Efficient Parallel Matrix Multiplication: Comparative Analysis of Pthreads and OpenMP Implementations

Agit Yesiloz
Department of Computing Sciences
Coastal Carolina University
Conway, SC, USA
ayesiloz@coastal.edu

***Abstract*—This report describes parallel implementations of matrix multiplication using the pthreads library and OpenMP directives in the C programming language. Parallelizing matrix multiplication is essential for enhancing performance, especially when dealing with large matrices. The pthreads implementation involves explicit thread management, while the OpenMP implementation offers a simpler and more concise approach through compiler directives.**

The study includes a comparative analysis of both implementations in terms of various performance metrics, such as computation time, I/O time, and overall execution time. Furthermore, it investigates the scalability of each approach concerning the number of threads or processes employed.

The code is organized to read input matrices from binary files, perform matrix multiplication in parallel, and save the resulting matrix to an output file. Performance assessment involves timing measurements using high-resolution timers to accurately capture computation and I/O overhead.

The results illustrate the effectiveness of parallelization techniques in reducing computation time, considering the overhead incurred by threading or parallelization constructs. The report concludes with insights into the strengths and limitations of each parallelization approach, providing guidance for selecting the most suitable method based on specific application requirements and hardware characteristics.

***Index Terms*—Parallel Computing, Matrix Multiplication, Pthreads, OpenMP, Parallelization Techniques**

I. Introduction

Matrix multiplication serves as a foundational operation in numerous scientific and engineering domains, underpinning algorithms in fields such as numerical simulations, signal processing, and machine learning. With the increasing size of matrices, the computational complexity of matrix multiplication escalates significantly. Consequently, optimizing the performance of matrix multiplication algorithms becomes imperative for effectively tackling large-scale problems [1].

Parallel computing emerges as a promising avenue to expedite matrix multiplication by distributing computational tasks across multiple processing units, including CPU cores or threads. In this context, pthreads and OpenMP represent two prevalent parallel programming paradigms in the C programming language. Pthreads furnishes a low-level interface for managing threads, affording fine-grained control over thread creation, synchronization, and communication. Conversely, OpenMP offers a high-level approach through compiler directives, streamlining the parallelization process and enabling developers to parallelize code with minimal alterations.

This report undertakes a comparative examination of parallel matrix multiplication implementations using pthreads and OpenMP in the C programming language. The objective is to assess the performance and scalability of each approach, providing insights into the trade-offs between explicit thread management and directive-based parallelization.

The study encompasses diverse facets of parallel matrix multiplication, encompassing input/output (I/O) operations, computation time, overall execution time, and scalability concerning the number of threads or processes employed. Performance metrics are gauged using high-resolution timers to precisely capture the time allocated to computation and I/O operations, facilitating a comprehensive analysis of the parallelization overhead.

The subsequent sections of this report are structured as follows: Section 2 furnishes an overview of the pthreads and OpenMP implementations, elucidating the parallelization strategies and code organization. Section 3 delineates the experimental setup, encompassing the hardware environment, input data, and performance evaluation methodology. Section 4 delineates the results of the performance analysis, juxtaposing the pthreads and OpenMP implementations concerning execution time and scalability. Finally, Section 5 deliberates on the findings, accentuating the strengths and limitations of each parallelization approach, and proffers recommendations for selecting the most appropriate method predicated on specific application requisites and hardware attributes.

This work makes the following contributions:

- It provides a detailed comparative analysis of parallel matrix multiplication implementations using pthreads and OpenMP in the C programming language. [2]By examining both approaches, this study offers insights into the trade-offs between explicit thread management and directive-based parallelization, helping developers

choose the most appropriate method for their specific requirements.

