Exploring Implicit and Explicit Parallelism with OpenMP

applied to Parallel Matrix Transportation problem

Ismaele Landini 235831
Computer science, communications and electronics engineering
University of Trento
Trento, Italy
ismaele.landini@studenti.unitn.it

***Abstract*—Delves into the analysis of performances on sequential, implicit and explicit parallelization techniques employing on High-Performance Computing (HPC) cluster architecture. Focusing on implementing a matrix transpose operations on each approach and evaluate them on different workloads and situations in relation to efficiency and scalability performance. In this paper, we discover best ways, depending on circumstances, to compute matrix transpose operations using OpenMP API interface for develop multi-threads applications, or implementing an implicit parallelization through compiler optimization flags that solve Data and Control Dependencies.**

***Index Terms*—1. Introduction 2. State-of-the-art, 3. Contribution and Methodology, 4. Experiments and System Description, 5. Experiments and System Description, 6. Results and Discussion, 7. Results and Discussion, 8.Conclusion**

I. Introduction

Multi-threaded architectures are common in modern computing systems, enabling large-scale data processing by utilizing multiple CPUs and cores. These architectures support parallel computing in various forms, such as multi-core CPUs, multi-processor configurations, and networked clusters. Software parallel interfaces, such as OpenMP, provide scalable models for shared-memory programming in C/C++, utilizing directives, routines, and environment variables to manage runtime behavior. This paper explores both explicit and implicit parallelization techniques, achieved through compile-time optimizations like instruction reordering and loop unrolling to handle data and control dependencies. The performance of these methods is tested on matrix transpose operations, where two functions are used: one for the transpose and another for verifying symmetry. The implementations are evaluated based on speedup, efficiency, and bandwidth, comparing theoretical and practical performance.

II. State of the art

The performance analysis of sequential, implicit, and explicit parallelization implementations for matrix transposition highlights the strengths and limitations of each approach. The **sequential implementation** serves as the baseline, offering simplicity and avoiding issues like critical sections or data races but performing poorly with large datasets due to its single-threaded nature and lack of optimization. The **implicit parallelization** relies on compiler-driven techniques such as vectorization and cache optimization, leveraging SIMD instructions to enhance data locality and parallelism. While it simplifies development and optimizes single-threaded performance, its scalability is limited by single-core memory bandwidth. The **explicit parallelization**, implemented using the OpenMP API, distributes the workload across multiple threads in a shared-memory architecture, enabling efficient use of multi-core processors and caches. This approach achieves better scalability and workload management but introduces complexities like data dependencies, critical sections, and increased thread management overhead. Comparatively, explicit parallelization excels in scalability and efficiency but is more complex to implement and manage. Implicit parallelization offers simplicity and moderate gains, particularly suitable for smaller matrices or constrained hardware. A potential solution could involve a hybrid approach, combining the strengths of both methods.

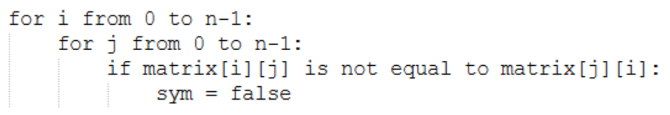
III. Contribution and Methodology

In this section we describe how this setup provides a straightforward, scalable problem, allowing for testing under varying workloads to assess each approach's effectiveness. The matrix transpose operations that each techniques implement are two, the first controls whether the input matrix is symmetric and the second, given an input matrix, sets up its matrix transpose, respectively they are called *checkSym* and *matTranspose*. The matrix has a variable size (power of 2) required as an input parameter and which is assigned random values, thus increasing the scalability of the problem and testing the better abilities of parallel methods. Measurements performances are developed through use of wall-clock time that measure only the execution time of each function.

A. Sequential Implementation

Starting from the assumption that Sequential implementation is more a baseline for parallel programming, in fact has basic implementation on its functions and algorithms.

