

# Assignment 2 (Matrix Multiplication Using Pthreads and OpenMP)

Anas Albaqeri

202004427

---

## Introduction

### Matrix Multiplication

Matrix multiplication is a binary operation that takes a pair of matrices, typically represented as two-dimensional arrays, and produces another matrix. It's a fundamental operation in linear algebra and has various applications in computer graphics, physics, statistics, and many other fields.

The process of multiplying two matrices, often denoted as  $A$  and  $B$ , results in a new matrix  $AB$ . The number of columns in matrix  $A$  must be equal to the number of rows in matrix  $B$  for the multiplication to be defined. The figure below explains the rule further.

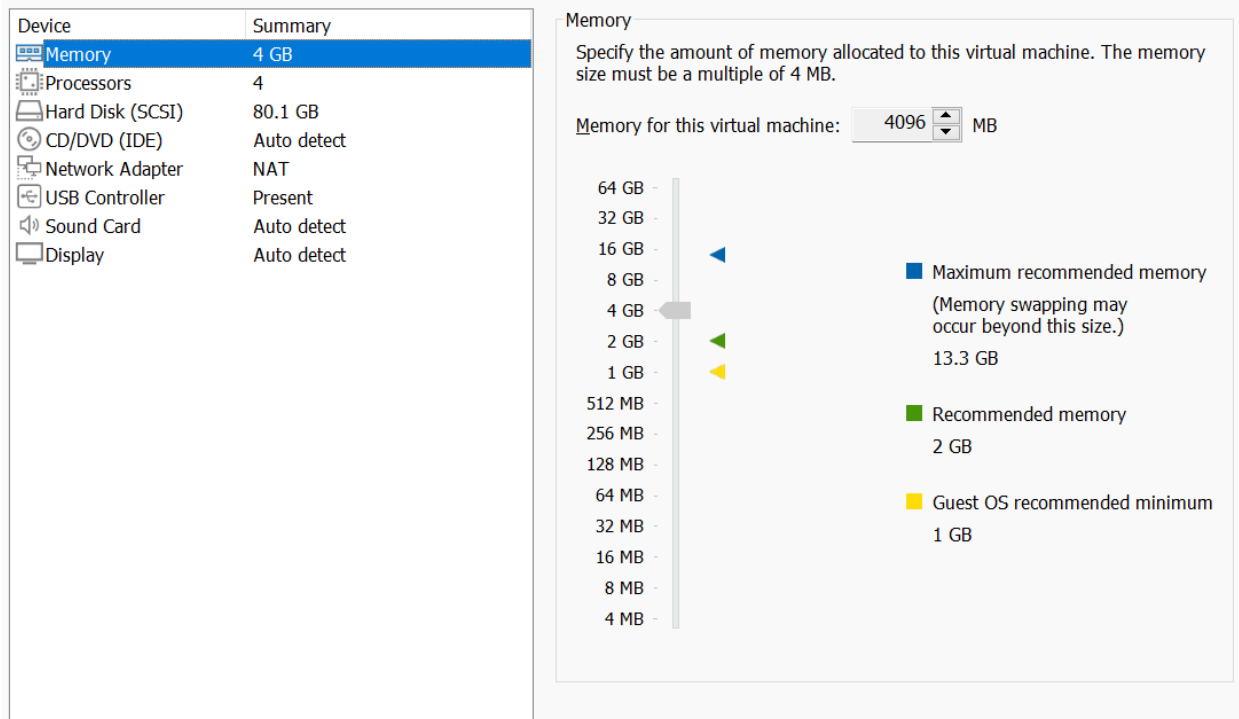
$$\begin{array}{c}
 \overbrace{\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix}}^A \quad \overbrace{\begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,l} \\ b_{2,1} & b_{2,2} & \dots & b_{2,l} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \dots & b_{m,l} \end{bmatrix}}^B \\
 \\
 \underbrace{\begin{bmatrix} \sum_{k=1}^m a_{1,k} b_{k,1} & \sum_{k=1}^m a_{1,k} b_{k,2} & \dots & \sum_{k=1}^m a_{1,k} b_{k,l} \\ \sum_{k=1}^m a_{2,k} b_{k,1} & \sum_{k=1}^m a_{2,k} b_{k,2} & \dots & \sum_{k=1}^m a_{2,k} b_{k,l} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^m a_{n,k} b_{k,1} & \sum_{k=1}^m a_{n,k} b_{k,2} & \dots & \sum_{k=1}^m a_{n,k} b_{k,l} \end{bmatrix}}^{AB}
 \end{array}$$

## Parallelizing the multiplication

### setup:

For this test, I am using Kali emulator with the following specification settings to run the code which I took from <https://www.geeksforgeeks.org/c-program-multiply-two-matrices/> and modified over it.

Kali Specifications:



for this task, I used 4 cores as 4 parallel processors working with 4 gb of RAM. This was the maximum my machine was able to handle for the Kali Emulator.

## Using Pthreads:

I used a standard code for multiplying matrices and adjusted to be parallelized using Pthreads, here is a breakdown of the code used:

```
kali-linux-2023.3-vmware-amd64 - VMware Workstation 17 Player (Non-commercial use only)
Player
~/Desktop/Assignment 2/MMpthreads.c - Mousepad
File Edit Search View Document Help
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <sys/time.h>
5
6 #define MATRIX_SIZE 1000
7
8
9 int MatrixA[MATRIX_SIZE][MATRIX_SIZE];
10 int MatrixB[MATRIX_SIZE][MATRIX_SIZE];
11 int ResultMatrix[MATRIX_SIZE][MATRIX_SIZE];
12
13
14 typedef struct {
15     int thread_id;
16     int num_threads;
17 } ThreadData;
18
19
20 void* MultiplyMatrices(void* arg) {
21     ThreadData* data = (ThreadData*)arg;
22     int start = data->thread_id * (MATRIX_SIZE / data->num_threads);
23     int end = start + (MATRIX_SIZE / data->num_threads);
24     for (int i = start; i < end; i++) {
25         for (int j = 0; j < MATRIX_SIZE; j++) {
26             ResultMatrix[i][j] = 0;
27             for (int k = 0; k < MATRIX_SIZE; k++) {
28                 ResultMatrix[i][j] += MatrixA[i][k] * MatrixB[k][j];
29             }
30         }
31     }
32     pthread_exit(NULL);
33 }
34
35 int main(int argc, char* argv[]) {
36
37     int num_threads = atoi(argv[1]);
38     for (int i = 0; i < MATRIX_SIZE; i++) {
39         for (int j = 0; j < MATRIX_SIZE; j++) {
40             MatrixA[i][j] = rand() % 10;
41             MatrixB[i][j] = rand() % 10;
42         }
43     }
44 }
```

```

    struct timeval startTime, endTime;
    gettimeofday(&startTime, NULL);

    // Multiply the matrices using pthreads
    pthread_t threads[num_threads];
    ThreadData threadArgs[num_threads];
    for (int i = 0; i < num_threads; i++) {
        threadArgs[i].thread_id = i;
        threadArgs[i].num_threads = num_threads;
        pthread_create(&threads[i], NULL, MultiplyMatrices, &threadArgs[i]);
    }
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    gettimeofday(&endTime, NULL);
    double executionTime = (endTime.tv_sec - startTime.tv_sec) + (endTime.tv_usec - startTime.tv_usec) / 1000000.0;
    printf("Execution time using %d threads: %f seconds\n", num_threads, executionTime);

    return 0;
}
```

## 1. Header Files and Constants:

- Include necessary header files like `<stdio.h>`, `<stdlib.h>`, `<pthread.h>`, and `<sys/time.h>`.

