Assignment 1

Gohil Dwijesh 2017CS50407
Prafful Ravuri 2017CS10369

January 2020

1 How does our program work?

Our algorithm is mainly divided into two parts. First part being the initialization of different matrices and the second part being the Gaussian Elimination technique to decompose the input matrix into an upper and lower triangular matrix. First part of the algorithm has order(n^2) time complexity.

Until k-1th iteration of outer most for loop:

- k-1 number of columns of the lower triangular matrix has been calculated. Though it may change in the next iterations due to swapping.
- k-1 number of rows of the upper triangular matrix has been calculated. Once calculated, they don't change.

Now consider the kth iteration of the outer most loop in the Gaussian Elimination technique. In the kth iteration:

- Gaussian Elimination algorithm assumes that the first row and first column element is largest compared to the first column elements of a matrix.
- To incorporate this assumption, we first will find out the row index of the same maximum element from a sub-matrix starting from (k,k)th element of bigger input matrix and swap kth row with found row index. Correspondingly we will swap few set of elements from lower matrix.
- We want to make kth column elements of to be zero except (k, k)th and above elements of kth column of the input matrix. We will store all these factors(that if we multiply ith factor with (k, k)th element of the input matrix and subtract it from ith row then the (i, k)th element will be zero) for corresponding rows into lower matrix at appropriate indices.
- Then we will apply elementary row operation and update the input matrix elements:

$$(k+i)^{\text{th}}{}_{\text{row}} = (k+i)^{\text{th}}{}_{\text{row}} - (k^{\text{th}}{}_{\text{row}}) * (\|factor^{\text{i}}\|) \quad \forall i >= 1$$

- We will go for the next iteration.

2 openMP

2.1 Implementation choice

2.1.1 Data structure

For all matrices, we used a vector of pointers. Each pointer is pointing to the vector of double elements. We used this data structure because it saves our order(n^2) time to swap two rows into order(n) time because we just have to swap two pointers. For permutation, we used a vector of integers.

Later we figured out that it would have been faster if we use arrays instead of vectors. It has something to do with caching.

2.2 How does our program partitions data, work and exploit parallelism?

openMP version of the program has in total two parallel blocks. One for matrix initialization and second inside the outer most loop of the Gaussian Elimination algorithm.

Initialization of matrices has order(n^2) time complexity. Typically for n=8000 it is 64000000 iterations. We used the following.

```
#pragma omp parallel for schedule(static, 4) collapse(2)
for(...)
    for(...)
        critical{
            <element initialization>
        }

```

We compared both sequential and parallel version to initialize and found parallel to be efficient.

Second parallel block is inside the outer most loop of the Gaussian Elimination algorithm. The input matrix is modified in each of its iterations so we can not parallelize the outer most loop.

- We tried both parallel and sequential approach to find the maximum element from a column and found sequential to be efficient.
- Except for the for loops, all other instructions (inside the outer most loop body) are done only by one thread. We used the following syntax:

```
#pragma omp single{...}  // provides implicit barrier
#pragma omp master{...}  // no implicit barrier

```

- For each for loops (inside the outer most loop body) we distributed the iteration to each threads. We also used nowait because we figured out that second last for loop and the for loop to swap lower matrix row elements are independent of each other. So it is not necessary for all threads to be joined there. It looks something like the following:

```
#prgma omp parallel{
    #pragma omp master{
        // swap pi vector, u[k][k] = max_element
    }
    #pragma omp for schedule(static, 4) nowait{
        // swap lower matrix
    }
    #pragma omp for schedule(static, 4){
        // update lower and upper matrix elements
    }
    #pragma omp for schedule(static, 4){
        for(...)
            for(...)
                // update input matrix elements
    }
}

```

2.3 How is the parallel work synchronized?

When do we need to synchronize our work? When threads can access and modify the shared resources. The type of matrices in our program is vector<vector<double>*>. That means that, whenever threads try to modify the same row, data races may occur and we need to synchronize it. We synchronized all such cases. Abstractly it is shown below:

```
#pragma omp critical{
    // random initialization of input matrix elements
    // swap lower matrix elements accross threads
    // update kth row elements of upper matrix
    // above tasks are done in separate critical blocks
}

```

2.4 Empirical data

Number of threads	Execution time	#cores used	Parallel efficiency
1	1117.9 sec	1	0.914518293
2	562.28 sec	2	0.909102227
4	519.167 sec	2	0.984596479
8	518.566 sec	2	0.985737592
16	520.092 sec	2	0.982845343
1(seq)	1022.34 sec	1	NA

Table 1: Number of threads and corresponding execution time in seconds. Tested on a quad core processor.

