Introduction to Parallel Computing IntroPARCO 2024 H1: Matrix transposition

Nicolò Bellè
ID: 238178 - nicolo.belle@studenti.unitn.it

***Abstract*—The goal of this project is to evaluate the execution time for carrying out the transposition of a matrix by comparing sequential implementation, implicit parallelization, and explicit parallelization using OpenMP library in order to find the most efficient for each one. You may notice a particular difference when using explicit parallelization with OpenMP but only when increasing the size of the matrix.**

I. Introduction

A. Background

Matrix transposition is the process of flipping a matrix over its diagonal: this means that the rows of the matrix become its columns and the other way round. This calculation becomes challenging when a matrix reaches larger dimensions.

B. Importance of the problem

The project aims to analyze various implementations that can reduce the time required to execute the transposition using some parallelization techniques and memory optimization. This is important because transposition is very often at the base of scientific applications such as machine learning, parallel computing, graphics and linear algebra calculations that require computational speed and high performance.

II. State-of-the-art

New research on matrix transposition has focused particularly on improving efficiency because, for example, it fundamentally contributes to the development of artificial intelligence that has been entering our daily lives in recent years.

The latest developments address the computational challenge on block-wise and cache-efficient methods to improve memory access patterns, reducing latency in large-scale data.

In the following, we analyze existing solutions to the problem and also point out their limitations.

A. Parallelization with OpenMP

OpenMP is a standard programming model in C, C++ and Fortran that simplifies parallel programming for shared-memory architectures. It allows developers to parallelize code by dividing work across multiple threads with minimal code modifications.

However, for small workloads, the cost of thread creation and synchronization may outweigh performance gains. Moreover, improper synchronization can lead to race conditions and bottlenecks, so you have to be careful.

B. Parallelization with MPI

MPI is a standard library for parallel programming in C, C++, Fortran and more, focusing on distributed-memory systems which is favorable for its scalability and flexibility. Furthermore, this model offers fine-grained control over data sharing and synchronization.

Processes on distributed memory systems communicate via messages which introduce delays compared to shared memory.

The main disadvantages are that mpi requires explicit communication management and debugging can be challenging.

C. Gap to fill in this project

This project aims to address the inefficiencies of single-thread operations in shared memory systems. In particular, it targets to reduce cache misses, by parallelizing row-column operations, and distribute work between threads while minimizing wall-clock time using OpenMP library.

III. Contribution and methodology

The project is written using C programming language. The main file is called *deliverable1.c* and contains the following methods. (for loops of all *checksym* are not intentionally interrupted, even though *check* value is already set to false, to have a better wall-clock time calculation)

A. Input matrix

There's two type of input function. The first one is *initMatrixSize* which implements the use of *scanf*, method contained within the library *stdio.h*. This method puts the program in a hold waiting for the user to enter the size of the matrix. Inside it, there are several controls on the typed value: this must be an integer greater than zero, therefore part of the range of integers and finally must be a power of two as required. If one of these points is not met, after a suitable warning, the input is requested again.

The previous function is commented on because the input is captured by another method called *checkInvalidInputAtoi*. It uses *atoi* method that gets the first parameter of the execute command *./deliverable1*. As before, the same controls have been implemented for this input but in this case, after the error message, the execution ends.

B. Sequential implementation

The *checksym* function has been implemented for sequential control of the symmetry of the matrix, which is equivalent to verify if the matrix is mirrored or not with respect to its diagonal. Pseudo code:

$$if(M[i][j]! = M[j][i])check = false; \tag{1}$$

This block of instructions is contained within two nested for loops which iterate all rows and columns starting from *i = 0* and *j = i + 1*. In this method, as in the others *checkSym* explained below, a safety check is carried out in which the matrix must not be null, otherwise it leaves without performing any operation.

Instead, in the function *matTranspose* sequential matrix transposition is performed without any code optimization, so two nested for loops iterate all rows and columns starting from *i=0* and *j=0* by performing the following operation:

$$Tc[j][i] = Mc[i][j]; \tag{2}$$

where *Tc* is a matrix which will contain the result of the transposition and will be used as return for this method while *Mc* is the starting matrix passed to the function as parameter. *Tc* has been dynamically allocated within this method by *malloc* as follows:

$$float **Tc = (float **)malloc(r * sizeof(float*)); \tag{3}$$

The problem of this implementation arises when matrices reach large sizes because it becomes expensive to iterate and copy all values one after another in a sequential way.

