COL 380

Introduction to Parallel and Distributed Programming

Assignment – 1

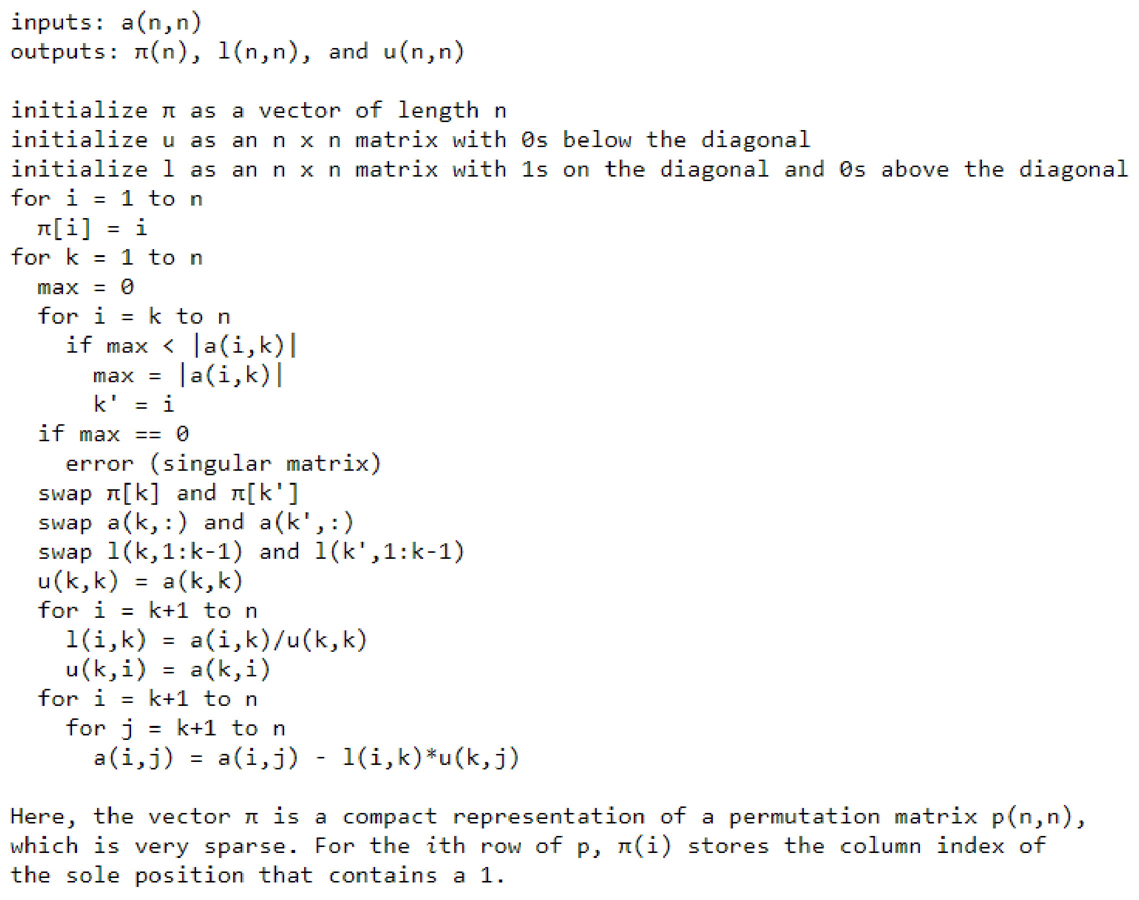
TEAM MEMBERS:

1. Banoth Chethan Naik - 2020CS10333
2. Varanasi Yaswanth Krishna - 2020CS10406
3. Vaddi Aditya - 2020CS50446

Introduction:

LU decomposition, a pivotal technique in numerical linear algebra, dissects a square matrix into lower and upper triangular matrices, facilitating efficient solving of linear equations and matrix operations. This assignment focuses on crafting two parallel implementations of LU decomposition with row pivoting, utilizing shared-memory programming models - Pthreads and OpenMP.