- The study evaluates the performance of pthreads and OpenMP implementations in terms of computation time, I/O time, and overall execution time. By employing high-resolution timers and carefully designed experiments, it accurately captures the performance characteristics of each parallelization technique and identifies factors influencing their efficiency.
- Through experiments varying the number of threads or processes, this work assesses the scalability of pthreads and OpenMP implementations. Understanding how performance scales with increasing computational resources is crucial for effectively utilizing parallel computing in large-scale applications.
- The report provides practical insights into parallel programming techniques, demonstrating how pthreads and OpenMP can be leveraged to improve the performance of matrix multiplication algorithms. It discusses common pitfalls, optimization strategies, and best practices for parallelization, empowering developers to optimize their code effectively.
- By presenting a comprehensive comparison and analysis, this work guides developers in selecting the most suitable parallelization approach for their specific application domains and hardware platforms. It highlights the strengths and limitations of each method and provides recommendations based on performance considerations and ease of implementation.

This paper is organized as follows. Section II provides an outline for the software framework design.

In Section **??**, the experiments are outlined and results are discussed to evaluate the enhancements. Section **??** concludes the paper.

II. Design

In this section, we'll delve into the design and implementation details of parallel matrix multiplication using the C programming language. Our goal here is to make matrix multiplication more efficient and scalable by tapping into parallel computing techniques. We'll discuss the choices we made regarding the program's structure, algorithms, and methods used for parallelization.

We'll walk through how we managed threads, synchronized their operations, and handled various aspects of parallel programming in C, focusing on both pthreads and OpenMP approaches. Additionally, we'll explain how we integrated high-resolution timers and devised methodologies for accurately measuring the impact of parallelization on computation time, I/O operations, and overall execution time.

This section aims to provide you with a clear understanding of how parallel matrix multiplication is implemented in C. We'll break down the key concepts and decisions behind our design, using examples and explanations to help you grasp the complexities of parallel programming and the potential benefits it offers for speeding up matrix computations..

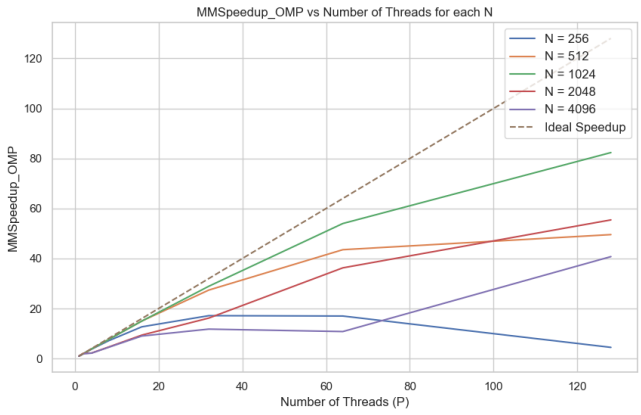

Fig. 1. Matrix Generation Process

A. Matrix Generation

To generate matrices for use in matrix multiplication experiments, a C program makematrix.c was developed. This program takes three command-line arguments: the number of rows (row), the number of columns (col), and the filename (filename) to which the matrix data will be written.

Dynamic Memory Allocation: The malloc2D() function is responsible for dynamically allocating memory for a 2D array of doubles using a contiguous block of memory. This ensures efficient memory usage and cache locality, as all elements of the matrix are stored in adjacent memory locations.

Matrix Initialization: The initialize2D() function populates the allocated matrix with sequential double values. By iterating over each element of the matrix and assigning a unique double value based on its position, this function ensures that each element has a distinct value, facilitating differentiation during matrix multiplication.

File Output: The write2D() function writes the matrix data to a binary file in row-major order. It begins by storing the dimensions of the matrix (rows and cols), followed by the double values of each element. This binary file format optimizes storage space and enables efficient reading of matrix data during subsequent matrix multiplication operations.

This matrix generation process provides a systematic and efficient approach to creating matrices of specified dimensions for use in matrix multiplication experiments.

1) Matrix Generation Process: The matrix generation process involves the following steps:

1) Specify the dimensions of the matrix (number of rows and columns) and the filename for the output matrix data file.
2) Allocate memory for the matrix using the malloc2D() function.
3) Initialize the matrix with sequential double values using the initialize2D() function.
4) Write the matrix data to a binary file using the write2D() function.

