Exploring Implicit and Explicit Parallelism with OpenMP

Mattia Bernabè
Bachelor's degree in Ingegneria Informatica, delle Comunicazioni ed Elettronica
Università di Trento
mattia.bernabe@studenti.unitn.it
ID: 237827
GitHub repository: https://github.com/brna23/matrixTransposition_ParCo

***Abstract*—This project explores different approaches to transposing a matrix in C++: a serial implementation, an implementation utilizing implicit parallelism, and an OpenMP parallelized implementation. The goal is to compare the performance of these approaches, analyze their scalability on different size configurations, and examine the benefits and challenges of parallel computation in matrix operations. Key findings demonstrate the significant speedup achieved by parallelized methods over the serial baseline.**

I. Introduction

Matrix transposition, a fundamental operation in scientific computing and data manipulation, involves swapping the rows and columns of a matrix. Efficiently transposing extensive datasets is essential in various domains, such as image processing, neural networks, and physical simulations. However, for large-scale applications, performing this operation in a serial manner can be computationally expensive.

II. Project Objectives

The objective of this project is to implement three different C++ methods for matrix transposition and evaluate their performance:

1) A baseline serial implementation.
2) A parallel implementation leveraging implicit parallelism.
3) A parallel implementation using OpenMP directives.

III. State-of-the-Art

A. Current Research and Developments

Matrix transposition is a key operation in scientific computing, with performance heavily influenced by cache efficiency and parallelization. Current research leverages OpenMP for parallelizing transposition, using techniques like tiling to improve cache locality and reduce memory latency. Implicit parallelism, provided by OpenMP and modern compilers, simplifies development but often falls short of the performance achieved by manually optimized solutions.

Identify applicable funding agency here. If none, delete this.

B. Existing Solutions and Limitations

1) **Explicit OpenMP** parallelization is efficient but requires significant manual tuning to address issues like load balancing and cache contention.
2) **Tiling and blocking** improve performance but add complexity and may not generalize well to all cases.
3) **Compiler-driven implicit parallelism** is easy to use but often leaves optimization potential untapped.
4) **GPU-based solutions** deliver high throughput but are less accessible and not always cost-effective for smaller workloads.

C. Gap and Aim of This Project

Existing methods struggle to balance performance, simplicity, and adaptability. This project seeks to improve OpenMP-based transposition by enhancing implicit parallelism through runtime optimizations and adaptive tiling strategies, reducing the need for manual tuning while delivering competitive performance on multicore systems.

IV. Contribution and Methodology

A. Unique Contributions

This project contributes a comparative study of serial and parallel approaches to matrix transposition, examining both performance and implementation complexity. The project also provides insights into the impact of OpenMP on matrix operations in shared-memory architectures.

B. Methodology Algorithms

The algorithms for checking the simmetry of the matrix is composed by 2 nested loops to iterate across the whole matrix and a counter that is incremented every time element[i,j] is different than the element[j,i]. This algorithm has a time complexity of $O(n^2)$ due to its nested loop structure iterating over all matrix elements When exiting from the for statement true is returned if the counter is equal to the matrix size, which means the matrix is symmetric. The algorithm for transposing the matrix is similar to the previous, 2 nested loops and where inside element[i,j] is swapped with element[j,i] in a new matrix. The code is the same for all the three implementation

C. Challenges and Solutions

The first challenge encountered was finding the best instruction for the compiler to optimize the implicit method; the solution I adopted was vectorization and loop unrolling, but with only the flags the code didn't work so I opted to use the flag -O2 that includes both instruction. Second challenge was understand what omp function to use, but this was easier because the library offer the istruction for for statement. To avoid race condition the omp reduction method was the way to parallelise the symmetry check function.

V. Experiments and System Description

A. System and Platform Specifications

The experiments were conducted on the Università di Trento cluster, submitting the job in the short_cpuQ with one node, 64 cpus, 64 threads and 1mb of memory. All this information are also written in the PBS file. The main library used for this project is ¡omp.h¿ that allows to use parallel istruction and taking the execution time, also the ¡vector¿ for creating the matrix.

B. Experimental Setup

All three implementation were benchmarked by transposing square matrices of various sizes, from 16 to 4096, randomically filled with float numbers. Execution walltime was measured using the C++ omp library and every matrix size - number of threads was repeated 10 times for better accuracy.

