Optimizing Matrix Transposition with Implicit and Explicit Parallelism

Riccardo Randon
University of Trento
Trento, Italy
Student ID: 231331
riccardo.randon@studenti.unitn.it

***Abstract*—This report explores the optimization of matrix transposition using sequential, implicit, and explicit parallelization techniques. Performance evaluations are conducted to compare the efficiency, speedup, and scalability of each approach across different matrix sizes. The results highlight the benefits and limitations of each method, along with potential improvements for enhanced performance.**

I. Introduction

Matrix transposition is a fundamental operation in many scientific and engineering applications, where it involves swapping rows and columns of a matrix. The performance of such operations is critical, especially when dealing with large datasets in parallel computing environments. This report focuses on optimizing the matrix transposition operation through various parallelization techniques to improve efficiency and scalability.

The primary objective of this project is to explore the impact of both implicit and explicit parallelism on the matrix transposition process. Initially, a sequential implementation is developed, followed by optimizations using implicit techniques like vectorization and memory access optimizations. Subsequently, explicit parallelization is achieved using OpenMP, a widely-used parallel programming framework. The performance of these implementations is compared in terms of speedup, efficiency, and scalability across varying matrix sizes and thread counts.

By examining the performance gains and potential bottlenecks, this project aims to provide insights into the effective use of parallelism in matrix operations.

II. STATE OF THE ART

Matrix transposition [1]is a key operation in many parallel computing applications, and optimizing its performance has been a subject of extensive research. Several parallelization strategies have been proposed to enhance its efficiency.

A. Sequential Matrix Transposition

In sequential implementations, the matrix is transposed by swapping elements, which results in a time complexity of $O(n^2)$ for an $n \times n$ matrix. While straightforward, this approach becomes inefficient for large matrices and does not exploit the capabilities of modern multi-core processors.

B. Implicit Parallelism

Implicit parallelism optimizes performance without requiring explicit thread management. This is achieved through techniques such as vectorization, memory prefetching, and cache optimization. Modern processors, particularly those equipped with Intel's Advanced Vector Extensions (AVX), can significantly enhance matrix transposition performance by executing multiple operations per cycle [2]. While compilers can automatically leverage these hardware-level optimizations, the design of efficient code remains crucial to fully realize their potential.

C. Explicit Parallelism with OpenMP

OpenMP [3] is a widely used framework for parallelizing matrix transposition by distributing tasks across multiple threads. Techniques like block-based transposition, which divide the matrix into smaller blocks for processing, have been shown to improve performance by maximizing cache utilization.

This project builds on these approaches by comparing sequential, implicit, and explicit parallelization techniques, focusing on performance improvements across different matrix sizes.

III. Contribution and Methodology

This project aims to analyze the performance of matrix transposition and check symmetry functions using three different parallelization strategies: sequential, implicit, and explicit parallelism via OpenMP. The methodology for implementing and evaluating the matrix transposition is as follows:

A. Sequential

In the sequential implementation, a single-threaded matrix transposition and symmetry check are performed without parallelization. As shown in Algorithm 1, the sequential matrix transposition is straightforward.

B. Implicit Parallelization

In the implicit parallelization, the matrix transposition and symmetry check functions are optimized using block-based algorithms. This approach aims to improve memory access and enhance cache performance by dividing the matrix into smaller blocks. Prefetching is applied to ensure that each block

Algorithm 1 Sequential Matrix Transposition

```
for $i = 0$ to $n - 1$ do
    for $j = 0$ to $n - 1$ do
        $T[i][j] \leftarrow M[j][i]$
    end for
end for
```

is loaded into the cache efficiently. The block size is optimized based on the current machine's architecture. Algorithm 2 show the pseudocode for the block-based transposition.

Algorithm 2 Block-Based Matrix Transposition

```
for $i = 0$ to $n - 1$ by blockSize do
    for $j = 0$ to $n - 1$ by blockSize do
        for $k = i$ to $i + blockSize - 1$ do
            for $l = j$ to $j + blockSize - 1$ do
                $T[k][l] \leftarrow M[k][l]$
            end for
        end for
    end for
end for
```

C. Explicit Parallelization with OpenMP

For explicit parallelization, OpenMP directives are used to parallelize the matrix transposition and symmetry check. After several attempts with different configurations, the most effective pragma for improving performance was `collapse(2)`, which allows for better load balancing across threads by collapsing two nested loops into a single parallel loop.

