Review of Storage Techniques for Sparse Matrices

Rukhsana Shahnaz, Anila Usman, Imran R. Chughtai
Pakistan Institute of Engineering and Applied Sciences (PIEAS),
Nilore, Islamabad. Pakistan.
{fac115, anila, imran }@pieas.edu.pk,

Abstract

This paper reviews the current state of knowledge of the storage formats for sparse linear systems. Here we consider the ways developed so far for storing a sparse matrix and their quoted effects on computational speed. The main idea behind these formats involves keeping both the indices and the non-zero elements in the sparse matrix in a single data structure. These specialized schemes not only save storage but also yield computational savings. Since the locations of the non-zero elements in the matrix are known explicitly, unnecessary computations involving zeros can be avoided [1, 2]. Thus the use of these formats reduces additional memory required in the usual indexing based storage schemes and gives promising performance improvements [3, 4].

1. Introduction

Sparse matrices are (usually large) matrices that contain a low percentage of non-zeros, which are fairly evenly distributed along the two dimensions, and usually very few non-zero elements per row [5, 6]. As the percentage of non-zero elements in a matrix decreases, there is a turning point where it is no longer efficient to operate on matrix elements since most operations are trivial with zeros. It then becomes more efficient to pay an additional cost for handling a more complicated storage scheme in order to avoid the trivial operations with zeros [7].

One of the main issues of storage is the reduction of "net" storage requirements. Here the idea of net storage is applied because with a sparse storage scheme, there are two types of storages concerned: "primary" which contains the actual non-zero matrix components and "overhead" which contains all the indexing information dealing with the primary storage [1]. The more sophisticated is the sparse storage scheme, the more indexing we have, and so a decrease in primary storage gives an increase in overhead storage. Therefore, our objective is the reduction of the net overall storage. At this point, besides that of memory use, there is another important issue to raise and it is the minimization of computational cost. Use of sparse storage scheme reduces the amount of zero multiplications compared to the dense storage (normal) case, but obviously this must be balanced against the increase in the cost of indexing per operation [8].

2. Sparse storage formats

For sparse matrices, the common practice is to store only the non-zero entries and to keep track of their locations in the matrix through an indexing scheme. A variety of such storage schemes are used to store and manipulate sparse matrices.

- General formats: Some storage formats are for general types of matrices, like the Compressed Row Storage (CRS) format, which make no assumption of the structure of distribution of non-zeros [9, 10].
- Specific formats: Some formats are optimized for specific types of matrices that mostly arise from a particular field. An example is Block Compressed Row Storage (BCRS), which is efficient for block matrices [9, 10].

The sparse storage formats can mainly be divided into following categories [11].

a) Point based storage formats

In point based storage scheme each entry is a single element of the matrix.

b) Block based storage formats

The block based storage scheme is one in which each entry defines a dense block of elements of any two dimensions.

2.1. Point based storage formats

2.1.1. **The coordinate storage (COO).** The most intuitive format for sparse matrix is in terms of coordinates of non-zero elements. Instead of storing the matrix densely, a list of coordinates of row and column indices is stored. Three data structures *{val, row_ind, col_ind}* are used in COO. *val()* contains all the non-zero elements of sparse matrix, *row_ind()* and *col_ind()* hold the row and column indices of non-zero matrix elements respectively [11].
This storage scheme is an inefficient one compared to the rest of the schemes that are available because of the amount of the indexing involved in accessing the elements of the matrix [12, 13].

6	0	9	0	0	4	0	0
0	0	0	0	0	4	0	0
0	5	0	0	0	0	0	0
0	0	3	5	8	0	0	0
0	0	0	0	6	0	0	0
0	0	0	0	0	5	0	0
0	0	0	0	0	4	3	0
0	0	0	0	0	0	2	2

Matrix A

val()	6	9	4	4	5	3	5	8	6	5	4	3	2	2
row_ind()	1	1	1	2	3	4	4	4	5	6	7	7	8	8
col_ind()	1	3	6	6	2	3	4	5	5	6	6	7	7	8

Figure 1. Coordinate storage format

2.1.2. **The compressed row storage (CRS).** The Compressed Row Storage (CRS) format for sparse matrices is perhaps the most widely used format when no sparsity structure of the matrix is required. In CRS rows are stored in consecutive order [12, 14]. The CRS format is specified by the arrays {*val, col_ind, row_ptr*}. *val()* contains the non-zero matrix elements taken in a row-wise fashion, *col_ind()* contains the column positions of the corresponding elements in *val()* and *row_ptr()* contains pointers to the first non-zero element of each row in array *val()* with $row\_ptr(N+1) = N_{nze}$ [15, 16].

