High Performance Computing

OpenMP + MPI Practice: Parallelization of Heat Diffussion Equation

Ariza Pérez, José Ramon
Buturuga, Mihaela Alexandra

University of Lleida
Higher Polytechnic School
Master's Degree in Informatics Engineering

Course 2024 - 2025

May 18th, 2025

Contents

1 Serial Code

This C program simulates heat diffusion in a 2D square area using a grid. It starts with two hot diagonal lines on a cold background. The program then calculates how the temperature changes over time, updating each point based on its neighbors, while keeping the edges cold. Finally, it creates a color image (BMP) showing the final heat distribution, with red being hot and blue being cold.

2 Analysis of the Hotspots

In the analysis of the heat diffusion simulation implemented in C, the main hotspots identified for parallelization using a hybrid approach of OpenMP and MPI are the initialization of the matrix and the solving of the heat equation. These sections are the most computationally intensive, especially with large grid sizes due to the repetitive operations over each cell.

By combining MPI for distributed memory parallelism across multiple processes (nodes) and OpenMP for shared memory parallelism within each process (core/thread), we aim to significantly reduce runtime by distributing the workload both across nodes and among threads within each node. This hybrid model maximizes hardware utilization and scalability.

2.1 Initialization of the Heat Matrix

The initialization of the matrix remains an ideal candidate for parallelization in the hybrid MPI and OpenMP implementation. This phase consists of assigning initial temperature values to each cell in the grid prior to the simulation. Since each cell is initialized independently, the task is embarrassingly parallel, with no data dependencies involved.

In this hybrid approach, the global grid is first partitioned among MPI processes, where each process is responsible for initializing a subdomain of the matrix. Within each process, OpenMP is then used to parallelize the initialization loop, distributing the workload across multiple threads. This two-level parallelization strategy is expected to reduce the setup time of the simulation by utilizing both inter-node (MPI) and intra-node (OpenMP) parallelism effectively.

2.2 Solving the Heat Diffusion Equation

The core of the heat diffusion simulation involves repeated temperature updates based on neighboring points. In the hybrid parallel approach, the grid is divided into row groups, each managed by an MPI process. OpenMP further accelerates this within each process by assigning parts of the row group to different threads. Updates can happen in parallel since they only rely on the previous state.

To ensure accuracy at the boundaries between these row groups, MPI processes exchange "ghost cell" data after each update, which introduces communication overhead. The fixed boundary conditions of the entire grid are also handled in parallel.

It's important to realize that if the total number of rows isn't perfectly divisible by the number of MPI processes, some processes will handle slightly more rows than others. This uneven workload can lead to a minor slowdown, especially with many processes or small grids.

2.3 Transforming the Heat Matrix into a BMP File

The final step of converting the heat data into a BMP image is currently done without parallelization in the hybrid setup. While important for visualization, this part doesn't take much time compared to the main simulation, especially for large problems.

Parallelizing the image writing is avoided because the function used to write to the file isn't safe for simultaneous access by multiple threads. Also, having multiple MPI processes write to the same file would be complicated and could introduce performance issues due to communication and coordination.

Therefore, the main process gathers all the heat data and writes the BMP image in a single, sequential step. This keeps things simple, ensures the image is correct, and doesn't significantly impact the overall performance.

3 Parallelization

3.1 Function `initialize_local_grid`

The `initialize_local_grid` function assigns initial temperature values to the portion of the simulation matrix handled by each MPI process. In the hybrid approach, the global 2D matrix is divided into rows and distributed. Each process initializes its local part, including ghost rows for communication.

A nested loop iterates through the local grid, using `global_start` to map local indices to global row indices for correct initial conditions.

OpenMP's `#pragma omp parallel for` parallelizes the outer loop (vertical axis `i`) within each MPI process. Loop variables `i` and `j` are `private` to prevent race conditions.

The implicit `static` scheduling of OpenMP evenly divides iterations among threads at compile time. This is efficient due to the uniform and independent nature of each cell's initialization, ensuring good load balancing with minimal overhead.

3.2 Function `solve_heat_equation`

The `solve_heat_equation` function uses both MPI and OpenMP to efficiently solve the heat diffusion problem on a distributed grid.

With the grid divided along the first dimension among MPI processes, each process manages a local subgrid. To update the boundary rows of these subgrids correctly, neighboring processes exchange their adjacent internal rows (ghost cells) before each update. This is done using the `exchange_ghost_cells` function with `MPI_Sendrecv`.

After the ghost cell exchange, the `update_local_grid` function updates the internal points of the local grid. This is parallelized with OpenMP using `#pragma omp parallel for private(i, j) collapse(2)`, which combines the nested loops for even workload distribution across threads. The `private` clause ensures that loop indices are local to each thread. The default static scheduling is used due to the uniform workload per iteration.

Following the internal updates, the `apply_boundaries` function applies Dirichlet boundary conditions. The first MPI process sets its top row boundary to zero, the last process sets its bottom row to zero, and all processes set their first and last columns to zero. These boundary updates are also parallelized with OpenMP using separate loops for rows and columns.

In summary, the function combines MPI for inter-process communication of ghost cells with OpenMP for parallel computation of local grid points and boundary conditions, aiming for maximum parallel efficiency due to the data independence of updates and explicit communication.

4 Performance

After parallelizing the heat diffusion code, performance was tested. The original serial code served as a comparison. The parallel OpenMP version was run with 2 and 4 threads (8 threads were ineffective on the 4-core machine as we already saw and thus excluded).

Tests were performed on grid sizes of 100x100, 1000x1000, and 2000x2000, with each size run for 100, 1,000, 10,000, and 100,000 time steps.

For MPI parallelization, the code was run on 2, 4, 8, 16, and 32 cores to assess its ability to scale across the Moore cluster's 8 nodes (each with 4 cores).

4.1 Execution Time

First, **??** shows the execution time of the serial code for different matrix sizes. As expected, the serial execution time increases significantly with both the matrix size and the number of simulation steps.

The computational workload per step scales with the number of grid cells ($nx \times ny$), and the total workload scales linearly with the number of steps. For instance, increasing the number of steps by a factor of 10 results in roughly a 10x increase in execution time for all matrix sizes. Similarly, increasing the matrix size from 100×100 to 1000×1000 (100 times more cells) leads to an approximate 100x increase in execution time, while going from 1000×1000 to 2000×2000 (4 times more cells) yields a 4x increase.

This proportional scaling in both problem dimensions causes very large execution times for large matrices and high step counts, motivating the need for parallelization to reduce runtime.

Now, **??**, **??**, **??**, **??**, **??** and **??** show the execution times for the hybrid OpenMP + MPI implementation.

