A JACOBI METHOD BY BLOCKS ON A MESH OF PROCESSORS

Domingo Giménez[1], Vicente Hernández[2*], Robert van de Geijn[3], 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. (vhernand@dsic.upv.es, avidal@dsic.upv.es)

[3]Department of Computer Sciences. University of Texas at Austin. Austin, Texas, 78712. (rvdg@cs.utexas.edu)

*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

contact author:

Domingo Giménez

e-mail: domingo@dif.um.es

street address: Departamento de Informática y Sistemas. Univ de Murcia. Aptdo 4021. 30001 Murcia. Spain. (domingo@dif.um.es)

telephone: 34 68 307100

fax: 34 68 364151

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 [21], 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 [21]. 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, the iPSC/860, the Alliant FX/80 and the PARSYS SN-1040.

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. More recently, novel algorithms based on invariant subspace decomposition that use matrix-matrix multiplication have been proposed [2, 16]. For a complete survey and references, we suggest the reader turn to [12, 14].

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 [5] and straight-forward parallelization [4, 7, 8, 11, 20, 21].

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 [1]. As a result, performance of a sequential implementation improves by overcoming the common discrepancy between CPU performance and bandwidth to main memory. 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 [9] 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 using the odd-even order to solve the symmetric eigenvalue problem 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.

Jacobi's Method

While we expect the reader to be well versed in the details of Jacobi's method, we briefly review the basics in this section.

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 plane-rotation (Jacobi rotation) in the (p, q) plane with $1 \leq p, q \leq n$. Under certain conditions [10], 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 with the identity matrix except in elements $q_{ii} = \cos\ \theta$, $q_{ij} = \sin\ \theta$, $q_{ji} = -\sin\ \theta$, and $q_{jj} = \cos\ \theta$, and [14]

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

Notice that 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 (and the corresponding symmetrical part). The convergence of the cyclic-by-row method is analyzed in [10]. More recently, [18, 19] demonstrate that the Jacobi method converges with other ordering schemes when the cyclic-by-row method is convergent. These orderings are intended to simplify parallelization of the method.

In this paper the odd-even order will be used [20]. Again, 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:

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

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. It can be shown that this ordering is equivalent to the row-cyclic ordering, where the rows and columns of the matrix are initially permuted. Grouping pairs of indices using the odd-even order, n iterations are needed to complete a sweep.

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, like matrix-matrix multiplication, can be arranged so that more computation is performed between memory accesses. This can be easily seen by observing that for matrices with dimension n, $O(n^3)$ operations are performed on $O(n^2)$ data. As a result, these operations can take advantage of hierarchical memories that include data caches. A suite of such matrix-matrix operations are part of the Basic Linear Algebra Subprograms (BLAS) [6], which are typically optimized for various architectures. As a result, an algorithm coded in terms of call to the BLAS becomes a portable high performance implementation. In this section, we show how blocking can be used for Jacobi's method.

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.

FIGURE 1

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.

FIGURE 2

The object is now to perform those flops efficiently. 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.

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 [3].

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, where the work is done on the lower triangular part of the matrix. In each block the algorithm works by making a sweep over the elements in the block, that is, zeroing each element in the block once by means of some order. 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 (cyclic by rows) 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.

$\boxed{\text{FIGURE 3}}$

$\boxed{\text{FIGURE 4}}$

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 $\{(1,4),(3,6),(5,8)\}$ (Figure 5). This requires swapping row and column blocks, yielding a new distribution (Figure 5). Now the work is done on the diagonal blocks of size $2s \times 2s$ marked in Figure 5, 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 [19].

FIGURE 5

The swapping of row and column blocks is illustrated in Figures 6 and 7. In these figures 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 $2s \times 2s$

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

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.

$\boxed{\text{FIGURE 6}}$

$\boxed{\text{FIGURE 7}}$

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.

In Figure 3, it can be seen that the number of blocks of size $s \times s$ is

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

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:

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

The two extreme values of s are 1 and $n/2$, so, the extreme values of (7) are $12n^2$ and $6n^3$, but optimal execution times will be obtained with moderate values of s.

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 pre-multiplying or post-multiplying 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:

$$\frac{q(q-1)}{2}\; 8s^2\,(n-2s) = 4n^3 - 12n^2 s + 8ns^2 \qquad \text{flops.} \tag{8}$$

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

$$\frac{3}{4}\frac{k_1}{k_3}. \tag{9}$$

Computing eigenvectors

To compute eigenvectors it is necessary to accumulate the orthogonal transformations, and it can also be done using matrix-matrix multiplications. After a sweep over a block of size $2s \times 2s$, it is necessary to multiply the $2s \times 2s$ matrix, where the transformations have been accumulated (Q), by the corresponding rows of matrix V, where the eigenvectors are computed. To do this efficiently a movement of rows on V is necessary after each step of the order, and this movement is done by permuting rows in the $2s \times 2s$ matrix where rotations were accumulated before updating V.

