# The performance of fast Givens rotations problem implemented with MPI extensions in multicomputers

L. Fernández and J. M. García
*Department of Informática y Sistemas, Universidad de Murcia, Campus de Espinardo s/n, 30080 Murcia, Spain*
*EMail: {lfmaimo, jmgarcia}@dif.um.es*

## Abstract

In this paper, issues related to implementing an MPI version of the fast Givens rotations problem are investigated. We have chosen this algorithm because it has the feature of having no predictable communication pattern. Message Passing Interface (MPI) is an attempt to standardise the communication library for distributed memory computing systems. The message-passing paradigm is attractive because of its wide portability and scalability. It is easily compatible with both distributed-memory multicomputers and shared-memory multiprocessors, with NOWs and also combinations of these elements. Currently, there are several commercial and free, public-domain, implementations of MPI. We have chosen the most common implementation of MPI called MPICH. In this paper we show the MPI algorithm of the fast Givens rotations and give some preliminary results about the performance in a network of personal computers. Our results will also point out the strength and weakness of the implementation.

## 1 Introduction

Many numeric calculation problems need to be solved in the shortest possible time, therefore requiring the availability of computers with high calculation power. In this sense, massively parallel computers have become the best alternative to achieve this objective. Their high processing speed is based on parallel execution of different processes which, properly combined, will produce the solution required. Parallel algorithm implementation is not immediate, as a great programming effort is often necessary, specially when these problems are specifically of sequential type. Another problem related to the parallel implementation is the election of the best parallel algorithm. Sometimes, the sequential and parallel behaviour is different. So, it is necessary to choose the algorithm with the best parallel execution.

This is the case of Givens and Householder transformations [4]. They are frequently used in many scientific applications, as lineal system resolution or eigenvalue problems. Among QR decompositions, Householder transformations are usually preferred when programming a serial computer, due

to their higher speed. However, Givens rotations are very well suited for parallel computers, because they exhibit a great potential parallelism. Moreover, there is an improved version of Givens rotations, known as fast Givens rotations [4].

In this paper, we focus on the fast Givens rotations problem. A Givens rotation can be defined by a transformation matrix $G(i,k,\theta)$ where $\theta$ is the rotation angle. The application of an $m \times n$ transformation matrix $G(i,k,\theta)$ to an $m \times n$ matrix A, annihilates the element $A_{ki}$, choosing the appropriate value of $\theta$. We have implemented this problem using the official standard in parallel computers, that is, the MPI extensions.

The goal of the Message Passing Interface is to offer a widely used standard for writing message-passing programs. The interface establishes a practical, portable, efficient, and flexible standard for message passing. MPI provides a wide variety of point-to-point and collective communication routines [1] [5]. Support is provided for *process groups*, so that a particular communication operation can be restricted to involve only a given set of processes. In MPI, a process is identified by a group and its rank within that group. A process may belong to several groups. In point-to-point communication, messages are regarded as labelled by a *communication context*, and a tag on that context. Communication contexts are means within MPI of ensuring that messages intended for receipt in one phase of an application cannot be incorrectly received in another phase. Communication contexts are managed by MPI and are not visible at the application level. Messages in MPI are typed, and *general datatypes* are supported. These may be used for communication array sections and irregular data structures.

Nowadays, the research community has a lot of interest in solving numerical problems using the MPI libraries. Moreover, it is necessary to evaluate the performance of the different MPI implementations. The most usual environments to implement the MPI paradigm are massive parallel machines (MPP) and networks of workstations (NOWs). In [3], we can see the performance of Householder and Givens transformations over a cluster of workstations. The performance of Givens and Householder transformations on vector computers is analysed in [7]. Also, we can find an MPI version of the BLACS routines with a deep explanation of the major changes in [9]. Finally, the issues of performance of some MPI implementations on workstation clusters can be seen in [8].

The goal of this paper is to code the fast Givens rotations algorithm with MPI extensions and to study the performance of a parallel implementation of this algorithm by using MPICH, a portable implementation of MPI. Our work environment is a network of personal computers communicating via Ethernet cards.

