Inverse Toeplitz Eigenproblem on Personal Computer Networks *

J. M. Badía[1] A. M. Vidal[2]

June 27, 2000

[1] Dpto. Informática. Univ Jaume I.
12071, Castellón, España.
badia@inf.uji.es
[2] Dpto. Sistemas Informáticos y Computación.
Univ. Politécnica de Valencia.
46071, Valencia, España.
avidal@dsic.upv.es

Abstract

In this paper we present a parallel algorithm for solving the inverse Toeplitz Eigenvalue Problem. The algorithm has been implemented by using a cluster of personal computers, interconnected by a high performance Myrinet network. We have utilized standard public domain parallel environments for implementing the calculation part as well as the communications, thus producing portable software. The results obtained allow us to confirm the scalability and efficiency of the algorithm. Besides, we have checked that by using the theoretical cost model provided by the ScaLAPACK we can predict the behaviour of the experimental results.

1 Introduction

The rapid development of parallel computers has made it possible to tackle problems which cannot be dealt with by classic sequential computers owning the storage and time requirements. Distributed Memory Machines are probably among the most extended computers in the market. This is due basically to the scalability of this kind of machines which allows us to increase the performance with the number of processors, even up to hundreds or thousands of processors.

The main reasons that prevented a widespread use of the massive parallel computers were the difficulty of programming and their high

*This paper was partially supported by the project CICYT TIC96-1062-C03: "Parallel Algorithms for the computation of the eigenvalues of sparse and structured matrices"

economic cost. However, the recent development of message passing environments such as the PVM [11] or the MPI [19], facilitates the implementation of efficient, portable and scalable algorithms on this kind of parallel architecture.

Nevertheless, the use of custom processors and, specially, the use of very fast interconnection networks, enormously increases the cost of this kind of computer, if we want to have a large number of processors (> 100). This problem is being alleviated by two phenomena: first, the use of standard processors in the MMP, which allows the price of these components to be decreased, and the platform to be easily updated as the technology evolves. Second, and in a more radical way, the appearance on the market of the high performance networks connecting personal computers, is enlarging the scope of the multiprocessors.

Nowadays, a big effort is being made in the development of high performance interconnection networks, which allow us to group several personal computers or workstations to form a multiprocessor architecture. In this sense, it is worth noticing the development of Fast Ethernet network (100 Mbits/s.) and, specially, the presence of networks with a bandwidth of 1 Gbit/s., such as the Myrinet networks [4], [18]. Just to cite an example, currently the most powerful computer, in peak performance, is a parallel computer of this kind, formed by thousands of Pentium processors [8].

Quite recently, multiprocessor architectures were restricted to a few universities, research centres and big companies. However, with the personal computer networks, multiprocessors can extend their application field enormously, reaching even small and medium size companies, and spreading their use to a larger number of different users.

One of the fields where the use of multiprocessors is specially adequate is the Numerical Linear Algebra. One of the problems in this field that is most complex and costly in terms of computational time, is represented by the inverse problems, because their solution involves the solution of several, sometimes many, direct problems. In this paper we focus on the inverse eigenvalue problem. This problem arises in a remarkable variety of applications, such as control design, seismic tomography, antenna array processing, system identifications, structural analysis, circuit theory, particle physics and so on.

In [6] a wide summary and a classification of a collection of inverse eigenvalue problems, and the most recently theoretical and algorithmic results related to these problems are presented. One of the types of the inverse problems identified in that paper is the structured eigenvalue problem, that is, the reconstruction of a matrix with a predetermined spectrum and with a definite structure, for example Toeplitz structure (all the elements in a diagonal have the same value). In this paper we present a parallel algorithm which solves the inverse eigenvalue problem with Toeplitz matrices, on a high performance network of personal computers.

On the other hand, several parallel libraries for numerical linear algebra have been recently developed on distributed memory environments. These libraries contain very efficient numerical methods to solve a large number of numerical and matrix problems. ScaLAPACK [3] and PLAPACK [21] are examples of this kind of library.

This paper focuses on three objectives. First, to analyze the possibility of using personal computers clusters with a high performance interconnection network to solve a problem with high computational cost, in an efficient and scalable way, by using parallel computing techniques. The idea is to take advantage of the excellent ratio price/performance of this kind of platforms to extend the field of high performance parallel computing.

Second, to implement a portable algorithm, based on the use of real standard software to perform the basic calculation operation (BLAS, PBLAS, LAPACK, ...) and to carry out the necessary communications (BLACS, MPI, ...). Moreover, we try to make a portable implementation by using a public domain operating system like LINUX, and standard C or FORTRAN compilers. The use of this kind of environments and programming tools allows us to obtain portable algorithms. Besides, the algorithms can be easily adapted to the new versions of different applications, executable on very spread platforms and with a performance that can be increased with new versions of the software or hardware.

The third objective of this paper is to study the validity of a theoretical cost and communication model, used by ScaLAPACK, to analyze the cost of an algorithm which combines different routines of this library and other routines that parallelize intermediate operations, and which also perform several redistributions of data.

