Blocked-Based Sparse Matrix-Vector Multiplication on Distributed Memory Parallel Computers

Rukhsana Shahnaz and Anila Usman
Department of Computer and Information Science, Pakistan Inst. of Eng. and Applied Sciences, Pakistan

Abstract: *The present paper discusses the implementations of sparse matrix-vector products, which are crucial for high performance solutions of large-scale linear equations, on a PC-Cluster. Three storage formats for sparse matrices compressed row storage, block compressed row storage and sparse block compressed row storage are evaluated. Although using BCRS format reduces the execution time but the improvement may be limited because of the extra work from filled-in zeros. We show that the use of SBCRS not only improves the performance significantly but reduces matrix storage also.*

Keywords: *Matrix-vector product, compressed storage formats, sparse matrix data structures, locality of matrix, parallel matrix computation, and block-based compressed storage.*

Received September 24, 2008; accepted May 17, 2009

1. Introduction

The performance of diverse applications in scientific computing, economic modeling, and information retrieval, among others, is dominated by Sparse Matrix-Vector Product (SMVP), $y \leftarrow y + A \times x$, where A is a sparse matrix, x and y are dense vectors. A variety of sparse matrix storage schemes [1, 5] are used to store and manipulate sparse matrices. These different data structures typically necessitate non-contiguous multiple memory system accesses which hinders performance. Furthermore, these multiple indirect accesses are difficult for the compiler to optimize, resulting in poor performance. Thus, investigating other approaches to efficiently store sparse matrices could be beneficial. Conventional implementations using Compressed Sparse Row (CSR) format storage usually run at 10% of machine peak or less on uniprocessors [10]. Higher performance requires a compact data structure and appropriate code transformations that best exploit properties of both the sparse matrix and the underlying machine architecture.

To achieve high performance on PC-cluster, it is desirable to increase both spatial and temporal locality of an application. The idea behind the temporal locality is to reuse as much as possible each data element that is brought into the memory system. Whereas the idea behind the spatial locality is the use of every element of data brought to the memory system. To increase temporal locality, blocking the computation [3] could be a beneficial choice. Here, instead of operating on entire rows or columns of a matrix, blocked algorithms operate on sub-matrices or, data blocks such as in Block Compressed Row Storage (BCRS). The goal is to maximize accesses to the data loaded into the cache before it is replaced. On the other hand, the way to ensure spatial locality is to access data contiguously in a data structure according to the memory layout of that data structure.

In this paper, we will show the performance improvement due to Sparse Block Compressed Row Storage (SBCRS) [8], the use of which increases both the spatial and temporal locality. The paper presents the parallel implementation of SMVP using three different storage formats on heterogeneous clusters [4, 9]. The experimental results obtained are presented and discussed.

The remaining paper is organized as follows: In section 2 we briefly present the CRS, BCRS and SBCRS formats and their algorithms for matrix-vector multiplication. Section 3 is about the parallel implementation of matrix-vector product. The experimental results and performance analysis is presented in section 4. Finally, in section 5, we give conclusions.

2. Sparse Storage Formats and their Implementations

The efficiency of an algorithm for the solution of linear system is determined by the performance of matrix-vector multiplication that depends heavily on the storage scheme used. A number of storage schemes have been proposed for sparse matrices. They have been proposed for various objectives, such as

simplicity, generality, performance, or convenience with respect to a specific algorithm.

In our previous work six storage formats including Coordinate Storage (COO), Compressed Row Storage (CRS), Compressed Column Storage (CCS), Jagged Diagonal Storage (JDS), Transposed Jagged Diagonal Storage (TJDS) and Bi-Jagged Diagonal Storage (Bi-JDS) were implemented and compared [6,7].

