Solving Eigenvalue Problems on Networks of Processors [⋆]

D. Giménez, C. Jiménez, M. J. Majado, N. Marín and A. Martín

Departamento de Informática, Lenguajes y Sistemas Informáticos.
Univ de Murcia. Aptdo 4021. 30001 Murcia, Spain.
{domingo,mmajado,nmarin}@dif.um.es

Abstract. In recent times the work on networks of processors has become very important, due to the low cost and the availability of these systems. This is why it is interesting to study algorithms on networks of processors. In this paper we study on networks of processors different Eigenvalue Solvers. In particular, the Power method, deflation, Givens algorithm, Davidson methods and Jacobi methods are analized using PVM and MPI. The conclusion is that the solution of Eigenvalue Problems can be accelerated by using networks of processors and typical parallel algorithms, but the high cost of communications in these systems gives rise to small modifications in the algorithms to achieve good performance.

1 Introduction

Within the different platforms that can be used to develop parallel algorithms, in recent years special attention is being paid to networks of processors. The main reasons for their use are the lesser cost of the connected equipment, the greater availability and the additional utility as a usual means of work. On the other hand, communications, the heterogeneity of the equipment, their shared use and, generally, the small number of processors used are the negative factors.

The biggest difference between multicomputers and networks of processors is the high cost of communications in the networks, due to the small bandwidth and the shared bus which allows us to send only one message at a time. This characteristic of networks makes it a difficult task to obtain acceptable efficiencies, and also lets one think of the design of algorithms with good performances on a small number of processors, more than of the design of scalable algorithms.

So, despite not being as efficient as supercomputers, networks of processors come up as a new environment to the development of parallel algorithms with a good ratio cost/efficiency. Some of the problems that the networks have can be overlooked using faster networks, better algorithms and new environments appropiate to the network features.

[⋆] 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.

The most used matricial libraries (BLAS, LAPACK [1], ScaLAPACK [2]) are not implemented for those environments, and it seems useful to programme these linear algebra libraries over networks of processors [3, 4, 5, 6]. The implementation of these algorithms can be done over programming environments like PVM [7] or MPI [8], which makes the work easier, although they do not take advantage of all the power of the equipment. The communications libraries utilized have been the free access libraries PVM version 3.4 and MPICH [9] (which is an implementation of MPI) version 1.0.11.

The results obtained using other systems or libraries could be different, but we are interested in the general behaviour of network of processors, and more particularly Local Area Networks (LANs), when solving Linear Algebra Problems. Our intention is to design a library of parallel linear algebra routines for LANs (we could call this library LANLAPACK), and we are working in Linear System Solvers [10] and Eigenvalue Solvers. In this paper some preliminary studies of Eigenvalue Solvers are shown. These problems are of great interest in different fields in science and engineering, and it is possibly better to solve them with parallel programming due to the high cost of computation [11]. The Eigenvalue Problem is still open in parallel computing, where it is necessary to know the eigenvalues efficiently and exactly. For that, we have carried out a study on the development of five methods to calculate eigenvalues over two different environments: PVM and MPI, and using networks with both Ethernet and Fast-Ethernet connections. Experiments have been performed in four different systems:

- A network of 5 SUN Ultra 1 140 with Ethernet connections and 32 Mb of memory on each processor.
- A network of 7 SUN Sparcstation with Ethernet connections and 8 Mb of memory on each processor.
- A network of 13 PC 486, with Ethernet connections, a memory of 8 Mb on each processor, and using Linux.
- A network of 6 Pentiums, with Fast-Ethernet connections, a memory of 32 Mb on each processor, and using Linux.

In this paper we will call these systems SUNUltra, SUNSparc, PC486 and Pentium, respectively.

The approximated cost of floating point operations working with double precision numbers and the theoretical cost of communicating a double precision number, in the four systems, is shown in table 1. The arithmetic cost has been obtained with medium sized matrices (stored in main memory). Also the quotient of the arithmetic cost with respect to the communication cost is shown. These approximations are presented to show the main characteristics of the four systems, but they will not be used to predict execution times because in networks of processors many factors, which are difficult to measure, influence the execution time: collisions when accessing the bus, other users in the system, assignation of processes to processors ... The four systems have also different utilization characteristics: SUNUltra can be isolated to obtain results, but the other three are shared and it is more difficult to obtain representative results.

Table 1. Comparison of arithmetic and communication costs in the four systems utilized.