The cost of updating V in each step is $2(2s)^2n = 8s^2n$, and $q(q-1)/2$ steps are needed per sweep. So, the cost per sweep of updating V is:

$$4n^3 - 4n^2s \qquad \text{flops.} \tag{10}$$

The cost of computing eigenvalues and eigenvectors is thus (formulas (7), (8) and (10)):

$$8k_3n^3 + (12k_1 - 16k_3)n^2s + 8k_3ns^2 \qquad \text{flops} \tag{11}$$

and in this case the same quotient (9) is obtained.

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 8). 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} (and with some additional blocks in the upper-triangular part of the matrix).

FIGURE 8

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

$$\begin{array}{cccc} A_{SW_{0,0}} & A_{SW_{0,1}} & \dots & A_{SW_{0,r-1}} \\ \dots & \dots & \dots & \dots \\ A_{SW_{r-1,0}} & A_{SW_{r-1,1}} & \dots & A_{SW_{r-1,r-1}} \end{array} \tag{12}$$

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 9). So, if we consider the lower-triangular part of 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 $i = 0, \dots, r-1$ and $j = 0, \dots, i$, these submatrices will be assigned to the processors in the following way: $A_{NW_{r-1-i,j}}$ is assigned to $P_{i,j}$, $i = 0, \dots, r-1$; $j = 0, \dots, i$, and submatrix $A_{SE_{i,r-1-j}}$ is assigned to $P_{i,j}$, $i = 0, \dots, r-1$; $j = r-1-i, \dots, r-1$.

FIGURE 9

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 $A_{SW_{i,j}^{l,m}}$, $A_{NW_{i,j}^{l,m}}$ and $A_{SE_{i,j}^{l,m}}$, with $l = 0, \ldots, t-1$, $m = 0, \ldots, t-1$, where $t = \frac{n}{2rs}$.

The basic parallel algorithm is given by Figure 10. In the following subsections, we will explain briefly how each step in the parallel scheme works.

FIGURE 10

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:

$$\text{even: } (1,2),(3,4),(5,6),(7,8),\ldots$$

$$\text{odd: } 1,(2,3),(4,5),(6,7),(8,9)\ldots$$

So, on even steps every antidiagonal processor will compute the same number of rotations, but on odd 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 even (Figure 11) and odd (Figure 12) 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 11

FIGURE 12

The procession from Figure 11 to Figure 12 and from Figure 12 to Figure 11 will be explained when we study how the data transference is performed.

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 13).

FIGURE 13

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 14, 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 14

Update matrix

Once the rotation matrices have been broadcasted, updating of matrix A proceeds as in the sequential algorithm, using BLAS3, except that rows and columns of blocks are now distributed among processors. The net improvement of BLAS3 vs. BLAS1 is not as dramatic, since in the parallel algorithm the multiplication of matrices is performed on smaller matrices.

Data transference

Data exchange between processors in even and odd steps is shown in Figures 15 and 16.

FIGURE 15

FIGURE 16

After each step, data movement consists of columns being communicated to the east and west, and rows to the north and south. 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.

Theoretical cost

Cost of Computing Rotations

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 $\frac{n}{2rs}-1=t-1$ rotations. So, the cost of this operation for each subsweep is:

$$48ts^3 = \frac{24ns^2}{r} \qquad \text{flops} \tag{13}$$

in the first step, and:

$$24ts^3 = \frac{12ns^2}{r} \qquad \text{flops} \tag{14}$$

in successive steps.

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

$$\frac{24ns^2}{r} + \left(\frac{n}{s} - 1\right)\frac{12ns^2}{r} = \frac{12n^2s}{r} + \frac{12ns^2}{r} \qquad \text{flops.} \tag{15}$$

Cost of Broadcasting Rotations

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 (also matrices sent from $P_{r-1,0}$ pass through $r-1$ links, but in this case half of the matrices are sent, Figure 13).

