A Library for Parallel Sparse Matrix Vector Multiplies*

Bora Uçar and Cevdet Aykanat
Department of Computer Engineering,
Bilkent University, 06800, Ankara, Turkey
{ubora,aykanat}@cs.bilkent.edu.tr

Abstract

We provide parallel matrix-vector multiply routines for 1D and 2D partitioned sparse square and rectangular matrices. We clearly give pseudocodes that perform necessary initializations for parallel execution. We show how to maximize overlapping between communication and computation through the proper usage of compressed sparse row and compressed sparse column formats of the sparse matrices. We give pseudocodes for multiplication routines which benefit from such overlaps.

Technical Report:	No: BU-CE-0506 Department of Computer Engineering, Bilkent University, 06800 Ankara, Turkey. Available at http://www.cs.bilkent.edu.tr/publications/ Date: August 31, 2005.

*This work is partially supported by the Scientific and Technical Research Council of Turkey under grant 103E028

1 Introduction

Parallel sparse matrix vector multiplies (SpMxV) of the form $y = Ax$ reside in the kernel of many scientific computations. One-dimensional (1D) [5, 13, 14, 16, 21] and two-dimensional (2D) [6, 7, 22] partitioning methods are proposed to balance the computational loads of the processors while minimizing the communication overhead. In this paper, we describe software that perform parallel SpMxV operations under 1D and 2D partitionings. Our aim is to ease the development of iterative methods. We give coding of the BiCGSTAB method as an example.

As noted in [20], software packages that implement only the parallel SpMxV operations are not common for several reasons. First, matrix-vector multiply is a simple operation; developers write their own routine. Second, there are different sparse matrix storage formats that fit different applications; it is difficult to design softwares that apply to all areas. Recently published sparse BLAS standard [10] even does not specify a data structure to store sparse matrices. It rather allows complete freedom for sparse BLAS library developers to optimize their own libraries [11]. However, there are numerous software packages (see the sparse iterative solvers having parallel mode in Dongarra's survey [9]) that include utilities for performing distributed SpMxV operations; see for example PETSc [1], Aztec [15, 20], and PSPARSLIB [18].

Common features of existing software utilities for SpMxV operations are as follows. Most of the packages target 1D partitioned matrices, where y and x vectors have the same processor assignment as that of the rows or the columns of the matrix. This symmetric partitioning on the input and output vectors restricts the packages to square matrices. Some packages enable the user of the library to plug the necessary communication subroutines which are called between the partial executions of the SpMxV routines in a reverse communication [8] loop.

The characteristics of our software are as follows. Its SpMxV routines apply to 1D and 2D partitioned matrices of any shape. It can handle symmetric and unsymmetric partitionings on the input and output vectors. Our software uses point-to-point communication operations internally to exploit sparsity during communications, i.e., there does not exist any redundancy in the communication. To our knowledge, there does not exist any package that uses point-to-point communication when the matrices have 2D partitions. Also, we are not aware of any SpMxV libraries targeting rectangular matrices. Our package include entry-level matrix construction process as prescribed in Sparse BLAS standard [10]. There are software utilities to set-up communication data structures using the partitioning indicators. The software exploits compressed sparse row (CSR) and compressed sparse column(CSC) storage formats to achieve maximum communication and computation overlap.

The organization of this report is as follows. In §2 and §3, we discuss the SpMxV operations under 1D and 2D partitioning of the sparse matrices, respectively. In §4, we discuss necessary steps to realize efficient implementation of the SpMxV routines. We clearly give pseudocodes that set up communication

and list issues that should be resolved to design parallel SpMxV routines along with our decisions. In §5, we give the interface of the library and its usability through actual implementations. We conclude the report with future work.

2 Parallel algorithms based on 1D partitioning

Suppose that the rows and columns of an $m \times n$ matrix A are permuted into a $K \times K$ block structure

$$A_{BL} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1K} \\ A_{21} & A_{22} & \cdots & A_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ A_{K1} & A_{K2} & \cdots & A_{KK} \end{bmatrix} \tag{1}$$

for *rowwise* or *columnwise* partitioning, where K is the number of processors. Block $A_{k\ell}$ is of size $m_k \times n_\ell$, where $\sum_k m_k = m$ and $\sum_\ell n_\ell = n$. In rowwise partitioning, each processor P_k holds the kth row stripe $[A_{k1} \cdots A_{kK}]$ of size $m_k \times n$. In columnwise partitioning, P_k holds the kth column stripe $[A_{1k}^T \cdots A_{Kk}^T]^T$ of size $m \times n_k$. In rowwise partitioning, the row stripes should have nearly equal number of nonzeros for having the computational load balance among processors. The same requirement exists for the column stripes in columnwise partitioning.

