Parallel Computing course First Midterm

Martina De Piccoli
235165
martina.depiccoli@studenti.unitn.it

***Abstract*—Nowadays, the matrix operations are widely used in various scientific and engineering applications, especially in data processing for artificial intelligence (AI) and machine learning systems. Therefore, with the continuous and rapid development of AI, it has become crucial to find the algorithms that best fit new architectures and results in shorter execution times for these applications.**

Efficient implementations of these operations are critical for high-performance applications. This project focuses on matrix transposition and symmetry check, which are computationally intensive for large matrices. Two main methods are analyzed in this work: naive and block-based approaches.

After evaluating the performance of the two algorithms, they are parallelized, and both implicit and explicit parallelization techniques are applied. In conclusion, this study aims to provide insights into the effectiveness of parallelization techniques for matrix transposition and check symmetry operations, comparing their efficiency, scalability, and overall impact on computational performance.

I. Introduction

Matrix transposition operations are widespread across scientific fields such as chemistry, physics, and economics [1] and in more technology-driven areas like image processing, cryptography, and AI [2]. In addition, modern computers are optimized for efficient row access in row-major 2D matrices due to their linear memory addressing, complicating column access to column-major 2D matrices and a common strategy to simplify operations on column-major matrices, is to transpose the matrix beforehand, centralizing the complexity within the transposition operation [1]. Consequently, optimizing matrix transposition is essential for enhancing performance, reducing execution times and improving cache utilization.

This research investigates two algorithms to perform *out-of-place* matrix transposition to square matrices: the naive and the block-based (BB) approach. After evaluating the most efficient algorithm formerly mentioned, parallelization techniques are applied and assessed, both implicit with GNU Compiler Collection's flags and explicit with OpenMP directives.

Moreover, since a symmetric matrix is defined as a square matrix that is equal to its own transpose [3], it may be beneficial to verify its symmetry in advance to avoid performing a transposition over a matrix that remains unchanged, hence, the methods mentioned above are also applied to a check symmetry function. Eventually, this study presents a benchmark analysis comparing the three approaches: sequential, implicit and explicit parallelization for the transposition.

II. State-of-the-art

The matrix transposition problem has been extensively tackled by previous research due to its importance in high-performance computing. The straightforward implementation consists in swapping each row with the corresponding column. This leads to poor data locality and doesn't optimize accesses, leading to a large amount of costly *read/write* operations for large matrices with a O(N^2) complexity, where N is the size of a square matrix. One revolutionary way of viewing the transposition has been first introduced by Eklundh [4]. Instead of using a sequential access pattern, Eklundh's algorithm performs *in-place* matrix transposition in different stages: two rows are read from memory and the appropriate elements are interchanged and stored back in the external memory. This transposition approach takes N number of stages to complete, reducing the memory accesses and complexity, becoming O(Nlog(N)), but requires that at least two rows fit in memory. Extension to the later algorithm were presented by Kaushik et al. [5] that reduces the number of *read* operations. Further implementations of matrix transposition were presented by Suh and Prasanna [6] who leveraged *collect* buffers to temporarily store permutations allowing writings to the memory to be performed all at once.

This work pertains to the *block-based* class of solutions, whose idea is to divide the matrix in block and to transpose them individually. Differently from existing approaches, to gain better performance and cache usage, implicit and explicit parallelization techniques are used and benchmarked. Implicit parallelization involves compiler optimizations and cache-aware memory access patterns to improve performance without explicit multithreading, while explicit parallelization techniques like OpenMP enable multi-threading for significant speedup, however they require careful tuning for load balancing and avoidance of race conditions.

III. Contribution and Methodology

In this section is presented an overview of two algorithms that exploit two different memory access patterns, and their optimization through parallelization techniques. Algorithms for checking symmetry follow essentially the same structure of the transposition ones: the inner assignment instruction is substituted by a comparison between element m_{ij} and m_{ji} of matrix M, and whether they differ, a boolean variable is set to false.

Note that this study focuses on random float matrices, hence the probability that m_{ij} and m_{ji} are equal is very low, therefore the results of the boolean variable is returned only after the loops are done, otherwise it serves no purpose to parallelize loops that are immediately broken after the first false result.

A. *Naive Implementation*

Given *n x n* matrix M, the naive approach consists of the rearrangement of the elements so that each row is transformed into a column, that is element m_{ij} swapped with element m_{ji}.

