COL380 Assignment 1

Aastha Rajani (2021CS10093), Disha Shiraskar (2021CS10578),
Srushti Junghare (2021CS50613), Priyadarshini Radhakrishnan (2021CS50614)

February 15, 2024

1 Introduction

LU decomposition is a widely used technique in linear algebra for solving systems of linear equations and matrix inversion. It decomposes a square matrix into lower and upper triangular matrices. Parallelizing LU decomposition can significantly enhance its performance, especially for large matrices. In this document, we describe a parallel implementation of LU decomposition using OpenMP and Pthreads. The algorithm contains the following parts :

1. Matrix Initialization: Generate a random square matrix A of size n x n.
2. Matrix Decomposition: Decompose A into lower triangular matrix L and upper triangular matrix U such that A = LU.
3. Partial Pivoting: Perform partial pivoting to improve numerical stability and prevent division by zero.
4. Parallelization: Parallelize the computation of L and U using OpenMP directives to exploit multi-core architectures.
5. Clean-up: Free dynamically allocated memory.

2 Possible Parallelization Strategies

Considering the given algorithm, we could parallelize following sections of the serial code:

1. Initialisation of matrix A,L,U,pi - We can parallelise this part by dividing the rows of each matrix evenly among the number of threads available. This section does not have any loop carried dependicies or critical sections to take care of.
2. The outer for loop (first for loop after initialisation of pi in the serial code) cannot be parallelised since it has more than 1 point of exit.
3. Finding maximum entry in the Column - We can divide the entire column evenly among the threads such that each thread has its private variables for max and k-max (row with the max element). After execution of all the threads, the overall global max and global k-max can be evaluated. Note that this section may or may not have any critical sections and it may require a barrier to ensure that all the threads have completed computing their private maximas (depending on the implementation).
4. Swapping two elements of the pi matrix is a single-step operation and does not require any parallelism.
5. Swapping rows of matrix A can be parallelized by dividing the total entries to be swapped evenly among the given number of threads. Similar logic for swapping rows of matrix L. Furthermore, we can also perform swapping of A and L parallelly since the operations executed in both the swappings are independent of each other.

6. For loop updating entries in L and U - In this for loop, one must note that that updates in L and U are completely independent of each other since l(i,k) uses u(k,k) but the statement 'u(k,i) = a(k,i)' never updates u(k,k) since i runs from k+1 to n. Therefore we may parallelize these updates.

7. Gaussian elimination step- The last for loop which updates A's entries can also be parallelized by dividing the iterations evenly among the threads. This section does not have any loop carried dependencies or critical sections to take care of.

In each of the possible strategies discussed above we must check for the conditions of false sharing before implementing parallelization.

3 Other Details

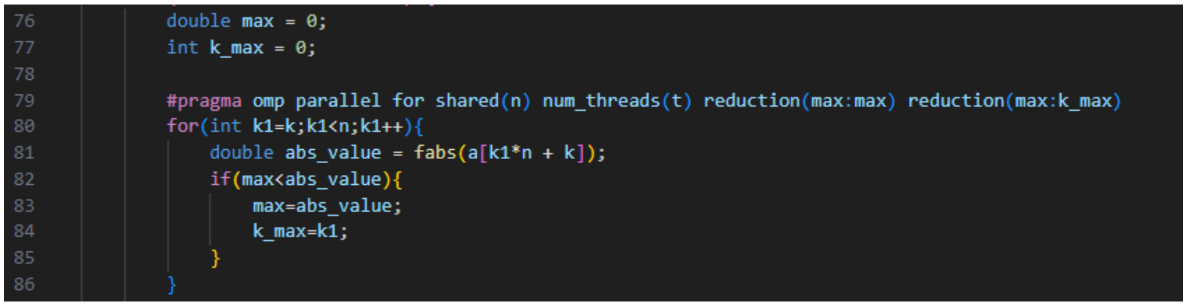
We have considered following points while implementing parallelism in OpenMP and Pthreads:

1. Using a Contiguous Layout for the Array -
   False sharing occurs when multiple threads modify different variables that reside on the same cache line. With a contiguous layout, where elements of the array are stored consecutively in memory, the likelihood of false sharing is minimized. Contiguous memory layout facilitates better cache utilization, as accessing neighboring elements of the array within the same cache line can result in fewer cache misses.

2. Creating matrices using pointers -
   We used double $**$ to represent 2D matrices and double $*$ for 1D matrix. This helped pass matrices as pointers in function arguments to avoid duplicacy.. Instead of swapping rows element by element, we instead swapped addresses of these rows (double) in O(1) time.

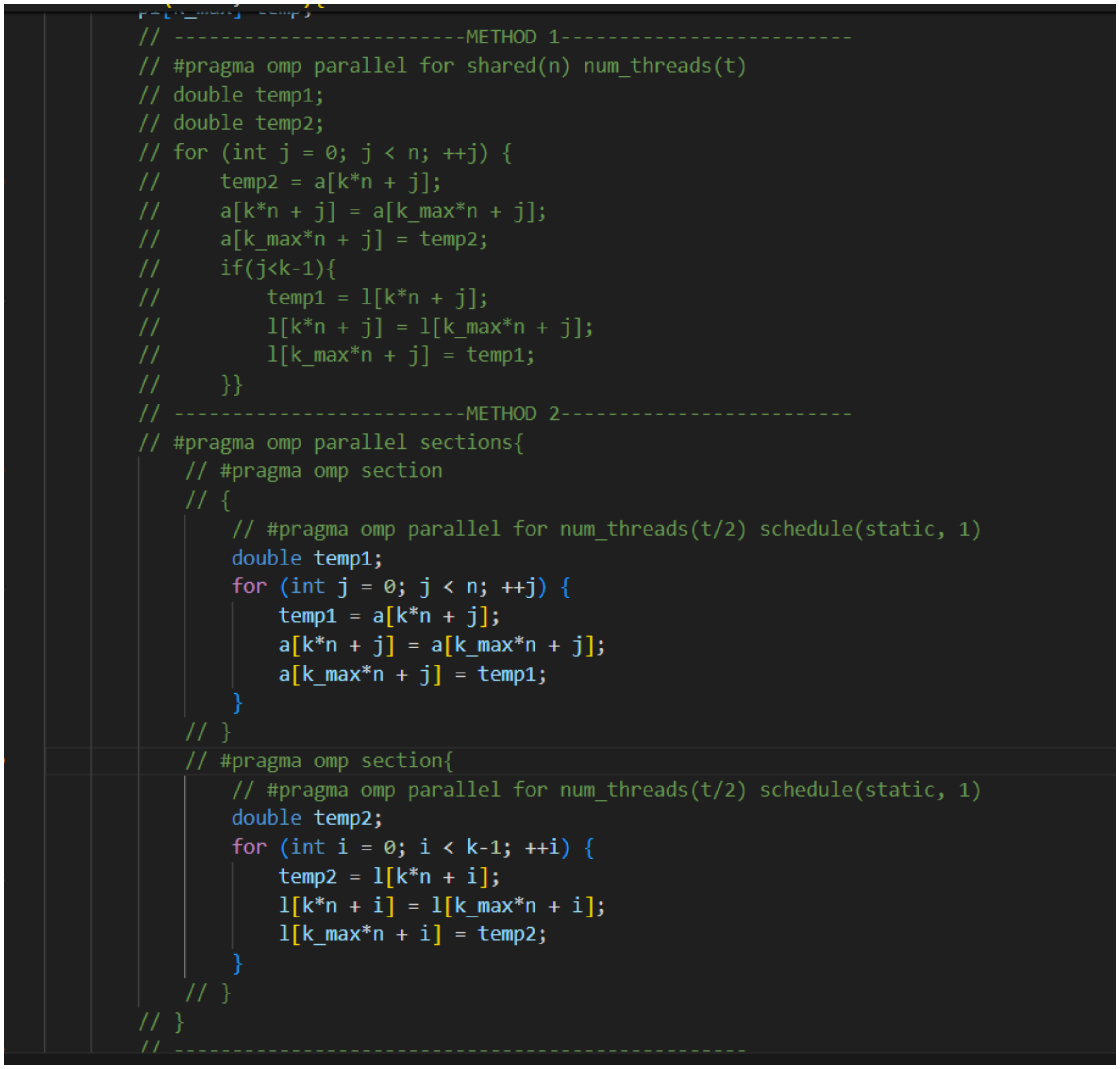
3. We ensured correctness by verifying the LU Decomposition by calculating L21Norm and ensuring that it is zero.