Figure 2 illustrates the matrix generation process flow.

```c
int main(int argc, char *argv[]) {
    int row;
    int col;
    char *filename;

    if (argc != 4) {
        printf("Usage: %s <row> <col> <filename>\n", argv[0]);
        return 1;
    }
    row = atoi(argv[1]);
    col = atoi(argv[2]);
    filename = argv[3];
    double **matrix = malloc2D(row, col);

    initialize2D(matrix, row, col);
    write2D(matrix, row, col, filename);

    free(matrix);
    return 0;

}
```

Fig. 2. Matrix Generation Process

B. Matrix Multiplication with Pthreads

The matrix multiplication algorithm implemented in `pth_matrix_matrix.c` utilizes the pthreads library to achieve parallelism. Below is an overview of the program's structure and key components:

- **Header Inclusions:** The program starts by including necessary header files such as `<stdio.h>` and `<stdlib.h>` to handle standard I/O operations and memory allocation.
- **Global Variables:** Global variables are declared to track timing information for computation and I/O operations.
- **Memory Allocation:** The `malloc2D` function dynamically allocates memory for 2D arrays of double precision floating-point numbers, ensuring efficient memory usage and contiguous storage for matrices.

```c
void *pth_matrix_matrix(void* rank) {
    long my_rank = (long) rank;
    int local_m = rowA/thread_count;

    int remainder = rowA % thread_count;
    int my_first_row = my_rank*local_m+MIN(my_rank, remainder);
    int my_last_row = (my_first_row+local_m)-(my_rank < remainder ? 0:1 );
    for (int i = my_first_row; i <= my_last_row; i++) {
        for(int j=0; j<colB; j++){
            C[i][j] = 0;
            for(int k=0; k<colA; k++){
                C[i][j] += A[i][k] * B[k][j];

            }

        }
    }
    return NULL;
}  /* Pth_mat_vect */
```

Fig. 3. Snippet of the `pth_matrix_matrix.c`

- **Matrix Multiplication Function:** The core of the program lies in the `pth_matrix_matrix` function, which performs matrix multiplication in parallel using pthreads. Each thread computes a subset of the output matrix elements.
- **Main Function:** The `main` function parses command-line arguments to specify input and output file names along with the desired number of pthreads threads to use. Input matrices A and B are read from binary files provided as command-line arguments.
- **Parallel Execution:** The program employs pthreads to parallelize the matrix multiplication operation. Threads are created using `pthread_create` and synchronized using `pthread_join`.
- **Result Output:** Once matrix multiplication is completed, the resulting matrix C is written to an output file specified in the command-line arguments. Timing information, including computation time and I/O time, is recorded and printed to the console.

This subsection provides insights into the implementation of matrix multiplication with pthreads, along with a snippet of the code for reference (see Figure 3).

C. Matrix Multiplication with OpenMP

The matrix multiplication algorithm is implemented in omp-matrix.c leverages the OpenMP library to achieve parallelism. Here's a concise summary of the program's structure and functionality:

- **Header Inclusions:** The program begins by including necessary header files such as `<stdio.h>` and `<stdlib.h>` for standard I/O operations and memory allocation, as well as `<omp.h>` for OpenMP support.
- **Memory Allocation:** Similar to the pthreads version, the `malloc2D` function is utilized to dynamically allocate memory for 2D arrays of double precision floating-point numbers.
- **Main Function:** The `main` function parses command-line arguments to specify input and output file names, along with the desired number of OpenMP threads to use.
- **Matrix Multiplication:** The core computation is performed within a parallel region using OpenMP directives. The `#pragma omp parallel for` directive distributes the iterations of nested loops across multiple threads for efficient parallel execution.
- **Result Output:** After matrix multiplication is completed, the resulting matrix C is written to an output file specified in the command-line arguments. Timing information, including computation time and I/O time, is recorded and printed to the console.

This subsection offers a succinct overview of the OpenMP-based matrix multiplication implementation, along with a snippet of the code for reference (see Figure 4).