The procedure for the transposition operation follows the Algorithm I shown below:

```
// ---- algorithm I ----
void naiveTransp(int** M, int** T, int size){
   for(int i = 0; i < size; i++)
      for(int j = 0; j < size; j++)
         T[j][i] = M[i][j];
}
```

B. *Block-based Implementation*

Given *n x n* matrix M, the block-based approach consists of dividing the matrix into four quadrants, and transpose each of them independently. The procedure for the transposition operation follows the Algorithm II shown below:

```
// ---- algorithm II ----
void blockTransp(int** M, int** T, int
    matrix_size, int block_size){
   for (int i=0; i<size; i+=block_size) {
      for (int j=0; j<size; j +=block_size) {
         for (int bi=i; bi<i+block_size &&
             bi<size; bi++) {
            for (int bj=j; bj<j+block_size &&
                bj<size; bj++) {
               T[bj][bi] = M[bi][bj];
}
```

C. *Algorithms analysis*

The way memory and cache are accessed is a determining factor and can significantly affect the bandwidth. As it was previously stated, the BB approach is the most efficient one in terms of exploitation of temporal and spatial locality and avoidance of unnecessary data loads [7], in fact, although no remarkable variation are observed in the reading step of the two algorithms, the effective improvement lays on the writing part. In row-major order, like C language, elements of a row are stored contiguously in memory, while elements of a column are strided. Therefore when transposing matrix M into T, the matrix M is accessed row-by-row, namely contiguously, whereas T is accessed column by column, resulting in non-contiguous memory access, causing frequent cache misses in the naive approach because each element of the column might reside in a different cache line. In BB approach, although elements are still written into columns, the block ensures that memory access remains localized within a small region, minimizing cache misses.

In order to ensure optimal cache usage, cache line and cache level sizes should be know. This issue is analyzed in detail in the 'Experiments') section.

D. *Parallelization Techniques*

The parallelization explored in this works is divided into two parts: implicit techniques, using compiler flags and exploiting instruction-level parallelism (ILP) and explicit techniques, using OpenMP directives.

Implicit parallelization : few code adjustments can result in better CPU usage, for instance, ILP can be exploited to maximize CPU's pipeline [8]. Since the transposition in this report is assumed to be *out-of-place*, *reads* and *writes* occur to different memory locations; therefore, there are no data dependencies between elements, and loop-unrolling techniques can be applied [8]. Other implicit parallelization are done by GNU compiler's flag that are *march=native*, which generates code for the specified processor of the local machine , and *-O* levels enabling, which are a set of optimizations [9].

These improvements are applied only to the block-based approach, since it already has better cache usage than the naive one, hence can itself be considered as implicit parallelization. A snippet of the loop-unrolling inside the BB transposition is shown below. The flags previously mentioned can be specified at compilation or within the code by pragmas or C attributes.

```
// ---- algorithm II implicit ----
... inner loop ...
for(bj=j; bj<j+block_size && bj<size; bj+=2){
   T[bj][bi] = M[bi][bj];
   T[bj][bi+1] = M[bi+1][bj];
   T[bj+1][bi] = M[bi][bj+1];
   T[bj+1][bi+1] = M[bi+1][bj+1];
}
```

Explicit parallelization: this part is tided to the concept of threads that run in parallel on multiple processing element using OpenMP directives [10] and since most modern hardware is multi-core, using multi-threading can significantly reduce computation time [7]. Despite, as already stated, BB is the most efficient access pattern for larger matrices, in this work OpenMP is used for both naive and BB approaches in order to asses the performance enhancement related to the utilization of more processing units; both naive and block-based implementations can be parallelized among threads by using pragma directives before the loops:

```
#pragma omp parallel for collapse(2)
```

The `for collapse` clauses used for nested loops helps avoiding the risk of oversubscription related to nested pragmas by combining multiple iteration spaces into a single one. Although for matrix transposition the parallelization is straightforward using for collapse(2), for the check symmetry function additional considerations must be taken into account when operating with a variable declared before the loops, hence shared among threads. To avoid race conditions, OpenMP offers the `reduction(&&:is_sym)` clause: this creates a copy of the variable `is_sym` for each thread, each thread updates its copy and once they've all completed their work, the final global value of `is_sym` is produced combining all the copies values using logical AND operator (&&).

IV. System Description and Experiments

A. Setup

Following benchmarks were conducted on a Linux system with an x86-64 architecture supporting 64-bit operations and configured with 96 physical CPUs distributed across 4 sockets. Each socket contains 24 cores, with 1 thread per core, enabling efficient parallel execution. The processor used is the Intel(R) Xeon(R) Gold 6252N CPU, it operates at 2.30 GHz and a hierarchical cache design as following: L1d and L1i of 32KB, L2 of 1024KB, and a shared L3 of 36MB cache per socket. The system is NUMA-based with 4 NUMA nodes. This architecture provides an environment for evaluating parallelization strategies, in particular for computationally intensive tasks such as matrix transposition, where cache efficiency, thread distribution, and memory locality are critical for performance.

