

Included below is an initial attempt at an MPI binding for the DRI. I've tried to put in what was decided at the last meeting but no doubt I've left out things. In addition, I've:

- * made names longer (e.g., gdo => global_data, part => partition, dist => distribution)

- * introduced iterators as a way to access indices of a distributed data structure (and gotten rid of the partition object as the way to determine which data a process owns)

- * used the partition object as the way to specify partition information for each dimension of a distribution

- * used an MPI-like approach for the communication calls

- * introduced predefined block and block cyclic distributions, and predefined packed layout

- * attempted to make it easy to do simple redistributions (e.g., corner turns)

In writing this "binding", I've taken the approach that the DRI interface would sit side-by-side with the MPI implementation and work in conjunction with the MPI implementation. For example, instead of using DRI_Dataspec, I used MPI_Datatypes.

The first included file is the corner turn example that James put out a while back. The second file is the proposal for an MPI binding for DRI.

--

Nathan Doss nathan.e.doss@lmco.com

=====
= Example
=====

```
/* Corner-turn example with MPI binding of data re-org interface */
/* Take a matrix distributed by rows and change it to being distributed
*/
/* by columns */
```

```
#include "mpi.h"
#include "dri.h"
```

```
int main(int argc, char **argv) {
```

```
/* Variable declarations */
DRI_Global_data gd;
float matrix1[1024][1024];
float matrix2[1024][1024];
int dims[] = {1024, 1024};
DRI_Partition part1[] = {DRI_PARTITION_INDIVISIBLE,
```

```

DRI_PARTITION_BLOCK};
DRI_Partition part2[] = {DRI_PARTITION_BLOCK,
DRI_PARTITION_INDIVISIBLE};
DRI_Group group;
DRI_Distribution dist1, dist2;

/* Initialization */
MPI_Init(&argc, &argv);
DRI_Init(&argc, &argv);

/* Create the global data & group */
DRI_Global_data_create(2, dims, MPI_FLOAT, &gd);
DRI_Group_create_from_commm(MPI_COMM_WORLD, &group);

/* Create the distributions */
DRI_Distribution_create(gd, group, part1, DRI_LAYOUT_PACKED,
&dist1);
DRI_Distribution_create(gd, group, part2, DRI_LAYOUT_PACKED,
&dist2);

/* Perform the redistribution */
DRI_Sendrecv(matrix1, 0, dist1, matrix2, 0, dist2);

/* Termination */
DRI_Finalize();
MPI_Finalize();
}

```

```

=====
= Proposal for MPI binding of DRI
=====

```

Table of Contents

```

=====

```

I. Introduction

A. Goals

B. Overview

C. General Guidelines

II. API

A. Environmental Management

1. Startup and Termination

Initialization

Termination

Querying status of initialization

Querying status of finalization

2. Profiling Interface

B. The Global Data Object

1. Data specification
2. Global Data Object
Constructors
Destructors
Accessors

C. Process Groups
Constructors
Destructors
Accessors

D. Partitions
Constructors
Destructors
Predefined partitions
Accessors

E. Memory Layout
Constructors
Destructors
Accessors

F. Distribution
Constructors
Destructors
Accessors

G. Communication

Appendix A. Constants

I. Introduction

=====

This document presents a proposal for a realization of an MPI binding of the Data Re-organization Interface (DRI). This proposal attempts to bind the guidelines and recommendations of the Data Re-organization committee to the Message Passing Interface (MPI) -- i.e., an MPI binding of the DRI API.

I.A Goals

In a parallel application, data locality is extremely important. The correct association of data with a particular processor at the right step in an algorithm has a direct impact on performance. Many parallel algorithms require extensive data reorganization between steps of the algorithm. One of the most classic forms is the "corner turn" reorganization of a 2-D array for the processing of SAR imagery. However, there is no well-defined API to implement this and other complex data reorganization methods. The Data Re-organization effort attempts to provide standard APIs that address data reorganization. The existence of such APIs allows the application programmer to concentrate on more important issues involved with complex numerical algorithms with the assurance the data reorganization is optimized for

the application platform.