2.1 Row-parallel algorithm

Consider matrix-vector multiply of the form $y \leftarrow Ax$, where y and x are column vectors of size m and n, respectively, and the matrix is partitioned rowwise. A rowwise partition of matrix A defines a partition on the output vector y. The input vector x is assumed to be partitioned conformably with the column permutation of matrix A. In particular, y and x vectors are partitioned as $y = [y_1^T \cdots y_K^T]^T$ and $x = [x_1^T \cdots x_K^T]^T$, where y_k and x_k are column vectors of size m_k and n_k, respectively. That is, processor P_k holds x_k and is responsible for computing y_k.

In [13, 18, 20, 21], authors discuss the implementation of parallel SpMxV operations where the matrix is partitioned rowwise. The common algorithm executes the following steps at each processor P_k:

1. For each nonzero off-diagonal block $A_{\ell k}$, send sparse vector $\hat{x}_k^\ell$ to processor P_ℓ, where $\hat{x}_k^\ell$ contains only those entries of x_k corresponding to the nonzero columns in $A_{\ell k}$.
2. Compute the diagonal block product $y_k^k = A_{kk} \times x_k$, and set $y_k = y_k^k$.
3. For each nonzero off-diagonal block $A_{k\ell}$, receive $\hat{x}_\ell^k$ from processor P_ℓ, then compute $y_k^\ell = A_{k\ell} \times \hat{x}_\ell^k$, and update $y_k = y_k + y_k^\ell$.

Since the matrix is distributed rowwise, we call the above algorithm *row-parallel*. In Step 1, P_k might be sending the same x_k-vector entry to different processors according to the sparsity pattern of the respective column of A. This multicast-like operation is referred to here as *expand* operation.

2.2 Column-parallel algorithm

Consider matrix-vector multiply of the form $y \leftarrow Ax$, where y and x are column vectors of size m and n, respectively, and the matrix A is partitioned columnwise. The columnwise partition of matrix A defines a partition on the input vector x. The output vector y is assumed to be partitioned conformably with the row permutation of matrix A. In particular, y and x vectors are partitioned as $y = [y_1^T \cdots y_K^T]^T$ and $x = [x_1^T \cdots x_K^T]^T$, where y_k and x_k are column vectors of size m_k and n_k, respectively. That is, processor P_k holds x_k and is responsible for computing y_k. Since the matrix is distributed columnwise, we derive a *column-parallel algorithm* for this case. The column-parallel algorithm executes the following steps at processor P_k:

1. For each nonzero off-diagonal block $A_{\ell k}$, form sparse vector $\hat{y}_\ell^k$ which contains only those results of $y_\ell^k = A_{\ell k} \times x_k$ corresponding to the nonzero rows in $A_{\ell k}$. Send $\hat{y}_\ell^k$ to processor P_ℓ.
2. Compute the diagonal block product $y_k^k = A_{kk} \times x_k$, and set $y_k = y_k^k$.
3. For each nonzero off-diagonal block $A_{k\ell}$ receive partial-result vector $\hat{y}_k^\ell$ from processor P_ℓ, and update $y_k = y_k + \hat{y}_k^\ell$.

The multinode accumulation on the w_k-vector entries is referred to here as *fold* operation.

3 Parallel SpMxV based on 2D partitioning

Consider the matrix-vector multiply of the form $y \leftarrow Ax$, where y and x are column vectors of size m and n, respectively, and the matrix is partitioned in two dimensions among K processors. The vectors y and x are partitioned as $y = [y_1^T \cdots y_K^T]^T$ and $x = [x_1^T \cdots x_K^T]^T$, where y_k and x_k are column vectors of size m_k and n_k, respectively. As before we have $\sum_k m_k = m$ and $\sum_\ell n_\ell = n$. Processor P_k holds x_k and is responsible for computing y_k. Nonzeros of a processor P_k can be visualized as a sparse matrix A^k

$$A^k = \begin{bmatrix} A_{11}^k & \cdots & A_{1k}^k & \cdots & A_{1K}^k \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ A_{k1}^k & \cdots & A_{kk}^k & \cdots & A_{kK}^k \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ A_{K1}^k & \cdots & A_{Kk}^k & \cdots & A_{KK}^k \end{bmatrix} \tag{2}$$