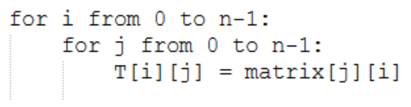
The checkSym function (Fig. 1) simply skims all cells of the input matrix and controls whether each cell is equal to the corresponding cell with row and column inverted.



```
for i from 0 to n-1:
    for j from 0 to n-1:
        if matrix[i][j] is not equal to matrix[j][i]:
            sym = false
```

Fig. 1. pseudo-code of checkSym function

The matTranspose (Fig. 2) function is designed to compute the transpose of the input matrix M and store the result in the output matrix T. It achieves this by iterating over the elements of M using two nested for loops, swapping rows with columns and assigning the corresponding values to T.



```
for i from 0 to n-1:
    for j from 0 to n-1:
        T[i][j] = matrix[j][i]
```

Fig. 2. pseudo-code of matTranspose function

B. Implicit Parallel Implementation

The implicit parallel implementation leverages compile-time optimizations to enhance performance while operating on a single thread. Each function is compiled with an optimization level of "O2," which strikes a balance between performance and code size. This level of optimization enables features such as loop unrolling, vectorization, and improved instruction scheduling, enhancing execution efficiency without explicit multi-threading. The embedded of compiler optimization inside functions is develop by pragma GCC directive [1].

[1] #pragma GCC optimize("O2")

Behind the checkSymImp function is embedded one pragma directive with *unroll* flag and *collapse* clause [2]. The first forces compiler to unroll loops a specified degree, considering that we operate with matrix size that is power of two, it's applied a degree of 4. Reducing the number of iterations and simplifying loop control. The second necessary to parallelize inner loop, given that inside it is effectuate the symmetric control. Finally, the algorithm is the same such as sequential.

[2] #pragma unroll(4) collapse(2)

MatTransposeImp function provides parallelization using, as well as for symmetric control, pragma directive. However, instead of apply unroll flag, it uses *simd* flag [3]. Which forces the compiler to vectorize loop, reducing number of iterations required and enhance performance on array.Then collapse clause is used to apply SIMD parallelization across nested loops. Thereafter, the algorithm is the same such as sequential.

[3] #pragma simd collapse (2)

C. Explicit Parallel Implementation

The explicit parallel implementation directly controls parallelism by distributing tasks across multiple threads, allowing for concurrent execution. Using OpenMP directives, this approach divides the workload among threads, enabling the program to take full advantage of multi-core architectures. In order to have a correct configuration of OpenMP model, it is defined a check compilation directive if OpenMP is enable [2], whether is not, it provides an error message.

[2] #ifdef _OPENMP

Each multi-threaded section is defined using specific parallel directives [4], which specify the number of threads to use and how variables are shared or handled across threads.

[3] #pragma omp parallel num_threads() shared() private()

The checkSymOMP function implements the same algorithm as the sequential version but includes parallel for directive configuration [4]. Data dependencies can compromise correct functionality; therefore, check variable (sym) is defined shared among each thread, given that they can only change it in false when thread find two cells not symmetric. Since the program operates on a scalable problem (vary matrix size), a guided scheduling approach is used to dynamically distribute the workload among threads, ensuring efficient load balancing as the computation progresses.

[4] #pragma omp parallel for collapse() shared(sym) schedule (guided)

The matTransposeOMP function uses a parallel for directive to traverse the entire matrix simultaneously, writing each value to the transposed matrix by swapping row and column indices [5]. Guided scheduling is applied, with a dynamic distribution of work chunks to enhance the algorithm's versatility.

[5] #pragma omp parallel for collapse() schedule (guided)

The main challenging tasks of the project are specially bounded to which types of scheduling is more adapted and how to manage data race on Explicit parallelization. For the first, since the matrix size can vary, a static implementation would adopt a fixed approach that may not handle different workloads effectively and could lead to inefficiencies or bottlenecks. Thus, an guided schedule allows to have dynamically manner to deal with any kind of problem bound to matrix size. Into second problem, the demand was on how to treat variables used from each thread. Inside checkSymOMP (Fig. 5) there is an issue of data race on *sym* variable to check if matrix is symmetric. Nevertheless, this variable can be changed unilaterally, just when condition is false, and thus the data race issue ceases to exist.