C. Implicit parallelization

For the reason just explained, the function *checkSymImplicit* was implemented using vectorization techniques that allow the same calculation to be performed on multiple elements. Code line added before the inner for compared to sequential implementation:

pragma simd

pragma unroll(4)

The second line of code has been implemented because unrolling reduces the overhead of loop control instructions by replicating the loop body multiple times.

To improve the efficiency of sequential transposition, a block approach has been implemented in the function *matTransposeImplicit*. This is done by creating four nested for loops in which the outer ones divide the matrix into blocks with a specified size within the variable *BLOCK SIZE* while the two inner ones iterate the elements inside each blocks.

Outer loops pseudo code:

$$for(i = 0; i < r; i+ = BLOCKSIZE) \tag{4}$$

$$for(j = 0; j < c; j+ = BLOCKSIZE) \tag{5}$$

Inner loops pseudo code:

$$for(bi = i; (bi < i + BLOCKSIZE)and(bi < r); bi + +) \tag{6}$$

$$for(bj = j; (bj < j + BLOCKSIZE)and(bj < c); bj + +) \tag{7}$$

This improves performance by leveraging better cache utilization.

D. OpenMP parallelization

By exploiting parallelization with OpenMP, in *checkSymOMP*, the efficiency of *checkSymImplicit* has been further improved by dividing the outer loop between different threads so that each one controls a subset of rows. *check* variable is set as shared variable so that each thread can have access to it. Later, the *flush(check)* method before the internal loop imposes the synchronization of the value set on *check* for each thread in order to ensure consistency. Key lines of code:

pragma omp parallel for shared(check)

pragma omp flush(check)

In the *matTransposeOMP* method, two nested for loops were used to iterate the matrix. The following directive tells the compiler to parallelize them:

pragma omp parallel for collapse(2) schedule(static)

The *collapse(2)* clause is necessary to combine the two outer loops into a single loop to allow for a more flexible distribution of iterations among threads and than with *schedule(static)* the iterations are divided statically, meaning each thread will be assigned a contiguous block of work. This blocks are created based on the size specified in the variable called *BLOCK SIZE* as in the implicit parallelization.

E. File .pbs

The file *deliverable1.pbs* is used to submit the job in the cluster specifying some parameters. Inside it you will find the following line needed to compile the *C* file:

gcc -o deliverable1 deliverable1.c -fopenmp -O2

The flag *-fopenmp* is necessary because in the main function are initialized threads to allow only the execution of functions *matTransposeOMP* and *checkSymOMP* in parallel. Otherwise, without that, they would always be performed with the same thread and this would not bring an improvement in efficiency as can be seen in the table below.

IV. Experiments and System Description

A. Description of computing system

The simulations were performed by connecting via GlobalProtect vpn to the high-performance computing cluster, which is a collection of interconnected computers, called nodes, that work together as a single system to perform computationally intensive tasks. It's based on PBS queue management system. It's composed of 65 TB of ram and the computing nodes have 142 CPU (for a total of 7674 cores) and 10 GPU (for a total of 48128 CUDA cores).

Total theoretical peak performance: 478.1 TFLOPs

Theoretical peak performance CPU: 422.7 TFLOPs

B. Methodologies and libraries used in the project

To run the simulations a job has been started in the short_cpuQ queue specifying the following parameters:

-l select=1:ncpus=64:ompthreads=64:mem=1gb

In the project it was thought to have each transposition and symmetry control function performed 10 times, so as to calculate the average time taken by each one and obtain a more accuracy result. The wall clock time function was used for calculating the time taken to transpose and check the symmetry of the matrix as required. The values were later saved in a *csv* file for comparison between the different implementations. In addition, to analyze the explicit implementation more thoroughly were calculated speed up, efficiency and bandwidth for each thread and size of the matrix. This implementation was performed by 1, 2, 4, 8, 16, 32, 64 threads and each one for ten times as explained above. The libraries included in the *deliverable1.c* file are:

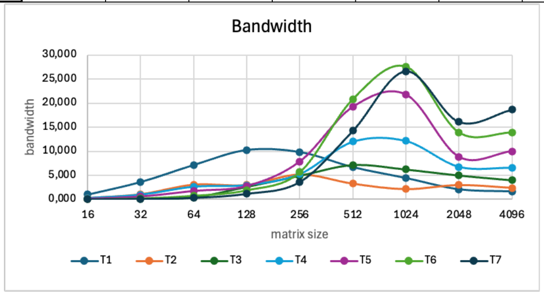
- omp.h for parallelization
- time.h for the random method used to generate random value to initialize the matrix
- ctype.h, limit.h and errno.h to check the input size value
- stdbool.h for type bool
- stdio.h and stdlib.h for some C language methods