4 Parallelization in OpenMP



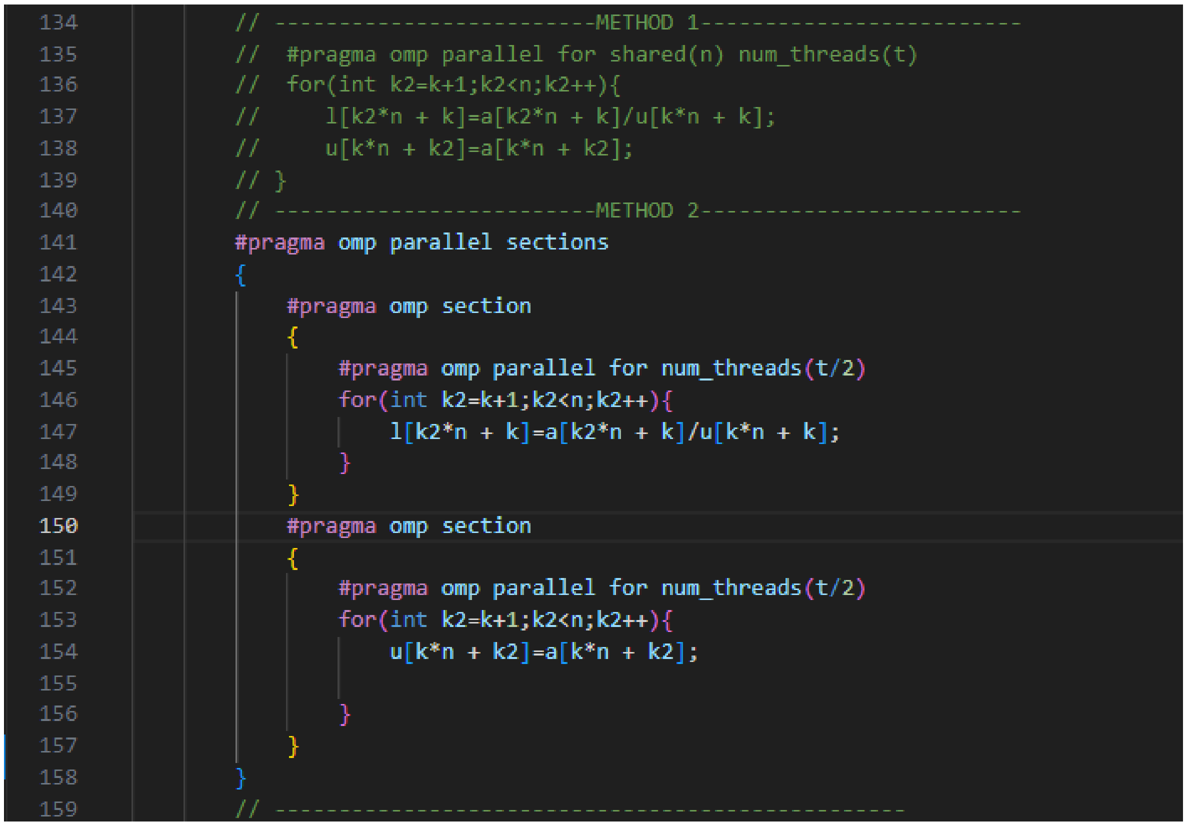
```
double max = 0;
int k_max = 0;

#pragma omp parallel for shared(n) num_threads(t) reduction(max:max) reduction(max:k_max)
for(int k1=k;k1<n;k1++){
    double abs_value = fabs(a[k1*n + k]);
    if(max<abs_value){
        max=abs_value;
        k_max=k1;
    }
}
```

- This code parallelizes finding the maximum absolute value in a matrix column using OpenMP. Threads run simultaneously, iterating through the column and updating a global maximum value with any larger value they find. Reduction clauses ensure the final maximum and its corresponding index are shared across all threads, providing the overall maximum element.



