Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Information Processing Letters 90 (2004) 87–92

Information Processing Letters

www.elsevier.com/locate/ipl

An optimal storage format for sparse matrices

Eurípides Montagne *, Anand Ekambaram

University of Central Florida, School of Electrical Engineering and Computer Science, Orlando, FL 32816, USA

Received 30 December 2002

Communicated by F. Dehne

Abstract

The irregular nature of sparse matrix–vector multiplication, $Ax = y$, has led to the development of a variety of compressed storage formats, which are widely used because they do not store any unnecessary elements. One of these methods, the Jagged Diagonal Storage format (JDS) is, in addition, considered appropriate for the implementation of iterative methods on parallel and vector processors. In this work we present the Transpose Jagged Diagonal Storage format (TJDS) which drew inspiration from the Jagged Diagonal Storage scheme but requires less storage space than JDS. We propose an alternative storage scheme which makes no assumptions about the sparsity pattern of the matrix and only needs three linear arrays instead of the four linear arrays required by JDS. Specifically, the data is aligned in such a way that the permutation array used in JDS, to permute the solution vector back to the original ordering, is unnecessary. This allow us to save the memory space required to store an integer vector of length n, where n stands for the number of columns in the sparse matrix A. This storage saving reaches, for the selection of matrices used in this work, from 14% up to 45% of the number of non-zero values of the sparse matrices. We present a case study of a 6×6 sparse matrix to show the data structures and the algorithm to compute $Ax = y$ using the TJDS format.

Keywords: Sparse matrix; Matrix vector product; Data structures

1. Introduction

In many scientific computations the manipulation of sparse matrices is considered the crux of the design. Generally the non-zero elements in a sparse matrix constitutes a very small percentage of data. This irregular nature of sparse matrix problems has led to the development of a variety of compressed storage formats [1,4], which are widely used. Although these formats suffer the drawback of indirect addressing, they have the advantage that they do not store any unnecessary elements [1]. One of these methods, the Jagged Diagonal Storage format (JDS) is, in addition, considered very convenient for the implementation of iterative methods on parallel and vector processors [3].

* Corresponding author.
E-mail address: eurip@cs.ucf.edu (E. Montagne).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2004.01.014

The main idea considered in compressed storage formats is to avoid the handling and storage of zero values. This is done by means of the storage of the non-zero elements of the sparse matrix in a contiguous way using a linear array. However, some additional arrays are needed for determining where the non-zero elements fit into the sparse matrix. The number of subsidiary arrays varies depending on the storage format used.

Research into sparse matrix reorganization has dealt with developing various static storage reduction schemes such as Compressed Row Storage (CRS), Compressed Column Storage (CCS), and Jagged Diagonal Storage (JDS). Departing from the CRS format we explain the JDS scheme in Section 2. In Section 3 we propose the Transpose Jagged Diagonal Storage (TJDS) format. Then in Section 4, we show a comparison of the storage required for a set of matrices using each storage format. Finally, in Section 5 we present our conclusions.

2. The jagged diagonal format

The Jagged Diagonal Storage format is based on the CRS scheme. In the Jagged Diagonal storage scheme, given any Adjacency Matrix A, we shift all the non-zero elements (*nze*) in each row to the left so that all the zeros are on the right, and this way we obtain the matrix A_{crs}, a compressed row version of matrix A. Simultaneously in a different array we note down the original column indices of the non-zero elements. Now as some of the rows may have more non-zero elements than others, we permute the rows of the matrix A_{crs} in decreasing number of non-zero elements they contain to create a A_{jds} matrix, which is the jagged diagonal version of A_{crs}. While performing the permutation, we create a permutation array that stores the original row indices and another array that stores the starting positions of each diagonal.

