Efficient Jacobi Algorithms on Multicomputers

Domingo Giménez[1], Vicente Hernández[2*] and Antonio M. Vidal[2*]

[1] Departamento de Informática y Sistemas. Univ de Murcia. Aptdo 4021. 30001 Murcia. Spain. (domingo@dif.um.es)

[2] Departamento de Sistemas Informáticos y Computación. Univ Politécnica de Valencia. Aptdo 22012. 46071 Valencia. Spain. (cpvhg@dsic.upv.es, cpavm@dsic.upv.es)

Abstract. In this paper, we study the parallelization of the Jacobi method to solve the symmetric eigenvalue problem on distributed-memory multiprocessors. To obtain a theoretical efficiency of 100% when solving this problem, it is necessary to exploit the symmetry of the matrix. The only previous algorithm we know exploiting the symmetry on multicomputers is that in [10], but that algorithm uses a storage scheme appropriate for a logical ring of processors, thus having a low scalability. In this paper we show how matrix symmetry can be exploited on a logical mesh of processors obtaining a higher scalability than that obtained with the algorithm in [10]. Algorithms for ring and mesh logical topologies are compared experimentally on the PARSYS SN-1040 and iPSC/860 multicomputers.

1 Introduction

The symmetric eigenvalue problem appears in many applications. In some applications in computational chemistry [3, 6] it is necessary to solve the eigenvalue problem obtaining all the eigenvalues of a symmetric, dense and real matrix of the size of some hundred or a few thousand. In this case it could be interesting to solve the problem by efficient Jacobi methods on multicomputers.

The basic idea behind Jacobi-like methods for finding the eigenvalues and eigenvectors of a real symmetric matrix is to construct a matrix sequence $\{A_l\}$ by means of

$$A_{l+1} = Q_l A_l Q_l^t \ , \ l = 1, 2, \ldots \tag{1}$$

where $A_1 = A$, and Q_l is a Jacobi rotation in the (p, q) plane with $0 \leq p, q \leq n-1$. Under certain conditions sequence $\{A_l\}$ converges to a diagonal matrix D,

$$D = Q_k Q_{k-1} \ldots Q_2 Q_1 A Q_1^t Q_2^t \ldots Q_{k-1}^t Q_k^t \tag{2}$$

whose diagonal elements are then the eigenvalues of A.

Each product $Q_l A_l Q_l^t$ represents a similarity transformation that annihilates a pair of nondiagonal elements, a_{ij} and a_{ji}, of matrix A_l. Matrix Q_l coincides

* Partially supported by ESPRIT III Basic Research Programme of the EC under contract No.9072 (Project GEPPCOM) and partially supported by Generalitat Valenciana Project GV-1076/93

with the identity matrix except in elements $q_{ii} = c$, $q_{ij} = s$, $q_{ji} = -s$, and $q_{jj} = c$, where $c = \cos\ \theta$, $s = \sin\ \theta$, and [5]

$$\tan\ 2\theta = \frac{2a_{ij}}{a_{ii} - a_{jj}}\ . \tag{3}$$

A cyclic Jacobi method works by making successive sweeps until some convergence criterion is fulfilled, normally until $off(A) < bound$. A sweep consists of successively nullifying the $n(n-1)/2$ nondiagonal elements in the upper-triangular part of the matrix (and the corresponding symmetrical part). To perform a sweep it is necessary to order the pairs of indices (i, j), $0 \leq i < j \leq n-1$, grouping them in $n-1$ disjointed sets, each one formed by $n/2$ pairs, then we can nullify the nondiagonal elements in $n-1$ iterations. Some of these orderings are intended to simplify parallelization of the method. To design efficient parallel Jacobi algorithms it is necessary to use a cyclic ordering with little movement of indices to pass from a set of pairs of indices to the next set of pairs, because this movement implies a data movement between processors. It is also necessary to combine the cyclic ordering with the topology we are using. In our implementations we have used the odd-even [8] and Eberlein [2] orderings with the ring topology, and the Round-Robin [1] ordering with a mesh topology.

