Communication Schemes in Parallel Sparse Matrix-Vector Multiplication on PC Cluster

Rukhsana Shahnaz and Anila Usman,

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

Summary
The numerical core of iterative algorithms is a matrix-vector multiplication (MVM) involving the large sparse matrix. In this work some issues related to effective parallel implementation of sparse MVM on PC clusters are discussed. Attention is focused on the interprocessor communications and compression step involved in TJDS-MVM. We present three different communication approaches for parallel implementation and evaluate the contributions on PC cluster.

Keywords:
Interprocessor communication, Matrices distribution, matrix-vector product, sparse storage formats, Cluster computing.

1. Introduction

Repeated matrix-vector multiplications (SpMxV) $y = Ax$ that involve the same large, sparse, structurally symmetric or nonsymmetric square or rectangular matrix are kernel operations in various iterative solvers. Efficient parallelization of these solvers requires matrix A to be partitioned among the processors in such a way that communication overhead is kept low while maintaining computational load balance.

A network of PCs can be viewed as a distributed-memory environment. However, the interprocessor communication overhead plays a much active role in the parallel implementation. The algorithm adopted in interprocessor communications determines the complexity of programming and the overall parallel performance, and therefore, the algorithm for interprocessor communication needs to be given more consideration when dealing with the parallel programming.

This paper presents three different schemes for the communication and compression of matrix in Transposed Jagged diagonal storage (TJDS) format for parallel MVM on heterogeneous cluster [1, 2]. The three strategies are global compression, local compression and modified local compression. There are two main idea behind these schemes: one is whether the compression step involved TJDS format is performed before or after matrix distribution and other is overlapping of communication and computation steps. 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.

Manuscript received May 5, 2025
Manuscript revised May 20, 2025
https://doi.org/10.22937/IJCSNS.2025.25.5.4

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) [2, 3, 4] were implemented and compared [5, 6]. 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}.

