Report for D1

1st Bilal Soussane
DISI UniTN
Trento, Italy
bilal.soussane@studenti.unitn.it

***Abstract*—Matrix transposition is a fundamental operation in various scientific and engineering applications, often serving as a critical component in algorithms for image processing, machine learning, and numerical simulations. This report investigates the performance optimization of matrix transposition through three different implementations: a sequential method, an implicit parallelism approach leveraging compiler optimizations, and an explicit parallelism technique using OpenMP directives. I evaluated and compared these methods across varying matrix sizes and thread counts on an Intel Xeon Gold 6252N system with 96 cores. The findings demonstrate that the explicit OpenMP implementation, optimized with blocking and loop unrolling (omp2), offers significant speedup over the sequential and implicit methods for large matrices, achieving up to 22 times speedup with 32 threads.**

I. Introduction

Matrix transposition involves rearranging a matrix such that the element at position (*i,j*) in the original matrix moves to position (*j,i*) in the transposed matrix. This operation is essential in numerous computational algorithms, including linear algebra routines, image processing and data analysis tasks. Efficient matrix transposition becomes increasingly important as the size of datasets grows, impacting the performance of high performance computing applications.

Traditional sequential implementations of matrix transposition may not fully utilize the capabilities of modern multicore processors, leading to suboptimal performance for large scale problems. Parallel computing techniques offer a pathway to enhance performance by distributing computational workloads across multiple processing units. This report aims to investigate and compare three implementations of matrix transposition:

1) Sequential Method: a standard implementation using nested loops without parallelization.
2) Implicit Parallelism: leveraging compiler optimizations and vectorization without explicit parallel directives.
3) Explicit Parallelism: utilizing OpenMP directives to explicitly parallelize the computation across multiple threads, with further optimizations such as blocking and loop unrolling.

The objective is to identify the most efficient implementation and understand how different optimization strategies impact performance across various matrix sizes.

II. State-of-the-Art

Matrix transposition is a fundamental operation in high-performance computing applications, including scientific simulations, image processing, and numerical linear algebra. The efficiency of this operation is crucial, especially as data sizes continue to grow. The primary challenge in optimizing matrix transposition lies not in computational complexity but in memory access patterns and bandwidth limitations. Modern processors are significantly faster than memory subsystems, creating a bottleneck where the time to move data outweighs the time to compute.

Researchers have explored various optimization techniques to address these challenges. Cache blocking improves data locality by dividing matrices into smaller sub-blocks that fit into the cache memory. By processing these blocks individually, the number of cache misses is reduced, minimizing costly data transfers between different levels of the memory hierarchy [1]. Loop unrolling decreases the overhead of loop control instructions and increases instruction-level parallelism. By expanding the loop body, the compiler can better optimize the code, leveraging superscalar execution and reducing branching penalties [2].

Vectorization utilizes Single Instruction, Multiple Data (SIMD) instructions available in modern CPUs to perform operations on multiple data points simultaneously. Compilers can automatically vectorize code under certain conditions, but explicit use of intrinsic functions or compiler directives can enhance the utilization of SIMD capabilities [3]. Parallel computing techniques further improve performance by distributing computations across multiple processing units. Implicit parallelism relies on the compiler to automatically parallelize loops and functions, but its effectiveness is limited by the compiler's ability to analyze code and predict data dependencies [4]. Explicit parallelism, using threading libraries like OpenMP, allows developers to manually control the distribution of work across threads, leading to significant performance improvements when managed effectively [5].

Despite these advancements, existing solutions face limitations. Cache blocking's effectiveness is sensitive to block size and may not yield significant benefits on architectures with large caches or for small matrices [6]. Manual loop unrolling can lead to code bloat and reduced maintainability, and modern compilers often perform automatic unrolling, reducing its benefits [7]. Automatic vectorization may be hindered by data dependencies and complex contrl flow, requiring code restructuring for optimal performance [8]. Implicit parallelism may not fully exploit hardware capabilities due to the compiler's conservative analysis, while explicit parallelism introduces overhead related to thread management and synchronization, potentially offsetting performance gains for smaller problem sizes [9].

