D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

October 21, 2019
This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 6

Groups, Contexts, Communicators, and Caching

6.1 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 [7] and [1] for further information on writing libraries in MPI, using the features described in this chapter.

6.1.1 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.

6.1.2 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 [4, 6, 8]) 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 operations between two groups of processes.

Caching. Communicators (see below) provide a "caching" mechanism that allows one to associate new attributes with communicators, on 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 7 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 7 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 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.

- 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 and collective communication in an related manner to intra-communicators. Users who do not need inter-communication in their applications can safely ignore this extension. Users who require inter-communication between overlapping groups must layer this capability on top of MPI.

6.2 Basic Concepts

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

6.2.1 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 pre-defined 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, is 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.*)

6.2.2 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 communication within one communicator do not interfere, and that communication 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, 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.*)

6.2.3 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 7), communicators may also "cache" additional information (see Section 6.7). 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.

6.2.4 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 or MPI_INIT_THREAD 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 represent disjoint groups 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 pre-defined 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.

6.3 Group Management

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

6.3.1 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)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(OUT) :: rank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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, const int ranks1[],
             MPI_Group group2, int ranks2[])
```

```
MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    INTEGER, INTENT(IN) ::  n, ranks1(n)
    INTEGER, INTENT(OUT) ::  ranks2(n)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL as the translated rank.

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)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    INTEGER, INTENT(OUT) ::  result
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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.

6.3.2 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)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Group), INTENT(OUT) ::  group
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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 the first group.

intersect all elements of the first group that are also in the second group, ordered as in the 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, const int ranks[],
             MPI_Group *newgroup)
```

```
MPI_Group_incl(group, n, ranks, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranks(n)
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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 ranks[0],..., ranks[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, const int ranks[],
             MPI_Group *newgroup)
```

```
MPI_Group_excl(group, n, ranks, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranks(n)
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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	a one-dimensional 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)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranges(3,n)
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

If ranges consists 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 ranges (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)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranges(3,n)
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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 whenenever 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.*)

6.3.3 Group Destructors

MPI_GROUP_FREE(group)

INOUT	group	group (handle)

```
int MPI_Group_free(MPI_Group *group)

MPI_Group_free(group, ierror)
    TYPE(MPI_Group), INTENT(INOUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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_GROUP, MPI_COMM_CREATE, MPI_COMM_DUP, and MPI_COMM_IDUP, 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.*)

6.4 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.*)

6.4.1 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)
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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 above), computing the size using MPI_GROUP_SIZE, and then freeing the temporary group 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 unless the number of processes has been changed by using the functions described in Chapter 10; note that the number of processes in MPI_COMM_WORLD does not change during the life of an MPI program.

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 $0 \ldots \mathsf{size}-1$, 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)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: rank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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 above), computing the rank using MPI_GROUP_RANK, and then freeing the temporary group 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)
    TYPE(MPI_Comm), INTENT(IN) ::  comm1, comm2
    INTEGER, INTENT(OUT) ::  result
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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.

6.4.2 Communicator Constructors

The following are collective functions that are invoked by all processes in the group or groups associated with comm, with the exception of MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.

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.*)

This chapter presents the following communicator construction routines: MPI_COMM_CREATE, MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO and MPI_COMM_SPLIT can be used to create both intracommunicators and intercommunicators; MPI_COMM_CREATE_GROUP and MPI_INTERCOMM_MERGE (see Section 6.6.2) can be used to create intracommunicators; and MPI_INTERCOMM_CREATE (see Section 6.6.2) can be used to create intercommunicators.

An intracommunicator involves a single group while an intercommunicator involves two groups. Where the following discussions address intercommunicator semantics, the two groups in an intercommunicator are called the *left* and *right* groups. A process in an intercommunicator is a member of either the left or the right group. From the point of view of that process, the group that the process is a member of is called the *local group*; the other group (relative to that process) is the *remote group*. The left and right group labels give us a way to describe the two groups in an intercommunicator that is not relative to any particular process (as the local and remote groups are).

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)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

MPI_COMM_DUP duplicates the existing communicator comm with associated key values and topology information. For each key value, the respective copy callback function determines 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. MPI_COMM_DUP returns in newcomm a new communicator with the same group or groups, same topology, and any copied cached information, but a new context (see Section 6.7.1).

Advice to users. This operation is used to provide a parallel library with a duplicate communication space that has the same properties as the original communicator. This includes any attributes (see below) and topologies (see Chapter 7). 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_DUP_WITH_INFO(comm, info, newcomm)

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

```
int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
```

```
MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
    INTEGER COMM, INFO, NEWCOMM, IERROR
```

MPI_COMM_DUP_WITH_INFO behaves exactly as MPI_COMM_DUP except that the hints provided by the argument info are associated with the output communicator newcomm.

Rationale. It is expected that some hints will only be valid at communicator creation time. However, for legacy reasons, most communicator creation calls do not provide an info argument. One may associate info hints with a duplicate of any communicator at creation time through a call to MPI_COMM_DUP_WITH_INFO. (*End of rationale.*)

MPI_COMM_IDUP(comm, newcomm, request)

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

```
int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
```

```
MPI_Comm_idup(comm, newcomm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS ::  newcomm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
    INTEGER COMM, NEWCOMM, REQUEST, IERROR
```

MPI_COMM_IDUP is a nonblocking variant of MPI_COMM_DUP. With the exception of its nonblocking behavior, the semantics of MPI_COMM_IDUP are as if MPI_COMM_DUP was executed at the time that MPI_COMM_IDUP is called. For example, attributes changed after MPI_COMM_IDUP will not be copied to the new communicator. All restrictions and assumptions for nonblocking collective operations (see Section 5.12) apply to MPI_COMM_IDUP and the returned request.

It is erroneous to use the communicator newcomm as an input argument to other MPI functions before the MPI_COMM_IDUP operation completes.

MPI_COMM_IDUP_WITH_INFO(comm, info, newcomm, request)

IN	comm	communicator (handle)
IN	info	info object (handle)
OUT	newcomm	copy of comm (handle)
OUT	request	communication request (handle)

```
int MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info,
              MPI_Comm *newcomm, MPI_Request *request)
```

