Sparse DDL version 2.1
User Guide

Tim Oliver
Institute for Advanced Scientific Computation
University of Liverpool

Thursday 17th August 1995

Contents

1 Introduction to the DDL **3**

2 DDL terms and conventions **4**
2.1 Distributed objects 4
2.2 Constants 4
2.3 Procedure specification 4
2.4 Language binding 5
2.5 Error handling 5

3 Distributed objects **6**
3.1 Distribution 6
3.2 Permutations 7
3.3 Object properties 7
3.4 Dense vectors 8
3.5 Sparse matrices 8
3.5.1 Single process sparse matrices 8
3.5.2 Distributed sparse matrices 9

4 Object management **12**
4.1 Object creation and deletion 12
4.2 Property query procedures 13
4.3 Property setting procedures 16
4.4 Copying and transforming objects 17

5 Level 1 BLAS **18**

6 Sparse matrix operations **20**
6.1 Sparse matrix vector multiply 20
6.2 Sparse triangular solve 21
6.3 Conversion between sparse formats 21

7 File I/O **24**
7.1 Parallel I/O 24
7.2 Host I/O 26

8 Direct access to object data **30**
8.1 Access procedures 30
8.2 Access methods 31
8.2.1 C 31
8.2.2 F77 32
8.2.3 F90 34

9 High-level operations **36**
9.1 Preconditioning for linear equation solvers 36
9.1.1 ILU preconditioning 36
9.1.2 ILUnc preconditioning 37

A Example DDL program **38**
A.1 Description of the example program 38
A.2 Listing 38

B Error handling **53**
B.1 The error mechanism 53
B.2 User error handlers 54

Chapter 1

Introduction to the DDL

This document is the User Guide to Version 2.1 of the DDL library. The specification of the Library is still under review and the implementation is far from complete. In particular, only the DDL_PCK_ROW **sparse distributed data format is fully implemented and there is little error checking. You have been warned!**

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

The Distributed Data Library (DDL) is a library system that permits a user to exploit the power of a distributed memory parallel computer from a single-threaded Fortran program. This data parallel programming model is achieved by letting the user call library routines which create, manage and operate on distributed objects such as matrices and vectors.

This document describes Version 2.1 of the Library. In this document we describe the DDL from a User's perspective. Programmers who are interested in developing their own DDL procedures or delving deeper into the design of the DDL should also read the DDL Technical Guide [2].

The document is arranged as follows. The next chapter gives a summary of the conventions used in the document. The following two chapters describe the distributed objects supported by the Library (Chapter 3) and the basic procedures for object management (Chapter 4) such as creation and deletion. Chapter 5 defines the Level 1 BLAS operations that have been implemented and Chapter 6 defines the matrix operations currently supported. File I/O procedures are defined in Chapter 7. Low-level access to the underlying data structures is discussed in Chapter 8. Finally, Chapter 9 describes the high level preconditioners for sparse linear equation solvers built on top of the base Library.

The Appendices include a complete example program (Appendix A). Appendix B contains information about error handling in the Library.

A bibliography and an index to the DDL procedure descriptions are given at the end of this document.

Chapter 2

DDL terms and conventions

This chapter explains the terms and conventions used by the DDL. Many of these are derived from the MPI standard [1].

2.1 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 supports several different types of distributed objects and allows new types to be added. In C each object type has a different handle: dense vectors are of type DDL_Vec, and sparse matrices are of type constDDL_Spa. The generic type DDL_Object is used to refer to an object of either type. In FORTRAN all handles are of type INTEGER.

The DDL manages system memory, allocating space for new objects and deallocating unwanted objects. When a new object is created the creation procedure will return a handle for the new object. When that object is deallocated the handle becomes invalid.

A user will often want to access the raw data values of a distributed object. In this case the DDL includes procedures that return a pointer to the part of an object's raw data that is allocated to the calling process. It is the user's responsibility to ensure that all changes to the raw data are consistent with the DDL specification.

2.2 Constants

The DDL makes considerable use of special values of the type INTEGER. These values are used to represent such things as error conditions, data types, and distributed object formats. These values are defined in header files which should be included in each user program. For FORTRAN use:

```
include "ddlf.h"
```

and for C use:

```
#include "ddl.h"
```

2.3 Procedure specification

This document uses a language independent notation to specify each DDL procedure.

The usage of each procedure argument is specified as one of IN, OUT or INOUT with the following meanings:

IN	The argument is used by the procedure but not changed.
OUT	The argument is not used by the procedure but may be changed.
INOUT	The argument is both used and changed by the procedure.

A special interpretation is used for object handles. If an argument is a handle on an object, and the object is used but not changed by the procedure, the argument is marked IN. If the object is changed but not used, the argument is marked OUT. Finally, if the object is both used and changed by the procedure then it is marked INOUT.

A short description of each argument is also given followed by the type of the argument. The argument type will be one of integer, float, double, logical, handle, choice or array. The handle type represents handles for any object type or DDL types and constants. The choice type represents an argument that can have one of a number of different types, such as different floating point precisions.

The language independent description is followed by language specific bindings for C and FORTRAN. These bindings specify the exact language types for each argument. The choice arguments are marked `void *` in C and <`type`> in FORTRAN.

2.4 Language binding

In C all procedures have names beginning with MPI_, and case is significant. Almost every procedure returns an integer indicating the error condition of the procedure on exit. DDL constants and types also have names beginning with MPI_. There are specific data types for handles to each object. Arrays are indexed from 0. The choice arguments are pointers of type `void *`.

In FORTRAN all procedure names are in upper case and begin with MPI_. The last argument of every procedure is an integer, IERR, indicating the error condition of the procedure on exit. DDL constant names are also in upper case and begin with MPI_. Handles for objects and DDL types and constants are represented by `INTEGER`s. Arrays are indexed from 1. Choice arguments may be one of several different types.

2.5 Error handling

Every Library procedure returns an error value to indicate whether the procedure was successful or not. The Library recognizes a number of different error conditions and represents these by the integer value ierr. In FORTRAN programs ierr is the last parameter in the Library procedure call. In C programs ierr is the result of the Library function call.

It is good practice to check the value of ierr after every procedure call. If the value is DDL_SUCCESS then the procedure executed successfully and the program may be continued. Any other value indicates that an error occurred in the procedure and the user should then decide whether further execution is possible or whether the program should be terminated.

The normal behaviour of the Library on encountering an error is to exit the procedure immediately and display an error message. The state of any OUT or INOUT parameters on exit with an error is not defined. In most instances these parameters will be unchanged but this is not guaranteed. Any IN parameters will be unchanged. The safest user response to an error is to terminate the program.

More information about error handling can be found in Appendix B.

Chapter 3

Distributed objects

The distributed objects in the DDL support 1- and 2-dimensional matrices, the most commonly used dimensions. Objects belong to one of two classes: dense data objects or sparse data objects. A dense representation can be used for objects which are 1-d vectors or 2-d matrices. The sparse representation is for 2-dimensional matrix objects. Each class is composed of a number of object types. Each of these different types 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.

3.1 Distribution

All the objects that the DDL supports have the same general distribution pattern (see Figure 3.1). This distribution is a subset of the distributions provided by HPF.

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.

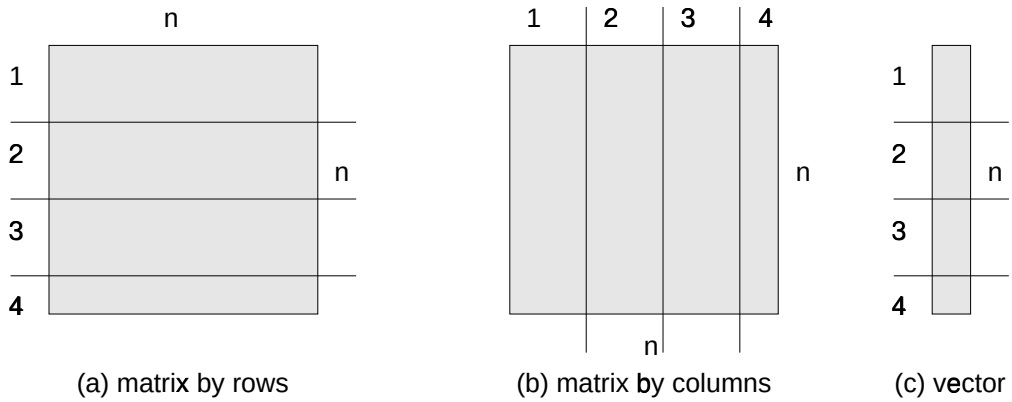

Figure 3.1: Distribution of matrices and vectors

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.

3.2 Permutations

The DDL allows the user to specify a permutation of an object's data. This 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.

3.3 Object properties

Each distributed object has a number of properties associated with it. The values of these properties define an instance of an object. The user specifies the object properties when the object is created, and he can later retrieve the properties of an object through a collection of query functions. Properties are of two types: either global properties or local properties. Global properties have the same value on all processes of the object's communicator. Local properties may have different values on each process in the communicator. The range of properties varies from one object type to another, but there are core properties common to all objects. These core properties are given below. The language independent type of each property is given in brackets with the property description. The language-specific type can be found by referring to Section 2.4.

objecttype (handle, global)

The base type of every distributed object or object property. This is stored so that DDL procedures, or the user, can check on the type of the object referenced by an object handle. This base type will determine which other properties an object has. objecttype can have the following values:

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

tag (handle, global)

This is used to give a unique name to each property of an object. This is needed since an object may have several properties of the same objecttype each representing different properties of the object. For 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_SPARSE	highest level sparse matrix object
DDL_VECTOR	highest level vector object
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

format (handle, global)
: The format of the object. format specifies the exact distributed data structure of an object. The values for format are type dependent. See the following sections for details.

comm (handle, global)
: The MPI communicator comm specifies the process group over which an object is distributed. The communicator includes information about the number of processes, the rank (or number) of each process in the group and logical configuration of the processes. The communicators of all objects involved in an operation must be compatible. For most operations this means that the communicators must be identical, i.e., MPI_COMM_COMPARE returns MPI_IDENT. A typical value for comm is MPI_COMM_WORLD.

gsize (array of integer, global)
: The global size of an object. The size of this array and the interpretation of each element are type dependent. See the following sections for details.

type (handle, global)
: The data type of each element of the distributed object. type may take a subset of the MPI data types as values:

C	FORTRAN	Description
MPI_FLOAT	MPI_REAL	single precision floating point
MPI_DOUBLE	MPI_DOUBLE_PRECISION	double precision floating point
MPI_INT	MPI_INTEGER	integer

See the MPI documentation for full details of MPI data types.

size (array of integer, local)
: The size of the local segment of an object. The size of this array and the interpretation of each element are type dependent. See the following sections for details.

offset (array of integer, local)
: The offset of the local segment of an object within the global object. The size of this array and the interpretation of each element are type dependent. See the following sections for details.

3.4 Dense vectors

Dense 1-D vector objects are of type DDL_Vec.

A vector is partitioned by blocks across the processes of the MPI communicator. Each process holds a block of consecutive elements of the vector determined by the rank of the process within the communicator. The local size property of the vector object is an array with a single element which is the number of elements in the block on this process. Similarly the offset property is an array with a single element which is the offset of the first element in the block on this process from the start of the vector. In FORTRAN the offset property is indexed from 1 following standard FORTRAN practice and so is 1 greater than the C value. The global property gsize is an array with a single element which is the size, *n*, of the global vector object.

3.5 Sparse matrices

3.5.1 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.

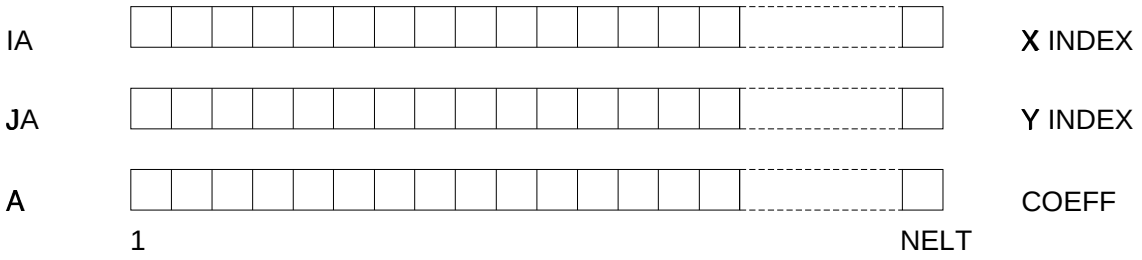

Figure 3.2: Triad format

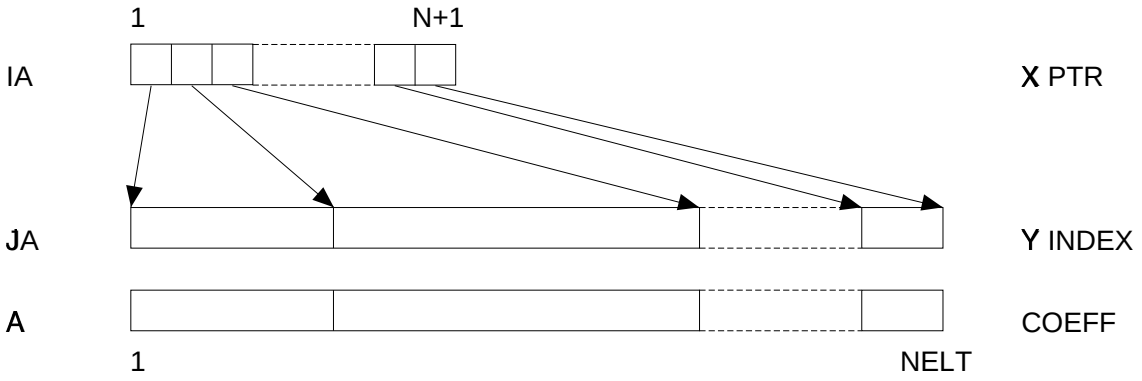

Figure 3.3: Row packed format

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 3.2). The elements are unordered.

