Sparse Matrix Operations in Vector and Parallel Processors

R. Doallo, J. Touriño & F.M. Hermo
Department of Electronics & Systems, University of La Coruña, Campus de Elviña s/n, 15071 La Coruña, Spain
Email: {doallo,juan}@udc.es

Abstract

Vector computers have been extensively used for years in matrix algebra to treat with large dense matrix problems. However, if matrices are sparse and we use special storage schemes for them, vectorization provides a poor performance due to the great amount of indirections in the code. An alternative option is the utilization of a multiprocessor (or a cluster of workstations); in this case, a data parallel programming model also fails because of the reason pointed out for vector computers. Therefore, the best choice is to parallelize the corresponding algorithms using message passing routines. In order to discuss these features, we will focus on solving sparse linear least squares problems, which appear in several scientific areas such as structural analysis, geodetic survey, molecular structure and many others. Experimental results are obtained for vector and parallel computer architectures.

1 Introduction: Linear Least Squares Problems

The Least Squares Problem (LSP) is defined as follows: $min\|Ax-b\|_2$, where $A \in \Re^{m\times n}, x \in \Re^n, b \in \Re^m$. Several methods have been developed to solve this kind of problems[4]: the augmentation method, the Peters-Wilkinson procedure, the method of normal equations and iterative algorithms. These problems can also be solved by adapting the algorithms which carry out the decomposition $A = QR$ (Q is an orthogonal matrix and R is upper triangular), which has good numerical stability. We have focused on two different QR algorithms[5]: the Modified Gram-Schmidt method (MGS) and the Householder procedure, both of them with column pivoting. In particular, the LSP is equivalent to solving the upper triangular system $R\Pi^T x = Q^T b$, by means of a back-substitution (Π is a permutation matrix generated by the column pivoting). We focus on the corresponding sequential algorithms

$$
\begin{array}{l}
rank = n; \\
for\ (j{=}0;\ j{<}n;\ j{+}{+}) \\
\quad norm_j = \sum_{i=0}^{m-1} a_{ij}^2; \qquad (1) \\
for\ (k{=}0;\ k{<}n;\ k{+}{+})\ \{ \\
\quad \text{Find } p,\ k \leq p < n, \text{ such that } norm_p = \max_{k \leq j < n} norm_j; \qquad (2) \\
\quad \text{if } (norm_p == 0)\ \{ \qquad (3) \\
\quad\quad rank = k; \quad \text{break;} \\
\quad \} \\
\quad \text{else swap } (norm_k,\ a_{0:m-1,k},\ r_{0:n-1,k}) \text{ and} \\
\quad\quad\quad (norm_p,\ a_{0:m-1,p},\ r_{0:n-1,p}); \qquad (4) \\
\quad r_{k\,k} = \sqrt{norm_k}; \qquad (5) \\
\quad for\ (i{=}0;\ i{<}m;\ i{+}{+}) \\
\quad\quad a_{i\,k} = a_{i\,k} / r_{k\,k}; \qquad (6) \\
\quad for\ (j{=}k{+}1;\ j{<}n;\ j{+}{+})\ \{ \\
\quad\quad r_{k\,j} = \sum_{i=0}^{m-1} a_{i\,k} \cdot a_{ij}; \qquad (7) \\
\quad\quad norm_j = norm_j - r_{k\,j}^2; \qquad (8) \\
\quad\quad for\ (i{=}0;\ i{<}m;\ i{+}{+}) \\
\quad\quad\quad a_{ij} = a_{ij} - a_{i\,k} \cdot r_{k\,j}; \qquad (9) \\
\quad \} \\
\}
\end{array}
$$

Figure 1: *MGS algorithm*

to perform the QR factorization, which is the core for solving the LSP.

In the MGS factorization (figure 1), matrix A is overwritten by matrix Q. In (1) the squares of the euclidean norms of the columns of matrix A are stored in vector *norm*. Then, an iterative procedure of n iterations is performed. This consists of the following actions: the pivot column (p) and the pivot element ($norm_p$) are selected (2); if the pivot is 0, the rank of the matrix is k and the factorization ends (rank-revealing factorization) (3); otherwise, a column swapping is performed (4). In (5-9) we update matrix A entries $a_{0:m-1,k:n-1}$ (6),(9) and matrix R entries $r_{k,k:n-1}$ (5),(7).