We can already make some observations:

- **Strong reduction vs Serial:** The hybrid approach significantly reduces execution time for large problems. For example, the 2000×2000 case with 100,000 steps drops from 4,314.34 seconds (serial) to 1,143.02 seconds (32 tasks, 2 threads), over a 3.7× speedup.
- **2 threads outperform 4 threads:** Configurations using 2 threads generally perform better than those with 4, due to the Moore cluster's 4-core nodes. More threads per node cause overhead from context switching and cache contention.
- **Scales well with larger problems:** As matrix size and step count increase, the hybrid implementation scales better. Execution time drops consistently from 2 to 32 MPI tasks for the 2000×2000 grid, especially with 2 threads.

- **Small problems perform worse:** For small grids like 100×100, hybrid parallelization may be slower than the serial version, as the overhead outweighs the computational benefit.
- **Communication and load imbalance:** With many MPI tasks, performance can stagnate or degrade due to communication overhead and minor load imbalance when the number of rows isn't divisible by the number of processes.
- **Scaling saturates with 4 threads:** With 4 threads, scaling worsens as MPI tasks increase. For instance, in the 1000×1000 grid with 100,000 steps, runtime goes from 1,424.65 s (2 tasks) to over 3,000 s (32 tasks), mainly due to oversubscription and communication costs.
- **Computation dominates for large problems:** For large matrices and high step counts, communication overhead becomes less significant, enabling better scalability and resource utilization.

4.2 Speedup and Scalability

To quantify the benefits of parallelization, we calculate the **speedup**, defined as:

$$\text{Speedup}(p) = \frac{T_{\text{serial}}}{T_p},$$

where T_{serial} is the execution time of the serial implementation and T_p is the execution time of the parallel version using p processing elements (MPI tasks and/or OpenMP threads).

The calculated speedup values, with 2 and 4 threads respectively, are presented in Figure 1 and Figure 4 (for 100x100), Figure 2 and Figure 5 (for 1000x1000), and Figure 3 and Figure 6 (for 2000x2000).

The related charts are Figure 1 and Figure 4 (for 100x100), Figure 2 and Figure 5 (for 1000x1000), and Figure 3 and Figure 6 (for 2000x2000).

The analysis of the speedup results combining OpenMP threads and MPI tasks across different matrix sizes and step counts reveals several important patterns:

- **Impact of matrix size:** Speedup improves significantly with increasing matrix size. For the smallest matrix (100×100), speedups are mostly below or around 1, indicating that the overhead of parallelization outweighs the computation time. For intermediate (1000×1000) and large (2000×2000) matrices, speedups increase substantially, reflecting that the larger workload better amortizes the parallel overhead.
- **Effect of step count:** Generally, increasing the number of steps tends to improve speedup, especially for larger matrices. This occurs because more computational work per

run increases the ratio of useful work to communication and synchronization overhead. However, for some configurations with very large step counts, speedups tend to sfigilize or slightly decrease, possibly due to memory hierarchy effects or resource contention over extended execution.

- **OpenMP Threads (2 vs 4):** Using 2 OpenMP threads combined with multiple MPI tasks consistently outperforms configurations with 4 OpenMP threads. In fact, the 4-thread OpenMP configurations often exhibit poor speedup, sometimes close to zero, likely due to increased thread management overhead and contention, which outweighs the parallel gain at this granularity.

- **Number of MPI Tasks:** Increasing the number of MPI tasks generally improves speedup for larger matrices. For the 2000×2000 matrix, speedup continues to increase up to 32 tasks when using 2 OpenMP threads, reaching values above 3.5–4. For smaller matrices, increasing MPI tasks does not help and may reduce speedup because of overhead dominating computation time.

- **Combined parallelization strategy:** The best speedup results are achieved by balancing a modest number of OpenMP threads (2) with a higher number of MPI tasks, especially for large matrices and higher step counts. This indicates that fine-grained threading within nodes paired with MPI parallelism across nodes is more effective than heavily multithreaded configurations alone.

- **Performance degradation for small matrices:** For the smallest matrix size (100×100), speedups often fall below 1, meaning parallel execution is slower than serial. This highlights the cost of parallel overhead such as thread creation, synchronization, and MPI communication that cannot be compensated by the limited computational workload.

In summary, to achieve meaningful speedup using hybrid OpenMP + MPI parallelism, it is crucial to apply the method to sufficiently large problem sizes and carefully tune the number of threads and MPI tasks to avoid excessive overhead and contention.

4.3 Efficiency

Parallel efficiency, $E_p = S_p/p$, measures how effectively processing resources are utilized compared to the ideal linear speedup ($E_p = 1$).

The calculated efficiency values, with 2 and 4 threads respectively, are presented in Table 14 and Table 17 (for 100x100), Table 15 and Table 18 (for 1000x1000), and Table 16 and Table 19 (for 2000x2000).

The related charts are Figure 7 and Figure 10 (for 100x100), Figure 8 and Figure 11 (for 1000x1000), and Figure 9 and Figure 12 (for 2000x2000).

Analyzing the efficiency of the hybrid OpenMP + MPI parallel configurations across different matrix sizes, step counts, threads, and tasks, we observe the following:

- **Efficiency generally decreases with increasing number of MPI Tasks:** For all matrix sizes, efficiency tends to be highest with fewer MPI tasks and decreases sharply as the number of tasks increases. This reflects the increased overhead of communication, synchronization, and load imbalance when more tasks are involved, which dilutes the useful computational work per task.

- **Matrix size has a positive impact on efficiency:** Larger matrices show better efficiency values compared to smaller ones. For the 2000×2000 matrix, efficiency reaches up to around 0.29 with 2 MPI tasks and 2 OpenMP threads, while for 100×100, efficiency is mostly below 0.15. This indicates that bigger problem sizes better amortize the overhead costs inherent in parallel execution.

- **Effect of step count on efficiency:** Efficiency generally decreases as the number of steps increases, especially with more MPI tasks. While more steps increase workload, they also increase runtime and the chance for overheads (such as synchronization delays or memory contention) to accumulate, slightly reducing efficiency.

- **OpenMP thread count influence:** Efficiency with 4 OpenMP threads is consistently poor across all matrix sizes and step counts, often near zero. This implies that increasing the number of OpenMP threads beyond 2 introduces significant overhead or contention that cannot be compensated by parallel gains at the tested granularity and workload.

- **Optimal balance between MPI Tasks and Threads:** The best efficiency values are achieved using 2 OpenMP threads combined with a low number of MPI tasks (2 or 4). This balance minimizes overhead and contention while still exploiting parallelism effectively, especially on larger matrices.

- **Small matrices suffer low efficiency:** Small problem sizes show very low efficiency regardless of parallel configuration, which means that the overhead of parallelism dominates the computation, resulting in ineffective parallel resource use.

