Sparse matrix multiplication on vector computers

Jocelyne Erhel

INRIA

UNITE DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1101

Programme 2
Structures Nouvelles d'Ordinateurs

SPARSE MATRIX MULTIPLICATION ON VECTOR COMPUTERS

Jocelyne ERHEL

Octobre 1989

RR-1101

Sparse matrix multiplication

on vector computers

Jocelyne ERHEL

Publication Interne n° 492
Septembre 1989

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

SPARSE MATRIX MULTIPLICATION ON VECTOR COMPUTERS

MULTIPLICATION DE MATRICES CREUSES SUR CALCULATEUR VECTORIEL [1]

Publication Interne n° 492 - 20 Pages

Jocelyne ERHEL
IRISA - INRIA

September 7, 1989

Abstract

An important kernel of scientific software is the multiplication of a sparse matrix by a vector. The efficiency of the algorithm on a vector computer depends on the storage scheme. With a storage by rows, performances are limited in general by the small vector length. Therefore a storage by so-called generalized columns has been designed, which provides long vectors and consequently good performances. However, it is not adapted to the symmetric case. A new type of storage, by sparse diagonals, has then been defined. It still exhibits long vectors, with performances as good as previoulsy, but it is also well-suited to symmetric matrices. Results on a CRAY2, with various sparse matrices, compare the three algorithms, and show the efficiency of the storage by sparse diagonals.

Résumé

Un noyau important des logiciels scientifiques est la multiplication d'une matrice creuse par un vecteur. L'efficacité de l'algorithme sur calculateur vectoriel dépend du mode de stockage. Avec un stockage par lignes, les performances sont limitées en général par la faible longueur des vecteurs. C'est pourquoi nous avons conçu un stockage par colonnes généralisées, qui donne de longs vecteurs, et par suite de bonnes performances. Cependant, il n'est pas adapté au cas symétrique. Nous avons alors défini un nouveau type de stockage, par diagonales creuses. On obtient encore de longs vecteurs, avec des performances aussi bonnes que précédemment, mais le stockage convient de plus pour le cas symétrique. Des résultats sur un CRAY2, avec diverses matrices creuses, comparent les trois algorithmes, et montrent l'efficacité du stockage par diagonales creuses.

[1]Article soumis pour publication au International Journal of HIGH SPEED COMPUTING

1 Introduction.

Many scientific applications require computations on large sparse matrices. Among them, the multiplication of the matrix by a vector represents an important kernel, which has to be optimized on vector computers to get efficient codes. This paper is focused on the design of a sparse storage and a corresponding algorithm to perform this matrix-vector multiplication. We deal with general sparse matrices, with a priori no specific pattern. Only the non-zero entries are stored, in order to save memory space, and to minimize the number of operations. Furthermore, for symmetric matrices, only the lower triangular part is stored.

We first investigate the classical *storage by rows*, associated to sparse dot-products. It requires rows with a large degree (we call degree of a row its number of non-zero entries) to become efficient on vector computers. For most sparse matrices, it yields poor performances, leading us to find other storage schemes.

For matrices with roughly the same degree per row, a compressed storage can be used to design an efficient vector algorithm ([1]). But it introduces fill-in, which becomes too important for general structures. Hence, we use a more general storage scheme, called *storage by generalized columns*, similar to the *stripe structure* ([2]), or to the *jagged diagonals* ([3]), and yielding good performances. However, this storage is not adapted to symmetric matrices.

For regular or band structures, the storage and multiplication by diagonals have been proved to be very performant on vector computers ([4]). For general sparse matrices, we define a *storage by sparse diagonals*, where only non-zero entries of the diagonals are stored. The corresponding algorithm is well-suited for symmetric matrices, and is quite efficient.

Our algorithms are executed on a CRAY2 computer[1]. Pipelined sparse operations (scatter and gather) are available, though the gather operation is quite slow (about three times slower than the scatter). To measure the performances of our algorithms, we use the *asymptotic speed* r_∞, and the *half performance length* $n_{1/2}$ ([5]). The speed of execution r for a vector length n can then be expressed by the formula:

$$r = r_\infty * n/(n + n_{1/2})$$

We have tested also the algorithms on various sparse matrices, coming from the Harwell collection ([6]). Results are given for the three storage schemes, in order to allow comparisons.

[1]Acces to the CRAY2 computer is provided by the Centre de Calcul Vectoriel de la Recherche, Palaiseau, France.

2 Position of the problem.

2.1 Non symmetric case.

Let $A \in R^{n \times m}$ be a sparse matrix, the entries of which are (α_{ij}). In order to save memory, only non-zero entries are stored. The set of indices corresponding to non-zero entries is denoted by I, and its cardinal by N.

$$\alpha_{ij} \neq 0 \Rightarrow (i,j) \in I$$

Let ν be a numbering of the non-zero entries of A, that is a one-to-one mapping from I into the set of integers $J = [1, N]$. The reciprocal mappings of ν are denoted by λ and γ, and are defined by:

$$i = \lambda(l) \; and \; j = \gamma(l) \Leftrightarrow \nu(i,j) = l$$

The non-zero entries of A are stored in the array $b = (\beta_l)_{l=1,N}$, where:

$$\nu(i,j) = l \Rightarrow \beta_l = \alpha_{ij}$$

From now on, x and y are vectors of order m and n respectively. Our goal is to design efficient storage and algorithm to perform the multiplication:

$$y := y + A * x \tag{1}$$

Operation 1 is translated into the following loop:

$$\begin{aligned} for \; l \; &= \; 1, N \\ &\quad y_{\lambda(l)} := y_{\lambda(l)} + \beta_l * x_{\gamma(l)} \\ end & \end{aligned} \tag{2}$$

We do not consider roundoff errors, so that the order of iterations is not relevant. It means that we can choose any numbering ν. Instruction 2 is vectorial if and only if λ is either constant (dot-product) or injective (vector triad). We are looking for partitions of J, $J = \bigcup J_k$, such that λ is constant or injective on each J_k.