```
MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS ::  newcomm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR)
    INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR
```

MPI_COMM_IDUP_WITH_INFO is a nonblocking variant of MPI_COMM_DUP_WITH_INFO. With the exception of its nonblocking behavior, the semantics of MPI_COMM_IDUP_WITH_INFO are as if MPI_COMM_DUP_WITH_INFO was executed at the time that MPI_COMM_IDUP_WITH_INFO is called. For example, attributes or info hints changed after MPI_COMM_IDUP_WITH_INFO will not be copied to the new communicator. All restrictions and assumptions for nonblocking collective operations (see Section 5.12) apply to MPI_COMM_IDUP_WITH_INFO and the returned request.

It is erroneous to use the communicator newcomm as an input argument to other MPI functions before the MPI_COMM_IDUP_WITH_INFO operation completes.

Rationale. The MPI_COMM_IDUP and MPI_COMM_IDUP_WITH_INFO functions are crucial for the development of purely nonblocking libraries (see [5]). (*End of rationale.*)

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)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Group), INTENT(IN) ::  group
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

If comm is an intracommunicator, this function returns a new communicator newcomm with communication group defined by the group argument. No cached information propagates from comm to newcomm. Each process must call MPI_COMM_CREATE with a group argument that is a subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. The processes may specify different values for the group argument. If a process calls with a non-empty group then all processes in that group must call the function with the same group as argument, that is the same processes in the same order. Otherwise, the call is erroneous. This implies that the set of groups specified across the processes must be disjoint. If the calling process is a member of the group given as group argument, then newcomm is a communicator with group as its associated group. In the case that a process calls with a group to which it does not belong, e.g., MPI_GROUP_EMPTY, then MPI_COMM_NULL is returned as newcomm. The function is collective and must be called by all processes in the group of comm.

Rationale. The interface supports the original mechanism from MPI-1.1, which required the same group in all processes of comm. It was extended in MPI-2.2 to allow the use of disjoint subgroups in order to allow implementations to eliminate unnecessary communication that MPI_COMM_SPLIT would incur when the user already knows the membership of the disjoint subgroups. (*End of rationale.*)

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 to sometimes 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) to further 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. When calling MPI_COMM_DUP, all processes call with the same group (the group associated with the communicator). When calling MPI_COMM_CREATE, the processes provide the same group or disjoint subgroups. For both calls, 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 must 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.*)

If comm is an intercommunicator, then the output communicator is also an intercommunicator where the local group consists only of those processes contained in group (see Figure 6.1). The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of newcomm. All processes in the same local group of comm must specify the same value for group, i.e., the same members in the same order. If either group does not specify at least one process in the local group of the intercommunicator, or if the calling process is not included in the group, MPI_COMM_NULL is returned.

Rationale. In the case where either the left or right group is empty, a null communicator is returned instead of an intercommunicator with MPI_GROUP_EMPTY because the side with the empty group must return MPI_COMM_NULL. (*End of rationale.*)

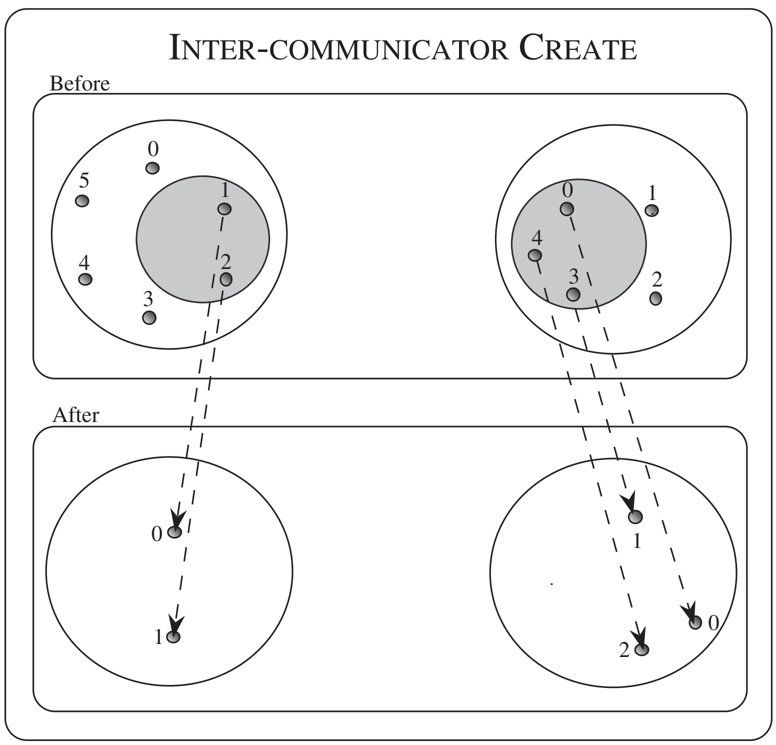

Figure 6.1: Intercommunicator creation using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

Example 6.1 The following example illustrates how the first node in the left side of an intercommunicator could be joined with all members on the right side of an intercommunicator to form a new intercommunicator.

```
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 intercommunicator: "inter_comm" */
...

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

MPI_Comm_create(inter_comm, group, &new_inter_comm);
MPI_Group_free(&group);
```

MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)

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

```
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,
             MPI_Comm *newcomm)
```