The best performing methods to date combine multiple optimization techniques to overcome individual limitations. An approach integrating cache blocking, loop unrolling, vectorization, and explicit parallelism using OpenMP has demonstrated significant speedups on modern multi-core processors [10]. By enhancing data locality, reducing loop overhead, leveraging SIMD instructions, and efficiently distributing workloads across threads, this comprehensive strategy addresses both computational and memory access challenges. Performance evaluations show that such optimized implementations can achieve near-linear speedup with increasing core counts on large matrices, outperforming naive parallel implementations and those relying solely on compiler optimizations.

III. Contribution and Methodology

Three versions of the matrix transposition algorithm were implemented: a sequential method, an implicit parallelism approach using compiler optimizations and an explicit parallelism technique utilizing OpenMP directives with additional optimizations. The goal was to evaluate and compare these methods to identify the most efficient one for large matrices.

A. Sequential Implementation

The sequential implementation served as the baseline for performance comparison. It employs a straightforward nested loop structure to swap elements across the main diagonal.

B. Implicit Parallelism with Compiler Optimizations

Recognizing the limitations of the sequential approach, especially for larger matrices, the next step involved exploring implicit parallelism through compiler optimizations. Based on insights from current research, after some testing, the block-based method was identified as a promising strategy for enhancing performance. Blocking improves data locality by partitioning the matrix into smaller submatrices or blocks that fit into the cache memory. This approach reduces cache misses and minimizes the latency associated with memory accesses. In implementing the block-based method, careful consideration was given to selecting an appropriate block size, which was chosen based on the cache hierarchy of the system to ensure that the blocks could be efficiently loaded and processed in the cache. This required an understanding of the hardware specifications, particularly the size of the L1 cache.

During the development of the implicit method, I experimented with both manual and automatic loop unrolling. Manual loop unrolling involves rewriting loops to reduce the overhead of loop control instructions, potentially increasing instruction-level parallelism. However, the performance measurements indicated that manual unrolling did not offer significant advantages. In some cases, it even led to reduced performance. This counterintuitive result was likely due to the compiler's inability to further optimize the manually unrolled code, as well as potential code bloat and increased instruction cache pressure.

To further investigate the impact of them, the code was compiled using various optimization flags (`-O1`, `-O2`, `-O3`) and some specific optimizations (`-funroll-loops`, `-fprefetch-loop-arrays`, `-ftree-vectorize`, `-march=native`). Each combination of these flags was tested to assess its effect on execution time.

1) Challenges Faced: One of the main challenges encountered was the limited benefit of manual loop unrolling in the implicit method. The expectation was that manual unrolling would reduce loop overhead and enhance performance. However, the compiler's advanced optimization capabilities often rendered manual unrolling inefficient. This highlighted the importance of understanding how compiler optimizations interact with code-level optimizations. Another challenge was determining the optimal set of compiler flags. The multitude of available flags and their possible combinations required systematic testing to identify which ones genuinely contributed to performance gains. This process was time consuming but essential for achieving the best possible results.

C. Explicit Parallelism with OpenMP

Four versions of the explicit method were developed, each introducing additional layers of optimization:

1) **Basic OpenMP Parallelization (`omp`)**: this initial version involved adding OpenMP pragmas to parallelize the outer loops of the transposition operation. It served as a foundation for further enhancements.
2) **OpenMP with Blocking (`omp1`)**: recognizing the benefits of blocking from the implicit method, this version incorporated loop blocking to improve cache performance while maintaining parallelization.
3) **OpenMP with Blocking and Manual Unrolling (`omp2`)**: this version added manual loop unrolling to the blocked parallel loops, aiming to reduce loop overhead and increase instruction-level parallelism.
4) **OpenMP with Blocking, Unrolling, and SIMD (`omp3`)**: the final version included SIMD (Single Instruction Multiple Data) directives to encourage the compiler to vectorize the loops, potentially leveraging the CPU's vector processing capabilities.

The development of the explicit methods involved iterative testing and performance measurement. Each version was carefully implemented, ensuring that the OpenMP pragmas and optimization techniques were correctly applied. Extensive benchmarking was conducted across various matrix sizes and thread counts to evaluate the effectiveness of each method.

