Parallel Computing

1st Leonardo Falsarolo, student ID 237821
Department of Information Engineering and Computer Science
University of Trento
Trento, Trentino Alto Adige
leonardo.falsarolo@studenti.unitn.it

***Abstract*—Matrix transpose is a fundamental operation in many computationally intensive applications, such as data processing, machine learning, and scientific simulations. This paper explores the implementation of parallel computing techniques to optimize the performance of a matrix transposition.**

***Index Terms*—Matrix Transpose, Parallel Computing, OpenMP, Data Parallelism, Performance Optimization.**

I. Introduction

Parallelization involves dividing a task into smaller subtasks that can be executed simultaneously across multiple processing units, such as cores or processors. This approach leverages the power of modern multi-core systems to speed up calculations and handle larger datasets. Parallelization can be implicit, where the system or compiler manages it automatically, or explicit, where the programmer manually defines the distribution of tasks.

Matrix transposition is a permutation, which can be factored into a product of disjoint cycles [5], and is a fundamental operation in various algorithms, such as solving linear systems, data analysis, and machine learning. It involves non-sequential data access, which can lead to inefficiencies in memory and cache management. This makes it an ideal benchmark for evaluating parallelization strategies.

The goal of this study is to compare implicit and explicit parallelization techniques for transposing an n × n matrix, evaluating their performance, scalability, and ease of implementation. Through benchmarking on matrices of various sizes, this work aims to determine which approach is more efficient and in which contexts, providing insights to help select the best technique based on specific computational needs.

II. State of the art

A. Current research and developments related to my project.

Matrix transposition is key for optimizing parallel computations. Recent research has focused on exploiting the hardware capabilities of Xeon Phi CPUs and Intel Xeon Phi coprocessors, using cache optimization techniques to improve performance. These systems achieve 82% and 67% of theoretical efficiency, respectively, by using loop tiling and "pragma simd" [1].

GPU-based approaches also enhance matrix transposition by leveraging thread management and memory coalescing. GPUs are particularly effective due to their higher memory bandwidth compared to CPUs, enabling faster handling of large matrices [2] [3].

Additionally, other studies explore alternative memory models, such as the RAM-model, I/O-model, cache-model, and cache-oblivious-models. Cache and I/O models focus on reducing memory latency and improving throughput by minimizing cache and TLB misses, while cache-oblivious algorithms aim to minimize data references [4].

B. Limitations

There are several limitations to consider. Challenges include balancing workload across cores, managing memory for large matrices, and reducing latency on heterogeneous platforms.

The Xeon Phi CPU relies on optimizations like loop tiling and SIMD instructions, while the Intel Xeon Phi coprocessor relies on efficient cache utilization. A common limitation is the overhead in thread management [1].

For GPUs, memory is often much more limited than CPUs. With out-of-place transposition, only half of the GPU's memory can be used for matrices, as the operation requires double the space of the original data [2] [3].

Regarding memory models, cache and I/O models face performance issues due to excessive data references, which reduce efficiency. Cache-oblivious models, however, do not directly address cache and TLB misses, which can also negatively impact performance [4].

C. gap my project aims to fill

My project offers a more practical and applied approach compared to previous studies. It is based on benchmarking real transposition algorithms using implicit compiler optimizations and explicit parallelization with OpenMP. Essentially, my project compares various approaches and aims to define the best algorithm for matrix transposition.

III. Contribution and Methodology

A. Unique contributions of my project.

The project focuses on the analysis and optimization of matrix transposition. The unique contributions of the project include:

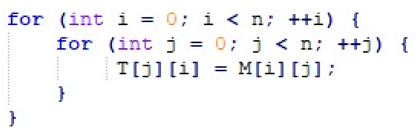
- **Comparison of transposition algorithms**: I tested various matrix transposition algorithms, including both sequential and parallel approaches, to compare their performance and determine which approach delivers the best results based on the hardware configuration.