When we apply the sequential Jacobi method to symmetric matrices we can exploit the symmetry by updating only the upper-triangular part of the matrix. In this case the cost of the method per sweep is $3n^3$ flops. It is not so easy to exploit the symmetry in parallel Jacobi methods. There are some papers where the symmetry is exploited on shared-memory multiprocessors [5] and systolic arrays [1]. Even the more recent works on Jacobi methods on distributed-memory multiprocessors applied to the Symmetric Eigenvalue Problem [7, 9] do not exploit the symmetry of the matrix, and the theoretical efficiency has an upper bound of 50%. The only previous algorithm we know exploiting the symmetry on a distributed-memory multicomputer is that presented in [10]. This algorithm has been designed for a logical ring, thus having the problem of a low scalability.

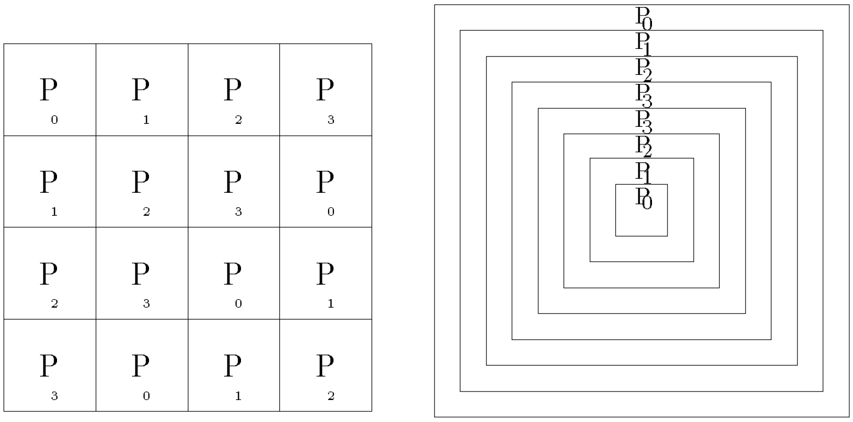
In section 2 two storage schemes, allowing us to exploit the symmetry on a logical ring, are presented. One of these storage schemes is that used in [10], and with the other storage scheme a better use of BLAS 1 can be achieved, due to a more compact use of the memory. In section 3 a parallel algorithm exploiting the symmetry on a logical mesh is proposed. In section 4 this algorithm is compared experimentally with those in section 2 on the PARSYS SN-1040 and iPSC/860 multicomputers. The experimental results confirm that the algorithm for mesh topology is more scalable than those for a ring topology.

2 Parallel Algorithms on a Logical Ring

To exploit the symmetry of the matrix on Jacobi methods on multicomputers it is necessary to obtain storage schemes storing symmetric elements in the same processor, and to combine the storage scheme with adequate orderings and topologies.

In [10] the *block Hankel-wrapped* storage scheme (figure 1.a) is combined with the odd-even ordering to obtain a parallel Jacobi method on a ring of processors.

For the purpose of comparing the proposed algorithm for a logical mesh, with the previous algorithm, we have implemented the algorithm using the *block Hankel-wrapped* storage scheme, and another algorithm also adequate for ring topology but with a more compact storage scheme that allows a better use of BLAS 1. We call the storage scheme used in this second algorithm a storage scheme by *frames* (figure 1.b) and it has been combined with the Eberlein ordering.



a) b)

Fig. 1. Storage schemes to exploit the symmetry on a logical ring of processors: a) *block Hankel-wrapped* storage scheme, b) storage scheme by *frames*.

These two algorithms for ring topology have a theoretical efficiency of 100%, but they present a low scalability.

3 A Parallel Algorithm on a Logical Mesh

In this section we show a storage scheme to exploit the symmetry of the matrix on a mesh of processors. Suppose $p = r^2$ processors are available and they are connected in a mesh, and the columns and rows of processors are numbered from 0 to $r-1$, and the processors designed by P_{ij}.