The Householder algorithm with column pivoting (figure 2) is similar to MGS, substituting expressions (5-9) by (10-14). Matrix A is overwritten by matrix R, and Q is not obtained because it is not explicitly necessary for the solution of the LSP. In (10), the *Householder vector* $v \in \Re^{m-k}$ is calculated, so that the product between $P{=}\ (I - 2vv^T/v^Tv)$ and the subcolumn $a_{k:m-1,k}$ is zero in all the components except for the first one; I is the identity matrix and P, an orthogonal matrix, is called a *Householder reflection* or *transformation* (P and $I \in \Re^{(m-k)\times(m-k)}$). In (11-13) a Householder reflection is

$$
\begin{aligned}
&\mathrm{v}_k = 1.0; \\
&\textit{for (i=k+1; i<m; i++)} \\
&\quad v_i = a_{i\,k}/(a_{k\,k} + sign(a_{k\,k}) \cdot \sqrt{norm_k}); && (10) \\
&\beta = -2 \ \Big/ \sum_{i=k}^{m-1} v_i^2; && (11) \\
&\textit{for (j=k; j<n; j++)} \ \{ \\
&\quad w_j = \beta \cdot \sum_{i=k}^{m-1} a_{ij} \cdot v_i; && (12) \\
&\quad \textit{for (i=k; i<m; i++)} \\
&\qquad a_{ij} = a_{ij} + v_i \cdot w_j; && (13) \\
&\quad norm_j = norm_j - a_{k\,j}^2; && (14) \\
&\}
\end{aligned}
$$

Figure 2: *Householder algorithm*

applied to submatrix $S \in \Re^{(m-k)\times(n-k)}$, made up of entries $a_{k:m-1,k:n-1}$, so that the original submatrix S is substituted by the product $P \times S$ in order to zero entries $a_{k+1:m-1,k}$. This product is obtained by performing the operation $S + vw^T$ (13), where $w = \beta S^T v \in \Re^{n-k}$ (12) and $\beta = -2/v^T v$ (11). Norms are updated in (14).

2 A Sparse Approach

Sparse linear least squares problems appear in several technical and scientific areas such as geodetic survey, structural analysis, tomography, photogrametry, molecular structure $\cdots$ Although a sparse problem could be treated with the corresponding dense program, the cost of ignoring sparsity, both in storage and execution time is very high.

The choice of the data storage scheme to support the sparse matrix is very important. Many different ways of storing sparse matrices have been developed to take advantage of the matrix pattern or the characteristics of the problems from which they arise[1,7,10]: Compressed Row/Column Storage (CRS/CCS), Block Compressed Row Storage, Compressed Diagonal Storage, Coordinate Format, Jagged Diagonal Format, Skyline Storage $\cdots$ The main factor we have taken into account in the selection of the data storage structure has been the nature of the algorithms, in which we only require efficient access by columns. Besides, in the QR decomposition problem, the structure of the matrices varies (due to the fill-in) and, therefore, it is convenient to use dynamic storage structures that support the changes in the pattern of the nonzero elements. Among the different types of dynamic storage structures we have chosen packed vectors[3]. As we can see in figure 3, each packed vector represents one column of the matrix. In

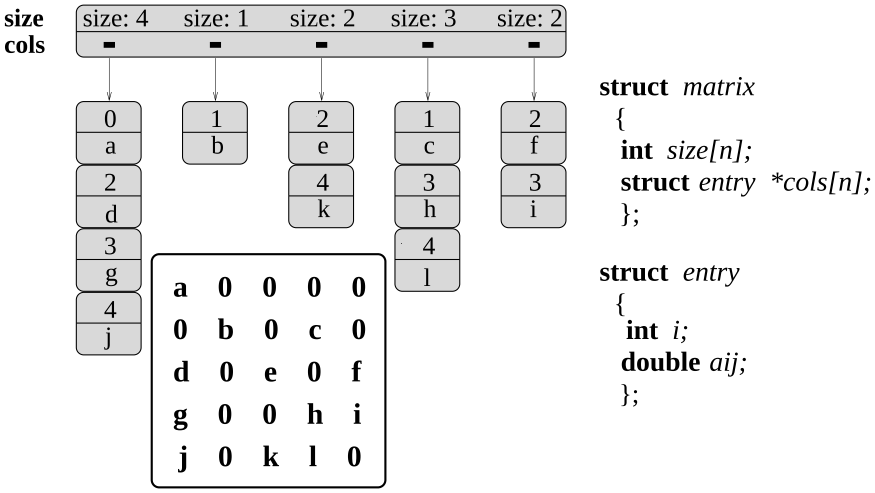

Figure 3: *Sparse matrix data structure*

fact, this structure follows a CCS format. We have also incorporated a new pivoting strategy[11] to preserve the sparsity of the resulting matrices Q and R. A threshold ϕ is defined, so that an entry a_{ij} is considered as zero if $|a_{ij}| < \phi$; in our algorithms, $\phi = 10^{-20}$.

