Indian Institute of Technology, Delhi


Parallel and Distributed Programming (COL 380)

Prof. Rijurekha Sen

# Assignment 1

Mani Sarthak (2021CS10095)
Harshit Gupta (2021CS10552)
Rishabh Verma (2021CS10581)

February 15, 2024

# Contents

## §1 Introduction

In this assignment we have to do LU decomposition of matrix and store the result using parrallelism to decrease the execution time. LU Decomposition is used by computers in the following algorithms:

- Solving system of Linear Equations
- Computing determinant of a matrix

Lower–Upper (LU) decomposition or factorization factors a given matrix as the product of a lower triangular matrix and an upper triangular matrix. Some assumptions/constraints we are taking in here are:-

- A is Singular, having rank k, then it's first k principal minors should be non zero (converse is not true)
- A is invertible and contains non zero leading principal minors (converse is also true).

Here LU decomposition can be applied to not only square matrices but also other matrices as well satisfying above constraints. But for this assignment we are Limited to Square matrices only. Our main purpose here is to Parallelise the Code for LU decomposition and Analyse the performance improvements. Hence we mainly focus on that.

*Note* : − Here we have the diagonal elements of L as 1, (we have defined it as given in the sequential pseudo code).

## §2 Logic and Implementation

### §2.1 Algorithm For LU decomposition

**Algorithm 2.1 —** LU Decomposition with Partial Pivoting

Input matrix $a(n, n)$
**Require:**
**Ensure:** $\pi(n)$, $l(n, n)$, and $u(n, n)$
Initialize $\pi$ as a vector of length $n$
Initialize $u$ as an $n \times n$ matrix with 0s below the diagonal
Initialize $l$ as an $n \times n$ matrix with 1s on the diagonal and 0s above the diagonal
**for** $i = 1$ to $n$ **do**
    $\pi[i] = i$
**end for**
**for** $k = 1$ to $n$ **do**
    max $= 0$
    **for** $i = k$ to $n$ **do**
        **if** max $< |a(i, k)|$ **then**
            max $= |a(i, k)|$
            $k' = i$
        **end if**
    **end for**
    **if** max $== 0$ **then**
        **error** (singular matrix)
    **end if**
    Swap $\pi[k]$ and $\pi[k']$
    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 $n$ **do**
        $l(i, k) = a(i, k)/u(k, k)$
        $u(k, i) = a(k, i)$
    **end for**
    **for** $i = k + 1$ to $n$ **do**
        **for** $j = k + 1$ to $n$ **do**
            $a(i, j) = a(i, j) - l(i, k) \times u(k, j)$
        **end for**
    **end for**
**end for**

- The LU factorization algorithm that is most commonly used on parallel machines is simply a reorganization of classic Gaussian Elimination. The basic algorithm proceeds row by row, attempt- ing to "eliminate" entries below the main diagonal. Multiples of row i are subtracted from rows below i in order to ensure that the part of column i below the main diagonal becomes zero. To enhance numerical stability pivoting, the swapping of rows to place a large value on the diagonal, is performed prior to each elimination step.

- Row interchanges (pivoting) is required to ensure existence of LU factorization and numerical stability of Gaussian elimination algorithm. After this process is completed, the solution of Ax=b can be obtained by forward and back substitution with L and U.

- In our implementation, we have taken the ijk ordering of the nested for loops, but they can be taken in any order, for total of 3! = 6 different ways of arranging loops. Different loop orders have different memory access patterns, which may cause their performance to vary widely, depending on architectural features such as cache, paging, vector registers, etc. While thinking about improving the efficiency of the program via the use of multi-threads and parallelization, it becomes very important to order these loops in a way that promises efficient parallel implementation. These forms only differ in accessing matrix by rows or columns, respectively.