2.2 Symmetric case.

Let A be a symmetric sparse matrix of order n. Only the lower triangular part L of A is stored, saving memory space. The main diagonal is stored in a separate array. Otherwise, notations are the same as before, applied to L. In particular, N is the number of non-zero entries in L.

Operation 1 is performed by the following loop:

$$\begin{aligned} for\ l &= 1, N \\ & \quad y_{\lambda(l)} := y_{\lambda(l)} + \beta_l * x_{\gamma(l)} \qquad (3) \\ & \quad y_{\gamma(l)} := y_{\gamma(l)} + \beta_l * x_{\lambda(l)} \qquad (4) \\ end & \end{aligned}$$

Operations 3 and 4 are grouped into the same loop to minimize the memory requirements.

Instruction 3 (resp. 4) is a vector one if and only if λ (resp. γ) is either constant or injective. We are then looking for partitions $J = \bigcup J_k$ such that on each J_k is satisfied one of the following:

- λ is constant and γ is injective (or the symmetric case),
- λ and γ are both injective.

The case where both functions are constant has obviously no interest!

3 Storage by rows.

The matrix A is stored by rows, with any numbering within each row. This storage defines a partition $(J_i)_{i=1,n}$, such that λ is constant and γ is injective on each J_i. This partition is therefore adapted to non symmetric and symmetric cases as well.

3.1 Non symmetric case.

The algorithm 2 is then rewritten as:

$$\begin{aligned} for\ i &= 1, n \\ for\ l &= 1, d_i \\ & \quad y_i := y_i + \beta_{l+t_i} * x_{\gamma(l+t_i)} \qquad (5) \\ end & \\ end & \end{aligned}$$

where d_i is the degree of row i, and t_i is defined by:

$$\begin{aligned} t_1 &= 0 \\ t_{i+1} &= t_i + d_i, \quad i = 1, n-1 \end{aligned}$$

Instruction 5 is then a sparse dot-product. The performances on CRAY2 are given by (figure 1):

$$\begin{aligned} r_\infty &= 55\ MFLOPS \\ n_{1/2} &= 150 \end{aligned}$$

Although the asymptotic speed is quite high, it requires a large vector length, greater than 300. The vector length is defined by the degree of the rows. But in most of sparse matrices, it is quite small and the average lies between 10 and 20 ([7]). Expected performances lie then in the range of 10 MFLOPS. Therefore, this approach is not efficient on the CRAY2, for general sparse matrices.

3.2 Symmetric case.

The symmetric case is similar to the previous one, except that only L is stored, and that the product by L and L^t are performed in the same loop. Instructions 3 and 4 are then rewritten as:

$$\begin{aligned} &for\ i = 1, n \\ &for\ l = 1, d_i \\ &\qquad y_i := y_i + \beta_{l+t_i} * x_{\gamma(l+t_i)} \\ &\qquad y_{\gamma(l+t_i)} := y_{\gamma(l+t_i)} + \beta_{l+t_i} * x_i \\ &end \\ &end \end{aligned} \tag{6}$$

The loop 6 is composed of a sparse dot-product and a sparse vector triad. the performances on CRAY2 are given by (figure 2):

$$\begin{aligned} r_\infty &= 60\ MFLOPS \\ n_{1/2} &= 40 \end{aligned}$$

The same conclusion applies in the symmetric case. In general, the degree of the rows are too small to get an efficient algorithm on the CRAY2.

4 Storage by generalized columns.

The poor performances of the storage by rows has led us to define a new type of storage, searching for long vectors. We want to find a partition such that, for non symmetric matrices, λ is injective on each subset of the partition, and such that the size of each subset is as large as possible (it will be the vector length).

The minimal number of subsets is the maximal degree of a row, say d. Conversely, the maximal size of a subset is the number of rows n. Therefore, we define I_k as the set of the k^{th} non-zero entries of all the rows. We then get d subsets, with most of them of size n. It should be noted that this numbering corresponds to the storage by columns for dense matrices. Therefore will call it the *generalized column storage* ([8]).

This storage scheme can still be improved by suppressing the indirection λ on the rows. Rows are renumbered by decreasing degree, such that each subset I_k contains rows from 1 to n_k, where n_k is the number of rows of degree at least k. However, the inverse permutation of rows must be performed on the resulting vector y.

The algorithm 2 is then rewritten as:

$$\begin{aligned} &for\ k &&= 1, d \\ &\quad for\ i &&= 1, n_k \\ &\qquad && y_i := y_i + \beta_{i+t_k} * x_{\gamma(i+t_k)} \\ &\quad end \\ &end \end{aligned} \tag{7}$$

Performances can be improved by unrolling the loop on k, if several subsets have the same size, or by adding some zero-entries. On the CRAY2, the measures give the following results (figure 1):

$$\begin{aligned} r_\infty &= 42\ MFLOPS \\ n_{1/2} &= 30 \end{aligned}$$

For most sparse matrices, almost all subsets have a size of n or slightly less than n, yielding long vectors, and consequently good performances on CRAY2. The vector length is greater than 300, allowing to obtain the asymptotic speed. However, memory conflicts tend to decrease the speed of computation. They could be controlled by choosing the ordering of the non-zero entries within each row.

It must be pointed out that this algorithm is adapted to any rectangular sparse matrix. But it cannot be extended to the symmetric case, where only L is stored.

