Acta Cybernetica **15** (2001) 137–149.

Parallel implementation for large and sparse eigenproblems*

E. M. Garzón† and I. García ‡

Abstract

This paper analyses and evaluates the computational aspects of an efficient parallel implementation for the eigenproblem. This parallel implementation allows to solve the eigenproblem of symmetric, sparse and very large matrices. Mathematically, the algorithm is supported by the Lanczos and Divide and Conquer methods. The Lanczos method transforms the eigenproblem of a symmetric matrix into an eigenproblem of a tridiagonal matrix which is easier to be solved. The Divide and Conquer method provides the solution for the eigenproblem of a large tridiagonal matrix by decomposing it in a set of smaller subproblems. The method has been implemented for a distributed memory multiprocessor system with the PVM parallel interface. A Cray T3E system with up to 32 nodes has been used to evaluate the performance of our parallel implementation. Due to the super-lineal speed-up values obtained for all the studied matrices, a detailed analysis of the experimental results is carried out. It will be shown that the management of the memory hierarchy plays an important role in the performance of the parallel implementation.

1 Introduction

Eigenproblems arise in a large number of disciplines of sciences and engineering. For example, they are used in: designing buildings, bridges and turbines; modeling queuing networks; analyzing stability of electrical networks; studying the fluid flow and so on. The matrices of these problems have a high dimension, a very low percentage of non-zero elements and, in general, they are non-symmetric. However, the symmetric eigenproblem constitutes the key in a lot of strategies to solve non-symmetric eigenproblems.

The computational cost and the memory requirements of the algorithms which provide a solution for the symmetric eigenproblem are very high when the matrix has a high dimension. In this context, the development of parallel algorithms and

*This work has been supported by the Ministry of Education of Spain (CICYT TIC99-0361).

†Department of Computer Architecture, University of Almería, 04120-Almería, Spain. e-mail: ester@ace.ual.es.

‡Department of Computer Architecture, University of Almería, 04120-Almería, Spain. e-mail: inma@ace.ual.es.

their efficient implementations on large scale supercomputer system are the only strategies which allow to solve this computationally expensive problem.

This paper deals with the parallel implementation of a strategy which provides a solution to the symmetric eigenproblem on a distributed memory multicomputer system. This strategy belongs to the so called direct methods and includes a divide and conquer technique.

The solution of the eigenproblem of a symmetric, large and sparse matrix $A \in R^{n \times n}$ can be obtained by the following transformations:

$$A = QTQ^T = QMDM^TQ^T = GDG^T \tag{1}$$

where D is a diagonal matrix whose non-zero elements represent the eigenvalues of A and T, and the columns of G and M are the eigenvectors of A and T, respectively.

The eigenproblem is solved by the following consecutive stages:

1. *Structuring the input matrix A.* This stage generates the tridiagonal matrix T and the orthonormal matrix Q, such that $A = QTQ^T$.

   The Lanczos Method with complete reorthogonalization is used at this stage. The complete reorthogonalization stage ensures the orthogonality of Q so that the spectrum of T is also the spectrum of A.

2. *Solving the eigenproblem of T.* At this stage the diagonal D and the orthogonal M matrices that give $T = MDM^T$ are the results of applying the Divide and Conquer method (DC). A divide-and-conquer method, developed by Cuppen [4], has been implemented at this stage. This method is decomposed in the following process:

   (a) Decomposing the eigenproblem of T in a set of $Sp_0 = 2^{Nv}$ subproblems $(T_1, T_2, \ldots, T_{Sp_0})$ of small dimension by rank-one transformations of T. Nv is the number of reconstruction levels.

   (b) Applying the QR method to every subproblem T_i. The corresponding eigenvalues and eigenvectors D_i and M_i, respectively, are obtained.

   (c) Reconstruction: From the set of matrices D_i and M_i $(i = 1, 2, \ldots, Sp_0)$, this procedure obtains the eigenvalue matrix D and the eigenvector matrix M. The reconstruction stage consists of a binary tree-structure of Nv levels.

3. *Computing the eigenvectors of A.* Let G be the orthonormal matrix whose columns g_i are the eigenvectors of A; i.e. $A = GDG^T$. This stage is solved by a matrix-matrix product, $G = QM$.

