Matrix Symmetry and Transpose: exploring Implicit and Explicit Parallelism with OpenMP.

Valerio Cassone ID 236359
University of Trento
Trento, Italy
valerio.cassone@studenti.unitn.it

***Abstract*—This project aims to deeply explore the use of Parallel Programming techniques in the forms of Implicit-Level Parallelism (ILP) and Explicit Parallelism with OpenMP (OMP) in order to optimize two different problems, the symmetry check and the transpose computation of a matrix, and subsequently benchmark and analyze the performance of both approaches, comparing their efficiency and scalability.**
The code and the results are available at the GIT repository [1].

I. Introduction

A. Problem identification

In linear algebra, the concepts of matrix symmetry and transposing are strictly correlated. Firstly, given $n \in \mathbb{N}$ and a square matrix M of $n \times n$ elements, M is symmetric if:

$$\forall i, j \in \{1, \dots, n\} \times \{1, \dots, n\},\ [M]_{ij} = [M]_{ji} \qquad (1)$$

Secondly, the transpose of a matrix is an operator which flips a matrix over its main diagonal: given $n, m \in \mathbb{N}$ and a matrix M of $n \times m$ elements, the transpose of M, denoted as M^T is a $m \times n$ matrix constructed by writing the rows of M as the columns of M^T or vice versa. In this way it is obtained that:

$$\forall i, j \in \{1, \dots, n\} \times \{1, \dots, m\},\ [M]_{ij} = [M^T]_{ji} \qquad (2)$$

By using the transpose operator definition, it is possible to state that a square matrix is symmetric if it is equal to its transpose, that is $M = M^T$.
The transpose operator is widely used in linear algebra for different purposes and computations, i.e. the inverse of an orthogonal matrix, and therefore it finds numerous applications in scientific fields.

B. Project objectives

The main objective of this project is creating a C program that sequentially computes the symmetry check and the transpose of a given square matrix M of floating-point numbers. Immediately after this, the proposed solution will be tested on a square matrix of $n \times n$ elements (where n is a power of two varying from 2^4 to 2^{12}) and the execution times of the two routines will be measured (in terms of wall-clock time). Thereafter there will be proposed optimizations of the sequential written code using Implicit-Level Parallelism (ILP) and Explicit Parallelism with OpenMP, in order to measure the new execution times, benchmark and analyze the approaches and compare their efficiency and scalability with respect to the sequential implementation in terms of speedup, efficiency and bandwidth.

II. State of the art

The state of the art in parallel matrix symmetry check and transposition has evolved to handle large-scale matrices efficiently across various parallel computing architectures, including distributed systems and multi-core processors. In the last years there were published researches showing different matrix transposition algorithms optimized and executed on parallel architectures.
The main strategies to efficiently solve the memory-related problem are using blocked, recursive, and hybrid data layouts in order to reduce the number of cache- and TLB-misses by storing the elements of a block contiguously. F. Gustavson, L. Karlsson and B. Kågström proposed an in-place matrix transposition using recursive blocked-format partitions of a matrix in order to parallelize the work and improve the efficiency of the cache usage [2].
One of the modern matrix-related problems is dealing with sparse data structures and in 2016 H. Wang, W. Liu, K. Hou and W. Feng proposed the algorithms **`ScanTrans`** and **`MergeTrans`** to significantly increase speedup of matrix transposition on parallel CPUs [3].
In 2020 B. Magalhaes and F. Schürmann exposed the central role MPI (Message Passing Interface) plays in the distributed transpose of large matrices. The key operation is a series of collective communication functions which facilitate efficient data transfer between processes [4]. These techniques represent the forefront of parallel matrix transposition, focusing on improving efficiency by minimizing communication overhead and leveraging the parallelism of modern computing hardware.
This project implements an out-of-place transposition and, rather than identifying new solutions to the described problems, tries to optimize existing direct approaches by exploiting parallelization techniques achievable with ILP or OMP and then analyze the obtained results in terms of speedup, efficiency and memory bandwidth with respect to the sequential baseline.