V. Result and Discussion

TABLE I
AVERAGE TIME SEQUENTIAL, IMPLICIT AND EXPLICIT IMPLEMENTATION

	AVG_TIME_SEQ	AVG_TIME_IMP
16	0,000000295	0,000000207
32	0,000000911	0,000000695
64	0,000003777	0,000002751
128	0,000015365	0,000011451
256	0,000068119	0,000053343
512	0,000383294	0,000297871
1024	0,002123201	0,001885868
2048	0,026073229	0,01604766
4096	0,123007308	0,081271028

	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	64 Thread
16	0,000002073	0,000008546	0,000185979	0,000009466	0,000017706	0,000041040	0,000089921
32	0,000002274	0,000007847	0,000154365	0,000008833	0,000014143	0,000035649	0,000082166
64	0,000004598	0,000010729	0,000087135	0,000012678	0,000018980	0,000041830	0,000099303
128	0,000012812	0,000043888	0,000050893	0,000044578	0,000047543	0,000068165	0,000112589
256	0,000053450	0,000100825	0,000107261	0,000100014	0,000066851	0,000092541	0,000144719
512	0,000313691	0,000642446	0,000293449	0,000175004	0,000109365	0,000100802	0,000147317
1024	0,001889523	0,003969087	0,001350668	0,000690910	0,000385858	0,000304514	0,000315072
2048	0,015923680	0,011265126	0,006776351	0,004977832	0,003808418	0,002410288	0,003688883
4096	0,080751914	0,056278055	0,033904088	0,020494908	0,013448344	0,009664299	0,007207968

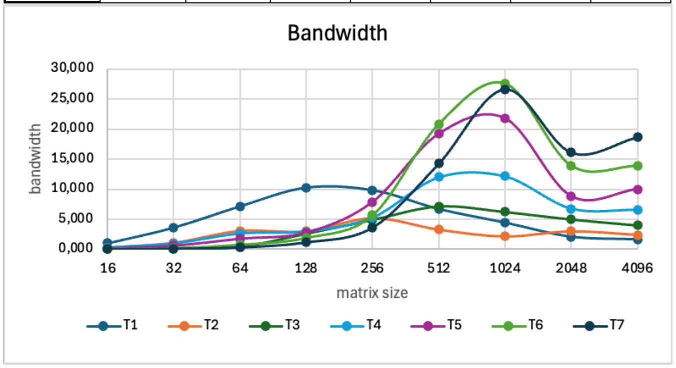


A. Table 1

In this graph is represented the average time taken by each thread to carry out the matrix transposition for each size (in green) compared to the average time taken for sequential (in blue) and implicit implementation (in orange). You can see how efficient parallelization is when the size exceeds 1024, especially with threads.

TABLE II
SPEED UP

	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	64 Thread
16	0,14	0,03	0	0,03	0,02	0,01	0
32	0,4	0,12	0,01	0,1	0,06	0,03	0,01
64	0,82	0,35	0,04	0,3	0,2	0,09	0,04
128	1,2	0,35	0,3	0,34	0,32	0,23	0,14
256	1,27	0,68	0,64	0,68	1,02	0,74	0,47
512	1,22	0,6	1,31	2,19	3,5	3,8	2,6
1024	1,12	0,53	1,57	3,07	5,5	6,97	6,74
2048	1,64	2,31	3,85	5,24	6,85	10,82	12,07
4096	1,52	2,19	3,63	6	9,15	12,73	17,07



B. Table 2

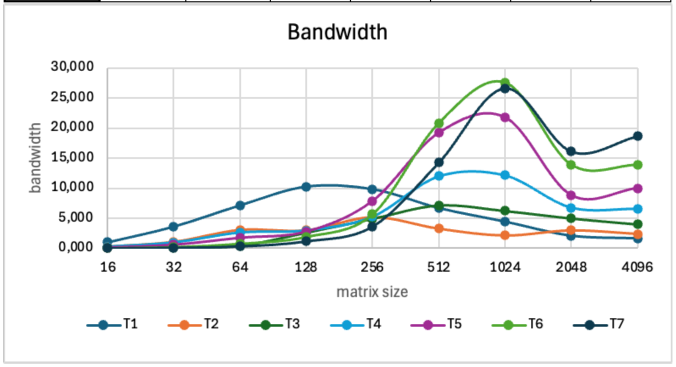
speed up has been calculated so that we have a data that measures how much the execution of a parallel application changes by implementing threads. This was done using the following formula:

$$SpeedUp = \frac{Ts}{Tp} \tag{8}$$

where Ts is the time of sequential implementation and Tp is the time of explicit parallelization.

