Ministry of higher education
and scientific research

University of El Manar

Faculty of Sciences of Tunis

وزارة التعليم العالي و البحث العلمي

جامعة تونس المنار

كلية العلوم بتونس

Parallel Computing Project Report

OpenMP Matrix Parallelization

Work developed by:

AROUA Rahma

BELHADJ Ghada

Supervised by:

Ms. SLAMA Yosr

Academic year : 2024/2025

Contents

1 Objectives of the Experimentation

The aim of this project is to implement and thoroughly analyze matrix multiplication using C and OpenMP while exploring the impact of parallelization on performance under various conditions. Specifically, the experimentation focuses on studying how different factors influence the efficiency of the computation. These factors include the size of the matrices, the number of processor cores utilized, and the loop parallelization strategies applied, such as static, dynamic, and block-based distribution. By leveraging OpenMP's predefined directives and primitives, we aim to evaluate the effectiveness of these approaches in optimizing computational performance. Furthermore, the study involves a comprehensive comparison of the results across multiple configurations through metrics such as makespan and speedup, offering valuable insights into the benefits and limitations of each strategy. This experimentation serves as a critical step in understanding how parallel computing frameworks can enhance the performance of computationally intensive tasks.

2 Matrix Generation and Multiplication

2.1 Matrix Allocation and Generation

```c
float** allocate_matrix(int size) {
    float** matrix = (float**)malloc(size * sizeof(float*));
    for (int i = 0; i < size; i++) {
        matrix[i] = (float*)malloc(size * sizeof(float));
    }
    return matrix;
}

void free_matrix(float** matrix, int size) {
    for (int i = 0; i < size; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

void generate_matrix(float** matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            matrix[i][j] = (float)(rand() % 100);
        }
    }
}
```

Listing 1: Matrix Allocation and Generation code

2.2 Sequential Matrix Multiplication

```
void matrix_multiply_sequential(float** A, float** B, float** C, int
    size) {
     for (int i = 0; i < size; i++) {
         for (int j = 0; j < size; j++) {
             C[i][j] = 0;
             for (int k = 0; k < size; k++) {
                 C[i][j] += A[i][k] * B[k][j];
             }
         }
     }
}
```

Listing 2: Sequential matrix Multiplication

3 Parallelization with OpenMP

3.1 Parallel Versions

We implemented three parallel versions using the #**pragma omp for** directives:

- Parallelization of the i loop.
- Parallelization of the j loop.
- Simultaneous parallelization of both i and j loops.

3.2 Parallelization Strategies

We tested these versions with different OpenMP strategies :

- **Static :** Fixed distribution of iterations among threads.
- **Dynamic :** Distribution as threads become available.
- **Bloc-Based :** Predefined blocks of iterations.

3.3 Experimental Configurations

- Tests were conducted with matrices of different sizes.
- The number of cores was varied from 1 to 8.

4 Performance Analysis

4.1 Matrix size: 512x512

Number of Threads	Strategy	Time (seconds)	Speedup
1	Static	0.541468	1.481358
1	Dynamic	0.539889	1.485692
1	Block-based	0.537133	1.493314
2	Static	0.545978	1.469124
2	Dynamic	0.532022	1.507662
2	Block-based	0.541520	1.481218
4	Static	0.592708	1.353294
4	Dynamic	0.560750	1.430421
4	Block-based	0.547804	1.464226
8	Static	0.517465	1.550072
8	Dynamic	0.552705	1.451241
8	Block-based	0.578601	1.386290

Table 1: Performance Comparison of Threading Strategies with Execution Time and Speedup (Size: N =512)

