Assignment 1: COL380

Raval Vedant Sanjay
2017CS10366

Shayan Aslam Saifi
2017CS10375

General Setup

- The parallelized implementation of the *LU-Decomposition* was done by the following two methods:
  - Open MP (code_omp.cpp)
  - Pthreads (code_pthreads.cpp)
- To run the *OpenMP* implementation, we need to run the following steps:

```
$ g++ -O3 -std=c++11 -o out_omp -fopenmp code_omp.cpp
$ ./out_omp 8000 4
```

- To run the *Pthreads* implementation, the following commands are required to be given:

```
$ g++ -O3 -std=c++11 -lpthread -o out_pthreads code_pthreads.cpp
$ ./out_pthreads 8000 4
```

- Here, (-O3) is a performance optimization flag, using loop vectorization and SIMD instructions for improvements. Though it comes handy in improving the overall performance, we couldn't observe much improvements with this, since it's already doing most of the optimization job which is supposed to be done by the threaded code.
- Our program is scalable in the sense that it can work on any size of input as well as any number of threads. Giving inadequate parameters to the program will print Error, giving the instructions to the user to provide inputs to the executable.
- All the data and the graphs included in the report are obtained corresponding to an octa-core laptop processor and input data of size 8000

Key Stategies

- The data-type used for the computations was *vector*, the reason being that they have many optimized built-in functions available

- We used the *drand_r* function and a *randBuffer* for generating the random numbers in a parallel way. This worked well with *OpenMP* since it managed the buffers going to different threads but with *Pthreads* that's not the case and involves some complications reducing the performance

- For all our parallel implementations using *Pthreads*, the data would be divided into **T** different sections (where T would be the number of threads), which would be in the range of **`[minVal + ((maxVal-minVal)/T)(threadNo), max(maxVal, minVal + ((maxVal-minVal)/T)(threadNo+1))].`** The parallelizations using *OpenMP* also uses such static data allocation in contiguous blocks

- We used the pseudo-code given for the algorithm and tried to parallelize the different loops involved in it. The outcomes of these different parallelizations are given below:

    - *Outermost k-loop*:
        - This entire loop can't be parallized because it is having data-dependancy within the single iterations of itself as well as from the previous iterations

    - *Calculating the row maximum*:
        - Parallelizing the Row Maximum is easier to do with *OpenMP* as compared to *Pthreads*. The *Critical* directive is used here for locking the *Row Maximum*.
        - Also there are many complications involved for checking the inter-loop dependencies in the case of a *Pthread* program and taking into account that this loop is not a bottleneck for the program, it becomes a fair choice to not parallelize this part using pthreads since the complications involved will only add some overhead to the total performance.

    - *Swapping the entire rows of A matrix*:
        - The built-in function optimizations for swapping an entire vector row makes it better to be done without any parallelization

    - *Swapping the rows of L matrix partially*:
        - This served as an example to why should we not try to parallelize each and every part of the code. Since this part of the program is a computationally very simple, so using threads here would have more overhead for their creation and deletion as compared to the somewhat performance improvement they achieve.
        - So for the *OpenMP* implementation, since two sections will already be parallelized before this loop executes, there will be a

considerable overhead for the thread generation and the other directives, so we didn't parallelize this loop.
        - Though, for the *Pthreads* code, there were no other threads before this section and so it was a good choice to generate the required number of threads here and use them for the subsequent computations (which are computationally expensive)

    - *Updating L and U*:
        - Since this was the most expensive among the O(n) loops, we parallelized it in both the implementations

    - *Computing A from the new L,U*:
        - This loop is a bottleneck for the entire program because of it's size, so it was parallelized in both the implementations. This resulted in most of the performance boost observed
        - For the *OpenMP* implementation, using the built-in module *collapse(2)* showed somewhat improvement in the performance because of the optimized ways of combining the 2 nested for-loops into a single for-loop
        - For *Pthreads,* no such module was there and thus combining the 2 nested for-loops into one didn't show any performance improvement

- This way, we parallelized some of the loops, taking into account the overhead and the improvements involved. Note that, for most the part of the program, there was no issue about synchronization since there were no loop dependencies.

- Note that, *OpenMP* does the work of creating and combining the threads by itself, while for *Pthreads* we were required to go into more details of initializing the different threads, partitioning the data, and joining the threads to combine their results and kill them.

- We even verified our efficiency by using `$ htop` and observing that for *n* number of threads (< totalCores), the CPU would utilize *n* cores at the efficiency of *>99%* during the parallel execution. Thus, the threads divide the work such that each core is at it's maximum utilization.

- We also verified the correctness of our programs by implementing a $O(n^3)$ function for computing the norm as asked and we observed that it was coming in the order of 10^{-12}