Exploiting the Symmetry on the Jacobi Method on a Mesh of Processors

D. Giménez
Departamento de Informática y Sistemas
Universidad de Murcia
Aptdo 4021. 30001 Murcia. Spain
domingo@dif.um.es

R. van de Geijn
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712
rvdg@cs.utexas.edu

V. Hernández* A. M. Vidal*
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Aptdo 22012. 46071 Valencia. Spain
{cpvhg,cpavm}@dsic.upv.es

Abstract

In this paper, we study the parallelization of the Jacobi method to solve the symmetric eigenvalue problem on a mesh of processors. To solve this problem obtaining a theoretical efficiency of 100% it is necessary to exploit the symmetry of the matrix. The only previous algorithm we know exploiting the symmetry on multicomputers is that in [1], but that algorithm uses a storage scheme adequate for a logical ring of processors, so 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 [1]. In addition, we show how the storage scheme exploiting the symmetry can be combined with a scheme by blocks to obtain a highly efficient and scalable Jacobi method for solving the symmetric eigenvalue problem for distributed memory parallel computers. We report performance results from the Intel Touchstone DELTA.

Keywords: Jacobi methods, symmetric eigenvalue problem, algorithms by blocks, multicomputers, mesh topology.

1 Introduction

Many different methods exist for computing the eigenvalues of a symmetric matrix, including bisection methods, the QR algorithm, divide and conquer (Cuppen's algorithm), and Jacobi's method. For a complete survey and references, we suggest the reader turn to [2, 3].

Jacobi's method is the oldest, dating back to the mid-1800's. It has fallen out of favor and been resurrected on many occasions. Its most recent resurgence is mainly due to better stability properties [4] and straight-forward parallelization [5, 6].

The primary purpose of this paper is to show how to obtain highly efficient parallel Jacobi algorithms. To achieve this goal we have used the very well known techniques of reformulating dense linear algebra algorithms to operate on blocks (submatrices) to obtain algorithms rich in matrix-matrix operations [7]. We combine this improvement in the sequential implementation with an assignment of data and work to the processors of a multicomputer to yield a parallel implementation that is both efficient and scalable.

The symmetric eigenvalue problem appears in many applications. In some applications in computational chemistry [8] it is necessary to solve the eigenvalue problem obtaining all the eigenvalues of a symmetric, dense and real matrix of 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.

In Section 2, we review some of the bases of Jacobi's method. In Section 3, a sequential Jacobi algorithm by blocks is given. In Section 4, the storage scheme that allows us to exploit the symmetry of the matrix on a logical mesh of processors is studied. In Section 5 the experimental results obtained on the Intel Touchstone DELTA are presented and analyzed. And in Section 6 the conclusions are given.

2 Jacobi's method

The basic idea behind Jacobi-like methods is to construct a matrix sequence $\{A_l\}$ by means of

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

where $A_1 = A$, and Q_l is a plane-rotation (Jacobi rotation) in the (p, q) plane with $1 \leq p, q \leq n$. Under

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

certain conditions [9], sequence $\{A_l\}$ converges to a diagonal matrix D,

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

whose diagonal elements are 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 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 [3]

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

This transformation only affects rows i and j when applied from the left, and columns i and j when applied from the right. The different ways of choosing pairs (i, j) have given rise to different versions of the Jacobi method. In the cyclic-by-row method the pairs are taken in the order

$$(1,2),(1,3),\ldots,(1,n),(2,3),(2,4),\ldots,$$
$$(2,n),\ldots,(n-1,n)\ . \qquad (4)$$

A cyclic Jacobi method works by making successive sweeps until some convergence criterion is fulfilled, normally until $\mathrm{off}(A) < \mathit{bound}$, where $\mathrm{off}(A) = \sqrt{\sum_{j=1, j\neq i}^{n} \sum_{i=1}^{n} a_{ij}^2}$. A sweep consists of successively nullifying the $n(n-1)/2$ nondiagonal elements in the lower triangular part of the matrix. The convergence of the cyclic-by-row method is analyzed in [9]. More recently, [10, 11] demonstrate that the Jacobi method converges with other ordering schemes when the cyclic-by-row method is convergent.