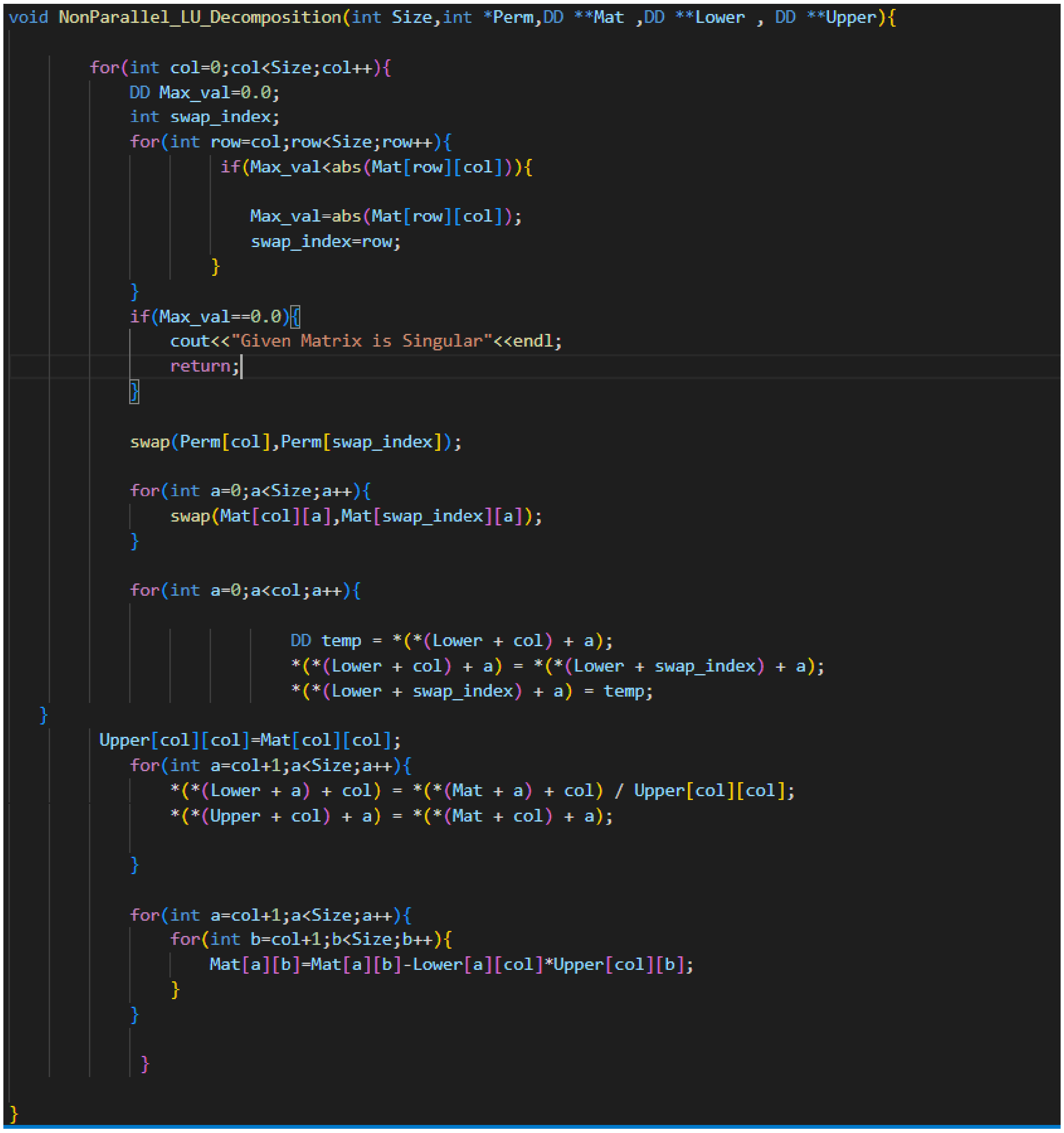
Serial Code:



```
inputs: a(n,n)
outputs: π(n), l(n,n), and u(n,n)

initialize π as a vector of length n
initialize u as an n x n matrix with 0s below the diagonal
initialize l as an n x n matrix with 1s on the diagonal and 0s above the diagonal
for i = 1 to n
  π[i] = i
for k = 1 to n
  max = 0
  for i = k to n
    if max < |a(i,k)|
      max = |a(i,k)|
      k' = i
  if max == 0
    error (singular matrix)
  swap π[k] and π[k']
  swap a(k,:) and a(k',:)
  swap l(k,1:k-1) and l(k',1:k-1)
  u(k,k) = a(k,k)
  for i = k+1 to n
    l(i,k) = a(i,k)/u(k,k)
    u(k,i) = a(k,i)
  for i = k+1 to n
    for j = k+1 to n
      a(i,j) = a(i,j) - l(i,k)*u(k,j)
```