Since this specific realization of the Data Re-organization API is based on MPI, we try to allow similar types of operations as one would have access to in MPI. Examples of this include:

- * No need for a transfer setup with synchronization between groups of processors before the transfer can be initiated.
- * Late binding of communication buffers
- * Blocking and non-blocking communication
- * Persistent communication
- * Communication modes: standard, synchronous, ready, buffered
- * Use of communicators/contexts to scope communication operations

I.B Overview

In order to specify a data reorganization operation, one must first specify the size, shape, and distribution of data over the sending group as well as the receiving group. The reorganization operation then is responsible for transporting and reorganizing the data from the sending distribution/layout of data to the receiving distribution/layout. The data reorganization API consists of routines that support the following:

- * Environment management - this part of the standard deals with issues such as error handling, startup and termination of the DRI environment, and querying the status of the DRI environment.
- * Global data - describes the shape and type of data that is to be distributed (e.g., a 2 row by 2 column matrix of integers).
- * Process group - is a list of processors over which the global data object will be distributed. It also includes the scope over which the communication will be performed (i.e., an MPI communicator)
- * Partition - is a description of how data is partitioned in one dimension. The different types of partitions include block, block cyclic, block with overlap, indivisible, etc.
- * Distribution - the combination of the global data object, the process group over which it's distributed, and partitions for each dimension. It's possible to infer a virtual process topology by looking at the distribution object.
- * Layout - describes, through the use of strides for each dimension, how data is physically laid out in processor memory.
- * Communication - the calls that either perform or assist in performing actual data movement

I.C General Guidelines

- Most DRI objects are read-only.
- If needed, DRI objects can be freed once passed into another DRI call. The semantics are that a DRI call, if it will need to reference an object that was passed into it, copies the object. Implementation note: Implementations do not necessarily need to copy the object but instead can use reference counts. For example:

```
DRI_Partition_create_block(..., &part[0]);  
DRI_Distribution_create(..., part, &dist);  
DRI_Partition_free(&part[0]);  
DRI_Sendrecv(..., dist, ...);
```

This corresponds to the approach used by MPI.

- MPI objects passed into DRI must not be freed until they are no longer used by the DRI library. This is necessary since the DRI library has no mechanism for incrementing the reference count of the MPI object.

(Discussion: Do accessors that access DRI objects actually create new copies of these objects (e.g., like MPI_Comm_group does in MPI)?

II. API

=====

The following sections describe the MPI binding of the DRI API.

(Implementation: Almost all DRI objects should be implemented as handles -- i.e., you should use reference counts so you know when the memory associated with the handles can be freed)

II.A Environmental Management

II.A.1 Startup and Termination

This document does not specify what the user must do to start DRI processes. The startup mechanism is implementation dependent (due to its dependence on MPI, it will likely be the same mechanism as used to start up MPI programs).

* Initialization

DRI implementations may require that some setup occur before DRI routines are called. To provide for this, DRI includes an initialization routine that must be called once (and only once) before other DRI or MPI calls are used (other than DRI_Initialized and MPI_Initialized).

```
int DRI_Init(int *argc, char ***argv);
```

The application programmer must call MPI_Init before DRI_Init is called. The DRI_Init is a collective call over all processes in MPI_COMM_WORLD (where "collective call" is defined as in the MPI-1 specification).

* Termination

This routine cleans up all DRI state. Once this routine is called, no other DRI routines may be called. DRI_Finalize must be called before MPI_Finalize.

```
int DRI_Finalize();
```

The DRI_Finalize is a collective call over all processes in MPI_COMM_WORLD (where "collective call" is defined as in the MPI-1 specification).

* Querying status of initialization

```
int DRI_Initialized(int *flag);
```

The flag is used to determine whether DRI_Init has been previously called. DRI_Initialized is the only routine that may be called before DRI_Init is called. The value returned to flag will be 1 if DRI was previously initialized, 0 otherwise.

* Querying status of finalization

```
int DRI_Finalized(int *flag);
```

The flag is used to determine whether DRI_Finalize has been previously called. DRI_Finalized is the only routine that may be called after DRI_Finalize is called. The value returned to flag will be 1 if DRI was previously finalized, 0 otherwise.

II.A.2 Profiling Interface

The DRI library contains a profiling interface using the same approach as the MPI library (see Chapter 8 of the MPI document).

* Profiling control

The DRI_PCONTROL routine allows the user to control the profiler dynamically at run time. Its purpose corresponds to the MPI_PCONTROL function found in the MPI specification.

```
int DRI_Pcontrol(const int level, ...);
```

II.A.3 Error Handling

DRI does not provide mechanisms for dealing with failures in the communication system. It is the job of the DRI implementor to insulate the user from unreliable hardware.

Two types of errors may occur in a DRI program:

- * program errors - when a DRI call is called with an incorrect argument that can be detected

- * resource error - when a program exceeds the amount of available system resources

The implementation is required to catch some errors, but not all. If the programmer introduces an error into a program and the implementation is not required to catch the error, that program is said to be erroneous.

Each DRI function returns an error value. DRI_SUCCESS is returned if the function executes correctly. Error codes that functions may return:

DRI_SUCCESS

DRI_ERR_ARG - argument passed into function was bad

DRI_ERR_UNKNOWN - we don't know what's wrong

DRI_ERR_OTHER - we don't have an error code for this error

<add others as necessary>

II.B The Global Data Object

The DRI_Global_data object provides a description of the cartesian data that will be distributed. The key pieces of information are the size and shape of the global data as well as the type of data (e.g., int, float).