In this paper the odd-even order will be used [6]. This order was intended to allow parallelization of the method, but, as we shall see, it also simplifies a block based implementation of the sequential algorithm. We will demonstrate this ordering by examining the case where $n = 8$. Numbering indices from 1 to 8, and initially grouping the indices in pairs $\{(1,2),(3,4),(5,6),(7,8)\}$, the sets of pairs of indices are obtained as follows:

$$\begin{array}{ll}
k=1 & \{(1,2),(3,4),(5,6),(7,8)\} \\
k=2 & \{2,(1,4),(3,6),(5,8),7\} \\
k=3 & \{(2,4),(1,6),(3,8),(5,7)\} \\
k=4 & \{4,(2,6),(1,8),(3,7),5\} \\
k=5 & \{(4,6),(2,8),(1,7),(3,5)\} \\
k=6 & \{6,(4,8),(2,7),(1,5),3\} \\
k=7 & \{(6,8),(4,7),(2,5),(1,3)\} \\
k=8 & \{8,(6,7),(4,5),(2,3),1\}
\end{array} \qquad (5)$$

where odd sets have four pairs of indices and even sets have three pairs of indices, so n iterations are needed to obtain the $n(n-1)/2$ pairs of indices.

3 A Jacobi algorithm by blocks

The underlying idea behind blocked algorithms is that traditional implementations of dense linear algebra algorithms encounter a bottleneck in modern architectures due to limited bandwidth between the CPU and main memory. The data and computation requirements of matrix-matrix operations can be arranged so that more computation is performed between memory accesses. As a result, these operations can take advantage of hierarchical memories. A suite of such matrix-matrix operations are part of the Basic Linear Algebra Subprograms (BLAS) [12]. In this section, we show how blocking can be used for Jacobi's method.

3.1 The Level-1 BLAS implementation

We start by giving in figure 1 a pseudo-code for a Jacobi's method implementation which is rich in vector-vector operations. In this figure, the operation `update matrix` consists of applying rotation to rows and columns i and j of A, which can be done using the level-1 BLAS routine `drot`. When taking advantage of symmetry, three calls to `drot` are required, rotating row and column pieces labeled by 1, 2 and 3 in figure 2. For each sweep, `update matrix` is executed $n(n-1)/2$ times at a cost of $6n$ floating point operations (flops) per execution. Thus the total cost per sweep is $3n^3$ flops.

```
WHILE convergence not reached DO
        FOR every pair (i,j) of indices in a sweep DO
                compute parameters c and s
                update matrix
        ENDFOR
ENDWHILE
```

Figure 1: **Basic Jacobi iteration.**

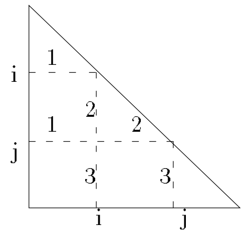


Figure 2: **Applying a single Jacobi transformation to matrix A using BLAS 1.**

In general, vector-vector operations like `drot` are memory intensive, and hence inefficient. Moreover, while applying a rotation from the left, the vectors require a stride through memory (when column-major storage is used), which is further added to the inefficiencies due to memory accesses.

3.2 The use of level-3 BLAS

In this subsection we show how to obtain a Jacobi algorithm by blocks. The method is similar to this used by Bischof for the SVD problem [13].

If matrix A is divided into columns and rows of square blocks of size s, it is possible to make a sweep over these blocks using some Jacobi order, *treating blocks as units*. For reasons that will become apparent later, we will use the odd-even order. So, with $n = 8s$, blocks are treated in the order indicated by the numbers in figure 3. In each block the algorithm works by making a sweep over the elements in the block. In our case, a cyclic-by-rows order has been used. Notice that some nondiagonal elements of the matrix are not covered by this method. In particular, the subdiagonal elements belonging to diagonal blocks in figure 3 will not be zeroed. To correct this, blocks 1, 2, 3 and 4 in figure 3 are considered to have size $2s \times 2s$, adding to each block the two adjacent diagonal blocks and the symmetrical block (figure 4.a). A sweep can now be performed covering all elements in these newly created blocks. Now the algorithm works on the blocks in the order $1, 2, 3, 4, \ldots$ until a sweep is completed, making successive sweeps until convergence is reached.