Here, the vector π is a compact representation of a permutation matrix p(n,n), which is very sparse. For the ith row of p, π(i) stores the column index of the sole position that contains a 1.

The below is the code for sequential LU decomposition



```
void NonParallel_LU_Decomposition(int Size,int *Perm,DD **Mat ,DD **Lower , DD **Upper){

        for(int col=0;col<Size;col++){
            DD Max_val=0.0;
            int swap_index;
            for(int row=col;row<Size;row++){
                     if(Max_val<abs(Mat[row][col])){

                        Max_val=abs(Mat[row][col]);
                        swap_index=row;
                     }
            }
            if(Max_val==0.0){
                cout<<"Given Matrix is Singular"<<endl;
                return;
            }

            swap(Perm[col],Perm[swap_index]);

            for(int a=0;a<Size;a++){
                swap(Mat[col][a],Mat[swap_index][a]);
            }

            for(int a=0;a<col;a++){

                            DD temp = *(*(Lower + col) + a);
                            *(*(Lower + col) + a) = *(*(Lower + swap_index) + a);
                            *(*(Lower + swap_index) + a) = temp;
    }
         Upper[col][col]=Mat[col][col];
            for(int a=col+1;a<Size;a++){
                *(*(Lower + a) + col) = *(*(Mat + a) + col) / Upper[col][col];
                *(*(Upper + col) + a) = *(*(Mat + col) + a);

            }

            for(int a=col+1;a<Size;a++){
                for(int b=col+1;b<Size;b++){
                    Mat[a][b]=Mat[a][b]-Lower[a][col]*Upper[col][b];
                }
            }

             }

}
```

Implementation and Design:

- We chose to employ pointers to represent 2-D arrays as the underlying data structure for defining matrices. This choice was made in favour of dynamic allocation using vectors, as it positively influenced execution time and facilitated the eventual decomposition of matrices.

- The utilization of an array of pointers was considered advantageous as it reduced the complexity of swapping entire columns of the input matrix from O(n) to O(1). However, this benefit is more pronounced for input matrices of relatively smaller sizes.

- To mitigate the risk of memory corruption or segmentation faults, every pointer was diligently freed after use.

Pthreads Implementation:

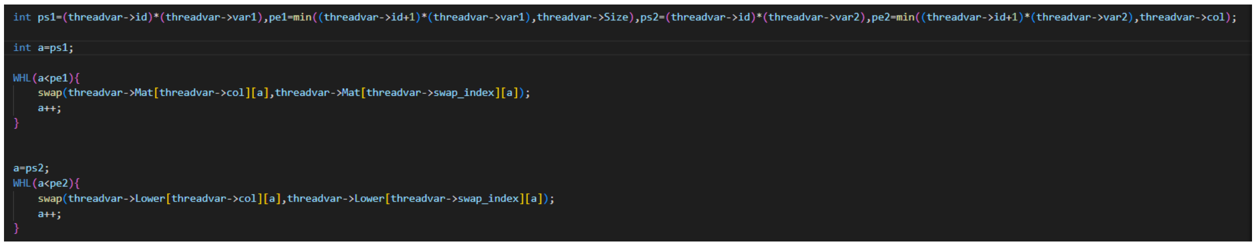
In the serial implementation we used the Threads at some instances where we can divide the work into the threads the below are the code snippets.