- Define a constant `MATRIX_SIZE` with a value of 1000, which represents the size of the square matrices to be multiplied.

## 2. Matrix Declarations:

- Declare three 2D arrays: `MatrixA`, `MatrixB`, and `ResultMatrix`, each of size `MATRIX_SIZE x MATRIX_SIZE`. These matrices are used for matrix multiplication, and `ResultMatrix` will store the result.

## 3. Thread Data Structure:

- Define a structure `ThreadData` to hold data to be passed to each thread. It includes `thread_id` and `num_threads`, which are used to identify each thread and determine the number of threads in the computation.

## 4. Matrix Multiplication Function (`MultiplyMatrices`):

- This is the function that each thread will execute.
- It takes a pointer to a `ThreadData` structure as an argument, allowing each thread to know its `thread_id` and the total number of threads.
- It calculates the range of rows to process based on the thread's `thread_id`.
- It performs matrix multiplication for the assigned rows and columns, storing the result in the `ResultMatrix`.

## 5. `main` Function:

- The program's entry point.
- It accepts the number of threads as a command-line argument (`num_threads`).
- Initializes and populates `MatrixA` and `MatrixB` with random values between 0 and 9.
- Records the start time using `gettimeofday`.

## 6. Parallel Matrix Multiplication with Pthreads:

- It initializes an array of `pthread_t` objects (`threads`) to represent the threads.
- It creates `ThreadData` structures (`threadArgs`) for each thread, setting `thread_id` and `num_threads`.

- It spawns multiple threads based on the value of `num_threads`, each executing the `MultiplyMatrices` function. This is the parallel part of the code.
- It waits for all threads to complete using `pthread_join`.

## 7. Record Execution Time:

- Records the end time using `gettimeofday` after all threads have finished.
- Calculates the execution time in seconds.
- Prints the execution time and the number of threads used.

## Parallelizing:

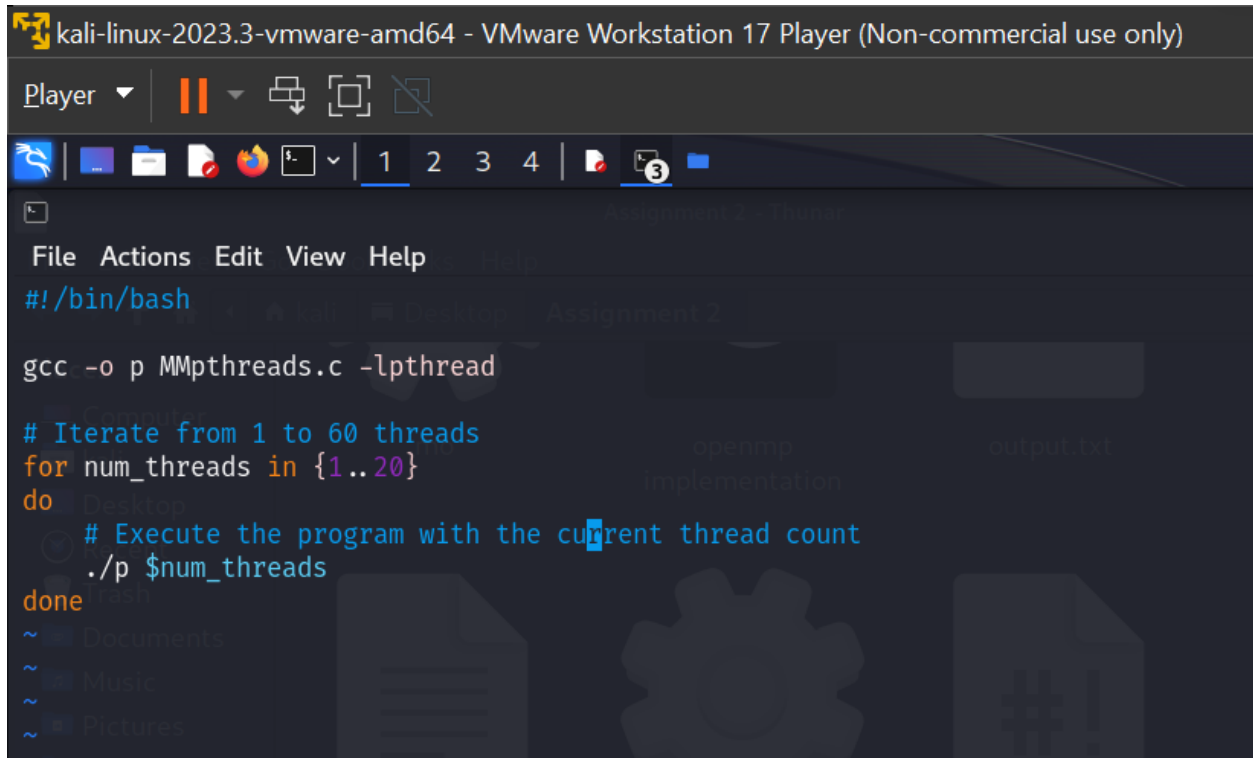
```
// Parallel Matrix Multiplication with Pthreads
pthread_t threads[num_threads];
ThreadData threadArgs[num_threads];
for (int i = 0; i < num_threads; i++) {
    threadArgs[i].thread_id = i;
    threadArgs[i].num_threads = num_threads;
    pthread_create(&threads[i], NULL, MultiplyMatrices, &threadArgs[i]);
}
for (int i = 0; i < num_threads; i++) {
    pthread_join(threads[i], NULL);
}
```

This part of the code demonstrates how matrix multiplication is parallelized using pthreads:

1. It creates an array of `pthread_t` objects ( `threads` ) to represent the threads.
2. It initializes an array of `ThreadData` structures ( `threadArgs` ) for each thread, setting `thread_id` and `num_threads`. This data structure allows each thread to identify itself and know the total number of threads involved.
3. It spawns multiple threads based on the value of `num_threads`. For each thread, it calls `pthread_create`. The `MultiplyMatrices` function is specified as the function to be executed by each thread.
4. It waits for all threads to complete using `pthread_join`. This synchronization point ensures that the main thread waits until all child threads have finished their work.