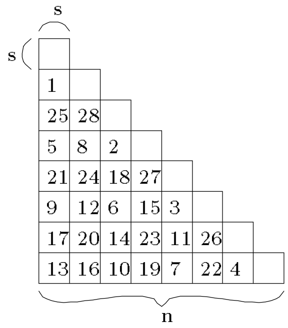


Figure 3: **Odd-even order by blocks.**

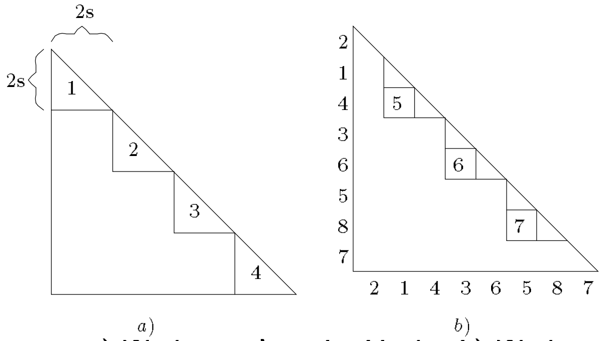


Figure 4: **a) Work on triangular blocks, b) Work on square blocks.**

We have now set the stage for an algorithm rich in level-3 BLAS operations (in this case matrix-matrix multiplications): Sweep over block 1 exploiting the symmetry of the matrix and using level-1 BLAS. In the process, the corresponding rotations are accumulated to form a matrix Q of size $2s \times 2s$. Finally, the first column of blocks of size $2s \times 2s$ is postmultiplied by this Q (except the $2s \times 2s$ block 1, obviously). This matrix-matrix multiplication can be done using level-3 BLAS. The work on blocks 2, 3 and 4 is done similarly.

After completing this first set of blocked rotations, a swap of column and row blocks is performed, using the odd-even order. This brings the next blocks of size $2s \times 2s$ to be zeroed to the diagonal, and the process continues similarly to operations performed on blocks 1, 2, 3 and 4. However, in this case the elements to be nullified are in square blocks of size $s \times s$ inside diagonal blocks of size $2s \times 2s$.

So, if initially the pairs of block indices are $\{(1,2),(3,4),(5,6),(7,8)\}$, after swapping indices a new set of pairs is obtained $\{(1,4),(3,6),(5,8)\}$. This requires swapping row and column blocks, yielding a new distribution (figure 4.b). Now the work is done on the diagonal blocks of size $2s \times 2s$ marked in figure 4.b, zeroing only elements in blocks 5, 6 and 7. After this the algorithm continues similarly until a sweep is completed. The Jacobi algorithm using this ordering by blocks converges, as can be easily proved using results in [11].

The swapping of row and column blocks is illustrated in figure 5. In this figure a bidirectional arrow $\longleftrightarrow$ represents a swap and a unidirectional arrow $\rightarrow$ a copy. One way to view such data movement is to recognize that it is equivalent to pre- or post-multiplying by a permutation matrix

$$P = \begin{pmatrix} \hat{P} & 0 & \cdots & 0 \\ 0 & \hat{P} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \hat{P} \end{pmatrix} ,$$

where $\hat{P}$ is the matrix of size $2s \times 2s$

$$\hat{P} = \begin{pmatrix} 0 & I \\ I & 0 \end{pmatrix} .$$

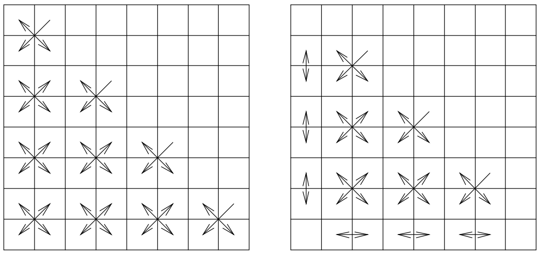
Next, since the updates of blocks involves an operation like $A = QAQ^t$, the permutation can be incorporated into Q, updating $A = (PQ)A(PQ)^t$ instead. Regardless, for the blocked algorithm, swapping represents a lower order operation.

3.3 Theoretical cost

To study the cost of the algorithm it suffices to consider only the number of flops. The additional time due to the data movement is inconsequential.

The number of blocks of size $s \times s$ is

$$\frac{\frac{n}{s}\left(\frac{n}{s}-1\right)}{2} = \frac{q\,(q-1)}{2}. \qquad (6)$$



a) b)

Figure 5: **Movement of blocks. a) Odd step, b) Even step.**

The first $q/2$ blocks are considered triangular blocks of size $2s \times 2s$, each requiring $3(2s)^3$ flops to make a sweep over the offdiagonal elements of the block and $3(2s)^3$ flops to accumulate the rotations, for a total of $6(2s)^3$ flops. In the rest of the blocks only s^2 elements are zeroed, thus needing $24s^3$ flops per block. So, the total number of flops to make the local sweeps over all blocks is:

$$\frac{q}{2}6(2s)^3 + \left(\frac{q^2}{2} - q\right) 24s^3 = 12n^2 s \text{ flops.} \quad (7)$$