The compiler installed is GCC 9.1.0

Implicit and explicit parallelization parts are assessed and studied individually; after evaluating the most efficient implementation for both of them, a final benchmark is presented, showing differences and improvements.

B. Block size estimate

As previously mentioned, the computation of a matrix operation can gain better performance by optimizing cache usage. This can be done by simply changing the memory access pattern, for instance by using the BB approach studied in this work. The choice of the block size is strictly related to the the cache of the system employed for the computation. It's usually more convenient to leverage the faster L1 and L2 caches [11], hence, estimation of the optimal block size of *float matrices* for Intel(R) Xeon(R) Gold 6252N CPU can be found below. For out-of-place transposition, both matrices have to be loaded into cache, hence two blocks are needed. For each block iteration, the cache usage will be:

$$\text{Cache Usage} = (B \times B \times 4) \times 2 = 8B^2$$

where B is the block size, 4 is the size of a float in bytes, and 2 accounts for the source and destination matrices.

The L1 cache size is 32 KB. For optimal cache utilization, the block size must satisfy the following condition:

$$8B^2 \leq 32 \text{ KB}$$

Thus, each block should be at most 64×64 elements.

The L2 cache size is 1 MB. For optimal cache utilization, the block size must satisfy the following condition:

$$8B^2 \leq 1 \text{ MB}$$

Thus, each block should be at most 256×256 elements.

TABLE I
MISS-RATIO (MR) FOR EACH (BLOCK SIZE)

matrix	MR (32)	MR (64)	MR (128)	MR(256)
256	11.76%	**11.22%**	-	-
512	1.73%	**1.52%**	1.62%	-
1024	4.53%	7.07%	3.25%	**2.55%**
2048	23.82%	29.13%	28.77%	**22.01%**
4096	83.72%	86.39%	89.34%	**81.76%**

At this point, the best block size can be evaluated through tools like `perf` [12] or `vtune` [13] which can be used to measure cache-miss ratio. Reducing cache-miss ratio generally leads to faster computation times by minimizing slower memory accesses [11]. As can be seen in the table, for matrices 1024x1024 or higher, the block that results in fewer *misses* is 256x256.

C. Implicit parallelization

Now that block size has been assessed, to get better performance from implicit parallelization the inner loop is unrolled (as previously shown in section III), and different flags are used to compile the program.

Note that `-O0` flag doesn't perform any optimization.

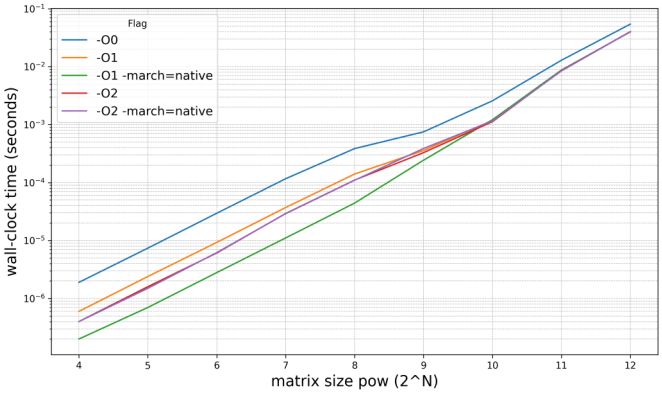


Fig. 1. Block-based loop-unrolled matrix transposition

D. Explicit parallelization

The block access pattern isn't useful only for implicit parallelism and exploiting data locality, it can also serve the purpose of distributing workloads among threads. In this explicit parallelization section, OpenMP directives (explained in section III) have been used to improve both naive and block-based approaches by using more threads.