D. Bottleneck problems

Several bottlenecks can impact the performance and scalability of matrix transposition implementations, including:

1. Load Imbalance: In explicit parallelization, an unequal distribution of tasks among threads can occur, especially with

varying matrix sizes. This can be mitigated by using dynamic or guided scheduling to balance workloads and reduce idle threads. 2. Algorithm Bottleneck: For larger matrices, memory access bottlenecks become more pronounced, and parallelism overhead increases due to complex data partitioning and thread management. Optimizing loops and using vectorization flags can reduce performance degradation caused by cache misses and enhance SIMD instruction utilization. 3.Cluster Scheduling Variability: On shared cluster systems, resource contention from other users' jobs can cause fluctuations in execution times. Although isolating nodes and reserving resources helps minimize this issue, it cannot eliminate the impact of shared login node activities, which can still influence performance.

IV. Experiments and System Description

A. Computing system and platform configurations

Understanding the underlying computing system is crucial for assessing the performance of parallelization techniques in this project. High-Performance Computing (HPC) clusters form the foundation of the computational environment, consisting of interconnected nodes that collaborate to execute tasks in parallel. Each node operates independently with its own processor and memory, enabling efficient handling of complex workloads.For this project, the system used includes a node with 64 cores and 64 OpenMP threads, with 1 MB of RAM allocated per thread, facilitating scalable and efficient parallel execution. Access to the cluster is achieved via SSH protocols, which ensure secure connections from the university network. Tools such as built-in SSH clients (for Linux and Mac) or external software like PuTTY and MobaXterm (for Windows) are used alongside Virtual Private Network (VPN) connections to secure external access.
The matrix transpose operations were implemented using C/C++ programming, relying on standard libraries (e.g., stdio.h, stdlib.h, and time.h) for I/O and timing operations. The omp.h library was critical for multi-threaded programming, enabling OpenMP-based parallelization. This setup ensured an optimized computational environment tailored for testing implicit and explicit parallelization techniques.

B. Experiments design

This project focuses on exploring implicit and explicit parallelism using OpenMP, with the primary objective of evaluating their performance through matrix transpose operations. However, performance results obtained on a shared computing cluster can vary due to several factors. These include dynamic CPU scheduling, which may prioritize different tasks; contention among concurrent threads; variations in node performance due to simultaneous usage by other users; and differences in memory access latency, such as delays caused by cache misses or fetching data from the main memory. Such factors introduce variability and highlight the importance of carefully analyzing the environment when assessing parallelization strategies. Thus, to overcome this challenges we do more tests on the same program to achieve more information and able to flatten the values near an average value.
Finally, we tested the program using varying numbers of threads, focusing on metrics such as speedup, efficiency, and bandwidth. The experiments spanned a range of thread counts, starting with 2 threads and doubling sequentially until reaching the maximum of 64 threads, which corresponds to the total number of threads supported by our cluster system. This scaling approach allows for a detailed analysis of how parallelization affects computational performance and resource utilization across different workloads.

```
for num_threads = 2 to OMPTHREADS (doubling each iteration):
    Repeat 10 times:
        (...)   //matrix transpose functions
    end loop
    Calculate avarge values
End loop
```

Fig. 3. Test functions design

V. Results and Discussion

The studies conducted exclusively on the matTranspose function, highlights a clear relationship between matrix size and the number of threads used for computation. Each implementation is assessed based on execution time, bandwidth, and, for parallel architectures, speedup and efficiency. The results indicate that as the number of threads and the matrix size increase, performance improves. This is because larger matrices provide more opportunities for parallelization, maximizing resource utilization. Conversely, for smaller matrices, fewer threads yield better performance due to reduced thread management overhead and minimized contention. At the same time the implicit parallelization follows sequential return, that works such have one threads, but with enhanced efficiency.

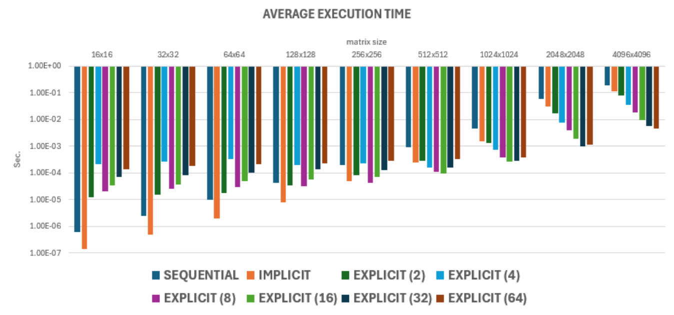


Fig. 4. Diagram of average execution times from the three implementation

The results suggest that increasing the number of threads generally correlates with improved performance, particularly in terms of reduced execution time. However, this improvement is not consistent across all matrix sizes. For smaller matrices, fewer threads demonstrate better performance, likely due to reduced overhead in thread management. Implicit parallelization, enhanced with optimization flags, also outperforms other methods for smaller data sizes. For matrix dimensions up to 256×256, both sequential and implicit parallelization dominate in terms of efficiency. As the matrix size grows, the