5 Storage by sparse diagonals.

We are now dealing with the symmetric case. We must find a partition $J = \bigcup J_k$, such that both λ and γ are injective, and such that the size of each J_k is large enough.

A solution could be to use a general graph coloring algorithm ([9]), or to define so-called stripes ([2]). But both methods implies the storage of both indirections.

To avoid this, the best solution would be to request a linear mapping for λ. However, it does not seem compatible with the storage of only L. Therefore, we look for a linear relation between both indirections λ and γ. A natural choice, well-suited to the symmetric case, is to store A by diagonals. The relation $\lambda - \gamma = k$, corresponds to the diagonal numbered k. Of course, only non-zero entries of the diagonals are stored, in association with γ. The algorithm 3, 4 can be rewritten as:

$$
\begin{array}{lcl}
for\ k & = & 1, m \\
for\ l & = & 1, n_k \\
 & & y_{\gamma(l+t_k)+k} := y_{\gamma(l+t_k)+k} + \beta_{l+t_k} * x_{\gamma(l+t_k)} \\
 & & y_{\gamma(l+t_k)} := y_{\gamma(l+t_k)} + \beta_{l+t_k} * x_{\gamma(l+t_k)+k} \\
end & & \\
end & &
\end{array}
\qquad (8)
$$

where:

- m is the bandwidth of the matrix,
- n_k is the number of non-zero entries in diagonal k, (possibly $n_k = 0$),
- t_k is given by: $t_1 = 0$, and $t_{k+1} = t_k + n_k$.

Instructions are sparse vector triads. Memory requirements are limited by using λ and β in the same loop.

This algorithm can also be applied to square non symmetric matrices, giving a similar loop, but with only the first instruction, and with negative diagonal numbers. However, it is not appropriate for rectangular matrices.

For the non symmetric case, measured performances on the CRAY2 are the following (figure 1):

$$
\begin{array}{rcl}
r_\infty & = & 28\ MFLOPS \\
n_{1/2} & = & 20
\end{array}
$$

For the symmetric case, we get better performances, due to a better ratio between memory transfers and floating-point operations (figure 2):

$$
\begin{array}{rcl}
r_\infty & = & 39\ MFLOPS \\
n_{1/2} & = & 15
\end{array}
$$

To be efficient, the storage must provide sufficiently long vectors, of length at least 40. In other words, diagonals must have enough non-zero entries. For general sparse matrices, diagonals can be scattered and very small. Hence a renumbering technique appears to be necessary, by applying a permutation matrix P, yielding the matrix P^tAP. Algorithms which minimize the bandwidth are quite efficient in practice, since reducing the number of diagonals increases their average length. Furthermore, these renumbering algorithms are often used in applications to solve the sparse linear system.

6 Numerical experiments.

6.1 Storage requirements.

We now compare the three storage schemes and the corresponding algorithms. First of all, it should be noted that they require roughly the same memory space.

In the non symmetric case, the three storage schemes need N real words to store the matrix β, and N integer words to store the column index γ. Additional pointers are necessary in each case to find the length of the rows, or the generalized columns, or the diagonals. The corresponding arrays are summarized below:

- storage by rows: $d_i \quad 1 \leq i \leq n$,
- storage by generalized columns: $n_k \quad 1 \leq k \leq d$, and the permutation of rows, $iperm_i \quad 1 \leq i \leq n$,
- storage by diagonals: $n_k \quad -m_1 \leq k \leq m_2$.

In the symmetric case, the main diagonal is stored in a separate array, so that both schemes by rows and by diagonals require $(N+n)$ real words to store the non-zero entries β, and N integer words to store the column index γ. The additional pointers are the same as previously and are summarized below:

- storage by rows: $d_i \quad 1 \leq i \leq n$,
- storage by diagonals: $n_k \quad 1 \leq k \leq m$.

Of course, all algorithms execute the same number of operations, which is $2*N$ (resp. $4*N+2*n$) in the non symmetric (resp. symmetric) case. Hence, we can use the rate of execution as a measure of comparison.

6.2 Test matrices.

We have tested the three methods on matrices arising from the Harwell collection. Concerning the symmetric matrices, we have chosen four matrices coming from structure problems, and four matrices coming from various applications. We have picked also three non symmetric matrices. The characteristics of all matrices are described in table 1. Two symmetric matrices (BCSSTK19 and BCSPWR09) before and after renumbering are depicted in figures 3 to 6. The non-zero entries are represented by points. The renumbering algorithm, provided by [10], is designed to minimize the bandwidth, and is derived from the original algorithm due to [11].

Symmetric matrices have been stored by rows and diagonals, with only the lower triangular part L, but also by generalized columns, with the non symmetric storage for the sake of comparison. Non symmetric matrices have been stored by rows, diagonals, and generalized columns.

The timings for the generalized columns algorithm include the permutation of the resulting vector y, to get the correct numbering of the rows.

6.3 Results on CRAY2.

Measures are done on a CRAY2, in dedicated mode. Results, in terms of rate of execution in MFLOPS, are given in table 2. Codes are written in all FORTRAN. They could be slightly improved, by writing them in cal, but we prefer to keep portability.

For most matrices, the storage by rows gives poor performances, less than 10 MFLOPS. For example, we get 1.7 MFLOPS for the matrix BCSPWR09. However, the results are fairly good for three matrices, namely BCSSTK24, ORANI678, and PSMIGR 1, with respectively 17, 13, and 29 MFLOPS. These three matrices have a large number of non-zero entries N, thus a large mean degree per row, so that the vector length is sufficiently long. But in other cases, the degree of the rows are too small, so that the algorithm is slow. Conclusions are identical in the symmetric and the non symmetric cases.