The storage savings for this approach is significant. Instead of storing n^2 elements, it requires only $(2 * N_{nze} + N + 1)$ storage locations [1].

val()	6	9	4	4	5	3	5	8	6	5	4	3	2	2
col_ind()	1	3	6	6	2	3	4	5	5	6	6	7	7	8
row_ptr()	1	4	5	6	9	10		11		13		14		

Figure 2. Compressed row storage format of matrix A given in figure1

2.1.3. **The compressed column storage (CCS).** Similar to CRS is the Compressed Column Storage (CCS) (also called the Harwell-Boeing Sparse Matrix Storage Format) [17, 12], which is constructed in exactly the same way as CRS but with the roles of rows and columns interchanged. One can also say that CCS is the transpose of CRS [12]. In CCS, the sparse matrix is scanned column-wise to create three arrays *{val, row_ind, col_ptr}*. *val()* contains the non-zero elements of matrix taken in a column wise fashion. *row_ind()* contains the row positions of the elements in *val()* and *col_ptr()* contains the pointers to the first non-zero elements in each column with $col\_ptr(N+1) = N_{nze}$ [15, 16].

The main reason for using this format instead of CRS is that some programming languages, particularly FORTRAN, traditionally stores matrices column-wise rather that row-wise [1].

val()	6	5	9	3	5	8	6	4	4	5	4	3	2	2
row_ind()	1	3	1	4	4	4	5	1	2	6	7	7	8	8
col_ptr()	1	2	3	5	6	8	12		14		14			

Figure 3. Compressed column storage format of matrix A given in figure 1

2.1.4. **The compressed diagonal storage (CDS).** Compressed Diagonal Storage (CDS) is a non-general sparse matrix storage format that makes use of the particular sparsity pattern. This format is used when the matrix is banded, that is, the non-zeros elements are within a diagonal band. Figure 4 depicts an example of how the three main diagonals of a banded matrix can be stored using the CDS method. This format is not suitable for general sparse matrices since few rows that exceed the diagonal band will result in storing a large number of zero values [1, 13].

1	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	1
0	0	0	0	2	1

val(:, -1)	0	3	7	8	9	2
val(:, 0)	1	9	8	7	9	1
val(:, +1)	3	6	7	5	1	0

Figure 4. Compressed diagonal storage format

2.1.5. **The jagged diagonal storage (JDS).** The Jagged Diagonal Storage (JDS) is a format that is specially tailored for sparse matrix vector

multiplications [12, 1]. The JDS format for a matrix requires four arrays *{val, col_ind, perm, jd_ptr}*. All the non-zero elements are shifted left leaving the zeros to the right. This gives a new matrix A_{crs}. A Jagged Diagonal Storage A_{jds} is obtained by reordering the rows of A_{crs} in decreasing order from top to bottom according to the number of nonzero elements per row. The nonzero elements of the A_{jds} matrix are stored in an array *val()*, one column after another. Each one of these columns is called a jagged diagonal. *col_ind()* stores the column indices of the non-zero elements, *perm()* permutes the resulting vector back to the original ordering and *jd_ptr(),* stores the starting position of the jagged diagonals in the array *val()* [17, 18, 15].