Each packed vector item stores the row index and the matrix element. The elements of each matrix column are arranged in growing order of the row index. We also need an array *size* to count the number of nonzero elements per column and an array of pointers *cols* to the beginning of each packed vector. Note that, by means of this data structure, the column pivoting operation is simply reduced to a pointer interchange. We have also implemented QR algorithms using one-dimensional doubly linked lists as data structure[12]. Nevertheless, packed vectors are more suitable because the elements of the columns are stored in adjacent memory positions, which exploits locality, increases the cache hit ratio, reduces memory fragmentation and avoids having to traverse the links of the lists to access to the matrix entries; it also implies an important decrease of the running times. Moreover, as packed vectors have no pointers between two consecutive elements, memory savings with respect to linked lists are considerable.

The procedure to support fill-in using packed vectors is the following: an auxiliary buffer (which is also a packed vector), with size enough to store one dense column, is allocated. During the updating process of a matrix column, performed in both MGS and Householder algorithms, each nonzero entry (a previously existing one or a new one (fill-in)) is added to the buffer. On the other hand, if an existing entry is zeroed during the factorization, then it is simply discarded and it is not stored in the buffer. After finishing the column updating procedure, the auxiliary buffer stores the new column of the matrix and it is reallocated to a memory area with the exact size required to store this new packed column, whereas the memory space of the original column is freed.

Matrix	m×n	#A	% #A	$\mathbf{t}_{mgs}$	$\mathbf{t}_{hou}$
BP1600	822×822	4841	0.72%	6.05s	12.96s
JPWH991	991×991	6027	0.61%	100.87s	89.18s
SHERMAN1	1000×1000	3750	0.37%	21.03s	24.33s
MAHINDAS	1258×1258	7682	0.49%	11.78s	16.22s

Table 1: *Harwell-Boeing test matrices: execution times on a vector processor*

```
for (j=0; j<n; j++)
  for (ptr=qr[j].cols; ptr<limit; ptr++)
    buf[ptr->i] += ptr->aij;

              (a)

     for (i=0; i<qr[j].size; i++)
       *(ptr+i) = *(qr[j].cols+i);

              (b)
```

Figure 4: *Example codes for a vector processor*

3 Implementation on a Vector Processor

The first step of our work consisted of running the sparse QR algorithms on a vector processor. We have used the Fujitsu VP2400/10 uniprocessor model, which has a scalar and a vector unit. The algorithms were written in C code. A set of directives, which begin with the keyword *#pragma*, can be inserted in the code to provide additional information to the vectorizing compiler. Table 1 shows the characteristics of the test matrices used (extracted from the Harwell-Boeing suite): dimensions (m×n), number of nonzero elements (#A) and the percentage of these elements (% #A); it also shows the corresponding execution times (in seconds) for solving LSP through MGS and Householder (t_{mgs}, t_{hou}).

What is really important, from our point of view, is that special characteristics of our sparse codes result in low vectorization rates, $v = t_{s1}/t_s$ (where t_{s1} is the scalar CPU time of the part of the program which can be vectorized, and t_s is the scalar CPU time for the whole program). Hence, poor performances are achieved because the vector unit is under-used. Figure 4 shows two significant examples taken from QR algorithms source code. Vectorization is extremely difficult due to the complex data addressing (figure 4b); it even involves a variable of type pointer (*ptr*) in the limits of a loop and its corresponding increment (figure 4a).

```
CDIR$ DO SHARED(i) on vp(i)
DO i = inf, sup
  vu(row(i)) = vp(i)
ENDDO
```

(a)

```
induc = pos
DO i= inf, sup
  IF (vu(i) .NE. 0.0) THEN
    vp(induc) = vu(i)
    row(induc) = i
    induc = induc + 1
  ENDIF
ENDDO
```

(b)

Figure 5: *Craft data-parallel codes*

4 Implementation on Parallel Machines

4.1 A Data Parallel Approach

In a data parallel programming model, the user writes code using a global index space and compiler directives to set data and work distributions. Data parallel compilers (such as Vienna Fortran[2], High-Performance Fortran[8] and Craft[9]), taking into account this information, reorganize the code, translate global into local addresses and insert communications to execute the program on a multiprocessor following a SPMD model.