Initially, prefetching optimizations were also applied, but these had to be removed due to performance degradation. This may be because the cache prefetching directives were causing cache contention reducing its effectiveness for the parallelized loops.

D. Performance Metrics

The performance of the three implementations was evaluated based on the following metrics:

- **Execution Time:** The total wall-clock time taken to complete the matrix transposition and symmetry check functions. This metric provides an indication of the computational efficiency of the different implementations.
- **Speedup:** The baseline of the speedup is the parallelized implementation with OpenMP with just one thread:
$$\text{Speedup} = \frac{\text{Time}_{\text{seq}}}{\text{Time}_{\text{parallel}}}$$
- **Efficiency:** The efficiency of the parallel implementation is calculated as the speedup divided by the number of threads used:
$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of threads}}$$
- **Bandwidth Test and Peak Performance:** The bandwidth test evaluates the memory throughput by measuring how efficiently data is transferred between the CPU and memory during the matrix transposition process. The bandwidth B is calculated as:
$$B = \frac{2 \times n^2 \times \text{sizeof(float)}}{\text{Time}_{\text{parallel}}}$$
where:
  - n is the dimension of the matrix ($n \times n$).
  - The factor 2 accounts for both the read and write operations performed on each element.

  The theoretical peak bandwidth B_{peak} is determined based on the system's memory specifications:
$$B_{\text{peak}} = \text{S (GHz)} \times \text{B (bytes)} \times \text{n}$$
where:
  - $S(GHz)$: The operating frequency of the system's memory.
  - $B(bytes)$: The width of the data bus connecting the memory.
  - n: The number of memory channels supported by the system.

  The observed bandwidth B is compared with B_{peak} to assess the efficiency of memory usage. Deviations from B_{peak} may indicate potential bottlenecks, such as limited thread scalability or suboptimal memory access patterns.

IV. Experiments and System Description

A. System Description

The experiments were conducted on a computing system with the following specifications:

- **Processor:** Intel Core i7-8665U, 4 cores, 8 threads, 1.90 GHz base frequency (up to 4.80 GHz)
- **Memory:** 16 GB DDR4 RAM, 2667 MHz, single 16 GB DIMM (Hynix Semiconductor)
- **Cache:**
  - L1 Cache: 128 KiB (per core)
  - L2 Cache: 1 MiB (per core)
  - L3 Cache: 8 MiB (shared)
- **Operating System:** Ubuntu 20.04 LTS
- **Compiler:** GCC 9.3.0 (GNU Compiler Collection)
- **OpenMP Version:** OpenMP 4.5
- **Programming Language:** C

Based on this notions the block size for the matrix operations was chosen according to the cache size of 32 MB to optimize memory access.

B. Experimental Setup

The experimental setup for this study is designed to benchmark the performance of matrix transposition across three different implementations. For each implementation, the following steps were carried out:

- A random matrix of size $n \times n$ with floating-point numbers was generated.
- The matrix was transposed after the symmetry check, with wall-clock time measured for both operations.

- The matrix size n was varied from 2^4 to 2^{12} to assess performance scalability across different problem sizes.
- For OpenMP-based implementations, the number of threads was varied from 1 to 32 (1, 2, 4, 8, 16, 32) to measure scalability with respect to the number of cores.
- For better reliability, each sample was taken 50 times, and the arithmetic mean time was recorded.

V. Results and Discussion

For better clarity, the following charts focus only on the matrix transpose operation. The symmetric check function yields similar results.

A. Performance Results

The performance of the three implementations—sequential, implicit parallelization, and OpenMP parallelization—is shown in Figure 1.

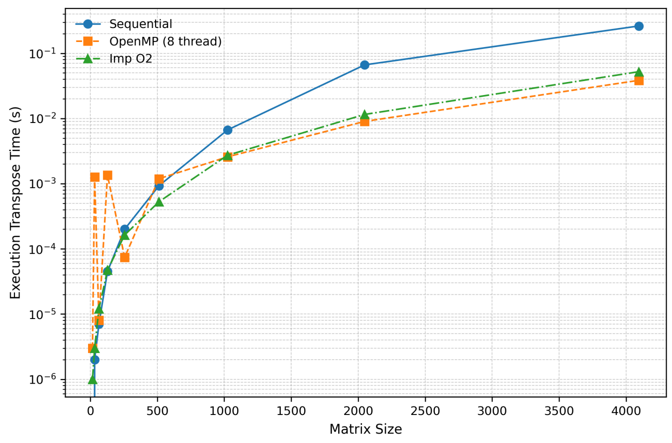

Fig. 1. Execution times for different matrix sizes using the sequential, implicit, and OpenMP parallelized implementations. (logarithmic scale)

As observed, the sequential implementation shows significantly poorer performance on larger matrices. Anomalies can be observed in the OpenMP implementation, particularly for smaller matrix sizes. This could be due to concurrency issues arising from block and thread management.

