A Manual for InterComm
Version 1.6

Norman Lo, Il-Chul Yoon, Jae-Yong Lee, Christian Hansen,
Henrique Andrade, Guy Edjlali, Alan Sussman
Department of Computer Science
University of Maryland
College Park, MD 20742
{normanlo,iyoon,jylee,chansen,edjlali,hcma,als}@cs.umd.edu

June 4, 2007

Contents

1 Introduction

InterComm is a framework for coupling distributed memory parallel components that enables efficient communication in the presence of complex data distributions. In many modern scientific applications, such as physical simuations that model phenomena at multiple scales and resolutions, multiple parallel and/or sequential components need to cooperate to solve a complex problem. These components often use different languages and different libraries to parallelize their data. InterComm provides abstractions that work across these differences to provide an easy, efficient, and flexible means to move data directly from one component's data structure to another.

The two main abstractions InterComm provides are the *distribution*, which describes how data is partioned and distributed across multiple tasks (or processors), and the *linearization* which provides a mapping from one set of elements in a distribution to another.

1.1 Distributions

InterComm classifies data *distributions* into two types, those in which entire blocks of an array are assigned to tasks, a block decomposition, and those in which individual elements of an array are assigned independently to a particular task, a translation table. In the case of the former, the data structure required to describe the distribution is relatively small and can be replicated on each of the participating tasks. In the case of the latter, there is a one-to-one correspondance between the elements of the array and the number of entries in the data descriptor, therefore, the descriptor itself can be large and must be partitioned across the participating tasks. InterComm provides two primitives for specifying these types of distrubtions (Section 3.2), as well as primitives to identify regions (sub-arrays) to transfer within these distributions (Section 3.3).

1.2 Linearization

A *linearization* is the method by which InterComm defines an implicit mapping between the source of a data transfer distributed by one data parallel program and the destination of the transfer distributed by another program. The source and destination data elements are each described by a set of regions.

One view of the linearization is as an abstract data structure that provides a total ordering for the data elements in a set of regions. The linearization for a region is provided by a data parallel library or directly by the application writer.

We represent the operation of translating from the set of regions S_A of A, distributed by `libX`, to its linearization, L_{S_A}, by parallel library ℓ_{libX}, and the inverse operation of translating from the linearization to the set of regions as ℓ_{libX}^{-1}:

$$L_{S_A} = \ell_{libX}(S_A)$$

$$S_A = \ell_{libX}^{-1}(L_{S_A})$$

Moving data from the set of regions S_A of A distributed by `libX` to the set of regions S_B of B distributed by library `libY` can be viewed as a three-phase operation:

1. $L_{S_A} = \ell_{libX}(S_A)$
2. $L_{S_B} = L_{S_A}$
3. $S_B = \ell^{-1}_{libY}(L_{S_B})$

The only constraint on this three-phase operation is to have the same number of elements in S_A as in S_B, in order to be able to define the mapping between data elements from the source to the destination.

The concept of linearization has several important properties:

- It does not require the explicit specification of the mapping between the source data and destination data. The mapping is implicit in the separate linearizations of the source and destination data structures.
- A parallel can be drawn between the linearization and the marshal/unmarshal operations for the parameters of a remote procedure call. Linearization can be seen as an extension of the marshal/unmarshal operations to distributed data structures.

1.3 Language Interfaces

InterComm supports programs written in both C and Fortran77. The functions provided for these languages are considered the *low-level* interfaces, as they provide great flexibilty in defining distributions, but require a large amount of attention to detail. In Section 3, the C and Fortran77 InterComm interfaces are presented. As a general rule, the Fortran functions are identical to their C counterparts. They only differ when a value is returned from the C function, in which case this value becomes the last parameter of the Fortran call, or when dealing with a C pointer type, which has been made to correspond with the integer type in Fortran. The C interface is specified in the `intercomm.h` source file in the InterComm software distribution.

InterComm also supports programs written in Fortran90 and C++/P++[2]. The main objective of this *high-level* interface is to encapsulate some of the complexity of describing and communicating data between the local and the remote applications. In Section 5, the C++/P++ and Fortran90 interfaces are described.

In addition to those interfaces, InterComm versions 1.5 and above provide a new interface, available with both low-level and high-level interfaces that employs an XML Job Description (XJD) to further decouple implementations of separate programs that communicate with InterComm, as will be further described in Sections 4 and 6.

1.4 Guidelines for using the API

In total, InterComm defines three types of API - low-level, high-level and XJD-based, and they provide different levels of granularity for handling InterComm program behavior. The low-level programming API, described in Section 3, provides the finest control over program behavior for initializing and finalizing InterComm, and for defining and handling regions in different programs. The high-level programming API, described in Section 5 is an encapsulation of the low-level programming API that provides simplified interfaces for C++, relying on the P++ parallel array class library, and for sequential Fortran90 programs. The low-level XJD-based programming API, described in Section 4 is another encapsulation of the low-level programming API that provides simplified interfaces for C and Fortran77 programs, and is described in Section 4. The XJD-based programming API is distinctive in that an XML Job Description (XJD) is used to create connections between InterComm programs, to enable different programs to communicate with each other with each program only specifying its side of the communication (an *import* or *export* operation), and not specifying anything about the program(s) on the other side of the communication operation. An XJD-based high-level API corresponding to the original InterComm high-level API is described in Section 6.

2 Downloading and Installation

The source package, as well as an online copy of this manual and a Programmer's Reference, can be obtained from the project website at http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/ic/.

InterComm depends on PVM [3] for interprocess communication, so the first step is to insure that you have a working and properly configured installation of PVM available. In particular, InterComm will need the environment variables PVM_ROOT and PVM_ARCH set during configuration.

InterComm uses the GNU Autotools [4] suite for building and installation. Generally, this involves the sequence of commands:

```
./configure
make
make install
```

The `configure` command takes a number of options; use of the `--help` flag will provide a full list of those available. Options that are specific to InterComm are:

`-- with-chaos=DIR`
: This causes the Chaos [5] extensions to be built (see Section A.1.1). DIR specifies where the Chaos headers and libraries can be found.

`-- with-mbp=DIR`
: This causes the MultiBlock Parti [6] extensions to be built (see Section A.1.2). DIR specifies where the MultiBlockParti headers and libraries can be found.

-- `with-ppp=DIR`
This causes the C++/P++ interface to be built. DIR specifies where the P++ headers and libraries can be found.

-- `enable-f77`
This causes the Fortran 77 interface to be built (default).

-- `enable-f90`
This causes the Fortran 90 interface to be built.

-- `enable-tests`
This causes the test scripts to be built to ensure the installation is successful.

-- `enable-mpi`
This causes the distribution test cases using MPI to be built (default).
If `--with-ppp=DIR` is used, this will be enabled automatically.

-- `enable-debug`
This causes debug statements to be shown while using InterComm to help find the source of bugs and other program problems.

3 Low-Level Programming Tasks

This section describes how to use the C and Fortran interfaces for InterComm.

3.1 Initializing the Library

Before InterComm is used to transfer data between programs, each program must first provide the runtime library with information regarding the programs participating in the communication. This is done with two calls, `IC_Init` and `IC_Wait`.

3.1.1 IC_Init

This function initializes the underlying communication library and creates an internal representation of the local program for use with other calls into the communication system.

Synopsis

C IC_Program* IC_Init(char* name, int tasks, int rank)

Fortran IC_Init(name, tasks, rank, myprog)
character name(*)
integer tasks, rank, myprog

Parameters

name the externally visible name of the program

tasks the number of tasks used

rank the rank of this task

Return value

an InterComm program data type

Example

```
IC_Program* myprog;
char* name = ‘‘cmpiexample’’;
int rank, tasks = 4;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
myprog = IC_Init(name, tasks, rank);
```

3.1.2 IC_Wait

This function contacts a remote program to arrange for subsequent communications. It returns an internal representation of the remote program for use in data exchange operations.

Synopsis

C IC_Program* IC_Wait(char* name, int tasks)

Fortran IC_Wait(name, tasks, prog)
character name(*)
integer tasks, prog

Parameters

name the name of the remote program

tasks the number of tasks used

Return value

an InterComm program data type

Example

```
IC_Program* prog;
char* name = ‘‘cpvmexample’’;
int tasks = 8;
prog = IC_Wait(name, tasks);
```

3.1.3 IC_Sync

This function call allows the establishment of a synchronization point between two programs by causing each to wait until both sides have made a matching call.

Synopsis

C int IC_Sync(IC_Program* myprog, IC_Program* prog)

Fortran IC_Sync(myprog, prog, status)
integer myprog, prog, status

Parameters

myprog the local program

prog the remote program

Return value

status, -1 indicates an error

Example

```
int sts;
sts = IC_Sync(myprog, prog);
```

3.2 Describing the Data Distribution

InterComm needs information about the distribution across tasks of a data structure, managed either by the application programmer or by a data parallel library employed by the application. In most parallel programs manipulating large multidimensional arrays, a distributed data descriptor is used to store the necessary information about the regular or irregular data distribution (e.g., a distributed array descriptor for MultiBlock Parti [6] or a translation table for Chaos [5]).