## Results:

I ran the test on 1000 x 1000 matrices and using 1 - 60 threads respectively of which I will be using only the first 10 results in my data comparison as no noticeable speedup is occurring after that.



```
kali-linux-2023.3-vmware-amd64 - VMware Workstation 17 Player (Non-commercial use only)
Player
File Actions Edit View Help
#!/bin/bash
gcc -o p MMpthreads.c -lpthread
# Iterate from 1 to 60 threads
for num_threads in {1..20}
do
    # Execute the program with the current thread count
    ./p $num_threads
done
~ Documents
~ Music
~ Pictures
```

## output:

kali-linux-2023.3-vmware-amd64 - VMware Workstation 17 Player (Non-commercial use)

Player ▾ | [Icons: Play, Full Screen, etc.]

[Icons: Home, Files, Firefox, etc.] | 1 2 3 4 | [Icons: Search, etc.]

File Edit Search View Document Help

[Icons: New, Open, Save, etc.]

MMpthreads.c

```
1 Execution time using 1 threads: 15.121880 seconds
2 Execution time using 2 threads: 8.127389 seconds
3 Execution time using 3 threads: 5.428974 seconds
4 Execution time using 4 threads: 4.074447 seconds
5 Execution time using 5 threads: 3.877891 seconds
6 Execution time using 6 threads: 4.254328 seconds
7 Execution time using 7 threads: 4.313621 seconds
8 Execution time using 8 threads: 4.088143 seconds
9 Execution time using 9 threads: 4.206521 seconds
10 Execution time using 10 threads: 4.246959 seconds
11 Execution time using 11 threads: 4.090062 seconds
12 Execution time using 12 threads: 4.249599 seconds
13 Execution time using 13 threads: 4.371964 seconds
14 Execution time using 14 threads: 4.392801 seconds
15 Execution time using 15 threads: 3.857918 seconds
16 Execution time using 16 threads: 4.176046 seconds
17 Execution time using 17 threads: 4.285926 seconds
18 Execution time using 18 threads: 4.112470 seconds
19 Execution time using 19 threads: 4.005331 seconds
20 Execution time using 20 threads: 4.125992 seconds
21 |
```

**Speedup Factor :**



$$S(P) = \frac{\text{Best Sequential Time}}{\text{Best Parallel Time}}$$

$$= \frac{15.12188}{3.857918} = 3.9197$$

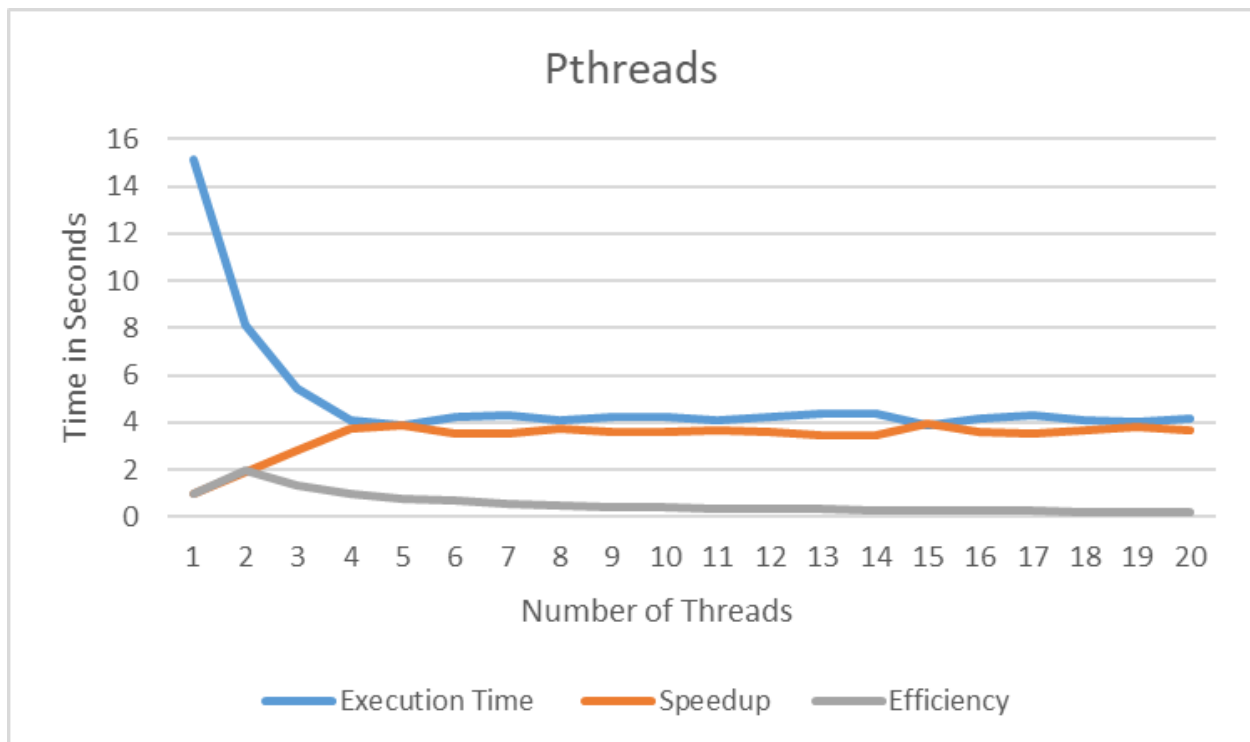
## Efficiency:

