Distributed Systems For AI
Parallel Programming
Assignment 1

Mohammed Sraj [GSR/5161/16]

November 13, 2024

1 Introduction

This report presents the work completed for Assignment 1 in Parallel Programming, where problems were implemented using the C programming language with Pthreads and OpenMP. To facilitate analysis and visualization, the numerical data generated by the C programs was exported and saved in CSV format. This data was then read by Python, which was used to process the information and create visualizations, providing deeper insights into the results.

2 Experiment Setup

The experiment was conducted on a **Dell** computer running **Ubuntu 22.04**. The system is equipped with an **Intel Core i5 processor**, which has **2 physical cores**, each capable of running **2 threads**, providing a total of **4 logical CPUs**. The system also has **16GB of RAM**. The program was compiled using **GCC version 11.4.0**,

3 Analysis(Results)

Problem 1

a

Increasing p boosts speedup by dividing the workload among more processors, but it reduces efficiency due to overhead ($\log_2(p)$), particularly for smaller n, where the overhead is more significant. On the other hand, increasing n improves both speedup and efficiency, as the larger workload (n^2) outweighs the overhead, resulting in better processor utilization and enhanced parallel performance. Thus, larger problem sizes are crucial for achieving effective parallelization and minimizing the impact of coordination overhead.

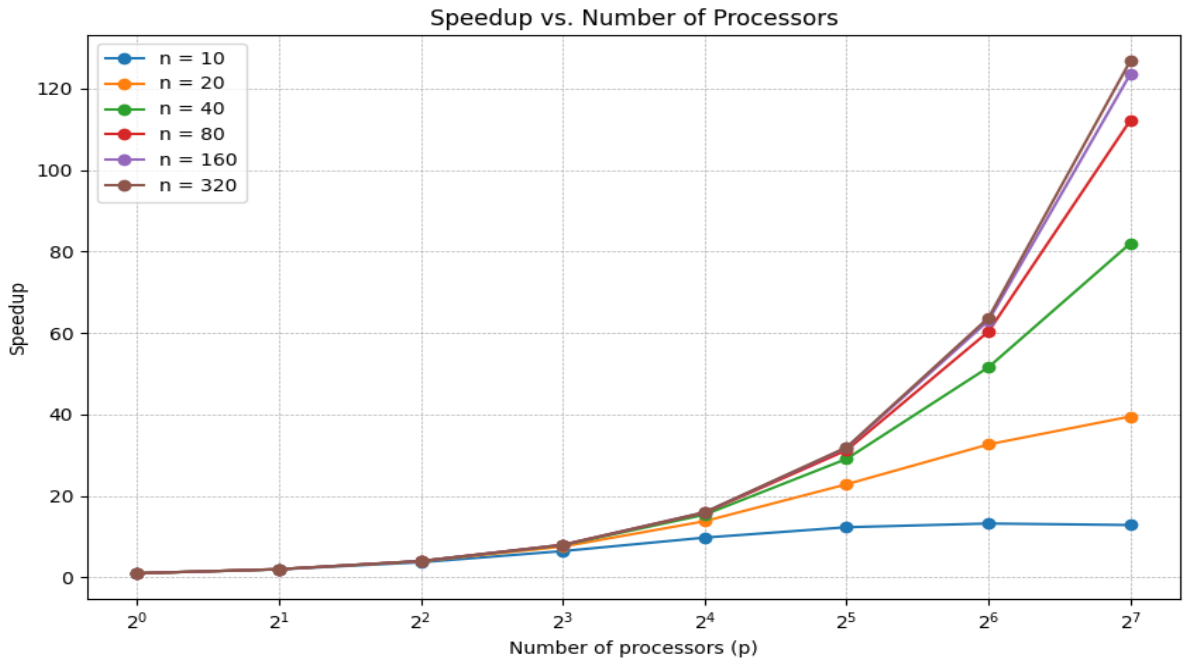

Figure 1: Speedup vs. Number of Processors

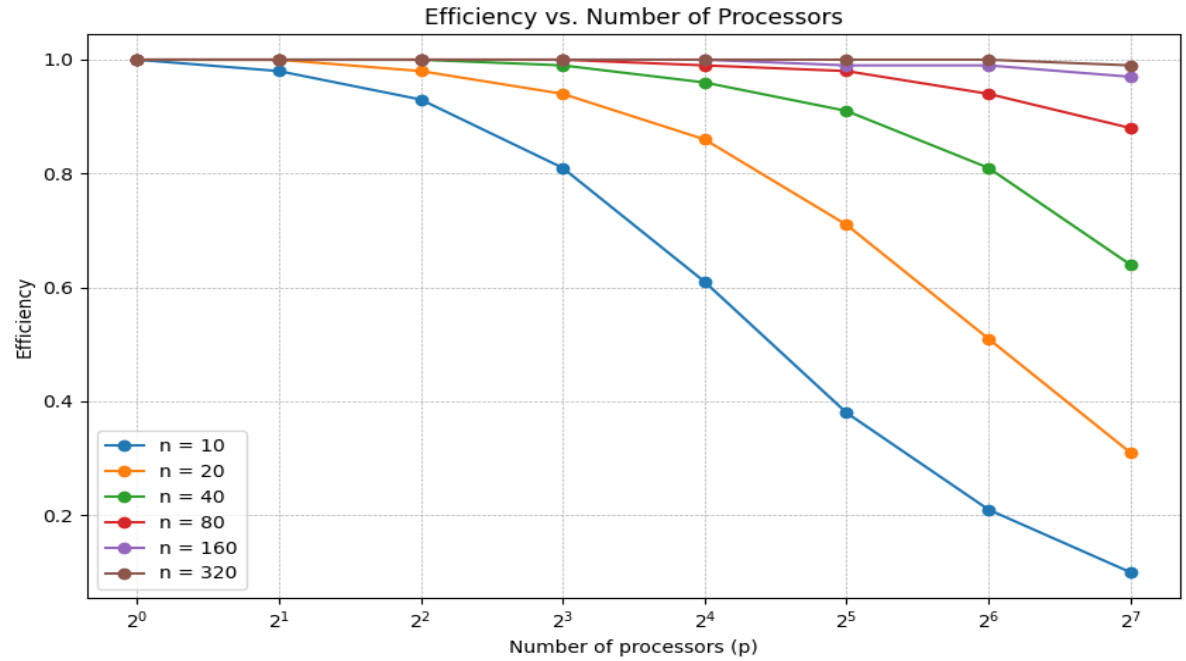

Figure 2: Efficiency vs. Number of Processors

b

Parallel efficiency improves with increasing problem size if T_{overhead} grows slower than T_{serial}, because the majority of the parallel runtime ($T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}$) is dominated by T_{serial}/p. This allows the parallel system to utilize processors more effectively. However, if T_{overhead} grows faster than T_{serial}, the overhead becomes a significant part of T_{parallel}, reducing the overall benefit of parallelization and leading to a decrease in parallel efficiency. This highlights the critical importance of minimizing T_{overhead} to maintain high efficiency in parallel programs.