```
MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  tag
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
```

MPI_COMM_CREATE_GROUP is similar to MPI_COMM_CREATE; however, MPI_COMM_CREATE must be called by all processes in the group of comm, whereas MPI_COMM_CREATE_GROUP must be called by all processes in group, which is a subgroup of the group of comm. In addition, MPI_COMM_CREATE_GROUP requires that comm is an intracommunicator. MPI_COMM_CREATE_GROUP returns a new intracommunicator, newcomm, for which the group argument defines the communication group. No cached information propagates from comm to newcomm. Each process must provide a group argument that is a subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. If a non-empty group is specified, then all processes in that group must call the function, and each of these processes must provide the same arguments, including a group that contains the same members with the same ordering. Otherwise the call is erroneous. If the calling process is a member of the group given as the group argument, then newcomm is a communicator with group as its associated group. If the calling process is not a member of group, e.g., group is MPI_GROUP_EMPTY, then the call is a local operation and MPI_COMM_NULL is returned as newcomm.

> *Rationale.* Functionality similar to MPI_COMM_CREATE_GROUP can be implemented through repeated MPI_INTERCOMM_CREATE and MPI_INTERCOMM_MERGE calls that start with the MPI_COMM_SELF communicators at each process in group and build up an intracommunicator with group group [3]. Such an algorithm requires the creation of many intermediate communicators; MPI_COMM_CREATE_GROUP can provide a more efficient implementation that avoids this overhead. (*End of rationale.*)

> *Advice to users.* An intercommunicator can be created collectively over processes in the union of the local and remote groups by creating the local communicator using MPI_COMM_CREATE_GROUP and using that communicator as the local communicator argument to MPI_INTERCOMM_CREATE. (*End of advice to users.*)

The tag argument does not conflict with tags used in point-to-point communication and is not permitted to be a wildcard. If multiple threads at a given process perform concurrent MPI_COMM_CREATE_GROUP operations, the user must distinguish these operations by providing different tag or comm arguments.

> *Advice to users.* MPI_COMM_CREATE may provide lower overhead than MPI_COMM_CREATE_GROUP because it can take advantage of collective communication on comm when constructing newcomm. (*End of advice to users.*)

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assignment (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)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  color, key
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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.

With an intracommunicator comm, a call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where processes that are members of their group argument provide color = number of the group (based on a unique numbering of all disjoint groups) and key = rank in group, and all processes that are not members of their group argument provide color = MPI_UNDEFINED.

The value of color must be non-negative or MPI_UNDEFINED.

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. For intracommunicators, MPI_COMM_SPLIT provides similar capability as MPI_COMM_CREATE to split a communicating group into disjoint subgroups. MPI_COMM_SPLIT is useful when some processes do not have complete information of the other members in their group, but all processes know (the color of) the group to which they belong. In this case, the MPI implementation discovers the other group members via communication. MPI_COMM_CREATE is useful when all processes have complete information of the members of their group. In this case, MPI can avoid the extra communication required to discover group membership. MPI_COMM_CREATE_GROUP is useful when all processes in a given group have complete information of the members of their group and synchronization with processes outside the group can be avoided.

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.

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

Rationale. color is restricted to be non-negative, so as not to confict with the value assigned to MPI_UNDEFINED. (*End of rationale.*)

The result of MPI_COMM_SPLIT on an intercommunicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see Figure 6.2). For those colors that are specified only on one side of the intercommunicator, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned to those processes that specify MPI_UNDEFINED as the color.

> *Advice to users.* For intercommunicators, MPI_COMM_SPLIT is more general than MPI_COMM_CREATE. A single call to MPI_COMM_SPLIT can create a set of disjoint intercommunicators, while a call to MPI_COMM_CREATE creates only one. (*End of advice to users.*)

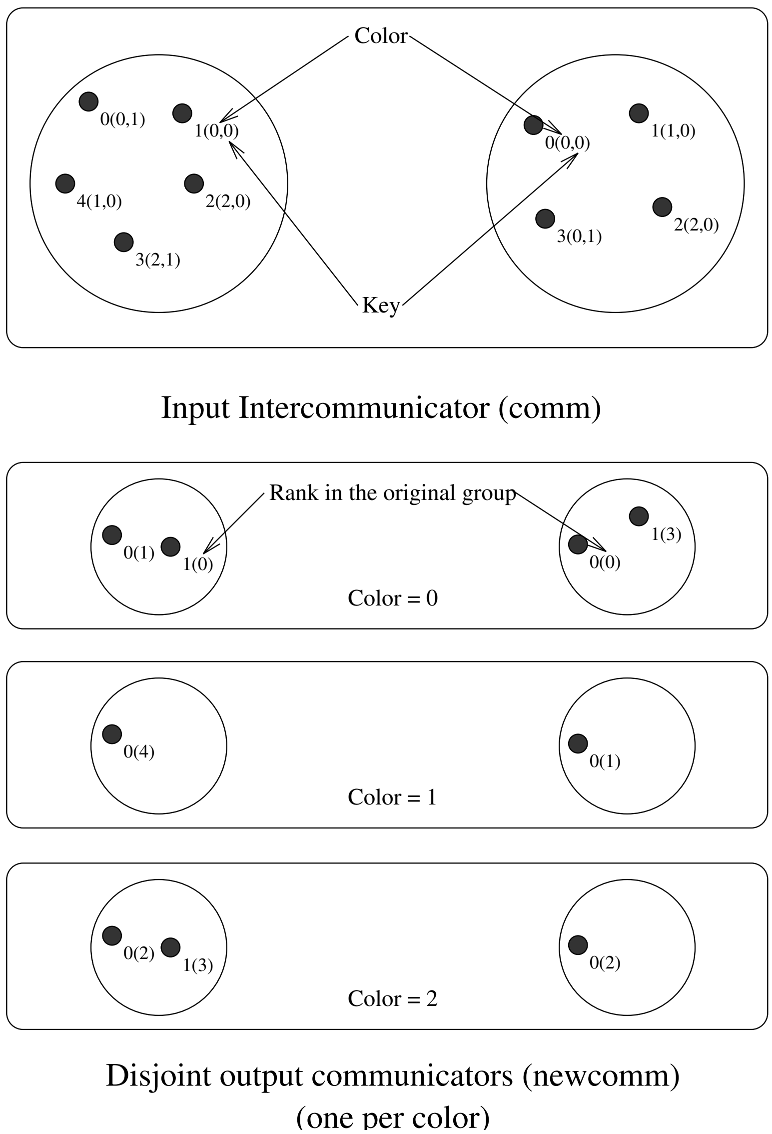

Figure 6.2: Intercommunicator construction achieved by splitting an existing intercommunicator with MPI_COMM_SPLIT extended to intercommunicators.

Example 6.2 (Parallel client-server model). The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

```
/* Client code */
MPI_Comm  multiple_server_comm;
MPI_Comm  single_server_comm;
int       color, rank, num_servers;

/* Create intercommunicator 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 intercommunicator */
MPI_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_server_comm;
int       rank;

/* Create intercommunicator with clients and servers:
   multiple_client_comm */
...

/* Split the intercommunicator for a single server per group
   of clients */
MPI_Comm_rank(multiple_client_comm, &rank);
MPI_Comm_split(multiple_client_comm, rank, 0,
               &single_server_comm);
```

MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)

IN	comm	communicator (handle)
IN	split_type	type of processes to be grouped together (integer)
IN	key	control of rank assignment (integer)
IN	info	info argument (handle)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
             MPI_Info info, MPI_Comm *newcomm)
```

```
MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: split_type, key
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
    INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
```

This function partitions the group associated with comm into disjoint subgroups, based on the type specified by split_type. Each subgroup contains all processes of the same type. 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. This is a collective call; all processes must provide the same split_type, but each process is permitted to provide different values for key. An exception to this rule is that a process may supply the type value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL .

For split_type, the following values are defined by MPI:

MPI_COMM_TYPE_SHARED — this type splits the communicator into subcommunicators, each of which can create a shared memory region.

MPI_COMM_TYPE_HW_SUBDOMAIN — all MPI processes in the group associated with newcomm share the same **hardware resource** (e.g., a network switch, a computing core, an L3 cache, a GPU) to which they are restricted.

Advice to implementors. A high quality implementation will return in the group of the output communicator newcomm the largest subset of MPI processes that fit the splitting criterion. (*End of advice to implementors.*)

Advice to users. The set of hardware resources to which an MPI process is restricted may change during the application execution (e.g., because of process relocation), in which case the communicators created with the value MPI_COMM_TYPE_HW_SUBDOMAIN before this change may not reflect the future hardware locality of such process. (*End of advice to users.*)

The user *constrains* with the info argument the splitting of the input communicator comm. To this end, the info key mpi_hw_subdomain_type is reserved and its value is an implementation defined string designating the type of the requested hardware resource (e.g., "NUMANode", "Package" or "L3Cache").

Advice to users. The set of implementation defined strings recognized by the MPI implementation can be retrieved with a call to the routine MPI_GET_HW_SUBDOMAIN_TYPES. (*End of advice to users.*)

As an exception, the value mpi_shared_memory is defined and reserved in order to produce the same communicators as the ones that would be created if the MPI_COMM_TYPE_SHARED value was used for the split_type parameter.