1	3	0	1	0	0
0	9	6	0	2	0
3	0	8	7	0	0
0	6	0	7	5	4
0	0	0	0	9	1
0	0	0	0	5	1

Matrix A

1	3	1	
9	6	2	
3	8	7	
6	7	5	4
9	1		
5	1		

Matrix A_{crs}

6	7	5	4
9	6	2	
3	8	7	
1	3	1	
9	1		
5	1		

Matrix A_{jds}

val( )	6	9	3	1	9	5;	7	6	8
	3	1	1;	5	2	7	1;	4;	
col_ind( )	2	2	1	1	5	5;	4	3	3
	2	6	6;	5	5	4	4;	6;	
perm( )	4	2	3	1	5	6			
jd_ptr( )	1	7	13	17					

Figure 5. Jagged diagonal storage format

The memory required for the JDS format is $(N_{nze} * 2) + N + N_{jd}$ [1].

2.1.6. **Transposed jagged diagonal storage (TJDS).** The Transpose Jagged Diagonal Storage (TJDS) is inspired from the Jagged Diagonal Storage format and makes no assumptions about the sparsity pattern of the matrix [17]. In TJDS all the non-zero elements are shifted upward instead of leftward as in JDS. This gives a new matrix A_{ccs}. A Transposed Jagged Diagonal Storage A_{tjds} is obtained by reordering the columns of A_{ccs} in decreasing order from left to right according to the number of nonzero elements per column. The nonzero elements of the A_{tjds} matrix are stored *val()*, one row after another. Each one of these rows is called a transposed jagged diagonal. Another array *row_ind()* is needed to store the row indices of the non-zero elements in the original matrix. Finally, a third array is also needed, *tjd_ptr(),* which stores the starting position of the transposed jagged diagonals in the array *val()*[17, 18].

Although TJDS suffers the drawback of indirect addressing, it does not need the permutation step. This format is suitable for parallel and distributed processing [17, 19].

1	3	0	1	0	0
0	9	6	0	2	0
3	0	8	7	0	0
0	6	0	7	5	4
0	0	0	0	9	1
0	0	0	0	5	1

Matrix A

1	3	6	1	2	4
3	9	8	7	5	1
	6		7	9	1
				5	

Matrix A_{ccs}

2	3	4	1	1	6
5	9	1	7	3	8
9	6	1	7		
5					

Matrix A_{tjds}

val( )	2	3	4	1	1	6;	5	9	1
	7	3	8;	9	6	1	7;	5;	
col_ind( )	2	1	4	1	1	2;	4	2	5
	3	3	3;	5	4	6	4;	6;	
tjd_ptr( )	1	7	13	17					

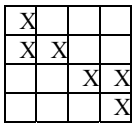
Figure 6. Transposed jagged diagonal storage format

2.2. Block based storage formats

2.2.1. **The block coordinate storage (BCOO).** The BCOO format is non-general sparse matrix format, which is used when the sparsity pattern of a sparse matrix is such that it comprises of dense sub-blocks. In BCOO a non-zero rectangular array *BA()* stores non-zero blocks in row fashion, *row_ind()* array stores the row indices of the first element of each nonzero block and *col_ind()* array stores the column indices of the first element of each nonzero block [20].

1	1	0	0	0	0	0	0
0	2	0	0	0	0	0	0
3	3	3	0	0	0	0	0
4	4	4	4	0	0	0	0
0	0	0	0	5	5	0	0
0	0	0	0	0	6	6	0
0	0	0	0	0	0	7	7
0	0	0	0	0	0	8	8

Matrix A



Block Matrix with block size = 2

BA()	1	1	0	2;	3	3	4	4;	3	0	4	4;
	5	5	0	6;	0	0	0	6;	7	7	8	8;

val ( )	1	5	9	13	17	21
col_ind ( )	1	1	2	3	4	4
row_ind ( )	1	2	2	3	3	4

Figure 7. The block coordinate storage format

2.2.2. **The block compressed row storage (BCRS).** This format is an extension of the CRS format with the difference that the elements in *val()* array are pointers to non-zero blocks which are stored row-wise in another array (Block array (*BA*)) that contains all the non-zero dense sub-blocks. BCRS format requires four arrays *{BA, val, col_ind, row_blk}*. *BA()* stores the non zero blocks in row wise fashion, *col_ind()* stores the column indices of the first element of each nonzero block and a pointer array *row_blk()* stores the pointer to each block row in the matrix [12].

