An Efficient Sparse Matrix-Vector Multiplication on Distributed Memory Parallel Computers

Rukhsana Shahnaz and Anila Usman,

Pakistan Institute of Engineering and Applied Sciences (PIEAS), Islamabad, Pakistan.

Summary
The matrix-vector product is one of the most important computational components of Krylov methods. This kernel is an irregular problem, which has led to the development of several compressed storage formats. We design a data structure for distributed matrix to compute the matrix-vector product efficiently on distributed memory parallel computers using MPI. We conduct numerical experiments on several different sparse matrices and show the parallel performance of our sparse matrix-vector product routines.

Key words:
Sparse matrices, matrix-vector product, sparse storage formats, distributed computing.

1. Introduction

Fast solution of linear equations with large sparse coefficient matrices is an essential requirement of advanced computations. We are planning to develop a new library for large-scale sparse matrix solutions that features a wide range of storage formats for sequential and distributed memory parallel architectures.

Krylov subspace methods are currently ubiquitous as a tool for solving linear systems, especially for very large sparse matrices. The efficient use of modern supercomputers strongly depends on a parallel, fast and memory saving implementation of the matrix-vector multiplication. A fast and efficient parallelization of SMVM computations is desirable which requires the distribution of nonzeros of the input matrix among processors in such a way that the computational loads of the processors are almost equal and the cost of interprocessor communication is low [1].

This paper presents a parallelization strategy of the SMVM using Transposed Jagged diagonal storage (TJDS) format on heterogeneous cluster [2, 3]. Two basic guidelines are defined for the parallel algorithm: one-dimensional data distribution and broadcast messages for all data communications. One-dimensional data distribution eases the processing workload balance on heterogeneous clusters [4]. The use of broadcast messages for every data communication is directly oriented to optimize performance on the most common cluster interconnection, Ethernet. Experimental results obtained in a local network of heterogeneous computers are presented.

The remaining paper is organized as follows: In Section 2 we briefly present the parallel implementation of matrix-vector multiplication using TJDS storage format. The Experimental results and performance analysis is presented in section 3. Finally, in Section 4 we give conclusions.

2. Implementation of matrix-vector multiplication

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.

In our previous work five storage formats including Coordinate Storage (COO), Compressed Row Storage (CRS), Compressed Column Storage (CCS), Jagged Diagonal Storage (JDS) and Transposed Jagged Diagonal Storage (TJDS) [3, 5, 6] were implemented and compared [7]. The TJDS achieve the high performance on distributed memory parallel architecture.

2.1 The transposed jagged diagonal storage (TJDS) format

The Transposed Jagged Diagonal Storage (TJDS) format is inspired from the Jagged Diagonal Storage (JDS) format and makes no assumptions about the sparsity pattern of the matrix. To illustrate the principles of the scheme, we introduce a 8 x 8 matrix A with nonzero elements a_{ij}.

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & a_{36} & 0 & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ 0 & 0 & 0 & a_{54} & a_{55} & 0 & a_{57} & 0 \\ 0 & 0 & 0 & a_{64} & 0 & a_{66} & a_{67} & 0 \\ 0 & 0 & 0 & a_{74} & a_{75} & a_{76} & a_{77} & 0 \\ 0 & 0 & 0 & a_{84} & 0 & 0 & 0 & a_{88} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix}$$

Manuscript received December 5, 2006.
Manuscript revised December 25, 2006.

