FACTA UNIVERSITATIS (NIŠ)
Ser. Math. Inform. 24 (2009), 39–51

PERFORMANCE COMPARISON OF STORAGE FORMATS FOR SPARSE MATRICES *

Ivan P. Stanimirović and Milan B. Tasić †

Abstract. The sparse data structure represents a matrix in space proportional to the number of non-zero entries. Many storage formats have been proposed to represent sparse matrices. In this paper we evaluate and compare the storage efficiency of various sparse matrix storage formats, and consider the performance results of matrix-vector multiplication using these storage formats.

1. Introduction

Most physical and social structures are sparse in the sense that the elements of these structures are loosely connected. For example, the atoms of very large molecules are directly connected only to a few other atoms, etc. A mathematical model of these connections is the adjacency matrix A, where $a_{ij} = 1$ if the element i is connected directly to element j, and $a_{ij} = 0$ otherwise. We can also represent a road network by an adjacency matrix, where a_{ij} is the distance or travel time between towns i and j directly connected by a road, and $a_{ij} = 1$ otherwise. Such adjacency matrices occur in many applications and they all have the characteristic that most of their elements are 0 or 1, [2, 4].

In most applications involving sparse matrices the size of the matrix is very large. Typically the dimensions are in the range $1,000 - 1,000,000$, with 220 nonzeros per row. Storing such large matrices is impossible, even on supercomputers. Besides, most of the storage would be wasted on zeros and, worse still, most of the calculations would be wasted on zeros. Sparse matrices (matrices with a substantial minority of nonzero elements, normally less than 10% nonzero elements) are pervasive in many mathematical and scientific applications. These matrices provide an opportunity to minimize storage and computational requirements by storing,

Received
2000 *Mathematics Subject Classification.* Primary 68N15; Secondary 97N60, 65F50
*Supported by Ministry of Science and Technological Development, Republic of Serbia, Grant no. 144011.
†Corresponding author

and performing arithmetic with, only the nonzero elements. The significant number of different storage formats gives us the source of a research. For example, consider the published Basic Linear Algebra Subroutines (*BLAS*) standard and the part dedicated to sparse matrices (*Sparse BLAS*), [5]. The (*Sparse BLAS*) do not state which storage formats must be supported or used. Each specific hardware vendor has the freedom (or problem) to select the storage format (or formats) that perform best for its hardware. In the context of iterative methods [1] and JAVA, this paper investigates the performance delivered by different storage formats considering a implementation in C++ and MATHEMATICA.

The structure of the paper is as follows. Section 2 introduces the most commonly used storage formats for sparse matrices. The Java Sparse Array (*JSA*) storage format was recently proposed in [6] to take advantage of Java arrays. The performance evaluation consider a specific kernel from iterative methods, namely matrix-vector multiplication, and compares this operation on two different computational platforms with nine different storage formats. The C++ and MATHEMATICA implementation of this matrix operation is described in Section 3. The performance study considers around 200 different sparse matrices representing various *CS*&*E* applications as recorded by the Matrix Market repository [7]. To the best of the authors knowledge, there is no other performance evaluation of storage formats for sparse matrices which consider such a variety of matrices and storage formats. Conclusions and future work are given in the last Section.

2. Storage formats for sparse matrices

The objective of storage formats for sparse matrices is to best exploit certain matrix properties by (1) reducing memory space, by storing only nonzero elements of a sparse matrix, and (2) by storing these elements in contiguous memory locations, for more efficient execution of subroutines on the matrix data. From an implementation point of view, there are two categories of storage formats. Point entry is used to categories storage formats where each entry in the storage format is a single element of the matrix. Block entry refers to storage formats where each entry defines a dense block of elements of any two dimensions. For both cases, programming languages provide static and dynamic data structures.

There are many documented versions of different storage formats for sparse matrices. One of the most complete sources is the book by Duff et al. [4], (for a historical source see [8]). Some examples of these storage formats follow.