B. Performance of Implicit Implementation with Different Flags

The implicit implementation was built with four different compilation flags: -O1, -O2, -O3, and -Ofast. Figure 2 illustrates that the impact on execution time is relatively small across these different flags.

Despite the differences in optimization levels, there is no significant variation in the execution time, indicating that the choice of compilation flag does not drastically affect the performance in this case.

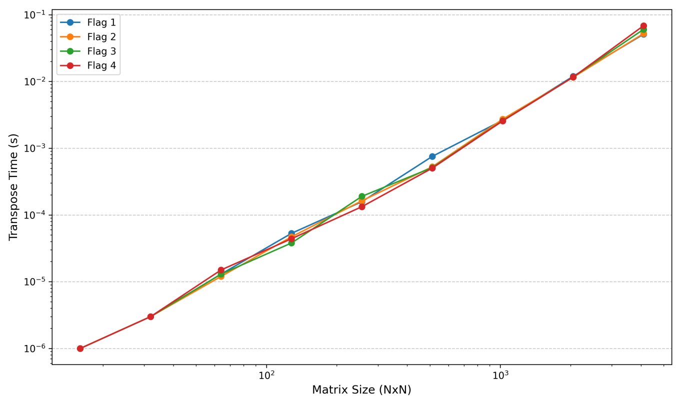

Fig. 2. Performance comparison across different compilation flags. (logarithmic scale)

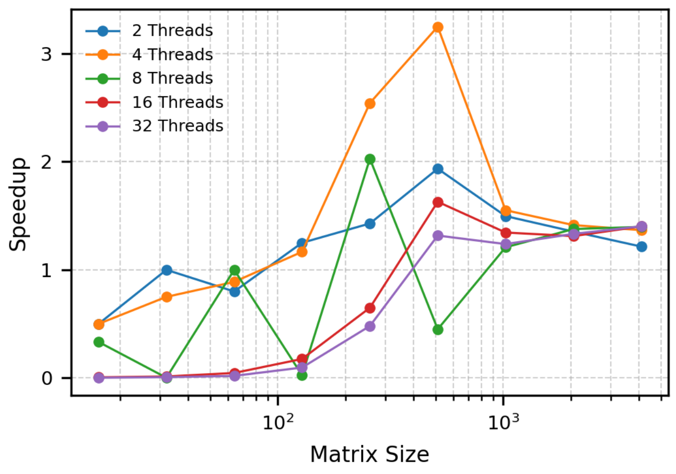

Fig. 3. Speedup comparison for different matrix sizes.(logarithmic scale)

C. Speedup

Speedup and efficiency were calculated for the OpenMP implementation relative to the sequential baseline. The results are summarized in Figure 3.

As expected, the speedup increases with the matrix size. However, the efficiency of the OpenMP implementation could potentially be further optimized by fine-tuning block sizes and workload distribution among threads. In particular, some configurations with specific numbers of threads show irregular patterns depending on the matrix dimension.

D. Efficiency

The efficiency, shown in Figure 4, drops significantly as the number of threads increases. This drop could be attributed to several factors, such as overhead from thread management, non-uniform workload distribution, or diminishing returns with respect to the available hardware resources.

E. Bandwidth

The observed bandwidth for the OpenMP implementation was compared with the theoretical peak bandwidth in Figure 5.

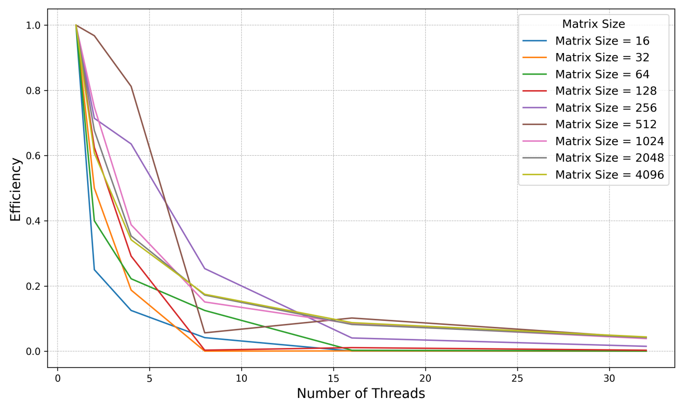

Fig. 4. Efficiency of OpenMP implementation as a function of matrix size and thread count.