In TJDS all the non-zero elements are shifted upward instead of leftward as in JDS. This gives a new matrix A_{ccs}.

$$A_{ccs} = \begin{bmatrix} a_{11} & a_{22} & a_{23} & a_{34} & a_{45} & a_{36} & a_{47} & a_{48} \\ 0 & a_{32} & a_{33} & a_{44} & a_{55} & a_{46} & a_{57} & a_{88} \\ 0 & 0 & a_{43} & a_{54} & a_{75} & a_{66} & a_{67} & 0 \\ 0 & 0 & 0 & a_{64} & 0 & a_{76} & a_{77} & 0 \\ 0 & 0 & 0 & a_{74} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{84} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix}$$

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 and reordering the elements of the vector x accordingly as if it were an additional row of A.

$$A_{tjds} = \begin{bmatrix} a_{34} & a_{36} & a_{47} & a_{45} & a_{23} & a_{22} & a_{48} & a_{11} \\ a_{44} & a_{46} & a_{57} & a_{55} & a_{33} & a_{32} & a_{88} & 0 \\ a_{54} & a_{66} & a_{67} & a_{75} & a_{43} & 0 & 0 & 0 \\ a_{64} & a_{76} & a_{77} & 0 & 0 & 0 & 0 & 0 \\ a_{74} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{84} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} x = \begin{bmatrix} x_4 \\ x_6 \\ x_7 \\ x_5 \\ x_3 \\ x_2 \\ x_8 \\ x_1 \end{bmatrix} y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix}$$

The rows of the compressed and permuted matrix A_{tjds} are called transposed jagged diagonals. Obviously, the number of these diagonals is equal to the maximum number *max_nz* of nonzeros per column. A suitable data structure required to compute Ax = y using TJDS scheme is shown in Figure 1. The *num_nz* nonzero elements of the A_{tjds} matrix are stored in a floating point linear array *value(:)*, one row after another. Another array of same length *row_ind(:)*, is needed to store the row indices of the non-zero elements in the original matrix. Finally, a third array of length *max_nz+1* is also needed, *tjd_ptr(:)*, which stores the starting position of the transposed jagged diagonals in the array *value(:)*. Figure 2 shows the matrix A considered above in the TJDS format.

```
TJDS_Matrix  =  record
      value      : array [1..num_nz] of REAL
      row_ind  : array [1..num_nz] of INTEGER
      tjd_ptr    : array [1..max_nz+1] of INTEGER
      X          : array [1..n] of REAL
      Y          : array [1..n] of REAL
end record
```

Fig. 1 Data structure of a n x n matrix in the TJDS scheme

$$value: \begin{Bmatrix} a_{34} & a_{36} & a_{47} & a_{45} & a_{23} & a_{22} & a_{48} & a_{11} & a_{44} & a_{46} & a_{57} & a_{55} & a_{33} \\ a_{32} & a_{88} & a_{54} & a_{66} & a_{67} & a_{75} & a_{43} & a_{64} & a_{76} & a_{77} & a_{74} & a_{84} & \end{Bmatrix}$$

$$row\_ind: \begin{Bmatrix} 3 & 3 & 4 & 4 & 2 & 2 & 4 & 1 & 4 & 4 & 5 & 5 & 3 \\ 3 & 8 & 5 & 6 & 6 & 7 & 4 & 6 & 7 & 7 & 7 & 8 & \end{Bmatrix}$$

$$tjd\_ptr: \{1 \;\; 9 \;\; 16 \;\; 21 \;\; 24 \;\; 25 \;\; 26\}$$

$$x: \{x_4 \;\; x_6 \;\; x_7 \;\; x_5 \;\; x_3 \;\; x_2 \;\; x_8 \;\; x_1\}$$

$$y: \{y_1 \;\; y_2 \;\; y_3 \;\; y_4 \;\; y_5 \;\; y_6 \;\; y_7 \;\; y_8\}$$

Fig. 2 Matrix A in the TJDS scheme

2.2 TJDS Matrix-Vector Multiplication

The Matrix Vector Multiplication (MVM) is performed along the transposed jagged diagonals providing an inner loop length equal to the diagonal length. To minimize the indirect memory accesses, vector X involved in the MVM is permuted initially once so that it automatically carry the same order as the matrix rows. The numerical core of the MVM is given in Figure 3.

```
for j = 1, …, max_nz do
   p = 1
   for i = 1, …, (tjd_ptr ( j+1 ) - tjd_ptr ( j )) do
      y ( row_ind ( p ) ) = y ( row_ind ( p ) +
                                val ( j ) *   x ( p )
    p = p+1
    end for i
end for j
```

Figure 3. MVM in the TJDS format: y = y + Ax.

