GROUPS, CONTEXTS, AND COMMUNICATORS

5.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a "higher level" of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [26] and [3] for further information on writing libraries in MPI, using the features described in this chapter.

5.1.1 FEATURES NEEDED TO SUPPORT LIBRARIES

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to "adorn" a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.

5.1.2 MPI'S SUPPORT FOR LIBRARIES

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- **Groups** of processes,
- **Virtual topologies**,
- **Attribute caching**,
- **Communicators**.

Communicators (see [16, 24, 27]) encapsulate all of these ideas in order to provide the appropriate scope for all communication operations in MPI. Communicators are divided into two kinds: intra-communicators for operations within a single group of processes, and inter-communicators, for point-to-point communication between two groups of processes.

Caching. Communicators (see below) provide a "caching" mechanism that allows one to associate new attributes with communicators, on a par with MPI built-in features. This can be used by advanced users to adorn communicators further, and by MPI to implement some communicator functions. For example, the virtual topology functions described in Chapter 6 are likely to be supported this way.

Groups. Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in MPI, but only communicators can be used in communication operations.

Intra-communicators. The most commonly used means for message passing in MPI is via intra-communicators. Intra-communicators contain an instance of a group, contexts of communication for both point-to-point and collective communication, and the ability to include virtual topology and other attributes. These features work as follows:

- **Contexts** provide the ability to have separate safe "universes" of message passing in MPI. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library

invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator.

- **Groups** define the participants in the communication (see above) of a communicator.
- A **virtual topology** defines a special mapping of the ranks in a group to and from a topology. Special constructors for communicators are defined in chapter 6 to provide this feature. Intra-communicators as described in this chapter do not have topologies.
- **Attributes** define the local information that the user or library has added to a communicator for later reference.

Advice to users. The current practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. This practice can be followed in MPI by using the predefined communicator MPI_COMM_WORLD. *Users who are satisfied with this practice can plug in* MPI_COMM_WORLD *wherever a communicator argument is required, and can consequently disregard the rest of this chapter.* (*End of advice to users.*)

Inter-communicators. The discussion has dealt so far with **intra-communication**: communication within a group. MPI also supports **inter-communication**: communication between two non-overlapping groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. This is especially convenient in a client-server computing paradigm, where either client or server are parallel. The support of inter-communication also provides a mechanism for the extension of MPI to a dynamic model where not all processes are preallocated at initialization time. In such a situation, it becomes necessary to support communication across "universes." Inter-communication is supported by objects called **inter-communicators**. These objects bind two groups together with communication contexts shared by both groups. For inter-communicators, these features work as follows:

- **Contexts** provide the ability to have a separate safe "universe" of message passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication

internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. There is no general-purpose collective communication on inter-communicators, so contexts are used just to isolate point-to-point communication.

- A local and remote group specify the recipients and destinations for an inter-communicator.
- Virtual topology is undefined for an inter-communicator.
- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They are used for point-to-point communication in a related manner to intra-communicators. Users who do not need inter-communication in their applications can safely ignore this extension. Users who need collective operations via inter-communicators must layer it on top of MPI. Users who require inter-communication between overlapping groups must also layer this capability on top of MPI.

5.2 Basic Concepts

In this section, we turn to a more formal definition of the concepts introduced above.

5.2.1 GROUPS

A **group** is an ordered set of process identifiers (henceforth processes); processes are implementation-dependent objects. Each process in a group is associated with an integer **rank**. Ranks are contiguous and start from zero. Groups are represented by opaque **group objects**, and hence cannot be directly transferred from one process to another. A group is used within a communicator to describe the participants in a communication "universe" and to rank such participants (thus giving them unique names within that "universe" of communication).

There is a special predefined group: MPI_GROUP_EMPTY, which is a group with no members. The predefined constant MPI_GROUP_NULL is the value used for invalid group handles.

> *Advice to users.* MPI_GROUP_EMPTY, which is a valid handle to an empty group, should not be confused with MPI_GROUP_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, in not a valid argument. (*End of advice to users.*)

> *Advice to implementors.* A group may be represented by a virtual-to-real process-address-translation table. Each communicator object (see below) would have a pointer to such a table.