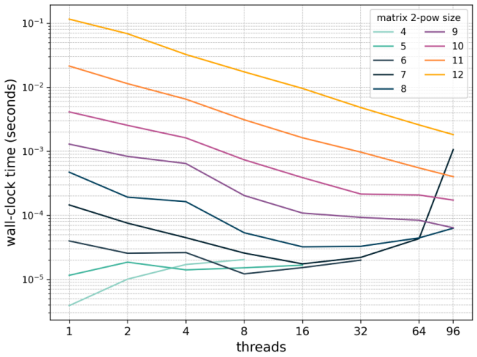


Fig. 2. Naive matrix transposition optimized with OpenMP

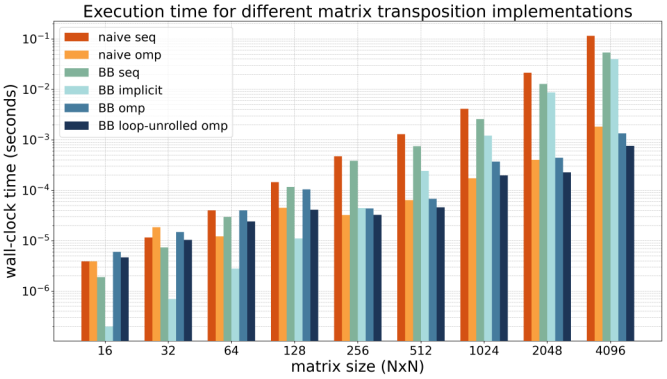
V. Results and Discussion

The observations and experiments conducted on the block size at section IV, allows to assert that:

- The optimal block for exploiting L1 and L2 caches for small matrices is 16x16; for medium ones is 64x64; for matrices equal or bigger than 1024x1024 floats, the favorable block is 256x256.
- The combination of compiler flags that best optimize the transposition, using block-based approach with loop-unrolling, is `'-O1 -march=native'`

- As regards as the number of threads for parallel executions, it is shown, at section IV, that for medium matrices the ideal number of threads is circa $1/8$ of the size of the matrix, while for bigger ones is 96 (which is the processors number for Intel(R) Xeon(R) Gold 6252N). For small matrices (less than 32x32 floats) the transposition results in less execution time for a single thread, meaning they don't benefit from the parallel execution. This can be explained by the overhead associated with managing multiple threads [10]: in fact, creating, synchronizing, and distributing work among threads, requires additional operations and resources, which takes a relatively high amount of time compared to the time required to transpose a small matrix.

In light of the above considerations, a final overall benchmark of naive and block-based matrix transposition, implicitly and explicitly (omp) parallelized, is provided:



Notice that the Y-axis is logarithmic, in order for the plots to be easily readable but this flattens the gaps: a difference of one unit between two bars represents an acceleration of 10×.

TABLE II
SPEEDUP INTRODUCED BY OPENMP TO NAIVE APPROACH

matrix	256	512	1024	2048	4096
Tseq(s)	0.00047	0.001297	0.004134	0.02149	0.11602
Tomp(s)	0.000032	0.000064	0.000173	0.00040	0.00183
Speedup	14.63	20.26	23.87	53.36	63.54
Efficiency	91.44%	21.10%	24.86%	55.58%	66.19%

Table II shows the speedup and efficiency related to the naive approach parallelization using OpenMP, for matrices greater than 256x256 (for smaller matrices we can see from the graph that there 's almost no speedup, or a even slowdown). Efficiency is calculated by choosing the number of threads that is more suitable for each matrix size according to observations previously made.

A. *Peak Performance Comparison*

It is valuable to further assess performance against the theoretical peak memory bandwidth (BW) of the architecture. For this purpose the number of threads used is the one that best fits within matrix size as explained at third dot of 'Results' section. Actual bandwidth can be calculated from the following formula, knowing that transposition requires 2 operations (read/write) and floats occupies 4 bytes:

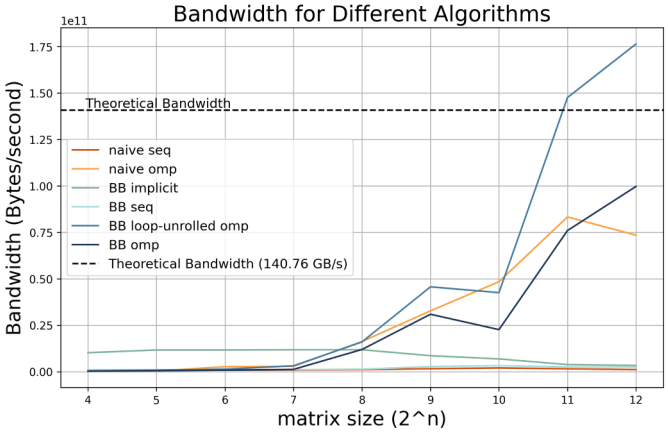
$$\text{Data Transferred} = (\text{Matrix Size})^2 \times 2 \times 4$$

$$\text{Bandwidth} = \text{Data Transferred}/\text{Time Taken}$$

Theoretical BW can be derived from hardware specifications of the architecture by using the formula:

$$\text{BW} = \text{Memory Speed} \times \text{Memory Channels} \times \text{Bus Width}$$

Intel Xeon Gold is equipped with a DDR4-2933 [14] that has a theoretical BW of **140.76GB/s**.