The innermost loop requires one store and four load operations (including one indirect load) to perform two floating-point operations (Flop). In other words the performance of the MVM is clearly determined by the quality of the memory access. To reduce the load operations associated with the vector y, we have splitted outer j loop into several loops over Transposed Jagged Diagonals with equal length by introducing an outer loop k.

```
for k = 1, …, diff_tjds do
  length=i_length(k)
  for j = j_start(k), …, j_end(k) do
    p = 1
    for i = 1, …, length do
      y ( row_ind ( p ) ) = y ( row_ind ( p ) +
                            val ( j ) *   x ( p )
      p = p+1
    end for i
  end for j
end for k
```

Figure 4. Modified MVM in the TJDS format with an outer k loop running over diff_tjds blocks of Transposed Jagged Diagonals with different loop lengths.

2. Parallel implementation

For parallel computers with distributed memory, we use a column-wise distribution of matrix and row-wise distribution of vector elements among the processors involved in MVM. Communication is required for all matrix elements that refers to vector data stored on remote processor (non-local vector elements) [8,9].

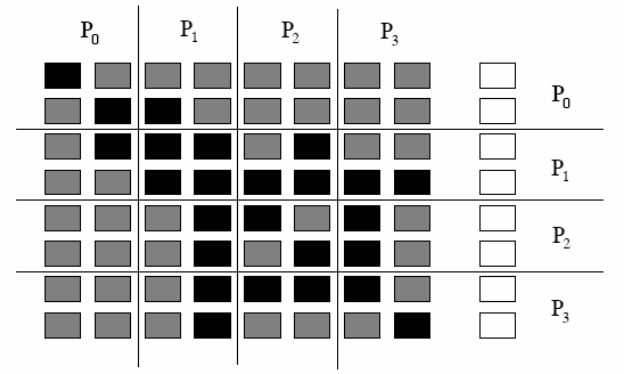

Figure 5. Distribution of a matrix A and vector X on 4 processors (P_0 – P_3). All nonzero matrix elements are black

Since the matrix does not change during the computation, a static communication scheme is predetermined beforehand. In this way, we can exchange data efficiently in anticipation of the overlapping of communication and computation in the each MVM. Table 1 shows a list of local variables involved in the communication.

Table 1. Variables representing the communication scheme

recnum	Number of processors from which vector elements are received
recid(recnum)	IDs of these processors
elrec(recnum)	Number of elements received from each of the recnum processors
sendnum	Number of processors to which vector elements are sent
sendid(sendnum)	IDs of these processors
elsend(sendnum)	Number of elements sent to each of the sendnum processors

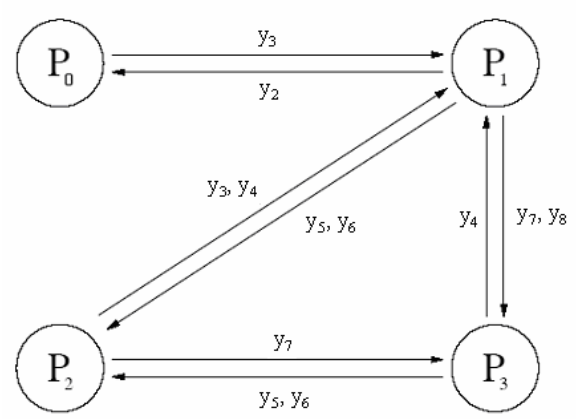

Figure 6. Communication scheme resulting from the data distribution of Figure 5

Since the matrix A is distributed column-wise among the processors, each processor transforms its local columns into the TJDS format.

