A Unified Approach to Parallel Block-Jacobi Methods for the Symmetric Eigenvalue Problem*

D. Giménez[1]**, V. Hernández[2]*** and A. M. Vidal[2]***

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

[2] Dpto. de Sistemas Informáticos y Computación. Univ Politécnica de Valencia Aptdo 22012. 46071 Valencia. Spain. {vhernand,avidal}@dsic.upv.es

Abstract. In this paper we present a unified approach to the design of different parallel block-Jacobi methods for solving the Symmetric Eigenvalue Problem. The problem can be solved designing a logical algorithm by considering the matrices divided into square blocks, and considering each block as a process. Finally, the processes of the logical algorithm are mapped on the processors to obtain an algorithm for a particular system. Algorithms designed in this way for ring, square mesh and triangular mesh topologies are theoretically compared.

1 Introduction

The Symmetric Eigenvalue Problem appears in many applications in science and engineering, and in some cases the problems are of large dimension with high computational cost, therefore it might be better to solve in parallel.

Different approaches can be utilized to solve the Symmetric Eigenvalue Problem on multicomputers:

- The initial matrix can be reduced to condensed form (tridiagonal) and then the reduced problem solved. This is the approach in ScaLAPACK [1].
- A Jacobi method can be used taking advantage of the high level of parallelism of the method to obtain high performance on multicomputers. In addition, the design of block methods allows us to reduce the communications and to use the memory hierarchy better. Different block-Jacobi methods have been designed to solve the Symmetric Eigenvalue Problem or related problems on multicomputers [2, 3, 4, 5].

* The experiments have been performed on the 512 node Paragon on the CSCC parallel computer system operated by Caltech on behalf of the Concurrent Supercomputing Consortium (access to this facility was provided by the PRISM project).

** Partially supported by Comisión Interministerial de Ciencia y Tecnología, project TIC96-1062-C03-02, and Consejería de Cultura y Educación de Murcia, Dirección General de Universidades, project COM-18/96 MAT.

*** Partially supported by Comisión Interministerial de Ciencia y Tecnología, project TIC96-1062-C03-01.

– There are other type of methods in which high performance is obtained because most of the computation is in matrix-matrix multiplications, which can be optimised both in shared or distributed memory multiprocessors. Methods of that type are those based in spectral division [6, 7, 8] or the Yau-Lu method [9].

In this paper a unified approach to the design of parallel block-Jacobi methods is analized.

2 A sequential block-Jacobi method

Jacobi methods work by constructing a matrix sequence $\{A_l\}$ by means of $A_{l+1} = Q_l A_l Q_l^t$, $l = 1, 2, \ldots$, where $A_1 = A$, and Q_l is a plane-rotation that annihilates a pair of nondiagonal elements of matrix A_l. A cyclic method works by making successive sweeps until some convergence criterion is fulfilled. A sweep consists of successively nullifying the $n(n-1)/2$ nondiagonal elements in the lower-triangular part of the matrix. The different ways of choosing pairs (i, j) have given rise to different versions of the method. The odd-even order will be used, because it simplifies a block based implementation of the sequential algorithm, and allows parallelization. With $n = 8$, 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:

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

When the method converges we have $D = Q_k Q_{k-1} \ldots Q_1 A Q_1^t \ldots Q_{k-1}^t Q_k^t$ and the eigenvalues are the diagonal elements of matrix D and the eigenvectors are the rows of the product $Q_k Q_{k-1} \ldots Q_1$.

The method works over the matrix A and a matrix V where the rotations are accumulated. Matrix V is initially the identity matrix. To obtain an algorithm working by blocks both matrices A and V are divided into columns and rows of square blocks of size $s \times s$. These blocks are grouped to obtain bigger blocks of size $2sk \times 2sk$.

The scheme of an algorithm by blocks is shown in figure 1.