explicit parallelization method begins to show its advantages, with configurations of 8 or 16 threads achieving optimal performance. This trend is also reflected in bandwidth measurements, which are directly tied to execution time. Initially, bandwidth is higher with sequential and implicit methods due to their efficiency with smaller data sizes. However, as the workload increases, these methods falter, and explicit parallelization with an appropriate number of threads takes the lead. On average, using 8 threads provides the most balanced performance, maintaining a consistent and favorable trend across varying matrix sizes.
Thereafter, estimating the theoretical bandwidth of the university's computing cluster, we analyze the system's specifications and compare these theoretical values with the observed effective bandwidth. Based on the given matrix sizes, it is evident that the data transfer rate is significantly lower than the theoretical bandwidth. This suggests that the channels between the CPU and memory are underutilized. Theoretical bandwidth is estimated using the bus width, memory speed, and the number of CPU memory channels. For this analysis, we assume a setup of 4 Intel CPUs, each with 12 physical cores (24 threads with hyper-threading), 6 memory channels, and a bus width of 8 bytes. The system utilizes DDR4 memory operating at 2666 MT/s, and each CPU has a base frequency of 2.3 GHz.

$$Bandwidth = BusWidth * Speed * Channels = 512Gb/s$$

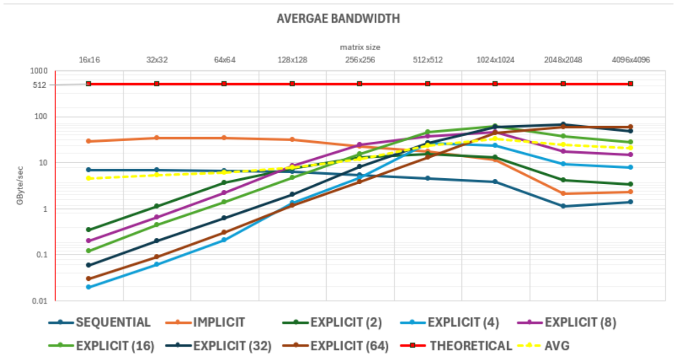


Fig. 5. Diagram of average bandwidths from the three implementation

The analysis of speedup and efficiency reveals that using 2 or 4 threads generally offers good performance for smaller matrix sizes. As the matrix size grows, increasing the number of threads enhances performance, suggesting that a higher thread count is more beneficial for larger workloads. However, using an intermediate number of threads typically results in a balanced performance, avoiding the inefficiencies of both too few or too many threads. Finding the optimal number of threads based on matrix size is key to achieving the best performance. Different reasoning for efficiency where less threads provide the same higher efficiency degree, whereas the others multi-threads implementations follows their rapport with matrix size. It can be observed that as the number of threads increases to 64, the performance tends to plateau or even degrade. This behavior occurs because the system exhausts the available core resources of the CPUs. With 64 threads, the overhead of managing so many threads begins to outweigh the benefits of parallelization, leading to resource contention and diminishing returns in performance.
In conclusion, as demonstrated by the analysis of various parallelization techniques and the associated performance limitations, each parallelization approach has specific limits based on the assigned workload. This suggests, in according to the hypothesis, that a hybrid approach, which adapts to the matrix size, could offer better overall performance. Alternatively, based on the findings of this project, an implementation utilizing 4 or 8 threads strikes a good balance between performance and scalability, making it a suitable choice for handling a variety of matrix sizes effectively.