1) Challenges Faced: A significant challenge was balancing the complexity introduced by optimizations against their performance benefits. As optimizations increased, so did the complexity of the code, which could potentially lead to difficulties in maintenance and reduced readability. Through rigorous and systematic testing, the effectiveness of each optimization was evaluated. This process was crucial to confirm that the performance improvements justified the added complexity.

Another challenge was minimizing synchronization overhead and ensuring efficient workload distribution among threads. Proper placement of OpenMP directives was crucial to avoid bottlenecks and achieve scalability. Careful attention was paid to data dependencies and potential race conditions to maintain the correctness of the parallel implementation.

IV. Experiments and System Description

A. *Computing Environment*

- **Processor**: 4 × Intel® Xeon® Gold 6252N CPU @ 2.30GHz
- **Total Cores**: 96 (24 cores per socket)
- **Cache Hierarchy**:
  - L1 Cache: 32 KB per core
  - L2 Cache: 1 MB per core
  - L3 Cache: 36 MB per socket
- **NUMA Nodes**: 4
- **Compiler**: GCC version 9.1.0 which supports OpenMP
- **Operating System**: Linux-based HPC environment
- **MobaXterm** was utilized to establish an SSH connection from my local Windows system to the university's cluster
- Standard C libraries and OpenMP directives were utilized

B. *Experimental Setup*

- **Matrix Sizes Tested**: From 2^4 (16×16) to 2^{12} (4096×4096).
- **Implementations Evaluated**:
  - Sequential Method
  - Implicit Method with various compiler flags
  - Explicit Methods (`omp`, `omp1`, `omp2`, `omp3`) with different thread counts
- **Compiler Flags for Implicit Method**:
  - Tested combinations of `-O1`, `-O2`, `-O3`, `-funroll-loops`, `-ftree-vectorize`, `-march=native`, etc.

C. *Experimental Procedures*

Each test involved running the transposition operation multiple times and recording the execution times. To account for variability and ensure statistical significance, 5 runs were performed for each configuration and the average execution time was calculated after removing outliers. Special attention was given to ensuring that all implementations operated on the same node, on identical data, and that memory allocation and initialization were consistent. This approach guaranteed that performance differences were attributable to the implementation strategies rather than extraneous variables.

D. *Experimental design was structured to test these hypotheses:*

- That compiler optimizations could significantly improve the performance of the implicit method, potentially reducing the need for manual code optimizations.
- That the explicit methods, particularly those combining multiple optimization techniques, would outperform the sequential and implicit implementations, especially for large matrices.
- That there would be an optimal number of threads for the explicit methods beyond which performance gains would diminish due to overhead and resource contention.

By systematically varying one parameter at a time (e.g., number of threads, compiler flags), the experiments provided clear insights into the impact of each factor on performance.

E. *Performance Metrics*

- **Execution Time**: Measured the wall-clock time for the transposition operation.
- **Speedup**: Ratio of sequential time to parallel time.
- **Efficiency**: Speedup divided by the number of threads.

V. Results and Discussion

A. *Implicit Method Performance*

The implicit method utilizes a block-based transposition algorithm aimed at improving cache utilization. To assess the impact of compiler optimizations, the code was compiled with different combinations of optimization flags. The execution times for varying matrix sizes and the best performing combinations of compiler flags are presented in **Table 1**.

c

Flags	n=10	n=11	n=12
noflags	0.0035558	0.0170424	0.0721600
-O1	0.0019304	0.0127038	0.0676564
-O2	0.0018430	0.0148564	0.0702352
-O2-funroll-loops-march=native	0.0018138	0.0137162	0.0720610
-O2 -funroll-loops	0.0018626	0.0148782	0.0726385

TABLE I
EXECUTION TIMES OF IMPLICIT METHOD (TIME IN SECONDS)