## §2.2 Analysis without Parallelism

Here we first tried and coded the above algorithm in without any parallelism we tried to find the part of codes which we can parallelise to reduce the time. Here below are our observations :-

- in the part of algrotithm from line 1 to 21 is basically finding the maximum absolute element in column i below the diagonal for each iteration i and then swap the row of maximum element with row , and also storing that maximum element as pivot in $\pi$ as part of permutation. Since this part of code is necessary to do before even trying to do other major parts we cannot parallelise it as if there is any delay doing this part before other are started we can have varying answers. the complexity of this part is $O(n^2)$.
- in the part from line 22 to 25 is a for loop for row interchanges with complexity of $O(n^2)$. this we can parallelise and we try to find if there were any significant improvements But we cannot find any major improvements and moreover answers were varying with larger input
- in the part from line 26 to 31 is final step called **Gaussian elimination** which is the main bottleneck in the algorithm has general form of triple-nested loop in which entries of L and U overwrite those of A with a complexity of $O(n^3)$ and we parallelised this step.

## §2.3 What we parallelised

Based on all these steps we experimented to first parallelised the first **double nested loop** but it could not give satisfactory results both with **pthreads and openMp**. so we did not parallelise it. in the next **tri-nested loop** since with $\mathbf{O(n^3)}$ complexity it is the bottleneck step which takes the major amount of time so we paralleised it and we saw major improvements.

So finally our **sequential execution step** ($t_s$) is the both double nested loops from **line 1 to 25** and our parallel step ($t_p$) is **tri-nested loop**

$\boxed{T_1 = t_s + t_p}$
Where $T_1$ is the sequential Time without any parallelism

So Parallel code time is
$\boxed{T_p = t_s + t_p/N}$
Where N is Number of processors

Hence speedup S is
$\boxed{S{=}T_1/T_p = \frac{t_s+t_p}{t_s+t_p/N}}$

## §3 Code Walkthrough

Below contains a brief walkthrough of our code, what main and helper functions we used and how pthreads and openMp is used to parallelise the Gaussian elimination step.

### §3.1 Main Function

Here it conatins the main function of our program named **LU_Decomposition** which takes input matrix **A(nxn)** as double pointer, initialised value of **L** and **U**, again as a double pointer and Permutation array $\pi$ and of course size n and number of threads as well
This function computes the main part of algorithm of finding the pivot element as well as row interchanges and then Gaussian Elimination. Below pics contain the main function when we do it without any parallelism.

```
// LU Decomposition
void LU_Decomposition(double** A, double** L, double** U, int* P, int n) {

    for (int k=0; k < n; k++){
        double max = 0.0;
        int k_pivot = -1;
        for (int i=k; i<n; i++){
            if (abs(A[i][k]) > max){
                max = abs(A[i][k]);
                k_pivot = i;
            }
        }
        if (k_pivot == -1){
            throw runtime_error("Singular matrix");
        }

        if (k_pivot != k){
            swap(P[k_pivot], P[k]);
            swap(A[k_pivot], A[k]);
            for (int j=0; j<k; j++){
                swap(L[k_pivot][j], L[k][j]);
            }
        }
```

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

Figure 1: part with row interchanges

```
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];
    }
}
```

Figure 2: Gaussian elimination

## §3.2 Helper functions

### §3.2.1 Print function

This function basically print the matrix which is given as its input, used to check for any error handling and answer verification.

```
void printMatrix(double** mat, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << mat[i][j] << " \n"[j == n-1];
        }
    }
}
```

### §3.2.2 Initialise function

This function initialises the **L** and **U** as any random number and all diagonal elements of **L** are set to 1

```
itialisation
 initialise(double** L, double** U, int* P, int n){
for (int i=0; i<n; i++){
    P[i] = i;
    for (int j=i; j<n; j++){
        U[i][j] = check;
    }
    L[i][i] = 1;
    for (int j=0; j<i; j++){
        L[i][j] = check;
    }
}
```

### §3.2.3 Multiply function

This Function is used to multiply the 2 matrices which are given as its input and produce the result. it is used to verify the result of LU decomposition

```
// Function to multiply two matrices
double** multiply(double** A, double** B, int n) {
    double** result = (double**)malloc(n * sizeof(double*));
    for (int i = 0; i < n; ++i) {
        result[i] = (double*)malloc(n * sizeof(double));
        for (int j = 0; j < n; ++j) {
            double temp = 0;
            for (int k = 0; k < n; ++k) {
                temp += A[i][k] * B[k][j]; // look for cache coherence in this.
            }
            result[i][j] = temp;
        }
    }
    return result;
}
```