III. Experimentation and Results

In this section, the experimental setup, execution procedure, and obtained results are discussed. The matrix multiplication algorithms implemented using pthreads and OpenMP were executed on the Expanse portal, a high-performance computing environment. Subsequently, the output files generated were analyzed using a Python script to generate performance graphs.

```
GET_TIME(COMPUTATION_START);
#pragma omp parallel for
for(int i=0; i<rowA; i++){
    for(int j=0; j<colB; j++){
        C[i][j] = 0;
        for(int k=0; k<colA; k++){
            C[i][j] += A[i][k] * B[k][j];

        }
    }
}
GET_TIME(COMPUTATION_END);
fwrite(&rowA, sizeof(int), 1, fileC);
fwrite(&colB, sizeof(int), 1, fileC);
for(int i=0; i<rowA; i++){
    for(int j=0; j<colB; j++){
       fwrite(&C[i][j], sizeof(double), 1, fileC);
    }
}
```

Fig. 4. Snippet of the `omp_matrix_matrix.c`

Finally, a comparison was conducted between the pthreads and OpenMP implementations in terms of execution time, speedup, and efficiency.

A. Experimental Setup

The experiments were conducted on the Expanse portal, a powerful computing platform equipped with multi-core processors. The pthreads and OpenMP implementations of matrix multiplication were compiled and executed using GCC compiler with optimization flags enabled.

B. Execution Procedure

The matrix multiplication programs were executed with various input sizes and thread counts to evaluate their performance under different configurations. Command-line arguments were provided to specify the input matrices' filenames, output filename, and the number of threads to utilize.

Upon execution, the programs generated output files containing the resulting matrix and recorded timing information for computation and I/O operations.

C. Analysis and Graph Generation

A Python script was developed to analyze the output files and extract relevant metrics such as computation time, I/O time, and matrix dimensions. Using the pandas and matplotlib libraries, the script generated graphs illustrating the relationship between the number of threads and various performance metrics.

```
#!/bin/bash
#SBATCH --job-name="omp"
#SBATCH --output="pth_matrix_vector.%j.%N.out"
#SBATCH --partition=compute
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=16GB
#SBATCH --account=ccu108
#SBATCH --export=ALL
#SBATCH -t 10:00:00

module purge
module load cpu
module load slurm
module load gcc

make clean all
for (( N=256; N<=4096; N*=2 ))
do
  ./make_matrix $N $N A.dat
  ./make_matrix $N $N X.dat
  ./matrix_matrix A.dat X.dat X2
   for (( P=1; P<=128; P*=2 ))
   do
     #Run the job
     echo "running $N program on $P threads"
      ./pth_matrix_matrix A.dat X.dat Y2 $P
      ./omp_matrix_matrix A.dat X.dat Y3 $P
    done

    echo "-------------------------------------------------"
done
```

Fig. 5. Experimental Setup on the Expanse Portal

```
# Add ideal line for speedup
if 'Speedup' in metric:
    sns.lineplot(x=n_data['P'], y=n_data['P'], label='Ideal Speedup', linestyle='--')

# Add ideal curve for specified metrics
if metric in ["OverallPthread", "MMPthread", "Overall_OMP", "MM_OMP"]:
    max_time = n_data[metric].max()
    ideal_time = max_time / n_data['P']
    sns.lineplot(x=n_data['P'], y=ideal_time, label='Ideal Time', linestyle='--')

plt.xlabel('Number of Threads (P)')
plt.ylabel(metric + ' (s)' if metric in time_metrics else metric)
plt.title(f'{metric} vs Number of Threads for each N')
plt.legend(loc='upper right', fontsize=12)
plt.savefig(f'../data/original_graphs/{metric}_vs_p.png')
```

Fig. 6. Snippet of Python Script that generates images

D. Comparison of Results

The performance metrics obtained from the pthreads and OpenMP implementations were compared in terms of execution time, speedup, and efficiency. Speedup was calculated as the ratio of the sequential execution time to the parallel execution time, while efficiency was computed as the ratio of speedup to the number of threads utilized.

1) Execution Time: The execution time of the pthreads and OpenMP implementations of matrix multiplication was measured to evaluate their computational efficiency. Execution time refers to the total elapsed time required to complete the matrix multiplication task, including both computation and I/O operations.

For each implementation, the execution time was recorded across different input sizes and thread counts. The aim was to analyze how the execution time varies with the number of threads utilized and the size of the input matrices.