2.1. Point entry storage formats

Coordinate Format (*COO*). Possibly the most intuitive storage format for a sparse matrix is in terms of coordinates. Instead of storing the matrix densely, a list of the coordinates in terms of row and column numbers is stored, with the associated nonzero values. *COO* requires no specific structure of the matrix and

is a very flexible format. It requires three (unordered) arrays and a single scalar recording the total *number of nonzero elements*, *nnz*. The combination of the three arrays provides a row i and column j coordinate pair for an element in the matrix along with its value a_{ij} . In general, for a matrix with *nnz*, *COO* requires three one-dimensional arrays of length *nnz* plus a scalar. By default, some implementation methods create matrices in *COO*. Using internal utilities subroutines, users can easily transform from *COO* to the other storage formats.

2.1.1. Compressed sparse row/column storage formats (*CSR*/*CSC*)

CSR and *CSC* storage formats are not based on any particular matrix property and hence can be used to store any sparse matrix. In *CSR*, the nonzero values of every row in the matrix are stored, together with their column number, consecutively in two parallel arrays, *Value* and *j*. There is no particular order with respect to the column number, *j*. The *Size* and *Pointer* for each row define the number of nonzero elements in the row and point to the relative position of the first nonzero element of the next row, respectively. The column based version, *CSC*, instead stores *Value* and *i*, in two parallel arrays and *Size* and *Pointer* of each column allows each member of *Value* to be associated with a column as well as the row given in *i*. The storage requirements are two arrays, each of length the number of rows (or columns), and two further arrays of length *nnz*, and a scalar to point to the next free location in the arrays *i* (or *j*) and *Value*.

Assume that A is a $m \times n$ sparse matrix and $A_1, \ldots, A_m$ are the rows of A. In *CSR* format matrix A is represented as the vector $\{l_1, \ldots, l_m\}$, where the vector l_i represents the ith row A_i. Each element l_i is the ordered pair of the form $\{j, a_{ij}\}$, where j indicates the column which contains a non-zero element a_{ij}. Each zero row is represented by an empty list.

Example 2.1. Consider the following matrix $A = \begin{bmatrix} a_{00} & 0 & 0 & a_{03} \\ 0 & a_{11} & 0 & 0 \\ a_{20} & 0 & a_{22} & 0 \end{bmatrix}$.

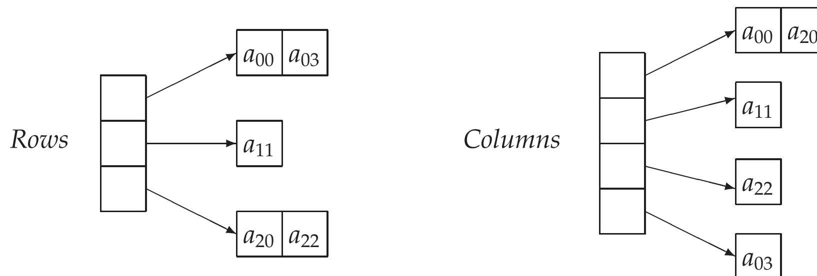

FIG. 2.1: An example sparse matrix A stored using *CSR* and *CSC*

2.2. Block entry storage formats

Block entry storage formats form an extension of certain point entry storage formats based on partitioning matrices into blocks of elements (i.e. sub-matrices). Block entry storage formats divide a matrix into blocks which can be squares or rectangles. These storage formats also define schemes to describe the memory position of a single block. If the block size remains fixed, for example, *Block Sparse Row/Column* (*BSR/BSC*) storage format can be obtained from *CSR/CSC*, respectively. Similarly, when the block size can vary the *Variable Block Compressed Sparse Row/Column* formats (*VBR/VBC*) are obtained. An example of a variable block matrix A is as follows:

$$A = \left[\begin{array}{cc|cccc|c} a11 & a12 & a13 & a14 & a15 & a16 & a17 \\ \hline a22 & a22 & a23 & a24 & a25 & a26 & a27 \\ a31 & a32 & a33 & a34 & a35 & a36 & a37 \\ \hline a41 & a42 & a43 & a44 & a45 & a46 & a47 \\ a51 & a52 & a53 & a54 & a55 & a56 & a57 \\ a61 & a62 & a63 & a64 & a65 & a66 & a67 \end{array}\right] \text{ or } \left[\begin{array}{ccc} A11 & A12 & A13 \\ A21 & A22 & A23 \\ A31 & A32 & A33 \end{array}\right]$$

where,

$$A11 = \left[\begin{array}{cc} a11 & a12 \end{array}\right], A13 = \left[\begin{array}{c} a17 \end{array}\right], A22 = \left[\begin{array}{cccc} a23 & a24 & a25 & a26 \\ a33 & a34 & a35 & a36 \end{array}\right], A33 = \left[\begin{array}{c} a47 \\ a57 \\ a67 \end{array}\right].$$

In the point entry storage formats, the storage format describes the position in the (*Value*) array of single matrix elements. Block entry storage formats (with length of block lb), instead have a scheme to describe the position of a single block in a $n/lb \times n/lb$ blocked matrix. Each block contains $lb2$ elements. In this way, most point entry storage formats can be blocked to generate *Block Coordinate* storage format (*BCO*), *Block Sparse Row/Column* storage format (*BSR/BSC*) and others where the block does not have constant dimensions (e.g. *Variable Block Compressed Sparse Row*).

2.3. Java sparse array (*JSA*)

Java Sparse Array (*JSA*) is a recently designed storage format, designed particularly to suit Java (see [6]), relying on declaring an array with individual elements being arrays arrays of arrays. *JSA* is a row oriented storage format, similar to *CSR*. The matrix is implemented as two arrays, each element of which is a pointer to an array. One of these arrays, Value, stores pointers to arrays which contain the matrix elements each row in the matrix has its elements in a separate array. All the separate arrays can be reached through the pointers in the Value array, that is an array of pointers to arrays. The second array Index stores pointers to arrays which contain the column indices of the matrix, again one array per row. Fig. 2.2. shows the matrix A introduced in Example 2.1 stored using *JSA*.

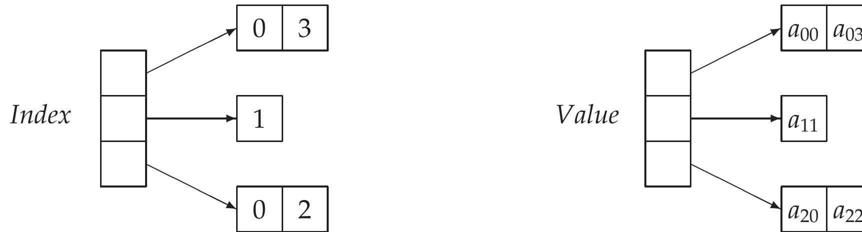

FIG. 2.2: The sparse matrix A stored using *JSA*

Short name	Name	Short name	Name
DNS	Dense	ELL	Ellpack-Itpack
BND	Linpack Banded	DIA	Diagonal
COO	Coordinate	BSR	Block Sparse Row
CSR	Compressed Sparse Row	SSK	Symmetric Skyline
CSC	Compressed Sparse Column	BSR	Nonsymmetric Skyline
MSR	Modified CSR	JAD	Jagged Diagonal
LIL	Linked List		

Table 2.1: Popular Storage Formats

2.4. Other storage formats with examples

Many sparse matrix storage formats are in use today, each of them having its acronym. Some of the most applied sparse matrix storage structures and their acronyms are shown in the Table 2.1. Please note that *CSR* is called *CRS*, *CCS* is *CSC*, and *SSK* is *SKS* in some references.

Example 2.2. *DNS* is a simple, row-wise, easy blocked format. For matrix

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix},$$

storage structure is $AA = [3\ 3\ 1.0\ 2.0\ 3.0\ 4.0\ 5.0\ 6.0\ 7.0\ 8.0\ 9.0]$.