The storage by generalized columns give in general good performances, ranging from 14 to 34 MFLOPS, for example 21 MFLOPS for the matrix BCSPWR09. However, two matrices (BCSSTK24 and ORANI678) give rise to poor results, less than 10 MFLOPS. This is due to memory conflicts. The column index γ is not injective, so that different rows may require the same index, slowind down the indirect access to the vector x. Although this is true for all matrices, it appears important only in these two cases. A thorough analysis of the column index γ shows that, for these two matrices, γ is very often the same for consecutive rows, sometimes even the identity for one generalized column. Conversely, for matrices PLAT1919 and LNS 3937, the column index γ is almost injective, so that few memory conflicts appear, explaining the high rate of execution (33 and 34 MFLOPS).

The results for the storage by diagonals, without renumbering, are comparable to those for the storage by generalized columns, except for two matrices, 1138 BUS (5 MFLOPS), and BCSSWR09 (4 MFLOPS). As can be seen in figure 5, the diagonals are scattered in the matrix, and have very few non-zero entries, providing very small vectors. A lot of diagonals have only one non-zero entry. Therefore, the rate of execution is very slow.

But after renumbering, the performances are much better. The figure 6 shows that now the diagonals are much longer, so that the vector length is sufficient to get a fast rate of execution. In general, for all tested matrices, the renumbering algorithm improves the results. After renumbering, the performances are very often better for the storage by diagonals than for the storage by generalized columns. Furthermore, it is more well-suited to the symmetric case, because it requires only the storage of the lower triangular part of the matrix. There are few memory conflicts, because the row index and the column index are both injective. Memory bank conflicts may still appear if the indices are equal modulo the number of memory banks B, but this happens for very sparse diagonals, with non-zero entries every B rows for example, which is not the case in practice.

7 Conclusion.

To perform efficiently a sparse-matrix-vector multiplication on vector computers, it is necessary to define a storage giving long vectors and minimizing memory requirements. The classical storage by rows yields for most sparse matrices too small vectors. For non symmetric matrices, the storage by generalized columns gives in general good performances, though it may be slowed down by memory conflicts. But for square and symmetric matrices, the storage by sparse diagonals becomes very competitive, for two reasons. First of all, the algorithm is in general faster than the two others. Secondly, it allows to store only half of the matrix, saving memory space. However, it requires sometimes a renumbering of rows and columns, but which is often used in practice for other purposes.

The algorithm by diagonals could be improved by allowing some fill-in, to obtain full diagonals, leaving only a few sparse diagonals. This scheme would eliminate indirections for most of the computations.

Other kernels than this sparse-matrix-vector multiplication are also crucial in scientific applications. In particular, the resolution of a sparse triangular system is very important. Our effort will now be focused on this operation.

