Matrix Transposition using Implicit and Explicit Parallelism OPENMP

Francesco Nodari ID: 234462
University of Trento
dept. di Ingegneria e Scienza dell'Informazione
Trento, Italy
email: francesco.nodari@studenti.unitn.it
Git repository: 1

***Abstract*—This project aim is to understand the implicit and explicit parallelism in order to optimize two different problems , the transpose and the symmetric check of a square matrix. Analyze the performance and compare it with the sequential code. The codes and the results are available at the Git repository[1].**

I. Introduction

A. Transpose of a matrix

In linear algebra the transpose of a matrix A is the matrix A^Tderived by making the first row of A the first column of A^T, the second row of A the second column of A^T, etc. In other words, when taking a transpose, the rows and columns are interchanged. Another way of saying this is that the subscripts have been interchanged, that is, if A^T =B =[$b_{i,j}$]. Then [$b_{i,j}$]=[$a_{j,i}$]. By using this definition, we can deduce that a matrix is symmetric when it is equal to its transpose, so A=A^T .

B. Project's objective

This project objective is to write a c++ program that sequentially computer the transposition and the symmetric check of a squared matrix of floating point numbers; then trying to improve it using implicit and explicit, OPENMP, paralleism. These programs will be tested by varying the matrix size (from 2^4 to 2^{12}) ; the number of threads (from one to sixty-four) and by measuring the time it takes to compute the transposition. Then it will be computed the strong scaling and the efficiency in order to analyze the performances.

II. State of the art

Matrix transposition is a fundamental matrix operation of linear algebra and arises in many scientific and engineering applications. On a uni-processor, an algorithm involving a transposed matrix may not actually require the matrix data to be transposed in physical memory. Instead, it may be accessed simply by exchanging the row and column indices. However, in a distributed memory multiprocessor environment, we cannot simply interchange the global row and column indices. Instead, the data must be physically moved from one processor to another. Transposition of a matrix is a redistribution of its elements. Many researchers have considered the problem for different architectures. 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. O'Leary implemented an algorithm for transposing an NX N matrix on a one-dimensional systolic array. Azari, Bo janczyk and Lee developed an algorithm for transposing an MX N matrix on an NX N mesh-connected array processor, Johnsson and Ho presented an algorithm for a Boolean n-cube, or hypercube. Modern high-performance computer architectures consist of a hybrid of the shared and distributed paradigms: distributed networks of multicore processors. The hybrid paradigm marries the high bandwidth low-latency interprocess communication featured by shared memory systems with the massive scalability afforded by distributed computing. One of the obvious advantages of exploiting hybrid parallelism is the reduction in communication relative to the pure MPI approach since messages no longer have to be passed between threads sharing a common memory pool. Another advantage is that some algorithms can be formulated, through a combination of memory striding and vectorization, so that local transposition is not required within a single MPI node (e.g. the multi-dimensional FFT1). The hybrid approach also allows smaller problems to be distributed over a large number of cores. A final reason in favour of the hybrid paradigm is that it is compatible with the modern trend of decreasing memory/core: the number of cores on recent microchips is growing faster than the total available memory. This restricts the memory available to individual pure MPI processes. This project implements aim is not to find new algorithms or solutions to improve this problem, but it tries to optimize existing approaches and to observe, understand and analyze using strong scaling and efficiency the differences between them.

III. Methodology

A. Sequential implementation

The implementation of the two problems has been implemented in a c++ program, as stated before. The symmetric check has been implemented inside a function called **checkSym**; this contains two nested for loops and inside the inner loop an if statement. In each iteration the if statement will check if the value in the position $M_{i,j}$ is not equal to the value in the position $M_{j,i}$ (**matrix[j * n + i]**!=**matrix[i * n + j]**); if this statement is true, then a local variable **check** is

set to false. The transposition has been implemented inside a function called **matTranspose**; this contains two nested loops which will copy the value of the original matrix in the positions $M_{i,j}$ and will paste it in the output matrix in the position $T_{j,i}$ (**trans[j * n + i]=matrix[i * n + j]**) .

B. Implicit Parallelism

The implicit parallelism is a technique which uses the ability of the compiler to recognize opportunities for parallelization and implement them without being told to do so. 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. The symmetric check has been implemented in a function called **checkSymImp**; this contains two cycles consisting in two nested for loops each. The first cycle iterates from the first row till half the number of rows and for the numbers of columns, the second iterates the other half of the rows and for the number of columns; inside each there is an if statement which will check if the value in the position $M_{i,j}$ is not equal to the value in the position $M_{j,i}$ (**matrix[j * n + i]!=matrix[i * n + j]**); if this statement is true, then a local variable **check** is set to false. The matrix transposition has been implemented in a function called **matTransposeImp**; this has been implemented like the checkSymImp function. This function contains two cycles consisting in two nested for loops each; the first cycle iterates from the first row till half the number of rows and for the numbers of columns; the second iterates the other half of the rows and for the number of columns. At each iteration, each set of loops will copy the value of the original matrix in the positions $M_{i,j}$ and will paste it in the output matrix in the position $T_{j,i}$ (**trans[j * n + i]=matrix[i * n + j]**) .