From the above algorithmic description, it can be seen that the parallel implementation of the eigenproblem consists of three consecutive parallel stages. Each stage manages a different kind of data structure. The first stage (Lanczos method) deals with an irregular sparse matrix; the second stage involves computations on a set of tridiagonal sub-matrices (structured data) and a set of dense matrices; the third stage is simply a product of dense matrices.

The parallel implementation of the first stage (Lanczos) is based on a decomposition in domains of the input sparse matrix which includes a computationally inexpensive pre-processing stage, namely Pivoting-Block [9]. This preprocessing stage guarantees that the input data partitions and its associated computations are balanced.

The parallel implementation of the Divide and Conquer method is based on a decomposition in domains of the input and output data [13]. The binary tree of tasks is distributed among Processing Element (PE) so that the same number of branches of the tree is allocated to each PE. When the number of PEs of the multiprocessor system P verifies that $P < Sp_0$, each PE starts the reconstruction process independently until the number of sub-problems is equal to P. When $P \geq Sp_0$, a set of PEs collaborates for the solution of a pair of subproblems. As the reconstruction stages evolve, the dimension of the subproblems are greater and the number of PEs collaborating for the same pair of subproblems increases.

The parallel product of matrices ($G = QM$) is carried out starting with a partition of Q and M by rows among PEs. We have implemented a strategy which reduces the memory requirements.

The main contributions of this paper consists of: (a) providing a parallel implementation for large sparse eigenproblems by linking the above described stages; (b) evaluating the proposed parallel implementation using a wide variety of problems and (c) doing a computational analysis to determine which factors are responsible for the super-lineal speed up values obtained from our experimental results.

This paper is organized as follows. In Section 2, the mathematical foundations of the applied method are briefly introduced. In Section 3, parallel implementations of every stage is described. Finally, in Section 4, new and non-standard performance indicators for evaluating parallel implementations are defined. Moreover, experimental results of the performance evaluation of our parallel implementation are shown and discussed. Performance evaluations were carried out by a multicomputer (Cray T3E) using no more than 32 processing elements.

2 Describing the Applied Methods

The eigenproblem of large and sparse matrices is solved by a direct method which allows to determine the matrix decompositions described by (1).

The Lanczos method, briefly described in Subsection 2.1, is applied to obtain T and Q (tridiagonal and orthogonal matrices, respectively). The eigenproblem of T is solved by the Divide and Conquer method [11] (Subsection 2.2), which obtains matrices D and M from T, where D is a diagonal matrix whose elements are the eigenvalues of T and A, and columns of $G = QM$ are the eigenvectors of A.

2.1 Structuring Sparse Matrix

The Lanczos Method with Complete Reorthogonalization has been used for finding a structured matrix with the same spectrum as the input sparse matrix. The

Lanczos method is considered an effective method for obtaining, from a symmetric matrix A, a symmetric tridiagonal matrix T and a set of orthonormal vectors, q_j $(0 < j \leq n)$. Given a symmetric matrix $A \in R^{n \times n}$ and a vector q_1 with unit norm, the Lanczos method generates the orthonormal matrix Q and the tridiagonal matrix T, in n iterative steps. This method is discussed in [1, 3, 12, 14]. The outer loop of the Lanczos algorithm is an iterative procedure (index j) which at the *j-th* iterative step computes the α_j and β_j coefficients of the tridiagonal matrix T and the vector q_{j+1}, where α_j and β_j denote the elements of the main and secondary diagonals of T, respectively. This loop includes a sparse matrix-vector product and a reorthogonalization process. The reorthogonalization procedure used in this work is the so called complete reorthogonalization (CR). CR is computationally expensive but it allows to ensure that the eigenvalues of A and T are the same. The Lanczos algorithm is particularly appropriate for structuring sparse matrices of high dimension.

2.2 Solving the eigenproblem of T

A solution for the eigenproblem of a tridiagonal matrix based on the Divide and Conquer method (DC) was proposed by Golub [11] and, lately, developed by Bunch, Nielsen and Sorensen [2] and Cuppen [4].

