AMATH 583 PS6
Rohan Tiwari

1 Norm

1.1 Look through the code for run() in norm_utils.hpp. How are we setting the number of threads for OpenMP to use?

In the run() method, there is a for loop which sets number of threads for OpenMP to use iteratively using the following call to omp_set_num_threads(nthreads).

```
#ifdef _OPENMP
      omp_set_num_threads(nthreads);
#endif
```

It overrides the setting of OMP_NUM_THREADS environment variable and sets the number of thread to use.

Inside the for loop, Number of threads is set to 1 initially and then incremented as power of 2 until it reaches the number of parts we want to split the problem into. The number of parts is the max number of threads (default value 8) that we want to use and is the command line argument to the calling driver program.

1.2 Which version of norm provides the best parallel performance? How do the results compare to the parallelized versions of norm from ps5?

Norm_Block_reduction performs the best. Overall, Norm_parfor and Norm_Block_reduction is faster than the parallelized versions in ps5 but the gap in performance reduces for larger problem sizes.

Norm_block_critical:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	1.97173	1.98536	2.02943	2.01318	1.98145	0	0	1.78938e-14	3.84812e-16
2097152	2.01561	2.06081	1.83121	1.85133	1.92172	0	1.53687e-14	8.43241e-15	1.27846e-14
4194304	2.10578	2.01657	1.76908	1.9475	1.83243	0	0	7.11596e-15	3.46182e-14
8388608	2.09925	1.96363	1.96363	1.96363	1.80168	0	8.92042e-14	7.24784e-14	6.36396e-14
16777216	1.39721	1.80251	1.72414	1.95784	1.94043	0	1.21165e-13	1.73093e-15	1.17318e-14
33554432	1.94599	2.09715	2.11034	1.98715	1.95084	0	0	3.45886e-13	4.28415e-13

Norm_block_reduction:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	1.95639	1.98732	3.14245	5.37746	9.95628	0	3.54027e-14	2.52052e-14	2.53976e-14
2097152	2.1303	2.08418	3.38636	3.57815	3.63607	0	1.29206e-14	6.80033e-15	1.04725e-14
4194304	2.14777	2.09501	3.38317	3.62269	3.36095	0	3.13487e-14	3.94262e-14	2.73099e-14
8388608	2.0011	2.09088	3.23635	3.54848	2.83782	0	6.48634e-14	5.98321e-14	6.21438e-14

16777216	1.9596	2.00279	2.91973	3.32983	3.42931	0	2.28867e-14	1.92325e-14	1.23088e-14
33554432	1.8222	1.93636	3.14012	3.43896	3.41894	0	3.39496e-13	3.29163e-13	3.22365e-13

Norm_cyclic_critical:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.0691	2.03766	1.18513	0.607971	0.310222	0	1.77013e-14	2.30887e-15	4.09825e-14
2097152	1.96468	1.95518	1.20408	0.640778	0.287494	0	7.20835e-15	1.25126e-14	1.83609e-14
4194304	1.83736	2.02655	1.05073	0.566515	0.280502	0	3.84646e-16	5.36582e-14	5.09656e-14
8388608	1.93108	1.95813	0.940427	0.600215	0.274928	0	4.78657e-14	2.37969e-14	1.00627e-14
16777216	1.62643	1.79362	1.16198	0.461986	0.257654	0	6.96217e-14	1.29242e-13	8.40461e-14
33554432	1.97711	1.11635	0.605052	0.442838	0.285084	0	9.44932e-14	2.80081e-13	1.7743e-13

Norm_cyclic_reduction:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	1.9795	1.81186	1.91176	1.70294	1.69006	0	3.15546e-14	2.11647e-14	2.82837e-14
2097152	2.11913	1.92172	1.81477	1.04858	0.790635	0	2.28491e-14	1.10165e-14	1.37367e-14
4194304	1.31544	1.40578	1.91829	1.41063	1.15901	0	4.11572e-14	3.59644e-14	3.05794e-14
8388608	1.79244	1.95996	1.94361	1.65914	1.14038	0	7.547e-14	8.68925e-14	7.99574e-14
16777216	1.72961	1.81451	1.78044	1.43584	1.04057	0	3.73111e-14	9.03928e-15	3.8465e-16
33554432	1.97711	1.96883	1.78346	1.34912	0.669749	0	3.90617e-13	3.17198e-13	3.20597e-13

Norm_parfor:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	1.96212	1.96403	3.2282	4.38163	6.20731	0	3.54027e-14	2.52052e-14	2.53976e-14
2097152	1.36138	1.74582	2.92569	3.50373	3.49767	0	1.29206e-14	6.80033e-15	1.04725e-14
4194304	2.09715	2.08222	2.96211	3.38317	3.45747	0	3.13487e-14	3.94262e-14	2.73099e-14
8388608	1.92047	2.08051	3.16313	3.68568	3.22639	0	6.48634e-14	5.98321e-14	6.21438e-14
16777216	2.08712	2.0693	3.22639	3.79311	3.45648	0	2.28867e-14	1.92325e-14	1.23088e-14
33554432	2.14699	2.1009	3.26677	3.67002	3.7106	0	3.39496e-13	3.29163e-13	3.22365e-13