C. Explicit Parallelism

Up to this point the proposed solutions have always been executed from a single process; using OPENMP it is possible to explicitly parallelize the written code using multiple threads. The **CODES** folder in the repository will contain five different programs concerning the explicit parallelism; all the files have the same functions but differs in the compiler flag assigned to each in the **main.pbs** file. A '**Compiler Flag**' in Computer Science refers to an additional optimization settings used during the compilation of code to enhance performance or customize behavior. This was done to underline the differences in performance using the same code and just changing the flag. The symmetric check has been implemented in a function called **checkSymOMP** using the same methodology used in the checkSymImp but using different **OPENMP pragmas** which let you define parallel regions in which work is done by threads in parallel. The first is **#pragma omp parallel num_threads(thr)** which lets the compiler know that the part inside the curly bracket has to be done in parallel and using a define amount of threads set by using the **thr** variable. The second is **#pragma omp for schedule(dynamic)** and is located before each two nested for loops. This tells the compiler that the code that it has to parallelize is a for loop and, using the **schedule(dynamic)**, that each thread executes a chunk of iterations then requests another chunk until none remain. The matrix transpose is done inside a function called **matTransposeOMP** using the same methodology used in the matTransposeImp function, but inside each loop at each iteration each set of loops will copy four values of the original matrix in the positions $M_{i,j}$ and will paste it in the output matrix in the position $T_{j,i}$. Since the performances of a parallel code changes changing the number of threads in the main function is implemented a for loop which iterates from one to sixty-four and the iterator is passed to both the checkSymOMP and the matTransposeOMP. Then the functions will put the value in the first pragma whose will set the number of threads in accordance using the **num_threads()** pragma. The program called "**OMP_no_flag.cpp**" will be compiled without any flag and will be used as a benchmark. The one called "**OMP O1.cpp**" will be compiled using the O1 compilation flag; which enables the core optimizations in the compiler and provides a good debug experience. The program called "**OMP O2.cpp**" will be compiled using the O2 compilation flag; is a higher optimization for performance compared to O1. It adds few new optimizations, and changes the heuristics for optimizations compared to O1. This is the first optimization level at which the compiler might generate vector instructions. It also degrades the debug experience, and might result in an increased code size compared to O1. The amount of loop unrolling that is performed might increase. Vector instructions might be generated for simple loops and for correlated sequences of independent scalar operations. The one called '**OMP O3.cpp**' will be compiled using the O3 compilation flag; is a higher optimization for performance compared to O2. This optimization level enables optimizations that require significant compile-time analysis and resources, and changes the heuristics for optimizations compared to O2. O3 instructs the compiler to optimize for the performance of generated code and disregard the size of the generated code, which might result in an increased code size. It also degrades the debug experience compared to O2.The differences when using O3 as compared to O2 are: The threshold at which the compiler believes that it is profitable to inline a call site increases. The amount of loop unrolling that is performed is increased. More aggressive instruction optimizations are enabled late in the compiler pipeline. At last the one called "**OMP Ofast.cpp**" will use the Ofast compilation flag; which disregard strict standards compliance. Ofast enables all O3 optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on -ffast-math and the Fortran-specific -fno-protect-parens and -fstack-arrays.

IV. Experiments

Theese experiments have been collected in the Git repository 1, and conducted with the followig comfiguration: Intel(R) Xeon(R) Gold 6252N, the architechture is x86_64 with 96 CPUs. The compiler used is **g++-9.1.0** . The experiments have been conducting using the methodologies explained before.

In every simulation the matrix is initialized with random float numbers using the function **randon_float** located in the **functions.h** file (located in the repository). The size of the matrix, as said before, is incremented from 2^4 to 2^{12}.

V. Results

Before discussing the results as a premise it will only be shown the results concerning the dimension 2^{12} ; the remaining graphics and data are located in the git repository in the **DATA AND GRAPHICS** folder.

A. Implicit Parallelism

As stated before this report will not show the results concerning the implicit parallelism simulations; although all the results are located in the git repository 1 . Analyzing the results, it's noticeable that for small matrices, the serial code is faster than the implicit counterpart; this is due to the fact that in the implicit there are more instruction and there is no advantage in parallelizing because the amount of memory that the compiler is using is relatively small. For medium size matrices, the situation is flipped due to the fact that there are more computations to do linked to the increased matrix size; and so parallelizing help improving the performances. For bigger matrices, the performances are almost equal, but it's noticeable that the serial code is slightly faster; that's because the implicit code, although is faster during the computational part, it suffers from cache misses and, when one happens, it has to stall the computations.

