AMATH 563 PS6 – Ben Francis

Question 1

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

Answer
A minimum size and maximum size of vectors are given, then the min is doubled until right before it exceeds the max. We then specify how many "parts" to divide the problem into. For each of the problems described before, we test it with one thread, then an additional number of tests where we double the number of threads each time, right until we would exceed the number of "parts" (threads) specified in the function call.

Question 2

ALSO RUN PARFOR

Should be the same

Which version of ``norm`` provides the best parallel performance?

Answer
The reductions are better. Specifically, *norm_block_reduction* is best for me. *parallel_for* also performed nearly identically to *norm_block_reduction*.

How do the results compare to the parallelized versions of ``norm`` from ps5?

Answer
In terms of ratio, this is better by a significant margin; here the 2 thread case gets around 2x better performance, the 4 thread case gets around 4x performance, and the 8 thread case gets around 2-4x performance, depending on problem size. In ps5 I was getting around 1.5x performance, with only a couple jumps to 2x depending on problem size.

However, retesting my pnorm right now I am getting better absolute performance from ps5, just a worse ratio of performance when thread count is increased. My single thread in ps5 is better than my 4 thread in ps6, for example. This might be due to something being different in the testing procedure.

Question 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)?

Answer
I'm still getting norm_block_reduction, with parallel_norm similar. The disparity of block and cyclic is sharper with larger problem sizes, with norm_block_reduction performing even better, with cyclic falling down in quality. Cyclic_reduction is also worse with more threads.

How do the results compare to the parallelized versions of ``norm`` from ps5?

These are better, with similar degrees of "betterness" (between block and cyclic) as indicated in Question 2; the 1:2, 1:4 ratios maintain for the set of the problem sizes.

Question 4

Which version of ``norm`` provides the best parallel performance for small problems (i.e., problems smaller than the low end of the default sizes in the drivers)?

Answer
Cyclic_reduction is better the smaller it gets, finally reaching a performance level similar to block_reduction/parallel_for at sizes 131072 and 262144.

How do the results compare to the parallelized versions of ``norm`` from ps5?

Answer
I am not seeing this "better the smaller it gets" from cyclic_reduction in ps5.

Sparse Matrix-Vector Product

Question 5

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

Answer
It starts with 1, then uses a for loop to multiply by 2 until it gets 8, so that it will run: 1, 2, 4, 8 threads (one loop for each). If OpenMP is defined (presumably it is!) then it sets the number of OMP threads to the corresponding value of the loop (1,2,4,8)

Question 6 for Piazza

(This discussion was conducted on piazza)

Question 7

Which methods did you parallelize?

Answer
I parallelized:

- CSR matvec

- CSC t_matvec

What directives did you use?

Answer

#pragma omp parallel for

For both of them.

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

Answer

Over 2x speedup for CSR and CSC^T (as expected) for 2 threads, with less speedup for larger problem sizes.

2-3x speedup for CSR and CSC^T for 4 threads, closer to 3x speedup for smaller sizes and closer to 2x for larger sizes (always better than 2 threads).

2-3x speedup for CSR and CSC^T for 8 threads, closer to 3x speedup for the middle sizes and closer to 2x for the smaller and larger sizes. For smaller, it is the worst, for middle it is near 4 threads, and for larger it is around the same speedup as 2 threads.

Sparse Matrix Dense Matrix Product (AMATH583 Only)

Question 8

Which methods did you parallelize?

Answer

CSR and CSC matmat.

What directives did you use?

Answer

#pragma omp parallel for

For CSR matmat: Before the outer *i* loop

For CSC matmat: Before the *k* loop (I rewrote it to move the *k* loop outside for clarity, so that it would also match the CSR version more closely)

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

Answer

All answers are only for size 2048.

CSR (matmat)
1 thread: 0.84
2 threads: 1.66
4 threads: 1.82
8 threads: 1.74

Analysis: Definite speedup for 2 threads, almost exactly 2x; since my computer is 2 core, I take this as a success!