In the row packed format (see Figure 3.3), `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 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`.

These single process sparse formats may be specified by the following format values:

DDL_ROW_FMT	single process row packed format
DDL_COL_FMT	single process column packed format
DDL_TRIAD_FMT	single process triad format

3.5.2 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.

A list of proposed sparse formats is given below by values for the format property:

DDL_PCK_ROW	distributed row packed format
DDL_PCK_COL	distributed column packed format
DDL_PAD_ROW	distributed row padded format
DDL_PAD_COL	distributed column padded format
DDL_TRIAD_ROW	row distributed triad format
DDL_TRIAD_COL	column distributed triad format
DDL_TRIAD	distributed triad format

The distributed row packed sparse format, DDL_PCK_ROW, is based on the row packed format (see Figure 3.4). The sparse matrix is distributed over the processes by whole rows, with each process holding a block of adjacent rows.

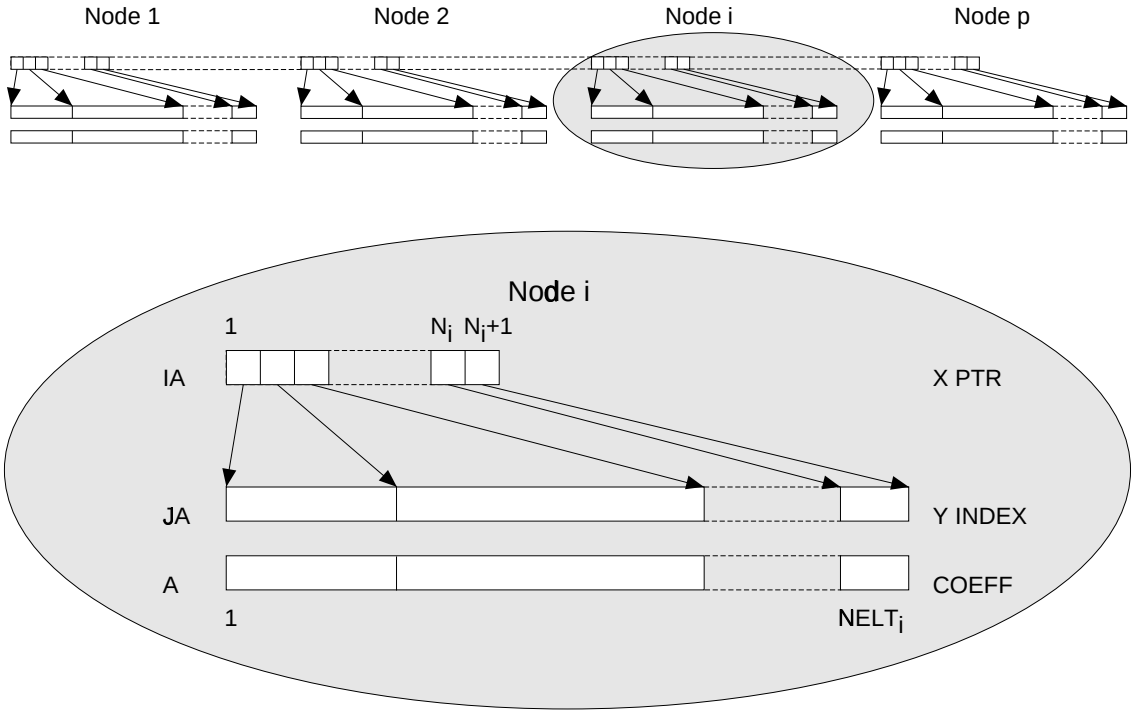

Figure 3.4: Distributed row packed format

The rows i allocated to a process are the same as the elements i of a dense vector that is partitioned over the same processes. 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.

The distributed column packed sparse format, DDL_PCK_COL, is similar to DDL_PCK_ROW except that processes hold blocks of columns stored locally in the column packed format.

The padded formats are similar to the packed formats. For example, the DDL_PAD_ROW distribution is the same as DDL_PCK_ROW except that each row is given an equal-sized block of storage on the process (see Figure 3.5). Thus each row may be padded out from the previous row by a number of empty storage locations. `ia` now contains pointers to the next empty element in each row. It is much easier to add extra elements to this format than to the packed formats.

The triad formats are again similar. In the DDL_TRIAD format data is stored in triad format on each process with no ordering to the elements at all. The DDL_TRIAD_ROW and DDL_TRIAD_COL formats partition the matrix by blocks of rows or columns. Within each process, however, the elements are stored in triad format with no ordering.

The distributed sparse objects have the same properties as the distributed dense objects. The type-specific properties are defined below for the sparse matrix object type:

gsize (array of integer, global)
: The gsize property of a sparse object is an array of 3 integers with the following interpretations:

1. the number of rows of the matrix,
2. the number of columns of the matrix,
3. the total number of non-zero elements that the object can hold.

type (handle, global)
: The type property gives the data type of the coefficient array `a`. The arrays `ia` and `ja` are arrays of integers.

size (array of integer, local)
: The local size property for a sparse object is an array of 2 elements with the following interpretations:

1. the number of rows (or columns) of the matrix stored on this process,
2. the total number of non-zero elements that this process can hold.

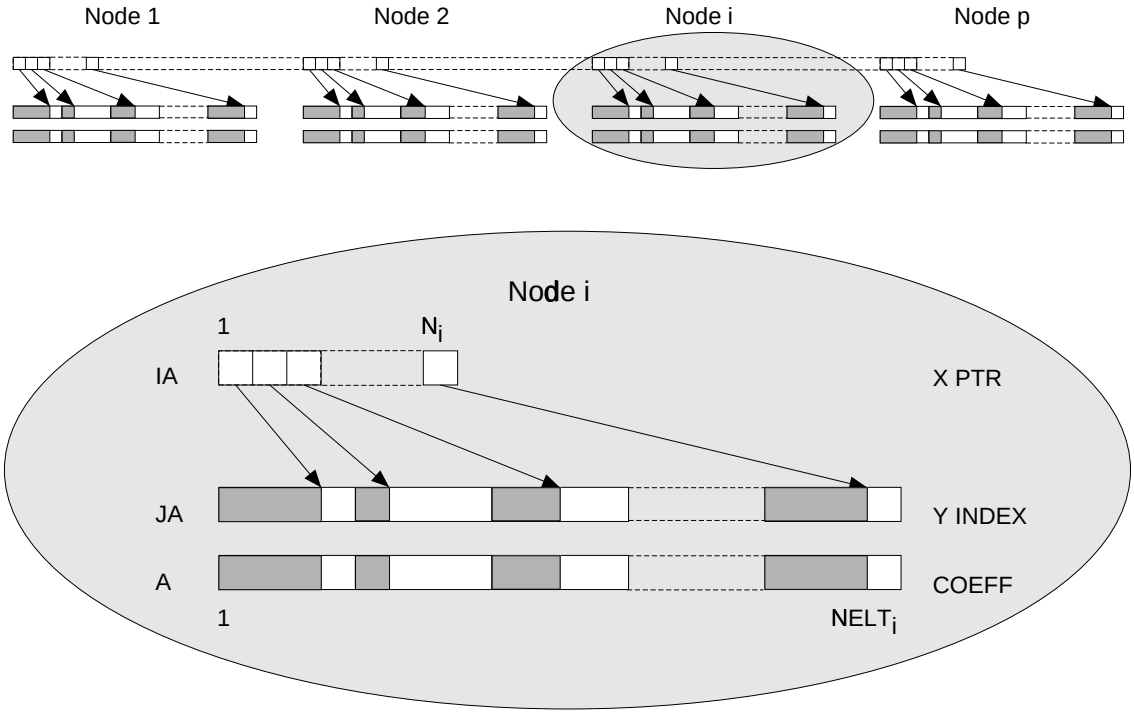

Figure 3.5: Distributed row padded format

offset (array of integer, local)
: The offset property is an array of 1 element which gives the offset to the first row (or column) held by this process. In FORTRAN the offset property is indexed from 1, and in C it is indexed from 0.

In addition, the sparse matrix object type has the following extra properties: subtype, permx, permy, nelt and gnelt.

subtype (handle, global)
: The subtype property of a sparse object specifies a special matrix type for the object. Currently this property can take the following values:

DDL_UPPER	the matrix is upper triangular,
DDL_LOWER	the matrix is lower triangular.

: If this property is not set then the matrix is assumed to be general sparse.

permx, permy (handle, global)
: These properties specify permutation vectors for the matrix. permx (DDL_Vec) specifies the row permutation for the matrix, and permy specifies the column permutation. (See Section 3.2.)

nelt (integer, local)
: The nelt property of a sparse object is the total number of non-zero elements held on the calling process.

gnelt (integer, global)
: The gnelt property of a sparse object is the total number of non-zero elements in the distributed object.

Chapter 4

Object management

This chapter describes how the user can manage DDL objects in his program. Management operations include creating and destroying objects, and querying and setting the properties of objects. All these operations are achieved through DDL procedure calls.

4.1 Object creation and deletion

A DDL object is created by calling a type specific DDL creation procedure. These procedures are DDL_Create_vector and DDL_Create_sparse. Depending on the type of the object to be created, different information must be provided by the user.

DDL_Create_vector(size, type, format, block, comm, vector)

IN	size	array of size 1 specifying the size of the vector (array of integer)
IN	type	the type of each element (handle)
IN	format	the format of the vector (handle)
IN	block	the size of indivisible blocks. Each node has a whole number of blocks. (integer)
IN	comm	communicator (handle)
OUT	vector	the new vector object (handle)

```
int DDL_Create_vector(int size[1], MPI_Datatype type, int format, int block,
            MPI_Comm comm, DDL_Vec *vector)
```

```
DDL_CREATE_VECTOR(SIZE, TYPE, FORMAT, BLOCK, COMM, VECTOR, IERR)
   INTEGER SIZE(1)
   INTEGER TYPE, FORMAT, BLOCK, COMM, VECTOR, IERR
```

DDL_Create_sparse(size, type, format, block, comm, sparse)

IN	size	array of size 3 specifying the size of the sparse matrix. The elements have the following meanings: 1. number of rows in the matrix 2. number of columns in the matrix

		3. maximum number of non-zero elements in matrix (array of integer)
IN	type	the type of each element (handle)
IN	format	the format of the sparse matrix (handle)
IN	block	the size of indivisible blocks. Each node has a whole number of blocks. (integer)
IN	comm	communicator (handle)
OUT	sparse	the new sparse matrix object (handle)

```
int DDL_Create_sparse(int size[3], MPI_Datatype type, int format, int block,
           MPI_Comm comm, DDL_Spa *sparse)
```

```
DDL_CREATE_SPARSE(SIZE, TYPE, FORMAT, BLOCK, COMM, SPARSE, IERR)
    INTEGER SIZE(3)
    INTEGER TYPE, FORMAT, BLOCK, COMM, SPARSE, IERR
```

All the creation procedures return a handle on the newly created object. The user must use this handle to refer to the object through the rest of his program.

When an object is no longer required it can be deallocated using a generic procedure DDL_Free.

DDL_Free(object)

OUT	object	the object to be deallocated (handle)

```
int DDL_Free(DDL_Object object)
```

```
DDL_FREE(OBJECT, IERR)
    INTEGER OBJECT, IERR
```

This procedure reclaims the memory allocated to the object so the object handle should not be used again to refer to the deleted object. Also any handles to sub-objects or pointers to raw data values become invalid and should not be used.

4.2 Property query procedures

If the user program needs to find out any property of an existing object, there is a set of procedures for querying an object. Most properties have a generic query procedure which can be used for any object type. The exact specification of the returned values may be type dependent. Full details of the object properties is given in Section 3.3.

The objecttype of an object can be found by calling DDL_Objecttype.

DDL_Objecttype(object, type)

IN	object	the object to query (handle)
OUT	type	the object type (handle)

```
int DDL_Objecttype(DDL_Object object, int *type)
```

```
DDL_OBJECTTYPE(OBJECT, TYPE, IERR)
    INTEGER OBJECT, TYPE, IERR
```

Other procedures to find global properties of an object are DDL_Tag to find the object tag; DDL_Format to get the object format; DDL_Gsize to return the global size of the object, gsize; DDL_Comm to get the object MPI communicator comm; and DDL_Type to retrieve the type of object elements.

DDL_Tag(object, tag)

IN	object	the object to query (handle)
OUT	tag	the object tag (handle)

```
int DDL_Tag(DDL_Object object, int *tag)
```

```
DDL_FORMAT(OBJECT, TAG, IERR)
   INTEGER OBJECT, TAG, IERR
```

DDL_Format(object, format)

IN	object	the object to query (handle)
OUT	format	the object format (handle)

```
int DDL_Format(DDL_Object object, int *format)
```

```
DDL_FORMAT(OBJECT, FORMAT, IERR)
   INTEGER OBJECT, FORMAT, IERR
```

DDL_Gsize(object, size)

IN	object	the object to query (handle)
OUT	size	the global size of the object (array of integers)

```
int DDL_Gsize(DDL_Object object, int *size)
```

```
DDL_GSIZE(OBJECT, SIZE, IERR)
   INTEGER(*) SIZE
   INTEGER OBJECT, IERR
```

The length of size in DDL_Gsize depends on the object type (see Section 3.3).

DDL_Comm(object, comm)

IN	object	the object to query (handle)
OUT	comm	the MPI communicator for the object (handle)

```
int DDL_Comm(DDL_Object object, MPI_Comm *comm)
```

```
DDL_TYPE(OBJECT, COMM, IERR)
   INTEGER OBJECT, COMM, IERR
```

DDL_Type(object, type)

IN	object	the object to query (handle)
OUT	type	the data type of object elements (handle)

```
int DDL_Type(DDL_Object object, MPI_Datatype *type)
```

```
DDL_TYPE(OBJECT, TYPE, IERR)
   INTEGER OBJECT, TYPE, IERR
```

Local properties of an object, which may return different values on different processes, can be found by calling DDL_Size to find the size of the local segment of an object and DDL_Offset to find the offset of the local segment.