The next section gives a brief overview of the classic and parallel algorithms to triangularizate a matrix by fast Givens rotations. The implementation of the algorithm with MPI extensions is discussed. In section 3

we present our execution environment. In section 4 we show our experimental results we have obtained, and we analyse these results. Finally, some concluding remarks and future works are made in section 5.

# 2 The fast Givens rotation algorithm

## 2.1 The classic algorithm

A Givens rotation can be defined by a transformation matrix:

$$G(i,k,\theta) = \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ . & . & . & & . & & . \\ 0 & \dots & c & \dots & s & \dots & 0 \\ . & & & & & & . \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ & & & & & & . \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix}$$
$$\begin{matrix} & & i & & k & & \end{matrix}$$

where c=cos(θ) and s=sin(θ) for some θ. Givens rotations are clearly orthogonal.

Premultiplication by $G(i,k,\theta)^T$ amounts to a counterclockwise rotation of θ radians in the (i,k) coordinate plane. Indeed, if $x \in \Re^n$ and $y=G(i,k,\theta)^T x$, then

$$y_i = \begin{cases} cx_i - sx_k & j = i \\ sx_i + cx_k & j = k \\ x_j & j \neq i,k \end{cases}$$

From these formulas it is clear that we can force $y_k$ to be zero by setting

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}} \qquad s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}}$$

Thus, it is a simple matter to zero a specified entry in a vector by using a Givens rotation.

## 2.2 The parallel algorithm of fast Givens rotations.

Fast Givens transformations allow for each rotation to be performed using fewer multiplications than the classic algorithm. To do this, the current matrix

is kept in a factored form as DA where D is an m × m diagonal matrix. (In our implementation D is stored as a vector.)

```
start_time=MPI_WTime();
if (my_process_number == 0)
{
   load matrix
   for(i=1;i<number_of_processes;++i)
        MPI_Send(rows type i to process i);
}
else
   MPI_Recv(my type rows);

MPI_Barrier();
do
{  MPI_IProbe(&received)
   if (received)
   {
        MPI_Recv(row from other process)
        insert row in local matrix
   }
   if (rows_counter>1)
   {
        extract two rows from local matrix
        calculate and apply a rotation
        insert the first row in the local matrix
        calculate destination process of second row
        MPI_Send(second row to destination process)
   }
} while not finished
MPI_Barrier()
end_time=MPI_WTime()
```

Figure 1. Basic parallel algorithm with MPI extensions.

To obtain a parallel implementation of the algorithm, we must take into account that the rotations of rows from A are totally independent and can be applied in any order. The only requirement for applying rotation to a pair of rows is that the first non zero elements of both rows occupy the same column position. So, any pair of rows of the same type can be processed in parallel with any other pair. Processes do not need to communicate during the rotation process. However, a row must be transferred to another process after each rotation in most cases.

The algorithm to triangularizate a matrix A and its associated coefficient vector b, based on fast Givens rotations, the theoretical proof of the algorithm and more information about Givens rotations can be found in [4]. We have followed the studies developed by Duato [2] in order to apply fast Givens rotations in a distributed-memory system.

The parallel implementation of the algorithm requires as many processes as columns the sparse matrix has. If we define the type of row as the column position occupied by its leftmost non zero element, then it is well known that only rows of the same type can be rotated together. Then we distribute the rows among processes so that each process stores all the rows of the same type. After a pair of rows have been rotated, one of them increases its type, being sent to the corresponding process to be rotated again. Empty rows are discarded and the algorithm finishes when there is almost a single row in each process. As the rotation of a pair of rows cannot produce a row of a lower type, a token is passed through all the processes to determine when the triangularization program has finished.

Our programming model allows that the communication is carried out asynchronously and in parallel with the processing. Figure 1 shows the basic algorithm with the MPI extensions.

It is interesting to note here that the algorithm has not a regular communication pattern. The communication pattern is not fixed and is varying throughout the time.

### 2.3 MPI extensions used in the algorithm.

In the former parallel algorithm, we use several MPI functions. Next, we are going to describe them briefly.