Rationale. The value mpi_shared_memory is defined in order to ensure consistency between the use of MPI_COMM_TYPE_SHARED and the use of MPI_COMM_TYPE_HW_SUBDOMAIN. (*End of rationale.*)

This mpi_hw_subdomain_type *info key is not a hint and is required (i.e., it must be provided and cannot be ignored by the* MPI *implementation) to perform the splitting operation, otherwise the result is implementation dependent.*

Advice to users. In heterogenous systems, the same value for the info key mpi_hw_subdomain_type may designate different hardware resource types (e.g., "LastLevelCache"). (*End of advice to users.*)

If the value provided for the info key mpi_hw_subdomain_type is not recognized by the MPI implementation or if all MPI processes involved in the splitting operation do not provide the same value, then
Solution 1:
comm and newcomm are handles for the same communicator object; the argument key is ignored and no new communicator is created by the call. More specifically, a call to the routine MPI_COMM_COMPARE(comm, newcomm, result) shall return MPI_IDENT in result regardless of the value of the parameter key.

Solution 2:
newcomm is a copy of the input communicator comm. The processes in the group associated with newcomm are ranked in the order defined by the value of the argument key with ties broken according to their rank in the group associated with comm. More specifically, a call to the routine MPI_COMM_COMPARE(comm, newcomm, result) shall return MPI_CONGRUENT or MPI_SIMILAR in result, depending on the value of the parameter key.

If the calling MPI process is not restricted to a particular instance of a resource of the requested type specified by the info key value then MPI_COMM_NULL is returned in newcomm for such process.

Example 6.3 (Splitting MPI_COMM_WORLD into NUMANode subcommunicators).

```
MPI_Info info;
MPI_Comm hwcomm;
int rank;

MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Info_create(&info);
MPI_Info_set(info,"mpi_hw_subdomain_type","NUMANode");
MPI_Comm_split_type(MPI_COMM_WORLD,
                    MPI_COMM_TYPE_HW_SUBDOMAIN,
                    rank,info,&hwcomm);
```

Advice to implementors. Implementations can define their own split_type values, or use the info argument, to assist in creating communicators that help expose platform-specific information to the application. (*End of advice to implementors.*)

6.4.3 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)
    TYPE(MPI_Comm), INTENT(INOUT) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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 6.7) are called in arbitrary order.

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

6.4.4 Communicator Info

Hints specified via info (see Chapter 9) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or minimize use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_COMM_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per communicator basis, in MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_SPLIT_TYPE, MPI_DIST_GRAPH_CREATE, and MPI_DIST_GRAPH_CREATE_ADJACENT, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_COMM_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

> *Advice to implementors.* It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

Info hints are not propagated by MPI from one communicator to another. The following info keys are valid for all communicators.

mpi_assert_no_any_tag (boolean, default: false): If set to true, then the implementation may assume that the process will not use the MPI_ANY_TAG wildcard on the given communicator.

mpi_assert_no_any_source (boolean, default: false): If set to true, then the implementation may assume that the process will not use the MPI_ANY_SOURCE wildcard on the given communicator.

mpi_assert_exact_length (boolean, default: false): If set to true, then the implementation may assume that the lengths of messages received by the process are equal to the lengths of the corresponding receive buffers, for point-to-point communication operations on the given communicator.

mpi_assert_allow_overtaking (boolean, default: false): If set to true, then the implementation may assume that point-to-point communications on the given communicator do not rely on the non-overtaking rule specified in Section 3.5. In other words, the application asserts that send operations are not required to be matched at the receiver in the order in which the send operations were posted by the sender, and receive operations are not required to be matched in the order in which they were posted by the receiver.

Advice to users. Use of the mpi_assert_allow_overtaking info key can result in nondeterminism in the message matching order. (*End of advice to users.*)

Advice to users. Some optimizations may only be possible when all processes in the group of the communicator provide a given info key with the same value. (*End of advice to users.*)

MPI_COMM_SET_INFO(comm, info)

INOUT	comm	communicator (handle)
IN	info	info object (handle)

```
int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
```

```
MPI_Comm_set_info(comm, info, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_COMM_SET_INFO(COMM, INFO, IERROR)
    INTEGER COMM, INFO, IERROR
```

MPI_COMM_SET_INFO updates the hints of the communicator associated with comm using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI_COMM_SET_INFO. MPI_COMM_SET_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Advice to users. Some info items that an implementation can use when it creates a communicator cannot easily be changed once the communicator has been created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI_COMM_SET_INFO. MPI_COMM_GET_INFO can be used to determine whether updates to existing info hints were ignored by the implementation. (*End of advice to users.*)

Advice to users. Setting info hints on the predefined communicators MPI_COMM_WORLD and MPI_COMM_SELF may have unintended effects, as changes to these global objects may affect all components of the application, including libraries and tools. Users must ensure that all components of the application that use a given communicator, including libraries and tools, can comply with any info hints associated with that communicator. (*End of advice to users.*)

MPI_COMM_GET_INFO(comm, info_used)

IN	comm	communicator object (handle)
OUT	info_used	new info object (handle)

```
int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
```

```
MPI_Comm_get_info(comm, info_used, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(OUT) ::  info_used
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
    INTEGER COMM, INFO_USED, IERROR
```

MPI_COMM_GET_INFO returns a new info object containing the hints of the communicator associated with comm. The current setting of all hints related to this communicator is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

6.5 Motivating Examples

6.5.1 Current Practice #1

Example #1a:

```
int 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();
  return 0;
}
```

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

```
int 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, &status);

   ...
   MPI_Finalize();
   return 0;
}
```

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

6.5.2 Current Practice #2

```
int 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();
    return 0;
  }
```

This example illustrates the use of a collective communication.

6.5.3 (Approximate) Current Practice #3

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

    MPI_Group_excl(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_buf,count, MPI_INT, MPI_SUM, 1, commslave);
      ...
      MPI_Comm_free(&commslave);
    }
    /* zero falls through immediately to this reduce, others do later... */
    MPI_Reduce(send_buf2, recv_buf2, count2,
               MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
  MPI_Group_free(&group_world);
  MPI_Group_free(&grprem);
  MPI_Finalize();
  return 0;
}
```

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.

6.5.4 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

int main(int argc, char *argv[])
{
  int me;
  MPI_Request request[2];
  MPI_Status status[2];
  MPI_Group group_world, subgroup;
  int ranks[] = {2, 4, 6, 8};
  MPI_Comm the_comm;
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &group_world);

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

  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);

  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(&the_comm);
      }

      MPI_Group_free(&group_world);
      MPI_Group_free(&subgroup);
      MPI_Finalize();
      return 0;
    }
```

6.5.5 Library Example #1

The main program:

```
int 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();
  return 0;
}
```

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);
}
```

6.5.6 Library Example #2

The main program:

```
   int main(int argc, char *argv[])
   {
     int ma, mb;
     MPI_Group 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, &group_world);

     MPI_Group_incl(group_world, size_list_a, list_a, &group_a);
     MPI_Group_incl(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);

     if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
     if(comm_b != MPI_COMM_NULL)
        MPI_Comm_rank(comm_b, &mb);

     if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);

     if(comm_b != MPI_COMM_NULL)
     {
       lib_call(comm_b);
       lib_call(comm_b);
     }

     if(comm_a != MPI_COMM_NULL)
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
       MPI_Comm_free(&comm_b);
     MPI_Group_free(&group_a);
     MPI_Group_free(&group_b);
     MPI_Group_free(&group_world);
     MPI_Finalize();
     return 0;
   }
```

The library:

```
   void lib_call(MPI_Comm comm)
   {
     int me, done = 0;
     MPI_Status status;
     MPI_Comm_rank(comm, &me);
     if(me == 0)
```

```
        while(!done)
        {
           MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
           ...
        }
     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 [8]). 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 6.9.

