Minor Project: OpenMP

Mian Qin, UIN: 725006574

1. ***C/C++ implementation of the Gaussian Process Regression using OpenMP directives.***

   The C++ implementation of **Gaussian Process Regression** follows the Matlab example code. Overall there are two major implementations (**non-opt & opt1** use ***triangular matrix inverse and matrix multiplication*** to calculate the final phase fstar=k'*(U\(L\f)); **opt2** uses simple ***forward/back substitution*** to calculate the final phase.). There are four main functions that implemented to fulfill GPR. **First**, LU factorization. I used the Cholesky-Crout Algorithm for factorizing the $tI + K$ matrix. It will generate two **transposable** triangular matrix as L and U. **Second**, triangular matrix inversion. To implement this function, I used the code from HW2 (as a separate source file). Due to the fact that L and U we got is transposable, we only need to do **single** matrix inverse for U and L can be derived by transpose the inversed U. **Third**, arbitrary matrix multiplication. This is used to calculate the final predicted value fstar using the inversed L and U. **Fourth**, forward/back substitution for calculating final phase of fstar.

   In order to parallel the algorithm using OpenMP, I mainly work on the first and third functions (since triangular matrix inverse has already be paralleled in HW2). The detailed description of the parallel strategies is discussed in section 2.

   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 GPR function prototype is

```
double GPR(int m, double rstar_l, double rstar_r)
```

      which takes a integer m and two double rstar_l and rstar_r as the input and return the predicted value fstar as double type.

      The LU factorization function prototype is

```
void CholeskyFactorization(double **K, double **U, int n)
```

      which takes a 2D double vector pointer K and matrix size n as input. The result is returned to another 2D double vector pointer U. (U need to be allocated by the called as same size as K and filled with zero using calloc).

      The upper triangular matrix inverse 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 arbitrary size matrix multiplication function prototype is

```
void Mmult(double **A, int rA, int cA, double **B, int rB, int cB, doub
le **C, int rC, int cC)
```

which takes two 2D double vector pointer A and B and corresponded matrix size as input. The result is written in the another 2D double vector C.

Forward/Back substitution function prototype is

```
void ForwardSubstitution(double **L, int nL, double *f, int nf, double
*y, int ny)
void BackSubstitution(double **U, int nU, double *f, int nf, double *y,
 int ny)
```

which takes triangular matrix and a vector to calculate the \ operation (for example L\f).

B) Build & Run

Compile: The code is compiled as follow:

```
icpc -qopenmp -std=c++11 GPR.cc Rinverse.cc -O3 -o GPR_icc
icpc -qopenmp -std=c++11 GPR.cc Rinverse.cc -O3 -o -DOPT1 GPR_icc
icpc -qopenmp -std=c++11 GPR.cc Rinverse.cc -O3 -o -DOPT2 GPR_icc
```

Note: There are two optimization macro (OPT1, OPT2) for the triangular matrix multiplication, see details in section 2, F&G).

Run: There is three program argument for setting the grid size and coordinates of rstar. Number of OpenMP threads is set through environment variable **OMP_NUM_THREADS**. Example as follow:

```
OMP_NUM_THREADS=16 OMP_PLACE=sockets ./GPR_icc 16 0.5 0.5
```

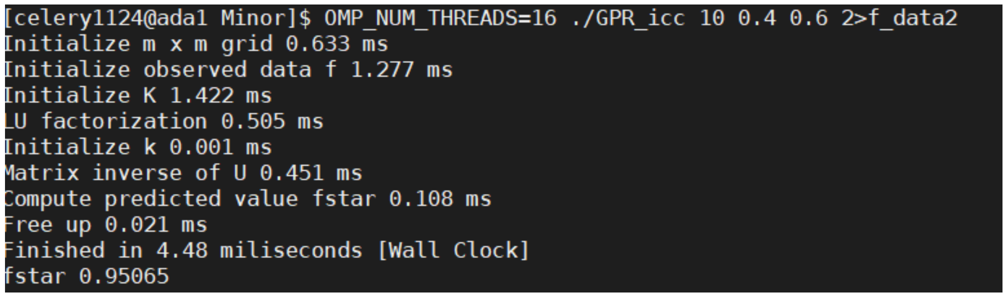
C) Note

The build script and job file for batch system is included in the homework submission.