### §3.2.4 Permute function

this function is used to multiply the original matrix with permutation matrix and then we can verify the result whether LU decomposition is correct or not by using the below identity
$\boxed{\text{PA=LU}}$

```
// Function to apply permutation vector to a matrix
double** permute(double** A, int* P, int n) {
    double** PA = (double**)malloc(n * sizeof(double*));
    for (int i = 0; i < n; ++i) {
        PA[i] = (double*)malloc(n * sizeof(double));
        for (int j = 0; j < n; j++) {
            PA[i][j] = A[P[i]][j];
        }
    }
    return PA;
}
```

### §3.2.5 MatricesEqual

we made a function called **arematricesEqual** which takes two matrices as a input and check if those 2 matrices are equal. since we are using **double**, so we are using a tolerance value and check if difference for each element is greater than this or not. Also we are computing error by using this formula:-

$$\boxed{\text{Err}=\Sigma_{j=0}^{n-1}\sqrt{\Sigma_{i=0}^{n-1}a_{ij}^2}}$$

```
bool areMatricesEqual(double** A, double** B, int n) {
    for (int j = 0; j < n; ++j) {
        double col_err = 0.0;
        for (int i = 0; i < n; ++i) {
            double x = abs(A[i][j] - B[i][j]);
            col_err += x * x;
            if (x > eps) {
                return false;
            }
        }
        norm_val += sqrt(col_err);
    }
    return true;
}
```

## §3.3 Pthreads

### §3.3.1 Introduction

Pthreads, short for **POSIX threads**, is a threading library for the C programming language. It provides an **API** for creating and manipulating threads within a program. Pthreads conform to the POSIX standard **(IEEE Standard for Information Technology—Portable Operating System Interface)**, which defines the API for creating and managing threads and synchronization between threads.

### §3.3.2 Code Implementation

- We used pointers to pass matrices to avoid duplicacy of function arguments. Instead of swapping rows element by element, we instead swapped addresses of these rows (**double **) in **O(1) time.**
- we used pthreads by first creating a structure which contains **thread_id** as counter to keep track of threads as well as the parameters, **A,L,U,n** as attributes to keep computation independent of others without conflicting threads so that threads sun smoothly

```
// Structure to pass data to threads
typedef struct {
    int thread_id;
    int n;
    double** A;
    double** L;
    double** U;
    int* P;
    int k;
} ThreadData;
```

- after that we define a function known as **matrix_update** which basically does the second part of inner loop. we first calculate the local start and local end of loop of arrays and then run the loop which update **A[i][j].**

```
void* matrix_update(void* threadarg) {
    ThreadData* my_data = (ThreadData*) threadarg;
    int tid = my_data->thread_id;
    int n = my_data->n;
    int k = my_data->k;
    double** A = my_data->A;
    double** L = my_data->L;
    double** U = my_data->U;
    int start = tid*(n-(k+1))/NUM_THREADS;
    int rows_to_process = (n-(k+1))/NUM_THREADS;
    if (tid == NUM_THREADS-1){
        rows_to_process = n - (k+1) - start;
    }

    // check for loop invariant here
    for(int i=(k+1) + start; i < (k+1)+(tid+1)*(n-(k+1))/NUM_THREADS; i++){
    // for(int i=(k+1) + start; i < (k+1)+start + rows_to_process; i++){
        for(int j=k+1; j < n; j++){
            A[i][j] -= L[i][k]*U[k][j];
        }
    }

    return NULL;
}
```

- after that we finally update the main **LU decomposition**function by defining a loop which run through thread array and then we define **thread[i]** and its attributes for each iteration of **inner loop** and then create the thread and then we finally we create a loop which joins each thread of **thread_array**

```
// parallel section O(n^3)
for (int i=0; i<NUM_THREADS; i++){
    thread_data_array[i].thread_id = i;
    thread_data_array[i].k = k;
    thread_data_array[i].n = n;
    thread_data_array[i].A = A;
    thread_data_array[i].L = L;
    thread_data_array[i].U = U;
    thread_data_array[i].P = P;

    int rc = pthread_create(&threads[i], NULL, matrix_update, (void*)(thread_data_array + i));
    if (rc != 0) {
        cout << "Error:unable to create thread," << rc << endl;
        exit(-1);
    }
    // for (int j=start; j<end; j++){
    //     A[i][j] = A[i][j] - L[i][k] * U[k][j];
    // }
}
for (int i=0; i<NUM_THREADS; i++){
    pthread_join(threads[i], NULL);
}
```

