Homework 1: Exploring Implicit and Explicit Parallelism with OpenMP

Chiara Sabaini — 238611
BsC Computer, Communication and Electronics Engineering
Dept. of Information Science and Engineering, University of Trento
chiara.sabaini@studenti.unitn.it
a.y. 2024/2025

***Abstract*—This paper provides a comprehensive analysis of implicit and explicit optimization techniques, evaluating their effectiveness using OpenMP for matrix transposition operations. By comparing the performance of these methods, the objective is to determine optimal strategies for various matrix sizes and hardware configurations. The study examines the influence of factors, such as thread count and matrix dimensions, on the performance improvements achieved through parallelization. The findings of this research offer insights into the determinants of parallel performance and can inform the development of efficient parallelization techniques for future applications. Further investigation and code refinement are required to broaden the application of optimization techniques in parallel computing.**

***Index Terms*—parallel computing, openmp, matrix operations, performance evaluation**

I. Introduction

In the domain of high-performance computing, parallel programming techniques are essential for accelerating computationally intensive tasks. OpenMP [1], a widely-used shared-memory programming model, provides a straightforward and efficient means to parallelize code. This project examines the effectiveness of implicit parallelism, achieved through compilation flags, and explicit parallelism, utilizing OpenMP, to optimize matrix transposition operations.

Matrix transposition is a fundamental operation in linear algebra and numerical analysis, frequently employed in scientific computing, machine learning, and data science. However, it can be computationally expensive, particularly for large matrices. By leveraging parallel programming techniques, it is possible to significantly reduce the execution time of matrix transposition.

This report explores the implementation and performance analysis of both implicit and explicit parallel approaches using OpenMP. It discusses the advantages and disadvantages of each approach and provides insights into the factors that influence parallel performance, such as matrix size and the number of threads. The results of this study can inform the development of efficient parallelization strategies for matrix transposition and other computationally intensive tasks.

II. State of the Art

Parallel programming has long been integral to high-performance computing. Numerous methodologies and tools have been devised to exploit the computational capabilities of multi-core and multi-processor systems [2]. OpenMP, a prevalent shared-memory programming model, facilitates parallelization by enabling developers to incorporate compiler directives that define parallel regions, loop parallelization, and thread management, thereby ensuring efficient execution across multiple threads.

A. Existing Solutions and Limitations

Previous studies have predominantly concentrated on explicit parallel programming methodologies utilizing OpenMP for matrix transposition. These methods can result in substantial performance enhancements but frequently necessitate meticulous manual parallelization, which can be prone to errors and time-intensiveness.

Explicit parallelism allows developers to specify exactly how loops and sections of code should be parallelized using OpenMP pragmas. This approach provides fine-grained control over the parallel execution, enabling optimizations tailored to specific hardware architectures and workloads. However, it requires a deep understanding of parallel programming concepts and careful management of thread synchronization and data dependencies to avoid issues such as race conditions and deadlocks.

Implicit parallelism, on the other hand, employs compiler optimizations to automatically parallelize code, thereby simplifying the programming process. This approach relies on the compiler's ability to analyze the code and determine which parts can be safely executed in parallel. While this can significantly reduce the complexity of parallel programming and make it more accessible to developers, it may not consistently attain optimal performance. The compiler's automatic parallelization may not always be able to exploit the full potential of the hardware, especially for complex or irregular code patterns. Additionally, the performance gains achieved through implicit parallelism can vary widely depending on the quality of the compiler and the specific characteristics of the code being parallelized.

Both approaches have their own set of limitations. Explicit parallelism, while powerful, can lead to increased code complexity and maintenance challenges. Implicit parallelism,

although easier to implement, may result in suboptimal performance and less predictable scalability. Therefore, the choice between implicit and explicit parallelism often depends on the specific requirements of the application, the expertise of the developers, and the target hardware environment.

B. Research Gap

Despite significant advancements in parallel programming, there remains a need for efficient and scalable algorithms for matrix transposition. Existing research has primarily focused on explicit parallelism using OpenMP, which, while effective, often requires intricate manual optimization. This project aims to fill this gap by systematically evaluating both implicit and explicit parallelism techniques using OpenMP. By comparing their performance across various matrix sizes and hardware configurations, I seek to identify the most effective strategies for optimizing matrix transposition.