of size $m \times n$, where $A = \sum A^k$. Here, the blocks in row-block stripe $A^k_{k*} = \{A^k_{k1}, \cdots, A^k_{kk}, \cdots, A^k_{kK}\}$ have row dimension of size m_k. Similarly, the blocks in column-block stripe $A^k_{*k} = \{A^k_{1k}, \cdots, A^k_{kk}, \cdots, A^k_{Kk}\}$ have column dimension of size n_k. The x-vector entries that are to be used by processor P_k are represented as $x^k = [x^k_1, \cdots, x^k_k, \cdots, x^k_K]$, where x^k_k corresponds to x_k and other x^k_ℓ are belonging to some other processor P_ℓ. The y-vector entries that processor P_k computes partial results for are represented as $y^k = [y^k_1, \cdots, y^k_k, \cdots, y^k_K]$, where y^k_k corresponds to y_k and other y^k_ℓ are to be sent to some other processor P_ℓ. Since the parallelism is achieved on nonzero basis rather than complete rows or columns, we derive a *row-column-parallel* SpMxV algorithm. This algorithm executes the following steps at each processor P_k:

1. For each $\ell \neq k$ having nonzero column-block stripe A^ℓ_{*k}, send sparse vector $\hat{x}^\ell_k$ to processor P_ℓ, where $\hat{x}^\ell_k$ contains only those entries of x_k corresponding to the nonzero columns in A^ℓ_{*k}.
2. Compute the column-block stripe product $y^k = A^k_{*k} \times x^k_k$.
3. For each nonzero column-block stripe $A^k_{*\ell}$, receive $\hat{x}^k_\ell$ from processor P_ℓ, then compute $y^k = y^k + A^k_{*\ell} \times \hat{x}^k_\ell$, and set $y_k = y^k_k$.
4. For each nonzero row-block stripe $A^k_{\ell*}$, form sparse partial-result vector $\hat{y}^k_\ell$ which contains only those results of $y^k_\ell = A^k_{\ell*} \times x^k$ corresponding to the nonzero rows in $A^k_{\ell*}$. Send $\hat{y}^k_\ell$ to processor P_ℓ.
5. For each $\ell \neq k$ having nonzero row-block stripe A^ℓ_{k*} receive partial-result vector $\hat{y}^\ell_k$ from processor P_ℓ, and update $y_k = y_k + \hat{y}^\ell_k$.

4 Implementation details

In order to implement the above algorithms, one has to follow some initialization steps:

1. Provide partitioning indicators on x and y vectors. In our implementation a central processor reads these partitioning indicators from different files and broadcast them to the other processors. We chose to provide each processor with partitioning indicators as a whole, i.e., each processor gets two arrays of size M and N one for output vector and one for input vector, respectively. Note that, processors usually need only a small portion of these partition arrays. The rationale behind our choice is to make the library handle arbitrary partitionings. That is, a processor can hold an x vector element and thus expand it even if it has not got a single nonzero in the corresponding column of A. Similarly, a processor can be set to responsible for folding on an element of y vector even if it does not generate partial result for that element. We refer reader to our previous work [21] to get taste of such unusual partitionings. Note that these indicators are usually available; however it is possible to efficiently construct them as discussed by Pınar [17] and Tunimaro et.al. [20]. If the matrix

```
SetupComm2D(A, xpartvec, ypartvec)
begin
(1)   for each nonzero a_ij in A do #i and j are global indices
(2)       if i is not marked then
(3)           mark i
(4)           increase ySendCount to processor p=ypartvec[i]
(5)           put p into ySendList
(6)       if j is not marked then
(7)           mark j
(8)           increase xRecvCount from processor p=xpartvec[j]
(9)           put p into xRecvList
(10)  end for
(11)  for i=1..M do
(12)      if i is not marked and myId=ypartvec[i] then
(13)          mark i; increase ySendCount[myId]
(14)  end for
(15)  for j=1..N do
(16)      if j is not marked and myId=xpartvec[j] then
(17)          mark j; increase xRecvCount[myId]
(18)  end for
(19)  AlltoAll communication
          #send xRecvCounts, receive into xSendCounts;
          #send ySend Counts, receive into yRecvCounts
(20)  #allocate space for indices to be sent and to be received
(21)  for each column j do
(22)      if j is marked then
(23)          put j into xIndexRecv list for processor p=xpartvec[j]
(24)  end for
(25)  for each row i do
(26)      if i is marked then
(27)          put i into yIndexSend list for processor p=ypartvec[i]
(28)  end for
(29)  for each processor in xRecvList do
(30)      send xIndexRecv list to processor p
(31)  end for
(32)  for each processor in xSend list do
(33)      receive into xIndexSend list for processor p
(34)  end for
(35)  for each processor in ySendList list do
(36)      send yIndexSend list to processor p
(37)  end for
(38)  for each processor in yRecvList list do
(39)      receive into yIndexRecv list for processor p
(40)  end for
end
```