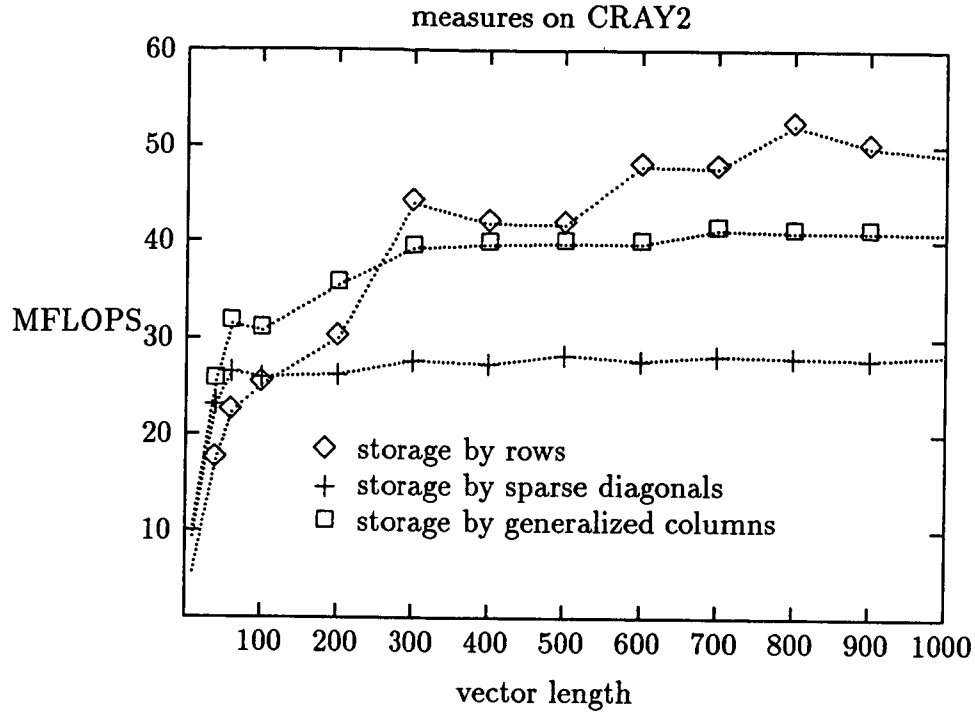

Figure 1: non symmetric case: rate of instruction 2

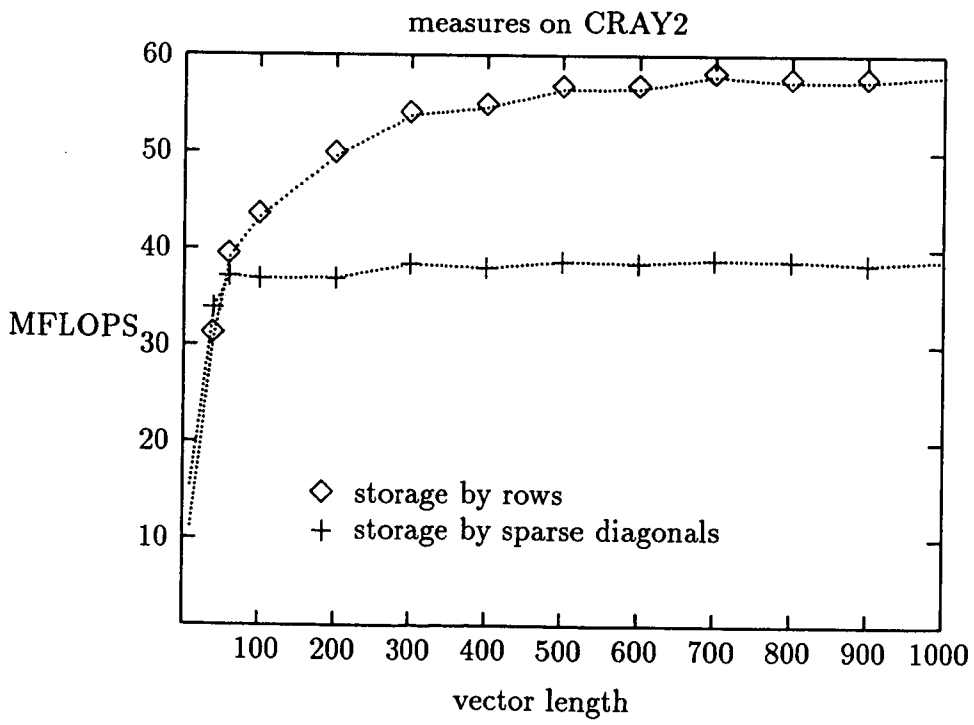

Figure 2: symmetric case: rate of instructions 3 and 4

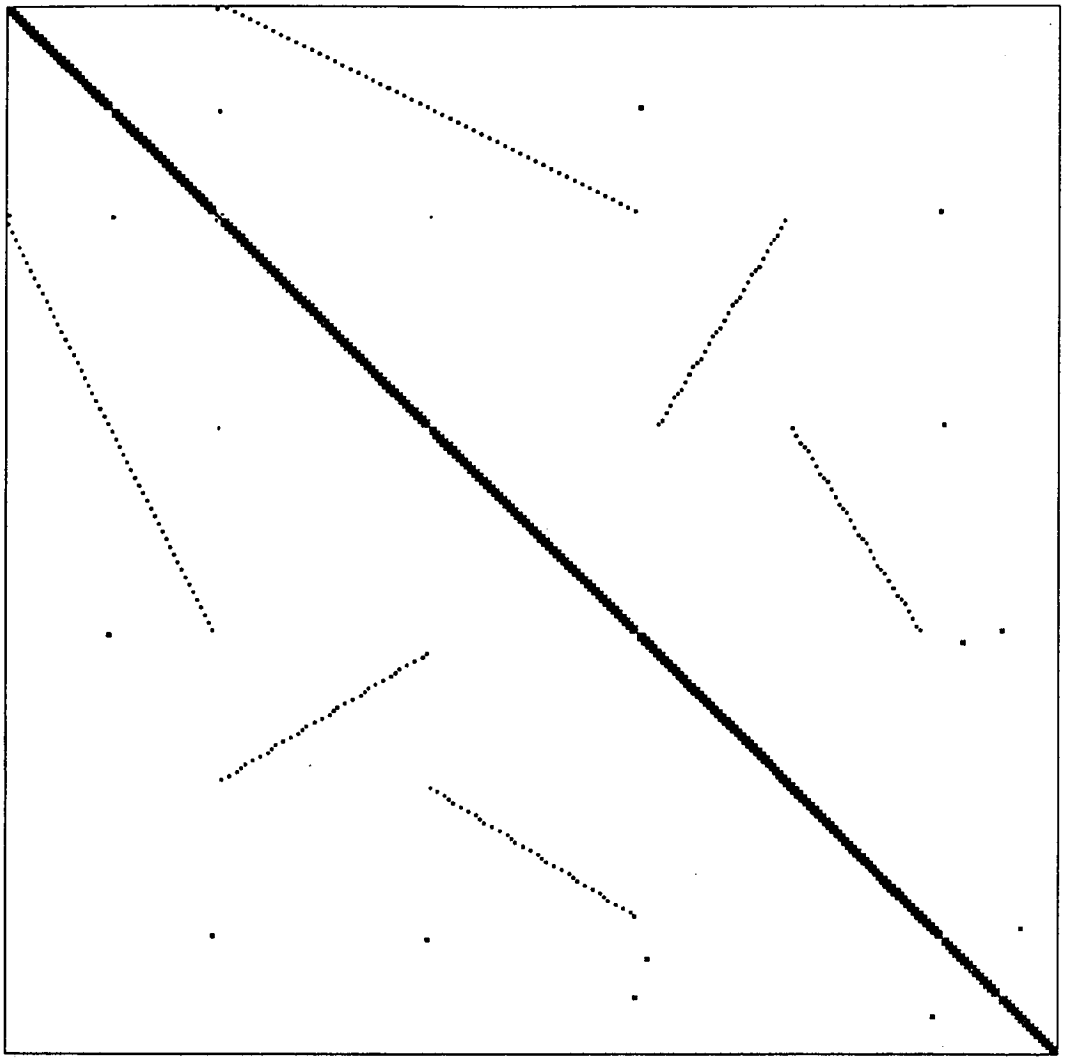

Figure 3: Matrix BCSSTK19 before renumbering

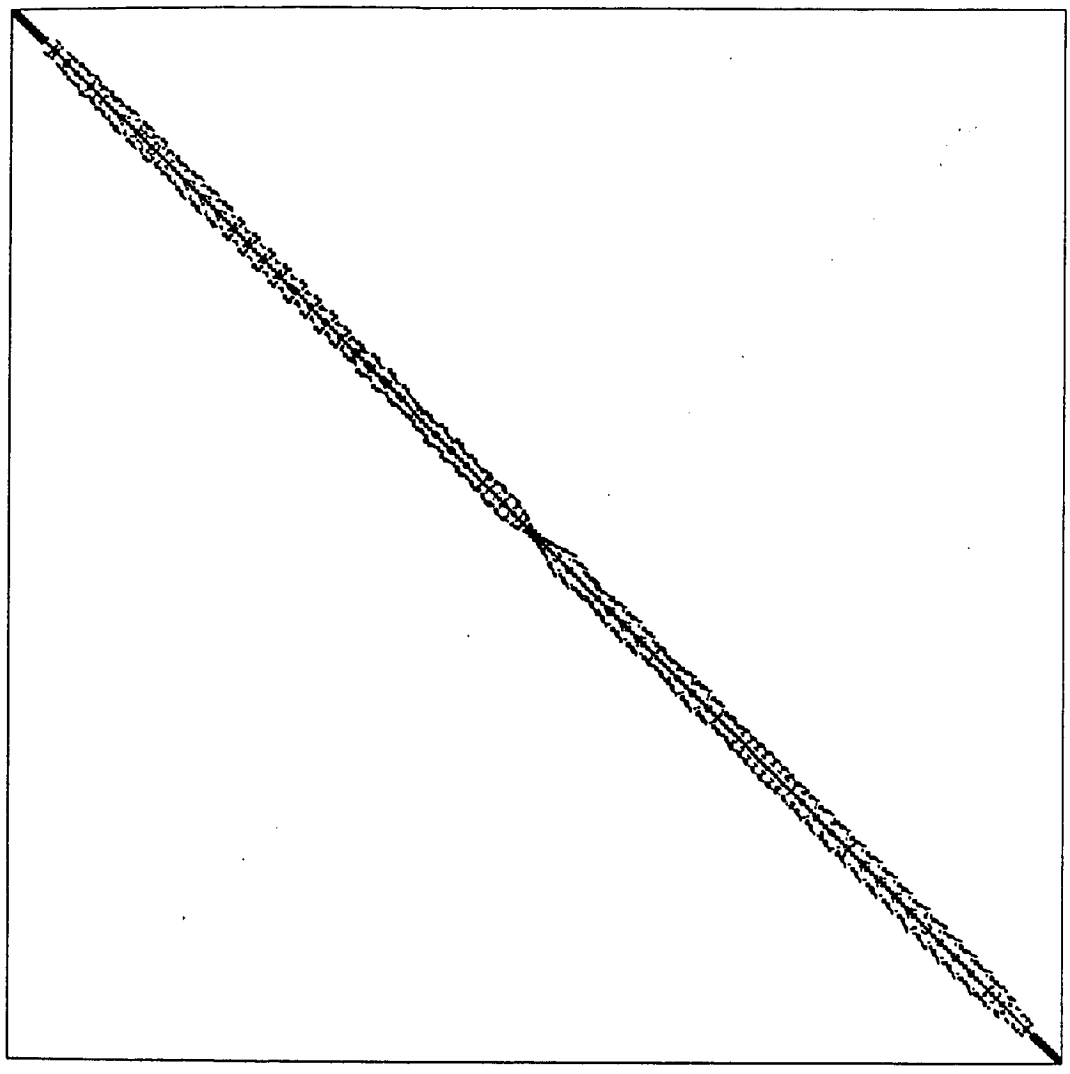

Figure 4: Matrix BCSSTK19 after renumbering

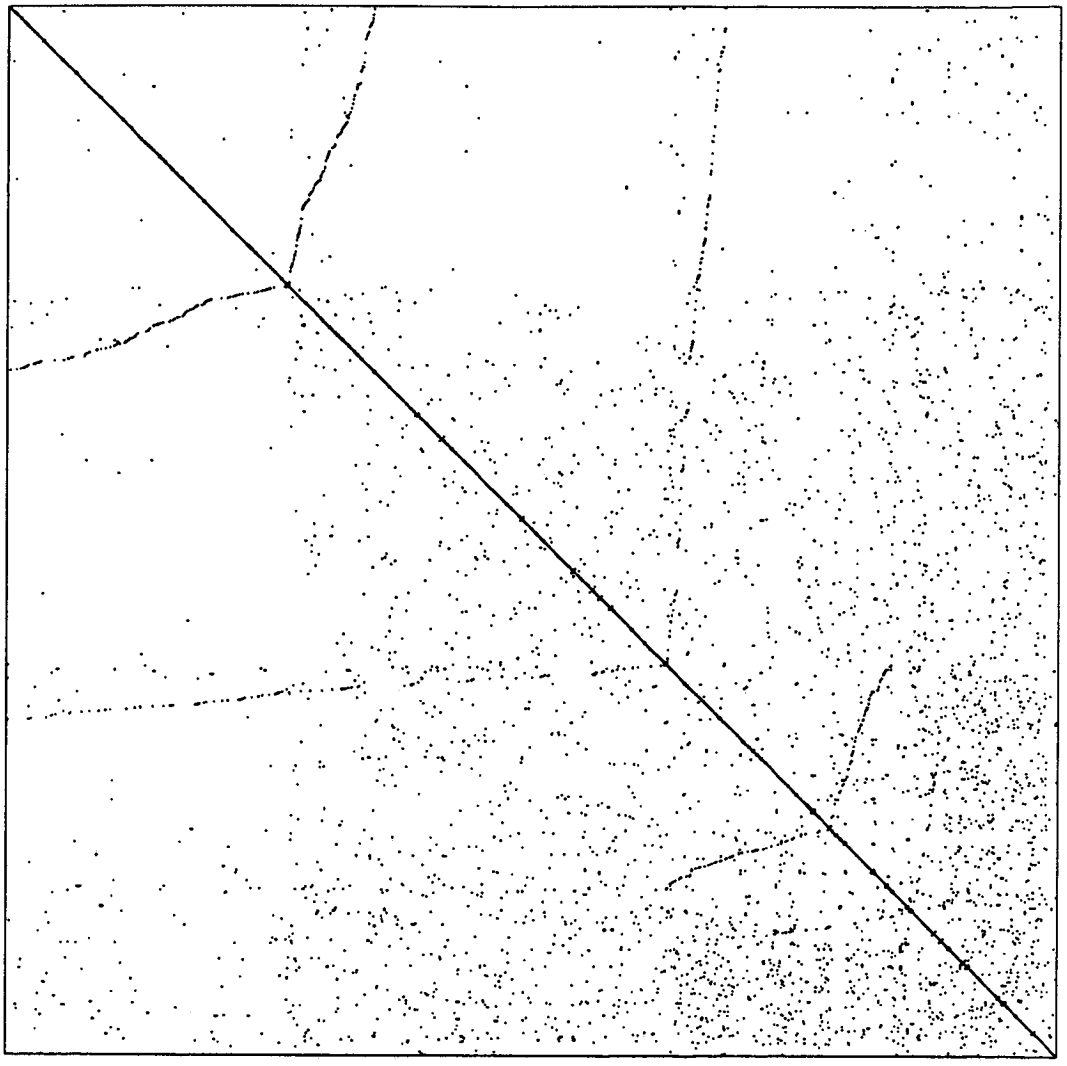

Figure 5: Matrix BCSPWR09 before renumbering

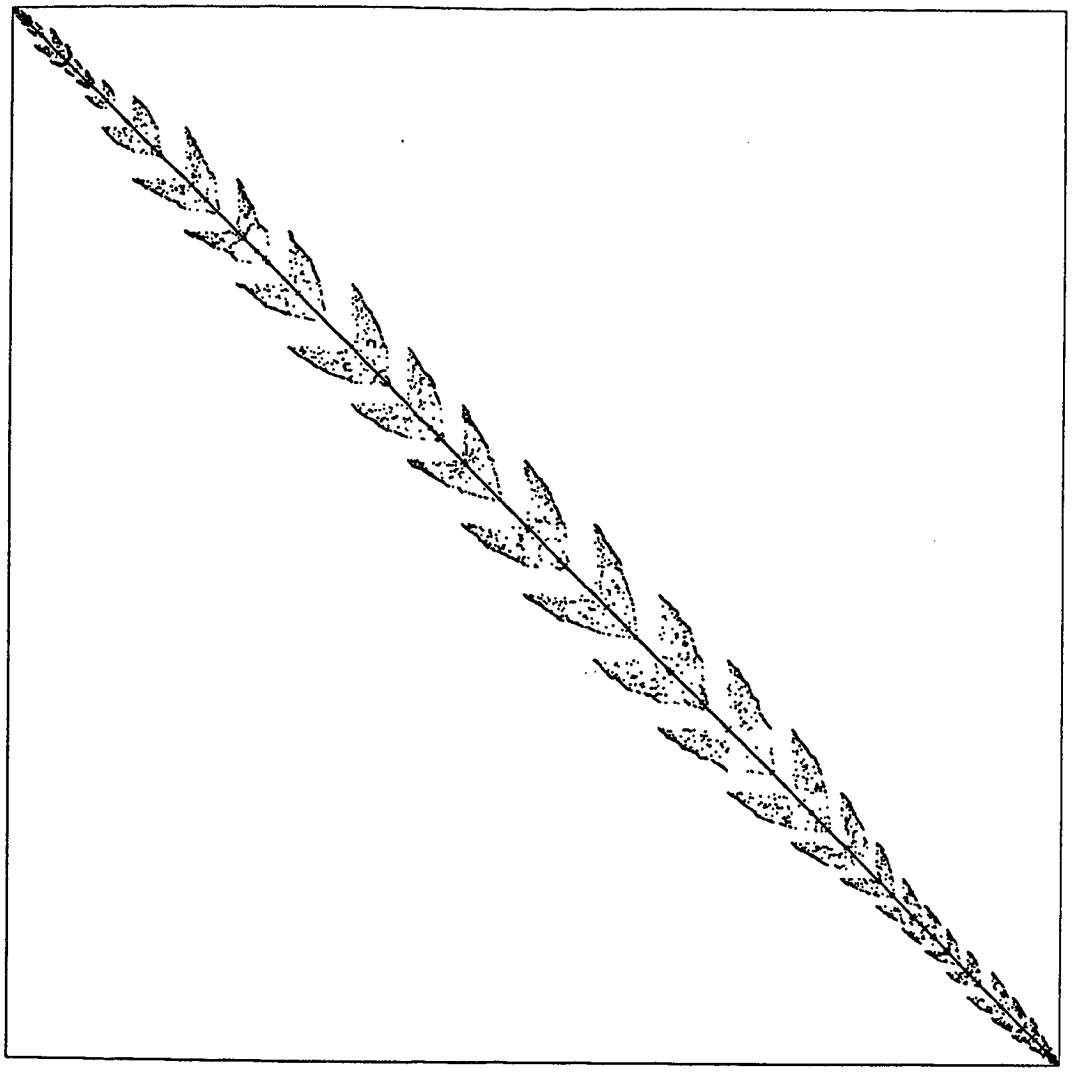

Figure 6: Matrix BCSPWR09 after renumbering

matrix name	order	number of entries	maximal degree	bandwidth before renumbering	bandwidth after renumbering
Symmetric matrices					
BCSSTK19	817	3835	11	567	18
BCSSTK12	1473	17857	33	650	62
BCSSTK23	3134	24156	31	449	350
BCSSTK24	3562	81736	57	3333	293
1138 BUS	1138	2596	18	1030	126
BCSPWR09	1723	4117	15	1663	116
PLAT1919	1919	17159	19	1297	80
ZENIOS	2873	15032	47	1844	30
Non symmetric matrices					
ORANI678	2529	90158	1110	2309	
LNS 3937	3937	25407	11	3202	
PSMIGR 1	3140	543162	2294	3124	

Table 1: characteristics of matrices