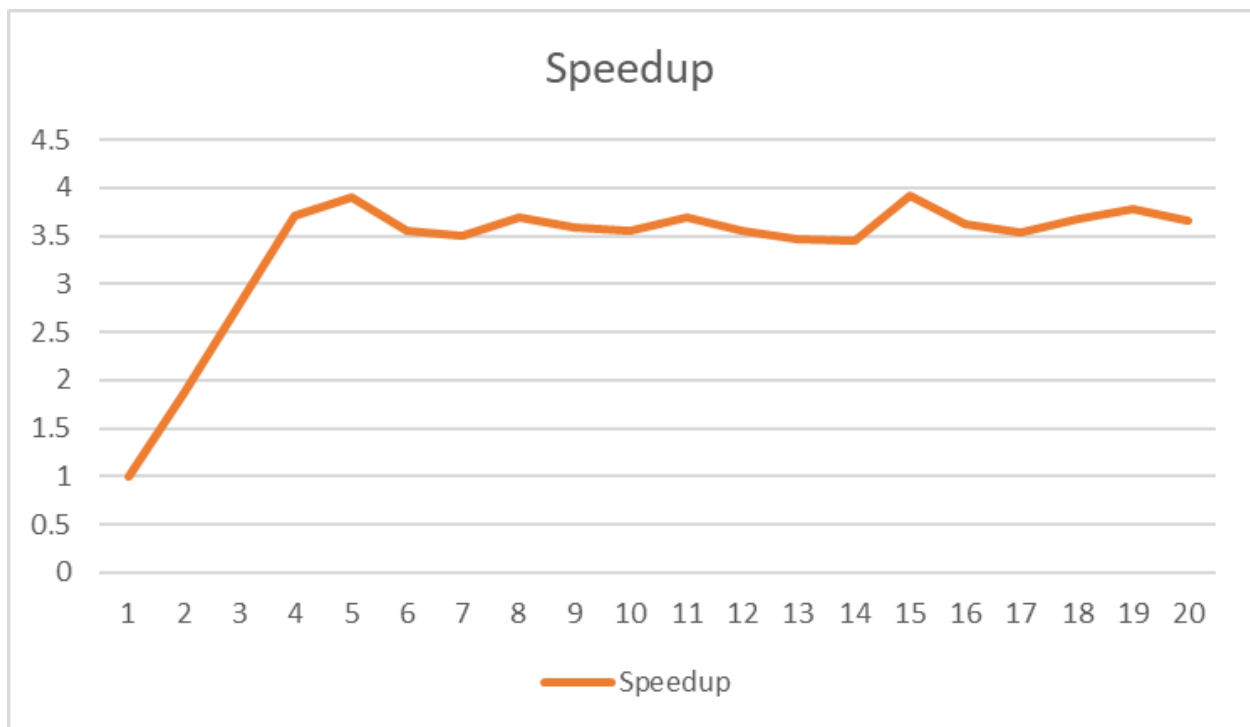
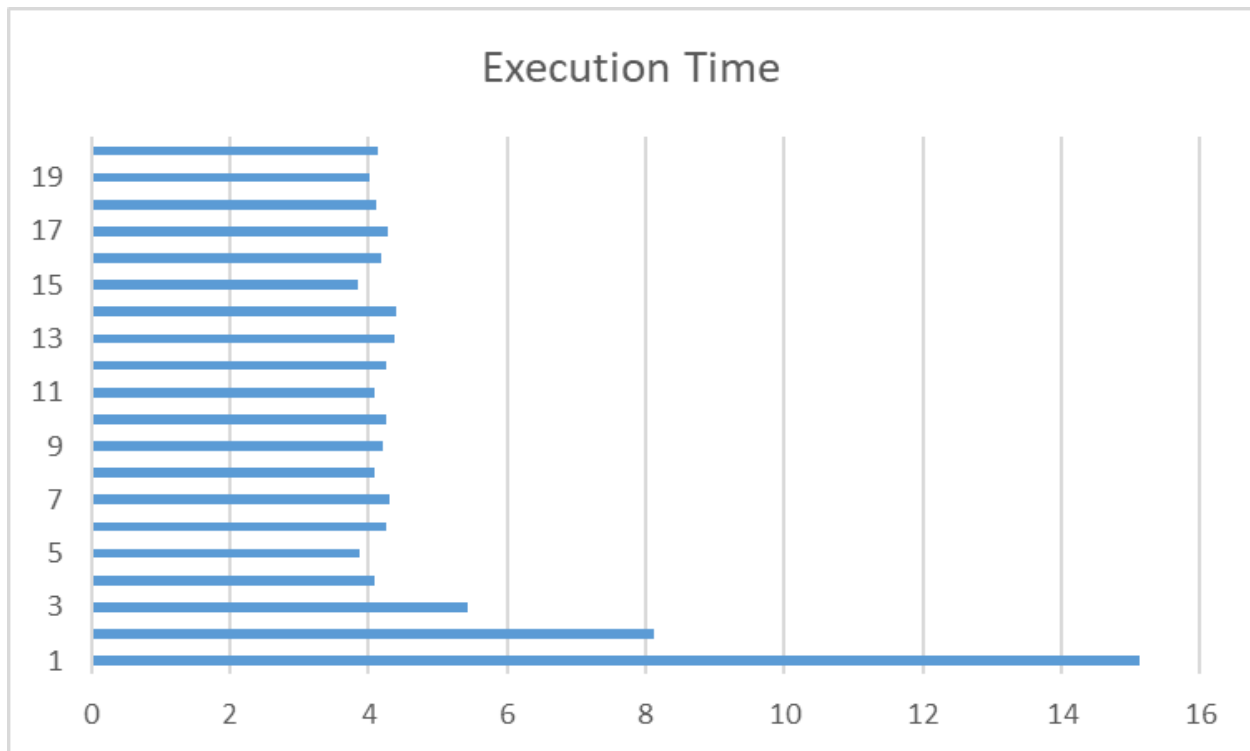
since the maximum speedup was acquired using 15 threads and since the differences are minute between using 4 to 20 threads we are going to use 4 threads as the optimal number

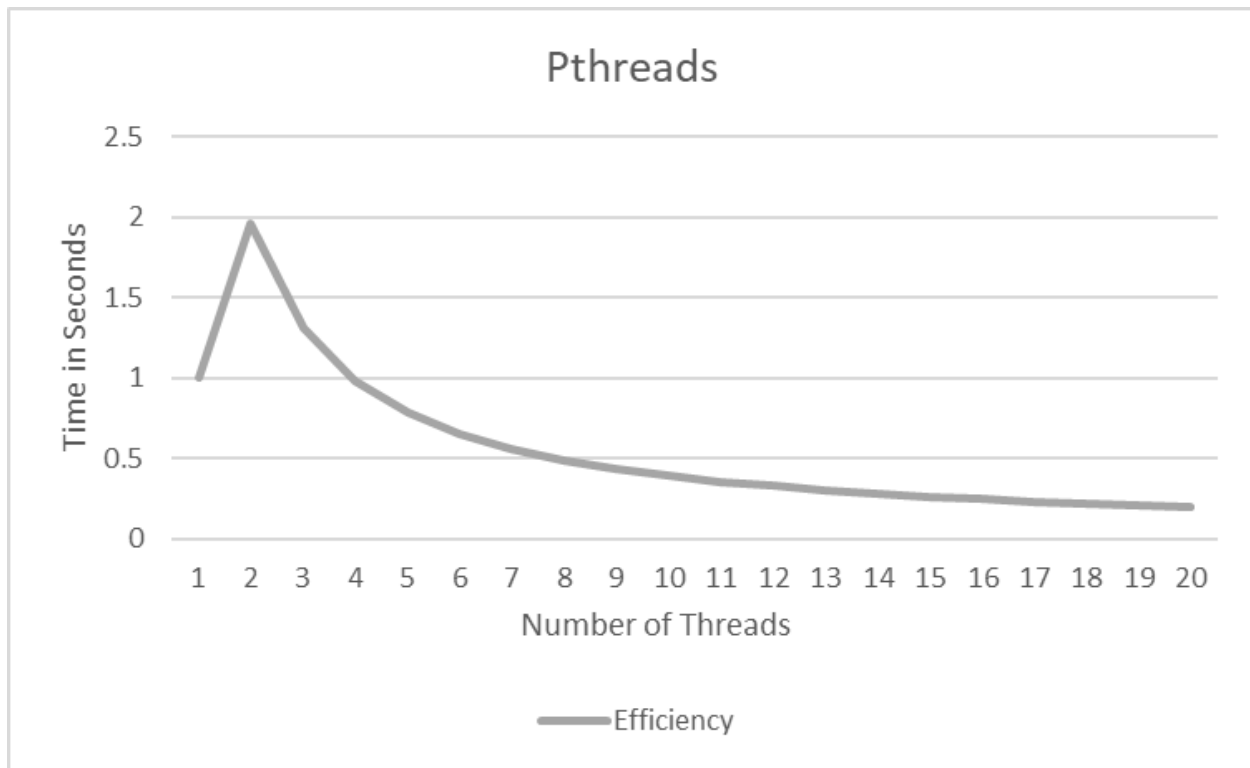
$$\text{Efficiency} = \frac{S(p)}{\text{Number of threads}} = 3.711394$$