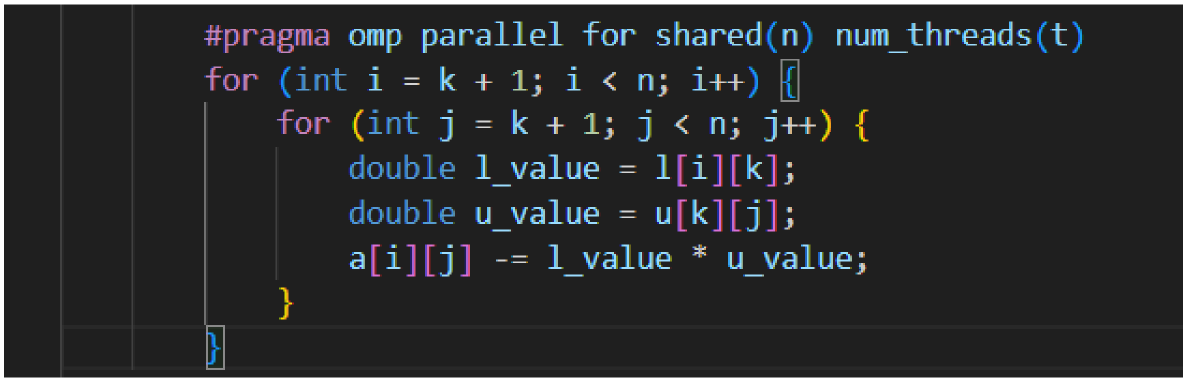
```
// -------------------------METHOD 1-------------------------
// #pragma omp parallel for shared(n) num_threads(t)
// double temp1;
// double temp2;
// for (int j = 0; j < n; ++j) {
//     temp2 = a[k*n + j];
//     a[k*n + j] = a[k_max*n + j];
//     a[k_max*n + j] = temp2;
//     if(j<k-1){
//         temp1 = l[k*n + j];
//         l[k*n + j] = l[k_max*n + j];
//         l[k_max*n + j] = temp1;
//     }}
// -------------------------METHOD 2-------------------------
// #pragma omp parallel sections{
    // #pragma omp section
    // {
        // #pragma omp parallel for num_threads(t/2) schedule(static, 1)
        double temp1;
        for (int j = 0; j < n; ++j) {
            temp1 = a[k*n + j];
            a[k*n + j] = a[k_max*n + j];
            a[k_max*n + j] = temp1;
        }
    // }
    // #pragma omp section{
        // #pragma omp parallel for num_threads(t/2) schedule(static, 1)
        double temp2;
        for (int i = 0; i < k-1; ++i) {
            temp2 = l[k*n + i];
            l[k*n + i] = l[k_max*n + i];
            l[k_max*n + i] = temp2;
        }
    // }
// }
// --------------------------------------------------
```

- For this section of code, we have tried two methods two parallelize it. The first method involves parallelizing it using the parallel for directive in which iterations of the loop are divided among t threads and by carefully checking the conditions of index, we are swapping the elements of rows of a and l within same iteration. Since this involves access to both l and a matrix, it leads to lot of cache misses. Therefore we decided to swap the elements of a and l in different loops.Further, these two swaps are independent of each other, so we have used parallel sections, so that they can do it simultaneously and alloted t/2 threads to each section in method 2.



```
// -------------------------METHOD 1-------------------------
//  #pragma omp parallel for shared(n) num_threads(t)
//  for(int k2=k+1;k2<n;k2++){
//     l[k2*n + k]=a[k2*n + k]/u[k*n + k];
//     u[k*n + k2]=a[k*n + k2];
// }
// -------------------------METHOD 2-------------------------
#pragma omp parallel sections
{
    #pragma omp section
    {
        #pragma omp parallel for num_threads(t/2)
        for(int k2=k+1;k2<n;k2++){
            l[k2*n + k]=a[k2*n + k]/u[k*n + k];
        }
    }
    #pragma omp section
    {
        #pragma omp parallel for num_threads(t/2)
        for(int k2=k+1;k2<n;k2++){
            u[k*n + k2]=a[k*n + k2];

        }
    }
}
// -------------------------------------------------
```

- We tried 2 ways to parallelize this section of code. The image demonstrates both the methods. In Method 1, we tried using simply the parallel for directive for t threads. In Method 2, we used the section directive of OpenMP which allows running two different sections in parallel. The code was broken into two separate for loops- one for updating L and other for U. Each of these for loops were assigned different section each with t/2 threads. Method 2 might have better cache utilization as reduces false sharing by ensuring threads within a section only access data relevant to their task (either l or u) and improves cache locality.



```
#pragma omp parallel for shared(n) num_threads(t)
for (int i = k + 1; i < n; i++) {
    for (int j = k + 1; j < n; j++) {
        double l_value = l[i][k];
        double u_value = u[k][j];
        a[i][j] -= l_value * u_value;
    }
}
```

- We have parallelized this section of code using parallel for directive in openmp. Since, each of the entry a[i][j] can be updated independent of other a[i][j]'s, this section can therefore be parallelized as much as possible and therefore,total iterations of both the loops are equally divided among t threads. We also tried using the collapse directive which equally divides the total iterations among the threads but it resulted in taking even more time. Therefore we decided to not use the collapse directive.

After trying all these possible parallelizations, we concluded that, for all code sections except the last one (nested for loops which involve update to a), the overhead of parallelization for a large input size and large number of threads is much greater than the reduction in time which parallelization causes.Moreover, the last section contains the Gaussian elimination step, which is the main bottleneck in the algorithm (taking up 97% of the execution time) has general form of triple-nested loop Therefore, to optimise our implementation, we have parallelized only the last section of the code.

5 Analysis of OpenMP Implementation

Thread Count	Matrix Size	Execution Time (secs)
1	500	0.148
	1000	1.161
	2000	9.675
	4000	91.645
	8000	720.523
2	500	0.094
	1000	0.677
	2000	6.397
	4000	68.008
	8000	562.575
4	500	0.066
	1000	0.596
	2000	4.737
	4000	69.752
	8000	337.497
8	500	0.072
	1000	0.592
	2000	4.167
	4000	62.267
	8000	290.890
16	500	0.2
	1000	0.785
	2000	4.432
	4000	30.339
	8000	285.334

Table 1: Run Times of LU Decomposition code using varied number of threads and size of A Matrix optimised using OpenMP