In this paper the data structures and the parallel implementations of matrix-vector product of one point based storage format, CRS and two block based storage formats BCRS and SBCRS are presented and analyzed. Figure 1 shows an example matrix and data structures of CRS, BCRS and SBCRS for this matrix.

$$\begin{pmatrix} 6 & 0 & 9 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 5 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 \end{pmatrix}$$

(a) Example matrix.

$$value = \{6\ \ 9\ \ 4\ \ 4\ \ 5\ \ 3\ \ 5\ \ 8\ \ 6\ \ 5\ \ 4\ \ 3\ \ 2\ \ 2\}$$
$$col\_ind = \{1\ \ 3\ \ 6\ \ 6\ \ 2\ \ 3\ \ 4\ \ 5\ \ 5\ \ 6\ \ 6\ \ 7\ \ 7\ \ 8\}$$
$$row\_ptr = \{1\ \ 4\ \ 5\ \ 6\ \ 9\ \ 10\ \ 11\ \ 13\ \ 14\}$$

(b) CRS format.

$$value = \{6\ \ 0\ \ 0\ \ 0\ \ 9\ \ 0\ \ 0\ \ 0\ \ 0\ \ 4\ \ 0\ \ 4\ \ 0\ \ 5\ \ 0\ \ 0\ \ 0\ \ 0$$
$$3\ \ 5\ \ 0\ \ 0\ \ 8\ \ 0\ \ 6\ \ 0\ \ 0\ \ 5\ \ 0\ \ 4\ \ 0\ \ 0\ \ 3\ \ 0\ \ 2\ \ 2\}$$
$$bcol\_ind = \{1\ \ 2\ \ 3\ \ 1\ \ 2\ \ 3\ \ 3\ \ 3\ \ 4\}$$
$$brow\_ptr = \{1\ \ 4\ \ 7\ \ 8\ \ 10\}$$

(c) BCRS format with (2x2) block size.

$$block\_array = \begin{Bmatrix} 6 & 9 & 4 & 4 & 5 & 3 & 5 & 8 & 6 & 5 & 4 & 3 & 2 & 2 \\ 1 & 1 & 1 & 2 & 1 & 2 & 2 & 2 & 1 & 2 & 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 & 2 & 1 & 2 & 1 & 1 & 2 & 2 & 1 & 1 & 2 \end{Bmatrix}$$
$$block\_ptr = \{1\ \ 2\ \ 3\ \ 5\ \ 6\ \ 8\ \ 9\ \ 11\ \ 12\}$$
$$bcol\_ind = \{1\ \ 2\ \ 3\ \ 1\ \ 2\ \ 3\ \ 3\ \ 3\ \ 4\}$$
$$brow\_ptr = \{1\ \ 4\ \ 7\ \ 8\ \ 10\}$$

(d) SBCRS format with (2x2) block size.

Figure 1. The data structures.

2.1. The Compressed Row Storage

The 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. The CRS format is specified by the arrays *{value, col_ind, row_ptr}*. The double-precision array *value( )* contains the non-zero matrix elements taken in a row-wise fashion, *col_ind( )* contains the column positions of the corresponding elements in *value( )* and *row_ptr( )* contains pointers to the first non-zero element of each row in array *value( )* with *row_ptr(N+1) = nze* where *nze* is the total number of nonzeros.

2.1.1. SMVP in CRS

The algorithm to perform the matrix-vector product $Ax = y$ using the CRS format is shown in Figure 2(a). It is parallelized at the outer loop, and thus the computation over the rows of matrix is distributed to different processors.

2.2. Block Compressed Row Storage

For BCRS, the matrix is split into *br*×*bc* submatrices (called blocks), where *br* and *bc* are fixed integers. BCRS stores the non-zero blocks (submatrices with at least one non-zero element) in a manner similar to CRS. Let *brows = rows/br* and *bnze* be the number of non-zero blocks in the matrix. BCRS is shown by three arrays {*value, bcol_ind, brow_ptr*}. The double precision array *value( )* of length *bnze*×*br*×*bc* stores the elements of the non-zero blocks: the first *br*×*bc* elements are of the first nonzero block, and the next *br*×*bc* elements are of the second non-zero block, etc. The integer array *bcol_ind( )* of length *bnze* stores the block column indices of the non-zero blocks. The integer array *brow_ptr( )* of length *(brows+1)* stores pointers to the beginning of each block row in the array *bcol_ind( )*.