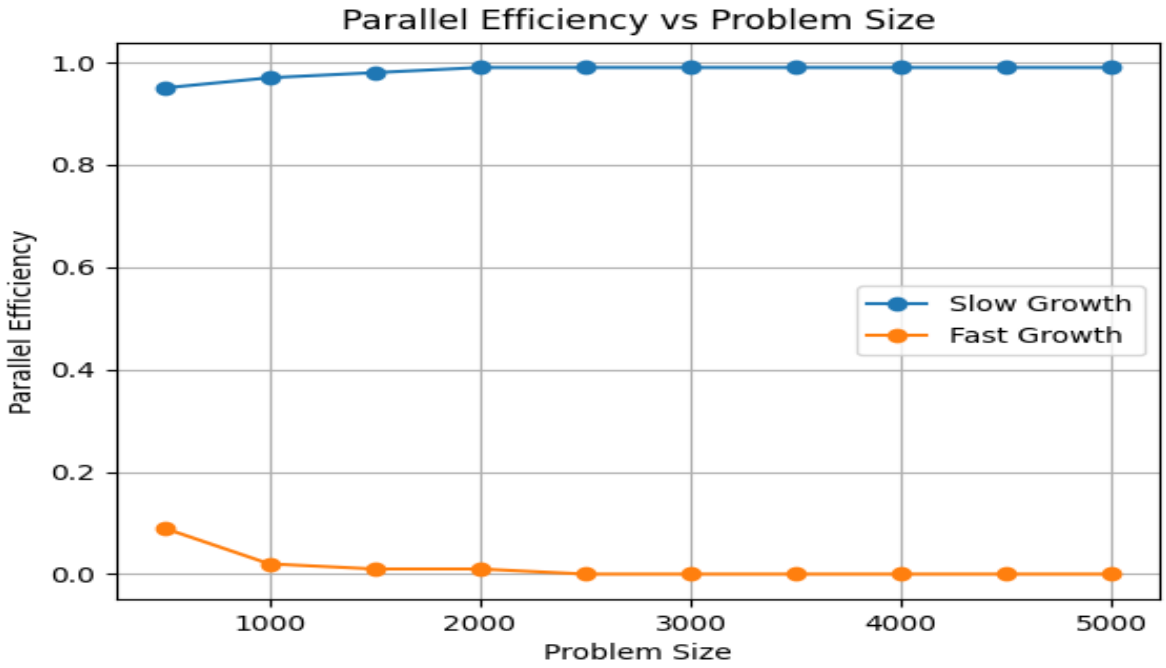

Figure 3: parallel Efficiency vs Problem Size

Problem 2

a

Threads	Version	Execution Time (ms)
2	Mutex	110.000455
2	Barrier	110.000245
2	SharedArray+Barrier	110.000236
4	Mutex	120.000909
4	Barrier	110.000649
4	SharedArray+Barrier	110.000536
8	Mutex	140.002060
8	Barrier	110.001929
8	SharedArray+Barrier	110.001711
10	Mutex	150.002019
10	Barrier	110.002580
10	SharedArray+Barrier	110.002373
12	Mutex	160.002775
12	Barrier	110.002529
12	SharedArray+Barrier	110.002943

Table 1: Execution times for different thread configurations and versions

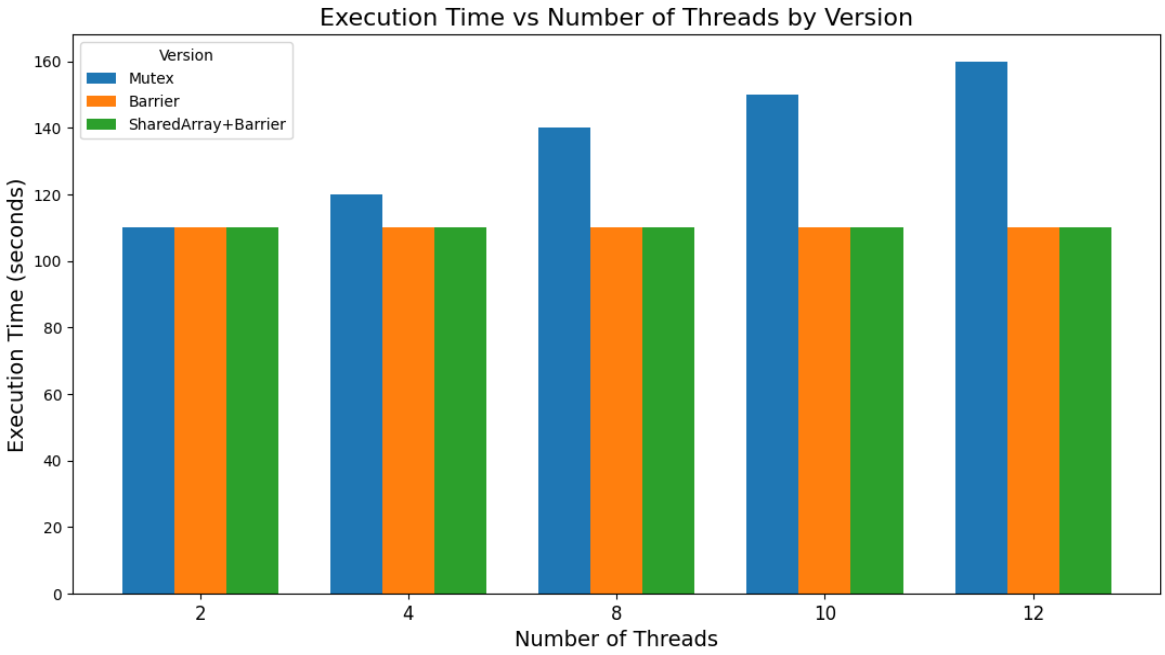

Figure 4: Execution times for different thread configurations and versions

b

In Version 3 (threads with barriers), as the number of threads increases, the parallel execution time also increases. This is because the parallel execution time is given by the formula:

$$T_{\text{parallel}} = \frac{T_{\text{serial}}}{p} + T_{\text{overhead}}$$

where T_{serial} is the serial execution time, p is the number of threads, and T_{overhead} is the overhead introduced by parallelization. As the number of threads exceeds a certain threshold, the overhead T_{overhead} becomes significant enough that the parallel execution time surpasses the serial execution time. Therefore, beyond a specific number of threads, parallelization becomes unprofitable as the parallel execution time exceeds the serial execution time, leading to no performance gains.