DDL_Size(object, size)

IN	object	the object to query (handle)
OUT	size	the local size of the object (array of integers)

```
int DDL_Size(DDL_Object object, int *size)
```

```
DDL_SIZE(OBJECT, SIZE, IERR)
    INTEGER(*) SIZE
    INTEGER OBJECT, IERR
```

The length of size in DDL_Size depends on the object type (see Section 3.3).

DDL_Offset(object, offset)

IN	object	the object to query (handle)
OUT	offset	the offset of the local segment of the object (array of integers)

```
int DDL_Offset(DDL_Object object, int *offset)
```

```
DDL_OFFSET(OBJECT, OFFSET, IERR)
    INTEGER(*) OFFSET
    INTEGER OBJECT, IERR
```

The length of offset in DDL_Offset depends on the object type (see Section 3.3).

The additional properties of the DDL_Spa object type, gused and used, can be found by calling DDL_Gused_sparse and DDL_Used_sparse respectively.

DDL_Gused_sparse(sparse, nelt)

IN	sparse	the sparse object to query (handle)
OUT	nelt	the number of non-zero elements used by the object (integer)

```
int DDL_Gused_sparse(DDL_Spa sparse, int *nelt)
```

```
DDL_GUSED_SPARSE(SPARSE, NELT, IERR)
    INTEGER SPARSE, NELT, IERR
```

DDL_Used_sparse(sparse, nelt)

IN	sparse	the sparse object to query (handle)
OUT	nelt	the number of non-zero elements used by the object on the calling process (integer)

```
int DDL_Used_sparse(DDL_Spa sparse, int *nelt)
```

```
DDL_USED_SPARSE(SPARSE, NELT, IERR)
    INTEGER SPARSE, NELT, IERR
```

DDL_Subtype_sparse returns the subtype of the sparse matrix (see Section 3.5).

DDL_Subtype_sparse(sparse, type)

IN	sparse	the sparse object to query (handle)
OUT	type	the sparse object subtype (handle)

```
int DDL_Subtype_sparse(DDL_Spa *sparse, int *type)
```

```
DDL_SUBTYPE_SPARSE(SPARSE, TYPE, IERR)
    INTEGER SPARSE, TYPE, IERR
```

4.3 Property setting procedures

Most object properties are set on creation and cannot be changed thereafter. However some properties may be changed after an object has been created. Changing most object properties after creation is considered to be a low-level function of the DDL, great care is needed and the user should be aware of the possible overheads incurred by changing some properties.

The DDL_Spa type suppports the following property setting functions: DDL_Setformat sets the format of the sparse matrix; DDL_Setsubtype sets the subtype of the sparse matrix (see Section 3.5); DDL_Setperm_sparse sets the permutation vectors for the sparse matrix; and DDL_Setused sets the local property used, the number of elements used on the calling process.

DDL_Setused_sparse(sparse, nelt)

INOUT	sparse	the sparse object to change (handle)
IN	nelt	the new number of non-zero elements used by the object on the calling process (integer)

```
int DDL_Setused_sparse(DDL_Spa sparse, int nelt)
```

```
DDL_SETUSED_SPARSE(SPARSE, NELT, IERR)
    INTEGER SPARSE, NELT, IERR
```

DDL_Setformat_sparse(sparse, format)

INOUT	sparse	the sparse object to change (handle)
IN	format	the new sparse object format (handle)

```
int DDL_Setformat_sparse(DDL_Spa sparse, int format)
```

```
DDL_SETFORMAT_SPARSE(SPARSE, FORMAT, IERR)
    INTEGER SPARSE, FORMAT, IERR
```

DDL_Setsubtype_sparse(sparse, type)

INOUT	sparse	the sparse object to change (handle)
IN	type	the new sparse object subtype (handle)

```
int DDL_Setsubtype_sparse(DDL_Spa sparse, int type)
```

```
DDL_SETSUBTYPE_SPARSE(SPARSE, TYPE, IERR)
    INTEGER SPARSE, TYPE, IERR
```

DDL_Setperm_sparse(sparse, permx, permy)

INOUT	sparse	the sparse object to change (handle)
IN	permx	the row permutation vector for the sparse object, or NULL if no permutation (handle)
IN	permx	the column permutation vector for the sparse object, or NULL if no permutation (handle)

```
int DDL_Setperm_sparse(DDL_Spa sparse, DDL_Vec permx, DDL_Vec permy)
```

```
DDL_SETPERM_SPARSE(SPARSE, PERMX, PERMY, IERR)
    INTEGER SPARSE, PERMX, PERMY, IERR
```

These procedures may be used, for example, while initializing a sparse matrix object. The user could update the underlying vectors `ia`, `ja` and `a` directly, and then inform the DDL of the changes using these procedures.

4.4 Copying and transforming objects

As well as simply creating an object and filling it with data, the user may want to copy data from one object to another. Simple copy operations could involve two objects of identical types, formats and sizes. These operations would not involve any communication and could be implemented efficiently. However, the user may also want to copy data between objects with different properties, such as different formats. These transformation operations will probably require communication and hence will be costly.

The DDL uses different procedures to provide simple object copying and more complicated object transformations. A generic copy procedure DDL_Copy allows the user to copy data between objects, provided the objects are similar enough such that no communication is required to perform the data copy, and that no data manipulation is required, such as reordering. The transformation procedure DDL_Transform_sparse allows the user to transform sparse objects from one data format to another, which may require communication.

DDL_Copy(src, dest)

IN	src	the source object (handle)
OUT	dest	the destination object (handle)

```
int DDL_Copy(DDL_Object src, DDL_Object dest)
```

```
DDL_COPY(SRC, DEST, IERR)
    INTEGER SRC, DEST, IERR
```

Note that all objects must use the same communicator and thus the same number of processes. In order to copy object data between objects using different numbers of processes the user must write the data to file and re-read it. This restriction to identical communicators is a general restriction in all DDL procedures and means that programs will in practice use a single communicator, e.g. MPI_COMM_WORLD, throughout.

Chapter 5

Level 1 BLAS

Some of the operations defined by the level 1 Basic Linear Algebra Subprograms (BLAS1) are available for manipulating DDL_Vec objects. The DDL procedures do not support strides. Both single and double precision floating point values are supported. All the parameters must be of the same type, i.e., MPI_FLOAT or MPI_DOUBLE.

DDL_Dot(x, y, dot)

IN	x	vector x (handle)
IN	y	vector y (handle)
OUT	dot	the dot product of x and y (choice)

```
int DDL_Dot(DDL_Vec x, DDL_Vec y, void *dot)
```

```
DDL_DOT(X, Y, DOT, IERR)
    INTEGER X, Y, IERR
    <type> DOT
```

DDL_Scal(alpha, x)

IN	alpha	the constant scale factor (choice)
INOUT	x	vector x (handle)

```
int DDL_Scal(double alpha, DDL_Vec x)
```

```
DDL_SCAL(ALPHA, X, IERR)
    <type> ALPHA
    INTEGER X, IERR
```

DDL_Nrm2(x, nrm2)

IN	x	vector x (handle)
OUT	nrm2	the 2-norm of the vector x choice)

```
int DDL_Nrm2(DDL_Vec x, void *nrm2)
```

```
DDL_NRM2(X, NRM2, IERR)
    INTEGER X, IERR
    <type> NRM2
```

DDL_Axpy(alpha, x, y)

IN	alpha	the constant scale factor for x (choice)
IN	x	vector x (handle)
INOUT	y	vector y (handle)

```
int DDL_Axpy(double alpha, DDL_Vec x, DDL_Vec y)
```

```
DDL_AXPY(ALPHA, X, Y, IERR)
    <type> ALPHA
    INTEGER X, Y, IERR
```

The copy operation is described in Section 4.4.

Two other useful operations are DDL_Rand which initialises the elements of a vector with random numbers, and DDL_Fill which initialises the elements of a vector with a constant.

DDL_Rand(alpha, x)

IN	alpha	the seed (double)
OUT	x	vector x (handle)

```
int DDL_Rand(double alpha, DDL_Vec x)
```

```
DDL_RAND(ALPHA, X, IERR)
    DOUBLE PRECISION ALPHA
    INTEGER X, IERR
```

DDL_Fill(alpha, x)

IN	alpha	the initialization constant (double)
OUT	x	vector x (handle)

```
int DDL_Fill(double alpha, DDL_Vec x)
```

```
DDL_FILL(ALPHA, X, IERR)
    DOUBLE PRECISION ALPHA
    INTEGER X, IERR
```

Chapter 6

Sparse matrix operations

6.1 Sparse matrix vector multiply

There is a sparse matrix, dense vector multiply routine DDL_Smv. This routine takes a sparse matrix A and a pair of dense vectors x and y and performs the following operation:

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

The matrix vector multiply algorithm must communicate data between processes. The communication pattern will depend on the sparsity pattern of the sparse matrix. To improve efficiency when repeated matrix vector multiplies are performed with the same matrix, the communication pattern is calculated before the computation is performed. The communication pattern is then saved for re-use in subsequent matrix vector multiplies. This is achieved by the user inserting a call to DDL_Smv_comm after the sparse data has been distributed but before DDL_Smv is called. Repeated calls to DDL_Smv can then be made without recalculating the communication pattern unless the sparsity pattern is changed by updating the sparse matrix. If the sparsity pattern is subsequently changed (or if the user has changed the sparse matrix but doesn't know whether the sparsity pattern has changed) the user should call DDL_Smv_comm again to recalculate the communication pattern.

If the user only performs a single matrix vector multiply or if the sparse matrix is not changed then calls to DDL_Smv_comm are not necessary since DDL_Smv will automatically calculate the communication pattern for a sparse matrix if the pattern has not been calculated before.

This procedure supports permutation vectors for the sparse matrix.

DDL_Smv(a, x, y, alpha, beta)

IN	a	sparse matrix (handle)
IN	x	dense vector (handle)
INOUT	y	dense vector (handle)
IN	alpha	scalar (choice)
IN	beta	scalar (choice)

```
int DDL_Smv(DDL_Spa a, DDL_Vec x, DDL_Vec y, double alpha, double beta)
```

```
DDL_SMV(A, X, Y, ALPHA, BETA, IERR)
    INTEGER A, X, Y, IERR
    <type> ALPHA, BETA
```

DDL_Smv_comm(sparse)

IN	sparse	sparse matrix (handle)

```
int DDL_Smv_comm(DDL_Spa sparse)
```

```
DDL_SMV_COMM(SPARSE, IERR)
    INTEGER SPARSE, IERR
```

6.2 Sparse triangular solve

There is a sparse triangular solve routine DDL_Trisolve. This routine takes a sparse triangular matrix A and a pair of dense vectors x and b and solves the triangular system of equations for x:

$$Ax = b.$$

The sparse matrix must be either upper triangular or lower triangular. This routine uses the property subtype to determine whether the matrix is upper or lower triangular. Hence the user must call DDL_Setsubtype (see Section 4.3) to set this property before calling DDL_Trisolve_comm or DDL_Trisolve.

The triangular solve algorithm must communicate data between processes. The communication pattern will depend on the sparsity pattern of the sparse matrix. To improve efficiency when repeated solves are performed with the same matrix, the communication pattern is calculated before the computation is performed. The communication pattern is then saved for re-use in subsequent solves. This is achieved by the user inserting a call to DDL_Trisolve_comm after the sparse data has been distributed but before DDL_Trisolve is called. Repeated calls to DDL_Trisolve can then be made without recalculating the communication pattern unless the sparsity pattern is changed by updating the sparse matrix. If the sparsity pattern is subsequently changed (or if the user has changed the sparse matrix but doesn't know whether the sparsity pattern has changed) the user should call DDL_Trisolve_comm again to recalculate the communication pattern.

If the user only performs a single triangular solve or if the sparse matrix is not changed then calls to DDL_Trisolve_comm are not necessary since DDL_Trisolve will automatically calculate the communication pattern for a sparse matrix if the pattern has not been calculated before.

This procedure supports permutation vectors for the sparse matrix.

DDL_Trisolve(a, x, b)

IN	a	sparse matrix (handle)
OUT	x	dense solution vector (handle)
IN	b	dense right hand side (RHS) vector (handle)

```
int DDL_Trisolve(DDL_Spa a, DDL_Vec x, DDL_Vec b)
```

```
DDL_TRISOLVE(A, X, B, IERR)
    INTEGER A, X, B, IERR
```

DDL_Trisolve_comm(sparse)

IN	sparse	sparse matrix (handle)

```
int DDL_Trisolve_comm(DDL_Spa sparse)
```

```
DDL_TRISOLVE_COMM(SPARSE, IERR)
    INTEGER SPARSE, IERR
```

6.3 Conversion between sparse formats

The distributed sparse object type supports several different formats which may be useful in different circumstances. The DDL provides a number of procedures to convert data from one of these formats to another.

> *Implementation note.* It is relatively simple to convert between certain distributed formats, but to provide general conversion between any formats might require considerable communication. Thus the procedures provided now

are sequential and may only be used to operate upon local sparse data structures. They may be useful for I/O and for converting local data to the format required by new user-written procedures.

Fully distributed conversion procedures can be developed if there is a need for them.

The sequential procedures are DDL_Triadtorow, DDL_Triadtocol and DDL_Rowtotriad, DDL_Coltotriad. These procedures convert between single process triad format (DDL_TRIAD_FMT) and single process row- or column-packed formats (DDL_ROW_FMT and DDL_COL_FMT). After these routines have been used to convert local sparse data structures from one format to another, DDL_Setformat_sparse must be called to inform the DDL of the change.

