Sparse DDL version 2.1
Technical Guide

Tim Oliver
Institute for Advanced Scientific Computation
University of Liverpool

Sunday 20th August 1995

Contents

1 Introduction **2**

2 Distributed objects **3**
2.1 Basic object types . . . 3
2.1.1 Dense vectors . . . 3
2.1.2 Sparse matrices . . . 3
2.2 Distributions . . . 4
2.3 Properties . . . 6
2.4 Permutations . . . 6
2.5 Communication patterns . . . 7
2.6 Extending DDL objects . . . 7

3 Data structures **9**
3.1 The header . . . 9
3.2 Basic distributed objects . . . 10
3.3 The property list . . . 12
3.4 Communication properties . . . 12
3.5 Permutation property . . . 16
3.6 Example sparse matrix data structure . . . 16

4 The future of the DDL **18**

A Example DDL procedure: sparse matrix-vector multiply **19**
A.1 Description . . . 19
A.1.1 DDL_Smv_comm . . . 19
A.1.2 DDL_Smv . . . 20
A.2 Listing . . . 21

Chapter 1

Introduction

WARNING: This document is the Technical Guide to version 2.1 of the DDL. The specification of the Library is still under review and the implementation is far from complete. Thus any similarity between the Library system described here and the Library software is entirely fortuitous!

You have been warned!

Any comments will be gratefully received…and maybe even acted upon!

This document describes the Sparse Distributed Data Library (Sparse DDL, or simply DDL), a library intended to ease the development of parallel sparse matrix applications. This document describes the design of the Library and its data structures, and shows how Library procedures may be written. It should be read in conjunction with the Library User's Guide [1] which details the high-level user interface to the Library. The Sparse DDL is built on a library providing core 2 dimensional distributed objects for image processing applications [2].

The document is arranged as follows. The next chapter gives an overview of distributed objects as implemented by the DDL. Then in Chapter 3 we describe the data structures used to implement the hierarchical, distributed object model. Finally, in Chapter 4 we sketch out ways in which the DDL could be enhanced in the future. Descriptions of two of the high-level procedures included in the DDL are given in Appendix A. These descriptions are intended to illustrate how Library procedures may be written.

Chapter 2

Distributed objects

The DDL uses the concept of opaque objects for the distributed data structures that it supports. This means that, in general, the user does not have direct access to DDL data, but instead manipulates the data through calls to DDL procedures. This hides the details of the implementation of the data structures, allowing the implementation to be changed at a later date, and also helps to avoid errors due to user programming. The user passes a DDL object to a procedure by providing a handle to that object. The DDL manages system memory, allocating space for new objects and deallocating unwanted objects. The DDL also allows a user to have direct access to the raw data values of a distributed object.

The DDL supports several different types of distributed objects and allows new types to be added. Currently, the DDL supports vector and sparse matrix distributed objects. Eventually we intend to support dense matrices as well.

For each type of object the DDL may support a number of different formats. Each of the different formats specifies the particular data structure that should be used to represent the object. The features of the different data structures may make one structure better than another for a particular application. Thus the choice of data structure is left to the user, so that he can select the optimal structure for his application. However, one of the aims of the DDL is to hide the details of the data structure and the data manipulations required to perform an operation. Hence DDL procedures should accept objects of any type compatible with the specified operation. As examples, I/O procedures should be able to read and write any of the object types, and a matrix-vector multiply procedure should accept both dense and sparse matrices of any type.

2.1 Basic object types

2.1.1 Dense vectors

Dense vector objects of type DDL_Vec are supported.

2.1.2 Sparse matrices

The DDL supports a sparse matrix object, DDL_Spa. Currently, sparse matrices must be square.

Single process sparse matrices

On a single process three vectors are used to represent a sparse matrix, `ia`, `ja` and `a`. `ia` and `ja` are vectors of type integer and `a` is a vector of type real or double. These three vectors are used for three main data structures for sparse data called the triad, row packed and column packed data structures.

With triad format, elements `ia(j)` and `ja(j)` contain the x and y indices of the matrix coefficient value in `a(j)` (see Figure 2.1). The elements are unordered.

In the row packed format (see Figure 2.2), `ia` contains a set of pointers, one for each row of the sparse matrix. These pointers point into the `ja` and `a` vectors to the start of matrix elements in that row of the

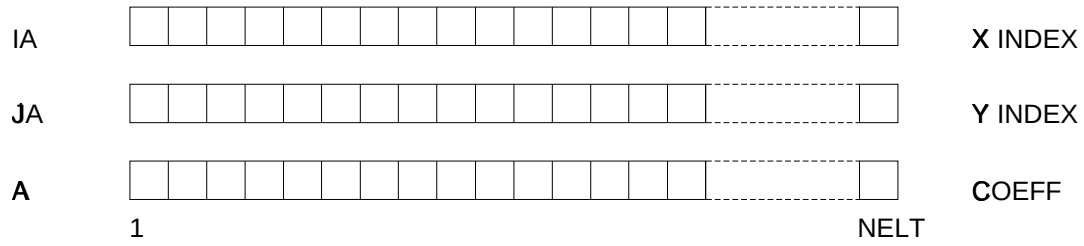

Figure 2.1: Triad format

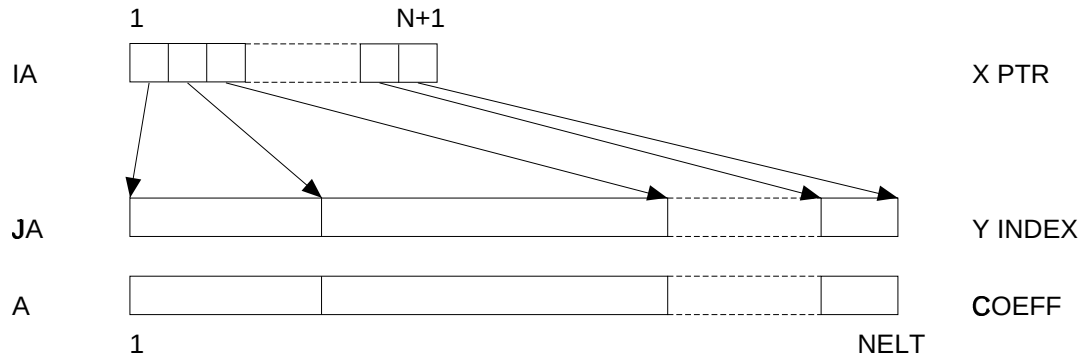

Figure 2.2: Row packed format

matrix. `a` again contains the coefficient values from the matrix and `ja` contains the column indices for the coefficients. Elements within each row are ordered from lowest column number to highest.

In the column packed format `ja` contains a set of pointers to the start of each column in `ia` and `a`, and `ia` contains the row indices of the coefficients stored in `a`. Elements within each column are ordered from lowest row number to highest.

In the row (and column) packed formats, for a matrix with `n` rows (columns) and `nelt` non-zero elements the pointer vector has element `n+1` set to `nelt+1`.

Distributed sparse matrices

Distributed sparse matrix objects are of type DDL_Spa.

The distributed sparse matrix formats are based on the single process sparse matrix formats.

Currently, the only fully supported format is the packed row distributed sparse matrix format, DDL_PCK_ROW. Many other distributed sparse matrix formats have been proposed including padded formats, where each row (or column) of the matrix is padded out with extra empty slots for later fill-in, and triad formats, where elements are stored as triads. The User Guide [1] includes more details of these proposed formats.

The distributed row packed sparse format is based on the row packed format (see Figure 2.3). The sparse matrix is distributed over the processes by whole rows, with each process holding a block of adjacent rows. On each process the rows are stored in the single process row packed format with minor changes. The variables `n` and `nelt` now refer to the number of rows held on a process and the number of elements on that process respectively. `ia` only has pointers for rows held by a process, with the rows renumbered locally from one.

2.2 Distributions

All the objects that the DDL supports have the same general distribution pattern (see Figure 2.4). This distribution is a subset of the distributions provided by HPF.