II.B.1 Data specification

MPI's MPI_Datatype is used to describe the type of data in a global data object.

The predefined values for MPI_Datatype and their association with C datatypes include:

MPI Datatype C datatype

MPI_CHAR signed char

MPI_SHORT signed short int

MPI_LONG signed int

MPI_UNSIGNED_CHAR signed long int

MPI_UNSIGNED_SHORT unsigned char

MPI_UNSIGNED unsigned short int

MPI_UNSIGNED_LONG unsigned int

MPI_FLOAT unsigned long int

MPI_DOUBLE float
MPI_LONG_DOUBLE double
MPI_BYTE long double

It's also possible for users to specify the type of data by using (committed) MPI derived datatypes. If MPI derived types are used, the application should not free the derived datatype until it is no longer needed by the DRI portion of the application

(Rationale: This is required since the DRI library has no way to increment the reference count to the datatype)

II.B.2 Global Data Object

The following functions are used to manipulate, access, and destroy global data objects.

* Constructors

DRI_Global_data_create is used to create a new DRI_Global_data object. This call is a local call.

```
int DRI_Global_data_create(int ndims, int dimsizes[],  
MPI_Datatype datatype,  
DRI_Global_data *gdh);
```

ndims - number of dimensions (e.g., 2 for a matrix)
dimsizes - size of each dimension (e.g., [2,3] for a 2x3 matrix)
datatype - type of elements in the gdo
gdh - handle for a newly created global data object

* Destructors

This frees a user's handle to a global data object. After the call, the output global data object will point to DRI_GLOBAL_DATA_NULL.

```
int DRI_Global_data_free(DRI_Global_data *gd);
```

* Accessors

The following are various accessors used to determine information about a specific global data object.

```
int DRI_Global_data_get_ndims(DRI_Global_data gdoh, int *ndims);
```

ndims - number of dimensions

```
int DRI_Global_data_get_dim(DRI_Global_data gdoh, int dim, int  
*dimsizes);
```

dim - dimension you wish to know the size of
dimsizes - size of the 'dim' dimension

```
int DRI_Global_data_get_dims(DRI_Global_data gdoh, int dimsizes[]);
```

dimsizes - array with sizes of each dimension

```
int DRI_Global_data_get_datatype(DRI_Global_data gdoh,  
MPI_Datatype *datatype);
```

datatype - type of each element of the global data object

II.C Process Groups

Process groups provide information about which processes a global data object is distributed across.

* Constructors

The following function creates a one-dimensional ordered set of processes.

```
int DRI_Group_create(int nprocs, int procs[], MPI_Comm comm,  
DRI_Group *group)
```

nprocs - number of processors in the group
procs - list of processor ranks in the group
comm - MPI intra-communicator over which the group is scoped.

The following function takes constructs a DRI group with all from an MPI communicator. The resulting group contains all processes originally found in the MPI communicator.

```
int DRI_Group_create_from_comm(MPI_Comm comm, DRI_Group *group)
```

comm - MPI intra-communicator containing the processing which will be in the resulting group. The comm is also used to scope all communications over the group.

For two groups to be involved in a communication operation, their scopes must be equivalent (i.e., same processors, same MPI context).

It is erroneous for the application programmer to free a communicator that is being used by a DRI_Group.

* Destructors

The following function destroys a DRI_Group. After the call, the output group will point to DRI_GROUP_NULL.

```
int DRI_Group_free(DRI_Group *group);
```

* Accessors

The following functions are used to access information about a DRI_Group.

```
int DRI_Group_get_rank(DRI_Group group, int *rank);
```

rank - rank of the current processor in the given group

If the calling processor is not a member of the group, DRI_UNDEFINED is returned as the value of rank.

```
int DRI_Group_get_size(DRI_Group group, int *size);
```

size - number of processors in the given group

```
int DRI_Group_get_comm(DRI_Group group, MPI_Comm *comm);
```

comm - communicator associated with the group

II.D Partitions

Partitions describe how data is partitioned in a single dimension. There are two basic types of partitions that can be created by the user: block and block cyclic. There are also 3 predefined partition objects.

* Constructors

The following routine is used to create a block partition.

```
int DRI_Partition_create_block_complex (int procs, int modulo,
int minimum,
int left_overlap,
int right_overlap, int pad,
DRI_Partition *partition);
```

procs - number of "chunks" over which the dimension is to be blocked

modulo - number of data elements that must be kept together when performing the distribution (i.e., a modulo of 128 would mean that data is allocated to each processor in 128 element chunks).