- **Optimization using implicit parallelizzation**:An additional contribution was the use of compiler flags to optimize the code and maximize hardware potential. Specifically, '-O1' applies basic optimizations like dead code elimination and loop unrolling, while '-O2' applies more aggressive optimizations such as loop optimization and improved instruction scheduling, leading to significant performance improvements.
- **Explicit parallelization with OpenMP**: I used `#pragma omp parallel for collapse(2) num_threads(num_of_threads)`, which allows me to execute a parallel loop distributed across multiple threads by collapsing the two nested loops into one.

B. Methodology

In this project, several algorithms were tested for matrix transposition, each with varying approaches to optimize performance. These algorithms include a classic double-loop method, a block-based approach for cache optimization, and a modified version of the double-loop method, where the second loop starts from the current position of the first loop + 1 to avoid processing the diagonal.

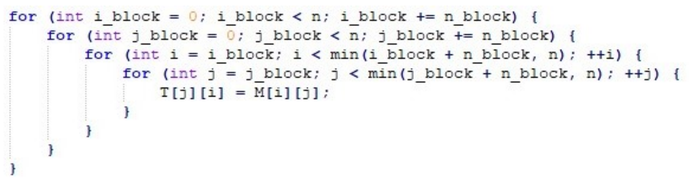
1) **Classic Double For-Loop (Naive Algorithm)**: This is the most straightforward approach to matrix transposition. It iterates through each element of the matrix using a nested loop, performing the transposition element by element. The outer loop runs over the rows of the matrix, and the inner loop runs over the columns. This approach is simple but may not be the most efficient for large matrices due to cache inefficiencies.



```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        T[j][i] = M[i][j];
    }
}
```

Fig. 1. Standard Nested 'for' Loop in C

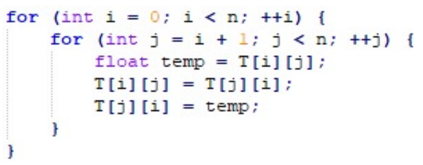
2) **Block-based Algorithm (Tiling)**: The block-based approach improves cache locality by dividing the matrix into smaller BLOCK SIZE × BLOCK SIZE submatrices, with each block transposed individually. This reduces cache misses and overhead from non-sequential memory access, significantly boosting performance for large matrices. The block size was optimized based on the hardware and matrix size, focusing on the most effective option to avoid excessive comparisons.



```
for (int i_block = 0; i_block < n; i_block += n_block) {
    for (int j_block = 0; j_block < n; j_block += n_block) {
        for (int i = i_block; i < min(i_block + n_block, n); ++i) {
            for (int j = j_block; j < min(j_block + n_block, n); ++j) {
                T[j][i] = M[i][j];
            }
        }
    }
}
```

Fig. 2. In-Place Matrix Transposition (Upper Triangle Approach)

3) **In-Place Matrix Transposition (Upper Triangle Approach)**:This method avoids redundant work by only transposing elements that haven't already been transposed and is more efficient, especially for large matrices. The matrix is being transposed without allocating additional memory for another matrix, elements are swapped directly in the original one.



```
for (int i = 0; i < n; ++i) {
    for (int j = i + 1; j < n; ++j) {
        float temp = T[i][j];
        T[i][j] = T[j][i];
        T[j][i] = temp;
    }
}
```

Fig. 3. In-Place Matrix Transposition (Upper Triangle Approach)

Apart from these 3 serial algorithms, I used some compilation flags to optimize execution (implicit parallelization) III-A. Furthermore, I have also implemented explicit parallel execution using `#pragma omp parallel for collapse(2) num_threads(num of threads)` [III-A].

C. Challenges faced and how they were addressed

During the execution of the tests on the university cluster, I had to pay attention to finding similar conditions for running my experiments, as other people could be running their own processes simultaneously. In some cases, the high load from other users on the system affected performance, making it difficult to obtain precise measurements. Therefore, I tried to run the tests when the cluster was less occupied, in order to minimize the impact of other activities on the results. Regarding the issue of outliers, I had to correct them manually.