The key of this method consists of dividing the input matrix of high dimension into several sub-matrices of lower dimension and solving the eigenproblem of high dimension from the solutions of eigenproblems of lower dimension. This strategy is very useful to solve the eigenproblem of very high dimensions. The DC method can be described as in Algorithm 1.

Algorithm 1 Divide and Conquer Algorithm: $DC(T) \rightarrow D, M$

1 *do* $i = 1, \ldots, Sp_0;\ i + 1$
2 Div $(i) \rightarrow T_i$ # SubDivision #
3 QR$(T_i) \rightarrow D_i, M_i$ # Solving Small Eigenproblems #
4 $Sp = Sp_0$;
5 *do* $k = 1, \ldots, Nv;\ k + 1$ # Reconstruction Levels #
6 *do* $i = 1, \ldots, Sp - 1;\ i + 2$ # Reconstructing Couples of SubMatrices #
7 Reconstruction $\rightarrow \hat{D}_{i/2}, \hat{M}_{i/2}$
8 First Deflation
9 Second Deflation
10 Solve Secular Equation $\rightarrow \hat{D}_{i/2}, V$
11 $\hat{M}_{i/2} = \begin{pmatrix} M_i & 0 \\ 0 & M_{i+1} \end{pmatrix} V$ # Intermediate matrix-matrix Product#
12 $Sp = Sp/2$

The subdivision process by rank one modifications of T is associated with line 2. This process generates the set of sub-matrices (T_i). Then, the eigenproblem of

every T_i is solved by the QR method [12] which generates the D_i (eigenvalues) and M_i (eigenvectors) matrices. (Sp_0 denotes the number of initial subproblems).

The loop with index k is associated with the reconstruction process (lines 5-12), where k denotes the level of reconstruction. The total number of levels of reconstruction is given by $Nv = log_2(Sp_0)$. The level of reconstruction k includes $Sp/2$ reconstruction processes for a couple of sub-matrices (T_i, T_{i+1}) whose eigenvalues and eigenvectors are known (D_i, M_i and D_{i+1}, M_{i+1}). The outputs of this process are $\hat{D}_{i/2}, \hat{M}_{i/2}$. These matrices are the solution of the eigenproblem for the sub-matrix which is the result of the association of a couple of tridiagonal submatrices (T_i, T_{i+1}). Details about the reconstruction process are described in [4]. This process may include deflations which reduce the computational cost of the secular equation solution and the dimensions of the matrices which are included in the so called Intermediate matrix-matrix Product (line 11). The number of subproblems at level k is denoted by Sp. When $Sp = 2$ the eigenproblem of T is solved ($\hat{D}_{i/2} = D, \hat{M}_{i/2} = M$).

2.3 Computing the eigenvectors of A

The matrix-matrix product $G = QM$ is computed in order to obtain the eigenvectors of the input matrix A. This last stage, which completes the solution of the eigenproblem of A, is computationally very expensive and needs large memory requirements. However, if the goal were only to compute the spectrum of the input matrix, this stage could be omitted.

3 Parallel Implementation

The method for solving the eigenproblem of a symmetric sparse matrix, discussed in Section 2, has an extremely high computational cost. Furthermore, this method demands large memory requirements. Consequently, its implementation on a distributed memory multiprocessor is necessary, specially, when the input matrix A is of high dimension.

The parallel implementation of the method has been carried out using a SPMD programming model and the PVM standard library. The whole solution of the eigenproblem (LDC) has been divided into a set of procedures: *Lanczos*, *DC* and *Final Product matrix-matrix*. These procedures link their executions in a sequential way because the data dependences prevent several procedures from simultaneous execution. Thus, every procedure must be independently parallelized.