5.1 Performance Graphs

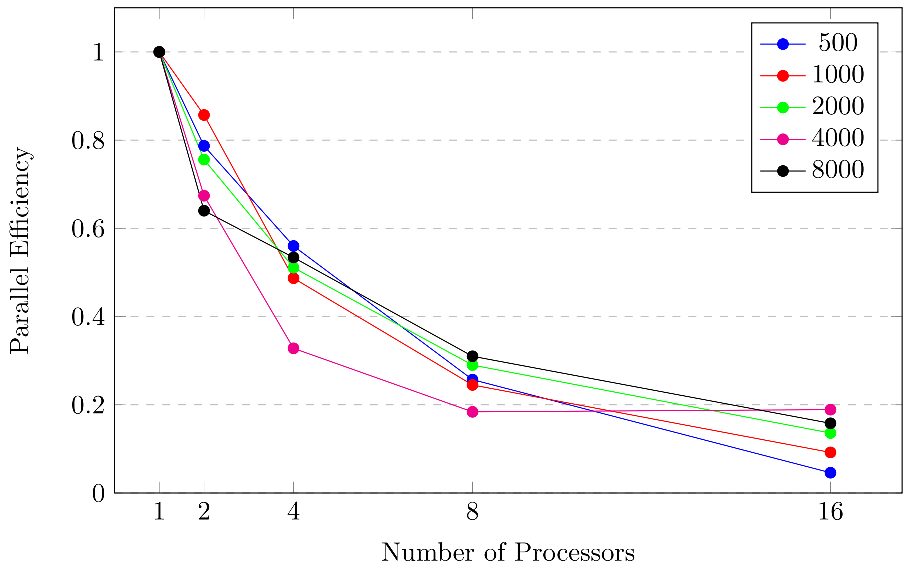


Figure 1: Parallel Efficiency vs. Number of Processors

5.2 Speedup Graphs

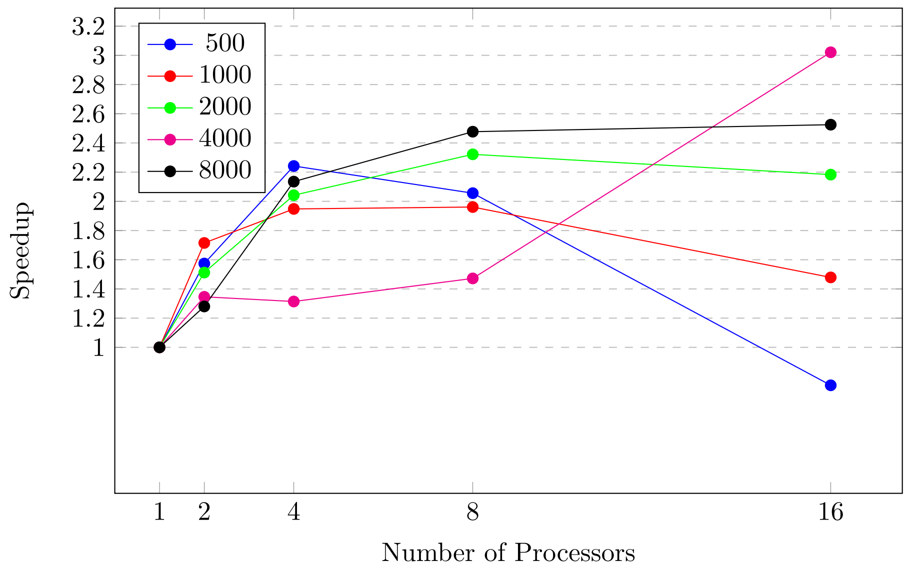
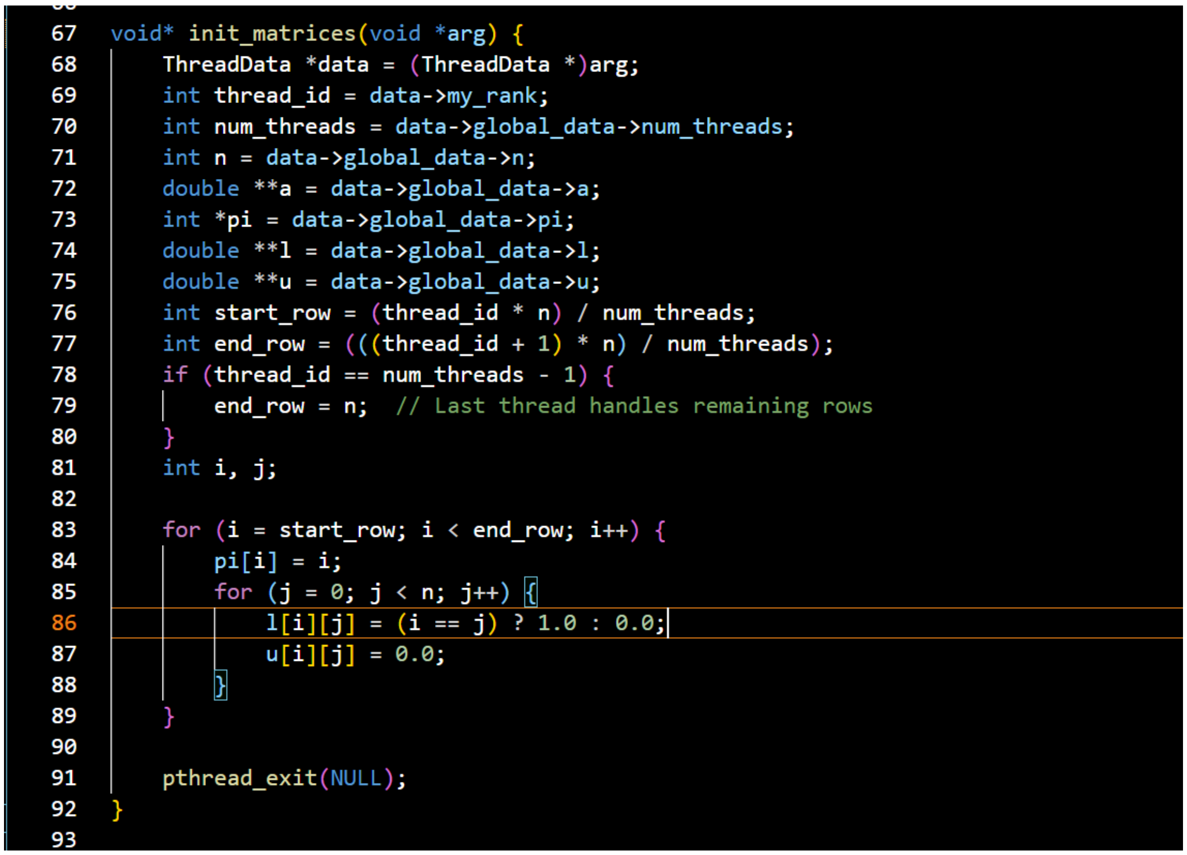