Simple implementations of MPI will enumerate groups, such as in a table. However, more advanced data structures make sense in order to improve scalability and memory usage with large numbers of processes. Such implementations are possible with MPI. (*End of advice to implementors.*)

5.2.2 CONTEXTS

A **context** is a property of communicators (defined next) that allows partitioning of the communication space. A message sent in one context cannot be received in another context. Furthermore, where permitted, collective operations are independent of pending point-to-point operations. Contexts are not explicit MPI objects; they appear only as part of the realization of communicators (below).

Advice to implementors. Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communications within one communicator do not interfere, and that communications over distinct communicators don't interfere.

A possible implementation for a context is as a supplemental tag attached to messages on send and matched on receive. Each intra-communicator stores the value of its two tags (one for point-to-point and one for collective communication). Communicator-generating functions use a collective communication to agree on a new group-wide unique context.

Analogously, in inter-communication (which is strictly point-to-point communication), two context tags are stored per communicator, one used by group A to send and group B to receive, and a second used by group B to send and for group A to receive.

Since contexts are not explicit objects, other implementations are also possible. (*End of advice to implementors.*)

5.2.3 INTRA-COMMUNICATORS

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, chapter 6), communicators may also "cache" additional information (see Section 5.7). MPI communication operations reference communicators to determine the scope and the "communication universe" in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local process. The source and destination of a message is identified by process rank within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the "spatial" scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one process to another.

5.2.4 PREDEFINED INTRA-COMMUNICATORS

An initial intra-communicator MPI_COMM_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI_INIT has been called. In addition, the communicator MPI_COMM_SELF is provided, which includes only the process itself.

The predefined constant MPI_COMM_NULL is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all processes that participate in the computation are available after MPI is initialized. For this case, MPI_COMM_WORLD is a communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynamically join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, MPI_COMM_WORLD is a communicator incorporating all processes with which the joining process can immediately communicate. Therefore, MPI_COMM_WORLD may simultaneously have different values in different processes.

All MPI implementations are required to provide the MPI_COMM_WORLD communicator. It cannot be deallocated during the life of a process. The group corresponding to this communicator does not appear as a predefined constant, but it may be accessed using MPI_COMM_GROUP (see below). MPI does not specify the correspondence between the process rank in MPI_COMM_WORLD and its (machine-dependent) absolute address. Neither does MPI specify the function of the host process, if any. Other implementation-dependent, predefined communicators may also be provided.

5.3 Group Management

This section describes the manipulation of process groups in MPI. These operations are local and their execution does not require interprocess communication.

5.3.1 GROUP ACCESSORS

MPI_GROUP_SIZE(group, size)

IN	group	group (handle)
OUT	size	number of processes in the group (integer)

```
int MPI_Group_size(MPI_Group group, int *size)

MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR
```

MPI_GROUP_RANK(group, rank)

IN	group	group (handle)
OUT	rank	rank of the calling process in group, or MPI_UNDEFINED if the process is not a member (integer)

```
int MPI_Group_rank(MPI_Group group, int *rank)

MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR
```

MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2)

IN	group1	group1 (handle)
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)
IN	ranks1	array of zero or more valid ranks in group1
IN	group2	group2 (handle)
OUT	ranks2	array of corresponding ranks in group2, MPI_UNDEFINED when no correspondence exists.

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
              MPI_Group group2, int *ranks2)

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
```

This function is important for determining the relative numbering of the same processes in two different groups. For instance, if one knows the ranks of certain processes in the group of MPI_COMM_WORLD, one might want to know their ranks in a subset of that group.

MPI_GROUP_COMPARE(group1, group2, result)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	result	result (integer)

