COL380 Assignment 1

Santhosh Deshineni : 2021CS10564
Pothamsetty Chethan Manogna Sai : 2021CS10561
Kashish Goel : 2021CS50602

February 2024

1 Parallelizing the sequential code

1.1 Overview of the sequential code

The main components of the algorithm involved the following :-

- Initialising a matrix of floating point numbers
- Performing the LU Decomposition comprising of the below components :-
    - Finding the largest element in a particular column
    - Swapping the rows in the matrices
    - Modifying the matrix entries for A,L,U
- Computing the error by calculating the Euclidean $L_{2,1}$ norm of the columns of the residual matrix.

1.2 Parallelizing Initialisation of the matrix

The initialisation of the matrix for which the decomposition is performed could be parallelised and the same was done in both pthread and OpenMP implementations.
To generate random numbers, `drand48()` being a thread safe function was used. To ensure that different threads don't generate the same sequence of numbers, random seed is made a **private variable** with each thread getting a different seed based on thread number ensuring that same numbers are not generated. During parallelisation, each thread is given contiguous rows to initialise for both pthreads and OpenMP.

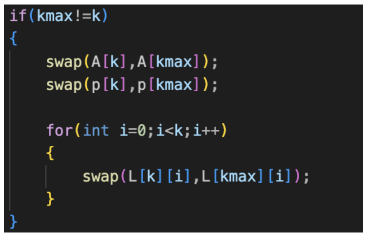
1.3 Parallelizing components in LU Decomposition

In the LU Decomposition function, we process the matrix column by column wherein we first find out the current maximum element in the column.

OpenMP: A team of threads is forked using the `parallel for` directive every time we need to parallelise a portion of the code.
PThreads: In pthreads, unlike openMP, there isn't any innate barrier and hence we need to put a barrier explicitly which has a lot of overhead associated with it. Due to this, we have to either create different team of threads every time or if using the same then use barriers. In our implementation of pthreads we have hence tried to maintain an optimal version with the right balance of overhead and parallelisation.

- **Parallelize finding max element in a column:** In order to parallise this step, we would have to use locks on the variable used to store the max element and the max index. The same needs to be done because of the presence of **data dependencies** in the loop. This could be done by using a mutex in case of pthreads and making the max variable shared in OpenMP after making the appropriate portion of code 'critical section'. However, doing the same was leading to increased time due to **overheads of acquiring and releasing the locks**. It was hence realised that it would be better to not parallelise this portion of the code.

- **Parallelize swapping of rows in matrix:** The portion of code as depicted in the code snippet below involves a pointer swap (among rows of A) which is an atomic operation made possible because of the design choice of array of pointers. The inner loop can be parallelised.

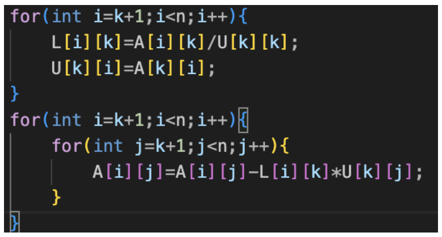
  

  ```
  if(kmax!=k)
  {
      swap(A[k],A[kmax]);
      swap(p[k],p[kmax]);

      for(int i=0;i<k;i++)
      {
          swap(L[k][i],L[kmax][i]);
      }
  }
  ```

  OpenMP : The same has been implemented in openMP using the `parallel for` directive.
  Pthreads: The same wasn't done for pthreads because of the overhead in creation and and destroying of threads.

- **Parallelize the matrix edit operations:** The below sequential part of code (in the code snippet) where the matrices' indexes have been changed are implemented as below:



```
for(int i=k+1;i<n;i++){
    L[i][k]=A[i][k]/U[k][k];
    U[k][i]=A[k][i];
}
for(int i=k+1;i<n;i++){
    for(int j=k+1;j<n;j++){
        A[i][j]=A[i][j]-L[i][k]*U[k][j];
    }
}
```

OpenMP: Both the loops were parallelised. Each thread was assigned a contiguous set of iterations as stated by the default scheme of scheduling in OPENMP.
PThreads: Parallelising both the loops was adding a lot of overhead. This is because, as stated previously, unlike openMP, to synchronize in pthreads we need to add barriers which has a lot of overhead associated with it. We hence chose to parallelize the second loop rather than the 1st. This was done because in the 1st loop, the cache utility is lower than that in the 2nd loop.

2 Implementation Choices

Storing the matrices: The matrices are stored as 2d arrays using array of pointers to 1-d arrays. This was done because the individual array rows are stored in contiguous memory locations, which can lead to better cache locality and improved memory access patterns whereas each vector in the vector of vectors will involve dynamic resizing and bound checking which may cause further overheads. In fact, a significant speedup from about 1270 seconds to 410 seconds in the serial code itself was observed when migrating from vectors to array of pointers.

Partitioning the data: While partitioning the matrix data for different threads, each thread is assigned a contiguous set of rows. This is done by using the default static scheduling of OPENMP, and assigning $\frac{total_iterations}{numberofthreads}$ rows to each thread starting from $\frac{total_iterations}{numberofthreads} * threadrank$ in pthreads.