As the InterComm functions for moving data require a data descriptor that is meaningful within the context of InterComm, several functions are provided for describing these two types of distributions. Ideally, these functions would be used by a data parallel library developer to provide a function that translates from their data descriptor specification to the one used by InterComm. Such translation functions are provided for MultiBlock Parti, Chaos and P++ in the InterComm distribution (see Section A).

3.2.1 IC_Create_bdecomp_desc

Block decompositions assign entire array sections to particular tasks (processes), allowing for a compact description of array element locations. This function is used to describe a regular block decomposition.

Synopsis

C IC_Desc* IC_Create_bdecomp_desc(int ndims, int* blocks, int* tasks, int count, int arrayOrder)

Fortran IC_Create_bdecomp_desc(ndims, blocks, tasks, count, desc, arrayOrder)
integer ndims, blocks(*), tasks(*), count, desc, arrayOrder

Parameters

ndims the dimension of the distributed array

blocks a three-dimensional array of block bound specifications (see example)

The first dimension of the **blocks** array corresponds to the number of blocks used to distribute the array (one per task – see information for the **tasks** array below). The second dimension is always two (i.e., each block is represented by *two* n-dimensional points as an n-dimensional rectangular box, where n is the number of dimensions of the distributed array for which the decomposition is being created. Each of the two points represents the corners of the rectangular box). The third dimension of the **blocks** array corresponds to n, where again n is the number of the dimensions of the distributed array for which the decomposition is being created.

tasks the corresponding task assignments for the individual blocks, starting from 0 in both C and Fortran

The k-th block in the **blocks** array (i.e., blocks[k][x][y]) is held by the k-th task (i.e., tasks[k])

count the number of blocks

arrayOrder the order (IC_ROW_MAJOR or IC_COLUMN_MAJOR) of the distributed array

Return value

an InterComm array descriptor data type. NULL or negative value means error

Example

```
/* 4 blocks, 2 points, 2-dimensional points */
int blocks[4][2][2] = {
  {{0,0},{3,3}}, /* block 0 */
  {{0,4},{3,7}}, /* block 1 */
  {{4,0},{7,3}}, /* block 2 */
  {{4,4},{7,7}}  /* block 3 */
};
int tasks[4] = {0,1,2,3};
IC_Desc* desc;

desc = IC_Create_bdecomp_desc(2, &blocks[0][0][0], tasks, 4, IC_ROW_MAJOR);
```

3.2.2 IC_Create_ttable_desc

The translation table is a representation of an arbitrary mapping between data elements and tasks in a parallel program. This information must be explicitly provided to this routine defining the descriptor. The

descriptor is distributed, and therefore partial for a given task. This function assigns array elements to tasks using a given partial map.

Synopsis

C IC_Desc* IC_Create_ttable_desc(int* globals, int* locals, int* tasks, int count)

Fortran IC_Create_ttable_desc(globals, locals, tasks, count, desc)
integer globals, locals(*), tasks(*), count, desc

Parameters

globals an array of global indices

locals the corresponding local indices

tasks a corresponding global index-to-task map, starting from 0 in both C and Fortran

count the number of global indices

Return value

an InterComm array descriptor data type

Example

```
/* local portion of the global index space */
int globals[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int locals[16] = {0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3};
int tasks[16] = {0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3};
IC_Desc* desc;

desc = IC_Create_ttable_desc(globals, locals, tasks, 16);
```

Since defining an entire block decomposition at once with IC_Create_bdecomp_desc may not be very convenient for a particular application, InterComm provides two additional methods for iteratively defining a distribution. Both of these functions operate upon an IC_Tree data type, which is not a complete descriptor, but a template which may be converted to one when all the partitions assigned to tasks have been specified.

3.2.3 IC_Create_bdecomp_tree

This function creates an empty IC_Tree data type with no partitions.

Synopsis

C IC_Tree* IC_Create_bdecomp_tree()

Fortran IC_Create_bdecomp_tree(root)
integer root

Return value

an InterComm partial decomposition data type

Example

```
IC_Tree* root;
root = IC_Create_bdecomp_tree();
```

3.2.4 IC_Section

This function creates a partition that will cut across all dimensions of the global array. Generally, one of these calls is made for each dimension and the order of calls needs to be from 1st to nth dimension.

Synopsis

C void IC_Section(IC_Tree* root, int dim, int count, int* indices)

Fortran IC_Section(root, dim, count, indices)
integer root, dim, count, indices(*)

Parameters

root a partial decomposition

dim the dimension to partition, starting from 0 in both C and Fortran

count the number of partitions

indices the upper bounding index for each partition

Return value

none

Example

```
int dim = 0, count = 2;
int indices[2] = {5, 10};
IC_Section(root, dim, count, indices);
```

3.2.5 IC_Partition

This function creates a partition for a particular subtree of the overall distribution, allowing for the recursive creation of irregular block decompositions. It returns an array of IC_Tree, which are the children of the subtree partitioned. These children can then in turn be further partitioned.

Synopsis

C IC_Tree* IC_Partition(IC_Tree* root, int dim, int *block, int count, int* indices)

Fortran IC_Partition(root, dim, block, count, indices)
integer root, dim, block(*), count, indices(*)

Parameters

root a partial decomposition

dim the dimension to partition, starting from 0 in both C and Fortran

block the way of traversing to the subtree to be partitioned on

count the number of partitions

indices the upper bounding index for each partition

Return value

an array of InterComm partial array descriptors

Example

```
IC_Tree *root;
int indices[2];
int dim = 0, count=2;
int *block0 = NULL; // since no partitions yet
int block1[1];
int block2[2];
indices[0] = 5; indices[1] = 10;
IC_Partition(root, dim, block0, count, indices);
dim = 1;
indices[0] = 4; indices[1] = 10;
block1[0] = 0; // parition on first child on first dimension
IC_Partition(root, dim, block1, count, indices);
indices[0] = 6; indices[1] = 10;
block1[0] = 1;
IC_Partition(root, dim, block1, count, indices);
dim = 2;
indices[0] = 3; indices[1] = 10;
block2[0] = 0; block2[1] = 1;
//partition on first child on first dimension, second child on second dimension
IC_Partition(root, dim, block2, count, indices);
```

3.2.6 IC_Verify_bdecomp_tree

Once all the partitions have been specified for a particular distribution, this function will verify that the distribution covers the entire global array and that no array element is assigned to multiple partitions. It returns an InterComm array descriptor that can then be used for schedule building and communication operations.

Synopsis

C IC_Desc* IC_Verify_bdecomp_tree(IC_Tree* root, int ndims, int* size, int* tasks, int count, int arrayOrder)

Fortran IC_Verify_bdecomp_tree(root, ndims, size, tasks, count, desc, arrayOrder)
integer root, ndims, size(*), tasks(*), count, desc, arrayOrder

Parameters

root a partial decomposition

ndims the number of dimensions

size the size of the global array

tasks the task assignment for each block, start from 0 for both C and Fortran

count the number of blocks

arrayOrder the order (IC_ROW_MAJOR or IC_COLUMN_MAJOR) of the array

Return value

a complete InterComm array descriptor

Example

```
IC_Desc* desc;
int size[2] = {10, 10};
int tasks[4] = {0, 1, 2, 3};
desc = IC_Verify_bdecomp_tree(root, 2, size, tasks, 4, IC_ROW_MAJOR);
```

3.3 Defining and Communicating Array Blocks

3.3.1 IC_Create_block_region

This function allocates a region (block or sub-array) designating the data elements for subsequent data communication operations.

Synopsis

C IC_Region* IC_Create_block_region(int ndims, int* lower, int* upper, int* stride)

Fortran IC_Create_block_region(ndims, lower, upper, stride, region)
integer ndims, lower(*), upper(*), stride(*), region

Parameters

ndims the number of dimensions of the array

lower the lower bounds of the region to be transferred

upper the upper bounds of the region to be transferred

stride the strides of the region to be transferred

Return value

an InterComm region data type

Example

```
int ndims = 3;
int lower[3] = {0,0,0};
int upper[3] = {2,2,2};
int stride[3] = {1,1,1};
IC_Region* region_set[1];
region_set[0] = IC_Create_block_region(ndims, lower, upper, stride);
```

3.3.2 IC_Create_enum_region

This function allocates a region (enumerated one element at a time) designating the data elements for importing or exporting operations.

Synopsis

C IC_Region* IC_Create_enum_region(int* indices, int size)

Fortran IC_Create_enum_region(indices, size, region)
integer indices(*), size, region

Parameters

indices an array of global indices

size the number of global indices enumerated

Return value

an InterComm region data type

Example

```
int indices[10] = {0,1,2,3,4,5,6,7,8,9};
int size = 10;
IC_Region* region_set[1];
region_set[0] = IC_Create_enum_region(indices, size);
```

3.3.3 IC_Compute_schedule

This function creates a communication schedule for communicating regions (blocks/sub-arrays) of an array. The types of array descriptors used on either end of the communication determine how and where this schedule is computed.