The rest of the work in the algorithm is the updating of rows and columns of blocks. After a sweep over one of the blocks of size $2s \times 2s$, matrices of sizes $2s \times (n-2s-i)$ and $2s \times i$ are updated (where i takes values $0,\ s,\ 2s, \ldots,\ n-2s$), and these updates are done by premultiplying or postmultiplying by matrices of size $2s \times 2s$, thus needing $2\,(2s)^2\,(n-2s) = 8s^2\,(n-2s)$ flops. Because the number of $s \times s$ blocks is $q(q-1)/2$, the total cost of "update matrix" is:

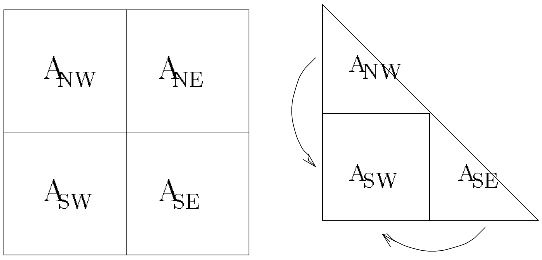
$$\begin{aligned} &\frac{q(q-1)}{2} 8s^2\,(n-2s) = \\ &4n^3 - 12n^2 s + 8ns^2 \text{ flops.} \end{aligned} \quad (8)$$

So, adding (7) and (8) the cost of the algorithm is approximately $4n^3$ for $s << n$. Comparing this to the cost of the level-1 implementation, $3n^3$, we could conclude that blocking is a bad idea. However, it should be noted that for the blocked algorithm almost all of the cost is in matrix-matrix multiplication, which can be expected to be performed at a faster rate than the vector-vector operations of the level-1 implementation. Letting the time required for a floating point operation be k_1 and k_3 for a level-1 and level-3 BLAS based implementation, respectively, the expected speedup by using the blocked method becomes

$$\frac{3\ k_1}{4\ k_3}. \quad (9)$$

4 A Jacobi algorithm by blocks on a mesh of processors

We now present a parallel implementation of the blocked algorithm for a logical mesh of processors. The assignment of elements of the matrix (storage scheme) we will use is obtained by dividing matrix A into four square submatrices of size $\frac{n}{2} \times \frac{n}{2}$ (figure 6.a). These four submatrices will be called A_{NW}, A_{NE}, A_{SW} and A_{SE}, where the subindices indicate the relative position of each submatrix in matrix A, and N, S, E, and W stand for north, south, east, and west, respectively. The matrix is symmetric, and so, it is necessary to work only with submatrix A_{SW} and the lower triangular parts of submatrices A_{NW} and A_{SE}.



a) b)

Figure 6: **a) Partition of the matrix by blocks, b) Folding.**

If we employ a square mesh with $p = r^2$ processors we divide submatrix A_{SW} into p square submatrices of size $\frac{n}{2r} \times \frac{n}{2r}$:

$$\begin{array}{cccc} A_{SW_{0,0}} & A_{SW_{0,1}} & \ldots & A_{SW_{0,r-1}} \\ \ldots & \ldots & \ldots & \ldots \\ A_{SW_{r-1,0}} & A_{SW_{r-1,1}} & \ldots & A_{SW_{r-1,r-1}} \end{array} \quad (10)$$

and assign submatrix $A_{SW_{i,j}}$ to processor $P_{i,j}$. To assign the lower-triangular parts of A_{NW} and A_{SE} to the processors, they are folded over A_{SW} (figure 6.b). So, if we consider matrices A_{NW} and A_{SE} divided into submatrices of size $\frac{n}{2r} \times \frac{n}{2r}$, $A_{NW_{i,j}}$ and $A_{SE_{i,j}}$, with $i = 0, \ldots, r-1$ and $j = 0, \ldots, i$, these submatrices will be assigned to the processors in the following way: $A_{NW_{r-1-i,j}}$ is assigned to $P_{i,j}$, $i = 0, \ldots, r-1$; $j = 0, \ldots, i$, and submatrix $A_{SE_{i,r-1-j}}$ is assigned to $P_{i,j}$, $i = 0, \ldots, r-1$; $j = r-1-i, \ldots, r-1$.

For simplicity we have assumed that $n = 2rk$, where k is the size of the submatrices being assigned to processors. To combine this method with the sequential blocked scheme, we will further assume that k is a multiple of $2s$. Naturally, an approximate decomposition can be used to overcome this restriction.