3.1 The Lanczos Method

In the solution of the eigenproblem, the *Lanczos* method is the only procedure which manages irregular data structures; i.e. a sparse input matrix A. Since the Lanczos algorithm works on mixed computations (dense-sparse), special care must be taken in the data distribution among processors in order to optimize the

work load balance for computations on both dense and sparse data structures. A data distribution called *Pivoting Block* [9] is used to balance the computational load of this procedure. Pivoting Block estimates a permutation of the rows of A obtaining an homogeneous density of the non-zero elements. Thus, the classical block partition applied to the permuted matrix is able to obtain a similar number of non-zero elements for the sparse sub-matrix allocated at every PE. Consequently, computations linked to dense and sparse structures are balanced. Details about parallel implementation of *Lanczos* algorithm can be found in [7, 8]. The outputs of this parallel algorithm are: the tridiagonal matrix T and the orthonormal matrix Q. The tridiagonal matrix T is stored at the local memory of every Processing Element (PE). When the parallel *Lanczos* procedure finishes, every PE stores $\lceil \frac{n}{P} \rceil$ rows of Q at its local memory, where P is the number of PEs in the multiprocessor system.

3.2 The *DC* Method

An efficient parallel implementation of *DC* method on share memory multiprocessors has been proposed by Dongarra and Sorensen [5]. Moreover, parallel implementations on a distributed memory multicomputer have been described by Ipsen and Jessup [13] and Tisseur and Dongarra [15]. In our implementation we have used most of the ideas described in [13].

The structure of the *DC* algorithm suggests a natural way to split and distribute the computational work among the set of PEs. The *DC* method can be represented by a binary tree of tasks which can be decomposed into P sets of tasks and distributed among PEs.

As an illustration, the example in Figure 1 starts with T subdivided into $SP_0 = 16$ sub-matrices and a multiprocessor system with $P = 4$ is considered. As it can be seen in Figure 1, for a problem which is subdivided into 16 subproblems, the *DC* method consists of 4 reconstruction levels. The reconstructions at levels $k = 1$ and 2 are carried out by every isolated PE. When $k = 3$, two groups of two PEs are defined. Thus, every group of PEs cooperates in the reconstruction of a couple of sub-matrices. At the final level ($k = 4$), the four PEs cooperate in the last reconstruction.

At the end of this stage, the non-zero elements of D and the rows of M are distributed among the set of PEs.

3.3 The Final matrix-matrix product

In order to complete the solution of the eigenproblem of the input matrix A, the matrix G is computed by the matrix-matrix product $G = QM$. The parallel implementation of this matrix-matrix product is more difficult than the standard one because every processor allocates only a subset of rows of Q and M. However, the communication time and the memory requirements have been optimized by re-using data structures defined and used at previous stages.

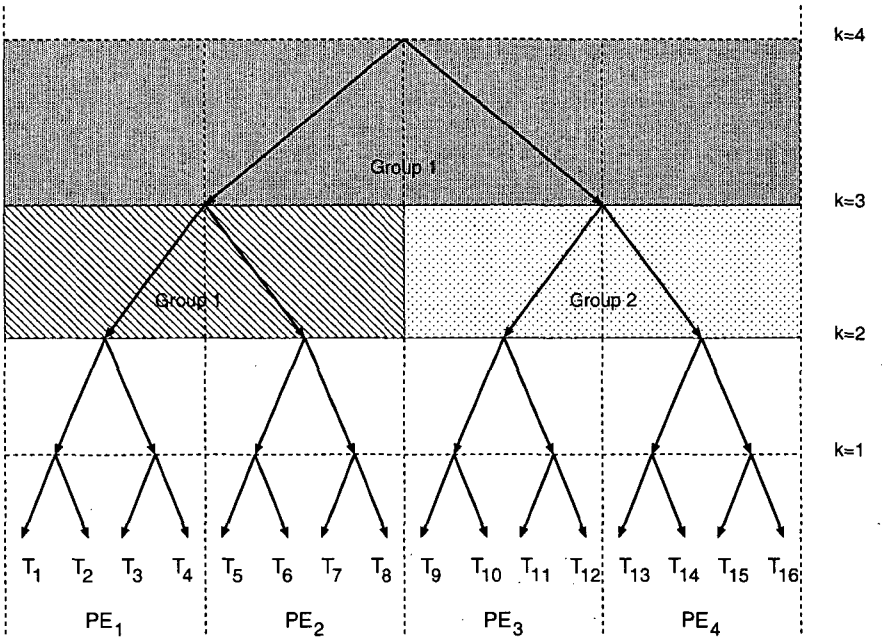

Figure 1: Binary tree of tasks for the *DC* method, and task distribution among PEs for a multiprocessor system with $P = 4$.