Example 2.3. Let us apply some known storage formats on the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}.$$

COO

The COO representation gives the following vectors:

AA = [12.0 9.0 7.0 5.0 1.0 2.0 11.0 3.0 6.0 4.0 8.0 10.0];

JR = [5 3 3 2 1 1 4 2 3 2 3 4];

JC = [5 5 3 4 1 4 4 1 1 2 4 3].

This method is simple, often used for entry.

CSR

The following arrays are generated here:

AA = [1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0];

JA = [1 4 1 2 4 1 3 4 5 3 4 5];

IA = [1 3 6 10 12].

The lengths of both AA and JA are nnz, and the length of IA is $n+1$. Denote that for each $j \in \{1, ..., n\}$ $IA(j)$ gives the index (offset) of the beginning of j-th row in AA and JA. According to some sources from Fortran programming language this representation gives fast row access, but slow column access.

MSR

Now we have the following vectors:

AA = [1.0 4.0 7.0 11.0 12.0 $*$ 2.0 3.0 5.0 6.0 8.0 9.0 10.0];

JA = [7 8 10 13 14 14 4 1 4 1 4 5 3].

This method gives importance to diagonal elements in the matrix, which are often nonzero or frequently accessed. The first n entries of the array AA are now the diagonal elements, the $(n+1)$-th entry is empty, and the rest of AA are the nondiagonal entries. The first $n+1$ entries in JA vector give the indices (offsets) of the beginnings of each row, as introduced earlier (denote that the array IA from the CSR representation can be generated from the array JA). The rest of JA vector are the column indices.

ELL

Here we form columns from first non-zero in each row and repeat this process. This representation assumes low number of nnz per row, equal to number of columns in $COEFF$ and $JCOEFF$.

$$COEF = \begin{bmatrix} 1.0 & 2.0 & 0 \\ 3.0 & 4.0 & 5.0 \\ 6.0 & 7.0 & 8.0 \\ 9.0 & 10.0 & 0 \\ 11.0 & 12.0 & 0 \end{bmatrix}, \; JCOEF = \begin{bmatrix} 1 & 3 & 1 \\ 1 & 2 & 4 \\ 2 & 3 & 5 \\ 3 & 4 & 4 \\ 4 & 5 & 5 \end{bmatrix}.$$

DIA (or CDS-Compressed Diagonal Storage)

If the matrix A is banded with bandwidth that is fairly constant from row to row, then it is worthwhile to take advantage of this structure in the storage scheme

by storing sub-diagonals of the matrix in consecutive locations. Not only can we eliminate the vector identifying the column and row, but we can pack the nonzero elements in such a way as to make the matrix-vector product more efficient. This storage scheme is particularly useful if the matrix arises from a finite element or finite difference discretization on a tensor product grid. We say that the matrix $A = (a_{i,j})$ is banded if there are nonnegative constants p, q, called the left and right half-bandwidth, such that $a_{i,j} \neq 0$ only if $i - p \leq j \leq i + q$. In this case, we can allocate for the matrix A an array $val(1 : n, -p : q)$. Usually, band formats involve storing some zeros. The *CDS* format may even contain some array elements that do not correspond to matrix elements at all.

Consider the nonsymmetric matrix A defined by

$$A = \begin{bmatrix} 10 & -3 & 0 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 0 & 0 & 8 & 7 & 5 & 0 \\ 0 & 0 & 0 & 9 & 9 & 13 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

Using the *CDS* format and an array $val = (6, -1 : 1)$, we store this matrix A as

$$DIA = \begin{bmatrix} 0 & 10.0 & -3.0 \\ 3.0 & 9.0 & 6.0 \\ 7.0 & 8.0 & 7.0 \\ 8.0 & 7.0 & 5.0 \\ 9.0 & 9.0 & 13.0 \\ 2.0 & -1.0 & 0 \end{bmatrix}.$$

Example 2.4. **LIL** is very similar to *CSR*, but rather than a flat *AA* vector, each row is a linked list of elements. First element of each row is accessed by *ROOT*, each element in *AA* has a corresponding *NEXT* entry.