matrix name	by rows	by generalized columns	by diagonals before renumbering	by diagonals after renumbering
Symmetric matrices				
BCSSTK19	4.0	20.5	12.0	31.2
BCSSTK12	10.2	14.5	26.7	32.3
BCSSTK23	6.6	19.4	26.2	27.1
BCSSTK24	17.2	9.9	22.0	31.6
1138 BUS	2.0	20.7	5.6	17.0
BCSPWR09	1.7	21.4	4.2	20.0
PLAT1919	6.6	33.1	21.8	27.7
ZENIOS	6.1	22.8	12.4	35.8
Non symmetric matrices				
ORANI678	13.0	8.8	14.2	
LNS 3937	3.1	34.0	18.3	
PSMIGR 1	29.3	27.7	19.4	

Table 2: Sparse matrix-vector multiplication - MFLOPS on CRAY2

References

[1] D.R. Kincaid, T.C. Oppe, D.M. Young, *Vector Computations for Sparse Linear Systems*, SIAM J. Alg. Disc. Meth., Vol. 7, No 1, pp. 99-112, Jan., 1986.

[2] R. Melhem, *Parallel Solution of Linear Systems with Striped Sparse Matrices*, Parallel Computing 6 , pp. 165-184, 1988.

[3] Y. Saad, *Parallel Iterative Methods for Sparse Linear and Nonlinear Equations*, Finite Element Analysis in Fluids, T.J. Chang, G.R. Karr, Eds, U. of Alabama in Huntsville Press, 1989.

[4] J.R. Bunch, J.J. Dongarra, C.B. Moler, G.W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, 1979.