## §3.4 OpenMp

### §3.4.1 Introduction

**OpenMP**, which stands for **Open Multi-Processing**, is an **application programming interface (API)** that supports multi-platform shared memory multiprocessing programming in C, C++, and **Fortran**. It enables developers to create parallel programs that can run efficiently on **multicore** processors, shared memory multiprocessors, and distributed memory systems. OpenMP simplifies parallel programming by providing a set of compiler directives, runtime library routines, and environment variables that enable developers to **parallelize** their code without needing to deal with **low-level threading details**.

### §3.4.2 Code Implementation

here we don't have to much with openMp we just included the **OpenMP library** by the name of **omp.h** . After that we updated the main function by first setting the number of threads at start of **LU_decomposition** function. After that in the Gaussian elimination loop.

- **pragma omp:** This is a preprocessor directive indicating that what follows is an OpenMP directive.
- **parallel for**: This directive creates a team of threads, and the enclosed for loop is divided among them for parallel execution. Each thread will execute a different chunk of the loop iterations.
- **collapse(2):** This clause indicates that a multi-level loop is being parallelized. In this case, it suggests that the loop immediately following is a nested loop with two levels. The iterations of both loops will be divided among the threads.
- **shared(A, L, U, k, n):** This clause specifies that variables **A, L, U, k, and n** are shared among all threads. This means that each thread will have its own copy of these variables, but changes made by one thread to these variables will be visible to other threads.

```
// Parallelize the update of matrix A
#pragma omp parallel for collapse(2) shared(A, L, U, k, n)
for (int i = k + 1; i < n; i++) {
    for (int j = k + 1; j < n; j++) {
        A[i][j] -= L[i][k] * U[k][j];
    }
}
```

## §3.5 Data Partitioning

- here we achieved data partitioning is achieved here in **matrix_Update** as for each **k** we first define chunk size of rows we have compute for each thread as $(n-k-1)/NUM\_THREADS$ and then we assign each chunk of rows to a thread by calculating its the index as defined as start
- then we compute the inner loop which runs from j =k+1 to n which basically calculates the resultant elements of the selected row i i.e A[i][k+1] to A[i][n-1]
- in this way Data is partitioned in pthreads and in **OpenMp** the partitioning is achieved by high level abstraction itself

## §3.6 Synchronisation of Parallel work

- Synchronization is achieved using **pthread_join**. After creating threads and assigning work to them, the main thread waits for all threads to finish their work before proceeding to the next iteration of the outer loop.
- pthread_join is called inside the loop over threads (for (int i=0; i less than NUM_THREADS; i++)). This ensures that the main thread waits for each thread to complete its work before moving on.
- By joining threads within the loop over k, the main thread waits for all threads to complete processing for the current iteration k before advancing to the next iteration. This synchronization ensures that the LU decomposition algorithm proceeds correctly and that each iteration is completed before moving on to the next.
- in **OpneMp** Data Synchronisation is achieved by the high level abstraction which openMp achieved by itself

# §4 Performance Analysis

For reference below is the time without use of threads to calculate the speedup.