In conclusion, efficiency analysis confirms that parallel overhead limits scalability, especially for small problems or excessive numbers of MPI tasks. Careful tuning of threads and tasks according to problem size is essential to achieve balanced and efficient parallel performance.

5 Conclusions

The performance evaluation of the hybrid OpenMP + MPI implementation reveals several important insights, although the overall results did not meet our initial expectations for parallel scalability and efficiency.

Firstly, the speedup achieved is strongly dependent on the problem size and the configuration of threads and tasks. Larger matrices (2000×2000) show moderate speedup, especially with 2 OpenMP threads and a moderate number of MPI tasks (4 to 32), reaching speedups up to around 4. However, smaller matrices (100×100 and 1000×1000) generally exhibit poor speedup or even slowdown, indicating that the overhead of parallelism outweighs the computational benefits for these cases.

Secondly, efficiency values were generally low across all configurations, with the highest efficiencies seen only for the largest matrix size combined with a low number of MPI tasks and 2 OpenMP threads. Increasing the number of OpenMP threads to 4 did not improve efficiency. In fact, it resulted in near-zero efficiency in most cases, likely due to increased overhead, thread contention, or insufficient workload per thread.

Furthermore, increasing the number of MPI tasks beyond a certain point consistently decreased both speedup and efficiency, reflecting communication and synchronization overheads that limit scalability. Also, increasing the number of simulation steps tends to reduce efficiency, possibly due to cumulative parallel overheads and increased synchronization demands.

In summary, the performance results highlight that the current parallel implementation struggles to fully exploit available parallelism, especially for smaller problem sizes or more aggressive thread/task configurations. This suggests the need for further optimization efforts, such as improving workload distribution, reducing communication overhead, or adapting parallel granularity, to achieve better scalability and resource utilization.

Overall, while the hybrid approach demonstrates potential on larger problems, the limited speedup and efficiency indicate significant challenges remain to achieve the desired performance gains.

Appendix

A Functions

A.1 Function `initialize_local_grid`

```
void initialize_local_grid(double *local_grid, int local_nx, int ny, int
    global_start) {
    int i, j;

    #pragma omp parallel for private(i, j)
    for (i = 1; i < local_nx - 1; i++) {
        int global_i = global_start + i - 1;
        for (j = 0; j < ny; j++) {
            if (global_i == j || global_i == ny - 1 - j) local_grid[i * ny + j]
                = T;
            else local_grid[i * ny + j] = 0.0;
        }
    }
}
```

Code 1: Function `initialize_local_grid` parallelized with OpenMP and MPI

A.2 Function `solve_heat_equation`

```
void solve_heat_equation(double *local_grid, double *new_local_grid, int steps,
    double r, int local_nx, int global_ny, int rank, int size) {
    for (int step = 0; step < steps; step++) {
        exchange_ghost_cells(local_grid, local_nx, global_ny, rank, size);
        update_local_grid(local_grid, new_local_grid, local_nx, global_ny, r);
        apply_boundaries(new_local_grid, local_nx, global_ny, rank, size);

        double *tmp = local_grid;
        local_grid = new_local_grid;
        new_local_grid = tmp;
    }
}
```

Code 2: Function `solve_heat_equation` parallelized with OpenMP and MPI

A.3 Function `exchange_ghost_cells`

```
void exchange_ghost_cells(double *local_grid, int local_nx, int global_ny, int
    rank, int size) {
    if (rank > 0) {
        MPI_Sendrecv(&local_grid[1 * global_ny], global_ny, MPI_DOUBLE, rank -
            1, 0,
                     &local_grid[0 * global_ny], global_ny, MPI_DOUBLE, rank -
                         1, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    if (rank < size - 1) {
        MPI_Sendrecv(&local_grid[(local_nx - 2) * global_ny], global_ny,
            MPI_DOUBLE, rank + 1, 0,
                     &local_grid[(local_nx - 1) * global_ny], global_ny,
                         MPI_DOUBLE, rank + 1, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

Code 3: Function `exchange_ghost_cells` parallelized with OpenMP and MPI

A.4 Function `update_local_grid`

```
void update_local_grid(double *local_grid, double *new_local_grid, int nx, int
    ny, double r) {
    int i, j;
    #pragma omp parallel for private(i, j) collapse(2)
    for (i = 1; i < nx - 1; i++) {
        for (j = 1; j < ny - 1; j++) {
            new_local_grid[i * ny + j] = local_grid[i * ny + j]
                + r * (local_grid[(i + 1) * ny + j] + local_grid[(i - 1) * ny +
                    j] - 2 * local_grid[i * ny + j])
                + r * (local_grid[i * ny + j + 1] + local_grid[i * ny + j - 1] -
                     2 * local_grid[i * ny + j]);
        }
    }
}
```

Code 4: Function `update_local_grid` parallelized with OpenMP and MPI

A.5 Function `apply_boundaries`

```
void apply_boundaries(double *local_grid, int local_nx, int ny, int rank, int
    size) {
    int i, j;

    if (rank == 0) {
        // First process sets the first row to 0
        #pragma omp parallel for private(j)
        for (j = 0; j < ny; j++) {
            local_grid[1 * ny + j] = 0.0;
        }
    } else if (rank == size - 1) {
        // Last process sets the last row to 0
        #pragma omp parallel for private(j)
        for (j = 0; j < ny; j++) {
            local_grid[(local_nx - 2) * ny + j] = 0.0;
        }
    }