Figure 1: Setting up communication for 2D partition.

partitioning is 1D then one of the partitioning indicators is used to partition the matrix as well.

2. Provide matrix nonzeros and x vector components to the processor. A central processor reads the matrix components and distribute them according to partition on the matrix. If the matrix partitioning is 2D then the central processor reads partitioning indicator on the nonzeros of A from a file. Here, the central processor sends all the matrix components of a processor in a single message.

3. Determine the communication pattern. This is a complicated task that takes more time than SpMxV takes. We show the pseudo-code, which is executed by each processor, for setting-up communication pattern for 2D case in Fig. 1. By removing lines pertaining to y vector one can obtain communication set-up procedure for 1D rowwise-partitioned matrices. Similarly, one can obtain the same procedure for 1D columnwise partitioned matrices by removing the lines pertaining to x vector. As seen in the figure, a certain processor sweeps (lines 1–10) its nonzeros to mark global indices of x vector entries that it needs and global indices of y vector entries on which it generates partial results. Note that after that sweeping processors know which x vector entries to be received from which processor and which y vector entries to be sent to which processor. Here, a processor increments the counters corresponding to its rank to compute its local matrix's row and column dimensions. Another sweep over row indices (lines 11–14) and column indices (lines 15–18) is necessary to handle arbitrary input and output vector partitionings. After the all-to-all communication (line 19), processors know the number of x vector entries to be sent and number of y vector partial results to be received on per processor basis. After allocating necessary space each processor become ready to exchange the indices of the vector components to be communicated later in SpMxV routines. In lines 21–28, processors build the lists that hold global indices of vector components. In the remaining of the method, processors exchange those global index lists. After executing the depicted steps each processor obtain the information on the vector components' indices to be sent and to be received. Besides, each processor obtain the row and column dimensions for the sparse matrix in its memory.

4. Determine local indices. For row-parallel algorithm, it is customary to renumber the x-vector entries that are accessed by processors in such a way that entries those belong to the same processor have contiguous indices; see [18, 20]. Analogously, for column-parallel algorithm, the y-vector entries that are to be sent to the same processor are renumbered contiguously. Combining these, it is preferable to renumber the x vector entries to be received from the same processor contiguously and y vector entries to be sent to the same processor contiguously in 2D case. In previous works [18, 20], developers renumbered the local vector components starting from 0, and then continue on the *external* vector components. We choose to renumber the vector components according to the rank of the processors responsible on the components. For example, processor P_k gives label to the external vector components belonging to some other processor P_j where $j < k$, then gives labels to the local vector components and then continue with labeling the external vector components belonging to

some other processor P_ℓ where $k < \ell$. Note that processor P_k can give labels to external vector components belonging to a processor P_ℓ in any order; $x[i]$ can get label that is less than the label of $x[j]$ even processor P_ℓ labels $x[j]$ before $x[i]$. Since processors communicate global indices in Fig. 1 this does not cause any problem.

5. Set local indices for vector components to be sent and to be received and also for matrix components. This is a straightforward task that is done locally by each processors. Each processor sweeps the local data structures holding the global indices of local matrix, `xSendList`, `xRecvList`, `ySendList`, and `yRecvList`.

6. Assemble the local sparse matrix. The local matrix is assembled using the labels determined in Step 4. In [18, 20], developers store the local matrix in CSR format for rowwise-partitioned matrices. Continuing their labeling procedure, this results in splitting the matrix into two, that is, *Aloc* and *Acpl*. Here *Aloc* contains nonzeros a_{ij} where $x[j]$ belongs to the associated processor, and *Acpl* contains nonzeros a_{ie} where $x[e]$ belongs to some other processor. Remember that the mentioned works address symmetric partitioning on x and y vectors, hence *Aloc* is a square matrix. In [18] developers mention that the matrices *Aloc* and *Acpl* can be stored in any format.