Like We also maintained the thread structure so that we can divide the can provide the start and end points where each thread can perform the task.

Each thread will do the respective task which is provided to it and later I joined the every thread at some instants .

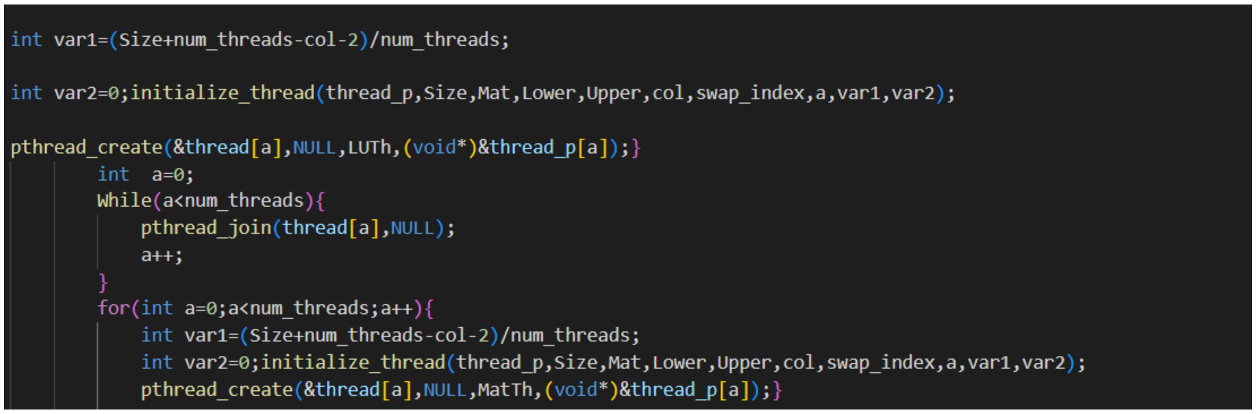
Swapping the Given Matrix elements and Lower triangular elements

I used the threads here



```
int ps1=(threadvar->id)*(threadvar->var1),pe1=min((threadvar->id+1)*(threadvar->var1),threadvar->Size),ps2=(threadvar->id)*(threadvar->var2),pe2=min((threadvar->id+1)*(threadvar->var2),threadvar->col);

int a=ps1;

WHL(a<pe1){
    swap(threadvar->Mat[threadvar->col][a],threadvar->Mat[threadvar->swap_index][a]);
    a++;
}


a=ps2;
WHL(a<pe2){
    swap(threadvar->Lower[threadvar->col][a],threadvar->Lower[threadvar->swap_index][a]);
    a++;
}
```