III. Methodology

A. Sequential Implementation

The solutions of the two proposed problems have been implemented in a C program. The symmetry check has

been implemented in a function called **checkSym** that contains two nested for loops checking, in each iteration **(i=1..n-1,j=0..i-1)**, if the input matrix **M** meets the symmetry condition **M[i][j] == M[j][i]**. When the condition is not met, a flag variable **check** is set to **false** to be returned by the function. At this point, however, **checkSym** is not yet correct. As a matter of fact, the **==** operator can't be used to compare floating-point numbers because it performs a bit-wise check and it does not take into account the representation problems that introduce approximation on these numbers [5]. The (numerically) correct way to perform the comparison is by checking whether the absolute value of the difference of the two operands is lower than a certain threshold **EPSILON** that defines the precision of test.
Proceeding with the transpose operator, it is implemented in the **matTranspose** function that contains two nested for loops copying, in each iteration **(i=0..n-1,j=0..n-1)**, the input matrix element **M[i][j]** in the symmetric position of the output matrix **T[j][i]**.

B. Implicit-Level Parallelism

Subsequently after implemented and tested the sequential solutions of the problems, it is possible to start searching for new approaches to reduce the execution times via using ILP. The first technique that can generally improve the execution time of a program is vectorization: the process of converting an algorithm from operating on a single value at a time to operating on a set of values at one time [6]. This is possible by fully loading a CPU register with multiple aligned homogeneous variables in order to perform a single operation instead of multiple ones. The said operation in the case of study is the subtraction **M[i][j] - M[j][i]**. Vectorization, however, can't be directly applied because the operands are not aligned in memory. In particular, if the elements of the matrix are organized in row major order [7], then **M[i][j]** retrieves aligned items for increasing **j** values, while this does not happen for **M[j][i]**. As a demonstration of this, trying to compile the code with **gcc -O2 -ftree-vectorize**, will result in two different errors: **inner-loop count not invariant** for **checkSym** because the inner loop doesn't have a prefixed number of iterations due to the condition **j < i** and **evolution of base is not affine** for **matTranspose**.
At this point, recognized that the real problem is strictly related to the loading/storing of data, rather than operations done on it, it is possible to change the memory access pattern of the routines in order to improve the efficiency of the data transfer. Instead of accessing row by row and column by column, which is optimal for the first and the worst for the latter, a trade-off solution is accessing the matrix in blocks. This aims to reduce the cache miss rate by working on the same region of the matrix for multiple iterations. The matrix will be divided into 32 X 32 blocks and then processed as stated before. Trying to automatically vectorize the new functions **checkSymImp** and **matTransposeImp** will lead in a **vectorization is not profitable** error by **gcc**.
Once the memory access pattern is optimized, it is possible to try prefetching the next matrix block, row by row, in order to reduce the reading time from the memory. This is obtained by using **_mm_prefetch(P,I)** that the compiler will replace with the prefetch function of the targeted architecture. **P** is the address of the data to be prefetched and **I** is a constant identifier for the cache level in which the data should be saved [8].
In addition to all of this, **gcc** offers a series of automatic optimizations that can be used to improve performance by adding option flags when executing the compile command. There are different levels of performance optimizations (**-O0**, **-O1**, **-O2**, **-O3**, **-Ofast**) whose results are compared in the following section.

C. Explicit Parallelism with OpenMP

Up to this point the proposed solutions have always been executed from a single process. Using OMP it is possible to explicitly parallelize the written code using multiple threads. Starting from the previously written code, the main idea is to equally divide the **for** loops iterations in multiple workloads, one for each spawned thread. This is obtained by adding **#pragma omp parallel for** above each **for** loop that is wanted to parallelize. The new parallelized functions will be **checkSymOMP** and **matTransposeOMP**. For the first one, however, parallelizing the loops will lead to a data race condition when changing the value of the **check** flag. This can be solved by adding **#pragma omp critical** the line above the assignment, but, considering that this instruction can be very frequently executed, all threads will wait at every iteration to access the variable, greatly increasing final execution time. A solution to this problem is instantiating a private flag for each thread and then joining all with a shared one at the end of all iterations (via a logical and). This approach requires to specify **reduction(&&: check)** at the beginning of the parallel section, reducing **check** access number to the amount of spawned threads.
OMP parallelization can be applied to the previous block solution: in particular, parallelizing the outermost for loop will divide the iterations such that every thread will work on its matrix block copied in its L1 cache. In this case, when **n** is lower than than 32 (prefixed block size), the new **size** will be obtained dividing **n** by the number of threads, in order to balance the work. In addition to this, the two for loops indexing the matrix blocks can be automatically collapsed into one by adding **collapse(2)** after **#pragma omp for**, but this can be done only in **matTransposeOMP**. In the other function, in fact, the inner for loop (indexing columns) depends on the outer, so this can't be collapsed. A solution could be using a single loop and mapping the row block and column block indexes into one using triangular numbers [9]. Furthermore OMP has a special clause called **schedule** that can be applied to parallelized loops to determine in which way the workload should be divided among threads. There are different "kinds" that can be used, in particular **static**

or **dynamic**, whose results will be compared in the following sections.