IV. Experiments and System Description

A. Description of the computing system and platform

For the tests I have used the UniTN's cluster . This experiments were conducted on a high-performance computing system equipped with 4 Intel Xeon Gold 6252N CPUs, each with 24 cores, totaling 96 logical processors, with NUMA architecture to optimize memory access. The performance of my parallel implementations are compared against their respective sequential algorithm, analyzing the impact of compiler optimizations and the scalability of parallel approaches for large matrix sizes."

B. Relevant specifications or configurations.

- To measure the execution time of matrix transposition, I used the "omp_get_wtime()" function from the "omp.h" library. This function returns high-resolution time in seconds based on the system clock. I recorded the start time before the main transposition loop and the end time after the loop. The difference between these values gave the execution time. This method ensures high precision as "omp_get_wtime()" measures wall-clock time, accounting for parallelization performance improvements without significant overhead.

- All tests were conducted using GCC version 9.1.0
- To create the charts, I used Python's Matplotlib library to process .csv files generated by my main program. This allowed me to plot speedup and efficiency for each matrix size as the thread count increased. Additionally, I compared the serial implementation with both implicit parallelization (using compilation flags) and explicit parallelization (with `#pragma Omp parallel for collapse(2)`).

C. Description of the experimental setup, procedures, and methodologies used in the project

For the experimental environment, I used the university computing cluster for all stages of the experiment. The same hardware and computing resources were employed for all simulations, ensuring that any differences in results were due solely to controlled variables such as compilation flags, thread count and matrix sizes. For each of the three transposition methods, I used compilation flags for implicit parallelization, including O1 and O2 [III-A] with other additional flags :

1) **-funroll-loops**: This option aims to transform loops into repetitive code, thereby reducing loop control overhead and improving performance in the case of loops with a large number of iterations. However, when the algorithm is simple the effect of loop unrolling is less significant. [6]
2) **-ftree-loop-im**: This optimization flag helps the compiler improve the performance of loops by rearranging or merging them. The goal is to make better use of the CPU cache and reduce unnecessary overhead in loop execution, making the program run faster. [6]
3) **-march-native**: This flag tells the compiler to generate code that is optimized for the specific CPU architecture of the machine where the program is being compiled. It ensures that the compiler uses the most efficient instructions and features available on that CPU, resulting in better performance. [6]

Parallel simulations were performed using 1, 2, 4, 8, 16, 32, 64, and 96 threads, with matrix sizes ranging from 2^4 to 2^{12}. For the block-based transposition method [III-B], a block size of 256 was chosen, which proved to be optimal. After execution, programs were used to generate graphs comparing the performance (speedup and efficiency) of each transposition method. Additionally, another program analyzed the .csv data to determine which optimization for implicit parallelization was the best for each matrix size, generating a final .csv file with the results.

D. Discussion on how experiments are designed to test the hypotheses or achieve the objectives.

The experiments are designed to test the hypothesis that the use of parallelization, either OMP or implicit, results in faster execution times compared to the serial implementation and that the speedup increases with the number of threads used up to a certain pointRegarding efficiency, it should generally decrease as the number of threads increases. This is due to the increased overhead from thread management when dividing the workload among too many threads.

V. Results and Discussion

A. Presentation of results.

In this section, I present the results of my experiments evaluating the performance of three matrix transposition methodologies, focusing on key metrics like execution time, speedup, and efficiency. The speedup (S) is calculated using the formula:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{1}$$

where T_{serial} is the execution time of the serial implementation, and T_{parallel} is the execution time of the parallel implementation.

The efficiency (E) is determined by the formula:

$$E = \frac{S}{N_{\text{threads}}}$$

where N_{threads} represents the number of threads used in the parallel execution. Efficiency measures how effectively the parallel resources are utilized.

B. Analysis and interpretation in context.

The first graph [4] shows how compiler optimizations, using GCC flags like O1 and O2 [III-A], drastically improve the execution time compared to the serial version. In particular, the use of O1 and O2 flags without extra flag options leads to a significant reduction in execution times. I used a normal scale in the graph to highlight how, for matrices of size 2^{12}, the execution time is almost halved compared to the serial version. Moreover, the `-funroll-loops` option further reduces execution time, but with minimal improvement compared to O1 and O2 optimizations. This is due to the fact that method 1 uses a relatively simple algorithm that doesn't benefit much from loop unrolling. Regarding `-ftree-loop-im` and `-march=native` optimizations have limited impact because the main bottleneck is caused by non-sequential memory access. In matrix transposition, the data access pattern is not contiguous, which prevents these optimizations from fully exploiting the cache and processor capabilities [IV-C].