6.6 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 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 except for MPI_COMM_IDUP and require that the local and remote groups be disjoint.

> *Advice to users.* The groups must be disjoint for several reasons. Primarily, this is the intent of the intercommunicators — to provide a communicator for communication between disjoint groups. This is reflected in the definition of MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the processes in the created intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to inter-communicators makes the most sense when the groups are disjoint. (*End of advice to users.*)

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

- The syntax of point-to-point and collective 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.
- 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_CART_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 can be 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 $\mathcal{P}$.
- $\mathbf{C}_{\mathcal{P}}$**.send_context** = $\mathbf{C}_{\mathcal{Q}}$**.receive_context** and the context is unique in $\mathcal{Q}$; $\mathbf{C}_{\mathcal{P}}$**.receive_context** = $\mathbf{C}_{\mathcal{Q}}$**.send_context** and this context is unique in $\mathcal{P}$.
- $\mathbf{C}_{\mathcal{P}}$**.source** is rank of **P** in $\mathcal{P}$ and $\mathbf{C}_{\mathcal{Q}}$**.source** is rank of **Q** in $\mathcal{Q}$.

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.*)

6.6.1 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)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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_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

Table 6.1: MPI_COMM_* Function Behavior (in Inter-Communication Mode)

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 or MPI_SIMILAR. In particular, 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_Comm_remote_size(MPI_Comm comm, int *size)
```

```
MPI_Comm_remote_size(comm, size, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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)
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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.*)

6.6.2 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 multithreaded, 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. 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. For applications that have used spawn or join, it may be necessary to first create an intracommunicator to be used as peer.

The application topology functions described in Chapter 7 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" 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	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)
    TYPE(MPI_Comm), INTENT(IN) ::  local_comm, peer_comm
    INTEGER, INTENT(IN) ::  local_leader, remote_leader, tag
    TYPE(MPI_Comm), INTENT(OUT) ::  newintercomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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.

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, newintracomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  intercomm
    LOGICAL, INTENT(IN) ::  high
    TYPE(MPI_Comm), INTENT(OUT) ::  newintracomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
```

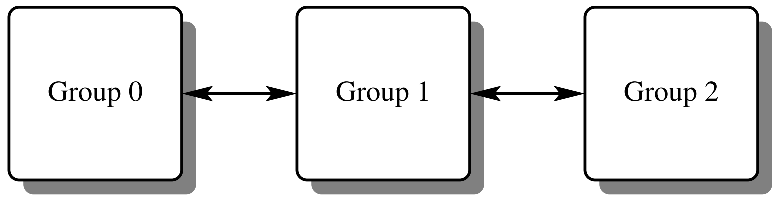

Figure 6.3: Three-group pipeline

```
INTEGER INTERCOMM, NEWINTRACOMM, 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.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

> *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.*)

6.6.3 Inter-Communication Examples

Example 1: Three-Group "Pipeline"

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1 inter-communicator.

```
int main(int argc, char *argv[])
{
  MPI_Comm   myComm;       /* intra-communicator of local sub-group */
  MPI_Comm   myFirstComm;  /* inter-communicator */
  MPI_Comm   mySecondComm; /* second inter-communicator (group 1 only) */
  int membershipKey;
  int rank;

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

  /* User code must generate membershipKey in the range [0, 1, 2] */
  membershipKey = rank % 3;
```

```
    /* Build intra-communicator for local sub-group */
    MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

    /* Build inter-communicators.  Tags are hard-coded. */
    if (membershipKey == 0)
    {                     /* Group 0 communicates with group 1. */
      MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
                           1, &myFirstComm);
    }
    else if (membershipKey == 1)
    {              /* Group 1 communicates with groups 0 and 2. */
      MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
                           1, &myFirstComm);
      MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
                           12, &mySecondComm);
    }
    else if (membershipKey == 2)
    {                     /* Group 2 communicates with group 1. */
      MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
                           12, &myFirstComm);
    }

    /* Do work ... */

    switch(membershipKey)  /* free communicators appropriately */
    {
    case 1:
       MPI_Comm_free(&mySecondComm);
    case 0:
    case 2:
       MPI_Comm_free(&myFirstComm);
       break;
    }

    MPI_Finalize();
    return 0;
  }
```

Example 2: Three-Group "Ring"

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

```
  int main(int argc, char *argv[])
  {
    MPI_Comm   myComm;      /* intra-communicator of local sub-group */
    MPI_Comm   myFirstComm; /* inter-communicators */
```

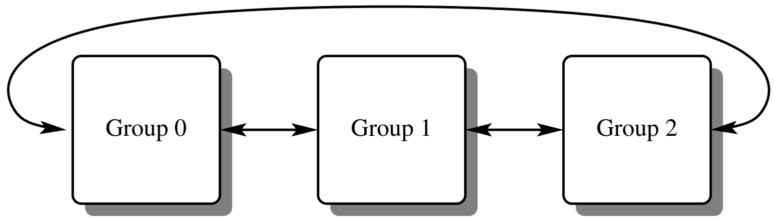

Figure 6.4: Three-group ring

```
MPI_Comm   mySecondComm;
int membershipKey;
int rank;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...

/* User code must generate membershipKey in the range [0, 1, 2] */
membershipKey = rank % 3;

/* Build intra-communicator for local sub-group */
MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

/* Build inter-communicators.  Tags are hard-coded. */
if (membershipKey == 0)
{             /* Group 0 communicates with groups 1 and 2. */
  MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
                       1, &myFirstComm);
  MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
                       2, &mySecondComm);
}
else if (membershipKey == 1)
{         /* Group 1 communicates with groups 0 and 2. */
  MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
                       1, &myFirstComm);
  MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
                       12, &mySecondComm);
}
else if (membershipKey == 2)
{        /* Group 2 communicates with groups 0 and 1. */
  MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
                       2, &myFirstComm);
  MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
                       12, &mySecondComm);
}

/* Do some work ... */
```

```
    /* Then free communicators before terminating... */
    MPI_Comm_free(&myFirstComm);
    MPI_Comm_free(&mySecondComm);
    MPI_Comm_free(&myComm);
    MPI_Finalize();
    return 0;
}
```

6.7 Caching

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects, communicators, windows, and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window, or datatype,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator MPI_COMM_SELF is a suitable choice for posting process-local attributes, via this attribute-caching mechanism. (*End of advice to users.*)

Rationale. In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (*End of rationale.*)

One difficulty is the potential for size differences between Fortran integers and C pointers. For this reason, the Fortran versions of these routines use integers of kind MPI_ADDRESS_KIND.

