A PVM-Based Library for Sparse Matrix Factorizations *

Juan Touriño and Ramón Doallo

Dep. of Electronics and Systems, University of A Coruña, Spain
{juan,doallo}@udc.es

Abstract. We present *3LM*, a C *L*inked *L*ist *M*anagement *L*ibrary for parallel sparse factorizations on a PVM environment which takes into account the fill-in, an important drawback of sparse computations. It is restricted to a mesh topology and is based on an SPMD paradigm. Our goal is to facilitate the programming in such environments by means of a set of list and vector-oriented operations. The result is a pseudo-sequential code, in which the interprocessor communications and the sparse data structures are hidden from the programmer.

1 Introduction

Sparse matrix operations appear in many scientific areas. Many libraries have been developed for managing sparse matrices, specially in linear algebra; for instance, the *NIST sparse BLAS* library [5] provides computational kernels for fundamental sparse matrix operations. This library is based on compressed storage formats which do not consider fill-in operations. Moreover, many linear algebra applications need to be solved in parallel due to memory and CPU requirements; so, parallel libraries such as *ScaLAPACK* [1], mainly oriented to dense computations, were developed. The *3LM* library was originally designed taking advantage of our experiences in programming sparse QR factorization algorithms on distributed-memory multiprocessors [6]. However, the routines of the library may be applied, without loss of generality, to several kinds of sparse algorithms involving fill-in (Cholesky, LU ...).

This paper is organized as follows: in §2 we describe the data structures and distributions available to users; §3 presents the programming model with *3LM*, focusing on the loop mapping. Different subsets of useful routines we have developed are briefly described in §4. A practical example of the use of *3LM* is shown in §5 and, finally, conclusions and future work are discussed in §6.

2 Library Data Structures and Distributions

The Linked List Column/Row Scatter scheme (*LLCS/LLRS*) was selected for representing sparse matrices, in order to support fill-in operations in a flexi-

* This work was supported by the Ministry of Education of Spain (project CICYT TIC96-1125-C03), Xunta de Galicia (XUGA20605B96) and by the Training and Mobility of Researchers Programme of the EU (ICARUS project at CINECA, Italy)

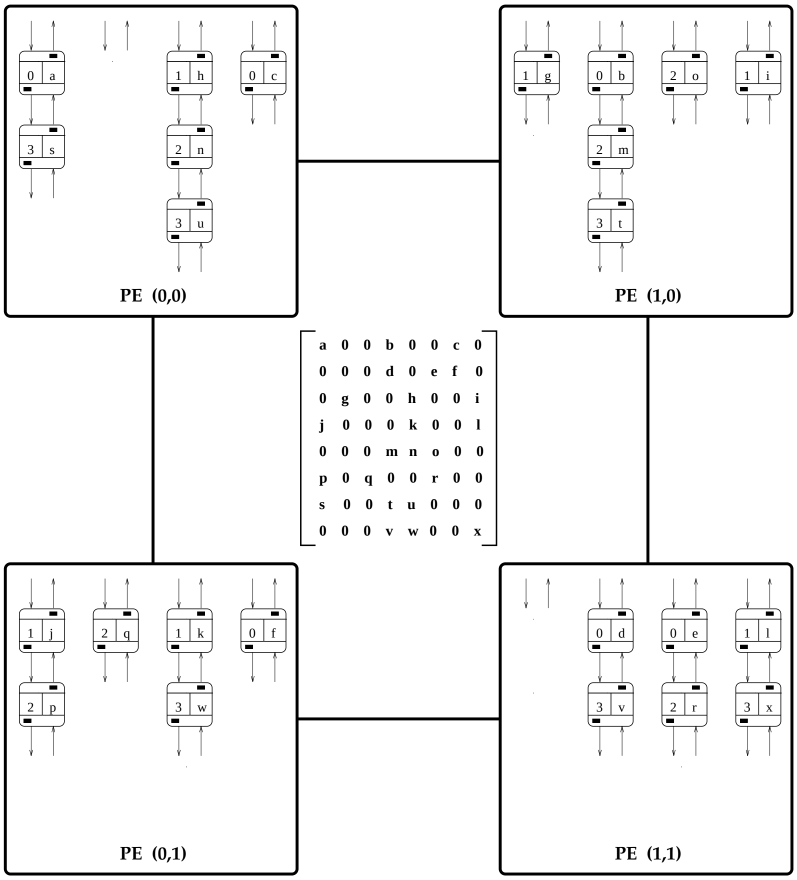

Fig. 1. *LLCS* scheme

ble way. It includes a data structure: linked lists, each one of them represents a column/row of the matrix, and a data distribution, a pseudo-regular cyclic distribution. This is accomplished by means of this *3LM* routine:

```
int  lll_dcs (char *file_n, int cols, dll *list_id)
```