```
int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR
```

MPI_IDENT results if the group members and group order is exactly the same in both groups. This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if the group members are the same but the order is different. MPI_UNEQUAL results otherwise.

5.3.2 GROUP CONSTRUCTORS

Group constructors are used to subset and superset existing groups. These constructors construct new groups from existing groups. These are local operations, and distinct groups may be defined on different processes; a process may also define a group that does not include itself. Consistent definitions are required when groups are used as arguments in communicator-building functions. MPI does not provide a mechanism to build a group from scratch, but only from other, previously defined groups. The base group, upon which all other groups are defined, is the group associated with the initial communicator MPI_COMM_WORLD (accessible through the function MPI_COMM_GROUP).

Rationale. In what follows, there is no group duplication function analogous to MPI_COMM_DUP, defined later in this chapter. There is no need for a group duplicator. A group, once created, can have several references to it by making copies of the handle. The following constructors address the need for subsets and supersets of existing groups. (*End of rationale.*)

Advice to implementors. Each group constructor behaves as if it returned a new group object. When this new group is a copy of an existing group, then one can avoid creating such new objects, using a reference-count mechanism. (*End of advice to implementors.*)

MPI_COMM_GROUP(comm, group)

IN	comm	communicator (handle)
OUT	group	group corresponding to comm (handle)

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

```
MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
```

MPI_COMM_GROUP returns in group a handle to the group of comm.

MPI_GROUP_UNION(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	union group (handle)

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
```

```
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	intersection group (handle)

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)
```

```
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	difference group (handle)

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)
```

```
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

The set-like operations are defined as follows:

union All elements of the first group (group1), followed by all elements of second group (group2) not in first.

intersect All elements of the first group that are also in the second group, ordered as in first group.

difference All elements of the first group that are not in the second group, ordered as in the first group.

Note that for these operations the order of processes in the output group is determined primarily by order in the first group (if possible) and then, if necessary, by order in the second group. Neither union nor intersection are commutative, but both are associative.

The new group can be empty, that is, equal to MPI_GROUP_EMPTY.

MPI_GROUP_INCL(group, n, ranks, newgroup)

IN	group	group (handle)
IN	n	number of elements in array ranks (and size of newgroup) (integer)
IN	ranks	ranks of processes in group to appear in newgroup (array of integers)
OUT	newgroup	new group derived from above, in the order defined by ranks (handle)

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

The function MPI_GROUP_INCL creates a group newgroup that consists of the n processes in group with ranks rank[0],. . ., rank[n−1]; the process with rank i in newgroup is the process with rank ranks[i] in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct, or else the program is erroneous. If n = 0, then newgroup is MPI_GROUP_EMPTY. This function can, for instance, be used to reorder the elements of a group. See also MPI_GROUP_COMPARE.

MPI_GROUP_EXCL(group, n, ranks, newgroup)

IN	group	group (handle)
IN	n	number of elements in array ranks (integer)
IN	ranks	array of integer ranks in group not to appear in newgroup
OUT	newgroup	new group derived from above, preserving the order defined by group (handle)

```
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0] ,. . . ranks[n−1]. The ordering of processes in newgroup is identical to the ordering in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct; otherwise, the program is erroneous. If n = 0, then newgroup is identical to group.

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)

IN	group	group (handle)
IN	n	number of triplets in array ranges (integer)
IN	ranges	an array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be included in newgroup
OUT	newgroup	new group derived from above, in the order defined by ranges (handle)

```
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
                MPI_Group *newgroup)
```

```
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

If ranges consist of the triplets

$$(first_1, last_1, stride_1), \ldots, (first_n, last_n, stride_n)$$

then newgroup consists of the sequence of processes in group with ranks

$$first_1, first_1 + stride_1, \ldots, first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, \ldots$$

$$first_n, first_n + stride_n, \ldots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$$

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous. Note that we may have $first_i > last_i$, and $stride_i$ may be negative, but cannot be zero.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the included ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet `(i,i,1)` in the argument ranges.

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)

IN	group	group (handle)
IN	n	number of elements in array ranks (integer)
IN	ranges	a one-dimensional array of integer triplets of the form (first rank, last rank, stride), indicating the ranks in group of processes to be excluded from the output group newgroup.
OUT	newgroup	new group derived from above, preserving the order in group (handle)

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
                         MPI_Group *newgroup)
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the excluded ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_EXCL. A call to MPI_GROUP_EXCL is equivalent to a call to MPI_GROUP_RANGE_EXCL with each rank i in ranks replaced by the triplet `(i,i,1)` in the argument ranges.

Advice to users. The range operations do not explicitly enumerate ranks, and therefore are more scalable if implemented efficiently. Hence, we recommend MPI programmers to use them whenever possible, as high-quality implementations will take advantage of this fact. (*End of advice to users.*)

Advice to implementors. The range operations should be implemented, if possible, without enumerating the group members, in order to obtain better scalability (time and space). (*End of advice to implementors.*)

5.3.3 GROUP DESTRUCTORS

MPI_GROUP_FREE(group)

INOUT	group	group (handle)