The rest of the paper is structured as follows: In section 2, the environment utilized for implementing the algorithm and the communication and computation cost model is presented. Section 3 contains a brief description of the problem to solve, the sequential algorithm utilized and its theoretical cost. In section 4, an outline of the parallel algorithm and its theoretical cost is presented. Experimental results and a thorough study of the scalability of the parallel algorithm in the environment used is presented in section 5. Finally, section 6 contains our conclusions.

2 Description and features of the environment

The target platform for our experimental study is a personal computer cluster connected through a Myrinet network [4]. More specifically speaking, the cluster consists of 32 PCs, based on 300MHz Pentium II processors, with 128 Mbytes of SDRAM each. The interconnection network consists of two switches of SAN type, from Myricom, model M2M-OCT-SW8, with 16 ports each.

A Myricom network card has been incorporated in each PC in order to connect it with one of the switches by means of a bidirectional link, with a bandwidth of 1,28 Gbits/s. Thus, a bisection bandwidth of 20.48 Gbits/s. on each switch of 16 ports is achieved. The connection between both switches allows this added bandwidth to be scaled.

Each switch is a crossbar, and both together allow the definition of any kind of topology by means of the manual or automatic stating of a set of paths among the different ports. Communications are carried out by using a cut-through protocol with low latency and flow control.

There exist specific implementations for Myrinet networks of some message passing environments such as MPI [17] which offer small latencies and high bandwidths. Below, we analyze the performance of the cluster and the interconnection network.

2.1 Communication cost

To analyze the communication cost we have adopted the same scheme used in [7]. We have used a well-known model to represent the cost, t_c, of performing a communication of m bytes through a link:

$$t_c = t_m + mt_v \tag{1}$$

Here, t_m stands for the startup time of the transference or latency time, and t_v represents the time of sending a byte through a link. Thus, the bandwidth of a link is given by $1/t_v$. It is worth noticing that in the communication costs, not only factors related to the speed of the physical links but also the environment utilized to implement the message-passing must be taken into account. In this case, the results shown have been obtained by using the MPI environment, GM version, which has been developed by the manufacturer of the interconnection network.

To determine the values of the constants in (1) we have utilized the double ping-pong algorithm, that is, a processor sends a set of packets of different sizes to another processor and the last one returns them. The time measured is the half of that required to send and return each packet. By sending packets of minimum size it is possible to obtain the value of t_m, while the value of t_v can be obtained by sending large size packets.

For the latency time, t_m, we have obtained a value of 33 μs. However, the bandwidth depends on the size of the messages sent. Sending a message of a few hundred bytes provides a bandwidth of 15 Mbytes/s. When the messages are of a few Kbytes the bandwidth reaches 23 Mbytes/s. Finally, for messages of tens of Kbytes, the bandwidth tends asymptotically to a maximum of 33 Mbytes/s. Thus, the maximum speed of transference through a link, obtained in this environment, gives a value of $t_v = 0,03$ μs/byte. This can be seen in Figure 1.

2.2 Arithmetic cost

To analyze the arithmetic performance of the processors it is important to distinguish between the peak performance and the real performance that we can obtain during the execution of a definite algorithm. The average execution time of a floating point operation (flop) in a concrete

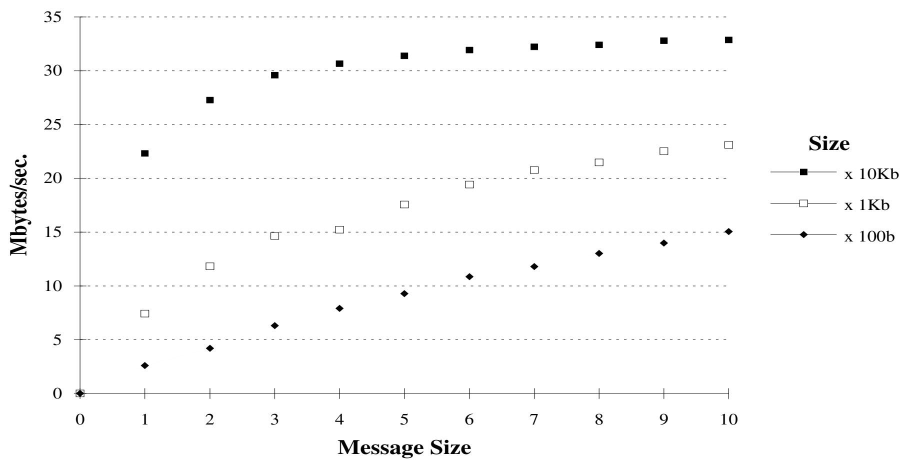

Figure 1: Bandwidth of the Myrinet network with messages of different sizes using MPI-GM.

	DGEMV		DGEMM		DGETRF	
n	MFlops	t_f (μs.)	MFlops	t_f (μs.)	MFlops	t_f (μs.)
200	84,42	1,18E-02	177,78	5,63E-03	132,83	7,53E-03
400	48,36	2,07E-02	180,28	5,55E-03	146,85	6,81E-03
600	47,93	2,09E-02	183,05	5,46E-03	154,65	6,47E-03
800	47,57	2,10E-02	184,84	5,41E-03	165,54	6,04E-03
1000	47,34	2,11E-02	185,53	5,39E-03	169,94	5,88E-03

Table 1: Arithmetic performance of the processors.

algorithm depends on the type of operation, on the memory access outline, and on the exploitation of all the features of the processor.