D) Validation

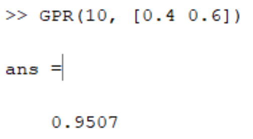
To validate the correctness of the OpenMP version of the code. I compare the result generated by the Matlab example code and my C++ implementation with the same input. Since the random function of C++ (drand48_r) and Matlab (rand) generate different results, I print out the C++ implementation observed data vector f and copied to Matlab code and make sure the input is identical and then compare the results.

C++ with OpenMP (8 threads) results (input argument 10 0.4 0.6) :



```
[celery1124@ada1 Minor]$ OMP_NUM_THREADS=16 ./GPR_icc 10 0.4 0.6 2>f_data2
Initialize m x m grid 0.633 ms
Initialize observed data f 1.277 ms
Initialize K 1.422 ms
LU factorization 0.505 ms
Initialize k 0.001 ms
Matrix inverse of U 0.451 ms
Compute predicted value fstar 0.108 ms
Free up 0.021 ms
Finished in 4.48 miliseconds [Wall Clock]
fstar 0.95065
```

Matlab result with same input:



```
>> GPR(10, [0.4 0.6])

ans =

    0.9507
```

2. ***Describe your strategy to parallelize the algorithm. Discuss any design choices you made to improve the parallel performance of the code.***

Before applying any parallel optimization to my code. I first analysis the **computation proportion** for each part of the code when running serial (single thread). This will give which part of the code takes most of the computation time and can benefit the most from parallelization. Table 1 the first row shows the **computation proportion** for each part of the code (the input parameters are 60 1 1). As the results shows, the majority of the computation lays on LU factorization, Matrix inverse and the final three matrix multiplication for non-opt and opt1 implementation. (overall takes up ~99%) For opt2 implementation using forward/back substitution, the main computation time spend on LU factorization (~99%). So, the main optimizations are on those parts.

Below are the optimization strategies for each part of the code.

A) **Initialize m x m grid of points**. This is a simple two levels loop with very little computation. Although this part takes near ~0% time to finish, I still use the **omp parallel for** with **static** schedule to parallel it.

B) **Initialize observed data vector f**. Similar with A), this part also takes nearly 0% to finish. I used the thread safe *drand48_r* to generate the random number. I also tried to parallel it with omp parallel for. However, the parallel performance is even worse compared to serial performance. Probably because there are lock contention issue or cache performance issue. So, this part I didn't apply any parallel optimization.

C) **Initialize k & Initialize K**. Similar with A) that doesn't cost much computation, I used the **omp parallel for** with **static** schedule to parallel it. For those two parallel sections. I use a cutoff threshold that only parallel when the loop size is large enough. Otherwise, let the code run in serial. This is a lesson learned from HW2.

D) **Compute LU factorization**. This is a major portion of the code that worth optimize. The algorithm for LU factorization I used is Cholesky-Crout algorithm. It contains three loops. The outer loop cannot be parallelized since the inner two loops has dependency. Each outer loop iteration compute 1 row of the result upper triangular matrix **U** from top to bottom and the later iteration needs to use the results generated from previous iterations. The major computation load happens in the last two level of inter loops which appears to have no dependency. So, I added the **omp parallel for** optimization here with a cutoff threshold (when j is large, run in serial). The upper inner loop used to calculate the diagnols is does little job that doesn’t worth to be parallelized.

```
    // Cholesky-Crout Algorithm
    for (int j = 0; j < n; j++) {
        double sum = 0;
        // Diagnols
        for (int k = 0; k < j; k++) sum += (U[k][j] * U[k][j]);
        U[j][j] = sqrt(K[j][j] - sum);

        #pragma omp parallel for private(sum) shared(n, j, K, U) schedule(
static) if (j < n-64)
        for (int i = j+1; i < n; i++) {
            sum = 0;
            for (int k = 0; k < j; k++) sum += (U[k][i] * U[k][j]);
            U[j][i] = (K[i][j] - sum) / U[j][j];
        }
    }
```

E) **Matrix inverse of U**. In order to compute the final results, the LU for K requires inverse computation. Since L and U are transposable, it only requires one matrix inverse for the U and a transpose to inv(U) equals inv(L). This inverse part borrows code from HW2. It mainly use two parallel optimizations, **omp task** for recursive jobs and **omp parallel for** for the matrix multiplications.

F) **Compute predicted value fstar**. This is the most computation intensive part of the whole GPR method. It computes three matrix multiplications (n x n * n x n, n x n * n x 1 and 1 x n * n x 1). The first one takes most of the time. I use **omp parallel for** with **static** schedule to parallel the matrix multiplication.

Another optimization to F) is on the algorithm itself. As the first nxn * nxn is multiplication of two triangular matrix (inv(U)*inv(L)). So, I did a optimization by reducing the amount of unnecessary float point multiplication (times zero). The highlighted code snippet shows the optimization. Table 1 demonstrated that this optimization hugely reduces the computation cost especially for single threads. However, it performs a little bit worse on threads scaling due to the unbalance of the inter loops computation.

```
#pragma omp parallel for shared (nU, U, L, C) collapse(2) schedule(static)
 if (nU > 8)
    for (int r = 0; r < nU; r++) {
        for (int c = 0; c < nU; c++) {
            double sum = 0;
            for (int i = std::max(r, c); i < nU; i++) {
                sum += U[r][i] * L[i][c];
            }
            C[r][c] = sum;
        }
    }
```

G) **Forward/Back substitution**. With forward/back substitution, we don't need to conduct the computation intensive matrix inversion and multiplication which is O(N^3) complexity. Rather, forward/back substitution only cost O(N^2) time. So, **opt2** performs the best in the three implementations. The two loops is not suitable for parallelization due to the dependency. For the outer loop each iteration needs results from last iteration. For the inner loop, it needs reduction on the **sum** variable. I tried to do **omp parallel for with reduction** for the inner loop. However, it will diminish the performance significantly. Besides, since for opt2, the forward/back substitution only takes up ~0.2 (Table 1) of the total run time. Thus, there is no need to parallel this part.

Table 1 Computation time break down for GPR

	Initialize m x m grid of points	Initialize observed data vector f	Initialize k	Initialize tI + K	Compute LU factorization	Matrix inverse of U	Compute predicted value fstar
Computation proportion (non-opt) (%)	0.000	0.000	0.000	0.141	13.671	13.746	72.442
16 thread speedup (non-opt)	1.386	1.154	1.403	11.939	14.954	13.136	15.395
Computation proportion (opt1) (%)	0.001	0.000	0.000	0.306	29.561	29.761	40.370
16 thread speedup (opt1)	1.384	1.121	1.270	11.977	14.899	13.024	10.430
Computation proportion (opt2) (%)	0.004	0.000	0.000	1.006	98.794	0	0.196
16 thread speedup (opt2)	1.449	1.138	1.500	12.204	15.180	NA	0.981

3. ***Compute the flop rate you achieve in the factorization routine and in the solver routine using all the cores. Compare this value with the peak flop rate achievable on a single core, and estimate the speedup obtained over one core and the corresponding efficiency/utilization of the cores on the node.***

For evaluation, I evaluated the two run cases (small: 40, 0.5 0.5, large: 60 1 1, extreme: 80 0.4 0.6). I ran each experiment three times to rule out some performance jitter. Table 2 shows the flop rate for factorization routine and GPR solver routine. (Here the flop contains add/sub, multiply, divide and exp and sqrt operations). The factorization achieves higher flop rate compared to overall solver due to solver also count other operations (int operation/branches). The GPR opt1 reduce the number of matrix multiplication operation thus achieve higher flop rate (efficiency).

Opt1 & opt2 share the similar overall flop rate which implies opt2 simply reduce the run time by reducing the amount of floating point operations.

Table 2 Flop rate for factorization and GPR solver routine

	LU factorization	GPR non-opt	GPR opt1	GPR opt2
Single core - small (GFLOPS)	14.358	2.584	3.490	3.495
20 cores - small (GFLOPS)	221.509	33.784	42.78	42.778
Single core - large (GFLOPS)	13.627	1.864	3.06	3.062
20 cores - large (GFLOPS)	248.616	30.337	45.65	45.646

Figure 1 illustrates the speedup efficiency results for non-opt routine. The baseline to calculate speedup is the runtime of single thread. As you can see the speedup is almost linear up till 18 cores (threads). For large and extreme case, the maximum speedup can achieve ~16.5x, for small test case, the maximum speedup is ~14x. Also, the efficiency is better for larger input size compared to smaller input size which is straightforward. This is due to the relative portion for serial part of the code is larger for smaller input (small). The 20 cores performance slightly dips. Also, from the error bar, the 20 cores results have a lot of variations. I guess it might be the machine's schedule jitter that cause these variations.

Figure 2 illustrates the speedup efficiency results for opt1 GPR routine. The baseline to calculate speedup is the runtime of single thread. For large and extreme case, the maximum speedup can achieve ~14.5x, for small test case, the maximum speedup is ~12x. Also, the efficiency is better for larger input size compared to smaller input size which is straightforward. Compared to non-opt GPR results, the scaling is better, i.e. the almost linear scaling. But the absolute speedup is lower. Due to the analysis in Section 2, it's because the optimize of the two triangular matrix multiplication reduce the balance of the code (the inner loops). So, the openmp schedule is less efficient.