	$floating\ point\ cost$	$word - sending\ time$	$quotient$
$SUNUltra$	0.025 μs	0.8 μs	32
$SUNSparc$	0.35 μs	0.8 μs	2.28
$PC486$	0.17 μs	0.8 μs	4.7
$Pentium$	0.062 μs	0.08 μs	1.29

```
given $v_0$
FOR $i = 1, 2, \ldots$
        $r_i = Av_{i-1}$
        $\beta_i = \| r_i \|_\infty$
        $v_i = \frac{r_i}{\beta_i}$
ENDFOR
```

Fig. 1. Scheme of the sequential Power method.

2 Eigenvalue Solvers

Methods of partial resolution (the calculation of some eigenvalues and/or eigenvectors) of Eigenvalue Problems are studied: the Power method, deflation technique, Givens algorithm and Davidson method; and also the Jacobi method to compute the complete spectrum. We are interested in the parallelization of the methods on networks of processors. Mathematical details can be found in many books ([12, 13, 14, 15]).

Power method. The Power method is a very simple method to compute the eigenvalue of biggest absolute value and the associated eigenvector. Some variations of the method allow us to compute the eigenvalue of lowest absolute value or the eigenvalue nearest to a given number. This method is too slow to be considered as a good method in general, but in some cases it can be useful. In spite of the bad behaviour of the method, it is very simple and will allow us to begin to analyse Eigenvalue Solvers on networks of processors.

A scheme of the algorithm is shown in figure 1. The algorithm works by generating a succession of vectors v_i convergent to an eigenvector q_1 associated to the eigenvalue λ_1, as well as another succession of values β_i convergent to the eigenvalue λ_1. The speed of convergency is proportional to $\frac{\lambda_2}{\lambda_1}$.

Each iteration in the algorithm has three parts. The most expensive is the multiplication matrix-vector, and in order to parallelize the method the attention must be concentrated on that operation. In the parallel implementation, a

```
master:
        given $v_0$
        FOR $i = 1, 2, \ldots$
                broadcast $v_{i-1}$ to the slaves
                receive $r_i^{(k)}$ from the slaves, and form $r_i$
                $\beta_i = \| r_i \|_\infty$
                compute norm
                broadcast norm to the slaves
                IF convergence not reached
                        $v_i = \frac{r_i}{\beta_i}$
                ENDIF
        ENDFOR

slave $k$, with $k = 0, 1, \ldots, p-1$:
        FOR $i = 1, 2, \ldots$
                receive $v_{i-1}$ from master
                $r_i^{(k)} = A^{(k)} v_{i-1}$
                send $r_i^{(k)}$ to master
                receive norm from master
        ENDFOR
```

Fig. 2. Scheme of the parallel Power method.

master-slave scheme has been carried out. Matrix A is considered distributed between the slave processes in a block striped partition by rows [16]. A possible scheme of the parallel algorithm is shown in figure 2. The multiplication matrix-vector is performed by the slave processes, but the master obtains β_i and forms v_i. These two operations of cost $O(n)$ could be done in parallel in the slaves, but it would generate more communications, which are very expensive operations in networks of processors.

The arithmetic cost of the parallel method is: $\frac{2n^2}{p} + 3n$ flops, and the theoretical efficiency is 100%.

The cost of communications varies with the way in which they are performed. When the distribution of vector v_i and the norm is performed using a broadcast operation, the cost of communications per iteration is: $2\tau_b(p) + \beta_b(p)(n+1) + \tau + \beta n$, where τ and β represent the start-up and the word-sending time, respectively, and $\tau_b(p)$ and $\beta_b(p)$ the start-up and the word-sending time when using a broadcast operation on a system with p slaves. If the broadcasts are replaced by point to point communications the cost of communications per iteration is $\tau(2p+1) + \beta(pn+n+p)$. Which method of communication is preferable depends on the characteristics of the environment (the communication library and the network of processors) we are using.

The parallel algorithm is theoretically optimum if we only consider efficiency, but when studying scalability, the isoefficiency function is $n \propto p^2$, and the scalability is not very good. This bad scalability is caused by the use of a shared bus which avoids sending data at the same time. Also the study of scalability is not useful in this type of system due to the reduced number of processors.