DDL_Coltotriad(n, nelt, ia, ja, a, type, isym)

IN	n	size of sparse matrix (integer)
IN	nelt	number of non-zero elements in matrix (integer)
INOUT	ia	vector of row indices (array of integer)
INOUT	ja	on entry, vector of column pointers; on exit, vector of column indices (array of integer)
INOUT	a	vector of coefficient values (array of choice)
IN	type	the type of the coefficient values (handle)
IN	isym	an integer specifying whether the matrix is symmetric or non-symmetric (integer)

```
int DDL_Coltotriad(int n, int nelt, int ia[], int ja[], void *a,
           MPI_Datatype type, int isym)

DDL_COLTOTRIAD(N, NELT, IA, JA, A, TYPE, ISYM, IERR)
   INTEGER N, NELT, IA(NELT), JA(NELT), TYPE, ISYM, IERR
   <type> A(NELT)
```

DDL_Rowtotriad(n, nelt, ia, ja, a, type, isym)

IN	n	size of sparse matrix (integer)
IN	nelt	number of non-zero elements in matrix (integer)
INOUT	ia	on entry, vector of row pointers; on exit, vector of row indices (array of integer)
INOUT	ja	vector of column indices (array of integer)
INOUT	a	vector of coefficient values (array of choice)
IN	type	the type of the coefficient values (handle)
IN	isym	an integer specifying whether the matrix is symmetric or non-symmetric (integer)

```
int DDL_Rowtotriad(int n, int nelt, int ia[], int ja[], void *a,
           MPI_Datatype type, int isym)

DDL_ROWTOTRIAD(N, NELT, IA, JA, A, TYPE, ISYM, IERR)
   INTEGER N, NELT, IA(NELT), JA(NELT), TYPE, ISYM, IERR
   <type> A(NELT)
```

DDL_Triadtocol(n, nelt, ia, ja, a, type, isym)

IN	n	size of sparse matrix (integer)
IN	nelt	number of non-zero elements in matrix (integer)
INOUT	ia	vector of row indices (array of integer)
INOUT	ja	on entry, vector of column indices; on exit, vector of column pointers (array of integer)
INOUT	a	vector of coefficient values (array of choice)
IN	type	the type of the coefficient values (handle)
IN	isym	an integer specifying whether the matrix is symmetric or non-symmetric (integer)

```
int DDL_Triadtocol(int n, int nelt, int ia[], int ja[], void *a,
            MPI_Datatype type, int isym)

DDL_TRIADTOCOL(N, NELT, IA, JA, A, TYPE, ISYM, IERR)
   INTEGER N, NELT, IA(NELT), JA(NELT), TYPE, ISYM, IERR
   <type> A(NELT)
```

DDL_Triadtorow(n, nelt, ia, ja, a, type, isym)

IN	n	size of sparse matrix (integer)
IN	nelt	number of non-zero elements in matrix (integer)
INOUT	ia	on entry, vector of row indices; on exit, vector of row pointers (array of integer)
INOUT	ja	vector of column indices (array of integer)
INOUT	a	vector of coefficient values (array of choice)
IN	type	the type of the coefficient values (handle)
IN	isym	an integer specifying whether the matrix is symmetric or non-symmetric (integer)

```
int DDL_Triadtorow(int n, int nelt, int ia[], int ja[], void *a,
            MPI_Datatype type, int isym)

DDL_TRIADTOROW(N, NELT, IA, JA, A, TYPE, ISYM, IERR)
   INTEGER N, NELT, IA(NELT), JA(NELT), TYPE, ISYM, IERR
   <type> A(NELT)
```

Chapter 7

File I/O

The DDL provides two sets of I/O procedures: one set performs direct parallel I/O between the file system and the process group; the other set performs ordinary I/O between the file system and the process on the host node. The operations provided in each set include file opening and closing, object reading and writing, and file querying.

7.1 Parallel I/O

The direct parallel I/O procedures are DDL_Open, DDL_Close, DDL_Read, DDL_Write and DDL_Fileformat. These procedures provide parallel I/O operations for a group of processes defined by an MPI communicator. All processes in the group must call these procedures, but the user does not need to make any distinction between processes that have I/O capability and those without that capability. The physical location of the file system is hidden from the user program. The only requirement is that one process in the process group has I/O capability.

These procedures allow the user to write distributed objects to file and later read the object data back from file to a distributed object. The user can use these procedures to read in initial data and write out result data, or to implement a simple form of checkpointing where, under user control, all the distributed objects can be dumped to file for later retrieval.

The data file format implemented in the DDL is intended to support checkpointing in a simple manner. All the essential information about a distributed object is recorded in a single file. This information includes the following object properties: the object DDLtype, format, gsize and type along with the raw data values for the object. The file format is independent of the size of the process group for an object, so programs using different numbers of processes can read and write the same data files.

Distributed objects can be written one after another to a single data file, and later read back in the same order. To facilitate reading objects from file the procedure DDL_Fileformat finds the object properties of the next object in the given data file. This information can then be used to create a compatible distributed object and that object filled with data from the file using DDL_Read.

> *Implementation note.* Currently the set of pairs of formats of the data file object and the distributed object that are accepted by the DDL_Read procedure is limited. For example, if a row packed sparse matrix object is written to file, it can only be read back into a row packed, row padded or row distributed triad object. This restriction may be relaxed in future if there is a demand for more flexible I/O.

The specifications for these parallel I/O procedures are given below.

DDL_Open(file, fname, mode, comm)

OUT	file	a handle on the newly opened file (handle)
IN	fname	the name of the file (array of character)
IN	mode	the mode for file access. This can have a number of values:

DDL_READ_MODE	Open file for reading from the beginning.
DDL_WRITE_MODE	Open file for writing, deleting any previous contents of file.

(handle)

IN	comm	the communicator specifying the process group which needs access to the file (handle)

```
int DDL_Open(FILE **file, char *fname, int mode, MPI_Comm comm)
```

```
DDL_OPEN(FILE, FNAME, MODE, COMM, IERR)
    CHARACTER*(*) FNAME
    INTEGER FILE, MODE, COMM, IERR
```

DDL_Close(file, comm)

IN	file	a handle on the opened file (handle)
IN	comm	the communicator specifying the process group which has access to the file (handle)

```
int DDL_Close(FILE *file, MPI_Comm comm)
```

```
DDL_CLOSE(FILE, COMM, IERR)
    INTEGER FILE, COMM, IERR
```

DDL_Fileformat(file, format, type, info, comm)

IN	file	a handle on the file (handle)
OUT	format	the format of the next object in the file (handle)
OUT	type	the type of the next object in the file (handle)
OUT	info	an array of extra information about the next object in the file. The size of this array and the interpretation of its elements is object type dependent. For DDL_Vec objects: 1. the number of elements in the vector object. For DDL_Spa objects: 1. the number of rows of the sparse matrix, 2. the number of columns of the sparse matrix, 3. the total number of non-zero elements in the sparse matrix. (array of integers)
IN	comm	the communicator specifying the process group which has access to the file (handle)

```
int DDL_Fileformat(FILE **file, int *format, MPI_Datatype *type, int *info,
           MPI_Comm comm)
```

```
DDL_FILEFORMAT(FILE, FORMAT, TYPE, INFO, COMM, IERR)
```

```
    INTEGER(*) INFO
    INTEGER FILE, FORMAT, TYPE, COMM, IERR
```

DDL_Read(object, file)

IN	object	the object to fill from file (handle)
IN	file	the file from which to get data (handle)

```
int DDL_Read(DDL_Object object, FILE *file)
```

```
DDL_READ(OBJECT, FILE, IERR)
    INTEGER OBJECT, FILE, IERR
```

DDL_Write(object, file)

IN	object	the object to write to file (handle)
IN	file	the file to which to write data (handle)

```
int DDL_Write(DDL_Object object, FILE *file)
```

```
DDL_WRITE(OBJECT, FILE, IERR)
    INTEGER OBJECT, FILE, IERR
```

7.2 Host I/O

The set of host I/O procedures mirror the direct parallel I/O procedures. The host I/O procedures must be called from a process having I/O capability, a *host process*. The data to be read or written is transferred between the host file system and data structures in the host process. These procedures can be used in sequential programs independently of the other DDL procedures; no DDL or MPI initialization is required for these procedures to work. This makes the procedures useful for taking data from other programs and massaging it into a format compatible with the DDL parallel I/O procedures, or vice versa.

The host I/O procedures are DDL_Open_host, DDL_Close_host, DDL_In_sparse, DDL_In_vector, DDL_Out_sparse, DDL_Out_vector, and DDL_Fileformat_host.

DDL_Open_host, DDL_Close_host and DDL_Fileformat_host are similar in specification to the equivalent parallel I/O procedures described in Section 7.1. DDL_In_sparse, DDL_In_vector, DDL_Out_sparse and DDL_Out_vector provide reading and writing of DDL format data files from data structures on the host process. Finally, DDL_Inhb can input the common Harwell-Boeing format data files. The host data structures used by these procedures are simple arrays for DDL_Vec objects and the data structures described in Section 3.5.1 for DDL_Spa objects.

The specification for these procedures follows.

DDL_Open_host(file, fname, mode)

OUT	file	a handle on the newly opened file (handle)
IN	fname	the name of the file (array of character)
IN	mode	the mode for file access. For full details see Section 7.1 (handle)

```
int DDL_Open_host(FILE **file, char *fname, int mode)
```

```
DDL_OPEN_HOST(FILE, FNAME, MODE, IERR)
    CHARACTER*(*) FNAME
    INTEGER FILE, MODE, IERR
```

DDL_Close_host(file)

IN	file	a handle on the opened file (handle)

```
int DDL_Close_host(FILE *file)
```

```
DDL_CLOSE_HOST(FILE, IERR)
   INTEGER FILE, IERR
```

DDL_Fileformat_host(file, format, type, info)

IN	file	a handle on the file (handle)
OUT	format	the format of the next object in the file (handle)
OUT	type	the type of the next object in the file (handle)
OUT	info	an array of extra information about the next object in the file. For full details see Section 7.1 (array of integers)

```
int DDL_Fileformat_host(FILE **file, int *format, MPI_Datatype *type,
            int *info)
```

```
DDL_FILEFORMAT_HOST(FILE, FORMAT, TYPE, INFO, IERR)
   INTEGER(*) INFO
   INTEGER FILE, FORMAT, TYPE, IERR
```

DDL_In_sparse(file, n, nelt, ia, ja, a, type, isym, format)

IN	file	a handle on the file from which to read (handle)
OUT	n	the number of rows/columns of the sparse matrix (integer)
OUT	nelt	the number of non-zero elements in the sparse matrix (integer)
OUT	ia	an array holding the row pointers or row indices of the sparse matrix (array of integers)
OUT	ja	an array holding the column pointers or column indices of the sparse matrix (array of integers)
OUT	a	an array holding the coefficient values of the sparse matrix (array of choice)
OUT	type	the type of coefficient elements of the sparse matrix (handle)
OUT	isym	an integer specifying whether the matrix is symmetric or non-symmetric (integer)
OUT	format	the format of the sparse matrix (handle)

```
int DDL_In_sparse(FILE *file, int *n, int *nelt, int ia[], int ja[], void *a,
            MPI_Datatype *type, int *isym, int *format)
```

```
DDL_IN_SPARSE(FILE, N, NELT, IA, JA, A, TYPE, ISYM, FORMAT, IERR)
   INTEGER(*) IA, JA
   (<type>) A
   INTEGER FILE, N, NELT, TYPE, ISYM, FORMAT, IERR
```

DDL_Out_sparse(file, n, nelt, ia, ja, a, type, isym, format)

IN	file	a handle on the file to which to write (handle)
IN	n	the number of rows/columns of the sparse matrix (integer)
IN	nelt	the number of non-zero elements in the sparse matrix (integer)
IN	ia	an array holding the row pointers or row indices of the sparse matrix (array of integers)
IN	ja	an array holding the column pointers or column indices of the sparse matrix (array of integers)
IN	a	an array holding the coefficient values of the sparse matrix (array of choice)
IN	type	the type of coefficient elements of the sparse matrix (handle)
IN	isym	an integer specifying whether the matrix is symmetric or non-symmetric (integer)
IN	format	the format of the sparse matrix (handle)

```
int DDL_Out_sparse(FILE *file, int n, int nelt, int ia[], int ja[], void *a,
           MPI_Datatype type, int isym, int format)

DDL_OUT_SPARSE(FILE, N, NELT, IA, JA, A, TYPE, ISYM, FORMAT, IERR)
   INTEGER(*) IA, JA
   (<type>) A
   INTEGER FILE, N, NELT, TYPE, ISYM, FORMAT, IERR
```

DDL_In_vector(file, vec, type, n)

IN	file	a handle on the file from which to read (handle)
OUT	vec	an array holding the element values of the vector (array of choice)
OUT	type	the type of the vector elements (handle)
OUT	n	the number of elements in the vector (integer)

```
int DDL_In_vector(FILE *file, void *vec, MPI_Datatype *type, int *n)

DDL_IN_VECTOR(FILE, VEC, TYPE, N, IERR)
   (<type>) VEC
   INTEGER FILE, TYPE, N, IERR
```

DDL_Out_vector(file, vec, type, n)

IN	file	a handle on the file to which to write (handle)
IN	vec	an array holding the element values of the vector (array of choice)
IN	type	the type of the vector elements (handle)
IN	n	the number of elements in the vector (integer)

```
int DDL_Out_vector(FILE *file, void *vec, MPI_Datatype type, int *n)

DDL_OUT_VECTOR(FILE, VEC, TYPE, N, IERR)
   (<type>) VEC
   INTEGER FILE, TYPE, N, IERR
```

DDL_Inhb(file, n, nelt, ia, ja, a, isym, soln, rhs, xinit, job)

IN	file	a handle on the file from which to read (handle)
OUT	n	the number of rows/columns of the sparse matrix (integer)
OUT	nelt	the number of non-zero elements in the sparse matrix (integer)
OUT	ia	an array holding the row indices of the sparse matrix (array of integers)
OUT	ja	an array holding the column pointers of the sparse matrix (array of integers)
OUT	a	an array holding the coefficient values of the sparse matrix (array of double)
OUT	isym	an integer specifying whether the matrix is symmetric or non-symmetric (integer)
OUT	soln	an array holding the solution vector, if requested and found (array of double)
OUT	rhs	an array holding the right hand side vector, if requested and found (array of double)
OUT	xinit	an array holding the initial x-vector, if requested and found (array of double)
INOUT	job	A flag indicating on entry what I/O operations to perform: 0 Read only the matrix. 3 Read matrix, rhs, xinit and soln (if present). On exit the flag indicates what operations were actually performed: -3 Unable to parse matrix "code" from input file to determine if only the lower triangle of matrix is stored. -2 Number of non-zeros (nelt) too large. -1 System size (n) too large. 1 Read in only the matrix *structure*, but no nonzero entries. hence, a is not referenced and has the same return values as on input. 3 Read in only the matrix. 7 Read in the matrix and rhs. 15 Read in the matrix, rhs, and xinit. 23 Read in the matrix, rhs, and soln. 31 Read in the matrix, rhs, soln and xinit. (integer)

```
int DDL_Inhb(FILE *file, int *n, int *nelt, int ia[], int ja[], double a[],
             int *isym, double soln[], double rhs[], double xinit[], int *job)

DDL_INHB(FILE, N, NELT, IA, JA, A, ISYM, SOLN, RHS, XINIT, JOB)
    INTEGER(*) IA, JA
    DOUBLE PRECISION(*) A, SOLN, RHS, XINIT
    INTEGER FILE, N, NELT, ISYM, JOB
```

DDL_Inhb can only input Harwell-Boeing format data files which specify floating point values using E for the exponent, not D.

