Class: AMATH 583 SPRING 2021 PS6 Name: Hongda Li

Preliminaries

I have done all the setting up for the environment, learned about the virtual memory management and the OS, evironmental variables, and setting them up. I didn't view the OMP video is too boring.

Warm Up

(1)

What happens when you have more threads than hardware concurrency? Are the thread IDs unique?

Yes, they all seems to be having unique IDs. And the number assigned to them are like the indices in an array, they are unique and indexed. Their IDs are probably given by the OS or something cause it looks random.

(2)

What effect does setting the environment variable OMP_NUM_THREADS have on this program?

The usage of the macros "#pragma omp parallel num_threads(nthreads)" will shadows the settings from the environmental variables. The compilation is successful and the macros makes use of a program variable.

(3)

After some fiddling, the use of critical will block the number of threads used when executing that section of code, and the use of critical inside the parallel seems eliminates the race conditions for the code inside of tha block.

5.1 Norm

5.1 Q1

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

This is the code snippt involved in running a callable function:

```
for (size_t nthreads = 1; nthreads <= n_parts; nthreads *= 2) {
    #ifdef _OPENMP
    omp_set_num_threads(nthreads);
    #endif
    t.start();
    for (size_t i = 0; i < ntrials; ++i) {
        norm1 = f(x);
    }
    t.stop();
    ms_times.push_back(t.elapsed());
    norms.push_back(norm1);
    }
```

The forloop calls the test function for different number of threads. Here, a macro is defined, an it tests for the existence of "_OPENMP", if it's there, then a code is compiled, and the code calls on the function 'omp_st_num_threads(nthreads)', and the variable is comming from the for loops itself (a variable that depends on the runtime of the program). And the function 'omp_st_num_threads(nthreads)' is different from where we see in the "nthread" from the omp_hello_world example, where, it's handle by the macros instead of a function involved during the runtime.

5.1 Q2

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

The best overall performance is achieved via block_reduction. This is the fastest. The results can be view in the appendix here: here.
The results from OMP is way better than the results from any of the approaches using "Threads + Multex", or "Async". It's just way faster, in the case of numerical operations.

5.1 Q3

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?

In the case of large problem size, the "parfor" and the "block reduction" produces the best performance. It's doubling the performance compare to what we had for ps5. The results are shown here

5.1 Q4

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?

The "block reduction" produces the best results ("parfor is the second best") for problem of smaller size. Compare to what we had for ps5, the speed is like, hundreds time faster. For small enough inputs, the Gflops/s from the previous assignment could be on the range of $0.4 \sim 0.01$. But here it's like "$16 \sim 13$". The results can be seen here.

5.2 Sparse Matrix Vector Product

5.2 Q5

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

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

The above code is the way where the driver programs uses, it's a macro that insert the code if the omp is defined. And it calls on the function "omp_set_num_threads(nthreads)" where the variable "nthreads" are from the for loop that the snippet of code is in.

5.2 Q7

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

I parallelized all the "mat_vec" and "t_mat_vec" methods for both the "CSR" and "CSC" clases, and then I removed the slow one and run the test again. I tried to use block partition and it didn't improve the speed. The directives I used are just "#pragma omp parallel for", and in the case where there is race conditions, I just used "#pragma omp critical", to block the thread's access to the vector. That means I just used "#pragma omp parallel for" for all the methods, but for CSR's Transposed multplication and CSC direct multplication, I imposed a critical section on the output vector. (Which causes things to be very slow and I decided to remove them by the end).

It seems like even OPENMP is unable to handle the problem with the problem of accessing the whole array. This causes significant slowdown for CSR Transposed and CSC method, where multiple threads have to write to the "y" vector in the critical region.
The results are here. As we can see, for the method that I parallelized (CSR, And CSC Transposed) there is a 50% speedup going from one thread to 8, but there is one significant for the second rows of all the runs for the CSR matrix, and it reaches around 10 Gflops/s for the 8 threads case. (5 times speedup).
Also observe the fact that, when multiple threads are sharing the same moery resources, there are no speed up at all, instead, there is slowdown, by a factor of hundreds.

5.3 Sparse Matrix Dense Matrix Product

5.3 Q8

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?

Due to the fact that some of the parallelization works tremendously slow on: "CSR" Transposed Matmat, and the "CSC" Matmat, I just didn't use parallelization. I tried and it didn't work out and it takes over 30 mins to get the results, so I just terminated and delted all my parallization code on those methods.
The directives I used are "#pragma omp parallel for".
Asymptoptically, as the size increase, the speed approaches a limit, because of the limited memory bandwidth. However, for the 8 thread case, the performance peaked for CSR multplication with one vector on the RHS, at around "8.4" Gflops/s(4 times speed up).
The same is tested with 32 colum s on the RHS, and there is a steady speed up, shown here. For small size of matrix, the speedup from multiple threads can be observed, and it's porportional to the number of threads involved.

5.4 Page Rank Reprise

5.4 Q9

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?

When reading the matrix, I used "read_cscmatrix" instead of "read_csrmatrix". I have done this because we are using the matrix transposed vector multiplication. And in the case of sparse column vector, transposed vector multiplication split the workloads when putting the results onto the output vector y.
There is absolutely no speedup for larger matrices, just like what happened for the matrix vector product experiment. The memory bandwidth is the problem.
There is a consistent, tiny improvement of "2ms" when I tried it on the "email-Enron-adj.mmio" example. I tried it enough and I thnk that is a speed up of about "20%" when 4 threads are introduced instead of 1.

5.4.1 Q10

I look over the methods in "amath583.cpp" and "amath583sparse.cpp", and I didn't see any good functions to parallelize.

5.5 Load Balancing

I read it. It's informative.

5.6 Load Balanced Partitioning with OpemMP

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?

I gained "50ms" speed up, which is around 20% for the "Web-NotreDame.mmio". And I used: " #pragma omp parallel for schedule(dynamic, 1024)" on the CSC Tranposed Multiply.
I tried to use "Dynamic" schedule only, resulting a tremendous slowdown. Which is somewhat expected I think OMP is not good with feed with the threads with very small tasks.

5.7 OMP SIMDs

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

Take a look at this, especially the first row.

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.33857	2.74375	3.32975	13.0595	67.039	2.52052e-14	2.50128e-14	2.559e-14	2.53976e-14
2097152	2.42987	2.63924	3.23983	3.82889	4.09242	7.07234e-15	8.43241e-15	8.56841e-15	8.97643e-15
4194304	2.46012	2.58437	3.30132	3.71474	3.81158	3.00024e-14	3.25026e-14	3.26949e-14	3.28873e-14
8388608	2.47306	2.55128	3.32881	3.59101	3.87644	7.50621e-14	7.5606e-14	7.5606e-14	7.6014e-14
16777216	2.40999	2.66305	3.27483	3.74106	3.94401	2.69255e-15	3.8465e-16	5.3851e-15	4.6158e-15
33554432	2.35352	2.76006	3.36989	3.82542	4.02194	3.3093e-13	3.25628e-13	3.23452e-13	3.23588e-13

Yes, there is significant speedup, way way more than 8 times, in the case of 8 threads. And this is what I did for the "norm_block_critical()" method using SIMDs:

```
double norm_block_critical(const Vector& x)
{
  double sum = 0;

  // Spawn the thread to work on each of the own works first and then.
  // combine the final works in a critical section.
  #pragma omp parallel
  {
    size_t tid       = omp_get_thread_num();
    size_t parts     = omp_get_num_threads();
    size_t blocksize = x.num_rows() / parts;
    size_t begin     = tid * blocksize;
    size_t end       = (tid + 1) * blocksize;
    if (tid == parts - 1)
    {
      end = x.num_rows();
    }
    double PartialSum = 0;

    #pragma omp simd
    for (size_t i = begin; i < end; ++i)
    {
      PartialSum += x(i) * x(i);
    }

    #pragma omp critical
    {
      sum += PartialSum;
    }
  }
  return std::sqrt(sum);
}
```

So I basically let each thread work on the partial sums with SIMDs first, and then combines them togeter. This doesn't work with cyclic norm. I think striding access patterns on memory is a party pooper, it breaks the cache line if we stride for more than 1 bytes, but in this case, we have unit striding access on the memory, which improves the speed a lot.

5.8 Epilogue

The most important thing that I learned from this assignment was, programming in c++ is indeed a pain in the ~~ass~~, without all the setup from the professor I will be pretty much in a swamp, and messing with makefile and vscode and wsl for hours without any actual productivities.

One thing that I am still not clear is, how exactly does the OS manage the virtual memory space? Does this has something to do with the slow down of multiple thread accessing the same resource? Why is that case that, if one thread is working on a particular element of an array, the WHOLE array is block for other

threads instead of just that one element is block to other threads? Why is system doing this? Why that is total braindead to me. So if we want to share the access for one array on different threads, we will have to partition the vector, so that each chunk is shared to one thread at a time?

Like, this is stupid, why are we working with data structure like this in a multiple thread environment, there has to be some kind of parallel data structures, like the, parallel queue, parallel array in C# or something right?

Appendix, Data

5.2 Execution Results

norm, block, critical:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.41727	2.43188	4.21629	5.26484	12.5698	0	3.54027e-14	2.52052e-14	2.53976e-14
2097152	2.41536	2.42696	3.02643	4.04331	5.92861	0	1.29206e-14	6.80033e-15	1.04725e-14
4194304	2.21757	2.33123	4.29103	5.7495	6.33691	0	3.13487e-14	3.94262e-14	2.73099e-14
8388608	2.39949	2.255	2.706	4.85452	4.77711	0	6.48634e-14	5.99681e-14	6.21438e-14
16777216	1.61798	1.89491	1.99364	4.25984	4.092	0	2.28867e-14	1.92325e-14	1.23088e-14
33554432	2.27819	2.28483	3.66429	4.52565	4.94486	0	3.39496e-13	3.29163e-13	3.22365e-13

norm, block, reduction:

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.3914	2.32774	3.47352	6.57245	16.0894	0	3.54027e-14	2.52052e-14	2.53976e-14
2097152	2.48361	2.4096	4.41409	6.60671	7.14366	0	1.29206e-14	6.80033e-15	1.04725e-14
4194304	2.43669	2.46012	3.97441	4.95598	6.24031	0	3.13487e-14	3.94262e-14	2.73099e-14
8388608	2.49364	2.39675	4.58895	6.33581	7.18203	0	6.48634e-14	5.98321e-14	6.21438e-14
16777216	2.43419	2.44511	4.37959	6.71089	6.92393	0	2.28867e-14	1.92325e-14	1.23088e-14
33554432	2.46724	2.44923	4.57858	6.86787	7.24941	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	2.41147	2.26228	2.93601	2.66028	2.8731	0	3.15546e-14	2.11647e-14	2.84761e-14
2097152	2.47449	2.06081	3.52819	3.76472	2.70636	0	2.28491e-14	1.10165e-14	1.37367e-14
4194304	2.16366	2.19146	4.13499	3.30132	2.49005	0	4.11572e-14	3.59644e-14	3.07717e-14
8388608	2.44994	2.52365	3.92725	2.43007	2.0341	0	7.547e-14	8.68925e-14	7.99574e-14
16777216	2.4841	2.44785	3.65946	3.26992	2.17668	0	3.73111e-14	9.03928e-15	3.8465e-16
33554432	2.25847	2.40657	4.04967	3.02292	2.61852	0	3.90617e-13	3.17198e-13	3.20597e-13

norm, cyclic, reduction

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.3633	2.12149	2.9576	3.09887	2.74375	0	3.15546e-14	2.11647e-14	2.82837e-14
2097152	2.44456	2.36728	3.95627	3.21408	2.35899	0	2.28491e-14	1.10165e-14	1.37367e-14
4194304	2.39955	2.35267	3.45747	3.35544	2.76597	0	4.11572e-14	3.59644e-14	3.05794e-14
8388608	2.47306	2.34057	3.92725	3.45495	2.63131	0	7.547e-14	8.68925e-14	7.99574e-14
16777216	2.4506	2.36555	3.63506	3.82638	2.23467	0	3.73111e-14	9.03928e-15	3.8465e-16
33554432	2.48289	2.39919	3.58597	3.12758	1.76602	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	2.3633	2.3914	3.95899	5.5711	13.4978	0	3.54027e-14	2.52052e-14	2.53976e-14
2097152	2.42987	2.37562	2.87575	4.38537	5.96358	0	1.29206e-14	6.80033e-15	1.04725e-14
4194304	2.12989	2.09073	4.20292	4.90844	6.07365	0	3.13487e-14	3.94262e-14	2.73099e-14
8388608	2.18909	2.32758	3.48364	4.78802	5.15271	0	6.48634e-14	5.99681e-14	6.21438e-14
16777216	2.32272	2.39938	4.26818	6.04166	6.49118	0	2.28867e-14	1.92325e-14	1.23088e-14
33554432	2.44159	2.10845	3.96758	6.63506	5.96145	0	3.39496e-13	3.29163e-13	3.22365e-13

norm, sequaltial

N	Sequential	1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
1048576	2.38008	2.30903	2.22229	2.41437	2.41727	0	0	0	0
2097152	2.39817	2.38684	2.39249	2.42696	2.36728	0	0	0	0
4194304	2.4396	2.44835	2.40802	2.44251	2.41655	0	0	0	0
8388608	2.33276	2.41329	2.47306	2.47598	2.47306	0	0	0	0
16777216	2.34017	2.38105	2.34773	2.47283	2.42338	0	0	0	0
33554432	2.36537	2.44159	2.46465	2.05855	2.20546	0	0	0	0

