LAPACK working note 50
Distributed Sparse Data Structures for Linear Algebra Operations *

Victor Eijkhout
Department of Computer Science
University of Tennessee, Knoxville
eijkhout@cs.utk.edu

September 15, 1992

Abstract

Distributed data structures for matrices and vectors representing sparse data, both structured and unstructured, are described. For unstructured data it is described how processors can derive communication information from the data structure.

1 Introduction

On distributed memory computers data structures are more complicated than on shared memory computers. Ideally, every processor handles a certain set of variables and needs only the data pertaining to those variables. However, since operations such as the matrix-vector product involve combining data from variables that may belong to different processors, the data structures need to be extended with connectivity information.

This paper will describe extended data structures for both problems on structured and unstructured grids. Rather than talking about processors directly, the discussion here will describe the data structures pertaining to *regions* in the physical domain. Multiple such regions may be assigned to a single processor.

Distributed storage of sparse matrices was considered in [4], where it was shown that 'integrity preserving' random assignments of nonzeros to processors have a high probability of generating an even distribution for matrices with limited numbers of nonzeros per row/column.

*. This work was supported by DARPA under contract number DAAL03-91-C-0047

2 Regular grid problems

On grids that are (topologically) a Cartesian product of intervals both vectors and matrices can be represented simply in terms of Fortran arrays. The easiest way to select a region from such a grid is to let it be a product of subintervals, so that a vector x can be allocated as

```
real x(ipts,jpts)
```

Sparse problems typically come from difference equations, such as the five point central difference stencil, where the matrix-vector multiplication $y = Ax$ on a point (i, j) of the grid takes the form

$$y_{ij} = a_{ij}x_{ij} - b_{ij}x_{ij+1} - c_{ij}x_{ij-1} - d_{ij}x_{i+1j} - e_{ij}x_{i-1j}.$$

In distributed computation, for some value of i and j one or more of the neighbouring elements of x will be part of another region.

It is obvious that some data will have to be moved from the owning region of the input data to the region storing the final result. Less clear is which of the two should do the computation involving the input data. Suppose that we are computing $b_{ij}x_{ij+1}$ and that x_{ij+1} is owned by a neighbouring region. In the usual regime the region that contains location $(i, j+1)$ will send x_{ij+1} to the region that has (i, j), and there the multiplication will be performed. This is called 'owner computes', but 'writer computes' would be a more accurate term, reflecting that a region does all of the computation that will be written in it.

Alternatively, following a 'reader computes' rule, the owner of $(i, j+1)$ can do all of the computation using its data as input, in this case performing the multiplication $b_{ij}x_{ij+1}$ and sending the completed result to the owner of location (i, j).

Since the matrix of a five-point stencil is structurally the Cartesian product of two tridiagonal matrices, we consider the multiplication with a tridiagonal matrix as atomic for the moment. This corresponds to an operation

$$y_i = a_i x_i - b_i x_{i-1} - c_i x_{i+1},$$

and let us assume for the moment that we are considering computing this for the range $i = 1, \ldots, n$.

In the 'writer computes' regime additional input elements x_0 and x_{n+1} are needed, so if we allocate

```
real x(0:n+1)
```

the above three-term averaging can be performed for all elements $1..n$ without exceptional conditions on the boundary. It corresponds to a matrix multiplication $y = Hx$ if the coefficients a_i, b_i, c_i are stored in a tridiagonal matrix H as

$$H_{i,i} = a_i, \qquad H_{i,i-1} = -b_i, \qquad H_{i,i+1} = -c_i.$$

Note that we require that the matrix includes nonstandard element $H_{1,0}$ and $H_{n,n+1}$. Storing this matrix H in a Fortran array `H` can be done by allocating

```
real H(n,-1:1)
```

with the conversion convention

$$\texttt{H(i,j)} = H_{i,i+j}\,.$$

This allocates the three nonzero diagonals of the matrix in contiguous storage, so that the matrix-vector multiplication can be performed by diagonals [3]. Allocating the matrix as

```
real H(-1:1,n)
```

