HW 2: Parallel Programming with OpenMP

Mian Qin, UIN: 725006574

1. C/C++ implementation of the Triangular Matrix Inverse that uses OpenMP directives.
   The C++ implementation of Triangular Matrix Inverse follows the same idea as the Matlab example code which partition the matrix to submatrix and recursively calculate inverse of each submatrix. In order to parallel the algorithm using OpenMP, I mainly used two OpenMP directives (parallel for and task). The detailed description of the parallel strategies is discussed in section

   A) Code description.

   The C++ implementation doesn't rely on any mathematical libraries. I implemented my own matrix multiple using three nested loops with OpenMP parallel for to parallelize. The recursive matrix partition is similar as the Matlab example which will not be explained here. Details of the implementation can be referred to the source code.
   My implementation takes matrix (upper triangular) of **double** as input and does **in-place** calculation.
   The main function prototype is

```
void OMPRinverse(double **A int n)
```

   which takes a 2D double vector pointer and matrix size as input (matrix is an upper triangular matrix). After the function returns, the result is updated in the 2D **double** vector A itself.
   The main recursive function to calculate inverse is

```
void _OMPRecurRinverse(double **A, int cOA, int n)
```

   The first and second input combined to represent the submatrix. (A gives the row offset and cOA gives the column offset). The third input stands for the size of the submatrix (also upper triangular)

   B) Build & Run

   Compile: I have 5 optimizations which is incrementally applied for the evaluation (see Section 2&3). Each individual optimization version is compiled through C++ Macro as follow:

   ***non optimized**: icpc -qopenmp -std=c++11 Rinverse.cc -O3 -o Rinv_nonopt*
   ***opt1**: icpc -qopenmp -std=c++11 Rinverse.cc -O3 -DOPT1 -o Rinv_opt1*
   ***opt2**: icpc -qopenmp -std=c++11 Rinverse.cc -O3 -DOPT2 -o Rinv_opt2*
   ***opt3**: icpc -qopenmp -std=c++11 Rinverse.cc -O3 -DOPT3 -o Rinv_opt3*
   ***opt4**: icpc -qopenmp -std=c++11 Rinverse.cc -O3 -DOPT4 -o Rinv_opt4*
   ***opt5**: icpc -qopenmp -std=c++11 Rinverse.cc -O3 -DOPT5 -o Rinv_opt5*

   To compile code with original and inversed matrix, compiled with **-DDBG** flag

Run: There is one program argument for setting the matrix size. Number of OpenMP threads is set through environment variable **OMP_NUM_THREADS**. Example of using 16 threads to inverse 2048x2048 matrix is as follow:

```
OMP_NUM_THREADS=16 OMP_NESTED=TRUE OMP_PLACE=sockets ./Rinv_opt1 2048
```

C) Note

The build script and job file for batch system is included in the homework submission. Optionally we can use gnu gcc to compile by

g++ -fopenmp -std=c++11 Rinverse.cc -O3 -DOPT5 -o Rinv_opt5

D) Validation

To validate the correctness of the OpenMP version of the code. I compare the result generated by the Matlab (using inv) and my C++ implementation with the same input.

C++ with OpenMP (8 threads) results:

```
[celery1124@ada1 HW2-689]$ OMP_NUM_THREADS=8 OMP_NESTED=TRUE OMP_PLACE=sockets ./Rinv_opt1_icc_dbg 11
Matrix size: 11

oringal matrix:

5.6394  0.1261  0.0473  0.7146  0.5460  0.8616  0.1684  0.6851  0.5296  0.6866  0.3283
0.0000  4.5378  0.9266  0.2238  0.4866  0.1022  0.9697  0.2312  0.7765  0.8927  0.9397
0.0000  0.0000  6.2530  0.8235  0.7864  0.7204  0.3522  0.7850  0.3161  0.3318  0.8134
0.0000  0.0000  0.0000  6.4050  0.3325  0.4546  0.1834  0.7994  0.7850  0.8378  0.5121
0.0000  0.0000  0.0000  0.0000  7.1183  0.2797  0.8069  0.6869  0.5481  0.1607  0.8860
0.0000  0.0000  0.0000  0.0000  0.0000  6.9166  0.3740  0.8306  0.0884  0.8364  0.1111
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  5.9618  0.2539  0.5713  0.5059  0.4199
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  7.5116  0.5017  0.3703  0.2507
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  5.0223  0.1258  0.2022
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  5.3982  0.0104
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  5.2595

Finished in 4.28 miliseconds [Wall Clock]
inversed matrix:

0.1773  -0.0049 -0.0006 -0.0195 -0.0123 -0.0202 -0.0006 -0.0105 -0.0120 -0.0141 -0.0047
0.0000  0.2204  -0.0327 -0.0035 -0.0113 0.0008  -0.0323 -0.0010 -0.0265 -0.0300 -0.0284
0.0000  0.0000  0.1599  -0.0206 -0.0167 -0.0146 -0.0056 -0.0112 -0.0030 -0.0025 -0.0185
0.0000  0.0000  0.0000  0.1561  -0.0073 -0.0100 -0.0032 -0.0147 -0.0216 -0.0207 -0.0119
0.0000  0.0000  0.0000  0.0000  0.1405  -0.0057 -0.0187 -0.0116 -0.0120 -0.0005 -0.0210
0.0000  0.0000  0.0000  0.0000  0.0000  0.1446  -0.0091 -0.0157 0.0001  -0.0205 -0.0015
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.1677  -0.0057 -0.0185 -0.0149 -0.0124
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.1331  -0.0133 -0.0088 -0.0058
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.1991  -0.0046 -0.0076
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.1852  -0.0004
0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.1901
```