The analysis revealed a significant discrepancy between the theoretical peak bandwidth and the observed values. While the observed bandwidth improves with larger matrices, the difference remains substantial. This could be due to factors like suboptimal memory access patterns, cache inefficiencies, or limitations in the parallelization strategy.

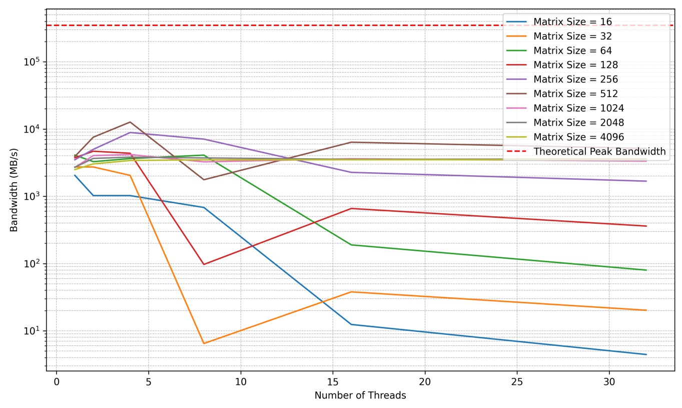

Fig. 5. Observed bandwidth for different matrix sizes and thread counts compared to the theoretical peak bandwidth.(logarithmic scale)

VI. Conclusions

In this work, we explored different parallelization techniques to optimize the matrix transposition operation. We compared three implementations: a sequential version, an implicit parallelization approach utilizing compiler optimizations, and an explicit parallelization strategy using OpenMP.

The sequential implementation served as the baseline, demonstrating the typical growth in execution time as the matrix size increased. The implicit parallelization approach, while improving performance slightly, still exhibited limitations due to the constraints of compiler optimizations, especially for larger matrix sizes.

The OpenMP-based parallelization showed the most significant improvements in performance. As the matrix size increased, the OpenMP implementation achieved substantial speedup, demonstrating the effectiveness of explicit parallelization in handling large-scale problems. The speedup and efficiency results confirmed that OpenMP parallelization could effectively reduce computation time and improve performance scalability, particularly for larger matrix sizes.

While the OpenMP implementation demonstrated promising results, there are still opportunities for optimization. Potential future work could explore advanced techniques, such as cache blocking, dynamic scheduling, or GPU-based parallelization, to further enhance performance.

Overall, the study highlights the power of parallelization in improving the efficiency of computationally intensive operations such as matrix transposition. By comparing various parallelization strategies, we demonstrated how careful optimization and parallel computing techniques can lead to significant performance improvements in real-world applications.

A. *Potential Optimizations and Future Work*

While the OpenMP implementation shows promising results, there are still areas for improvement and challenges to address. One notable issue lies in the symmetry check function. First, the current implementation assumes ideal conditions, direct comparisons of floating-point numbers are prone to precision issues, making it preferable to include a threshold for approximate equality during comparisons. Second, the OpenMP implementation of the symmetry check is inefficient in most cases because the loop never breaks early when it determines that the matrix is not symmetric, leading to unnecessary computations.

Additionally, while all experiments were conducted on a local machine for simplicity and accessibility, testing on an HPC cluster would provide a more interesting and realistic analysis of performance, particularly for larger matrix sizes and more complex workloads. Future work could also explore GPU-based parallelization using CUDA for very large matrices, as well as more OpenMP technics.

VII. Git and Instructions for Reproducibility

To ensure the reproducibility of the results, the source code and instructions for running the experiments are provided through a GitHub repository. The repository contains all the necessary files, including the implementation of the matrix transposition routines (sequential, implicit parallelization, and OpenMP), as well as the required dependencies and setup instructions.

The project can be found at the following GitHub repository link:

https://github.com/rikirandon/Intro_PARCO_H1

References

[1] Wikipedia, ”Matrix Transpose” *https://en.wikipedia.org/wiki/Transpose*
[2] Wikipedia, ”Advanced Vector Extensions” *https://en.wikipedia.org/wiki/Advanced Vector Extensions*
[3] Wikipedia, ”OpenMP” *https://en.wikipedia.org/wiki/OpenMP*