Thus, to combine this Storage Scheme with the design of an algorithm rich in matrix-matrix multiplications, it is necessary to divide each submatrix $A_{SW_{i,j}}$, $A_{NW_{i,j}}$ and $A_{SE_{i,j}}$ into submatrices of size $s \times s$ that we will call $A_{SW^{l,m}_{i,j}}$, $A_{NW^{l,m}_{i,j}}$ and $A_{SE^{l,m}_{i,j}}$, with $l = 0, \ldots, t-1$, $m = 0, \ldots, t-1$, where $t = \frac{n}{2rs}$.

The basic parallel algorithm is given by figure 7. We will explain briefly how each step in the parallel scheme works.

```
In parallel for each P_{i,j}  i = 0,...,r-1; j = 0,...,r-1:
       WHILE off(A) > bound DO
              FOR k = 1 TO n/s
                     IF i = r-1-j
                            Compute rotations
                     ENDIF
                     Broadcast Rotations
                     Update matrix
                     Transfer data
              ENDFOR
       ENDWHILE
```

Figure 7: **Parallel Jacobi iteration.**

4.1 Compute rotations

For the design of the algorithm we have used the odd-even ordering. With this ordering different nullification schemes are obtained on odd and even steps. On odd steps every antidiagonal processor will compute the same number of rotations, but on even steps one processor computes rotations for one pair less than the other antidiagonal processors (processor $P_{r-1,0}$).

We will show how blocks are stored in memory with an example with $p = 4$ and $\frac{n}{4s} = 4$, on odd (figure 8) and even (figure 9) steps. In these figures there are blocks of memory containing blocks of matrices (marked with X) and other additional memory blocks used in some parts of the program. Some blocks of size $2s \times 2s$ have been marked. The subsweeps are done over these blocks, and so, it is necessary to accumulate rotations corresponding to each one of these blocks and then to broadcast the resulting matrices.

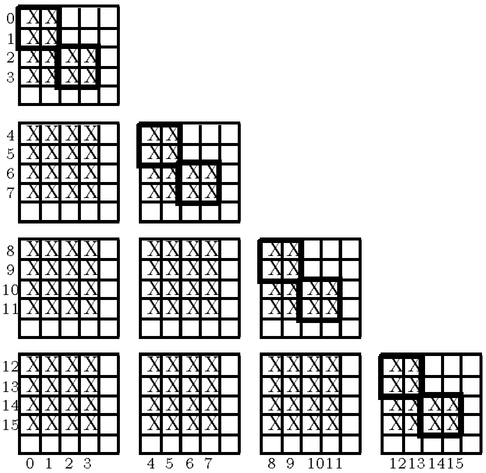


Figure 8: **Computing rotations. Odd step.**

The procession from figure 8 to figure 9 and from figure 9 to figure 8 will be explained when we study how the data transference is performed.

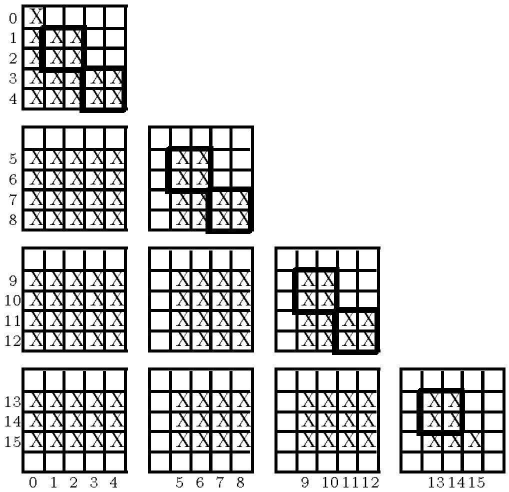


Figure 9: **Computing rotations. Even step.**

4.2 Broadcast rotations

The broadcast of rotations is done in the same way on each step.

To update a $2s\times 2s$ block it is necessary to know the matrices where the rotations modifying the diagonal blocks in the same row or column as the block to be updated have been accumulated. So, the matrices where rotations have been accumulated are sent to processors containing blocks in the same row or column, that is, to the processors on the same row or column of processors where the diagonal matrices are. But it isn't necessary to send all the matrices of rotations because we work only with the lower triangular part of the matrix. It is therefore necessary to send these parameters to the south and the west of the matrix. In the mesh, the rotation matrices computed in A_{NW} are sent west, south and north, and the matrices computed in A_{SE} are sent west, east and south (figure 10).