When finding the values of I also added the parallelism

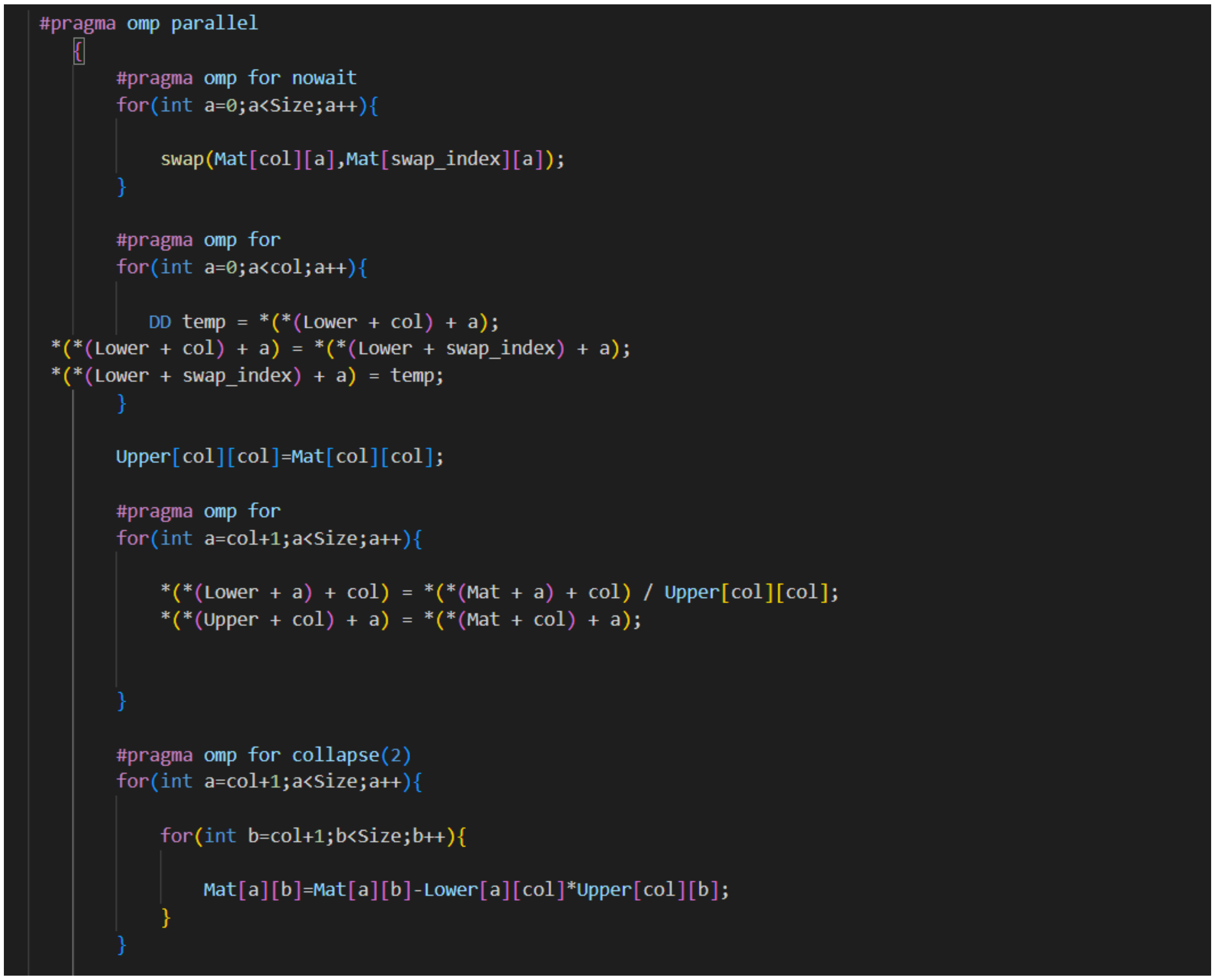


```
int var1=(Size+num_threads-col-2)/num_threads;

int var2=0;initialize_thread(thread_p,Size,Mat,Lower,Upper,col,swap_index,a,var1,var2);

pthread_create(&thread[a],NULL,LUTh,(void*)&thread_p[a]);}
        int  a=0;
        While(a<num_threads){
            pthread_join(thread[a],NULL);
            a++;
        }
        for(int a=0;a<num_threads;a++){
            int var1=(Size+num_threads-col-2)/num_threads;
            int var2=0;initialize_thread(thread_p,Size,Mat,Lower,Upper,col,swap_index,a,var1,var2);
            pthread_create(&thread[a],NULL,MatTh,(void*)&thread_p[a]);}
```

OpenMP Implementation:

In the serial Implementation we used the pragma directive for the for loops and also added some barrier points to get the above work done after only we need to proceed after the series of checks we fixed the pragma directives.