Synopsis

C IC_Sched* IC_Compute_schedule(IC_Program* myprog, IC_Program* prog, IC_Desc* desc, IC_Region** region_set, int set_size)

Fortran IC_Compute_schedule(myprog, prog, desc, region_set, set_size, sched)
integer myprog, prog, desc, region_set(*), set_size, sched

Parameters

myprog the InterComm application descriptor for the local program

prog the InterComm application descriptor for the remote program

desc the InterComm array descriptor

region_set an array of regions describing the data to be communicated

set_size the number of regions in the array

Return value

an InterComm schedule data type

Example

```
IC_Sched* sched;
sched = IC_Compute_schedule(myprog, prog, desc, region_set, 1);
```

3.3.4 IC_Send_TYPE

This function is used for sending a set of array regions to a remote application. There is a send function for each supported basic data TYPE (char, short, int, float, double).

Synopsis

Fortran 90 IC_Send(to, sched, data, tag, status)
Note that the Fortran 90 IC_Send function is polymorphic, i.e., it does not require calling a particular version depending on the type of the **data** being received as do the C and Fortran 77 counterparts.

C int IC_Send_char(IC_Program* to, IC_Sched* sched, char* data, int tag)

Fortran IC_Send_char(to, sched, data, tag, status)
integer to, sched, tag, status
integer*1 data(*)

C int IC_Send_short(IC_Program* to, IC_Sched* sched, short* data, int tag)

Fortran IC_Send_short(to, sched, data, tag, status)
integer to, sched, tag, status
integer*2 data(*)

C int IC_Send_int(IC_Program* to, IC_Sched* sched, int* data, int tag)

Fortran IC_Send_int(to, sched, data, tag, status)
integer to, sched, tag, status
integer data(*)

C int IC_Send_float(IC_Program* to, IC_Sched* sched, float* data, int tag)

Fortran IC_Send_float(to, sched, data, tag, status)
integer to, sched, tag, status
real data(*)

C int IC_Send_double(IC_Program* to, IC_Sched* sched, double* data, int tag)

Fortran IC_Send_double(to, sched, data, tag, status)
integer to, sched, tag, status
real*8 data(*)

Parameters

to the InterComm application descriptor for the receiving program

sched the InterComm communication schedule

data (pointer to) the local array

tag a message tag for identifying this communication operation

Return value

status, -1 indicates an error

Example

```
int sts, tag;
float* data = &A[0][0][0];
sts = IC_Send_float(prog, sched, data, tag);
```

3.3.5 IC_Recv_TYPE

This function is used for receiving a set of regions from a sending program. There is a receive function for each supported basic data TYPE.

Synopsis

Fortran 90 IC_Recv(from, sched, data, tag, status)

Note that the Fortran 90 IC_Recv function is polymorphic, i.e., it does not require calling a particular version depending on the type of the **data** being received as do the C and Fortran 77 counterparts.

C int IC_Recv_char(IC_Program* from, IC_Sched* sched, char* data, int tag)

Fortran IC_Recv_char(from, sched, data, tag, status)
integer from, sched, tag, status
integer*1 data(*)

C int IC_Recv_short(IC_Program* from, IC_Sched* sched, short* data, int tag)

Fortran IC_Recv_short(from, sched, data, tag, status)
integer from, sched, tag, status
integer*2 data(*)

C int IC_Recv_int(IC_Program* from, IC_Sched* sched, int* data, int tag)

Fortran IC_Recv_int(from, sched, data, tag, status)
integer from, sched, tag, status
integer data(*)

C int IC_Recv_float(IC_Program* from, IC_Sched* sched, float* data, int tag)

Fortran IC_Recv_float(from, sched, data, tag, status)
integer from, sched, tag, status
real data(*)

C int IC_Recv_double(IC_Program* from, IC_Sched* sched, double* data, int tag)

Fortran IC_Recv_double(from, sched, data, tag, status)
integer from, sched, tag, status
real*8 data(*)

Parameters

from the sending program

sched the communication schedule

data (pointer to) the local array

tag a message tag identifying this communication operation

Return value

status, -1 indicates error

Example

```
int sts, tag;
float data[500][500][500];
tag = 99;
sts = IC_Recv_float(prog, sched, data, tag);
```

3.3.6 IC_Bcast_local_TYPE

This function broadcasts a data block located in a single sender task (the one making this call) to all tasks in the receiving program. Since the underlying message passing system broadcast function is used directly to perform the communication, no communication schedule is necessary. Correct use of this function requires that it must be invoked by *exactly one* task in the sending program. There is a broadcast function for each supported basic data TYPE.

Synopsis

Fortran 90 IC_Bcast_local(to, data, nelems, tag, status)
Note that the Fortran 90 IC_Bcast_local function is polymorphic, i.e., it does not require calling a particular version depending on the type of the **data** being received, as do the C and Fortran 77 counterparts.

C int IC_Bcast_local_char(IC_Program* to, char* data, int nelems, int tag)

Fortran IC_Bcast_local_char(to, data, nelems, tag, status)
integer to, nelems, tag, status
integer*1 data(*)

C int IC_Bcast_local_short(IC_Program* to, short* data, int nelems, int tag)

Fortran IC_Bcast_local_short(to, data, nelems, tag, status)
integer to, nelems, tag, status
integer*1 data(*)

C int IC_Bcast_local_int(IC_Program* to, int* data, int nelems, int tag)

Fortran IC_Bcast_local_int(to, data, nelems, tag, status)
integer to, nelems, tag, status
integer*1 data(*)

C int IC_Bcast_local_float(IC_Program* to, float* data, int nelems, int tag)

Fortran IC_Bcast_local_float(to, data, nelems, tag, status)
integer to, nelems, tag, status
integer*1 data(*)

C int IC_Bcast_local_double(IC_Program* to, double* data, int nelems, int tag)

Fortran IC_Bcast_local_double(to, data, nelems, tag, status)
integer to, nelems, tag, status
integer*1 data(*)

Parameters

to the receiving program

data (pointer to) the local array

nelems the number of elements in the local array

tag a message tag used to identify this communication operation

Return value

status, -1 indicates error

Example

```
IC_Program* partner
int tag=99, nelems=10;
float data[10];
IC_Bcast_local_float(partner, data, nelems, tag);
```

3.3.7 IC_Recv_local_TYPE

This function is used to receive a data block broadcast from a task in a partner program. Since the underlying message passing system broadcast function is used directly to perform the communication, no communication schedule is necessary. Correct use of this function requires that it must be invoked by *all* tasks in the receiving program. There is a receive function for each supported basic data TYPE.

Synopsis

Fortran 90 IC_Recv_local(from, data, nelems, tag, status)
Note that the Fortran 90 IC_Recv_local function is polymorphic, i.e., it does not require calling a particular version depending on the type of the **data** being received as do the C and Fortran 77 counterparts.

C int IC_Recv_local_char(IC_Program* from, char* data, int nelems, int tag)

Fortran IC_Recv_local_char(from, data, nelems, tag, status)
integer from, nelems, tag, status
integer*1 data(*)

C int IC_Recv_local_short(IC_Program* from, short* data, int nelems, int tag)

Fortran IC_Recv_local_short(from, data, nelems, tag, status)
integer from, nelems, tag, status
integer*1 data(*)

C int IC_Recv_local_int(IC_Program* from, int* data, int nelems, int tag)

Fortran IC_Recv_local_int(from, data, nelems, tag, status)
integer from, nelems, tag, status
integer*1 data(*)

C int IC_Recv_local_float(IC_Program* from, float* data, int nelems, int tag)

Fortran IC_Recv_local_float(from, data, nelems, tag, status)
integer from, nelems, tag, status
integer*1 data(*)

C int IC_Recv_local_double(IC_Program* from, double* data, int nelems, int tag)

Fortran IC_Recv_local_double(from, data, nelems, tag, status)
integer from, nelems, tag, status
integer*1 data(*)

Parameters

from the sending program

data (pointer to) the local array

nelems the number of elements in the local array

tag a message tag used to identify this communication operation

Return value

status, -1 indicates error

Example

```
IC_Program* partner
int tag=99, nelems=10;
float data[10];
IC_Recv_local_float(partner, data, nelems, tag);
```

3.4 Releasing Library Resources

3.4.1 IC_Free_program

This function releases memory used for holding an InterComm application descriptor and disconnects the program from the underlying communication infrastructure.

Synopsis

C void IC_Free_program(IC_Program* prog)

Fortran IC_Free_program(prog)
integer prog

Parameters

prog an InterComm data type representing a partner program

Return value

none

Example

```
IC_Free_program(prog);
```

3.4.2 IC_Free_desc

This function releases memory used for an InterComm array descriptor.