B. OPENMP

Looking at the speedup graphics 1 , it shows clearly that the compiler flag improved the performances and the peak performance is obtained with four threads and using the Ofast flag; whose uses all the functions enabled from the O1, O2 and O3 flags all together and disregard strict standards compliance. Analyzing the efficiency graphic 2, it's noticeable that from eight threads, there is a significant decline in performances. This project matrix size is relatively small and it needs a small amount of threads to compute the transposition; so if the compiler has a larger number of them; when it organizes the parallel sections, it has to organize more threads than it needs, and the excess ones cause overhead that slows the compiler. But it's important to notice that the efficiency of the "OMP no flag" , used as benchmark, is very low compared to the others; this underlines the importance of flags and their improvement in performance.

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. This study represents a step toward understanding best practices for optimizing parallel applications, offering insights for further

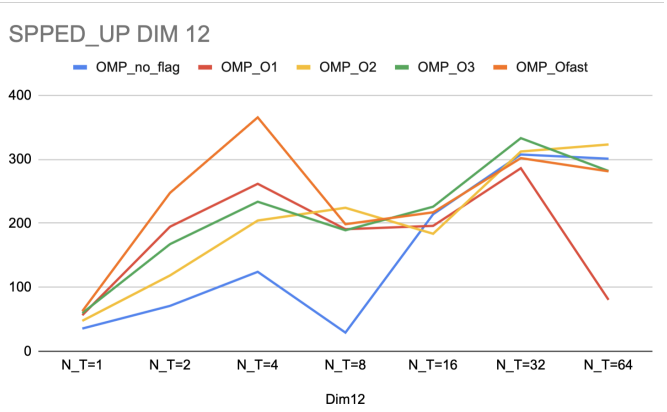

Fig. 1. SPEEDUP DIM 12

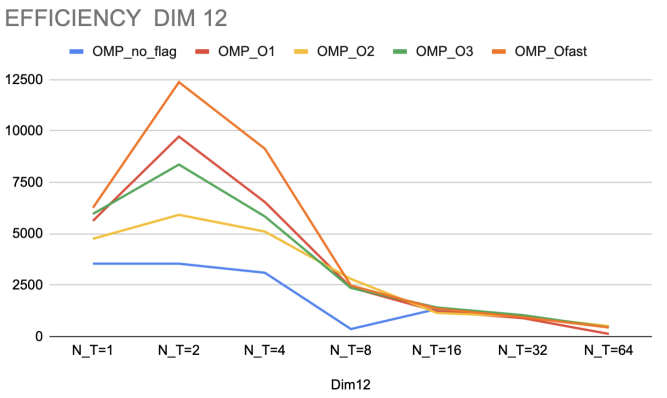

Fig. 2. EFFICIENCY DIM 12

		Speed_up explicit parallelism			
Dim12	**OMP_no_flag**	**OMP_O1**	**OMP_O2**	**OMP_O3**	**OMP_Ofast**
N_T=1	35,44528162	56,27952306	47,54815266	59,65038024	62,62365829
N_T=2	70,8719201	194,5846514	118,4222779	167,3475241	247,4979282
N_T=4	124,0803816	261,6061647	204,2084411	233,5976423	365,397839
N_T=8	29,09994961	190,7000332	224,133356	189,0286367	198,493229
N_T=16	214,0803619	195,8987708	183,5647097	225,803427	217,0372288
N_T=32	307,3760364	285,9503214	312,0401181	332,9509219	301,7368007
N_T=64	300,8217809	80,32161004	323,0360007	281,6279712	281,1040909

Fig. 3. Speedup dim 12 data

		efficiency explicit parallelism			
Dim12	**OMP_no_flag**	**OMP_O1**	**OMP_O2**	**OMP_O3**	**OMP_Ofast**
N_T=1	3544,528162	5627,952306	4754,815266	5965,038024	6262,365829
N_T=2	3543,596005	9729,232568	5921,113893	8367,376207	12374,89641
N_T=4	3102,00954	6540,154117	5105,211026	5839,941057	9134,945974
N_T=8	363,7493701	2383,750415	2801,66695	2362,857959	2481,165363
N_T=16	1338,002262	1224,367318	1147,279436	1411,271419	1356,48268
N_T=32	960,5501137	893,5947542	975,1253691	1040,471631	942,9275022
N_T=64	470,0340327	125,5025157	504,7437511	440,043705	439,2251421

Fig. 4. Efficiency dim 12 data

research on optimization and scalability in multi-core environments.

REFERENCES

[1] Git Repository : Link to the repository

[2] Adaptive Matrix Transpose Algorithms for Distributed Multicore Processors. John C. Bowman,University of Alberta, and Malcolm Roberts,University of Strasbourg. link to the page

[3] 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. link to the page

[b4] PEC Flag Description for ARM Compiler. Link to the page