where `file_n` is the name of the file in which the matrix is stored in Harwell-Boeing format [3] or in coordinate format; if `file_n`=0 (or NULL), the structure is only set up (this is necessary for sparse matrices which are generated at run-time); `cols` is the number of columns of the matrix, and `list_id` is an identifier of the matrix which contains the list orientation and an array of pointers to the beginning and to the end of the lists. In Figure 1, an 8×8 matrix is distributed onto a 2×2 mesh using this scheme. Similar routines have been developed [6] for a row-oriented scheme (`lll_drs`), for singly-linked lists (`lll_scs`, `lll_srs`), and for two-dimensional linked lists (`lll_srcs`, `lll_drcs`).

In matrix algebra, vector operations such as scalar-vector multiplication, vector addition, dot product, vector multiply or saxpy operation are very common. Besides, the characteristics of many problems force the programmer to manage vectors distributed (and even replicated) in a row or column of processors of the virtual mesh to take advantage of data locality and to minimize communications. In order to make this kind of operations easier, we use the routine:

```
void *lll_InitVector (char *file_n, int nels, vector *vec_id,
                       int dir, int dtype)
```

which distributes a dense vector of `nels` entries, stored in `file_n`, onto a mesh in a cyclic way, on each row or column of processors (that is, replicated), depending on `dir` (`XDirection` or `YDirection`); `dtype` is the data type of the elements: `DataInt`, `DataFloat`, `DataDouble` (predefined constants). This routine stores in `vec_id` (vector identifier) the following information: orientation of the vector (row-oriented or column-oriented), type of the elements and a pointer to the beginning of the vector, which is also returned by the routine.

A complete set of routines for data redistribution or reorientation, both for sparse matrices and for dense vectors, has been also developed in [6].

3 Loop Mapping

Let us assume a mesh topology in which each processor is identified by coordinates *(pidx, pidy)*. *3LM* programs must begin with the sentence:

```
lll_BeginMesh (npex, npey, progname)
```

which sets a mesh of dimensions `npey`×`npex` and executes the program `progname` in all the processors, using a PVM environment, for a cluster of workstations or for a Cray T3D/T3E multiprocessor (by setting `progname`=0/NULL). Programs must end with `lll_EndMesh()`.

As an example, considering that a matrix A is stored in an *LLCS* format, a double loop which performs a column access, is mapped on a double local loop, as shown in Figure 2.

For instance, let us consider the operation of multiplying the elements of the submatrix $A_{a:c,b:d}$ by a scalar constant named `value`. The corresponding pseudo-sequential *3LM* code is expressed as:

```
for (j=fcol(b); j<fcol(d+1); j++)
   lll_doper(j, listA_id, value, OpMul, frow(a), frow(c+1));
```

where `listA_id` is the matrix A identifier, `OpMul` is the operation (product) of each element of the list with `value`, from global row index `a` up to index `c`. The procedure `lll_doper` goes only through the links of the corresponding lists instead of traversing the whole iteration space `a:c`.

A similar procedure for vectors is `lll_voper`, which has a vector identifier `vec_id` as parameter instead of a scalar `value`; it operates each entry of list `j` with the corresponding entry of `vec_id`. There are predefined operations: `OpAdd, OpSub, OpMul, OpDiv, Nop`, as well as user-defined ones.

for (j=j1; j<j2; j++)   for (j=f_{col}(j1); j<f_{col}(j2); j++)
    for (i=i1; i<i2; i++) $\Longrightarrow$ for (i=f_{row}(i1); i<f_{row}(i2); i++)
        $A_{ij} = \cdots$ $A_{ij} = \cdots$

being

$$f_{col}(x) = \left\lfloor \frac{x}{npex} \right\rfloor + \begin{cases} 1 \text{ if } pidx < (x \ mod \ npex) \\ 0 \text{ otherwise} \end{cases}$$

$$f_{row}(x) = \left\lfloor \frac{x}{npey} \right\rfloor + \begin{cases} 1 \text{ if } pidy < (x \ mod \ npey) \\ 0 \text{ otherwise} \end{cases}$$