III. Contribution and Methodology

A. Statement of the Task

This work investigates the effectiveness of implicit and explicit parallelization for matrix transposition. Aiming to compare the performance of these approaches and identify the factors that influence their efficiency.

B. Problem Description

Matrix transposition is a common operation in scientific computing and various data processing tasks. However, for large matrices, the execution time can become a significant bottleneck. This project explores the use of compilation flags and OpenMP, a shared-memory programming model, to accelerate matrix transposition by leveraging parallel processing capabilities of modern computers.

C. Approaches and Comparison

This work evaluates two approaches to parallelize matrix transposition using OpenMP.

1) Implicit Parallelism: This approach relies on the compiler to automatically parallelize loops within the matrix transposition code. It offers a simpler programming experience as manual parallelization is not required.

Advantages:

- Easier to implement, requires minimal code modification.
- Compiler optimizations can potentially improve performance.

Disavantages:

- Less control over the parallelization strategy.
- May not achieve optimal performance across different hardware architectures or matrix sizes.

In this project, implicit parallelization is achieved by adding a single line immediately before the definition of the function to be parallelized, as demonstrated in the following code snippet:

Listing 1. Implicit Parallelization using GCC Optimization

```
#pragma GCC optimize ("O2,tree-vectorize")
```

In this context, the $\#pragma$ directive serves as a compiler instruction for the GNU Compiler Collection (GCC) to optimize functions using the designated flags. The flags employed in this directive are:

- O2: Optimization Level 2
  Enables a higher level of optimization, including loop optimizations, instruction scheduling, and register allocation. It aims to balance compilation time and performance improvement.
- tree-vectorize: Tree Vectorization
  Identifies and vectorizes loops, especially those with simple arithmetic operations. It transforms the loop into a series of vector instructions, allowing the processor to process multiple data elements simultaneously.

While the directive itself doesn't directly implement parallelism, it enables compiler optimizations that can lead to implicit parallelization. The compiler can identify opportunities for parallel execution, such as loop iterations that are independent and can be executed concurrently on multiple cores.

2) Explicit Parallelism: This approach involves manually inserting OpenMP directives within the code to explicitly define parallel code areas and thread management. This requires a deeper understanding of OpenMP constructs.

Advantages:

- Provides finer-grained control over parallelization strategy.
- Can be optimized to achieve better performance for specific hardware configurations and matrix sizes.

Disadvantages:

- Requires more complex implementation compared to implicit parallelism.
- More prone to errors if parallelization is not implemented correctly.

In this project, explicit parallelization is achieved by adding OpenMP directives to the code. For example, the following directives parallelize loops using OpenMP:

Listing 2. matTranspose() explicitly parallelized using OpenMP

```
#pragma omp parallel for collapse(2)
```

Listing 3. checkSym() explicitly parallelized using OpenMP

```
#pragma omp parallel for collapse(1) reduction(&:isSym)
```

In these contexts, the $\#pragma$ directive instructs the compiler to parallelize the subsequent loop using OpenMP. The omp keyword specifies that the directive pertains to OpenMP, while the $parallel for$ keywords indicate that the for loop should be executed in parallel. The $collapse$ clause specifies the number of nested loops to be combined into a single loop for parallel execution, and the $reduction$ clause is used to perform a reduction operation on a variable across all threads, ensuring that at the end of the parallel region, the variable is aggregated using the designated operator.

TABLE I
HARDWARE SPECIFICATIONS OF THE EXPERIMENTAL SYSTEM.

Hardware Specification	Value
CPU	Intel(R) Xeon(R) Gold 6252N CPU @ 2.30GHz
Cores	96
Threads per Core	1
Sockets	4
L1d Cache	32 KB per core
L1i Cache	32 KB per core
L2 Cache	1 MB per core
L3 Cache	36.6 MB shared across all cores
NUMA Nodes	4
Compiler	GCC 9.1.0

D. Methodology

The methodology employed in this study involves the implementation and evaluation of matrix transposition using both implicit and explicit parallelization techniques. The process involves setting up a high-performance computing environment, implementing matrix transposition and symmetry check functions in C, and applying both implicit and explicit parallelization techniques on the functions. Performance is evaluated by measuring execution times for various matrix sizes and thread counts, followed by data analysis to identify factors influencing performance.