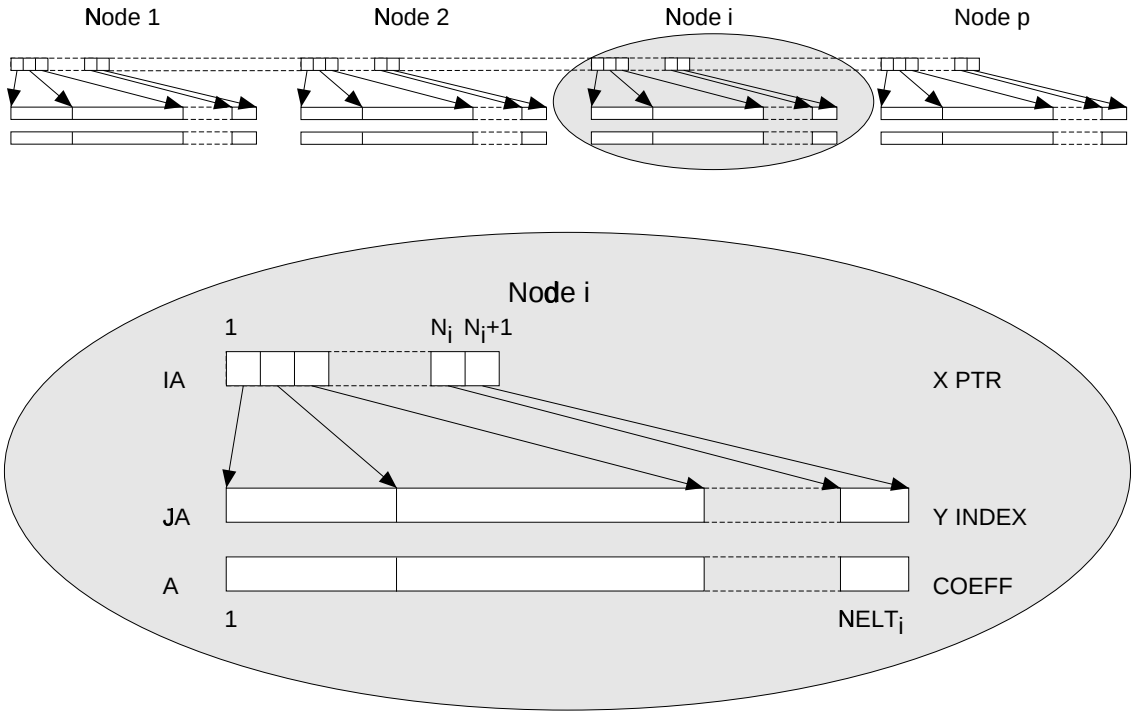

Figure 2.3: Distributed row packed format

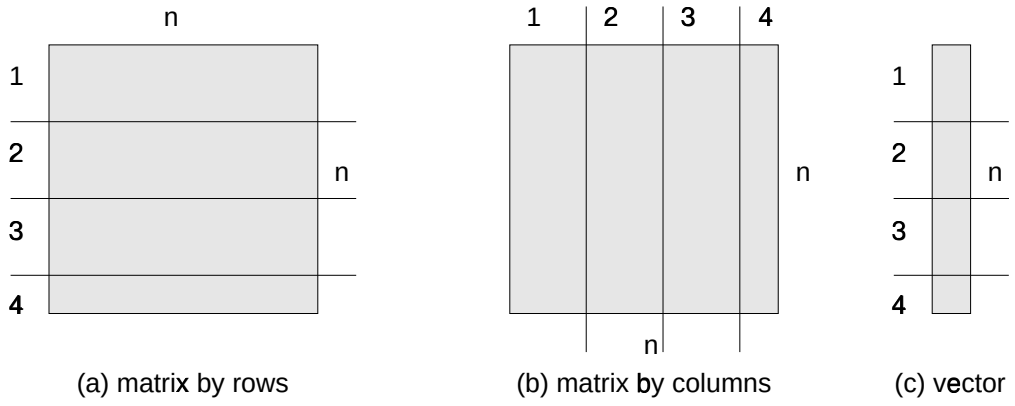

Figure 2.4: Distribution of matrices and vectors

The distribution of a vector of length n from a program running on p_{max} processes is as follows. The vector is partitioned into blocks of contiguous elements, with all processes but the last having the same number of elements. This might mean that not all of the p_{max} processes can be used. The size of these blocks is $b = \lceil n/p_{max} \rceil$. The actual number of processes that can be used is then $p = n/b$. The last process block is of size $n - b(p-1)$.

A matrix may be distributed by either rows or columns. The rows (or columns) of the matrix are partitioned into contiguous blocks in the same manner as for a vector.

The block size used by this distribution does not provide the most even balance across the processes. Neither does it always make use of all the available processes. However, in many instances it is the easiest partitioning to work with since you know that all used processes except the last have the same block size.

Objects can also be distributed using a parameter, block, which specifies that blocks of block contiguous elements are indivisible. This forces the distribution mechanism to partition the dimension into blocks which are a multiple of block in size.

2.3 Properties

It is often useful to be able to associate information or properties with a distributed object. The DDL implements a property list to allow the user to specify optional properties for an object. The Library provides procedures to add, find and remove properties from the property list. Properties can include such things as permutations (Section 2.4) and communication patterns (Section 2.5).

The property list also allows the user to extend the functionality of the Library by defining new properties for distributed objects (see Section 2.6).

For more information about the property list see Section 3.3.

2.4 Permutations

The DDL implements a permutation property which allows the user to specify a reordering of an object's data. The permutation specifies a 1-to-1 mapping between the indices of an object's data in physical storage and the logical data indices. The physical index of an element is the index used to locate that element in distributed memory. The logical index of an element is the "mathematical" index for that element. For example, the element of a vector, $v(i)$, has a logical or mathematical index i, but this element might be mapped to a physical storage location with index j, i.e., $v(j)$. A permutation is represented by a vector p such that $p(i)$ specifies the logical index of physical index i.

Currently, only the sparse matrix object makes use of this property, allowing the user to specify a row and/or column permutation for the matrix. Permuting the rows and/or columns of a sparse matrix is often useful to get the nonzeros into a structure that is more suited to the application.

Note that specifying a permutation property does not cause the object's data to be redistributed using the new indices. The physical storage remains the same, with the permutation property specifying the new logical indices. This avoids a potentially very costly communication operation. It would be useful to have procedures which redistributed the data according to a given permutation but these have not been implemented yet.

Note also that not all DDL procedures use the logical indices given by the permutation properties. The mathematical procedures, such as matrix-vector multiply and triangular solve, do use the logical indices, but the I/O procedures still use the physical indices.

Permutations may increase the cost of an operation such as the matrix-vector multiply or triangular solve. This is because the regular alignment of elements given by the initial distribution may be lost once a permutation is applied. Thus more communication may well be required to fetch all the data that a procedure needs. The increase in cost when permutations are applied will vary from procedure to procedure and between different permutations. In view of this, it may be worth considering permuting the entire input data set. Thus the data is distributed in its permuted form and no additional permutation properties are required. Now the procedures can use the more efficient non-permuted algorithms. If different permutations are required at different points in an application the permutation properties can be changed accordingly.

For more information about the implementation of permutations see Section 3.5.

2.5 Communication patterns

Communication systems for distributed memory parallel systems, such as MPI and PVM, have recognised the need for flexible, efficient, general purpose communications procedures. In particular, these systems provide all-to-all collective operations that exchange blocks of contiguous elements between processes. However, for sparse matrix communications the elements are usually not contiguous in memory. Using MPI or PVM, the sender must therefore prepack the scattered elements into a buffer before calling the communication procedure. The receiver might then have to unpack the elements from the receive buffer before continuing. Recognising that such operations are very common in sparse matrix applications the DDL defines communication properties for sparse matrix objects. The communication properties provide the data necessary to invoke an all-to-all collective communication of scattered data elements. Associated with the communication property data structures is the DDL all-to-all collective communication procedure.

By holding these communication patterns as object properties they can be calculated once at the start of a user program and then reused each time the relevant Library procedure is invoked. This reduces the run-time of a program at the expense of more memory.

The communication patterns can be described as follows. Each process occupied by an object has a number of data items (these might be elements of a vector for example). The communication pattern specifies for each process a subset of these data items which should be sent to each other process. A process can send different data subsets to different processes including an empty set. The data structure holds the indices of data items to be sent to each process.

These data structures are intended to be general purpose but a number of slightly different communication data structures may be required by different applications. Currently, the Library defines two communication property structures, one with the information ordered by process number, and the other with the information ordered by index number.

For more information about the implementation of communication properties see Section 3.4.