- *MPI_WTime*: This function returns a floating-point number of seconds, representing elapsed wall-clock time since time in the past. The returned times are local to the node that called the function. We use this function to calculate the execution time of the algorithm.
- *MPI_Send* and *MPI_Recv*: These functions allow communication of the processes. MPI provides blocking and non-blocking send and receive functions. Non-blocking functions allow the overlap of message transmittal with computation. Moreover, there are several modes for point-to-point communication. The mode allows the user to choose the semantics of communication and to influence in the underlying protocol for the transfer of data. We use a blocking communication with the standard mode. In this mode, MPI decides whether outgoing messages will be buffered. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons.
- *MPI_Barrier*: This function blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call. We use this function in our algorithm with synchronization purposes across all processes.
- *MPI_IProbe*: This function allows incoming messages to be checked for, without actually receiving them. We use this function to select either to receive a new row or to rotate a pair of rows.

# 3 Description of our implementation.

## 3.1 Testing environment

The algorithm was executed on a cluster of PC 486DX4 at 133 Mhz. with 8Mb RAM and 1 Gb. of HD running LINUX. We used a public domain distribution of MPI, named MPICH which make use of sockets for communications.

The matrices used belong to the Harwell-Boeing library. This library consists of a lot of matrices used on the resolution of real problems in all sciences [6]. These matrices have the characteristic of being sparse.

## 3.2 Some features in the implementation of the algorithm.

To measure the throughput of the algorithm, initially we assigned each process a row type, that is, a process per column. Because of the size of the matrices we have worked with (squared of 512-1856 rows), we decided to represent them internally in a way that made good use of the memory.

The size of the matrices caused a great generation of processes. Since each process was assigned to a column, either each processor was busy only on context exchange, or the local memory was quickly consumed.

Because of this, we decided to assign more than one row type to each process, in order to simulate whichever of the processes having one from those row types.

In addition to this, we have exploited the matrix dispersion by implementing the sparse matrix abstract data type by means of linked lists, so the amount of memory used by the matrices is dramatically reduced. This causes an increment of the access time to an element (O(n) vs. O(1)).

The communication between processes is carried out by sending a row of the matrix. Because we use sparse matrices, each process only sends those elements distinct from zero in the message. In this way, we have a reduced communication time.

# 4 Experimental results

In this section, we show some preliminary results from the tests we have carried out. We have evaluated the fast Givens rotations algorithm with square matrices of several sizes, from 512 rows up to 1856 rows. Also, we have evaluated the sequential algorithm. In the following figures we plot the results for several processors (from 2 to 10) and also plot the result for the sequential execution (plotted for the one processor). However, it must be noted that we use two different algorithms for the sequential case and for the parallel case.

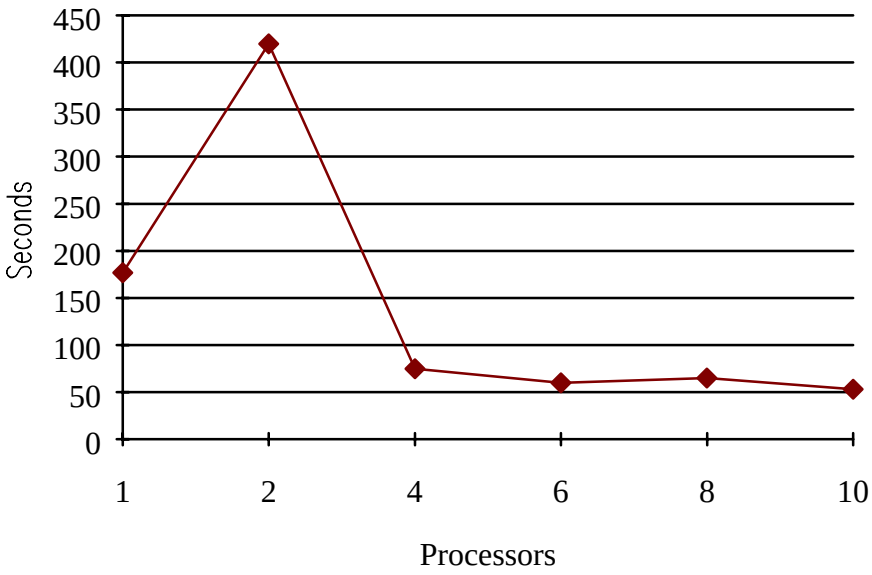


Figure 2. Execution time vs. processors with a 512×512 matrix.