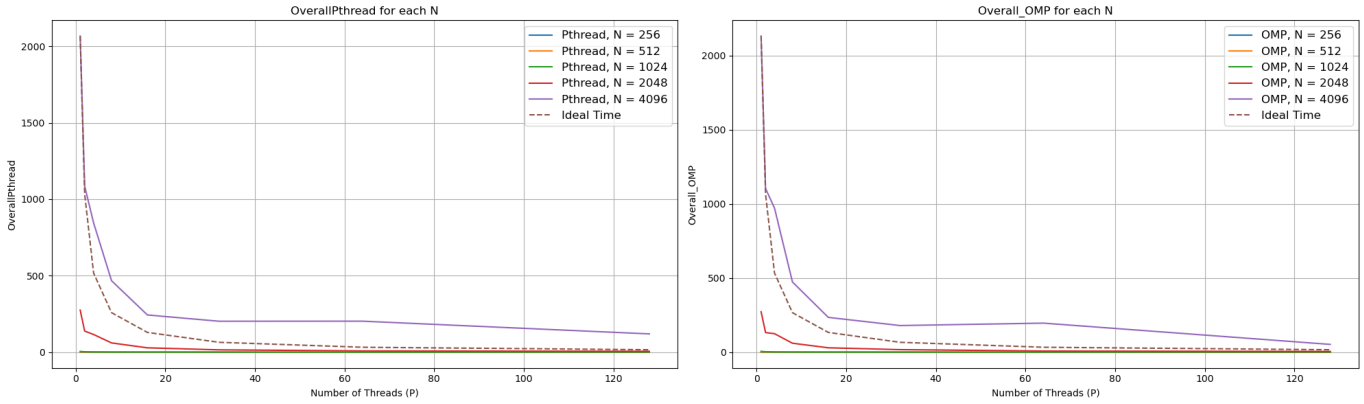

Fig. 7. Overall time using Pthread and OpenMP

The similarity in overall execution time between pthreads and OpenMP (OMP) implementations, as depicted in the graph, indicates that both parallelization approaches are effective in achieving comparable performance for overall task. Both Pthread and OpenMP exhibit similar performance trends. As the number of threads increases, there is a sharp decline in time, which eventually. This suggests that both Pthread and OpenMP benefit from increased parallelism up to a point, after which adding more threads does not significantly improve performance. The similarity in matrix multiplication time between pthreads and

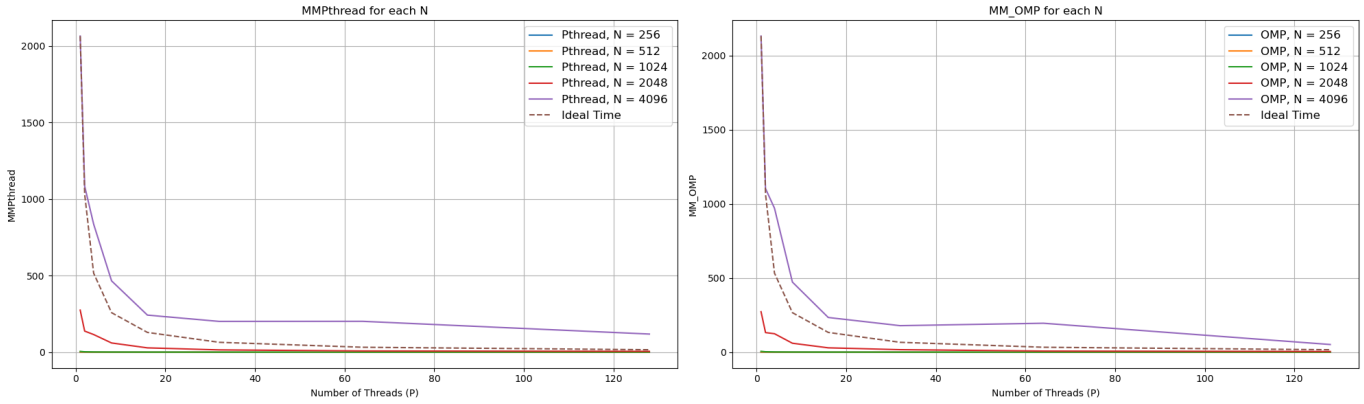

Fig. 8. Matrix multiplication time using Pthread and OpenMP

OpenMP (OMP) implementations suggests that both parallelization approaches achieve comparable efficiency in distributing the computational workload across multiple threads.