2.6 Extending DDL objects

Currently, only a small number of distributed objects are supported, but the design of the DDL is intended to be flexible and allow for enhancements to the data structures. These enhancements can take place on two levels. Firstly, each DDL object includes a property list (see Section 2.3), which allows the user to associate particular properties with a distributed object, such as a permutation. The property list allows the user to extend the functionality of the Library by defining new properties for distributed objects. For example, future solvers will need new communications structures.

The second way in which a user can extend the Library is by defining new high-level distributed object types. These new objects are defined in terms of the existing distributed objects. Defining new high-level objects can be used to hide data complexity from the calling program. For example, if an application needs to manipulate a matrix in its LU factorized form then a new DDL sparse matrix distributed object can be defined which contains the sparse L and U sub-matrices. A possible structure for this object is shown in Figure 2.5. (See Chapter 3 for a full description of the data structures.)

Thus we see that objects can be extended in two directions. First, an object can be extended by adding new properties to the property list. These properties all apply to the basic object property at the head of the property list. The second direction of extension is hierarchical: new objects defined in terms of existing objects. In this case the highest-level object contains embedded sub-objects which may in turn contain more sub-objects. This results in a tree structure of objects. Each node in this tree, an entire object, may include its own property list. How, and whether, object properties in the tree are applied to objects at different (lower) levels of the tree, for example to create default properties, is not yet certain. The work on this aspect of DDL objects is only beginning.

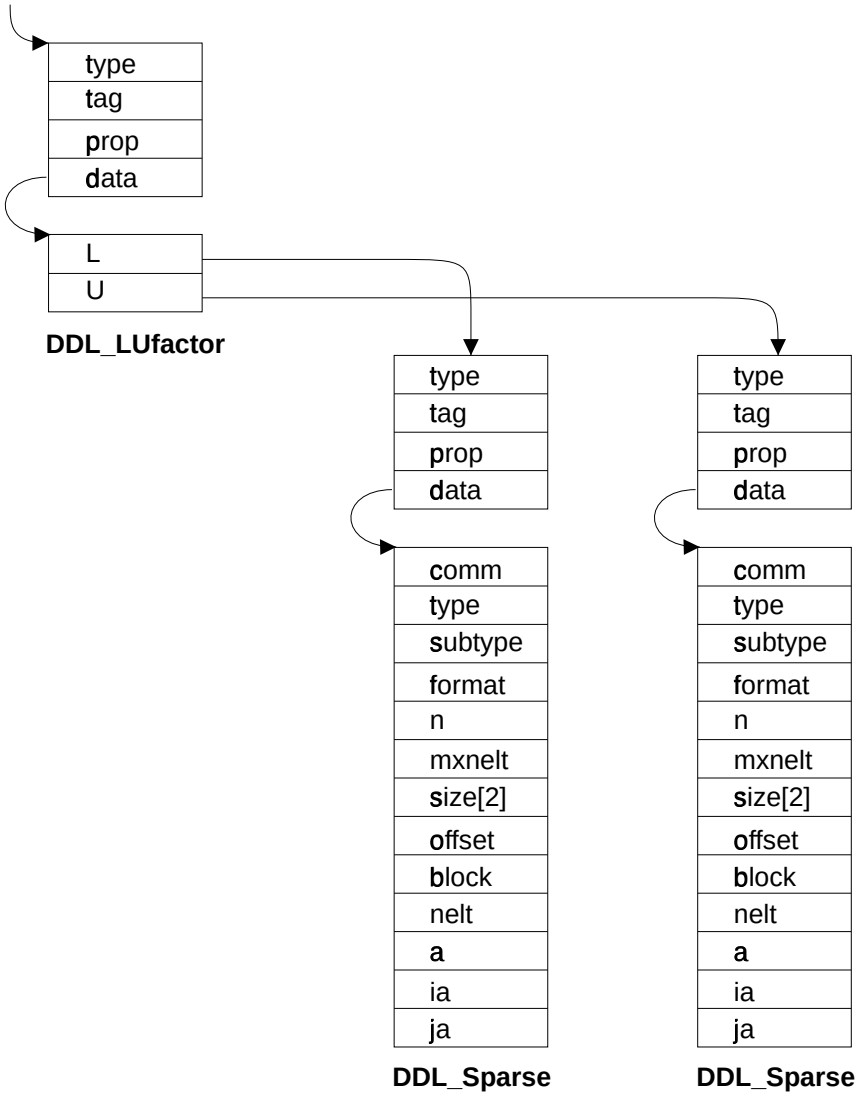

Figure 2.5: Matrix DDL object stored as LU Factors

Chapter 3

Data structures

The Sparse DDL is based on a number of basic data structures used to represent the distributed objects which the user's program manipulates. This chapter describes those data structures and shows how these structures can be used to enhance the Library.

A DDL distributed object is represented by a group of data structures which specify properties of the distributed object. The number and type of properties for any object may vary, but all objects are built from the same base set of property data structures.

3.1 The header

Every property data structure starts with a header record, DDL_Prop, which identifies the property and thus the data structure (Figure 3.1). The entries are defined as follows:

type (integer)
: The type of the property. This is used to identify the property that is passed to a Library procedure so that the appropriate action can be taken. Currently supported values are:

DDL_PROP_VECTOR	dense vector
DDL_PROP_SPARSE	sparse matrix
DDL_PROP_GLOB	communications pattern (process ordered)
DDL_PROP_IND	communications pattern (index ordered)

There is a different value for each different property data structure. Some property values represent distributed objects (such as vectors and matrices) and other values represent properties of distributed objects (such as communication patterns). New property types can be defined to extend the functionality of the Library (see Section 2.6).

tag (integer)
: The tag field is used to give a unique name to a property of an object. This is needed since an object may have several properties of the same type each representing different properties of the object. For

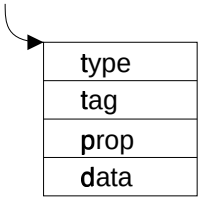

Figure 3.1: Header record

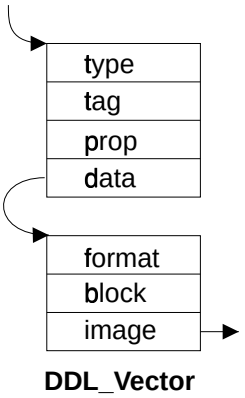

Figure 3.2: Vector data structure

example a sparse matrix may require two DDL_PROP_VECTOR properties to specify the row and column permutations of the matrix. Currently supported values are:

DDL_PX	row permutation for a matrix
DDL_PY	column permutation for a matrix
DDL_IPX	inverse row permutation for a matrix
DDL_IPY	inverse column permutation for a matrix
DDL_COMM_PX	row communication pattern for matrix-vector multiply
DDL_COMM_PY	column communication pattern for matrix-vector multiply and linear equation solver

New tags can be defined to extend the functionality of the Library.

prop (pointer)
The pointer to the next property data structure in the property list.

data (pointer)
The pointer to the type -specific record containing the actual property information.

The handle a user program has for a distributed object is a pointer to the property header record for the object.

3.2 Basic distributed objects

The current Library defines two basic distributed object types. These are a distributed vector, DDL_Vec, and a distributed sparse matrix, DDL_Spa. The Library also defines a DDL_Object type which can represent both DDL_Vec and DDL_Spa. All of these types are identical to the header DDL_Prop and may be used interchangeably. The Library uses different types to improve the readability of the code. Users are encouraged to use the types DDL_Vec and DDL_Spa. All Library procedures have parameters of type DDL_Object. Some procedures permit both vector and matrix object parameters whilst others may accept more than one object type in the future. All procedures should check the header field objecttype (with the DDL_Objecttype procedure) of an object to ensure that the object is of the correct type. This check is not currently implemented.

DDL_Vec and DDL_Spa consist of a header record DDL_Prop with the data entry pointing to a DDL_Vector and DDL_Sparse data structure respectively. These data structures are shown in Figures 3.2 and 3.3.

The distributed vector entries are defined as follows:

format (integer)
The format of the dense vector data. Not currently used.