C. Design of Experiments

Each implementation's performance was evaluated based on execution time and scalability as matrix size increased. Finally speedup and efficiency were calculated for all the omp data based on the serial time.

VI. Results and Discussion

A. Presentation of Results

The first graph represent the run time per matrix size and thread number

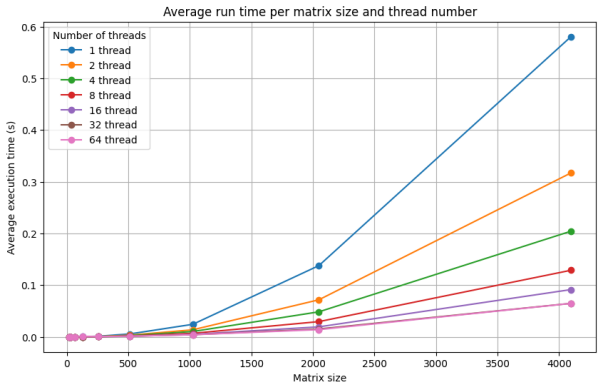

Fig. 1. Execution time

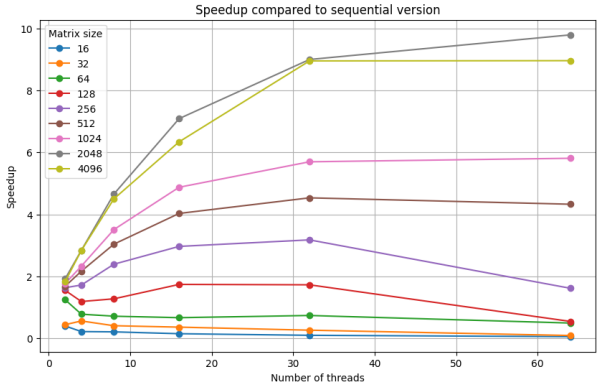

Fig. 2. SpeedUp

it shows that with a small matrix size the execution time is more or less the same, but incrementing that number is clearly that with more threads number the time spent is less.

In fig.2 is represented how the speedup, between the serial code and the omp, which formula is

$$S_p = \frac{T_s}{T_p} \tag{1}$$

where T_s is the execution time of the serial program (1 thread) and T_p is the execution time with p number of threads. One can observe that with small size matrix the speedup is quite linear because the execution time is very low but incrementing the size increase of it we see that the speedup grows pretty fast before the 32 threads than it starts to slow.

Finally there is the graph about the efficiency of the parallel code compared to the serial one.

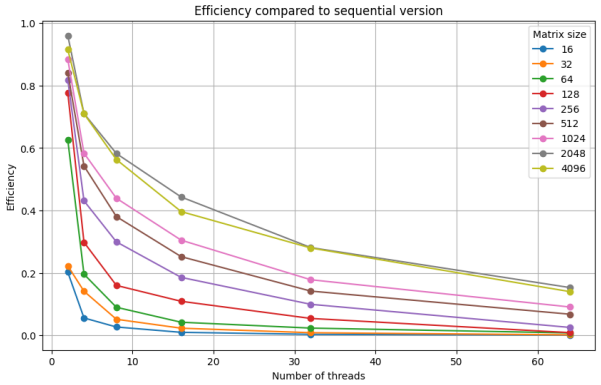

Fig. 3. Efficiency as a function of threads

The formula is

$$E_p = \frac{S_p}{p} = \frac{\frac{T_s}{T_p}}{p} = \frac{T_s}{p \cdot T_p} \tag{2}$$

where S_p is the speedup and p is the number of threads. It is observable that the real efficiency of parallel approach is present when working with big matrix size, in fact before 32 threads we have a good efficiency than it starts to decrease.

The slowdown beyond 32 threads is likely due to memory bandwidth limitations, as the cache becomes a performance bottleneck.

VII. Conclusions

This project successfully demonstrated the effectiveness of OpenMP for matrix transposition, achieving a peak speedup of 20x for large matrices and high thread counts. While implicit parallelism provided moderate gains, it lacked scalability due to compiler limitations. Explicit OpenMP parallelization emerged as the optimal choice for shared-memory systems. Future work could explore hybrid strategies, integrating tiling with advanced runtime schedulers or GPU acceleration to address diminishing returns observed with high thread counts.