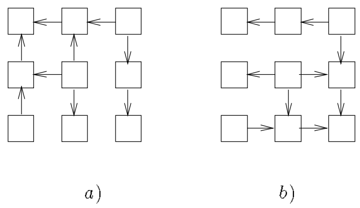


Figure 10: **Broadcast rotations. a) rotations computed in A_{NW}. b) rotations computed in A_{SE}.**

The matrices generated by rotations corresponding to A_{NW} and A_{SE} are sent together to the adjacent processors, and are sent in the order:

processor $P_{0,r-1}$ to $P_{0,r-2}$ and $P_{1,r-1}$

processor $P_{r-1,0}$ to $P_{r-2,0}$ and $P_{r-1,1}$

processor $P_{i,r-1-i}$ with $i = 1, \ldots, \lfloor \frac{r-1}{2} \rfloor$, to $P_{i,r-2-i}$, $P_{i+1,r-1-i}$, $P_{i,r-i}$ and $P_{i-1,r-1-i}$

and processor $P_{i,r-1-i}$ with $i = \lfloor \frac{r-1}{2} \rfloor + 1, \ldots, r-2$, to $P_{i,r-i}$, $P_{i-1,r-1-i}$, $P_{i,r-2-i}$ and $P_{i+1,r-1-i}$.

This order of broadcasting is in figure 11, where the numbers by the rows indicate the order in which the corresponding messages are sent. Processors not in the main antidiagonal merely pass the rotation matrices they receive.

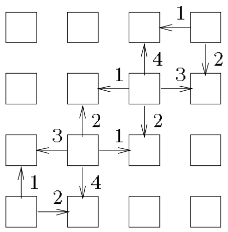


Figure 11: **Order in which rotation matrices are sent.**

4.3 Update matrix

Once the rotation matrices have been broadcasted, updating of matrix A proceeds as in the sequential algorithm, except that rows and columns of blocks are now distributed among processors.

4.4 Data transference

Data exchange between processors in even and odd steps is shown in figures 12 and 13.

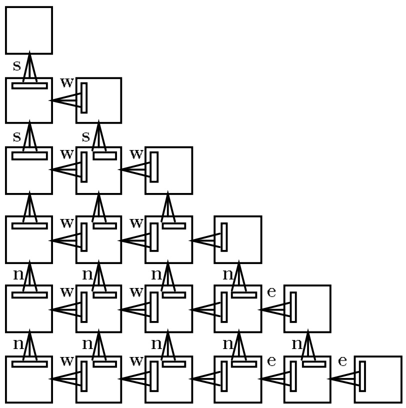


Figure 12: **Data transference. Odd step.**

After each step, data movement consists of columns being communicated to the east and west, and rows to the north and south in the matrix (what causes in the mesh the data movement shown in figures 12 and

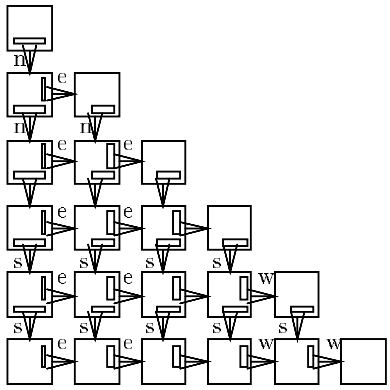


Figure 13: **Data transference. Even step.**

13). In addition, after the exchange of data an internal movement of data is needed to obtain the new grouping of blocks, and this movement can be incorporated into the rotation matrices by appropriate permutation, as in the sequential algorithm.

4.5 Theoretical cost

During each step, antidiagonal processors compute $\frac{n}{2s}$ or $\frac{n}{2s} - 1$ rotation matrices on a subsweep, computing on each antidiagonal processor $\frac{n}{2rs} = t$ or $\frac{n}{2rs} - 1 = t - 1$ rotation matrices. So, the cost of this operation for each subsweep is:

$$t48s^3 = \frac{24ns^2}{r} \qquad \text{flops} \tag{11}$$

in the first step, and:

$$t24s^3 = \frac{12ns^2}{r} \qquad \text{flops} \tag{12}$$

in successive steps.

So, the total cost of "compute rotations" per sweep is:

$$\frac{24ns^2}{r} + \left(\frac{n}{s} - 1\right) \frac{12ns^2}{r} = \frac{12n^2s}{r} + \frac{12ns^2}{r} \text{ flops.} \tag{13}$$

The most expensive communication associated with broadcasting rotation matrices is the one from processor $P_{0,r-1}$, since those matrices need to pass through $r-1$ links.

