97728	9.56458
4194304	3.42672	2.76669	4.48109	7.54371	8.1285
8388608	3.22639	2.59647	5.34568	8.72415	8.32457
16777216	3.28965	2.69978	5.01883	9.47101	8.96492
33554432	3.25771	2.728	4.72598	9.51899	9.38586

fnorm
---

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

The performance is shown below. We again see 2x performance improvement going from 1 thread to 2 threads to 4 threads, as well
as higher performance than sequential , as the problem size increases.
Using 8 threads does not lead to gain over 4 Threads.

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

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

The performance is shown below. We lose all the benefits of multi-threading using deferred launch policy and there is no difference between the sequenctial
and parallel implementation in terms of performance.

N	Sequential	1 thread	2 threads	4 threads	8 threads
1048576	10.0082	8.42356	10.0082	9.91007	9.71949
2097152	5.16874	5.56201	5.6542	5.44367	5.68561
4194304	3.32881	3.26659	3.25645	3.25645	3.24637
8388608	3.32475	3.27483	3.27483	3.2847	3.23596
16777216	3.28046	3.23528	3.2532	3.23528	3.24421
33554432	3.33046	3.24197	3.24197	3.24982	3.21865

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

In ``partitioned_two_norm_a`` each thread runs right away and computes its partial sum while in ``partitioned_two_norm_b`` runs only when partial sums are being accumulated using .get(), for the total sum.

cnorm
---

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

The performance is shown below. There is 2x improvement going from 1 thread to 2 threads to 4 threads and no benefit of using 8 threads. However, parallel implementaion is still slower than the sequential version. Performance of 1 Thread is much worse(3x) than sequential.

N	Sequential	1 thread	2 threads	4 threads	8 threads
1048576	8.56633	0.942057	1.72791	3.75772	3.09121
2097152	5.71738	0.963663	1.854	2.85869	2.88285
4194304	3.40447	0.871634	1.87917	2.68866	2.67494
8388608	3.31465	0.959964	1.99364	2.80339	2.66631
16777216	3.30818	1.10169	1.53517	2.57545	2.6691
33554432	3.2736	1.10467	1.82361	2.63172	2.66305

* 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 loses all the benefits of multi-threading and perform 3-4x worse than 4 thread blocked algorithm. Since, load operations have a huge overhead, the contiguog storage of vector elements in the blocked partitioning aids makes
it much faster than interleaved loading. This could significantly slow down each thread, especilly with increasing number of threads.

rnorm
---

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

The performance is shown below. There is almost no benefit of increasing the number of threads. The performance drops with increasing number of threads.
This because the recursive algorithm can be parallelized only at the base level. All the higher level computations are sequential.

N	Sequential	1 thread	2 threads	4 threads	8 threads
1048576	8.15183	3.15884	2.10589	1.81152	0.971016
2097152	5.33026	3.0733	1.99885	1.60914	1.22125
4194304	3.47211	2.44994	2.30456	2.01649	1.66177
8388608	3.24559	2.54794	2.40202	1.95434	1.592
16777216	3.23528	2.6273	2.706	2.42646	2.1237
33554432	3.34708	2.71696	2.66834	2.61633	2.45371

* 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.

The performance for deferred version is shown below. There is very little difference between asyn and deferred. This is because any base level thread computation cannot be used by the higher level until all the base level threads are completed.

N	Sequential	1 thread	2 threads	4 threads	8 threads
1048576	9.44698	3.71628	2.66709	1.4338	1.04641
2097152	5.50221	2.36353	2.40237	1.61421	1.27132
4194304	3.1775	2.52669	2.15313	2.01262	1.68041
8388608	3.32475	2.69264	2.42877	2.09715	1.7732
16777216	3.28046	2.66305	2.542	2.41151	1.87905
33554432	3.30586	2.62657	2.5663	2.47178	2.43148

Divide and Conquer + Tasks(AMATH583 Only)

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

Same as the rnorm question above.

General
---

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

The performance p_norm and f_norm was the best (with 4 threads). This was because of structure of the algorithms which allowed parallelization, as well as locality in space for loading operations. cnorm suffered due to non-contiguous access of vector elements, while rnorm has very little scope for parallelization.

* 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.)

There are 2 doubles (16 bytes) and 2 flops in each loop for the norm calculation. Tus the numeric intensity of the algorithm is 1/8 = 0.125. Since, the variable x is shared, The peak performance is limited by L3 cache. For 4 threads as well 8 threads, this should be in the range of 20 GFlops/s from the roofline plots. For very large matrices, exceeding the L3 cache size, DRAM would determine the performance which roughly amounts to 10 GFlops/sec.

Conundrum #1
---

1. What is causing this behavior?

2. How could this behavior be fixed?

3. Is there a simple implementation for this fix?

Parallel matvec
---

* Which methods did you implement?

We partitioned rows (for CSR) and columns (for CSC) and assigned one block to each thread. We implemented the algorithm using std::futures and std::async.

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

Even though the parallel versions passed the test cases, we don't see any speedup due to parallelization. With more number of thread the performance degrades. The performance is an order of magnitude worse than the COO implementations.

Conundrum #2
---

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

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?