To analyze the arithmetic performance of the processor three widespread standard algorithms have been used: the first one, `DGEMV`, performs the matrix-vector product and is integrated in the level 2 of the computational kernel BLAS. The second one, `DGEMM`, performs matrix products and belongs to BLAS level 3, and the third one, `DGETRF`, performs the LU decomposition of a matrix and is integrated in the LAPACK library [1]. In the case of the routines included in the BLAS kernel, we have used a version specially devised to take advantage of Pentium processors, which is incorporated in the ASCI Red Pentium Pro BLAS 1.1.N [12], [13].

In table 1 we present the results obtained by the previous routines in a Pentium II-300 processor, included in the cluster utilized to perform the experiments with the parallel algorithm. We can verify that the performance obtained by the two last routines approaches the 200 MFlops, while the first one achieves a clearly inferior performance. This is undoubtedly due to the ratio between the number of operations and the number of memory accesses in each subroutine. While in the first one we use the level 2 of BLAS, in the second and third ones we refer to BLAS 3.

The different performances of the previous subroutines result in very different values for the average time of execution of a floating point operation. While in the case of the two last subroutines this value is approximately 0,006 μs, in the first one the value is around 0,02 μs. Thus, the value of constant t_f, which is more appropriate to model our system, strongly depends on the characteristics of the arithmetic operations to be performed and on the exploitation of the different levels of BLAS.

3 Description of the problem and its sequential solution

In this section we briefly describe the inverse eigenproblem to be solved, the sequential algorithm and its theoretical cost. This problem has been previously studied in [15] and [10]. To obtain a more detailed information of the algorithm [20] and [2] can also be consulted. Let $t = [t_0, t_1, \ldots, t_{n-1}]$ be a real n-vector. We say that $T(t)$ is a Real Symmetric Toeplitz Matrix (TRS) generated by t if

$$T(t) = \left(t_{|i-j|}\right)_{i,j=1}^{n} .$$

This kind of matrix appears in the solution of many problems in Physics or Engineering.

Given n real values such as

$$\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n, \tag{2}$$

the inverse eigenvalue problem consists in computing a generator t, so that the spectrum of the TRS matrix associated coincides with (2).

The TRS matrices verify some properties [5], [15] that allow their spectrum to be divided in two parts with the same number of eigenvalues, known as even and odd eigenvalues, and associating them with the symmetric and skew-symmetric eigenvectors, respectively. On the other side, it is possible to compute both spectra separately, substantially reducing the cost of calculating the eigenvalues and eigenvectors of the matrix.

In [16] a method for solving the inverse eigenproblem with TRS matrices is proposed. This method is equivalent to the Newton method. This algorithm is improved in [20] by means of the adequate exploitation of the previous spectral properties.

3.1 Sequential Algorithm

In this section, we briefly describe the sequential method proposed in [20]. We will call $p_1(t), \ldots, p_r(t)$ the symmetric eigenvectors of $T(t)$

and $q_1(t), \ldots, q_s(t)$ its skew-symmetric eigenvectors. On the other side, we will denote the target spectrum as

$$\Lambda = [\mu_1, \mu_2, \ldots, \mu_r, \nu_1, \nu_2, \ldots, \nu_s] \tag{3}$$

where the even and odd spectra have been separated, and each one has been written in increasing order.

Let t_0 be a n-vector and let Λ be the target spectrum, as defined in (3). By using t_0 as an initial generator, the method computes a sequence t^m, $m = 1, 2, \ldots$ as the solution of the equations:

$$\begin{array}{ll} p_i(t^{m-1})^T T(t^m) p_i(t^{m-1}) = \mu_i, & 1 \leq i \leq r \\ q_j(t^{m-1})^T T(t^m) q_j(t^{m-1}) = \nu_j, & 1 \leq j \leq s \end{array} \tag{4}$$

where $r = \lceil n/2 \rceil$ and $s = \lfloor n/2 \rfloor$. The previous equations can be written as a linear system of dimension $n = r + s$; and we can obtain t^m from t^{m-1}, thus producing an iterative method.

In each iteration of the algorithm, we start by constructing the matrix associated with the linear system in order to solve (4). This is performed from the eigenvectors of the TRS matrix of the previous iteration, which have been computed with a small cost by separating the odd and even spectra. Then, the linear system is solved, thus obtaining a new generator for a TRS matrix, whose spectrum is calculated. The convergence of the algorithm is reached when the difference between the computed spectrum and the target spectrum is smaller than a given error ϵ_0.

Broadly speaking, the sequential algorithm we have implemented is the following:

```
REPEAT
    Λ(T(t^{m-1})) ← Compute the spectrum (eigenvalues and eigenvectors)
                    of T(t^{m-1})
    C ← Build the linear system from (4) and the computed eigenvectors.
    t^m ← Solve the linear system C t^m = Λ
UNTIL |Λ(T(t^m)) − Λ| < ε_0
```

The method described is equivalent to Newton's method. Since the Newton method is not globally convergent, this algorithm does not necessarily converge to a solution of the problem. In [20] several improvements are proposed in order to achieve the convergence of the method. Specifically, if the convergence fails, we try to linearly converge to a modified spectrum with a less restrictive stopping criterion. When this new convergence is reached, the computed spectrum is used as a starting point to quadratically converge on the target spectrum.