This methodology provides a comprehensive framework for assessing the effectiveness of parallelization strategies in matrix transposition, offering insights into optimizing parallel performance for various computational tasks.

IV. EXPERIMENTS AND SYSTEM DESCRIPTION

A. Experimental Setup

The experiments were conducted on a high-performance computing (HPC) node identified as $hpc - c12 - node15.unitn.it$. The system specifications are summarized in Table I.

The matrices utilized in the experiments, with dimensions ranging from $2^4 \times 2^4$ to $2^{16} \times 2^{16}$, were randomly populated of float values. The performance of the matrix transposition operation was assessed using both implicit and explicit parallelization techniques. Execution time was recorded for different thread counts to evaluate the scalability of the parallel implementations.

B. Relevant Specifications

The implementation was conducted entirely in C, utilizing the OpenMP library for parallelization. The experiments were executed using the GCC compiler version 9.1.0, which supports OpenMP directives for parallel programming. Performance evaluation involved measuring the execution time of the matrix transposition and symmetry check functions across various matrix sizes and thread counts. Data analysis was performed to identify factors influencing parallel performance and scalability, using Python ran in a Jupyter notebook.

C. Implementation Details

The code implementation and performance evaluation were conducted using C and OpenMP on a multi-core system I. The complete source code is available in the GitHub repository (VII).

The following code snippets show the base sequential implementations for matrix transposition and symmetry checking functions.

Listing 4. matTranspose() function implementation in C

```
void matTranspose(float** M, float** T, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            T[j][i] = M[i][j];
        }
    }
}
```

Listing 5. checkSym() function implementation in C

```
bool checkSym(float** M, int n) {
    bool isSym = true;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (M[i][j] != M[j][i]) {
                isSym = false;
                break;
            }
        }
        if (!isSym){
            break;
        }
    }
    return isSym;
}
```

The $matTranspose()4$ function performs the matrix transposition operation, while the $checkSym()4$ function checks the symmetry of a matrix.

D. Experimental Design

The experiments were meticulously designed to achieve the objectives outlined in the introduction. Each experiment was structured to isolate specific variables and measure their effects accurately.

The hypotheses formulated for the experiments are the following:

- **Hypothesis 1:** Explicit parallelism improves performance regardless of matrix size.
- **Hypothesis 2:** Explicit parallelism offers superior performance compared to sequential implementations.
- **Hypothesis 3:** Implicit parallelism can achieve performance comparable to explicit parallelism.

The experiments were conducted in two phases:

- **Experiment 1:** Tested the $checkSym()$ function on symmetric matrices of different sizes, using varying numbers of threads to accurately measure the execution time and efficiency.
- **Experiment 2:** Tested the $matTranspose()$ function on randomly initialized matrices of different sizes, using varying numbers of threads to measure execution time and scalability.

V. Results and Analysis

A. Hypothesis 1: Explicit parallelism improves performance regardless of matrix size.

Based on Fig. 1 and 2, it can be concluded that explicit parallelism significantly enhances the performance of both the $matTranspose()$ and $checkSym()$ functions, particularly for larger matrices. Notably, there is an anomaly where the sequential implementation (using a single thread) outperforms the parallel implementation for small $2^4 \times 2^4$ matrices. Beyond this anomaly, the execution time consistently decreases as the number of threads increases, demonstrating effective utilization of computational resources through parallelization. The linear decrease in execution time with an increasing number of threads indicates good scalability of the parallel implementations. This observation confirms that explicit parallelism improves performance for larger matrices, thereby partially supporting Hypothesis 1.