4 Evaluation

The evaluation of the parallel implementation for the solution of the eigenproblem has been carried out on a multiprocessor system Cray T3E using a set of n dimensional input matrices. Some of the matrices belong to the Harwell-Boeing collection of test matrices [6]. Moreover, a subset of the test matrices has been designed to analyze the parallel implementation of the *DC* method. These test matrices are obtained by permutating a set of tridiagonal matrices denoted by [1,μ,1] or [1,k,1] [13], where $\alpha_k = k\mu$ or $\alpha_k = k$, respectively, and $\beta_k = 1$ $(k = 1 \ldots n)$. These matrices are denoted by "tn", where n is the dimension of the input matrix. The selection of the initial Lanczos vector (q_1) allows us to control the number of deflations the *DC* method produces. Consequently, it is possible to generate test problems with a high or a low computational cost for the *DC* method.

In Table 1, three matrices of the set of test matrices used in the evaluation of our parallel implementation are characterized by parameters like the dimension of the matrix (n) and the percentage of non-zero elements (γ). The last two columns of Table 1 provide numerical results which specify the accuracy of the applied methods. Specifically, numerical results for the parameters $log_{10}\frac{R}{\|A\|_F}$ and $log_{10}Ort$ are given; where R is the norm of the residual related to the eigenproblem solution $(R = \| AG - GD \|_F)$ and $Ort = \| G^TG - I \|_F$ provides a measurement of the

orthogonality of the eigenvectors ($\| \cdot \|_F$ denotes the Frobenius matrix norm).

Table 1: Numerical results of the accuracy of LDC method for several test matrices.

Matrix	n	γ	$log_{10} \frac{R}{\|A\|_F}$	$log_{10} Ort$
BFW782B	782	0.9 %	-15	-12
t1024	1024	0.3 %	-14	-13
t2048	2048	0.1 %	-14	-12

The parallel performance evaluation was carried out executing the algorithm with $P = 1, 2, 4, 8, 16$ and 32 PEs and subdividing the tridiagonal matrix T into P sub-matrices; i.e. $Sp_0 = P$ for the DC method. However, executions using only one or a few PEs were only possible for some of the test matrices (the smaller ones) because of the fact that the computer ran out of memory for large matrices; for example for the matrix $t7168$ the multiprocessor system ran out of memory when less than 16 PEs were used, so execution times were only obtained for $P = 16$ and 32 PEs. Under these circumstances it was not possible to compute the standard values of the speed-up for evaluating the performance of the parallel implementation. As an alternative to the speed-up, we have defined a new parallel performance estimator called Incremental Speed-up ($IncSpUp$) which provides information about how much the computing time diminishes when the number of PEs increases. $IncSpUp$ is defined as follows:

$$IncSpUp(2^i) = \frac{T(P = 2^{i-1})}{T(P = 2^i)}, \tag{2}$$

where T(P) is the run time of the execution with P PEs. For ideal parallel implementations, the value of the Incremental Speed-up should be equal to $IncSpUp = 2$, which corresponds to a lineal speed-up [10].

The experimental values for the Incremental Speed-up obtained from executions of our parallel implementation have been plotted in Figure 2. A set of eight matrices whose dimension n ranges between 782 and 7168 was used as test matrices; two of the matrices belong to the Harwell-Boeing collection (BFW728B and BCSSTK27), the remaining test matrices belong to the above described kind of matrices (tn). For every tn matrix the parallel algorithm was run twice; one of them producing many deflations and the other few deflations. As it was previously described, many or few deflations may appear depending on the value of the initial Lanczos vector (q_1). In Figure 2, tn and tn^* graphs correspond to the same matrix but for execution with few and many deflations, respectively.