If N_b is the dimension of each block and N_{nzb} is the number of non zero blocks in the n x n matrix A, then the total storage needed is $(N_{nze} = N_{nzb} * N_b^2)$ [1].

BCRS is useful when the sparse matrix is comprised of square dense blocks of non-zeros in some regular pattern. This format can offer significant cost savings when the sub-blocks are large since we eliminate the need to explicitly store the positional information for each non-zero element. Apart from the usefulness of BCCS, the size of the dense sub-blocks often has to be pre-calculated in order to choose its optimal value and this induces an additional overhead for creating the matrix stored in this way.

BA ()	1	1	0	2;	3	3	4	4;	3	0	4	4;
	5	5	0	6;	0	0	0	6;	7	7	8	8;

val ( )	1	5	9	13	17	21
col_ind ( )	1	1	2	3	4	4
row_blk ( )	1	2	4	6	7	

Figure 8. Block compressed row storage format of matrix A given in figure 7

2.2.3. **The block compressed column storage (BCCS).** This format is an extension of the CCS format. Similar to he CCS, BCCS format requires three arrays *{val, row_ind, col_blk}*. *val()* stores the non zero blocks in column wise fashion. An array *row_ind()* stores the row indices of the first element of each nonzero block and a pointer array *col_blk()* stores the pointer to each block column in the matrix. The advantages and disadvantages of BCCS are those of BCRS [20].

BA ( )	1	0	1	2;	3	4	3	4;	3	4	0	4;
	5	0	5	6;	0	6	0	0;	7	8	7	8;

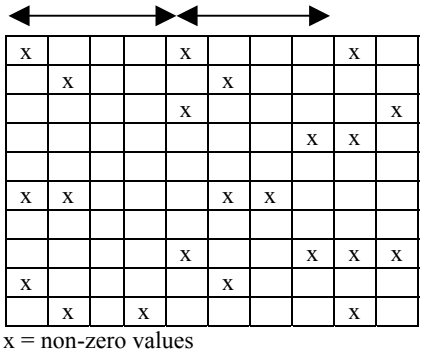
val ( )	1	5	9	13	17	21
row_ind ( )	1	2	2	3	3	4
col_blk ( )	1	3	4	5	7	

Figure. 9. Block compressed column storage format of matrix A given in figure 7

2.2.4. **The block based compression storage (BBCS).** In block based compression format, the matrix is partitioned in Vertical Blocks (VBs). For each vertical block, all the non-zero elements are stored row-wise in increasing row number order [15, 5]. An example of such a partitioning is graphically depicted in Figure 10. Each VB is stored in the main memory as a sequence of 6-tuple entries. The fields of such a data entry are as follows [15, 21, 22]:

1. **Value**: It specifies the value of a non-zero elements of matrix if ZR = 0. Otherwise it denotes the number of subsequent block rows with no nonzero matrix elements.
2. **Column-Position (CP)**: It specifies the matrix element column number within the block. Thus for a matrix element a_{ij} within the vertical block of size m, it is computed as j mod m.
3. **End-of-Row Flag (EOR)**: EOR is 1 when the current data entry describes the last non-zero element of the current block row and 0 otherwise.
4. **Zero-Row Flag (ZR)**: ZR is 1 when the current block row contains no nonzero value and 0 otherwise. When this flag is set, the Value field denotes the number of subsequent block rows that have no non-zero values.
5. **End-of-Block Flag (EOB)**: When EOB is 1, it indicates that the current matrix element is the last non-zero one within the VB.
6. **End-of-Matrix Flag (EOM)**: EOM is 1 only at the last entry of the last VB of the matrix.