Chapter 8

Direct access to object data

The functionality provided by the DDL procedures will often not be sufficient for the user's application. In this case the DDL allows the user to get direct access to the object data values. Thus the user can implement any data manipulations required by his application. The DDL provides the user with a means to access object data values held on the local process. If the data manipulations require data held on other processes then the user must include explicit communications using MPI. It is also the user's responsibility to make sure that the data structures he accesses are maintained in a DDL-consistent state. For example, if the user adds fill-in elements to a sparse matrix using direct access to the object data, he must maintain the object format and inform the DDL of his changes by updating the used property to reflect the new number of elements, nelt, held locally on each process.

8.1 Access procedures

The DDL access procedures all return pointers to the arrays of data values that constitute the object. The procedure DDL_Get_vector returns a pointer to the array of data values which constitutes the local part of a vector object. Similarly, DDL_Get_ia, DDL_Get_ja and DDL_Get_a return pointers to the local parts of the ia, ja and a vectors of a sparse matrix, respectively.

DDL_Get_vector(vector, ptr)

IN	vector	a vector object (handle)
OUT	ptr	a pointer to the local vector data values (handle)

```
int DDL_Get_vector(DDL_Vec vector, void **ptr)
```

```
DDL_GET_VECTOR(VECTOR, PTR, IERR)
    INTEGER VECTOR, PTR, IERR
```

DDL_Get_ia(sparse, ia)

IN	sparse	a sparse matrix object (handle)
OUT	ia	a pointer to the local ia data values (handle)

```
int DDL_Get_ia(DDL_Spa sparse, int **ia)
```

```
DDL_GET_IA(SPARSE, IA, IERR)
    INTEGER SPARSE, IA, IERR
```

DDL_Get_ja(sparse, ja)

IN	sparse	a sparse matrix object (handle)
OUT	ja	a pointer to the local ja data values (handle)

```
int DDL_Get_ja(DDL_Spa sparse, int **ja)
```

```
DDL_GET_JA(SPARSE, JA, IERR)
   INTEGER SPARSE, JA, IERR
```

DDL_Get_a(sparse, a)

IN	sparse	a sparse matrix object (handle)
OUT	a	a pointer to the local a data values (handle)

```
int DDL_Get_a(DDL_Spa sparse, void **a)
```

```
DDL_GET_A(SPARSE, A, IERR)
   INTEGER SPARSE, A, IERR
```

8.2 Access methods

The methods for using the return pointer are language-dependent, and are described below with example code fragments.

8.2.1 C

In C the return pointer is a normal C pointer. Thus, the pointer can be used in the usual manner to reference elements of the array.

Example 8.1 *Initialise a distributed vector object to zero. Notice the agreement in type between the pointer* `double *data` *and the* type *property of the vector object* MPI_DOUBLE.

```
DDL_Vec y;
double *data;
MPI_Datatype type;
...
y = DDL_VECTOR_NULL;
type = MPI_DOUBLE;
ierr = DDL_Create_vector(n, type, block, comm, &y);
...
/* initialize vector y */
ierr = DDL_Get_vector(y, &data);
ierr = DDL_Size_vector(y, &size);
for (i=0;i<size;i++) data[i] = 0.0;
```

Example 8.2 *Initialise a distributed sparse matrix object to a diagonal matrix. The matrix is initially generated in triad format, which is usually the easiest format to generate. But notice that the matrix is then converted to row packed format to be consistent with the DDL* format *property for the object. Note also that the code updates the* used *property.*

```
DDL_Spa m;
double *a;
int *ia, *ja;
int sizes[3];
int format, i, size, offset;
...
```

```
/* create DDL sparse matrix */
format = DDL_PCK_ROW;
ierr = DDL_Create_sparse(sizes, type, format, block, comm, &m);
...
/* update sparse matrix vectors directly */
ierr = DDL_Get_ia(m, &ia);
ierr = DDL_Get_ja(m, &ja);
ierr = DDL_Get_a(m, &a);
ierr = DDL_Size_sparse(m, sizes);
size = sizes[0];
ierr = DDL_Offset_sparse(m, &offset);
/* remember FORTRAN indexing and local */
/* row ordered triad format */
/* just create diagonal matrix */
for (i=0;i<size;i++) {
  ia[i] = i+1;
  ja[i] = offset+i+1;
  a[i] = offset+i+1;
}
/* convert to row packed format as specified at creation */
ierr = DDL_Triadtorow(size, size, ia, ja, a, type, 0);
/* update sparse object properties */
ierr = DDL_Setused_sparse(m, size);
```

8.2.2 F77

In F77 the return pointer is held in an INTEGER since the F77 standard does not support pointers. Thus, the program cannot directly access the array referenced by the pointer. Two access methods are supported by the DDL, both of which require non-standard F77.

Using the POINTER data type

The simplest access method makes use of the POINTER data type supported by many F77 compilers including those from IBM, SUN and SGI. This data type associates a FORTRAN array variable with a pointer by using the pointer value as the base for all accesses to the array. The pointer variable is passed as a parameter to the access procedure, e.g., DDL_Get_vector, and the associated array variable is used to access elements of the DDL object. (Refer to your compiler documentation to check whether this extension to F77 is supported by your compiler.)

Example 8.3 *Initialise a distributed vector object to a value. Notice the agreement in type between the array* `double precision vec(10)` *and the* type *property of the vector object* MPI_DOUBLE_PRECISION.

```
      program demo
      integer i, y, n, type
      double precision vec(10)
      pointer (vecptr, vec)
      double precision val
      ...
c     create vector object
      type = MPI_DOUBLE_PRECISION
      call DDL_Create_vector(n, type, block, comm, y, ierr)
      ...
c     initialize vector
      call DDL_Get_vector(y, vecptr, ierr)
      call DDL_Size_vector(y, n, ierr)
      val = 0.0
      do i=1, n
         vec(i) = val
      end do
      ...
      end
```

Using the DDL_Access procedure

The second method for accessing the DDL object array in F77 uses a user-written subroutine. This subroutine is passed as a parameter to DDL_Access, along with the pointer and other relevant data. DDL_Access in turn calls the user's subroutine passing the array in as a normal F77 array.

The procedure DDL_Access does not conform strictly to the F77 standard since it allows a variable number of arguments. However, most F77 compilers allow this use. The arguments to DDL_Access are as follows:

1	user's F77 access subroutine name,
2	the number, n, of DDL pointer type arguments in this call,
3 to $n+2$	the n pointer arguments,
$n+3$	the number, m, of standard F77 arguments in this call,
$n+4$ to $n+m+3$	the m standard F77 arguments,
$n+m+4$	the ierr error argument.

The user's access subroutine, named in the first argument to DDL_Access, must have an argument list that matches the arguments passed to DDL_Access. The arguments passed to DDL_Access are in turn passed to the user's subroutine in the following order:

1 to n	the n arrays referenced by the pointer arguments,
$n+1$ to $n+m$	the m standard F77 arguments,
$n+m+1$	the ierr error argument.

The arrays passed to the user's subroutine are standard F77 arrays and can now be accessed using normal FORTRAN statements.

This method is more clearly illustrated by the following examples which are similar to the C examples given above.

Example 8.4 *Initialise a distributed vector object to a value. Notice the agreement in type between the array* `double precision data(n)` *and the* type *property of the vector object* MPI_DOUBLE_PRECISION.

```
      program demo
c     user access procedures must be declared external
      external initvec
      integer y, data, n, type
      double precision val
      ...
c     create vector object
      type = MPI_DOUBLE_PRECISION
      call DDL_Create_vector(n, type, block, comm, y, ierr)
      ...
      call DDL_Get_vector(y, data, ierr)
      call DDL_Size_vector(y, n, ierr)
      val = 0.0
c     call access procedure to initialize vector
      call DDL_Access(initvec, 1, data, 2, n, val, ierr)
      ...
      end

c     access raw data values of vectors
      subroutine initvec(ierr, data, n, val)
      integer n, ierr
      double precision data(n), val
      ...
      do i=1, n
         data(i) = val
      end do
      end
```

Example 8.5 *Initialise a distributed sparse matrix object to a diagonal matrix. The matrix is initially generated in triad format, which is usually the easiest format to generate. But notice that the matrix is then converted to row packed format to be consistent with the DDL* format *property for the object. Note also that the code updates the* used *property.*

```
      program demo
c     user access procedures must be declared external
      external initsparse
      integer sparse, ia, ja, a, type, ierr
      integer maxnelt, num_rows, offset
      ...
c     create DDL sparse matrix
      type = MPI_DOUBLE_PRECISION
      call DDL_Create_sparse(sizes, type, format, block, comm,
     &                       sparse, ierr)
      ...
c     update sparse matrix vectors directly
      call DDL_Get_ia(sparse, ia, ierr)
      call DDL_Get_ja(sparse, ja, ierr)
      call DDL_Get_a(sparse, a, ierr)
      call DDL_Size_sparse(sparse, sizes, ierr)
      maxnelt = sizes(2)
      num_rows = sizes(1)
      call DDL_Offset_sparse(sparse, offset, ierr)
c     call worker procedure to initialize sparse matrix
      call DDL_Access(initsp, 3, ia, ja, a, 5, maxnelt, num_rows,
     &                offset, type, sparse, ierr)
      ...
      end

c     access raw data values of sparse matrix
      subroutine initsparse(ierr, ia, ja, a, maxnelt, num_rows,
     &                      offset, type, sparse)
      integer maxnelt, ierr, num_rows, offset, type, sparse
      integer ia(maxnelt), ja(maxnelt)
      double precision a(maxnelt)
      ...
c     remember local indexing
c     row ordered triad format
c     just create diagonal matrix
      do i=1, num_rows
         ia(i) = i
         ja(i) = offset+i-1
         a(i) = offset+i-1
      end do
c     convert to row packed as specified at creation
      call DDL_Triadtorow(num_rows, num_rows, ia, ja, a, type, 0, ierr)
c     complete sparse object entries
      call DDL_Setused_sparse(sparse, num_rows, ierr)
      end
```

8.2.3 F90

F90 supports the pointer data type, so programs can access the arrays referenced by the DDL return pointers directly. F90 code dealing with return pointers is very similar to C code.

Example 8.6 *Initialise a distributed vector object to a value. Notice the agreement in type between the array pointer* `double precision, pointer, dimension(:) :: data` *and the* type *property of the vector object* MPI_DOUBLE_PRECISION.

```
      program demo
      integer y, n, type
      double precision val
      double precision, pointer, dimension(:) :: data
```

```
      ...
c     create vector object
      type = MPI_DOUBLE_PRECISION
      call DDL_Create_vector(n, type, block, comm, y, ierr)
      ...
c     initialise array
      call DDL_Get_vector(y, data, ierr)
      call DDL_Size_vector(y, n, ierr)
      val = 0.0
      do i=1, n
         data(i) = val
      end do
      end
```

Chapter 9

High-level operations

This chapter describes some high level routines to support the solution of sparse systems of linear equations. A complete DDL program using these routines and illustrating the use of the DDL is given in Appendix A.

9.1 Preconditioning for linear equation solvers

These routines are intended to be used in conjunction with iterative linear equation solvers as preconditioning techniques. They work by converting the original row packed sparse matrix into a factorized block diagonal matrix. During the iterative solve, this preconditioning matrix can be used to improve the performance of the solver.

Only FORTRAN callable versions of the procedures are provided.

9.1.1 ILU preconditioning

Incomplete LU preconditioning is supported by two routines DDL_Ilu_block and DDL_Ilu_solve.

DDL_Ilu_block forms the incomplete LU factorization of a block diagonal matrix. The input matrix, a, is in row-packed format. The output matrices, l and u, are in a special internal format (the SLAP column format) which is only understood by the ILU procedures. Since the factorization is incomplete no fill-ins will occur.

DDL_Ilu_block(a, l, u, dinv)

IN	a	matrix to factorize (handle)
OUT	l	factored lower triangular matrix (handle)
OUT	u	factored upper triangular matrix (handle)
OUT	dinv	vector for inverse diagonal of a (handle)

```
DDL_ILU_BLOCK(A, L, U, DINV, IERR)
   INTEGER A, L, U, DINV, IERR
```

DDL_Ilu_solve solves the equation $LUx = b$ where LU is the factorized block diagonal matrix formed by DDL_Ilu_block.