2.2.1. SMVP in BCRS

The algorithm to perform parallel matrix–vector product for BCRS is shown in Figure 2(b). A larger *br* reduces the number of load instructions for the elements of the vector *x*, and a larger *bc* works as the unrolling of the inner loop, but this wastes memory and CPU power because of the zero elements in the non-zero blocks.

2.3. Sparse Block Compressed Row Storage

The SBCRS format is similar to the BCRS regarding its relation with CRS. The format is constructed as follows: The Sparse matrix is divided into *(S×S)* sized blocks. The SBCRS structure consists of two parts: (a) A CRS-like structure where the elements are pointers to non-empty S^2*-blockarrays* representing non-empty *(S×S)* sparse blocks in the matrix and (b) the collection of all S^2*-blockarrays*. In essence, this is a variation of the BCRS, where the blocks are not assumed to be dense and of arbitrary size, but rather are *(S×S)* sparse blocks. All the non-zero values as well as the positional information combined are stored in a row-wise fashion in an array (S^2*-blockarray*) in memory. The positional data consists of column and row position of the non-zero elements within the sub-matrix or block (*row_pos* and *col_pos*). Thus if we choose $S<256$, we only need to store 8-bits for each

row and column position. This is significantly less than other sparse matrix storage schemes where at least a 32-bit entry has to be stored for each non-zero element. Three more arrays {*block_ptr, bcol_ind, brow_ptr*} are required for specifying SBCRS format. The two arrays *bcol_ind( )* and *brow_ptr( )* are similar as in BCRS while third array *block_ptr( )* contains the pointers to the S^2-*blockarrays*.

2.3.1. SMVP in SBCRS

The algorithm to perform the matrix-vector product using the SBCRS format is shown in Figure 2(c) where *S* represents the size of the block and *row_pos( )* and *col_pos( )* are row and column index within block stored in 8-bits each.

```
for i = 0 to rows-1
  y ( i )=0
  for j = row_ptr ( i ) to ( row_ptr ( i+1 ) – 1 )
    y ( i ) = y ( i ) + value ( j-1 ) * x ( col_ind ( j-1 ) )
  end for j
end for i
```

(a) CRS.

```
z=0
for b = 0 to brows-1
   y ( b )=0
   for j = brow_ptr ( b ) to ( brow_ptr ( b+1 ) – 1 )
      for k = 0 to br-1
         for t = 0 to bc-1
            y ( bc*b+k ) = y (bc*b+k ) + val (z ) * x
( bc*bcol_ind ( j-1 )+t )
            z++
         end for t
      end for k
   end for j
end for b
```

(b) BCRS.

```
for b = 0 to brows-1
  y ( b ) = 0
  for j = brow_ptr ( b ) to ( brow_ptr ( b+1 ) – 1 )
    for z = block_ptr ( j ) to block_ptr ( j+1 ) - 1
      y(S*b+(row_pos(z)-1)) = y(S*b+(row_pos(z)-1))
     +block_array(z)*x(S*(bcol_ind(j)-1) +col_pos(z)-1)
    end for z
  end for j
end for b
```

(c) SBCRS matrix-vector product.

Figure 2. The algorithms.

3. Parallel Implementation

For parallel computers with distributed memory, we use a row-block distribution of matrix and vector elements among the processors involved in the matrix computation as shown in Figure 3. The row-block size is directly proportional to the computer relative computing power to achieve a good balance of workload.

Matrix elements requiring access to vector data stored on remote processors cause communication. Components corresponding to internal interface points will be sent to the neighboring processors and components corresponding to external interface points would be received from the neighboring processors. Local pre-processing on each processor is needed to facilitate the implementation of the communication tasks and to have good efficiency during the matrix-vector multiplication. For minimization of the communication cost the rows are permuted according to the column position of elements i.e., rows having common column indexes are combined on one processor. During the communication process we use non-blocking send-routines to have the possibility to continue with other useful work after sending the required blocks. This allows an efficient way of exchanging data as well as an overlapping of communication and computation.