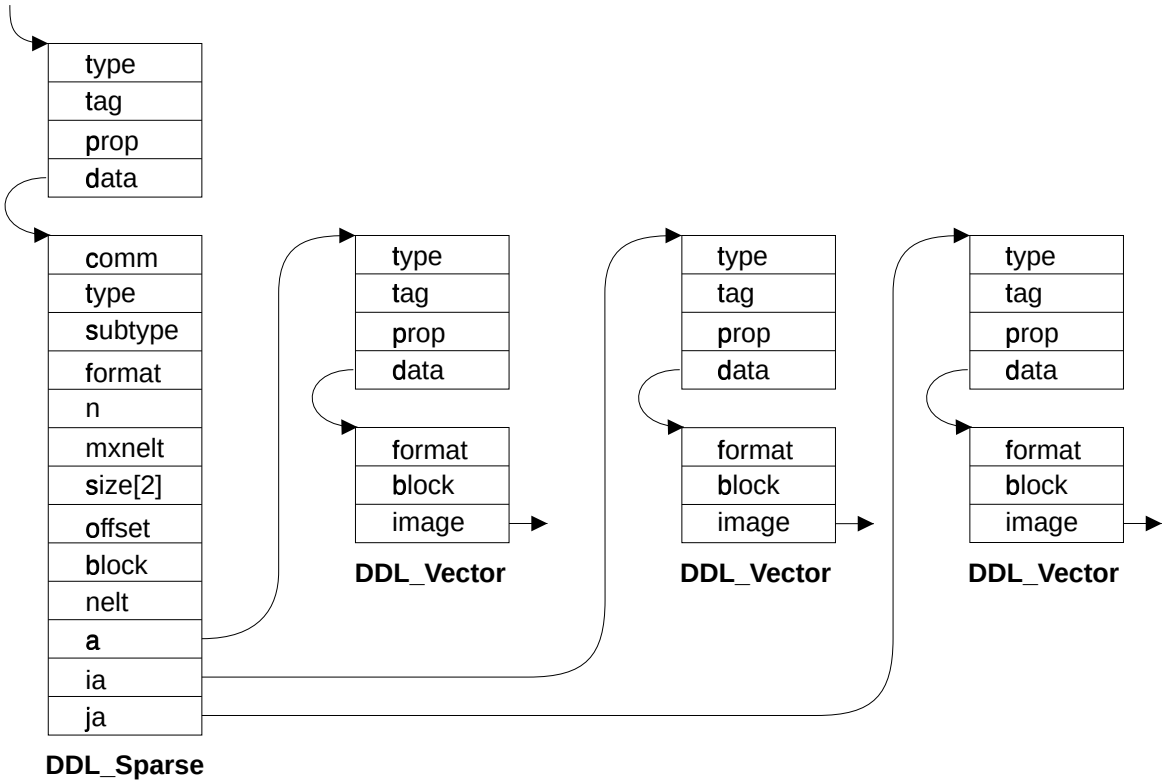

Figure 3.3: Sparse matrix data structure

block (integer)
The block size used in calculating the distribution of the vector. Elements of the vector are allocated to processes in blocks of size block.

image (pointer)
A pointer to a DDL Image object which contains the actual element data values and other information [2].

The sparse matrix distributed object is an example of how the Library can be extended by defining new object types in terms of existing object types. In this case the new sparse matrix object is defined in terms of the dense vector object. Another example of extending the Library is given in Section 2.6.

The sparse matrix entries are defined as follows:

comm (handle)
The MPI communicator for the object.

type (integer)
The data type of elements of the sparse matrix. Valid values are:

MPI_DOUBLE	double precision floating point
MPI_FLOAT	single precision floating point

subtype (integer)
The subtype of the matrix. Valid values are:

DDL_UPPER	upper triangular matrix
DDL_LOWER	lower triangular matrix

All other values are considered to be general sparse matrices.

format (integer)
: The format of the sparse matrix. This entry specifies how the sparse matrix data is stored in the three distributed vectors a, ia and ja. See Section 2.1.2 and the User Guide [1] for full details.

n (integer)
: The number of rows (columns) in the matrix.

mxnelt (integer)
: The maximum number of nonzero elements that the distributed object can hold.

size[2] (array of integer)
: size[0] gives the number of rows (columns) held on this process. size[1] gives the maximum number of nonzero elements that can be held on this process.

offset (integer)
: The number of the first row (column) held on this process.

block (integer)
: The block size used in calculating the distribution of the matrix. Rows (columns) of the matrix are allocated to processes in blocks of size block.

nelt (integer)
: The number of nonzero elements currently held on this process.

a (DDL_Vec)
: The coefficient values for the local sparse matrix elements.

ia (DDL_Vec)
: Row indices or pointers for the local sparse matrix elements.

ja (DDL_Vec)
: Column indices or pointers for the local sparse matrix elements.

For more details of the different sparse matrix formats see Section 2.1.2 and the User Guide [1].

3.3 The property list

The two basic distributed object data structures described above allow the user to define a distributed object and fill it with data values. In addition to this basic information it is often useful to be able to associate other information or properties with the distributed object. The DDL implements a property list to allow the user to add and remove optional properties to an object.

The property list is a chain of property data structures connected by the prop pointer in the property header. The Library provides procedures to add, find and remove properties from the property list.

The Library currently includes two different property structures in addition to the DDL_Vec and DDL_Spa object structures. These are DDL_Glob and DDL_Ind. These data structures are identified by the following values for the type entries in the property header: DDL_PROP_GLOB and DDL_PROP_IND. The structures are described in the following sections.

The property list allows the user to extend the functionality of the Library by defining new properties for distributed objects (see Section 2.6).

3.4 Communication properties

The DDL_Glob and DDL_Ind properties (see Figures 3.4 and 3.5) represent general communication patterns required for Library procedures. By holding these communication patterns as object properties they can be calculated once at the start of a user program and then reused each time the relevant Library procedure is invoked. This reduces the run-time of a program at the expense of more memory.

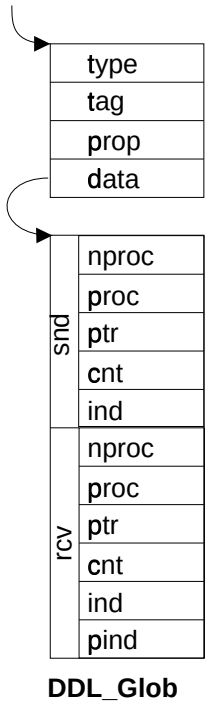

Figure 3.4: Communications data structure (process ordered)

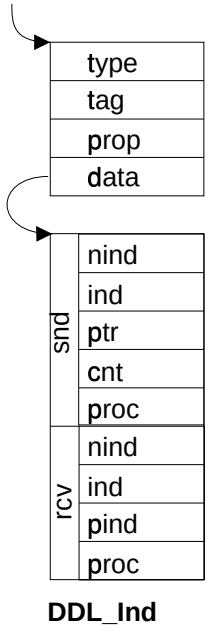

Figure 3.5: Communications data structure (index ordered)

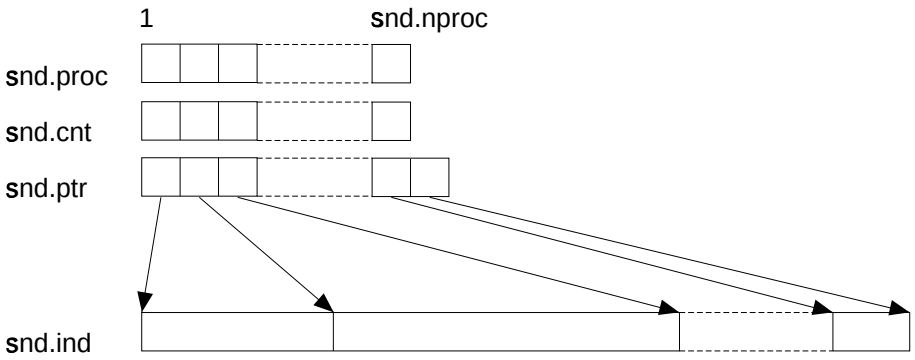

Figure 3.6: DDL_Glob send structure

The communication patterns can be described as follows. Each process occupied by an object has a number of data items (these might be elements of a vector for example). The communication pattern specifies for each process a subset of these data items which should be sent to each other process. A process can send different data subsets to different processes including an empty set. The data structure holds the indices of data items to be sent to each process.