The matrix vector product $Ax = y$ is expressed in matrix form as follows:

$$\begin{bmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & a_{25} & 0 \\ a_{31} & 0 & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}.$$

By applying the CRS scheme to the sparse matrix A we have,

$$\begin{bmatrix} a_{11} & a_{12} & a_{14} & 0 & 0 & 0 \\ a_{22} & a_{23} & a_{25} & 0 & 0 & 0 \\ a_{31} & a_{33} & a_{34} & 0 & 0 & 0 \\ a_{42} & a_{44} & a_{45} & a_{46} & 0 & 0 \\ a_{55} & a_{56} & 0 & 0 & 0 & 0 \\ a_{65} & a_{66} & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}.$$

Thus matrix A has become A_{crs} wherein the rows are compressed such that the non-zero elements are at the left end. Hence, we obtain A_{jds} a Jagged Diagonal Storage version of A_{crs} by reordering the rows of A_{crs} in decreasing order from top to bottom according to the number of non-zero elements per row. It can be seen that the rows one and four of the matrix A_{crs} were permuted as shown below.

$$\begin{bmatrix} a_{42} & a_{44} & a_{45} & a_{46} & 0 & 0 \\ a_{11} & a_{12} & a_{14} & 0 & 0 & 0 \\ a_{22} & a_{23} & a_{25} & 0 & 0 & 0 \\ a_{31} & a_{33} & a_{34} & 0 & 0 & 0 \\ a_{55} & a_{56} & 0 & 0 & 0 & 0 \\ a_{65} & a_{66} & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}.$$

We now store the A_{jds} matrix as a single dimension array. The non-zero elements of the A_{jds} matrix are stored in a floating-point linear array *value_list*, one column after another. Each one of these columns is called a jagged diagonal (*jd*). The length of the array *value_list* is equal to the number of non-zeros elements in A_{jds}. We also need another array, *column_indexes*, of the same length as *value_list* to store the column indices of the non-zero elements. Since the rows have been permuted to obtain a different matrix, A_{jds}, from the original matrix A, we need an array, *perm_vector*, to permute the resulting vector back to the original ordering. Obviously the size of this array is n for an n-dimensional matrix. A fourth array is also needed, *start_positions*, which stores the starting position of the jagged diagonals in the array *value_list*. The length of this array is equal to the maximum number of non-zero values per row in matrix A. These arrays are shown below:

value_list	a_{42}	a_{11}	a_{22}	a_{31}	a_{55}	$a_{65;}$	a_{44}	a_{12}	a_{23}	a_{33}	a_{56}	$a_{66;}$	a_{45}	a_{14}	a_{25}	$a_{34;}$	a_{46}

column_indexes	2	1	2	1	5	5	4	2	3	3	6	6	5	4	5	4	6

start_positions	1	7	13	17	18

perm_vector	4	2	3	1	5	6

X	x_1	x_2	x_3	x_4	x_5	x_6

In the above data structures, each jagged diagonal is separated by a semi-colon (;) in the array *value_list*. The memory required for the above storage scheme will be:

$$S_{jds} = (N_{nze} * 2) + (N_{nze} * 1) + N + N_{jd}$$

considering that floating point elements will consume 2 words and integer values will consume a single word. In the above mentioned formula,

S_{jds} denotes the storage required for storing the sparse matrix in the JDS format,

$(N_{nze} * 2)$ gives the storage required to store the *nze* of A,

and the subsidiary data required by this storage format is given as follows:

$(N_{nze} * 1)$ denotes the storage required to store the array indices of the non-zero elements,

N_{nze} denotes the number of non-zeros in the sparse matrix,

N denotes the number of rows in the sparse matrix (for the permutation array),

N_{jd} denotes the number of jagged diagonals.