DDL_Ilu_solve(l, u, dinv, b, x)

IN	l	factored lower triangular matrix (handle)
IN	u	factored upper triangular matrix (handle)
IN	dinv	vector with inverse diagonal of unfactored matrix (handle)
IN	b	RHS vector (handle)
OUT	x	solution vector (handle)

```
DDL_ILU_SOLVE(L, U, DINV, B, X, IERR)
```

```
    INTEGER L, U, DINV, B, X, IERR
```

9.1.2 ILUnc preconditioning

Incomplete LU preconditioning is supported by two routines DDL_Ilunc_block and DDL_Ilunc_solve.

DDL_Ilunc_block forms the incomplete LU factorization of a block diagonal matrix. The input matrix, a, is in row-packed format. The output matrices, l and u, are in a special internal format (the SLAP column format) which is only understood by the ILU procedures. Since the factorization is incomplete no fill-ins will occur.

DDL_Ilunc_block(a, d, ivec, nc, expinv)

IN	a	matrix to factorize (handle)

```
DDL_ILUNC_BLOCK(A, D, IVEC, NC, EXPINV, IERR)
    INTEGER A, D, IVEC, NC, IERR
    LOGICAL EXPINV
```

DDL_Ilunc_solve solves the equation $LUx = b$ where LU is the factorized block diagonal matrix formed by DDL_Ilunc_block.

DDL_Ilunc_solve(a, d, ivec, x, b, nc, expinv)

IN	b	RHS vector (handle)
OUT	x	solution vector (handle)

```
DDL_ILUNC_SOLVE(A, D, IVEC, X, B, NC, EXPINV, IERR)
    INTEGER A, D, IVEC, NC, B, X, IERR
    LOGICAL EXPINV
```

Appendix A

Example DDL program

The following complete program gives an example of the use of some of the distributed sparse matrix routines. The source for this program is included as part of the sparse DDL distribution.

A.1 Description of the example program

A Transpose Free Quasi Minimal Residual algorithm is implemented in order to illustrate some of the features of the DDL library, with particular regard to sparse matrix support. TFQMR is a non-stationary iterative method which attempts to remedy the problem of irregular convergence behaviour associated with Bi Conjugate Gradient (BCG) and Conjugate Gradient Squared (CGS). The iterates of a TFQMR solve are characterized by a quasi minimization of the residual norm, thus leading to a smoother convergence curve. Unlike QMR, TFQMR (as the name suggests) requires no matrix vector multiplications with A^T, which makes it a more attractive algorithm, especially in a parallel environment. One more point of note is that, like CGS, TFQMR can be prone to breakdown under certain circumstances. Therefore, for a robust implementation of the method, look ahead steps should be incorporated into the algorithm. None of the Harwell-Boeing data sets tested produced breakdown, though some converged very slowly, when solved without preconditioners.

The two block preconditioning techniques introduced into this version of the demonstration enhance convergence rates considerably. These methods firstly form a block diagonal matrix by reducing the allocated rows of the original matrix to the block diagonal on each processor. An incomplete LU factorization then takes place on each block in order to form the preconditioning matrix. In the current demonstration, the program automatically allocates one block per processor. Future solvers will allow the user to specify the number of blocks independently of the number of processors required.

For a discussion of the theory behind TFQMR, and a listing of the algorithm see Freund's paper[1]. It can be seen that the source code reads exactly like a sequential fortran implementation, apart from the creation of distributed objects and calls to DDL rather than BLAS routines. The demonstration is provided with the COSI 001 test problem `test4001.hbf` (now in Harwell-Boeing format).

A.2 Listing

```
 1  c     $Id: tfqmrnc.f,v 1.2 1995/06/05 10:49:52 tim Exp tim $
 2
 3  c     MPI sparse DDL demo in FORTRAN (Andy's tfqmrnc)
 4
 5  c     Copyright 1994 IASC
 6
 7  c     DESCRIPTION   DDL based iterative solver for Ax=b.
 8  c                   Block preconditioning techniques are
 9  c                   incorporated into this version.
10
```

