Parallel Computing for Matrix Transposition

Deliverable 1 – Introduction to Parallel Computing 146209

Alessandro Luzzani
Dip.Ingegneria e Scienza
dell'Informazione
University of Trento (IT)
Bachelor Degree Student cod.235065
Lodrone, Tn, Italy
alessandro.luzzani@studenti.unitn.it

***Abstract*—This document tries to explain some possible implementations and usage of matrix transposition functions in a code, developed with C++. The main aim is to show how different techniques that lead to the same solution can change execution time, memory usage and CPU burst. As parallel programming is one of the most effective possibilities in optimized programming, I would desire to show especially how parallel computing possibilities give noticeable results with just a few efforts by the programmers' side by including just a few libraries or techniques more than usual.**

I. INTRODUCTION

Matrices are probably the first topic covered by teachers in most of geometry and linear algebra programs around Italian universities. They are basically a set of information arranged in columns and rows. Each position contains a given value of the data that the matrix contains. With integers it could be (0,1,19) with letters could be (a,w,u). Matrixes seem to be quite abstract and ambiguous entities, but they actually rule the world of physics, maths and also computer science. Just to make a few examples, matrixes can be used to calculate possible changings of the reference system in a lot of physical problems while in electrical engineering are used to solve in an easy way equations simplifying many scenarios. In Computer Science, they are used to represent images, as they can filled by numbers (integers or decimals) that could be connected to a specific colour. Each slot can be interpreted as a pixel whose tonality is decided by the number it contains.

Figure 1:exemple of matrix filter and edge detector and matrix filter effect on a photo[2]

The importance of matrices and their wide applications led developers to focus on coding thinking always about new efficient functions that are used to make calculus or transformation on them. During the years were published various libraries for C++ e.g. Eigen that can help in a sophisticate way arithmetical operations. However, there are still some reasons that could encourage to self-develop our tools. This is precisely the case we will explore, with development of tailor-made functions.

Transposition is one of the most frequent operations. It consists in changing position of all data inside the matrix, following the symmetry of the main diagonal.

In this project, we have only used square matrixes that can contain floats for simplicity. We will see some transposition methods as well as many other functions that revealed to be fundamental in performance and correctness evaluations.

II. CODE AND FUNCTIONS' IMPLEMENTATION

A. Introduction to the code

I decided to split my job into different parts that all co-operate for the final execution. I developed two .h files, one dedicated to time management, the other (nouned Matrice.h) to matrices creation and matrices' functions. Both .h files can be included in cpp files and their functionalities are widely used in different ways by the main function.

B. Matrice.h , the heart of the executable

Matrice.h is the key point of my solution. It contains everything useful to work on matrixes. Actually Matrice.h is a header file which contains just one class called Matrice, whose attributes are a *static const int* that defines the size and a statically allocated 2d array *m* of floats. The default constructor *Matrice()* and the function *createM()* both are used to create a new matrix when needed, with the first one that allocates the necessary space in the stack and the second one that assigns pseudo-random float numbers in each slot. Also the *createMsymmetric(float x)* function acts similarly, but instead of random numbers, assigns a float number taken as an input. This last function creates matrixes with all same values and can be called for specific reasons.

As matrixes were not dynamically allocated, the compiler must know the size before launching the program. Size can be change only by opening the code in the .h file in which it is defined. Although it could be changed to check scalability, size of matrixes was by default decided as N=1000 to start, so that we can dispose of a quite high amount of data to process. With each float number occupying 4 bytes we get ***4 x 1000 x 1000 ≈ 3,8 MB*** .

Then there are other general functions that can be used for various reasons such as *stampamatrice()* that, as the name suggests, prints matrixes; a functions that indicates if one matrix is the transpose of another *checkTransposeOf(const Matrice& m2)*, and other two functions that copy the value of one matrix into another given one.

C. CheckSym and matTranspose: three different ways of working.

CheckSym() and *matTranspose()* are the two most important functions inside the struct Matrice. They are used respectively

to check if a matrix is symmetric and calculating the matrix transpose.

Both functions can be found in three distinguished implementations: one serial, one with implicit parallelization techniques and lastly with the usage of OpenMP, so with an effective parallel procedure.