3. The transpose jagged diagonal format

A very popular format for storing sparse matrices is the Compressed Column Storage (CCS) format or Harwell–Boeing sparse matrix format. In this storage scheme a matrix A is compressed along the columns by shifting all the non-zero elements (*nze*) upwards and then each column of the compressed matrix is stored in a linear array, say *value_list*, one column after another. Thus the array *value_list* contains all *nze* stored by columns. The length of this vector is equal to the number of non-zero elements (N_{nze}) in A. Another array, *row_indexes*, is required. This array contains the row index of each *nze* in A. The length of this array is equal to the length of *value_list*. It is straight forward to figure out that for any index i, with $1 \leqslant i \leqslant N_{nze}$, *row_indexes*$[i]$ is the row index of the *nze* stored in *value_list*$[i]$. A third array, *start_position*, is required and it contains a pointer to the

first *nze* in each row. The length of this array is equal to the number of columns of matrix A plus one, and the value of $start\_position[i+1] = N_{nze} + 1$. By applying the CCS to the following sparse matrix-vector product matrix:

$$\begin{bmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & a_{25} & 0 \\ a_{31} & 0 & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}.$$

It becomes,

$$\begin{bmatrix} a_{11} & a_{12} & a_{23} & a_{14} & a_{25} & a_{46} \\ a_{31} & a_{22} & a_{33} & a_{34} & a_{45} & a_{56} \\ 0 & a_{42} & 0 & a_{44} & a_{55} & a_{66} \\ 0 & 0 & 0 & 0 & a_{65} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}.$$

Hence, the Transpose Jagged Diagonal Storage (TJDS) scheme can be obtained from the CCS by reordering the columns of A_{ccs} in decreasing order from left to right according to the number of non-zero elements per column as shown below. We note that there is no need to store the permutation vector because the *row_indexes* array will indicate the exact position to store each partial result of the matrix-vector computation in the resulting vector Y.

$$\begin{bmatrix} a_{25} & a_{12} & a_{14} & a_{46} & a_{11} & a_{23} \\ a_{45} & a_{22} & a_{34} & a_{56} & a_{31} & a_{33} \\ a_{55} & a_{42} & a_{44} & a_{66} & 0 & 0 \\ a_{65} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_5 \\ x_2 \\ x_4 \\ x_6 \\ x_1 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}.$$

The corresponding arrays, *value_list*, *row_indexes*, and *start_position* are obtained as follows. The non-zero elements of compressed ordered matrix are stored in a floating point linear array *value_list*, one row after another. Each one of these rows is called a transpose jagged diagonal (*tjd*). Another array of the same length as *value_list* is used to store the column indexes of the elements of the compressed ordered matrix. This new array that we denote, *row_indexes*, is of type integer. Finally, a third array called *start_position* stores the starting position of each *tjd* stored in array *value_list*. These arrays are shown below:

value_list	a_{25}	a_{12}	a_{14}	a_{46}	a_{11}	$a_{23;}$	a_{45}	a_{22}	a_{34}	a_{56}	a_{31}	$a_{33;}$	a_{55}	a_{42}	a_{44}	$a_{66;}$	a_{65}

row_indexes	2	1	1	4	1	2	4	2	3	5	3	3	5	4	4	6	6

start_position	1	7	13	17	18

X	x_5	x_2	x_4	x_6	x_1	x_3

The sequential algorithm to perform the matrix-vector product $Ax = y$ using the TJDS format is shown below. In this algorithm *num_tjdiag* stands for the number of transpose jagged diagonals:

```
for i ← 1 to num_tjdiag
    k ← 1
    for j := start_position[i] to start_position[i + 1] − 1
        p ← row_indexes[k]
        Y[p] := Y[p] + value_list[j] ∗ X[k]
        k ← k + 1
    endfor
endfor.
```

4. Evaluation of TJDS

In order to evaluate the storage requirements of TJDS compared to CRS, CCS, and JDS, we present a selection of sparse matrices from the Matrix-Market collection [2]. As show in Table 1, for each matrix we present its name, dimension, N_{nze}, and the longest N_{nze} per column.

We see that to store the non-zero elements that constitutes the actual data, we also need subsidiary data, which will vary for each storage scheme. This subsidiary data includes storing of row/column indices (CCS, CRS, JDS, TJDS), the starting points of the rows/columns (CCS, CRS), the permutation vector (JDS), the number of jagged diagonals (JDS, TJDS) etc. As you can see in the case of JDS, we need an array of size N to store the original permutation. Our TJDS eliminates the need for this array of size N. In Table 2, we show that the number of elements of the permutation vector represent, for the selected matrices used in this work, up to 45% of the number of non-zero elements of matrix A.