In each block the algorithm works by making a sweep over the elements in the block. Blocks corresponding to the first Jacobi set are considered to have size $2s \times 2s$, adding to each block the two adjacent diagonal blocks. A sweep is performed covering all elements in these blocks and accumulating the rotations to form a matrix Q of size $2s \times 2s$. Finally, the corresponding columns and rows

```
WHILE convergence not reached DO
       FOR every pair (i,j) of indices in a sweep DO
              perform a sweep on the block of size 2s × 2s formed
                by the blocks of size s × s, A_ii, A_ij and A_jj,
                accumulating the rotations on a matrix Q of size 2s × 2s
              update matrices A and V performing matrix-matrix
                multiplications
       ENDFOR
ENDWHILE
```

Fig. 1. Basic block-Jacobi iteration.

of blocks of size $2s \times 2s$ of matrix A and the rows of blocks of matrix V are updated using Q.

After completing a set of blocked rotations, a swap of column and row blocks is performed. This brings the next blocks of size $s \times s$ to be zeroed to the subdiagonal, and the process continues nullifying elements on the subdiagonal blocks.

Because the sweeps over each block are performed using level-1 BLAS, and matrices A and V can be updated using level-3 BLAS, the cost of the algorithm is:

$$8k_3n^3 + (12k_1 - 16k_3)\,n^2s + 8k_3ns^2 \quad flops, \tag{1}$$

when computing eigenvalues and eigenvectors. In this formula k_1 and k_3 represent the execution time to perform a floating point operation using level-1 or level-3 BLAS, respectively.

3 A logical parallel block-Jacobi method

To design a parallel algorithm, what we must do first is to decide the distribution of data to the processors. This distribution and the movement of data in the matrices determine the necessities of memory and data transference on a distributed system. We begin analyzing these necessities considering processes but not processors, obtaining a logical parallel algorithm.

Each one of the blocks of size $2sk \times 2sk$ is considered as a process and will have a particular necessity of memory. At least it needs memory to store the initial blocks, but some additional memory is necessary to store data in sucessive steps of the execution.

A scheme of the method is shown in figure 2. The method using this scheme is briefly explained below.

```
On each process:
WHILE convergence not reached DO
       FOR every Jacobi set in a sweep DO
              perform sweeps on the blocks of size 2s × 2s
                corresponding to indices associated to the processor,
                accumulating the rotations
              broadcast the rotation matrices
              update the part of matrices A and V associated
                to the process, performing matrix-matrix multiplications
              transfer rows and columns of blocks of A and rows
                of blocks of V
       ENDFOR
ENDWHILE
```

Fig. 2. Basic parallel block-Jacobi iteration.

Each sweep is divided into a number of steps corresponding each step to a Jacobi set.

For each Jacobi set the rotations matrices can be computed in parallel, but working only processes associated to blocks $2sk \times 2sk$ of the main diagonal of A. On these processes a sweep is performed on each one of the blocks it contains corresponding to the Jacobi set in use, and the rotations on each block are accumulated on a rotations matrix of size $2s \times 2s$.

After the computation of the rotations matrices, they are sent to the other processes corresponding to blocks in the same row and column in the matrix A, and the same row in the matrix V. And then the processes can update the part of A or V they contain.

In order to obtain the new grouping of data according to the next Jacobi set it is necessary to perform a movement of data in the matrix, and that implies a data transference between processes and additional necessities of memory. It is illustrated in figure 3, where $\frac{n}{s} = 16$, the rows and columns of blocks are numbered from 0 to 15, and the occupation of memory on an odd and an even step is shown. In the figure blocks containing data are marked with $\times$.

Thus, to store a block of matrix A it is necessary to reserve a memory of size $(2sk + s) \times (2sk + s)$, and to store a block of V is necessary a memory of size $(2sk + s) \times 2sk$.

4 Parallel block-Jacobi methods

To obtain parallel algorithms it is necessary to assign each logical process to a processor, and this assignation must be in such a way that the work is balanced.