Fig. 2. Mapping global loops onto local loops

4 Library Routines

The *3LM* routines we have shown above and the ones we will see next, have been specified for column-oriented operations, that is, using an *LLCS* distribution for lists and using column vectors (`YDirection`). This was accomplished in order to simplify their explanation. There exist analogous procedures for singly-linked lists (`lll_s*`) and for 2-D linked lists (`lll_2d*, lll_2s*`) [6].

However, the same routines can be also applied to row-oriented operations (when using an *LLRS* scheme and row vectors) because these routines obtain the orientation from the identifiers of lists and vectors, and they operate accordingly.

Next, we introduce additional subsets of routines we found interesting for helping the user to program parallel algorithms in our application context.

4.1 Replication Operations

Sometimes, a column (in an *LLCS* distribution) of the matrix is required to perform calculations with data located in other processors.

The procedure `lll_drepl(j, list_id, vec_id, low, high)` replicates column (list) `j` of the matrix on the corresponding processors. This column is stored in the vector defined by `vec_id`, from entry with index `low` up to entry `high` (not inclusive). Internally, this procedure broadcasts a compressed vector instead of a full-size vector to reduce the size of the message to be broadcast. There are analogous procedures to replicate dense vectors (`lll_vrepl`).

4.2 Gather Operations

They are used for vectors which are distributed on each row (or column) of processors, and other processors need to obtain non-local data of these vectors. Function `lll_vgather(vec_id, j1, j2)` returns the value of entry `j1` of vector identified by `vec_id` to the processors which own entry `j2`. If `j2=All` (`All` is a predefined constant of the library), this value is returned to all the processors.

4.3 Reduction Routines

3LM provides a set of reduction instructions, both for lists and for dense vectors. For instance: `lll_vmaxval/lll_vminval(vec_id, low, high)` returns the maximum/minimum element of vector `vec_id`, from index `low` up to index `high`. Similarly, `lll_vmaxloc/lll_vminloc(vec_id, low, high)`, returns the index of the maximum/minimum element. There are also reduction routines for other operations such as sum, product ..., and for user-defined operations.

4.4 Fill-in Routines

In the sparse computations we are considering, an important drawback is the generation of new nonzero entries in the matrix, with the corresponding problems of storage and treatment of these entries. This is solved by means of the linked list data structure. Let us consider the following iteration space:

for (j=b; j<d+1; j++)
    for (i=a; i<c+1; i++)
        $A_{ij} = A_{ij} + vec_i$

where *vec* is a vector distributed and replicated in each column of processors. Fill-in appears in this computation and is confined in the local processor which executes its own set of iterations. We can solve this fact by using this routine:

```
for (j=fcol(b); j<fcol(d+1); j++)
   lll_dfillin(j, listA_id, vec_id, OpAdd, frow(a), frow(c+1));
```

where `OpAdd` is the operation between the entries of the list and vector `vec`. Generalizing, an operation g_{oper} can be a predefined or a user-defined function. According to this, the procedure `lll_dfillin` carries out the following actions:

$$If\ A^*_{ij} \leftarrow g_{oper}(A_{ij}, vec_i) \begin{cases} \neq 0\ and\ A_{ij} \neq 0 & Entry\ A_{ij}\ updated\ in\ the\ list\ as\ A^*_{ij} \\ \neq 0\ and\ A_{ij} = 0 & New\ entry\ A^*_{ij}\ inserted\ in\ the\ list \\ 0 & Entry\ A_{ij}\ deleted\ of\ the\ list \\ A_{ij} & No\ actions\ are\ taken \end{cases}$$

The routine `lll_dupdate(i, j, aij, list_id)` sets element `(i, j)` in the matrix identified by `list_id` to `aij` (insertion and deletion operations are assumed depending on the value of `aij`).