(Discussion: What if 128 does not divide evenly into the number of elements? Who gets the leftovers?)

By using MPI derived types, specifically MPI_Type_contig, you can get the same effect without using Modulo)

minimum - minimum number of elements that must be assigned to each processor

(Discussion: What if minimum X #procs < total data size?)

Which processor would get less than the minimum?)

left_overlap - overlap on the "left" side of the data

right_overlap - overlap on the "right" side of the data

pad - flag that indicates whether padding should be used

partition - output partition object

The value of "procs" can be specified as 0 in which case the implementation will determine the value for this parameter. The implementation determines the value for procs when the partition object is used to build a distribution (see next section). If the same partition object is used to build two different distribution objects, the value chosen for "procs" by the DRI implementation may be different in the two distributions.

If pad is 0, data is always packed together (i.e., there are no "holes" in the data). If pad is 1 (and overlap > 0), space will be left for overlap even if no overlapping data is available.

The following routine is used to create a simple block partition and can be thought of as a complex block distribution with defaults for some of the parameters.

```
int DRI_Partition_create_block(int procs, DRI_Partition *partition);
```

procs - number of "chunks" over which the dimension is to be blocked

partition - output partition object

Just as with DRI_Partition_create_block_complex, the value of "procs" can be specified as 0.

This function is equivalent to calling DRI_Partition_create_block_complex with modulo = 1, minimum = 1, left and right overlap = 0, and pad = 0.

The next function creates a block cyclic partition.

```
int DRI_Partition_create_block_cyclic_complex(int procs, int block_size, int left_overlap, int right_overlap, int pad, DRI_Partition *partition);
```

procs - number of processors over which the data will be cycled

block_size - size of each "card" or block that is handed to each processor during each cycle

left_overlap - overlap on the "left" side of each block

right_overlap - overlap on the "right" side of each block

pad - flag that indicates whether padding should be used

partition - output partition object

The value of "procs" can be specified as 0 in which case the implementation will determine the value for this parameter. The implementation determines the value for procs when the partition object is used to build a distribution (see next section). If the same partition object is used to build two different distribution objects, the value chosen for "procs" by the DRI implementation may be different in the two distributions.

If pad is 0, data is always packed together (i.e., there are no "holes" in the data). If pad is 1 (and overlap > 0), space will be left for overlap even if no overlapping data is available.

The next function creates a simple block cyclic partition.

```
int DRI_Partition_create_block_cyclic(int procs, int block_size,
DRI_Partition *partition);
```

procs - number of processors over which the data will be cycled

block_size - size of each "card" or block that is handed to each processor during each cycle

partition - output partition object

Just as with DRI_Partition_create_block_cyclic_complex, the value of "procs" can be specified as 0.

This function is equivalent to calling DRI_Partition_create_block_cyclic_complex with left and right overlap = 0 and pad = 0.

(Discussion: What should happen when data size is not evenly divisible by the block size?)

* Destructors

The following function destroys a DRI_Partition. After the call, the output partition will point to DRI_PARTITION_NULL.

```
int DRI_Partition_free(DRI_Partition *partition);
```

* Predefined partitions

There are 3 predefined partition objects:

- DRI_PARTITION_BLOCK - a partition object that specifies that data is to be partitioned in block fashion. The number of processors over which data is to be partitioned is left to the implementation (procs = 0), right and left overlap is 0, modulo is 1, and the minimum is 1.

- DRI_PARTITION_CYCLIC - a partition object that specifies that data is to be partitioned in cyclic fashion. The number of processors data is to be cycled to is left to the implementation. This is equivalent to a user created block cyclic distribution with the block size equal to 1, the number of processors left undefined, and a right and left overlap of 0.

- DRI_PARTITION_INDIVISIBLE - this partition indicates that data is not to be partitioned in the dimension to which it refers. This is a DRI_PARTITION_BLOCK partition with the exception that the number of processors over which data is partitions is set to 1.

(Discussion: Should we have a DRI_PARTITION_REPLICATED? What are the ramifications to the rest of the spec?)

* Accessors

The following function is used to determine the type of a partition.

```
int DRI_Partition_get_type(DRI_Partition partition, int *type);
```

The returned 'type' will be one of the following constants:

- DRI_PARTITION_TYPE_BLOCK
Returned for partitions created with either of the block partition constructors.

- DRI_PARTITION_TYPE_BLOCK_CYCLIC
Returned for partitions created with either of the block cyclic partition constructors.