2) Speedup: Speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. [3] It's a measure of the effectiveness of parallelization and is defined as:

$$S = \frac{T_1}{T_p}$$

where:

- S is the speedup,

- T_1 is the execution time of the sequential algorithm,
- T_p is the execution time of the parallel algorithm using p processors.

The ideal speedup is linear, i.e., doubling the number of processors doubles the speed. However, in practice, due to overheads such as communication and synchronization between processors, the speedup is often sub-linear.

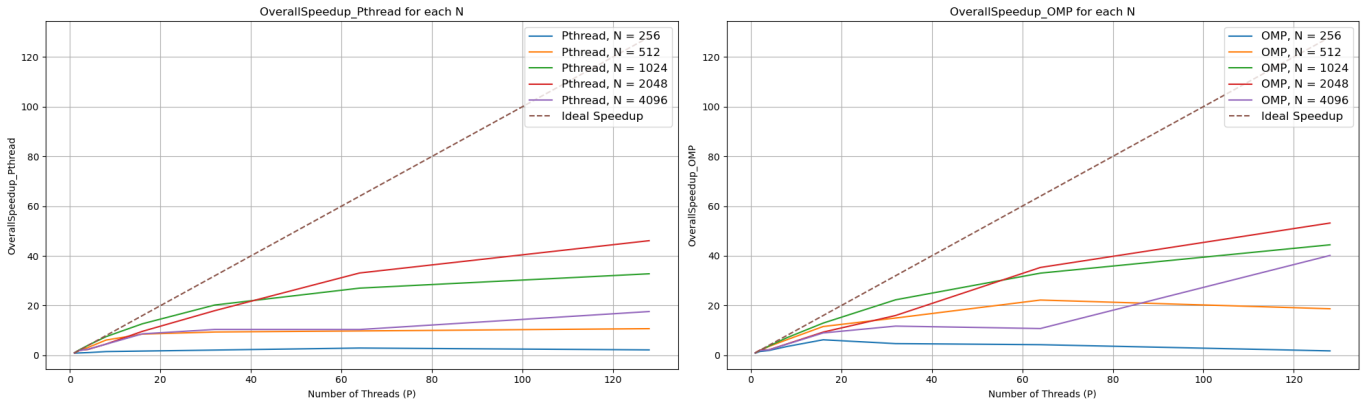

Fig. 9. Overall Speedup for Pthread and OpenMP

From the graphs, it's evident that both PThread and OMP exhibit similar performance trends. As the number of threads increases, so does the overall speedup but it doesn't reach the ideal speedup. This suggests that both PThread and OMP benefit from increased parallelism up to a point, after which adding more threads does not significantly improve performance.