[1] R. W. Freund,*A Transpose-Free Quasi-Minimal Residual Algorithm for non-Hermitian Linear Systems*, SIAM J. Sci. Comput, **14**, No. 2, pp.470-482, 1993.

```
11        program tfqmrnc
12
13        implicit none
14        include 'mpif.h'
15        include 'ddlf.h'
16
17        character hbfilename*(100), ddlfilename*(100), datadir*(100)
18        character solnfilename*(100), summaryfilename*(100)
19        data hbfilename /'test4001.hbf\0'/
20        data datadir /'../parsim/data/'/
21        data ddlfilename /'tfqmrdata\0'/
22        data solnfilename /'soln\0'/
23        data summaryfilename /'summary\0'/
24
25  c     Workspace for ilu blocks & solves 9/1/95
26        integer maxiwork, maxrwork
27        parameter(maxiwork=1000000, maxrwork=1000000) ! local size
28        integer iwork(maxiwork)
29        double precision rwork(maxrwork)
30
31        integer i,n,iter,itmax,loop
32        double precision error,tol(10),gamma,delta,epsilon
33        double precision epsilon2
34        integer nelt, tolno, nc
35        logical rhsexist,ilu,ilunc,expinv
36        logical nopc
37        character*80 ans
38
39        integer infile, outfile
40        integer ierr
41        integer block, format
42        integer comm
43        integer a_ddl, x_ddl, b_ddl, xtrue_ddl, tmp_ddl
44        integer type, fformat, sizes(3), len, precon
45
46  c     initialise MPI, DDL
47        call MPI_Init(ierr)
48        call DDL_Init_sparse(MPI_COMM_WORLD, comm, ierr)
49
50        if (DDL_Ishost()) then
51          print *,' '
52          print *,'               TFQMR SPARSE DDL SOLVER'
53          print *,'              ************************'
54
55  c*********************************************************************
56  c*                                                                   *
57  c*                  READ ITERATION PARAMETERS                        
58  c*                                                                   *
59  c*********************************************************************
60
61          write(*,*) 'Input demonstration options'
62          write(*,*) '(for default [in brackets] just press RETURN)'
63          write(*,*) ' '
64
65          write(*,*) 'Input name of data file [test4001.hbf]'
66          read(*,'(a)') ans
67          if (ans .eq. ' ') then
68             hbfilename =
```

```
      +       datadir(1:index(datadir,' ')-1) // 'test4001.hbf'
         else
            hbfilename = datadir(1:index(datadir,' ')-1) // ans
         end if
c        form C file name string for DDL procedures
         len=index(hbfilename,' ')
         hbfilename(len:len) = '\0'

         print *,'Is the problem supplied with right hand'
         print *,'side and solution vectors y/n? [Y]'
         read(*,'(a)') ans
         if (ans .eq. ' ') then
            ans = 'y'
         end if
         if ((ans.eq.'Y').or.(ans.eq.'y')) then
            rhsexist = .TRUE.
         else
            rhsexist = .FALSE.
         end if
         print *,' '

         print *,'Enter Max. No. Iterations [100]'
         read(*,'(a)') ans
         if (ans .eq. ' ') then
            itmax = 100
         else
            read (ans, *) itmax
         end if

         print *,'Enter no. of tolerance levels required (max 10) [1]'
         read(*,'(a)') ans
         if (ans .eq. ' ') then
            tolno = 1
         else
            read (ans, *) tolno
         end if
         print *,' '

         print *,'Enter set of tolerances (smallest last)'

         do i = 1,tolno
            print *,'Enter tolerance ',i,' [1e-6]'
            read(*,'(a)') ans
            if (ans .eq. ' ') then
               tol(i) = 1e-6
            else
               read (ans, *) tol(i)
            end if
         end do
         print *,' '

         print *,'Enter e, where x_initial = x_true + e(rand(0,1))'
         print *,'(typically e should lie between 0.001 and 1.0):'
         print *,'An entry of zero gives an initial guess of zero [0]'
         read(*,'(a)') ans
         if (ans .eq. ' ') then
            epsilon = 0.0D0
         else
```

```
127            read (ans, *) epsilon
128         end if
129
130         print *,'Is block preconditioning to be applied [Y]'
131         read(*,'(a)') ans
132         if (ans .eq. ' ') then
133            ans = 'y'
134         end if
135         if ((ans.eq.'Y').or.(ans.eq.'y')) then
136   10       print *,'Choose the preconditioner'
137            print *,'type "ilu" for incomplete LU factorization'
138            print *,'     "inc" for incomplete LUnc factorization [INC]'
139            read(*,'(a)') ans
140            if (ans .eq. ' ') then
141               ans = 'inc'
142            end if
143            if ((ans.eq.'ILU').or.(ans.eq.'ilu')) then
144               ilu = .TRUE.
145               print *,' '
146               print *,'ILU block preconditioner to be applied'
147               print *,' '
148               ilunc = .FALSE.
149               nopc = .FALSE.
150            else if ((ans.eq.'INC').or.(ans.eq.'inc')) then
151               ilunc = .TRUE.
152               print *,' '
153               print *,'ILU(nc) block preconditioner to be applied'
154               print *,' '
155               print *,'Enter nc parameter [4]'
156               read(*,'(a)') ans
157               if (ans .eq. ' ') then
158                  nc = 4
159               else
160                  read (ans, *) nc
161               end if
162               print *,'Do you want to form explicit cell diagonals? [N]'
163               read(*,'(a)') ans
164               if (ans .eq. ' ') then
165                  ans = 'n'
166               end if
167               if ((ans.eq.'Y').or.(ans.eq.'y')) then
168                  expinv = .TRUE.
169               else
170                  expinv = .FALSE.
171               end if
172               ilu = .FALSE.
173               nopc = .FALSE.
174            else
175               print *,' '
176               print *,'*Unrecognizeable input* - try again y/n'
177               print *,' '
178               read (*,'(a)') ans
179               if ((ans.eq.'Y').or.(ans.eq.'y')) then
180                  go to 10
181               else
182                  print *,' '
183                  print *,'No Preconditioning applied'
184                  print *,' '
```

```
185                  nopc = .TRUE.
186                  ilu = .FALSE.
187                  ilunc = .FALSE.
188               end if
189            end if
190          else
191             print *,' '
192             print *,'No Preconditioning applied'
193             print *,' '
194             nopc = .TRUE.
195             ilu = .FALSE.
196             ilunc = .FALSE.
197          end if
198
199          print *, 'OK. Starting run...'
200
201          call createinput(hbfilename, ddlfilename, rhsexist)
202
203        end if ! DDL_Ishost
204
205  c     broadcast user input to all processes
206        call MPI_Bcast(itmax, 1, MPI_INTEGER, 0, comm, ierr)
207        call MPI_Bcast(rhsexist, 1, MPI_LOGICAL, 0, comm, ierr)
208        call MPI_Bcast(ilu, 1, MPI_LOGICAL, 0, comm, ierr)
209        call MPI_Bcast(ilunc, 1, MPI_LOGICAL, 0, comm, ierr)
210        call MPI_Bcast(nc, 1, MPI_INTEGER, 0, comm, ierr)
211        call MPI_Bcast(expinv, 1, MPI_LOGICAL, 0, comm, ierr)
212        call MPI_Bcast(nopc, 1, MPI_LOGICAL, 0, comm, ierr)
213        call MPI_Bcast(epsilon, 1, MPI_DOUBLE_PRECISION, 0, comm, ierr)
214        call MPI_Bcast(tolno, 1, MPI_INTEGER, 0, comm, ierr)
215        do i = 1, tolno
216           call MPI_Bcast(tol(i), 1, MPI_DOUBLE_PRECISION, 0, comm, ierr)
217        end do
218
219  c     open input and output files
220        call DDL_Open(infile, ddlfilename, DDL_READ_MODE, comm, ierr)
221        call DDL_Open(outfile, solnfilename, DDL_WRITE_MODE, comm, ierr)
222
223  c     read in the information heading
224        call DDL_Fileformat(infile, fformat, type, sizes, comm, ierr)
225        n = sizes(1)
226        nelt = sizes(2)
227        format = DDL_PCK_ROW
228        block = 4
229
230  c     set up size vector
231        sizes(1) = n
232        sizes(2) = n
233        sizes(3) = nelt*2
234
235  c     create sparse matrix
236        a_ddl = DDL_NULL
237        call DDL_Create_sparse(sizes, type, format,
238       &                       block, comm, a_ddl, ierr)
239
240  c     create vector for solution
241        x_ddl = DDL_NULL
242        call DDL_Create_vector(n, type, block,
```

```
243       &                       comm, x_ddl, ierr)
244   c     create vector for result
245         b_ddl = DDL_NULL
246         call DDL_Create_vector(n, type, block,
247       &                       comm, b_ddl, ierr)
248   c     create vector for true solution
249         xtrue_ddl = DDL_NULL
250         call DDL_Create_vector(n, type, block,
251       &                       comm, xtrue_ddl, ierr)
252   c     create temp vector
253         tmp_ddl = DDL_NULL
254         call DDL_Create_vector(n, type, block,
255       &                       comm, tmp_ddl, ierr)
256
257   c     read data from file to sparse object
258         call DDL_Read(a_ddl, infile, ierr)
259
260         if (.not. rhsexist) then
261   c        initialise xtrue to one
262            call DDL_Fill(xtrue_ddl, 1.0D0, ierr)
263            call DDL_Smv(a_ddl, xtrue_ddl, b_ddl, gamma, delta, ierr)
264         else
265   c        read rhs from file
266            call DDL_Read(b_ddl, infile, ierr)
267   c        read soln vector from file
268            call DDL_Read(xtrue_ddl, infile, ierr)
269         endif
270
271   c     initialise x
272         call DDL_Rand(1.0d0, tmp_ddl, ierr)
273         call DDL_Copy(xtrue_ddl, x_ddl, ierr)
274         if (epsilon.eq.0.0d0) then
275            call DDL_Scal(0.0d0, x_ddl, ierr)
276         else
277            call DDL_Axpy(epsilon, tmp_ddl, x_ddl, ierr)
278         end if
279
280         if (nopc) precon = 0
281         if (ilu) precon = 1
282         if (ilunc) precon = 2
283
284         call DDL_Linsolve(a_ddl, x_ddl, b_ddl, itmax, iter,
285       &     tol, tolno, precon, nc, expinv, error,
286       &     iwork, maxiwork, rwork, maxrwork, ierr)
287
288   c     Why have we stopped?
289         if (DDL_Ishost()) then
290            if(iter .lt. itmax) THEN
291               write (*,*)
292               write (*,'(A22,I4,A11)') 'TFQMR converged after',iter,
293       +           'iterations'
294               write (*,*)
295            else
296               write (*,*)
297               write (*,'(A37)') ' Specified Max. Iterations reached:'
298               write (*,'(A15,I4,A25,I4)') ' Max.iters = ',itmax,
299       +           'Iterations reached = ',iter
300               write (*,*)
```

```
301           end if
302         end if
303
304         call DDL_Write(x_ddl, outfile, ierr)
305
306         call DDL_Close(outfile, comm, ierr)
307         call DDL_Close(infile, comm, ierr)
308
309   c     check solution is correct
310         if (DDL_Ishost()) then
311            call check(ddlfilename, solnfilename, rhsexist, error)
312            print *, 'Max error is', error
313         end if
314
315   c     delete distributed objects
316         call DDL_Free(a_ddl, ierr)
317         call DDL_Free(x_ddl, ierr)
318         call DDL_Free(b_ddl, ierr)
319         call DDL_Free(xtrue_ddl, ierr)
320         call DDL_Free(tmp_ddl, ierr)
321
322   c     tidy up and exit
323         call DDL_Finalize_sparse(comm, ierr)
324         call MPI_Finalize(ierr)
325
326         stop
327         end
328
329
330   c     convert Harwell-Boeing file to row-packed triad file
331   c     HB file has matrix; and  rhs, soln if .rhsexist.
332   c     output matrix; rhs, soln if .rhsexist.
333         subroutine createinput(hbfilename, ddlfilename, rhsexist)
334
335         implicit none
336         include 'mpif.h'
337         include 'ddlf.h'
338
339         character*(*) hbfilename, ddlfilename
340         logical rhsexist
341
342         integer maxn, mxnelt, np
343         parameter (maxn = 5005, mxnelt = 40000, np = 10)
344
345         integer infile, outfile
346         integer ierr, format
347         integer type
348         integer ia(mxnelt), ja(mxnelt)
349         double precision a(mxnelt)
350         integer nelt, job
351         integer n, i, isym, size
352         double precision soln(maxn), rhs(maxn), xinit(maxn)
353
354   c     read in sparse matrix from file
355         job = 3
356         n = maxn
357         nelt = mxnelt
358   c     read in as column gathered
```

```
359        call DDL_Open_host(infile, hbfilename, DDL_READ_MODE, ierr)
360        call DDL_Inhb(infile, n, nelt, ia, ja, a,
361       &     isym, soln, rhs, xinit, job, ierr)
362        call DDL_Close_host(infile, ierr)
363        type = MPI_DOUBLE_PRECISION
364  c     convert to triad
365        call DDL_Coltotriad(n, nelt, ia, ja, a, type, isym, ierr)
366  c     convert triad to row
367        call DDL_Triadtorow(n, nelt, ia, ja, a, type, isym, ierr)
368  c     convert row to triad
369        call DDL_Rowtotriad(n, nelt, ia, ja, a, type, isym, ierr)
370  c     write out as row ordered triad format
371  c     this data file will be used for rest of demo
372        format = DDL_TRIAD_ROW
373        call DDL_Open_host(outfile, ddlfilename, DDL_WRITE_MODE, ierr)
374        call DDL_Out_sparse(outfile, n, nelt, ia, ja, a, type,
375       &     isym, format, ierr)
376
377        if (rhsexist) then
378          if (job .eq. 23 .or. job .eq. 31) then
379  c         write out rhs & sol
380            call DDL_Out_vector(outfile, rhs, MPI_DOUBLE_PRECISION,
381       &                        n, ierr)
382            call DDL_Out_vector(outfile, soln, MPI_DOUBLE_PRECISION,
383       &                        n, ierr)
384          else
385            print *, 'Error reading RHS and SOLN vectors'
386          end if
387        end if
388
389        call DDL_Close_host(outfile, ierr)
390        return
391        end
392
393
394  c     check result vector
395        subroutine check(ddlfilename, solnfilename, rhsexist, maxerr)
396
397        implicit none
398        include 'mpif.h'
399        include 'ddlf.h'
400
401        character*(*) solnfilename, ddlfilename
402        logical rhsexist
403        double precision maxerr
404
405        integer maxn, mxnelt, np
406        parameter (maxn = 5005, mxnelt = 40000, np = 10)
407
408        integer infile, outfile
409        integer ierr, format
410        integer type
411        integer n, i
412        double precision soln(maxn), correct(maxn), error
413
414  c     get calculated solution vector
415        call DDL_Open_host(infile, solnfilename, DDL_READ_MODE, ierr)
416        n = maxn
```

```
417         call DDL_In_vector(infile, soln, type, n, ierr)
418         call DDL_Close_host(infile, ierr)
419   c     get correct solution vector
420         if (rhsexist) then
421            call DDL_Open_host(infile, ddlfilename, DDL_READ_MODE, ierr)
422            call DDL_Skip_host(infile, ierr)
423            call DDL_Skip_host(infile, ierr)
424            n = maxn
425            call DDL_In_vector(infile, correct, type, n, ierr)
426            call DDL_Close_host(infile, ierr)
427         else
428   c        set to 1
429            do i=1, n
430               soln(i) = 1.0D0
431            end do
432         end if
433
434   c     check for differences
435         maxerr = 0.0D0
436         do i=1, n
437            error = ABS(ABS(soln(i))-ABS(correct(i)))
438            if (error .gt. maxerr) then
439      maxerr = error
440            end if
441         end do
442
443         return
444         end
445
446
447   c     DDL procedure to solve sparse linear system of equations
448         subroutine DDL_Linsolve(a_ddl, x_ddl, b_ddl, itmax, iter,
449        &     tol, tolno, precon, nc, expinv, error,
450        &     iwork, iwork_dim, rwork, rwork_dim, ierr)
451
452         implicit none
453         include 'mpif.h'
454         include 'ddlf.h'
455
456   c     DDL object handles
457         integer a_ddl, x_ddl, b_ddl
458
459         integer itmax, iter, tolno, precon, nc, ierr
460         double precision tol(tolno), error
461         logical expinv
462         integer iwork_dim
463         integer iwork(iwork_dim)
464         integer rwork_dim
465         double precision rwork(rwork_dim)
466
467   c     local variables
468   c     tfqmrnc variables
469         integer i,n,m
470         double precision alpha,theta,betan,
471        + tau,omega,rhon,rhonm1,const,xnrm,c,sigma,malpha,oldcgs,
472        + rnrm,rnrm0,eta,xtnrm,bnrm,gamma,delta
473         integer nelt, tolcount
474         logical ilu,ilunc
```

```
475
476  c     DDL/MPI
477        integer infile, outfile
478        integer block, format
479        integer comm
480        integer m_ddl, l_ddl, u_ddl, dinv_ddl
481        integer pip_ddl, rcgs_ddl, r0_ddl
482        integer p_ddl, v_ddl, d_ddl, tmp_ddl, q_ddl, w_ddl, z_ddl
483        integer uv_ddl, rvec_ddl, ivec_ddl
484        integer type, fformat, sizes(3), len
485        integer numblks, blocksize, lnelt, memory_int, memory_real
486
487  c     check preconditioning type
488        if (precon .eq. 0) then
489           ilu = .FALSE.
490           ilunc = .FALSE.
491        else if (precon .eq. 1) then
492           ilu = .TRUE.
493           ilunc = .FALSE.
494        else if (precon .eq. 2) then
495           ilu = .FALSE.
496           ilunc = .TRUE.
497        end if
498
499  c     find problem size
500        call DDL_Gsize(a_ddl, sizes, ierr)
501        n = sizes(1)
502        nelt = sizes(3)
503        call DDL_Type(a_ddl, type, ierr)
504        call DDL_Format(a_ddl, format, ierr)
505        call DDL_Comm(a_ddl, comm, ierr)
506        call DDL_Block(a_ddl, block, ierr)
507
508  c     check enough memory is allocated
509        if (ilunc) then
510           call DDL_Size(a_ddl, sizes, ierr)
511           blocksize = sizes(1)
512           call DDL_Used_sparse(a_ddl, lnelt)
513           numblks = blocksize/nc
514           memory_int = 10+nc+2*(numblks+blocksize)+ lnelt+blocksize+1
515           memory_real = 8*blocksize+nc*(blocksize+3*nc)+ lnelt
516           if (memory_int.gt.iwork_dim.or.memory_real.gt.rwork_dim) then
517              ierr=2
518              goto 1000
519           end if
520        end if
521
522        if (ilu) then
523           sizes(1) = n
524           sizes(2) = n
525           sizes(3) = nelt*2
526           l_ddl = DDL_NULL
527           call DDL_Create_sparse(sizes, type, format,
528       &                          block, comm, l_ddl, ierr)
529           u_ddl = DDL_NULL
530           call DDL_Create_sparse(sizes, type, format,
531       &                          block, comm, u_ddl, ierr)
532        end if
```

```
533
534       if (ilu) then
535          dinv_ddl = DDL_NULL
536          call DDL_Create_vector(n, type, block, comm, dinv_ddl, ierr)
537  c       factorize A to block preconditioner M
538          call DDL_Ilu_block(a_ddl, l_ddl, u_ddl, dinv_ddl,
539       &        iwork, iwork_dim, rwork, rwork_dim, ierr)
540       end if
541
542       if (ilunc) then
543  c       create integer vector for ilunc calls
544          ivec_ddl = DDL_NULL
545          call DDL_Create_vector(10*n, MPI_INTEGER, block,
546       &                         comm, ivec_ddl, ierr)
547          rvec_ddl = DDL_NULL
548          call DDL_Create_vector(10*n, MPI_DOUBLE_PRECISION, block,
549       &                         comm, rvec_ddl, ierr)
550  c       factorize A to block preconditioner ILUnc
551          call DDL_Ilunc_block(a_ddl, rvec_ddl, ivec_ddl, nc, expinv,
552       &        iwork, iwork_dim, rwork, rwork_dim, ierr)
553       end if
554
555  c     create vector for residual
556       rcgs_ddl = DDL_NULL
557       call DDL_Create_vector(n, type, block,
558       &                      comm, rcgs_ddl, ierr)
559  c     create vector for initial residual
560       r0_ddl = DDL_NULL
561       call DDL_Create_vector(n, type, block,
562       &                      comm, r0_ddl, ierr)
563  c     create vectors for tfqmr soln method
564       uv_ddl = DDL_NULL
565       call DDL_Create_vector(n, type, block,
566       &                      comm, uv_ddl, ierr)
567       p_ddl = DDL_NULL
568       call DDL_Create_vector(n, type, block,
569       &                      comm, p_ddl, ierr)
570       v_ddl = DDL_NULL
571       call DDL_Create_vector(n, type, block,
572       &                      comm, v_ddl, ierr)
573       d_ddl = DDL_NULL
574       call DDL_Create_vector(n, type, block,
575       &                      comm, d_ddl, ierr)
576       tmp_ddl = DDL_NULL
577       call DDL_Create_vector(n, type, block,
578       &                      comm, tmp_ddl, ierr)
579       q_ddl = DDL_NULL
580       call DDL_Create_vector(n, type, block,
581       &                      comm, q_ddl, ierr)
582       w_ddl = DDL_NULL
583       call DDL_Create_vector(n, type, block,
584       &                      comm, w_ddl, ierr)
585       if(ilu .or. ilunc) then
586          z_ddl = DDL_NULL
587          call DDL_Create_vector(n, type, block,
588       &                         comm, z_ddl, ierr)
589       end if
590
```

```
591        gamma = 1.0d0
592        delta = 0.0d0
593  c     calculate mv communications
594        call DDL_Smv_comm(a_ddl, ierr)
595
596        call DDL_Nrm2(b_ddl, bnrm, ierr)
597
598        call DDL_Copy(b_ddl, r0_ddl, ierr)
599        call DDL_Smv(a_ddl, x_ddl, r0_ddl, -1.0d0, gamma, ierr)
600
601        call DDL_Nrm2(x_ddl, xnrm, ierr)
602
603        if (ilu) then
604           call DDL_Copy(r0_ddl, tmp_ddl, ierr)
605           call DDL_Ilu_solve(l_ddl, u_ddl, dinv_ddl, tmp_ddl,
606       &                      r0_ddl, ierr)
607           call DDL_Copy(b_ddl, tmp_ddl, ierr)
608           call DDL_Ilu_solve(l_ddl, u_ddl, dinv_ddl, tmp_ddl,
609       &                      b_ddl, ierr)
610        else if (ilunc) then
611           call DDL_Copy(r0_ddl, tmp_ddl, ierr)
612           call DDL_Ilunc_solve(a_ddl, rvec_ddl, ivec_ddl, tmp_ddl,
613       &        r0_ddl, nc, expinv,
614       &        iwork, iwork_dim, rwork, rwork_dim, ierr)
615           call DDL_Copy(b_ddl, tmp_ddl, ierr)
616           call DDL_Ilunc_solve(a_ddl, rvec_ddl, ivec_ddl, tmp_ddl,
617       &        b_ddl, nc, expinv,
618       &        iwork, iwork_dim, rwork, rwork_dim, ierr)
619        end if
620
621        call DDL_Copy(r0_ddl, p_ddl, ierr)
622        call DDL_Copy(r0_ddl, uv_ddl, ierr)
623        call DDL_Copy(r0_ddl, rcgs_ddl, ierr)
624        call DDL_Scal(0.0d0, d_ddl, ierr)
625
626  c     initialise v to Ap
627        call DDL_Smv(a_ddl, p_ddl, v_ddl, gamma, delta, ierr)
628
629        call DDL_Nrm2(x_ddl, xnrm, ierr)
630        call DDL_Nrm2(rcgs_ddl, rnrm, ierr)
631        call DDL_Nrm2(b_ddl, bnrm, ierr)
632
633        rnrm0 = rnrm
634        omega = rnrm
635        tau = rnrm
636        theta = 0.0d0
637        eta = 0.0d0
638        iter = 0
639        tolcount = 1
640        error = rnrm/bnrm
641        rhonm1 = 1.0d0
642
643        call DDL_Dot(r0_ddl, rcgs_ddl, rhonm1, ierr)
644
645  C*****************************************************************
646  C*****************************************************************
647  C
648  C                  START ITERATIVE LOOP
```

```
C
C*****************************************************************
C*****************************************************************


      do while ((error.GE.(tol(tolno))).AND.(iter.LT.itmax))
        iter = iter + 1


C*****************************************************************
C
C                     SECTION (c)
C
C*****************************************************************

         call DDL_Dot(r0_ddl, rcgs_ddl, rhon, ierr)

         betan = rhon/rhonm1

         if (iter.eq.1) then
            call DDL_Copy(rcgs_ddl, uv_ddl, ierr)
            call DDL_Copy(rcgs_ddl, p_ddl, ierr)
         else
            call DDL_Copy(rcgs_ddl, uv_ddl, ierr)
            call DDL_Axpy(betan, q_ddl, uv_ddl, ierr)
            call DDL_Copy(q_ddl, tmp_ddl, ierr)
            call DDL_Axpy(betan, p_ddl, tmp_ddl, ierr)
            call DDL_Copy(uv_ddl, p_ddl, ierr)
            call DDL_Axpy(betan, tmp_ddl, p_ddl, ierr)
         end if

         call DDL_Smv(a_ddl, p_ddl, v_ddl, gamma, delta, ierr)

         if (ilu) then
            call DDL_Copy(v_ddl, tmp_ddl, ierr)
            call DDL_Ilu_solve(l_ddl, u_ddl, dinv_ddl, tmp_ddl,
     &                         v_ddl, ierr)
         else if (ilunc) then
            call DDL_Copy(v_ddl, tmp_ddl, ierr)
         call DDL_Ilunc_solve(a_ddl, rvec_ddl, ivec_ddl, tmp_ddl,
     &           v_ddl, nc, expinv,
     &           iwork, iwork_dim, rwork, rwork_dim, ierr)
         end if

C*****************************************************************
C
C                     SECTION (a)
C
C*****************************************************************

         call DDL_Dot(r0_ddl, v_ddl, sigma, ierr)
         alpha = rhon/sigma
         malpha = -alpha
         oldcgs = rnrm
         call DDL_Copy(uv_ddl, q_ddl, ierr)
         call DDL_Axpy(malpha, v_ddl, q_ddl, ierr)
         call DDL_Copy(uv_ddl, w_ddl, ierr)
         call DDL_Axpy(1.0d0, q_ddl, w_ddl, ierr)
```

```
707           if (ilu) then
708              call DDL_Smv(a_ddl, w_ddl, z_ddl, gamma, delta, ierr)
709              call DDL_Copy(z_ddl, tmp_ddl, ierr)
710              call DDL_Ilu_solve(l_ddl, u_ddl, dinv_ddl, tmp_ddl,
711       &                         z_ddl, ierr)
712              call DDL_Axpy(malpha, z_ddl, rcgs_ddl, ierr)
713           else if (ilunc) then
714              call DDL_Smv(a_ddl, w_ddl, z_ddl, gamma, delta, ierr)
715              call DDL_Copy(z_ddl, tmp_ddl, ierr)
716              call DDL_Ilunc_solve(a_ddl, rvec_ddl, ivec_ddl, tmp_ddl,
717       &           z_ddl, nc, expinv,
718       &           iwork, iwork_dim, rwork, rwork_dim, ierr)
719              call DDL_Axpy(malpha, z_ddl, rcgs_ddl, ierr)
720           else
721              call DDL_Smv(a_ddl, w_ddl, tmp_ddl, gamma, delta, ierr)
722              call DDL_Axpy(malpha, tmp_ddl, rcgs_ddl, ierr)
723           end if
724
725     C*****************************************************************
726     C
727     C                     SECTION (b)
728     C
729     C*****************************************************************
730
731           call DDL_Nrm2(rcgs_ddl, rnrm, ierr)
732           omega = dsqrt(oldcgs*rnrm)
733           m = 2*iter-1
734           const = theta*theta*eta/alpha
735           theta = omega/tau
736           c = 1.0d0/(dsqrt(1.0d0+theta*theta))
737           tau = tau*theta*c
738           eta = c*c*alpha
739           call DDL_Scal(const, d_ddl, ierr)
740           call DDL_Axpy(1.0d0, uv_ddl, d_ddl, ierr)
741           call DDL_Axpy(eta, d_ddl, x_ddl, ierr)
742           error = dsqrt(dble(m+1))*dabs(tau)/bnrm
743           omega = rnrm
744           m = 2*iter
745           const = theta*theta*eta/alpha
746           theta = omega/tau
747           c = 1.0d0/(dsqrt(1.0d0+theta*theta))
748           tau = tau*theta*c
749           eta = c*c*alpha
750           call DDL_Scal(const, d_ddl, ierr)
751           call DDL_Axpy(1.0d0, q_ddl, d_ddl, ierr)
752           call DDL_Axpy(eta, d_ddl, x_ddl, ierr)
753           error = dsqrt(dble(m+1))*dabs(tau)/bnrm
754           call DDL_Nrm2(x_ddl, xnrm, ierr)
755
756           if (error.LT.(tol(tolcount))) then
757              tolcount = tolcount + 1
758           end if
759
760           rhonm1 = rhon
761        end do
762
763     c     remove distributed objects
764           if (ilu) then
```

```
765           call DDL_Free(l_ddl, ierr)
766           call DDL_Free(u_ddl, ierr)
767           call DDL_Free(dinv_ddl, ierr)
768           call DDL_Free(z_ddl, ierr)
769        end if
770        if (ilunc) then
771           call DDL_Free(ivec_ddl, ierr)
772           call DDL_Free(rvec_ddl, ierr)
773           call DDL_Free(z_ddl, ierr)
774        end if
775        call DDL_Free(rcgs_ddl, ierr)
776        call DDL_Free(r0_ddl, ierr)
777        call DDL_Free(uv_ddl, ierr)
778        call DDL_Free(p_ddl, ierr)
779        call DDL_Free(v_ddl, ierr)
780        call DDL_Free(d_ddl, ierr)
781        call DDL_Free(tmp_ddl, ierr)
782        call DDL_Free(q_ddl, ierr)
783        call DDL_Free(w_ddl, ierr)
784   1000 end
```