CSC (matmat)
1 thread: 0.91
2 threads: 1.63
4 threads: 1.52
8 threads: 1.43

Analysis: Also around 2x speedup.

How does the parallel speedup compare to sparse matrix by vector product?

Answer

CSR (matvec)
1 thread: 1.32
2 threads: 1.96
4 threads: 2.11
8 threads: 1.22

CSC (t_matvec)
Spare matrix by vector (for reference):
1 thread: 1.17
2 threads: 2.11
4 threads: 1.32
8 threads: 2.13

These also get around 2x, a little less.

PageRank Reprise

Question 9
** Describe any changes you made to pagerank.cpp to get parallel speedup.*

Answer
Changed CSR to CSC

How much parallel speedup did you get for 1, 2, 4, and 8 threads?

Answer
CSC
1 thread: 233
2 threads: 138
4 threads: 250
8 threads: 135

CSR
1 thread: 288
2 threads: 422
4 threads: 265
8 threads: 462

Comparative speedup from CSC:CSR
1 thread: 20%
2 threads: 67%
4 threads: 5%
8 threads: 70%

(Results summarized above)

Question 10

** (EC) Which functions did you parallelize?*

Answer

(1): Stochastify in CSC matrix, *#pragma omp parallel for*, before second inner for loop

(2): Two norm in amath583.cpp, *#pragma omp parallel reduction(+:sum)* before for loop

How much additional speedup did you achieve?

Results from stochastify parallelization:
1 thread: 230
2 threads: 122
4 threads: 122
8 threads: 125

Speedup due to stochastify (using results from Question 9):
1 thread: 1%
2 threads: 12%
4 threads: 51%
8 threads: 7%

Noticable speedup is seen in the 4 thread case, which is promising.

Results from two_norm parallelization:

1 thread: 232
2 threads: 127
4 threads: 119
8 threads: 115

Speedup due to two_norm:
A little better, but very similar results as stochastify. It is unclear if this is due to computational noise, or anything substantial underneath.

Answer

(1) Stochastify: 4290 / 4596 is approximately 90%, so around 10% speedup
(2) Two norm: No improvement; not sure why, but such is life.

Load Balanced Partitioning with OpenMP

Question 11

What scheduling options did you experiment with?

Answer
I experimented with all of them: *static, dynamic, auto, guided*. I experimented with chunk sizes 2, 8, and 32. Number of threads: 4.

Dataset: Hollywood-2009.mmio (878 MB)
I tried some smaller stuff, but this had most definitive results.

Are there any choices for scheduling that make an improvement in the parallel performance (most importantly, scalability) of pagerank?

Answer

Some results:
Chunk size 2, threads 8:
1. Static: 121
2. Dynamic: 116
3. Auto: 115
4. Guided: 112

Chunk size 8, threads 8:
1. Static: 117
2. Dynamic: 111
3. Auto: 140
4. Guided: 116

Chunk size 32, threads 8:

1. Static: 115
2. Dynamic: 136
3. Auto: 128
4. Guided: 110

Experimentally, I didn't really see that much reliable difference. Between the different chunk sizes shown above, there is not enough of a difference for me to say if there was one better than another. Furthermore, selection of chunk size is another variable. The best seems to be guided, though the margin is always in only tens of milliseconds, so this may be computational noise. Auto performed the worst.

OpenMP SIMD

Question 12

Which function did you vectorize with OpenMP?

Answer
norm_block_reduction

How much speedup were you able to obtain over the non-vectorized (sequential) version?

Answer
I will compare:
(1) simd and omp
(2) simd and sequential

(1) simd and omp:
simd performed about the same for all thread numbers (as expected), with about 5.8 GF/s at n = 1048576. This beats omp for 1 and 2 threads, but omp is better at 4 and 8 threads. (This makes sense; simd speedup is thread independent, omp's is thread dependent). So speedup is dependent on thread count.

(2) simd and sequential
Sequential alone at 1048576 gives 1.6 GF/s, so this is in the range of ¼ the performance, which makes sense as we would go from 64 size registers to the 256 vector registers. So 4x speedup.