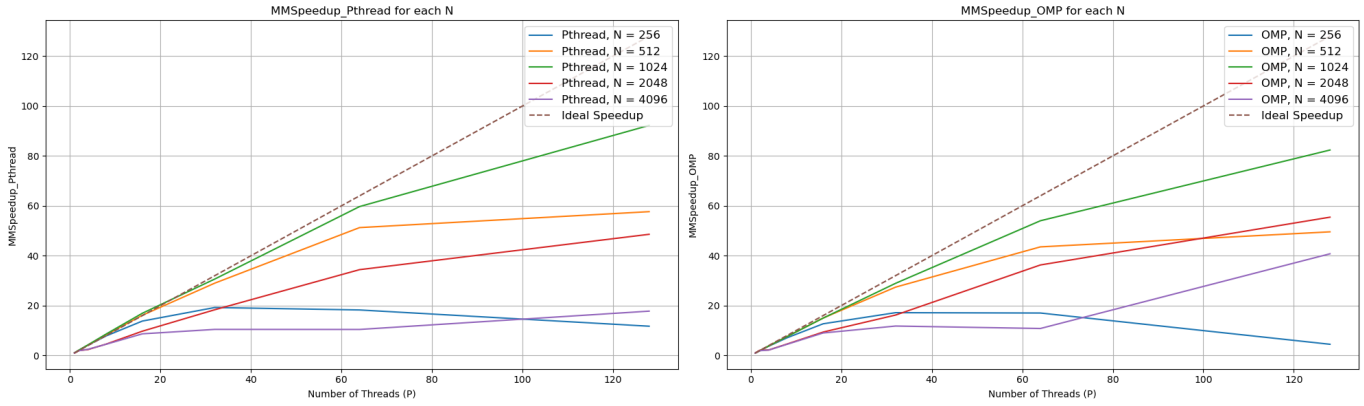

Fig. 10. Matrix multiplication speedup for Pthread and OpenMP

3) Efficiency: Efficiency in parallel computing refers to the utilization of resources, particularly how effectively the available processors are used to perform a computation. [**?**] It is calculated as the ratio of the speedup achieved by a parallel algorithm to the number of processors used. The efficiency formula is as follows:

$$E = \frac{S}{P} \times 100\%$$

Where:

- E is the efficiency,
- S is the speedup, and
- P is the number of processors.

Efficiency values range from 0% to 100%. An efficiency of 100% indicates perfect utilization of resources, where doubling the number of processors doubles the speedup. However, in practice, efficiency values are often less than 100% due to overheads such as communication and synchronization among processors.

In summary, efficiency quantifies how effectively the available computing resources are utilized in parallel execution, providing insights into the scalability and performance of parallel algorithms.

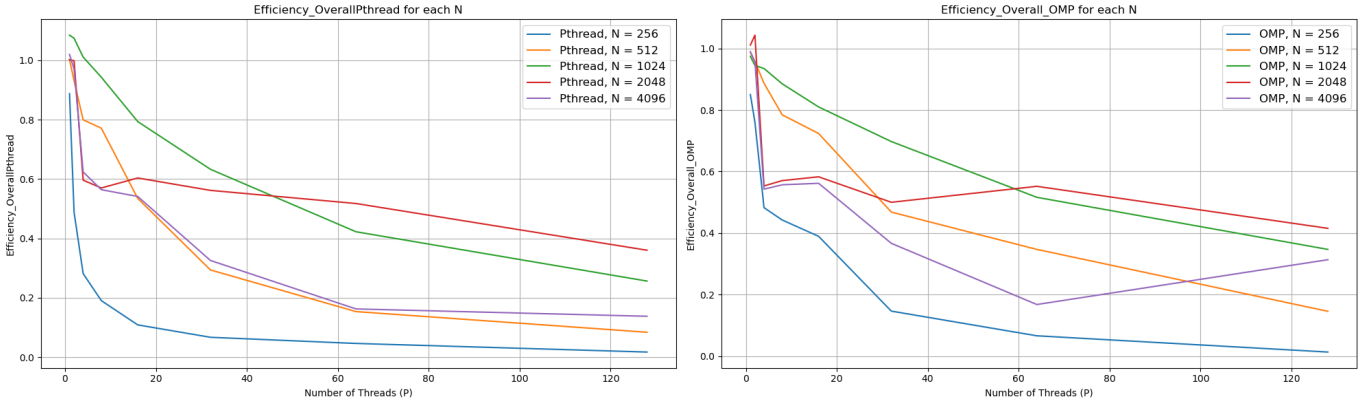

Fig. 11. Overall Efficiency for Pthread and OpenMP

From the graphs, it's apparent that both PThread and OpenMP demonstrate analogous performance trends. With an increase in the number of threads, there is a noticeable decrease in efficiency. This observation implies that although augmenting the thread count initially enhances performance, there exists a saturation point beyond which further thread additions result in less efficient utilization of computational resources. [1]