Figure 2: Parallel Efficiency vs. Number of Processors

6 Parallelization in Pthreads

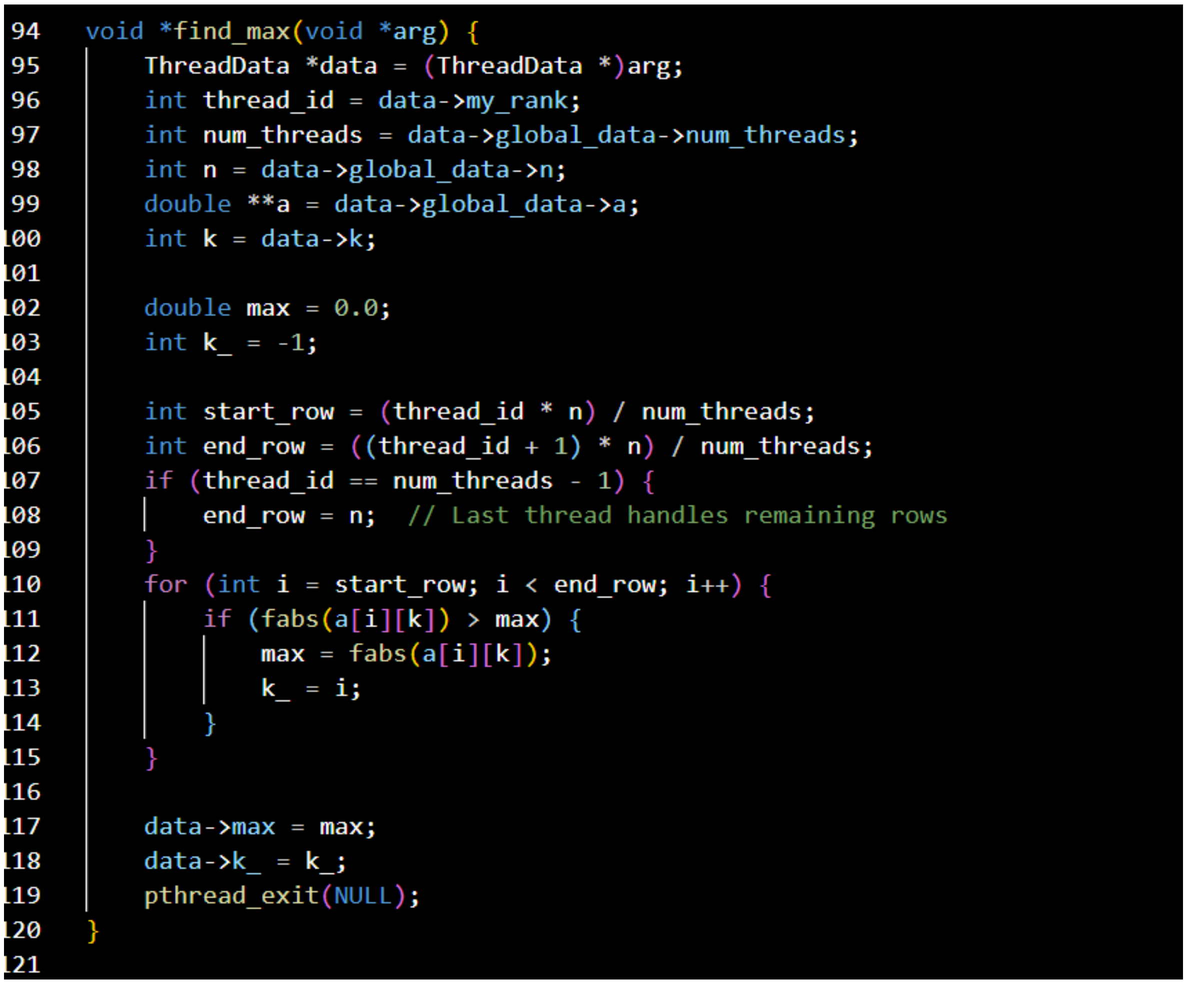
As mentioned in the section of Possible Parallelization Strategies, we tried parallelizing the mentioned sections of code using pthreads as described below:

1. Initialization of pi, L, U: I have created the function ”init matrices” to paralleize the work among threads, each thread iterates through its assigned rows, setting values in pi and initializing elements in l and u starting from their corresponding start row and end row which are functions of their rank(thread id). I have created a global struct called globaldata(contains n, pi, a, l, u) so that all threads can access its content as shared variables and have thread id as a private variable to that particular thread and all these variables I stored in another struct called data. This function is called in pthread create with pointer to ThreadData struct as argument. We have also taken care of the fact that in case where n is not exactly divisible by t then the last thread is responsible for it’s computation starting from it’s own assigned start row uptill the last row to ensure not missing some of the last leftover rows.



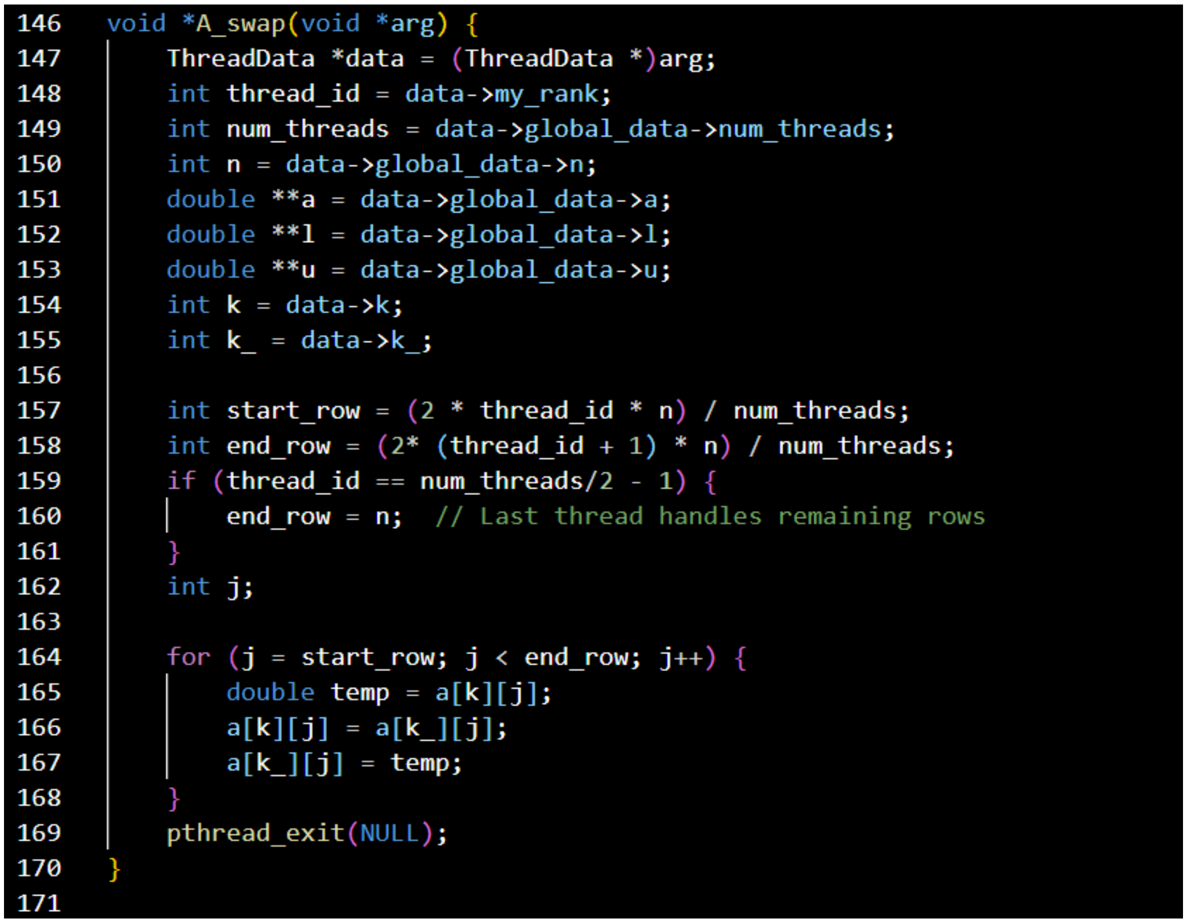
```c
void* init_matrices(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    int thread_id = data->my_rank;
    int num_threads = data->global_data->num_threads;
    int n = data->global_data->n;
    double **a = data->global_data->a;
    int *pi = data->global_data->pi;
    double **l = data->global_data->l;
    double **u = data->global_data->u;
    int start_row = (thread_id * n) / num_threads;
    int end_row = (((thread_id + 1) * n) / num_threads);
    if (thread_id == num_threads - 1) {
        end_row = n;  // Last thread handles remaining rows
    }
    int i, j;

    for (i = start_row; i < end_row; i++) {
        pi[i] = i;
        for (j = 0; j < n; j++) {
            l[i][j] = (i == j) ? 1.0 : 0.0;
            u[i][j] = 0.0;
        }
    }

    pthread_exit(NULL);
}
```