[5] R.W. Hockney, C.R. Jesshope, *Parallel Computers*, Adam Hilger, Bristol, 1981.

[6] I.S. Duff, R.G. Grimes, J.G. Lewis, *Sparse Matrix Test Problems*, Report CSS 191, Harwell Laboratory, England, 1987.

[7] A.K. Dave, I.S. Duff, *Sparse Matrix Calculations on the CRAY 2*, Parallel Computing, Vol. 5, pp 55-64, 1987.

[8] J. Erhel, B. Philippe, *Multiplication of a Vector by a Sparse Matrix on Supercomputers*, Parallel Processing, M. Cosnard, M. Barton, M. Vanneschi, Eds, North-Holland, 1988.

[9] J. Erhel, *Finite Element Methods on Parallel and Vector Computers*, Applications in Fluid Dynamics, Supercomputing, E. Houstis, T, Papatheodorou, C. Polychronopoulos, Eds, Springer-Verlag, 1987.

[10] H. Crane, N. Gibbs, Jr. Poole, P. Stockmeyer, *Algorithm 508: Matrix bandwidth minimization and profile reduction*, ACM TOMS Vol 2, No 4, pp.375-377, 1976.

[11] E. Cuthill, J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, Proc. 24th Nat. Conf. ACM, Brandon Systems Press, pp.157-172.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 486 **SYNTHESIS OF A NEW SYSTOLIC ARCHITECTURE FOR THE ALGEBRAIC PATH PROBLEM**