The difference between the two property structures is the ordering of the communications. The DDL_Glob structure orders the communications by processes. Thus all the indices of data items to be sent to a process are stored contiguously in the structure. This structure is useful when all the data items for a process are available at the same time and a single message containing all these data items can be sent to the process. The DDL_Ind structure orders the communications by the data item indices. Thus the structure stores contiguously the numbers of all the processes which will be sent a data item. This structure is used when data items will only be available one at a time and so a small message containing the single item is sent to each process. This type of communication is clearly fine-grained and should be avoided where possible, but sometimes it will be required, for example in a triangular solve procedure.

Figure 3.6 illustrates the use of the send structure of DDL_Glob. The receive structure is similar except for the addition of a permuted index vector pind. The send entries are defined as follows:

snd.nproc (integer)
: The number of processes which this process sends messages to.

snd.proc (array of integer)
: A vector containing the process numbers of the processes which this process sends messages to.

snd.ptr (array of integer)
: A vector of indices into the snd.ind vector. snd.ptr[i] is the index of the first element in snd.ind that is to be sent to process snd.proc[i].

snd.cnt (array of integer)
: A vector giving the number of elements to be sent to each process. snd.cnt[i] is the number of elements in snd.ind to be sent to process snd.proc[i].

snd.ind (array of integer)
: A vector of indices of elements to be sent to processes. The indices are first ordered by process number and then the indices for each process are arranged in increasing order.

The receive entries have similar interpretations to the send entries, with the following addition:

rcv.pind (array of integer)
: A vector of logical permuted indices of elements to be received from other processes. The indices are ordered by process number, but unordered within each process. This vector is required by some procedures which need to know both the physical index of an item, given in rcv.ind, and the logical

index of the item after permutations are applied, given here. rcv.ind[i] is the physical index of the element with logical permuted index rcv.pind[i]. See Section 2.4 for more details about permutations.

The index ordered communication structure, DDL_Ind, is similar to the process ordered structure, DDL_Glob. The send entries are defined as follows:

snd.nind (integer)
The number of indices which this process must send.

snd.ind (array of integer)
A vector containing the indices which this process must send.

snd.ptr (array of integer)
A vector of indices into the snd.proc vector. snd.ptr[i] is the index of the first process number in snd.proc which is to be sent the index snd.ind[i].

snd.cnt (array of integer)
A vector giving the number of processes each index is to be sent to. snd.cnt[i] is the number of processes in snd.proc to be sent index snd.ind[i].

snd.proc (array of integer)
A vector of process numbers to be sent each index. The process numbers are first ordered by index number and then the processes for each index are arranged in increasing order.

The receive entries are different from the send entries:

rcv.nind (integer)
The number of indices which this process must receive.

rcv.ind (array of integer)
A vector containing the indices which this process must receive arranged in ascending order.

rcv.pind (array of integer)
A vector of logical permuted indices of elements to be received from other processes. The indices are unordered. This vector is required by some procedures which need to know both the physical index of an item, given in rcv.ind, and the logical index of the item after permutations are applied, given here. rcv.ind[i] is the physical index of the element with logical permuted index rcv.pind[i]. See Section 2.4 for more details about permutations.

rcv.proc (array of integer)
A vector of process numbers indicating where each index is to be received from. Index rcv.ind[i] is to be received from process rcv.proc[i].

The two communication property structures described above can be used for general purpose communication patterns. Specific instances of these structures are used to hold the communications patterns required by different Library procedures. To distinguish between the different instances, the header tag entry is used. Currently three specific instances are defined by tags:

DDL_COMM_PX	row communication pattern; matrix-vector multiply and triangular solve
DDL_COMM_PY	column communication pattern; matrix-vector multiply
DDL_COMM_IND	column communication pattern; triangular solve

Appendix A describes more fully the use of communication properties in the matrix-vector multiply procedures.

3.5 Permutation property

A permutation property can be associated with each dimension of any distributed object. Currently only the sparse matrix object type makes use of permutation properties. Different instances of a permutation are, like other property types, identified by different tag values. The sparse matrix object may have a total of four different permutations: a row permutation (and its inverse) and a column permutation (and its inverse).

DDL_PX	row permutation
DDL_PY	column permutation
DDL_IPX	inverse row permutation
DDL_IPY	inverse column permutation

A permutation is represented by a distributed vector object, DDL_Vec, the same size as the dimension of the object to be permuted. Each entry in the permutation vector specifies the logical index of the associated physical index of the object. See Section 2.4 for a description of permutations.

3.6 Example sparse matrix data structure

Figure 3.7 shows an example data structure for a sparse matrix distributed object. As well as the basic sparse matrix data structure, consisting of three distributed vectors, the structure also includes a property list. This property list has process-ordered communications property, an index-ordered communications property, and a permutation property.

Another example of a typical sparse matrix structure might be that of the matrix used in the tfqm-rnc solver example program provided with the Sparse DDL. This matrix structure includes a DDL_Glob communications property for use in the matrix-vector multiply and triangular solve procedures, a DDL_Ind communications property for the triangular solve, and up to three DDL_Vec permutation properties for a row, column and inverse row permutation.

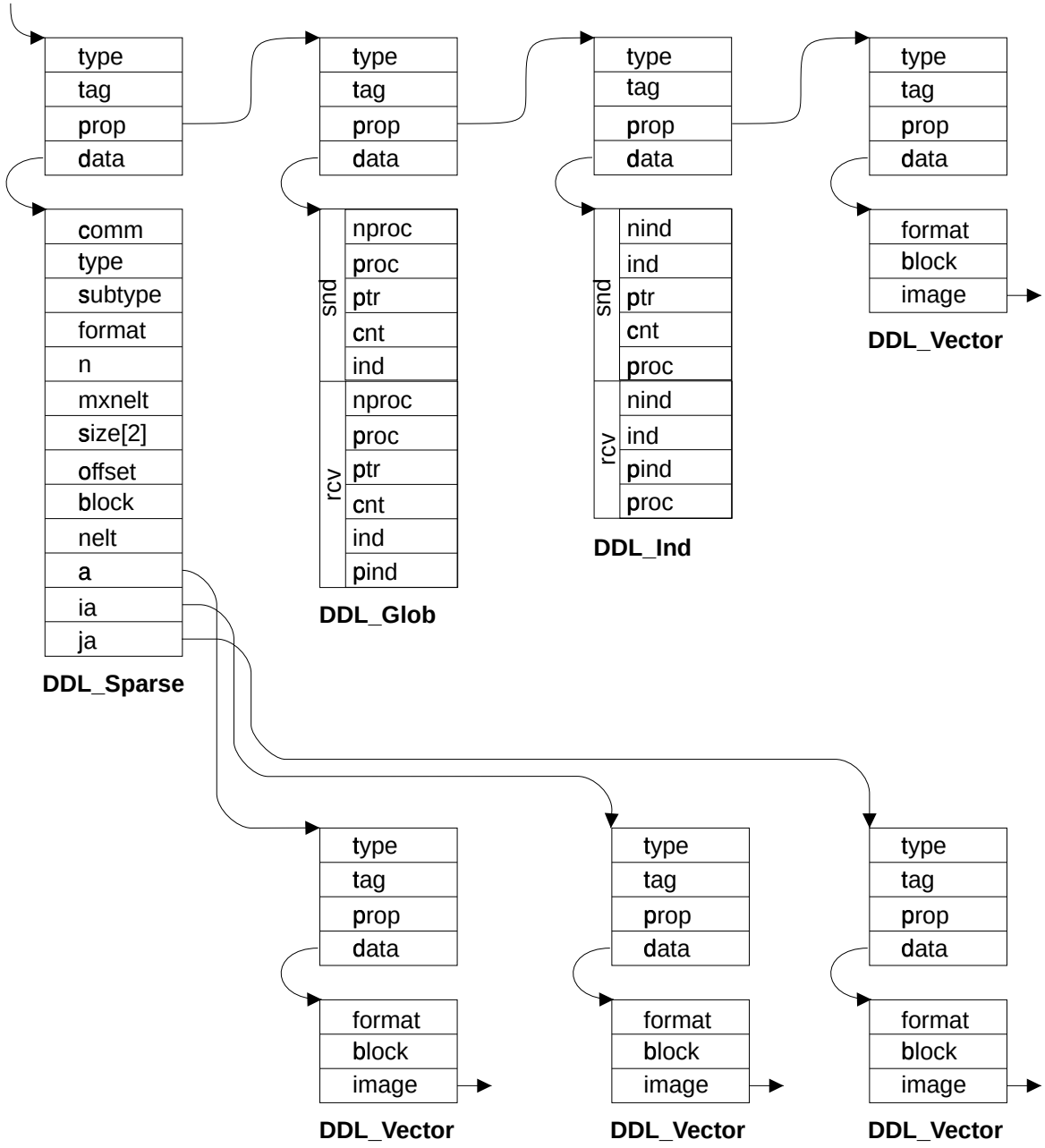

Figure 3.7: Structure of a typical sparse matrix

Chapter 4

The future of the DDL

Work on the design of a distributed data library began at Liverpool during the Esprit Supernode I project in 1988. Back then, the work was based around OCCAM and transputers. In the following years, the library work progressed through a number of stages. The implementation language has changed from OCCAM, to FORTRAN and OCCAM, to FORTRAN and C. The communications has moved from OCCAM channels to PVM and then to MPI. The current implementation, written mainly in C with high-level FORTRAN procedures and using the MPI system, is portable and uses mainstream languages, and provides a good foundation for development over the next few years.