3.2 Analysis of the theoretical cost

Each iteration of the previous sequential method performs three basic tasks: the computation of the spectrum of a TRS matrix, the construction of the coefficient matrix of a linear system and, finally, the solution of the linear system.

By exploiting the spectral properties of the TRS matrices, the cost of computing their spectrum is

$$t_1 = 11n^3/6 + n^2/4 \quad \textit{flops}.$$

The construction of the coefficient matrix implies a cost of

$$t_2 = n^3/2 + 3n^2 \quad \textit{flops}.$$

Finally, the solution of the linear system by means of gaussian elimination produces a cost of:

$$t_3 = 2n^3/3 \quad \textit{flops}.$$

Thus, if we call it the number of necessary iterations to reach the convergence, the total cost of the sequential algorithm is given by:

$$t_2 = (t_1 + t_2 + t_3) = (3n^3 + 13n^2/4) * it \quad \textit{flops}. \tag{5}$$

4 Outline of the parallel algorithm

The parallelization of the sequential method is based on the use of the ScaLAPACK parallel linear algebra library [3]. In this environment the algorithms use a SPMD model and a block cyclic data distribution among the processors of a logical bidimensional mesh.

The algorithm parallelizes the three main steps of the sequential version and performs the communications needed to redistribute the data appropriately in order to start each step. A very simplified version of the parallel algorithm is summarized in the following pseudocode:

```
WHILE not converged
    Compute the odd and even spectra of T(t^{m-1}) in parallel.
    Gather the eigenvectors in the first row of processors.
    IF the processor is in the first row of the mesh THEN
        Compute the corresponding rows of matrix C in the linear system.
    Redistribute matrix C among all the processors in the mesh.
    Solve the linear system in parallel.
```

In order to accomplish the parallel solution of the linear system and the computation of the spectra, we have used several ScaLAPACK routines, namely, `PDGETRF`, `PDGETRS` and `PDSYEV`. To compute the coefficient matrix for the linear system we have exploited the fact that each row depends on one eigenvector, and therefore, all matrix rows can be computed in parallel.

Due to the mesh topology of the environment and the data dependencies of the problem, we have to perform some redistributions of the data in each iteration of the algorithm. These communications greatly increase the cost of the parallel algorithm.

First, in each iteration, we must gather the eigenvectors in the first row of processors in order to compute the coefficient matrix of the linear system. On the other side, we compute the odd and even spectra separately, obtaining two matrices distributed in the whole mesh. To obtain a properly distributed matrix containing all the eigenvectors, once we have the eigenvectors in the first row of processors, we have to redistribute them. Finally, once we have built the coefficient matrix, we must redistribute its elements among all the processors of the mesh in order to solve the linear system. The computation and communication outline in each iteration of the parallel algorithm is shown in figure 2.

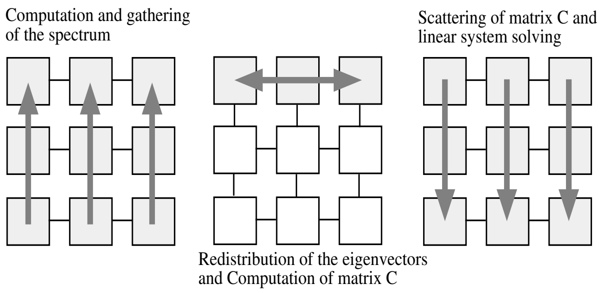

Figure 2: Computation and communication outline of the parallel algorithm.

To perform all the communications of the algorithm we have used the routines in BLACS [9] and the auxiliary redistribution routines included in the ScaLAPACK library.

4.1 Theoretical cost analysis

In this section we use the ScaLAPACK model in order to analyze the theoretical cost of the parallel algorithm. In this model, p processors are distributed in a square mesh and the matrices of size $n \times n$ are distributed by using a block cyclic scheme with block size $nb \times nb$. The cost of a driver routine in ScaLAPACK ([3], pag. 97) is given by:

$$T(n,p) = C_f \frac{n^3}{p} t_f + C_v \frac{n^2}{\sqrt{p}} t_v + C_m \frac{n}{nb} t_m, \tag{6}$$

where $C_f n^3/p$ represents the total number of floating point operations, $C_v n^2/\sqrt{p}$ represents the total number of bytes communicated through the algorithm, and $C_m n/nb$ represents the number of messages transferred.

We make some assumptions in order to simplify the cost analysis. First, we are going to represent the cost in the case of a square mesh, though, as we will see in the following sections, the configuration of the mesh greatly affects the experimental results. Second, we perform the redistribution of the matrices using messages of size $nb \times nb$, and we suppose that there is no overlapping among these messages.