Synopsis

C void IC_Free_desc(IC_desc* desc)

Fortran IC_Free_desc(desc)
integer desc

Parameters

desc an InterComm distributed array descriptor

Return value

none

Example

```
IC_Free_desc(desc);
```

3.4.3 IC_Free_region

This function is used to release memory for holding an InterComm region descriptor.

Synopsis

C void IC_Free_region(IC_region* region)

Fortran IC_Free_region(region)
integer region

Parameters

region an InterComm region descriptor representing an array block

Return value

none

Example

```
IC_Free_region(region);
```

3.4.4 IC_Free_sched

This function releases memory for holding an InterComm communication schedule.

Synopsis

C void IC_Free_sched(IC_Sched* sched)

Fortran IC_Free_sched(sched)
integer sched

Parameters

sched an InterComm communication schedule

Return value

none

Example

```
IC_Free_sched(sched);
```

3.4.5 IC_Quit

Shuts down the InterComm communication subsystem and frees the local program data structure.

Synopsis

C int IC_Quit(IC_Program* myprog)

Fortran IC_Quit(myprog, status)
integer myprog, status

Parameters

myprog the local program

Return value

status, -1 indicates an error

Example

```
int sts;
sts = IC_Quit(myprog);
```

4 Low-Level XJD-based Programming Tasks

This section describes how to use the C and Fortran interfaces that employ an XML Job Description (XJD). This interface makes an InterComm program more like a component by utilizing an externally defined XJD. The XJD describes programs and connections between programs, using program and region names (identified by strings) that are defined by each InterComm program. The XJD provides the configuration information needed to allow programs to communicate with each other, thereby making each program independent from the programs it communicates with, potentially increasing the potential for program reuse. For detailed information on the format of the XJD, see Appendix B.

4.1 Initializing the Library

4.1.1 IC_Initialize

This function initializes the underlying communication library and creates the internal representation of the programs and regions, parsing the given XML Job Description (XJD) This function essentially encapsulates the functionality of the IC_Init, IC_Wait and IC_Sync described in Section 3, using the XJD file to supply the required information about other programs this one is connected to.

Synopsis

C IC_XJD* IC_Initialize(char* progname, int rank, char*xjdname, int* status)

Fortran IC_Initialize(progname, rank, xjdname, xjd_id, status)
character, (len=*) progname
character, (len=*) xjdname
integer rank, xjd_id, status

Parameters

progname my program name

rank my rank

xjdname XJD file name

xjd_id id of the IC_XJD maintained by InterComm (Fortran 77)

status status, if negative indicates an error

Return value

a pointer to an IC_XJD datatype, to contain program and region information

Example

```
IC_XJD* xjd;
int rank, *status;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
xjd = IC_Initialize(‘‘myprog’’, rank, ‘‘./example.xjd’’, status);
```

4.2 Describing the Data Distribution

The data distribution operations ares the same as for the low-level programming API described in Section 3.2.

4.3 Defining and Communicating Array Blocks

The low-level programming API described in Section 3.3 is used to describe array blocks. In addition, this section describes the functions required to register the created array blocks, so that InterComm can automatically build communication schedules for those regions as required by the connection specification in the XJD file, and allow the user to subsequently transfer regions between programs using InterComm.

4.3.1 IC_Register_region

Register detailed information on a set of regions (a set of array blocks) into the InterComm internal representation.

Synopsis

C int IC_Register_region(IC_XJD* xjd, IC_Region** rgnset, int set_size, char* set_name, IC_Desc* desc, void* local_data, int* status)

Fortran IC_Register_region(xjd, rgnset, set_size, set_name, desc, localdata, status)
integer xjd, rgnset, set_size
character, (len=*) set_name
integer desc
void* localdata
integer status

Parameters

xjd the InterComm descriptor for programs and regions, returned from the call to IC_Initialize

rgnset an array of regions describing the data to be communicated

set_size the number of regions in the array

set_name name of the region set (a string)

desc the InterComm global array descriptor

localdata pointer to the local data array

status status, if negative indicates an error

Return value

none

Example

```
int* status;
IC_Region* rgnset[2];
float* data = &A[0][0][0];
IC_Register_region(xjd, rgnset, 2, ‘‘rgnset1’’, desc, data, status);
```

4.3.2 IC_Commit_region

Informs InterComm that all regions have been defined, so that communication schedules can be built for all the connections this program participates in, and store the schedules into the internal program representation.

Synopsis

C void IC_Commit_region(IC_XJD* xjd, int* status)

Fortran IC_Commit_region(xjd, status)
integer xjd, status

Parameters

xjd the InterComm program descriptor for programs and regions

status status, if negative indicates an error

Return value

none

Example

```
int* status;
sts = IC_Commit_region(xjd, status);
```

4.3.3 IC_Export

Export a region. Note that this function does not need the destination program name, as is needed for the low-level API described in Section 3, since that comes from the connection information in the XJD file, acquired by InterComm in the call to IC_Initialize. InterComm exports the specified region into one or more destination programs as specified in the XJD. If the XJD does not specify any connections for the specified region, so that no program will import the region, the function becomes a no-op.

Synopsis

C int IC_Export(IC_XJD* xjd, char* rgnset_name)

Fortran IC_Export(xjd, rgnset_name, status)
integer xjd, status
character(*) rgnset_name

Parameters

xjd the InterComm program descriptor for programs and regions

rgnset_name name of the region set (a string)

Return value

status, -1 indicates error

Example

```
IC_Export(xjd, ‘‘rgnset1’’);
```

4.3.4 IC_Import

Import a region. This function also does not need the source program name for the import operation, as for IC_Export, since that is specified in the XJD file. If no connection is specified for the imported region in the XJD, an error will be reported.

Synopsis

C int IC_Import(IC_XJD* xjd, char* rgnset_name)

Fortran IC_Import(xjd, rgnset_name, status)
integer xjd, status
character(*) rgnset_name

Parameters

xjd the InterComm program descriptor for programs and regions

rgnset_name name of the region set

Return value

status, -1 indicates error

Example

```
IC_Import(xjd, ‘‘rgnset1’’);
```

4.3.5 IC_Bcast_local

Broadcast a memory block to all partner program processes. Strictly speaking, the memory block is different from a normal region, which is defined as part of a distributed array in all processes of a program. However, the block must be named in the XJD file, and have commtype *1xN* in the XJD. In that way, InterComm can determine the partner program associated with the broadcast. Broadcast communication for the block does not need a communication schedule, so the user does not need to register the block. Note that only *one*

process (the process with rank 0) in the sender program performs the broadcast, while *all* processes in the receiver program perform the corresponding receive operation. As IC_Export does, InterComm broadcasts the specified region into one or more destination programs as specified in the XJD. If the XJD does not specify any connections for the specified region, so that no program will receive the region, the function becomes a no-op.

Synopsis

C int IC_Bcast_local(IC_XJD* xjd, char* rgnset_name, void* data, int nelems)

Fortran IC_Bcast_local(xjd, rgnset_name, data, nelems, status)
integer xjd, nelems, status
character(*) rgnset_name
void* data

Parameters

xjd the InterComm application descriptor for programs and regions

rgnset_name name of the region set

data pointer to the memory block to broadcast

nelems the number of elements in the block

Return value

status, -1 indicates error

Example

```
int* status;
int bcast[1] = {10};
IC_Bcast_local(xjd, ‘‘rgnset1’’, bcast, 1);
```

4.3.6 IC_Recv_local

Receive a memory block from a process in a partner program. Strictly speaking, the memory block is different from a normal region, which is defined as part of a distributed array in all processes of a program. However, the block must be named in the XJD file, and have commtype *1xN*. In that way, InterComm can determine the partner program associated with the broadcast. Broadcast communication for the block does not need a communication schedule, so the user does not need to register the block. Note that only *one* process in the sender program performs the broadcast, while *all* processes in the receiver program perform the corresponding receive operation. If no connection is specified for the received region in the XJD, an error will be reported.

Synopsis

C int IC_Recv_local(IC_XJD* xjd, char* rgnset_name, void* data, int nelems)

Fortran IC_Recv_local(xjd, rgnset_name, data, nelems, status)
integer xjd, nelems, status
character(*) rgnset_name
void* data

Parameters

xjd the InterComm application descriptor for programs and regions

rgnset_name name of the region set

data pointer to the memory block to receive into

nelems the number of elements in the block

Return value

status, -1 indicates error

Example

```
int brecv[10];
IC_Recv_local(xjd, ‘‘rgnset1’’, brecv, 10);
```

4.4 Releasing Library Resources

4.4.1 IC_Finalize

Release the allocated memory for the internal representation of programs and regions.