## Scalability:

the graphs below shows the scalability across different numbers of threads.







## Using OpenMp

I used a standard code for multiplying matrices and adjusted to be parallelized using OpenMP library, here is a breakdown of the code used:

```
File Edit Search View Document Help
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 #include <time.h>
5
6 #define MATRIX_SIZE 1000
7 #define BLOCK_SIZE 32
8
9 int MatrixA[MATRIX_SIZE][MATRIX_SIZE];
10 int MatrixB[MATRIX_SIZE][MATRIX_SIZE];
11 int ResultMatrix[MATRIX_SIZE][MATRIX_SIZE];
12
13 void multiply_matrices(int start_row, int end_row) {
14     for (int i = start_row; i < end_row; ++i) {
15         for (int j = 0; j < MATRIX_SIZE; ++j) {
16             for (int k = 0; k < MATRIX_SIZE; ++k) {
17                 ResultMatrix[i][j] += MatrixA[i][k] * MatrixB[k][j];
18             }
19         }
20     }
21 }
22
23 int main(int argc, char* argv[]) {
24     if (argc != 2) {
25         fprintf(stderr, "Usage: %s <num_threads>\n", argv[0]);
26         return 1;
27     }
28
29     int num_threads = atoi(argv[1]);
30     //limiting test to 20 threads as no speed up apparent after few increases
31     if (num_threads < 1 || num_threads > 20) {
32         fprintf(stderr, "Number of threads must be between 1 and 20.\n");
33         return 1;
34     }
35 }
```

```

36  int i, j, k;
37  double start_time, end_time;
38  double elapsed;
39  srand(time(NULL));
40
41  // Initialize the matrices with random values
42  for (i = 0; i < MATRIX_SIZE; i++) {
43      for (j = 0; j < MATRIX_SIZE; j++) {
44          MatrixA[i][j] = 2;
45          MatrixB[i][j] = 2;
46      }
47  }
48
49  start_time = omp_get_wtime();
50  omp_set_num_threads(num_threads);
51
52  // Perform matrix multiplication with blocking
53  #pragma omp parallel
54  {
55      int num_threads = omp_get_num_threads();
56      int thread_id = omp_get_thread_num();
57      int rows_per_thread = MATRIX_SIZE / num_threads;
58      int start_row = thread_id * rows_per_thread;
59      int end_row = (thread_id == num_threads - 1) ? MATRIX_SIZE : start_row + rows_per_thread;
60
61      multiply_matrices(start_row, end_row);
62  }
63
64  end_time = omp_get_wtime();
65
66  printf("Execution time: %f seconds using %d threads\n", end_time - start_time, num_threads);
67
68  return 0;

```

the code used is similar to the previous one with the main following changes in penalization method used:

```

for (int i = start; i < end; i++) {
    for (int j = 0; j < MATRIX_SIZE; j++) {
        ResultMatrix[i][j] = 0;
        for (int k = 0; k < MATRIX_SIZE; k++) {
            ResultMatrix[i][j] += MatrixA[i][k] * MatrixB[k][j];
        }
    }
}

```

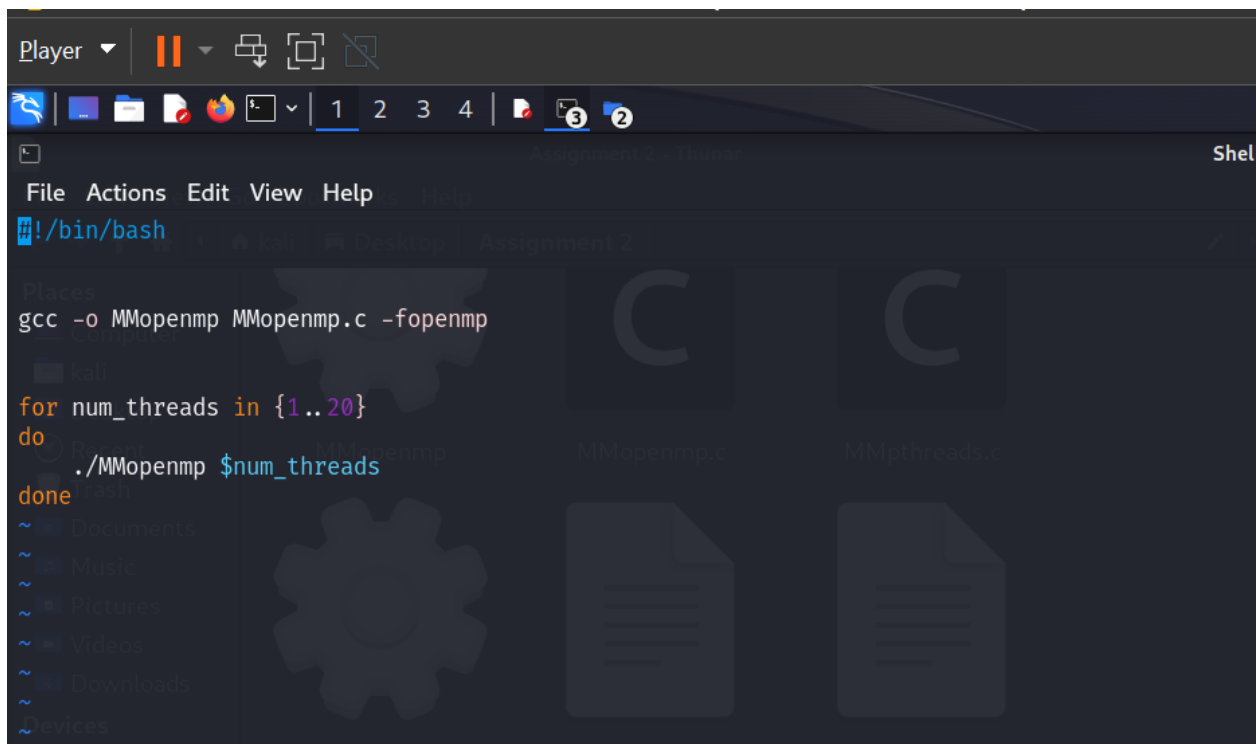
1. The outer loop ( `for (int i = start; i < end; i++)` ) iterates over a subset of rows. Each thread is responsible for a range of rows based on its `start` and `end` values. This divides the work among multiple threads.
2. The middle loop ( `for (int j = 0; j < MATRIX_SIZE; j++)` ) iterates over columns of the result matrix. This is common to all threads and is not parallelized.