Each processor on the main antidiagonal computes $\frac{t}{2}$ rotation matrices of size $2s \times 2s$ from matrix A_{NW} and also $\frac{t}{2}$ from matrix A_{SE}. Hence, on each dispatch, processor $P_{0,r-1}$ sends $4ts^2 = \frac{2ns}{r}$ elements, sending first on link 1 and then on link 2 (figure 11). Thus, the cost of broadcasting rotations is:

$$r\alpha + 2ns\beta, \tag{14}$$

and the cost per sweep is:

$$\frac{n}{s}\left(r\alpha + 2ns\beta\right) = \frac{nr}{s}\alpha + 2n^2\beta, \quad (15)$$

where α is the start-up time and β is the word sending time.

While updating the matrix, during even steps each processor not on the main antidiagonal has the same amount of work, and updates a total of $\frac{n^2}{2p}$ elements, requiring $\frac{4n^2s}{p}$ flops. During odd steps, processors on column 0 must also update $2s\frac{n}{2r}$ elements, which is done in $\frac{4s^2n}{r}$ flops. Hence $\frac{4n^2s}{p} + \frac{4s^2n}{r}$ flops per sweep are required to update the matrix on odd steps. Thus the cost per sweep of "update matrix" is:

$$\frac{n}{2s}\frac{4n^2s}{p} + \frac{n}{2s}\left(\frac{4n^2s}{p} + \frac{4s^2n}{r}\right) = \frac{4n^3}{p} + \frac{2sn^2}{r} \text{ flops.} \quad (16)$$

During odd steps some processors send $\frac{n}{r}s$ elements to the west and $\frac{n}{2r}s$ to the east, $\frac{n}{r}s$ to the north and $\frac{n}{2r}s$ to the south. The cost is $4\alpha + \frac{3ns}{r}\beta$. During even steps some processors send $\frac{n}{2r}s$ elements to the north, $\frac{n}{r}s$ to the south, $\frac{n}{r}s$ to the east and $\frac{n}{2r}s$ to the west. And the cost is $4\alpha + \frac{3ns}{r}\beta$. Thus, the cost per sweep of exchanging data is:

$$\frac{n}{2s}\left(8\alpha + \frac{6ns}{r}\beta\right) = \frac{4n}{s}\alpha + \frac{3n^2}{r}\beta. \quad (17)$$

The total arithmetic cost per sweep, given by adding (13) and (16) is

$$4k_3\frac{n^3}{p} + (2k_3 + 12k_1)\frac{n^2s}{r} + 12k_1\frac{ns^2}{r} \text{ flops,} \quad (18)$$

where k_3 and k_1 are the constants affecting the execution times obtained using BLAS 3 and BLAS 1, respectively. The communication cost per sweep, given by adding (15) and (17), is

$$(4+r)\frac{n}{s}\alpha + \left(2 + \frac{3}{r}\right)n^2\beta. \quad (19)$$

Since the overhead related with the communication and load imbalance is of lower order, 100% efficiency can be approached by this algorithm when the matrix size is large enough.

5 Experimental results

The results presented here have been obtained on the Touchstone DELTA. This multicomputer consists of 512 i860 processors connected by a mesh network of dimension 16 × 32. The codes were written in C, using double precision arithmetic. Test matrices were generated by taking elements with values uniformly distributed between -10 and 10. The number of sweeps to reach the convergence is similar to those obtained by cyclic methods and is approximately $log_2\ n$. This allows us to report only times per sweep.

5.1 Sequential algorithm

Table 1 shows the execution time per sweep in seconds when only eigenvalues are computed on an i860 processor. This table reports a comparison of the algorithms that use BLAS 1 and 3, respectively. The optimal value of block size increases with the matrix size, and for the test matrices was between 16 and 32.

size	B1	B3. 4	B3. 8	B3. 16	B3. 32	B3. 64
256	9.2	8.8	5.7	**4.9**	5.4	7.6
512	88.5	68.0	40.1	**30.8**	31.8	43.6
768	306.2	220.1	126.7	94.1	**92.3**	117.4
1024	738.1	510.5	291.6	211.1	**207.1**	244.3

Table 1: **Execution times of sequential algorithms using BLAS 1 and BLAS 3 with different block sizes.**