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 14). Thus, the cost of broadcasting rotations originated by processor $P_{0,r-1}$, and hence the entire system, is:

$$\alpha + \frac{2ns}{r}\beta + \left(\alpha + \frac{2ns}{r}\beta\right)(r-1) = r\alpha + 2ns\beta, \tag{16}$$

and the cost per sweep is:

$$\frac{n}{s}\left(r\alpha + 2ns\beta\right) = \frac{nr}{s}\alpha + 2n^2\beta, \tag{17}$$

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

Cost of Updating Matrix

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 $2\left(\frac{n}{2r}\frac{n}{2r}\right) = \frac{n^2}{2r^2}$ elements, requiring $\frac{4n^2s}{r^2}$ flops. During odd steps, processors on columns 0 must also update $2s\frac{n}{2r}$ elements, which is done in $\frac{4s^2n}{r}$ flops. Hence $\frac{4n^2s}{r^2} + \frac{4s^2n}{r}$ flops per sweep are required to update the matrix on odd steps. Thus the cost per sweep of "update matrix" is:

$$\frac{n}{2s}\frac{4n^2s}{r^2} + \frac{n}{2s}\left(\frac{4n^2s}{r^2} + \frac{4s^2n}{r}\right) = \frac{4n^3}{p} + \frac{2sn^2}{r} \qquad \text{flops.} \tag{18}$$

Cost of Transfer Data

During even 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 odd 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:

$$\frac{n}{2s}\left(8\alpha + \frac{6ns}{r}\beta\right) = \frac{4n}{s}\alpha + \frac{3n^2}{r}\beta. \qquad (19)$$

Total cost

The total arithmetic cost per sweep, given by adding (15) and (18) is

$$4k_3\frac{n^3}{p} + (2k_3 + 12k_1)\frac{n^2 s}{r} + 12k_1\frac{ns^2}{r} \qquad \text{flops,} \qquad (20)$$

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 (17) and (19), is

$$(4+r)\frac{n}{s}\alpha + \left(2n^2 + \frac{3n^2}{r}\right)\beta. \qquad (21)$$

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.

Scalability

To study theoretically the scalability of the method it is possible to study the isoefficiency function of the algorithm [17]. This function is of order $O(p)$ if we consider n the problem size and p the number of processors. But this is also the isoefficiency function of the algorithm for a logical ring presented in [21]. However, the mesh topology is more appropriate to obtain good scalability. Just to compare the storage scheme we present in this paper and that in [21] we have implemented algorithms using both storage schemes. In Table 1 the execution time in seconds obtained per sweep in different multicomputers, and with different number of processors and matrix size, are shown. In the table appear times obtained with the storage scheme for a ring when a ring topology is used to perform communications (RING), with the storage scheme for a ring when a hypercube topology is used to perform communications (CUBE), and with the storage scheme for a mesh (MESH). We can see the use of the storage scheme for a logical mesh reduces considerably the execution time even comparing with the storage scheme for a logical ring but using a hypercube topology to perform communications.

$\boxed{\text{TABLE 1}}$

Experimental results

A lot of experiments have been performed on other different systems: a multiprocessor Alliant FX/80 with 8 processors, a multicomputer iPSC/860 with 32 processors, and a multicomputer Supernode PARSYS SN-1040 with 16 processors. The implementation of the parallel algorithms on the Alliant FX/80 and the iPSC/860 has been carried out with BLAS [13]. The results are similar in the different systems and they confirm the theoretical results. Because we are mainly interested in the scalability of the algorithm, we only present results 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 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.

Sequential algorithm

Table 2 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.

TABLE 2

Table 3 shows the times per sweep when eigenvalues and eigenvectors are computed on an i860 processor. These are times obtained with the algorithm using BLAS 3, and the optimal values are marked.

TABLE 3

In this case the optimal value of s increases more quickly than when only eigenvalues are computed. This is 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.

Table 4 shows the quotient between the execution times obtained using BLAS 3 with optimal block size and using BLAS 1, when only eigenvalues are computed. This quotient tends to 1/4 when the matrix size increases.

$\boxed{\text{TABLE 4}}$

The sequential algorithms can be easily modified to obtain algorithms working in shared memory multiprocessors. Table 5 shows results obtained in an Alliant FX/80 with 8 processors. In the table the quotient between the execution times obtained using BLAS 3 and BLAS 1 are shown. The results are similar to those obtained in an i860.

$\boxed{\text{TABLE 5}}$

And Table 6 shows the Mflops obtained with the algorithm on an i860 using BLAS 3 with optimal block size, when only eigenvalues are computed and when eigenvalues and eigenvectors are computed. Again, when eigenvalues and eigenvectors are computed, a better performance is obtained.

$\boxed{\text{TABLE 6}}$

Other experiments have been done to improve this algorithm, but without success:

1. Always working with all the blocks of size $2s \times 2s$ (making each subsweep over these blocks as a complete sweep and not only over the block of size $s \times s$). In some cases, with small matrix sizes, a reduction of 1 or 2

sweeps is obtained, but the total execution time is worse than that obtained with the presented algorithm.

2. Increasing the number of subsweeps made over each block in the first sweeps. In some cases, with small matrix sizes, a small reduction in the number of sweeps is obtained, but the total execution time is also worse in this case.