The development of parallel iterative solvers using the DDL, and their incorporation into an oil reservoir simulation package, within the PARSIM project, has demonstrated that the basic Library design is usable and flexible. In order to develop the Library further it is essential to get other application developers using the Library so that the strengths and weaknesses of the Library can be revealed to direct future development work.

At this stage there are already a number of areas where future work is desirable:

- performance testing on a range of current parallel machines,
- adding support for dense matrices,
- providing full support for more sparse matrix formats, as described in the User Guide [1],
- investigating new high-level objects, such as the LU factored matrix,
- investigating new object properties,
- increasing the range of high-level operations supported,
- investigating integration of the Library with HPF.

Appendix A

Example DDL procedure: sparse matrix-vector multiply

This chapter describes the sparse matrix dense vector multiply procedures from the DDL as an example of the way in which DDL procedures may be written. These procedures are listed in full in Section A.2. The next section comments on the matrix vector code, describing the code structure and highlighting important features. The code makes use of many DDL procedures. Some of these procedures are frequently used in user-level programs and are described in the DDL User's Guide [1]. Other low-level procedures have no documentation other than the comments in the DDL source code for the procedure. A Library writer will therefore need to be reasonably familiar with the source for the existing procedures, which provide much of the functionality required by a general Library procedure.

A.1 Description

The sparse matrix dense vector multiply may be written as

$$y = \alpha Ax + \beta y$$

where A is the square, sparse matrix; x is the initial dense vector; y is the result vector; and α and β are constants. These three distributed objects must all be distributed over the same processes, using the same MPI communicator.

The sparse matrix-vector multiply procedures assume that the matrix is distributed by rows, and allow the rows and/or columns to be permuted. The procedures use the property list both for the permutation properties DDL_PX and DDL_PY, and for the matrix-vector multiply communication pattern properties. The communication patterns DDL_COMM_PX and DDL_COMM_PY are calculated by the procedure DDL_Smv_comm (lines 5–148), whilst the actual matrix vector multiply is performed by DDL_Smv (lines 151–355). For a given sparse matrix, DDL_Smv_comm only needs to be called once by a user program, then DDL_Smv can be called any number of times provided the structure of the matrix is not changed.

The property list procedures are: DDL_Create_prop to add a property to the list; DDL_Find_prop to find an existing property in the list; and DDL_Free_prop to remove a property in the list.

A.1.1 DDL_Smv_comm

This procedure calculates the communications properties for the sparse matrix, and only needs to be called once provided the structure of the matrix is not changed.

The procedure starts with some rudimentary error checking (lines 17–26). Lines 28–38 extract important information about the matrix using procedures described in the User Guide [1].

Lines 40–91 calculate the column communication pattern property DDL_COMM_PY. This pattern represents the elements of the initial dense vector x that must be communicated between processes for the

matrix vector multiply. If a process has a nonzero matrix element in column i then that process needs element i of the vector x. DDL_COMM_PY is a data structure that holds the information about which processes need which elements of x and which processes these elements are held on. This information is held in the process-ordered data structure DDL_Glob. See Sections 2.5 and 3.4 for further details about communications properties.

First the procedure checks if the property already exists (line 41). If the property does not exist then it is created (line 48) and space for the communications structure is allocated (lines 49–57). If the property does already exist then the procedure will recalculate the communications property assuming that the user has changed the matrix structure. Some of the allocated arrays in the communications structure are potentially very large and to save memory they are truncated to the minimum size necessary once the structure has been calculated. Therefore, before recalculating the communication pattern, the procedure reallocates these arrays to full size (lines 60–65).

If the matrix has a column permutation (checked in line 70) then the procedure gets a pointer to the permutation vector data (line 73) and allocates space in the communications property structure for the permuted index vector (lines 74–76). The procedure DDL_Col is then called to calculate the column communication pattern and store it in the allocated arrays (lines 78–86). The arrays are then truncated in lines 88–91.

Lines 93–146 calculate the possible row communication pattern property DDL_COMM_PX. This pattern represents the elements of the result vector y that must be communicated between processes for the matrix vector multiply. This property is only required if the matrix has a row permutation. For the matrix vector multiply, if a process holds logical row i of the sparse matrix then that process will calculate element i of the result vector y. Before the multiply starts these elements of y must be fetched from their owner processes and when the multiply is complete, the updated elements of y must be returned to their owners. DDL_COMM_PX is a data structure that holds the information about which processes will calculate which elements of y and which processes these elements must be returned to. This information is again held in the process-ordered data structure DDL_Glob.

Lines 95–96 check for a row permutation. If none exists then nothing needs to be done. If there is a permutation then the communications structure is calculated using code very similar to that already described for the column permutation.

A.1.2 DDL_Smv

This procedure performs the actual matrix vector multiply.

Lines 177–209 perform checks on the parameters to ensure they are compatible. These checks include checking that all objects have the same datatype (lines 189–191); the objects have compatible dimensions (lines 192–194); the objects have the same MPI communicator (lines 197–204); and the matrix is distributed by rows (lines 206–209).

Lines 211–222 extract important information about the objects.

Lines 224–228 check whether the matrix-vector multiply communications properties have been calculated already. If they have not, then they are calculated here.

Lines 230–245 get pointers to the row and column communications structures DDL_Glob.

The matrix vector multiply result elements are stored in a temporary vector on each process (line 248).

Lines 250–275 fetch the required initial elements of the result vector to each process if there is a row permutation. This makes use of the row communication data structure DDL_COMM_PX calculated by DDL_Smv_comm. Firstly, two temporary buffers are allocated on each process(lines 252–256): one buffer yrcv_buf will hold the initial values of the result vector sent from other processes; while the other buffer ysnd_buf will hold the elements of the vector that this process must send to other processes. Lines 258–260 pack the vector elements into the send buffer as directed by the communications data structure. Lines 261–170 then perform a collective all-to-all communication operation DDL_Alltoallvs using these data buffers and the information in the communication data structure. After the communication operation the initial result vector elements in the receive buffer are unpacked into the temporary result vector (lines 272–274).

If there is no row permutation then lines 277–279 copy the initial result values from the distributed result vector y to the temporary result vector.

Lines 282–307 fetch elements of the x vector using the column communications data structure. The code is very similar to the code which fetches the initial result values.

A local sparse matrix-vector multiply is performed in lines 309–313 using the local sparse matrix, the temporary initial y values and the temporary x values.

Lines 315–342 update the distributed y vector with the result values. If there was a row permutation then lines 315–337 return the calculated result elements to their owner processes using the row communication data structure and the collective communication DDL_Alltoallvs. Otherwise, the procedure just copies the new result values into the result vector object (lines 338–342).

The remaining lines 344–354 just tidy up by freeing the temporary arrays allocated by the procedure.