    // Lateral boundaries are set to 0
    #pragma omp parallel for private(i)
    for (i = 0; i < local_nx; i++) {
        local_grid[i * ny + 0] = 0.0;
        local_grid[i * ny + (ny - 1)] = 0.0;
    }
}
```

Code 5: Function `apply_boundaries` parallelized with OpenMP and MPI

B Tables

B.1 Execution Times

B.1.1 Serial Execution Time

	Steps			
Matrix Size	**100**	**1,000**	**10,000**	**100,000**
100 x 100	0.01	0.11	1.02	10.20
1000 x 1000	1.17	12.45	110.92	1,073.50
2000 x 2000	4.66	49.19	476.57	4,314.34

Table 1: Serial Execution Time (in seconds)

B.1.2 Execution Time (2 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

	MPI Tasks (2 OpenMP Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.02	0.02	0.27	0.36	1.01
1,000	0.14	0.13	9.65	3.37	9.61
10,000	0.69	1.21	96.06	96.05	95.81
100,000	16.83	10.74	960.25	962.09	957.83

Table 2: Execution Time with OpenMP (2 threads) + MPI Tasks for 100×100 matrix

B.1.3 Execution Time (2 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

	MPI Tasks (2 OpenMP Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	1.98	0.39	0.90	1.41	1.79
1,000	18.26	11.55	8.79	10.93	10.90
10,000	174.56	91.25	105.04	105.36	101.20
100,000	1,482.58	1,348.17	1,183.07	1,050.14	1,005.52

Table 3: Execution Time with OpenMP (2 threads) + MPI Tasks for 1000×1000 matrix

B.1.4 Execution Time (2 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

	MPI Tasks (2 OpenMP Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	4.60	2.56	2.35	2.06	2.47
1,000	42.05	21.76	17.90	14.22	12.82
10,000	415.33	233.92	176.83	138.48	117.71
100,000	4,052.52	2,555.92	1,826.81	1,325.09	1,143.02

Table 4: Execution Time with OpenMP (2 threads) + MPI Tasks for 2000×2000 matrix

B.1.5 Execution Time (4 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

	MPI Tasks (4 OpenMP Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	1.10	2.66	3.32	3.08	2.92
