LANLAPACK: Solving Eigenvalue Problems on Networks of Processors *

Javier Cuenca†, Domingo Giménez‡, María José Majado‡, Norberto Marín‡, Isidro Verdú‡

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 multicomputers, 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.

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, 7]. The implementation of these algorithms can be done over programming environments like PVM [8] or MPI [9], which makes the work easier, although they do not take advantage of all the power of the equipment. Clearly, a way to increase the performance of programs over networks is to use environments (PVM, MPI, ...) specifically designed for them [10], but we are interested in algorithms and we will analyse them using non

*Partially supported by Comisión Interministerial de Ciencia y Tecnología, project TIC96-1062-C03-02.

†Departamento de Arquitectura y Tecnología de Ordenadores. Univ de Murcia. Aptdo 4021. 30001 Murcia, Spain. (jcuenca@ditec.um.es)

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

optimized routines. The communication libraries utilized have been the free access libraries PVM version 3.4 and MPICH [11] (which is an implementation of MPI) version 1.0.13.

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 on Linear System Solvers [12, 13, 14] and Eigenvalue Solvers [15, 16]. 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 [17]. In this paper we will analyse the behaviour of different Eigenvalue Solvers over networks of processors. 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 using algorithms accessing the data by rows, with the matrices also stored by rows (we use C to programme the algorithms) and with medium sized matrices stored in main memory. The communication costs shown in table 1 are obtained from the theoretical speed of the networks: the bandwidth of Ethernet is 10Mb/seg and the bandwidth of Fast-Ethernet 100Mb/seg. These are theoretical bandwidths, but the speeds achieved experimentally are lower [12, 18], especially in the network with Fast-Ethernet, where the experimental bandwidth is less than half of the theoretical bandwidth. Also the quotient of the communication cost with respect to the arithmetic 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
*PC*486	0.17 μs	0.8 μs	4.7
Pentium	0.062 μs	0.08 μs	1.29

```
given v_0
FOR i = 1, 2, ...
        r_i = A v_{i-1}
        β_i = || r_i ||_∞
        v_i = r_i / β_i
ENDFOR
```

Figure 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 ([19, 20, 21, 22, 23]).

2.1 Power method

2.1.1 The sequential 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. The method not always converges, but in most of the cases it does, for example when the matrix has one dominant eigenvalue. This method is too slow to be considered as a good method in general, but in some cases it can be useful, for example, when the matrix is sparse, when a good approximation of the eigenvector associated to the dominant eigenvalue is known, or when the dominant eigenvalue is much bigger than the other eigenvalues. 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 $\frac{\lambda_2}{\lambda_1}$.

```
master:
        given v_0
        FOR i = 1, 2, ...
                broadcast v_{i-1} to the slaves
                receive r_i^(k) from the slaves, and form r_i
                β_i = || r_i ||_∞
                compute norm
                broadcast norm to the slaves
                IF convergence not reached
                        v_i = r_i / β_i
                ENDIF
        ENDFOR

slave k, with k = 0, 1, ..., p - 1:
        FOR i = 1, 2, ...
                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
```

Figure 2: Scheme of the parallel Power method.

2.1.2 A basic parallel implementation

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-slave scheme has been carried out. Matrix A is considered distributed between the slave processes in a block striped partition by rows [24]. A possible scheme of the parallel algorithm is shown in figure 2, and a scheme of communications and computations in each step of the algorithm is shown in figure 3. 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.

2.1.3 Execution time

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

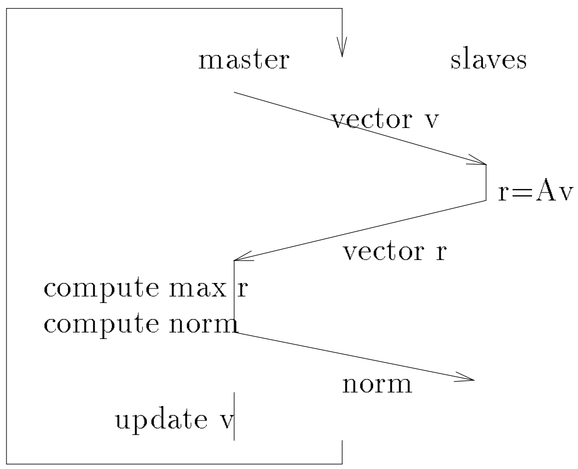

Figure 3: Scheme of communications and computations performed by the master and slaves processes in the parallel Power method.

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, and to study the algorithms in order to decide in which way to perform communications it is necessary to do a previous study of the experimental values of these parameters in the system.

2.1.4 Scalability

The parallel algorithm is theoretically optimum if we only consider efficiency (the theoretical efficiency when n increases with a fixed number of processors is 100%), 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.

Of course, the presented algorithm is a very simple algorithm and it could be improved in some ways. Some modifications will be analysed further.

2.1.5 Experimental results

We will experimentally analyse the algorithm in the most and the least adequate systems for parallel processing (Pentium and SUNUltra, respectively).

Experimental results of the sequential method. We begin by analyzing the performance achieved by the sequential algorithm in a machine of SUNUltra and Pentium.

SUNUltra has five processors of the same type, but one of them (`camelot`) manages the file system, and for this reason the processors cannot be considered as equal. In table 2 the execution time, and the Mflops of the sequential Power method when executed in `camelot` and in other of the machines (`merlin`) is shown. Also the results obtained in a processor of Pentium and the iterations needed to achieved the convergence are shown (the stopping criterium is that

Table 2: Execution time (in seconds) and Mflops of the sequential Power method on two different processors of SUNUltra and in a processor of Pentium varying the matrix size.