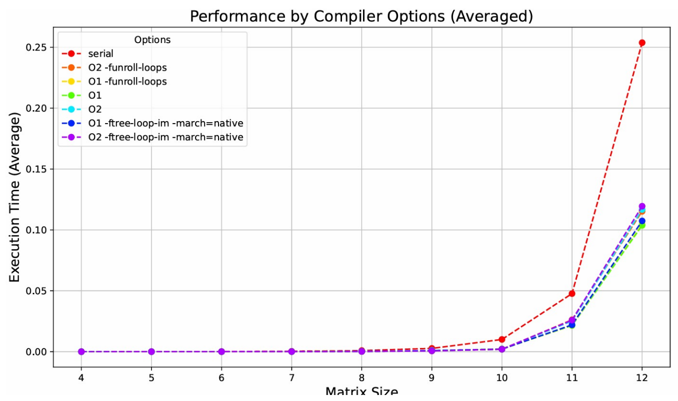


Fig. 4. Implicit using compilation flags

In the second graph [5], instead, it can be seen how, as the number of threads increases, the execution time also decreases significantly, going from an execution time of a transpose of a $2^{12} \times 2^{12}$ matrix of approximately 10^{-1} to about 10^{-2} with 64 threads, reducing the time by approximately 90%. This graph demonstrates the use of explicit parallelization with the OMP library, using the directive `#pragma omp parallel for collapse(2) num_threads(num of threads).` [III-A]

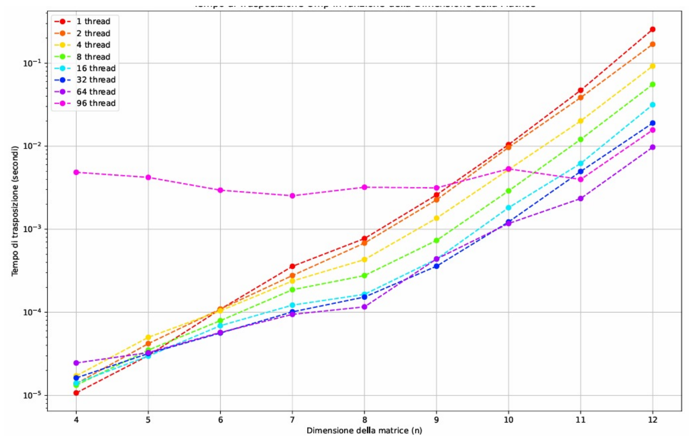


Fig. 5. OpenMp using different number of threads

Regarding speedup [6], the graph clearly shows that as the number of threads increases, it approaches the ideal speedup [IV-D], **especially** for matrices of size 2^{11} and 2^{12}, where we can observe a peak around 64 threads. However, after 64 threads, the performance start to decrease due to the overhead associated with thread management, and we can see that the actual speedup starts to decline.

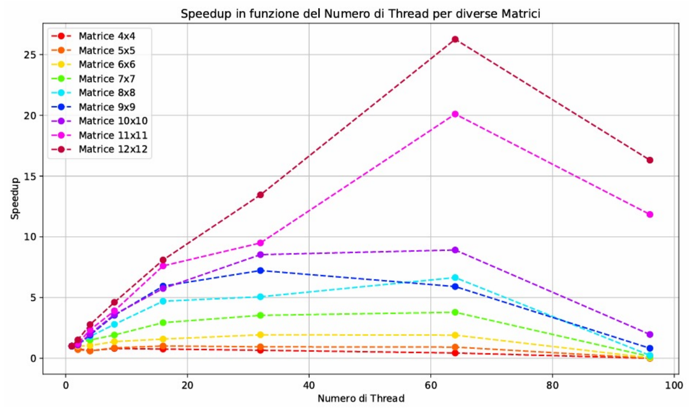


Fig. 6. Speedup for different matrix sizes

Regarding efficiency [7], we notice that as the number of threads increases, efficiency tends to decrease [IV-D]. This occurs mainly due to the overhead introduced by thread management. As the number of threads increases, the ability to fully utilize all available resources is reduced. Consequently, efficiency decreases, and the full computational power added by the threads cannot be fully exploited.

