Matrix Transposition using MPI

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 Message Passing Interface (MPI) method of 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 different MPI routines. Theese programs will be tested by varying the matrix size (from 2^4 to 2^{12}) ; the number of processors and by measuring the time it takes to compute the transposition. Then it will be computed the strong and week 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 and weak 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 the 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]**) . It has been implemented a second sequential function which transposes a matrix in blocks called **matTransposeBlock** . This has been coded using four two nested for loops, each nested block will operate on a quarter of the matrix and will transpose its assigned chunk by simply using the same logic as in the matTranspose function before.

B. MPI implementations

Up to this point the program uses a single processor; with MPI we can parallelize the code using multiple processors and different MPI routines. MPI is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory. In the Git repository there are five different programs, each uses a different routine or implementation. The " **point to point Async.cpp** " file uses a point to point type of communication, it's the simplest form of message passing; this mechanism enables the transmission of data between a pair of processes, one side sending (source), the other receiving (destination). Using asynchronous send, the sender does not get an information message when the message it's received, it only knows when it is sent. This has been implemented using a single and two nested for loops; the first iterates on the column number and sends the columns of the original matrix, this is done by using the process number 0 only (using **MPI Send** function) to the process number 1; whose will receive the message, (using **MPI_Recv** function), in a vector inside two nested for loops, theese iterates on the numbers of columns, the first, and rows, the second, and will paste the values in the transposed matrix with the indexes swapped. The " **Buff Non Block.cpp** " file uses still a point to point type of communication but it implements a Buffered-Non-Blocking type of communication, with this type operations operations the sender post data to a buffer, but the sender process does not wait for the completion of the communication. It proceeds with its execution while the communication operation is ongoing in the background. This is implemented as the previous code, the difference is that we use the **MPI Wait** function (MPI_Wait(&request_send,&stat)) to let the sender wait before sending or receiving the next column and we use the **Isend** and **Irecv** functions which initiates a non-blocking send and receive. The "**point to point NBB Sync.cpp**" file uses a non-buffered-blocking type of communication, the sender directly sends data and is blocked until the receiver has successfully received the message, using synchronous send, requires the receiving process to provide a "ready" signal, in order to initiate the send; therefore, the receiving process will always be ready to receive data from the sender. This is implemented as the codes before with the difference that it has been used the **Ssend** function for the sender which initiates a blocking synchronous send. Up to now the methodology used was similar; the last two codes however use a different type of routine which is the **Scatter and Gather**. The scatter divides a big array into a number of smaller parts equal to the number of processes and sends each process (including the source) a piece of the array in rank order. The gather does the opposite, receives data stored in small arrays from all the processors (including the source or root) and concatenates it in the receive array in rank order. The "**Scatter Gather.cpp**" firstly makes sure that the number of process is a perfect square and that the size of the matrix can be divided by the squared number of processors, if this two conditions aren't satisfied the program will not run. If, on the other hand, the conditions are satisfied the program uses the scatter function to send equal blocks of the matrix to the processors, using a two local sub-matrices to help the computation. Then with the gather function that collects all the processes and puts the values of each in the final transpose matrix using two nested for loops. The "**Scatter Gather2.cpp**" operates in the same way as the code just explained but it differs size given to each process; because in this case the number of processor won't be squared; the compiler will simply check if the matrix size can be divided by the number of processor and give to calculated size to each. In doing this each CPU will get a smaller portion of the matrix. The symmetric check is done after the transposition and each file uses the methodology described for it put instead of doing the transposition it checks if the chuck send matches with the corresponding chunk int the transposed matrix, if it does sets a boolean variable **check** to true if not to false. One of the main challenges faced was the coordination of processes especially in the "**Scatter Gather.cpp**" and "**Scatter Gather2.cpp**" because the number of processes operating on the transpose matrix is bigger so when copying values from the local buffer to the transpose could happen that the rows and columns were not in the correct position and also dirty reads were common; the first problem was solved by fixing the sizes of the local vectors passing the data; the second was addressed by adding a local 2D vector, which ads overhead because the value from the receive buffer in the scatter function have to be copied to the vector but it reduces dirty reads when collecting the processes using the gather function.

IV. Experiments

Theese experiments have been collected in the Git repository [], 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** , with **mpich-3.2.1**. 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}. For the weak scalability the matrix dimension in incremented from 2^4 to 2^{10}.

V. Results

Before discussing the results of this project as a premise it will only be shown the results related to the dimension 2^{12} referring to the speedup, it will also be shown the speedup for the dimension 2^8, and efficiency; the remaining results and

