# Parallel Computing first project

Francesco Nodari
*Ingegneria Informatica, Comunicazioni, Elettronica dept.DISI*

ID: 234462
email address: francesco.nodari@studenti.unitn.it

***Abstract*—The project consists in writing a sequential code that composes the transposition of a matrix and to find the best way to parallelize that code to improve performances by using implicit and explicit parallelization techniques. For the implicit parallelism there are several techniques to improve the code; I divided the matrix in two submatrices. For the explicit parallelism I used the same technique I used for the implicit parallelism as base and I used openMP pragmas to give the compiler commands to parallelize the code. Analyzing the data I realized that the best technique of the three is the explicit parallelism but the way we implement it is not universal; we must also consider the dimensions of the matrix, because the size of the matrix makes a big differences in deciding the technique and also how many threads to use.**

## I. Introduction of the problem and importance

Matrix transposition is the process of swapping the rows and columns of a matrix; it's a fundamental matrix operation of linear algebra and arises in many scientific and engineering applications. Its relevance is due to the fact that today's commodity computers are providing to users an enormous computational power that needs to be exploited in a suitable way. Sparse matrix transposition features an irregular memory access pattern that depends on the input matrix, and thereby its dependencies cannot be known before its execution. This situation demands from the programmer an important effort to develop an optimized parallel version of the code. Understanding its importance in parallel programming helps improve performance and efficiency in handling large-scale problems. The main objective of this project is to enlighten the differences in performance between implicit and explicit parallelization and serial code; understanding the different ways to parallelize a serial code, see first hand and calculate the actual improvement in time efficiency.

## II. State of the art

As I said before in the introduction the matrix transpose is a fundamental operation and many scientists, engineers and programmers have developed different algorithms to improve that.

### A. S. Pissanet- zky's algorithm

As a case study and motivating application, the Sparse Matrix Transposition algorithm proposed by S. Pissanet- zky has been considered one of the most efficient sequential algorithms for the sparse transposition. It works with matrices stored in the CRS (Compressed Row Storage) format, which is a widely used space-efficient scheme for representing sparse matrices. Sparse matrix transposition features an irregular memory access pattern that depends on the input matrix, and thereby, its dependencies cannot be known before its execution. In turn, this code can be useful as a benchmark to test transactional systems as the dependence pattern can be tuned by selecting a given input matrix.

### B. Data partition

A well known approach to solve the parallelization of this code is by following a data-partitioning strategy; basically, the input sparse matrix needs to be partitioned into smaller submatrices, also sparse, that needs to be distributed among threads. This partition can be as complex as desired (block, cyclic, ...) and it implies an index translation from the CRS structure of the input matrix to those of the submatrices. A simple partition can consist of partitioning the matrix by rows, with all submatrices of the same size. In this case the partitioning procedure can be executed fully in parallel. As we are assuming a shared memory architecture, parallelism has been expressed in the figure by using OpenMP notation. After partitioning the input matrix, each thread invokes the sequential transposition applied to the submatrix or submatrices of which is in charge. In a final step, all transposed submatrices have to be gathered to get the transposed of the input matrix. Depending on the partitioning strategy, this step can be more or less complex, as different column data may need to be interleaved to reconstruct the original matrix structure. As discussed, the amount of information about the problem can guide the programmer towards a more efficient solution when parallelizing codes with complex memory patterns, as is the case for sparse transposition. Transactional memory abstractions can make the programmer's work easier in these cases, because it allows exploiting concurrency optimistically without needing extra information. Many researchers have considered the problem for different architectures.

### C. Eklundh's solution

In 1972, Eklundh considered the problem of directly accessing rows or columns of a matrix when its size is larger than the available high-speed storage. The first algorithm required O(NlogN) disk accesses for performing the in-place transposition of matrices stored on disk. The algorithm required that at least two rows of the matrix fit into main memory.

### D. Others's possible solutions

O'Leary implemented an algorithm for transposing an N X N matrix on a one-dimensional systolic array. Azari, Bo janczyk and Lee developed an algorithm for transposing an M X N matrix on an N X N mesh-connected array processor; Johnson and Ho presented an algorithm for a Boolean n-cube, or hypercube. Systolic Arrays, introduced by Kungand Leiserson,are very popular implementations for parallel matrix computation. However, there seems to be no consensus on the best way to transpose a matrix for use in a systolic array algorithm. For Example, Bojanczyk, Brent, and Kung suggest using "a buffer that supports fast two-dimensional addressing" and the speed of their algorithm depends critically on the speed of this hardware.