In our implementation we explicitly split a processor's matrix into two sparse matrices *Aloc* and *Acpl* for row-parallel algorithm. Here *Aloc* contains all nonzeros a_{ij} where $x[j]$ is local to the processor even if $y[i]$ belongs to some other processor and *Acpl* contains all nonzeros a_{ie} where $x[e]$ belongs to some other processor. We store *Aloc* and *Acpl* in CSC format. Our aim is to maximize communication and computation overlap without incurring any extra operation. In [18] developers perform the first two steps of the row-parallel algorithm above by overlapping communication in the first step with the computation in the second one. After receiving all external x vector components they continue with multiplication using *Acpl* as the third step. With our approach we again obtain the same overlap in the first two steps and also overlap communication and computation in the third step as well, i.e., we implement the third step of row-parallel algorithm as written in §2. When a processor receives a message in the third step containing some external x components, it can continue multiplying before waiting all external x components to arrive through exploiting the CSC format. Note that, using CSC format instead of CSR here is essential. In this format, we have explicit and immediate access to the row indices that has nonzeros in a given column. Hence, given an $x[j]$ one can update those $y[i]$'s where there is a nonzero a_{ij} sequentially without any search. Similarly for column-parallel algorithm we store *Aloc* and *Acpl* in CSR format to maximize the communication computation overlap. In using CSR format here, our gain is the overlap between the messages a processor receives and the associated gathering of partial sums in step 3 of the column-parallel algorithm. In row-column parallel algorithm we benefit both of the overlaps by using the same constructs in row and column-parallel algorithms.

5 Examples using the library

In Fig. 2, we give the listing of the interface to the library and a call to external BiCGSTAB solver we have developed using the SpMxV routines of the library. We used LAM implementation [3] of message passing interface (MPI). In Fig. 2, `buMatrix` data structure is used the store the sparse matrices, either in CSR or CSR format. The data structure also has fields to hold the row, column, and nonzero counts and to distinguish the storage formats. Each local matrix will be of this type (`loc and cpl` in the figure). `parMatrix` is used to store distributed matrices for SpMxV operation. It has `loc` and `cpl` fields to store *Aloc* and *Acpl* as discussed in Section 4. The `parMatrix` structure also has fields to describe and implement communications. The communication handle `in` is used in communications regarding the input vectors of the SpMxV operation. The handle `out` is used in communications regarding the output vectors of SpMxV operations. We carry those communication handles along with matrices, however, they are used with vectors that appear in a SpMxV operation with the associated matrix. The field `scheme` designates the partitioning scheme on the matrix, which is used to decide on the SpMxV subroutine to call.

In Fig. 2, `initParLib` initializes the library. Note that this call creates a communication world under the current communication world (in the figure, the parent communication world is MPI's default `MPI_COMM_WORLD`). We hide the world that library's communication exist from the user, however, there are necessary subroutines which returns library's communication world handle. Such a distinct communication world is necessary in order to distinguish messages that are performed inside and outside the library (see Chapter 5 in [19]) to avoid message conflicts. The routine `readMatrixCoordinates` fills coordinate format storage area through communication. In `setup2D`, the initialization steps discussed in Section 4 are executed. It also assembles the matrices *Aloc* and *Acpl* from the coordinate format. The vector x is created with the size of the local x-vector entries which is mostly available explicitly without any computation. We choose to decouple the size of the local vectors from the local matrices' dimensions to free the user from parallel programming details. Similarly, vector `b` generated at the end of `mxv` routine holds only the components of b that are folded in this processor. Finally, we delete the library's communication world by a call `quitParLib`. After this call, any attempts to call library's facilities will fail with a proper message.