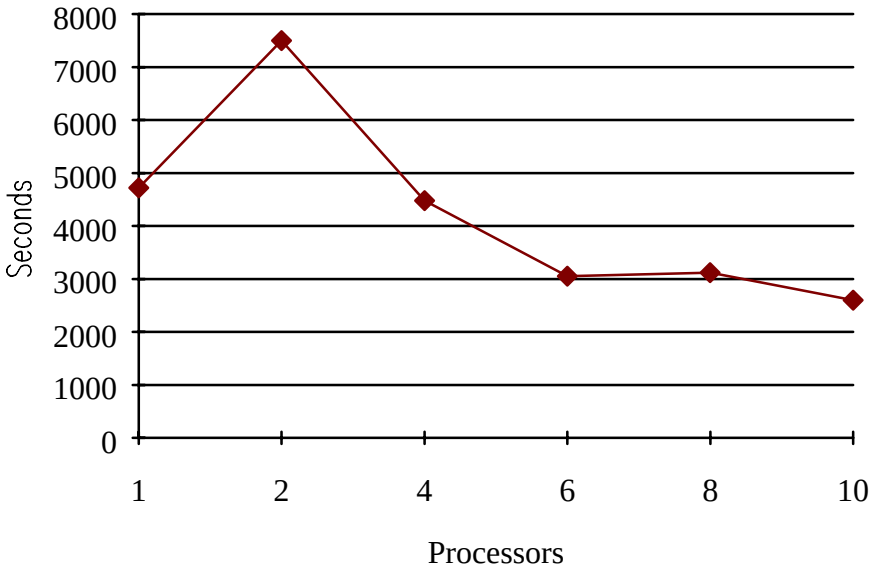


Figure 3. Execution time vs. processors with a 1374×1374 matrix.

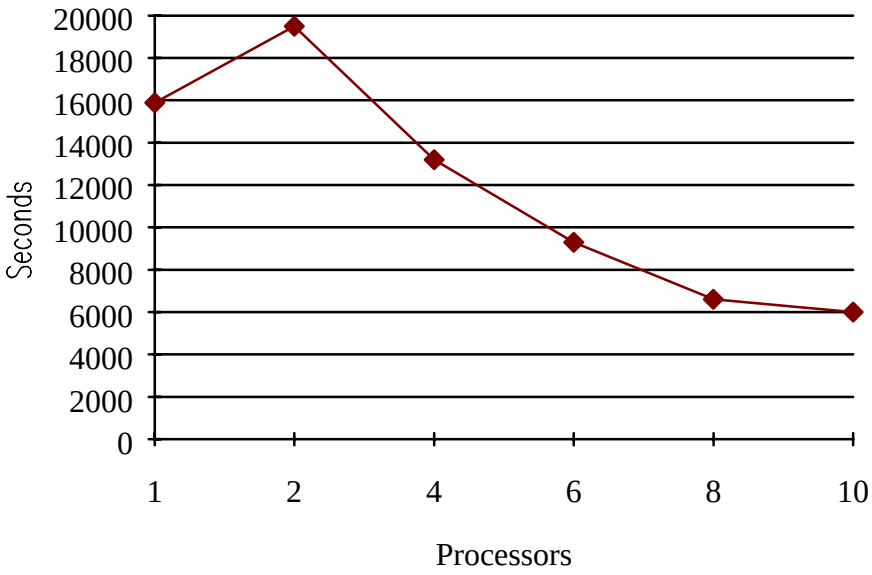


Figure 4. Execution time vs. processors with a 1856×1856 matrix.