### E. Ullman's proposal

Ullman discusses a systolic algorithm of Allah and Kosaraju which transposes a matrix in place. Current advanced architecture computers possess hierarchical memories in which accesses to data in the upper levels of the memory hierarchy (registers, cache, and/or local memory) are faster than those in lower levels (shared or off -processor memory).

### F. Conclusions

To exploit the power of such machines, block-partitioned algorithms are preferred for dense linear algebra computations, in which operations are performed on submatrices, rather than distributing matrix data over processors we therefore assume individual matrix elements. In distributing matrix data over processors we therefore assume a block scattered composition; the block scattered decomposition can reproduce the most common data distributions used in dense linear algebra.

## III. Contribution and Methodology

It's known and I realized first-hand by talking to my colleagues and using a different compiler on my laptop how the operating system and the compiler influence the performance of the code; my project is helpful to understand how a computer like mine parallelizes these functions.

### A. Serial code

For the serial part I used a double for loop for both the transposition function and the check of the symmetry, using a single thread. For the transposition function I simply changed the indexes of the rows and the columns between the matrix passed as argument and the transpose; then I returned the transpose. I used a matrix allocated dynamically to be able to have a bigger size compared to the static declaration which gives, for large sizes, error in the bus. We know that a squared matrix is symmetric if it is the transpose to itself; so for the symmetry check I checked if the parameters inside the matrix at a certain indexes were the same in the same matrix with the same indexes but swapped; I used a Boolean variable as support, in doing so I could firstly let the compiler run the whole function and not stop at the first iteration, so I could see how much time it took to complete the task and of course return the value of the variable to let the user know if the matrix is symmetric or not.

### B. Implicit parallelism

For the implicit parallelization, I divided the matrix into two smaller matrices and I applied the same technique of swapping indexes between the matrix and the transpose. I did this because my computer would recognize that I accessed different memory locations sequentially so this section can be parallelized using different threads without having memory access problems; I tried using two variables as support in the transposition function to break even more the steps so I could parallelize a little bit more, but it decreases the response because the compiler would have used the same variable to get the value from the start matrix and store it in the transpose; in doing so it couldn't parallelize this section because I was accessing the same memory location, so it had to wait for the first thread to finish and then re-access the same value memory location and then compute the next computation. I used, at the end, the vectorization technique and I accessed the memory in a more efficient way.

### C. Explicit parallelism

For the explicit parallelism I used the same technique I used in the implicit parallelism, but now I chose to divide the matrix transposition in four in the same cycle; I did this by accessing four different memory locations in both the start matrix and the transpose, in doing so for every loop I would have swapped eight different values to the transpose matrix, four per for cycle. For the symmetric check I used the same technique I used in the implicit symmetric check code; I couldn't do like I did in the transpose function because I would have accessed the same boolean variable memory location eight times per cycle and I would have created a bottle neck problem in which each thread would have to wait their turn to access the variable. I also used the openMP pragmas section to explicitly tell the compiler to parallelize that part and how. In the pragmas sections, in both the transposition and symmetric check functions, I passed how many threads I wanted to be used to collect the data and understand how many threads were needed to complete the check.

### D. Challenges

In this project I faced problems in the implicit and explicit parallelism sections; in the implicit I had to find a way to let the compiler know in a smart way that that part could be parallelized, for example without accessing the same memory location sequentially and use vectorization. For the explicit I had to find the right pragmas to improve the parallelism on my computer, for a simple project like this using too many flags it could backfire and take more time because some, could be redundant and slow down the program; and also I had to find a way to use the threads in a smart way trying to avoid threads overhead.

## IV. Experiments and System Description

### A. My system

For this project I used a MacBook pro with a M1 silicon chip with a 10 core CPU, the version is 14.5; I run my simulations in the university cluster and on Visual Studio Code, Apple clang version 16.0.0.

### B. The code

In the code I included both running options, for running on the Visual Studio Code compiler it's compulsory to comment lines 32 and 33, these lines take the input from the PBS file, and take off the comments on lines 30 and 31, these lines allow the user to put the input itself. When the user lunches the PBS file in the cluster it will show automatically all the dimension's outputs I took into consideration, in doing so the user doesn't have to change the input in the PBS file every time. For changing the threads number the user has to change it manually at lines 178 and 206. The outputs show the time it took the program to execute different functions; this allow the user to collect data and calculate the speed up and the efficiency of the different functions and choose the best fit for the need.