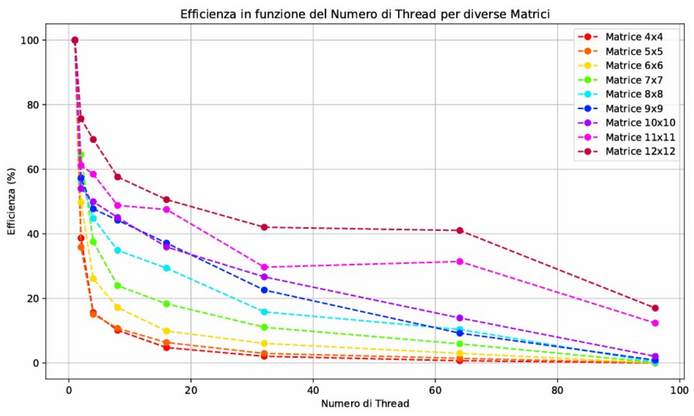


Fig. 7. Efficency for different matrix sizes

These tests were conducted using all available matrix transposition methods to determine the fastest one [III-A]. I concluded that the best method depends on the matrix size. Based on the data, I've summarized the fastest methods for implicit parallelization in the table below.

Matrix Dimension	Min Time	Computation Option
4	4.38839×10^{-7}	O1=256
5	1.46851×10^{-6}	O2 -funroll-loops=method3
6	5.33946×10^{-6}	O2 -funroll-loops=method3
7	1.78952×10^{-5}	O2 -funroll-loops=256
8	5.19291×10^{-5}	O1 -ftree-loop-im -march-native=method3
9	2.36649×10^{-4}	O2=method3
10	7.36079×10^{-4}	O2 -funroll-loops=method3
11	6.20201×10^{-3}	O2 -funroll-loops=256
12	4.83283×10^{-2}	O1 -funroll-loops=method3

C. *Comparison with state of art*

One of the main contributions of my work is the exploration of compiler optimizations, including explicit parallelization with OpenMP. While previous studies have focused on SIMD and GPU-based optimizations, my comparative analysis of implicit vs. explicit parallelization provides new insights, especially in environments without GPU resources. Additionally, my results show that while memory access optimization and efficient memory management are crucial for performance, proper thread management can reduce some of the overhead seen in other studies.

CONCLUSIONS

The results show that the additional methods, Block-based and Upper Triangle Approach methods [III-A], are overall the most efficient. However, the optimal method depends on the matrix size. For small matrices, the Upper Triangle Approach generally perform better due to the simplicity of the algorithm and lower overhead. As the matrix size increases, the block-based method becomes more dominant, effectively leveraging cache memory and minimizing inefficient memory accesses.

GIT-REPOSITORY

https://github.com/FlsLeonardo/Parallel-computing-mid-term

References

[1] Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors Andrey Vladimirov for Colfax Research August 12, 2013

[2] NVIDIA, ”Optimizing Matrix Transposition on GPUs,” Proceedings of PPoPP, 2014.

[3] J. Gómez-Luna, I. -J. Sung, L. -W. Chang, J. M. González-Linares, N. Guil and W. -M. W. Hwu, ”In-Place Matrix Transposition on GPUs,” in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 3, pp. 776-788, 1 March 2016, doi: 10.1109/TPDS.2015.2412549.

[4] Cache-Efficient Matrix Transposition,Siddhartha Chatterjeey,Sandeep Senz

[5] T. Hungerford, Algebra astratta: un’introduzione,1997.

[6] 3.11 Options That Control Optimization gcc.gnu.org