We will experimentally analyse the algorithm in the most and the least adequate systems for parallel processing (Pentium and SUNUltra, respectively). In table 2 the execution time of the sequential and the parallel algorithms on the two systems is shown, varying the number of slaves and the matrix size. Times have been obtained for random matrices, and using PVM and the routine `pvm_mcast` to perform the broadcast in figure 2. The results greatly differ in the two systems due to the big difference in the proportional cost of arithmetic and communication operations. Some conclusions can be obtained:

- Comparing the execution time of the sequential and the parallel algorithms using one slave, we can see the very high penalty of communications in these systems, especially in SUNUltra.
- The best execution times are obtained with a reduced number of processors because of the high cost of communications, this number being bigger in Pentium. The best execution time for each system and matrix size is marked in table 2.
- The availability of more potential memory is an additional benefit of parallel processing, because it allows us to solve bigger problems without swapping. For example, the parallel algorithm in SUNUltra is quicker than the sequential algorithm only when the matrix size is big and the sequential execution produces swapping.
- The use of more processes than processors produces in Pentium with big matrices better results than one process per processor, and this is because communications and computations are better overlapped.

The basic parallel algorithm (figure 2) is very simple but it is not optimized for a network of processors. We can try to improve communications in at least two ways:

- The broadcast routine is not optimized for networks, and it would be better if we replace this routine by point to point communications.
- The diffusion of the norm and the vector can be assembled in only one communication. In that way more data are transferred because the last vector need not be sent, but less communications are performed.

In table 3 the execution time of the basic version (version 1), the version with point to point communications (version 2), and the version where the diffusion of norm and vector are assembled (version 3) are compared. Versions 2 and 3 reduce the execution time in both systems, and the reduction is clearer in SUNUltra because of the bigger cost of communication in this system.

Until now, the results shown have been those obtained with PVM, but the use of MPI produces better results (obviously it depends on the versions we are using). In table 4 the execution time obtained with the basic version of the

Table 2. Execution time (in seconds) of the Power method using PVM, varying the number of processors and the matrix size.

	sequential	1 slave	2 slaves	3 slaves	4 slaves	5 slaves	6 slaves	7 slaves	8 slaves
SUNUltra									
300	**0.069**	0.149	0.255	0.289	0.381	0.547	0.526	0.552	0.638
600	**0.292**	0.497	0.450	0.491	0.629	0.864	0.838	0.849	0.954
900	**0.599**	0.882	0.679	0.735	1.065	1.491	1.457	1.284	1.603
1200	4.211	7.582	1.277	**1.062**	1.426	1.722	1.940	2.004	
1500	23.613	54.464	**1.481**	1.796	1.901	2.233	2.324	2.421	
Pentium									
300	**0.172**	0.281	0.250	0.188	0.292	0.323	0.407	0.405	0.436
600	0.592	1.153	0.639	**0.569**	0.608	0.682	0.599	0.662	0.656
900	1.302	1.762	1.903	1.272	0.908	0.892	**0.842**	0.891	1.171
1200	2.138	8.141	1.544	1.750	1.568	1.224	1.275	1.231	**1.169**
1500	3.368	254.42	2.776	1.904	4.017	2.431	1.911	1.750	**1.680**

Table 3. Comparison of the execution time of the Power method using PVM, with different communication strategy.

	2 slaves			3 slaves			4 slaves		
	ver 1	ver 2	ver 3	ver 1	ver 2	ver 3	ver 1	ver 2	ver 3
SUNUltra									
300	0.255	0.187	**0.168**	0.289	0.262	**0.187**	0.381	0.339	**0.203**
600	0.450	0.418	**0.363**	0.491	0.430	**0.386**	0.629	0.605	**0.341**
900	0.679	0.570	**0.566**	0.735	0.666	**0.567**	1.065	0.688	**0.542**
1200	1.277	1.089	**0.877**	1.062	1.045	**0.854**	1.426	0.949	**0.911**
Pentium									
300	0.250	**0.157**	0.271	0.188	0.188	**0.141**	0.292	0.294	**0.192**
600	0.639	**0.574**	0.585	0.569	**0.516**	0.599	0.608	0.603	**0.557**
900	1.903	**1.879**	2.714	1.272	**0.983**	1.090	0.908	**0.813**	0.985

programme using MPI on SUNUltra is shown. Comparing this table with table 2 we can see the programme with MPI works better when the number of processors increases.