In the previous conditions, we have computed the values of the constants in (6) which corresponds to the part of the parallel algorithm that is not computed using ScaLAPACK drivers. Therefore, the following constants include the computation of the coefficient matrix and the data redistributions in each iteration.

$$C_f \approx \frac{p}{2\sqrt{p}} \tag{7}$$

$$C_v \approx \frac{3}{2}\left(\sqrt{p}+1\right) \tag{8}$$

$$C_f \approx \frac{3}{2}\left(\sqrt{p}-1\right)\frac{n}{nb} \tag{9}$$

If we want to obtain the total cost of the parallel algorithm we have to add the values associated to the ScaLAPACK routines used in each iteration ([3], table 5.8) to the previous constants. We also have to recall that we compute the odd and even spectra separately, and so, we call the routine `PDSYEV` twice with two matrices of size $n/2$.

In figure 3 we show the performance of the parallel algorithm obtained by applying the previous theoretical model. Specifically, we have substituted in (6) the values of the parameter t_f, t_v and t_m which corresponds to our parallel architecture (see §3). If we analyze the computations developed in the parallel algorithm, we can see a combination of several BLAS levels. Therefore, we have used a value of $t_f = 0{,}015\mu$s., corresponding to an intermediate point among BLAS levels 2 and 3.

The results in figure 3 show that the speedups are quite far from the maximum. This behaviour is mainly due to the large communication cost obtained if we substitute (8) and (9) in (6). Taking into account our assumptions, both expressions represent maximum bounds for the communication cost.

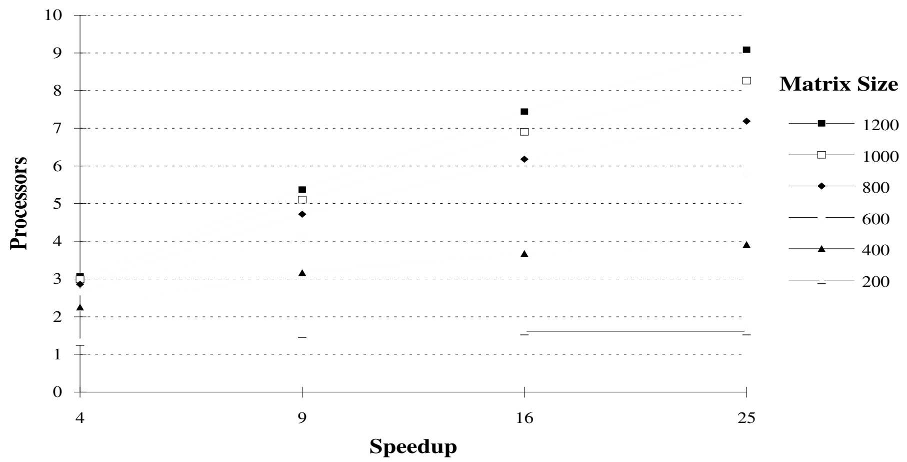

Figure 3: Performance using the theoretical model.

Figure 4 shows the influence of the three factors in (6) in the theoretical performance of the algorithms. Specifically, this figure represents their value using 25 processors for matrices of several sizes. We can see that the computation cost grows more quickly than the communications cost, but it is larger only with matrices of size 600 or more. Therefore, the overload due to the communications limits the speedup that we can obtain with the parallel algorithm except for very large matrices.

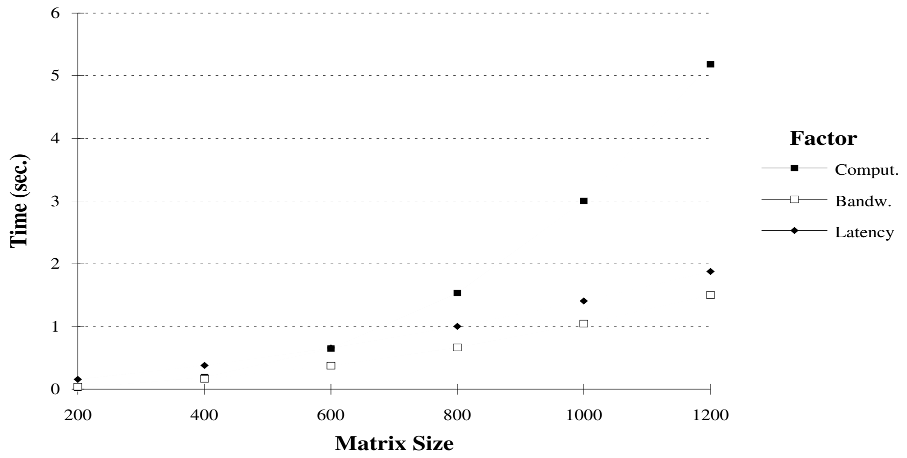

Figure 4: Effect of the communications and computations in the duration of the parallel algorithm. (p=25).

5 Experimental analysis

First, in table 2 we show the execution time of both, sequential and parallel algorithm, with matrices of several sizes and using different numbers of processors. This table shows the absolute performance of the algorithms in the experimental environment. We can see a clear reduction in the duration of the algorithm when we use the parallel algorithm. For example, with a matrix size of $n = 1200$ we reduce the duration from 2677.39 seconds in the sequential version, to 268.21 seconds in the parallel version using 25 processors.

In the following sections we study the effect of the configuration of the mesh in the performance of the parallel algorithm and we analyze its scalability using different metrics.

5.1 Effect of the configuration of the mesh

In this section we study the effect of the configuration of the mesh in the performance of the parallel algorithm. This factor has a large influence in our algorithm, because an important part of the communication cost depends on it.