1,000	11.40	26.50	30.02	29.66	30.10
10,000	272.41	271.16	288.08	304.20	301.02
100,000	1,171.69	2,930.66	2,964.58	3,040.74	3,021.46

Table 5: Execution Time with OpenMP (4 threads) + MPI Tasks for 100×100 matrix

B.1.6 Execution Time (4 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

	MPI Tasks (4 OpenMP Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	2.75	3.42	3.33	4.14	3.81
1,000	36.17	34.08	31.35	32.43	31.98
10,000	355.71	302.14	311.25	315.69	312.61
100,000	1,424.65	2,934.94	2,882.44	3,164.78	3,092.54

Table 6: Execution Time with OpenMP (4 threads) + MPI Tasks for 1000×1000 matrix

B.1.7 Execution Time (4 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

	MPI Tasks (4 OpenMP Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	6.32	5.08	4.27	4.87	4.49
1,000	55.15	43.87	37.53	35.92	34.08
10,000	410.44	432.88	372.09	346.53	326.51
100,000	3,090.22	3,608.40	3,041.00	3,415.37	3,235.95

Table 7: Execution Time with OpenMP (4 threads) + MPI Tasks for 2000×2000 matrix

B.2 Speedup

B.2.1 Speedup (2 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

	MPI Tasks (OpenMP 2 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.50	0.51	0.04	0.03	0.01
1,000	0.79	0.82	0.01	0.03	0.01
10,000	1.49	0.84	0.01	0.01	0.01
100,000	0.61	0.95	0.01	0.01	0.01

Table 8: Speedup for OpenMP (2 threads) + MPI Tasks on 100×100 matrix

B.2.2 Speedup (2 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

	MPI Tasks (OpenMP 2 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.59	2.98	1.31	0.83	0.65
1,000	0.68	1.08	1.42	1.14	1.14
10,000	0.64	1.22	1.06	1.05	1.10
100,000	0.72	0.80	0.91	1.02	1.07

Table 9: Speedup for OpenMP (2 threads) + MPI Tasks on 1000×1000 matrix

B.2.3 Speedup (2 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

	MPI Tasks (OpenMP 2 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	1.01	1.82	1.98	2.27	1.89