Synopsis

C void IC_Finalize(IC_XJD* xjd, int* status)

Fortran IC_Finalize(xjd, status)
integer xjd, status

Parameters

xjd the InterComm application descriptor for programs and regions

status status, if negative indicates an error

Return value

status, if negative indicates an error

Example

```
int* status;
IC_Finalize(xjd, status);
```

5 High-Level Programming Tasks

In some cases, the utilization of InterComm can be simplified by relying on a few higher-level function calls. The functions described in Section 3 provide a generic way for initializing and finalizing InterComm, for defining array decompositions, and for establishing communication between a pair of programs, among other tasks. On the other hand, some applications can make use of a simplified interface that encapsulates most of the complexities of InterComm. However, this high-level API, albeit simpler, does not provide as much flexibility as the low-level interface. For example, in order to use the high-level interface, array distributions must be one of a small number *canonical* distributions that will be described in this section. The high-level API is available for C++ programs relying on the P++ library and also for *sequential* Fortran 90 programs.

5.1 Creating Endpoints

The high-level API relies on the concept of a *communication endpoint*. The endpoint is an abstraction that corresponds to a communication channel between a pair of programs that need to exchange entire arrays or array sections. The establishment of a communication endpoint is the first step to ensure that data can flow between a pair of programs.

5.1.1 EndPoint(Constructor)

Synopsis

C++ IC_EndPoint::IC_Endpoint(const char* endpointName, const unsigned mynproc, const unsigned onproc, const unsigned myao, const unsigned oao, int& status)

Fortran 90 ic_endpoint_constructor(icce, endpointName, mynproc, onproc, myao, oao, status)
ic_obj icce
character, (len=*) endpointName
integer mynproc, onproc, myao, oao, status

Parameters

icce *returns* a handle to the InterComm communication endpoint descriptor

endpointName the name for the endpoint
This must be in the format *first program* : *second program* (e.g., `simulation1:simulation2`). The Fortran 90 version requires explicitly appending a CHAR(0) to the end of the string.

mynproc the number of processes used by the local program

onproc the number of processes used by the remote (the other) program

myao the array *ordering* used by the local program
The valid inputs for this parameter are `IC_ROW_MAJOR` and `IC_COLUMN_MAJOR`.

oao the array *ordering* used by the remote (the other) program
The valid inputs for this parameter are `IC_ROW_MAJOR` and `IC_COLUMN_MAJOR`.

status *returns* the result of the endpoint creation operation

This result should always be checked to ensure that the endpoint is in a sane state. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *constructor call*. The Fortran version returns a *reference* to the endpoint in its *icce* parameter. Both calls return the status of the operation in the *status* parameter.

Example

C++:

```
IC_EndPoint right_left_ep("right:left",1,1,
    IC_EndPoint::IC_COLUMN_MAJOR,IC_EndPoint::IC_COLUMN_MAJOR,ic_err);
```

Fortran 90:

```
type(ic_obj) :: icce
call ic_endpoint_constructor(icce,'left:right'//CHAR(0),1,1,
    IC_COLUMN_MAJOR,IC_COLUMN_MAJOR,ic_err)
```

5.2 Communicating Array Sections

Once the communication endpoint is created, the two applications can start exchanging data, by exporting and importing arrays or arrays subsections.

5.2.1 ExportArray

Synopsis

C++ IC_EndPoint::exportArray(const intArray& array, int msgtag, int& status)

Fortran 90 ic_export_array(icce, array, msgtag, status)
ic_obj icce
integer, dimension (:) array
integer msgtag
integer status

C++ IC_EndPoint::exportArray(const floatArray& array, int msgtag, int& status)

Fortran 90 ic_export_array(icce, array, msgtag, status)
ic_obj icce
real, dimension (:) array
integer msgtag
integer status

C++ IC_EndPoint::exportArray(const doubleArray& array, int msgtag, int& status)

Fortran 90 ic_export_array(icce, array, msgtag, status)
ic_obj icce
double precision, dimension (:) array
integer msgtag
integer status

Parameters

icce the exporting InterComm communication endpoint handle

array the array or array section to be transfered

The C++ method and the Fortran 90 function call are both *polymorphic*, which means that the operation is correctly performed regardless of the type of the array elements.

msgtag the message tag for exporting a region set

status *returns* the result of the export operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation by setting the *status* parameter.

Example

C++:

```
doubleArray DOUBLES(10,10);
Index I(3,6), J(1,4);
int msgtag = 2001;
right_left_ep.exportArray(DOUBLES(J,I), msgtag, ic_err);
```

Fortran 90:

```
double precision, dimension (10,10) :: DOUBLES
integer msgtag
call ic_export_array(icce,DOUBLES(2:5,4:9), msgtag, ic_err);
```

5.2.2 ImportArray

If one application is exporting data, the other (remote) application is importing data.

Synopsis

C++ IC_EndPoint::importArray(const intArray& array, int msgtag, int& status)

Fortran 90 ic_import_array(icce, array, msgtag, status)
ic_obj icce
integer, dimension (:) array
integer msgtag
integer status

C++ IC_EndPoint::importArray(const floatArray& array, int msgtag, int& status)

Fortran 90 ic_import_array(icce, array, msgtag, status)
ic_obj icce
real, dimension (:) array
integer msgtag
integer status

C++ IC_EndPoint::importArray(const doubleArray& array, int msgtag, int& status)

Fortran 90 ic_import_array(icce, array, msgtag, status)
ic_obj icce
double precision, dimension (:) array
integer msgtag
integer status

Parameters

icce the importing InterComm communication endpoint handle

array the array or array section to be received
The C++ method and the Fortran 90 function call are both *polymorphic*, which means that the operation will be correctly performed regardless of the type of the array elements.

msgtag the message tag for importing a region set

status *returns* the result of the import operation
This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation in the *status* parameter.

Example

C++:

```
floatArray FLOATS(10);
Index I(3,6);
int msgtag = 2001;
right_left_ep.importArray(FLOATS(I), msgtag, ic_err);
```

Fortran 90:

```
real, dimension (10) :: FLOATS
integer msgtag
call ic_import_array(icce,FLOATS(2:5), msgtag, ic_err);
```

5.2.3 Broadcast Array

Broadcast a local array to all processes of the other (remote) application. This method broadcast an array by directly invoking the message passing system broadcast API underlying all InterComm data movement calls. Thus a communication schedule is unnecessary.

Synopsis

C++ IC_EndPoint::bcastLocalArray(const intArray& array, int nelems, int msgtag, int& status)

Fortran 90 ic_bcast_local_array(icce, array, nelems, msgtag, status)
ic_obj icce
integer, dimension (:) array
integer nelems, msgtag
integer status

C++ IC_EndPoint::bcastLocalArray(const floatArray& array, int nelems, int msgtag, int& status)

Fortran 90 ic_bcast_local_array(icce, array, nelems, msgtag, status)
ic_obj icce
real, dimension (:) array
integer nelems, msgtag
integer status

C++ IC_EndPoint::bcastLocalArray(const doubleArray& array, int nelems, int msgtag, int& status)

Fortran 90 ic_bcast_local_array(icce, array, nelems, msgtag, status)
ic_obj icce
double precision, dimension (:) array
integer nelems, msgtag
integer status

Parameters

icce the broadcasting InterComm communication endpoint handle

array the local array or array section to broadcast

The C++ method and the Fortran 90 function call are both *polymorphic*, which means that the operation will be correctly performed regardless of the type of the array elements.

nelems the number of elements in the array

msgtag the message tag for importing a region set

status *returns* the result of the import operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation in the *status* parameter.

Example

C++:

```
floatArray FLOATS(10);
right_left_ep.bcastLocalArray(FLOATS, 10, 2006, ic_err);
```

Fortran 90:

```
real, dimension (10) :: FLOATS
call ic_bcast_local_array(icce, FLOATS, 10, 2006, ic_err);
```

5.2.4 Receive Broadcast Array

Receive a broadcast array from a (remote) application. This method receives an array by invoking the message passing system receive API underlying all InterComm data movement calls. Thus a communication schedule is unnecessary.

Synopsis

C++ IC_EndPoint::recvLocalArray(const intArray& array, int nelems, int msgtag, int& status)

Fortran 90 ic_recv_local_array(icce, array, nelems, msgtag, status)
ic_obj icce
integer, dimension (:) array
integer nelems, msgtag
integer status

C++ IC_EndPoint::recvLocalArray(const floatArray& array, int nelems, int msgtag, int& status)

Fortran 90 ic_recv_local_array(icce, array, nelems, msgtag, status)
ic_obj icce
real, dimension (:) array
integer nelems, msgtag
integer status

C++ IC_EndPoint::recvLocalArray(const doubleArray& array, int nelems, int msgtag, int& status)

Fortran 90 ic_recv_local_array(icce, array, nelems, msgtag, status)
ic_obj icce
double precision, dimension (:) array
integer nelems, msgtag
integer status

Parameters

icce the broadcasting InterComm communication endpoint handle

array the local array or array section to broadcast

The C++ method and the Fortran 90 function call are both *polymorphic*, which means that the operation will be correctly performed regardless of the type of the array elements.

nelems the number of elements in the array

msgtag the message tag for importing a region set

status *returns* the result of the import operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation in the *status* parameter.