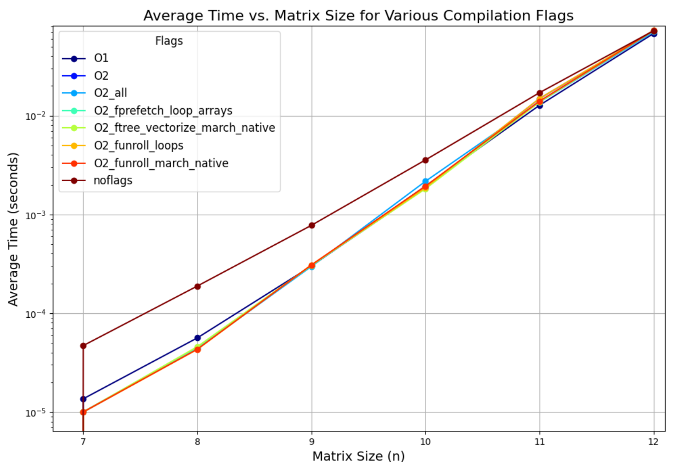

Fig. 1. Comparison of various compiler flags from n=7 to n=12

1) *Observations:*

1) **Significant Performance Improvement with Optimization Flags**: As observed from Figure 1, compiling with optimization flags drastically reduces execution time compared to no optimization (`noflags`), especially with smaller matrix sizes.

2) **Comparison Between `-O1` and `-O2`**: At larger matrix sizes ($n = 12$), `-O1` slightly outperforms `-O2` variants. On the other hand, for smaller to medium matrix sizes ($n = 7$ to $n = 10$), `-O2` and its variants perform better. This suggests that the less aggressive optimization strategies of `-O1` may better align with the manual optimizations in the code, avoiding over-optimization which can sometimes degrade performance.
3) **Effectiveness of Specific Flags**:
   - The addition of `-funroll-loops` and `-march=native` to `-O2` marginally improves performance at smaller matrix sizes.
   - The differences among `-O2` variants are minimal, suggesting limited impact from these additional flags for this specific code and hardware.

2) Conclusions Implicit Method: Based on these observations, adopting the `-O2 -funroll-loops -march=native` flags is recommended as a standard practice for achieving balanced performance across a range of matrix sizes. For scenarios where the absolute maximum performance is critical, particularly with very large matrices, the `-O1` flag may be more advantageous, providing a tailored approach that complements specific manual optimizations.

B. Explicit Method Performance

Four explicit methods were evaluated: `omp`, `omp1`, `omp2`, and `omp3`.

a) Comparison Among Methods: `omp1` performs well at lower thread counts but is overtaken by `omp2` as threads increase. `omp3` does not provide consistent benefits, possibly due to overhead from SIMD directives not being fully exploited.

b) Overall Performance Analysis: `omp2` consistently outperforms the other three methods across most matrix sizes and thread counts, especially for larger matrices and when using an optimal number of threads (32 threads), as noted by examining the data. Increasing the thread count beyond 32 does not yield better performance and may introduce overhead. At smaller matrix sizes, such as $n = 7$ and $n = 8$, the performance differences are less pronounced, but `omp2` often still holds a slight advantage.

Threads	omp (s)	omp1 (s)	omp2 (s)	omp3 (s)
n=12				
16	0.00318316	0.00222580	0.00220046	0.0023195
32	0.0024558	0.0021906	0.0020336	0.0022946
64	0.0076668	0.0103394	0.0040275	0.086220
n=11				
16	0.0141394	0.0072689	0.0069544	0.0113810
32	0.0069517	0.0055595	0.0051207	0.0069140
64	0.0086378	0.0125392	0.0064570	0.0291900

TABLE II
T2: EXECUTION TIMES AT N=12 AND N=11

VI. SPEEDUP AND EFFICIENCY ANALYSIS

The speedup and efficiency of the OpenMP implementation (explicit method without using blocking: `omp`) were computed relative to the sequential baseline (serial method).

- Speedup $= \dfrac{T_{\text{serial}}}{T_{\text{parallel}}}$
- Efficiency $= \dfrac{\text{Speedup}}{\text{Number of Threads}}$

Threads	Serial Time (s)	Parallel Time (s)	Speedup	Efficiency
2	0.1119592	0.0611780	1.83	0.915
4	0.1119592	0.0312528	3.58	0.895
8	0.1119592	0.0163870	6.84	0.855
16	0.1119592	0.00908625	12.35	0.772
32	0.1119592	0.00620088	18.03	0.563
64	0.1119592	0.0055408	17.11	0.267
96	0.1119592	0.0296340	3.78	0.039