Deflation technique. Deflation technique is used to compute the next eigenvalue and its associated eigenvector starting from a previously known one. This technique is used to obtain some eigenvalues and it is based on the transformation of the initial matrix to another one that has got the same eigenvalues, replacing λ_1 by zero.

To compute `numEV` eigenvalues of the matrix A, the deflation technique can

Table 4. Execution time (in seconds) of the Power method using MPI, varying the number of processors and the matrix size, on SUNUltra.

	1 slave	2 slaves	3 slaves	4 slaves	5 slaves	6 slaves
300	0.15	**0.14**	0.22	0.28	0.27	0.38
600	0.39	**0.31**	**0.31**	0.37	0.55	0.62
900	0.73	0.52	**0.47**	0.56	0.61	0.66
1200	3.58	0.76	**0.70**	0.71	0.79	0.93
1500	22.16	1.37	**0.97**	0.99	1.06	0.94

```
$A_1 = A$
FOR $i = 1, 2, \ldots, numEV$
        compute by the Power method $\lambda_i$ and $q_i^{(i)}$
        update matrix $A$: $A_{i+1} = B_{i+1} A_i$
        compute $q_i$ from $q_i^{(i)}$
ENDFOR
```

Fig. 3. Scheme of the deflation technique.

be used performing `numEV` steps (figure 3), computing in each step, using the Power method, the biggest eigenvalue (λ_i) and the corresponding eigenvector ($q_i^{(i)}$) of a matrix A_i, with $A_1 = A$. Each matrix A_{i+1} is obtained from matrix A_i using $q_i^{(i)}$, which is utilized to form matrix B_{i+1}:

$$B_{i+1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & -q_1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & -q_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & -q_{k-1} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & -q_{k+1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & -q_n & 0 & \cdots & 1 \end{bmatrix} ,$$

where ${q_i^{(i)}}^t = (q_1, \ldots, q_{k-1}, 1, q_{k+1}, \ldots, q_n)$.

The eigenvalue λ_i is the biggest eigenvalue in absolute value of matrix A_i, and it is also the i-th eigenvalue in absolute value of matrix A. The eigenvector q_i associated to the eigenvalue λ_i in matrix A is computed from the eigenvector $q_i^{(i)}$ associated to λ_i in the matrix A_i. This computation is performed repeatedly applying the formula $q_i^{(w)} = q_i^{(w+1)} + \frac{{a_k^{(w)}}^t q_i^{(w+1)}}{\lambda_i - \lambda_w} q_w^{(w)}$, where ${a_k^{(w)}}^t$ is the k-th

Table 5. Execution time (in seconds) of the deflation method using PVM, varying the number of processors and the matrix size, when computing 5% of the eigenvalues.

	sequential	1 slave	2 slaves	3 slaves	4 slaves	5 slaves
SUNUltra						
300	**11.4**	29.6	25.2	28.5	32.4	33.7
600	**184.7**	308.3	239.7	252.1	244.2	273.2
900	914.4	1258.3	862.5	826.6	**765.7**	831.0
Pentium						
300	25.0	32.4	29.9	23.0	22.4	**22.2**
600	407.0	462.4	288.9	239.5	195.8	**194.6**
900	2119.0	2316.5	1460.3	993.7	823.2	**720.7**

column of matrix A_w, with k the index where $q_w^{(w)} = 1$.

The most costly part of the algorithm is the application of the Power method, which has a cost of order $O\left(n^2\right)$ per iteration. Therefore, the previously explained Power method can be applied using a scheme master-slave. The cost of the deflation part (update matrix) is $2n^2$ flops, and the cost of the computation of q_i depends on the step and is $5n(i-1)$ flops. These two parts can be performed simultaneously in the parallel algorithm: the master process computes q_i while the slaves processes update matrix A_i. The deflation part of the parallel algorithm (update matrix and compute q_i) is not scalable if a large number of eigenvalues are computed. As we have previously mentioned, scalability is not very important in these types of systems. In addition, this method is used to compute only a reduced number of eigenvalues, due to its high cost when computing a large number of them.

In table 5 the execution time of the sequential and parallel algorithms are shown on SUNUltra and Pentium, using PVM and the basic version of the Power method. Compared with table 2, we can see the behaviour of the parallel Power and deflation algorithms is similar, but that of the deflation technique is better, due to the high amount of computation, the work of the master processor in the deflation part of the execution and the distribution of data between processes in the parallel algorithm.