The entire matrix is stored as a sequence of VBs and there is no need for an explicit numbering of the VBs.



x = non-zero values

Matrix A with vertical block of size 4

	1	2	3	4	5	6
x	1	1	0	0	0	0
x	2	1	0	0	0	0
3	-	1	1	0	0	0
x	1	0	0	0	0	0
x	2	1	0	0	0	0
2	-	1	1	0	0	0
x	1	1	0	0	0	0
x	2	0	0	0	0	0
x	4	1	0	1	0	0

Storage of First Vertical Block

Figure 10. Block based compressed storage format

When compared with other sparse matrix representation formats, BBCS requires a lower memory overhead and bandwidth since the index values associated with each non zero element are restricted within the VB boundaries.

The BBCS has two main disadvantages [5]:

- **Indexed loads/stores**. BBCS format reduces the amount of indexed memory accesses that need to be performed.
- **Flexibility**. Another problem related to BBCS is the inflexibility of altering the contents of the matrix.

2.2.5. **The Hierarchical Sparse Matrix Storage (HiSM).** The HiSM format is designed by taking into consideration the advantages and disadvantages that the BBCS format offers [18]. The HiSM format has a more intuitive organization that retains the advantages of BBCS and alleviates its disadvantages. This is mainly achieved in two ways [7, 21]:

(a) The matrix is partitioned in such a way that each partition block has a limited amount of rows and columns in order to alleviate the indexed memory accesses.

(b) A layered description of positional information is introduced in order to make the structure more flexible to changes.

The entire matrix is divided hierarchically into blocks with the lowest level containing the actual value of the nonzero elements and the higher levels containing pointers to the non-empty blocks of one level lower. The advantages that the format offers are low storage requirements, a flexible structure for element manipulations and allowing for efficient operations [7].

2.3. Java Sparse Array (JSA)

The Java Sparse Array (JSA) format is a new concept for storing sparse matrices that is unique for Java [10]. JSA has been created to exploit Java's flexible definition of multi-dimensional arrays. In Java every array is an object storing either primitive types or other objects [11]. A two-dimensional array is formed as an array of arrays. This definition enables developers to create both rectangular and jagged arrays. JSA is a row oriented storage format similar to CSR. It uses two arrays, each element of which is itself an array (object). One of these arrays is for storing references to the *val()* arrays (one for each row) and the other is for storing references to *index()* arrays (one for each row) [8]. The memory requirements to store a sparse matrix in JSA are $2N_{nze} + 2N$ array locations [11].

3. Comparison

Current storage formats, are either too specific for their application area and types of matrices (non-general) or suffer from several drawbacks including the following [12, 23, 4, 24, 25]:

- **Short vectors**: This problem is particularly evident in CRS but it is a more general phenomenon that is directly related to the fact that most sparse matrices only have a small number of non-zero elements in each row or column. Small means that when to construct a vector by using the non-zero elements in a row and operate on it, the overhead cost is significant. The JD storage format successfully deals with this problem when the matrix has sufficiently large dimensions since a vector of length of the order of the dimension of the matrix is obtained. However it does not deal with the other problems, moreover it is specially designed for sparse matrix vector multiplication and not efficient in other operations.

- **Indexed Accesses**: This problem is particular to JD and CRS, both general storage formats. The problem is related to the fact that since no assumption is made of the sparsity pattern, the positional information of each non-zero element (e.g. the column position in CRS and JD) is stored explicitly. This results in indexed memory accesses whenever one needs to access other elements using this positional information. This problem is mainly addressed by non-general storage formats such as BCRS and CDS that partly store data with implicit positional information.
- **Positional Information Storage Overhead**: This problem is related to the previously described problem. However, the focus here is on the overhead of storing the positional information. Formats like CRS and JD store the column position for each element (the row position is implied by the index used to access the row) which requires a full word extra storage for each non-zero element.

3.1. Format Storage Efficiency

The issue of storing the additional positional information is one of the causes of inefficient operation on sparse matrices [27]. Reducing the amount of positional information reduces the memory bandwidth allowing better resource utilization especially in memory bound operations.