From Figure 2 performance of the parallel implementation can be analyzed for every value of the number of PEs, P. For every tested matrix such that $n \geq 2048$, the values of the $IncSpUp$ estimator were greater than 2 when $2 \leq P \leq 16$ but for $P = 32$ only the largest matrices ($t5120$ and $t7168$) gave $IncSpUp \geq 2$. From the definition of the $IncSpUp$ it is easy to see that from the values of the $IncSpUp$ for $2, 4, \ldots, P$ PEs, it is possible to obtain the value of the Speed-up for P PEs because

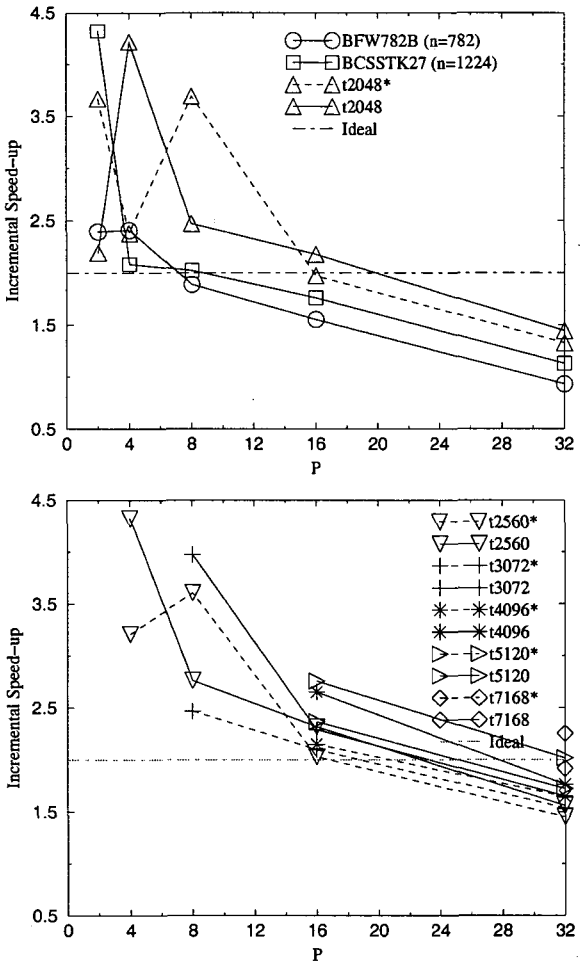

Figure 2: Incremental Speed-up of the parallel execution of LDC against the number of processors (P) for several input matrices.

$SpUp(P) = IncSpUp(P) \times IncSpUp(P/2) \times \ldots \times IncSpUp(2)$. The values of $IncSpUp$ estimator obtained in our experimental results are equivalent to efficiency values higher than 1, it means that our implementation exhibits a super-lineal behavior.

Notice that, as the number of PEs increases, the computational work load of every processor diminishes but the interprocessor communications and delays for synchronizing tasks do not decrease but even may increase. Values of the $IncSpUp$ less than 2 can have been produced as a consequence of long delays for synchronization, which are mainly due to work load unbalances among processors, or long

interprocessor communications.

In an attempt to determine the causes for both the super-lineal Speed-up behavior as well as the decreasing of the $IncSpUp$ when the number of PEs increases, a detailed analysis of the performance was carried out. This analysis was made through a pair of additional parameters: the number of cache faults and the execution profiles. Execution profiles provide measurements of the percentage of the computational work related to every procedure involved in the parallel algorithm. Experimental results will show that the management of the memory hierarchy plays an important role in algorithms with large memory requirements.