The following routines are used to retrieve useful information about either block or block cyclic partitions.

```
int DRI_Partition_get_left_overlap(DRI_Partition, int *left_overlap);  
int DRI_Partition_get_right_overlap(DRI_Partition, int  
*right_overlap);  
int DRI_Partition_get_pad(DRI_Partition, int *pad);  
int DRI_Partition_get_left_overlap(DRI_Partition, int *left_overlap);
```

The following routines are used to retrieve useful information about block partitions. The value DRI_UNDEFINED will be returned if a block cyclic distribution is passed into the function:

```
int DRI_Partition_get_modulo(DRI_Partition, int *modulo);  
int DRI_Partition_get_minimum(DRI_Partition, int *minimum);
```

The following routine is used to retrieve useful information about block cyclic partitions. The value DRI_UNDEFINED will be returned if a block distribution is passed into the function:

```
int DRI_Partition_get_block_size(DRI_Partition, int *block_size);
```

II.E Memory layout

The memory layout describes, through the use of strides for each dimension, how data is physically laid out in processor memory.

* Constructors

```
int DRI_Layout_create(int ndims, int strides[], DRI_Layout *layout);
```

The "strides" array is used to specify how "fast" each dimension moves.

In other words, for a particular dimension, the stride value for that dimension indicates the distance between two consecutive elements in that dimension.

* Destructors

```
int DRI_Layout_free(DRI_Layout *layout);
```

After this call returns, the value of layout will equal DRI_LAYOUT_NULL;

* Predefined layouts

There is one predefined layout:

DRI_LAYOUT_PACKED

which can be used to specify that dimensions are layed out in memory in order (i.e., the first dimension moves "fastest", the last dimension moves "slowest"). For example, in C, this would correspond to the fact that elements in the same row of a matrix are contiguous.

* Accessors

The following routine returns the number of dimensions in the layout object.

```
int DRI_Layout_get_ndims(DRI_Layout layout, int *ndims);
```

The following routine returns the stride of a particular dimension:

```
int DRI_Layout_get_stride(DRI_Layout layout, int dim, int *stride);
```

The following routine returns the array of strides for a layout:

```
int DRI_Layout_get_strides(DRI_Layout layout, int strides[]);
```

A value of DRI_UNDEFINED is returned for each of the accessor routines if they are called with DRI_LAYOUT_PACKED (Rationale: DRI_LAYOUT_PACKED is a special layout that can be used with data of

any dimension).

II.F Distribution

The DRI distribution object combines the following information into one object:

- * the global data,
- * the process group over which the global data is distributed,
- * and partitions for each dimension.

The distribution object pulls together in one place all the information needed to describe a distributed data distribution. This object is used not only to describe a data reorganization operation but also to allow the user to determine on which processor individual pieces of a distributed data structure data are actually located.

* Constructors

The following routine is used to create a distribution object:

```
int DRI_Distribution_create(DRI_Global_data global_data,  
DRI_Group group,  
DRI_Partition partition[],  
DRI_Distribution *dist);
```

global_data - describes the shape of the data structure to be distributed

group - group of processors over which the data will be distributed

partition - array of partitions. The size of this array must equal the number of dimensions in "global_data".

dist - newly created distribution

If one or more of the partitions in the partition array was specified as having 0 "procs", the distribution constructor will attempt to provide a reasonable value for that particular dimension. It attempts to select a balanced distribution of processors per coordinate direction, depending on the number of processes in the group to be balanced and upon the number of processors specified in the other partitions. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. An error will occur if the number of nodes in the group is not a multiple of the product of non-zero "procs" in the partitions. The call will allocate processors to each partition in non-increasing order. For example, if two "procs" values are specified as 0 for a two dimensional distributed data structure, the value of the "procs" associated with the first (index 0 element in the partition array) will be greater than or equal to the "procs" value associated with the second entry (index 1 element in the partition array).

Although the distribution constructor associates processor counts with partitions that were originally created with "procs" = 0, the original partitions are not modified. Upon return from the constructor, each of the partitions in the partition array will be as they were before the call to the distribution constructor.

In the following examples, assume we have been given:

Object Contents

Global data 3 dims, 100 x 500 x 10
Group 20 processors

Example 1:

Object Contents

Partitions { p1(procs=0), p2(procs=5), p3(procs=2) }

Creating the distribution object will assign 2 processors to the p1 partition.

Example 2:

Object Contents

Partitions { p1(procs=0), p2(procs=0), p3(procs=0) }

Creating the distribution object will assign: p1(procs=5), p2(procs=2), p3(procs=2).

Example 3:

Object Contents

Partitions { p1(procs=3), p2(procs=2), p3(procs=0) }

An error will be generated since the product of the number of processors(20) is not a multiple of the product of non-zero procs(6).

Example 4:

Object Contents

Partitions { p1(procs=5), p2(procs=4), p3(procs=0) }

Creating the distribution object will assign: p3(procs=1).