To distribute the matrix in the multiprocessors system we divide the upper-triangular part of matrix A in three submatrices of size $\frac{n}{2} \times \frac{n}{2}$ (figure 2.a), distribute the square submatrix upon the mesh (figure 2.b), and fold the upper-triangular matrices over the processors in the way shown in figure 2.c). So, we call this storage scheme by *folding*. If matrix A is divided in blocks A_{ij} with

$i = 0, 1, \ldots, 2r-1$, $j = i, i+1, \ldots, 2r-1$, the data assignation is done in the following way:
to $P_{i,r-1-i}$, with $i = 0, 1, \ldots, r-1$, blocks A_{ii}, $A_{2r-1-i,2r-1-i}$ and $A_{i,2r-1-i}$,
to P_{ij}, with $i+j < r-1$, blocks $A_{i,j+r}$ and $A_{i,r-1-j}$,
and to P_{ij}, with $i+j > r-1$, blocks $A_{i,j+r}$ and $A_{2r-1-i,j+r}$.

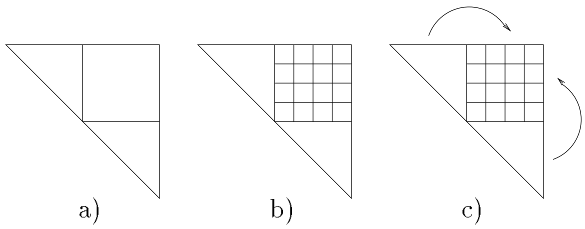


Fig. 2. Data storage with the storage scheme by *folding*.

We have combined the storage scheme by folding with the Round-Robin ordering to obtain a parallel algorithm exploiting the symmetry of the matrix on an open, square mesh of processors. The scheme used to compute the eigenvalues is:

IN PARALLEL: FOR $i = 0, 1, \ldots, r-1$; $j = 0, 1, \ldots, r-1$ IN P_{ij}:
    REPEAT
        FOR $k = 1, 2, \ldots, n-1$
            IF $i = r-1-j$ THEN
                Compute Rotations
            ENDIF
            Transfer Parameters
            Update Matrix
            Transfer Columns
            Transfer Rows
        ENDFOR
        Compute $off(A)$
    UNTIL $off(A) < bound$

Here is a brief explanation of each one of the procedures in the algorithm:

Only processors in the main antidiagonal of the mesh ($i = r-1-j$) compute rotations, because they contain the blocks in the main diagonal of the matrix.

The transference of parameters is not a Broadcast. The parameters obtained from the block A_{ii} must be transferred to processors containing blocks A_{ik}, with $i+1 \leq k \leq 2r-1$, and A_{ki}, with $0 \leq k \leq i-1$, so they must be transferred in the column and row of processors where the block is. But not all the communications are necessary, because parameters of blocks A_{ii}, $i < r$, must be transferred horizontally to all the processors in the row, but vertically only to the north; and parameters of blocks A_{ii}, $r \leq i$, must be transferred vertically to all the processors in the column, but horizontally only to the east. In figure 3 the data transference from processor P_{11} with $p = 9$ is shown. In a) the movement of data in the matrix is represented, and in b) the corresponding movement in the mesh is represented.

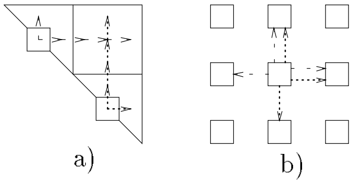


Fig. 3. Transference of parameters: a) transference in the matrix, b) transference in the mesh.

In `Update Matrix` each processor updates the part of the matrix it contains. So, this work is balanced.

In `Compute` $off(A)$ each processor computes the part of $off(A)$ corresponding to elements it contains. After this, the partial results are accumulated in P_{00} and the final result is broadcasted from P_{00} to the other processors in the system.

To explain procedures `Transfer Columns` and `Transfer Rows` we use an example with $n = 24$ and $r = 3$.

After `Transfer Columns` and `Transfer Rows` the upper-triangular part of A must be modified. We will see how these data are modified in `Transfer Rows`, and then we will deduce what data must be modified by `Transfer Columns`.