Table 2 shows the Mflops obtained with the algorithm using BLAS 3 with optimal block size, when only eigenvalues are computed and when eigenvalues and eigenvectors are computed. When eigenvalues and eigenvectors are computed, a better performance is obtained, due to the fact that the amount of computation is greater when eigenvalues and eigenvectors are computed and hence using level-3 BLAS becomes beneficial sooner.

size	256	384	512	640	768	896	1024
values	13.6	15.9	17.4	18.3	19.6	20.3	20.7
vectors	18.1	20.6	22.6	22.9			

Table 2: **Mflops obtained using the sequential algorithm with BLAS 3 and optimal block sizes.**

5.2 Parallel algorithm

In table 3 the execution times obtained on the Touchstone DELTA are shown, with different mesh and matrix sizes, and with block sizes 16 and 32. The optimal block size is always 16, but when the matrix size increases the difference between times obtained with block sizes 16 and 32 decreases as in the sequential algorithm. In the parallel case the pre- and post-multiplication of rows and columns of blocks by a rotation matrix is performed by rows and columns of processors, each processor working thus with smaller matrices and obtaining less benefit from the work with BLAS 3.

	processors					
	4		16		64	
	block size					
	16	32	16	32	16	32
512	**10.8**	13.3	**4.5**	5.9	**2.0**	
1024	**66.3**	71.1	**23.5**	27.9	**9.1**	12.2
1536	**202.7**	203.0	**64.8**	72.5	**23.6**	
2048			**138.0**	147.4	**47.7**	56.8

Table 3: **Execution time per sweep of the parallel algorithm on the Touchstone DELTA.**

To analyze experimentally the scalability of the algorithm the Mflops/node obtained with meshes of different sizes when using the same memory per node are shown in table 4. Also the matrix size used are shown in the table. We consider the obtained results satisfactory, showing good scalability (approximately 3.5 Gflops have been obtained using 256 processors with matrices of size 6144).

processors	1	16	25	36	64	256
Mflops/node	15.2	13.8	13.5	13.7	13.6	13.5
matrix size	384	1530	1920	2302	3072	6144

Table 4: **Mflops per node obtained by the parallel algorithm on the Touschtone DELTA.**

6 Conclusions

The use of BLAS 3 to solve the symmetric eigenvalue problem by Jacobi methods can produce efficient algorithms. A small block size (between 16 and 32) gives good results. The optimal block size increases with the matrix size. When eigenvalues and eigenvectors are computed a bigger optimal block size is obtained. This is due to a higher amount of computations, which produces a better utilization of BLAS 3.

This sequential algorithm can be combined with an appropriate Storage Scheme (folding) to obtain an algorithm by blocks on a mesh of processors that presents good scalability properties.

Acknowledgments

This work was performed in part using the Intel Touchstone Delta System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Intel Supercomputer Systems Division and the California Institute of Technology. In addition, this work was performed in part on the 32 node Intel iPSC/860 operated by the University of Texas Center of High Performance Computing.

References

[1] 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.

[2] K. A. Gallivan, R. J. Plemmons and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, 1990.

[3] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.

[4] J. Demmel and K. Veselić. Jacobi's method is more accurate than QR. *SIAM Journal on Matrix Analysis and Applications*, 13:1204–1245, (1992).

[5] R. P. Brent and F. T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM Journal on Scientific and Statistic Computation*, 6(1):69–84, (1985).

[6] G. W. Stewart. A Jacobi-like algorithm for computing the Schur decomposition of a nonhermitian matrix. *SIAM Journal on Scientific and Statistic Computation*, 4:853–864, (1985).

[7] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.

[8] A. Edelman. Large dense linear algebra in 1993: The parallel computing influence. *The International Journal of Supercomputer Applications*, 7(2):113–128, 1993.

[9] G. E. Forsythe and P. Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Transactions of American Mathematical Society*, 94:1–23, (1960).

[10] F. T. Luk and H. Park. A proof of convergence for two parallel Jacobi SVD algorithms. *IEEE Transactions on Computers*, 28(6):806–811, (1989).

[11] G. Schroff and R. Schreiber. On the convergence of the cyclic Jacobi method for parallel block orderings. *SIAM Journal on Matrix Analysis and Applications*, 10(3):326–346, (1989).

[12] J. J. Dongarra, J. Du Croz, S. Hammarling and I. Duff. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. on Math. Soft.*, 16(1):1–17, (1990).

[13] C. H. Bischof. Computing the singular value decomposition on a distributed system of vector processors. *Parallel Computing*, 11:171–186, 1989.