$$value : \{a_{34} \quad a_{23} \quad a_{44} \quad a_{33} \quad a_{54} \quad a_{43} \quad a_{64} \quad a_{74} \quad a_{84}\}$$

$$row\_ind : \{3 \quad 2 \quad 4 \quad 3 \quad 5 \quad 4 \quad 6 \quad 7 \quad 8\}$$

$$tjd\_ptr : \{1 \quad 3 \quad 5 \quad 7 \quad 8 \quad 9 \quad 10\}$$

Figure 7. Local portion of A on processor P_1 in TJDS format

Vectors involved in MVM automatically carry the same permuted order as the matrix rows. During MVM, each matrix element has to find the matching vector element it is multiplied with by using the information of row indices. No permutation information is required during MVM as the vector is already permuted according to the matrix. The local and non-local elements of vector X are copied into a separate array recvec(:) which serves as a vector in MVM. It is apparent that a Transposed Jagged Diagonal can only be released for computation if the corresponding non-local elements have successfully been received. The parallel MVM implementation is based on MPI. The Transposed Jagged Diagonals are processed as a whole during MVM.

3. Experimental results and performance analysis

We have selected the sparse matrices from the Matrix-Market [10] collection to evaluate the SMVM for TJDS format.

3.1 Sequential performance

The MVM for TJDS is similar in the sense that they both have four load operations and one store operation to compute each partial result. But TJDS outperforms JDS because the permutation step needed in the JDS is not required for the TJDS [6]. Figure 8 gives the execution time for the matrix vector multiplication using JDS and TJDS formats. The two algorithms are executed sequentially on Intel Pentium-IV 3GHz processor with 512MB RAM.

Table 2. Selection of sparse matrices from matrix market

	Matrix	Dimension	N_{nze}
1	dw2048	2048 x 2048	10114
2	add32	4960 x 4960	23884
3	bcsstk23	3134 x 3134	24156
4	add20	2395 x 2395	17319
5	rw5151	5151 x 5151	20199
6	bcsstk15	3948 x 3948	60882
7	mhd3200a	3200 x 3200	68026

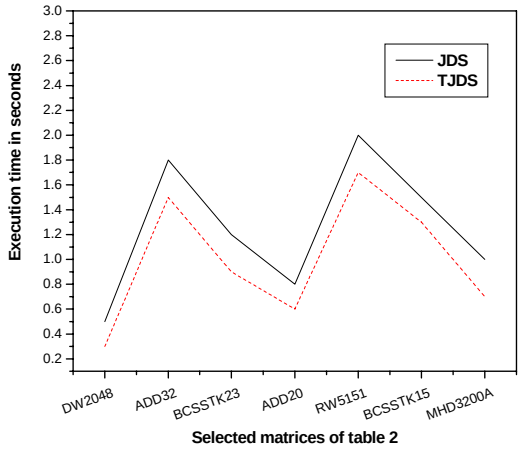

Figure 8. MVM execution time using JDS and TJDS formats

3.2 Parallel performance

In the parallel MVM the current processor exchanges data with its neighbors and then computes the product locally. After completing local computation the contribution from all processors is summed up by using MPI_ALLREDUCE. The matrices are partitioned among processors in column-wise block form. Timings are measured using MPI_Wtime(). We reported three different timings, total execution time, communication time and computation time. From these results we found that for small matrices, the communication exceeds the computation time and parallel algorithm performs worst than even sequential.

Total execution time is increased for systems ranging from 1000 to 10000 with the addition of a processor. This is due to the communication time because these systems are not very large and have less computation time as compared to communication time. For systems ranging from 50000 to 5000000 the total execution time is decreased with the addition of new processors in the cluster because these systems have more computation time than communication time.

A single processor system is unable to solve the linear system of order greater than one million (10^6) and 2-processor system could not solve a system of order greater than three million due to memory requirements. Figure 9 show the results of the TJDS SMVP on different processors with memory consideration.