Synchronizing the parallel Work: In OpenMP, there is an implicit barrier and hence any piece of code that is run by multiple threads, will be synchronized itself. For achieving synchronisation in PThreads, `pthread_join` was used.
False sharing and Cache coherence: The work allocated to individual threads is row-wise and the rows themselves are contiguous. This ensures that the thread on its own is cache efficient. Further, false sharing is deterred as no other threads access this row.

3 Timings and speedups for various programs

Input size: 8000*8000 matrix with floating point nos. in the range of 0 to 1.
Time taken for serial code: 409.706 seconds (We observe that the single thread implementations of OpenMP and Pthreads take slightly longer than the serial code because of some overhead of thread creation)

The time taken for parallel execution in seconds is as below. The euclidean norms obtained were in the 10^{-11} to 10^{-9} range.

Number of Threads ->	1	2	4	8	16
OPENMP	410.376	213.457	122.605	107.745	110.562
PTHREADS	460.941	238.784	145.151	125.973	126.234

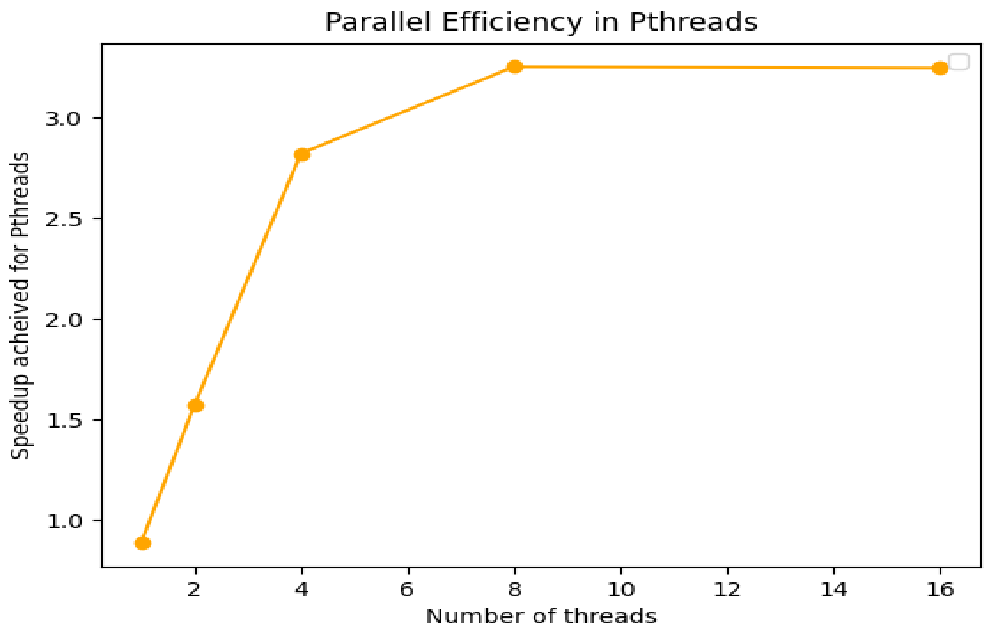
Table 1: Time vs number of threads

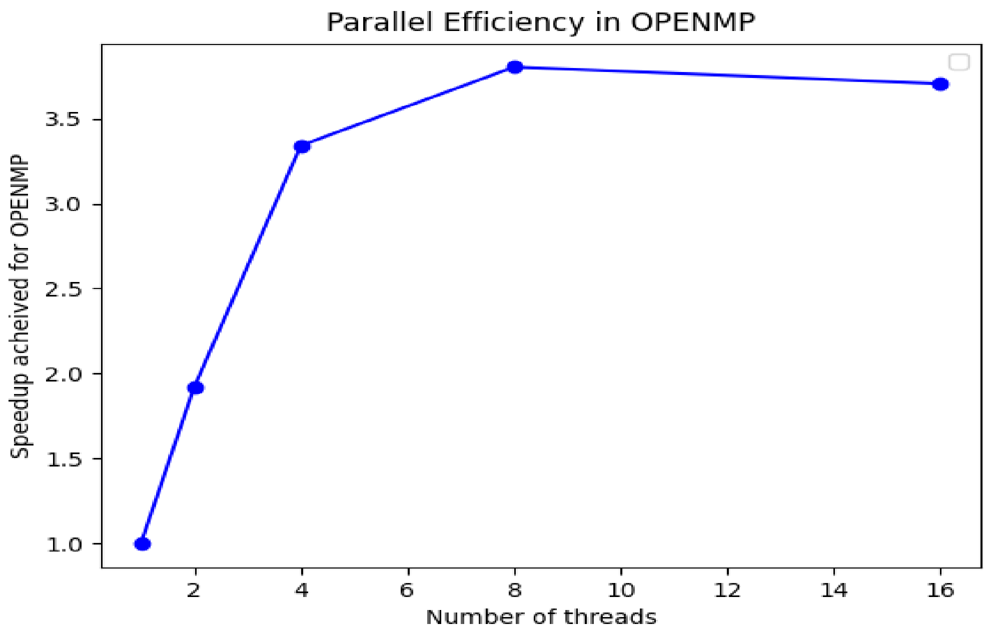
The speedup achieved for parallel execution is:-