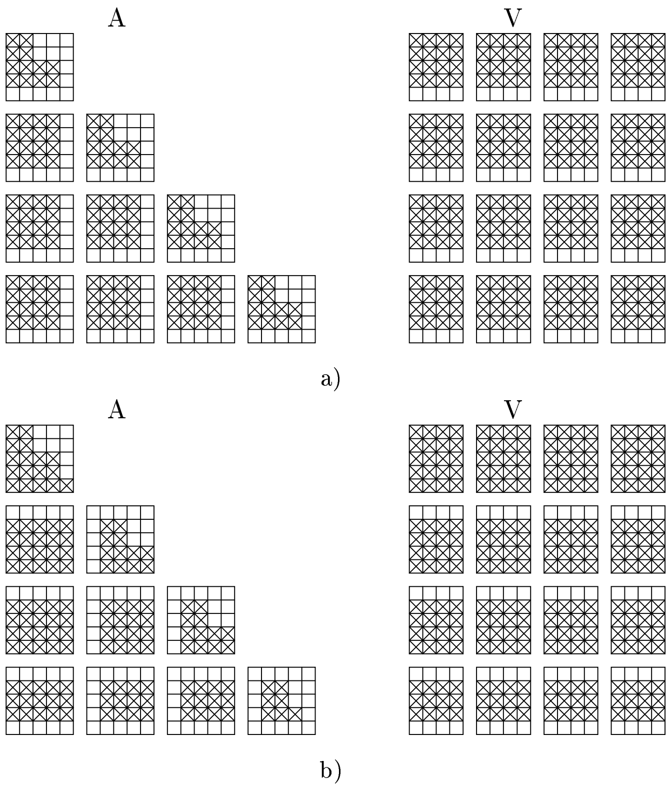

Fig. 3. Storage of data: a) on an odd step, b) on an even step.

The most costly part of the algorithm is the updating of matrices A and V (that produces the cost of order $O\left(n^3\right)$). To assign the data in a balanced way it suffices to balance the updating of the matrices only. In the updating of non-diagonal blocks of matrix A, $\frac{n}{2sk} \times \frac{n}{2sk}$ data are updated pre- and post-multiplying by rotations matrices. In the updating of diagonal blocks only elements in the lower triangular part of the matrix are updated pre- and post-multiplying by rotations matrices. And in the updating of matrix V, $\frac{n}{2sk} \times \frac{n}{2sk}$ data are updated but only pre-multiplying by rotations matrices. So, we can see the volume of computation on the updating of matrices on processes corresponding to a non-diagonal block of matrix A is twice that of processes corresponding to a block of V or a diagonal block of A. This must be had in mind when designing parallel algorithms.

An algorithm for a ring Considering $q = \frac{n}{2sk}$ and a ring with $p = \frac{q}{2}$ processors, P_0, P_1, ..., P_{p-1}, a balanced algorithm can be obtained assigning to each processor P_i, rows i and $q-1-i$ of matrices A and V. So, each processor P_i contains blocks A_{ij}, with $0 \leq j \leq i$, $A_{q-1-i,j}$, with $0 \leq j \leq q-1-i$, and V_{ij}

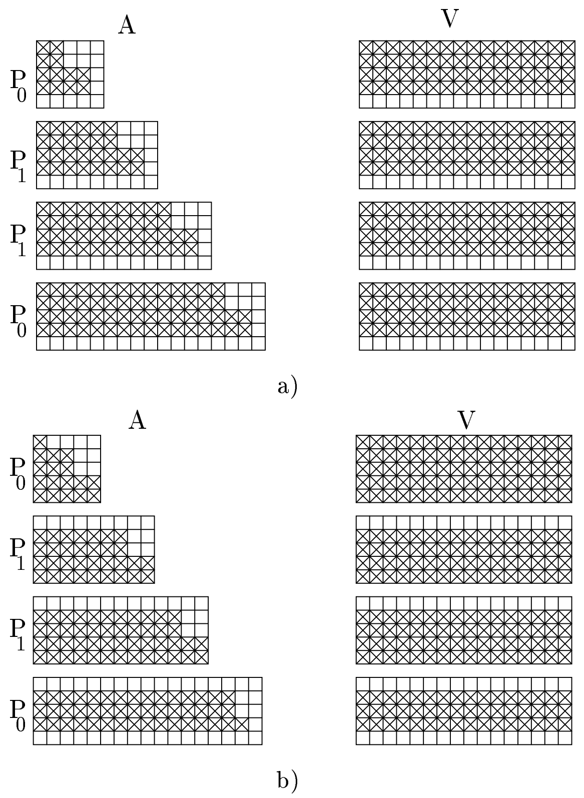

Fig. 4. Storage of data: a) on an odd step, b) on an even step. Algorithm for ring topology.

and $V_{q-1-i,j}$, with $0 \leq j < q$.