Norm_seq:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.00916	1.94504	1.58986	1.64311	1.92641	0	0	0	0
2097152	1.90722	1.97042	2.08418	2.04621	2.08633	0	0	0	0

4194304	2.06125	2.12106	1.9776	2.06333	2.05298	0	0	0	0
8388608	2.02038	2.06007	1.99539	2.04401	2.01262	0	0	0	0
16777216	1.99729	1.99729	2.04026	1.98096	1.95434	0	0	0	0
33554432	1.90496	1.9065	2.00753	2.05316	1.86414	0	0	0	0

1.3 Which version of norm provides the best parallel performance for larger problems (i.e., problems at the top end of the default sizes in the drivers or larger)? How do the results compare to the parallelized versions of norm from ps5?

Norm_parfor seems to provide parallel performance for larger problems. Comparing with PS5, we see block partitioning with tasks does best and close but norm_parfor is still slightly better over multiple runs compared to block partitioning with tasks.

1.4 Which version of norm provides the best parallel performance for small problems (i.e., problems smller than the low end of the default sizes in the drivers)? How do the results compare to the parallelized versions of norm from ps5?

Norm_block_reduction seems to provide parallel performance for smaller problems. Comparing with PS5, we see it still does better – almost 2x-3x faster.

2 Sparse Matrix-Vector Product

2.1 How does pmatvec.cpp set the number of OpenMP threads to use?

In the main() method, there is a for loop which sets number of threads for OpenMP to use iteratively using the following call to omp_set_num_threads(nthreads).

```
#ifdef _OPENMP
    omp_set_num_threads(nthreads);
#endif
```

It overrides the setting of OMP_NUM_THREADS environment variable and sets the number of thread to use.

Inside the for loop, Number of threads is set to 1 initially and then incremented as power of 2 until it reaches the max number of threads. The max number of threads is defaulted to 8 but is also a command line argument to the calling driver program.

2.2 (For discussion on Piazza. You can discuss this question on Piazza but you have to answer this question independantly in your submission.) What characteristics of a matrix would make it more or

less likely to exhibit an error if improperly parallelized? Meaning, if, say, you parallelized CSCMatrix::matvec with just basic columnwise partitioning – there would be potential races with the same locations in y being read and written by multiple threads. But what characteristics of the matrix give rise to that kind of problem? Are there ways to maybe work around / fix that if we knew some things in advance about the (sparse) matrix?

If we partition CSRMatrix::matvec on x we end up in race conditions on y because multiple threads will potentially modify the same index in y. The race condition will not happen if the inner loop runs just once i.e. only one update is to be made to y(i). This implies that each row will have only 1 nonzero element e.g. if it is a diagonal matrix with nonzero elements only on the diagonal. In the other case of multiple nonzero elements per row there will be a race condition as there will be multiple writes and reads from y(i). Therefore, if we knew this characteristic about the matrix we could partition on x. In the same way we can argue about CSCMatrix::matvec.

2.3 Which methods did you parallelize? What directives did you use? How much parallel speedup did you see for 1, 2, 4, and 8 threads?

CSR matvec and CSC t_matvec were parallelized.

Directive added was "#pragma omp parallel for" for both implementations.

4 threads is the best performing implementation. There is better speed up observed for smaller problem sizes of 2-3x for CSR and CSC^T and it drops down to about 1.5x for 4 threads. 8 threads tend to do better than 2 threads on smaller problem sizes but not so much on larger problem sizes.

1 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	0.922289	0.944041	2.56586	0.971537	1.00068	2.59918
128	16384	81408	0.907644	0.946176	2.7478	1.00799	1.00799	2.63933
256	65536	326656	0.770064	0.79112	1.22743	0.87296	0.850953	1.23492
512	262144	1308672	0.772648	0.793135	1.21033	0.879779	0.876098	1.22449
1024	1048576	5238784	0.748398	0.700535	0.912818	0.656632	0.757812	1.11567
2048	4194304	20963328	0.506667	0.619988	0.758854	0.636458	0.672171	0.724435

2 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	0.930868	0.909712	4.44748	0.930868	0.971537	3.77616
128	16384	81408	0.82547	0.588238	3.58195	0.924375	0.891508	4.09366
256	65536	326656	0.611863	0.664022	1.2425	0.61002	0.64705	1.20552
512	262144	1308672	0.702643	0.784223	1.28459	0.827619	0.865238	1.15684
1024	1048576	5238784	0.732474	0.743778	1.28868	0.83675	0.800612	1.10543
2048	4194304	20963328	0.772842	0.767536	1.26571	0.84487	0.783676	1.34165