4.5 Swapping Operation

In many matrix calculations, explicit pivoting operations are required. This feature is a drawback in sparse computations due to the storage scheme and to the fill-in, which changes the number of elements per column of the matrix. A high-level swap operation is implemented to make the programming easier:

```
lll_dswap(j1, j2, list_id, rows)
```

being `j1` and `j2` the global indices of the columns to be swapped in the mesh according to the *LLCS* scheme, and `rows` is the row dimension of the matrix. Compressed vectors are used to reduce message sizes.

4.6 Other Routines

More remarkable procedures for list management, among others developed, are:
`lll_dvdp(j, list_id, vec_id, low, high)` returns to all the processors which own list (column) `j` the dot product of that column and vector `vec_id`.
`lll_dunpack(j, list_id, vec_id, low, high)` copies elements of list (column) `j`, from index `low` up to index `high` on the corresponding positions of the column vector defined by `vec_id`; the rest of entries of this vector are zero.

There are also *3LM* low-level routines [6] to handle directly the data structures, as well as to determine the actions on each processor of the mesh, for special operations which cannot be performed with the high-level set described above.

5 Sparse QR Factorization: an Application Example

The code of Figure 3 shows an example of the use of the *3LM* routines for the rank-revealing sparse Modified Gram-Schmidt (MGS) QR algorithm, with column pivoting. An `m`×`n` matrix A is decomposed into the product of an orthogonal matrix Q (which is originally matrix A) and an upper triangular matrix R (consult [4, Chap.5]). Lines 27-37 of Figure 3 correspond with the column pivoting stage of the algorithm. The generation of each row k of matrix R is performed in line 40: $R_{k,k} \leftarrow pivot$, and in line 47: $R_{k,k+1:n-1} \leftarrow Q^T_{0:m-1,k} Q_{0:m-1,k+1:n-1}$. Finally, the core of the stage of updating matrix Q is carried out in line 41: $Q_{0:m-1,k} \leftarrow Q_{0:m-1,k}/pivot$, and principally in line 51, where fill-in appears: $Q_{0:m-1,k+1:n-1} \leftarrow Q_{0:m-1,k+1:n-1} - Q_{0:m-1,k} R_{k,k+1:n-1}$.

A detailed parallel implementation which uses message-passing routines explicitly is described in [2]. As we can see, the *3LM* code is not very broad, whereas the corresponding parallel code mentioned above can fill about 2000 lines.

Figure 4 shows the efficiencies obtained for the code of Figure 3 on a Cray T3E, for five sparse matrices selected from the Harwell-Boeing collection [3]. A strategy to preserve sparsity during the factorization was also included. The legend of this figure indicates the dimensions of the matrices, as well as the number of nonzero entries. As we can see, the algorithm scales rather well. Nevertheless, the execution times are not good in comparison with the implementation which uses message-passing explicitly. This is because that implementation is very optimized and the *3LM* routines are generic, not specific for a particular algorithm. The ease of programming using *3LM* involves higher execution times.