B. Hypothesis 2: Explicit parallelism offers superior performance compared to sequential implementations.

Speedup and efficiency are calculated using the following formulas:

$$Speedup = \frac{T_{sequential}}{T_{parallel}} \tag{1}$$

$$Efficiency = \frac{Speedup}{N_{threads}} \tag{2}$$

where $T_{sequential}$ is the execution time of the sequential implementation, $T_{parallel}$ is the execution time of the parallel implementation, and $N_{threads}$ is the number of threads used in the parallel implementation.

From Fig. 3 and 4, it is evident that explicit parallelism significantly enhances performance compared to the sequential implementation for both the $matTranspose()$ and $checkSym()$ functions. The observed speedup ratios consistently exceed 1, indicating that the parallel implementation outperforms its sequential counterparts. Furthermore, the efficiency of the parallel implementation, as depicted in Figures 5 and 6, underlines the efficacy of explicit parallelism in leveraging computational resources. Although efficiency values tend to decrease as the number of threads increases, this is an anticipated outcome due to the overhead associated with thread management. These findings substantiate that explicit parallelism offers superior performance relative to sequential implementations, thereby supporting Hypothesis 2.

C. Hypothesis 3: Implicit parallelism can achieve performance comparable to explicit parallelism.

Fig. 7 and 8 illustrate the execution time of the $matTranspose()$ and $checkSym()$ functions for both sequential,implicitly and explicitly parallel implementations. The results indicate that implicit parallelism, achieved through compiler optimizations, can achieve performance comparable to explicit parallelism using OpenMP. The execution times of the implicit and explicit parallel implementations are similar across different matrix sizes and thread counts, suggesting that implicit parallelism can be as effective as explicit parallelization in certain scenarios. These findings partially support Hypothesis 3, indicating that implicit parallelism can achieve comparable performance to explicit parallel

VI. Conclusions

Based on the results and analysis, we conclude that:

- Explicit parallelism significantly enhances performance for larger matrices, confirming that it improves performance regardless of matrix size, although it may not be beneficial for very small matrices.
- Explicit parallelism offers superior performance compared to sequential implementations, with observed speedup ratios consistently exceeding 1, demonstrating effective utilization of computational resources.
- Implicit parallelism can achieve performance comparable to explicit parallelism in certain scenarios, suggesting that compiler optimizations can be effective in leveraging parallel execution.

This paper presented an analysis of implicit and explicit parallelism using OpenMP for matrix transposition operations. The performance of both approaches was evaluated for varying matrix sizes and thread counts to identify the factors that influence parallel performance. The results of this study can inform the development of efficient parallelization strategies for matrix transposition and other computationally intensive tasks. Future work will focus on optimizing the parallel implementations further and exploring additional parallel programming techniques to enhance performance.

VII. GIT and Instructions for Reproducibility

The source code and experimental data for this project are available in the GitHub repository at https://github.com/chiarasabaini/parco-homework. The repository contains the C code for the matrix transposition and symmetry check functions, as well as the Python scripts used for data analysis and visualization. Detailed instructions for reproducing the experiments are provided in the README file in the repository.

References

[1] "The OpenMP API specification for parallel programming", https://www.openmp.org/ (Accessed: Dec. 1, 2024)

[2] J. Choi, J. J. Dongarra and D. W. Walker (1993, Oct.), "Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers", https://www.netlib.org/lapack/lawnspdf/lawn65.pdf

[3] https://www.openmp.org/wp-content/uploads/OpenMPRefGuide-5.2-Web-2024.pdf

[4] https://gcc.gnu.org/onlinedocs/gcc/Function-Specific-Option-Pragmas.html

[5] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Appendix

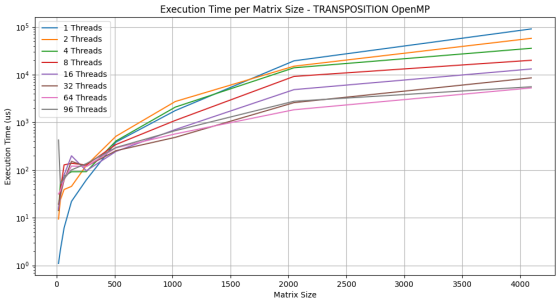

Fig. 1. Execution time for $matTranspose()$ Function Explicitly Parallelized with OpenMP.