IV. Experiments

The proposed solutions in the last section of this document have been collected in a GIT repository [1] and tested on the following configuration: Intel(R) Xeon(R) Gold 6252N CPU @ 2.30GHz, with 512 MB of dedicated RAM and 32 selected cores. The compiler used is **gcc-9.1.0** with **OMP 2015.11**. The experiments have been conducted following and combining the methodologies described in the previous section. In every simulation, the input $n \times n$ square matrix **M** has been initialized with random **double** values and then processed by the symmetry check and matrix transposer functions. The dimension n is a power of 2 varying from 2^4 to 2^{12}. The tested solutions, among with their identifier are the following:

Code	Description
S	sequential execution
V	manual vectorization
B	block access pattern
BP	**B** with manual prefetching
BO1	**B** with **-O1** optimization
BO2	**B** with **-O2** optimization
BO3	**B** with **-O3** optimization
BOf	**B** with **-Ofast** optimization
O	**S** with OMP parallel execution
OR	**O** with collapse and reduction on **check**
OB	**B** with OMP, collapse and reduction.
OBT	**OB** with triangular numbers.
OB_S	**OB** with static scheduling.
OB_D	**OB** with dynamic scheduling.
OBf	**OB_D** with **-Ofast** optimization.

These simulations aim to test the proposed solutions in different combinations of methodologies, in order to study how they behave when increasing the input data in terms of FLOPS for the symmetry check and effective bandwidth for the matrix transpose. In addition to this, speedup and efficiency are compared for OMP approaches.

V. Results

A. Implicit-Level Parallelism

The Table I shows the computed FLOPS and effective bandwidth of each simulation, along with the value of n. Starting from these results, it is possible to make a series of consideration about the methodologies proposed. First of all, it is evident that the vectorization on the symmetry check has not brought a significant improvement (as the compiler stated): this is partly due to the amount of the overhead added, but also because it confirms that the symmetry check is an heavily memory-bound problem rather than computational. For this exactly reason, using the block access pattern greatly increase the performances of both routines. By analyzing **S** and **B** with the **perf** command **perf stat -e cache-references,cache-misses** it shows that, for the same amount of data, **S** requires significantly more accesses to higher-level caches rather than **B**; subsequently, **B** has a lower execution time because it reduces the memory stalls for higher-level cache misses (especially when dealing with bigger matrices). Furthermore, the results show that prefetching values doesn't always improve performances (in some cases it worsens them). In general, prefetching is valuable when the compiler can do other operations before needing new data, but in the case of symmetry, numbers' comparison is so computationally short that the prefetching overhead slows the execution. Finally, the **gcc** optimizations resulted in a significant performance improvement.

B. OpenMP

The results of the simulations with OMP have been organized in different plots showing the speedup and the efficiency over the number of threads (Fig 1, Fig 2, Fig 3, Fig 4). To facilitate interpretation, this report only shows the case $n = 12$ (with very smaller matrices, the overhead of creating thread slows the program more than the sequential execution), even though all the results can be visualized from the GIT repository. The first thing that stands out from the symmetry check graph is the significant decrease of speedup and efficiency when directly parallelizing the routine with the shared variable **check** (code **O**). This clearly demonstrates that the concurrent access to **check** at every iteration creates a bottleneck that is immediately removed when using private variables **c** (from code **OR** onward). Furthermore, for the second time it is possible to observe how the block access pattern significantly increases performance and even more when using scheduling strategies. In some cases, super-linear speedup is achieved with an efficiency greater than 100%. This can be observed with **OBf** (with **-Ofast** optimization) or even with **OBT** (triangular numbers allow a perfect linear workload division among threads, resulting in very high efficiency). Furthermore, static scheduling it has proven to be the best strategy for matrix transposing against the dynamic one.