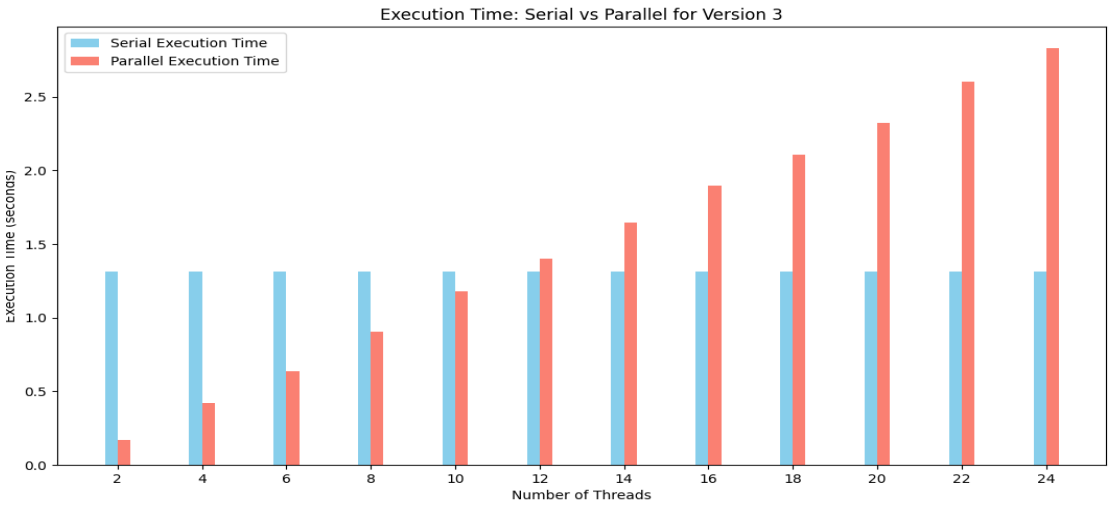

Figure 5: Execution times serial vs parallel

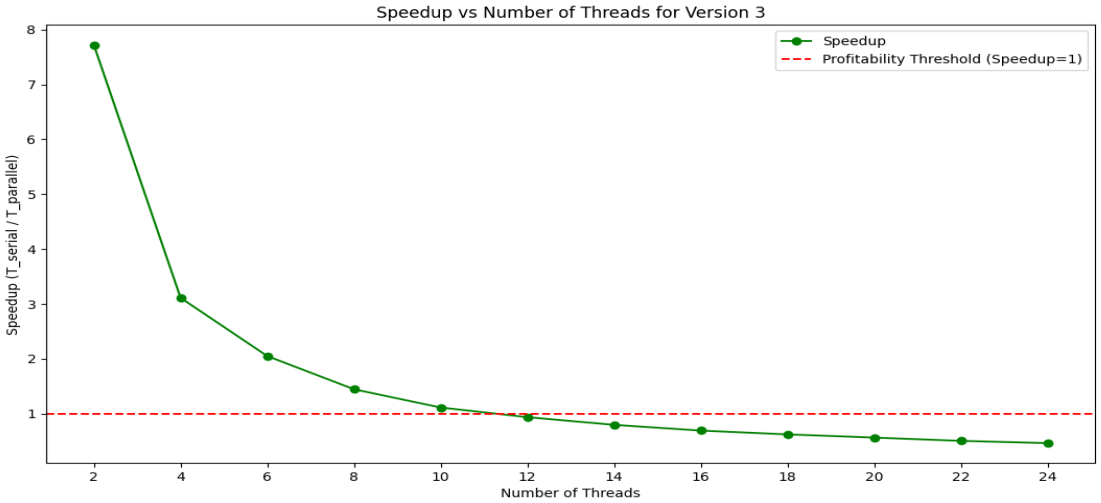

Figure 6: Execution times serial vs parallel

Problem 3

In this question, the data is generated by executing the stream program with the command `OMP_NUM_THREADS=[number of threads] ./stream`, where the number of threads is varied to create a dataset for different thread counts.

The maximum time, minimum time, and average time decrease as the number of threads increases, except in the case of the best rate for all functions, such as `copy` and `triad scale` and `add`.