Davidson method. The Power method is a very simple but not very useful method to compute the biggest eigenvalue of a matrix. Other more efficient methods, as for example Davidson methods [17] or Conjugate Gradient methods [18, 19], can be used.

The Davidson algorithm lets us compute the highest eigenvalue (in absolute value) of a matrix, though it is especially suitable for large, sparse and symmetric matrices. The method is valid for real as well as complex matrices. It works by building a sequence of search subspaces that contain, in the limit, the desired eigenvector. At the same time these subspaces are built, so approximations to the desired eigenvector in the current subspace are also built.

```
$V_0 = [\ ]$
given $v_1$
$k = 1$
WHILE convergence not reached
       $V_k = [V_{k-1}|v_k]$
       orthogonalize $V_k$ using modified Gram-Schmith
       compute $H_k = V_k^c A V_k$
       compute the highest eigenpair $(\theta_k, y_k)$ of $H_k$
       obtain the Ritz vector $u_k = V_k y_k$
       compute the residual $r_k = Au_k - \theta_k u_k$
       IF $k = k_{max}$
              reinitialize
       ENDIF
       obtain $v_{k+1} = (\theta_k I - D)^{-1} r_k$
       $k = k + 1$
ENDWHILE
```

Fig. 4. Scheme of the sequential Davidson method.

Figure 4 shows a scheme of a sequential Davidson method. In successive steps a matrix V_k with k orthogonal column vectors is formed. After that, matrix $H_k = V_k^c A V_k$ is formed and the biggest eigenvalue in absolute value (θ_k) and its associated eigenvector (y_k) are computed. This can be done using the Power method, because matrix H_k is of size $k \times k$ and k can be kept small using some reinitialization strategy (when $k = k_{max}$ the process is reinitialized). The new vector v_{k+1} to be added to the matrix V_k to form V_{k+1} can be obtained with the succession of operations: $u_k = V_k y_k$, $r_k = Au_k - \theta_k u_k$ and $v_{k+1} = (\theta_k I - D)^{-1} r_k$, with D the diagonal matrix which has in the diagonal the diagonal elements of matrix A.

To obtain a parallel algorithm the cost of the different parts in the sequential algorithm can be analysed. The only operations with cost $O(n^2)$ are two matrix-vector multiplications: Av_k in the computation of H_k and Au_k in the computation of the residual. AV_k can be accomplished in order $O(n^2)$ because it can be decomposed as $[AV_{k-1}|Av_k]$, and AV_{k-1} was computed in the previous step. The optimum value of k_{max} varies with the matrix size and the type of the matrix, but it is small and it is not worthwhile to parallelize the other parts of the sequential algorithm. Therefore, the parallelization of the Davidson method is done basically in the same way as the Power method: parallelizing matrix-vector multiplications.

This method has been parallelized using a master-slave scheme (figure 5), working all the processes in the two parallelized matrix-vector multiplications, and performing the master process non parallelized operations. In that way,

```
master:
    $V_0 = [\ ]$
    given $v_1$
    $k = 1$
    WHILE convergence not reached
        $V_k = [V_{k-1}|v_k]$
        orthogonalize $V_k$ using modified Gram-Schmith
        send $v_k$ to slaves
        in parallel compute $Av_k$ and accumulate in the master
        compute $H_k = V_k^c\,(AV_k)$
        compute the highest eigenpair $(\theta_k, y_k)$ of $H_k$
        obtain the Ritz vector $u_k = V_k y_k$
        send $u_k$ to slaves
        in parallel compute $Au_k$ and accumulate in the master
        compute the residual $r_k = Au_k - \theta_k u_k$
        IF $k = k_{max}$
            reinitialize
        ENDIF
        obtain $v_{k+1} = (\theta_k I - D)^{-1}\, r_k$
        $k = k + 1$
    ENDWHILE

slave $k$, with $k = 1, \ldots, p-1$:
    WHILE convergence not reached
        receive $v_k$ from master
        in parallel compute $Av_k$ and accumulate in the master
        receive $u_k$ from master
        in parallel compute $Au_k$ and accumulate in the master
        IF $k = k_{max}$
            reinitialize
        ENDIF
        $k = k + 1$
    ENDWHILE
```

Fig. 5. Scheme of the parallel Davidson method.

matrix A is distributed between the processes and the result of the local matrix-vector multiplications is accumulated in the master and distributed from the master to the other processes.