Table 1
Selection of sparse matrices from Matrix Market

Matrix	Dimension	N_{nze}	Longest N_{nzec}
gre_185	185 × 185	1005	6
Memplus	17758 × 17758	126150	353
FS_183_3	183 × 183	1069	105
Bcsstk2	485 × 485	1810	11
Blckhole	2132 × 2132	8502	20
Cry2500	2500 × 2500	12349	6
Ibm32	32 × 32	126	7
Tols4000	4000 × 4000	8784	22

Table 2
Percentage ratio between number of non-zero elements and matrix dimension

Matrix	Dimension ($M = N$)	N_{nze}	% ratio
gre_185	185 × 185	1005	18.4%
Memplus	17758 × 17758	126150	14.1%
FS_183_3	183 × 183	1069	17.1%
Bcsstk2	485 × 485	1810	26.8%
Blckhole	2132 × 2132	8502	25.1%
Cry2500	2500 × 2500	12349	20.2%
Ibm32	32 × 32	126	25.4%
Tols4000	4000 × 4000	8784	45.5%

Table 3
Subsidiary storage required for each matrix using the four formats

Matrix	CRS	CCS	JDS	TJDS
gre_185	1190	1190	1196	1011
Memplus	143908	143908	144261	126503
FS_183_3	1252	1252	1324	1174
Bcsstk20	2295	2295	2306	1821
Blckhole	10634	10634	10654	8522
Cry2500	14849	14849	14854	12355
Ibm32	158	158	166	133
Tols4000	12784	12784	12874	8806

The formulas to calculate the subsidiary storage requirements for the CRS, CCS, JDS, and TJDS formats are:

CRS: $(N_{nze} * 1) + N$,

CCS: $(N_{nze} * 1) + M$,

JDS: $(N_{nze} * 1) + N + N_{jd}$,

TJDS: $(N_{nze} * 1) + N_{tjd}$,

where N_{nze} denotes the number of non-zero elements of matrix A, N denotes the number of rows and M denotes the number of columns. Since we are considering square matrices, $N = M$. In the case of JDS, N stands for the length of the *perm_vector*, N_{jd} denotes the number of jagged diagonals, N_{tjd} denotes the number of transpose jagged diagonals. In the above formulas, $(N_{nze} * 1)$ gives the storage required to store the array indices of the non-zero elements. In Table 3 we show the subsidiary storage required for the selected matrices using the four storage formats. On the other hand, it is worthwhile mentioning that the number of jagged diagonals in any sparse matrix is equal to the maximum number of non-zero elements per row. Likewise, in the case of the TJDS format the number of transpose jagged diagonals is equal to the maximum number of non-zero elements per column. Hence, for symmetric matrices we have $N_{jd} = N_{tjd} = \mathit{longest\_N}_{nzpr} \leqslant N$.

5. Conclusions

We have presented a solution to the sparse matrix vector product problem using TJDS, a new storage format that requires less storage space than the CRS, CCS and JDS formats. We have also shown that TJDS does not need the permutation vector required by JDS to permute the resulting vector back to the original ordering. The number of elements of the permutation vector can be, for the selected matrices used in this work, up to 45% of the number of non zero elements of matrix A. This new format is suitable for parallel and distributed processing because the data partition scheme inherent to the data structures used allows splitting the algorithm in two well defined phases which grant local computation.

References

[1] J. Dongarra, Sparse matrix storage formats, in: Z. Bai, et al. (Eds.), Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, SIAM, Philadelphia, 2000. Electronic version available at: http://www.cs.utk.edu/~dongarra/etemplates/node372.html.
[2] Matrix Market, Electronic version available at: http://math.nist.gov/MatrixMarket/.
[3] Y. Saad, Krylov subspace methods on supercomputers, Siam. J. Sci. Stat. Comp. 10 (6) (1989) 1200–1232.
[4] Y. Saad, SPARSKIT: A basic tool kit for sparse matrix computations Report RIACS-90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.