- The innermost loop ( `for (int k = 0; k < MATRIX_SIZE; k++)` ) calculates the matrix multiplication by iterating over the shared dimension of the two input matrices. This is the computationally intensive part, and parallelization happens here. Each thread calculates the partial results for its assigned rows concurrently.

## Results

the test was run on same conditions as the Pthreads test with 1000 x 1000 matrices and using threads from 1 to 20 respectively.

The following results were acquired after running the script



MMOpenmp.c	
1	Execution time: 17.894962 seconds using 1 threads
2	Execution time: 7.600897 seconds using 2 threads
3	Execution time: 5.617599 seconds using 3 threads
4	Execution time: 5.029495 seconds using 4 threads
5	Execution time: 4.770310 seconds using 5 threads
6	Execution time: 4.724276 seconds using 6 threads
7	Execution time: 4.867918 seconds using 7 threads
8	Execution time: 4.625773 seconds using 8 threads
9	Execution time: 4.621660 seconds using 9 threads
10	Execution time: 4.636850 seconds using 10 threads
11	Execution time: 4.674454 seconds using 11 threads
12	Execution time: 4.351825 seconds using 12 threads
13	Execution time: 4.654945 seconds using 13 threads
14	Execution time: 4.585658 seconds using 14 threads
15	Execution time: 4.453260 seconds using 15 threads
16	Execution time: 4.540402 seconds using 16 threads
17	Execution time: 4.497482 seconds using 17 threads
18	Execution time: 4.501193 seconds using 18 threads
19	Execution time: 4.518787 seconds using 19 threads
20	Execution time: 4.537096 seconds using 20 threads

## Speedup Factor :

$$S(P) = \frac{\text{Best Sequential Time}}{\text{Best Parallel Time}}$$

$$= \frac{17.894962}{4.351825} = 4.1091$$

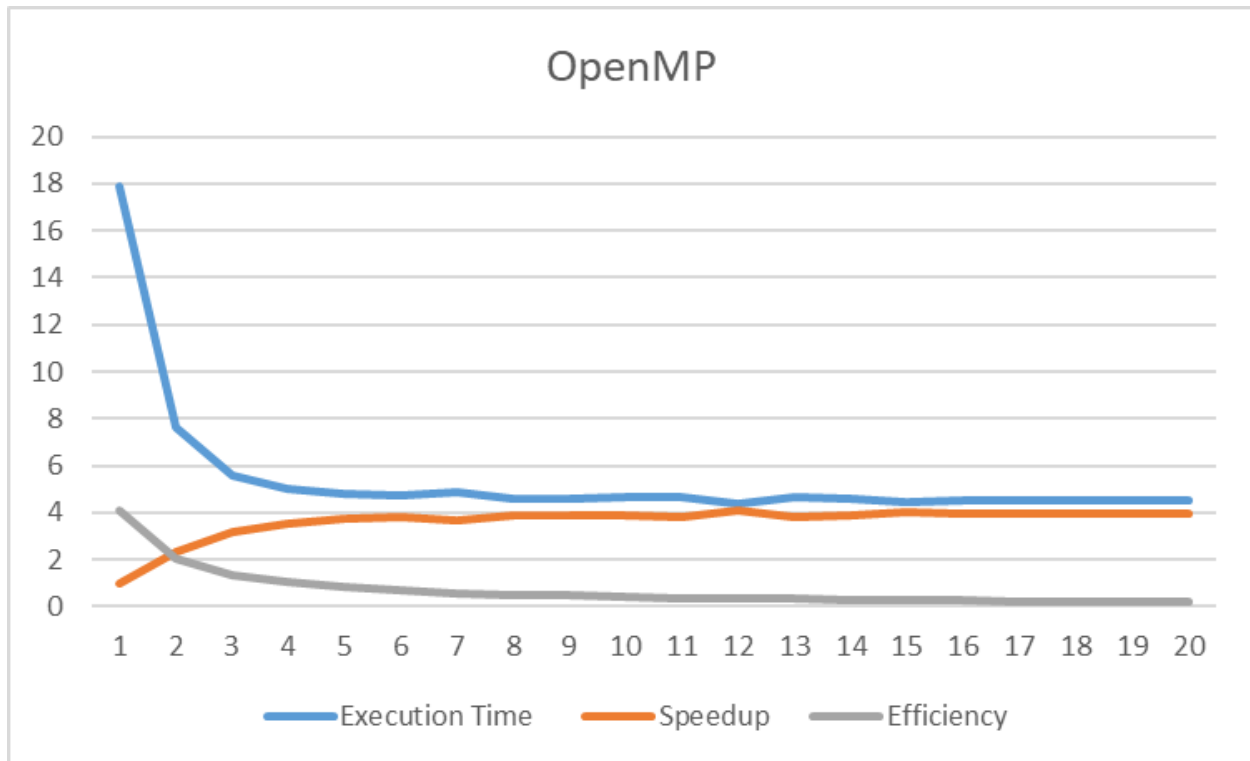
## Efficiency:

since the maximum speedup was acquired using 15 threads and since the differences are minute between using 4 to 20 threads we are going to use 4 threads as the optimal number