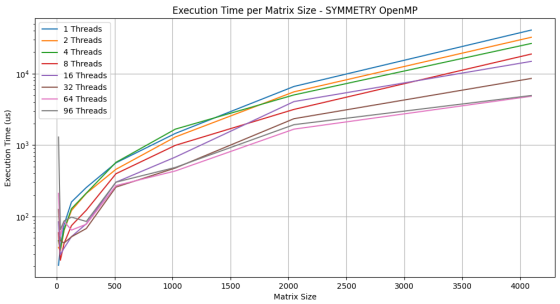

Fig. 2. Execution time for $checkSym()$ Function Explicitly Parallelized with OpenMP.

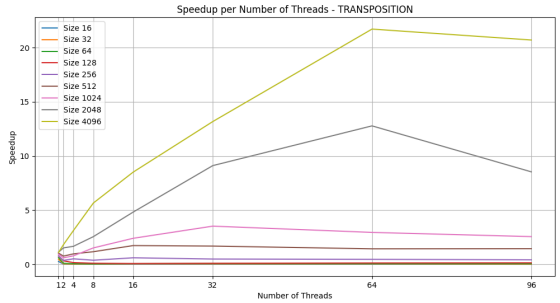

Fig. 3. Speedup Ratio of the OpenMP Implementation for the $matTranspose()$ function.

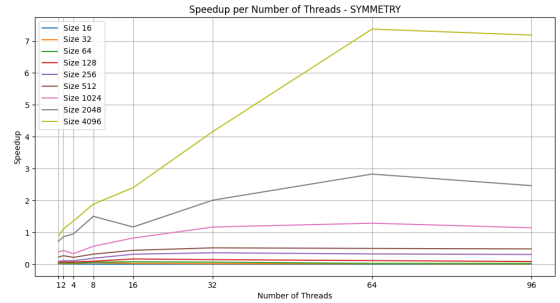

Fig. 4. Speedup Ratio of the OpenMP Implementation for the $checkSym()$ function.

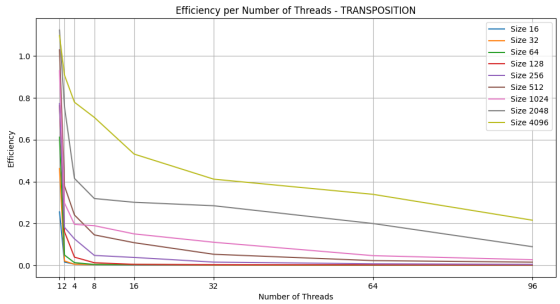

Fig. 5. Efficiency of the OpenMP Implementation for the $matTranspose()$ function.

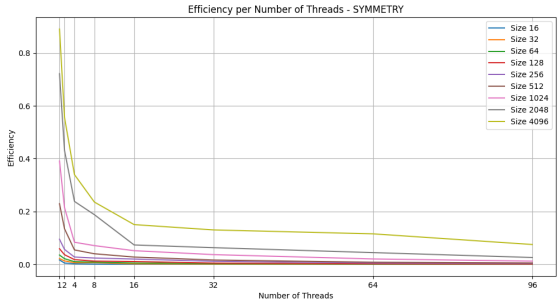

Fig. 6. Efficiency of the OpenMP Implementation for the $checkSym()$ function.

Fig. 7. Execution Time of the $matTranspose()$ Function, for Different Implementations.

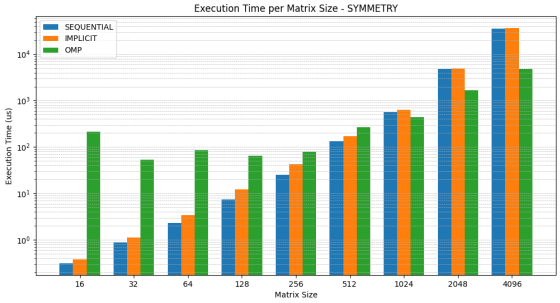

Fig. 8. Execution Time of the $checkSym()$ Function, for Different Implementations.