This library has been also used to program other sparse orthogonal factorizations, such as Householder reflections and Givens rotations, using *LLCS* and *LLRS* schemes, respectively [6]. A 2-D linked list (*LLRCS* scheme) would be suitable for a right-looking LU factorization. Sparse Cholesky factorization can also be approached by means of the *3LM* library, using an *LLCS* distribution.

```
#include "lll.h"

void main()
{
 int m, n, pesx, pesy;
 int i, j, k, rank, pivot_index;
 double pivot, tempnorm;
 double *norm, *vsum, *vcol, *temp;
 vector norm_id, vsum_id, vcol_id, temp_id;
 dll listQ_id, listR_id;

 pesx=4; pesy=4;
 lll_BeginMesh(pesx, pesy, "qr_mgs");
 m=1000; n=1000;
 norm=lll_InitVector(0, n, &norm_id, XDirection, DataDouble);
 vsum=lll_InitVector(0, n, &vsum_id, XDirection, DataDouble);
 vcol=lll_InitVector(0, m, &vcol_id, YDirection, DataDouble);
 temp=lll_InitVector(0, m, &temp_id, YDirection, DataDouble);
 lll_dcs("matrix.dat", n, &listQ_id);
 lll_dcs(NULL, n, &listR_id);
 rank=n;
 for (j=fcol(0); j<fcol(n); j++) {
    lll_dunpack(j, listQ_id, temp_id, frow(0), frow(m));
    norm[j]=lll_dvdp(j, listQ_id, temp_id, frow(0), frow(m));
   }
 for (k=0; k<n; k++) {
    pivot=lll_vmaxval(norm_id, fcol(k), fcol(n));
    pivot_index=lll_vmaxloc(norm_id, fcol(k), fcol(n));
    if (pivot < 1.0e-20) {
       rank=k; break;
      }
    lll_dswap(k, pivot_index, listQ_id, m);
    lll_dswap(k, pivot_index, listR_id, n);
    tempnorm=lll_vgather(norm_id, k, pivot_index);
    for (j=fcol(pivot_index); j<fcol(pivot_index+1); j++)
       norm[j]=tempnorm;
    pivot=sqrt(pivot);
    for (j=fcol(k); j<fcol(k+1); j++) {
       for (i=frow(k); i<frow(k+1); i++)
          lll_dupdate(i, j, pivot, listR_id);
       lll_doper(j, listQ_id, pivot, OpDiv, frow(0), frow(m));
      }
    lll_drepl(k, listQ_id, vcol_id, 0, m);
    for (j=fcol(k+1); j<fcol(n); j++) {
       vsum[j]=lll_dvdp(j, listQ_id, vcol_id, frow(0), frow(m));
       for (i=frow(k); i<frow(k+1); i++)
          lll_dupdate(i, j, vsum[j], listR_id);
       norm[j]=norm[j]-vsum[j]*vsum[j];
       for (i=frow(0); i<frow(m); i++)
          temp[i]=vcol[i]*vsum[j];
       lll_dfillin(j, listQ_id, temp_id, OpSub, frow(0), frow(m));
      }
   }
 lll_EndMesh();
}
```

Fig. 3. Sparse MGS code using *3LM* routines

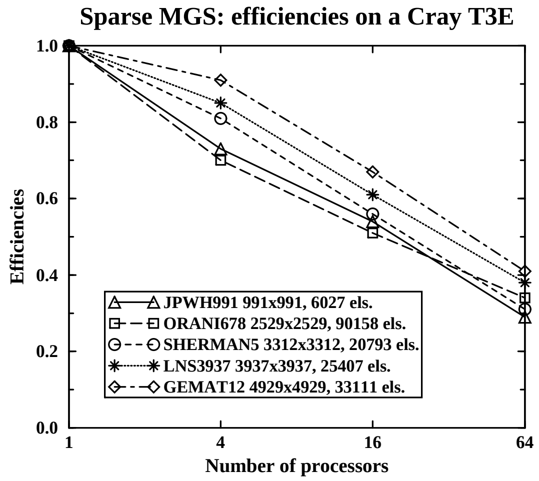

Fig. 4. Efficiencies for the MGS algorithm using *3LM* routines

6 Conclusions and Future Work

3LM provides an environment to develop effortlessly codes in the field of sparse direct factorizations and their applications (linear systems of equations, least squares problems ...). Besides, some of these routines can be used to extend the capabilities of a data-parallel compiler to handle sparse matrix computations [6]. As future work, we intend to code this library using Fortran 90/MPI, as well as to extend the library to include a wider class of problems.

References

1. Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK: a Portable Linear Algebra Library for Distributed Memory Computers -Design Issues and Performance. Tech. Report CS-95-283, Dep. Computer Science, University of Tennessee (1995).
2. Doallo, R., Fraguela, B.B., Touriño, J., Zapata, E.L.: Parallel Sparse Modified Gram-Schmidt QR Decomposition. In Int'l Conference on High-Performance Computing and Networking, HPCN'96, Brussels. Lecture Notes in Computer Science, Vol. 1067, Springer-Verlag (1996) 646–653.
3. Duff, I.S., Grimes, R.G., Lewis, J.G.: User's Guide for the Harwell-Boeing Sparse Matrix Collection. Tech. Report TR-PA-92-96, CERFACS (1992).
4. Golub, G.H., van Loan, C.F.: Matrix Computations. The Johns Hopkins University Press, second edition (1989).
5. Remington, K.A., Pozo, R.: NIST Sparse BLAS User's Guide. National Institute of Standards and Technology (1996).
6. Touriño, J.: Parallelization and Compilation Issues of Sparse QR Algorithms. PhD Thesis, Dep. of Electronics and Systems, University of A Coruña, Spain (1998).