4. Experimental Results and Performance Analysis

The experiments are carried out on a cluster of PCs interconnected with 100Mb/s Ethernet. All the PCs are installed with Linux operating system and MPICH2 is used as a message-passing programming model, which is considered as the most appropriate for parallel processing on clusters. We implemented the two block-based data structures on parallel processors and compared their performance with that of conventional compressed row storage format. We experimented with the collection of matrices selected from the Matrix Market [2]. Table 1 shows the characteristics of each test matrix used in the experiments.

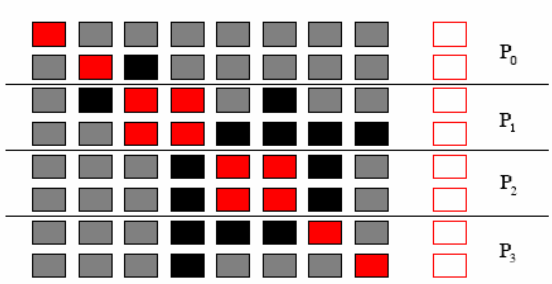


Figure 3. Distribution of matrix and vector elements on four processors. Matrix elements causing communication are marked black and local matrix elements are colored red.

Table 1. Test matrices for the experiments selected from Matrix Market [10].

#	Matrix Name	Dimension	Non-zeros	#	Matrix Name	Dimension	Non-zeros
1	Qc2534	2534 x 2534	463360	17	Bcsstk17	10974 x 10974	219812
2	Psmigr_1	3140 x 3140	543162	18	Bcsstk18	11948 x 11948	80519
3	Cavity16	4562 x 4562	138187	19	Fidap019	12005 x 12005	259863
4	Bcssth16	4884 x 4884	147631	20	Fidapm29	13668 x 13668	186294
5	Olm5000	5000 x 5000	19996	21	Bcsstk25	15439 x 15439	133840
6	Sherman3	5005 x 5005	20033	22	Fidap011	16614 x 16614	1091362
7	Rw5151	5151 x 5151	20199	23	E40r0000	17281 x 17281	553956
8	S3rmt3m3	5357 x 5357	106526	24	memplus	17758 x 17758	126150
9	S1rmt3m1	5489 x 5489	112505	25	Fidap035	19716 x 19716	218308
10	S1rmq4m1	5489 x 5489	143300	26	Fidapm11	22294 x 22294	623554
11	Fidap018	5773 x 5773	69335	27	Af23560	23560 x 23560	484256
12	Utm5940	5940 x 5940	83842	28	Bcsstk30	28924 x 28924	1036208
13	Fidap015	6867 x 6867	96421	29	Bcsstk31	35588 x 35588	608502
14	Dw8192	8192 x 8192	41746	30	Bcsstk32	44609 x 44609	1029655
15	Fidapm37	9152 x 9152	765944	31	S3dkt3m2	90449 x 90449	1921955
16	Fidapm15	9287 x 9287	98519	32	S3dkq4m2	90449 x 90449	2455670

The main theme behind the performance improvement here is to increase both spatial and temporal locality of an application. To increase temporal locality, blocking the computation [3] could be used. The goal is to maximize accesses to the data loaded into the cache before the data is replaced. The BCRS format stores the non-zero blocks (sub-matrices *br×bc* with at least one non-zero element) in a manner similar to CRS. As BCRS includes all the zero values within the block these extra filled-in zeros wastes memory and CPU power. Since a larger block size can result in more overheads, so when performance does not improve significantly we still reduce the overall storage. Hence there is an optimum block size for which BCRS outperforms CRS.