Parallel algorithm

In Table 7 the execution times obtained on the Touchstone DELTA are shown, with different mesh and matrix sizes, and with block sizes 8, 16 and 32. We can observe that the optimal block size is practically 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, and this is more important when the number of processors increases.

$\boxed{\text{TABLE 7}}$

In Table 8 the Mflops/node obtained with meshes of different sizes are shown. Each value corresponds to the biggest matrix texted. This value decreases when the number of processors increases (due to the communications, the synchronizations and the use of BLAS 3 with smaller matrices).

$\boxed{\text{TABLE 8}}$

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 9. Approximately 75000 double precision elements are stored per processor. 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).

$\boxed{\text{TABLE 9}}$

To compare the experimental results with those obtained theoretically we need to know the values of the constants k_3 and k_1, and these values greatly depend on the sizes of the vectors and matrices and on the data type. Because we have worked in double precision and the optimal block size is between 16 and 32, we have performed matrix-matrix multiplications and vector rotations of the type used in the algorithm, and have obtained a mean performance of approximately 20.62 Mflops when using BLAS 3, and 7.29 Mflops when using BLAS 1. In this way, the values of k_3 and k_1 are 0.048 and 0.137 μs, respectively. The values of the start-up time and the word-sending time can be approximated by 61 and 0.149 μs, respectively [15]. With these data the expected and real times obtained with different number of processors and the matrix sizes of table 9, are shown in table 10. The expected times have been obtained using formulas 7 and 8 for the sequential algorithm, and formulas 20 and 21 for the parallel algorithm. Also the quotient between the expected and the real time is represented. We can see that when the number of processors increases the difference between the real and the expected times increases, and this is due to the overlapping of communications and computation in some parts of the parallel algorithm.

$\boxed{\text{TABLE 10}}$

Conclusions

The use of BLAS 3 to solve the symmetric eigenvalue problem by Jacobi methods can produce efficient algorithms. The performance of the method depends on the size of the blocks into which the matrix is divided, and 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.

The use of BLAS 3 permits the design of efficient algorithms independently of the machine where the program is run (supposing BLAS 3 is available for this machine).

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] 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.

[2] L. Auslander and A. Tsao. On parallelizable eigensolvers. *Advances in Applied Mathematics*, 13:253–261, 1992.

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

[4] 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.

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

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

[7] P. J. Eberlein. On one-sided Jacobi methods for parallel computation. *SIAM Journal on Algebraic Discrete Mathematic*, 8(4):790–796, 1987.

[8] 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.

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

[10] 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.

[11] G. Fox. Eigenvalues of Symmetric Tridiagonal Matrices. Unpublished, 1984.

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

[13] D. Giménez. Jacobi methods for solving the Symmetric Eigenvalue Problem on Multicomputers. Ph. D. Thesis, (in spanish). Dept de Sistemas Informáticos y Computación. Univ Politécnica de Valencia. 1995.

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

[15] R. W. Hockney and E. A. Carmona. Comparison of communications on the Intel iPSC/860 and Touchstone Delta. *Parallel Computing*, 18:1067–1072, 1992.

[16] S. Huss-Lederman, A. Tsao and G. Zhang. A parallel implementation of the invariant subspace decomposition algorithm for dense symmetric matrices. In *Proceedings Sixth SIAM Conf. on Parallel Processing for Scientific Computing*. SIAM, 1993.

[17] Vipin Kumar and Anshul Gupta. Analyzing Scalability of Parallel Algorithms and Architectures. *Journal of Parallel and Distributed Computing*, 22:379–391, 1994.

[18] 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.

[19] 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.

[20] 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.

[21] Robert 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.

Table 1: Execution times in seconds per sweep of algorithms with different storage schemes and topologies.

multicomputer	processors	size	RING	CUBE	MESH
Touchstone DELTA	64	1024	41.56	25.67	18.97
iPSC/860	16	512	14.99	12.80	11.23
PARSYS SN-1040	16	320	39.24	38.04	34.67

Table 2: Execution times in seconds per sweep of sequential algorithms on an i860 using BLAS 1 and BLAS 3 with different block sizes. Computing eigenvalues.

size	BLAS1	BLAS3. 4	BLAS3. 8	BLAS3. 16	BLAS3. 32	BLAS3. 64
128	1.14	1.27	0.92	**0.89**	1.10	
256	9.25	8.86	5.73	**4.92**	5.48	7.64
384	35.36	28.77	17.44	**14.18**	15.10	21.12
512	88.54	68.09	40.11	**30.83**	31.81	43.62
640	179.20	128.87	74.65	**57.21**	57.72	76.37
768	306.29	220.16	126.73	94.15	**92.30**	117.47
896	505.91	344.50	196.94	146.41	**141.28**	175.10
1024	738.13	510.51	291.65	211.12	**207.18**	244.39
1152	1128.88	718.58	407.78	298.87	**279.90**	332.86