4 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	0.935218	0.944041	7.14774	1.00068	1.00571	6.90127
128	16384	81408	0.915933	0.928654	8.02357	0.988125	0.997957	8.72127
256	65536	326656	0.703218	0.611863	2.97833	0.794222	0.869213	7.78949
512	262144	1308672	0.644269	0.79615	1.30867	0.876098	0.632591	1.04694
1024	1048576	5238784	0.725856	0.774868	1.24862	0.83098	0.816895	1.19299
2048	4194304	20963328	0.680351	0.771065	0.901649	0.595761	0.589478	1.18521

8 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	0.840911	0.893467	5.71819	0.962196	0.918058	6.25427
128	16384	81408	0.822087	0.937333	7.1639	0.937333	0.988125	8.72127
256	65536	326656	0.686531	0.655426	4.05053	0.705668	0.651211	4.70992
512	262144	1308672	0.610459	0.594851	1.11376	0.73728	0.81158	1.1965
1024	1048576	5238784	0.736954	0.635842	1.21098	0.777368	0.857595	1.40107
2048	4194304	20963328	0.687322	0.774626	1.34165	0.828181	0.789208	1.31021

3 583 only – Sparse Matrix Dense Matrix Product

3.1 Which methods did you parallelize? What directives did you use? How much parallel speedup did you see for 1, 2, 4, and 8 threads? How does the parallel speedup compare to sparse matrix by vector product?

CSR matmat and CSC matmat are parallelized.

The "#pragma omp parallel for" directive is added.

Tried changing order but the loop ordering ijk seems to perform best for parallel for.

If we compare CSR we see 1.1x, 1.43x and 1.43x for speed up for 2,4 and 8 threads compared to 1 thread for N=2048 which is the largest matrix size. If we compare CSC we see 3.56x, 4.43x and 5.13x speedup for 2,4 and 8 threads compared to 1 thread for N=2048 which is the largest matrix size. 8 threads generally tends to do well for CSC compared to CSR.

For smaller problem sizes in CSR we see speedups of 1.78x, 2.13x and 1.94x for 2,4 and 8 threads compared to 1 thread for N=64. For smaller problem sizes in CSC we see speedups of 1.75x, 3.375x and 4.04x for 2,4 and 8 threads compared to 1 thread for N=64.

Compared to matvec, CSR speedups seem to follow a similar pattern but CSC speed ups are higher.

1 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	1	0.644603	1.51889	0.641474

128	16384	81408	1	0.610748	0.98172	0.640252
256	65536	326656	1	0.600814	1.12152	0.598143
512	262144	1308672	1	0.596357	1.07891	0.614507
1024	1048576	5238784	1	0.574113	1.07462	0.603025
2048	4194304	20963328	1	0.591707	0.922914	0.233854

2 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	1	0.629255	2.69681	1.12943
128	16384	81408	1	0.578743	2.94516	1.15245
256	65536	326656	1	0.577606	1.51216	1.02735
512	262144	1308672	1	0.471122	1.1877	0.851425
1024	1048576	5238784	1	0.476253	1.23266	1.01601
2048	4194304	20963328	1	0.582315	1.02978	0.829058

4 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	1	0.626273	3.22302	2.16629
128	16384	81408	1	0.613575	4.9086	2.32513
256	65536	326656	1	0.465683	2.0705	1.28174
512	262144	1308672	1	0.513951	1.03924	0.936004
1024	1048576	5238784	1	0.590286	1.0222	1.01601
2048	4194304	20963328	1	0.595307	1.32201	1.02978

8 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	1	0.635306	2.93652	2.59105
128	16384	81408	1	0.543165	4.57008	1.92076
256	65536	326656	1	0.492975	5.8514	3.84521
512	262144	1308672	1	0.565346	1.07891	1.13069
1024	1048576	5238784	1	0.484512	1.38546	1.22366
2048	4194304	20963328	1	0.57773	1.31608	1.1882

4 Page Rank Reprise

4.1 Describe any changes you made to pagerank.cpp to get parallel speedup. How much parallel speedup did you get for 1, 2, 4, and 8 threads?

Changed it to use CSCMatrix instead of CSRMatrix. This now uses the t_matvec of CSCMatrix which is partitioned on vector y to avoid race conditions. t_matvec of CSCMatrix has the #pragma omp parallel for directive.

File web-Google.mtx was used in the test.