5.3 Execution Results

"pmatvec" Execution Results, all with openmp directives.

1 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	2.44069	2.70455	2.70455	0.0596533	0.0601915	2.77968
128	16384	81408	2.35987	2.08947	2.86556	0.0596637	0.0596814	2.86556
256	65536	326656	1.14422	1.14422	1.33241	0.0600079	0.0600079	1.26579
512	262144	1308672	1.03147	1.01153	1.24635	0.059367	0.0560309	1.23169
1024	1048576	5238784	0.97961	0.983608	1.1421	0.0595169	0.0596938	1.25513

2048	4194304	20963328	0.901649	0.926556	1.06143	0.0586695	0.0598525	1.21968

2 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	2.35455	2.04221	4.00273	0.0275102	0.0186364	4.88138
128	16384	81408	2.41674	2.44621	5.42133	0.0230192	0.0230298	5.57193
256	65536	326656	1.14422	1.15072	1.74592	0.0263569	0.017156	1.42624
512	262144	1308672	0.964919	0.872448	1.41478	0.0268446	0.0247532	1.25382
1024	1048576	5238784	0.956286	0.926862	1.28868	0.0203913	0.0273659	1.3314
2048	4194304	20963328	0.972212	0.91894	1.35247	0.0226845	0.0205939	1.48413

4 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	2.41129	2.15201	6.06475	0.0135045	0.0145798	5.26676
128	16384	81408	2.22877	2.47641	9.55187	0.0124204	0.0115881	6.07846
256	65536	326656	1.13779	1.17748	7.50099	0.0146483	0.0125996	2.10965
512	262144	1308672	1.14419	1.17633	1.58627	0.0122736	0.0109096	1.45408
1024	1048576	5238784	1.0041	0.995802	1.65058	0.0107948	0.0115392	1.36149
2048	4194304	20963328	0.952879	0.924003	1.39756	0.00968255	0.0104892	1.52461

8 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	1.55145	2.08476	5.55935	0.00695837	0.00695644	6.67122
128	16384	81408	1.98603	2.15687	11.1439	0.0068905	0.00717826	10.5573
256	65536	326656	1.11893	1.20552	15.579	0.00706998	0.00683404	9.64413
512	262144	1308672	0.930611	0.787171	1.29252	0.0066449	0.00701983	1.61067
1024	1048576	5238784	0.98764	0.995802	1.59592	0.00729238	0.0073321	1.52522
2048	4194304	20963328	0.961069	1.01949	1.53157	0.00717093	0.00722936	1.3803

"pmatvec" Execution Results, only parallelizing the computations without resources sharing.

1 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	2.44069	2.66849	2.66849	2.8591	3.12714	2.7416
128	16384	81408	2.35987	2.44621	2.8252	2.99387	2.11147	2.8252
256	65536	326656	1.05483	1.08303	1.28181	1.31511	1.31511	1.27375
512	262144	1308672	1.0214	1.00667	1.44405	1.40529	1.21737	1.22449
1024	1048576	5238784	0.860657	0.523878	0.88924	0.909374	0.941344	0.621093
2048	4194304	20963328	0.842747	0.685917	0.853469	1.06143	0.91643	1.01028

2 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	2.38258	2.1067	4.16951	2.70455	1.57588	5.00342
128	16384	81408	2.33243	2.38797	5.42133	2.50737	2.99387	5.27867
256	65536	326656	0.858164	0.82664	2.10965	0.983139	0.79112	1.39674
512	262144	1308672	0.556882	0.621328	1.09056	1.10787	0.72704	1.23169
1024	1048576	5238784	0.741489	0.779884	0.983608	1.01254	0.956286	1.03872
2048	4194304	20963328	0.752048	0.810177	0.975039	0.947495	0.88035	1.09972