puts the rows of the matrix in consecutive storage, and requires a conversion convention

$$\texttt{H(i,j)} = H_{i+j,j}\,.$$

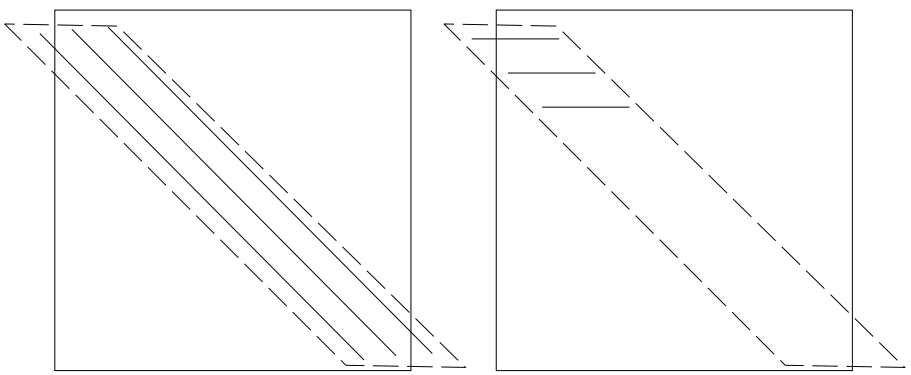

Figure 1: Diagonal and row storage for 'writer computes' rule.

Note that, whereas the input vector requires two additional elements, the output vector can simply be allocated with the elements $1..n$.

In the 'reader computes' regime we need no additional input elements, but we perform the extra computations $b_{n+1}x_n$ and c_0x_1, to be sent to the right and left neighbouring region respectively. Storing the a_i, b_i, c_i coefficients again in a tridiagonal matrix H, we now need nonstandard elements $H_{-1,1}$ and $H_{n+1,n}$. Such a matrix can be stored in a Fortran array declared as

```
real H(n,-1:1)
```

(storing diagonals contiguously) with a conversion convention

$$\texttt{H(i,j)} = H_{i+j,i},$$

or (storing columns contiguously) as

```
real H(-1:1,n)
```

with a conversion convention

$\mathtt{H(i,j)} = H_{i+j,j}$.

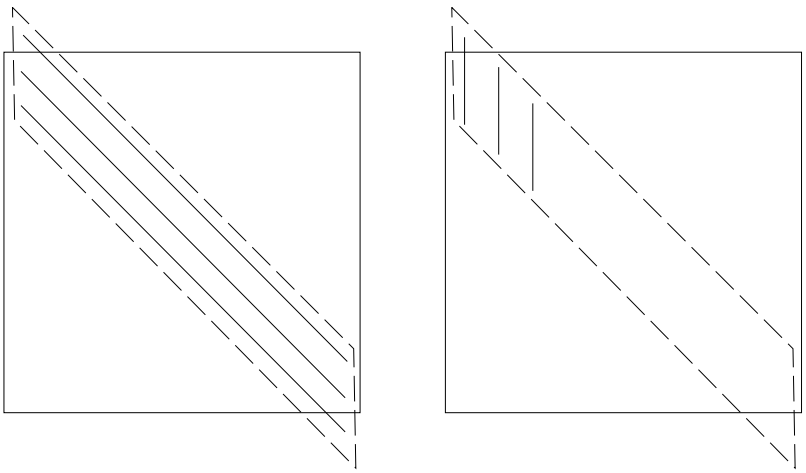

Figure 2: Diagonal and column (Linpack) storage for 'reader computes' rule.

This latter storage mode is the band storage used in Linpack [2] and Lapack [1].

Generalizing the above storage modes to multidimensional problems implies that vectors may have to be stored as

```
real x(0:ipts+1,0:jpts+1, ... )
```

and that the off-diagonals of the matrix have to contain some nonstandard elements.