2. Finding Maximum entry in Column: I have created ”find max” function for parallel execution by multiple threads to determine the maximum absolute value within a specified column. Each thread is assigned a portion of the array’s rows based on its thread ID, by iterating through its assigned rows, each thread identifies the maximum absolute value and its corresponding row index within the specified column. When all the threads are joined one by one, I have calculated the maximum value of the private maximum values the individual threads have calculated.



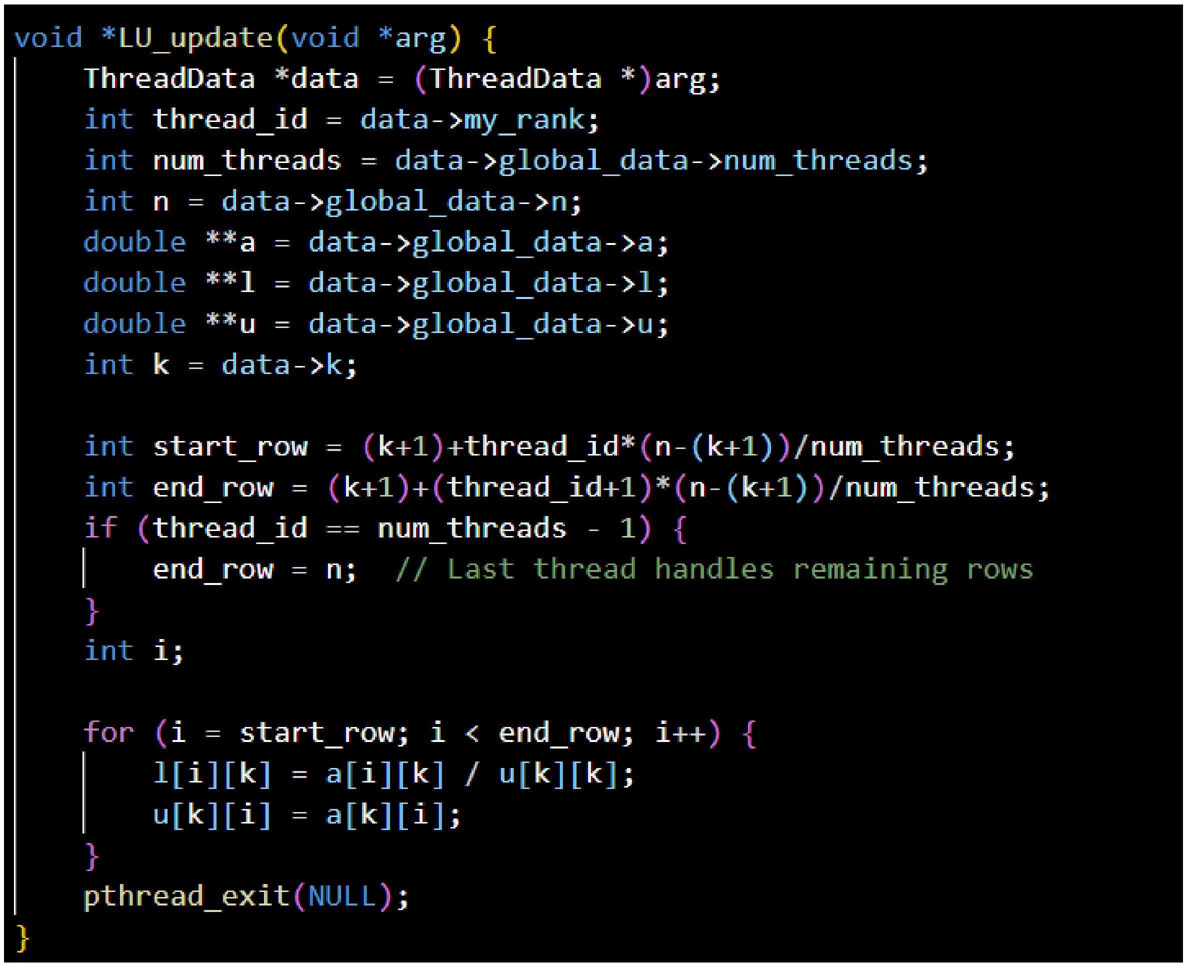
```
void *find_max(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    int thread_id = data->my_rank;
    int num_threads = data->global_data->num_threads;
    int n = data->global_data->n;
    double **a = data->global_data->a;
    int k = data->k;

    double max = 0.0;
    int k_ = -1;

    int start_row = (thread_id * n) / num_threads;
    int end_row = ((thread_id + 1) * n) / num_threads;
    if (thread_id == num_threads - 1) {
        end_row = n;  // Last thread handles remaining rows
    }
    for (int i = start_row; i < end_row; i++) {
        if (fabs(a[i][k]) > max) {
            max = fabs(a[i][k]);
            k_ = i;
        }
    }

    data->max = max;
    data->k_ = k_;
    pthread_exit(NULL);
}
```

3. Swapping rows of matrix a and l: We can perform swapping rows of a and l parallelly since the operations executed in both the swappings are independent of each other. So we decided to give half of the threads for executing the a matrix swap and other half of the threads for executing l swap and atlast joining all the threads, since the contents of a are accessed in the next line so we need to ensure that the threads finish their work.