TABLE III
T3: SPEEDUP AND EFFICIENCY OF OPENMP IMPLEMENTATION AT N=12

1) Observations: In examining the performance metrics of multi-threaded processes, it is observed that speedup increases with thread count up to 32 threads, achieving a maximum speedup of 18.03. However, speedup tends to decrease beyond this point due to overhead. Additionally, efficiency is highest at lower thread counts, for example, 0.915 at 2 threads, and there's a notable decrease in efficiency at higher thread counts, such as 0.039 at 96 threads. This suggests a performance degradation at high thread counts; specifically, execution time increases when using 96 threads compared to 32 threads, indicating that the overhead associated with higher thread counts can outweigh the benefits.

VII. CONCLUSIONS

This study demonstrates that an explicit OpenMP implementation of matrix transposition, optimized with blocking and loop unrolling (`omp2`), significantly outperforms both the sequential and implicit methods for large matrices.

Key findings include:

- **Optimal Performance**: Achieved with `omp2` using 32 threads, yielding up to 22 times speedup over the sequential implementation.
- **Compiler Optimizations**: Leveraging compiler flags in the implicit method improved performance but did not match the explicit method's efficiency.

The analysis also underscores the need to consider hardware limitations and overhead when scaling parallel applications. Optimal performance is achieved by balancing the number of threads and utilizing effective optimization strategies.

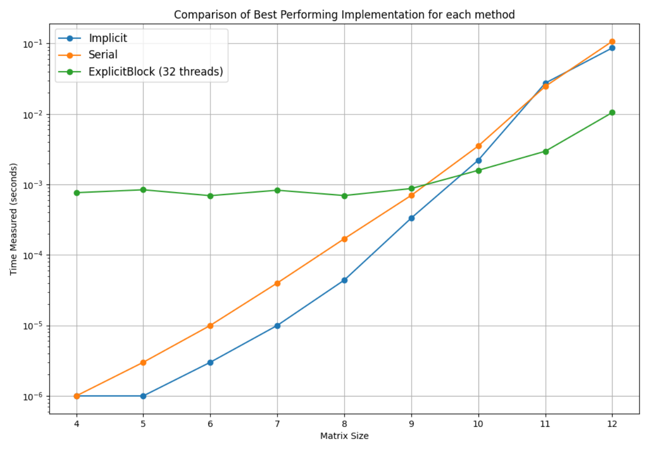

Fig. 2. Serial vs Implicit (with Blocking) vs Explicit (omp2 32 threads)

REFERENCES

[1] Lam, M. S., Rothberg, E. E., & Wolf, M. E. (1991). “The cache performance and optimizations of blocked algorithms”. *ACM SIGARCH Computer Architecture News*, 19(2), 63-74.

[2] M. Booshehri, A. Malekpour, and P. Luksch, ”An Improving Method for Loop Unrolling,” *International Journal of Computer Science and Information Security (IJCSIS), vol. 11, no. 5, pp. xx-xx*, May 2013.

[3] Maleki, S., Gao, W., Garzarán, M. J., Wong, T., & Padua, D. (2011). An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques* (pp. 372-382).

[4] P. Bone, Z. Somogyi, and P. Schachte, ”Estimating the overlap between dependent computations for automatic parallelization,” *The University of Melbourne and National ICT Australia (NICTA)*, pp. xx-xx, (2011).

[5] Dagum, L., & Menon, R. (1998). OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), 46-55. DOI: 10.1109/99.660313

[6] Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. (1999). Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science* (pp. 285-297). IEEE. DOI: 10.1109/SFFCS.1999.814600

[7] J. W. Davidson and S. Jinturkar, ”Aggressive loop unrolling in a retargetable, optimizing compiler,” in *Compiler Construction, T. Gyimóthy, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg*, 1996, pp. 59–73, DOI: 10.1007/3-540-61053-7_53.

[8] Larsen, S., & Amarasinghe, S. (2000). Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (pp. 145-156). DOI: 10.1145/349299.349320

[9] S. A. Haque, M. Moreno Maza, and N. Xie, ”A Many-core Machine Model for Designing Algorithms with Minimum Parallelism Overheads,” 2014.

[10] Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4), 65-76. DOI: 10.1145/1498765.1498785