It can be seen that OpenMP parallelizations lead to higher bandwidth for large matrices, in particular the bandwidth related to the loop-unrolled block-based transposition even exceeds the theoretical DDR4-2933 one. This behavior could be hypothetically due to 3 factors:

- BB exploits spatial and temporal locality, avoiding the potential bottleneck of accessing strided memory locations;
- cache usage related to BB algorithm and loop-unrolling: BB was set up to fits in L1 and L2 caches, thus much of the data transfer might not actually involve main memory, especially when using 96 cores (hence 96 L1 and L2 caches);
- concurrent work avoids some waiting to access the memory, which instead the sequential execution is subject to.

VI. CONCLUSIONS

The matrix transposition operation can benefit significantly from implicit and explicit parallelization techniques. Using a block-based access pattern and leveraging Instruction Level Parallelism, can help overcome hardware limitations related to matrix storing, by exploiting spatial and temporal locality. Moreover, parameters can be carefully tuned to align within the cache hierarchy, minimizing cache misses and access to main memory, leading to markedly improvements of the performance.

Explicit parallelization, such as those introduced by OpenMP API, can further enhance execution time, by dividing workloads among multiple threads. This approach is particularly efficient for large matrices, however, for small matrices, the naive sequential approach remains more effective as it avoids the overhead introduced by parallel process management.

In conclusion, for high-intensity operations the combination of parallel processing and cache-aware optimization proves to be the most efficient strategy, whereas for low-intensity tasks, the naive approach remains the most expedient and straighforward solution.

References

[1] A. K. Pandey and P. Gupta. “Applications of Matrices in Modern Scenario”. In: *Journal of Applied Science and Education (JASE)* 1.1 (2021), pp. 1–14. DOI: 10.54060/JASE/001.01.003.

[2] Paul Godard, Vincent Loechner, and Cédric Bastoul. “Efficient Out-of-core and Out-of-place Rectangular Matrix Transposition and Rotation”. In: *IEEE Transactions on Computers* 70.11 (2021). hal-02960539, p. 7. DOI: 10.1109/TC.2020.3030592.

[3] Gilbert Strang. *Linear Algebra and Its Applications*. 4th. See page 66 for the definition of symmetric matrices. Boston: Cengage Learning, 2016.

[4] J.O. Eklundh. “A Fast Computer Method for Matrix Transposing”. In: *IEEE Transactions on Computers* C-21.7 (1972), pp. 801–803. DOI: 10.1109/T-C.1972.223584.

[5] S.D.Kaushik C.H.Huang R.W.Johnson P. Sadayappan J.R. Johnson. “Efficient transposition algorithms for large matrices”. In: *Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (1993), pp. 656–665. DOI: 10.1145/169627.169814.

[6] Jinwoo Suh and V.K. Prasanna. “An efficient algorithm for out-of-core matrix transposition”. In: *IEEE Transactions on Computers* 51.4 (2002), pp. 420–438. DOI: 10.1109/12.995452.

[7] Valentin Volokitin et al. “Case Study for Running Memory-Bound Kernels on RISC-V CPUs”. In: (Aug. 2023), pp. 51–65. DOI: 10.1007/978-3-031-41673-6_5.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th. See page 190 on loop unrolling. San Francisco: Morgan Kaufmann, 2017.

[9] Richard M. Stallman and the GCC Developer Community. *Using GCC: The GNU Compiler Collection Reference Manual*. Comprehensive coverage of GCC, including compiler flags. Boston, MA: GNU Press, 2003.

[10] Mark C. Deakin. *Programming Your GPU with OpenMP*. Comprehensive guide to GPU programming with OpenMP. Cambridge, MA: MIT Press, 2021.

[11] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5th. Burlington, MA: Morgan Kaufmann, 2017.

[12] Linus Torvalds and the Linux Kernel Contributors. *perf: Performance Analysis Tools*. Accessed: 2024-11-29. 2024. URL: https://github.com/torvalds/linux/tree/master/tools/perf.

[13] Intel Corporation. *Get Started with Intel® VTune™ Profiler*. Accessed: 2024-11-29. 2024. URL: https://www.intel.com/content/www/us/en/docs/vtune-profiler/get-started-guide/2024-0/overview.html.

[14] Intel Corporation. *Intel Xeon Gold 6252N Processor: Specifications*. Accessed: 2024-11-30. URL: https://www.intel.com/content/www/us/en/products/sku/193951/intel-xeon-gold-6252n-processor-35-75m-cache-2-30-ghz/specifications.html.