Figure 4 shows the transference of rows. With I, S, N, E and W we mark data that must be transferred. I indicates an internal transference, and S, N, E and W indicate external tranferences to the south, north, east and west, respectively. Each X represents an element that must be modified in the corresponding movement of data, but which does not participate in the movement.

We need also to store some blocks 2×2 in the lower-triangular part of A. These blocks appear in the figure with a letter, and they are stored in the processor where their symmetric blocks are stored (we want to exploit the symmetry).

In `Transfer Columns` we need to transfer data modifying those in figure 4. Figure 5 shows how to do this tranference.

In each step, each processor in the main antidiagonal computes $\frac{n}{2\sqrt{p}}$ rotations. So, the cost of `Compute Rotations` per sweep is

$$\frac{11n^2}{2\sqrt{p}} \qquad flops. \tag{4}$$

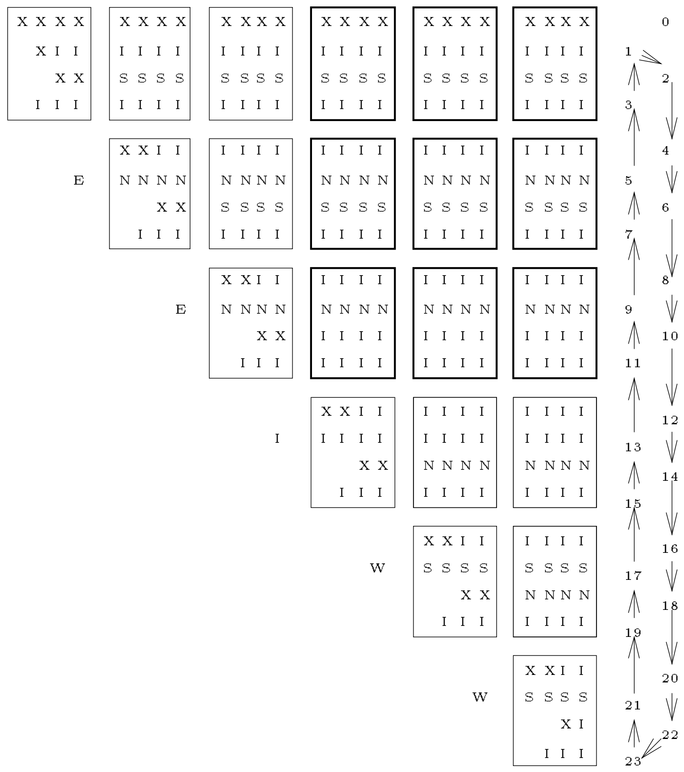


Fig. 4. Transference of rows with the storage scheme by *folding* and the Round-Robin ordering.

In `Update Matrix` each processor not in the main antidiagonal updates $\frac{n^2}{8p}$ blocks 2×2, and each processor in the main antidiagonal updates $\frac{n^2}{8p} + \frac{n}{4\sqrt{p}}$ blocks 2×2. So, the cost per sweep of `Update Matrix` is

$$\frac{3n^3}{p} + \frac{6n^2}{\sqrt{p}} - \frac{3n^2}{p} \qquad flops. \tag{5}$$

And the arithmetic cost per sweep is

$$\frac{3n^3}{p} + \frac{23n^2}{2\sqrt{p}} - \frac{3n^2}{p} \qquad flops. \tag{6}$$

If β is the start-up time and τ the word-sending time, in each step the transference of rotations has a cost of $\sqrt{p}\beta + n\tau$, the transference of columns expends a time of $6\beta + \left(\frac{4n}{\sqrt{p}} + 4\right)\tau$, and the transference of rows expends a time of