Here is the result from a sample run:

Unaccelerated time: 3335ms

Accelerated (with CSC):

1 thread: 2651ms

2 thread: 2171ms

4 thread: 1778ms

8 thread: 1866ms

1 thread gives 1.25x, 2 thread gives 1.53x, 4 thread gives 1.88x and 8 thread gives 1.78x speedup. This was the pattern over multiple runs. 4 threads gives best performance.

4.2 Which functions did you parallelize? How much additional speedup did you achieve?

Added "#pragma omp parallel reduction (+:sum)" to two_norm in amath583.cpp.

File web-Google.mtx was used in the test.

Here is the result from a sample run:

Unaccelerated time: 3335ms

Accelerated (with CSC from Question 9):

1 thread: 2651ms

2 thread: 2171ms

4 thread: 1778ms

8 thread: 1866ms

Accelerated (with parallelism to two_norm):

1 thread: 2675ms

2 thread: 1870ms

4 thread: 1773ms

8 thread: 1830ms

There is an additional speedup observed for 2 thread case which is 1.17x of what was seen in question 9. Other options do not yield much speedup.

5 Load Balanced Partitioning with OpenMP

5.1 What scheduling options did you experiment with? Are there any choices for scheduling that make an improvement in the parallel performance (most importantly, scalability) of pagerank?

For options (Static, dynamic, guided), chunk_sizes 2,4,8 and 16 were experimented with. I tried with different number of threads as well. I observed that when the chunk_sizes are small, adding more threads does not improve performance (sometimes degrades it) so it seems like a balance is required between chunk sizes and number of threads for the best performance so to give enough work to the threads.

File web-Google.mtx was used in the test.

Listed below are some of the top performing times that were observed across threads and scheduling options.

1. Best performance for 8 threads was achieved with omp_set_schedule(omp_sched_dynamic,8); gave the best performance of 1688ms.
2. Best performance for 8 threads was achieved with omp_set_schedule(omp_sched_guided,32) at 1770ms.
3. Best performance for 2 threads was achieved with omp_set_schedule(omp_sched_dynamic,4); gave the best performance of 1840ms.

Generally, dynamic and guided scheduled options performed well with a reasonable chunk size probably making sure that threads have enough work to do. For a given thread count, increasing chunk size for dynamic helps until a certain point but then it degrades performance. Guided option does not seem to have this problem because as per the documentation it adjusts the chunk size along the way.

6 OpenMP SIMD

6.1 Which function did you vectorize with OpenMP? How much speedup were you able to obtain over the non-vectorized (sequential) version?

Norm_Block_Critical is vectorized.

OpenMP with SIMD:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	1.07549	2.47376	4.19868	9.76296	29.576	0.75	0.75	0.75	0.75
2097152	1.70748	2.53658	3.50373	3.81444	3.88033	0.646447	0.646447	0.646447	0.646447
4194304	1.71425	2.57786	3.48099	3.67472	3.88391	0.646447	0.646447	0.646447	0.646447
8388608	1.67237	2.64125	3.51871	3.59101	3.57876	0.646447	0.646447	0.646447	0.646447
16777216	1.00185	2.14458	2.90418	2.91973	3.77996	0.646447	0.646447	0.646447	0.646447
33554432	1.14186	2.58395	3.37473	3.62471	3.87592	0.646447	0.646447	0.646447	0.646447

OpenMP without SIMD:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	1.97173	1.98536	2.02943	2.01318	1.98145	0	0	1.78938e-14	3.84812e-16
2097152	2.01561	2.06081	1.83121	1.85133	1.92172	0	1.53687e-14	8.43241e-15	1.27846e-14
4194304	2.10578	2.01657	1.76908	1.9475	1.83243	0	0	7.11596e-15	3.46182e-14
8388608	2.09925	1.96363	1.96363	1.96363	1.80168	0	8.92042e-14	7.24784e-14	6.36396e-14
16777216	1.39721	1.80251	1.72414	1.95784	1.94043	0	1.21165e-13	1.73093e-15	1.17318e-14
33554432	1.94599	2.09715	2.11034	1.98715	1.95084	0	0	3.45886e-13	4.28415e-13

We see that for smaller problem sizes there is tremendous speedup. E.g. 8 threads is almost 14x faster, 4 threads is almost 4x faster, 2 threads is 2x and 1 threads is 1.5x faster for N=1048576.

When we look at larger problem sizes, we see that the speed up with SIMD is not that high but still good. E.g. 8 threads is 2x faster, 4 threads is 1.8x faster, 2 threads is 2x faster and 1 thread is 1.23x faster. I think generally SIMD is easier to do for block partition methods and not for striding or cycling because those do not do contiguous access.

7 About PS6

Answer the following questions (append to Questions.rst):

7.1 The most important thing I learned from this assignment was…

I learned about various OpenMP directives and when to use what especially combining SIMD directives with other ones.

7.2 One thing I am still not clear on is…

I am not clear on scheduling options fully as in which is the best one under what scenario.