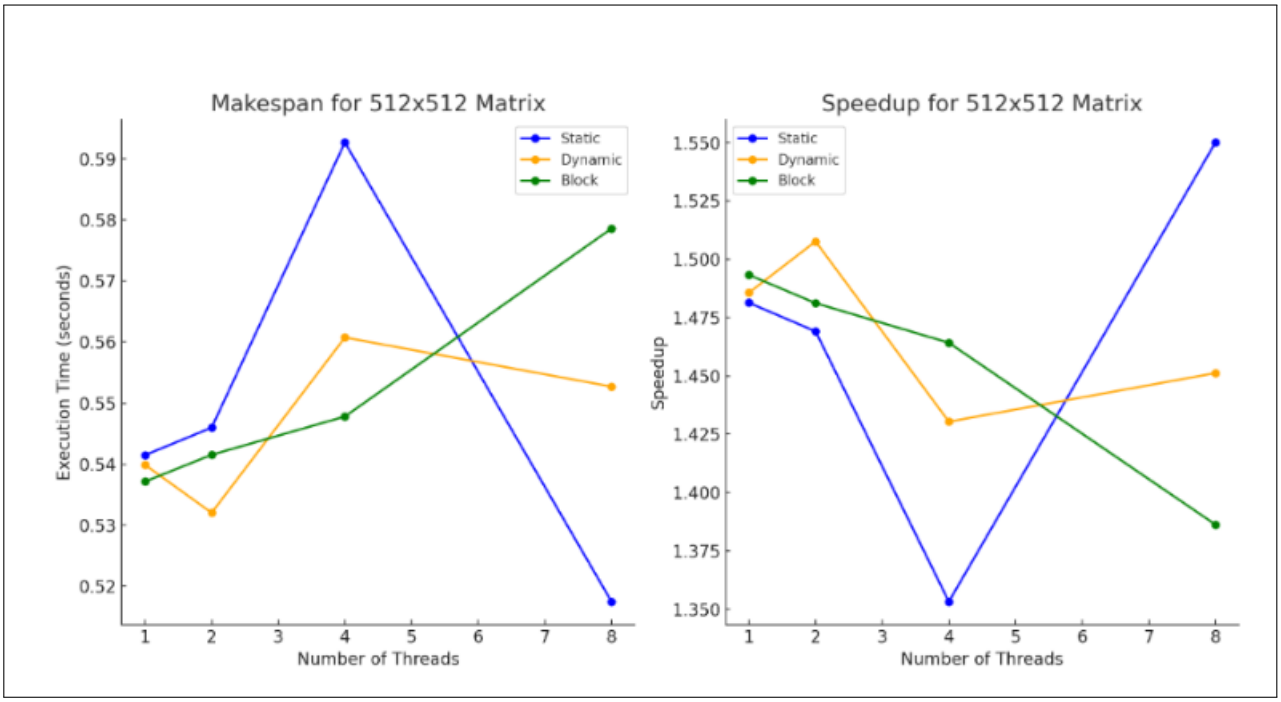

Figure 1: The curves for 512x512 matrix

Analysis of Results

Makespan Curve

- Execution time generally decreases as the number of threads increases, but improvements are limited beyond 4 threads.
- The block-based strategy is slightly faster than others for multi-threaded configurations.
- The dynamic strategy is comparable to block-based but may show a slight increase in time at 8 threads.

Speedup Curve

- Performance gains are optimal from 1 to 4 threads, but speedup plateaus or slightly decreases beyond that.
- The static strategy offers better speedup at 8 threads, while dynamic and block-based show smaller gains at this level.

4.2 Matrix size: 1024x1024

Number of Threads	Strategy	Time (seconds)	Speedup
1	Static	8.244665	1.023292
1	Dynamic	7.883330	1.070195
1	Block-based	8.021963	1.051701
2	Static	7.888849	1.069447
2	Dynamic	7.763590	1.086701
2	Block-based	7.657876	1.101703
4	Static	7.629343	1.105823
4	Dynamic	7.750442	1.088545
4	Block-based	7.895242	1.068581
8	Static	7.690167	1.097077
8	Dynamic	11.302524	0.746444
8	Block-based	18.474206	0.456675

Table 2: Performance Comparison of Threading Strategies with Execution Time and Speedup (Size: N =1024).