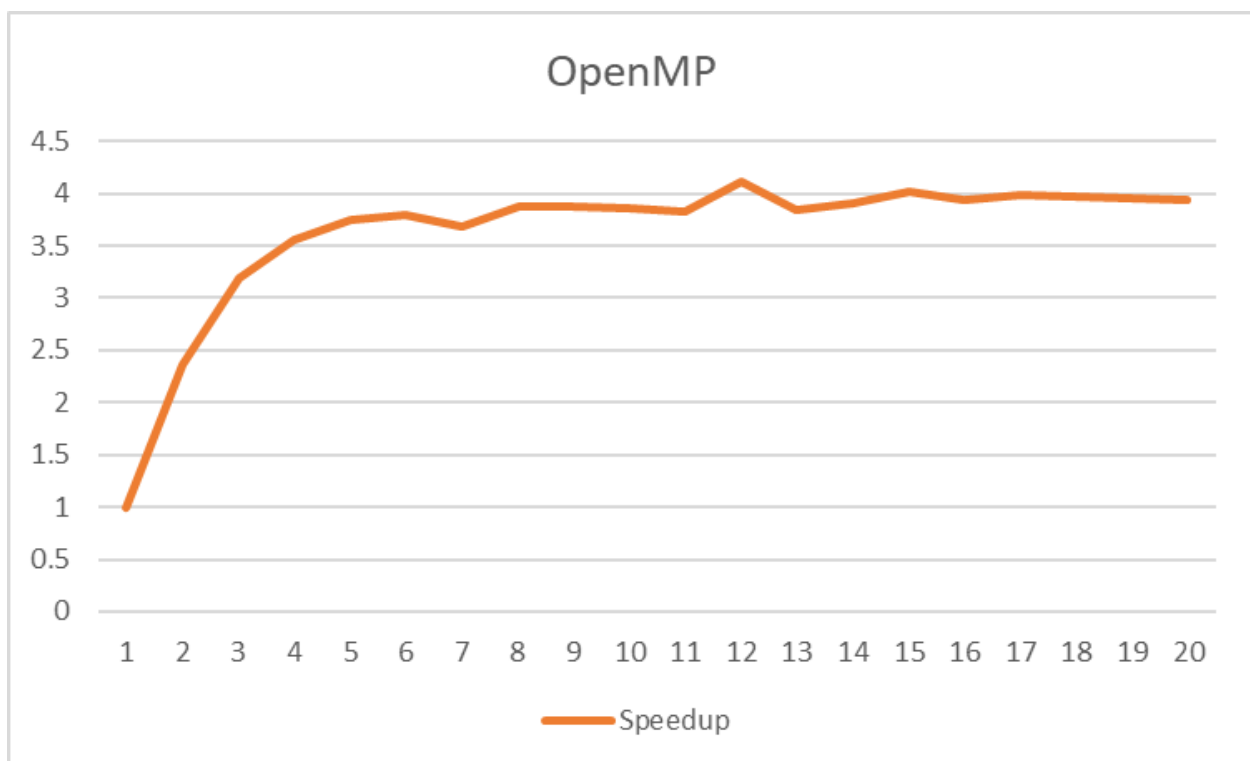
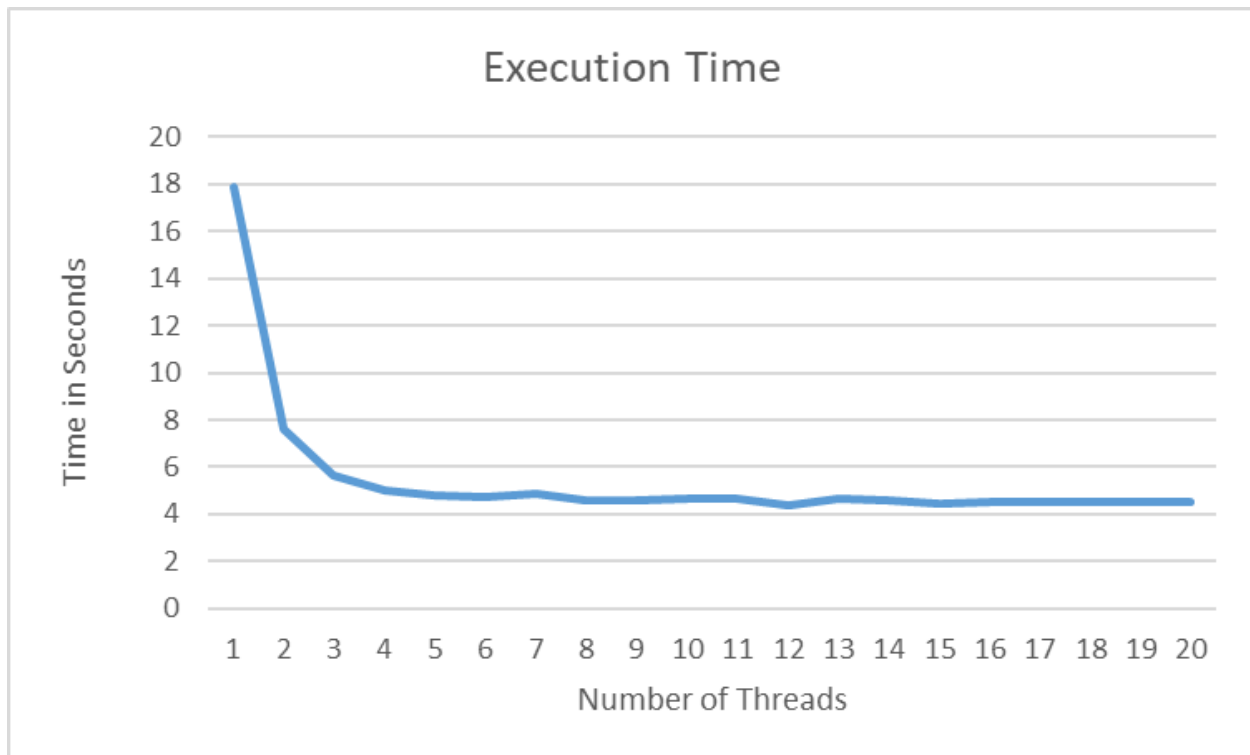
$$Efficiency = \frac{S(p)}{\text{Number of threads}} = 3.5505$$