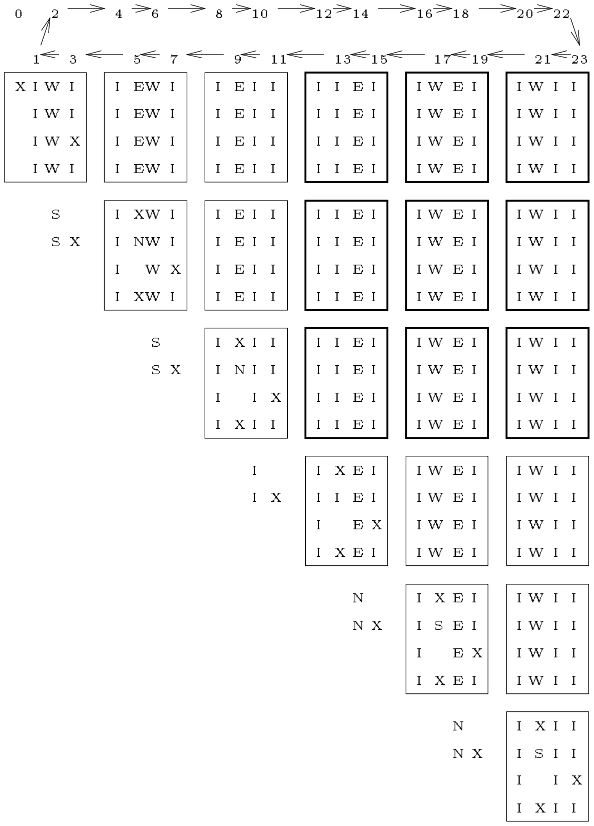


Fig. 5. Transference of columns with the storage scheme by *folding* and the Round-Robin ordering.

$6\beta+\left(\frac{4n}{\sqrt{p}}+2\right)\tau$. Hence, the communication cost per sweep can be approximated by

$$n\,(12+\sqrt{p})\,\beta+n^2\left(1+\frac{8}{\sqrt{p}}\right)\tau. \tag{7}$$

Because the cost per sweep of a sequential algorithm exploiting the symmetry is $3n^3$, the theoretical efficiency of the parallel algorithm is 100%.

4 Experimental Results

In order to compare our proposed algorithm for a logical mesh of processors, with the algorithms for a logical ring in section 2, we have implemented the algorithms on the multicomputers PARSYS SN-1040 and iPSC/860. We will call these algorithms: HANKEL (the algorithm for a logical ring using the *block Hankel-wrapped* storage scheme), FRAME (the algorithm for a logical ring using the storage scheme by *frames*) and MESH (the algorithm for a logical mesh using the storage scheme by *folding*).

In table 1 we show the results obtained with HANKEL and MESH using 4 and 16 processors on the PARSYS SN-1040, and in table 2 we show the results obtained with FRAME and MESH using 4 and 16 processors on an iPSC/860. Dense random matrices with entries varying between -10 and 10, and different sizes have been utilized. The codes were written in C language and the time in the tables are times in seconds per sweep. Also the times obtained with a sequential algorithm exploiting the symmetry and using the cyclic-by-row order, and the efficiencies, are shown in the tables.

Table 1. Execution time per sweep in seconds, and efficiencies of the algorithms for ring and mesh topologies. In the PARSYS SN-1040.

	192	256	320
SEQUENTIAL	90.52	214.52	417.79
HANKEL 4	30.59	69.69	132.80
efficiency RING 4	0.74	0.77	0.79
MESH 4	28.99	65.92	125.56
efficiency MESH 4	0.78	0.81	0.83
HANKEL 16	9.98	21.37	39.23
efficiency RING 16	0.57	0.63	0.67
MESH 16	8.64	18.75	34.62
efficiency MESH 16	0.65	0.72	0.75

The theoretical efficiency is 100%, but the experimental efficiency is still far from this limit when the matrix size is not very great, and this is due mainly to the broadcast, the contention time in the synchronizations of `Transfer Rows` and `Transfer Columns`, and the overhead due to the irregular distribution of data in the multiprocessor. This difference is bigger in the iPSC/860 than in the PARSYS SN-1040, and this is due to the fact that in the implementations for the iPSC/860 BLAS 1 has been used, reducing the execution time of the arithmetic part of the programs, and also to a greater cost of the communications in relation to the arithmetic operations in the iPSC/860.

The algorithm on mesh topology presents better performances (when the number of processors increases) than the algorithms on ring topology, and this is due to a lower cost of communications and to the possible overlapping of arithmetic and communication work on the algorithm for mesh.