Starting with the serial one, whose main aim is just to calculate a result showing a base case scenario, with *matTranspose()* we see two nested for loops (very common to see with matrixes or 2d elements), one that runs from a local variable 0 to the size of the matrix, and the inner one starting from the variable +1 also ending to the size of the matrix. This basic algorithm doesn't clearly have anything special. It is noticeable the fact that in this way, the function does not care about the diagonal at all: it would be very useless, as the main diagonal is never considered in the process of transposition, and works only on half of the matrix avoiding useless repetitions. The swap is done in the simplest way possible, avoiding to ask for external functions as we need to limit overhead in our job with a local float variable c.

```
for (int i = 0; i < size; i++)
{

    for (int j = i+1; j < size; j++)
    {

        float c = m[i][j];
        m[i][j] = m[j][i];
        m[j][i] = c;
```

Figure 2: nested loop for matTranspose()

As regards the serial version of the function checking the symmetry, the situation is very similar: two nested loops checking each couple. However, the first time one couple does not have same values, it is enough for us to know that a matrix is not symmetric, so we are allowed to break the loop safely.
Moving on, we can see how the implicit parallelized functions are barely different from the serial ones. In the function working on transposition, it was just changed the way the swap acts. Instead of calling a new *float c* every time the loop iterates, it was decided to create two floats *c* and *e* before the nested loops begin, and then using the two variables overwriting them each time and then copying their values in the destination's slots.
It behaves quite like an unroll, changing a little bit the instruction but without using an effective pragma unroll.

```
c = m[i][j];
e = m[j][i];
m[j][i] = c;
m[i][j]= e;
```

Figure 3: swapping numbers in implicit parallelization case, the different approach

It could seem that adding an operation to the swap procedure might ask more effort by the system, however this is not always true when other methods are considered. Compilations flags, for example, are a tool that helps the compiler to optimize for size, speed or other factors. They should go well for this precise case. As a matter of fact, the key point of implicit parallelization speedup in this job, is the -O2 used in compilation, that guarantees efficiency without compromising security aspects. To be more specific -O2 is composed by a good quantity of flags, it enables all the flags part of -O1 adding around 50 more flags.[2] .It was decided to avoid using -O3 or even -Ofast as they act too aggressively. Furthermore, also other pragmas such as *pragma unroll* did not work as expected: they just added useless latency, so they were not included in this case. The function to check symmetry does not differ from the one in the serial case, but benefits from the advantages of -O2 too.

Last but not least, we can analyse the version of both functions parallelized with OpenMp. OpenMp is a set of compiler directives, libraries, and environment variables that provide a simple and flexible interface for parallel programming in C, C++, and Fortran. It is often used in multi-core programming with a single memory. I decided to go for a soft and smart approach, starting from the algorithm of the first serial function. Usually, the best solution for nested loops is to collapse them with an instruction. In this case things did not go as expected for issues related to the fact that the two nested loops run on different offsets. For this reason, I changed the way of parallelization, choosing just to parallelize the outer loop with a *#pragma omp parallel for private(c).* Threads' count can be managed outside the program*

In this case also checkSymOMP() has its own implementation, I added a Boolean variable *symmetric* which is shared between all threads and initialized to true, it switches to false at the first pair not satisfying symmetry requests. Other pragmas that could fit well for its conditions were added, i.e. *#pragma omp parallel for shared(symmetric)*, *#pragma omp critical* and *#pragma omp cancel for* to interrupt the for in case of non-symmetry detected before the end of the full loop. Race conditions are avoided in both scenarios: thanks to the critical section while updating *symmetric* and maintaining *c* as a private variable in all iterations.

D. Timecheck_h and main file

The header file timecheck.h is used to compute time by giving a struct named *Timer* with default tools that can be really useful in many occasions. The main file can be personalized and structured in many ways, and it is programmers' responsibility to guarantee the best user experience using desired function calls. *

III. EXECUTION AND RESULTS ANALYSIS

Although the program could run quite everywhere, the testing procedure and comparisons reported in this deliverable are all based on different runs on the same system, which is one node of the HPC Cluster open for research purposes to members of the University of Trento. The cluster has plenty of space available for tests, as it is composed totally by 65 TB of total RAM, as well as a stunning total theoretical peak performance evaluated to be 478.1 TFLOP/s,. In order to run programs on that platform, we need to open an interactive session or put jobs in queue with a pbs file. I chose from the beginning to use pbs, as they are easier to launch and can be distributed easily to public and re-used with small changings. More information related to Pbs files can be found on my GitHub.