1,000	1.17	2.26	2.75	3.46	3.84
10,000	1.15	2.04	2.70	3.44	4.05
100,000	1.06	1.69	2.36	3.26	3.77

Table 10: Speedup for OpenMP (2 threads) + MPI Tasks on 2000×2000 matrix

B.2.4 Speedup (4 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

	MPI Tasks (OpenMP 4 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.01	0.00	0.00	0.00	0.00
1,000	0.01	0.00	0.00	0.00	0.00
10,000	0.00	0.00	0.00	0.00	0.00
100,000	0.01	0.00	0.00	0.00	0.00

Table 11: Speedup for OpenMP (4 threads) + MPI Tasks on 100×100 matrix

B.2.5 Speedup (4 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

	MPI Tasks (OpenMP 4 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.42	0.34	0.35	0.28	0.31
1,000	0.34	0.37	0.40	0.38	0.39
10,000	0.31	0.37	0.36	0.35	0.35
100,000	0.75	0.37	0.37	0.34	0.35

Table 12: Speedup for OpenMP (4 threads) + MPI Tasks on 1000×1000 matrix

B.2.6 Speedup (4 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

	MPI Tasks (OpenMP 4 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.74	0.92	1.09	0.96	1.04
1,000	0.89	1.12	1.31	1.37	1.44
10,000	1.16	1.10	1.28	1.38	1.46
100,000	1.40	1.20	1.42	1.26	1.33

Table 13: Speedup for OpenMP (4 threads) + MPI Tasks on 2000×2000 matrix

B.3 Efficiency

B.3.1 Efficiency (2 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

	MPI Tasks (OpenMP 2 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.13	0.06	0.00	0.00	0.00
1,000	0.20	0.10	0.00	0.00	0.00
10,000	0.37	0.10	0.00	0.00	0.00
100,000	0.15	0.12	0.00	0.00	0.00

Table 14: Efficiency for OpenMP (2 threads) + MPI Tasks on 100×100 matrix

B.3.2 Efficiency (2 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

	MPI Tasks (OpenMP 2 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.15	0.37	0.08	0.03	0.01
1,000	0.17	0.13	0.09	0.04	0.02
10,000	0.16	0.15	0.07	0.03	0.02
100,000	0.18	0.10	0.06	0.03	0.02