Number of Threads ->	1	2	4	8	16
OPENMP	0.998	1.919	3.341	3.803	3.705
PTHREADS	0.888	1.572	2.822	3.2523	3.2456

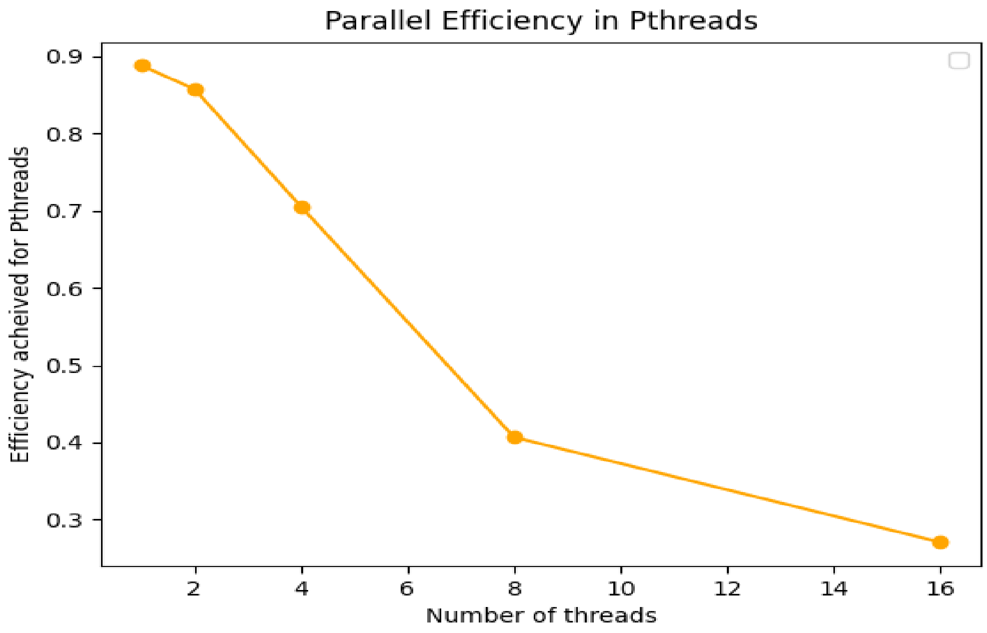
Table 2: Speedup vs Number of threads

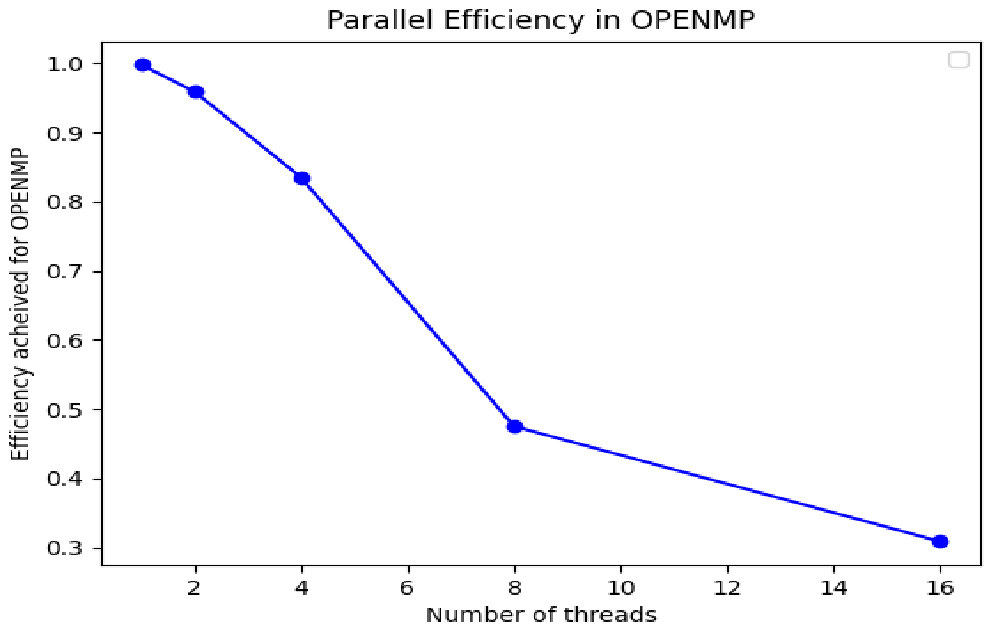
Below are the graphs depicting the speedup for pthreads and openmp





Below are the graphs depicting the efficiency for pthreads and openmp





Observations are as follows:-

- Achieved the peak of speedup for 8 threads
- Achieved greater speedup for OPENMP than for Pthreads

3.1 Machine Specifications

The machine used to run the various programs and note down the timings was a Macbook Air M2 with 8 cores. We can clearly observe that the maximum speedup achieved is with 8 threads which appears to match up with the hardware given the various overheads of pthreads and OpenMP. The verification function was also parallelized for the OpenMP case.