Example

C++:

```
floatArray FLOATS(10);
right_left_ep.recvLocalArray(FLOATS, 10, 2006, ic_err);
```

Fortran 90:

```
real, dimension (10) :: FLOATS
call ic_recv_local_array(icce, FLOATS, 10, 2006, ic_err);
```

5.3 Termination

When the pair of applications reach a point where data is no longer being exchanged, both applications are expected to destroy their end of the communication endpoint to ensure a clean shutdown of InterComm and also of the underlying communication infrastructure.

5.3.1 EndPoint(Destructor)

Synopsis

C++ IC_EndPoint::~ IC_Endpoint()

Fortran 90 ic_endpoint_destructor(icce)
ic_obj icce

Parameters

icce the InterComm endpoint to be shutdown

Return value

Note that the C++ call is an object destructor. It need not be called explicitly. In reality, the destructor is automatically called for deallocating communication endpoint objects statically created. The application should invoke the *delete* C++ operator to ensure that the destructor properly finalizes InterComm.

Example

C++:

```
IC_EndPoint* right_left_ep = new IC_EndPoint("right:left",1,1,
    IC_EndPoint::IC_COLUMN_MAJOR,IC_EndPoint::IC_COLUMN_MAJOR,ic_err);
delete right_left_ep; // indirectly invoking the object destructor
```

Fortran 90:

```
type(ic_obj) :: icce
call ic_endpoint_destructor(icce)
```

5.4 Error Codes

All the high-level InterComm calls, with the exception of the destructor call, return the status of the operation. For successful operations `IC_OK` is returned. For unsuccessful operations, a variety of error codes can be returned. A list of all possible return values is shown in Table 1.

InterComm provides an auxiliary function to help application developers handle erroneous operations. The following function call/method invocation can be employed to print out a message stating the nature of the error condition:

5.4.1 PrintErrorMessage

Synopsis

C++ IC_EndPoint::printErrorMessage(const char* msg, int& status)

Fortran 90 ic_print_error_message(msg, status)
character (len=*) msg
integer status

Parameters

msg an error message
The msg string will precede the text for the error code. For the Fortran 90 call the string must have a CHAR(0) as the last character.

status the error code for which the error message will be printed out

Return value

Error Condition	Error Message
IC_OK	no error
IC_GENERIC_ERROR	generic error
IC_INVALID_NDIM	invalid number of array dimensions
IC_CANT_ALLOC_REGION	can't allocate InterComm array regions
IC_CANT_GET_DA_DESCRIPTOR	can't obtain distributed array descriptor
IC_CANT_COMPUTE_COMM_SCHEDULE	can't compute communication schedule
IC_COMM_FAILURE	communication (send/recv) failure
IC_INVALID_ENDPOINT_NAME	invalid endpoint name
IC_INITIALIZATION_FAILURE	local program initialization failed
IC_CANT_CONNECT_TO_REMOTE	can't connect to remote program
IC_PVM_ERROR	PVM error
IC_MPI_ERROR	MPI error
IC_POINTER_TABLE_ERROR	internal pointer translation table error – possibly too many InterComm descriptors (regions, distributions, etc) have been defefined

Table 1: InterComm high-level API errors

The C++ call is C++ *void* method invocation. The Fortran version also does not return a value. The function prints out a message warning the user that the error code is invalid, if *status* does not hold any of the values displayed in Table 1.

Example

C++:

```
right_left_ep.printErrorMessage("API call failed",ic_err);
```

Fortran 90:

```
call ic_print_error_message('API call failed'//CHAR(0),ic_err);
```

6 High-Level XJD-based Programming Tasks

The high-level API described in Section 5 provides a simplified interface that encapsulates InterComm complexities, and it allows users to transfer any array or subarray at any program execution point. However, it makes programs strongly tied each other, and therefore they must be rewritten to be coupled with other programs.

In this section, we describes an XJD-based high-level API that keeps the simplicity of the high-level API and also utilizes the XJD (XML Job Description) to componentize a program with an externally visible program

name and data port names, and to allow the programs to exchange data as described for the low-level XJD-based API in Section 4. Using this API, an InterComm program can register arrays and import/export the arrays between pairs of programs.

However, this API has same constraints that the High-level API in Section 5 has. In addition, all arrays in a program must be registered and committed before the actual data import/export calls can be used to move data between the programs. The XJD-based high-level API is available for C++ programs relying on the P++ library and also for *sequential* Fortran 90 programs.

6.1 Creating EndpointSet

The XJD-based high-level API utilizes the concept of an endpointset. It encapsulates multiple endpoints, which a program takes part in as an exporter or importer for each array. Through the information provided by a given XJD to run a set of coupled programs, multiple endpoints are established for each program depending on how many programs it exchanges data with.

6.1.1 EndPointSet(Constructor)

Synopsis

C++ IC_EndPointSet::IC_EndPointSet(const char* xjdfile, const char* compname, int& status)

Fortran 90 ic_endpointset_constructor(icepset, xjdfile, compname, status)
ic_obj icepset
character, (len=*) xjdfile
character, (len=*) compname
integer status

Parameters

icepset a handle to the set of endpoints

xjdfile the XJD file name
The Fortran 90 version requires explicitly appending a CHAR(0) to the end of the string.

compname my program name

status *returns* the result of the set creation operation
This result should always be checked to ensure that the set is in a sane state. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *constructor call*. The Fortran version returns a *reference* to the set in its *icepset* parameter. Both calls return the status of the operation in the *status* parameter.

Example

C++:

```
IC_EndPointSet icepset("test.xjd", "comp1", ic_err);
```

Fortran 90:

```
type(ic_obj) :: icepset
call ic_endpointset_constructor(icepset,'test.xjd'//CHAR(0), &
                                'comp1'//CHAR(0), ic_err)
```

6.2 Registering and committing arrays

After communication endpoints are created, the regions to communicate between programs must be registered and committed before data communication.

6.2.1 RegisterArray

Synopsis

C++ IC_EndPointSet::registerArray(const char* arrname, const intArray& array, int& status)

Fortran 90 ic_register_array(icepset, arrname, array, status)
ic_obj icepset
character, (len=*) arrname
integer, dimension(:) array
integer status

C++ IC_EndPointSet::registerArray(const char* arrname, const floatArray& array, int& status)

Fortran 90 ic_register_array(icepset, arrname, array, status)
ic_obj icepset
character, (len=*) arrname
real, dimension(:) array
integer status

C++ IC_EndPointSet::registerArray(const char* arrname, const doubleArray& array, int& status)

Fortran 90 ic_register_array(icepset, arrname, array, status)
ic_obj icepset
character, (len=*) arrname
double, dimension(:) array
integer status

Parameters

icepset the handle to the set of endpoints

arrname the array name (this string must be used in the XJD)
The Fortran 90 version requires explicitly appending a CHAR(0) to the end of the string.

array the array or array section to be transferred

The C++ method and the Fortran 90 function call are both *polymorphic*, which means that the operation is correctly performed regardless of the type of the array elements.

status *returns* the result of the array registration operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation by setting the *status* parameter.

Example

C++:

```
floatArray FLOATS(10);
Index I(3,6);
epset.registerArray("array1", FLOATS(I), ic_err);
```

Fortran 90:

```
real, dimension (10) :: FLOATS
call ic_register_array(icepset, 'array1'//CHAR(0), FLOATS(2:6), ic_err)
```

6.2.2 CommitArrays

This method will generate the communication schedules for all registered arrays. After registering all arrays necessary for a program, the arrays must be committed by invoking this method.

Synopsis

C++ IC_EndPointSet::commitArrays(int& status)

Fortran 90 ic_commit_arrays(icepset, status)
ic_obj icepset
integer status

Parameters

icepset a handle to the set of endpoints

status *returns* the result of the set creation operation

This result should always be checked to ensure that the set is in a sane state. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation by setting the *status* parameter.

Example

C++:

```
epset.commitArrays(ic_err);
```

Fortran 90:

```
ic_commit_arrays(icepset, ic_err)
```

6.3 Communicating Array Sections

Once the communication endpoint is created and all arrays are registered and committed, the two applications can start exchanging data, by exporting and importing arrays.

6.3.1 ExportArray

Synopsis

C++ IC_EndPointSet::exportArray(const char* arrname, int& status)

Fortran 90 ic_export_array(icepset, arrname, status)
ic_obj icepset
integer, dimension (:) array
integer status

Parameters

icepset a handle to the set of endpoints

arrname the array name

The Fortran 90 version requires explicitly appending a CHAR(0) to the end of the string.

status *returns* the result of the export operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation by setting the *status* parameter.

Example

C++:

```
epset.exportArray("array1", ic_err);
```

Fortran 90:

```
call ic_export_array(icepset, 'array1'//CHAR(0), ic_err)
```

6.3.2 ImportArray

Synopsis

C++ IC_EndPointSet::importArray(const char* arrname, int& status)

Fortran 90 ic_import_array(icepset, arrname, status)
ic_obj icepset
integer, dimension (:) array
integer status

Parameters

icepset a handle to the set of endpoints

arrname the array name

The Fortran 90 version requires explicitly appending a CHAR(0) to the end of the string.

status *returns* the result of the export operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation by setting the *status* parameter.