Abdelhamid BENAINI, Patrice QUINTON, Yves ROBERT, Yannick SAOUTER, Bernard TOURANCHEAU
34 Pages, Juillet 1989.

PI 487 **PLANS SIMULATION USING TEMPORAL LOGICS**
Eric RUTTEN, Lionel MARCE
40 Pages, Juillet 1989.

PI 488 **ON FINITE LOOPS IN LOGIC PROGRAMMING**
Philippe BESNARD
20 Pages, Septembre 1989.

PI 489 **LTA : UN LANGAGE DE TRAITEMENT D'ARBRES**
Dalila HATTAB
24 Pages, Septembre 1989.

PI 490 **THE SIGNAL SOFTWARE ENVIRONMENT FOR REAL-TIME SYSTEM SPECIFICATION, DESIGN, AND IMPLEMENTATION**
Albert BENVENISTE, Paul LE GUERNIC
34 Pages, Septembre 1989.

PI 491 **PHYSIQUE QUALITATIVE : PRESENTATION ET COMMENTAIRES**
Qinghua ZHANG
48 Pages, Septembre 1989.

PI 492 **SPARSE MATRIX MULTIPLICATION ON VECTOR COMPUTERS**
Jocelyne ERHEL
20 Pages, Septembre 1989.

PI 493 **THE SUPERIMPOSITION OF ESTELLE PROGRAMS : A TOOL FOR THE IMPLEMENTATION OF OBSERVATION AND CONTROL ALGORITHMS**
Benoît CAILLAUD
30 Pages, Septembre 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique