1. (70 points) In this assignment, you have to develop an OpenMP-based parallel code to compute f(x,y) for a given point (x,y). The code should use a single shared-memory node on ADA, and should be parallelized to exploit all the cores on the processing node. The code should initialize the grid points and observed data values using equations (1) and (2). Next, the matrix A = (tI+K) should be computed. This should be followed by LU factorization of A. These factors should be used to compute the solution of the system Az=f using the L and U factors obtained in the previous step. Finally, the predicted value should be computed as $f(x,y)=kTz$. You must develop your own code to compute the LU factors and to solve the triangular systems. LU factorization can be replaced by Cholesky factorization, which is a more efficient algorithm for symmetric positive definite matrices.

I have used Standard LU Factorization + substitution to solve the assignment. I have done some work to make sure the generation of Matrix K and observed data become less compute heavy and I have spent a lot of time finding a way to increase the speed of LU factorization. My code works and has been extensively tested. For example, after substitution, I have multiplied my matrix to the result of the substation to see if I get the right-hand side. For LU factorization I have extracted a complete L and U matrix and multiplied them to see if I get my original matrix. I believe these testing become important for two reasons, 1) it is easy to make mistake with the indices, and 2) OpenMP sometimes does surprising things. I spent a lot of time with OpenMP because parallelizing the inner loop is not as efficient as having an external loop parallelized and while trying "creative" solutions I managed to produce many unintended consequences. In my code I have sued the following variable names for substation:

$$LY = B$$

$$UX = Y$$

So each matrix is named $matriL, matrixY, matrixB(observedValues)$, matrixU, matrixX, natrixY . I have parallelized all parallelable parts of the program from the final multiplication of $k^T(kTranspose)$ by $matrixY$ to other parts of generating the data at the beginning. Although for orders of m I have not parallelized since it would have more overhead than actual benefit.

In terms of development I usually develop the code first in Visual Studio to test the code logic and then use ADA HRPC. I transfer my code and compile it. Once it is compiled, I schedule my job file. For this project since I am using 20 cores, I cannot use the command line directly since it is limited to 8 cores and usually they fluctuate a lot more than the scheduled job.

2. (20 points) Describe your strategy to parallelize the algorithm. Discuss any design choices you made to improve the parallel performance of the code.

Well the main issue with LU is that the algorithm needs the inner loop parallelized, this is especially painful because you cannot start the parallel region outside the loop. So, you must initiate a parallel region every time you start iteration. I used $\#pragma\ omp\ single$ so that I can run the outer loop only once but then it gave me an error when running the parallel loop inside the single area. I created a single region outside the function but then the loop was not running correctly. I then gave up trying to initiate the parallel region outside the inner for loop. I tried many ways so I was convinced that not much can be done. So, then I focused on scheduling to speed up the processes. I used dynamic and static scheduling with different chuck sizes, I found chunk size of 25 to work best. Sometimes better than just static sometimes worse. I also tried to use collapse to see if that helps, but it only slowed the performance since the inner loop does not really have much computation in it anyway. I have tried at least 14 different version of the LU to make sure I do not miss any opportunity. I have included these 14 different ways in appendix. Since

the dynamic was having so much overhead, using dynamic was out of the equation. So I started playing with static. The issue is that with fixed size of chunk at the beginning when n-k is large, you may end up having small slices and once n-k become small then you get improvements since the iterations are not divided to very fine tasks. For this problem it is especially important to consider that the tasks in each iteration is shrinking every time the inner loop is called again. For instance, if n is 100, for k = 0 (the outer loop) the first iteration would have 99 operations but when k = 10, each iteration would have 9 operations. So this is why a fixed size chunk also improve some part but not all. Therefore, I decided to an unfixed chunk size based on value of the k and n. The code snippet shows the final form of my RU factorize:

```
for (uint64_t k = 0; k < n - 1; k++)
{
    uint64_t chunkSize = 100000;
    if( (chunkSize / (n - k)) < 20)
    {
        chunkSize = chunkSize / (n - k);
        #pragma omp parallel shared(matrix, k, n, chunkSize)
        {
            #pragma omp for nowait schedule(static,chunkSize)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                }
        }
    }
    else
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait schedule(static)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                }
        }
    }
}
```

The table below shows the performance based on 3 different chunk size and 2 different coordinates and M = 50.

20 Threads	M =50 x=0.5 y=0.5				M =50 x=0.2 y=0.9			
Chunk Size	Test1	Test2	Test3	Avg	Test1	Test2	Test3	Avg
10,000	0.379688	0.355591	0.552314	0.429198	0.534644	0.419004	0.636816	0.530155
100,000	0.240001	0.263433	0.539687	0.347707	0.252887	0.480933	0.489467	0.407762
200,000	0.490237	0.539786	0.502577	0.510867	0.258541	0.400844	0.252203	0.303863
Not Specified	0.510609	0.493688	0.501294	0.501864	0.24695	0.255459	0.555312	0.352574

10 Threads	M =50 x=0.5 y=0.5				M =50 x=0.2 y=0.9			
Chunk Size	Test1	Test2	Test3	Avg	Test1	Test2	Test3	Avg
10,000	0.572639	0.515350	0.516682	0.53489	0.554037	0.517204	0.526164	0.532468
100,000	0.447977	0.443304	0.436022	0.442434	0.455362	0.482523	0.450069	0.462651
200,000	0.443343	0.441646	0.490237	0.458409	0.458462	0.443738	0.451566	0.451255
Not Specified	0.454695	0.448923	0.452473	0.45203	0.451654	0.482172	0.454940	0.462922

Overall, result is not very concrete but after many testing beside the one presented above, in most cases chunk sizes (take note that this is not directly fed to the static schedular, it has been used to compute a relationship that consider n and k as well) of 100,000 works better than the others. The main problem with our measurements is that the clock is not fixed on the CPU end and as a result no results can be compared easily.

3. (10 points) 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. You may choose appropriate values for the grid size to study the features of your implementation.

To calculate flop we first need to find the number of floating point operations in our algorithm and divide that by the amount of time taken to compute the algorithm. Then we can find the flop rating of the intel processor in ADA.

First, we try to count the number of operations in LU factorization. In my LU factorization we have 3 loops. Initially we assume the first loop does not exist. Everything inside the deepest loop run $(n-k)^2$ and everything in the second loop run $(n-k)$. Inside the most inner loop we have one multiplication and one subtraction, that would mean 2 operation. In the second loop we have one division, which would also mean 1 operation.

These add up to a total of

$$\sum_{k=0}^{k=n-1} 2(n-k)^2 + (n-k)$$

Based on wolfram alpha this would be 10,413,541,250 operations for n = 2500 or m = 50.

The multiplication of 1Xn matrix by nX1 matrix at the end is also n multiplication and n addition resulting in another 2500 operations.

So the total operations are 10,413,543,750 FLOP for doubles.

Now we take the best performing time for a m=50 case from table provided in question 2 and as it can be seen 0.240001s seems to be the best record (m=50 x=0.5 y=0.5 and 100,000 chunk size in the program):

10,413,543,750 FLOP / 0.240001s = 43,389,584,835 FLOP/s

Ada nodes have 2 Intel Xeon E5-2670 v2 with 30.97 GFLOPS each based on [2]. So the total flops of an Ada node is ~62 GFLOPS. Now 100 X 43GFLOPS / 62GLOPS = 69%

Now let's compare that to single core performance. For the same m=50 x=0.5 y=0.5 and 100,000 chunk size in the program and the best single thread performance is 4.777377s.

10,413,543,750 FLOP / 4.777377s = 2,179,761,771 FLOP/s

Now to find the speed up: Sp = 43.390/2.180 =19.90

Efficiency = 19.90/20 = 0.995

Although it is important to point out that this number is misleading because the open MP does not make it truly single core, the overhead still exist. While the multi core performance benefit from the overhead the single core does not benefit from it. So a purely single core code would perform better.

[1] https://www.wolframalpha.com/input/?i=%E2%88%91%282%282500-k%29%5E2+%2B+%282500-k%29%29+where+k+from+1+to+2499

[2] https://ranker.sisoftware.co.uk/show_run.php?q=c2ffc9f1d7b6d7eadbead8eddbe9cfbd80b096f396ab9bbdcef3cb&l=en

Appendix Different way of Paralyzing LU:

```
void computeLUMatrixCompactV1(double matrix[], const uint64_t n)
{
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }
    }
}
```

```
void computeLUMatrixCompactV2(double matrix[], const uint64_t n)
{
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait schedule(static,10)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }
    }
}
```

```
void computeLUMatrixCompactV3(double matrix[], const uint64_t n)
{
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait schedule(dynamic)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    matrix[i * n + k] /= matrix[k * n + k];
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] -= (matrix[i * n + k] * matrix[k * n + j]);
                    }
                }
        }
    }
}
```

```
void computeLUMatrixCompactV4(double matrix[], const uint64_t n)
{
    for (uint64_t k = 0; k < n - 1; k++)
    {
        uint64_t i;
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait private(i)
                for (i = k + 1; i < n; i++)
                {
                    matrix[i * n + k] /= matrix[k * n + k];
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] -= (matrix[i * n + k] * matrix[k * n + j]);
                    }
                }
        }
    }
}
```

```
void computeLUMatrixCompactV5(double matrix[], const uint64_t n)
{
    double *tempArray = new double[n];
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait collapse(2)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    // double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    // matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - matrix[i * n + k] / matrix[k * n + k] * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }

        for (uint64_t i = k + 1; i < n; i++)
        {
            matrix[i * n + k] = matrix[i * n + k] / matrix[k * n + k];
        }
    }
}
```

```
void computeLUMatrixCompactV6(double matrix[], const uint64_t n)
{
    double *tempArray = new double[n];
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait collapse(2) schedule(static,10)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    // double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    // matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - matrix[i * n + k] / matrix[k * n + k] * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }
        for (uint64_t i = k + 1; i < n; i++)
        {
            matrix[i * n + k] = matrix[i * n + k] / matrix[k * n + k];
        }
    }
}
```

```
void computeLUMatrixCompactV7(double matrix[], const uint64_t n)
{
    double *tempArray = new double[n];
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait collapse(2) schedule(dynamic)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    // double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    // matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - matrix[i * n + k] / matrix[k * n + k] * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }
        for (uint64_t i = k + 1; i < n; i++)
        {
            matrix[i * n + k] = matrix[i * n + k] / matrix[k * n + k];
        }
    }
}
```

```
void computeLUMatrixCompactV8(double matrix[], const uint64_t n)
{
    #pragma omp parallel
    {
        for (uint64_t k = 0; k < n - 1; k++)
        {
            #pragma omp for
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }
    }
}
```

```
void computeLUMatrixCompactV9(double matrix[], const uint64_t n)
{
    #pragma omp parallel
    {
        for (uint64_t k = 0; k < n - 1; k++)
        {
            #pragma omp for nowait
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }

            #pragma omp barrier
        }
    }
}
```

```
void computeLUMatrixCompactV10(double matrix[], const uint64_t n)
{
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait schedule(static,50)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }
    }
}
```

```
void computeLUMatrixCompactV11(double matrix[], const uint64_t n)
{
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait schedule(static,25)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }
    }
}
```

```
void computeLUMatrixCompactV12(double matrix[], const uint64_t n)
{
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait schedule(static,75)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }
    }
}
```

```
void computeLUMatrixCompactV13(double matrix[], const uint64_t n)
{
    for (uint64_t k = 0; k < n - 1; k++)
    {
        #pragma omp parallel shared(matrix, k, n)
        {
            #pragma omp for nowait schedule(static,100)
                for (uint64_t i = k + 1; i < n; i++)
                {
                    double toBeL = matrix[i * n + k] / matrix[k * n + k];
                    matrix[i * n + k] = toBeL;
                    for (uint64_t j = k + 1; j < n; j++)
                    {
                        matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                    }
                    // matrix[i * n + k] = toBeL;
                }
        }
    }
}
```

```
void computeLUMatrixCompactV14(double matrix[], const uint64_t n)
{
    #pragma omp parallel shared (matrix, n)
    {
        #pragma omp single
        {
            for (uint64_t k = 0; k < n - 1; k++)
            {
                for (uint64_t i = k + 1; i < n; i++)
                {
                    #pragma omp task shared(matrix) firstprivate(i, k, n)
                    {
                        double toBeL = matrix[i * n + k] / matrix[k * n + k];
                        matrix[i * n + k] = toBeL;
                        for (uint64_t j = k + 1; j < n; j++)
                        {
                            matrix[i * n + j] = matrix[i * n + j] - toBeL * matrix[k * n + j];
                        }
                    }
                }
                #pragma omp taskwait
            }
        }
    }
}
```

I have collected logs for all these but I did not have time to process all the data.