Example

C++:

```
epset.importArray("array1", ic_err);
```

Fortran 90:

```
call ic_import_array(icepset, 'array1'//CHAR(0), ic_err)
```

6.3.3 Broadcast Array

Broadcast a local array to all processes of another (remote) application. This method broadcasts an array by directly invoking the message passing system broadcast API underlying all InterComm data movement calls. Thus a communication schedule is unnecessary.

Synopsis

C++ IC_EndPointSet::bcastLocalArray(const char* arrname, const intArray& array, int nelems, int& status)

Fortran 90 ic_bcast_local_array(icepset, arrname, array, nelems, status)
ic_obj icepset
character, (len=*) arrname
integer, dimension(:) array
integer nelems
integer status

C++ IC_EndPointSet::bcastLocalArray(const char* arrname, const floatArray& array, int nelems, int& status)

Fortran 90 ic_bcast_local_array(icepset, arrname, array, nelems, status)
ic_obj icepset
character, (len=*) arrname
real, dimension(:) array
integer nelems
integer status

C++ IC_EndPointSet::bcastLocalArray(const char* arrname, const doubleArray& array, int nelems, int& status)

Fortran 90 ic_bcast_local_array(icepset, arrname, array, nelems, status)
ic_obj icepset
character, (len=*) arrname
double, dimension(:) array
integer nelems
integer status

Parameters

icepset the handle to the set of endpoints

arrname the array name (this string must be used in the XJD)

The Fortran 90 version requires explicitly appending a CHAR(0) to the end of the string.

array the array or array section to be transferred

The C++ method and the Fortran 90 function call are both *polymorphic*, which means that the operation is correctly performed regardless of the type of the array elements.

nelems the number of elements in the array

status *returns* the result of the array registration operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation by setting the *status* parameter.

Example

C++:

```
floatArray FLOATS(10);
epset.bcastLocalArray("array1", FLOATS, 10, ic_err);
```

Fortran 90:

```
real, dimension (10) :: FLOATS
call ic_bcast_local_array(icepset, 'array1'//CHAR(0), FLOATS, 10, ic_err)
```

6.3.4 Receive Broadcast Array

Receive a broadcast array from a (remote) application. This method receives an array by invoking the message passing system receive API underlying all InterComm data movement calls. Thus a communication schedule is unnecessary.

Synopsis

C++ IC_EndPointSet::recvLocalArray(const char* arrname, const intArray& array, int nelems, int& status)

Fortran 90 ic_recv_local_array(icepset, arrname, array, nelems, status)
ic_obj icepset
character, (len=*) arrname
integer, dimension(:) array
integer nelems
integer status

C++ IC_EndPointSet::recvLocalArray(const char* arrname, const floatArray& array, int nelems, int& status)

Fortran 90 ic_recv_local_array(icepset, arrname, array, nelems, status)
ic_obj icepset
character, (len=*) arrname
real, dimension(:) array
integer nelems
integer status

C++ IC_EndPointSet::recvLocalArray(const char* arrname, const doubleArray& array, int nelems, int& status)

Fortran 90 ic_recv_local_array(icepset, arrname, array, nelems, status)
ic_obj icepset
character, (len=*) arrname
double, dimension(:) array
integer nelems
integer status

Parameters

icepset the handle to the set of endpoints

arrname the array name (this string must be used in the XJD)

The Fortran 90 version requires explicitly appending a CHAR(0) to the end of the string.

array the array or array section to be transferred

The C++ method and the Fortran 90 function call are both *polymorphic*, which means that the operation is correctly performed regardless of the type of the array elements.

nelems the number of elements in the array

status *returns* the result of the array registration operation

This result should always be checked to ensure that the operation was correctly performed. In case of success, status is set to `IC_OK`.

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. Both calls return the status of the operation by setting the *status* parameter.

Example

C++:

```
floatArray FLOATS(10);
epset.recvLocalArray("array1", FLOATS, 10, ic_err);
```

Fortran 90:

```
real, dimension (10) :: FLOATS
call ic_recv_local_array(icepset, 'array1'//CHAR(0), FLOATS, 10, ic_err)
```

6.4 Termination

After a program completes all communication with other programs, it must destroy all endpoints to ensure a clean termination of InterComm.

6.4.1 EndPointSet(Destructor)

Synopsis

C++ IC_EndPointSet::~ IC_EndpointSet()

Fortran 90 ic_endpointset_destructor(icepset)
ic_obj icepset

Parameters

icepset a handle to the set of endpoints

Return value

Note that the C++ call is an object destructor. It need not be called explicitly. The destructor is automatically called for deallocating communication endpoints for the statically created endpoint set.

Example

C++:

```
IC_EndPointSet epset("test.xjd","comp1", ic_err);
delete epset; // indirectly invoking the object destructor
```

Fortran 90:

```
type(ic_obj) :: icepset
call ic_endpointset_destructor(icepset)
```

6.5 Error Codes

All the XJD-based high-level InterComm calls, with the exception of the destructor call, return the status of the operation. For successful operations `IC_OK` is returned. For unsuccessful operations, a variety of error codes can be returned. In addition to the errors define for the high-level API described in Table 1, Table 2 shows additional errors defined for the XJD-based high-level API.

InterComm provides an auxiliary function to help application developers handle erroneous operations. The following function call/method invocation can be employed to print out a message stating the nature of the error condition.

6.5.1 PrintErrorMessage

Synopsis

C++ IC_EndPointSet::printErrorMessage(const char* msg, int& status)

Error Condition	Error Message
IC_EPSET_INIT_FAILURE	failure to initialize EndPointSet from XJD
IC_ARRAY_REGISTER_FAILURE	failure to register an array
IC_COMMIT_ARRAY_FAILURE	failure to commit the registered arrays

Table 2: InterComm XJD-based high-level API errors

Fortran 90 ic_print_error_message(msg, status)
character (len=*) msg
integer status

Parameters

msg an error message

This string will precede the actual text corresponding to the error code. For the Fortran 90 call the string must have a CHAR(0) as the last character.

status the error code for which the error message will be printed out

Return value

The C++ call is a C++ *void* method invocation. The Fortran version also does not return a value. The function prints out a message warning the user that the error code is invalid, if *status* does not hold any of the values displayed in Table 1.

Example

C++:

```
ep.printErrorMessage("API call failed", ic_err);
```

Fortran 90:

```
call ic_print_error_message('API call failed'//CHAR(0), ic_err)
```

7 Compilation and Program Startup

This section contains some very basic guidelines for compiling and running coupled InterComm applications. More detailed information for a particular platform can be gleaned from examining the generated Makefiles in the `examples` directory of the distribution.

7.1 Compiling

The following example commands show the libraries needed for an InterComm application written in one of the supported languages. The compiler names used are chosen to be generic, they may be different for a particular system or communication library (i.e. MPI).

```
cc -o cexample cexample.c -lIC -lgpvm3 -lpvm3
f77 -o fexample fexample.f -lICf77 -lIC -lfpvm3 -lgpvm3 -lpvm3
c++ -o pppexample pppexample.cpp -lICppp -lIC -lgpvm3 -lpvm3 -lPpp -lPpp_static
f90 -o f90example f90example.f90 -lICf90 -lIC -lfpvm3 -lgpvm3 -lpvm3
```

Depending on your compiler you may need to link in the math library (`-lm`) as well.

Use of the MultiBlockParti or Chaos descriptor translation functions requires linking additional libraries, (`-lICmbp` and `-lICchaos`, respectively).

7.2 Running

InterComm depends upon PVM for inter-program communication, so the first step for any successful coupling is to start the PVM daemon on all the target machines. For this particular example, it is assumed that there is no scheduler or access constraints for the hosts on which the programs will run and that these hosts have been listed in a file called `hosts`.

```
echo ‘‘quit’’ | pvm hosts
mpirun -np 4 cexample
mpirun -np 8 fexample
echo ‘‘halt’’ | pvm
```

More complicated scenarios for starting the virtual machine are outside of the scope of this document and are best handled by consulting the PVM documentation. However, some scripts for working around the constraints of IBM's LoadLeveler[8] scheduler can be found in the `scripts` directory.

A Utility Functions

The following sections provide utility functions for interfacing with external libraries.

A.1 Descriptor Translation Functions

A.1.1 IC_Translate_parti_descriptor

If support for MultiBlockParti is enabled during configuration (Section 2), inclusion of the header file `mbp2bdecomp.h` will allow the use a translation function for converting MultiBlockParti `DARRAY` descriptors to InterComm block decomposition descriptors.

Synopsis

C IC_Desc* IC_Translate_parti_descriptor(DARRAY* darray)

Fortran IC_Translate_parti_descriptor(darray, desc)
integer darray, desc

Parameters

darray a MultiBlockParti distributed array descriptor

Return value

an InterComm array descriptor data type

Example

```
DARRAY* darray;
IC_Desc* desc;

/* ...create DARRAY using MultiBlockParti calls... */
desc = IC_Translate_parti_desciptor(darray);
```

A.1.2 IC_Translate_chaos_descriptor

If support for Chaos is enabled during configuration (Section 2), inclusion of the header file `chaos2ttable.h` will allow the use of a translation function for converting Chaos `TTABLE` descriptors to InterComm translation table descriptors.