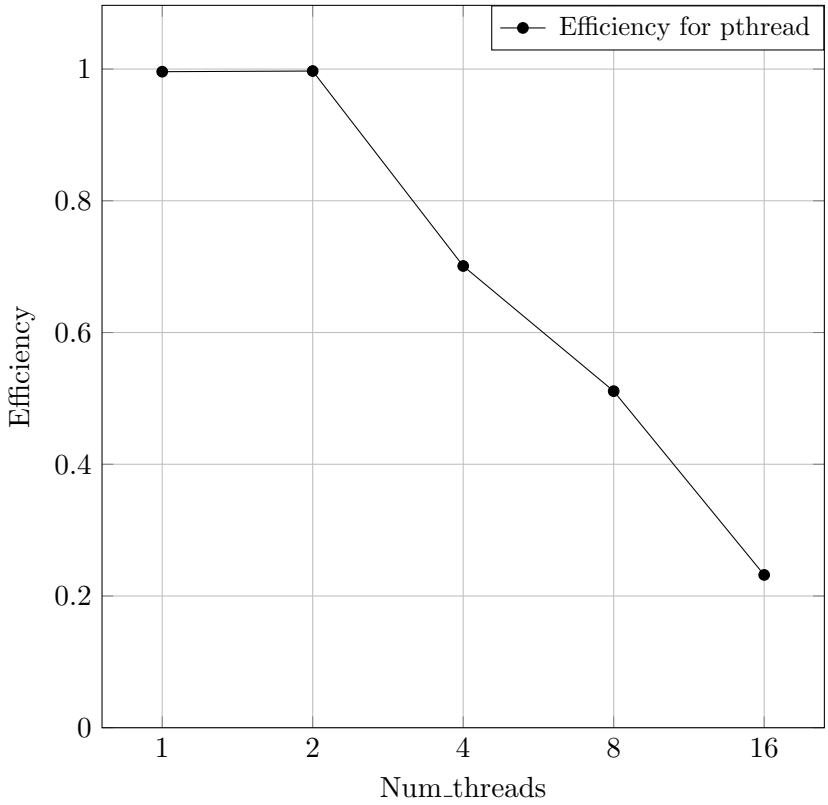
| N (Matrix Size) | Execution Time ($T_1$) |
|---|---|
| 500 | 0.169987 |
| 1000 | 1.35567 |
| 2000 | 10.9822 |
| 4000 | 87.193 |
| 8000 | 710.247 |