On the other hand SBCRS stores the non-zero blocks like BCRS where the blocks are not assumed to be dense and of arbitrary size rather they are sparse blocks i.e. no overhead of fill-in zeros is involved. The positional information of non-zero elements within the block is also stored in memory, which increases the spatial locality. As we choose *block_size<256*, we only require 8-bits for each row and column position. This is significantly less than other sparse matrix storage formats where at least a 32-bit entry has to be stored for each non-zero element. Since there is no overhead of extra zeros in SBCRS, increasing the size of block will reduce the storage appreciably as depicted in Figure 4.

Figure 5 shows the comparison of the subsidiary storage required for the selected matrices using the three storage formats CRS, BCRS and SBCRS with optimal block size *br×bc* for BCRS and *256×256* for SBCRS. The optimal block sizes for which the performance is better as compared to the other block sizes for the same format are written at the top of respective bars. It is seen that SBCRS is more space efficient than CRS and BCRS except for s3rmt3m3, as the structure of the matrix is such that the elements are clustered along the diagonal and for this particular block size (3x3) there is very less overheads of fill-in zeros.

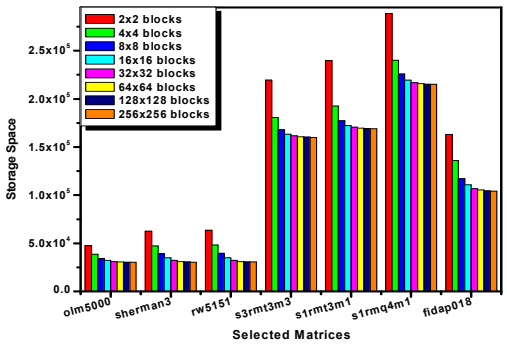


Figure 4. Storage space required by SBCRS format for different block sizes.

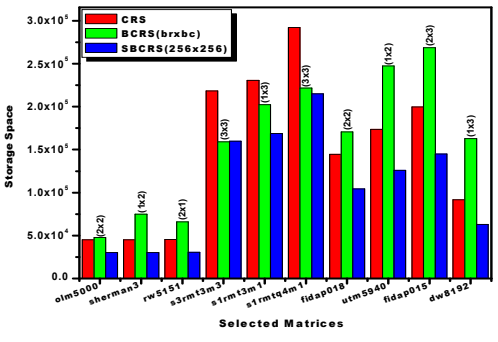
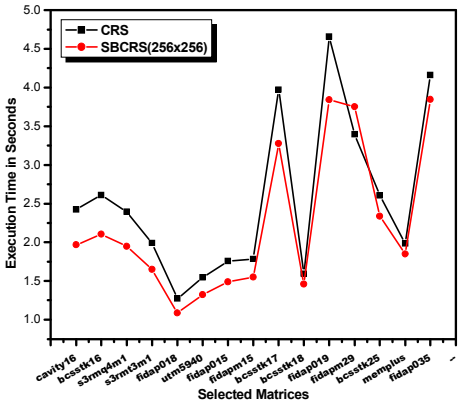
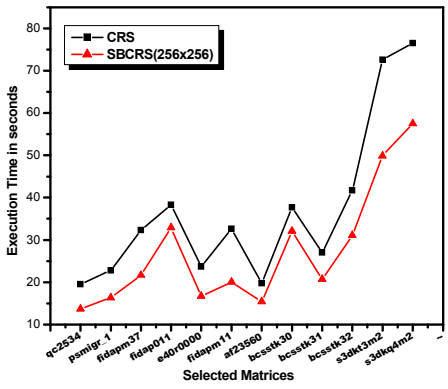


Figure 5. Storage space required by CRS, BCRS and SBCRS.

For BCRS the best performance is attained by choosing the optimum block size mentioned as *br×bc*. As mentioned in Figure 4, the optimum block size is different for different matrices. The BCRS performance is much better than CRS for diagonal matrices. As these matrices have high locality value i.e., the elements are clustered along the diagonal and are not scattered uniformly around the matrix, which leads to less overhead and most of the blocks (sub-matrices) are dense.