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

6.7.1 Functionality

Attributes can be attached to communicators, windows, and datatypes. Attributes are local to the process and specific to the communicator to which they are attached. Attributes are not propagated by MPI from one communicator to another except when the communicator is duplicated using MPI_COMM_DUP or MPI_COMM_IDUP (and even then the application must give specific permission through callback functions for the attribute to be copied).

> *Advice to users.* Attributes in C are of type void *. Typically, such an attribute will be a pointer to a structure that contains further information, or a handle to an MPI object. In Fortran, attributes are of type INTEGER. Such attribute can be a handle to an MPI object, or just an integer-valued attribute. (*End of advice to users.*)

> *Advice to implementors.* Attributes are scalar values, equal in size to, or larger than a C-language pointer. Attributes can always hold an MPI handle. (*End of advice to implementors.*)

The caching interface defined here requires that attributes be stored by MPI opaquely within a communicator, window, and datatype. Accessor functions include the following:

- obtain a key value (used to identify an attribute); the user specifies "callback" functions by which MPI informs the application when the communicator is destroyed or copied.
- store and retrieve the value of an attribute;

> *Advice to implementors.* Caching and callback functions are only called synchronously, in response to explicit application requests. This avoids problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)
>
> The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators.
>
> A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid access to attributes through the use of pointers in communicators (to find the attribute table) and cleverly chosen key values (to retrieve individual attributes). In light of the efficiency "hit" inherent in the minimal interface, the more complete interface defined here is seen to be superior. (*End of advice to implementors.*)

MPI provides the following services related to caching. They are all process local.

6.7.2 Communicators

Functions for caching on communicators are:

MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, extra_state)

IN	comm_copy_attr_fn	copy callback function for comm_keyval (function)
IN	comm_delete_attr_fn	delete callback function for comm_keyval (function)
OUT	comm_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
             MPI_Comm_delete_attr_function *comm_delete_attr_fn,
             int *comm_keyval, void *extra_state)
```

```
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
             extra_state, ierror)
    PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn
    PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn
    INTEGER, INTENT(OUT) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

Generates a new attribute key. Keys are locally unique in a process, and opaque to user, though they are explicitly stored in integers. Once allocated, the key value can be used to associate attributes and access them on any locally defined communicator.
The C callback functions are:

```
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
             void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
             void *attribute_val, void *extra_state);
```

which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.
With the mpi_f08 module, the Fortran callback functions are:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
      TYPE(MPI_Comm) :: oldcomm
      INTEGER :: comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL :: flag
```

and

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
  attribute_val, extra_state, ierror)
      TYPE(MPI_Comm) ::  comm
      INTEGER ::  comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
```

With the `mpi` module and `mpif.h`, the Fortran callback functions are:

```
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
              ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

```
SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The comm_copy_attr_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP or MPI_COMM_IDUP. comm_copy_attr_fn should be of type MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns flag = 0 or .FALSE., then the attribute is deleted in the duplicated communicator. Otherwise (flag = 1 or .TRUE.), the new attribute value is set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_DUP or MPI_COMM_IDUP will fail).

The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a function that does nothing other than returning flag = 0 or .FALSE. (depending on whether the keyval was created with a C or Fortran binding to MPI_COMM_CREATE_KEYVAL) and MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1 or .TRUE., returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. These replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN, whose use is deprecated.

Advice to users. Even though both formal arguments attribute_val_in and attribute_val_out are of type void *, their usage differs. The C copy function is passed by MPI in attribute_val_in the *value* of the attribute, and in attribute_val_out the *address* of the attribute, so as to allow the function to return the (new) attribute value. The use of type void * for both is to avoid messy type casts.

A valid copy function is one that completely duplicates the information by making a full duplicate copy of the data structures implied by an attribute; another might just make another reference to that data structure, while using a reference-count mechanism. Other types of attributes might not copy at all (they might be specific to oldcomm only). (*End of advice to users.*)

Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (*End of advice to implementors.*)

Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows. The comm_delete_attr_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_COMM_DELETE_ATTR. comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.

This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_FREE will fail).

The argument comm_delete_attr_fn may be specified as MPI_COMM_NULL_DELETE_FN from either C or Fortran. MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE), is erroneous.

The special key value MPI_KEYVAL_INVALID is never returned by MPI_COMM_CREATE_KEYVAL. Therefore, it can be used for static initialization of key values.

Advice to implementors. The predefined Fortran functions MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and MPI_COMM_NULL_DELETE_FN are defined in the `mpi` module (and `mpif.h`) and the `mpi_f08` module with the same name, but with different interfaces. Each function can coexist twice with the same name in the same MPI library, one routine as an implicit interface outside of the `mpi` module, i.e., declared as `EXTERNAL`, and the other routine within `mpi_f08` declared with `CONTAINS`. These routines have different link names, which are also different to the link names used for the routines used in C. (*End of advice to implementors.*)

Advice to users. Callbacks, including the predefined Fortran functions MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and MPI_COMM_NULL_DELETE_FN should not be passed from one application routine that uses the `mpi_f08` module to another application routine that uses the `mpi` module or `mpif.h`, and vice versa; see also the advice to users on page **??**. (*End of advice to users.*)

MPI_COMM_FREE_KEYVAL(comm_keyval)

INOUT	comm_keyval	key value (integer)

```
int MPI_Comm_free_keyval(int *comm_keyval)
```

```
MPI_Comm_free_keyval(comm_keyval, ierror)
```

```
    INTEGER, INTENT(INOUT) :: comm_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
    INTEGER COMM_KEYVAL, IERROR
```

Frees an extant attribute key. This function sets the value of keyval to MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use, because the actual free does not transpire until after all references (in other communicators on the process) to the key have been freed. These references need to be explictly freed by the program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance, or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed communicator.

MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)

INOUT	comm	communicator from which attribute will be attached (handle)
IN	comm_keyval	key value (integer)
IN	attribute_val	attribute value

```
int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)

MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

This function stores the stipulated attribute value attribute_val for subsequent retrieval by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback function comm_delete_attr_fn was executed), and a new value was next stored. The call is erroneous if there is no key with value keyval; in particular MPI_KEYVAL_INVALID is an erroneous key value. The call will fail if the comm_delete_attr_fn function returned an error code other than MPI_SUCCESS.

MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)

IN	comm	communicator to which the attribute is attached (handle)
IN	comm_keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key (logical)

```
int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
              int *flag)
```

```
MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

Retrieves attribute value by key. The call is erroneous if there is no key with value keyval. On the other hand, the call is correct if the key value exists, but no attribute is attached on comm for that key; in such case, the call returns flag = false. In particular MPI_KEYVAL_INVALID is an erroneous key value.

Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the *value* of the attribute; the call to MPI_Comm_get_attr passes in attribute_val the *address* of the location where the attribute value is to be returned. Thus, if the attribute value itself is a pointer of type void*, then the actual attribute_val parameter to MPI_Comm_set_attr will be of type void* and the actual attribute_val parameter to MPI_Comm_get_attr will be of type void**. (*End of advice to users.*)

Rationale. The use of a formal parameter attribute_val of type void* (rather than void**) avoids the messy type casting that would be needed if the attribute value is declared with a type other than void*. (*End of rationale.*)

MPI_COMM_DELETE_ATTR(comm, comm_keyval)

INOUT	comm	communicator from which the attribute is deleted (handle)
IN	comm_keyval	key value (integer)

```
int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
```

```
MPI_Comm_delete_attr(comm, comm_keyval, ierror)
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
```

Delete attribute from cache by key. This function invokes the attribute delete function comm_delete_attr_fn specified when the keyval was created. The call will fail if the comm_delete_attr_fn function returns an error code other than MPI_SUCCESS.