4 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	1.64046	2.04221	5.4091	2.59918	2.81883	5.26676
128	16384	81408	2.35987	2.60506	5.73112	2.90709	2.7478	9.1177
256	65536	326656	0.794222	0.761379	1.85804	0.892188	0.900119	5.06317
512	262144	1308672	0.883492	0.538271	1.21737	0.956107	1.04173	1.23898
1024	1048576	5238784	0.854553	0.833855	1.27505	1.11567	1.08551	1.34628
2048	4194304	20963328	0.961069	0.961069	1.38601	1.11433	1.089	1.43339

8 threads

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T
64	4096	20224	2.38258	2.06326	5.71819	1.92439	2.50171	6.90127
128	16384	81408	1.68562	2.35987	10.0295	2.27942	2.60506	10.0295
256	65536	326656	0.908192	0.900119	9.20576	1.37773	0.861816	8.43861
512	262144	1308672	0.81158	0.872448	1.45408	1.0683	1.0214	1.46425
1024	1048576	5238784	0.902562	0.919787	1.46942	1.16983	1.18129	1.4968
2048	4194304	20963328	0.921465	0.88969	1.38601	1.13315	1.09972	1.41525

5.3 Exeuction Results

"pmatmat.exe" with only one vector on the RHS

1 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	1	1.48476	1.59209	2.20239
128	16384	81408	1	1.4564	1.61625	2.17266
256	65536	326656	1	0.934599	1.29406	1.12152
512	262144	1308672	1	0.831392	1.05475	0.923768
1024	1048576	5238784	1	0.667893	1.05435	1.08156
2048	4194304	20963328	1	0.873472	0.808503	0.758363

2 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	1	1.48476	2.16629	1.97229
128	16384	81408	1	1.32532	3.15553	2.17266
256	65536	326656	1	1.07666	1.84359	1.14053
512	262144	1308672	1	0.877867	1.28488	0.981504
1024	1048576	5238784	1	0.692732	1.03482	1.0222
2048	4194304	20963328	1	0.87088	1.20776	1.06722

4 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	1	1.51889	2.64287	1.97229
128	16384	81408	1	1.2503	5.76227	2.13762
256	65536	326656	1	1.02735	4.07825	1.18055
512	262144	1308672	1	0.851425	1.34606	1.04694
1024	1048576	5238784	1	0.77254	1.27001	1.00384
2048	4194304	20963328	1	0.858148	0.793207	1.09103

8 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	1	1.48476	2.31831	1.71615
128	16384	81408	1	1.23862	5.52218	2.13762
256	65536	326656	1	0.928154	8.41139	1.35942
512	262144	1308672	1	0.929846	1.53627	1.1585
1024	1048576	5238784	1	0.745072	1.32001	0.947125
2048	4194304	20963328	1	0.868304	1.1979	1.07899

"pmatmat.exe" with 32 columns on the RHS.

1 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	32	12.355	3.57645	4.11834
128	16384	81408	32	4.87398	3.5138	3.87419
256	65536	326656	32	4.86186	3.42721	3.80109
512	262144	1308672	32	4.65306	3.40468	3.70597
1024	1048576	5238784	32	4.45558	3.37816	3.6149
2048	4194304	20963328	32	4.42425	3.35413	3.53067

2 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	32	11.3254	6.79526	3.99721
128	16384	81408	32	4.87398	5.81128	3.21475
256	65536	326656	32	4.26653	4.75136	3.60448
512	262144	1308672	32	4.45505	5.04548	3.64152
1024	1048576	5238784	32	3.64437	3.64437	3.45652
2048	4194304	20963328	32	4.41696	4.9235	3.67073

4 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	32	9.70752	7.55029	2.42688
128	16384	81408	32	4.72166	5.21011	2.69809
256	65536	326656	32	4.64577	5.80722	3.66772
512	262144	1308672	32	4.02668	4.60192	3.12519
1024	1048576	5238784	32	4.44082	5.40778	3.60518
2048	4194304	20963328	32	4.38091	4.06562	3.64827

8 threads

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC
64	4096	20224	32	12.355	13.5905	3.88301
128	16384	81408	32	5.03644	10.7924	2.74715
256	65536	326656	32	4.64577	5.36051	2.90361
512	262144	1308672	32	4.06578	4.86948	3.17254
1024	1048576	5238784	32	4.56166	4.6567	3.62467
2048	4194304	20963328	32	4.35602	3.81694	3.62364