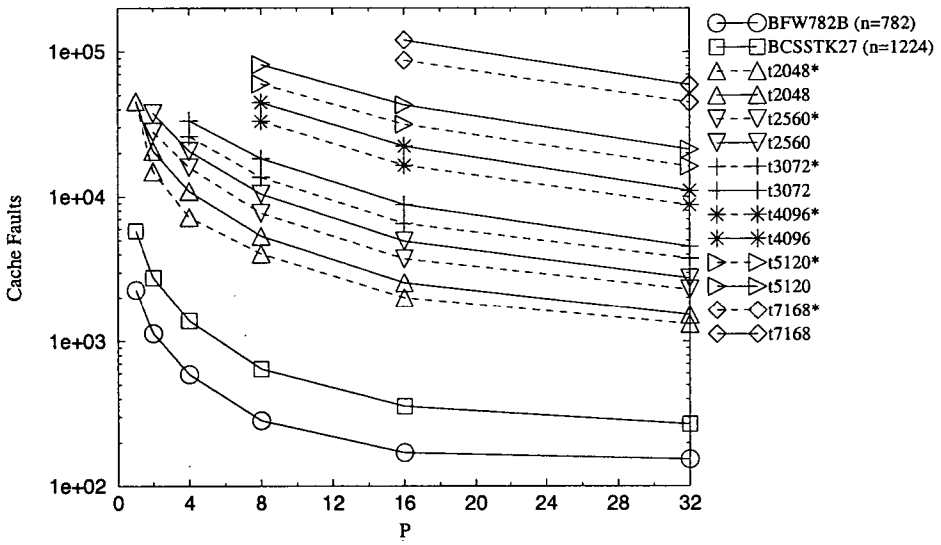

Figure 3: Cache faults versus the number of PEs for several dimensions of input matrices. (*) means many deflations.

Figure 3 shows a log plot of the number of cache faults against P. It can be seen that the number of cache faults diminishes considerably as P increases and this fact is more relevant for small values of P. This is mainly due to the fact that the percentage of the total data that can be allocated at the cache memory is greater when more PEs are used. This means that the time spent on accessing to data memory decreases as the number of PEs increases. This justifies that super-lineal Speed-up values ($IncSpUp \geq 2$) have been obtained in our experimental results.

Experimental results for execution profiles are shown in the Table 2. The procedures included in the LCD algorithm that have stronger computational work load are:. *Lanczos* (lc), QR method (QR), intermediate product of matrices (IP, line 11 of DC) and final product of matrices (FP). The characters in brackets are referred to the notation of the column head in Table 2. Moreover, this table has two additional columns which specify the percentages of the run time related to the waiting time for synchronization (wt) and the interprocessor communication process (c).

Table 2: Execution profile of *LDC* for several test matrices. *lc, FP, IP, QR, wt* and *c* denote the percentage of the work load associated to the Lanczos method, the Final Product of matrices, the Intermediate Product of matrices, the QR method, the wait for messages time and the interprocessor communications, respectively.

	Many Deflations					*Few Deflations*					
P	*lc*	*FP*	*QR*	*wt*	*c*	*lc*	*FP*	*IP*	*QR*	*wt*	*c*
	t2048										
2	39	55	2	3	-	24	33	25	2	14	-
4	30	66	-	1	-	32	25	33	-	6	-
8	46	45	-	4	1	33	32	21	-	9	-
16	42	42	-	2	10	33	33	18	-	4	6
32	28	27	-	5	31	23	23	11	-	6	26
	t3072										
2	47	48	-	3	-	32	32	25	1	7	-
4	55	42	-	1	-	23	27	30	-	17	-
8	46	47	-	3	-	32	32	23	-	9	-
16	45	44	-	1	6	34	33	19	-	4	4
32	33	33	-	4	14	27	27	14	-	5	18
	t5120										
8	49	47	-	1	-	28	26	36	-	6	-
16	47	46	-	-	3	33	31	21	-	9	2
32	38	38	-	3	15	30	30	17	-	5	11
	t7168										
16	50	45	-	-	2	30	29	28	-	9	-
32	42	41	-	2	10	32	31	18	-	6	8

On the left side of Table 2, the results are associated with executions of the *DC* procedure that include many deflations, so the reconstruction process and the intermediate products (*IP*) are not very hard from a computational point of view. On the right hand side, we can see the results associated with the *DC* procedure that includes few deflations. So, the intermediate products represent a relatively large percentage of the total computational work.

From data in Table 2, it can be seen that the communication processes are computationally irrelevant except for execution with $P = 16$ and $P = 32$, but their importance decreases when n increases.

Notice that the values of the *wt* parameter are also estimations of the work load balance among processors since a synchronization stage always precedes every communication among processors. For all the analyzed cases the value of *wt* is extremely small. Thus, the parallel implementation has a good work load balance. In [13], from the point of view of parallel implementation, deflations have been described as a serious drawback, as they can produce load unbalance. Nevertheless, the values of *wt* obtained in our experimental results show that deflations do not produce a relevant work load unbalance.

