Solving the Symmetric Eigenvalue Problem on Distributed Memory Systems*†

D. Giménez, M. J. Majado, I. Verdú
Departamento de Informática y Sistemas
Univ de Murcia
Aptdo 4021. 30001 Murcia. SPAIN
{domingo,mmajado,iverdu}@dif.um.es

Abstract ***In this paper we compare the performance obtained when solving the Symmetric Eigenvalue Problem with ScaLAPACK (routine*** `pxsyevx`***) and with a Jacobi method working by blocks on a logical triangular mesh of processes. The two methods are experimentally compared on a Paragon with 512 processors and on a network of processors using MPI. On the Paragon each process can be assigned to a processor, but in a network the number of processors is reduced and the processes must be assigned to the processors obtaining a balanced distribution. The conclusion is that the Jacobi methods are more suitable for parallelization and highly efficient Jacobi methods can be more adequate in solving the Symmetric Eigenvalue Problem on a system with a large number of processors.***

Keywords: symmetric eigenvalue problem, Jacobi methods, multicomputers, networks of processors

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

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

1 Introduction

ScaLAPACK [1] is a package for solving Dense Linear Algebra problems on Distributed Memory Multiprocessors. It is being developed and the intention is to include in ScaLAPACK all the routines in LAPACK. The Symmetric Eigenvalue Problem can be solved with the ScaLAPACK routine `pxsyevx`.

The Jacobi method for solving the Symmetric Eigenvalue Problem seems to be more scalable than the method used in ScaLAPACK, and it is promising for Multicomputers. Highly efficient and scalable Jacobi methods can be designed [2]. In this paper we compare the performance of a Jacobi method and the routine `pxsyevx` of ScaLAPACK.

2 Solving the Symmetric Eigenvalue Problem

2.1 The routine `pxsyevx`

The Symmetric Eigenvalue Problem can be solved by reducing the symmetric matrix A to tridiagonal form ($QAQ^T = T$), and then computing the eigenvalues (and optionally the eigenvectors) of the tridiagonal matrix T. In that way, the eigenvalues of matrix A are computed with a cost $\frac{4}{3}n^3+O\left(n^2\right)$. In ScaLAPACK the reduction is performed with routines of level 3 PBLAS, which exploit the parallelism. The execution time of the routine is ([3]):

$$(1) \qquad k_3 \frac{2}{3}\frac{n^3}{p} + k_2 \left(\frac{2}{3}\frac{n^3}{p} + 120\frac{n^2}{p}\right)$$
$$+60\frac{n^2}{p} + 21\alpha n \log p + 5\beta \frac{n^2}{\sqrt{p}} \log p$$

when computing eigenvalues.

In this formula, and in the following formulae in this section, k_3, k_2 and k_1 represent the cost of an operation performed using BLAS 3, 2 or 1, respectively, and α and β represent the start-up time and the word-sending time of a communication.

2.2 Jacobi methods

The Jacobi method has a cost of approximately $3n^3 \log n$ flops when computing eigenvalues. The computational cost of the method is higher than the cost of the routine `pxsyevx`, but Jacobi methods present better scalability properties and that could produce better performance on systems with a great number of processors (MPPs) or where the cost of communications is extremely large (networks of processors).

It is possible to design Jacobi methods based on BLAS 3 [4]. In this way, the cost of the method is approximately $4n^3 \log n$ flops when computing eigenvalues. But the use of BLAS 3 allows a reduction on the execution time with respect to the non blocked methods.

To explain the method we consider a matrix A of size $n \times n$ divided into blocks of size $t \times t$, and the algorithm will work considering each one of these blocks as elements, and working only with the lower triangular part of the matrix. In figure 1 the logical division in blocks of the matrix and the memory reserved to store the matrix on odd and even steps is represented. In this figure $\frac{n}{t} = 16$ and the rows and columns of blocks are numbered from 0 to 15, and each block of size $t \times t$ is represented by an `x`. These blocks are grouped in bigger blocks, grouping $2k$ consecutive rows and columns of blocks of size $t \times t$ (in the figure $k = 2$).