Table 3: Execution times in seconds per sweep of sequential algorithm on an i860 using BLAS 3 with different block sizes. Computing eigenvalues and eigenvectors.

size	BLAS3. 4	BLAS3. 8	BLAS3. 16	BLAS3. 32	BLAS3. 64
128	2.08	1.33	**1.15**	1.27	
256	17.47	9.87	7.56	**7.38**	9.06
384	59.40	32.05	23.59	**21.96**	26.53
512	143.53	75.88	53.57	**48.23**	57.15
640		144.92	102.84	**91.48**	104.01

Table 4: Quotient between execution times of sequential algorithms on an i860 using BLAS 3 and BLAS 1. Computing eigenvalues.

size	128	256	384	512	640	768	896	1024	1152
quotient	0.78	0.53	0.40	0.34	0.31	0.30	0.27	0.28	0.24

Table 5: Quotient between execution times of the algorithms on an Alliant FX/80 with 8 processors using BLAS 3 and BLAS 1. Computing eigenvalues.

size	128	192	256	320	384	448	512	576	640
quotient	0.64	0.52	0.47	0.45	0.39	0.29	0.16	0.25	0.24

Table 6: Mflops obtained using the sequential algorithm on an i860 with BLAS 3 and optimal block sizes.

$size$	128	256	384	512	640	768	896	1024	1152
$values$	9.42	13.64	15.97	17.41	18.32	19.63	20.36	20.73	21.84
$values, vectors$	14.58	18.18	20.62	22.26	22.92				

Table 7: Execution time in seconds per sweep of the parallel algorithm on the Touchstone DELTA.

	processors								
	4			16			64		
	block size								
	8	16	32	8	16	32	8	16	32
512	12.82	**10.83**	13.36	4.62	**4.52**	5.99	**1.98**	2.01	
1024	86.92	**66.30**	71.18	27.14	**23.53**	27.90	9.62	**9.16**	12.22
1536		**202.74**	203.00		**64.80**	72.56	26.51	**23.64**	
2048					**138.08**	147.46	55.64	**47.70**	56.89

Table 8: Mflops per node obtained by the parallel algorithm on the Touschtone DELTA with different mesh sizes.

processors	1	4	9	16	25	36	64	256
Mflops/node	21.84	17.87	17.44	17.34	16.38	16.37	15.19	13.50

Table 9: Mflops per node obtained by the parallel algorithm on the Touschtone DELTA with different mesh sizes and fixed memory per node.

processors	1	4	9	16	25	36	64	256
Mflops/node	15.97	14.77	14.12	13.81	13.57	13.78	13.68	13.50
matrix size	384	768	1152	1530	1920	2302	3072	6144

Table 10: Expected and real times in seconds per sweep obtained on the Touschtone DELTA with different mesh sizes.

processors	1	4	9	16	25	36	64	256
expected time	13.42	31.45	49.83	65.05	86.53	104.55	142.61	330.25
real time	14.18	30.66	48.11	64.80	83.43	98.56	132.37	268.29
quotient	0.94	1.02	1.03	1.00	1.03	1.05	1.07	1.23
matrix size	384	768	1152	1530	1920	2302	3072	6144

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

Figure 1: Basic Jacobi iteration.

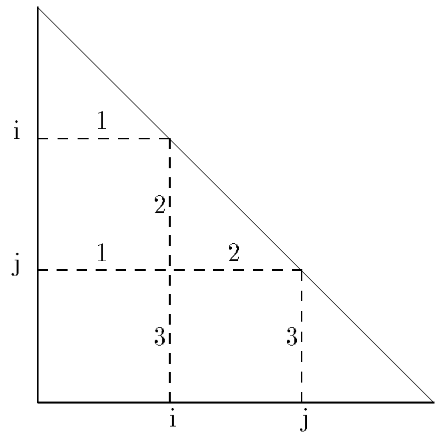


Figure 2: Applying a single Jacobi transformation to matrix A using BLAS 1, taking advantage of symmetry by updating only the lower-triangular portion of A.