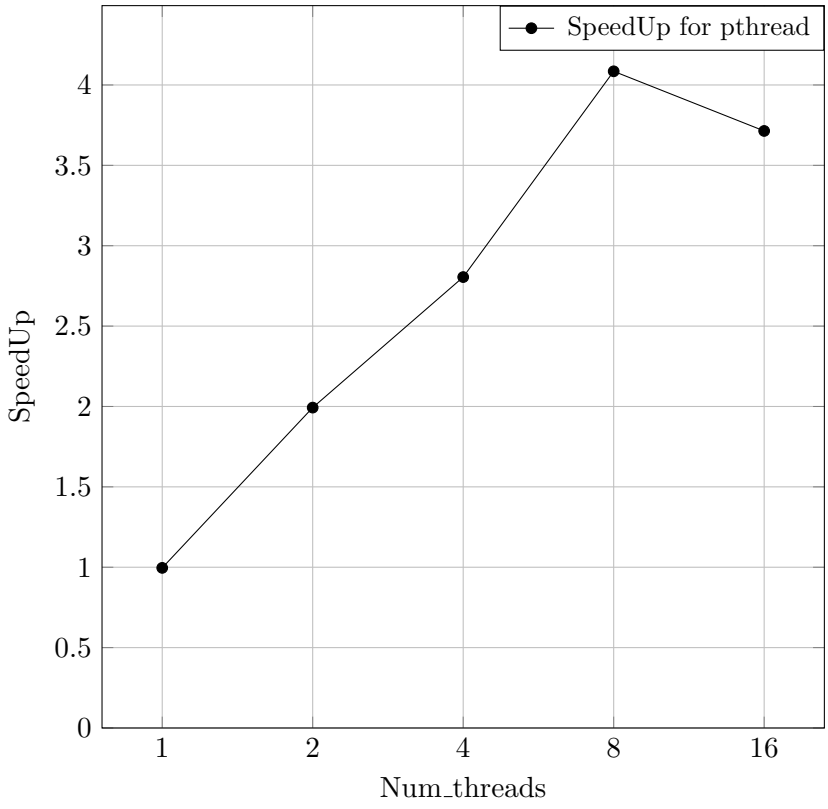
## §4.1 Analysis For Pthreads

### §4.1.1 Run Times

| threads(n) | N (Matrix Size) | $T_p$ | $E = \frac{T_1}{T_p n}$ | $S = \frac{T_1}{T_p}$ |
|---|---|---|---|---|
| 1 | 500 | 0.184722 | 0.920 | 0.920 |
| | 1000 | 1.38048 | 0.982 | 0.982 |
| | 2000 | 10.9905 | 0.999 | 0.999 |
| | 4000 | 87.7915 | 0.993 | 0.993 |
| | 8000 | 692.924 | 0.996 | 0.996 |
| 2 | 500 | 0.109807 | 0.774 | 1.548 |
| | 1000 | 0.721372 | 0.939 | 1.878 |
| | 2000 | 5.56111 | 0.987 | 1.974 |
| | 4000 | 43.6624 | 0.998 | 1.996 |
| | 8000 | 346.205 | 0.997 | 1.994 |
| 4 | 500 | 0.073127 | 0.581 | 2.324 |
| | 1000 | 0.448516 | 0.828 | 3.312 |
| | 2000 | 3.98507 | 0.689 | 2.756 |
| | 4000 | 31.7516 | 0.687 | 2.748 |
| | 8000 | 253.139 | 0.701 | 2.804 |
| 8 | 500 | 0.081845 | 0.260 | 2.080 |
| | 1000 | 0.391378 | 0.475 | 3.800 |
| | 2000 | 2.76089 | 0.497 | 3.967 |
| | 4000 | 22.5199 | 0.484 | 3.872 |
| | 8000 | 173.857 | 0.511 | 4.088 |
| 16 | 500 | 0.111296 | 0.095 | 1.520 |
| | 1000 | 0.425843 | 0.218 | 3.488 |
| | 2000 | 2.83063 | 0.242 | 3.872 |
| | 4000 | 22.3233 | 0.244 | 3.904 |
| | 8000 | 191.235 | 0.232 | 3.712 |