Table 15: Efficiency for OpenMP (2 threads) + MPI Tasks on 1000×1000 matrix

B.3.3 Efficiency (2 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

	MPI Tasks (OpenMP 2 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.25	0.23	0.12	0.07	0.03
1,000	0.29	0.28	0.17	0.11	0.06
10,000	0.29	0.25	0.17	0.11	0.06
100,000	0.27	0.21	0.15	0.10	0.06

Table 16: Efficiency for OpenMP (2 threads) + MPI Tasks on 2000×2000 matrix

B.3.4 Efficiency (4 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

	MPI Tasks (OpenMP 4 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.00	0.00	0.00	0.00	0.00
1,000	0.00	0.00	0.00	0.00	0.00
10,000	0.00	0.00	0.00	0.00	0.00
100,000	0.00	0.00	0.00	0.00	0.00

Table 17: Efficiency for OpenMP (4 threads) + MPI Tasks on 100×100 matrix

B.3.5 Efficiency (4 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

	MPI Tasks (OpenMP 4 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.05	0.02	0.01	0.00	0.00
1,000	0.04	0.02	0.01	0.01	0.00
10,000	0.04	0.02	0.01	0.01	0.00
100,000	0.09	0.02	0.01	0.01	0.00

Table 18: Efficiency for OpenMP (4 threads) + MPI Tasks on 1000×1000 matrix

B.3.6 Efficiency (4 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

	MPI Tasks (OpenMP 4 Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	0.09	0.06	0.03	0.01	0.01
1,000	0.11	0.07	0.04	0.02	0.01
10,000	0.15	0.07	0.04	0.02	0.01
100,000	0.17	0.07	0.04	0.02	0.01

Table 19: Efficiency for OpenMP (4 threads) + MPI Tasks on 2000×2000 matrix

B.4 Charts

B.4.1 Speedup (2 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

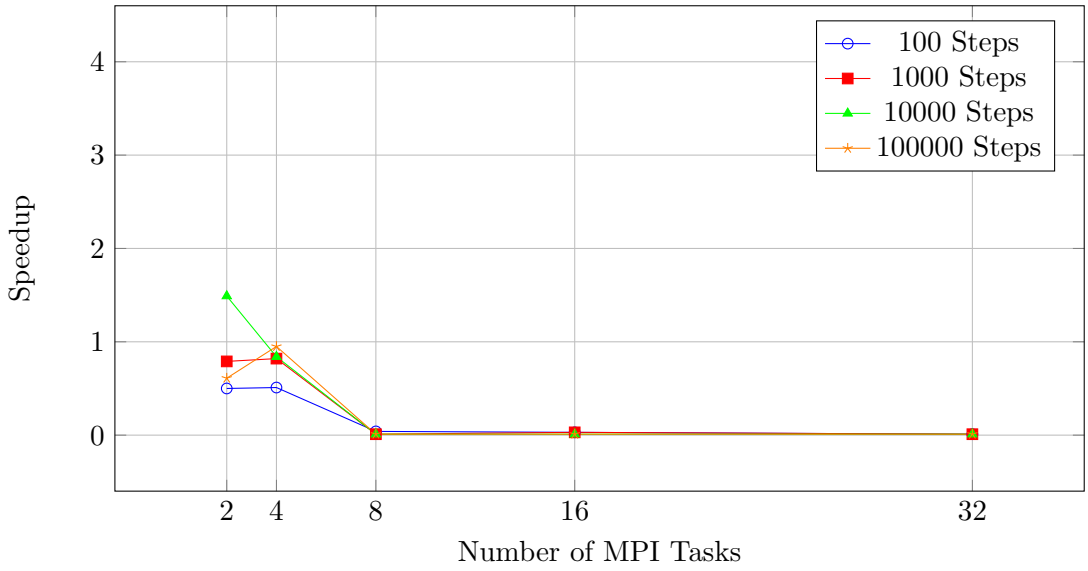

Figure 1: Speedup for OpenMP (2 Threads) + MPI Tasks on 100×100 matrix

B.4.2 Speedup (2 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

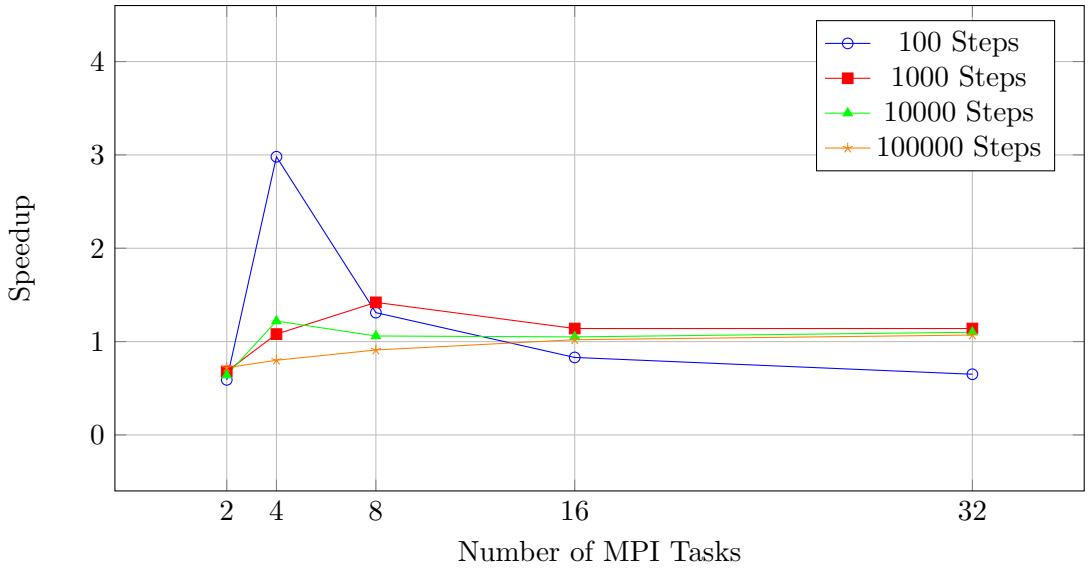

Figure 2: Speedup for OpenMP (2 threads) + MPI Tasks on 1000×1000 matrix

B.4.3 Speedup (2 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