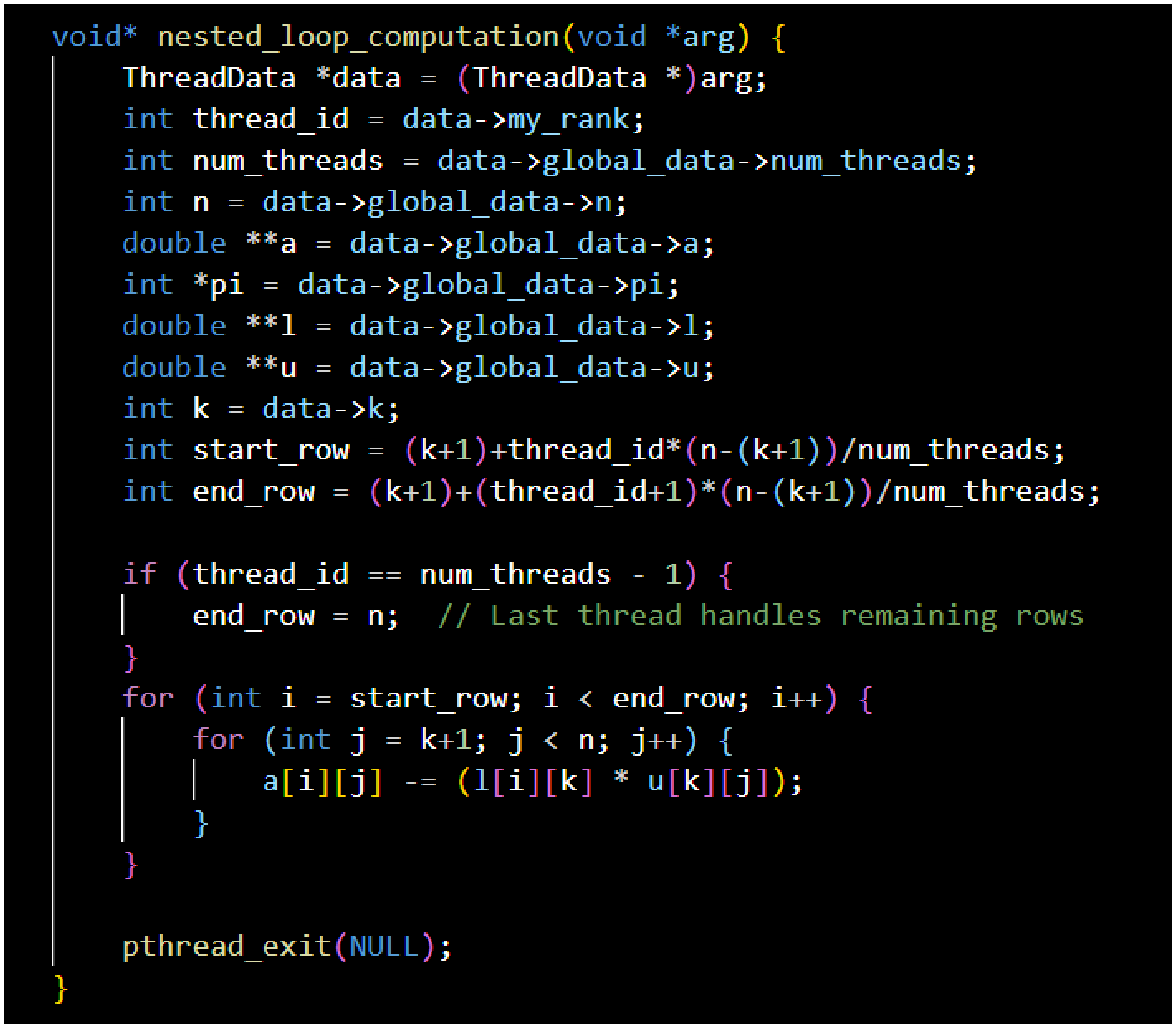
```
void *A_swap(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    int thread_id = data->my_rank;
    int num_threads = data->global_data->num_threads;
    int n = data->global_data->n;
    double **a = data->global_data->a;
    double **l = data->global_data->l;
    double **u = data->global_data->u;
    int k = data->k;
    int k_ = data->k_;

    int start_row = (2 * thread_id * n) / num_threads;
    int end_row = (2* (thread_id + 1) * n) / num_threads;
    if (thread_id == num_threads/2 - 1) {
        end_row = n;  // Last thread handles remaining rows
    }
    int j;

    for (j = start_row; j < end_row; j++) {
        double temp = a[k][j];
        a[k][j] = a[k_][j];
        a[k_][j] = temp;
    }
    pthread_exit(NULL);
}

```

4. Updating the values of l and u: Now since the upadation of l and u is independent of each other, though upadation of l uses contents of u but it uses only those which are not being updated, so we can parallelize this loop entirely. I have created a function called ”LU upadte” for the same.



```
void *LU_update(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    int thread_id = data->my_rank;
    int num_threads = data->global_data->num_threads;
    int n = data->global_data->n;
    double **a = data->global_data->a;
    double **l = data->global_data->l;
    double **u = data->global_data->u;
    int k = data->k;

    int start_row = (k+1)+thread_id*(n-(k+1))/num_threads;
    int end_row = (k+1)+(thread_id+1)*(n-(k+1))/num_threads;
    if (thread_id == num_threads - 1) {
        end_row = n;  // Last thread handles remaining rows
    }
    int i;

    for (i = start_row; i < end_row; i++) {
        l[i][k] = a[i][k] / u[k][k];
        u[k][i] = a[k][i];
    }
    pthread_exit(NULL);
}
```

5. Gaussian elimination step: The outer loop of the last loop is parallelized among the threads as it does not have any loop carried dependencies or critical sections to take care of. I have created the function ”nested loop computation” for the same.



```
void* nested_loop_computation(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    int thread_id = data->my_rank;
    int num_threads = data->global_data->num_threads;
    int n = data->global_data->n;
    double **a = data->global_data->a;
    int *pi = data->global_data->pi;
    double **l = data->global_data->l;
    double **u = data->global_data->u;
    int k = data->k;
    int start_row = (k+1)+thread_id*(n-(k+1))/num_threads;
    int end_row = (k+1)+(thread_id+1)*(n-(k+1))/num_threads;

    if (thread_id == num_threads - 1) {
        end_row = n;  // Last thread handles remaining rows
    }
    for (int i = start_row; i < end_row; i++) {
        for (int j = k+1; j < n; j++) {
            a[i][j] -= (l[i][k] * u[k][j]);
        }
    }

    pthread_exit(NULL);
}
```

After experimenting with various parallelization methods, we determined that, except for the final section (involving nested for loops that update variable ”a”), the overhead of parallelization outweighs the time reduction benefits for large input sizes and a high number of threads. Additionally, the last section, which encompasses the Gaussian elimination step, is the primary bottleneck in the algorithm, accounting for 97 percent of the execution time and featuring a general triple-nested loop structure. Consequently, in order to optimize our implementation, we have only parallelized the final section of the code.