To obtain an algorithm on a logical triangular mesh it suffices to assign to each proces

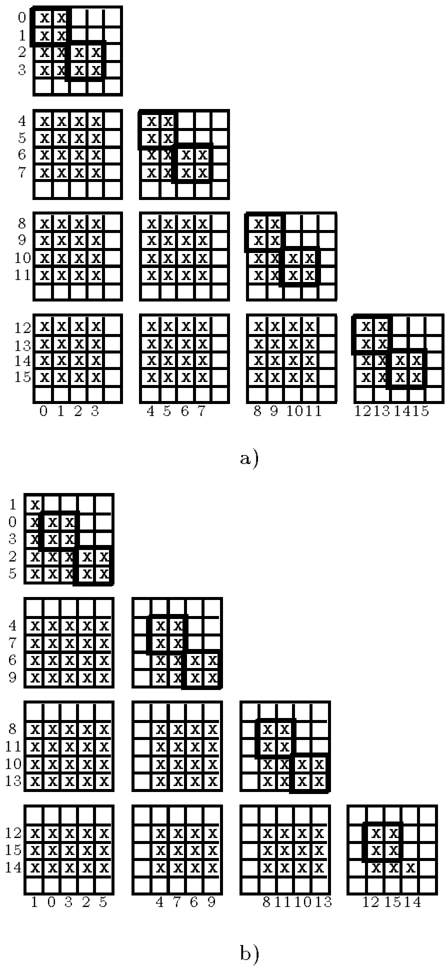

Figure 1: Logical distribution of the matrix on odd and even steps.

one of the blocks of size $2tk \times 2tk$. On a MPP we have a great number of processors and the best way is to assign each proces to a diferent processor, but on a network the number of processors is small (normally) and we have more processes than processors, and it is necessary to obtain a balanced assignation of the processes to the processors.

After updating the matrix, a movement of rows and columns of blocks is necessary in order to obtain the new grouping of blocks according to the odd-even ordering. This data movement implies a data transference between processors and generates the logical division and storage scheme of the matrix shown in figure 1 b).

A set of sweeps are performed over the blocks of size $2t \times 2t$ marked in the figure. But in the first step the sweeps were performed nullifying once each nondiagonal element, and in the successive steps the sweeps are performed nullifying only elements in the subdiagonal blocks of size $t \times t$, because the elements on blocks of size $t \times t$ in the main diagonal of blocks were nullified in the first step.

After the second step, a new movement of rows and columns of blocks is necessary to obtain the new grouping of data. In that way, the blocks are stored again as in figure 1 a). The work continues in the same way until $\frac{n}{t}$ steps are performed, and performing successive sweeps until convergence is reached.

When computing only eigenvalues the cost per sweep is approximately:

$$(2) \quad k_3\left(8\frac{n^3}{q^2}+2\frac{n^2t}{q}\right)+k_1 12\left(\frac{n^2t}{q}+\frac{nt^2}{q}\right) +\alpha\left(4+q\right)\frac{n}{t}+\beta\left(2+\frac{6}{q}\right)n^2$$

with $q=\frac{-1+\sqrt{1+8p}}{2}$. When the number of processors increases $q \simeq \sqrt{2}\sqrt{p}$, and (2) can be approximated by:

$$(3) \quad k_3\left(4\frac{n^3}{p}+\sqrt{2}\frac{n^2t}{\sqrt{p}}\right)+k_1 6\sqrt{2}\left(\frac{n^2t}{\sqrt{p}}+\frac{nt^2}{\sqrt{p}}\right) +\alpha\left(4+\sqrt{2}\sqrt{p}\right)\frac{n}{t}+\beta\left(2+\frac{3\sqrt{2}}{\sqrt{p}}\right)n^2$$

But on a network of processors with an ethernet connection the data are transferred through a common bus, and the cost of the transference of data is:

$$(4) \quad \beta\left(2n^2t^2+n^2t+\frac{nt^2}{2}\right)(q-1)$$

which produces a drastic reduction on the performance when the number of processors increases.

We can conclude that `pxsyevx` is faster than the Jacobi method. But the Jacobi algorithm has smaller communication times, and it is possible to overlap communications and computations in some parts of the execution.

Table 1: Comparison of the execution time of `pdsyevx` and the Jacobi method.