(a) For small.

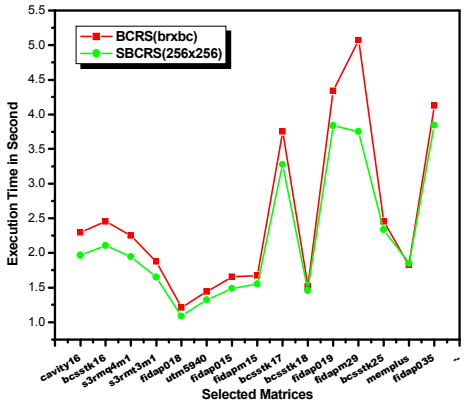


(b) For large.

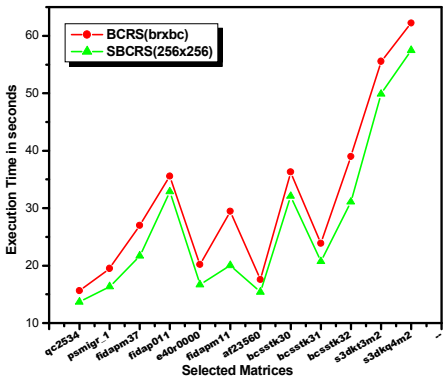
Figure 6. Execution times (in seconds) of matrix-vector products of CRS and SBCRS formats.

Figures 6 and 7 show the execution times in seconds for 1000 iterations of matrix vector products in various storage formats. The performance of SBCRS is appreciable as compared to CRS as shown in Figure 6 and BCRS as shown in Figure 7 as the large block size reduces the number of load instructions for the elements of the vector *x*. SBCRS outperforms CRS for matrices fidapm37, fidapm11 as they are less sparse i.e., they have large value of non-zeros and larger the data in the block reduces number of blocks and also number of instruction to load and store the two vectors (multiplier and resultant) respectively. For matrix fidapm29 the performance of SBCRS reduces as compare to CRS because of high locality (nonzeros spread throughout the matrix) and consequently low clustering.

The performance of SBCRS for matrices s3dkt3m2 and s3dkq4m2 is nearly same as BCRS (less percentage improvement as compare to overall percentage improvement) due to reason that matrix elements are clustered along the diagonal and the extra work from filled-in zeros is almost ineffective and also the large blocks of SBCRS lie on the diagonal producing equal effectiveness due to reduction in load store instructions. The results show that the use of SBCRS not only reduces matrix storage as shown in Figure 5 but also improves the performance significantly as shown in Figure 6 and Figure 7. These results lead to the observation that the performance is improved by optimizing the choice of the matrix storage format and the best storage format differs for different matrices.



(a) For small.



(b) For large.

Figure 7. Execution times (in seconds) of matrix-vector products of BCRS and SBCRS formats.

Figure 8 shows the execution time in seconds for 1000 iterations of matrix vector products using SBCRS for different number of processors. Initially the parallelization is performed on cluster of eight processors, which will be further extended for larger clusters in future.

The speed-up ratios for the parallel matrix-vector products using SBCRS are shown in Table 2. The parallelization speed-ups for the matrix-vector product are nearly ideal in most cases while some super linear speed-ups are observed for P=8 may be due to the

reason that smaller data size increases the data locality much which improves the cache hits. The parallel implementation on more than eight processors is under way.

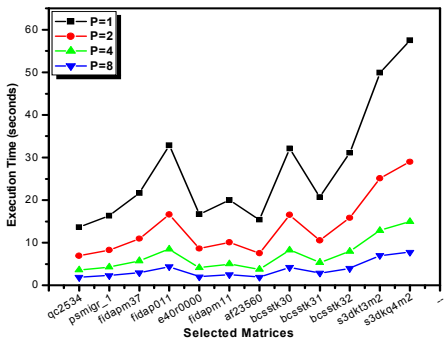


Figure 8. Execution times (in seconds) of 1000 iterations of SBCRS matrix-vector products on different number of processors.