* Destructors

The following routine frees a distribution. :

```
int DRI_Distribution_free(DRI_Distribution *dist);
```

After this call returns, the value of dist will equal DRI_DISTRIBUTION_NULL;

* Accessors

DRI distributions require a wide range of accessors. These accessors answer questions such as:

- What are the objects that make up this distribution (i.e., what were the objects passed into the constructor)?
- How much data do I own? How much space does a processor need to allocate for a particular distribution? How much data does a particular processor have?
- Who has a certain element (e.g., who owns array element (3,2,4) of a 3 dimensional array)? What elements does a certain processor own?

The following routines return information about the arguments passed into the distribution constructor:

```
int DRI_Distribution_get_global_data(DRI_Distribution dist,  
DRI_Global_data *gd);
```

```
int DRI_Distribution_get_group(DRI_Distribution dist,  
DRI_group *group);
```

```
int DRI_Distribution_get_partition(DRI_Distribution dist,  
DRI_partition partition[]);
```

The following calls are used to determine how much data individual processors hold (data with overlap and pad) and own (data without overlap or pad).

```
int DRI_Distribution_get_count(DRI_Distribution dist,  
int rank, int *count);
```

dist - the distribution

rank - rank of the processor we want to find out about

count - how many elements are stored by processor "rank". This includes overlap or pad. The count is not a byte count -- it is a count of how many elements relative to the datatype specified during construction of the global data.

```
int DRI_Distribution_get_owned_count(DRI_Distribution dist,
```

```
int rank, int *count);
```

dist - the distribution

rank - rank of the processor we want to find out about

count - how many elements are owned by processor "rank".
This does not include overlap or pad.

This function can be used in conjunction with the MPI extent function to determine how much memory space is required by a distribution on a particular processor.

For any distribution, an element has a local index (for each dimension) and a global index (for each dimension). The local index is the index relative to the processors local buffer. The global index is the index relative to the global data structure. The following functions are used to aid in mapping between local and global indices:

```
int DRI_Distribution_location(DRI_Distribution dist,  
int global[],  
int *rank);
```

dist - the distribution

global - the global indices of the data element

rank - rank of the processor that owns the element

```
int DRI_Distribution_global_to_local(DRI_Distribution dist,  
int rank, int global[],  
int local[]);
```

dist - the distribution

rank - rank of the processor relative to which the mapping should be made

global - the global indices of the data element

local - the resulting local mapping. If the data element does not exist on the "rank" processor, DRI_UNDEFINED is returned for each of the indices.

```
int DRI_Distribution_local_to_global(DRI_Distribution dist,  
int rank, int local[],  
int global[]);
```

dist - the distribution

rank - rank of the processor relative to which the mapping should be made

local - the local indices of the data element

global - where the element exists in the global data structure

It's sometimes useful to obtain a listing of the indices in each dimension that a particular processor either owns or has access to (i.e., overlap/pad). Distribution iterators are used to allow the user to iterate through the indices owned by a particular processor.

The following function is used to create a distribution iterator. The constructed iterator will iterate through each index resident in the specified processor's memory (i.e., including pad & overlap).

```
int DRI_Distribution_iterator_create(DRI_Distribution dist,  
int rank, int dim,  
DRI_Distribution_iterator *it)
```

dist - the distribution

rank - rank of the processor to which this iterator will refer

dim - dimension of the global data this iterator will cover

it - newly created distribution iterator

The following function is used to create a distribution iterator. The constructed iterator will iterate only through each index owned by the specified processor (i.e., it will NOT include pad & overlap).

```
int DRI_Distribution_iterator_create_owned(DRI_Distribution dist,  
int rank, int dim,  
DRI_Distribution_iterator  
*it)
```

dist - the distribution

rank - rank of the processor to which this iterator will refer

dim - dimension of the global data this iterator will cover

it - newly created distribution iterator

The following operation frees a distribution iterator.

```
int DRI_Distribution_iterator_free(DRI_Distribution_iterator *it)
```

it - distribution iterator to free. It will point to DRI_DISTRIBUTION_ITERATOR_NULL after the routine returns.

The following operations are used to traverse an iterator. DRI_Distribution_iterator_next increments the iterator to the next available index and returns its value. If DRI_Distribution_iterator_next is called on a previously unused iterator, it returns the first index.

```
int DRI_Distribution_iterator_next(DRI_Distribution_iterator it,  
int *index, int *flag)
```

it - distribution iterator

index - the "next" index. The index is a global index.

flag - is used to indicate whether or not the returned index value is valid. If flag is 1, the end of the iterator has not been reached -- the value of index is valid. If flag is 0, the end of the iterator has been reached, the value of index is undefined.