We have developed BiCGSTAB [2, 23] to test the usability of the developed SpMxV library and give the code in Fig. 3. Once we have designed the SpMxV routine with proper interface, development of iterative methods becomes an easy task. One has to deal with vector operations only. We have provided a few linear vector operations such as `dotv`, `normv`, and `v_plus_cw` as well. These operations perform dot product of two vectors, compute the norm of a vector, and compute "scalar c w plus v" as in SAXPY of BLAS1 with different resulting vector. With these routines, the code looks like its pseudocode listing.

```
int main(int argc, char *argv[])
{
 int myId, numProcs, i, myXsize;
 int *rowIndices, *colIndices;  double *val;
 buMatrix *mtrx, *loc, *cpl;
 int partScheme;
 parMatrix *A;
 comm *in, comm *out;
 buVector *x, *b;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
 MPI_Comm_rank(MPI_COMM_WORLD, &myId);

 mtrx  = (buMatrix*) malloc(sizeof(buMatrix));
 loc = (buMatrix *) malloc(sizeof(buMatrix));
 cpl = (buMatrix *) malloc(sizeof(buMatrix));
 in = allocComm();
 out = allocComm();

 initParLib(MPI_COMM_WORLD);
 partScheme = PART_2D;
 readMatrixCoordinates(&rowIndices,&colIndices,
      &val, &(mtrx->nnz), &(mtrx->gm),
      &(mtrx->gn), &(mtrx->outPart),
      &(mtrx->inPart),argv[1], MPI_COMM_WORLD);
 setup2D(rowIndices,colIndices, val, mtrx->nnz,
      mtrx, loc, cpl, in, out, MPI_COMM_WORLD);
 A = (parMatrix *) malloc(sizeof(parMatrix));
 A->loc=loc; A->cpl=cpl; A->in=in; A->out=out;
 A->scheme = partScheme;
 /*fill a local vector x;later enlarge       *
  *into a temporary vector in mxv routine to *
  *hold external variables as well.          */
 myXsize=A->loc->n-A->in->recv->all[numProcs]
 x = allocVector(myXsize);
 for(i  = 0 ; i < x->sz; i++)
     x->val[i] = 1;
 b = (buVector *)malloc(sizeof(buVector));
 b->sz = 0;
/*will be adjusted in mxv routine to         *
 * include only the elements that I own      */
 mxv(A, x, b, MPI_COMM_WORLD);
/*compute b = A.1  */
 for( i  = 0 ; i < x->sz; i++) /*reset x=0   */
     x->val[i] = 0.0;
 bicgstab(A,x,b,200,1.0e-12,MPI_COMM_WORLD);

 freeMatrix(mtrx); freeMatrix(loc);
 freeMatrix(cpl);
 free(A); freeVector(x); freeVector(b);
 freeComm(out);  freeComm(in);
 quitParLib(MPI_COMM_WORLD);
 MPI_Finalize();
 return 0;
}
```


Figure 2: A simple C program that uses library with 2D partitioning.

```
void
bicgstab(parMatrix *A,buVector *x,buVector *b,
   int maxIter,double tol,MPI_Comm parentComm)
{
 buVector *rhat, *r, *p, *v, *w, *z;
 double c,old_rho,rho,alpha;
 double old_omega,omega,beta;
 double res, res0;
 int k, myId, inSz, outSz;
 c= old_rho=alpha=old_omega= 1.0;
 z = (buVector *)malloc(sizeof(buVector));
 z->sz = 0;
 /*initial residual will be computed*/
 mxv(A, x, z, parentComm);
 v_plus_cw(b, z, c, r);
 inSz = x->sz;  outSz = z->sz;
 p = allocVector(inSz); v=allocVector(inSz);
r = allocVector(outSz);
 rhat = allocVector(outSz);
 w = allocVector(inSz + outSz);
 vcopy_vv(r, rhat);
 res0 = normv(b); k = 0;
 do /*main BiCGSTAB loop*/
     {
     k ++;
     rho = dotv(rhat, r);
     beta =(rho/old_rho) * (alpha/old_omega);
/* compute new p as                        */
/* p = r - beta * z # z=p -old_omega * v*/
     v_plus_cw(p, v, -old_omega, z);
     v_plus_cw(r, z, beta, p);
/*compute new v, r, and alpha*/
     mxv(A, p, v, parentComm);
     alpha = rho/dotv(rhat, v);
     v_plus_cw(r, v, -alpha, r);
     if(normv(r)/res0 < tol)
       { v_plus_cw(x, p, alpha, x); break;}
/*compute new omega*/
     mxv(A, r, z, parentCOMM);
     omega = dotv_div_dotv(z, r, z, z);
/*compute new x and new r*/
     v_plus_cw(x, p, alpha, w);
     v_plus_cw(w, r, omega, x);
     v_plus_cw(r, z, -omega, r);
     res = normv(r);
     old_rho = rho;
     old_omega = omega;
     if(myId == 0)
       printf("\titeration=");
       printf("%9d residual %e\n",k,res/res0);
     } while((res/res0>tol) && (k<maxIter));
 freeVector(p);freeVector(v);freeVector(w);
 freeVector(z);freeVector(r);freeVector(rhat);
 return;
}
```


Figure 3: A simple C program that uses library to develop BiCGSTAB.