	Paragon			
	pdsyevx		Jacobi	
size	time	pr.	time	pr.
128	2.32	10	1.98	10
256	3.91	36	1.98	136
384	5.69	36	2.86	300
512	7.45	64	6.62	136
640	9.33	64	8.30	210
	Network			
	pdsyevx		Jacobi	
size	time	pr.	time	pr.
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
640	86.36	3	449.70	13

3 Experimental comparison

The experiments have been performed on a 512 node Paragon and on a network of PCs connected by ethernet and using MPI.

The results have been obtained with randomly generated matrices with double precision values between -10 and 10, and similar times are obtained in other cases.

The performance of the routine `pdsyevx` (double precision `pxsyevx`) depends on the block size, and a block size of 8 has been chosen to obtain these results. With this block size the execution time is close to the optimum [1]. Also the performance of the blocked Jacobi algorithm depends on the block size, and a block size of 16 has been used to obtain the results.

Table 1 shows the execution time of `pdsyevx` and the Jacobi method when computing eigenvalues. The table shows the lowest execution times (in seconds) obtained with each method, and the number of processors with which this execution time has been obtained.

On the Paragon we can see the Jacobi method is clearly better than `pdsyevx` for medium size matrices, but with bigger matrices `pdsyevx` tends to be better. The number of processors used with the Jacobi method is restricted because $n = 2tkq$, with $q=\frac{-1+\sqrt{1+8p}}{2}$.

Table 2: Comparison of the efficiency of `pdsyevx` and the Jacobi method.

	Paragon					
pr:	10		36		136	
met:	syev	Jac	syev	Jac	syev	Jac
128	0.19	0.28	0.04	0.05	0.01	0.01
256	0.37	0.49	0.11	0.28	0.02	0.13
384	0.45	0.58	0.16	0.42	0.03	0.24
512	0.52	0.64	0.21	0.47	0.04	0.26
640	0.58	0.65	0.26	0.52	0.06	0.29
768	0.62	0.65	0.30	0.54	0.07	0.36
896	0.65	0.67	0.34	0.55	0.09	0.35
1024	0.67	0.72	0.37	0.62	0.10	0.43
	Network					
pr:	5		7		9	
met:	syev	Jac	syev	Jac	syev	Jac
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
640	0.29	0.49	0.18	0.33	0.16	0.32

This is why the optimum number of processors varies greatly. The routine `pdsyevx` has been executed with the same number of processors as the Jacobi method and using a mesh as similar to a square mesh as possible, which produces an execution time near to the optimum [1].

On a network the number of processors is bigger with the Jacobi method than with `pdsyevx`, but the high cost of communications produces a smaller number, and Jacobi cannot reduce the execution time of `pdsyevx`. A network with 13 processors has been used.

To analyse how the methods work when the number of processors and the matrix' size vary, in table 2 the efficiency of `pdsyevx` and the Jacobi method is compared. The efficiencies are obtained comparing each parallel algorithm with itself executed on one processor.

Analysing table 2 we can conclude the Jacobi algorithm is more suitable for parallelization than the method used in `pdsyevx`. When the number of processors increases the efficiency of the Jacobi algorithm is clearly greater than the efficiency of `pdsyevx`, and this difference increases with the number of processors. The Jacobi method presented has a low efficiency with a small number of processors, which is mainly due to the imbalance in the arithmetic computation. But this imbalance is compensated when using a large number of processors with the overlap of arithmetic operations and communications.

4 Conclusions

For solving the Symmetric Eigenvalue Problem with a large number of processors computing all the eigenvalues of a dense matrix the Jacobi methods can be competitive with other methods based on reduction of the original matrix to condensed form (as the method used in the routine `pdsyevx` of ScaLAPACK).

On a MPP, in some cases (depending on the matrix' size, the number of processors, ...), it is preferable to solve the problem using a Jacobi method; but on a network of processors `pdsyevx` clearly surpasses the Jacobi method because the high efficiency of Jacobi methods cannot be totally exploited due to the high cost of communications.

References

[1] ScaLAPACK User's Guide. 1996.

[2] D. Giménez. A comparison of the Symmetric Eigenvalue Problem with ScaLAPACK and Jacobi methods. In *Eight SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1997.

[3] J. Demmel and K. Stanley. The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 528–533. SIAM, 1995.

[4] R. Schreiber. Solving eigenvalue and singular value problems on an undersized systolic array. *SIAM J. Sci. Stat. Comput.*, 7(2):441–451, 1986.