DRI_Distribution_iterator_prev decrements the iterator to the previous index and returns its value. If DRI_Distribution_iterator_prev is called on a previously unused iterator, it returns the last index.

```
int DRI_Distribution_iterator_prev(DRI_Distribution_iterator it,  
int *index, int *flag)
```

it - distribution iterator

index - the "previous" index. The index is a global index.

flag - is used to indicate whether or not the returned index value is valid. If flag is 1, the end of the iterator has not been reached -- the value of index is valid. If flag is 0, the end of the iterator has been reached, the value of index is undefined.

The following function resets an iterator to its original "unused" state.

```
int DRI_Distribution_iterator_reset(DRI_Distribution_iterator it);
```

The following function is used to determine whether the current index pointed to by the iterator is an overlap/pad index.

```
int DRI_Distribution_iterator_overlap(DRI_Distribution_iterator it,  
int *flag)
```

it - distribution iterator

flag - returns 1 if the current index is overlap or pad, returns 0 otherwise

As an example of how distribution iterators might be used, consider

how one would iterate through each element of a 3 dimensional matrix available in processor 0:

```
DRI_Distribution_iterator it_i, it_j, it_k;
int i, j, k;
int f_i, f_j, f_k;
DRI_Distribution_iterator_create(dist, 0, 0, &it_i);
DRI_Distribution_iterator_create(dist, 0, 1, &it_j);
DRI_Distribution_iterator_create(dist, 0, 2, &it_k);

DRI_Distribution_iterator_next(it_i,&i,&f_i);
while (f_i) {
DRI_Distribution_iterator_next(it_j,&j,&f_j);
while (f_j) {
DRI_Distribution_iterator_next(it_k,&k,&f_k);
while (f_k) {

/* do whatever to global element i,j,k */

DRI_Distribution_iterator_next(it_k,&k,&f_k);
}
DRI_Distribution_iterator_reset(it_k);
DRI_Distribution_iterator_next(it_j,&j,&f_j);
}
DRI_Distribution_iterator_reset(it_j);
DRI_Distribution_iterator_next(it_i,&i,&f_i);
}

DRI_Distribution_iterator_free(&it_i);
DRI_Distribution_iterator_free(&it_j);
DRI_Distribution_iterator_free(&it_k);
```

(Discussion: As a variation of this, you could have the `iterator_next` and `iterator_prev` return blocks of information. In other words, instead of returning a single index, it would return two indices -- begin and end.)

II.G Communication

This section details the routines used for performing the actual data reorganizations from one distribution to another.

The basic calls used to perform a redistribution operation are:

```
int DRI_Send(void *buffer, int tag,
DRI_Distribution local_dist,
DRI_Distribution remote_dist)
```

buffer - buffer containing data local to the processor

tag - tag value must match on sender and receiver

local_dist - describes the distribution of data on the sending side of the reorganization operation

remote_dist - describes the distribution of data on the receiving side of the reorganization operation

```
int DRI_Recv(void *buffer, int tag,  
DRI_Distribution local_dist,  
DRI_Distribution remote_dist)
```

buffer - buffer containing data local to the processor

tag - tag value must match on sender and receiver

local_dist - describes the distribution of data on the receiving side of the reorganization operation

remote_dist - describes the distribution of data on the sending side of the reorganization operation

These functions require that the local distribution of the sender match the remote distribution of the receiver and vice-versa. The DRI_Send call has semantics similar to the standard MPI_Send call with respect to buffering requirements.

(To do: Describe "safe" and "unsafe" situations with simply using DRI_Send and DRI_Recv. For example, if the groups of the local and remote distributions are the same, calling DRI_Send followed by DRI_Recv may or may not work -- i.e., it's an unsafe program. Whether it worked or not would depend on how it's implemented. One possible implementation is to issue a bunch of MPI_Isend calls followed by a MPI_Waitall. If the MPI implementation has enough buffer space, the call will work. If not, the call will fail.)

(Rationale: Both the local and remote distribution information is specified by the sender and receiver. It's possible to provide a layered library that does not require distribution information about both sides. The layered library could use a name server approach to finding out the remote distribution and then use these calls to actually implement the redistribution)

In addition to the standard mode DRI_Send, the DRI interface has 3 additional modes that correspond to the buffered, ready, and synchronous MPI send modes:

```
int DRI_Bsend(void *buffer, int tag,  
DRI_Distribution local_dist,  
DRI_Distribution remote_dist)
```

This function has similar semantics to that of MPI_Bsend. It either uses an MPI_Bsend variant to implement itself or (if the implementation is by the MPI vendor) uses the buffer space provided to the implementation by MPI_Buffer_attach.

```
int DRI_Ssend(void *buffer, int tag,  
DRI_Distribution local_dist,
```

DRI_Distribution remote_dist)

This call will not return until all processors in the remote_dist processor group have called a matching receive operation.

```
int DRI_Rsend(void *buffer, int tag,  
DRI_Distribution local_dist,  
DRI_Distribution remote_dist)
```

This call assumes all processors in the remote_dist processor group have called a matching receive operation prior to the sender issuing the DRI_Rsend.