graphics are located in the Git repository1. Before discussing and analyzing the results of the MPI routines it is due to point out the differences in performance between the two serial function of transposition. As it is shown in 5 the block transposition is slower than the normal one up until the dimension 2^6. That's because it uses four block with two nested for loops in each, so dividing a small matrix in four smaller equally sized ones adds only useless time. For bigger matrices on the other hand this approach uses the implicit parallelism technique, the compilator understands that the loops are isolated from each others and assigns a thread to each to improve the performances. Considering the speedup, which is a measure of how effective parallelization is with respect to the number of cores used, shown in the first graphic 1, it's clearly visible the fact the MPI routines have not brought an improvement, whereas with the openMP parallelization done in the previous project it improved it substantially. The graphic also shows that the "Buff non Block" and the "Point to Point NBB" with four threads were very close to the serial one respect to the others; but as the number of threads increases the speedup of every routine declines; this is also noticeable in the efficiency graphic 2 . It's due to notice that the "Scatter Gather" and "Scatter Gather2" were not always possible to run because it was not possible to assign an integer size to each or the process number was not a perfect square. It's important to notice that for the dimensions 2^8(graphic 3), 2^9, 2^{10} the speedup of theese routines, except for the "Scatter Gather" and "Scatter Gather2", were greater than the serial one; but in accordance to the graphic shown for the dimension 2^{12}, the efficiency is compared to the serial one very poor, it's even poorer if compared with the openMP parallelization. Considering the weak scaling graphic 4, which describes the ability of a system to maintain roughly the same level of performance when adding additional resources; in terms of computing, this means that the application can take advantage of additional computing resources to serve more customers or process more data, while still achieving the same performance. The graphic shows that compared to the serial code the MPI routines worsen the performance as the number and size of processor increases and from 8 processors is almost canceled. Ideally an effective parallel algorithm weak scaling would be close to 1, indicating that the program efficiently scales with the workload and the number of processors. Theese poor performances could be linked to the fact that this method core is the message passing, theese could add an overhead and slow the program down; it is also compulsory to notice that MPI was designed to operate on distributed memory so each processor has its own memory and operates accessing it and when it's done it passes the results using messages to the other processors. That said it is necessary to point out that in this project uses a shared memory, so each processor has to copy its portion of data before operating it in, this adds a cost to the overall performance.

A. *MPI vs openMP*

OpenMP is an API that allows developers to easily write shared-memory parallel applications in C/C++ and Fortran. A shared memory model runs on a single system and utilizes its multiple processing units or cores to achieve concurrent computation using the same memory space. OpenMP provides high-level compiler directives. This includes **pragma omp parallel**, used to identify which parts of the code need to run concurrently **pragma omp critical** to identify critical sections. It then automatically deals with many of the low-level complications like handling threads (each core's line of execution) and locks (constructs used to avoid problems like race and deadlock). It even handles reductions and compiles the results of each thread into a final answer. **OpenMPI** is an implementation of the Message Passing Interface (MPI), used in distributed memory architectures. Unlike shared memory, described above, distributed memory uses a collection of independent core memory pairs that synchronize using a network, mostly found in supercomputers. This means that each core or node has a memory of its own and does not require locks like shared memory. However, synchronization is still required to distribute the computation and collect results and that is done through message passing. OpenMPI provides API calls such as `MPI_Send` and `MPI_Recv` to allow communication between computation nodes. Unlike OpenMP, here each computational unit has to send its results to a master and it manually compiles and aggregates the final result.

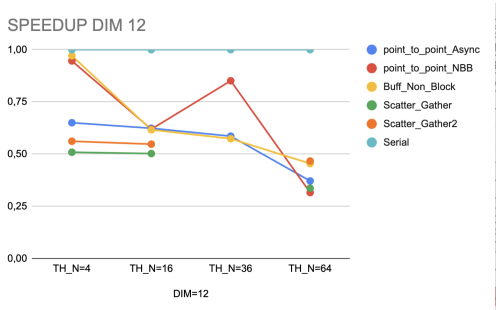

Fig. 1. Speedup DIM 12.

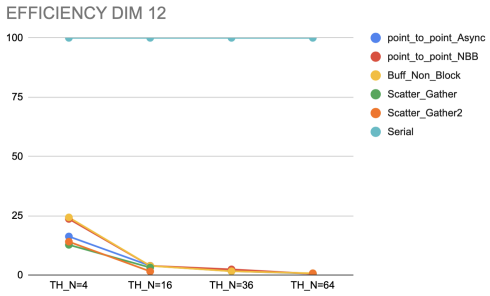

Fig. 2. Efficiency DIM 12.

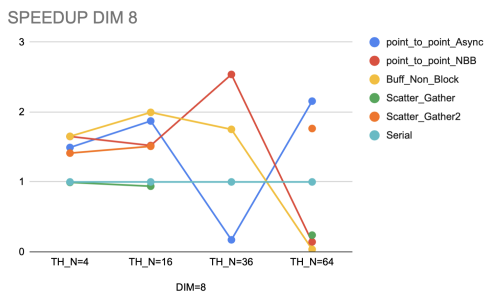

Fig. 3. Speedup DIM 8.

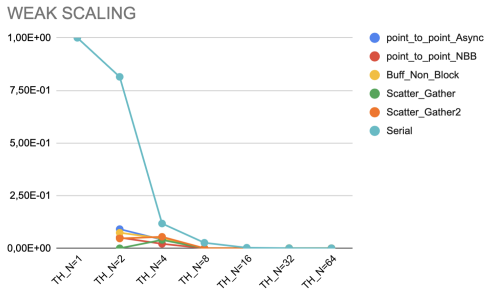

Fig. 4. WEAK SCALING

CONCLUSIONS

As we can see from the graphics and as was stated in the Results part the MPI programs are much slower than the sequential one. This is not desired because the aim of parallelization is to improve the response time of the program especially when is dealing with a large amount of data; this is due to the fact that MPI uses messages to operate and theese add a cost on the performance. It is due to notice that this method does not use a shared memory, each processor operates on its own memory, so, as said before, each processor has to make its own copies, this also adds up to the costs. MPI is a great method to program on distributed memory devices; if a shared memory is in place then openMP has to be preferred.

Serial	Normal	Blocks
DIM=4	2,41E-06	3,11E-06
DIM=5	2,96E-06	3,34E-06
DIM=6	2,05E-05	1,53E-05
DIM=7	9,24E-05	5,95E-05
DIM=8	0,000913588	0,000368582
DIM=9	0,00398396	0,00077586
DIM=10	0,0209209	0,00368798
DIM=11	0,0818043	0,0207799
DIM=12	0,391874	0,281668

Fig. 5. WEAK SCALING

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
[4] What is the message passing interface (MPI)? link to the page