Craft is implemented on the Cray T3D multiprocessor. It consists of Cray FORTRAN77, some Fortran90 extensions (array syntax, for instance) and compiler directives, which begin with the keyword *CDIR$*. We coded the corresponding sequential algorithms in Fortran (there is not a C version of Craft), we used a CCS data structure and we dealt with fill-in by means of copying the data structure in each iteration of the algorithms to include the new generated elements. Craft (and most data parallel languages) only provides the classical $BLOCK(x)$ distribution for dense matrices, where x indicates that each Processing Element (PE) receives x contiguous elements of an array starting on PE #0 (if $x = 1$, we have a cyclic distribution). These distributions are not suitable for sparse matrix computations as we will show by means of our QR example codes: we have found code segments such as those shown in figure 5. The DO SHARED directive in figure 5a distributes the work of the loop; it assigns the iteration i to the PE which owns $vp(i)$. This piece of code unpacks a packed vector. The assignment statement with indirections at the left-hand side prevents locality exploitation (it is not guaranteed $vu(row(i)), row(i)$ and $vp(i)$ to be located in the same processor) because none of the dense data distributions provided by Craft are appropriate for the CCS sparse data structure. Code in figure 5b, packs the entries of a sparse vector. Craft fails when compiling this piece

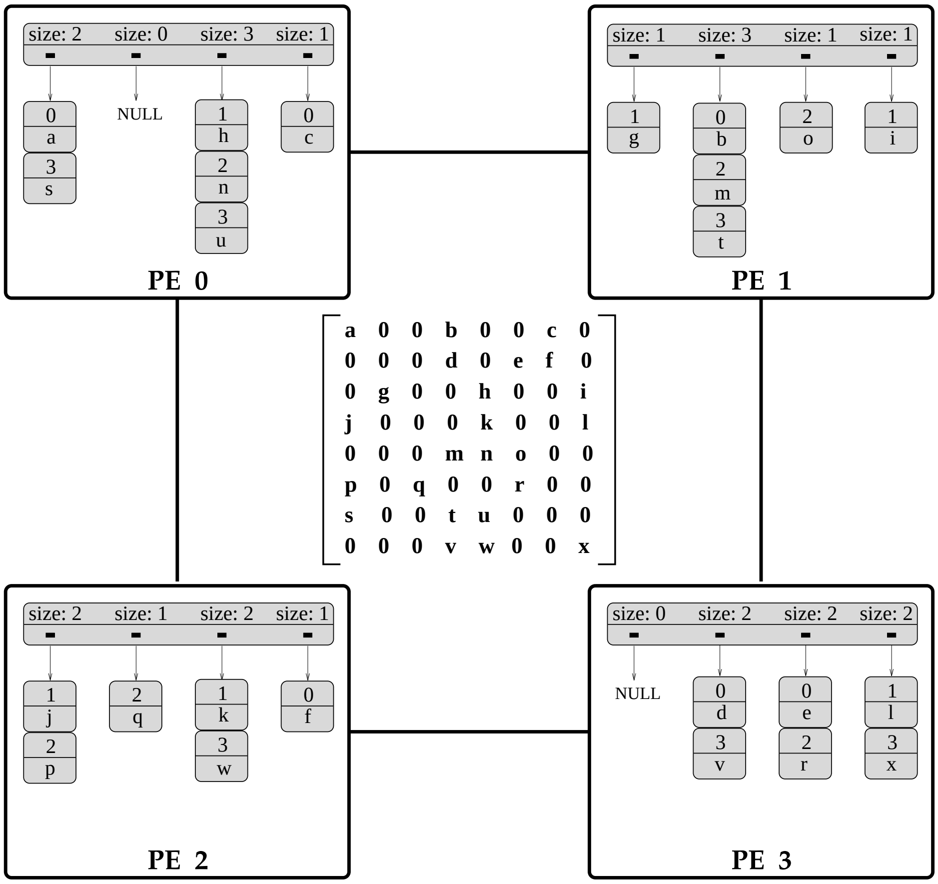

Figure 6: *Sparse matrix data distribution scheme*

of code because of the use of an induction variable *induc* and its increment inside a conditional statement.