We have shown in section 4 that in each iteration of the algorithm we must perform two redistributions of a matrix among all the proces-

	Matrix size					
proc.	200	400	600	800	1000	1200
1	5,99	60,96	282,30	782,94	1453,61	2677,39
4	6,84	37,85	104,83	210,28	274,19	731,62
9	8,23	36,48	88,72	154,58	199,81	404,47
16	8,45	30,67	30,67	128,54	155,76	288,94
25	9,57	36,51	72,87	112,39	180,56	268,21

Table 2: Duration of the sequential and parallel algorithms in seconds.

sors in the mesh. Specifically, we must gather the eigenvectors in the first row of processors and we must redistribute the coefficient matrix to all the processors. Both operations force an important communication cost whose real value depends on the configuration of the mesh. If we use an unidimensional mesh with only one row of processors, the cost of these communications is zero, while if we only use one column of processors, its cost is the maximum one. However, with large matrices we must also take into account that the behaviour of the ScaLAPACK routines is better with square meshes, and that this class of meshes improve the load balance of the whole algorithm.

In figure 5 we can see how if we use meshes with a large number of rows, the speedups decrease due to the larger cost of the redistributions, while the best performance with large matrices (n=1200) is obtained using square or almost square meshes.

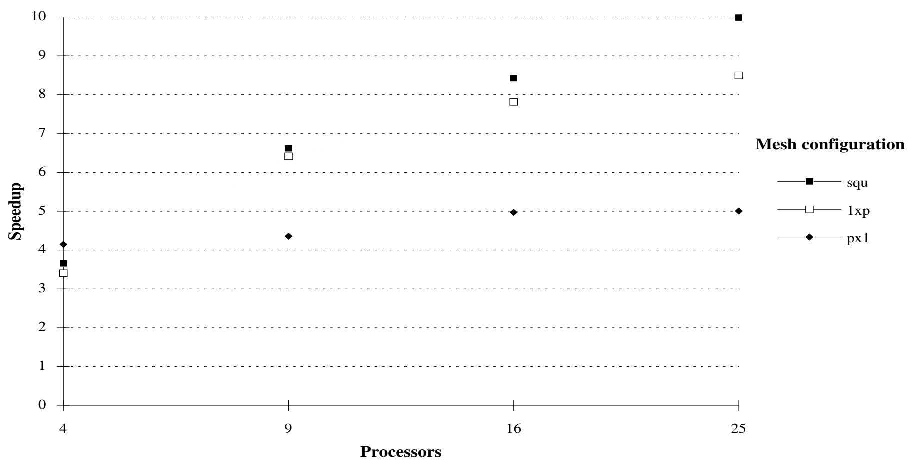

Figure 5: Effect of the configuration of the mesh in the speedups. ($n = 1200$).

5.2 Scalability analysis

In this section we analyze the scalability of the parallel algorithm. By scalability we mean the capacity of the algorithm to maintain the performance when we increase the number of processors. To achieve this behaviour we have to increase the size of the problem appropriately while increasing the number of processors.

We use two metrics to analyze the scalability. In the first metric, called *isotemporal*, we increase the size of the problem so that the duration of the parallel algorithm can be the same as the duration of the sequential algorithm. In the second metric, called *isospatial*, we keep the size of the problem constant in each processor, thus maintaining the memory usage per node.

5.3 Isotemporal scalability

We use the concept of scaled speedup defined in [14] to represent the isotemporal scalability:

$$Sp = \frac{pW}{T(p, pW)},$$

where W represents the cost of the sequential algorithm and $T(p, pW)$ represents the cost of the parallel algorithm to solve a problem of cost pW using p processors.

As we are dealing with an algorithm with sequential cost $W = O(n^3)$, we must increase the matrix size, n, with $\sqrt[3]{p}$, if we want to keep the same computation cost in the parallel algorithm.

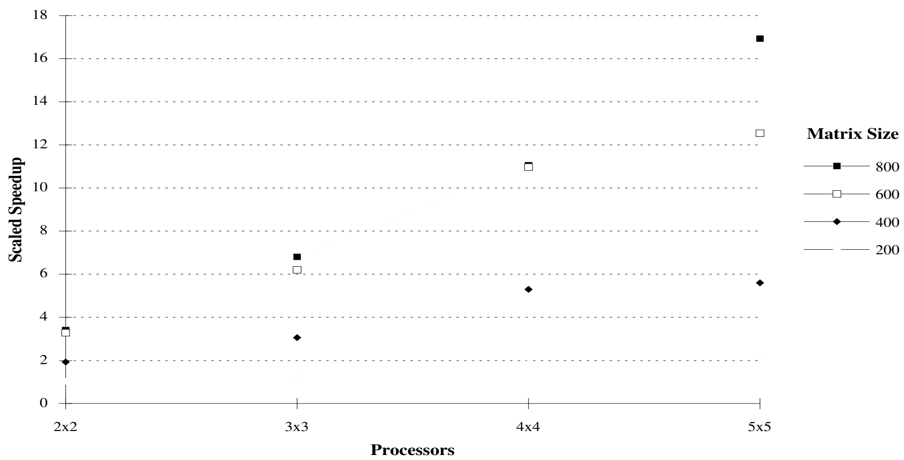

Figure 6: Scaled speedup using an isotemporal metric.