A. Measuring time

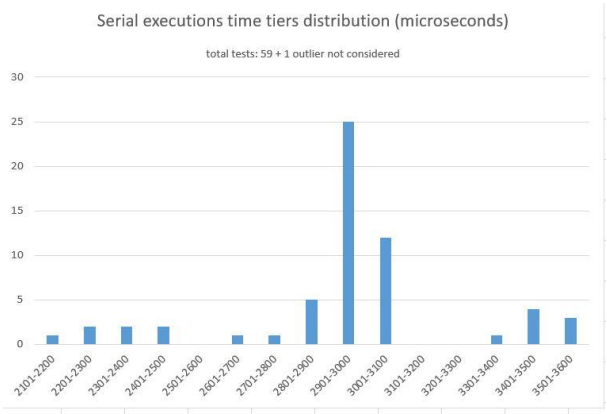
The first part of testing consists in evaluating individual time of functions in single separate runs on different matrixes. In this way, we can be sure that data locality and other caching possibilities could not affect our results. As a matter of fact, in many previous tries I did, I noticed how measurement of time

**for further and more specific information see the README on Github https://github.com/luciangorie/int.ParallelComputingDeliverable1.git*

appeared inferior when applying transposition on the same matrix more times.

The first comparison we need is just a comparison between times of the three implementations for both functions (more than 50 tries, removing outliers) ,keeping the same matrix size equal for every case for now (N=1000) and also same number of parallel processes when threads are used (threads=8).

Let's analyse results for *matTranspose()*. Here is a bar chart showing the distribution of time measures.

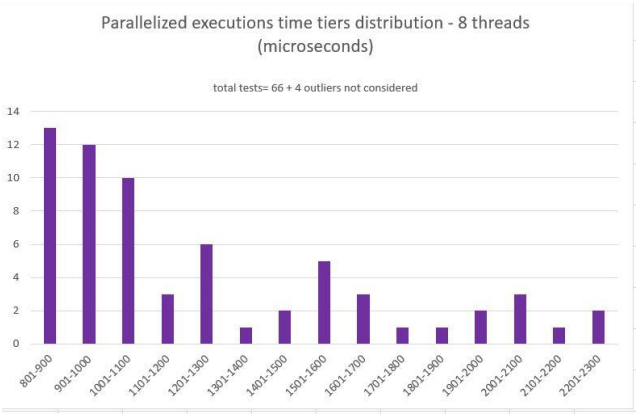


As you can clearly see, this first distribution appears similar to a Gaussian distribution, with most executions between 2801-2900 microseconds. We avoided to consider just one outlier out of 60 that was extremely higher than the others. The arithmetic mean is 2955 µs, and the standard deviation close to 300 µs, which is relatively low, meaning that the values obtained do not differ too much from the mean. For this reason, we can say that the function is reliable and consistent, although the average time could be improved.

As a matter of fact, the implicitly parallelized transposition goes faster than the serial version. The mean of 59 tries (like before I had to eliminate one value for the same reason) is pretty surprising as it is 1388 µs. Like in the other case, we identify one main column that depicts the most frequent scenario, between 1201 and 1300 µs. Standard deviation is a little bit bigger.

Figures 6: time needed for checkSym(), quite surprisingly the best option is the one implicitly parallelized,

arithmetic mean for checking symmetry (µs)		
serial	-	2241
implicitly parallelized	-	1275
parallel	-	1630



Also in the last case, 66 tries are put in a bar chart. What jumps into eye, is that we don't have a main column. All executions are widely distributed in more time slots. This result changes the standard deviation making it 439 µs, which is 46% more than the one of serial case. However, parallelization acts as expected, even with just 8 threads, the mean drops down to 1250 µs, which is not extremely lower than the implicit one, but is a good point to start to make some intermediate considerations: if it is true that we did not get an impressive speedup using more parallel processes, it is also true that we got many tries under 1ms which is an important border.

As regards *checkSym()*, the best option is the one with -o2 as a compiler flag (figure 6). The model with OpenMP can't guarantee enough gain, since computational complexity of a series of if -clauses is very low compared to the transpose.

B. Calculating reliable intervals with T-Student table

The t-student distribution is very similar to the gaussian one, but it can be distinguished for its heavier tails and for the possibility of weighing the amount of data collected. It is more suitable in cases in which number of findings is limited, like in our research. We would like to study a range close to the arithmetic mean that covers 80% of probability in order to have a better understanding of results. [6]

Figure 8

serial	-	[2904 - 3005] µs
implicitly parallelized	-	[1329 - 1446] µs
parallel	-	[1180-1321] µs

As we could predict, all functions operate in distinguished time slots in their interval between cumulated probability is 0,1 and 0,9. These values also confirm the hypothesis that even if parallel transposition had a more relevant variance with more disperse data, can still be valid. If we start to change conditions or change operations in matrixes, parallelization with OpenMp could have great potential.