We also added the pragma parallel directive according to the given threads it will divide the work.



```
#pragma omp parallel
    {
        #pragma omp for nowait
        for(int a=0;a<Size;a++){

            swap(Mat[col][a],Mat[swap_index][a]);
        }

        #pragma omp for
        for(int a=0;a<col;a++){

           DD temp = *(*(Lower + col) + a);
 *(*(Lower + col) + a) = *(*(Lower + swap_index) + a);
 *(*(Lower + swap_index) + a) = temp;
        }

        Upper[col][col]=Mat[col][col];

        #pragma omp for
        for(int a=col+1;a<Size;a++){

            *(*(Lower + a) + col) = *(*(Mat + a) + col) / Upper[col][col];
            *(*(Upper + col) + a) = *(*(Mat + col) + a);


        }

        #pragma omp for collapse(2)
        for(int a=col+1;a<Size;a++){

            for(int b=col+1;b<Size;b++){

                Mat[a][b]=Mat[a][b]-Lower[a][col]*Upper[col][b];
            }
        }
```

Observations (Tables and Graphs):

We list the table and on various randomly generated matrices and done the thread and size variations. The below are the listed table and graphs

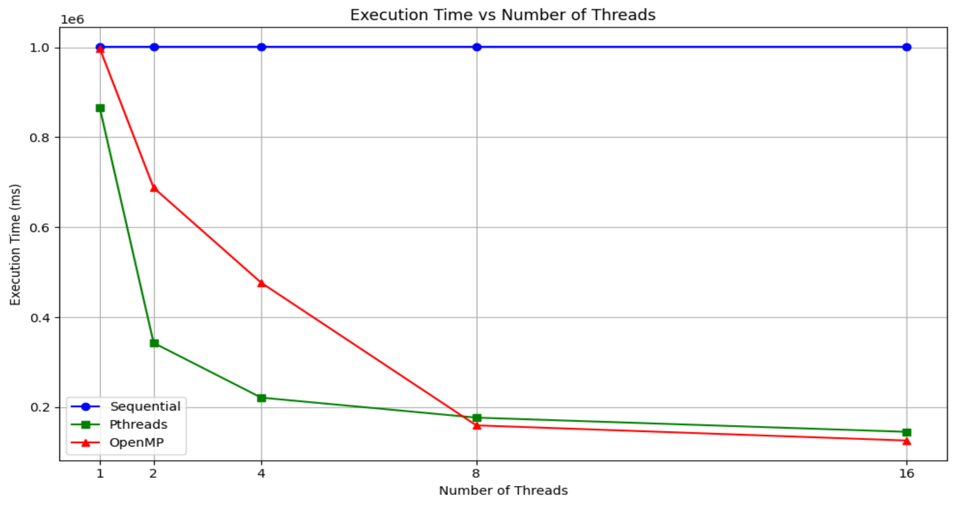
Pthread:

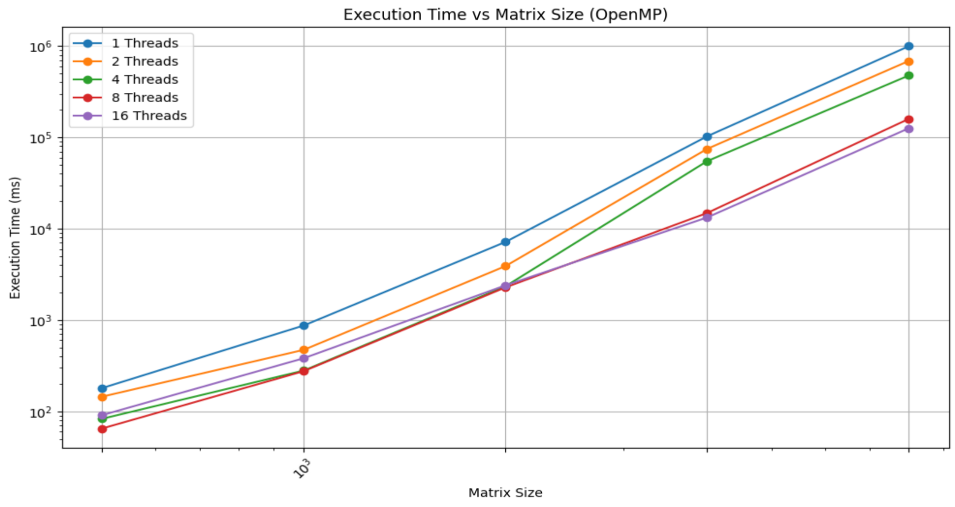
Matrix Size	Threads	Execution Time(ms)
500	1	**300**
	2	**140**
	4	**98**
	8	**132**
	16	**181**
1000	1	**2712**
	2	**976**
	4	**540**
	8	**764**
	16	**848**
2000	1	**15832**
	2	**6553**
	4	**4312**
	8	**3856**
	16	**4112**
4000	1	**110456**
	2	**52332**
	4	**48795**
	8	**13846**
	16	**16327**
8000	1	**865000**
	2	**342983**
	4	**221374**
	8	**176897**
	16	**145366**