3 Irregular grid problems

If the physical domain no longer has a simple product topology it becomes harder to know which variables border on a certain region. Thus, in addition to the real array storing values of variables a number of integer arrays with information about the domain, its partitioning into regions, and the connectivity of those regions, will have to be declared.

3.1 Assumptions

In a distributed computing environment it is not only a question how connectivity is arranged, but also how communicating regions get to know this

information. The discussion in this section will assume that a matrix has been centrally constructed, and that processors receive some part of it and construct the connectivity information themselves based on what information about the matrix they receive.

A 'writer computes' mode of computation is assumed.

3.2 Vector storage

Since for irregular grids the concept of dimension of the physical domain disappears, we have to store variables in a linear array. As in the case of regular grids above, we store for a region both the owned variables and the bordering variables the information of which is needed to compute values of the owned variables.

Even under the assumption that the global numbering of variables is such that every region has variables that are numbered consecutively, the numbers of bordering variables need not obey any pattern (for the regular grids they came in sequences with constant stride). Thus, skips in numbering between bordering variables and local variables can be arbitrarily large, and since we want to keep local storage to a size proportional to the number of local values, we have to remap the global numbering into a local numbering that is contiguous. We need

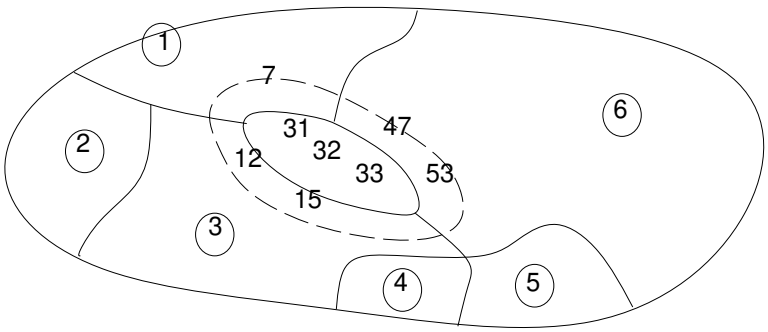

Figure 3: Example domain, with owned and bordering variables of a certain region indicated.

one real array for the data, and an integer array of the same size with the renumbering information:

```
real x(n_vars)
integer global_num(n_vars)
```

In order to distinguish between owned and bordering variables it is necessary to have another integer array:

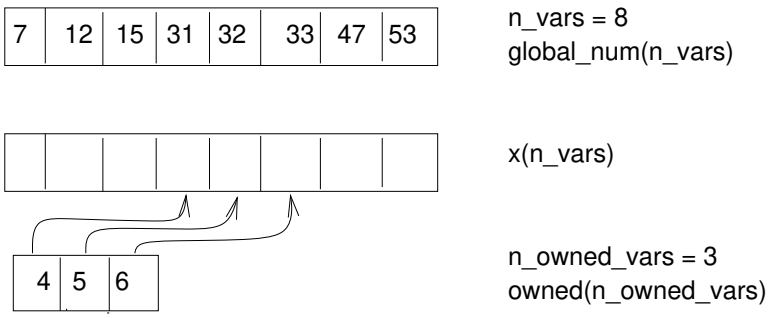

Figure 4: Remapped input vector and information on which indexes are owned.

```
integer owned(n_owned_vars)
```

Computation then will take the form

```
do i=1,n_owned_vars
    x(owned(i)) = ...
end do
```

Under the simplifying assumption that all owned variables in a region are numbered consecutively we can reduce this to a single integer storage

```
integer owned_low
```

with the computation taking the form

```
do i=owned_low,owned_low+n_owned_vars-1
    x(i) = ...
end do
```