C. Peak Performance Comparison

In order to compute the theoretical peak bandwidth it is necessary to know the RAM clock speed, the data transfer rate (2 for DDR architectures), the bus width and the number of channels. The information on the RAM frequency are not directly available without knowing the whole system or having super user privileges. For this reason the peak bandwidth is estimated based on the documentation on the CPU and its maximum theoretical available performance. The Intel(R) Xeon(R) Gold 6252N CPU @ 2.30GHz CPU [10] supports DDR4-2933 memory type, so the RAM peak frequency will be 2.933 GHz times 2. Furthermore, it has 6 8-bytes memory channels. The peak bandwidth will be:

$$PB = 2.933\,GHz \cdot 2 \cdot 8\,B \cdot 6 = 281.568\,GB/s \quad (3)$$

The computed peak bandwidth is then plotted along with the effective bandwidth of the matrix transpose routines when $n = 12$ in order to compare the theoretical and the effective values (Fig 5). The higher computed bandwidth is $28.24\ GB/s$, but it still is significantly lower than the peak. This probably depends on wrong assumptions (physical machine with a different memory system) or other environmental factors.

TABLE I
FLOPS AND EFFECTIVE BANDWIDTH FROM EXPERIMENTS

ID	Symmetry check (GFLOPS)									Matrix transpose (effective bandwidth in GB/s)								
n	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
S	0.318	0.286	0.292	0.274	0.273	0.159	0.145	0.103	0.051	6.87	6.25	6.39	2.75	2.25	0.947	0.914	0.287	0.151
V	0.183	0.205	0.209	0.233	0.231	0.16	0.155	0.117	0.057	6.67	6.26	6.09	3.02	2.12	0.962	0.967	0.361	0.151
B	0.15	0.119	0.183	0.239	0.261	0.245	0.242	0.223	0.126	6.77	5.43	5.74	5.84	5.44	5.04	4.79	2.93	2.45
BP	0.141	0.126	0.193	0.247	0.262	0.247	0.237	0.149	0.097	6.39	6.1	6.44	6.61	6.03	5.77	5.33	2.9	2.46
BO1	0.512	0.559	0.841	0.995	0.927	0.604	0.589	0.427	0.213	34.33	33.82	30.98	31.89	19.59	11.88	11.73	6.33	4.9
BO2	0.558	0.571	0.856	1.01	0.937	0.612	0.591	0.423	0.222	34.42	31.32	30.75	31.28	19.68	11.83	11.73	6.31	5.18
BO3	0.557	0.555	0.854	0.998	0.933	0.609	0.59	0.426	0.218	33.3	37.35	30.95	31.87	19.7	11.87	11.73	6.3	5.11
BOf	0.561	0.617	0.852	0.999	0.936	0.607	0.591	0.425	0.22	23.27	24.21	23.81	24.06	19.54	11.87	11.73	6.31	5.21