Matlab results:

```
>> inv(test)

ans =

    0.1773   -0.0049   -0.0006   -0.0195   -0.0123   -0.0202   -0.0006   -0.0105   -0.0120   -0.0141   -0.0047
         0    0.2204   -0.0327   -0.0035   -0.0113    0.0008   -0.0323   -0.0010   -0.0265   -0.0300   -0.0284
         0         0    0.1599   -0.0206   -0.0167   -0.0146   -0.0056   -0.0112   -0.0030   -0.0025   -0.0185
         0         0         0    0.1561   -0.0073   -0.0100   -0.0032   -0.0147   -0.0216   -0.0207   -0.0119
         0         0         0         0    0.1405   -0.0057   -0.0187   -0.0116   -0.0120   -0.0005   -0.0210
         0         0         0         0         0    0.1446   -0.0091   -0.0157    0.0001   -0.0205   -0.0015
         0         0         0         0         0         0    0.1677   -0.0057   -0.0185   -0.0149   -0.0124
         0         0         0         0         0         0         0    0.1331   -0.0133   -0.0088   -0.0058
         0         0         0         0         0         0         0         0    0.1991   -0.0046   -0.0076
         0         0         0         0         0         0         0         0         0    0.1852   -0.0004
         0         0         0         0         0         0         0         0         0         0    0.1901
```

2. Describe your strategy to parallelize the algorithm.

There are mainly two categories of optimizations using OpenMP.

A) First, to parallel the recursive call for computing the inverse of the submatrix (each recursive call can be paralleled to a new threads and compute in parallel, it needs a barrier to synchronize all children threads to guarantee correctness). This is done through **omp task** directive. However, when applying this optimization, when the recursive call reaches the bottom (i.e. the submatrix is partitioned to 2x2 or 1x1), the task is too small for a separate thread (thread schedule overhead is much more than computation itself). We'd better let the small recursive

call run in serial fashion. This is similar as the Matlab example by calling a serial **inv** function when the matrix size is smaller or equal than 16. I set this threshold to 32.

B) Second, after applying the task directive to recursive calls. The speedup is neglectable. I did some profiling by time each part of the code for serial code running. I found that the major time is consumed by the two matrices multiplication function to calculate the top right submatrix. So, the second optimization is to use **opm parallel for** to parallel the matrix multiplication. Same thing applies, we can further optimize the code by skipping the parallel for region (nested parallel) for smaller matrix multiplication.

Besides, I also tried different schedule type for **opm parallel for** (static and dynamic).

The optimization matrix for different implementation for evaluation is demonstrated in the below table.

	task recursive	skip recursion task for small partition	parallel for matrix multiplication	skip parallel for small matrix multiplication	dynamic schedule	static schedule
Opt1	✓	✗	✗	✗	✗	✗
Opt2	✓	✓	✗	✗	✗	✗
Opt3	✓	✓	✓	✗	✗	✗
Opt4	✓	✓	✓	✓	✓	✗
Opt5	✓	✓	✓	✓	✗	✓

C) A side note regarding memory copy. In the matrix multiplication implementation, my implementation also does an in-place update. (MmultL and MmultR will update into the left and right operand respectively). However, inside the multiplication function, it requires a temporary matrix to hold the results and copy back to the operand 2D vector. I implement this using **memcpy** which proves to be very fast (usually SIMD optimized). I did some profiling. This memcpy loop runtime is neglectable compared to matrix multiplication computation. Only when the memcpy loop is really large that I start parallel them (threshold to be 1024).

3. Determine the speedup and efficiency obtained by your routine on 1, 2, 4, 10, and 20 processors.

   For evaluation, I evaluated the above 5 optimization versions for inversing a 2048x2048 size matrix (runtime in range of seconds to tens of seconds). I ran each experiment three times to rule out some performance jitter. The baseline to calculate speedup is the runtime of serial code (without OpenMP directives).