Consider the next matrix:

$$A = \begin{bmatrix} 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 6 & 4 \end{bmatrix} \rightarrow$$

i	*IA*	*JA*	*AA*	*NEXT*
1	4	3	2	−1
2	3	2	2	5
3	2	2	1	1
4	8	1	7	−1
5	7	4	2	−1
6	–	5	6	6
7	–	6	4	−1
8	–	5	5	−1

Number −1 indicates the end of a row. Column lookup take $O(nnz)$. Store a column wise index in the same way as rows. Very good element insertion time, but more memory.

3. The implementation of some storage format

Here we implement some storage formats in two different programming languages. The idea is to show the implementation of the sparse-vector functions when the exact storage format is known.

3.1. The implementation of *CSR* and *CSC* formats in C++

When implementing the *CSR*/*CSC* storage formats in programming language C++, we use the following pointer representation.

```
typedef float type;

typedef struct element
{   int row;  int col;  type val;
    struct element *NextInRow;  struct element *NextInCol;
} Element;

typedef struct RPM
{   Element **rows;  Element **cols;  short int dim;
} SparseMatrix;
```

Denote that our sparse matrix is implemented as a structure containing the matrix size stored in the variable *dim*, and two arrays of the pointers to the elements of the matrix, which are stored in the variables *rows* and *cols*. Also, each element is stored as a structure that contains its value, position in the matrix denoted as its row and column number and, eventually, two pointers to the elements which are successive in Row and Column. These two pointers are stored using *NextInRow* and *NextInCol* statements. Notice that the memory required for *CSR*/*CSC* matrix storage is equal to:

$$NumberOfNonzeroElements \times StorageSizeOfIndividualElement.$$

3.1.1. Converting sparse matrix to the matrix-vector

The function that converts sparse matrix to the matrix vector introduced by *CSR* storage format requires two parameters: the matrix in standard form with elements of the corresponding type and the matrix size. Here we give one way to implement this function.

```
SparseMatrix Convert(type **mat, int n)
{   SparseMatrix A;  A.dim=n;
    A.rows = (Element **) malloc(n*sizeof(Element *));
    A.cols = (Element **) malloc(n*sizeof(Element *));
    for(int i=0; i<n; i++) A.rows[i]=NULL;
    for(int i=0; i<n; i++) A.cols[i]=NULL;
    for(int i=0; i<n; i++)
```

```
        for(int j=0; j<n; j++)
            if(mat[i][j])
            {   Element *novi=(Element *) malloc(sizeof(Element));
                novi->row=i; novi->col=j;
                novi->val=mat[i][j];
                novi->NextInCol = novi->NextInRow = NULL;
                if (A.rows[i]==NULL) A.rows[i]=novi;
                  else {Element *p=A.rows[i];
                        while (p->NextInRow) p=p->NextInRow;
                        p->NextInRow=novi; }
                if (A.cols[j]==NULL) A.cols[j]=novi;
                  else {Element *p=A.cols[j];
                        while (p->NextInCol) p=p->NextInCol;
                        p->NextInCol=novi;}}
    return A;}
```