3.3 Connectivity information

For the bordering variables it is necessary to know to what bordering region they belong, or rather, for each region sending bordering values it is necessary to know where these have to be stored in the not-owned portion of the vector. To this end we need two integer arrays

```
integer in_locs(n_in_vars)
integer in_regions(n_in_regions+1)
```

such that if region i sends incoming data, the number of items is

$$\texttt{in_locs}(i+1) - \texttt{in_locs}(i)$$

and they have be stored in $\texttt{x}(\texttt{in_locs}(k))$ for

$$k = \texttt{in_regions}(i), \ldots, \texttt{in_regions}(i+1) - 1.$$

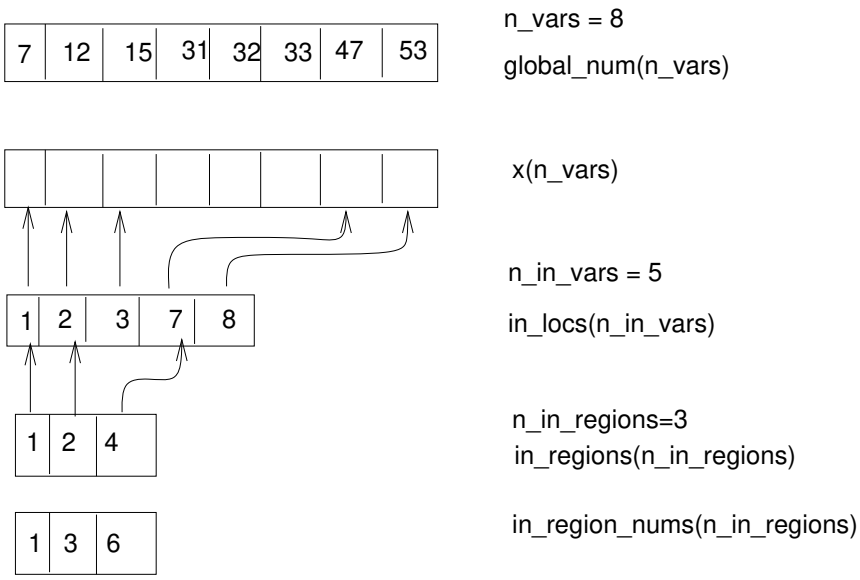

Figure 5: Pointer structure for incoming data items.

Analogously, a pointer structure is needed for sending outgoing data items to bordering regions.

3.4 Matrix storage

In this section we will consider an extension of Compressed Row Storage to distributed computation. Ordinary CRS is based on one real and two integer arrays

```
real matrix_elements(n_nonzero)
integer column_nums(n_nonzero)
integer row_first_locs(n_rows)
```

The nonzero elements of row i are stored in locations

$$\texttt{row_first_locs}(i)..\texttt{row_first_locs}(i+1)-1$$

of `matrix_elements`, where the corresponding element in `column_nums` gives the column number of the nonzero.

We now consider the case where a processor handling a region receives the rows corresponding to the variables of that region. First of all we note that, because of the remapping from global to local variable numbering, the array `column_nums` has to be updated accordingly.

Strictly speaking, a region needs only the rows corresponding to its owned variables in the course of the computation. However, in addition arrays such as `in_regions` are needed, and if possible we want to construct those in a distributed manner. The following two assumptions make that possible.

1. *Each region knows the partitioning of all variables over the set of regions.*
2. *Each region has all rows of the global matrix that have a nonzero coefficient a_{ij} where i or j is an owned variable.*

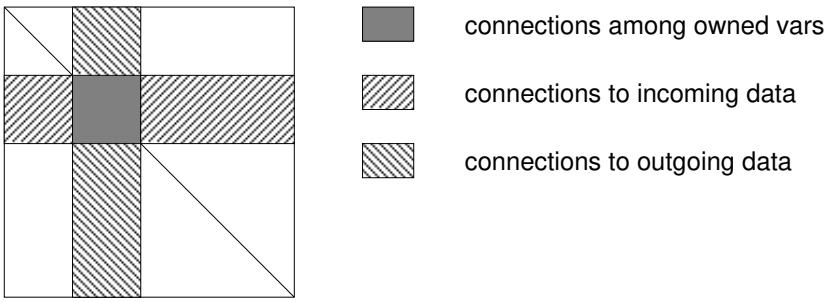

Figure 6: Nonzero structure needed for parallel matrix operations.