Fig. 1. Symmetry check: Speedup vs Number of threads

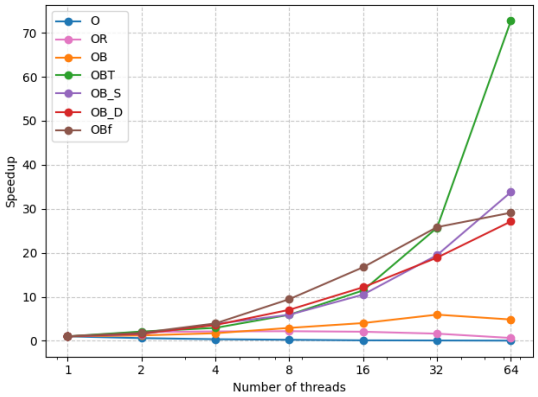


Fig. 2. Symmetry check: Efficiency vs Number of threads

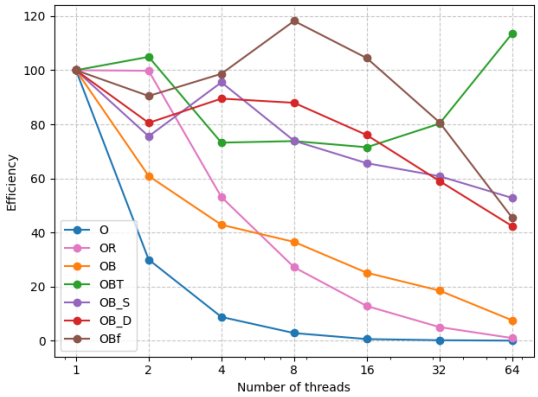


Fig. 3. Transpose: Speedup vs Number of threads

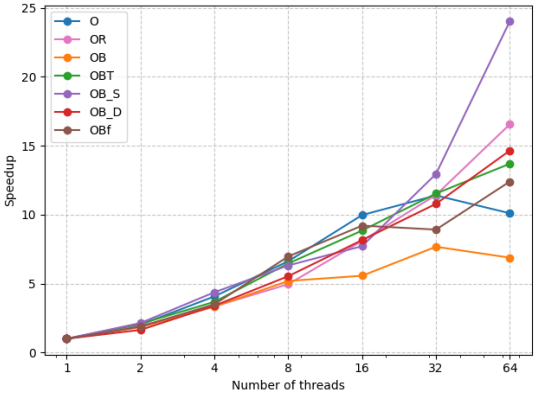


Fig. 4. Transpose: Efficiency vs Number of threads

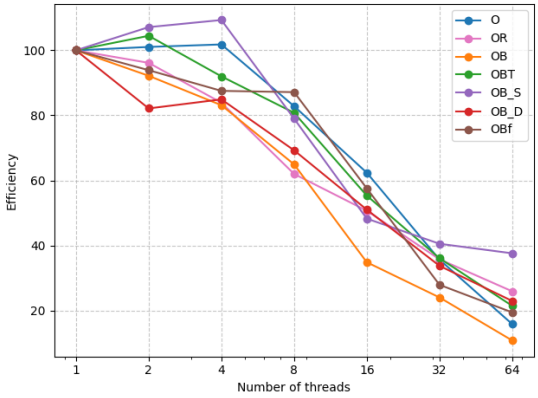
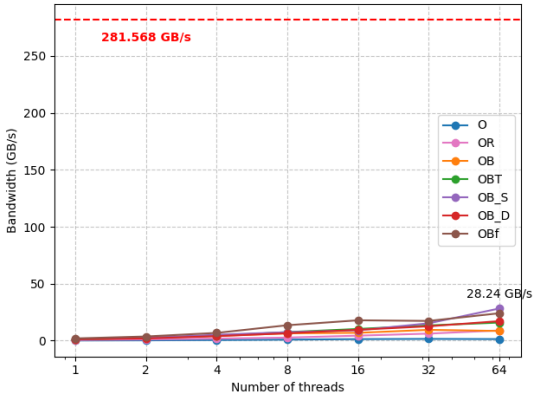


Fig. 5. Bandwidth vs Number of threads (n = 4096)



VI. CONCLUSIONS

The impact of optimizations, parallelization techniques and scheduling strategies has been analyzed on the efficiency and performance of sequential and parallel application. The results highlight how selecting optimal configurations can significantly improve speedup and efficiency. Specifically, compiler optimizations demonstrated performance improvements, although with limited gains compared to sequential baseline. Scheduling strategies provided specific advantages on the workload structure. Bandwidth analysis revealed discrepancies between theoretical and effective values. This study represents a step toward understanding best practices for optimizing parallel applications, offering insights for further research on optimization and scalability in multi-core environments.

REFERENCES

[1] GIT Repository with the implemented code, reproducibility instructions and results. https://github.com/ValeMX/parco-h1-236359

[2] Fred Gustavson, Lars Karlsson, and Bo Kågström. 2012. Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion. ACM Trans. Math. Softw. 38, 3, Article 17 (April 2012), 32 pages. https://doi.org/10.1145/2168773.2168775

[3] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel Transposition of Sparse Data Structures. In Proceedings of the 2016 International Conference on Supercomputing (ICS '16). Association for Computing Machinery, New York, NY, USA, Article 33, 1–13. https://doi.org/10.1145/2925426.2926291

[4] Magalhaes, Bruno & Schürmann, Felix. (2020). Efficient Distributed Transposition Of Large-Scale Multigraphs And High-Cardinality Sparse Matrices. 10.48550/arXiv.2012.06012. https://arxiv.org/abs/2012.06012

[5] Floating-Point Comparison

[6] Vectorization: A Key Tool To Improve Performance On Modern CPUs

[7] Row Major Order and Column Major Order

[8] What it takes to transpose a matrix

[9] Triangular number

[10] Intel® Xeon® Gold 6252N Processor