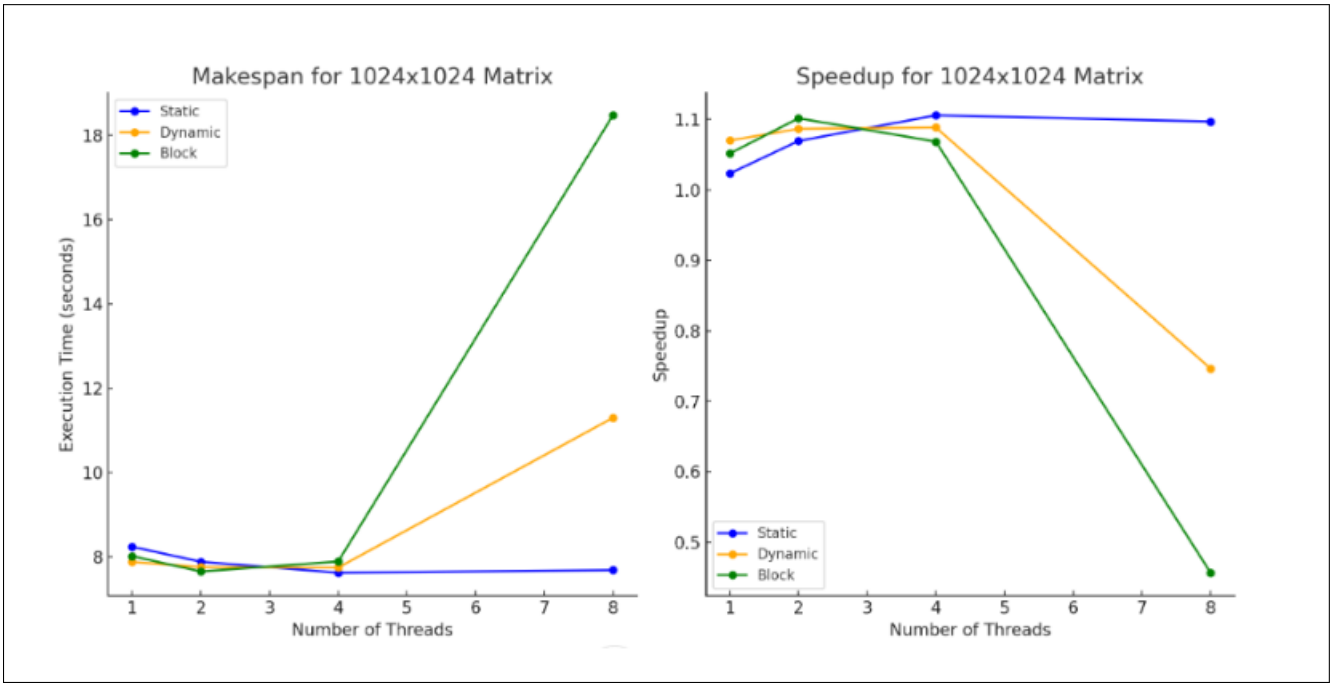

Figure 2: The curves for 1024x1024 matrix

Analysis of Results

Makespan Curve

- The dynamic and block-based strategies are faster for 2 to 4 threads, but static becomes less efficient beyond 4 threads.
- Beyond 4 threads, execution times plateau or slightly increase.

Speedup Curve

- Speedup is more linear compared to the 512x512 matrix up to 4 threads, but a plateau is observed beyond that.
- Dynamic and block-based strategies continue to show significant gains with 4 threads, unlike static, which stabilizes earlier.

4.3 Matrix size: 2048x2048

Number of Threads	Strategy	Time (seconds)	Speedup
1	Static	0.541468	1.481358
1	Dynamic	0.539889	1.485692
1	Bloc-Based	0.537133	1.493314
2	Static	0.545978	1.469124
2	Dynamic	0.532022	1.507662
2	Bloc-Based	0.541520	1.481218
4	Static	0.592708	1.353294
4	Dynamic	0.560750	1.430421
4	Bloc-Based	0.547804	1.464226
8	Static	0.517465	1.550072
8	Dynamic	0.552705	1.451241
8	Bloc-Based	0.578601	1.386290

Table 3: Performance Comparison of Threading Strategies with Execution Time and Speedup (Size: N =2048)