7 Analysis of Pthreads Implementation

Thread Count	Matrix Size	Execution Time (secs)
1	500	0.228
	1000	1.553
	2000	10.806
	4000	88.128
	8000	626.641
2	500	0.141
	1000	0.941
	2000	6.207
	4000	50.354
	8000	388.624
4	500	0.137
	1000	0.709
	2000	5.082
	4000	43.512
	8000	351.222
8	500	0.208
	1000	0.638
	2000	3.347
	4000	29.212
	8000	264.597
16	500	0.379
	1000	0.809
	2000	4.007
	4000	28.150
	8000	276.775

Table 2: Run Times of LU Decomposition code using varied number of threads and size of A Matrix optimised using Pthreads

7.1 Performance Graphs

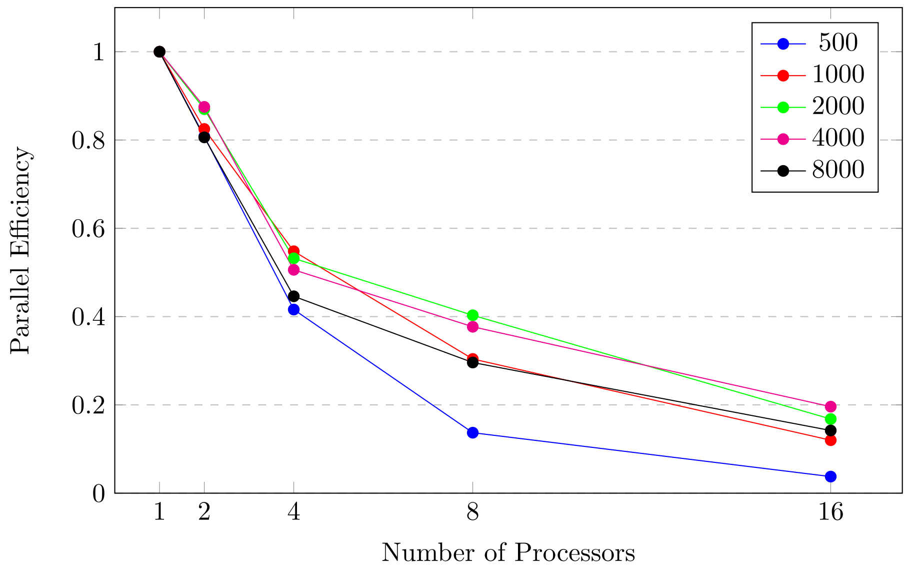


Figure 3: Parallel Efficiency vs. Number of Processors

7.2 Speedup Graphs

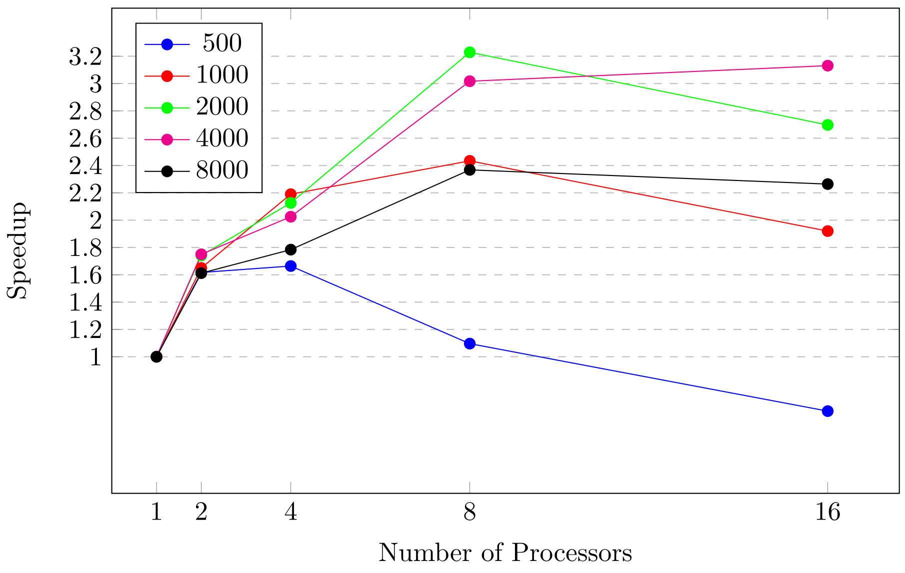


Figure 4: Parallel Efficiency vs. Number of Processors

8 Inference

Here are some observations:

- Speedup:
  Speedup refers to the improvement in performance achieved by parallelizing a task compared to executing it sequentially.Speedup can be calculated using the formula:
  Speedup= $\frac{Sequential\ Execution\ Time}{Parallel\ Execution\ Time}$
  From the data, we can calculate the speedup for each configuration (Thread Count and Matrix Size) by comparing the execution time of the sequential execution (Thread Count = 1) with the execution time of parallel execution (Thread Count > 1).

- Efficiency:
  Efficiency measures the utilization of resources in a parallel system. Efficiency can be calculated using the formula:
  Efficiency=$\frac{Speedup}{Number\ of\ Threads}$
  Efficiency gives us an idea of how well the available resources (threads) are utilized for parallel execution.

- Observations:
  - As the number of threads increases, the execution time generally decreases, indicating improved performance due to parallelization.
  - For larger matrix sizes (e.g., 4000 and 8000), the speedup tends to be higher compared to smaller matrix sizes.
  - However, the speedup does not scale linearly with the number of threads. In some cases, the speedup may plateau or even decrease with a higher number of threads, indicating diminishing returns.
  - The efficiency also tends to decrease with an increasing number of threads especially for smaller sized matrices. This suggests that the overhead of managing multiple threads starts to outweigh the benefits of parallelism.

9 Comparative Analysis of Open MP and Pthreads

We observed varying performance between pthreads and openmp for varying values of size of the matrix (n).

- For smaller matrix sizes (n=500 and n=1000), even though both OpenMP and Pthreads showed relatively comparable performance, Open MP exhibited slightly shorter execution times than Pthreads for varying number of threads.
- For the medium-sized matrix(n=2000), both Open MP and Pthreads performed equally well depending on the number of threads.
- However, as the matrix size(n) increased, Pthreads consistently outperformed OpenMP, with significantly shorter execution times observed for larger matrices (n=4000 and n=8000).

Overall, based on the provided data, Pthreads generally performed better than OpenMP for matrix multiplication tasks across all sizes considered.

References

[1] "LU Decomposition." https://en.wikipedia.org/wiki/LU_decomposition.

[2] "drand48." https://linux.die.net/man/3/drand48.

[3] "Matrix norm: L2,1 norm." https://en.wikipedia.org/wiki/Matrix_norm#L2.2C1_norm.