A.2 Listing

```
 1  /* sparse matrix vector multiply with permutations */
 2  /* assumed row packed format */
 3
 4  /* calculate communication pattern for sparse matrix-vector multiply */
 5  int DDL_Smv_comm(DDL_Object a)
 6  {
 7    int ierr = DDL_SUCCESS;
 8    int nproc, proc, dummy, offset, ret;
 9    int rcv_tot_data, snd_tot_data;
10    int *ia, *ja, *py, *px, *tmp;
11    int nr, nc, n, size[3], i, j, low, high, subtype;
12    int format, block;
13    MPI_Comm comm;
14    DDL_Prop p;
15    DDL_Glob g;
16
17    /* check parameters */
18    if (a == NULL) {
19      ierr = DDL_Error(DDL_ERR_UNALLOCATED, DDL_ERR_HARD, DDL_SMV_COMM);
20      if (ierr > 0) return ierr; /* cleanup and exit with error */
21    }
22    /* only packed rows supported at the moment */
23    ierr = DDL_Format_sparse(a, &format);
24    if (format != DDL_PCK_ROW) {
25      ierr = DDL_Error(DDL_ERR_IFORMAT, DDL_ERR_HARD, DDL_SMV_COMM);
26      return ierr;}
27
28    ierr = DDL_Get_ia(a, &ia);
29    ierr = DDL_Get_ja(a, &ja);
30    ierr = DDL_Gsize_sparse(a, size);
31    nr = size[0]; /* total number of rows */
32    nc = size[1]; /* total number of columns */
33    ierr = DDL_Size_sparse(a, size);
34    n = size[0]; /* number of rows I have */
35    ierr = DDL_Block_sparse(a, &block);
36    ierr = DDL_Comm(a, &comm);
37    ierr = MPI_Comm_size(comm, &nproc);
38    ierr = MPI_Comm_rank(comm, &proc);
39
40    /* create COMM_PY global comms data structure for matrix */
41    ierr = DDL_Find_prop(a, DDL_COMM_PY, &p);
42    if (ierr != DDL_SUCCESS) {
43      ierr = DDL_Error(ierr, DDL_ERR_HARD, DDL_SMV_COMM);
44      if (ierr > 0) return ierr; /* cleanup and exit with error */
45    }
```

```
 46    if (p == NULL) {
 47      /* create communication structure */
 48      ierr = DDL_Create_prop(a, DDL_COMM_PY, &p);
 49      g = (p->data);
 50      g->rcv.proc = malloc(nproc*sizeof(int));
 51      g->rcv.ind = malloc(nc*sizeof(int));
 52      g->rcv.cnt = malloc(nproc*sizeof(int));
 53      g->rcv.ptr = malloc((nproc+1)*sizeof(int));
 54      g->snd.proc = malloc(nproc*sizeof(int));
 55      g->snd.ind = malloc(n*nproc*sizeof(int));
 56      g->snd.cnt = malloc(nproc*sizeof(int));
 57      g->snd.ptr = malloc((nproc+1)*sizeof(int));
 58    }
 59    else {
 60      /* structure already exists-just make ind vectors full size */
 61      g = (p->data);
 62      free(g->rcv.ind);
 63      free(g->snd.ind);
 64      g->rcv.ind = malloc(nc*sizeof(int));
 65      g->snd.ind = malloc(n*nproc*sizeof(int));
 66    }
 67
 68    /* calculate column communications
 69       ie which elements of result vector x I need */
 70    ierr = DDL_Find_prop(a, DDL_PY, &p);
 71    py = NULL;
 72    if (p != NULL) {
 73      ierr = DDL_Get_vector(p, &py);
 74      /* allocate space for permuted receive index vector */
 75      free(g->rcv.pind); /* in case there was an old vector */
 76      g->rcv.pind = malloc(nc*sizeof(int));
 77    }
 78    ierr = DDL_Col(ia, ja, n, nc, block, py,
 79   proc, nproc, comm,
 80   &(g->rcv.nproc), &(g->snd.nproc),
 81   &rcv_tot_data, &snd_tot_data,
 82   g->rcv.proc, g->snd.proc,
 83   g->rcv.ptr, g->snd.ptr,
 84   g->rcv.cnt, g->snd.cnt,
 85   g->rcv.ind, g->snd.ind,
 86   g->rcv.pind);
 87
 88    /* now truncate rcv.ind, and snd.ind since these are *BIG* */
 89    ierr = DDL_Truncate(&(g->rcv.ind), rcv_tot_data);
 90    ierr = DDL_Truncate(&(g->snd.ind), snd_tot_data);
 91    if (py != NULL) ierr = DDL_Truncate(&(g->rcv.pind), rcv_tot_data);
 92
 93    /* look at row swapping;
 94       if no row permutation vector px then skip this */
 95    ierr = DDL_Find_prop(a, DDL_PX, &p);
 96    if (p != NULL) {
 97      ierr = DDL_Get_vector(p, &px);
 98      /* need COMM_PX row data structure for elements of RHS vector b */
 99      /* create COMM_PX comms data structure for matrix */
100      ierr = DDL_Find_prop(a, DDL_COMM_PX, &p);
101      if (ierr != DDL_SUCCESS) {
102        ierr = DDL_Error(ierr, DDL_ERR_HARD, DDL_SMV_COMM);
103        if (ierr > 0) return ierr; /* cleanup and exit with error */
```

```
104      }
105      if (p == NULL) {
106        /* create communication structure */
107        ierr = DDL_Create_prop(a, DDL_COMM_PX, &p);
108        g = (p->data);
109        g->rcv.proc = malloc(nproc*sizeof(int));
110        g->rcv.ind = malloc(n*sizeof(int));
111        g->rcv.pind = malloc(n*sizeof(int));
112        g->rcv.cnt = malloc(nproc*sizeof(int));
113        g->rcv.ptr = malloc((nproc+1)*sizeof(int));
114        g->snd.proc = malloc(nproc*sizeof(int));
115        g->snd.ind = malloc(n*nproc*sizeof(int));
116        g->snd.cnt = malloc(nproc*sizeof(int));
117        g->snd.ptr = malloc((nproc+1)*sizeof(int));
118      }
119      else {
120        /* structure already exists-just make ind vectors full size */
121        g = (p->data);
122        free(g->rcv.ind);
123        free(g->rcv.pind);
124        free(g->snd.ind);
125        g->rcv.ind = malloc(n*sizeof(int));
126        g->rcv.pind = malloc(n*sizeof(int));
127        g->snd.ind = malloc(n*nproc*sizeof(int));
128      }
129
130      /* calculate row communications
131         ie which elements of RHS vector b I need */
132      ierr = DDL_Row(ia, ja, n, nr, block, px,
133     proc, nproc, comm,
134     &(g->rcv.nproc), &(g->snd.nproc),
135     &rcv_tot_data, &snd_tot_data,
136     g->rcv.proc, g->snd.proc,
137     g->rcv.ptr, g->snd.ptr,
138     g->rcv.cnt, g->snd.cnt,
139     g->rcv.ind, g->snd.ind,
140     g->rcv.pind);
141
142      /* now truncate rcv.ind and snd.ind since these are *BIG* */
143      ierr = DDL_Truncate(&(g->rcv.ind), rcv_tot_data);
144      ierr = DDL_Truncate(&(g->snd.ind), snd_tot_data);
145      ierr = DDL_Truncate(&(g->rcv.pind), rcv_tot_data);
146    }
147    return ierr;
148  }
149
150  /* parallel sparse matrix, dense vector multiply */
151  int DDL_Smv(DDL_Object sp, DDL_Object x, DDL_Object y,
152      double alpha, double beta)
153  {
154    int ierr, i, j, low, high, next, ret;
155    int x_size, x_itype, x_offset, y_offset;
156    MPI_Comm x_comm;
157    MPI_Datatype x_type;
158    void *x_data;
159    int y_size, y_itype;
160    MPI_Comm y_comm;
161    MPI_Datatype y_type;
```

```
162    void *y_data;
163    int sp_size[3], sp_itype;
164    MPI_Comm sp_comm;
165    MPI_Datatype sp_type;
166    int *ia, *ja;
167    void *a, *tmp, *res, *ysnd_buf, *yrcv_buf, *snd_buf, *rcv_buf;
168    int *snd_cnt, *rcv_cnt;
169    int format, size;
170    MPI_Aint typesize;
171    int n, m, nelt;
172    DDL_Prop p, p1;
173    DDL_Glob gx, gy;
174