Whenever a communicator is replicated using the function MPI_COMM_DUP or MPI_COMM_IDUP, all call-back copy functions for attributes that are currently set are invoked (in arbitrary order). Whenever a communicator is deleted using the function MPI_COMM_FREE all callback delete functions for attributes that are currently set are invoked.

6.7.3 Windows

The functions for caching on windows are:

MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)

IN	win_copy_attr_fn	copy callback function for win_keyval (function)
IN	win_delete_attr_fn	delete callback function for win_keyval (function)
OUT	win_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
             MPI_Win_delete_attr_function *win_delete_attr_fn,
             int *win_keyval, void *extra_state)
```

```
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
             extra_state, ierror)
    PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
    PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
    INTEGER, INTENT(OUT) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
    INTEGER WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or MPI_WIN_DUP_FN from either C or Fortran. MPI_WIN_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_WIN_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.

The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN from either C or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

```
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
             void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
             void *attribute_val, void *extra_state);
```

With the mpi_f08 module, the Fortran callback functions are:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
      TYPE(MPI_Win) ::  oldwin
      INTEGER ::  win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL ::  flag
```

and

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
  extra_state, ierror)
      TYPE(MPI_Win) ::  win
      INTEGER ::  win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
```

With the mpi module and mpif.h, the Fortran callback functions are:

```
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
              ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

```
SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is erroneous.

MPI_WIN_FREE_KEYVAL(win_keyval)

INOUT	win_keyval	key value (integer)

```
int MPI_Win_free_keyval(int *win_keyval)
```

```
MPI_Win_free_keyval(win_keyval, ierror)
    INTEGER, INTENT(INOUT) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
    INTEGER WIN_KEYVAL, IERROR
```

MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)

INOUT	win	window to which attribute will be attached (handle)
IN	win_keyval	key value (integer)
IN	attribute_val	attribute value

```
int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
```

```
MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)

IN	win	window to which the attribute is attached (handle)
IN	win_keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key (logical)

```
int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
             int *flag)
```

```
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

MPI_WIN_DELETE_ATTR(win, win_keyval)

INOUT	win	window from which the attribute is deleted (handle)
IN	win_keyval	key value (integer)

```
int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
```

```
MPI_Win_delete_attr(win, win_keyval, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
```

6.7.4 Datatypes

The new functions for caching on datatypes are:

MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval, extra_state)

IN	type_copy_attr_fn	copy callback function for type_keyval (function)
IN	type_delete_attr_fn	delete callback function for type_keyval (function)
OUT	type_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
             MPI_Type_delete_attr_function *type_delete_attr_fn,
             int *type_keyval, void *extra_state)
```

```
MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
             extra_state, ierror)
    PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
    PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
    INTEGER, INTENT(OUT) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
```

```
    INTEGER TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or MPI_TYPE_DUP_FN from either C or Fortran. MPI_TYPE_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.

The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN from either C or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

```
typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
             int type_keyval, void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
             int type_keyval, void *attribute_val, void *extra_state);
```

With the mpi_f08 module, the Fortran callback functions are:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
      TYPE(MPI_Datatype) ::  oldtype
      INTEGER ::  type_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL ::  flag
```

and

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
  attribute_val, extra_state, ierror)
      TYPE(MPI_Datatype) ::  datatype
      INTEGER ::  type_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
```

With the mpi module and mpif.h, the Fortran callback functions are:

```
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

```
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE), is erroneous.

MPI_TYPE_FREE_KEYVAL(type_keyval)

INOUT	type_keyval	key value (integer)

```
int MPI_Type_free_keyval(int *type_keyval)

MPI_Type_free_keyval(type_keyval, ierror)
    INTEGER, INTENT(INOUT) :: type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
    INTEGER TYPE_KEYVAL, IERROR
```

MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)

INOUT	datatype	datatype to which attribute will be attached (handle)
IN	type_keyval	key value (integer)
IN	attribute_val	attribute value

```
int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
             void *attribute_val)

MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)

IN	datatype	datatype to which the attribute is attached (handle)
IN	type_keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key (logical)

```
int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
             void *attribute_val, int *flag)
```

```
MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

MPI_TYPE_DELETE_ATTR(datatype, type_keyval)

INOUT	datatype	datatype from which the attribute is deleted (handle)
IN	type_keyval	key value (integer)

```
int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)

MPI_Type_delete_attr(datatype, type_keyval, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
```

6.7.5 Error Class for Invalid Keyval

Key values for attributes are system-allocated, by MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL. Only such values can be passed to the functions that use key values as input arguments. In order to signal that an erroneous key value has been passed to one of these functions, there is a new MPI error class: MPI_ERR_KEYVAL. It can be returned by MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE, MPI_KEYVAL_FREE, MPI_{TYPE,COMM,WIN}_DELETE_ATTR, MPI_{TYPE,COMM,WIN}_SET_ATTR, MPI_{TYPE,COMM,WIN}_GET_ATTR, MPI_{TYPE,COMM,WIN}_FREE_KEYVAL, MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DISCONNECT, and MPI_COMM_FREE. The last four are included because keyval is an argument to the copy and delete functions for attributes.

6.7.6 Attributes Example

Advice to users. This example shows how to write a collective communication operation that uses caching to be more efficient after the first call. (*End of advice to users.*)

```
   /* key for this module's stuff: */
```

```
static int gop_key = MPI_KEYVAL_INVALID;

typedef struct
{
   int ref_count;          /* reference count */
   /* other stuff, whatever else we want */
} gop_stuff_type;

void Efficient_Collective_Op(MPI_Comm comm, ...)
{
  gop_stuff_type *gop_stuff;
  MPI_Group       group;
  int             foundflag;

  MPI_Comm_group(comm, &group);

  if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
  {
    if ( ! MPI_Comm_create_keyval(gop_stuff_copier,
                             gop_stuff_destructor,
                             &gop_key, (void *)0)) {
    /* get the key while assigning its copy and delete callback
       behavior. */
    } else
        MPI_Abort(comm, 99);
  }

  MPI_Comm_get_attr(comm, gop_key, &gop_stuff, &foundflag);
  if (foundflag)
  { /* This module has executed in this group before.
       We will use the cached information */
  }
  else
  { /* This is a group that we have not yet cached anything in.
       We will now do so.
    */

    /* First, allocate storage for the stuff we want,
       and initialize the reference count */

    gop_stuff = (gop_stuff_type *) malloc(sizeof(gop_stuff_type));
    if (gop_stuff == NULL) { /* abort on out-of-memory error */ }

    gop_stuff->ref_count = 1;

    /* Second, fill in *gop_stuff with whatever we want.
       This part isn't shown here */