3.1.2. C++ implementation of matrix-vector multiplication

For matrix-vector multiplication, two for-cycles are carried out, creating the products of matrix *A* row-vectors and matrix *B* column-vectors. After that, the implementation of matrix-vector multiplication present an if-then-else code structure separating special cases.

```
SparseMatrix Multiply(SparseMatrix A, SparseMatrix B)
{   SparseMatrix C; C.dim=A.dim;
    C.rows = (Element **) malloc(C.dim*sizeof(Element *));
    C.cols = (Element **) malloc(C.dim*sizeof(Element *));
    for(int i=0; i<C.dim; i++) C.rows[i]=NULL;
    for(int i=0; i<C.dim; i++) C.cols[i]=NULL;
    for(int i=0; i<C.dim; i++)
        for(int j=0; j<C.dim; j++)
        {   Element *p=A.rows[i], *q=B.cols[j];
            int s=0;
            while(p)
            {   while(q && (p->col > q->row)) q=q->NextInCol;
                if(q && (p->col == q->row)) s+=p->val*q->val;
                p=p->NextInRow;}
            if (s)
            {   Element *novi=(Element *)malloc(sizeof(Element));
                novi->row = i;
                novi->col = j;
                novi->val = s;
                novi->NextInCol=novi->NextInRow=NULL;
                if(!C.rows[i]) C.rows[i]=novi;
                  else {Element *p=C.rows[i];
                        while (p->NextInRow) p=p->NextInRow;
                        p->NextInRow=novi;}
                if(!C.cols[j]) C.cols[j]=novi;
                  else {Element *p=C.cols[j];
                        while (p->NextInCol) p=p->NextInCol;
                        p->NextInCol=novi;} } }
    return (C);}}
```

3.2. Representation of COO format in MATHEMATICA

In the COO representation, a sparse matrix is represented as a vector of appropriate ordered triples. Each ordered triple corresponds to a non-zero element a_{ij}, and possesses the form $\{i, j, a_{ij}\}$. When we deal with *sparse matrices*, computer algebra comes close to the methods of numerical calculation [3]. For this purpose in this paper are written functions for implementation of basic operations in matrix algebra, which are applicable to sparse matrix representation. The function *Sparse[mat_]* uses the formal parameter *mat_*, representing a matrix in the usual ("dense") form, required in MATHEMATICA [9].

The local variable l denotes an adequate *sparse representation* of the matrix *mat*:

$$l = \{\{i_1, j_1, mat_{i_1,j_1}\}, \{i_2, j_2, mat_{i_2,j_2}\}, \ldots, \{i_u, j_u, mat_{i_u,j_u}\}\}.$$

Local variables $a1$ and $b1$ are assigned to store the lists which contain different elements from the lists $\{i_1, i_2, \ldots, i_u\}$ and $\{j_1, j_2, \ldots, j_u\}$, respectively. More precisely, the list $a1$ represents the union of first elements from the list l, while elements of the list $b1$ are the union of the second elements from the lists l.

The result of the function *Sparse[mat_List]* is the ordered triple $\{a1, b1, l\}$.

```
Sparse[mat_List]:= Block[{a1=b1=l={},m,n,i,j},
{m,n}=Dimensions[mat];
  For[i=1, i<=n, i++,
   For[j=1, j<=m, j++, If[mat[[i,j]] =!= 0,l=Append[l,{i,j,mat[[i,j]]}]]]];
  For[i=1,i<=Length[l],i++,a1=Union[a1,{l[[i,1]]}]; b1=Union[b1,{l[[i,2]]}]];
  {a1,b1,l}];
```

Let us mention that an arbitrary sparse row-matrix a is represented in the form $\{\{1, i_1, a_{1,i_1}\}, \cdots, \{1, i_k, a_{1,i_k}\}\}$, and a sparse column-matrix is represented by the list $\{\{i_1, 1, a_{i_1,1}\}, \cdots, \{i_k, 1, a_{i_k,1}\}\}$.