## V. Results and Discussion

Below I included the results of my tests and the graphic of the speed up and the efficiency, I included all the data, including the graphic of the speed up and the efficiency related to the tests on the Visual Studio Code compiler, in the "PARALLEL COMPUTING 1 PROJECT DATA" file. As we can see from the graphics we have the peek performance when we used only four threads, that is due to the simplicity of the task and the small dimension of the matrix and because when we have too many threads the compiler wastes time organizing them and deciding to assign to whom what. We can also see that, regardless of the amount of threads, we are are decreasing the response time as we increase the matrix size, that's because we have to do more computations and that's where the parallel system works best because we run different computation in parallel, and the time the compiler takes to organize the threads is irrelevant compared to when we have a small matrix size. In comparison to the state-of-the-art here we are not operating on a sparse matrix ad the size in reduced enormously, however we can notice the improvement using the vectorization technique and a better memory access and the data partition, so we can accurately say that these technique are useful and work even for not sparse matrix having a smaller size; nonetheless to have a more accurate comparison we gonna have to use a sparse matrix and increase the dimension.

The data presented were obtained are the average value of several simulations ran on the university cluster.

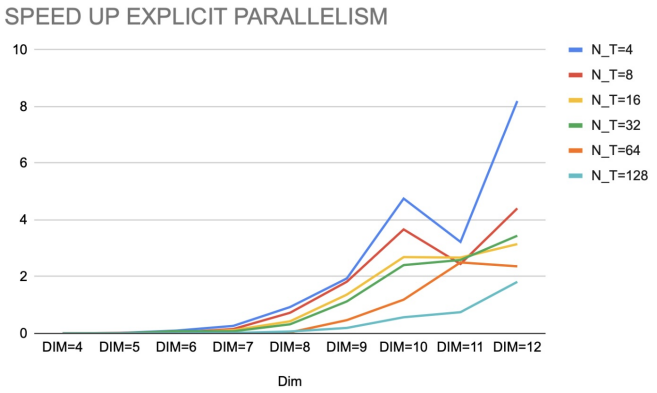


Fig. 1. Graphic of the speed up

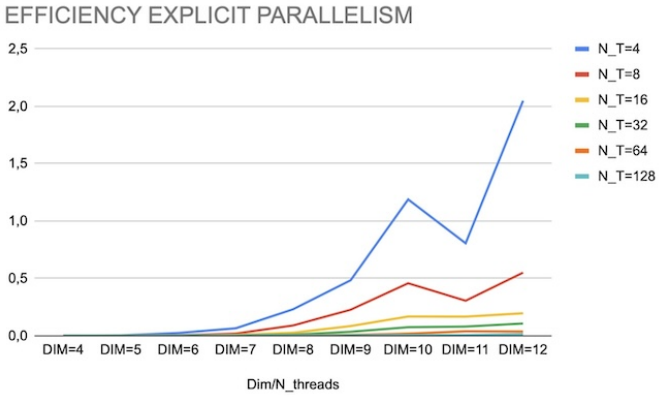


Fig. 2. Graphic of the efficiency

| threads | T=4 | T=8 | T=16 | T=32 | T=64 | T=128 |
|---|---|---|---|---|---|---|
| DIM=4 | 0.00443 | 0.00311 | 0.00204 | $9.62e^{-4}$ | $5.88e^{-4}$ | $4.59e^{-4}$ |
| DIM=5 | 0.01407 | 0.00519 | 0.00318 | 0.00374 | 0.00108 | 0.00141 |
| DIM=6 | 0.0907 | 0.0133 | 0.00367 | 0.00538 | 0.00508 | 0.0163 |
| DIM=7 | 0.2562 | 0.1525 | 0.1172 | 0.0339 | 0.005906 | 0.002525 |
| DIM=8 | 0.9237 | 0.7262 | 0.3435 | 0.0537 | 0.03077 | 0.0662 |
| DIM=9 | 1.937 | 1.2806 | 1.365 | 1.0215 | 1.0626 | 0.9116 |
| DIM=10 | 4.7525 | 3.6844 | 2.6894 | 2.407 | 1.1887 | 0.5662 |
| DIM=11 | 3.2246 | 2.4459 | 2.617 | 2.5862 | 2.5 | 0.7468 |
| DIM=12 | 8.1955 | 4.4059 | 3.1464 | 3.4426 | 2.364 | 1.6815 |