To save memory and to improve computation some of the processes in the logical method can be grouped, obtaining on each processor four logical processes, corresponding to the four rows of blocks of matrices A and V contained in the processor. In figure 4 the distribution of data and the memory reserved are shown, for $p = 2$ and $\frac{n}{s} = 16$. In that way, $(2sk + s)(3n + 2sk + 2s)$ possitions of memory are reserved on each processor.

The arithmetic cost per sweep when computing eigenvalues and eigenvectors is:

$$8k_3 \frac{n^3}{p} + (12k_1 - 8k_3)\frac{n^2 s}{p} + 12k_1 \frac{ns^2}{p} \quad flops. \tag{2}$$

And the cost per sweep of the communications is:

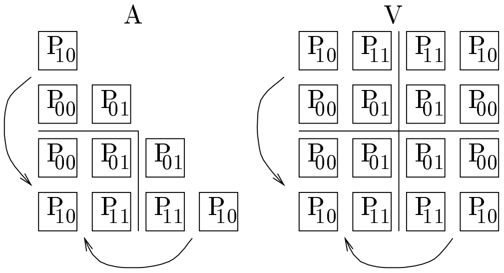

Fig. 5. Folding the matrices in a square mesh topology.

$$\beta(p+3)\frac{n}{s} + \tau\left(8n^2 + 2ns - \frac{2n^2}{p}\right) \quad , \tag{3}$$

where β represents the start-up time, and τ the time to send a double precision number.

An algorithm for a square mesh In a square mesh, a way to obtain a balanced distribution of the work is to fold the matrices A and V in the system of processors, such as is shown in figure 5, where a square mesh with four processors is considered.

To processor P_{ij} are assigned the next blocks: from matrix A block $A_{\sqrt{p}+i,j}$, if $i \leq \sqrt{p}-1-j$ block $A_{\sqrt{p}-1-i,j}$, and if $i \geq \sqrt{p}-1-j$ block $A_{\sqrt{p}+i,2\sqrt{p}-1-j}$; and from matrix V blocks $V_{\sqrt{p}+i,j}$, $V_{\sqrt{p}-1-i,j}$, $V_{\sqrt{p}+i,2\sqrt{p}-1-j}$ and $V_{\sqrt{p}-1-i,2\sqrt{p}-1-j}$. The memory reserved on each processor in the main antidiagonal is $(2sk+s)(14sk+3s)$, and in each one of the other processors $(2sk+s)(12sk+2s)$.

This data distribution produces an inbalance in the computation of the rotations matrices, because only processors in the main antidiagonal of processors work in the sweeps over blocks in the diagonal of matrix A. On the other hand, this inbalance allows us to overlap computations and communications.

The arithmetic cost per sweep when computing eigenvalues and eigenvectors is:

$$8k_3\frac{n^3}{p} + (12k_1 + 2k_3)\frac{n^2 s}{\sqrt{p}} + 12k_1\frac{ns^2}{\sqrt{p}} \quad flops. \tag{4}$$

And the cost per sweep of the communications is:

$$\beta(\sqrt{p}+4)\frac{n}{s} + \tau n^2\left(2 + \frac{7}{\sqrt{p}}\right) \quad . \tag{5}$$

Comparing equations 4 and 2 we can see the arithmetic cost is lower in the algorithm for a ring, but only in the terms of lower order. Furthermore, communications and computations can be overlapped in some parts of the algorithm for a mesh.

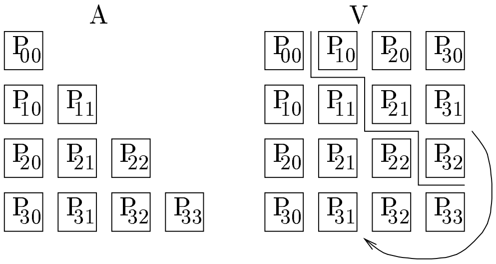

Fig. 6. The storage of matrices in a triangular mesh topology.