C. Analysing the approaches in different matrices' size

As stated before, one function does not necessarily work better than the others in all scenarios. To understand functions' operating behaviour, we have to test them more times with different matrixes' size N. We change N opening *matrice_h* as it was hard coded. We still keep threads=8.

In this case, creating a table is the best way to visualize results. In the slots of the table (figure 8) is reported the average of between 5 and 10 tries.

**for further and more specific information see the README on Github https://github.com/luciangorie/int.ParallelComputingDeliverable1.git*

AVERAGE TRANSPOSITION TIME (microseconds)				
value of N (total float numbers)		serial	implicitlyParallelized	parallel with threads=8
16	256	<1	<1	15
32	1024	2.5	1	20,5
64	4096	8	3,5	45
128	16384	31,5	13	123
256	65536	145	184	114
512	262144	822,5	959,5	454,5
1024	1048576	9.943	5.384	2.711,4
2048	4194304	43.342	24.253	10.871
4096	16777216	210.355	167.206	60.600,50

Figure 9: average times for three versions of transposal

Each function shows a trend: the serial time grows more as the matrix size increases, the implicitly parallelized one too, even if not so extremely. On the contrary, parallel function's time of execution grows with regularity. What we learn from this, is the fact that calling parallelization techniques with threads costs in term of time, used to open and manage threads, and pays back if we have bigger matrixes: OpenMp starts to have an evident impact from 512 x 512 matrixes on. Looking at data, it could seem useless to have an implicitly parallelized function like mine, as it just shows advantages on the serial when the fully parallel with OMP gets more efficient (overtaking it), however I should remind that, as the solutions could work on many systems, the implicitly parallelized one still is excellent if we take care of resources needed, like it in everyday computing often happens.

IV. UNDERSTANDING RESULTS AND FURTHER ANALYSIS

The goal of this last paragraph is to understand the state of what was developed with scientifically recognized instruments. If before we had a look on data collection and how to visualize them, now it is time to focus on their interpretation. From now on, we will just consider the transposition function, since it is more adequate to the further analysis for its wider computational impact.

A. Speedup

Speedup is the most common evaluation that can be done in parallelization experiments. With speedup calculation we would like to quantify how one function of transposal is faster than another one. Although it can be defined by a number, I preferred to used percentages. [3]

The theorical speedup (Amdahl) for *matTransposeOMP()* is **1/(1-P)+(P/Nc)** and considering we parallelized the full function **P=1**, so it becomes just **1/(1/Nc)** which is **Nc,** the number of cores, which is the same as the number of threads so Nc=8. So, it is 8 times faster **(87%).** The theorical speedup for *matTransposeIMP()* is not defined, but goes around **25-30%** most of the times.

SPEEDUP IN DIFFERENT N size *(shown in percentage (new - old)/old*100%)*			
value of N (total float numbers)		speedUp implicit	speedup Parallel with threads=8;
512	262144	*no speedup*	44,7%
1024	1048576	45,9%	72,7%
2048	4194304	44,0%	74,9%
4096	16777216	20,5%	71,2%
average:	-	**36,8%**	**65,9%**

Figure 11: speedup with 4 different N sizes and average

The empirical speedup is calculated in relation to the serial execution for four chosen N sizes. Figure 9 contains speedup for each size for both implicit and parallel functions. Last line shows the arithmetic mean in order to have a better general idea. Speedup looks like the one we expected, 36,8% by just using *-o2* and small changings is a striking result that demonstrates the utility of compilation flags. The speedup percentage of the function using threads is not as high as the theorical one but even not too far from it, considered time needed for opening and managing threads. As a matter of fact, if we wouldn't consider the matrix with 512 values per row, which is the one most affected by threads overhead, the speedup would be more than 70%.

B. Scalability and Performance

Scalability is also a very important indicator (the most relevant in HPC) and it is defined as the measure of a system's ability to increase or decrease in performance and cost in response to changes. Strong scalability studies the way performance improves with more resources available; weak scalability checks how performance remain stable if we increase both problem size and system capability. [4] [5] I decided to opt strong scalability (keeping N=1000), and to plot a graph running the same program with different threads, from 2 to 64, taking the average of some tries.