- COO is an inefficient storage scheme as compared to the rest of the schemes that are available because of the amount of the indexing involved in accessing the elements of the matrix [12].
- CRS storage structure saves space on disk because unlike COO that stores one entry per nonzero element in row vector, CRS stores only one element for each row of the matrix [26].
- CCS storage will have a definite advantage over COO, but loose to CRS, because of less compression achieved in column vector of CCS than compression in row vector of CRS [1].
- JDS is more space efficient than CDS at the cost of indexing (gather/scatter) operations [12].
- TJDS format needs less storage space than the JDS format because the permutation array is not required [17, 18].
- The larger the dimension of the block in BCRS, the higher is the number of zeros to be stored, which requires more disk space [12].
- BBCS outperforms other general schemes like JDS and CRS by 1.7 to 4.1 times [12].
- Block Based Compression Storage (BBCS) format and Hierarchical Sparse Matrix (HiSM) storage require 72% to 78% of the storage space needed for Compressed Row Storage (CRS) or the Jagged Diagonal (JD) storage [12].
- HiSM format can achieve 40% reduction of storage space when comparing to the Compressed Row Storage (CRS) and Jagged Diagonal (JD) storage methods [26].
- JSA is competitive with traditionally CRS scheme on matrix computation routines and gives flexibility without loosing efficiency [10].

4. Conclusions

This paper reviews the storage formats for sparse linear systems. The main property of all sparse matrix storage formats is that they try to minimize the amount of zeros stored and at the same time provide an efficient way of accessing and operate on non-zero elements. Doing so (a) decreases the storage space and the bandwidth requirements for storing and accessing the matrix respectively and (b) allows avoiding the trivial operations on zero elements [28, 29].

The sparse matrix storage formats, COO, CRS and CCS, are quite similar to each other with the difference in column and row vector. CRS has minimal storage requirements. CDS can give a considerable performance benefit if the matrix is banded. TJDS format needs less storage space than the JDS format because the permutation array is not required [12, 26, 30].

The block entry storage formats do not perform significantly better with different block size. Both BBCS and HiSM achieve a considerable performance speedup when compared to CRS and JD [11, 12, 26]. JSA is competitive with traditionally CRS scheme on matrix computation routines and gives flexibility without loosing efficiency [10, 11].

5. References

[1] R. Barrett et al., *Templates for the solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

[2] Y. Saad, "SPARSKIT: A Basic Toolkit for Sparse Matrix Computations", *Technical Report*, Computer Science Department, University of Minnesota, June 1994.

[3] A. Pinar and M. Heath, "Improving Performance of Sparse Matrix-Vector Multiplication", *In Proceedings of Supercomputing*, 1999.