Table 1: Properties of test matrices.

Matrix	M	N	NNZ
memplus	17758	17758	126150
bcsstk25	15439	15439	252241
onetone2	36057	36057	254595
pig-very	174193	105882	463303
lhr34	35152	35152	799064

6 Experiments

We have conducted experiments on a few sparse matrices. Properties of these matrices are listed in Table 1. In the table, M, N, and NNZ denote number of rows, number of columns, and number of nonzeros, respectively, of the matrices obtained from University of Florida Sparse Matrix Collection[1], Matrix Market[2], and [12].

The matrices are partitioned among 24 processors using PaToH software [4] to obtain rowwise, columnwise, fine-grain on nonzero basis, and checkerboard partitionings [5, 6, 7] to test our 1D and 2D parallel algorithms. We report the timings in Table 2 in milliseconds. Timings are obtained using `MPI_Wtime()` function. The columns having label R list the time consumed while reading the matrix, the vectors, and partitioning indicators and providing each processor with the necessary data. The columns having labels S list the time consumed during setting up the communication and local matrix data structures. The columns having labels M list the time for an SpMxV operation.

Except for the matrix *bcsstk25*, the row-column-parallel algorithm based on checkerboard partitioning is the best among algorithm-partitioning combinations. The algorithm based on fine-grain partitioning is the worst except for matrix *pig-very*. In order to investigate these results we give the communication pattern for parallel SpMxV computations. In Tables 3 and 4 we give the communication patterns for 1D partitioning and 2D partitioning based algorithms, respectively. For the checkerboard partitioning we assumed a processor mesh of size 6×4. In these tables, `TM` and `MM` correspond to total number of messages and maximum number of messages per processor; and `TV` and `MV` correspond to total volume of messages and maximum volume of messages per processor. These metrics refer to the *send* operations. Note that PaToH minimizes the total volume metric; in fine-grain partitioning case it minimizes the sum of the volumes in fold and expand steps; in checkerboard partitioning case it minimizes the total volumes in fold and expand phases separately. Note that in all cases except the *pig-very* matrix the total number of messages are doubled in fine-grain partitionings. Hence, even in the case of *memplus* in which total volume in fine-grain partitioning shrinks to 1/3 of other partitionings, the row-column-parallel algorithm based on fine-grain partitioning takes more time than

[1] http://www.cise.ufl.edu/~davis/sparse/
[2] http://math.nist.gov/MatrixMarket/

Table 2: Parallel times on 24 processors. R reading time(msecs.), S setup time(msecs.), M SpMxV time(msecs.).

	1D partitioning					
	Row-Parallel			Column-Parallel		
Matrix	R	S	M	R	S	M
memplus	1220	25	3.25	1260	27	3.14
bcsstk25	1950	53	1.53	2000	47	1.37
onetone2	2550	70	2.24	2600	39	1.97
pig-very	6900	1060	5.63	6980	139	6.72
lhr34	6590	54	6.02	6310	56	4.56
	2D partitioning					
	Fine Grain			Checkerboard		
Matrix	R	S	M	R	S	M
memplus	1890	42	4.95	1880	35	1.97
bcsstk25	3340	72	1.68	3360	38	1.66
onetone2	3880	58	3.77	3880	58	2.11
pig-very	9530	187	6.03	8950	165	5.46
lhr34	10850	82	6.23	10620	89	5.96

Table 3: Communication pattern for parallel computations based on 1D partitionings.

	Row-Parallel				Column-Parallel			
Matrix	TM	MM	TV	MV	TM	MM	TV	MV
memplus	522	23	12016	1070	509	23	10754	1689
bcsstk25	59	4	6855	377	62	5	6702	403
onetone2	132	7	5959	556	103	8	7890	835
pig-very	511	23	10196	1573	496	23	24172	3686
lhr34	233	15	24184	1482	239	13	24967	1323

other combinations. Note also that the checkerboard partitioning produces the smallest total number of messages in all cases. Combined with the advantage of bounding the maximum number of messages per processor, the checkerboard partitioning delivers the fastest SpMxV where there are significant differences on the metrics pertaining to number of messages.

7 Conclusion and future work

We have developed a sparse matrix vector multiplication library that works under 1D and 2D partitioning of sparse matrices. The matrices can be square and rectangular. The library can handle vector distributions that are different than the matrix distribution. In our implementation of the SpMxV routines, processors perform scalar multiplications as soon as the associated data are available.