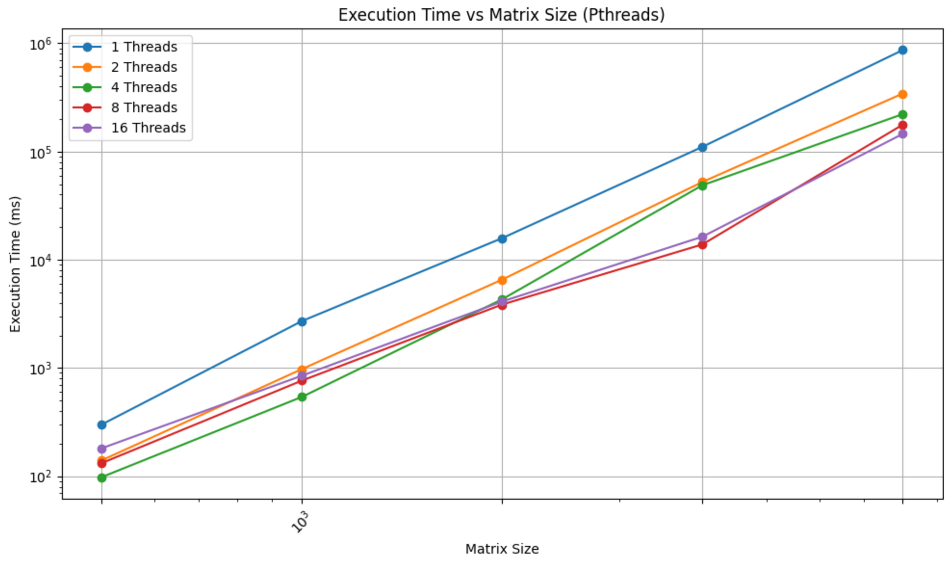
Open MP:

Matrix Size	Threads	Execution Time(ms)
500	1	**180**
	2	**145**
	4	**83**
	8	**65**
	16	**91**
1000	1	**872**
	2	**472**
	4	**280**
	8	**276**
	16	**382**
2000	1	**7165**
	2	**3896**
	4	**2342**
	8	**2279**
	16	**2398**
4000	1	**102848**
	2	**74873**
	4	**54972**
	8	**14875**
	16	**13289**
8000	1	**997327**
	2	**687873**
	4	**476788**
	8	**159700**
	16	**125987**

Graphs:


Execution Time vs Number of Threads
1e6
1.0
0.8
0.6
0.4
0.2
Execution Time (ms)
Sequential
Pthreads
OpenMP
1
2
4
8
16
Number of Threads


Execution Time vs Matrix Size (OpenMP)
1 Threads
2 Threads
4 Threads
8 Threads
16 Threads
10^6
10^5
10^4
10^3
10^2
Execution Time (ms)
10^3
Matrix Size


Execution Time vs Matrix Size (Pthreads)
1 Threads
2 Threads
4 Threads
8 Threads
16 Threads
10^6
10^5
10^4
10^3
10^2
Execution Time (ms)
10^3
Matrix Size

Numerical Stability:

We checked with the L21verification. Performed the L21 decomposition on the given matrix A to obtain matrices L and U.

We computed the values and checked whether it is less than the bound.

We checked the values on our randomly generated matrices of size 100,500,1000,2000,3000,4000,5000,6000,7000,8000.All are below the bound.