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

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
```

3. Parallel implementation

To keep memory requirements as low as possible on multiprocessor systems with distributed memory, the matrix data is spread over the processors involved in computation. Matrix elements requiring access to vector data stored on remote processors (non-local vector elements) cause communication. Since the matrix does not change during 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 MVM step. Three different techniques for this purpose are presented here.

3.1 The Global compression scheme

In the global compression scheme the matrix compression is performed before the matrix distribution. Fig. 1 shows the matrix A and vector X. All nonzero matrix elements are filled. Matrix elements causing communication are marked black and local matrix elements are colored red. Fig. 2 shows the matrix A with all the nonzero elements shifted upward. Fig. 3 shows Transposed Jagged Diagonal Storage obtained by reordering the columns of upward shifted matrix 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. Fig. 4 illustrates the communication of vector elements in MVM.

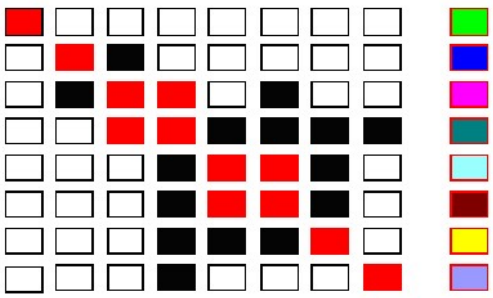

Fig. 1 The original matrix A and vector X.

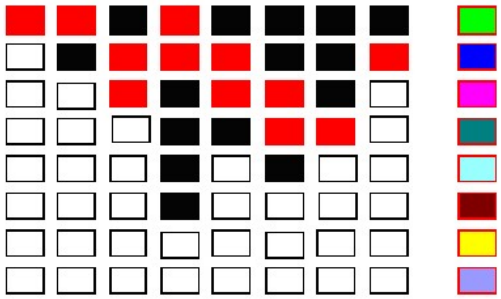

Fig. 2 The upward compressed matrix.

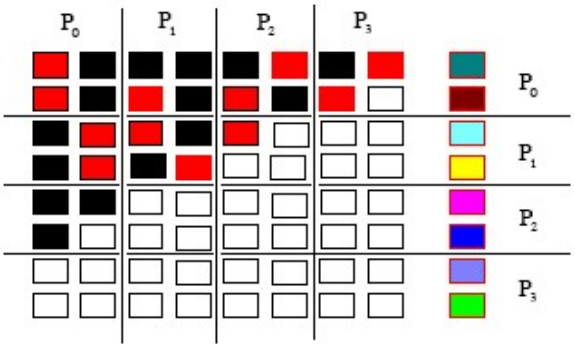

Fig. 3 The reordering of compressed matrix column and vector X. Also the distribution of TJDS-matrix and vector X on 4 processors ($P_0 - P_3$).

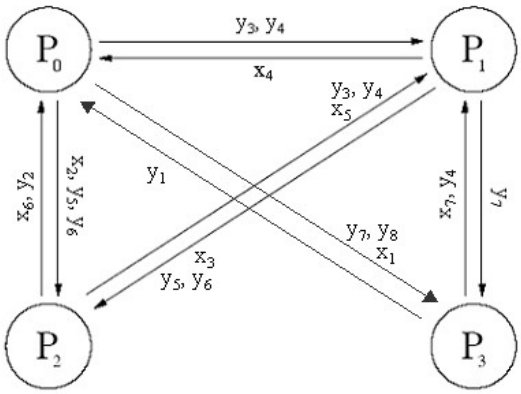

Fig. 4 The communication scheme resulting from the data distribution of fig. 3

3.2 The Local compression scheme

The global compression involve a lot of communication volume, as shown in fig. 4, leading to perform the compression step by each processor. In the local compression scheme the matrix compression is performed locally after the matrix distribution [6]. Each processor transforms its local columns into the TJDS format as illustrated in fig. 6. The Jagged Diagonals can only be released for computation if the corresponding non-local vector elements have successfully been received. Fig. 7 shows the communication required during one MVM.

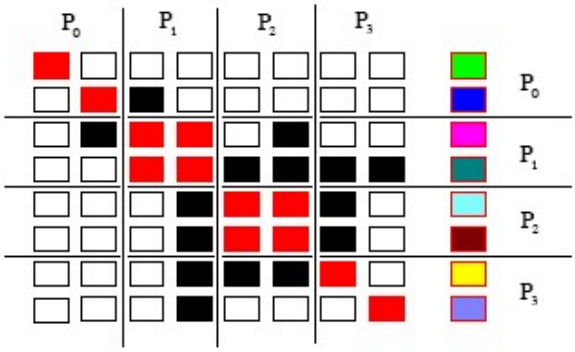

Fig. 5 Distribution of a matrix A and vector X on 4 processors ($P_0 - P_3$).

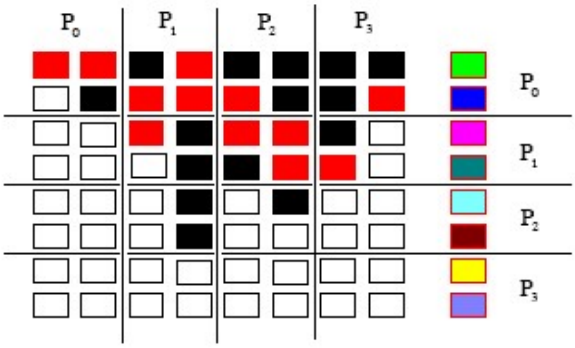

Fig. 6 Transformation of local columns of matrix in TJDS format.

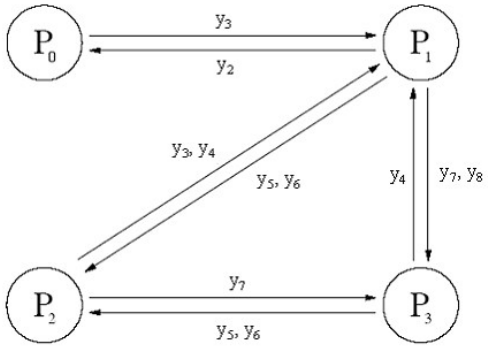

Fig. 7 Communication scheme resulting from local compression

3.3 Modified local compression with rearrangement of local and non-local sub-blocks

In principle the implementation of local compression scheme allows overlapping of communication and computation. At this point the parallel TJDS scheme as described above has one serious drawback: The processor-local compression step might cause a strong mixture of local and non-local matrix elements when shifting the nonzero elements to the left. For instance, process P_2 and P_3 in fig. 6 requires at least one receive-operation to be completed before starting any computation when using the TJDS transformation in a straightforward way. For that reason we have slightly modified the parallel compression step as follows:

a. Shift local matrix elements to upward only within the local sub-block on each processor.
b. Fill each local sub-block with the non-local matrix elements from downward.

If the diagonal of the matrix is full this algorithm generates at least one local *Transposed Jagged Diagonal* as can be seen in fig. 8 below.

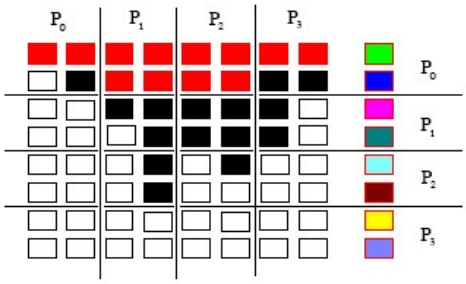

Fig. 8 Distribution of a matrix A and vector X as in figure 5 with modified TJDS compression step.

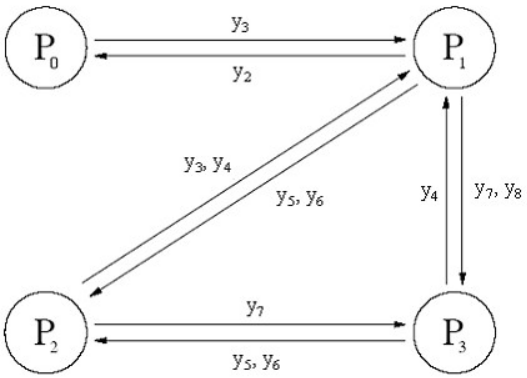

Fig. 9 Communication scheme resulting from modified local compression

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. 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.

4. Experimental results and performance analysis

We have selected the sparse matrices from the Matrix-Market [7] collection to evaluate the TJDS-MVM for different communication and compression schemes.