Figure 3 illustrates the speedup efficiency results for opt2 GPR routine. The baseline to calculate speedup is the runtime of single thread. For large and extreme case, the maximum speedup can achieve ~15-18x, for small test case, the maximum speedup is ~14x. Also, the efficiency is better for larger input size compared to smaller input size which is straightforward.

I also demonstrated the speedup of opt1 & opt2 GPR compared to non-opt GPR routine in Figure 4. As the results shows, the opt1 GPR can speedup almost 2x in 20 cores scenario compared to non-opt. The opt2 GPR can speed up to 126x in 20 cores compared to non-opt implementation thanks to forward/back substitution.

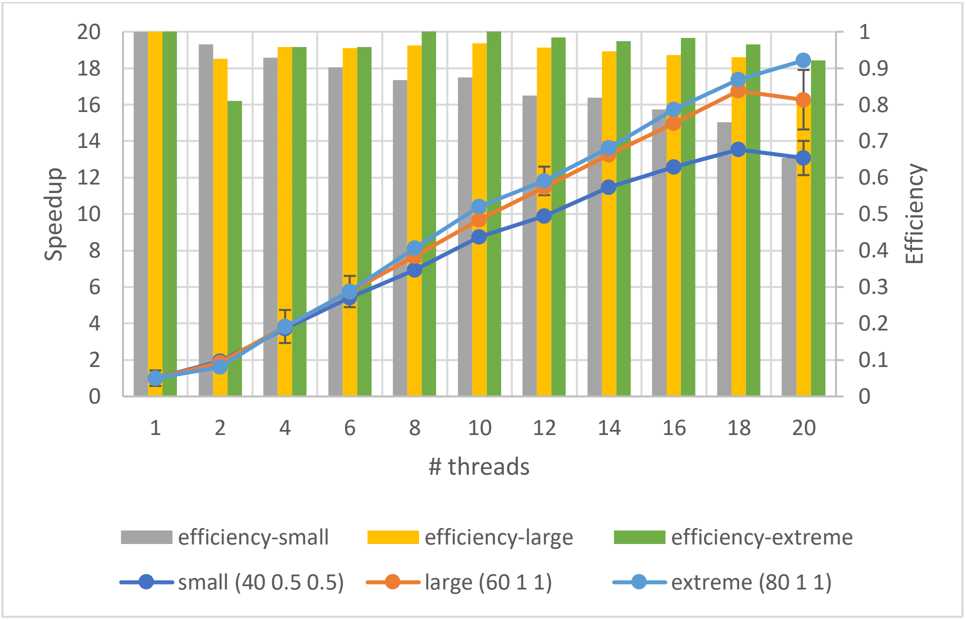


Figure 1 Speedup & efficiency results for non-opt GPR

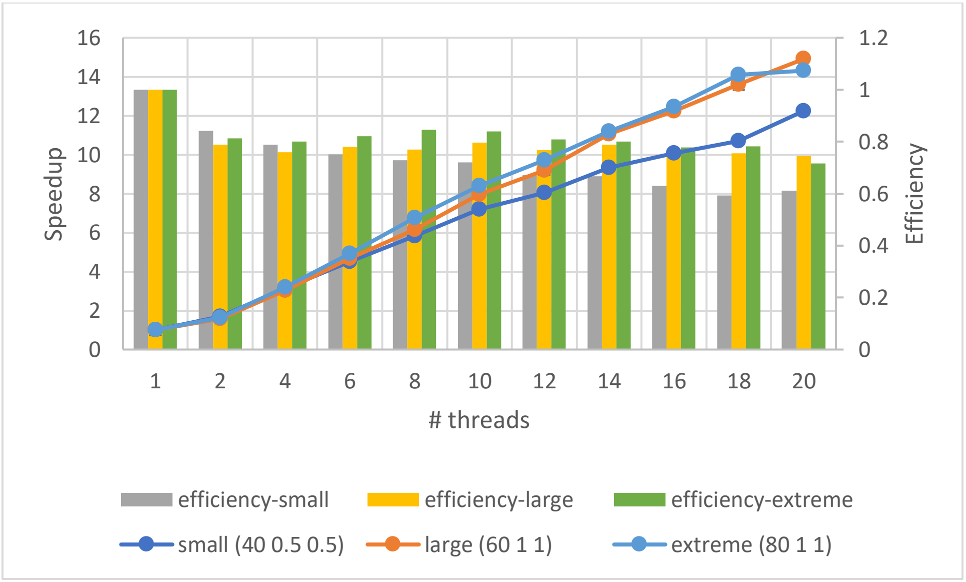


Figure 2 Speedup & efficiency results for opt1 GPR

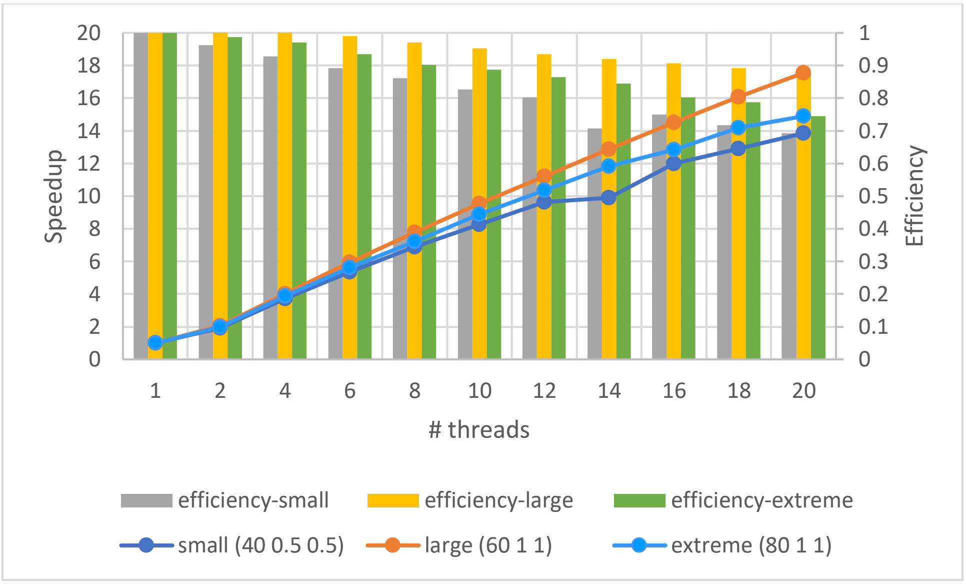


Figure 3 Speedup & efficiency results for opt2 GPR

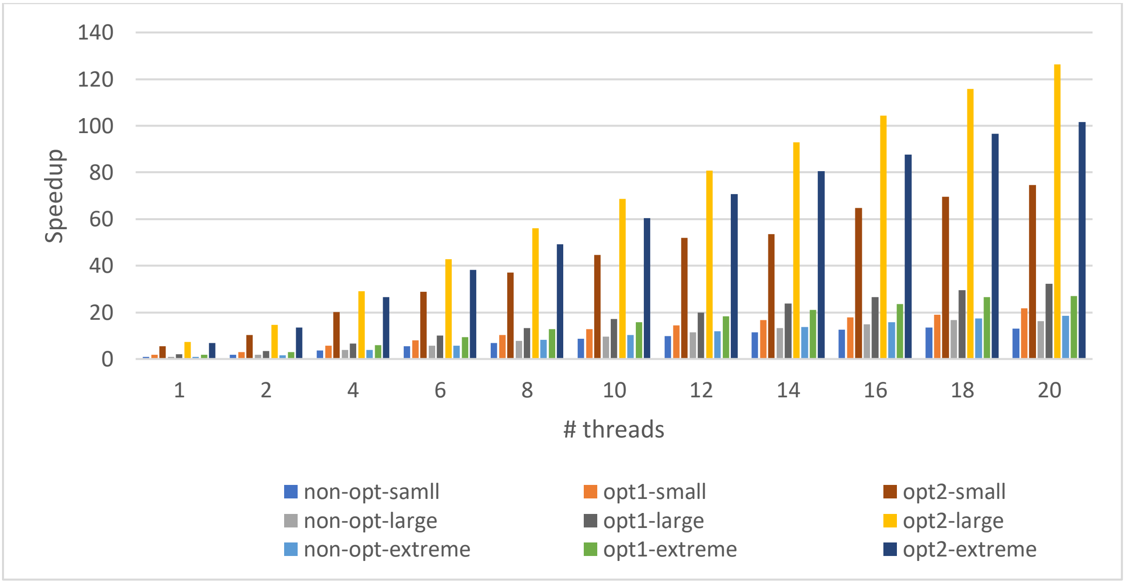


Figure 4 Speedup comparison for opt1, opt2 and non-opt GPR