Table 2. Speed-up ratios for parallel matrix vector products.

#	Matrix Name	P=1	P=2	P=4	P=8
1	E40r0000	1.00	1.93	3.95	8.18
2	Fidapm11	1.00	1.98	3.99	9.43
3	Bcsstk30	1.00	1.90	3.42	7.62
4	Bcsstk31	1.00	1.96	3.85	7.55
5	Bcsstk32	1.00	1.96	3.89	7.76
6	S3dkt3m2	1.00	1.98	3.87	7.10

5. Conclusions

In the present paper, we have discussed the parallel performance of matrix-vector product routine that are crucial for high performance implementation of iterative linear solvers for SBCRS storage format on a PC-cluster and is compared with CRS and BCRS. Although using BCRS format reduces the execution time but the improvement may be limited because of extra work from filled-in zeros. The use of SBCRS improves the performance significantly and reduces matrix storage also. The SBCRS format performs better for large block size as the large block size reduces the number of load and store instructions for the elements of the vectors. To achieve high performance, it is desirable to increase both spatial and temporal locality of an application. SBCRS increases both the temporal and spatial localities by using sparse blocking. It outperforms for the diagonal and banded diagonal matrices while its performance improvement is very less for matrices with high locality (zeros are spread throughout the matrix).

References

[1] Barrett R., Berry M., Chan T., Demmel J., Donato J., Dongarra J., Eijkhout V., Pozo R., Romine C., and Vorst H., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, 1994.

[2] Boisvert R., Pozo R., *Remington K., Barrett R., and Dongarra J., The Matrix Market: A Web Resource for Test Matrix Collections*, Chapman and Hall, London, pp. 125-137, 1997.

[3] Pinar A. and Heath M., "Improving Performance of Sparse Matrix-Vector Multiplication," *in Proceedings of Supercomputing*, Portland, pp. 159-163, 1999.

[4] Rosenberg A., "Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better," *in Proceedings of 3rd IEEE International Conference on Cluster Computing*, pp. 124-124, 2001.

[5] Saad Y., *Iterative Methods for Sparse Linear Systems*, SIAM, USA, 2003.

[6] Shahnaz R. and Usman A., "Implementation and Evaluation of Sparse Matrix-Vector Product on Distributed Memory Parallel Computers," *in Proceedings of Cluster IEEE International Conference on Cluster Computing*, Barcelona, pp. 1-6, 2006.

[7] Shahnaz R. and Usman A., "An Efficient Sparse Matrix Vector Multiplication on Distributed Memory Parallel Computers", Computer *International Journal of Computer Science and Network Security*, vol. 7, no. 1, pp. 77-82, 2007.

[8] Smailbegovic F., Gaydadjiev G., and Vassiliadis S., "Sparse Matrix Storage Format," in *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, pp. 445-448, 2005.

[9] Tinetti F., "Performance of Scientific Processing in NOW: Matrix Multiplication Example," *Journal of Computer Science and Technology*, vol. 1, no. 4, pp. 78-87, 2001.

[10] Vuduc R., Demmel J., Yelick K., Kamil S., Nishtala R., and Lee B., "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply," i*n Proceedings of the IEEE/ACM Conference on Supercomputing*, pp. 458-463, 2002.

Rukhsana Shahnaz received PhD scholar in the Department of Computer and Information Sciences, Pakistan Institute of Engineering and Applied Sciences. She is being funded by Higher Education Commission of Pakistan under a PhD fellowship program. Her research interests include performance improvement of sparse systems and cluster computing.

Anila Usman is a faculty member in the Department of Computer and Information Sciences, Pakistan Institute of Engineering and Applied Sciences since 1991. Currently, she is serving as the head of Department of Computer and Information Sciences. She received PhD in numerical computing from the University of Manchester, UK in 1998, and post doctorate from the same university in 2004, on high performance computing. Her research interests include numerical computing and parallel/distributed computing.