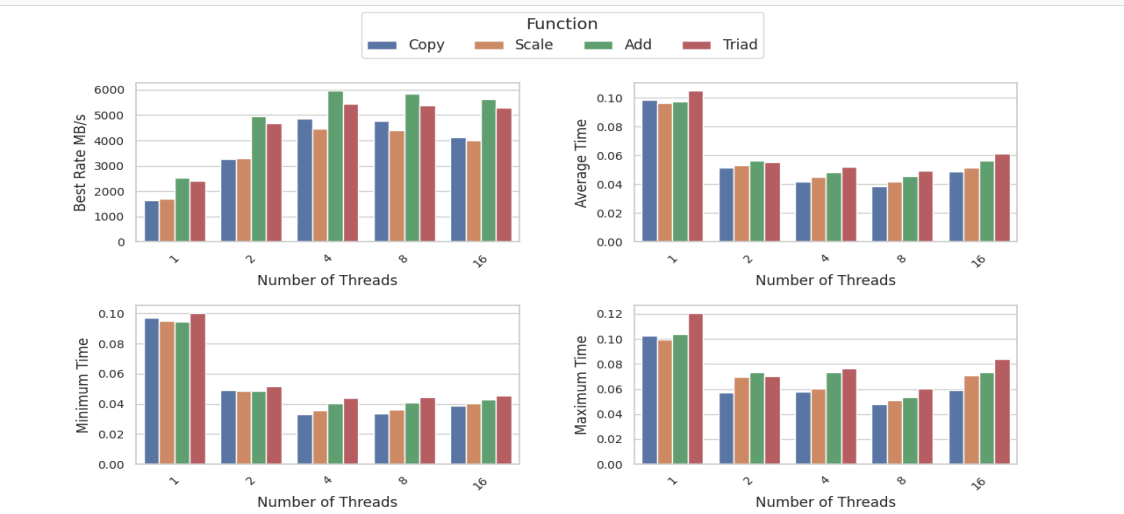

Figure 7: Execution times serial vs parallel

Problem 4

1,a and b

Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency	Validation
1	0.002755	0.003013	0.914378	0.914378	VALID
2	0.002755	0.003241	0.850070	0.425035	VALID
4	0.002755	0.002379	1.158048	0.289512	VALID
8	0.002755	0.002223	1.239275	0.154909	VALID
16	0.002755	0.004820	0.571577	0.035724	VALID