In effect, we require each processor to have the nonzero variables in the shaded regions in figure 6.

A few remarks about this.

- If the variables are numbered in such a way that each region ownes a block of consecutively numbered variables, then the first assumption can be satisfied with for each region only two integers per region extra storage.
- The rows a_{i*} for which i is an owned variable will be named 'essential' since they are necessary for the computation of Ax under the 'writer computes' rule. Rows a_{i*} for which i is not owned will be called 'non-essential'. However, we will see in the next section that they make it possible that a region can construct its own connectivity information.
- Including some non-essential rows gives each region those rows that are necessary to compute the matrix-vector products with both A and A^t. The multiplication with A^t involves some, but not all, elements in the non-essential rows of A.
- The extra rows are those values of i for which there is a j such that $a_{ij} \neq 0$ and j is an owned variable. Such i-values correspond to variables bordering on the region, and their number is usually of a lower order than the number of variables in the region. Thus the amount of extra storage needed is not prohibitive.
- From the previous point it follows that the number of rows is `n_vars` (see section 3.2), that the array `global_num` describes what rows they are, and

that the array `owned` gives the numbers of the rows that are needed for the matrix-vector product with A.

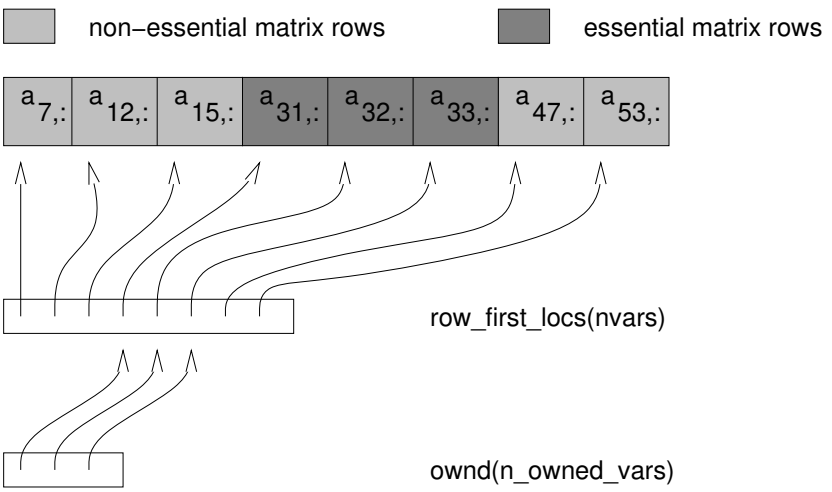

Figure 7: Row compressed storage of the essential and non-essential matrix rows.

3.5 Construction of connectivity information

With the extended matrix described above, it is easy for a region to construct its connectivity information. Let r be a neighbouring region, then

- region r sends variable j as incoming data if j is not an owned variable and $a_{ij} \neq 0$ for some i that is an owned variable;
- region r is sent variable j as outgoing data if j is an owned variable and $a_{ij} \neq 0$ for some i that is not an owned variable.

Note that a region knows what incoming data to expect from its essential rows, but that it needs the non-essential rows to figure out what outgoing data to send. For the incoming data, the sending region determined the need for this from its non-essential rows.

In conclusion we can state that, under the assumption that the regions know how the variables are partitioned, it is enough if each region has copies of certain matrix rows as described above, plus the array `global_num`.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.

[2] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK Users' Guide*. SIAM, 1979.

[3] N.K. Madsen, G.H. Rodrigue, and J.I. Karush. Matrix multiplication by diagonals on a vector/parallel processor. *Inform. Proc. Letters*, 5:41–45, 1976.

[4] Andrew T. Ogielski and William Aiello. Sparse matrix algebra on parallel processor arrays. Technical report, Bell Communications Research, 1991.