```
int MPI_Group_free(MPI_Group *group)

MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR
```

This operation marks a group object for deallocation. The handle group is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to MPI_COMM_CREATE and MPI_COMM_DUP, and decremented for each call to MPI_GROUP_FREE or MPI_COMM_FREE; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

5.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (*End of advice to implementors.*)

5.4.1 COMMUNICATOR ACCESSORS

The following are all local operations.

MPI_COMM_SIZE(comm, size)

IN	comm	communicator (handle)
OUT	size	number of processes in the group of comm (integer)

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

Rationale. This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see below), computing the size using MPI_GROUP_SIZE, and then freeing the group temporary via MPI_GROUP_FREE. However, this function is so commonly used, that this shortcut was introduced. (*End of rationale.*)

Advice to users. This function indicates the number of processes involved in a communicator. For MPI_COMM_WORLD, it indicates the total number of processes available (for this version of MPI, there is no standard way to change the number of processes once initialization has taken place).

This call is often used with the next call to determine the amount of concurrency available for a specific library or program. The following call, MPI_COMM_RANK indicates the rank of the process that calls it in the range from 0...size−1, where size is the return value of MPI_COMM_SIZE. (*End of advice to users.*)

MPI_COMM_RANK(comm, rank)

IN	comm	communicator (handle)
OUT	rank	rank of the calling process in group of comm (integer)

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

```
MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR
```

Rationale. This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see below), computing the size using MPI_GROUP_RANK, and then freeing the group temporary via MPI_GROUP_FREE. However, this function is so commonly used, that this shortcut was introduced. (*End of rationale.*)

Advice to users. This function gives the rank of the process in the particular communicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.

Many programs will be written with the master-slave model, where one process (such as the rank-zero process) will play a supervisory role, and the other processes will serve as compute nodes. In this framework, the two preceding calls are useful for determining the roles of the various processes of a communicator. (*End of advice to users.*)

MPI_COMM_COMPARE(comm1, comm2, result)

IN	comm1	first communicator (handle)
IN	comm2	second communicator (handle)
OUT	result	result (integer)

```
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)
```

```
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR
```

MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical in constituents and rank order; these communicators differ only by context. MPI_SIMILAR results if the group members of both communicators are the same but the rank order differs. MPI_UNEQUAL results otherwise.

5.4.2 COMMUNICATOR CONSTRUCTORS

The following are collective functions that are invoked by all processes in the group associated with comm.

Rationale. Note that there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. The base communicator for all MPI communicators is predefined outside of MPI, and is MPI_COMM_WORLD. This model was arrived at after considerable debate, and was chosen to increase "safety" of programs written in MPI. (*End of rationale.*)

MPI_COMM_DUP(comm, newcomm)

IN	comm	communicator (handle)
OUT	newcomm	copy of comm (handle)

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

```
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
```

MPI_COMM_DUP Duplicates the existing communicator comm with associated key values. For each key value, the respective copy callback function deter-

mines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group, any copied cached information, but a new context (see Section 5.7.1).

Advice to users. This operation is used to provide a parallel library call with a duplicate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see chapter 6). This call is valid even if there are pending point-to-point communications involving the communicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end of the call. Other models of communicator management are also possible.

This call applies to both intra- and inter-communicators. (*End of advice to users.*)

Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information. (*End of advice to implementors.*)

MPI_COMM_CREATE(comm, group, newcomm)

IN	comm	communicator (handle)
IN	group	Group, which is a subset of the group of comm (handle)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR
```

This function creates a new communicator newcomm with communication group defined by group and a new context. No cached information propagates from comm to newcomm. The function returns MPI_COMM_NULL to processes that are not in group. The call is erroneous if not all group arguments have the same value, or if group is not a subset of the group associated with comm. Note that the call is to be executed by all processes in comm, even if they do not belong to the new group. This call applies only to intra-communicators.

Rationale. The requirement that the entire group of comm participate in the call stems from the following considerations:

- It allows the implementation to layer MPI_COMM_CREATE on top of regular collective communications.
- It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.

- It permits implementations sometimes to avoid communication related to context creation. (*End of rationale.*)

Advice to users. MPI_COMM_CREATE provides a means to subset a group of processes for the purpose of separate MIMD computation, with separate communication space. newcomm, which emerges from MPI_COMM_CREATE can be used in subsequent calls to MPI_COMM_CREATE (or other communicator constructors) further to subdivide a computation into parallel sub-computations. A more general service is provided by MPI_COMM_SPLIT, below. (*End of advice to users.*)

Advice to implementors. Since all processes calling MPI_COMM_DUP or MPI_COMM_CREATE provide the same group argument, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

Important: If new communicators are created without synchronizing the processes involved then the communication system should be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (*End of advice to implementors.*)

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assigment (integer)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. A process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for color and key.

A call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where all members of group provide color = 0 and key = rank in group, and all processes

that are not members of group provide color = MPI_UNDEFINED. The function MPI_COMM_SPLIT allows more general partitioning of a group into one or more subgroups with optional reordering. This call applies only to intra-communicators.

Advice to users. This is an extremely powerful mechanism for dividing a single communicating group of processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra.

Multiple calls to MPI_COMM_SPLIT can be used to overcome the requirement that any call have no overlap of the resulting communicators (each process is of only one color per call). In this way, multiple overlapping communication structures can be created. Creative use of the color and key in such splitting operations is encouraged.

Note that, for a fixed color, the keys need not be unique. It is MPI_COMM_SPLIT's responsibility to sort processes in ascending order according to this key, and to break ties in a consistent way. If all the keys are specified in the same way, then all the processes in a given color will have the relative rank order as they did in their parent group. (In general, they will have different ranks.)

Essentially, making the key value zero for all processes of a given color means that one doesn't really care about the rank-order of the processes in the new communicator. (*End of advice to users.*)

5.4.3 COMMUNICATOR DESTRUCTORS

MPI_COMM_FREE(comm)

INOUT	comm	communicator to be destroyed (handle)

```
int MPI_Comm_free(MPI_Comm *comm)

MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR
```

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 5.7) are called in arbitrary order.

Advice to implementors. A reference-count mechanism may be used: the reference count is incremented by each call to MPI_COMM_DUP, and decre-

mented by each call to MPI_COMM_FREE. The object is ultimately deallocated when the count reaches zero.

Though collective, it is anticipated that this operation will normally be implemented to be local, though the debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

5.5 Motivating Examples

5.5.1 CURRENT PRACTICE #1

Example #1a:

```
main(int argc, char **argv)
{
  int me, size;
  ...
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  (void)printf ("Process %d size %d\n", me, size);
  ...
  MPI_Finalize();
}
```

Example #1a is a do-nothing program that initializes itself legally, and refers to the the "all" communicator, and prints a message. It terminates itself legally too. This example does not imply that MPI supports `printf`-like communication itself.

Example #1b (supposing that `size` is even):

```
main(int argc, char **argv)
{
   int me, size;
   int SOME_TAG = 0;
   ...
   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &me);   /* local */
   MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */

   if((me % 2) == 0)
   {
      /* send unless highest-numbered process */
      if((me + 1) < size)
         MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
   }
```

```
    else
        MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD);

    ...
    MPI_Finalize();
}
```

Example #1b schematically illustrates message exchanges between "even" and "odd" processes in the "all" communicator.

5.5.2 CURRENT PRACTICE #2

```
main(int argc, char **argv)
{
  int me, count;
  void *data;
  ...

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  if(me == 0)
  {
      /* get input, create buffer ''data'' */
      ...
  }

  MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);

  ...

  MPI_Finalize();
}
```

This example illustrates the use of a collective communication.

5.5.3 (APPROXIMATE) CURRENT PRACTICE #3

```
main(int argc, char **argv)
{
  int me, count, count2;
  void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
  MPI_Group MPI_GROUP_WORLD, grprem;
  MPI_Comm commslave;
  static int ranks[] = {0};
  ...
  MPI_Init(&argc, &argv);
```

```
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);  /* local */

  MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem);  /* local */
  MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);

  if(me != 0)
  {
    /* compute on slave */
    ...
    MPI_Reduce(send_buf,recv_buff,count, MPI_INT, MPI_SUM, 1, commslave);
    ...
  }
  /* zero falls through immediately to this reduce, others do later... */
  MPI_Reduce(send_buf2, recv_buff2, count2,
             MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  MPI_Comm_free(&commslave);
  MPI_Group_free(&MPI_GROUP_WORLD);
  MPI_Group_free(&grprem);
  MPI_Finalize();
}
```

This example illustrates how a group consisting of all but the zeroth process of the "all" group is created, and then how a communicator is formed (commslave) for that new group. The new communicator is used in a collective call, and all processes execute a collective call in the MPI_COMM_WORLD context. This example illustrates how the two communicators (that inherently possess distinct contexts) protect communication. That is, communication in MPI_COMM_WORLD is insulated from communication in commslave, and vice versa.

In summary, "group safety" is achieved via communicators because distinct contexts within communicators are enforced to be unique on any process.

5.5.4 EXAMPLE #4

The following example is meant to illustrate "safety" between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT      50

main(int argc, char **argv)
{
  int me;
  MPI_Request request[2];
```

```
    MPI_Status status[2];
    MPI_Group MPI_GROUP_WORLD, subgroup;
    int ranks[] = {2, 4, 6, 8};
    MPI_Comm the_comm;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

    MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
    MPI_Group_rank(subgroup, &me);      /* local */

    MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_conn);

    if(me != MPI_UNDEFINED)
    {
        MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                        the_comm, request);
        MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                        the_comm, request+1);
    }

    for(i = 0; i < SOME_COUNT, i++)
      MPI_Reduce(..., the_comm);
    MPI_Waitall(2, request, status);

    MPI_Comm_free(t&he_comm);
    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Group_free(&subgroup);
    MPI_Finalize();
}
```

5.5.5 LIBRARY EXAMPLE #1

The main program:

```
main(int argc, char **argv)
{
  int done = 0;
  user_lib_t *libh_a, *libh_b;
  void *dataset1, *dataset2;
  ...
  MPI_Init(&argc, &argv);
  ...
  init_user_lib(MPI_COMM_WORLD, &libh_a);
  init_user_lib(MPI_COMM_WORLD, &libh_b);
```

```
    ...
    user_start_op(libh_a, dataset1);
    user_start_op(libh_b, dataset2);
    ...
    while(!done)
    {
       /* work */
       ...
       MPI_Reduce(..., MPI_COMM_WORLD);
       ...
       /* see if done */
       ...
    }
    user_end_op(libh_a);
    user_end_op(libh_b);

    uninit_user_lib(libh_a);
    uninit_user_lib(libh_b);
    MPI_Finalize();
}
```

The user library initialization code:

```
void init_user_lib(MPI_Comm comm, user_lib_t **handle)
{
  user_lib_t *save;

  user_lib_initsave(&save); /* local */
  MPI_Comm_dup(comm, &(save -> comm));

  /* other inits */
  ...

  *handle = save;
}
```

User start-up code:

```
void user_start_op(user_lib_t *handle, void *data)
{
  MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
  MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
}
```

User communication clean-up code:

```
void user_end_op(user_lib_t *handle)
```

```
{
    MPI_Status *status;
    MPI_Wait(handle -> isend_handle, status);
    MPI_Wait(handle -> irecv_handle, status);
}
```

User object clean-up code:

```
void uninit_user_lib(user_lib_t *handle)
{
    MPI_Comm_free(&(handle -> comm));
    free(handle);
}
```

5.5.6 LIBRARY EXAMPLE #2

The main program:

```
main(int argc, char **argv)
{
    int ma, mb;
    MPI_Group MPI_GROUP_WORLD, group_a, group_b;
    MPI_Comm comm_a, comm_b;

    static int list_a[] = {0, 1};
#if  defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
    static int list_b[] = {0, 2 ,3};
#else/* EXAMPLE_2A */
    static int list_b[] = {0, 2};
#endif
    int size_list_a = sizeof(list_a)/sizeof(int);
    int size_list_b = sizeof(list_b)/sizeof(int);

    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

    MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
    MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);

    MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
    MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);

    MPI_Comm_rank(comm_a, &ma);
    MPI_Comm_rank(comm_b, &mb);
```

```
    if(ma != MPI_UNDEFINED)
       lib_call(comm_a);

    if(mb != MPI_UNDEFINED)
    {
      lib_call(comm_b);
      lib_call(comm_b);
    }

    MPI_Conn_free(&comm_a);
    MPI_Conn_free(&comm_b);
    MPI_Group_free(&group_a);
    MPI_Group_free(&group_b);
    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Finalize();
  }
```

The library:

```
  void lib_call(MPI_Conn comm)
  {
    int me, done = 0;
    MPI_Comm_rank(comm, &me);
    if(me == 0)
       while(!done)
       {
          MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm);
          ...
       }
    else
    {
      /* work */
      MPI_Send(..., 0, ARBITRARY_TAG, comm);
      ....
    }
#ifdef EXAMPLE_2C
    /* include (resp, exclude) for safety (resp, no safety): */
    MPI_Barrier(comm);
#endif
  }
```

The above example is really three examples, depending on whether or not one includes rank 3 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates that, despite contexts, subsequent calls to lib_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is added. What this demonstrates

is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back-masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no back-masking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [28]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity—deleting either feature removes the guarantee that back-masking cannot be required.

Algorithms that try to do non-deterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of "collective calls" implemented with point-to-point operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 5.8.

5.6 Inter-Communication

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All point-to-point communication described thus far has involved communication between processes that are members of the same group. This type of communication is called "intra-communication" and the communicator used is called an "intra-communicator," as we have noted earlier in the chapter.

In modular and multi-disciplinary applications, different process groups execute distinct modules and processes within different modules communicate with one another in a pipeline or a more general module graph. In these applications, the most natural way for a process to specify a target process is by the rank of the target process within the target group. In applications that contain internal user-level servers, each server may be a process group that provides services to one or more clients, and each client may be a process group that uses the services of one or more servers. It is again most natural to specify the target process by rank within the target group in these applications. This type of communication is called "inter-communication" and the communicator used is called an "inter-communicator," as introduced earlier.

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the "local group," that is, the sender in a send and the receiver in a receive. The group containing the target process

is called the "remote group," that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking and require that the local and remote groups be disjoint in order to avoid deadlock.

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point communication is the same for both inter- and intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- An inter-communicator cannot be used for collective communication.
- A communicator will provide either intra- or inter-communication, never both.

The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_COMM_CREATE).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

group

send_context

receive_context

source

For inter-communicators, **group** describes the remote group, and **source** is the rank of the process in the local group. For intra-communicators, **group** is the communicator group (remote=local), **source** is the rank of the process in this group, and **send context** and **receive context** are identical. A group is represented by a rank-to-absolute-address translation table.

The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process **P** in group $\mathcal{P}$, which has an inter-communicator $\mathbf{C}_{\mathcal{P}}$, and a process **Q** in group $\mathcal{Q}$, which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$. Then

- $\mathbf{C}_{\mathcal{P}}$.**group** describes the group $\mathcal{Q}$ and $\mathbf{C}_{\mathcal{Q}}$.**group** describes the group $\mathcal{P}$.

- C_P.**send_context** = C_Q.**receive_context** and the context is unique in Q; C_P.**receive_context** = C_Q.**send_context** and this context is unique in P.
- C_P.**source** is rank of **P** in P and C_Q.**source** is rank of **Q** in Q.

Assume that **P** sends a message to **Q** using the inter-communicator. Then **P** uses the **group** table to find the absolute address of **Q**; **source** and **send_context** are appended to the message.

Assume that **Q** posts a receive with an explicit source argument using the inter-communicator. Then **Q** matches **receive_context** to the message context and source argument to the message source.

The same algorithm is appropriate for intra-communicators as well.

In order to support inter-communicator accessors and constructors, it is necessary to supplement this model with additional structures, that store information about the local communication group, and additional safe contexts. (*End of advice to implementors.*)

5.6.1 INTER-COMMUNICATOR ACCESSORS

MPI_COMM_TEST_INTER(comm, flag)

IN	comm	communicator (handle)
OUT	flag	(logical)

```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
```

```
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG
```

This local routine allows the calling process to determine if a communicator is an inter-communicator or an intra-communicator. It returns true if it is an inter-communicator, otherwise false.

When an inter-communicator is used as an input argument to the communicator accessors described above under intra-communication, the following table describes behavior.

MPI_COMM_* Function Behavior (in Inter-Communication Mode)	
MPI_COMM_SIZE	returns the size of the local group.
MPI_COMM_GROUP	returns the local group.
MPI_COMM_RANK	returns the rank in the local group.

Furthermore, the operation MPI_COMM_COMPARE is valid for inter-communicators. Both communicators must be either intra- or inter-communicators, or else MPI_UNEQUAL results. Both corresponding local and remote groups must compare correctly to get the results MPI_CONGRUENT and MPI_SIMILAR. In par-

ticular, it is possible for MPI_SIMILAR to result because either the local or remote groups were similar but not identical.

The following accessors provide consistent access to the remote group of an inter-communicator:

The following are all local operations.

MPI_COMM_REMOTE_SIZE(comm, size)

IN	comm	inter-communicator (handle)
OUT	size	number of processes in the remote group of comm (integer)

```
int MPI_Conn_remote_size(MPI_Conn comm, int *size)

MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

MPI_COMM_REMOTE_GROUP(comm, group)

IN	comm	inter-communicator (handle)
OUT	group	remote group corresponding to comm (handle)

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)

MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
```

Rationale. Symmetric access to both the local and remote groups of an inter-communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE have been provided. (*End of rationale.*)

5.6.2 INTER-COMMUNICATOR OPERATIONS

This section introduces four blocking inter-communicator operations. MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-communicator; the function MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote groups of an inter-communicator. The functions MPI_COMM_DUP and MPI_COMM_FREE, introduced previously, duplicate and free an inter-communicator, respectively.

Overlap of local and remote groups that are bound into an inter-communicator is prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If a process is multi-threaded, and MPI calls block only a thread, rather than a process, then "dual membership" can be supported. It is then the user's responsibility to make sure that calls on behalf of the two "roles" of a process are executed by two independent threads.)

The function MPI_INTERCOMM_CREATE can be used to create an inter-

communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator (the two leaders could be the same process). Furthermore, members of each group know the rank of their leader.

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

In standard MPI implementations (with static process allocation at initialization), the MPI_COMM_WORLD communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. In dynamic MPI implementations, where, for example, a process may spawn new child processes during an MPI execution, the parent process may be the "bridge" between the old communication universe and the new communication world that includes the parent and its children.

The application topology functions described in chapter 6 do not apply to inter-communicators. Users that require this capability should utilize MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topology-oriented intra-communicator. Alternatively, it may be reasonable to devise one's own application topology mechanisms for this case, without loss of generality.

MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm)

IN	local_comm	local intra-communicator (handle)
IN	local_leader	rank of local group leader in local_comm (integer)
IN	peer_comm	"peer" intra-communicator; significant only at the local_leader (handle)
IN	remote_leader	rank of remote group leader in peer_comm; significant only at the local_leader (integer)
IN	tag	"safe" tag (integer)
OUT	newintercomm	new inter-communicator (handle)

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
        MPI_Comm peer_comm, int remote_leader, int tag,
        MPI_Comm *newintercomm)
```

```
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
                NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR
```

This call creates an inter-communicator. It is collective over the union of the local and remote groups. Processes should provide identical local_comm and local_leader arguments within each group. Wildcards are not permitted for remote_leader, local_leader, and tag.

This call uses point-to-point communication with communicator peer_comm, and with tag tag between the leaders. Thus, care must be taken that there be no pending communication on peer_comm that could interfere with this communication.

> *Advice to users.* We recommend using a dedicated peer communicator, such as a duplicate of MPI_COMM_WORLD, to avoid trouble with peer communicators. (*End of advice to users.*)

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)

IN	intercomm	Inter-Communicator (handle)
IN	high	(logical)
OUT	newintracomm	new intra-communicator (handle)

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)
```

```
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
    INTEGER INTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH
```

This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the "low" group before the "high" group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

> *Advice to implementors.* The implementation of MPI_INTERCOMM_MERGE, MPI_COMM_FREE and MPI_COMM_DUP are similar to the implementation of MPI_INTERCOMM_CREATE, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)