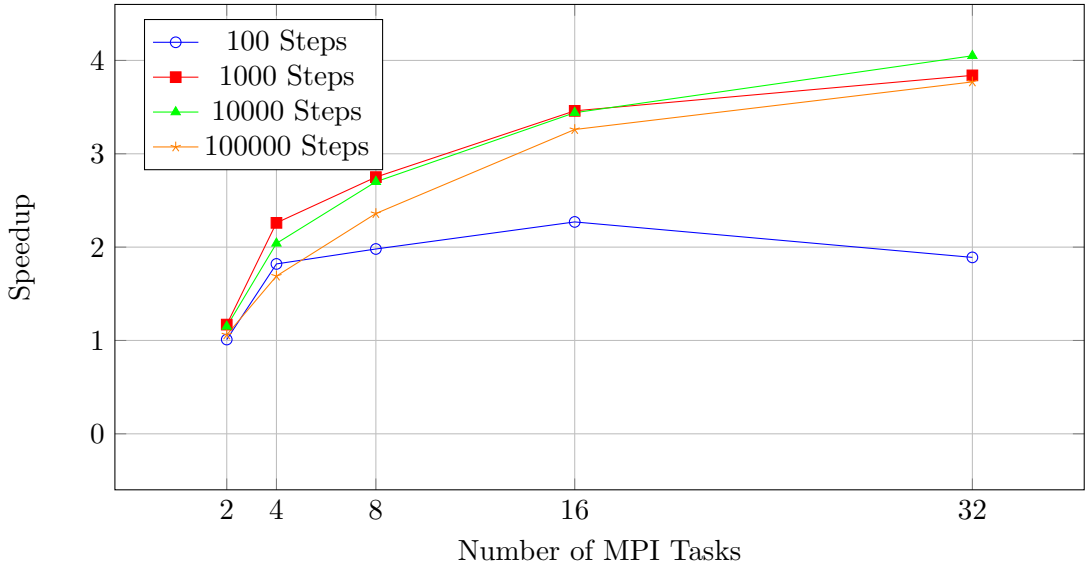

Figure 3: Speedup for OpenMP (2 threads) + MPI Tasks on 2000×2000 matrix

B.4.4 Speedup (4 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

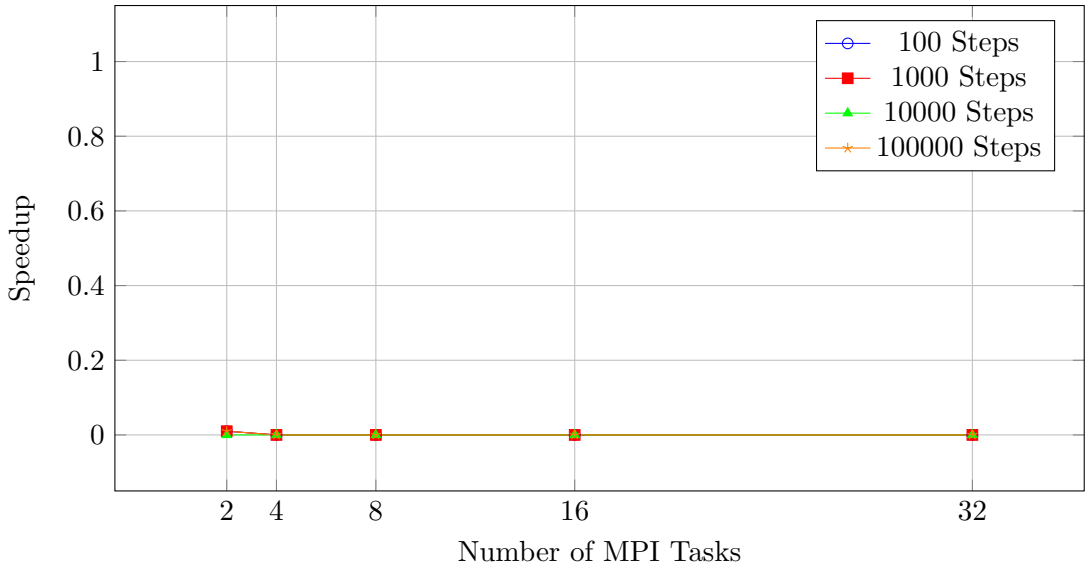

Figure 4: Speedup for OpenMP (4 threads) + MPI Tasks on 100×100 matrix

B.4.5 Speedup (4 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

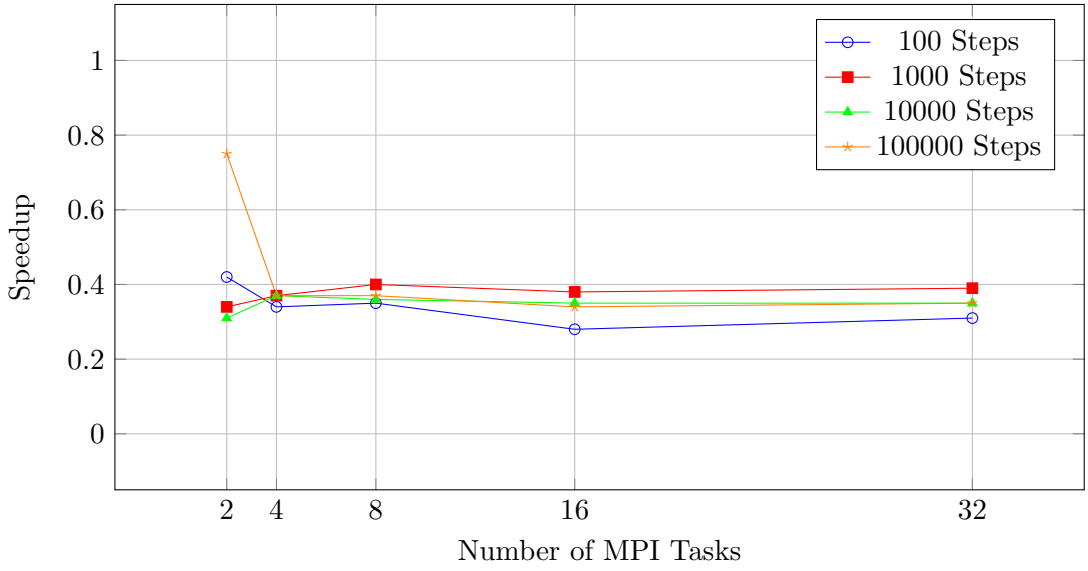

Figure 5: Speedup for OpenMP (4 threads) + MPI Tasks on 1000×1000 matrix

B.4.6 Speedup (4 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

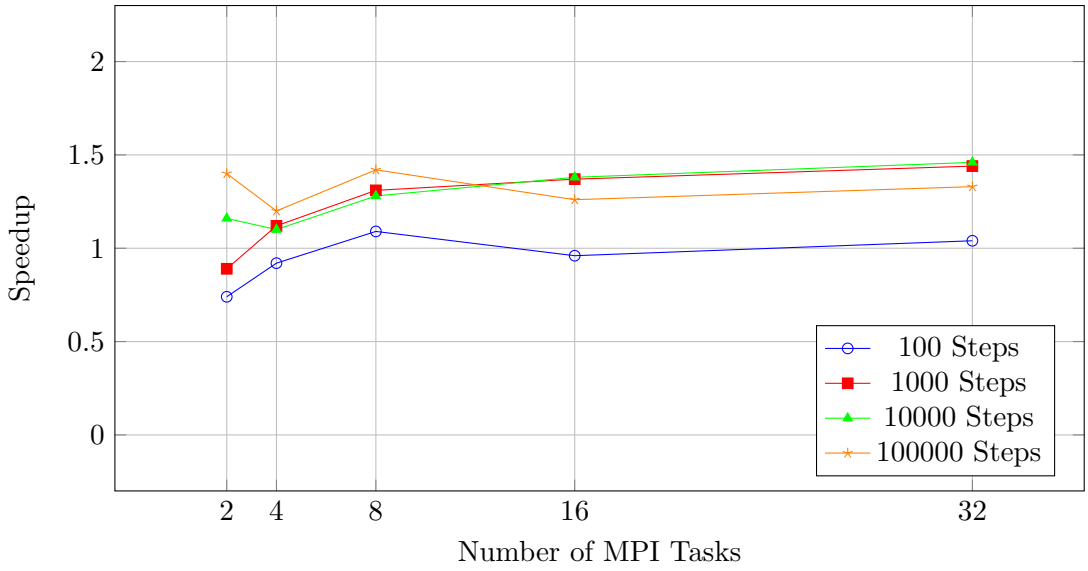

Figure 6: Speedup for OpenMP (4 threads) + MPI Tasks on 2000×2000 matrix

B.4.7 Efficiency (2 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

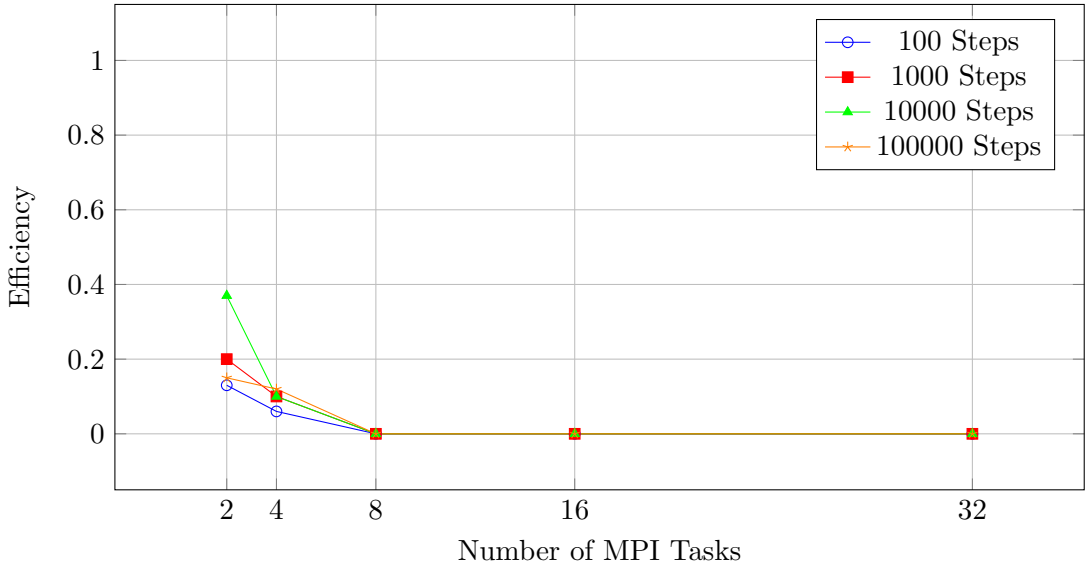

Figure 7: Efficiency for OpenMP (2 threads) + MPI Tasks on 100×100 matrix

B.4.8 Efficiency (2 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

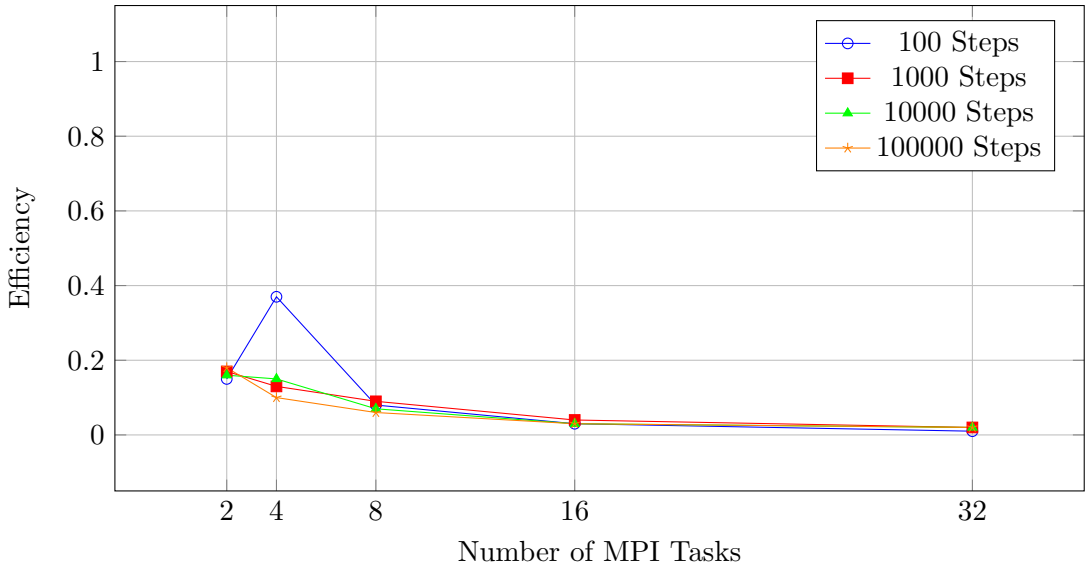

Figure 8: Efficiency for OpenMP (2 threads) + MPI Tasks on 1000×1000 matrix

B.4.9 Efficiency (2 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

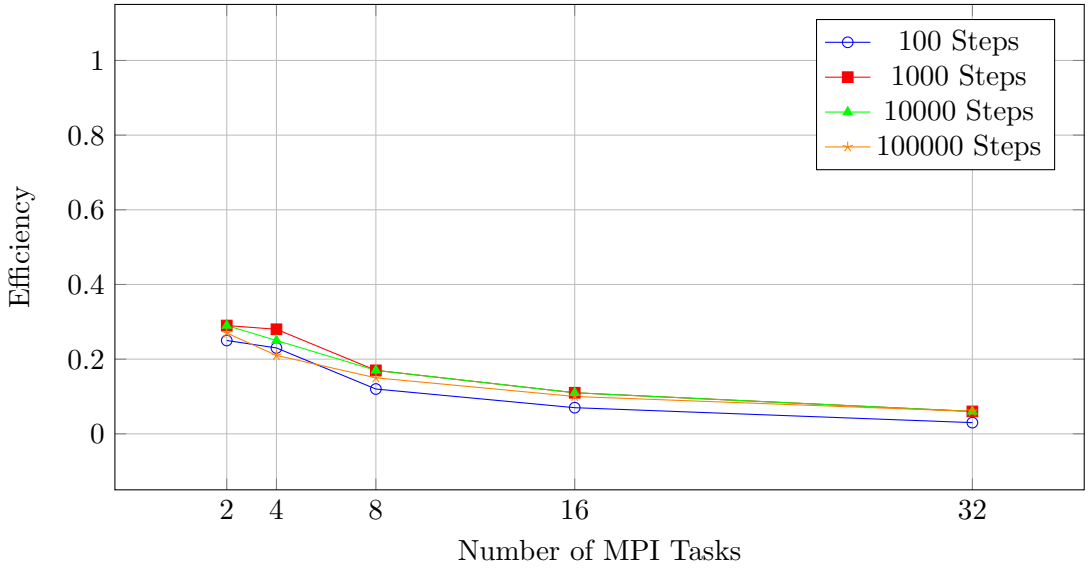

Figure 9: Efficiency for OpenMP (2 threads) + MPI Tasks on 2000×2000 matrix

B.4.10 Efficiency (4 OpenMP Threads + MPI Tasks) in a 100x100 Matrix

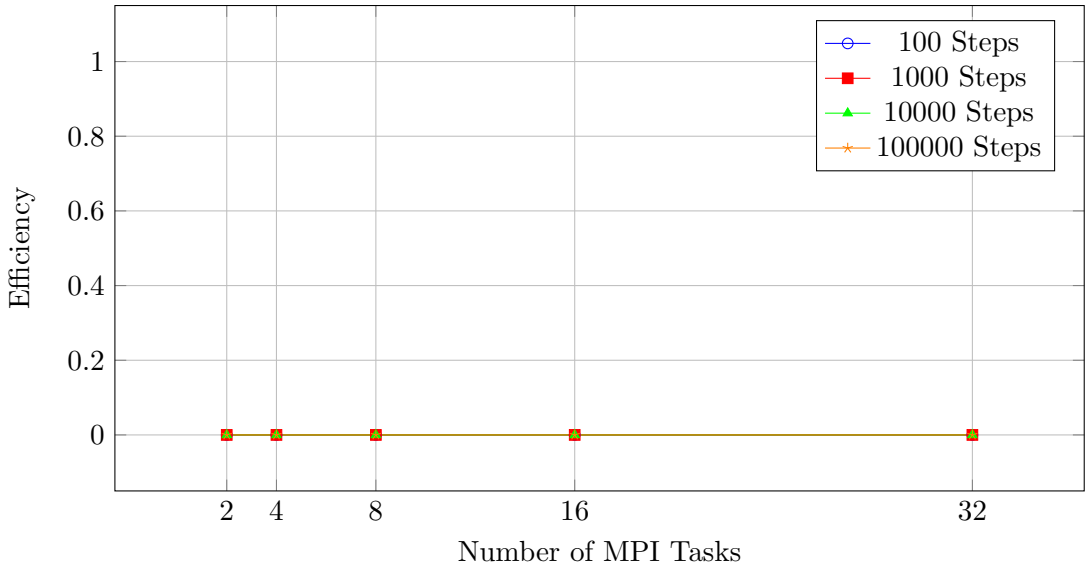

Figure 10: Efficiency for OpenMP (4 threads) + MPI Tasks on 100×100 matrix

B.4.11 Efficiency (4 OpenMP Threads + MPI Tasks) in a 1000x1000 Matrix

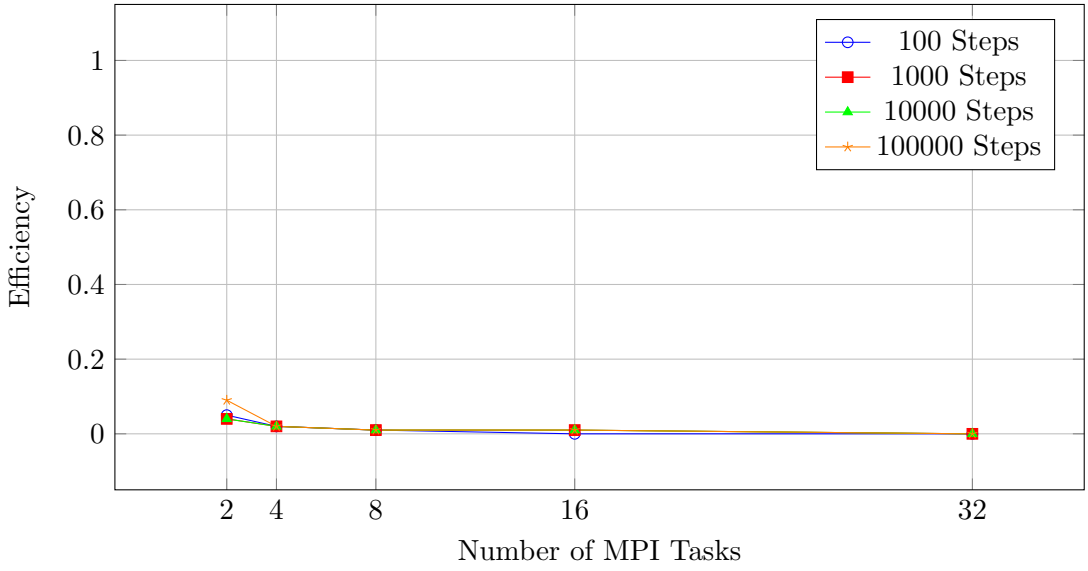

Figure 11: Efficiency for OpenMP (4 threads) + MPI Tasks on 1000×1000 matrix

B.4.12 Efficiency (4 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

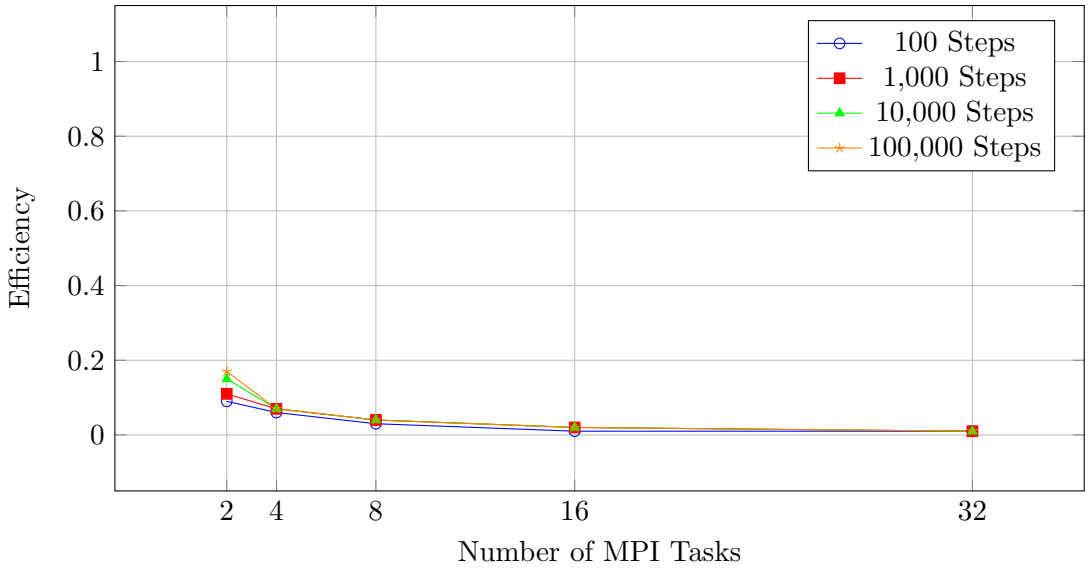

Figure 12: Efficiency for OpenMP (4 threads) + MPI Tasks on 2000×2000 matrix