### §4.1.2 Plots for 8000 * 8000 matrix using Pthreads



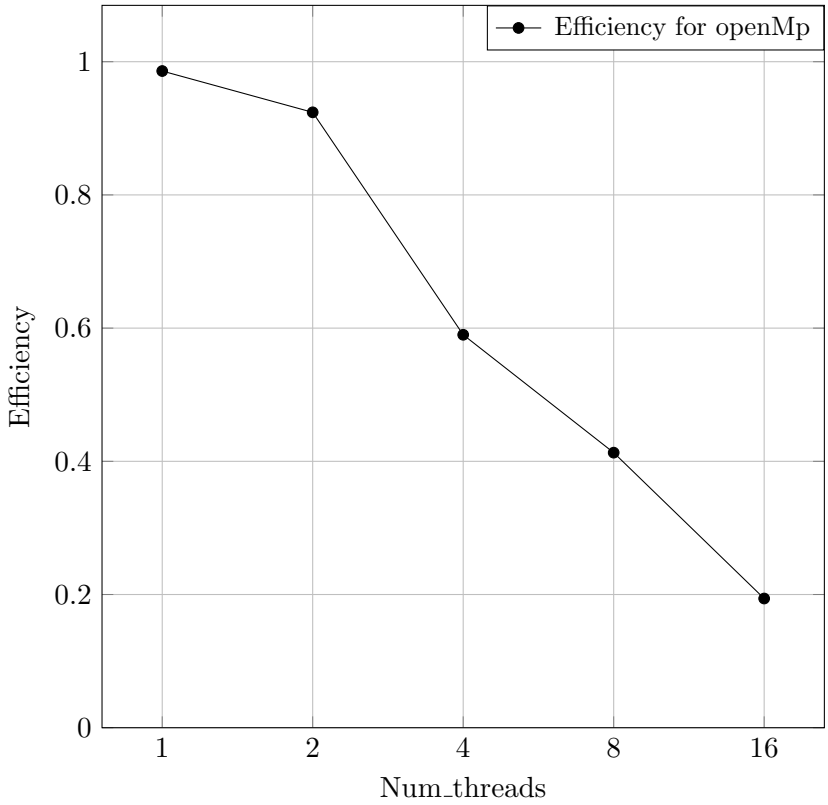



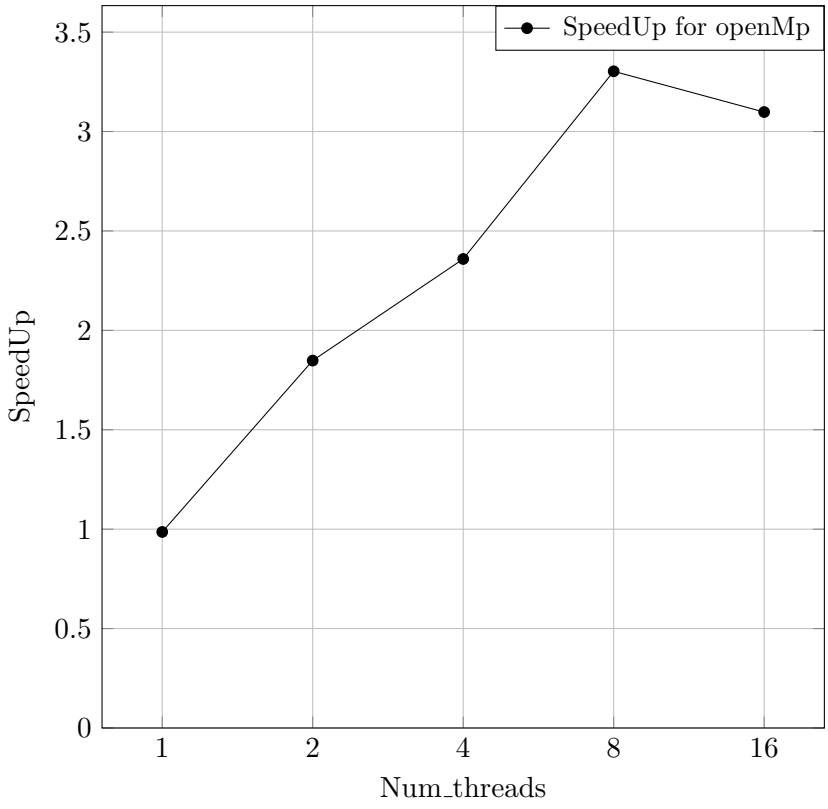
## §4.2 Analysis For OpenMp

### §4.2.1 Run Times

| threads(n) | N (Matrix Size) | $T_p$ | $E = \frac{T_1}{T_p n}$ | $S = \frac{T_1}{T_p}$ |
|---|---|---|---|---|
| 1 | 500 | 0.181686 | 0.936 | 0.936 |
| | 1000 | 1.41048 | 0.961 | 0.961 |
| | 2000 | 11.3593 | 0.967 | 0.967 |
| | 4000 | 90.4002 | 0.965 | 0.965 |
| | 8000 | 720.336 | 0.986 | 0.986 |
| 2 | 500 | 0.100121 | 0.849 | 1.698 |
| | 1000 | 0.731726 | 0.926 | 1.852 |
| | 2000 | 5.9664 | 0.920 | 1.840 |
| | 4000 | 48.0034 | 0.908 | 1.816 |
| | 8000 | 384.32 | 0.924 | 1.848 |
| 4 | 500 | 0.062768 | 0.677 | 2.708 |
| | 1000 | 0.582805 | 0.582 | 2.328 |
| | 2000 | 4.69199 | 0.585 | 2.340 |
| | 4000 | 37.9585 | 0.574 | 2.296 |
| | 8000 | 301.038 | 0.590 | 2.360 |
| 8 | 500 | 0.066586 | 0.319 | 2.552 |
| | 1000 | 0.476082 | 0.356 | 2.848 |
| | 2000 | 3.52229 | 0.390 | 3.120 |
| | 4000 | 27.6615 | 0.394 | 3.152 |
| | 8000 | 215.025 | 0.413 | 3.304 |
| 16 | 500 | 0.081221 | 0.131 | 2.096 |
| | 1000 | 0.538108 | 0.157 | 2.512 |
| | 2000 | 3.75323 | 0.183 | 2.928 |
| | 4000 | 29.3371 | 0.186 | 2.976 |
| | 8000 | 229.254 | 0.194 | 3.104 |

### §4.2.2 Plots for 8000 * 8000 matrix using OpenMP



## §4.3 Some Observations

- we can clearly see that as size of matrices increases speed up increased as well. for size=500 the speedup is approximately 1 for openMP and pthreads but as we size increases to size=8000 there is significant speedup of around 3
- for num_threads =1 we are adding an extra overhead for pthreads and openMp without any gain in parallel part, so efficiency is less than 1
- we also see graphs for both openMp and pthreads is almost similar in shape for both in efficiency and speedup