An algorithm for a triangular mesh In a triangular mesh, matrix A can be assigned to the processors in an obvious way, and matrix V can be assigned folding the upper triangular part of the matrix over the lower triangular part (figure 6).

To processor P_{ij} $(i \geq j)$ are assigned the blocks A_{ij}, V_{ij} and V_{ji}. The memory reserved on each processor in the main diagonal is $(2sk + s)(4sk + s)$, and in each one of the other processors $(2sk + s)(6sk + s)$.

Only processors in the main diagonal work in the sweeps over blocks in the diagonal of matrix A. In this case, as happens in a square mesh, the inbalance allows us to overlap computations and communications.

If we call r to the number of rows and columns in the processors system, r and p are related by the formula $r = \frac{-1+\sqrt{1+8p}}{2}$, and the arithmetic cost per sweep of the algorithm when computing eigenvalues and eigenvectors is:

$$16k_3\frac{n^3}{r^2} + (12k_1 + 2k_3)\frac{n^2 s}{r} + 12k_1\frac{ns^2}{r} \quad flops. \tag{6}$$

And the cost per sweep of the communications is:

$$\beta(r+7)\frac{n}{s} + \tau\left(2n^2 + \frac{6n^2}{r} + 4ns\right) \quad . \tag{7}$$

The value of r is a little less than $\sqrt{2p}$. Thus, the arithmetic cost of this algorithm is worse than that of the algorithm for a square mesh, and the same happens with the cost of communications. But when p increases the arithmetic costs tend to be equal, and the algorithm for triangular mesh is better than the algorithm for a square mesh, due to a more regular distribution of data.

5 Comparison

Comparing the theoretical costs of the algorithms studied it is possible to conclude the algorithm for a ring is the best and the algorithm for a triangular mesh

Table 1. Theoretical costs per sweep of the different parts of the algorithms.

	update matrices	*comp. rotations*	*broadcast*	*trans. data*
ring	$8\frac{n^3}{p} - 8\frac{n^2 s}{p}$	$12\frac{n^2 s}{p} + 12\frac{ns^2}{p}$	$(p-1)\frac{n}{s}\beta$ $\left(2n^2 - 2\frac{n^2}{p}\right)\tau$	$4\frac{n}{s}\beta$ $\left(6n^2 + 2ns\right)\tau$
sq. mesh	$8\frac{n^3}{p} + 2\frac{n^2 s}{\sqrt{p}}$	$12\frac{n^2 s}{\sqrt{p}} + 12\frac{ns^2}{\sqrt{p}}$	$\sqrt{p}\frac{n}{s}\beta$ $2n^2\tau$	$4\frac{n}{s}\beta$ $7\frac{n^2}{\sqrt{p}}\tau$
tr. mesh	$16\frac{n^3}{r^2} + 2\frac{n^2 s}{r}$	$12\frac{n^2 s}{r} + 12\frac{ns^2}{r}$	$r\frac{n}{s}\beta$ $2n^2\tau$	$7\frac{n}{s}\beta$ $\left(6\frac{n^2}{r} + 4ns\right)\tau$

is the worst. This can be true when using a small number of processors, but it is just the opposite when the number of processors and the matrix size increase, due to the overlapping of computations and communications on the algorithms for a mesh.

Some attention has been paid to the optimization of parallel Jacobi methods by overlapping communication and computation [10, 11], and in the mesh algorithms here analysed the inbalance in the computation of rotation matrices makes possible this overlapping. Adding the arithmetic and the communication costs in equations 2 and 3, 4 and 5, and 6 and 7, the total cost per sweep of the algorithms for ring, square mesh and triangular mesh, respectively, can be estimated; but these times have been obtained without regard to the overlapping of computation and communication. In the algorithm for a ring there is practically no overlapping because the computation of rotations is balanced, and after the computation of the rotation matrices each processor is involved in the broadcast, and the updating of matrices begins only after the broadcast finishes. But in the algorithms for mesh the computation of rotations is performed only by the processors in the main diagonal or antidiagonal in the system of processors, and this makes the overlapping possible.