Then, in order to see how much the slope is close to -1, we need to convert both x and y – axis values to their ln (). To maintain visibility, graphs are left on my GitHub. [7] We must recognize that we are not close to a good level of scalability, with negative slope just a little bit more than 0,3. However we should also pay attention to the fact that biggest count of threads information is misleading. If we just evaluate scalability until 16 threads, it is the slope is -0,57 which is not that worrying. I believe that this is normal and reflects the nature of the existence of OpenMP, which is to guarantee simple multithreading. If we begin to consider system with lots of CPUs/ALUs, other relevant techniques may win in term of scalability.

To conclude, we would like to see the comparison between system and program best performance. As reported before, the HPC cluster has a peak of 478 TFLOP/s. Program best average performance with N=1000 is the one with 16 threads, measuring 543 µs. So, peak performance in FLOPs/s is **(500x1000)-500 so 1000x499 loops x 3 operations/loop**, which is **1497000 FLOPs /543 x 10 $^{-6}$ s = approx. 2,8 MFLOPs.** If we can estimate the peak performance of core used (16) as 16 out of 7674 cores in percentage of that we find maximum (ideal) system speed as **956 GFLOPs** which still remains wildly extreme if compared to the result got before. Interpreting this fact, we can be sure that the limit of the transpose is definitely not the system, as it disposes of much more performance. It is more probable the function is just memory-bounded. It sounds realistic, since big matrices contain data that must be swapped continuously in the cache*.

V. FINAL CONSIDERATIONS AND CONCLUSION

The approaches analysed show how parallelization and other techniques can effectively make the program gain speed and efficiency. The solutions proposed may lack of complexity and elegance, but they are based on strong foundations, such as the possibility of customization combined with a reduced need of external libraries, that avoids useless stress in the compiling procedure and security issues. Even though we did not find a clear scalability, functions can operate on different systems, with a general speed working in time ranges close to the ones we could expect from theorical previsions.

**for further and more specific information see the README on Github https://github.com/luciangorie/int.ParallelComputingDeliverable1.git*

REFERENCES

Between brackets numbers referred to references.

[1] Manifold Software, How Matrix Filters Work, https://manifold.net/doc/mfd9/how_matrix_filters_work.htm, visited on 20/11/2024

[2] Free software foundation Inc., Options That Control Optimization, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, visited on 20/11/2024

[3] Stack Overflow, how to calculate percentage improvement in response time for performance testing,

https://stackoverflow.com/questions/28403939/how-to-calculate-percentage-improvement-in-response-time-for-performance-testing

Visited on 20/11/2024

[4] Luxe Quality, How to Do Scalability Testing: And Why It Matters?, https://luxequality.com/blog/how-to-do-scalability-testing/, visited on 27/11/2024

[5] Gartner, Scalability, https://www.gartner.com/en/information-technology/glossary/scalability#:~:text=Scalability%20is%20the%20measure%20of,application%20and%20system%20processing%20demands , visited on 26/11/2024

[6] Table of T-Student distribution, https://image.slidesharecdn.com/t-distributiontable-170723122602/95/t-distribution-table-1-638.jpg?cb=1500812899

[7] *Following as an example:* Natalie Perlin, Joel P. Zysman, Ben P. Kirtman, Rosenstiel School of Marine and Atmospheric Sciences, University of Miami, Miami, FL, 33149 2 - Center of Computational Science, University of Miami, Coral Gables, FL, 33146, Practical scalability assessment for parallel scientific numerical applications, section III letter E, https://arxiv.org/pdf/1611.01598

[8] (EXTRA SOURCES USED FOR THE DELIVERABLE)

De Natele Francesco, TM-p1 Tecnologie Multimediali, Bachelor Degree in ICT Engineering, University of Trento,2024.

Vella Flavio, L5-Modeling and Benchmarks, Introduction to parallel Computing, Bachelor Degree in ICT Engineering, University of Trento,2024.

Eigen, https://eigen.tuxfamily.org/index.php?title=Main_Page, visited on 21/11/2024

Deakin, Mattson, Pogramming your GPU with OpenMP, digital version available at

https://didatticaonline.unitn.it/dol/pluginfile.php/1884066/mod_resource/content/1/Scientific%20and%20Engineering%20Computation.Tom%20Deakin%2C%20Timothy%20G.%20Mattson%20-%20Programming%20Your%20GPU%20with%20OpenMP_%20Performance%20Portability%20for%20GPUs-The%20MIT%20Press%20%282023%29.pdf , 2023

**for further and more specific information see the README on Github https://github.com/luciangorie/int.ParallelComputingDeliverable1.git*