## §4.4 Challenges in measuring performance

- First of all, all of us have different hardwares in our computer so we have to standerdise one of our's for time checking and reporting. Lets say Mani's MacBook M1 was decided with specification of 8 cpu cores.
- Now given the apple hardware we dont have all those 8 cores for calculation in real, 4 of them are performance and rest 4 are for efficiency. Also at any point of time we have several compulsory system processes and threads running which cant be avoided. Hence so at any point of time if we pass the **num_threads** as x then while the process is running (observing on Activity Monitor or using **ps -ef | grep pthread** ) we see that on average less than x amount of cpu cores is provided for computation even when x is less than 8. (eg. for x = 4 we find that about 3.2 cores on average is used).
- Also running the program in almost identical environment gives different execution time, which sometimes become way too off (several reasons like background applications running, heat produced raising thermal interrupts, dynamic branch predictors getting saturated for many runs of same program, uncontrolled execution of system processes by operating system and many more).
- Because of these reasons the actual plots presented in the report are not actual figures (ie. **num_threads** =4 dont represent 4 actually but a little smaller value say 3.2, interpret it as average number of cores provided to it.) Also the time reported in the tables and that was used for making plots were hence taken as the average on many runs in the hope that it will normalise the error generated by uncontrolled factors (basically **Central Limit Theorem**).
- We tried testing the scripts on **baadalvm**, and several other **virtual machines** as well, for which we had access to. But all had these issues in them.
- Since the hardware bottleneck is that we can have atmost 8 cores at any given time, so we cant actully measure the actual speedup or efficiency for 16 cores, and hence using more than 8 cores should give worse speedup (accounting for **parallel overhead**, **pigeonhole principle** for cores allocation and **context switching**). This effect can be observed easily in the plots.
- These were all the effects that show the possible inaccuracies that could be observed in the measurement. However many factors like how the compilers are dealing with the programs, how the os is scheduling the threads and how the actual hardware is there in the machine (eg. Apple silicon or Intel hardware, the assembly instructions RISC or CISC) etc. are centailnly not in our control and hence we cant observe or predict them.

# §5 Special Points

## §5.1 Random generation

- we used the **drand48** function in test generation to create matrices. we output an random number from 0 to 1 for each entry in the matrix and then multiply it by 100 so that range of elements of matrices is from 0 to 100
- we used the **srand48** for seeding the drand48 function. Seeding is the process of initializing the random number generator with a starting value, called the seed (here we have seeded with **time(NULL)**). By providing a seed, we can ensure that the sequence of random numbers generated by the random number generator is reproducible.

## §5.2 Usage of double pointers for calculation and IO

- **Parallel Access:** With a array of pointers (each representing a row to the matrix), different threads or processes can work on different portions of the array concurrently without interfering with each other. Each thread is assigned a subset of the array, leading to parallelism gains.
- **Improved Cache Locality:** Pointers can sometimes improve cache locality. When processing small chunks of data, each pointed-to vector can fit entirely within a cache line, reducing cache misses compared to accessing elements in a large contiguous array.
- **Local variables when reading/writing a row:** When working with a matrix (of type `double**`) operation which requires reading/writing elements of a row, using a local variable to save the pointer to that row (of type `double*`) greatly improves performance. This occurs because with no optimization flags, `A[i][j]` requires first reading address of row using `A[i]`, then reading address of its $j'th$ element. However, if a local variable $A_{row}$ is used to save memory address of row `A[i]`, then $A_{row}[j]$ only requires 1 memory access.

## §5.3 False Sharing

- **Avoiding False Sharing:** It is where different threads modify data in the same cache line, can lead to performance bottlenecks. Using separate data vectors pointed to by different pointers help mitigate this issue by reducing contention for cache lines.

## §5.4 Pthreads vs OpenMp

- Based on the analysis we can clearly see pthreads gives better results such as efficiency and speedUp compared to OpenMp Based on the currnet setup we have. Because OpenMp is high level abstratction which also compute th way it parallelise the code in contrast to pthreads where we ourselves define the what part of code and how to compute it in different threads. so we may have extra overhead in OpenMp so it could be the reason pthreads is more efficient.

## §5.5 norm values

- Based on the various matrices we computed for n=500,1000,2000,4000,8000 the norm values we get were of the order of $10^{-9}$ . We also observed that there is no associativity $a \cdot (b \cdot c) \neq (a \cdot b) \cdot c$ in floating point arithmetic so the way we multiply the matrix also matters in norm values calculation.