4.2 Parallel Algorithms using Message Passing Libraries

The parallel algorithms were implemented on the *Fujitsu* AP1000[6] MIMD distributed memory computer, which has SPARC processors connected by a 2D torus topology. The algorithms were written in C code plus the own AP1000 libraries for message passing. Entries of the matrix are distributed on the PEs in a cyclic way (per rows and per columns) onto a mesh, as if the matrix were dense, but only the nonzero elements are stored using the data structure shown in figure 3. This distribution provides two features: data balancing, and load balancing for those cases where index space is progressively reduced through iterations, such as our QR algorithms. An example for an 8×8 matrix distributed on a 2×2 mesh is displayed in figure 6.

Let us consider the MGS sequential algorithm (see Section 1) to see how it can be executed in parallel. First, each PE obtains the local norms corresponding to the column segments it contains. By means of a reduction

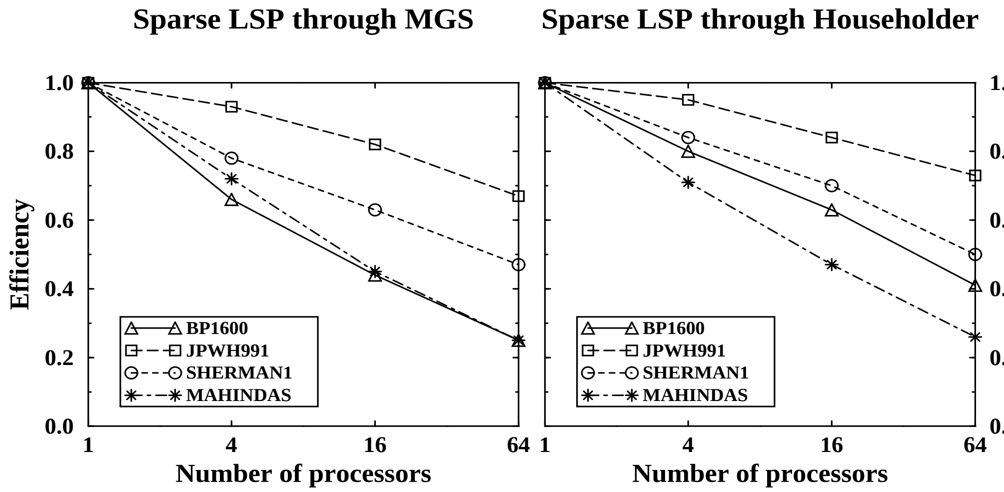

Figure 7: *Efficiencies of the parallel algorithms*

instruction (sum by columns), the vector *norm* of each column of PEs contains the norms of the corresponding global columns (1). Then, the local maximum norm of each PE is calculated (this value is the same for each column of PEs); the global maximum is obtained by means of a reduction instruction which finds the maximum norm by rows of PEs. As a result, the pivot element, as well as p (the index of the pivot column), is stored in every PE (2). If columns k and p are located in different PEs, we use a buffer for exchanging data, in order to perform the pivoting described in (4). The corresponding column of matrices Q and R, as well as the square of the norm, are placed in consecutive memory positions and sent in a single message. Regarding the MGS procedure, the current column k of matrix Q is normalized by dividing it by its norm (6). This column is stored in a packed vector called *vcol*, which is distributed and replicated on each column of PEs by means of a broadcast instruction. For each column with a global index greater than k, the vector called *vsum* (not packed) stores the local dot product of this column and the normalized current column. Finally, the global dot product for each column is obtained in *vsum* by means of a reduction instruction. The elements of *vsum* will be the row k of matrix R (7), from column k up to the last one, since R is upper triangular. Then, the corresponding elements a_{ij} of Q are updated: $a_{ij} = a_{ij} - vcol_i \cdot vsum_j$ (9). Note that this is a local operation and communications are not required. When this operation is carried out, if the original $a_{ij} = 0$ and $vcol_i, vsum_j \neq 0$, then a_{ij} becomes a nonzero element (fill-in). Norm updating (8) is also local.