Table 1. Selection of sparse matrices from matrix market

	Matrix	Dimension	N_{nze}	Max. N_{nzec}	Max. N_{nzer}
1	cry10000	10000x10000	49699	6	5
2	bcsstk17	10974x10974	219812	150	150
3	bcsstk18	11948x11948	80519	49	49
4	bcsstk25	15439x15439	133840	59	59
5	memplus	17758x17758	126150	353	353
6	af23560	23560x23560	484256	21	21

For parallel computers with distributed memory, we use a column-wise distribution of matrix and row-wise distribution of vector elements among the processors. 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.

4.1 Communication performance

We have tested our implementation on cluster of 8 PCs with 3.0 GHz processors running LINUX. Each processor is equipped with 1024 MB local memory. One of the machines is the master node and the others are slave nodes. The slave nodes have a cut down version of LINUX, which contains the bare minimum for the machine to operate in the cluster. The slaves and the master node are connected via a network (Ethernet 100 Mbps).
Three different way are presented here for the communication and compression step for TJDS format. The main aim is to find the way by which we can exchange data efficiently in anticipation of the overlapping of communication and computation in the MVM step. The timing results are shown in table 2.

Table 2. Elapsed time of computation on cluster of PCs

Matrix info				P	Elapsed time (sec) in		
Matrix	n	cols	nnz	#	Sch1	Sch2	Sch3
cry10000	10000	5000	25000	2	2.158	2.096	2.065
		2500	12550	4	1.922	1.812	1.757
		1250	6325	8	1.192	1.054	0.985
bcsstk17	10974	5487	118014	2	5.014	4.914	4.864
		2744	64266	4	3.221	3.077	3.005
		1372	32916	8	2.402	2.24	2.159
bcsstk18	11948	5974	40899	2	3.192	3.082	3.027
		2987	21978	4	2.137	2.003	1.936
		1494	11514	8	1.895	1.731	1.649
bcsstk25	15439	7720	66964	2	4.687	4.567	4.507
		3860	34087	4	3.212	3.068	2.996
		1930	17091	8	2.062	1.88	1.789
memplus	17758	8879	71619	2	5.853	5.677	5.589
		4440	36131	4	3.416	3.348	3.314
		2220	22875	8	1.971	1.951	1.941
af23560	23560	11780	242128	2	9.024	8.742	8.601
		5890	121506	4	5.698	5.472	5.359
		2945	60781	8	3.371	3.191	3.101

The total runtimes for different interprocessor communication schemes with various numbers of processors and fixed problem size (for 1000 iterations) are presented in Fig. 10.

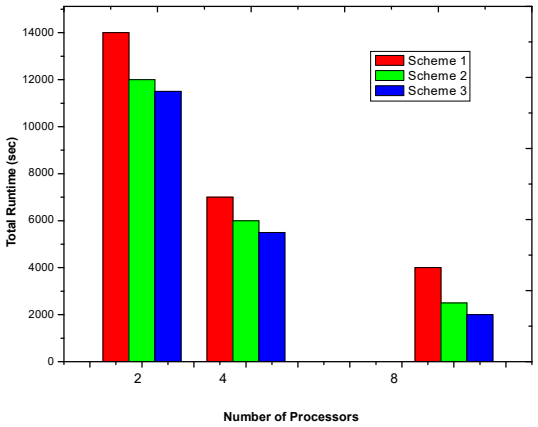

Fig. 10 Total runtime with different interprocessor communication schemes

4. Conclusions

This paper presents some issues related to interprocessor communication in parallelizing TJDS based sparse matrix-vector multiplication on a PC cluster. Three communication and compression schemes that have different programming complexities and different performances are presented. In the global compression scheme the matrix compression is performed before the matrix distribution. The global compression involves a lot of communication volume, leading to perform the compression step by each processor. In the local compression scheme the matrix compression is performed locally after the matrix distribution. But in this case the processor-local compression step might cause a strong mixture of local and non-local matrix elements when shifting the nonzero elements to the left. For this purpose the local compression scheme is slightly modified. So that the computation starts on every processor before the completion of first receive operation allowing maximum overlapping of communication and computation. The experimentation shows that the implementation of this modified approach obtains good results.

References

[1] 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.
[2] R. Barrett et al., *Templates for the solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.
[3] 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.

[4] 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.

[5] 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.

[6] Rukhsana Shahnaz, Anila Usman, An efficient sparse matrix-vector multiplication on distributed memory parallel computers, *IJCSNS, International Journal of Computer Science and Network Security*, Vol.7, No.1, January 2007, pp. 77-82.

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