[4] S. Toledo, "Improving Memory-System Performance of Sparse Matrix-Vector Multiplication", In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.
[5] S. Vassiliadis, S.D. Cotofana, and P.T. Stathis, "Block Based Compression Storage Expected Performance", *In Proceedings of HPCS2000*, Victoria, March 2000, pp. 389-406.
[6] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee., "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply", *In Proceedings of the IEEE/ACM Conference on Supercomputing*, 2002.
[7] P.T. Stathis, S. Vassiliadis, and S.D. Cotofana, "A Hierarchical Sparse Matrix Storage Format for Vector Processors", *In Proceedings of 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003, pp. 61
[8] Geir Gundersen, *The Use of Java Arrays in Matrix Computation*, Cand. Scient Thesis, University of Bergen, Bergen, Norway, April 2002.
[9] Eun-Jin Im and K. Yelick, "Optimizing Sparse Matrix Vector Multiplication on SMPs", *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.
[10] G. Gundersen and T. Steihung, " Data Structures in Java for Matrix Computations", *Concurrency and Computation: Practice and Experience*, Volume 16, Number 8, July 2004, pp. 799-815.
[11] Mikel Lujan, Anila Usman, T.L. Freeman, Patrick Hardie, and John R. Gurd, "Storage Formats for Sparse Matrices in Java", *In Proceedings of the International Conference on Computational Sciences-ICCS 2005*, Lecture Notes in Computer Sciences. Springer-Verlag, 2005. To appear.
[12] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington, "A Sparse Matrix Library in C++ for High Performance Architectures", *Proceedings of the Second Object Oriented Numerics* Conference, 1994, pp. 214-218.
[13] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghil, "Compiling Imperfectly Nested Sparse matrix Codes with Dependencies", *Technical Report TR2000-1788*, Cornell University, Computer Science, March 2000.
[14] Manuel Ujaldon, Shamik Sharma, Joel H. Saltz, Emilio Zapata, "Runtime Techniques for Parallelizing Sparse Matrix Applications", *Proceedings of the Parallel Algorithms for Irregularly Structured Problems, Second International Workshop, IRREGULAR '95*, 1995, pp. 43-57.
[15] S. D. Cotofana, P.T. Stathis and S. Vassiliadis, "Direct and Transposed Sparse Matrix-Vector Multiplication", In *Proceedings of the 2002 Euromicro conference on Massively-parallel computing systems, MPCS-2002*, Ischia, Italy, April 2002, pp. 1-9.
[16] N. Goharian, T. El-Ghazawi, D. Grossman, "Enterprise Text Processing: A Sparse Matrix Approach", *Proceedings of the IEEE Int. Conference on Information Technology, Computing and Coding (ITCC01),* LV, 2001.
[17] A. Ekambaram and E. Montagne, "An Alternative Compressed Storage Format for Sparse Matrices", *ISCIS XVIII - Eighteenth International Symposium on Computer and Information Sciences, LNCS 2869*, November 2003, pp. 196-203.
[18] E. Montagne and Anand Ekambaram, "An Optimal Storage Format for Sparse Matrices", *Information Processing Letters*, Elsevier Science Publishers, Volume 90, Issue 2, April 2004, pp. 87-92.
[19] Roman Geus and S. R ollin, " Towards a Fast Parallel Sparse Matrix-Vector Multiplication", *In Proceedings of the International Conference on Parallel Computing (ParCo),* Imperial College Press, 1999, pp. 308-315.
[20] V. Kotlyar, K. Pingali, and P. Stodghill, "Compiling Parallel Code for Sparse Matrix Applications", *In Supercomputing*, 1997.
[21] P.T. Stathis, S. Vassiliadis, and S.D. Cotofana, "Design Considerations of a Multiple Inner Product and Accumulate Vector Functional Unit", *Proceedings of ProRISC 2002,* Veldhoven, the Netherland, January 2002, pp. 481-485.
[22] P. Stathis, S. Vassiliadis, and S. Cotofana, "Sparse Matrix Vector Multiplication Evaluation Using the BBCS Scheme", *In 8th PCI Proceedings*, November 2001.
[23] Nikolay Mateev, *A Generic Programming System for Sparse Matrix Computations,* PhD thesis, Cornell University, 2000.
[24] W. Pugh and T. Shpeisman, "Generation of efficient code for sparse matrix computations", *In Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing, LNCS*, August 1998.
[25] Wagdi G. Habashi, Laura C. Dutto, Claude Y. Lepage , "Effect of the Storage Format of Sparse Linear Systems on Parallel CFD Computations", *Comput. Methods Appl. Mech. Engrg*., 2000, pp. 441-453.
[26] Eduardo F. D, Mark R. Fahey and Richard Tran Mills, " Vectorized Sparse Matrix Multiply for Compressed Row Storage Format", *International Conference on Computational Science (1)*, May 2005, pp. 99-106.
[27] Eun-Jin Im, *Optimizing the Performance of Sparse Matrix - Vector Multiplication*, PhD thesis, University of California at Berkeley, May 2000.
[28] Goharian, N., E1-Ghazawi, T., Grossman, D., and Chowdhury, A. "On the Enhancements of a Sparse Matrix Information Retrieval Approach", *Proceedings of the International Conference on Parallel and distributed Processing Techniques and Applications*, 1999.
[29] Manuel Ujaldon, Shamik Sharma, Emilio Zapata and Joel H. Saltz, "Experimental Evaluation of Efficient Sparse Matrix Distributions", *Proceedings of the 1996 International Conference on Supercomputing*, 1996, pp. 78-86.
[30] Y. Saad, E.C. Anderson, " Solving Sparse Triangular Systems on Parallel Computers", International J. High Speed Comp., 1989, pp. 73-96.