Synopsis

C IC_Desc* IC_Translate_chaos_descriptor(TTABLE* ttable)

Fortran *There is no Fortran interface to Chaos*

Parameters

ttable a Chaos translation table descriptor

Return value

an InterComm array descriptor data type

Example

```
TTABLE* ttable;
IC_Desc* desc;

/* ...create TTABLE using Chaos calls... */
desc = IC_Translate_chaos_desciptor(ttable);
```

B XML Job Description and HPCALE

This section describes the structure of the XML Job Description (XJD) and introduces the High-Performance Computing Application Launching Environment (HPCALE), which is an environment to launch multiple high-performance computing applications on multiple resources with minimal user intervention.

B.1 XML Job Description

An XJD (XML Job Description) describes a configuration for a coupled simulation that consists of multiple data parallel programs. In an XJD, the programs used in a coupled simulation are described in the *components* section, and the data communication patterns between the program pairs are described in the *connections* section.

Decoupling such information into a separate description file helps users to write each program more independently from the programs that it may be coupled with. Moreover, since the information for program coupling is decoupled from the source code, a program can be reused without code modification or with minimal code modification for different coupled simulations.

Every program implemented using the XJD-based programming API, as described in Sections 4 and 6, requires an XJD for InterComm initialization. HPCALE also uses an XJD to launch components using additional information that must be added in the *components* section for each component. The following example shows the overall XJD structure.

Example

```
<?xml version="1.0" encoding="utf-8"?>
<ICXJD>
    <version>1.5</version>
    <components>
        <component>...</component>
        <component>...</component>
    </components>
    <connections>
```

```
        <connection>...</connection>
    </connections>
</ICXJD>
```

InterComm provides different internal implementations for the same function call based on the XJD *version*. Thus, user must specify the correct InterComm version for the components in an XJD are supposed to use. The only current legal value for the version is 1.5.

B.1.1 Component

The term *component* is used in an XJD instead of *program* because an InterComm program behaves much like a (parallel) software component when it is implemented using the XJD-based programming API. Each component exposes a component name and the names of the regions it defines for the data communication. The entries for a component section and an example is given below. Note that this is the minimal information for a component description. To launch a component on a resource via HPCALE, more entries must be specified for a component. The cluster name where a component is to be run is one such entry.

Elements

id the unique identifier for a component

name the externally visible component name (should match the program name given in an IC_Initialize() call or IC_EndPointSet constructor)

nNode the number of processes a component runs on

arrayorder the memory ordering of the arrays for a component. IC_ROW_MAJOR or IC_COLUMN_MAJOR are the two possibilities.

Example

```
<component>
    <id>component1</id>
    <name>RECEIVER</name>
    <nNode>8</nNode>
    <arrayorder>IC_ROW_MAJOR</arrayorder>
</component>
```

B.1.2 Connection

A *connection* specifies how to match regions in two components. In the IC_Initialize() call or IC_EndPointSet constructor, each component determines how to communicate with other components using the XJD connection information. In the IC_Commit_region() call a communication schedule is built for each connection, and then components can communicate with each other via IC_export() and IC_import() calls. Note that InterComm cannot completely verify the syntactic and semantic correctness of the connection information,

but does checks for several types of semantic errors in an XJD. For example, if there are multiple connections with the same importer and different exporters, InterComm will report an error since there must be one source for any imported object.

For a multicast operation (i.e. an exported object in one program that is imported by multiple importing programs), multiple *connection*s are needed, one for each path. For example, if component A should send a data object to components B, C, and D, three different *connection*s, one for A-B, one for A-C and one for A-D are needed. All the *connections* should have the same *exporter* and *exportport* specification, and different *importer*, *importport*, and *msgtag* specifications.

Elements

id the unique identifier for a connection

type a basic datatype for the data object being transferred (legal values are **char**, **int**, **float**, and **double**)

commtype a communication type for a region. **MxN** is used for data redistribution between two (parallel) components, and **1xN** for broadcasting an array.

msgtag a unique message tag (an integer between 0 and 65535) for a connection between the exporter and importer

exporter the exporter component name (should match a component ID in the component section of the XJD)

exportport a region name for the exporter component (should match the a name registered for an object in the exporter program)

importer the importer component name (should match a component ID in the component section of the XJD)

importport a region name for the importer component (should match the a name registered for an object in the importer program)

Example

```
<connection>
    <id>conn1</id>
    <type>float</type>
    <commtype>MxN</commtype>
    <msgtag>100</msgtag>
    <exporter>comp1</exporter>
    <exportport>port1</exportport>
    <importer>comp2</importer>
    <importport>port1</importport>
</connection>
```

B.2 High-Performance Computing Application Launching Environment (HPCALE)

High-Performance Computing Application Launching Environment (HPCALE) provides a convenient environment for launching complex applications, such a set of coupled InterComm programs, on one or more computational resources.

To achieve this, an XJD must be located in a directory an InterComm program can access. Thus, to launch programs installed on multiple resources, the XJD must be placed in the correct file system locations on the resources where each program is launched. In addition, several entries are needed in the component section of the XJD. For example, the *cluster* and *node* entries must be specified to let HPCALE know where to launch a component, and on how many nodes.

Such information can be retrieved from the HPCALE information repository by registering the information before launching the components. Otherwise, the user can specify the information directly in the XJD (and that will override any information in the repository). The additional entries HPCALE requires for each component are listed below. See the HPCALE manual [9] for the complete description for the entries.

Elements

cluster the computational resource to launch a component on

argument program arguments for the component

sendlist the list of input files that must be transferred to the resource a component is launched on

recvlist the list of output files that must be transferred back from the resource to the launching site

launchprog external launching tool for a component (e.g., *mpirun*, *mpiexec*)

launcharg the arguments for the launching program if a launching program such as *mpirun* or *mpiexec* is used for a component

xjdoption the XJD options defined by the component when it accepts an XJD name as a command-line argument with an optional delimiter

Although writing an XJD is not very complex, to create an XJD a user must have information about available resources and the component details. HPCALE provides a Web-based tool to help create an XJD for components and connections. The tool utilizes information pre-registered into the HPCALE information repository (likely by owners of the computational resources used). Users can employ the tool to create an XJD. The created XJD will have all the information needed for launching a set of coupled components, and HPCALE will use the XJD to launch components on one or more computational resources. More details on using HPCALE can be found in [9].

C Example Code

There are several example programs in the `example` directory of the distribution. They demonstrate the basic InterComm functionality needed to couple a pair of parallel programs. There are examples written in both C and Fortran, each using either PVM or MPI to acheive parallelism. The files in this directory include:

- `Makefile`
  This builds the various codes and can be used as a template for other InterComm projects.
- `cpvmexample.c`
  This code uses a translation table type data distribution and is meant to communicate with either the fpvmexample or the cmpiexample.
- `fpvmexample.f`
  This code uses a block decomposition data distribution and is meant to communicate with either the fmpiexample or the cpvmexample.
- `cmpiexample.c`
  This code uses a block decomposition data distribution and is meant to communicate with either the cpvmexample or the fmpiexample.
- `fmpiexample.f`
  This code uses a translation table type data distribution and is meant to communicate with either the cmpiexample or the fpvmexample.

C.1 Wave Equation

The code in the `WaveEquation` directory illustrates how one can couple multiple applications using InterComm's high-level interface. In this example, a 2-D wave equation simulation was split into two halves on the x-axis. Because of the split grid and the periodic boundary conditions, the two programs must exchange data, which is accomplished by an Interpolator class (that, in the future, will be able to manipulate the data and the grid as well). This class uses a communication endpoint. In this example, there are two implementations for a communication endpoint: plain files - files are created, read from, and written to (good for debugging purposes) - and InterComm.

C.2 Rainbow

This code is a sample code that utilizes an XJD file and the XJD-based API to communicate between two programs.

References

[1] InterComm, *http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/ic/*

[2] Object-Oriented Tools for Solving PDEs in Complex Geometries, *http://www.llnl.gov/CASC/Overture/*

[3] PVM: Parallel Virtual Machine, *http://www.csm.ornl.gov/pvm/pvm_home.html*

[4] GNU Autoconf, Automake, and Libtool, *http://www.gnu.org/directory/*

[5] Chaos Tools, *http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/tools.html*

[6] MultiBlockParti, *http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/tools.html*

[7] The Message Passing Interface (MPI) Standard, *http://www-unix.mcs.anl.gov/mpi/*

[8] IBM LoadLeveler, *http://publib.boulder.ibm.com/clresctr/windows/public/llbooks.html*

[9] High Performance Computing Application Launching Environment (HPCALE), *http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/hpcale/*