Because the operations parallelized are matrix-vector multiplications, as in the Power method, the behaviour of the parallel algorithms must be similar, but better results are obtained with the Davidson method because in this case the master process works in the multiplications.

Table 6 shows the execution time of 50 iterations of the Davidson method

Table 6. Execution time of the Davidson method using MPI, varying the number of processors and the matrix size, on PC486.

	sequ	p=2	p=3	p=4	p=5	p=6	p=7	p=8
300	0.048	0.051	0.048	0.045	0.043	**0.041**	**0.041**	0.042
600		0.129	0.097	0.083	0.076	0.072	0.065	**0.064**
900				0.165	0.131	0.118	0.119	**0.112**

Table 7. Efficiency of the Givens method using PVM, varying the number of processors, on SUNUltra and SUNSparc, with matrix size 100.

	$p=2$	$p=3$	$p=4$	$p=5$	$p=2$	$p=3$	$p=4$	$p=5$
	all the eigenvalues				*20% of the eigenvalues*			
SUNUltra	0.72	0.63	0.55	0.49	0.52	0.38	0.26	0.19
SUNSparc	1.17	0.94	0.60	0.64	0.81	0.61	0.36	0.37

for symmetric complex matrices on PC486 and using MPI, varying the number of processors and the matrix size.

Givens algorithm or bisection. As we have seen in previous paragraphs, on parallel Eigenvalue Solvers whose cost is of order $O\left(n^2\right)$ only a small reduction in the execution time can be achieved in networks of processors in some cases: when the matrices are big or the quotient between the cost of communication and computation is small.

In some other Eigenvalue Solvers the behaviour is slightly better. For example, the bisection method is an iterative method to compute eigenvalues in an interval or the k biggest eigenvalues. It is applicable to symmetric tridiagonal matrices. This method is especially suitable to be parallelized, due to the slight communication between processes, which factor increases the total time consumed in a network. When computing the eigenvalues in an interval, the interval is divided in subintervals and each process works in the computation of the eigenvalues in a subinterval. When computing the k biggest eigenvalues, each slave knows the number of eigenvalues it must compute. After that, communications are not necessary but inbalance is produced by the distribution of the spectrum. More details on the parallel bisection method are found in [20].

The eigenvalues are computed by the processes performing successive iterations, and each iteration has a cost of order $O(n)$. Despite the low computational cost good performance is achieved because communications are not necessary after the subintervals are broadcast. Table 7 shows the efficiency obtained using this method to calculate all the eigenvalues or only 20% of them, on SUNUltra and SUNSparc for matrix size 100. The efficiencies are clearly better than in the previous algorithms, even with small matrices and execution time.

Table 8. Theoretical speed-up of the Jacobi method, varying the number of processes and processors.

	$p = 2$	$p = 3$	$p = 4$	$p = 5$	$p = 6$	$p = 7$	$p = 8$	$p = 9$	$p = 10$
3	2	2	2	2	2	2	2	2	2
6		3	3	4.5	4.5	4.5	4.5	4.5	4.5
10			4	4	5.3	5.3	8	8	8

Jacobi method. The Jacobi method for solving the Symmetric Eigenvalue Problem works by performing successive sweeps, nullifying once on each sweep the $n(n-1)/2$ nondiagonal elements in the lower-triangular part of the matrix.

It is possible to design a Jacobi method considering the matrix A of size $n \times n$, dividing it into blocks of size $t \times t$ and doing a sweep on these blocks, regarding each block as an element. The blocks of size $t \times t$ are grouped into blocks of size $2kt \times 2kt$ and these blocks are assigned to the processors in such a way that the load is balanced. Parallel block Jacobi methods are explained in more detail in [21].

In order to obtain a distribution of data to the processors, an algorithm for a logical triangular mesh can be used. The blocks of size $2kt \times 2kt$ must be assigned to the processors in such a way that the work is balanced. Because the most costly part of the execution is the updating of the matrix, and nondiagonal blocks contain twice more elements to be nullified than diagonal blocks, the load of nondiagonal blocks can be considered twice the load of diagonal blocks.

Table 8 shows the theoretical speed-up of the method when logical meshes of 3, 6 or 10 processes are assigned to a network, varying the number of processors in the network from 2 to 10. Higher theoretical speed-up is obtained increasing the number of processes. This produces better balancing, but also more communications and not always a reduction of the execution time. It can be seen in table 9, where the execution time per sweep for matrices of size 384 and 768 is shown, varying the number of processors and processes. The shown results have been obtained in PC486 and SUNUltra using MPI, and the good behaviour of the parallel algorithm is observed because a relatively large number of processors can be used reducing the execution time.