To compare in more detail the three methods, in table 1 the costs per sweep of each part of the algorithms are shown.

The three algorithms have an isoefficiency function $f(n) = p$, but the algorithms for mesh are more scalable in practice. The value of the isoefficiency function appears from the term corresponding to rotations broadcast, which has a cost $O\left(n^2 p\right)$, but in the algorithm for a ring this is the real cost, because the matrices A and V can not be updated before the rotations have been broadcast. It is different in the algorithms for a mesh topology, where the execution times obtained are upper-bounds. In these algorithms the rotations broadcast can be overlapped with the updating of the matrices (as shown in [12] for systolic arrays) and when the size of the matrices increases the total cost can be better approximated by adding the costs of table 1 but without the cost of broadcast, which is overlapped with the updating of the matrices. In this way, the isoefficiency function of the algorithms for mesh topology is $f(n) = \sqrt{p}$, and these methods are more scalable.

In addition, few processors can be utilized efficiently in the algorithm for a ring, for example, with $n = 1024$, if $p = 64$ the block size must be lower or equal to 8, but when using 64 processors on the algorithm for a square mesh the block size must be lower or equal to 64.

The overlapping of communication and computation in the algorithm for triangular mesh is illustrated in figure 7. In this figure matrices A and V are shown, and a triangular mesh with 21 processors is considered. The first steps of the computation are represented writing into each block of the matrices which part of the execution is carried out: R represents computation of rotations, B broadcast of rotation matrices, U matrix updating, and D transference of data. The numbers represent which Jacobi set is involved, and an arrow indicates a movement of data between blocks in the matrices and the corresponding communication of data between processors. We will briefly explain some of the aspects in the figure:

- Step 1: Rotation matrices are computed by the processors in the main diagonal of processors.
- Step 2: The broadcast of rotation matrices to the other processors in the same row and column of processors begins.
- Step 3: Computation and communication are overlapped. All the steps in the figure have not the same cost (the first step has a cost $24k_1\frac{ns^2}{p}$ and the second $2\beta + 4\frac{ns}{p}\tau$), but if a large size of the matrices is assumed the cost of the computational parts is much bigger than that of the communication parts, therefore communication in this step finishes before computation.
- Step 4: More processors begin to compute and the work is more balanced.
- Step 5: Update of matrices has finished in processors in the main diagonal and the subdiagonal of processors, and the movement of rows and columns of blocks begins in order to obtain the data distribution needed to perform the work corresponding to the second Jacobi set.
- Step 6: The computation of the second set of rotation matrices begins in the diagonal before the updating of the matrices have finished. All the processors are involved in this step, and the work is more balanced than in the previous steps. If the cost of computation is much bigger than the cost of communication, the broadcast could have finished and all the processors could be computing.
- Step 7: Only four diagonals of processors can be involved at the same time in communications. Then, if the number of processors increases the cost of comunication becomes less important.
- Step 8: First and second updating are performed at the same time.
- Step 9: After this step all the data have been moved to the positions corresponding to the second Jacobi set.
- Step 10: When the step finishes the third set of rotations can be computed.

As we can see, there is an inbalance at the beginning of the execution, but that is compensated by the overlapping of communication and computation, which