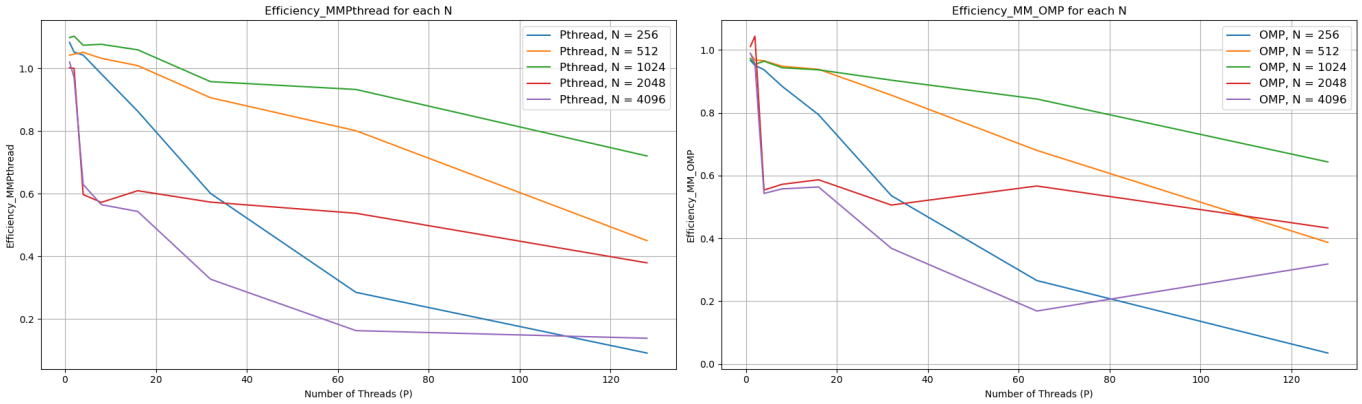

Fig. 12. Matrix Multiplication Efficiency for Pthread and OpenMP

As the number of threads increases, the efficiency decreases. This suggests that while increasing the number of threads does improve performance initially, there is a point of diminishing returns after which adding more threads leads to less efficient utilization of computational resources.

E. Performance Comparison

It is evident that OpenMP (OMP) exhibits better speedup compared to PThread in terms of parallel computing. This suggests that OpenMP achieves faster execution times, leading to improved overall performance. However, despite this difference in speedup, both PThread and OpenMP demonstrate similar efficiency levels. This indicates that although OpenMP may provide faster computation through better speedup, both approaches utilize computational resources with comparable efficiency, ensuring effective utilization of available hardware resources. [2]

IV. Conclusions

In conclusion, the implementation of matrix multiplication using both PThread and OpenMP parallelization techniques provided valuable insights into parallel computing performance. Through experimentation and analysis, several key conclusions can be drawn:

1) **Performance Trends:** Both PThread and OpenMP implementations exhibited similar performance trends, with increasing parallelism leading to improved overall execution time.
2) **Speedup and Efficiency:** While OpenMP demonstrated better speedup in parallel computing compared to PThread, both approaches showed similar efficiency levels. This suggests that although OpenMP achieves faster execution times, both PThread and OpenMP utilize computational resources effectively.
3) **Scalability:** The scalability of both PThread and OpenMP implementations was evident, with performance improvements observed as the number of threads increased. However, there were diminishing returns in speedup beyond a certain point, indicating limitations in scaling parallelism.
4) **Optimization Potential:** Further optimization of parallel algorithms and resource management strategies could enhance performance and efficiency. Fine-tuning parameters such as thread count and workload distribution may lead to better utilization of available resources.
5) **Considerations for Future Work:** Future work could explore additional parallelization techniques, optimize memory access patterns, and investigate alternative algorithms to further improve performance and scalability.

In summary, the comparison between PThread and OpenMP implementations provides valuable insights into the trade-offs between speedup, efficiency, and scalability in parallel computing. By leveraging the strengths of each approach and optimizing parallel algorithms, significant performance gains can be achieved in computational tasks, contributing to advancements in parallel computing research and applications.

References

[1] G. Schryen, "Speedup and efficiency of computational parallelization: A unifying approach and asymptotic analysis," *Department of Management Information Systems, Paderborn University*, Year.

[2] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Title of the technical report," Technical Report, March 1989. [Online]. Available: https://www.osti.gov/biblio/6215375

[3] Wikipedia. (2024) Amdahl's law. [Online]. Available: