Inter-communicator Extensions to MPI in the MPIX (MPI eXtension) Library*†

Anthony Skjellum‡ Nathan E. Doss§ Kishore Viswanathan

Department of Computer Science &
NSF Engineering Research Center for Computational Field Simulation§
Mississippi State University
PO Box 6176
Mississippi State, MS 39762

August 3, 1994

Abstract

MPI is the new standard for multicomputer and cluster message passing introduced by the Message-Passing Interface Forum (MPIF) in April 1994. This paper describes the current inter-communicator interface found in MPI and the reasons for its current design. We also motivate the need for additional inter-communicator operations and introduce the extensions we have included in MPIX (MPI eXtension Library), a library of extensions to MPI that we are currently developing.

Inter-communicators may be used for a variety of purposes such as in client/server applications (i.e., I/O and graphics servers) and for process management in dynamic process environments and multi-protocol implementations; MPI's definitions are unnecessarily restrictive, so we extend them here. We discuss the inter-communicator collective operations defined in MPIX and illustrate their use. We also discuss additional inter-communicator construction routines not in the original MPI interface, but that are provided in MPIX. Our final contribution is a strategy for extending virtual topologies to support inter-communicators with the MPIX inter-communicator library.

1 Introduction

The final MPI (Message-Passing Interface) document was released in April 1994 [7]. The MPI Forum intentionally did not address some important issues related to message passing because of the lack of standard practice in such areas as parallel I/O and dynamic process management. Furthermore, inter-communicators play an important role in these areas, but are not fully developed in some parts of the standard. To explore these areas of research further, we have begun writing MPIX (MPI eXtension Library), which includes extensions to MPI in the areas of topologies, datatypes, and collective operations, as well as others. The goals of MPIX are as follows (roughly, in order of importance):

- Allow immediate experimentation with functionality too remote from previous current practice to be accepted in the first version of MPI.
- Stipulate those calls that can be layered on top of MPI as specified, but which could be more efficiently implemented if incorporated into a subsequent version of the standard.

*Submitted to *Integrated Computer-Aided Engineering* (ICAE) Journal.

†Work supported by the NSF Engineering Research Center for Computational Field Simulation, Mississippi State University. Revision 1.0; July 21, 1994.

‡Corresponding author. e-mail: *tony@cs.msstate.edu*; phone: 601-325-8435.

- Provide upward compatibility while not violating the MPI standard.

This paper discusses those extensions to MPI provided by MPIX that relate to inter-communicators. Inter-communicators *per se* are important because they represent a non-trivial generalization of the SPMD[1] programming model to multiple, cooperating SPMD computations, in a way for which we can explain the semantics clearly and for which we can continue to provide efficient implementations with MPI-style semantics.

This paper is organized as follows. Sections 2 briefly describes the current inter-communicator interface provided by the MPI standard, and describes an implementation of communicators. In section 3, we discuss how the semantics of the intra-communicator-only constructors can be extended to inter-communicators and introduce the MPIX calls that implement these construction routines. Section 4 discusses the semantics and implementation of MPIX inter-communicator collective operations. This is followed in section 5 by a description of how topologies are applied to inter-communicators in MPIX. Sections 6, 7, and 8 describe related work, the current status of our MPIX implementation, and future work respectively. Conclusions follow.

Each of these section discusses the usefulness of the extensions provided as well as a discussion of implmentation issues. For some functions we also provide examples of their use. For instance, we indicate where explicit inclusion of certain features in the standard would lead to inherently high-performance.

We assume familiarity with MPI inter-communicators in what follows.

2 MPI Communicators

MPI communciators provide safe, separate communication spaces, which provides the scoping needed to assure that messages occuring in separate communication spaces do not interfere with each other. A communicator is the encapsulation of a communication "context" which provides scope and a process group that describes the set of processes in the communication space. MPI provides two types of communicators: intra-communicators for communication within a single group and inter-communicators for communication between two distinct groups[2]. This section discusses how communicators (both intra- and inter-communicators) are implemented in MPICH [4], the Argonne/Mississippi State MPI implementation.

```
struct {
    int       communicator_type;
    int       send_context;
    int       recv_context;
    MPI_Group local_group;
    MPI_Group remote_group;
    MPI_Comm  collective_comm;
    /* ... */
} MPI_Comm;
```

Figure 1: Simplified communicator structure from the MPICH implementation.

Figure 1 shows a simplified picture of the communicator structure for the MPICH implementation. Communicators contain one or more groups. Groups of processes are represented as an enumerated virtual-to-real process-address-translation table. In addition to groups, a communicator contains one or more contexts. Contexts are system-managed integer tags added to the message header when a message is composed. This tag is then checked when a message is received. Point-to-point messages and collective messages have to be kept separate. The MPICH implementation provides this safety by using different communicators for point-to-point and collective operations; however, the user only ever sees the point-to-point communicator. When

[1] Single Program, Multiple Data

[2] The groups may not legally overlap in membership unless multiple-threaded programming is in use.

a collective operation is called, a special collective communicator which is "hidden" within the point-to-point communicator is used [5].

In this paper, we use two different ways of describing the groups in an inter-communicator. They can be described from a global perspective as having a left and a right group or from a process perspective as having a local group and a remote group. For inter-communicators, the two groups are distinct and non-overlapping. Contexts can also be viewed in this same manner. From the global perspective, the send context of the left group is equal to the receive context of the right group and vice-versa. From a process perspective, the *send_context* is used when sending a message to the remote group. The *recv_context* is used when receiving a message from the remote group. For inter-communicators, the contexts may or may not be equal. Although our discussion centers on inter-communicators, we note that the implementation of intra-communicators is conceptually similar to that of inter-communicators, where the two contexts and the two groups are equal. The MPI document includes a similar discussion [7, Chapter 5, Section 6].

Although most collective communications are not currently defined by MPI on inter-communicators, it is still necessary for inter-communicators to have a hidden, collective inter-communicator. This hidden inter-communicator also contains a hidden intra-communicator. These extra communicators are necessary for the implementation of the `MPI_INTERCOMM_MERGE` and `MPI_INTERCOMM_DUP` functions, which are in fact collective.

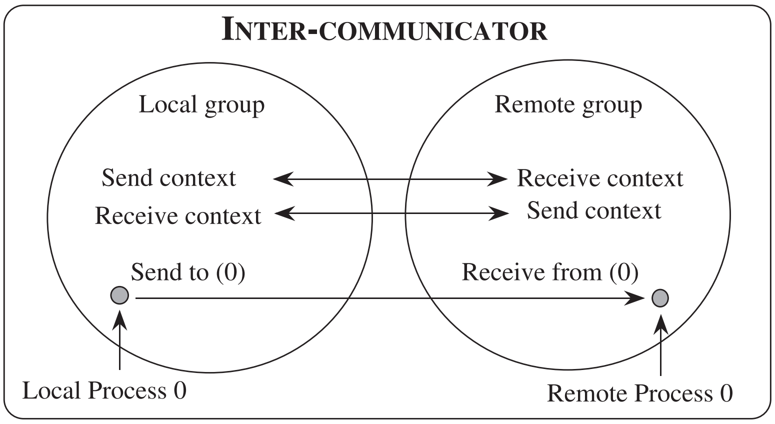

Figure 2: Schematic of nn MPI inter-communicator.

Figure 2 illustrates the structure of an inter-communicator and shows how a send in the local group to node "0" is in fact a send to the remote node "0." A more complete discussion of MPI inter-communicators can be found in the MPI standard document [7, Chapter 5]; see also Chapter 7 of *Using MPI* [9].

3 MPIX Inter-communicator Constructors

The current MPI interface provides only two inter-communicator construction routines:

- `MPI_INTERCOMM_CREATE`, creates an inter-communicator from two intra-communicators,
- `MPI_COMM_DUP`, duplicates an existing inter-communicator (or intra-communicator).

The other communicator constructors, `MPI_COMM_CREATE` and `MPI_COMM_SPLIT`, currently apply only to intra-communicators. We are convinced that these operations have well-defined semantics for inter-communicators and so we provide functions in MPIX for this purpose. We also provide two other inter-communciator constructors for partitioning intra-communicators into multiple inter-communicators and simplifying inter-communicator construction in cases where the two intra-communicators intersect, whereas intersection is prohibited currently, subject to various conditions [7, Section 5.6.2].

One important aspect of our portable MPIX implementation is that each of these MPIX inter-communicator constructors requires at least three more global synchronizations than a native implementation would require, with many more global operations needed for some operations. In MPICH, an initial implementation of MPI [4, 5, 8], an inter-communicator also contains (unseen by the user) an intra-communicator over the local group as well as a duplicate inter-communicator. These "extra" communicators are useful for the implementation of the MPI_COMM_DUP and MPI_COMM_MERGE operations and can be created efficiently. They are also useful for more efficient native implementation of these MPIX inter-communciator constructors. Part of the overhead in our portable implementation comes from the fact that we are unable to access the hidden intra- and inter-communicators already present in an MPI communicator. We must therefore create these duplicate communicators and cache them on all inter-communicators. In what follows, we exploit the notation found in [7].

3.1 MPIX_INTERCOMM_CREATE and MPIX_COMM_DUP

MPIX_INTERCOMM_CREATE and MPIX_COMM_DUP are included in the MPIX library even though to the user, they seem to perform exactly the same function in MPIX as they do in MPI. The MPIX versions create an additional cached, duplicate inter-communciator and a cached intra-communicator (the cached intra-communicator contains only those processes in the local group). These cached communicators are needed to implement the other MPIX inter-communicator constructors and MPIX collective functions.

```
MPIX_INTERCOMM_CREATE (l_comm, l_leader, peer_comm, r_leader, tag, comm_out)
  IN   l_comm            local communicator
  IN   l_leader          local leader
  IN   peer_comm         peer communciator
  IN   r_leader          remote leader
  IN   tag               safe tag
  OUT  comm_out          new communicator

MPIX_COMM_DUP (comm_in, comm_out)
  IN   comm_in           orignal communicator
  OUT  comm_out          new communicator
```

3.2 MPIX_COMM_CREATE

This function is similar to the MPI_COMM_CREATE function in the standard with the added functionality of also applying to inter-communicators.

```
MPIX_COMM_CREATE (comm_in, group, comm_out)
  IN   comm_in           orignal communicator
  IN   group             group of processes to be in new
                         communicator
  OUT  comm_out          new communicator
```

If the comm_in argument is an intra-communicator, then its functionality is the same as that of MPI_COMM_CREATE. If, however, comm_in is an inter-communicator, then the output communicator is also an inter-communicator where the local side consists only of those processes contained in group (see figure 3). The group argument should only contain those processes in the local group of the input inter-communicator that are to be a part of comm_out. If the group does not specify at least one process on the local side of the inter-communicator, or if the calling process is not included in the group, MPI_COMM_NULL is returned.

3.2.1 Native vs. Portable Implementation

The construction of an inter-communicator using MPIX_COMM_CREATE has the same communication overhead as that of duplicating an inter-communicator if it were implemented as part of the standard. However, in our portable MPIX implementation, we implemented MPIX_COMM_CREATE first by using MPI_COMM_CREATE to create an intra-communicator for each side of the inter-communicator and then by using MPI_INTERCOMM_CREATE to create the new inter-communicator (a step that required the swapping of remote leader information).

Figure 4 illustrates how MPIX_COMM_CREATE may be implemented portably:

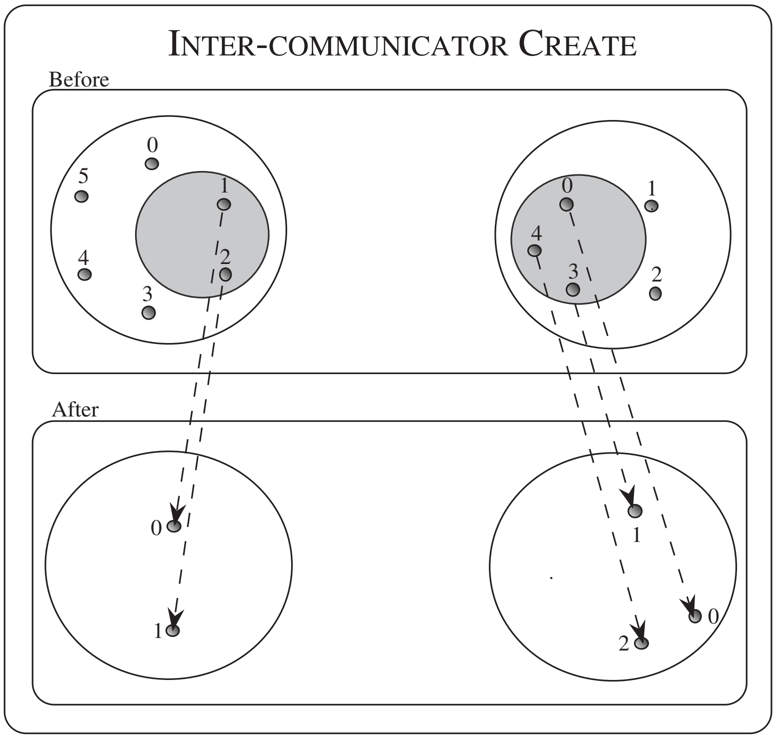

Figure 3: Inter-communicator create

3.2.2 Example

The following example illustrates how the first node in the "left" side of an inter-communicator could be joined with all members on the "right" side of an inter-communciator to form a new inter-communicator.

```
MPI_Comm  inter_comm, new_inter_comm;
MPI_Group local_group, group;
int       rank = 0; /* rank on left side to include in new inter-comm */

/* Construct the original inter-communicator: "inter_comm" */
...

/* Construct the group of processes to be in new inter-communicator */
if (/* I'm on the left side of the inter-communicator */) {
  MPI_Comm_group ( inter_comm, &local_group );
  MPI_Group_incl ( local_group, 1, &rank, &group );
}
else
  MPI_Comm_group ( inter_comm, &group );

MPIX_Comm_create ( inter_comm, group, &new_inter_comm );
```

Without MPIX, users would have to implement code of this complexity themselves.

3.3 MPIX_COMM_SPLIT

This function splits an existing communicator into multiple communicators based on color and key arguments (analogous to MPI_COMM_SPLIT), but works on inter-communicators as well as intra-communicators.

```c
#include <mpi.h>
int MPIX_Comm_create ( comm_in, group, comm_out )
MPI_Comm   comm_in, *comm_out;
MPI_Group  group;
{
  MPI_Comm   cached_intra, cached_inter, new_cached_intra, new_cached_inter;
  MPI_Group  comm_in_group;
  MPI_Status status;
  int        group_rank = 0;
  int        local_leader, remote_leader, flag, rank;

  /* test to see if we should call MPI_Comm_create for intra-comms */
  MPI_Intercomm_test ( comm_in, &flag );
  if (!flag)  return (MPI_Comm_create ( comm_in, group, comm_out ));

  /* retrieve cached information ... */

  /* translate rank 0 in group to rank in original comm_in */
  MPI_Comm_group ( comm_in, &comm_in_group );
  MPI_Group_translate_ranks ( group, 1, &group_rank, comm_in_group, &local_leader );

  /* swap leader information with remote group */
  MPI_Comm_rank ( comm_in, &rank );
  if (rank == 0)
    MPI_Sendrecv (&local_leader,  1, MPI_INT, 0, 0, &remote_leader, 1, MPI_INT, 0, 0,
                  cached_inter, &status );

  /* broadcast leader information */
  MPI_Bcast (&remote_leader, 1, MPI_INT, 0, cached_intra );

  /* make a new intra communicator */
  MPI_Comm_create ( cached_intra, group, &new_cached_intra );

  /* if we're not going to be a part of the new inter-comm, return */
  if (new_cached_intra == MPI_COMM_NULL) {
    (*comm_out) = MPI_COMM_NULL;
    return (MPI_SUCCESS);
  }

  /* use intercomm_create to create new inter-communicator */
  MPI_Intercomm_create ( new_cached_intra, 0, cached_inter, remote_leader, 0, comm_out );
  MPI_Comm_dup ( (*comm_out), &new_cached_inter );

  /* cache the two communicators ... */
  return (MPI_SUCCESS);
}
```

Figure 4: Example portable implementation of MPIX_COMM_CREATE.

```
MPIX_COMM_SPLIT (comm_in, color, key, comm_out)
  IN   comm_in            original communicator
  IN   color              color of subset assignment
  IN   key                color of rank assignment
  OUT  comm_out           new communicator
```

The result of split on an inter-communicator is that those processes on the left with the same `color` as those processes on the right combine to create a new inter-communicator. The `key` argument describes the relative rank of processes on each side of the inter-communicator (see figure 5). For those colors that are specified only on one side of the inter-communicator but not the other, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned to those processes that specify MPI_UNDEFINED as the color.

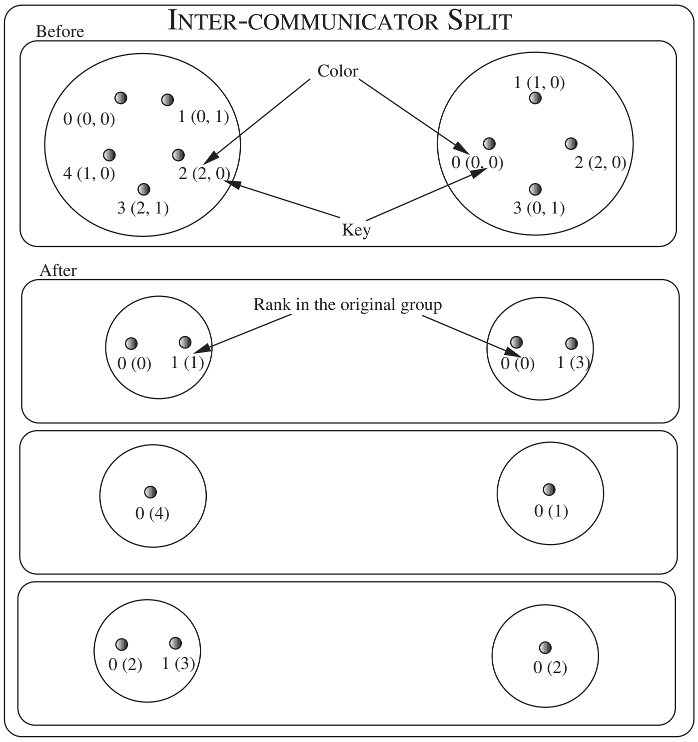

Figure 5: Inter-communicator construction by splitting an existing inter-communicator.

3.3.1 Native vs. Portable Implementation

MPIX_COMM_SPLIT for inter-communicators can be implemented natively with the same communication overhead as the code to split intra-communicators in MPI_COMM_SPLIT plus one additional collective operation. Our portable MPIX implementation first collects all color and key information in all processes, then uses this information to determine leader information for the MPIX_INTERCOMM_CREATE call, which must be called for each color. If, for example, there are five different colors given, then there would at least be five MPIX_INTERCOMM_CREATE calls.

3.3.2 Example (Parallel client-server model)

The following client code illustrates how clients on the left side of an inter-communicator could be assigned to a single server from a pool of servers on the right side of an inter-communicator.

```
/* Client code */
MPI_Comm  multiple_server_comm;
```

```
MPI_Comm  single_server_comm;
int       color, rank, num_servers;

/* Create inter-communicator with clients and servers: multiple_server_comm */
...

/* Find out the number of servers available */
MPI_Comm_remote_size ( multiple_server_comm, &num_servers );

/* Determine my color */
MPI_Comm_rank ( multiple_server_comm, &rank );
color = rank % num_servers;

/* Split the inter-communicator */
MPIX_Comm_split ( multiple_server_comm, color, rank, &single_server_comm );
```

The following is the corresponding server code:

```
/* Server code */
MPI_Comm  multiple_client_comm;
MPI_Comm  single_client_comm;
int       rank;

/* Create inter-communicator with clients and servers: multiple_client_comm */
...

/* Split the inter-communicator */
MPI_Comm_rank ( multiple_client_comm, &rank );
MPIX_Comm_split ( multiple_client_comm, rank, 0, &single_client_comm );
```

3.4 MPIX_INTERCOMM_PARTITION

MPIX_INTERCOMM_PARTITION partitions an intra-communicator (single group) into a single inter-communicator (local and remote groups). There is no MPI analogue to MPIX_INTERCOMM_PARTITION.

```
MPIX_INTERCOMM_PARTITION (intracomm, flag, key, intercomm)
  IN   intracomm          original intra-communicator
  IN   flag               left/right flag (boolean)
  IN   key                sorting order
  OUT  intercomm          new inter-communicator
```

The flag argument determines whether a process will be on the left or right side of the resulting inter-communicator. The key argument provides ordering information for each side of the inter-communicator. There must be at least one process with flag = TRUE and at least one process with flag = FALSE, otherwise, the resulting inter-communicator will be MPI_COMM_NULL.

3.4.1 Discussion

This function can be implemented natively with the same complexity as MPI_COMM_CREATE. Our portable MPIX version first splits the original intra-communicator with MPI_COMM_SPLIT, then uses MPI_INTERCOMM_CREATE to construct the inter-communicator.

3.4.2 Example

The example given below shows how the original MPI_COMM_WORLD can be divided into a client/server inter-communicator.

```
MPI_Comm  client_server_comm;
int       rank, flag;

/* Get my rank in the world */
MPI_Comm_rank ( MPI_COMM_WORLD, &rank );

/* Set the flag that determines my function in life */
if (/* I'm to be a server */)
  flag = TRUE;
else /* I'm to be a client */
  flag = FALSE;

/* Partition the world */
MPIX_Intercomm_partition ( MPI_COMM_WORLD, flag, rank, &client_server_comm );
```

Users will find this function convenient as compared to using MPI_COMM_SPLIT and MPIX_INTERCOMM_CREATE.

3.5 MPIX_INTERCOMM_OVERLAP_CREATE

When constructing an inter-communicator with MPI_INTERCOMM_CREATE, the two intra-communicators are not allowed to overlap [4]. We provide this function for easier construction of inter-communicators when the two original intra-communicators do overlap. As shown in Figure 6, if there is overlap, the processes in the overlap are included on one side (the left side) of the inter-communicator, but not the other. There is no need for a peer communicator or tag (as in MPI_INTERCOMM_CREATE) as a result of the overlap.

```
MPIX_INTERCOMM_OVERLAP_CREATE (left_intra, right_intra, intercomm)
  IN   left_intra        intra-communicator to make up left side
                         of the resulting inter-communicator
  IN   right_intra       intra-communicator to make up right side
                         of the resulting inter-communicator
  OUT  intercomm         new inter-communicator
```

All processes that belong to both the left and right intra-communicators must order the arguments to MPIX_INTERCOMM_OVERLAP_CREATE in exactly the same way. If there is no overlap, one intra-communicator argument must be MPI_COMM_NULL. If there is no overlap, this is an erroneous usage.

3.5.1 Discussion

This function can be implemented natively with the same communication overhead as an MPI_INTERCOMM_CREATE. Our MPIX version needs several additional communication steps. First, it duplicates the original right hand intra-communicator, then creates a new right side intra-communicator which does not contain the processes in the overlap. Several synchronizations are then needed to determine a leader in the intersection for the left intra-communicator and a leader for the right side that is not in the intersection. MPIX_INTERCOMM_CREATE is then used to create the new inter-communicator with the original right intra-communicator as the peer communicator.

4 MPIX Inter-communicator Collective Operations

In the current draft, collective operations only apply to intra-communicators [7, Section 4.2, pg. 90]; however, most MPI collective operations can be easily extended to inter-communicators. This realization is a major contribution of our study, though it is evident in hindsight that we should have included these capabilities in the initial MPI standard. To understand how MPI can be extended, we place most MPI operations into one of the following categories; which itself is useful for understanding collective operations:

All-To-All All processes contribute to the result. All processes receive the result.

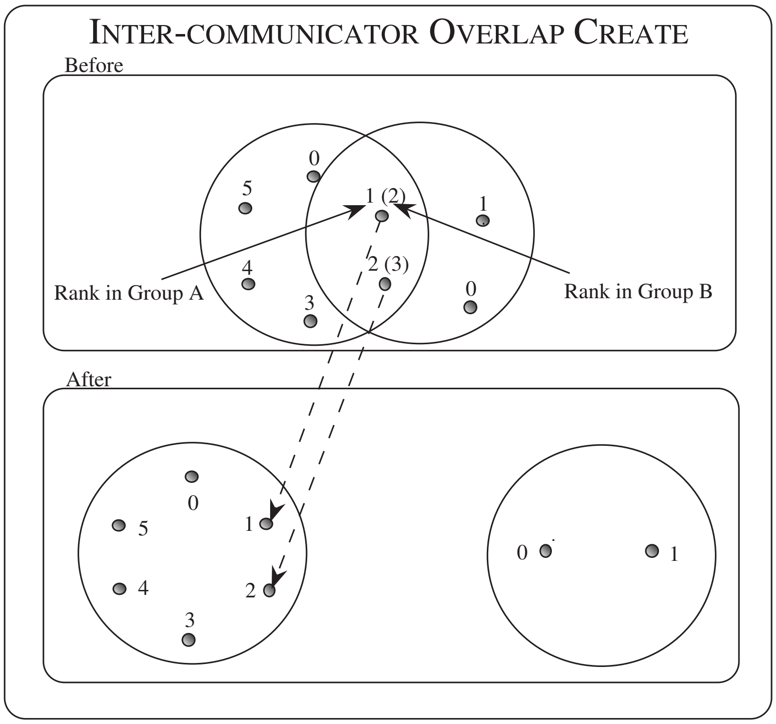

Figure 6: Inter-communicator construction from two overlapping intra-communicators.

- MPI_Allgather, MPI_Allgatherv
- MPI_Alltoall, MPI_Alltoallv
- MPI_Allreduce, MPI_Reduce_scatter

All-To-One All processes contribute to the result. One process receives the result.

- MPI_Gather, MPI_Gatherv
- MPI_Reduce

One-To-All One process contributes to the result. All processes receive the result.

- MPI_Bcast
- MPI_Scatter, MPI_Scatterv

Other Collective operations that do not fit into one of the above categories.

- MPI_Scan
- MPI_Barrier

The `MPI_Barrier` operation does not fit into this classification since no data is being moved. `MPI_Scan` does not fit into this classification since its data movement pattern does not fit our taxonomy.

Communicators contain a "left" and a "right" group (as was discussed in Section 2). Collective operations can be described in terms of these two groups. For example, an All-To-All `MPI_Allgather` operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see figure 8). As another example, a One-To-All `MPI_Bcast` operation sends data from one member of one group to all members of the other group (see figure 7). For intra-communicators, these two groups are the same. For inter-communicators, these two groups are distinct.

MPIX provides implementations of the following collective operations:

- MPIX_BCAST,
- MPIX_GATHER, MPIX_GATHERV,
- MPIX_SCATTER, MPIX_SCATTERV,
- MPIX_ALLGATHER, MPIX_ALLGATHERV,
- MPIX_ALLTOALL, MPIX_ALLTOALLV,
- MPIX_REDUCE, MPIX_ALLREDUCE,
- MPIX_REDUCE_SCATTER.

These functions use exactly the same argument list as their MPI counterparts and also work on intra-communicators, as expected.

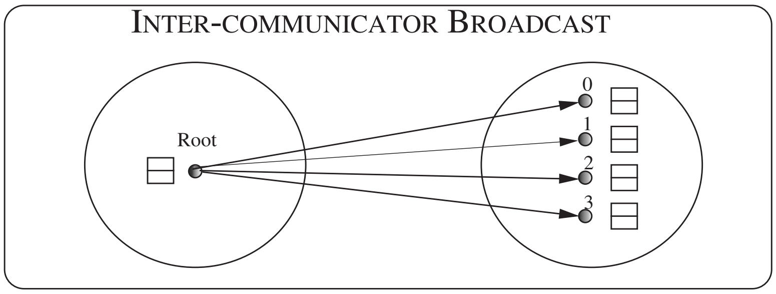

Figure 7: Inter-communicator broadcast.

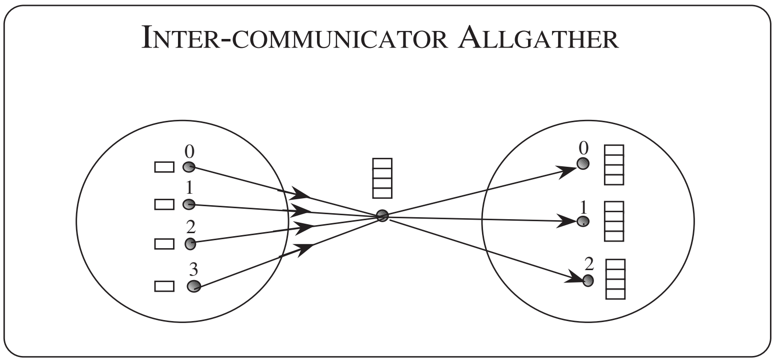

Figure 8: Inter-communicator allgather. The focus of data to one process is represented, not mandated by the semantics.

In order to implement these calls portably, we need a safe collective communication space. This can be obtained by duplicating the original inter-communicator and caching this duplicate inter-communicator with the original inter-communicator. We also use a local intra-communicator to communicate within each side of an inter-communicator. We therefore duplicate (or create) intra-communicators for each side of the inter-communicator and also cache this in the inter-communicator. All of the inter-communicator constructors mentioned in [3] do this (which further adds to the overhead of constructing inter-communicators non-natively). We have also provided a function similar to MPI_INTERCOMM_CREATE to create an inter-communicator that can be used with the inter-communicator collective calls. We mentioned earlier that in MPICH [4], inter-communicators already have these additional hidden communicators, so that if the inter-communciator collective operations were implemented natively, this additional overhead would be avoided. Thus, our operations are good candidates for later inclusion in MPI, for these performance reasons.

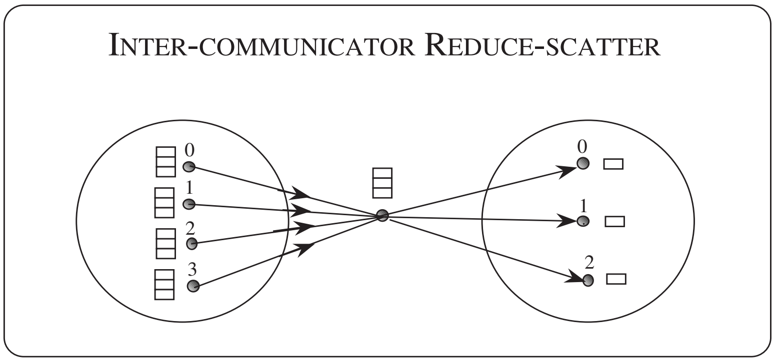

Figure 9: Inter-communicator reduce-scatter. The focus of data to one process is logical, not mandated by the semantics.

The interface to the MPIX inter-communicator collective operations is similar to that of intra-communicator calls. The one exception is that some indication must be given as to which side of the inter-communicator is the sending side and which is the receiving side. This can be handled easily for both *rooted* and *non-rooted* collective calls.

4.1 Rooted Example

For the rooted case, a special flag, `MPIX_ROOT`, is used to indicate that the process is the root of the inter-communicator operation. This flag is given for the root argument. In all processes on the other side of an inter-communicator, the rank of the root process (relative to the other group) is given. Rooted collective inter-communicator operations are not collective over all processes in the inter-communicator. On the root side of the inter-communicator, only the root process participates, while all processes on the other side participate (see figure 7).

The following example illustrates the use of `MPIX_BCAST`.

```
int      rank, intercomm_rank, size, flag, root = 0;
MPI_Comm intercomm;

/* Get size of world and my rank in it */
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

/* Partition the world into an inter-communicator */
flag = rank % 2;
MPIX_Intercomm_partition ( MPI_COMM_WORLD, flag, rank, &intercomm );

/* Perform the broadcast */
MPI_Comm_rank (intercomm, &intercomm_rank);
if (flag == 0) {
  if (intercomm_rank == 0)
    MPIX_Bcast ( &rank, 1, MPI_INT, MPIX_ROOT, intercomm );
}
else
  MPIX_Bcast ( &rank, 1, MPI_INT, root, intercomm );
```

4.2 Non-Rooted Example

For non-rooted `MPIX` collective calls, a `NULL` buffer parameter (`MPIX_BUFFER_NULL`) is used to flag whether the process is on the sending or receiving side of the operation. An `MPIX_ALLREDUCE` is illustrated below.

```
int      rank, size, flag, side;
MPI_Comm intercomm;
int      data;

/* Get my rank in the world */
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

/* Partition the world into an inter-communicator */
flag = rank % 2;
MPIX_Intercomm_partition ( MPI_COMM_WORLD, flag, rank, &intercomm );

/* Initialize buffer and determine my side */
if (flag = 0) {
  data = rank % 5;

  /* Perform the allreduce */
  MPIX_Allreduce(&data, MPIX_BUFFER_NULL, 1, MPI_INT, MPI_SUM, intercomm);
}
else {

  /* Perform the allreduce */
  MPIX_Allreduce(MPIX_BUFFER_NULL, &data, 1, MPI_INT, MPI_SUM, intercomm);
}
```

The advantage of our approach is that the interface remains the same for both intra- and inter-communicators. An alternate approach would be to have different interfaces for intra- and inter-communicators. Inter-communicator collective operations could have either send or receive buffer information, but not both. An additional flag could then be passed to indicate whether or not it was on the send or receive side of the operation.

5 MPIX Inter-communicator Topologies

The topology functions in chapter 6 of the MPI document [7] do not apply to inter-communicators [7, pg. 128]. In some applications, it is useful for an inter-communicator to have a topology for at least one side or possibly both sides of an inter-communicator. MPIX provides a library of routines similar to the ones provided by MPI on intra-communicators for adding, inquiring about, and manipulating topologies on inter-communicators. Figure 10 lists the topology functions provided in MPIX.

MPI provides two functions for creating new communicators with associated topology information: MPI_CART_CREATE and MPI_GRAPH_CREATE. MPIX provides four similar calls with similar arguments for inter-communicators. All of these calls create a new inter-communicator with attached topology information. For MPIX_CART_CREATE and MPIX_GRAPH_CREATE, the topology information provided in each process applies to the local side of the inter-communicator, while for MPIX_CART_REMOTE_CREATE and MPIX_GRAPH_REMOTE_CREATE the topology information provided in each process applies to the remote side of the inter-communicator. Unlike intra-communicator topology constructors, all processes on one side of the inter-communicator should give the same topology information or the results are undefined.

Once an inter-communicator has been created with any one of these calls, both local and remote topology information can be retrieved. In order to provide this information, we provide both "local" and "remote" versions of the topology inquiry functions found in the MPI document that get the local and remote topology information respectively. For example, we have both an MPIX_CART_GET that gets the local topology information and an MPIX_CART_REMOTE_GET that gets the topology information of the remote group.

We also provide MPIX_CART_SUB and MPIX_CART_REMOTE_SUB for partitioning inter-communicators similar to manner in which MPI_CART_SUB partitions intra-communicators (see figure 11). These calls partition the

Existing MPI Functions	Corresponding MPIX Functions
MPI_Cart_create	MPIX_Cart_create, MPIX_Cart_remote_create
MPI_Cart_get	MPIX_Cart_get, MPIX_Cart_remote_get
MPI_Cartdims_get	MPIX_Cartdims_get, MPIX_Cartdims_remote_get
MPI_Cart_rank	MPIX_Cart_rank, MPIX_Cart_remote_rank
MPI_Cart_coords	MPIX_Cart_coords, MPIX_Cart_remote_coords
MPI_Cart_sub	MPIX_Cart_sub, MPIX_Cart_remote_sub
MPI_Cart_shift	MPIX_Cart_shift (intra-communicators only)
MPI_Graph_create	MPIX_Graph_create, MPIX_Graph_remote_create
MPI_Graph_get	MPIX_Graph_get, MPIX_Graph_remote_get
MPI_Graphdims_get	MPIX_Graphdims_get, MPIX_Graphdims_remote_get
MPI_Graph_neighbors_count	MPIX_Graph_neighbors_count, MPIX_Graph_remote_neighbors_count
MPI_Graph_neighbors	MPIX_Graph_neighbors, MPIX_Graph_remote_neighbors

Figure 10: Existing MPI Topology Functions and their MPIX Counterparts.

inter-communicator into lower-dimensional sub-communicators. Assume that we have a two-dimensional topology on the left side of an inter-communicator and a three dimensional topology on the right side of the inter-communicator. The inter-communicator could then be partitioned into rows on the left side of a communicator that would then correspond to a plane on the right side. A restriction on this call is that the product of the removed dimensions on the left side of the inter-communicator must be equal to the product of the removed dimensions on the right side of the communicator. The local version of this call, `MPIX_CART_SUB`, allows processes to specify how the local topology is to be subdivided. The remote version, `MPIX_CART_REMOTE_SUB`, allows processes to specify how the remote topology is to be subdivided.

Our topology library contains exactly the same functionality as the `MPI` document specifies (the functions work with intra-communicators also) with the additional functionality of working with inter-communicators. From our MPI implementation experiences, we believe that a portable implementation of these topology functions can be as efficient as a native implementation.

6 Related Work

Many existing message-passing systems do not use groups other than the group of all processes (*i.e.*, p4 [2], NX [10]) or do not have a mechanism for communicating between groups (*i.e.*, PVM [1], Zipcode [11], Chimp [6]). MPI provides a mechanism for inter-communication and therefore a framework to build upon that other message-passing systems do not provide.

Chandy et. al. discuss communication between groups and use similar language to that found in this paper and in the MPI document when discussing group-to-group communication [3]. The purpose of their discussion is to demonstrate how libraries can be coupled together, with distinct groups each running different library code. Communication between the groups takes the form of either point-to-point or collective communication. They do not discuss how inter-group communication is to be safely implemented, instead focusing on such things as data distribution and data structure information that can be used to couple library codes together. They do not seem to recognize the relationship of their work to the MPI inter-communicator paradigm.

Inter-communicators also find an application in the efficient implementation of parallel I/O services. A group of file server processes can provide file access to a group of client processes via an inter-communicator composed of the two groups. Marc Snir proposed I/O extensions to MPI based on this model with processes being able to perform both collective and individual I/O [12]. However, his proposal, which is preliminary, fails to utilize the inter-communicator paradigm.

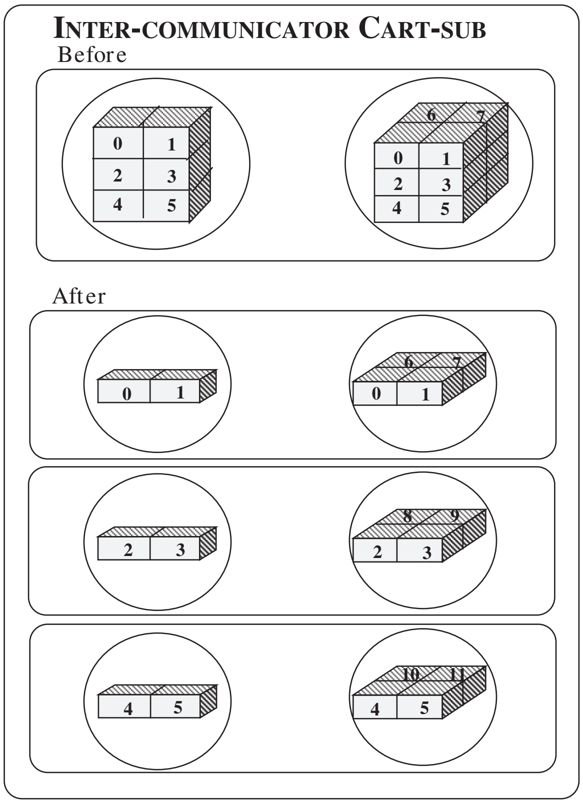

Figure 11: Inter-communicator cartesian subdivision

7 Status of Portable Implementation

Presently we have implemented and tested the C interface for the following operations as part of our portable MPIX implementation.

The following inter-communicator constructors have been implemented:

- `MPIX_Intercomm_create`,
- `MPIX_Comm_dup`,
- `MPIX_Comm_create`,
- `MPIX_Intercomm_partition`,
- `MPIX_Intercomm_overlap_create`.

and the following inter-communicator collective operations have been implemented as well:

- `MPIX_Bcast`,
- `MPIX_Gather`, `MPIX_Gatherv`, `MPIX_Allgather`, `MPIX_Allgatherv`,
- `MPIX_Scatter`, `MPIX_Scatterv`,
- `MPIX_Alltoall`, `MPIX_Alltoallv`,
- `MPIX_Reduce`, `MPIX_Allreduce`,
- `MPIX_Reduce_scatter`.

Topology functions and `MPIX_Intercomm_split` remain to be implemented.

8 Future Work

We are currently implementing the C interface for the portable versions of all the MPIX virtual topology functions that we have listed in figure 10; in the previous section, we indicated our progess. We will implement the C interface for the native version of all the MPIX functions we have mentioned in this paper. As mentioned earlier, we are convinced that our native implementation will, in general be faster than the portable implementation. This is mainly a result of the increased number of global synchronizations required in the portable implementation. To estimate the performance penalty from the overheads in the portable implementation, we will conduct a head-to-head performance comparison of the two versions.

Inter-communicators find application in the efficient implementation of parallel I/O services. A group of file server processes can provide file access to a group of client processes via an inter-communicator between the client and the server groups [12]. We intend to implement parallel I/O services as part of the MPIX library.

9 Conclusions

MPI is a powerful library for describing message-passing and has already embraced most of the complexity of inter-communicators but has stopped short. MPI has already accepted most of the implied implementation overheads of what we propose in this paper, by inclusion of certain features (*e.g.*, `MPI_INTERCOMM_MERGE`). Therefore, we advocate that most of the functions introduced in this paper be changed (as part of an MPI2 effort) to work with inter-communicators for the following three reasons:

- consistency,
- usefulness of the proposed functionality,
- expense of a portable implementation.

We propose that all collective operations, `MPI_COMM_CREATE` and `MPI_COMM_SPLIT` should be extended to work with inter-communicators. We do not propose that the topology extensions we've introduced be adopted by MPI2 because since a portable implementation can be written with approximately the same performance as a native implementation.

In summary, these are the contributions of this work:

- A portable implementation is possible for all of the functions presented.
- Based on our work with MPIX and MPICH, we can describe the relative efficiency of layering as compared to direct standard support.
- Inter-communicator collective operations open a new area for algorithmic research (*e.g.,* pipelined linear algebra libraries).
- Inter-communicator collective operations provide a paradigm in which to discuss generalized collective operations.
- We have extended the SPMD model logically.

Furthermore, we note that:

- Most of the concepts and functions in MPI that currently do not apply to inter-communicators do make sense with inter-communicators.
- A native implementation of the communicator construction and collective functions we propose can be much more efficient than a layered (portable) implementation.
- Inter-communication is not supported in most existing message passing libraries. With `MPIX`, we provide a rich environment in which to experiment and hope to have some impact on things that may be included in MPI-2. This will lead to further algorithmic experimentation (pipelined libraries, parallel client-server computation).

- Applying all the functions in the `MPI` document to inter-communicators makes it more consistent as a whole.
- Threads and parallel I/O remain the next logical extensions to the generalized SPMD programming model we present here.
- We have written this library in order to advance MPI, by providing enhanced functionality, that suggests how MPI should evolve in the immediate future.

10 Acknowledgements

We acknowledge the editor of this special issue, Dr. Salim Hariri, for his patience and cooperation in the production and submission of this paper. We are also grateful to our colleagues Purushotham V. Bangalore and Aswini K. Chowdappa, who have provided feedback on the manuscript.

Where to Learn More about MPI

To find out more about MPI, see the document [7] available on **netlib**. A quick way to access this is to send a mail message to *netlib@ornl.gov* containing the line `send index from MPI`. Alternatively, one can anonymous ftp the source from **netlib2.cs.utk.ed**. An online version of the document is available through the WWW at **http://www.mcs.anl.gov/mpi/index.html**. An MPI frequently asked questions (FAQ) list can be retrieved by anonymous ftp from **aurora.cs.msstate.edu** in **pub/mpi/faq** and is also available through the WWW at **http://www.cs.msstate.edu/dist_computing/mpi-faq.html**. An MPI usenet newsgroup (**comp.parallel.mpi**) has also been recently created.

References

[1] A. Beguelin, G. A. Geist, W. Jiang, R. Manchek, K. Moore, and V. Sunderam. The PVM project. Technical report, Oak Ridge National Laboratory, February 1993.

[2] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992.

[3] K. Mani Chandy, Rajit Manohar, Berna L. Massingill, and Daniel L. Meiron. Integrating Task and Data Parallelism with the Collective Communication Archetype. Technical Report CRPC-94-6, California Institute of Technology, CRPC., May 30 1994.

[4] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. A model implementation of mpi. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993. in preparation.

[5] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. Implementing the message-passing interface standard (MPI). in preparation, 1994.

[6] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992.

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. To appear in the International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.

[8] William Gropp and Ewing Lusk. An Abstract Device Definition to Support the Implementation of a High-Level Point-to-Point Message-Passing Interface. Technical Report MCS-P342-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[9] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994. In preparation.

[10] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[11] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994.

[12] Marc Snir, Peter Corbett, Dror Feitelson, and Jean-Peirre Prost. Draft Document for a Parallel MPI IO Library. (Draft document presented for informal consideration in MPI-2 standardization process)., January 14 1994.