Each of these send and receive functions are blocking where the terms blocking and non-blocking are defined as in the MPI-1 specification. After a blocking send call returns, the application may write into the buffer with full confidence that the changes will not be transmitted to the receiving group. After a non-blocking send call returns, the application may not write into the buffer since changes made at that point may be transmitted to the receiving group. After a blocking receive, valid data is guaranteed to be in the buffer (i.e., data received from the send group). Data is not guaranteed to be valid after a non-blocking receive.

For DRI_Recv and each of the various DRI send functions, a non-blocking version of the operation is provided:

```
int DRI_Irecv(void *buffer, int tag,  
DRI_Distribution local_dist,  
DRI_Distribution remote_dist,  
DRI_Request *request)
```

```
int DRI_Isend(void *buffer, int tag,  
DRI_Distribution local_dist,  
DRI_Distribution remote_dist,  
DRI_Request *request)
```

```
int DRI_Ibrecv(void *buffer, int tag,  
DRI_Distribution local_dist,  
DRI_Distribution remote_dist,  
DRI_Request *request)
```

```
int DRI_Issend(void *buffer, int tag,  
DRI_Distribution local_dist,  
DRI_Distribution remote_dist,  
DRI_Request *request)
```

```
int DRI_Irsend(void *buffer, int tag,  
DRI_Distribution local_dist,  
DRI_Distribution remote_dist,  
DRI_Request *request)
```

Various functions can be used to determine if a non-blocking operation has completed. The following routine waits until a non-blocking request has completed.

DRI_Wait(DRI_Request *request);

request - request will be set to DRI_REQUEST_NULL upon completion of this function (unless the request is a persistent request).

The following routine tests to see if a non-blocking operation has finished.

DRI_Test(DRI_Request *request, int *flag);

request - request will be set to DRI_REQUEST_NULL if the request has completed (and it is not a persistent request), otherwise it will not be modified.

flag - is set to 1 if the request has completed, 0 otherwise.

(To do: Add DRI variations of the other MPI completion functions: DRI_Waitany, DRI_Waitany, DRI_Waitany, DRI_Testany, DRI_Testany, DRI_Testany)

In order to reduce the overhead associated with setting up a distribution operation, persistent communication operations are also available (based on MPI persistent communication):

int DRI_Recv_init(void *buffer, int tag,
DRI_Distribution local_dist,
DRI_Distribution remote_dist,
DRI_Request *request)

int DRI_Send_init(void *buffer, int tag,
DRI_Distribution local_dist,
DRI_Distribution remote_dist,
DRI_Request *request)

int DRI_Bsend_init(void *buffer, int tag,
DRI_Distribution local_dist,
DRI_Distribution remote_dist,
DRI_Request *request)

int DRI_Ssend_init(void *buffer, int tag,
DRI_Distribution local_dist,
DRI_Distribution remote_dist,
DRI_Request *request)

int DRI_Rsend_init(void *buffer, int tag,
DRI_Distribution local_dist,
DRI_Distribution remote_dist,
DRI_Request *request)

The requests created with these operations can be started by calling:

```
int DRI_Start(DRI_Request *request);
```

A request must complete before it can be restarted.

The following routine is provided for cases where a processor participates as both sender and receiver:

```
int DRI_Sendrecv(void *send_buffer, int stag, DRI_Distribution send_dist,  
void *recv_buffer, int rtag, DRI_Distribution recv_dist);
```

A persistent version of this call is available:

```
int DRI_Sendrecv_init(void *sendbuffer, int stag, DRI_Distribution senddist,  
void *recvbuffer, int rtag, DRI_Distribution recvdist,  
DRI_Request *request);
```

(Discussion: Should we have DRI_Bsendrecv, DRI_Ssendrecv,
DRI_Rsendrecv?
Should we then have persistent versions of each of these?)

Appendix A. Constants

=====

```
DRI_GLOBAL_DATA_NULL  
DRI_GROUP_NULL  
DRI_PARTITION_NULL  
DRI_DISTRIBUTION_NULL  
DRI_LAYOUT_NULL  
DRI_DISTRIBUTION_ITERATOR_NULL  
DRI_UNDEFINED  
DRI_PARTITION_TYPE_BLOCK  
DRI_PARTITION_TYPE_BLOCK_CYCLIC  
DRI_LAYOUT_PACKED  
<add missing ones>
```