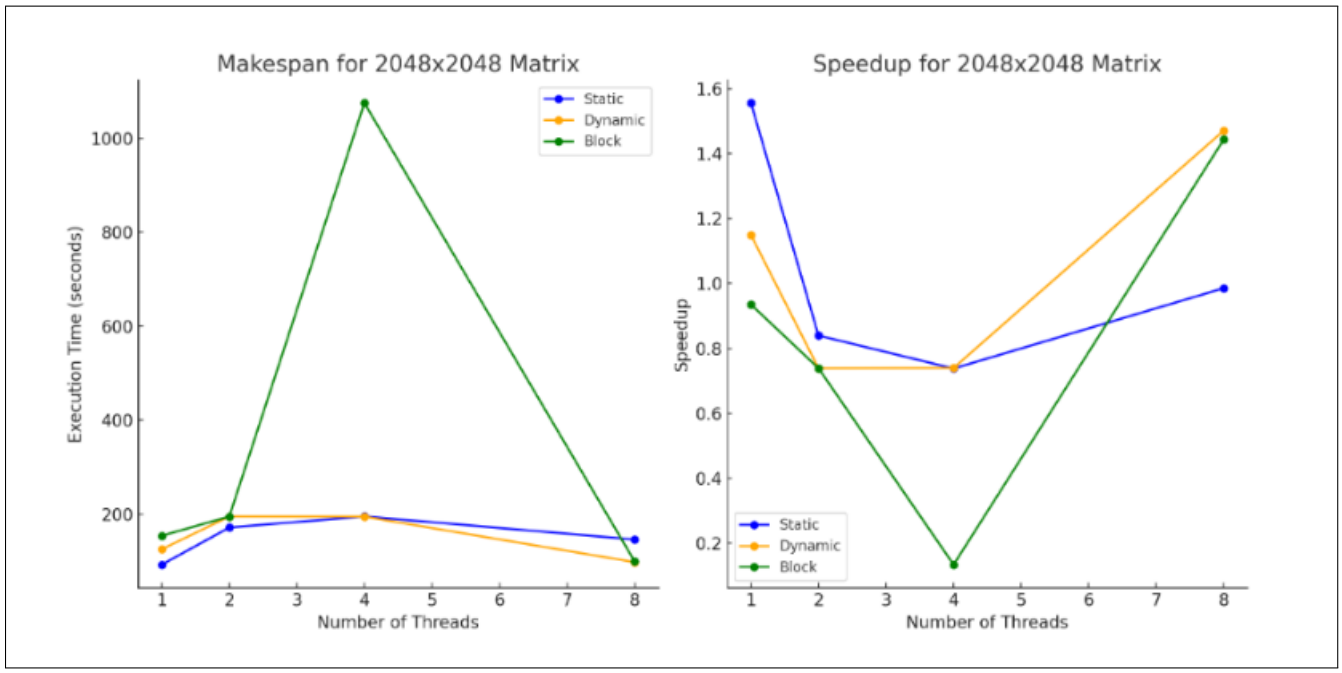

Figure 3: The curves for 2048x2048 matrix

Analysis of Results

Makespan Curve

- The static strategy is significantly faster up to 4 threads.
- Dynamic surpasses static at 8 threads, while block-based becomes inefficient beyond 2 threads.
- There is a marked degradation in the performance of the block-based strategy with a large number of threads.

Speedup Curve

- Static shows good gains up to 4 threads, but its speedup decreases afterward.
- Dynamic provides the best gains at 8 threads, while block-based shows negative impacts beyond 2 threads, indicating parallelization overhead.

5 Comparison of Matrix Size Impact

5.1 Effect on Makespan

- Larger matrices significantly increase overall execution time, but the impact of adding threads is more pronounced for 2048×2048 matrices.
- Strategies like block-based are less effective for large matrices, likely due to the overhead of block management.

5.2 Effect on Speedup

- Speedup is more stable for medium-sized matrices (1024×1024), while for very large matrices (2048×2048), strategies need to be chosen carefully (dynamic is preferred).
- Small matrices (512×512) benefit less from parallelization as the overhead becomes significant relative to the task size.

5.3 Behavior of Strategies

- **Static:** Performs well for small to medium-sized matrices and fewer threads. Loses efficiency for larger matrices or a high number of threads.
- **Dynamic:** Provides good adaptability, particularly useful for larger matrices and a higher number of threads.
- **Block-Based:** Performs well for small and medium-sized matrices but is unsuitable for large sizes and a high number of threads.

6 Conclusion

- **Matrix Size:** Larger matrices benefit more from parallelization, but appropriate strategies must be chosen to avoid overhead.
- **Strategies:** The dynamic strategy is generally the most suitable for large matrices and a high number of threads.
- **Number of Threads:** Adding more threads is beneficial up to a certain threshold (typically 4 for these experiments). Beyond that, speedup gains diminish or may become negative.