TABLE III
EFFICIENCY

	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	64 Thread
16	14,25%	1,73%	0,04%	0,39%	0,10%	0,02%	0,01%
32	40,06%	5,80%	0,15%	1,29%	0,40%	0,08%	0,02%
64	82,14%	17,60%	1,08%	3,72%	1,24%	0,28%	0,06%
128	119,93%	17,50%	7,55%	4,31%	2,02%	0,70%	0,21%
256	127,44%	33,78%	15,88%	8,51%	6,37%	2,30%	0,74%
512	122,19%	35,83%	32,65%	27,38%	21,90%	11,88%	4,07%
1024	112,37%	40,75%	39,30%	38,41%	34,39%	21,79%	10,53%
2048	163,74%	115,73%	96,19%	65,47%	42,79%	33,80%	11,04%
4096	152,33%	109,29%	90,70%	75,02%	57,17%	39,78%	26,66%



C. *Table 3*

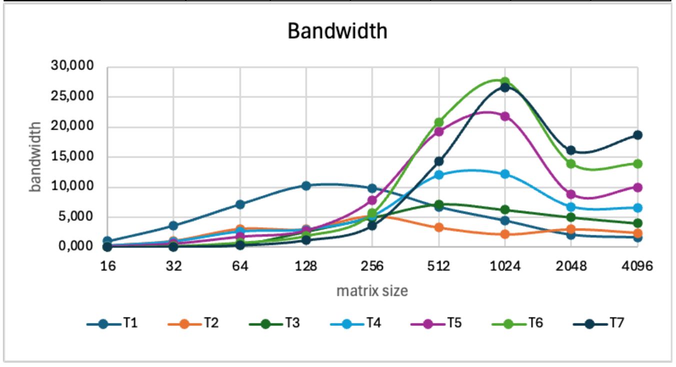
Efficiency has been calculated to know how many computational resources are actually used to perform the parallel transposition. This was done using the following formula:

$$Efficiency = \frac{SpeedUp}{Nthreads} \times 100$$

where *SpeedUp* is the value computed before and *Nthreads* is the number of threads.

TABLE IV
BANDWIDTH

	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	64 Thread
16	0,988	0,240	0,.011	0,216	0,116	0,050	0,023
32	3,603	1,044	0,053	0,927	0,579	0,230	0,100
64	7,126	3,054	0,376	2,585	1,726	0,783	0,330
128	10,231	2,986	2,575	2,940	2,757	1,923	1,164
256	9,809	5,200	4,888	5,242	7,843	5,665	3,623
512	6,685	3,264	7,147	11,983	19,176	20,805	14,236
1024	4,440	2,113	6,211	12,141	21,740	27,548	26,624
2048	2,107	2,979	4,952	6,741	8,811	13,921	16,096
4096	1,662	2,385	3,959	6,549	9,980	13,888	18,621



D. *Table 4*

Bandwidth describe for each matrix size his effective data transfer rate from memory to CPU achieved during real operations and the comparison is done for each thread. This was done using the following formula:

$$Bandwidth = \frac{DataTransfered}{Tp}$$

where

$$DataTransfered = n \times n \times n \times size\_Of\_Float$$

Conclusion

Explicit parallelization has proved to be a good method for greatly improving the efficiency of matrix transposition, while implicit parallelization implements improvements much smaller. The most difficult obstacles to overcome are to speed up access to memory and ensure, in addition to parallelism, also the synchronization and consistency of the operations that are performed. To solve this problem, the use of openmp was effective because it took two more lines of code to notice a big computational difference. It is a matter of finding the right clauses to apply to the code in order to optimize it at its best.

References

[1] ”wiki.info”, OpenMp library. https://hpc-wiki.info/hpc/OpenMP
[2] ”rivier.edu”, Matrix transposition. https://www2.rivier.edu/journal/ROAJ-Fall-2023/J1274_Malita%20and_Stefan_Fall-2023.pdf
[3] ”Harnessing Parallel Computing Power of OpenMP Optimization for Large Matrices”. https://researchtech.net/index.php/2024/01/ harnessing-parallel-computing-openmp-optimization-matrices/

GIT repository Nicolò Bellè https://github.com/Mystic-10/IntroPARCO-2024-H1/blob/main/README.md