We can notice that the performance of the parallel algorithm is better than the sequential one from four processors or more. The main reason for this improvement is the partitioning made : columnwise block striping, where each processor is assigned a rank of adjacent row types. So when a row changes its type to another one that belongs to the same rank, it is not sent through the network but internally stored into the local memory of the processor. In this way, the communication time dramatically decreases.

| Processors. | Theor. communic. | Real communic. |
|---|---|---|
| 2 | 8711 | 23 |
| 4 | 8711 | 62 |
| 6 | 8710 | 99 |
| 8 | 8708 | 134 |
| 10 | 8710 | 166 |

Table 1. Theoretical and real communications with a 512×512 matrix.

| Processors. | Theor. communic. | Real communic. |
|---|---|---|
| 2 | 43622 | 38 |
| 4 | 43468 | 123 |
| 6 | 44075 | 211 |
| 8 | 43405 | 287 |
| 10 | 43562 | 373 |

Table 2. Theoretical and real communications with a 1374×1374 matrix.

In Table 1 and Table 2 we can see an example of the great difference between the theoretical and real communications made during the algorithm execution. With this partitioning mode, we reduce the computation cost of the sequential algorithm by the number of processors without increasing the communication cost.

The mean length of the sent rows varies from 63 elements (512×512 matrix) to 120 elements (1856×1856 matrix) for the matrices used. This gives us a mean message length that varies from 252 to 480 bytes for the array of indexes and from 504 to 960 bytes for the array of values. By sending these two messages, we spend less time than by sending one message where the two arrays are joined.

Finally, we can observe that from a certain number of processors, the execution time is higher than the sequential case. This is caused by the increase of communications. In our tests, we have found that from 14 processors the parallel algorithm begins to get worse.

## 5 Conclusions and future works.

In this paper we have presented the MPI version of the fast Givens rotations problem. We have used this algorithm to the resolution of the matrix triangularization problem. We have implemented this algorithm with a few MPI extensions. We have used a standard blocking made to communicate the processes.

In this paper we evaluated the performance of the fast Givens rotations problem over MPICH, the most usual implementation of MPI.

Despite the high cost that involves the communication time in an ordinary network (Ethernet) using sockets, the computational cost of the algorithm ($O(mn^2)$) makes its parallel execution profitable.

Instead of using Ethernet, we plan to conduct our next tests in two ways: on a NOW using Fast Ethernet and on a parallel computer (IBM-SP2) which has a fast interconnection network.

Besides, we want to generalise the results to matrix sets with similar sizes and to study higher and non-squared matrices.

## References


[1] Dongarra, J. J., Otto, S. W., Snir, M. & Walker, D. A Message Passing Standard for MPP and Workstations. *Communications of the ACM*. July 1996, Vol. 39, No. 7, pp. 84-90.

[2] Duato, J. Parallel triangularization of a sparse matrix on a distributed-memory multiprocessor using fast Givens rotations. *Linear Algebra and its Applications*, 1989, 121, 582-592.

[3] Egecioglu, Ö. & Srinivasan, A. *Givens and Householder Reductions for Lineal Least Squares on a Cluster of Workstations*. Tech. Rep. TRCS95-10, Department of Computer Science, University of California, Santa Barbara, 1995.

[4] Golub, G.H. & Van Loan, C.F. *Matrix computations*, 2nd edition, The Johns Hopkins University Press, Baltimore, 1990.

[5] Gropp, W., Lusk, E. & Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Mass., 1994.

[6] Harwell-Boeing library. URL: ftp://ftp.cerfacs.fr/pub/harwell_boeing/

[7] Mattingly, R. B., Meyer, C.D. & Ortega, J. M. Orthogonal Reduction on Vector Computers. *SIAM J. Sci. Statist. Comput.*, 1989, 10, 372-381.

[8] Nupairoj, N. & Ni, L. M. Performance Evaluation of Some MPI Implementations on Workstation Clusters, *Procc. of the 1994 Scalable Parallel Libraries Conference*, October 1994, 98-105.

[9] Walker, D. W. An MPI version of the BLACS. *Procc. of the 1994 Scalable Parallel Libraries Conference*, October 1994.