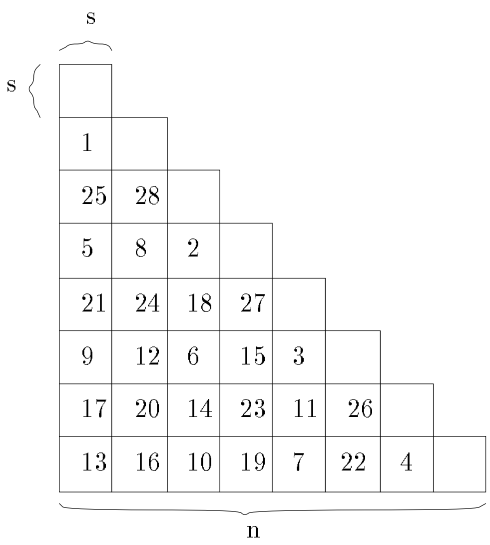


Figure 3: Odd-even order by blocks.

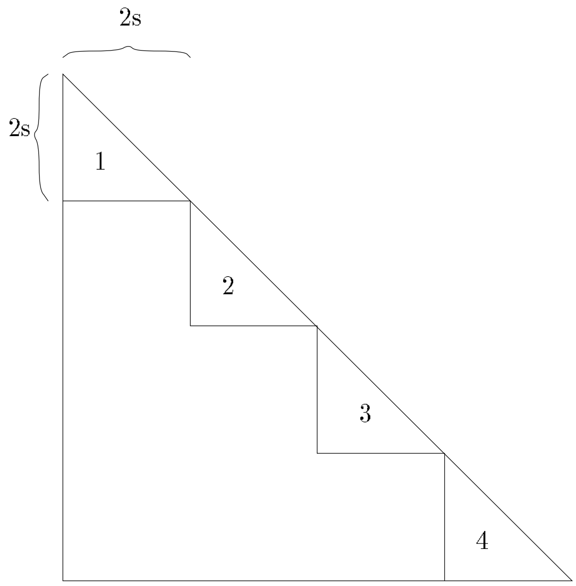


Figure 4: Work on triangular blocks.

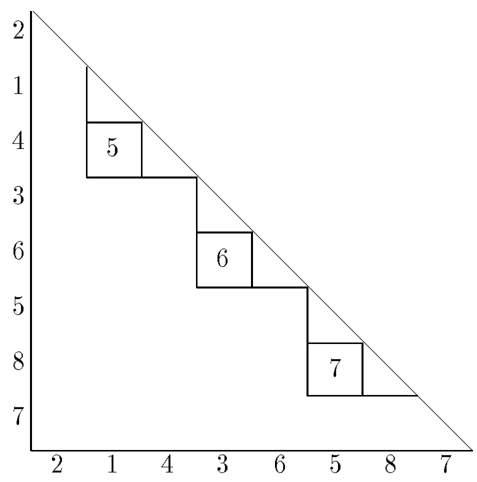


Figure 5: Work on square blocks.

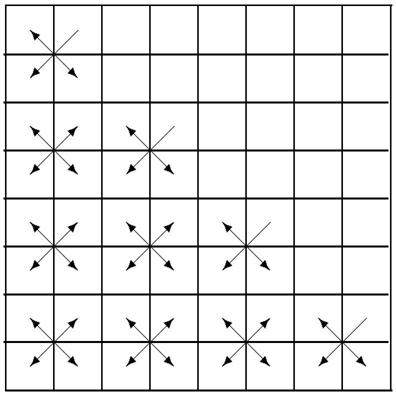


Figure 6: Movement of blocks. Odd step.

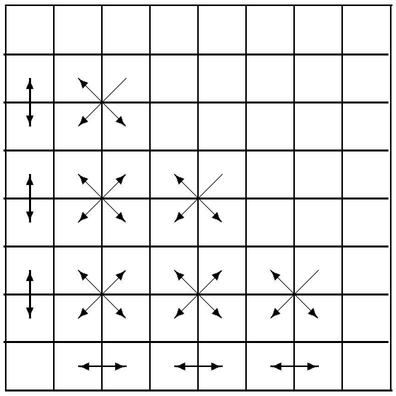


Figure 7: Movement of blocks. Even step.