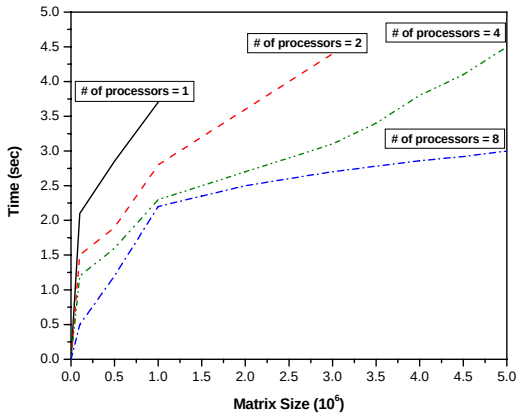

Figure 9. Performance of parallel TJDS matrix-vector product on different processors

We have tested our implementation on cluster of 8 PCs connected by a network of relatively low performance (Ethernet 100Mb/s). The results are shown in table 3.

Table 3. Elapsed time of computation on cluster of PCs

Matrix info				P	Elapsed time (sec)		
Matrix	n	cols	nnz	#	t_{comp}	t_{comm}	t_{exe}
cry10000	10000	5000	25000	2	1.103	0.031	2.096
		2500	12550	4	0.647	0.055	1.812
		1250	6325	8	0.239	0.069	1.054
bcsstk17	10974	5487	118014	2	3.714	0.050	4.914
		2744	64266	4	1.855	0.072	3.077
		1372	32916	8	1.009	0.081	2.24
bcsstk18	11948	5974	40899	2	1.877	0.055	3.082
		2987	21978	4	0.886	0.067	2.003
		1494	11514	8	0.499	0.082	1.731
bcsstk25	15439	7720	66964	2	2.967	0.06	4.567
		3860	34087	4	1.368	0.072	3.068
		1930	17091	8	1.004	0.091	1.88
memplus	17758	8879	71619	2	3.940	0.088	5.677
		4440	36131	4	1.858	0.034	3.348
		2220	22875	8	0.801	0.01	1.951
af23560	23560	11780	242128	2	7.501	0.141	8.742
		5890	121506	4	4.213	0.113	5.472
		2945	60781	8	2.001	0.090	3.191

Some parameters in the table are described below.

- The columns of *cols* and *nnz* are the maximum number of columns and the maximum number of nonzero entries, respectively, for a local matrix. In this work we partition the matrices for 2, 4 and 8 processors. For a balanced partitioning, both *cols* and *nnz* of all local matrices should be approximately the same and we double the number of processors, they should reduce in half. When these properties are found in the local matrices, we say that the local balance among the processors is good.
- The column of *P#* is the number of processors used in the calculation.
- The column of t_{comp} is the maximum total elapsed time for the slowest processor to compute the local matrix-vector product. Because the computing time for a local matrix-vector product is proportional to number of nonzeros, the slowest processor should be the one whose number of nonzero entries of the local matrix is maximum. The elapsed time for this processor represents the computing time. For matrices partitioned with a good load balance, the computing time for all processors should be approximately the same. The matrix af23560 has the maximum computation time of all selected matrices of table 3 as it has largest value of nonzeros.
- Before computing a local matrix-vector product, all processors exchange data with their neighbors. The column of t_{comm} is the maximum total elapsed time of the processor that does the most communication with its neighbors. The communication cost is another factor to be taken into account when partitioning a matrix for parallel computing. The communication time of the current processor is proportional to both the number of neighboring processors and the amount of exchanged data. With many neighbors, there aremany times to initialize latency of the network. The data is transferred through the network, so the larger the amount of data, the more time is required to transfer them.
- The column of t_{exe} is the maximum total elapsed time of all computations for the slowest processor.

An overview over the total runtime based on TJDS-MVM implementation is presented in figure 10 for fixed matrix size using different number of processors. The behavior can be attributed to different vector lengths of the inner loop of MVM, which is determined by the matrix dimension.