3 Conclusions

Our goal is to develop a library of linear algebra routines for LANs. In this paper some previous results on Eigenvalue Solvers are shown. The characteristics of the environment propitiate small modifications in the algorithms to adapt them to the system. In these environments we do not generally have many processors, and also, when the number of processors goes up, efficiency unavoidably goes down. For these reasons, when designing algorithms for networks of processors it is preferable to think on good algorithms for a small number of processors,

Table 9. Execution time per sweep of the Jacobi method on PC486 and SUNUltra using MPI.

	p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=8	p=9	p=10
SUNUltra: 384 (non swapping)										
1	6.25									
3		4.78	**4.55**							
6			7.08	7.31	8.26					
SUNUltra: 768 (non swapping)										
1	50.78									
3		28.37	**25.50**							
6			39.51	33.11	32.18					
PC486: 384 (non swapping)										
1	43.26									
3		30.32	26.81							
6			28.01	23.15	19.81	18.53				
10				42.20	27.15	30.79	21.46	19.82	**15.49**	17.44
PC486: 768 (swapping)										
1	698.8									
3		217.4	195.6							
6			187.2	142.1	111.6	104.6				
10				158.9	145.6	106.5	96.9	83.5	76.5	**72.5**

and not on scalable algorithms. Because of the great influence of the cost of communications, a good use of the available environments -MPI or PVM- is essential.

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, 1995.
2. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R. C. Whaley. ScaLAPACK User's Guide. SIAM, 1997.
3. J. Demmel and K. Stanley. The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers. In D. H. Bailey, P. E. Bjørstad, J. R. Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon and L. T. Watson, editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 528–533. SIAM, 1995.
4. D. Giménez, M. J. Majado and I. Verdú. Solving the Symmetric Eigenvalue Problem on Distributed Memory Systems. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications. PDPTA'97*, pages 744–747, 1997.

5. Kuo-Chan Huang, Feng-Jian Wang and Pei-Chi Wu. Parallelizing a Level 3 BLAS Library for LAN-Connected Workstations. *Journal of Parallel and Distributed Computing*, 38:28–36, 1996.
6. Gen-Ching Lo and Yousef Saad. Iterative solution of general sparse linear systems on clusters of workstations. May 1996.
7. A. Geist, A. Begelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.. The MIT Press, 1995.
8. Message Passing Interface Forum. A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, (3), 1994.
9. Users guide to mpich. preprint.
10. F. J. García and D. Giménez. Resolución de sistemas triangulares de ecuaciones lineales en redes de ordenadores. Facultad de Informática. Universidad de Murcia. 1997.
11. A. Edelman. Large dense linear algebra in 1993: The parallel computing influence. *The International Journal of Supercomputer Applications*, 7(2):113–128, 1993.
12. G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1989. Segunda Edición.
13. L. N. Trefethen and D. Bau III. *Numerical Linear Algebra.* SIAM, 1997.
14. David S. Watkins. *Matrix Computations.* John Wiley & Sons, 1991.
15. J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, 1965.
16. V. Kumar, A. Grama, A. Gupta and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* The Benjamin Cummings Publishing Company, 1994.
17. E. R. Davidson. The Iterative Calculation of a Few of the Lowest Eigenvalues and Corresponding Eigenvectors of Large Real-Symmetric Matrices. *Journal of Computational Physics*, 17:87–94, 1975.
18. W. W. Bradbury and R. Fletcher. New Iterative Methods for Solution of the Eigenproblem. *Numerische Mathematik*, 9:259–267, 1966.
19. A. Edelman and S. T. Smith. On conjugate gradient-like methods for eigenvalue-like problems. *BIT*, 36(3):494–508, 1996.
20. J. M. Badía and A. M. Vidal. Exploiting the Parallel Divide-and-Conquer Method to Solve the Symmetric Tridiagonal Eigenproblem. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, Madrid, January 21-23*, 1998.
21. D. Giménez, V. Hernández and A. M. Vidal. A Unified Approach to Parallel Block-Jacobi Methods for the Symmetric Eigenvalue Problem. In *Proceedings of VECPAR'98*, 1998.