With respect to Householder, the current column is divided by a given value in order to get vector v (10); thus, only the column of PEs that stores the current column is working. As v contains many null elements and is broadcast to the corresponding PEs on the X axis, it is stored in a packed vector. Once this previous step is completed, there is a copy of the distributed vector v on each column of PEs, and the value β is obtained in every PE (11). Then, vector $w = \beta S^T v$ (12) is calculated, from element w_k up to w_{n-1}, so that w is distributed and replicated on each row of PEs. Our data distribution scheme is adequate to follow an optimal path through the submatrix S distributed on the PEs. The strategy consists of traversing the columns corresponding to this submatrix, starting from the last column. The process ends when is reached a column whose global index is less than k; within each column, we traverse the corresponding packed vector starting from the end and continuing until we reach a row whose global index is less than k. It is not necessary to transpose submatrix S due to the distribution of the data. Submatrix S is updated as $a_{ij} = a_{ij} + v_i \cdot w_j$ (13), being, as in MGS algorithm, a local operation where fill-in may show up.

In conclusion, figure 7 shows the efficiencies of these algorithms for the sparse matrices described in table 1. As we can see, high efficiencies are obtained just selecting a suitable data structure and distribution, which are not available in data parallel languages.

Acknowledgments

We gratefully thank Xunta de Galicia and Ministry of Education and Science of Spain for financing this work under projects XUGA20605B96 and CICYT TIC96-1125-C03, respectively; we also thank the following institutions for giving us access to their computers: Centro de Supercomputación de Galicia, Santiago de Compostela, Spain (Fujitsu VP2400/10); Imperial College/Fujitsu Parallel Computing Research Centre, London (Fujitsu AP1000); Ecole Polytechnique Federale de Laussane, Switzerland (Cray T3D).

Keywords: Sparse matrices, least squares, QR factorization, vectorization, parallel algorithms.

References

1. Barret, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. & van der Vost, H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Siam Press, 1994.

2. Chapman, B., Mehrotra, P. & Zima, H. Programming in Vienna Fortran, *Scientific Programming*, 1992, **1**, 31-50.

3. Duff, I.S., Erisman, A.M. & Reid, J.K. *Direct Methods for Sparse Matrices*, Clarendon Press, 1986.

4. George, J.A., Heath, M.T. & Ng, E.G.Y. A Comparison of some Methods for Solving Sparse Linear Least Squares Problems, *SIAM J. Sci. Stat. Computing*, 1983, **4**, 177-187.

5. Golub, G.H. & van Loan, C.F. Orthogonalization and Least Squares, Chapter 5, *Matrix Computations*, pp. 193-259, The Johns Hopkins University Press, 1989.

6. Ishihata, H., Horie, T. & Shimizu, T. Architecture for the AP1000 Highly Parallel Computer, *Fujitsu Sci. Tech. J.*, 1993, **29(1)**, 6-14.

7. Meier, U., Skinner, G. & Gunnels, J. *A Collection of Codes for Sparse Matrix Computations*, CSRD Report No.1134, University of Illinois at Urbana-Champaign, Center for Supercomputing Res.& Dev., 1991.

8. Merlin, J. & Hey, A. An Introduction to High Performance Fortran, *Scientific Programming*, 1995, **4**, 87-113.

9. Pase, D.M., MacDonald, T. & Meltzer, A. The CRAFT Fortran Programming Model, *Scientific Programming*, 1994, **3**, 227-253.

10. Saad, Y. *SPARSKIT: a Basic Tool Kit for Sparse Matrix Computation*, CSRD Report No.1029, University of Illinois at Urbana-Champaign, Center for Supercomputing Res.& Dev., 1990.

11. Touriño, J. & Doallo, R. An Approach to Decrease Fill-in in Sparse Orthogonalizations on a MIMD Computer, pp. S-B-1 to S-B-5, *Proceedings of the Sixth Parallel Computing Workshop, PCW'96*, Kawasaki, Japan, 1996, Fujitsu Laboratories Ltd., 1996.

12. Touriño, J., Doallo, R., Asenjo, R., Plata, 0. & Zapata, E.L. Analyzing Data Structures for Parallel Sparse Direct Solvers: Pivoting and Fill-In, pp. 151 to 168, *Proceedings of the Sixth Workshop on Compilers for Parallel Computers, CPC'96*, Special issue of the volume Konferenzen des Forschungszentrums Jülich, Vol.21, Aachen, Germany, 1996.