Since the problem size is fixed, the utilization of processors decreases with increasing number of processors due to the communication overhead. The parallel features of the implementation can be demonstrated in more detail by the parallel efficiency.

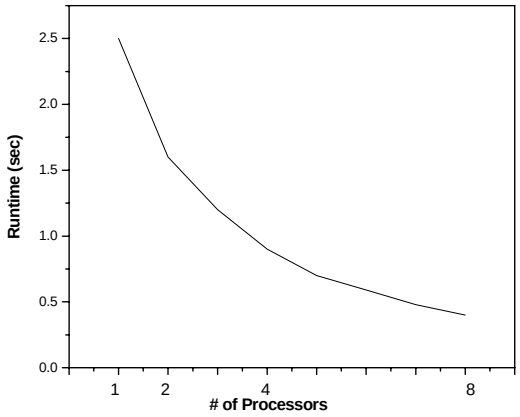

Figure 10. Runtimes of the TJDS-MVM on different number of processors

Table 4 shows the speed-up ratio for parallel matrix vector products using TJDS format. The parallelization speed-ups are nearly ideal in most cases.

Table 4. Speed-up ratios for parallel matrix-vector products

Matrix	P=1	P=2	P=4	P=8
cry10000	1.00	1.93	3.83	7.04
bcsstk17	1.00	1.99	3.97	7.84
bcsstk18	1.00	1.97	3.83	7.04
bcsstk25	1.00	1.90	3.99	7.97
memplus	1.00	2.00	4.03	10.88
af23560	1.00	1.93	3.63	7.00

Speedup ratio determines how much faster the parallel version runs than does the serial version. As this is expressed as a ratio of the serial runtime over the parallel runtime, if the parallel is faster then the ratio is greater than 1. A perfect speedup occurs when this ratio is exactly equal to the number of processors that are parallelized.

4. Conclusions

This paper presents some results of a simple but effective approach for parallelizing linear algebra operation such as TJDS based sparse matrix-vector multiplication. Also specific examples and experimental performance results are presented for the operation to be solved in parallel on Ethernet-based heterogeneous cluster, using the advantages of a transposed jagged diagonal storage format. The experimentation shows that the implementation obtains good results in parallel.

The number of computation for matrix-vector product is linear with respect to the number of nonzero elements. TJDS is suitable for parallel and distributed processing because the data partition scheme inherent to the data structure keeps the locality of reference on the non-zero values of the matrix and the elements of the x array.

We will also work toward high performance iterative linear solvers using these kernel routines and effective preconditioners for the solvers, with the goal of developing a sparse linear solver for sequential and distributive memory parallel architectures.

References

[1] A. T. Ogielski and W. Aiello, Sparse matrix computations on parallel processor arrays, *SIAM Journal on Scientific Computing*, 14 (1993), pp. 519-530.

[2] Arnold L. Rosenberg, Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better, *cluster, 3rd IEEE International Conference on Cluster Computing (CLUSTER'01)*, 2001, pp. 124.

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

[4] B. Hendrickson and T. G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing, *SIAM J. on Sci. Comp.*, 21(6), 2000, pp.2048–2072.

[5] 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.

[6] 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.

[7] Rukhsana Shahnaz, Anila Usman, Implementation and Evaluation of Sparse Matrix-Vector Product on Distributed Memory Parallel Computers, *Proc. Cluster2006, IEEE International Conference on Cluster Computing*, Barcelona, 2006.

[8] Fernando G. Tinetti, Walter J. Aroztegui, Antonio A. Quijano, "Sparse Equation Systems in Heterogeneous Clusters of Computers," *aina, 19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 2 (INA,, USW,, WAMIS,, and IPv6 papers)*, 2005, pp. 471-474.

[9] S. Riyavong, "Experiments on Parallel Matrix-Vector Product", *Working Note WN/PA/03/127, CERFACS*, Toulouse, France, 2003.

[10] Matrix Market. *http://math.nist.gov/MatrixMarket*.