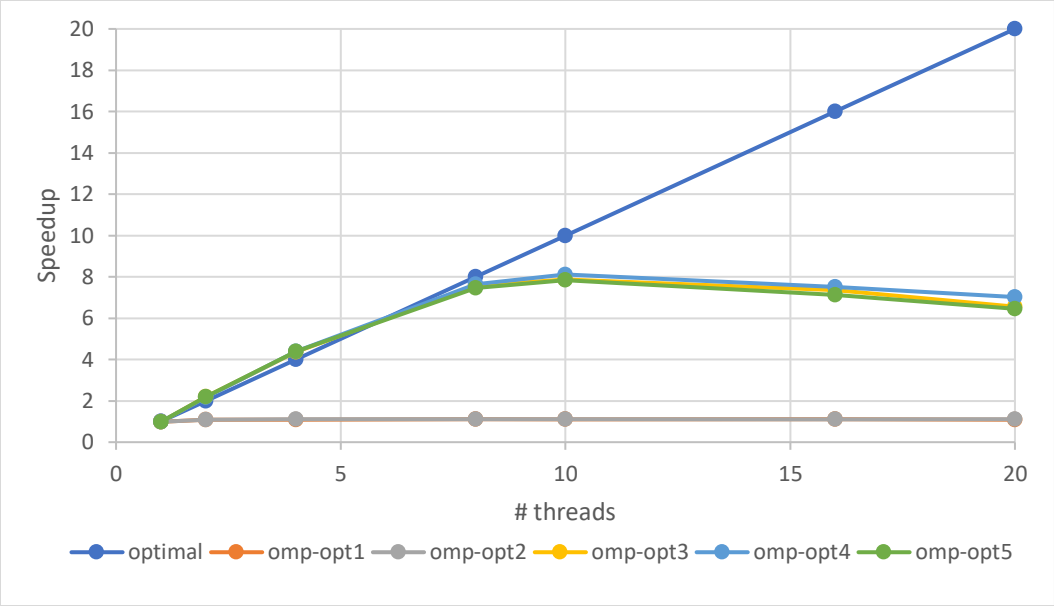

Figure 1 intel icc evaluation results

Figure 1 illustrates the speedup results for different optimizations (code is compiled using intel icc). From the results, we see that only parallel the recursive calls merely improve performance (speedup is almost constant 1 across 2 to 20 threads). As we analyzed in section 2, this is due to the fact that matrix multiplication takes the majority of computation time (the final 2 1024x1024 matrix multiplication takes up ~80% of the overall runtime).

However, one observation that doesn't make much sense is that OPT4 & OPT5 that skip the parallelization for small matrix multiplication doesn't improve performance much. Thus, the performance scaling is not perfect (stop scaling at 10 threads). I compiled with gnu g++ and got different results. For g++ compiled binary, the OPT4&OPT5 get much better performance scaling compared to OPT3 as shown in Figure2.

Between static and dynamic scheduling, dynamic performs slightly better.

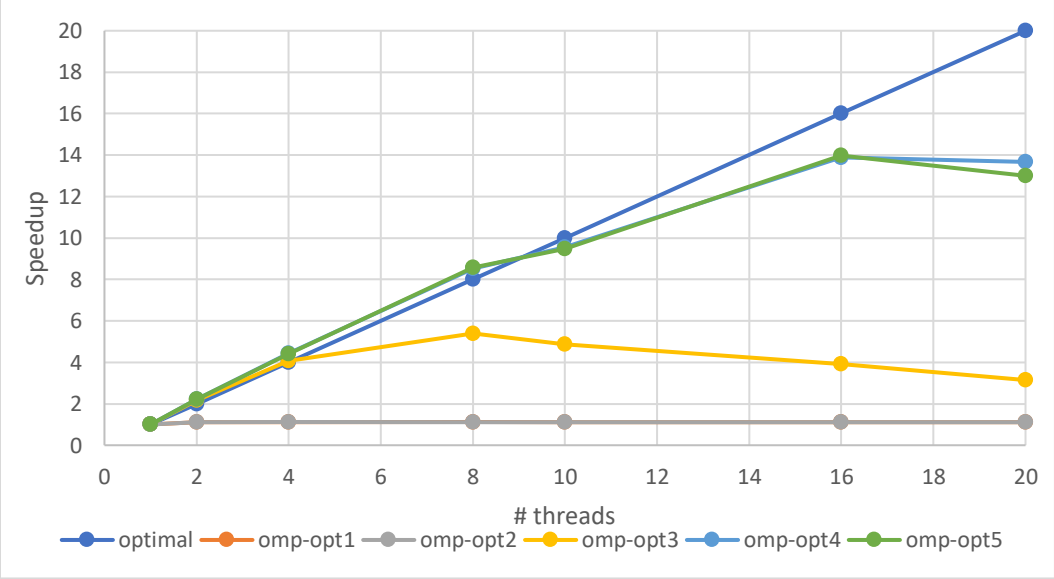

Figure 2 gnu gcc evaluation results