Table 2: Performance Results for Parallel PI Calculation using pthread

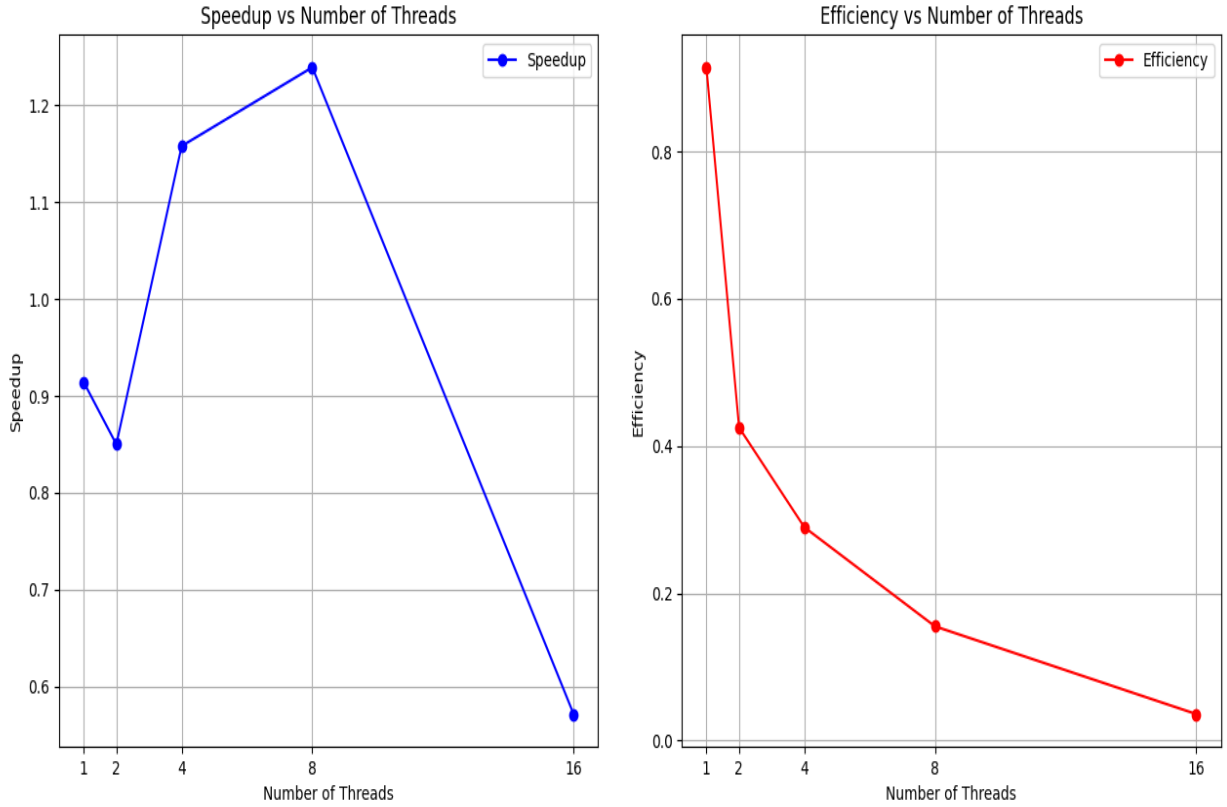

Figure 8: speed up and efficiency pthread

c

The code for serial and pthread parallel implementation is provided in the source code of the assignment

2 a and b

Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency	Validation
1	0.183168	0.198789	0.921420	0.921420	VALID
2	0.183168	0.105739	1.732269	0.866134	VALID
4	0.183168	0.119340	1.534844	0.383711	VALID
8	0.183168	0.105552	1.735337	0.216917	VALID
16	0.183168	0.101488	1.804824	0.112801	VALID

Table 3: Performance Results for Parallel PI Calculation using openMP

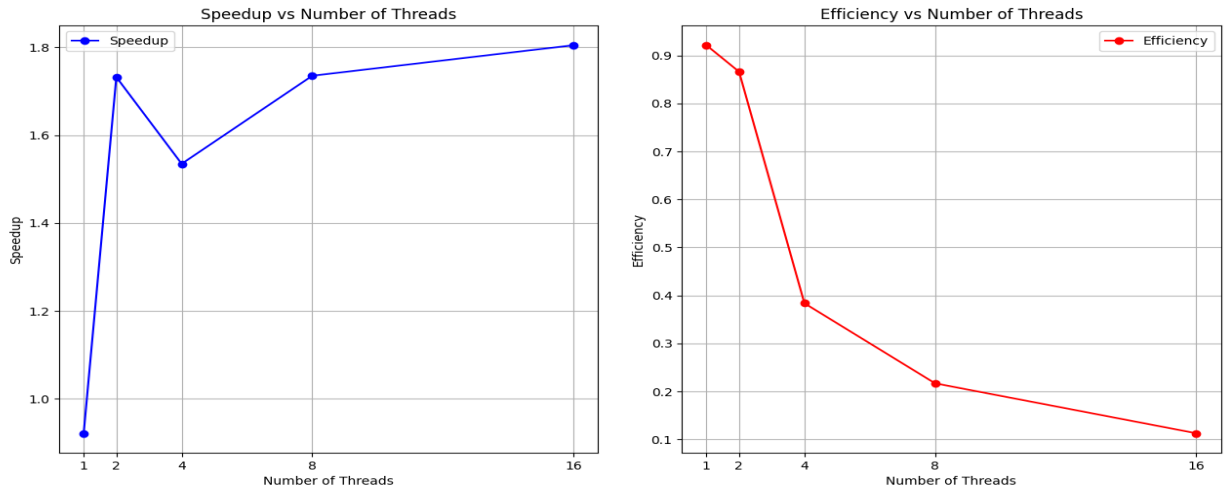

Figure 9: speed up and efficiency OpenMP

c

The code for serial and openMP parallel implementation is provided in the source code of the assignment
When comparing the implementation of π calculation using **pthread** and **OpenMP**, we observe different performance behaviors as the number of threads increases. With **pthread**, the performance tends to decline as the number of threads increases. In contrast, with **OpenMP**, the speedup remains consistent even as the number of threads increases. This is because OpenMP manages threads internally, optimizing their handling and abstracting low-level details. However, in both cases, the efficiency decreases as the number of threads increases. This reduction is attributed to the overhead caused by increased input/output operations as the number of threads grows.

4 Conclusions

From this assignment, I have learned that increasing the number of cores or threads generally improves performance but does not guarantee continuous improvement. Beyond a certain threshold, additional cores can lead to diminishing returns or even decreased speedup and efficiency. This is due to factors such as the limits imposed by Amdahl's Law, increased overhead from thread management and synchronization, hardware resource contention, and the bottleneck created by input/output operations.