TABLE I
Speed Up Explicit Parallelism for Varying Dimensions and Threads

| threads | T=4 | T=8 | T=16 | T=32 | T=64 | T=128 |
|---|---|---|---|---|---|---|
| Dim=4 | 0.00111 | $3.9e^{-4}$ | $1.7e^{-4}$ | $1.92e^{-4}$ | $1.99e^{-7}$ | $3.9e^{-7}$ |
| Dim=5 | 0.003519 | 0.00194 | $3.82e^{-4}$ | $1.117e^{-4}$ | $2.96e^{-6}$ | $9.28e^{-7}$ |
| Dim=6 | 0.02454 | 0.009314 | 0.002309 | 0.002561 | $8.633e^{-6}$ | $3.119e^{-6}$ |
| Dim=7 | 0.05671 | 0.0190 | 0.008972 | $5.33e^{-4}$ | $7.36e^{-6}$ | $393e^{-5}$ |
| Dim=8 | 0.2318 | 0.09078 | 0.0266 | 0.01129 | $6,185e^{-5}$ | 0.00013 |
| Dim=9 | 0.484 | 0.227 | 0.0355 | 0.0391 | 0.00216 | 0.00104 |
| Dim=10 | 1.188 | 0.4359 | 0.16809 | 0.07253 | 0.0148 | 0.004442 |
| Dim=11 | 8.061 | 0.35574 | 1.169 | 0.108 | 0.00928 | 0.00583 |
| Dim=12 | 2.048 | 0.5507 | 1.196 | 0.107 | 0.0363 | 0.01419 |

TABLE II
Efficiency Explicit Parallelism for Varying Dimensions and Threads

## VI. Conclusions

As we can see from the data and the graphics there isn't a best way to compute the matrix transposition it all depends

on the size.

### A. Small Matrices

For small matrices the serial code is the best choice because, as we can see from the data, the differences are very slim; we hate to consider that we are operating on a very small chunk of memory and the number of computations are very low so we are not gaining any advantage in running things in parallel and also because we are wasting most of the time in preparing the parallel section and organizing the threads. We have also to consider that we are withholding threads which are not giving an actual improvement and that could be used elsewhere.

### B. Medium Matrices

If we consider a relatively medium matrix we can see that the serial code is slower than the parallel counterpart, but we also notice that the implicit code is faster than its counterpart. That's because in the implicit parallelism the compiler doesn't have to open a parallel section and gather a set number of threads like in the explicit parallelism.

### C. Big Matrices

If we consider big matrices however we notice that the best choice is the explicit parallelism by a considerable margin, now the time it takes to open the parallel section, get the threads and organize them is relatively irrelevant because of the quickness it takes to complete the tasks compared to the other techniques.

### D. Thread's problem

If we consider the threads though we notice that we have peek performance with only four, that's because, as I already said, the task and the amount of memory is relatively simple and small; so if we have more threads than we need and so it creates like a bottle neck problem, we have a lot of threads in the parallel region but just a few can run so the others creates an overhead.

### E. Conclusions

To sum up everything unfortunately there is not a rule to follow to decide the method and if we use the explicit parallelism the threads number. We gonna have to understand the problem and how many resources we are gonna use and need to solve it and how many functions and processes we are gonna use, test the code changing threads number or method, calculate the speed up and the efficiency and decide the best fit for the problem in our hands.

### F. Contributions

## REFERENCES


[1] "Parallelizing the sparse matrix transposition: reducing the programmer effort using transactional memory." by Miguel A. Gonzalez-Mesa, Eladio D. Gutierrez, Oscar Plata Dept. Computer Architecture, University of Malaga, 29071 Malaga, Spain

[2] " Parallel matrix transpose algorithms on distributed memory concurrent computers" by Jaeyoung Choi, Jack J. Dongarra, David W. Walker. Department od Computer Science University of Tennessee

[3] " IN-MEMORY PARALLEL PROCESSING FOR MATRIX TRANSPOSE" by Mihaela Malita, Ph.D. Associate Professor, Department of Mathematics and Computer Science, Rivier University and Gheorghe M. Ştefan** Professor, National University of Science and Technology Politehnica Bucharest, Romania

[4] "SystolicArraysforMatrixTransposeandOtherReorderings" by DIANNE P.O'LEARY