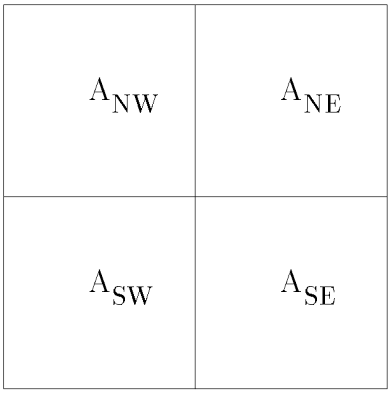


Figure 8: Partition of the matrix by blocks

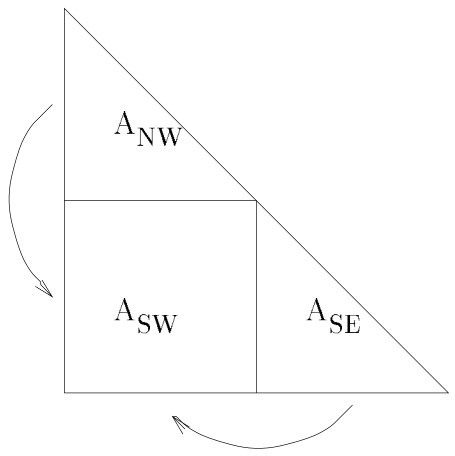


Figure 9: Folding

```
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 10: Parallel Jacobi iteration.

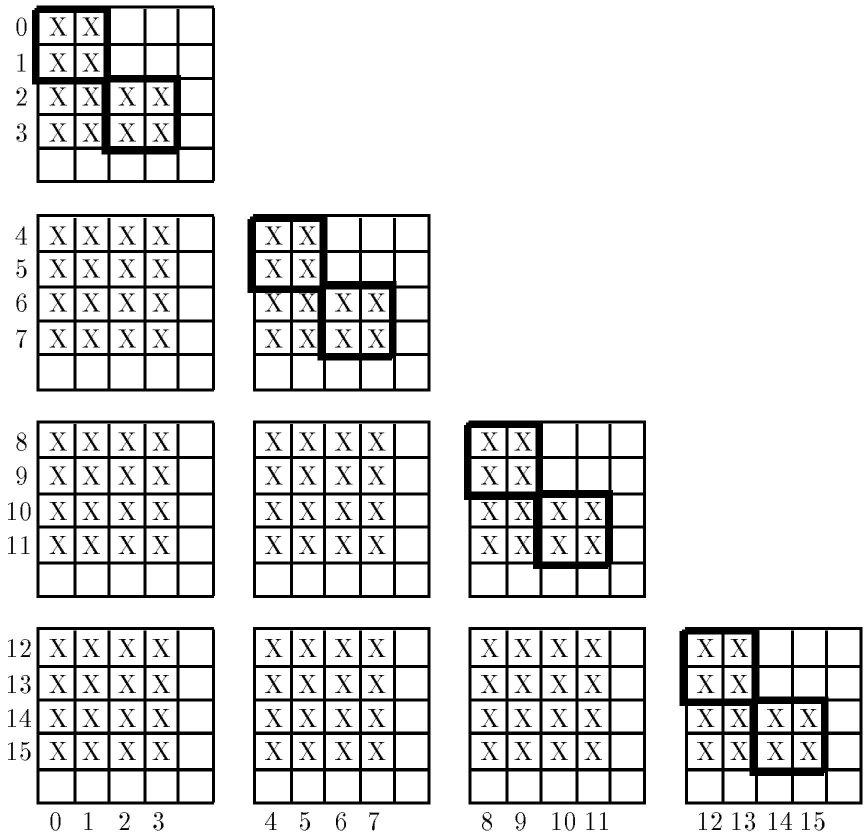


Figure 11: Computing rotations. Even step

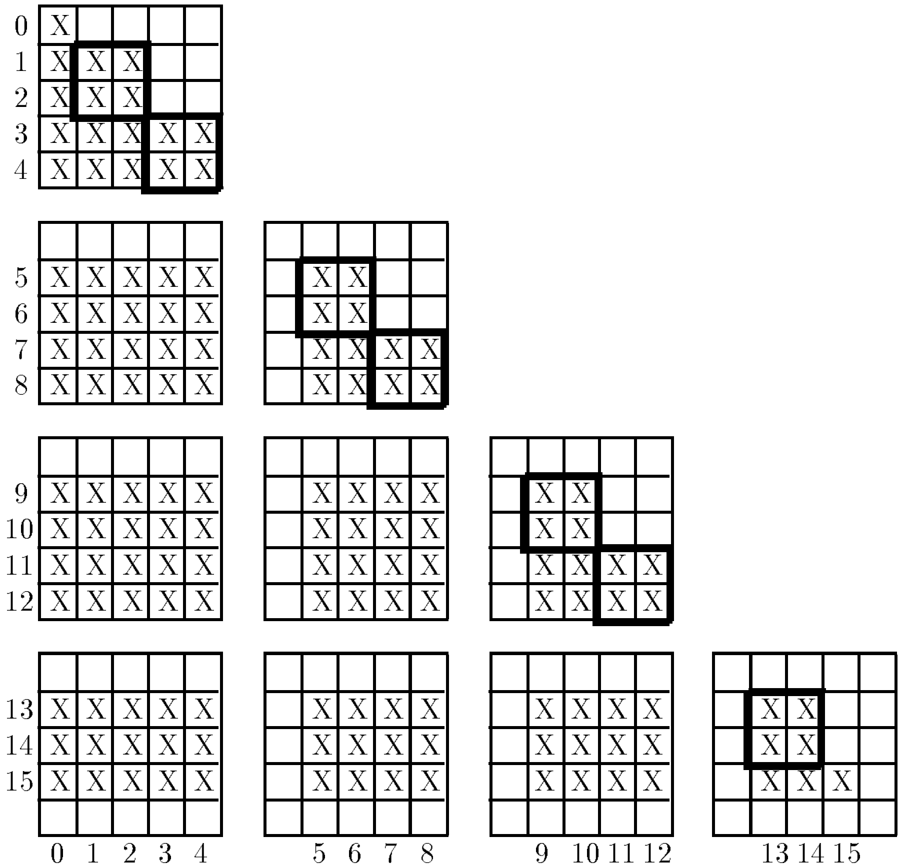
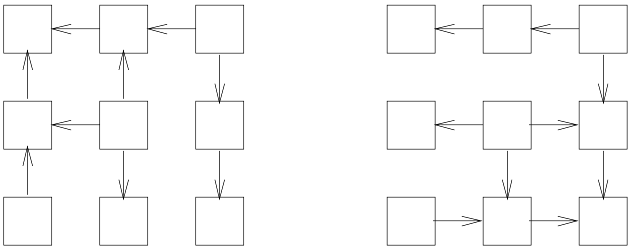