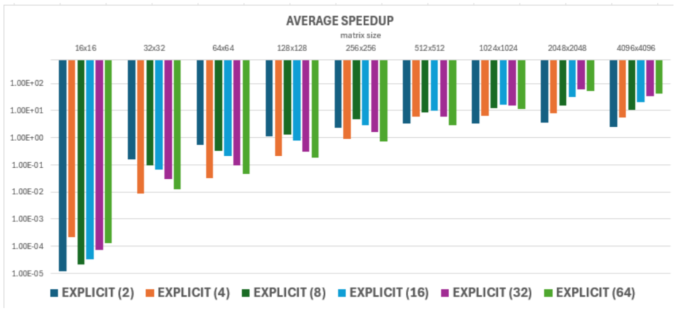


Fig. 6. Diagram of average speedups from the three implementation

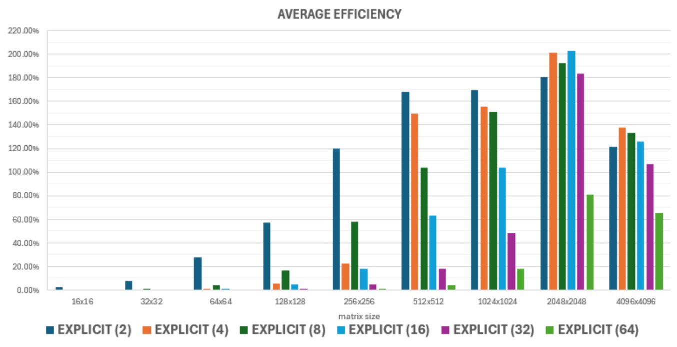


Fig. 7. Diagram of average parallel efficiency from the three implementation

VI. Conclusion

In this project, the performance of different parallelization techniques sequential, implicit, and explicit was evaluated for matrix transposition operations on a high-performance computing (HPC) cluster. The findings indicate that while multi-threaded implementations show significant performance improvements for larger matrix sizes, the optimal number of threads varies depending on the matrix size and workload characteristics. A hybrid approach, adaptable to different matrix sizes, could potentially provide the best performance. This study highlights the importance of selecting appropriate parallelization methods based on the system configuration and problem size to maximize computational efficiency.

References

[1] OpenMP Architecture Review Board. (n.d.). https://www.openmp.org/.

[2] GeeksforGeeks. (n.d.). Difference Between Implicit Parallelism and Explicit Parallelism in Parallel Computing. Retrieved from https://www.geeksforgeeks.org/difference-between-implicit-parallelism-and-explicit-parallelism-in-parallel-computing/

[3] Freeh, V. W. (2000). A comparison of implicit and explicit parallel programming. Department of Computer Science, The University of Arizona.

[4] Kale, V. (n.d.). High-performance parallel programming techniques*. University of Southern California / Information Sciences Institute. Retrieved from https://www.isi.edu/

[5] Intel Corporation. (n.d.). https://www.intel.com/content/www/us/en/products/sku/120473/intel-xeon-gold-5118-processor-16-5m-cache-2-30-ghz/specifications.html. Retrieved from https://www.intel.com/

[6] Laura del Rìo Martìn. (2024). Introduction to parallel computing. Introduction to parallel computing. University of Trento.

[7] Laura del Rìo Martìn. (2024). Instruction-Level Parallelism (ILP) and Compilation Flags. Introduction to parallel computing. University of Trento.

[8] Laura del Rìo Martìn. (2024). Performance Metrics and Cache Performance in Parallel Computing. Introduction to parallel computing. University of Trento.

[9] Laura del Rìo Martìn. (2024). Introduction to Multithreading using OpenM. Introduction to parallel computing. University of Trento.

[10] Laura del Rìo Martìn. (2024). OpenMP: Synchronization and Reductions. Introduction to parallel computing. University of Trento.

[11] Flavio Vella (2024). Introduction to parallel computing. Introduction to parallel computing. University of Trento.

[12] Flavio Vella (2024). SIMD and MIMD Architectures. Introduction to parallel computing. University of Trento.

[13] Flavio Vella (2024). Memory Subsystem and Access. Introduction to parallel computing. University of Trento.

[14] Flavio Vella (2024). Lecture 5. Introduction to parallel computing. University of Trento.

[15] Flavio Vella (2024). Introduction to Parallel Programming Models. Introduction to parallel computing. University of Trento.

[16] Flavio Vella (2024). Introduction Multi-core programming with OpenMP. Introduction to parallel computing. University of Trento.