By means of the function *ColS[a,k]* it is possible to form the vector whose elements are taken from the k-th column of the sparse representation a of the matrix A.

```
ColS[a_List,k_]:= Block[{Ak={},i},
   For[i=1,i<=Length[a],i++,If[a[[i,2]]==k,Ak=Append[Ak,{a[[i,1]],1,a[[i,3]]}]]];
  Ak];
```

The function *MaxDim[a,k]* computes the maximal index between the indices of rows which contain a non-zero element, in the case $k = 1$. Similarly, in the case $k = 2$, the result of this function is the maximum in the set of indices of columns containing a non-zero element. It is assumed that a is the sparse representation of the matrix A.

```
MaxDim[a_List,k_]:=Block[{l={},i},
   For[i=1,i<=Length[a],i++,l=Append[l,a[[i,k]]]]; Max[l]];
```

By means of the function *InPos[a,i1,i2]* we compute the value $A[[i1,i2]]$, denoted by S, and the position *pos* of this value in the sparse representation a of the matrix A.

```
InPos[a_List,i1_,i2_]:= Block[{find=True,i=1,s,pos},
  While[find,If[(a[[i,1]]==i1)&&(a[[i,2]]==i2),
       s=a[[i,3]];pos=i;find=False,
       If[i==Length[a],s=0;find=False,i++]]];
  {s,pos}];
```

The result of the function *MultMat[a,b]* is the product of two matrices a and b, given in the form of ordered triples.

```
MultMat[a_List,b_List]:= Block[{C={},Amb={},i,j,k,m1,m2,n1,n2,m,n},
  m1=MaxDim[a,1];m2=MaxDim[b,1];n1=MaxDim[a,2];n2=MaxDim[b,2];
  m=Max[m1,m2];n=Max[n1,n2];
  For[i=1,i<=m,i++, For[j=1,j<=n,j++,
    C=Append[C,{i,j,Sum[InPos[a,i,k][[1]]*InPos[b,k,j][[1]],{k,1,n}]}]]];
  For[i=1,i<=Length[C],i++,If[C[[i,3]]=!=0,Amb=Append[Amb,C[[i]]]]];
  Simplify[Amb]];
```

The function *MultSk[a,b]* gives the matrix which is equal to the product of the matrix a with the scalar b.

```
MultSk[a_List,b_]:= Block[{A=a,i},
  For[i=1,i<=Length[A],i++,A[[i,3]]=A[[i,3]]*b]; Simplify[A]];
```

The result of the function *Rcpv[a]* is a matrix whose sparse representation is determined by reciprocal values of elements contained in the sparse representation a.

```
Rcpv[a_List]:=Block[{A=a,i},
 For[i=1,i<=Length[A],i++,A[[i,3]]=1/A[[i,3]]]; Simplify[A]];
```

Transpose of a given sparse matrix a can be generated by means of the function *HermitS[a]*.

```
HermitS[a_List]:=Block[{A=a,t,i},
  For[i=1,i<=Length[A],i++,t=A[[i,1]];A[[i,1]]=A[[i,2]];A[[i,2]]=t]; A];
```

The result of the function *SubMat[a_List,b_List]* is equal to the difference of matrices a and b, represented in the sparse form. The sum of matrices a and b can be computed using the function call *SubMat[a,MultSk[b,-1]]*.

```
SubMat[a_List,b_List]:= Block[{B=b,g,M={},C={},t,t1,i,s1,s2},
 For[i=1,i<=Length[a],i++,s1=a[[i,1]];s2=a[[i,2]];g=InPos[B,s1,s2];t=g[[1]];t1=g[[2]];
   If[Not[t=!=0],C=Append[C,a[[i]]],
      If[a[[i,3]]=!=t,C=Append[C,{s1,s2,a[[i,3]]-t}]]; M=Append[M,B[[t1]]]]];
 B=Complement[B,M];
 For[i=1,i<=Length[B],i++,C=Append[C,{B[[i,1]],B[[i,2]],-B[[i,3]]}]];
 Simplify[C]];
```

Table 4.1: Timings for matrix-vector multiplication of different sizes and densities.