Figure 7 shows the speedups obtained without scaling, that is, if we maintain the size of the problem while increasing the number of processors. If we compare this figure with figure 6, we can see the effect of scaling in the performance of the algorithm.

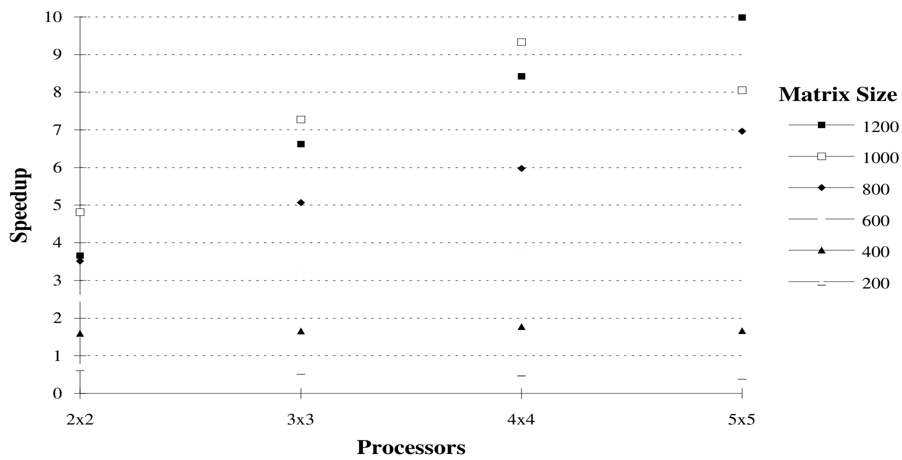

Figure 7: Speedup without scaling.

Figure 7 also allows us to confirm experimentally the results predicted by the theoretical analysis in section 4.1 and represented in figure 3. The patterns in both figures are similar, and the differences are due to several reasons. First, the theoretical computation cost is based on a value for the parameter t_f that does not have to coincide with the mean cost of a flop in our algorithm, and that is based on the performance of several BLAS and LAPACK routines. In general, the computation performance of our algorithm is quite far from the peak of the machine, and it is nearer to the values obtained with routine `DGEMV` than to the values obtained with routine `DGEMM`, (see table 1).

Second, the communication cost in the theoretical model is based on some simplifications that do not take into account the possible overlapping among different messages, the use of pipelined communications, the real topology of the interconnection network, the possible overlapping among computations and communications, etc. We must also take into account that the bandwidth, and thus the value of t_v, depends, as we can see in figure 1, on the size of the message. All these aspects of the implementation define the real cost of the communications and can justify the possible differences between the theoretical and experimental results.

5.4 Isospatial scalability

When we use the isospatial scalability we must maintain the size of the problem in each processor. In the case of the ScaLAPACK library, if we hold constant the ratio n^2/p while increasing the number of processors, the efficiency of the driver routines is almost maintained. When this condition holds, it is said that these routines scale isoefficiently.

In figure 8 we show the performance obtained when we use an isospatial scaling in our parallel algorithm. First, we can see a clear decrease of the MFlops/s. per node when we increase the number of processors. This behaviour is due to the fact that we are not using a ScaLAPACK routine, but a combination of some of them, with some routines implemented specifically for this algorithm. Besides we perform some redistributions of the data in each iteration that clearly affect the global scalability of the algorithm.

On the other hand, we must point out that the biggest reduction occurs when we go from the sequential version to the parallel version. If we increase the number of processors in the parallel version, the performance is almost maintained. This behaviour is due to the overload of the parallelization, and mainly, to the communication cost. Besides, when we begin with large matrices, the decrease of the performance is not so large and we go from 64 MFlop/s. per node in the sequential case to 33 MFlop/s. per node using 25 processors.

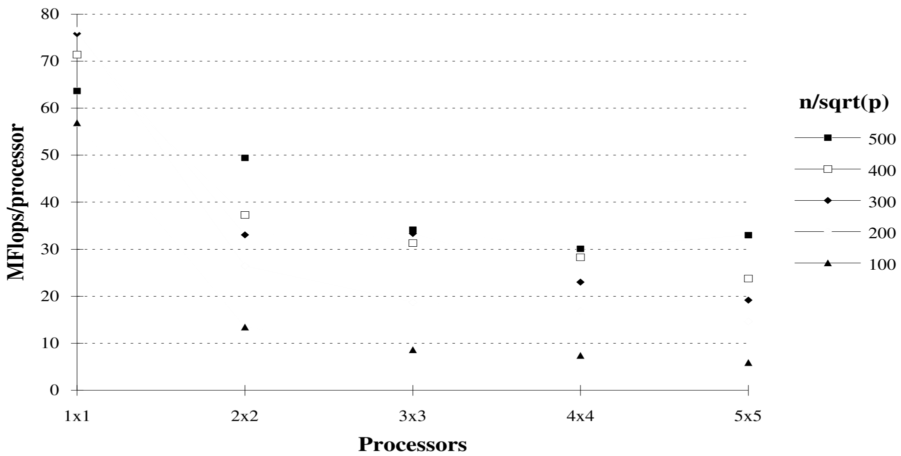

Figure 8: Megaflops per processor using an isospatial scaling.

6 Conclusions

In this paper we present a parallel algorithm that solves efficiently and in a quite scalable way the inverse eigenproblem for real symmetric Toeplitz matrices. We show the possibility of implementing this type of algorithm on an architecture with an excellent rate cost/performance. Specifically, we have used a cluster of personal computers connected with a high performance network.

We must always take into account that we are dealing with a complex problem that involves a large number of communications. This factor is crucial in the performance that we can obtain working with a distributed memory multicomputer. We have tested this effect both theoretically and experimentally.

To implement the algorithms we have used a standard environment based mainly on public domain and very well known tools (Linux, MPI, BLAS, LAPACK, ScaLAPACK, ...). Therefore, we have obtained a portable algorithm for a large range of parallel architectures. Moreover, the performance of the algorithm can improve with the quick evolution of the characteristics of personal computers and of high performance networks (Fast Ethernet, Gigabit, Myrinet, ...).

The utilization of the ScaLAPACK parallel linear algebra library imposes a program model based on a bidimensional mesh and a block cyclic distribution of the matrices. In the case of our algorithm, and due to its communication pattern, the configuration of the mesh greatly affects the performance.

At the same time, we have applied the theoretical cost analysis model of the ScaLAPACK to our algorithm. Even taking into account several important simplifications and the effect of the communications, the results offered by the model allow us to approach the general behaviour of the parallel algorithm and permits the analysis of the influence of the different factors involved, such as the computation cost, bandwidth and latency of the communications.

We must also point out that the implementation of this type of algorithms proves that it is possible to obtain good performances by applying parallel programming techniques and tools to architectures based on clusters of personal processors. It is not necessary to use big supercomputers, with very expensive hardware and specifically designed software to obtain good result in the solution of complex linear algebra problems.

Besides, we have obtained a considerable degree of scalability on a cluster of personal computers connected with an external network with excellent performance as the Myrinet. This result is very promising, as it proves the possibility of increasing the area of application of parallel algorithms to architectures based on standard components with low cost and using standard software tools.

References

[1] E. Anderson, Z. Bay, and C. Bischof. *LAPACK User's Guide*. SIAM, 1992.

[2] J.M. Badia and A.M. Vidal. Parallel solution of the inverse eigenproblem for real symmetric toeplitz matrices. Tech. Report DI 01-04/99, Dpt. Informatica, Univ. Jaume I, 1999.

[3] L.S. Blackford, J. Choi, and A. Cleary. *ScaLAPACK Users' Guide*. Software, Environment, Tools. SIAM, 1997.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[5] A. Cantoni and F. Butler. Eigenvalues and eigenvectors of symmetric centrosymmetric matrices. *Lin. Alg. Appl*, (13):275–288, 1976.

[6] Moody T. Chu. Inverse eigenvalue problems. *SIAM Review*, 40(1):1–39, March 1998.

[7] J. J. Dongarra and T. Dunigan. Message-passing performance of various computers. Technical Report UT-CS-95-299, Dpt. of Computer Science, Univ. of Tennessee, July 1995.

[8] Jack J. Dongarra, Hans W. Meuer, and Erich Strohmaier. TOP500 supercomputer sites. Technical Report UT-CS-98-391, Department of Computer Science, University of Tennessee, June 1998.

[9] Jack J. Dongarra and R. Clint Whaley. LAPACK working note 94: A user's guide to the BLACS v1.0. Technical Report UT-CS-95-281, Department of Computer Science, University of Tennessee, March 1995.

[10] Shamuel Friedland. Inverse eigenvalue problems for symmetric Toeplitz matrices. *SIAM Journal on Matrix Analysis and Applications*, 13(4):1142–1153, October 1992.

[11] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[12] G. Henry. ASCI red pentium pro BLAS 1.1N. Technical report, www.cs.utk.edu/ ghenry/distrib, 1999.

[13] B. Kagstrom, P. Ling, and C. van Loan. GEEM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. Technical Report UT-CS-95-315, Department of Computer Science, University of Tennessee, October 1995. Fri, 27 Aug 99 3:05:19 GMT.

[14] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* The Benjamin/Cumming Pub. Company, Redwood, California, 1994.

[15] H.J. Landau. The inverse eigenvalue problem for real symmetric toeplitz matrices. *J. Amer. Math Soc.*, (7):749–767, 1994.

[16] Dirk P. Laurie. A numerical approach to the inverse Toeplitz eigenproblem. *SIAM Journal on Scientific and Statistical Computing*, 9(2):401–405, March 1988.

[17] Myricom. The GM message-passing system. Technical report, Myricom Inc., 1998.

[18] VITA Standards Org. Myrinet-on-VME protocol specification. Draft Standard. Technical Report 26-199x Draft 1.1., VITA, 1998.

[19] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: the complete reference.* MIT Press, Cambridge, MA, USA, 1996.

[20] William F. Trench. Numerical solution of the inverse eigenvalue problem for real symmetric Toeplitz matrices. *SIAM Journal on Scientific Computing*, 18(6):1722–1736, November 1997.

[21] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* MIT Press, Cambridge, MA, USA, 1997. With contributions by Philip Alpatov and others.