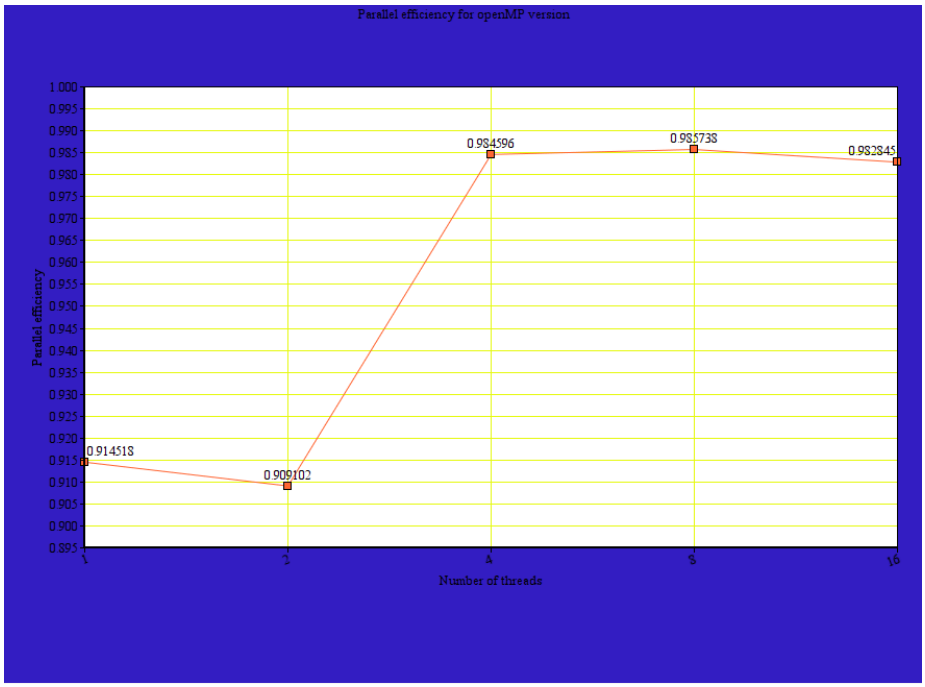

Figure 1: Parallel efficiency v/s number of threads for openMP version of the program

3 pthread

3.1 Implementation choice

3.1.1 Data structure

For all matrices input matrix, lower and upper triangular matrix, we used a two-dimensional discrete array of doubles in the sense that consider a one-dimensional array of size n(= matrix size). Each of the elements of this array points to n sized row arrays. After using vectors in the first part we realized that read and write operation on dynamic memory is costlier than static memory access. We used a one-dimensional array of integers for the permutation vector.

3.2 How does our program partitions data, work, and exploits parallelism?

Our pthread implementation is mainly divided into two parallel blocks. All matrices input matrix, lower and upper triangular matrix and permutation 1D array are shared among all the threads of two parallel blocks.

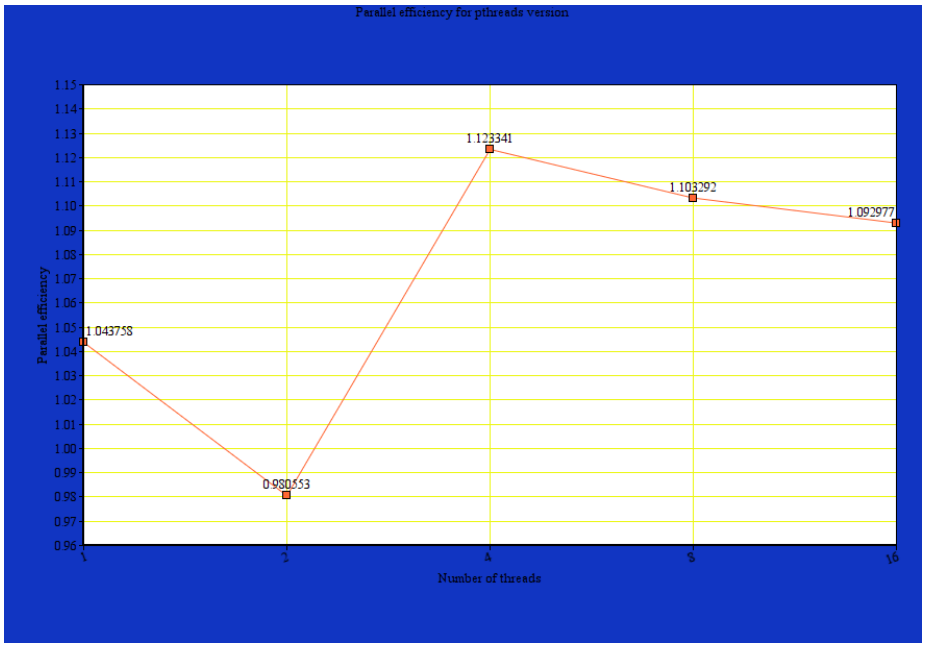

Figure 2: Parallel efficiency v/s number of threads for pthreads version of the program

We parallelized input matrix initialization. The total of n^2 iterations are divided into threads. We implemented the following openmp directive using pthreads.

```
#pragma omp parallel for schedule(static) num_threads(NUM_THREADS)

```

We used second parallel block for swapping lower triangular matrix rows, updating lower and upper triangular column values, and mainly updating sub-matrix of input matrix from a given index (k+1, k+1). For loop iterations are distributed among the threads. We implemented the following openMP directive

```
#pragma omp parallel for schedule(static) num_threads(NUM_THREADS)

```

3.3 How is the parallel work synchronized?

There are two types of synchronization possible: mutex lock and adding an explicit barrier where ever there are some data dependencies. We used 2D array to store each matrix element. Now a[i][j] is implemented as *(a + i*c + j). So read, modify and load operations on different elements of the matrix are independent of each other. So in our implementation, we don't need to use mutex lock. Although we used explicit barrier to make sure that while

modifying the input matrix, lower and upper matrix elements are accessed only after they are modified. Code snippet is given below.

```
// update lower and upper matrix elements
pthread_barrier_wait(&barrier)
// a[i, j] = fxn(lower[p, q], upper[r, s])

```

3.4 Empirical data

Number of threads	Execution time	#cores used	Parallel efficiency
1	671.765 sec	1	1.043757862
2	357.533 sec	2	0.980552844
4	312.087 sec	2	1.123340607
8	317.758 sec	2	1.103292443
16	320.757 sec	2	1.092976926
1(seq)	701.16 sec	1	NA

Table 2: Number of threads and corresponding execution time in seconds. Tested on a quad core processor.