Figure 12: Computing rotations. Odd step



a b

Figure 13: Broadcast rotations. a) rotations computed in A_{NW}. b) rotations computed in A_{SE}

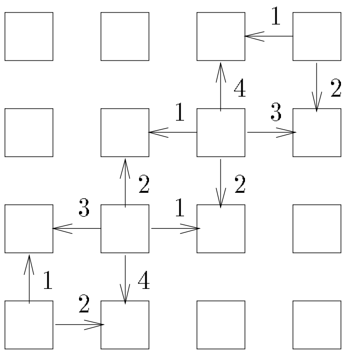


Figure 14: Order in which rotation matrices are sent

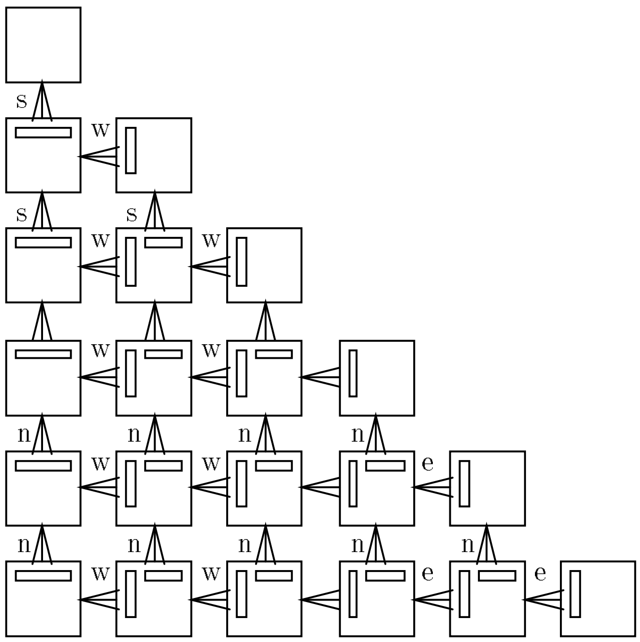


Figure 15: Data transference. Even step

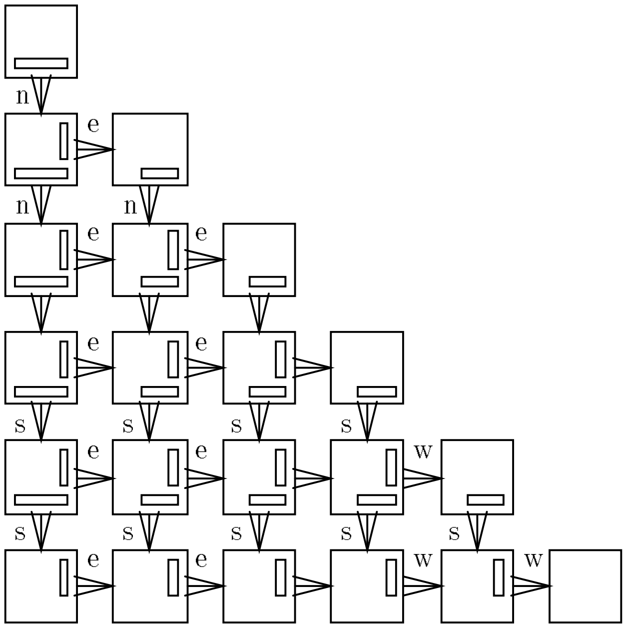


Figure 16: Data transference. Odd step