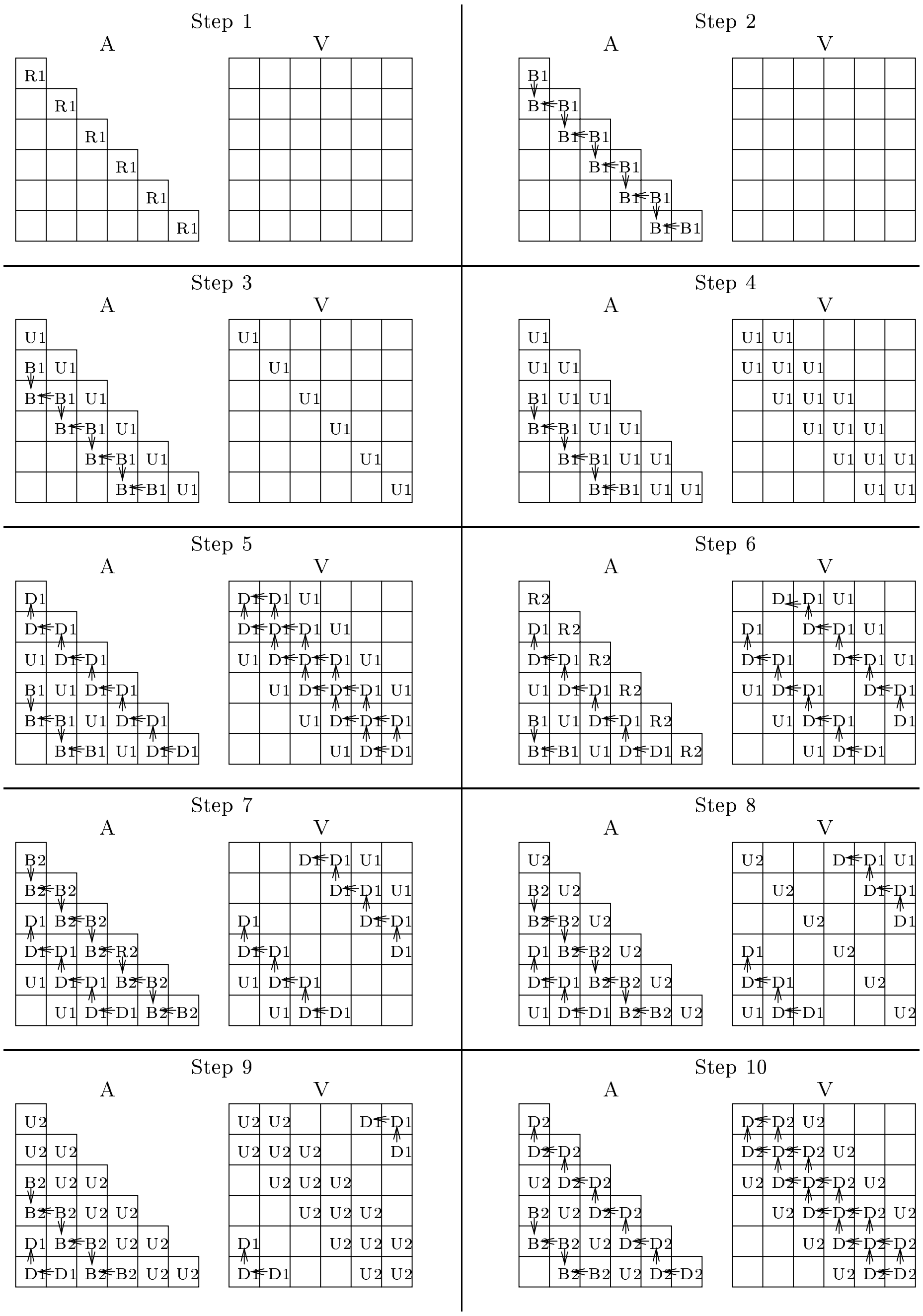

Fig. 7. Overlapping communication and computation on the algorithm for triangular mesh.

Table 2. Comparison of the algorithms for mesh topology. Execution time per sweep (in seconds), on an Intel Paragon.

matrix size :	512		1024	
processors	*triangular*	*square*	*triangular*	*square*
3	10.92		77.61	
4		7.40		47.96
10	3.67		22.71	
16		2.76		15.02
36	1.39		7.27	
64		1.21		5.53
136	0.66		2.74	
256		0.96		2.50

makes it possible to overlook the cost of broadcast to analyze the scalability of the algorithm. The same happens with the algorithm for square mesh.