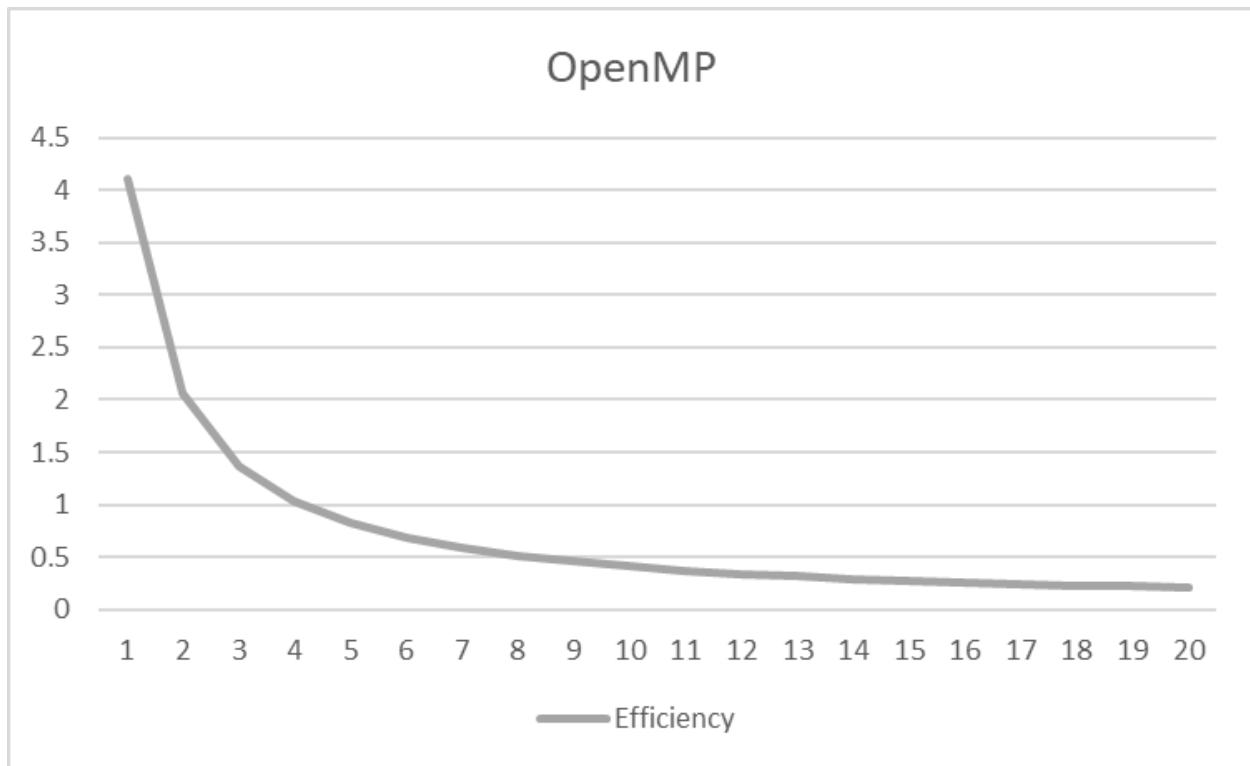
## Scalability:

the graphs below shows the scalability across different numbers of threads.

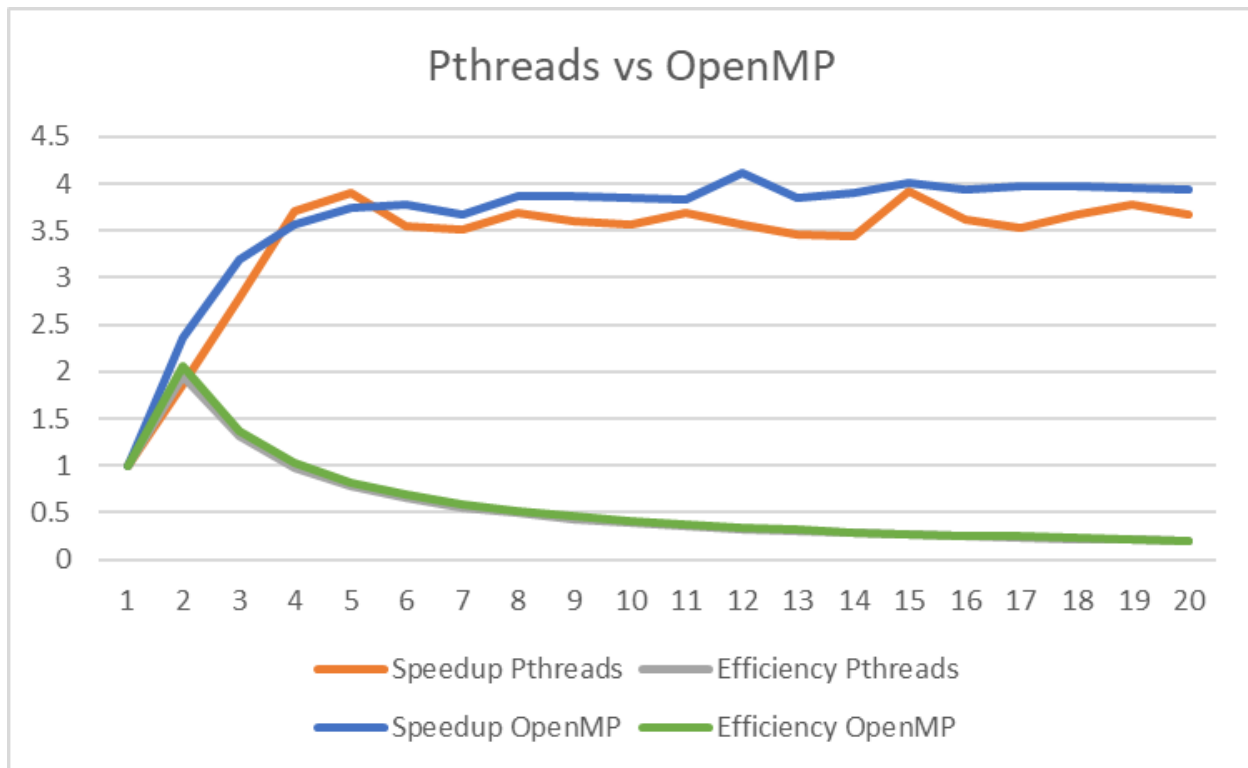








## Comparison and Discussion



As the results of the tests are plotted above, we can come to some inferred conclusions. of the top, both of the two methods showed clear pattern of improved speedup along with increasing the number of parallel threads. Pthreads on one hand shoed a high speedup of 3.71 working 4 parallel threads while OpenMP on the other hand landed a speedup of around 4.11 but far at 12 thread. OpenMP, however; was able to achieve a speedup of 3.55 when only using 4 parallel threads. The fact that OpenMP had a better speedup but at a later stage is indicating much lower efficiency. Objectively, both the methods started to lose “efficiency” after 4 parallel threads as even the fixed efficiency that pthreads had after 4 processors is not justified as better but rather less worse. On the other hand, we realize that OpenMP’s efficiency dropped significantly after only 2 parallel processors. This suggests increased overhead thread management and synchronization and clearly the optimal thread count appears to be four for both the methods but with a slight advantage to Pthreads. The answer to which of the two methods is better is hard as there might be other factors that could have played a role such as workload nature and emulator machine optimization as these could have affected the course of the results. However, based from our tests and taken all other unmeasurable factors aside, we can conclude that Pthreads was more efficient and is the better method.