175    ierr = DDL_SUCCESS;
176
177    /* check input parameters */
178    ierr = DDL_Gsize_vector(x, &x_size);
179    ierr = DDL_Gsize_vector(y, &y_size);
180    ierr = DDL_Gsize_sparse(sp, sp_size);
181    m = sp_size[1];
182    ierr = DDL_Type_vector(x, &x_type);
183    ierr = DDL_Type_vector(y, &y_type);
184    ierr = DDL_Type_sparse(sp, &sp_type);
185    ierr = DDL_Comm_vector(x, &x_comm);
186    ierr = DDL_Comm_vector(y, &y_comm);
187    ierr = DDL_Comm_sparse(sp, &sp_comm);
188
189    if ((x_type != y_type) || (x_type != sp_type)) {
190      ierr = DDL_Error(DDL_ERR_ITYPE, DDL_ERR_HARD, DDL_SMV);
191      return ierr;}
192    if ((x_size != sp_size[1]) || (y_size != sp_size[0])) {
193      ierr = DDL_Error(DDL_ERR_ISIZE, DDL_ERR_HARD, DDL_SMV);
194      return ierr;}
195    /* check communicators are **identical**
196       same number of processes and ranking of processes */
197    ierr = MPI_Comm_compare(x_comm, y_comm, &ret);
198    if (ret != MPI_IDENT) {
199      ierr = DDL_Error(DDL_ERR_ICOMM, DDL_ERR_HARD, DDL_SMV);
200      return ierr;}
201    ierr = MPI_Comm_compare(x_comm, sp_comm, &ret);
202    if (ret != MPI_IDENT) {
203      ierr = DDL_Error(DDL_ERR_ICOMM, DDL_ERR_HARD, DDL_SMV);
204      return ierr;}
205
206    ierr = DDL_Format_sparse(sp, &format);
207    if (format != DDL_PCK_ROW) {
208      ierr = DDL_Error(DDL_ERR_IFORMAT, DDL_ERR_HARD, DDL_SMV);
209      return ierr;}
210
211    /* get pointers to raw data values */
212    ierr = DDL_Get_vector(x, &x_data);
213    ierr = DDL_Get_vector(y, &y_data);
214    ierr = DDL_Get_ia(sp, &ia);
215    ierr = DDL_Get_ja(sp, &ja);
216    ierr = DDL_Get_a(sp, &a);
217    ierr = DDL_Size_vector(x, &x_size);
218    ierr = DDL_Offset_vector(x, &x_offset);
219    ierr = DDL_Offset_vector(y, &y_offset);
```

```
220    ierr = DDL_Size_sparse(sp, sp_size);
221    n = sp_size[0];
222    ierr = MPI_Type_size(x_type, &typesize);
223
224    /* check global comms. data exists, else create */
225    ierr = DDL_Find_prop(sp, DDL_COMM_PY, &p);
226    ierr = DDL_Find_prop(sp, DDL_COMM_PX, &p1);
227    if (p == NULL && p1 == NULL)
228      ierr = DDL_Smv_comm(sp);
229
230    /* get global comms strcutures */
231    ierr = DDL_Find_prop(sp, DDL_COMM_PY, &p);
232    if (ierr != DDL_SUCCESS) {
233      ierr = DDL_Error(ierr, DDL_ERR_HARD, DDL_SMV);
234      if (ierr > 0) return ierr; /* cleanup and exit with error */
235    }
236    gy = (DDL_Glob)(p->data);
237    ierr = DDL_Find_prop(sp, DDL_COMM_PX, &p);
238    if (ierr != DDL_SUCCESS) {
239      ierr = DDL_Error(ierr, DDL_ERR_HARD, DDL_SMV);
240      if (ierr > 0) return ierr; /* cleanup and exit with error */
241    }
242    if (p != NULL)
243      gx = (DDL_Glob)(p->data);
244    else
245      gx = NULL;
246
247    /* temporary buffer for result vector */
248    res = malloc(n*typesize);
249
250    /* get elements of result y I need for permutation */
251    if (gx != NULL) {
252      /* allocate temporary buffers */
253      size = gx->rcv.ptr[gx->rcv.nproc];
254      yrcv_buf = malloc(size*typesize);
255      size = gx->snd.ptr[gx->snd.nproc];
256      ysnd_buf = malloc(size*typesize);
257
258      /* pack ysnd_buf */
259      for (i=0;i<gx->snd.ptr[gx->snd.nproc];i++)
260        ((double*)ysnd_buf)[i] = ((double*)y_data)[gx->snd.ind[i]];
261      /* do alltoall */
262      ierr = DDL_Alltoallvs(ysnd_buf, gx->snd.nproc,
263    gx->snd.proc,
264    gx->snd.cnt,
265    gx->snd.ptr, x_type,
266    yrcv_buf, gx->rcv.nproc,
267    gx->rcv.proc,
268    gx->rcv.cnt,
269    gx->rcv.ptr, x_type,
270    sp_comm);
271
272      /* unpack yrcv_buf to res */
273      for (i=0;i<gx->rcv.ptr[gx->rcv.nproc];i++)
274        ((double*)res)[gx->rcv.ind[i]-y_offset] = ((double*)yrcv_buf)[i];
275    }
276    else {
277      /* gx == NULL just copy my elements of y */
```

```
278       for (i=0;i<n;i++)
279         ((double*)res)[i] = ((double*)y_data)[i];
280     }
281
282     /* now get elements of x I need */
283     /* allocate temporary buffers */
284     size = gy->rcv.ptr[gy->rcv.nproc];
285     rcv_buf = malloc(size*typesize);
286     size = gy->snd.ptr[gy->snd.nproc];
287     snd_buf = malloc(size*typesize);
288     tmp = malloc(m*typesize);
289
290     /* pack snd_buf */
291     for (i=0;i<gy->snd.ptr[gy->snd.nproc];i++)
292       ((double*)snd_buf)[i] = ((double*)x_data)[gy->snd.ind[i]];
293
294     /* do alltoall */
295     ierr = DDL_Alltoallvs(snd_buf, gy->snd.nproc,
296   gy->snd.proc,
297   gy->snd.cnt,
298   gy->snd.ptr, x_type,
299   rcv_buf, gy->rcv.nproc,
300   gy->rcv.proc,
301   gy->rcv.cnt,
302   gy->rcv.ptr, x_type,
303   sp_comm);
304
305     /* unpack rcv_buf to tmp */
306     for (i=0;i<gy->rcv.ptr[gy->rcv.nproc];i++)
307       ((double*)tmp)[gy->rcv.ind[i]] = ((double*)rcv_buf)[i];
308
309     /* do matrix-vector multiply */
310     ierr = DDL_Smv_seq(n, m, ia, ja,
311        a, sp_type,
312        nelt, format, tmp, res,
313        alpha, beta);
314
315     /* send back elements of result vector if permutation */
316     /* opposite to fetching y values at start:
317        use rcv.ind to pack sending vector etc */
318     if (gx != NULL) {
319       /* pack yrcv_buf */
320       for (i=0;i<gx->rcv.ptr[gx->rcv.nproc];i++)
321         ((double*)yrcv_buf)[i] = ((double*)res)[gx->rcv.ind[i]-y_offset];
322
323       /* do alltoall sending rcv structure */
324       ierr = DDL_Alltoallvs(yrcv_buf, gx->rcv.nproc,
325     gx->rcv.proc,
326     gx->rcv.cnt,
327     gx->rcv.ptr, x_type,
328     ysnd_buf, gx->snd.nproc,
329     gx->snd.proc,
330     gx->snd.cnt,
331     gx->snd.ptr, x_type,
332     sp_comm);
333
334       /* unpack ysnd_buf to y_data */
335       for (i=0;i<gx->snd.ptr[gx->snd.nproc];i++)
```

```
336         ((double*)y_data)[gx->snd.ind[i]] = ((double*)ysnd_buf)[i];
337     }
338     else {
339       /* gx == NULL just copy my result elements to y */
340       for (i=0;i<n;i++)
341         ((double*)y_data)[i] = ((double*)res)[i];
342     }
343 
344     /* free temporary arrays */
345     if (gx != NULL) {
346       free(ysnd_buf);
347       free(yrcv_buf);
348     }
349     free(snd_buf);
350     free(rcv_buf);
351     free(res);
352     free(tmp);
353 
354     return ierr;
355   }
```

Bibliography

[1] Tim Oliver. *Sparse DDL version 2.1: User's Guide*. Institute for Advanced Scientific Computation, University of Liverpool, July 1995.

[2] Bruce Stevens, Cliff Addison, and Rod Cook. *A users' guide to the DDL: Distributed Data Library*. Institute for Advanced Scientific Computation, University of Liverpool, April 1994.