We are going to implement a reference model for the vectors, matrices, and

Table 4: Communication pattern for parallel computations based on 2D fine-grain and checkerboard partitionings.

Matrix	Fine-grain partitioning							
	Expand				Fold			
	TM	MM	TV	MV	TM	MM	TV	MV
memplus	488	23	2407	347	472	23	2298	127
bcsstk25	47	4	1227	92	56	4	6094	362
onetone2	173	20	2783	349	132	10	3653	260
pig-very	525	23	12781	1553	61	5	169	19
lhr34	167	11	4185	449	234	13	22631	1533
Matrix	Checkerboard partitioning							
	Expand				Fold			
	TM	MM	TV	MV	TM	MM	TV	MV
memplus	118	5	5998	365	72	3	6005	556
bcsstk25	10	1	1679	230	48	3	5717	450
onetone2	35	4	1332	266	65	3	7360	647
pig-very	116	5	6200	714	72	3	17497	957
lhr34	60	5	11261	792	72	3	20131	1324

communicators using integers. The aims are to enable inter-operability of the library with Fortran codes and to hide the complexity of data structures.

We are going to implement a reference model for the vectors, matrices, and communicators. In doing so, the gain is two fold. First, we are going to use integers as references which will enable the inter-operability of the library within Fortran codes. Second, the data structures are a little bit cumbersome. Once we have designed the reference model these data structures will be hidden.

References

[1] B. Balay, W. Gropp, L. C. McInnes, and B. Smith. PETSc 2.0 users manual. Technical report, Argonne National Laboratories, 1996.

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

[3] Greg Burns, Raja Daoud, and James Vaigl. LAM: an open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[4] Ü. V. Çatalyürek and C. Aykanat. PaToH: A multilevel hypergraph partitioning tool, version 3.0. Technical Report BU-CE-9915, Computer Engineering Department, Bilkent University, 1999.

[5] Ü. V. Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE T. Par. Dist. Sys.*, 10(7):673–693, 1999.

[6] Ü. V. Çatalyürek and Cevdet Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Proceedings of 8th International Workshop on Solving Irregularly structured Problems in Parallel (Irregular 2001)*, April 2001.

[7] Ü. V. Çatalyürek and Cevdet Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proc. of Scientific Computing 2001 (SC2001)*, pages 10–16, Denver, Colorado, November 2001.

[8] J. Dongarra, V. Eijkhout, and A. Kalhan. Reverse communication interface for linear algebra templates for iterative methods. Technical Report UT-CS-95-291, University of Tennessee at Knoxville, 1995.

[9] Jack Dongarra. Freely available software for linear algebra on the web. http://www.netlib.org/utk/people/JackDongarra/la-sw.html, July 2001.

[10] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM TOMS*, 28:239–267, June 2002.

[11] I. S. Duff and C. Vömel. Algorithm 818: A reference model implementation of the sparse BLAS in fortran 95. *ACM Trans. Math. Software*, 28(2):268–283, June 2002.

[12] M. Hegland. Description and use of animal breeding data for large least squares problems. Technical Report TR-PA-93-50, CERFACS, Toulouse, France, 1993.

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

[14] B. Hendrickson and R. Leland. *The Chaco user's guide, version 2.0.* Sandia National Laboratories, Albuquerque, NM, 1995.

[15] S. A. Hutchinson, J. N. Shadid, and R. S. Tunimaro. The aztec user's guide - version 1.0. Technical report, Sandia National Laboratories, Albuquerque, NM, 1995.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995.

[17] Ali Pınar. *Combinatorial Algorithms in Scientific Computing.* PhD thesis, Department of Computer Science at University of Illinois at Urbana-Champaign, June 2001.

[18] Yousef Saad and Adrei V. Malevsky. P-SPARSLIB: A portable library of distributed memory sparse iterative solvers. Technical Report R95-71, Centre de Recherche en Calcul Applique, Jun 1995.

[19] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[20] R. S. Tunimaro, J. N. Shadid, and S. A. Hutchinson. Parallel sparse matrix vector multiply software for matrices with data locality. *Concurrency: Practice and Experience*, 10(3), March 1998.

[21] Bora Uçar and Cevdet Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *Siam J. Sci. Comp.*, submitted 2002.

[22] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. Technical report, Dept .of Mathematics, Utrecht University, May 2002.

[23] H. A. Van Der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of bi-cg for the solution of non-symmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.