	SUNUltra: merlin		SUNUltra: camelot		Pentium		
	time	Mflops	time	Mflops	time	Mflops	iterations
100	0.0076	63.39	**0.0073**	65.68	0.021	22.98	24
200	**0.023**	65.72	0.063	24.21	0.085	17.81	19
300	**0.069**	41.55	0.192	15.03	0.172	16.72	16
400	**0.134**	35.93	0.793	6.06	0.290	16.53	15
500	**0.197**	35.55	0.540	12.97	0.421	16.61	14
600	**0.292**	32.04	0.667	14.03	0.593	15.80	13
700	**0.370**	34.46	1.136	11.21	0.831	15.33	13
800	**0.447**	34.39	1.499	10.25	1.024	15.00	12
900	**0.599**	32.43	1.553	12.52	1.302	14.93	12
1000	**1.283**	18.71	1.907	12.58	1.609	14.92	12
1100	2.568	11.31	**1.810**	16.05	1.950	14.89	12
1200	4.211	7.52	2.315	13.68	**2.139**	14.81	11
1300	9.107	4.08	3.356	11.08	**2.518**	14.77	11
1400	16.403	2.63	7.553	5.71	**3.024**	14.26	11
1500	23.613	2.63	11.243	4.40	**3.369**	14.69	11
1600	23.614	2.10	20.025	2.81	**3.907**	14.41	11
1700	30.137	1.87	432.230	0.147	**14.287**	4.45	11
1800	388.717	0.164	456.644	0.156	**42.635**	1.67	11
1900	**414.968**	0.171	468.225	0.154			11

the norm is lower than 1.0E-10). The results have been obtained for random matrices with double precision numbers. For each matrix size, the lowest execution time is marked.

With reference to this table we observe the following:

- As we have mentioned, `camelot` and `merlin` are machines of the same type, but the results obtained in them are very different. `merlin` is quicker than `camelot` (that controls the file system) for small matrices (until size 1000×1000), for medium matrices (from 1100 until 1600) `camelot` is quicker, and for big matrices (bigger than 1700) `merlin` is again quicker. The only conclusion we can obtain is that even with SUNUltra being an homogeneous system, **the processors in the system have different capacities**, and to obtain good performance it is preferable not to use `camelot` because it manages the file system.

- The results in SUNUltra have been obtained with an optimized C, with compilation option `-fast`. It produces good performance with small matrices (65 Mflops are achieved), but when the matrix size increases the system makes swapping and the performance decreases very quickly. This happens normally with matricial programs [18], and to avoid that algorithms by blocks are designed [1]. The most costly operation in the Power method is the matrix-vector multiplication, which is a level 2 operation, and a **block version of the algorithm cannot be designed**.

- The use of parallel programming will allow us to distribute the data between the processors and **delay swapping**, obtaining superlinear speed-up.

Table 3: Mflops of matrix-matrix multiplications on a processor of SUNUltra, with square matrices and varying the matrix size.

	no BLAS	BLAS1	BLAS3
40	69.04	53.55	80.59
80	72.55	62.36	78.74
120	72.55	66.18	72.13
160	68.47	62.16	73.99
200	43.34	41.11	56.14
240	38.07	30.55	41.65
280	12.69	12.16	37.52
320	8.07	7.91	36.21
360	10.03	9.84	35.35
400	8.71	8.55	34.78
440	8.63	8.45	34.63
480	6.78	6.70	34.40

- If we compare the performance with that obtained in the multiplication matrix-matrix, we can conclude the **performance achieved by the sequential Power method is acceptable** and comparable to that achieved in the system. Table 3 shows the Mflops obtained in the matrix-matrix multiplication for different matrix sizes, and when the multiplication is performed non using BLAS, using BLAS 1 or using BLAS 3.

- In Pentium the GNU C compiler and the option `-O3` have been used. Comparing the results of SUNUltra and Pentium we can see the machines in SUNUltra are quicker than that in Pentium, but this difference is caused in part by the compiler. When the matrix size is big better results are obtained in Pentium, but with parallel programming we will distribute data to avoid swapping.

Experimental results with PVM. In table 4 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. The execution time fluctuates in some cases, and that is because in some of the experiments we do not control which processors work in the solution of the problem, and it is possible to have less than the maximum number of processors in the system, or `pvm` can assign more than one process to the same processor, or can assign one process to `camelot` while it could be preferable to use another processor. In spite of all these inconveniences, 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 4.

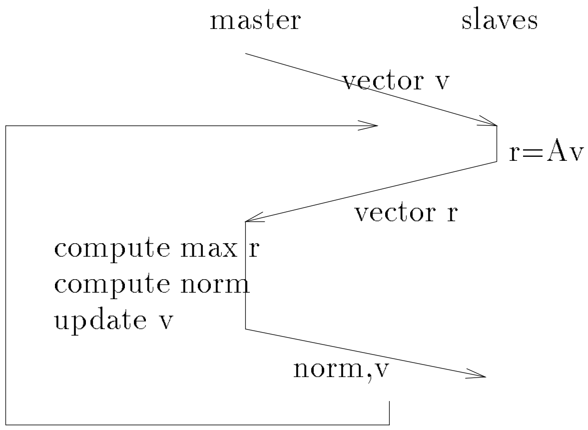

Figure 4: Scheme of communications and computations performed by the master and slaves processes in the parallel Power method.

- 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 lower execution time is smaller in SUNUltra than in Pentium. Pentium has a quicker network, but it does not compensate its bigger computational cost.

Improving the basic algorithm. 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. A scheme of communications and computations in each step of the algorithm is shown in figure 4.

In table 5 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 SUNUltra, but this reduction is not so clear in Pentium. The reduction is clear in SUNUltra because of the bigger cost of communication in this system, but in Pentium the reduction happens only in