The algorithm for a triangular mesh is in practice the most scalable due to the overlapping of computation and communication and also to the regular distribution of data, which produces a lower overhead than the algorithm for a square mesh. In tables 2 and 3 the two algorithms for mesh are compared. The results have been obtained on an Intel Paragon XP/S35 with 512 processors. Because the number of processors possible to use is different in both algorithms, the results have been obtained for different numbers of processors. The algorithm for a square mesh has been executed on a physical square mesh, but the algorithm for a triangular mesh has not been executed in a physical triangular mesh, but in a square mesh. This could produce a reduction on the performance of the algorithm for triangular mesh, but this reduction does not happen.

In table 2 the execution time per sweep when computing eigenvalues is shown for matrix sizes 512 and 1024.

In table 3 the Mflops per node obtained with approximately the same number of data per processor are shown. Due to the inbalance in the parallel algorithms the performance is low with a small number of processors, but when the number of processors increases the inbalance is less important and the performance of the parallel algorithms approaches that of the sequential method.

The performance of the algorithm for triangular mesh is much better when the number of processors increases. For example, for a matrix size of 1408, in a square mesh of 484 processors 2.99 Gflops were obtained, while in a triangular mesh of 465 processors 4.23 Gflops were obtained.

6 Conclusions

We have shown how parallel block-Jacobi algorithms can be designed in two steps: first associating one process to each block in the matrices, and then ob-

Table 3. Comparison of the algorithms for mesh topologies. Mflops per node with approximately the same number of data per processor, on an Intel Paragon XP/S35.

data/pro. :	4068		8192		18432		32768	
seq. :	13.10		14.71		18.63		18.39	
proc.	*tr*	*sq*	*tr*	*sq*	*tr*	*sq*	*tr*	*sq*
4		10.52		9.34		15.73		18.14
6	10.95		13.63		14.41		17.00	
15	11.46		12.78		15.53		18.30	
16		10.36		12.16		15.62		17.87
36	10.73	10.21	14.38	12.13	17.04	15.41	18.93	
64		10.15		12.14		15.46		17.65

taining algorithms for a topology by grouping processes and assigning them to processors.

Scalable algorithms have been obtained for mesh topologies and the more scalable in practice is the algorithm for a triangular mesh.

References

1. J. Demmel and K. Stanley. The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers. In David H. Bailey, Petter E. Bjørstad, John R. Gilbert, Michael V. Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia J. Torczon and Layne T. Watson, editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 528–533. SIAM, 1995.
2. Robert Schreiber. Solving eigenvalue and singular value problems on an undersized systolic array. *SIAM J. Sci. Stat. Comput.*, 7(2):441–451, 1986.
3. Gautam Schroff and Robert Schreiber. On the convergence of the cyclic Jacobi method for parallel block orderings. *SIAM J. Matrix Anal. Appl.*, 10(3):326–346, 1989.
4. Christian H. Bischof. Computing the singular value decomposition on a distributed system of vector processors. *Parallel Computing*, 11:171–186, 1989.
5. D. Giménez, V. Hernández, R. van de Geijn and A. M. Vidal. A block Jacobi method on a mesh of processors. *Concurrency: Practice and Experience*, 9(5):391–411, May 1997.
6. L. Auslander and A. Tsao. On parallelizable eigensolvers. *Ad. App. Math.*, 13:253–261, 1992.
7. 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.
8. Xiaobai Sun. Parallel Algorithms for Dense Eigenvalue Problems. In *Whorkshop on High Performance Computing and Gigabit Local Area Networks, Essen, Germany, 1996*, pages 202–212. Springer-Verlag, 1997.
9. Stéphane Domas and Françoise Tisseur. Parallel Implementation of a Symmetric Eigensolver Based on the Yau and Lu Method. In José M. L. M. Palma

and Jack Dongarra, editor, *Vector and Parallel Processing-VECPAR'96*, pages 140–153. Springer-Verlag, 1997.
10. Makan Pourzandi and Bernard Tourancheau. A Parallel Performance Study of Jacobi-like Eigenvalue Solution. Technical report, 1994.
11. El Mostafa Daoudi and Abdelhak Lakhouaja. Exploiting the symmetry in the parallelization of the Jacobi method. *Parallel Computing*, 23:137–151, 1997.
12. Richard P. Brent and Franklin T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Stat. Comput.*, 6(1):69–84, 1985.