To analize the scalability of the algorithms, in table 3 the efficiencies are compared on the iPSC/860. The table shows the efficiencies obtained with the

Table 2. Execution time per sweep in seconds, and efficiencies of the algorithms for ring and mesh topologies. In an iPSC/860.

	256	384	512
SEQUENTIAL	9.36	35.35	88.19
FRAME 4	4.59	11.30	22.85
efficiency RING 4	0.51	0.78	0.96
MESH 4	4.77	13.01	28.68
efficiency MESH 4	0.49	0.68	0.77
FRAME 16	4.67	8.59	14.99
efficiency RING 16	0.13	0.26	0.37
MESH 16	2.77	6.00	11.23
efficiency MESH 16	0.21	0.37	0.47

algorithms FRAME and MESH with 4 and 16 processors and with different matrix sizes, maintaining the number of elements per processor constant when the number of processors changes. The efficiency decreases in both algorithms when the number of processors increases, but with the topology of ring a big reduction in the efficiency is obtained, and with the topology of mesh this reduction is considerably lower. So a better experimental scalability is obtained with the algorithm we propose.

Table 3. Comparison of the scalability of the algorithms FRAME and MESH, comparing the efficiency when the number of processors increases maintaining the number of elements per processor. On the iPSC/860.

processors	matrix size	FRAME	MESH
4	128	0.18	0.24
16	256	0.13	0.21
4	192	0.33	0.40
16	384	0.26	0.37
4	256	0.51	0.49
16	512	0.37	0.47

5 Conclusions

From the analysis of the previous results we can conclude that it is possible to exploit the symmetry of the matrix when computing eigenvalues of symmetric matrices on a logical mesh of processors. The algorithm presented has been compared experimentally with the only previous algorithm we know exploiting the symmetry in the symmetric eigenvalue problem on multicomputers [10], and with another algorithm for a logical ring allowing a better use of BLAS 1. The algorithm for mesh presents a better experimental scalability, solving the problem of the previous algorithm. In addition, these storage schemes can be combined with schemes to work by blocks to obtain more efficient algorithms [4].

Acknowledgments. This work was performed in part on the 32 node Intel iPSC/860 operated by the University of Texas Center of High Performance Computing. We thank Professor Robert van de Geijn of the Department of Computer Sciences of the University of Texas for allowing and facilitating the use of this equipment.

References

1. R. P. Brent and F. T. Luk. A systolic architecture for almost linear-time solution of the symmetric eigenvalue problem. Technical Report TR-CS-82-10, Department of Computer Science, Australian National University, Camberra, August 1982.
2. P. J. Eberlein and H. Park. Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 8:358–366, 1990.
3. A. Edelman. Large dense linear algebra in 1993: The parallel computing influence. *The International Journal of Supercomputer Applications*, 7(2):113–128, 1993.
4. D. Giménez, V. Hernández, R. van de Geijn and A. M. Vidal. A jacobi method by blocks to solve the symmetric eigenvalue problem on a mesh of processors. ILAS Conference, 1994.
5. G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1989.
6. I. N. Levine. *Molecular Spectroscopy.* John Wiley and Sons, 1975.
7. M. Pourzandi and B. Tourancheau. A Parallel Performance Study of Jacobi-like Eigenvalue Solution. Technical report, March 1994.
8. G. H. Stewart. A Jacobi-like algorithm for computing the Schur decomposition of a nonhermitian matrix. *SIAM J. Sci. Stat. Comput.*, 4:853–864, 1985.
9. V. Strumpen and P. Arbenz. Improving Scalability by Communication Latency Hiding. In D. H. Bailey, P. E. Bjørstad, J. R. Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon and L. T. Watson, editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 778–779. SIAM, 1995.
10. R. A. van de Geijn. Storage schemes for Parallel Eigenvalue Algorithms. In G. H. Golub and P. Van Dooren, editor, *Numerical Linear Algebra. Digital Signal Processing and Parallel Algorithms*, volume 70 of *NATO ASI Series.* Springer-Verlag, 1991.