Density	50	100	150	200	250	300	350	400	450	500
100%	0.2	0.4	0.85	1.6	3.2	6.1	10.8	16.5	28.3	42
75%	0.1	0.2	0.75	1.3	3	4.5	6.7	10.9	19.2	27.8
50%	0	0.2	0.7	1.2	2.3	3.1	3.9	6.2	10.6	14.1
25%	0	0.1	0.4	0.75	1	1.5	2.1	2.7	4.8	7.2
10%	0	0.1	0.3	0.55	0.8	1	1.2	1.6	1.9	2.3
Standard	0	0.1	0.35	0.7	1	1.3	1.8	2.3	3	3.7

4. CSC/CSR storage format performance results

Table 4.1 shows the performance results of matrix-vector multiplication using *CSC*/*CSR* storage format implemented in programming language C++. All timings in the table are given in seconds. No distinction is made between symmetric and non-symmetric matrices. These performance results of each different matrix density and matrix size are calculated as the average of the several testings. All matrix elements in these test-examples are randomly generated. The results are ordered by increasing the total number of non-zero elements.

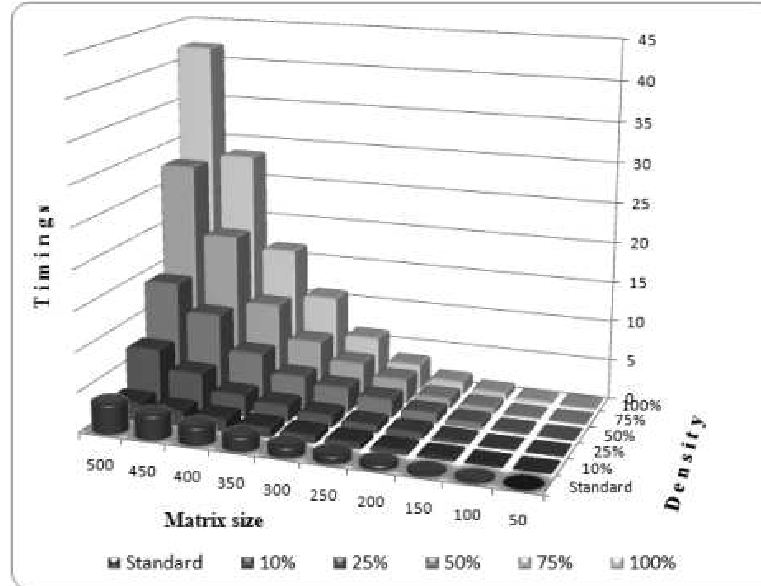

Fig. 4.1: Time results for different matrix densities using *CSR* storage format.

5. Conclusions

It would be presumptuous to say that all the storage formats for sparse matrices are covered by this work, especially since there are many minor variations which can create entirely new storage formats. Nonetheless, this paper has presented a comprehensive performance comparison of storage formats for sparse matrices. Future work underway is to include a similar set of experiments with Data Base storage system implementations and the other operations supported by SQL.

REFERENCES

1. R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, (1994).
2. D. O'Connor, *Algorithms & Data Structures*, (2002).
3. J.H. Davenport, Y. Siret and E. Tournier, *Computer Algebra*, Academic Press, (1988).
4. I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, (1986).
5. I. S. Duff, M. A. Heroux, and R. Pozo, *An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum*, ACM Transactions on Mathematical Software, 2002, pp. 239-267.
6. G. Gundersen, T. Steihaug, *Data structures in Java for matrix computations*, Concurrency and Computation: Practice and Experience, 2004, 799-815.
7. The matrix market, *http://math.nist.gov/MatrixMarket/*.
8. U. W. Pooch, A. Nieder, *A survey of indexing techniques for sparse matrices*, ACM Computing Surveys, 1973, 109-133.
9. S. Wolfram, *The Mathematica Book, 4th ed.*, Wolfram Media/Cambridge University Press, (1999).

Ivan P. Stanimirović
Faculty of Science and Mathematics,
Department of Mathematics and Informatics,
P. O. Box 224, Višegradska 33,
18000 Niš, Serbia
ivan.stanimirovic@gmail.com

Milan B. Tasić
Faculty of Science and Mathematics,
Department of Mathematics and Informatics,
P. O. Box 224, Višegradska 33,
18000 Niš, Serbia
milan12t@ptt.rs