5 Conclusions

In this paper a parallel implementation of the eigenproblem of symmetric sparse and large matrices is proposed and evaluated. The solution is based on a direct method which mainly consists of three consecutive stages. Parallel implementations of every isolated stage have been described in the bibliography [7, 8, 13, 15]. However, a parallel implementation for the whole eigenproblem solution which includes these methods has not been reported anywhere. Our proposal is able to provide a solution for very large matrices which can not be solved with a uniprocessor system due to both the high computational complexity and the large memory requirements. We have solved all the problems associated to the work load unbalance which frequently appear when sparse matrices are involved in parallel computations.

A detailed analysis of the parallel implementation has been carried out through the experimental values of the Incremental Speed-up, the number of cache faults and the execution profiles. It has been proved that the designed parallel implementation is very efficient since it includes specific devices which allow: (a) distributing the computational work load associated with all the procedures in a balanced way; (b) establishing interprocessor communications that do not increase considerably the run time, and what is more, (c) improving the mamory data access time, especially for irregular data.

References

[1] Berry, M.W. Large-scale sparse singular value computations. *The International Journal of Supercomputer Applications*, 6(1):13–49, Spring 1992.

[2] Bunch, J.R.; Nielsen C.P. and Sorensen, D.C. Rank one modification of the symmetric eigenproblem. *Numerische Mathematik*, 31(1):31–48, 1978.

[3] Cullum, J.K. and Willoughby, R.A. *Lanczos algorithms for large symmetric eigenvalue computations*, volume 1: Theory, volume 2: Programs. Birkhäuser, Stuttgart, 1985.

[4] Cuppen, J.J.M. A divide and conquer method for the symmetric eigenproblem. *Numerische Mathematik*, 36:177–195, 1981.

[5] Dongarra, J.J. and Sorensen, D.C. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM Journal on Scientific and Statistical Computing*, 8(2):139–154, March 1987.

[6] Duff, I.S.; Grimes, R.G. and Lewis, J.G. User's guide for the Harwell-Boeing sparse matrix collection. Technical report, Research and Technology Division, Boeing Computer Services, 1992.

[7] García, I; Garzón, E.M.; Cabaleiro, J.C.; Carazo, J.M. and Zapata, E.L. Parallel tridiagonalization of symmetric matrices based on Lanczos method. In

M. Valero, E. Onate, M. Jane, J. L. Larriba, and B. Suarez, editors, *Parallel Computing and Transputer Applications*, pages 236–245, Amsterdam, 1992. IOS Press.

[8] Garzón, E.M. and García, I. Parallel implementation of the Lanczos method for sparse matrices: Analysis of data distributions. In ACM, editor, *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing*, pages 294–300, New York, 1996. ACM Press.

[9] Garzón, E.M. and García, I. Evaluation of the work load balance in irregular problems using Value Based Data Distributions. *Proceedings of the IASTED International Conference Parallel and Distributed Systems. Euro-PDS'97*, pages 137–143, 1997.

[10] Garzón, E.M. and García, I. A parallel implementation of the eigenproblem for large, symmetric and sparse matrices. In J.J. Dongarra, E. Luque, and T. Margalef, editors, *Recent advances in PVM and MPI*, volume 1697 of *Lecture Notes in Computer Science*, pages 380–387. Springer-Verlag, 1999.

[11] Golub, G.H. Some modified matrix eigenvalue problems. *SIAM Review*, 15(2):318–344, April 1973.

[12] Golub, G.H. and Van Loan C. F. *Matrix computations.* Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

[13] Ipsen, I.C.F. and Jessup, E.R. Solving the symmetric tridiagonal eigenvalue problem on the hypercube. *SIAM Journal on Scientific and Statistical Computing*, 11(2):203–229, March 1990.

[14] Simon, H.D. Analysis of the symmetric Lanczos algorithm with reorthogonalization methods. *Linear Algebra and its Applications*, 61:101–131, 1984.

[15] Tisseur F. and Dongarra J. A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures. *SIAM Journal on Scientific Computing*, 20(6):2223–2236, November 1999.