Table 4: 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	9 slaves
SUNUltra										
100	**0.0076**	0.0957	0.218	0.225	0.345	0.380	0.401	0.464	0.394	
200	**0.023**	0.124	0.222	0.327	0.326	0.462	0.578	0.470	0.577	
300	**0.069**	0.150	0.255	0.289	0.417	0.547	0.526	0.552	0.535	
400	**0.134**	0.225	0.321	0.336	0.502	0.735	0.692	0.630	0.638	
500	**0.197**	0.405	0.354	0.412	0.614	0.769	0.806	0.895	0.951	
600	**0.292**	0.497	0.450	0.573	0.707	0.864	0.839	0.849	0.954	
700	**0.370**	0.619	0.642	0.647	0.843	1.055	0.994	0.920	1.180	
800	**0.447**	0.723	0.591	0.621	0.885	1.337	1.140	1.194	1.086	
900	**0.599**	0.883	0.680	0.735	1.065	1.491	1.457	1.284	1.604	
1000	1.283	1.040	**0.808**	0.900	1.197	1.634	1.414	1.573	1.686	
1100	2.568	1.542	1.106	**1.012**	1.391	1.841	1.850	1.808		
1200	4.211	7.205	1.173	**1.062**	1.427	1.723	1.880	2.007		
1300	9.107	27.360	**1.187**	1.572	1.548	2.066	1.965	2.002		
1400	16.403	43.871	**1.308**	1.717	1.771	2.104	2.014	2.249		
1500	23.613	53.752	**1.482**	1.797	2.021	2.233	2.325	2.422		
1600	23.614	76.590	7.329	**2.452**	2.645	2.869	2.457	2.619		
1700	30.137	383.088	9.942	**2.491**	2.646	2.693	2.685	2.511		
1800	388.717			**3.638**						
1900	414.968			**2.816**						
Pentium										
100	**0.0209**	0.138	0.142	0.176	0.200	0.251	0.580	0.307		
200	**0.0853**	0.224	0.160	0.171	0.184	0.252	0.477	0.440		
300	**0.172**	0.189	0.250	0.189	0.191	0.265	0.560	0.557	0.436	
400	0.290	0.431	0.293	0.307	**0.263**	0.298	0.540	0.462		
500	0.421	0.519	0.477	**0.321**	0.481	0.442	0.612	0.639		
600	0.593	2.150	**0.486**	0.544	0.619	0.502	0.589	0.631	0.656	
700	0.831	0.941	1.376	0.825	**0.589**	0.709	0.667	0.764		
800	1.024	3.007	1.243	1.040	0.876	**0.750**	0.753	0.818		
900	1.302	1.578	0.877	1.150	0.970	0.852	**0.799**	0.886	1.171	
1000	1.609	1.914	1.787	1.779	1.226	0.950	**0.942**	1.019		
1100	1.950	2.235	1.246	1.119	1.305	1.003	**0.975**	1.110	1.162	1.291
1200	2.139	8.142	1.544	1.751	1.569	1.225	1.275	1.232	**1.170**	1.333
1300	2.518	78.453	5.097	1.297	1.779	1.429	1.240	1.360	**1.236**	1.539
1400	3.024	27.522	2.278	3.552	2.024	1.724	1.821	2.286	**1.512**	1.940
1500	3.369	254.426	2.776	1.904	4.017	2.432	1.912	1.751	**1.681**	2.255
1600	3.907	126.415	40.014	4.246	2.574	2.478	2.291	2.177	**1.960**	3.267
1700	14.287		7.290	4.468	3.823	**1.994**	2.684	2.710	2.873	7.223
1800	42.635		8.404	7.260	3.171	**2.100**	2.799	3.376	2.718	8.974
1900				35.295	6.335	6.388	2.644	**2.643**	2.927	8.235

Table 5: Comparison of the execution time of the Power method using PVM, with different communication strategy.

	2 slaves			3 slaves			4 slaves			5 slaves		
	ver 1	ver 2	ver 3	ver 1	ver 2	ver 3	ver 1	ver 2	ver 3	ver 1	ver 2	ver 3
SUNUltra												
100	0.218	0.137	**0.114**	0.225	0.193	**0.139**	0.345		**0.150**	0.380		**0.217**
200	0.222	0.193	**0.137**	0.327	0.226	**0.162**	0.326		**0.162**	0.462		**0.226**
300	0.255	0.188	**0.168**	0.289	0.263	**0.187**	0.417	0.339	**0.203**	0.547		**0.270**
400	0.321	0.236	**0.230**	0.336	0.320	**0.234**	0.502		**0.237**	0.735		**0.316**
500	0.354	0.299	**0.298**	0.412	0.532	**0.282**	0.614		**0.277**	0.769		**0.345**
600	0.450	0.418	**0.364**	0.573	0.673	**0.386**	0.707	0.605	**0.342**	0.864		**0.492**
700	0.642	0.497	**0.435**	0.647	0.681	**0.425**	0.843		**0.399**	1.055		**0.601**
800	0.591	0.512	**0.485**	0.621	0.616	**0.503**	0.885		**0.424**	1.337		**0.555**
900	0.680	0.570	**0.567**	0.735	0.667	**0.568**	1.065	0.688	**0.542**	1.491		**0.691**
1000	0.808	0.656	**0.643**	0.900	0.746	**0.641**	1.197		**0.694**	1.634		**0.805**
1100	1.106	0.956	**0.826**	1.012	1.183	**0.783**	1.391		**1.141**	1.841		**1.682**
1200	1.173	1.090	**0.877**	1.062	1.046	**0.855**	1.427	0.949	**0.911**	1.723		**1.307**
1300	1.187	**0.969**	0.985	1.572	1.302	**0.958**	**1.548**		1.554	2.066		**1.559**
1400	**1.308**	1.595		1.717	1.317	**1.070**	1.771		**1.725**	**2.104**		4.554
1500	**1.482**	1.798		1.797	1.564		**2.021**		14.469	**2.233**		5.321
Pentium												
100	0.142	**0.106**		0.176	**0.134**		**0.200**	0.204		0.251	**0.250**	
200	0.160	**0.127**		**0.171**	0.272		0.184	**0.146**		**0.252**	0.271	
300	0.250	**0.157**	0.271	0.189	0.188	**0.141**	**0.191**	0.294	0.192	0.265	0.291	**0.175**
400	0.293	**0.280**		0.307	**0.248**		**0.263**	0.346		**0.298**	0.375	
500	0.477	**0.444**		**0.321**	0.435		0.481			**0.442**	0.496	
600	**0.486**	0.574	0.585	0.544	**0.516**	0.599	0.619	**0.532**	0.557	**0.502**	0.518	0.591
700	1.376	**1.059**		**0.825**	1.033		**0.589**	0.792		**0.709**	1.466	
800	**1.243**	1.576		1.040	**0.883**		0.876	**0.821**		**0.750**	0.797	
900	**0.877**	1.577	2.714	1.150	**0.983**	1.090	0.970	**0.813**	0.985	**0.852**	0.866	2.071
1000	**1.787**	2.223		1.779	**1.608**		**1.226**	1.391		**0.950**	1.022	
1100	**1.246**	2.438		**1.119**	2.483		**1.305**	1.424		**1.003**	1.112	
1200	**1.544**	9.938	6.731	**1.751**	2.088	11.826	**1.569**	1.624	4.494	**1.225**	1.263	2.647

small problems, where to optimize communications is more critical. Anyway, the experiments performed in SUNUltra are more confidential than those in Pentium, because SUNUltra can be used without interferences from other users, and also because in SUNUltra more experiments have been performed.

If we do not use a master process more processes are working in the resolution of the problem and a more reduced execution time can be obtained. One of the processes can work as the master and also work in the multiplication matrix-vector. The accumulation and diffusion of the vector and the norm can be done without centralizing the work in a process, and not centralizing these communications reduces the execution time [13].

Experimental results with MPI. 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 6 the execution time obtained with the basic version of the programme using MPI on SUNUltra is shown. Comparing this table with table 4 we can see the programme with MPI works better when the number of processors increases.

In table 7 the execution time of the Power method with programmed broadcast and assembled communications (version 3) is shown. There is no difference between these times and those in

Table 6: Execution time (in seconds) of the Power method (basic version) 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**	0.23	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

Table 7: Execution time (in seconds) of the Power method (version 3) using MPI, varying the number of processors and the matrix size, on SUNUltra

	Packed communication						Copy of data		
	1 slave	2 slaves	3 slaves	4 slaves	5 slaves	6 slaves	1 slave	2 slaves	3 slaves
300	0.16	**0.13**	0.17	0.24	0.62	0.60	0.16	0.14	0.17
600	0.40	**0.31**	**0.31**	0.35	0.42	0.53	0.40	**0.31**	**0.31**
900	0.74	0.52	**0.49**	0.58	0.61	0.69	0.87	0.81	0.79
1200	4.62	0.79	**0.70**	0.75	0.84	0.96	1.34	0.81	0.79
1500	21.17	1.13	**0.97**	0.98	1.04	1.11	22.00	1.35	1.14

table 6. The optimization of communications is not so important in MPI as in PVM, possibly because MPI is more optimized, as we can conclude by the better execution time obtained with MPI. In the table, the time obtained when the messages are packed using the routine of MPI are compared with the time obtained when the packing is done in the programme. It is preferable to use the MPI routine, especially when the matrix size increases.

Experimental Speed-up. In order to compare the possibilities of parallelism with different systems, in table 8 the speed-up achieved in the different systems is shown. The systems are SUNUltra with PVM, SUNUltra with MPI, Pentium with PVM and SUNSparc with PVM. The speed-up in the three first systems are obtained from the previous tables, and the results in SUNSparc are obtained from [15]. The speed-up are obtained dividing the sequential time in the system with respect to the smaller execution time in the system. The table also shows the number of slaves from which the smaller execution time is obtained.

Comparison with LAPACK. When designing Eigenvalue Solvers (sequential or parallel) it is interesting to compare the implemented algorithm with standard libraries ([1, 2]). The routine `dsyevx` of LAPACK can be used to compute a number of eigenvalues and eigenvectors of a matrix. Table 9 shows the execution time of `dsyevx` when computing the highest eigenvalue and its associated eigenvector in a processor of Pentium. `dsyevx` is a routine of general purpose and it is not optimized for the computation of one eigenvalue, which makes the Power method preferable. The comparison can be done also with ScaLAPACK, but this package solves the problem reducing the matrix to tridiagonal form (as LAPACK does) and then solving the tridiagonal problem, and because the sequential Power method is quicker than `dsyevx` and ScaLAPACK

Table 8: Comparison of the speed-up achieved in different systems.

	SUNUltra PVM		SUNUltra MPI		Pentium PVM		SUNSparc PVM	
	speed-up	slaves	speed-up	slaves	speed-up	slaves	speed-up	slaves
100	0.079	1			0.20	2	1.12	1
200	0.19	1			0.67	2		
300	0.46	1	0.53	2	1.22	3		
400	0.60	1			1.19	3		
500	0.95	2			1.31	3	3.52	6
600	0.85	4	0.94	2	1.22	2		
700	0.93	4			1.41	4		
800	1.05	4			1.37	5		
900	1.11	4	1.27	3	1.63	6		
1000	2.00	3			1.71	6	4.82	6
1100	3.28	3			2.00	6		
1200	4.93	3	6.02	3	1.83	7		
1300	9.51	3			2.04	7		
1400	15.33	3			2.00	7		
1500	15.93	2	24.34	3	2.00	7		
1600	9.63	3			1.99	7		
1700	12.10	3			7.16	4		
1800	106.85	3			20.30	4		
1900	147.36	3						

Table 9: Execution time of the routine `dsyevx` of LAPACK in Pentium, when computing the highest eigenvalue and its associated eigenvector.

100	200	300	400	500	600
0.168	0.542	1.850	4.443	8.986	32.194

does not work well in networks of processors [4, 5, 18], the comparison with ScaLAPACK is not useful. It could be more adequate to compare with sparse software, as ARPACK [25], and we are working in this comparison.

2.2 Deflation technique

2.2.1 The sequential method

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 be used performing `numEV` steps (figure 5), 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

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

Figure 5: Scheme of the deflation technique.

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 column of matrix A_w, with k the index where $q_w^{(w)} = 1$. In that way, q_i is obtained computing $q_i^{(i-1)}$ from $q_i^{(i)}$, $q_i^{(i-2)}$ from $q_i^{(i-1)}$, etcetera, until $q_i = q_i^{(1)}$ is obtained from $q_i^{(2)}$.

2.2.2 A parallel implementation

The most costly part of the algorithm is the application of the Power method, which has a cost of order $O(n^2)$ 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. In that way the master process works in some parts of the computation, but because it needs additional memory, it will produce memory problems earlier than the Power method. Table 10 shows the execution time of the deflation part of the algorithm (computation of A_{i+1} from A_i in the slaves, and computation of q_i from $q_i^{(i)}$ in the master) when computing 20% of the eigenvalues in SUNSparc, and we can see the overlapping of computation in master and slave processes produces good results. The computation of the eigenvector of matrix A from the eigenvector of matrix A_i could be parallelized to avoid memory

Table 10: Execution time of the deflation part in SUNSparc, when computing 20% of the eigenvalues.

	seq	1 slave	2 slaves	3 slaves	4 slaves	5 slaves	6 slaves	7 slaves	8 slaves	10 slaves
100	4.07	2.17	2.22	2.32	2.26	2.22	2.64	2.29	2.10	**1.86**
500	92.98	77.40	44.22	28.15	29.97	24.80	**23.47**	26.91	31.24	32.28
1000	715.73	343.23	276.78	213.31	211.61	194.64	**128.15**	178.81	192.08	197.54

```
master:
        FOR i = 1,2,...,numEV
                obtain λ_i and q_i^(i) applying the parallel Power method to A_i
                receive row a_m corresponding to the index m with v_i(m) = 1
                compute q_i from q_i^(i)
        ENDFOR

slave k, with k = 0,1,...,p-1:
        FOR i = 1,2,...,numEV
                obtain λ_i and q_i^(i) applying the parallel Power method to A_i
                IF index m with v_k(m) = 1 corresponds to processor P_k
                        broadcast a_m to master and slaves
                ELSE
                        receive a_m
                ENDELSE
                update the part of A it contains
        ENDFOR
```

Figure 6: Scheme of the parallel deflation technique.

problems, but we have not done this parallelization. The experiments performed have been with not very big matrices, and memory problems do not appear.

The scheme of the parallel algorithm is shown in figure 6, and the computations and communications in figure 7.

2.2.3 Scalability

The deflation part of the parallel algorithm (update matrix and compute q_i) is not scalable if a large number of eigenvalues are computed, because the computation of the eigenvector of matrix A from the eigenvector of A_i is not parallelized. 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.

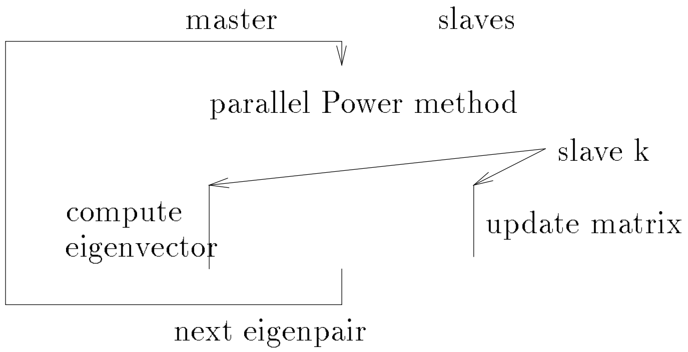

Figure 7: Scheme of communications and computations performed by the master and slave processes in the parallel deflation technique.

2.2.4 Experimental results

Execution time. In table 11 the execution time of the sequential and parallel algorithms are shown on SUNUltra and Pentium. The number of iterations performed is much bigger in that case than in the Power method, which produces a longer execution time. The speed of convergence depends on the separation of the eigenvalues, and can vary greatly from one matrix to another. The matrices used are symmetric and with eigenvalues chosen to ensure they do not converge too slowly.

Some comparisons can be done:

- Compared with table 4, 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 and the work of the master process in the deflation part of the execution.
- MPI is normally better than PVM, as we observed in the Power method.
- SUNUltra is more powerful than Pentium, and for small matrix size better execution time is obtained in SUNUltra, but the quicker network in Pentium makes it possible to obtain lower execution time in Pentium than in SUNUltra for big matrices.

Comparison with LAPACK. As we have said, the speed of convergence depends on the distribution of the spectrum of the matrix, but normally a high execution time is needed to compute a not too small number of eigenvalues, which makes the method not competitive with other methods.

In table 12 the execution time of the routine `dsyevx` when computing a variable number of the eigenvalues and eigenvectors is shown. Comparing with table 9 we can see the execution time does not increases to much with respect to the execution time obtained when computing only one eigenvalue and eigenvector. This is caused by the way in which `dsyevx` works reducing the initial matrix to tridiagonal form and solving the reduced problem. In general, we can consider the deflation technique appropriate only when a very small number of eigenpairs are computed.

Table 11: Execution time (in seconds) of the deflation method, 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	iterations
SUNUltra PVM							
100		2.5	2.8	3.9			363
200		14.6	14.5	21.0			1259
300	**11.5**	29.6	25.3	28.5	32.4	33.7	2647
400		90.7	92.0	135.3			4491
500		179.8	165.9	262.9			6725
600	**184.7**	308.4	239.7	252.1	244.2	273.3	9576
700		588.1	572.1				12931
800		892.4	907.7	1106.8			16619
900	914.4	1258.3	862.6	826.6	**765.8**	831.1	20755
1000		1958.9	1743.8	2354.7			25374
SUNUltra MPI							
100		1.06	1.47	2.18	2.90		363
300	**11.5**	24	22	29	39	45	2647
500		146	116	120	151		6725
900	914.4	1188	826	**796**	881		20755
Pentium PVM							
100		1.6	2.2	2.5			363
200		11.7	11.1	9.9	13.0	13.8	1259
300	25.0	42.5	29.9	23.0	22.4	**22.2**	2647
400		113.9	67.8	59.8	75.8	77.4	4491
500		252.1	148.1	113.7	105.9	143.8	6725
600	407.0	462.4	288.9	232.0	195.8	**194.6**	9576
700		964.9	545.1	412.1	346.7	372.2	12931
800		1615.9	944.5	659.5	560.3	524.3	16619
900	2119.0	2316.5	1460.3	993.7	823.2	**720.7**	25374

2.3 Davidson method

2.3.1 The sequential 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 [26] or Conjugate Gradient methods [27, 28], 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.

Figure 8 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

Table 12: Execution time of the routine `dsyevx` of LAPACK in Pentium, when computing a variable number of the eigenvalues and eigenvectors.

	Number of eigenvalues					
	5	10	15	20	50	all
100	0.139	0.191	0.196	0.256	0.411	0.739
200	0.795	0.950	0.970	1.083	1.519	3.922
300	2.599	2.786	2.901	3.070	3.942	11.356
400	6.360	7.104	6.930	7.339	8.495	25.259

$V_0 = [\ ]$
`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 (`θ_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`

Figure 8: Scheme of the sequential Davidson method.

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.

2.3.2 A parallel implementation

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.

Table 13: 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 14: Speed-up of the matrix-vector multiplication and the Davidson method using MPI, varying the number of processors and the matrix size.

$matrix - vector$							
	$p=2$	$p=3$	$p=4$	$p=5$	$p=6$	$p=7$	$p=8$
100	0.91	0.94	1.10	1.33	1.45	1.52	1.45
200	1.47	1.84	2.40	2.95	3.08	3.15	3.08
300	1.71	2.58	2.96	3.55	3.93	3.97	4.07
400	1.65	2.33	2.65	3.77	4.22	4.65	4.76
$Davidson\ method$							
	$p=2$	$p=3$	$p=4$	$p=5$	$p=6$	$p=7$	$p=8$
100	1.00	0.96	0.95	0.93	0.99	0.99	0.93
200	1.00	1.09	1.06	1.10	1.12	1.13	1.11
300	0.93	1.00	1.06	1.10	1.15	1.16	1.13
400	1.64	1.98	2.45	2.68	1.91	3.04	3.02

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 9), working all the processes in the two parallelized matrix-vector multiplications, and performing the master process non parallelized operations. In that way, 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.

2.3.3 Experimental results

Table 13 shows the execution time of 50 iterations of the Davidson method for symmetric complex matrices on PC486 and using MPI, varying the number of processors and the matrix size. The values have been taken from [16].

Table 14 shows the speed-up of the matrix-vector multiplication and the Davidson method. We can see the Davidson method produces an important reduction in the execution time with big matrices, but the speed-up achieved with the Davidson method is smaller than that obtained in the matrix-vector multiplications, which are the only operations parallelized in the parallel Davidson algorithm.

```
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
```

Figure 9: Scheme of the parallel Davidson method.

Table 15: Execution time of the Givens method using PVM, varying the number of processes and the matrix size, on SUNUltra.

	seq	p=1	p=2	p=3	p=4	p=5	p=6	p=10
all the eigenvalues								
100	0.044	0.045	0.027	0.020	**0.017**	0.026	0.033	
300	0.346	0.356	0.184	0.133	**0.101**	0.121	0.141	
600	1.279	1.307	0.672	0.492	0.353	**0.311**	0.476	
900	2.735	2.803	1.452	1.007	0.784	0.642	0.975	**0.624**
1200	4.907	4.980	2.591	1.805	1.398	**1.069**	1.833	1.125
20% of the eigenvalues								
100	0.008	0.010	0.008	**0.007**	0.008	0.011	0.013	
300	0.065	0.067	0.039	0.028	**0.024**	0.035	0.040	
600	0.238	0.240	0.126	0.089	**0.071**	0.098	0.128	
900	0.505	0.508	0.265	0.182	**0.144**	0.151	0.179	0.204
1200	0.916	0.919	0.479	0.328	**0.265**	0.271	0.326	0.274

2.4 Givens algorithm or bisection

2.4.1 The algorithm

As we have seen in previous sections, 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. Some technique must be used to avoid overflow problems [29], but here we are interested in the parallelism of the method and not in precision problems.

The method is especially suitable to be parallelized, due to the slight communication between processes. 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 [29].

2.4.2 Experimental results

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 15 shows the execution time and the speed-up when the method is used to calculate all the eigenvalues or only 20% of them, on SUNUltra, and table 16 shows the speed-up. The speed-up is clearly better than in the previous algorithms, even with small matrices and execution time.

Comparing the execution time of the sequential method and the parallel method with only one slave, we can see the low influence of communications in this algorithm. The reduction of

Table 16: Speed-up of the Givens method using PVM, varying the number of processors and the matrix size, on SUNUltra.

	$p=1$	$p=2$	$p=3$	$p=4$	$p=5$	$p=6$	$p=10$
all the eigenvalues							
100	0.98	1.63	2.20	2.59	1.69	1.33	
300	0.97	1.88	2.60	3.43	2.86	2.45	
600	0.98	1.90	2.60	3.62	4.11	2.69	
900	0.98	1.88	2.72	3.49	4.26	2.81	4.38
1200	0.99	1.89	2.72	3.51	4.59	2.68	4.36
20% *of the eigenvalues*							
100	0.80	1.00	1.14	1.00	0.73	0.62	
300	0.97	1.67	2.32	2.71	1.86	1.63	
600	0.99	1.89	2.67	3.35	2.43	1.86	
900	0.99	1.91	2.77	3.51	3.34	2.82	2.48
1200	1.00	1.91	2.79	3.46	3.38	2.81	3.33

efficiency when the number of slaves increases is caused more by the inbalance in the distribution of the spectrum than by the communications.

The best execution time is obtained with a high number of processes, which is equal or close to the number of processors. The master and one slave process can be assigned to the same processor without great penalization because the master only works in the distribution and accumulation of data and results. In some cases the use of more processes than processors give the best execution times, and this can be used to avoid inbalance when the spectrum is not uniformly distributed.

2.5 Jacobi method

2.5.1 A block two-sided method

The Jacobi method for solving the Symmetric Eigenvalue Problem ([30]) 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. The sequential method has a cost of $4n^3$ flops per sweep when computing only eigenvalues, and $8n^3$ flops when computing eigenvalues and eigenvectors.

2.5.2 A parallel block two-sided method

Parallel block Jacobi methods are explained in more detail in [31, 32].

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 [5]. Because the most costly part of the execution is the updating of

Table 17: 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

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.

The arithmetic cost of the parallel algorithm is $4\frac{n^3}{p}$ flops per sweep when computing eigenvalues, and the theoretical efficiency is of 100%, but the cost of communications produces a reduction in the experimental efficiency, and this reduction is specially important in a network. In a multicomputer data can be transferred in some parts of the algorithms overlapping communications, and the cost of communications per sweep can be approximated by $\left(2+\frac{3\sqrt{2}}{\sqrt{p}}\right)n^2$ (this is the term corresponding to the work sending time), but in a network communication of data cannot be overlapped, and the work sending time is approximatelly $\left(2n^2t^2+n^2t+\frac{nt^2}{2}\right)\frac{\sqrt{1+8p}-3}{2}$.

2.5.3 Experimental results

Despite the high cost of communications in a network, the behaviour of the method is acceptable. Table 17 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. Normally, higher theoretical speed-up is obtained by increasing the number of processes. This produces better balancing, but not always a reduction of the execution time is achieved due to more communications being necessary. It can be seen in table 18, 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. This good behaviour can be observed in table 19, where the efficiency obtained in PC486 is shown.

2.5.4 Comparison with ScaLAPACK

Jacobi methods are more suitable for parallel processing than methods based in reduction to condensed form, and this is way Jacobi methods can be competitive with ScaLAPACK in some cases. In [5] the routine `pdsyevx` of ScaLAPACK and the Jacobi method here presented are compared in a multicomputer (an Intel Paragon with 512 processors) and a network (PC486). In the Paragon, lower execution time is obtained with the Jacobi method in some cases (with matrices of size under a 1000). In PC486 the efficiency of the Jacobi method is much bigger than that of `pdsyevx`, but Jacobi method is slower than `pdsyevx` due to the high cost of communications of the Jacobi method in a network. That can be observed in tables 20 and 21. Table 20 shows the efficiency of `pdsyevx` and Jacobi in PC486, with different matrix size and number of processors and using MPI. The efficiency of Jacobi is clearly better, but table 21 shows the lower execution time obtained with `pdsyevx` and Jacobi in the network and the number of processors from which these times have been obtained. More processors can be used efficiently with

Table 18: 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**

Jacobi, but this number of processors is not enough to obtain an execution time lower than the execution time obtained with `pdsyevx`.

2.5.5 Block one-sided Jacobi methods

Due to the high cost of communications of two-sided Jacobi methods, it can be better to use one-sided methods when working on a network. In [33] two one-sided block Jacobi methods are designed, and in [34] they are compared with a two-sided method. One of the one-sided methods is equivalent to the two-sided method and the other one-sided method has lower execution time but also more precision problems. In tables 22, 23 and 24 a two-sided block method for a ring and the equivalent one-sided method are compared in SUNUltra (table 22), PC486 (table 23) and Pentium (table 24). They are compared when computing eigenvalues and eigenvectors because

Table 19: Efficiency of the Jacobi method using MPI in PC486, varying the number of processors and the matrix size.

	$p=3$	$p=5$	$p=7$	$p=9$
128	0.51	0.08	0.04	0.16
256	0.55	0.25	0.18	0.24
384	0.52	0.30	0.25	0.29
512	0.55	0.35	0.33	0.34
614	0.70	0.49	0.33	0.32

Table 20: Efficiency of `pdsyevx` and the Jacobi method in PC486 using MPI, varying the number of processors and the matrix size.

	$p=5$		$p=7$		$p=9$	
	pdsyevx	*Jacobi*	*pdsyevx*	*Jacobi*	*pdsyevx*	*Jacobi*
128	0.05	0.05	0.03	0.03	0.01	0.11
256	0.10	0.25	0.05	0.18	0.03	0.25
384	0.16	0.43	0.09	0.28	0.07	0.31
512	0.22	0.35	0.13	0.33	0.10	0.34
614	0.29	0.49	0.18	0.33	0.16	0.32

Table 21: Execution time of `pdsyevx` and the Jacobi method in PC486 using MPI.

	pdsyevx		*Jacobi*	
	time	*processors*	*time*	*processors*
128	2.55	1	14.91	3
256	10.91	1	69.37	9
384	27.89	2	161.31	9
512	53.44	3	315.21	9
614	86.36	3	449.70	13

one-sided methods compute both. The methods are compared varying the matrix size and the number of processors. In tables the lowest execution time is marked for each matrix and system size. When communications are very costly in comparison with the arithmetic cost (table 1) the one-sided method is clearly better than the two-sided method, as happens in SUNUltra. But when the cost of communications is not so important the one-sided method is better than the two-sided method only when the number of processors increases. The arithmetic cost of the two-sided method is $8\frac{n^3}{p}$ flops per sweep when computing eigenvalues and eigenvectors, and that of the one-sided method is $9\frac{n^3}{p}$ flops per sweep, but when the number of processors increases lower execution time is obtained with the one-sided method because in this method the broadcast of rotation matrices is avoided.

In table 25 the scalability of the two methods is compared. The table shows the execution time when the matrix size varies proportionally to the number of processors. For a small number of processors the two-sided method is quicker than the one-sided method due to the lower theoretical execution time, but when the number of processors increases the one-sided method becomes better, due to its reduced communication cost.

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. Small modifications must be made in the algorithms to adapt them to the characteristics of the system.

In these environments we do not generally have many processors, and when designing algorithms for networks of processors it is preferable to think of good algorithms for a small number of processors, and not of scalable algorithms.

Table 22: Comparison of one-sided and two-sided method on SUNUltra using MPI. Execution time per sweep.

	p=2		p=3		p=4		p=5	
	two	one	two	one	two	one	two	one
128	0.80	**0.26**						
192			1.89	**0.62**				
256	3.34	**1.58**			4.56	**1.37**		
320							9.25	**3.15**
384	12.03	**4.63**	**11.47**	16.31				
512	32.08	**10.62**			36.10	**7.11**		
576			31.80	**13.72**				
640	60.	**22.70**					54.5	**11.2**
768	80.	**35.97**	123.38	**33.82**	96.55	**21.48**		
896	126.	**53.29**						
960			125.04	**61.62**			155.09	**39.64**
1024	250.	**89.**			165.62	**50.02**		
1152			220.07	**105.61**				

Table 23: Comparison of one-sided and two-sided method on PC486 using MPI. Execution time per sweep.

	p=2		p=3		p=4		p=5		p=6		p=7	
	two	one	two	one	two	one	two	one	two	one	two	one
128	**2.89**	2.95										
192			**6.97**	7.33								
256	**21.21**	22.21			**13.15**	13.32						
320							21.71	**20.77**				
384	69.87	**69.63**	49.01	**48.61**					32.51	**30.55**		
448											47.38	**43.39**
512	175.43	**162.43**			97.15	**96.39**						
576			165.68	**163.31**								

Also processors can have different computational ability, and even in an environment where the computational ability of processors is equal they can not be considered as equal because other factors can modify this ability (for example, the behaviour of the processor managing the file system is different). For this reason, studying the distribution of the work to obtain a good balance is very important, and in order to design a linear algebra library for Local Area Networks (LANLAPACK) it could be necessary to design installation routines to analyse the characteristics of the system (the different computational power of the processors, the cost of communications, ...) and to design the routines in the library in such a way that the distribution of data depend on some parameters obtained by the instalation routines. In that way, a static library could be designed. It could be preferable for some problems of high cost to design dynamic libraries in which the parameters of the system are tested periodically and the data are redistributed.

Table 24: Comparison of one-sided and two-sided method on Pentium using MPI. Execution time per sweep.

	p=2		p=3		p=4		p=5		p=6	
	two	one	two	one	two	one	two	one	two	one
128	0.66	**0.64**								
192			**1.45**	1.49						
256	**4.46**	4.86			2.81	**2.75**				
320							4.59	**3.92**		
384	**16.41**	18.73	**11.17**	13.55					6.97	**5.79**
512	**38.69**	40.07			**23.82**	27.51				
576			**41.65**	43.74						
640	**82.31**	88.72					38.51	**38.02**		

Table 25: Execution time of two-sided and one-sided block Jacobi methods in Pentium using MPI, and with matrix size scaled with the size of the system ($n = 64p$).

$Time(seconds)$	$p=2$	$p=3$	$p=4$	$p=5$	$p=6$
$two-sided$	0.63	1.45	2.81	4.59	6.97
$one-sided$	0.64	1.49	2.75	3.92	5.80

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] ScaLAPACK User's Guide. 1996.

[3] 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: A Linear Algebra Library for Message-Passing Computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, CD-ROM*. SIAM, 1997.

[4] 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.

[5] 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.

[6] 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.

[7] Gen-Ching Lo and Yousef Saad. Iterative solution of general sparse linear systems on clusters of workstations. May 1996.

[8] 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.

[9] Message Passing Interface Forum. A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, (3), 1994.

[10] Antonio Flores and José M. García. Improving the Performance of Scientific Parallel Applications in a Cluster of Workstations. In *Proceedings of PARA'98*. Springer-Verlag. Lecture Notes in Computer Science, 1998.

[11] Users guide to mpich. Preprint.

[12] Francisco Javier García Ros. Resolución de sistemas triangulares de ecuaciones lineales en redes de ordenadores. Proyecto Fin de Carrera. Facultad de Informática. Universidad de Murcia. Director Domingo Giménez. 1997.

[13] Agustín Caballero. Paralelización de métodos tipo Gradiente Conjugado para resolución de problemas de álgebra lineal. Proyecto Fin de Carrera (en preparación). Facultad de Informática. Universidad de Murcia. Director Domingo Giménez. 1998.

[14] Ginés Carrillo Cava. Resolución de sistemas de ecuaciones lineales en redes de ordenadores. Proyecto Fin de Carrera (en preparación). Facultad de Informática. Universidad de Murcia. Director Domingo Giménez. 1998.

[15] Celso Jiménez González. Algoritmos para el cálculo de algunos valores propios en multicomputadores. Proyecto Fin de Carrera. Facultad de Informática. Universidad de Murcia. Director Domingo Giménez. 1996.

[16] Adolfo Martín González. Métodos de Davidson para el cálculo de valores propios. Proyecto Fin de Carrera. Facultad de Informática. Universidad de Murcia. Directores Domingo Giménez y Norberto Marín. 1997.

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

[18] David Montejo Rodríguez. Estudio de las Prestaciones de Librerías de Paso de Mensajes y de Computación Matricial. Proyecto Fin de Carrera. Facultad de Informática. Universidad de Murcia. Directores Domingo Giménez y Isidro Verdú. 1998.

[19] Carlos Conde Lázaro and Gabriel Winter Althaus. *Métodos y Algoritmos Básicos del Álgebra Numérica*. Editorial Reverté, 1990.

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

[21] L. N. Trefethen and D. Bau III. *Numerical Linear Algebra*. SIAM, 1997.

[22] David S. Watkins. *Matrix Computations.* John Wiley & Sons, 1991.

[23] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, 1965.

[24] V. Kumar, A. Grama, A. Gupta and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* The Benjamin Cummings Publishing Company, 1994.

[25] R. B. Lehoucq, D. C. Sorensen and C. Yang. *ARPACK Users' Guide. Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* SIAM, 1998.

[26] 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.

[27] W. W. Bradbury and R. Fletcher. New Iterative Methods for Solution of the Eigenproblem. *Numerische Mathematik*, 9:259–267, 1966.

[28] A. Edelman and S. T. Smith. On conjugate gradient-like methods for eigenvalue-like problems. *BIT*, 36(3):494–508, 1996.

[29] 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.

[30] C. G. J. Jacobi. Uber ein leichtes Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen. *Journal für die Reine and Anbewante Mathematic*, 30:51–94, 1846.

[31] 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, 1998.

[32] 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'98-3rd International Meeting on Vector and Parallel Processing, Porto, June 21-23*, pages 139–152, 1998.

[33] D. Giménez, J. Cuenca, R. M. Ralha and A. J. Viamonte. One-sided Block Jacobi Methods for the Symmetric Eigenvalue Problem. In *Proceedings of VECPAR'98-3rd International Meeting on Vector and Parallel Processing, Porto, June 21-23*, pages 687–692, 1998.

[34] Javier Cuenca and Domigo Giménez. Implementation of parallel one-sided block Jacobi methods for the symmetric eigenvalues problem. Unpublished.