Appendix B

Error handling

B.1 The error mechanism

When a Library procedure detects an error several pieces of information are available:

error number (integer, global)
: Each different type of error that the Library recognizes is given a unique number. This error number is returned to the user as the value ierr. In FORTRAN programs ierr is the last parameter in the Library procedure call. In C programs ierr is the result of the Library function call. To aid the programmer in using this value, a set of integer constants are defined, one for each different error number from the error mechanism. In C the values are constants, and in FORTRAN they are parameters. (See Table B.1 for a list of error values.)

error severity (integer, global)
: Each type of error is also given a severity to reflect how serious the error is. This value is not currently used.

procedure (integer, global)
: The Library procedure in which the error occurred is represented by an integer. To aid the programmer in using this value, a set of integer constants are defined by the Library. The value of each integer constant is the value that the error mechanism gives to the corresponding procedure. In C the values are constants with names such as DDL_CREATE_SPARSE, and in FORTRAN they are parameters with names such as DDL_PERR_CREATE_SPARSE.

By default, on encountering an error a Library procedure exits immediately printing an error message giving the error number and description along with the procedure in which the error occurred. However, if the user desires, he can change the way in which the Library responds to errors by providing his own error handling procedure.

The error handling mechanism works as follows. When a Library procedure detects an error it calls the error handler passing three parameters: the error number, the error severity, and the procedure name. These parameters are all integers and are described above. The error handler must then decide how to process the error information. Typically, the error handler may print an error message. On exit, the error handler must return an error condition to the calling procedure which is used to control subsequent execution of that procedure. The returned error conditions fall into three categories:

ierr > 0 The calling Library procedure is to exit immediately returning ierr as the error value. Typically, the return error condition will be the same as the error number passed to the error handler (which is always positive) and thus the effect is to exit the procedure returning the original error number to the user. This is the behaviour of the default error handler.

ierr $= 0$ This return value is the same as DDL_SUCCESS. The calling Library procedure is to continue execution as if no error occurred. This return value might be useful if an error only represents a warning and will not cause problems later in the procedure.

ierr < 0 The calling Library procedure is to take corrective action and then continue execution as if no error occurred. The corrective action is specified by the value of ierr and the possible actions are given in the Library procedure documentation.

The error handler may respond differently to each error number in each Library procedure if required. The default error handler responds to all errors by returning the error number passed to it, i.e., ierr > 0, thus causing the Library procedure to exit. All the errors currently encountered by the Library require this response. Returning DDL_SUCCESS

to ignore an error will probably cause a Library procedure to crash ungracefully at a later stage. Also, none of the existing procedures provide corrective actions for any errors, so returning ierr < 0 will probably result either in a crash or with the procedure exiting with an unusual error condition which may confuse the calling program.

Thus, currently, the only reason for a user to provide his own error handler is to change the printed error information. However, as the Library is enhanced these features of the error mechanism may become useful.

B.2 User error handlers

A user's error handler may be written in C or FORTRAN and must have a fixed calling sequence.

For a C error handler:

```
#include "ddl.h"

int F77(ddl_error)(int *err_no, int *err_severity, int *ddl_routine)
{
  int ierr;
  /* optional: display error information */
  ierr = F77(ddl_print_error)(err_no, err_severity, ddl_routine);

  /* an error value must be returned */
  /* for example: return error number to force termination */
  ierr = *err_no;

  return ierr;
}
```

For a FORTRAN error handler:

```
      subroutine ddl_error(err_no, err_severity, ddl_routine)
      integer err_no, err_severity, ddl_routine
      integer ierr

      include "mpif.h"
      include "ddlf.h"

c     optional: display error information
      call DDL_Print_error(err_no, err_severity, ddl_routine)

c     an error value must be returned
c     for example: return error number to force termination
      ierr = err_no

      return ierr

      end
```

In order to link in the user's error handler instead of the default error handler two methods may be used:

1. for simple user programs, include the error handler source code in the user's single source file, or,
2. place the error handler source code in a separate file and link the corresponding object file before the standard DDL library files.

DDL_ERR_SUCCESS	No error
DDL_ERR	Unspecified error
DDL_ERR_UNALLOCATED	Unallocated object
DDL_ERR_MEMORY	Out of memory
DDL_ERR_SIZE	Invalid SIZE parameter
DDL_ERR_TYPE	Invalid TYPE parameter
DDL_ERR_FORMAT	Invalid FORMAT parameter
DDL_ERR_BLOCK	Invalid BLOCK parameter
DDL_ERR_COMMTOP	Invalid communicator topology
DDL_ERR_COMMDIM	Invalid communicator dimension
DDL_ERR_USED	Object in use
DDL_ERR_FILE	Invalid file handle
DDL_ERR_NOTIMPLEMENTED	Operation not implemented
DDL_ERR_TOOSMALL	Object too small to hold data
DDL_ERR_FILEFORMAT	Invalid file format
DDL_ERR_ISIZE	Incompatible object size
DDL_ERR_ITYPE	Incompatible object type
DDL_ERR_IFORMAT	Incompatible object format
DDL_ERR_ICOMM	Incompatible object communicator
DDL_ERR_DDLTYPE	Invalid object type
DDL_ERR_TOOBIG	Data space too small for object data
DDL_ERR_IOBJ	Incompatible objects
DDL_ERR_OVER	Invalid overlap
DDL_ERR_DIST	Not all processes used by distribution
DDL_ERR_IPROPTAG	Illegal property tag
DDL_ERR_NOTEXIST	Not exist
DDL_ERR_NOCOMMTAG	No COMM tag

Table B.1: Error values

Bibliography

[1] Message Passing Interface Forum. Draft document for a standard message-passing interface. 10 May 1993.

[2] Tim Oliver. *Sparse DDL version 2.1: Technical Guide*. Institute for Advanced Scientific Computation, University of Liverpool, July 1995.

Index

DDL_Access, 33
DDL_Axpy, 19

DDL_Close, 25
DDL_Close_host, 27
DDL_Coltotriad, 22
DDL_Comm, 14
DDL_Copy, 17
DDL_Create_sparse, 13
DDL_Create_vector, 12

DDL_DDLtype, 13
DDL_Dot, 18

DDL_Fileformat, 25
DDL_Fileformat_host, 27
DDL_Fill, 19
DDL_Format(object, format), 14
DDL_Free, 13

DDL_Get_a, 31
DDL_Get_ia, 30
DDL_Get_ja, 31
DDL_Get_vector, 30
DDL_Gsize, 14
DDL_Gused_sparse, 15

DDL_Ilu_block, 36
DDL_Ilu_solve, 37
DDL_Ilunc_block, 37
DDL_Ilunc_solve, 37
DDL_In_sparse, 27
DDL_In_vector, 28
DDL_Inhb, 29

DDL_Nrm2, 18

DDL_Offset, 15
DDL_Open, 25
DDL_Open_host, 26
DDL_Out_sparse, 28
DDL_Out_vector, 28

DDL_Rand, 19
DDL_Read, 26
DDL_Rowtotriad, 22

DDL_Scal, 18
DDL_Setformat_sparse, 16
DDL_Setperm_sparse, 17
DDL_Setsubtype_sparse, 16
DDL_Setused_sparse, 16
DDL_Size, 15
DDL_Smv, 20
DDL_Smv_comm, 21
DDL_Subtype_sparse, 16

DDL_Triadtocol, 23
DDL_Triadtorow, 23
DDL_Trisolve, 21
DDL_Trisolve_comm, 21
DDL_Type, 14

DDL_Used_sparse, 15

DDL_Write, 26