```

```
    /* Third, store gop_stuff as the attribute value */
    MPI_Comm_set_attr(comm, gop_key, gop_stuff);
  }
  /* Then, in any case, use contents of *gop_stuff
     to do the global op ... */
}

/* The following routine is called by MPI when a group is freed */

int gop_stuff_destructor(MPI_Comm comm, int keyval, void *gop_stuffP,
                         void *extra)
{
  gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
  if (keyval != gop_key) { /* abort -- programming error */ }

  /* The group's being freed removes one reference to gop_stuff */
  gop_stuff->ref_count -= 1;

  /* If no references remain, then free the storage */
  if (gop_stuff->ref_count == 0) {
    free((void *)gop_stuff);
  }
  return MPI_SUCCESS;
}

/* The following routine is called by MPI when a group is copied */
int gop_stuff_copier(MPI_Comm comm, int keyval, void *extra,
               void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
{
  gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
  gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
  if (keyval != gop_key) { /* abort -- programming error */ }

  /* The new group adds one reference to this gop_stuff */
  gop_stuff_in->ref_count += 1;
  *gop_stuff_out = gop_stuff_in;
  return MPI_SUCCESS;
}
```

6.8 Naming Objects

There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

MPI_COMM_SET_NAME (comm, comm_name)

INOUT	comm	communicator whose identifier is to be set (handle)
IN	comm_name	the character string which is remembered as the name (string)

```
int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)

MPI_Comm_set_name(comm, comm_name, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    CHARACTER(LEN=*), INTENT(IN) ::  comm_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
    INTEGER COMM, IERROR
    CHARACTER*(*) COMM_NAME
```

MPI_COMM_SET_NAME allows a user to associate a name string with a communicator. The character string which is passed to MPI_COMM_SET_NAME will be saved inside the MPI library (so it can be freed by the caller immediately after the call, or allocated on the stack). Leading spaces in name are significant but trailing ones are not.

MPI_COMM_SET_NAME is a local (non-collective) operation, which only affects the name of the communicator as seen in the process which made the MPI_COMM_SET_NAME call. There is no requirement that the same (or any) name be assigned to a communicator in every process where it exists.

Advice to users. Since MPI_COMM_SET_NAME is provided to help debug code, it is sensible to give the same name to a communicator in all of the processes where it exists, to avoid confusion. (*End of advice to users.*)

The length of the name which can be stored is limited to the value of MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C to allow for the null terminator. Attempts to put names longer than this will result in truncation of the name. MPI_MAX_OBJECT_NAME must have a value of at least 64.

Advice to users. Under circumstances of store exhaustion an attempt to put a name of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be viewed only as a strict upper bound on the name length, not a guarantee that setting names of less than this length will always succeed. (*End of advice to users.*)

Advice to implementors. Implementations which pre-allocate a fixed size space for a name should use the length of that allocation as the value of MPI_MAX_OBJECT_NAME. Implementations which allocate space for the name from the heap should still define MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate space for a string of up to this size when calling MPI_COMM_GET_NAME. (*End of advice to implementors.*)

MPI_COMM_GET_NAME (comm, comm_name, resultlen)

IN	comm	communicator whose name is to be returned (handle)
OUT	comm_name	the name previously stored on the communicator, or an empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)

```
int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
```

```
MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
    INTEGER COMM, RESULTLEN, IERROR
    CHARACTER*(*) COMM_NAME
```

MPI_COMM_GET_NAME returns the last name which has previously been associated with the given communicator. The name may be set and retrieved from any language. The same name will be returned independent of the language used. name should be allocated so that it can hold a resulting string of length MPI_MAX_OBJECT_NAME characters. MPI_COMM_GET_NAME returns a copy of the set name in name.

In C, a null character is additionally stored at name[resultlen]. The value of resultlen cannot be larger than MPI_MAX_OBJECT_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger than MPI_MAX_OBJECT_NAME.

If the user has not associated a name with a communicator, or an error occurs, MPI_COMM_GET_NAME will return an empty string (all spaces in Fortran, "" in C). The three predefined communicators will have predefined names associated with them. Thus, the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system may have chosen to give a default name to a communicator does not prevent the user from setting a name on the same communicator; doing this removes the old name and assigns the new one.

Rationale. We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated from the heap.
- To make the attribute key useful additional code to call `strdup` is necessary. If this is not standardized then users have to write it. This is extra unneeded work which we can easily eliminate.

- The Fortran binding is not trivial to write (it will depend on details of the Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.

(*End of rationale.*)

Advice to users. The above definition means that it is safe simply to print the string returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was no name.

Note that associating a name with a communicator has no effect on the semantics of an MPI program, and will (necessarily) increase the store requirement of the program, since the names must be saved. Therefore there is no requirement that users use these functions to associate names with communicators. However debugging and profiling MPI applications may be made easier if names are associated with communicators, since the debugger or profiler should then be able to present information in a less cryptic manner. (*End of advice to users.*)

The following functions are used for setting and getting names of datatypes. The constant MPI_MAX_OBJECT_NAME also applies to these names.

MPI_TYPE_SET_NAME (datatype, type_name)

INOUT	datatype	datatype whose identifier is to be set (handle)
IN	type_name	the character string which is remembered as the name (string)

```
int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)

MPI_Type_set_name(datatype, type_name, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    CHARACTER(LEN=*), INTENT(IN) ::  type_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
    INTEGER DATATYPE, IERROR
    CHARACTER*(*) TYPE_NAME
```

MPI_TYPE_GET_NAME (datatype, type_name, resultlen)

IN	datatype	datatype whose name is to be returned (handle)
OUT	type_name	the name previously stored on the datatype, or a empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)

```
int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,
             int *resultlen)

MPI_Type_get_name(datatype, type_name, resultlen, ierror)
```

```
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) ::  type_name
    INTEGER, INTENT(OUT) ::  resultlen
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER DATATYPE, RESULTLEN, IERROR
    CHARACTER*(*) TYPE_NAME
```

Named predefined datatypes have the default names of the datatype name. For example, MPI_WCHAR has the default name of MPI_WCHAR.

The following functions are used for setting and getting names of windows. The constant MPI_MAX_OBJECT_NAME also applies to these names.

MPI_WIN_SET_NAME (win, win_name)

INOUT	win	window whose identifier is to be set (handle)
IN	win_name	the character string which is remembered as the name (string)

```
int MPI_Win_set_name(MPI_Win win, const char *win_name)
```

```
MPI_Win_set_name(win, win_name, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    CHARACTER(LEN=*), INTENT(IN) ::  win_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
    INTEGER WIN, IERROR
    CHARACTER*(*) WIN_NAME
```

MPI_WIN_GET_NAME (win, win_name, resultlen)

IN	win	window whose name is to be returned (handle)
OUT	win_name	the name previously stored on the window, or a empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)

```
int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
```

```
MPI_Win_get_name(win, win_name, resultlen, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) ::  win_name
    INTEGER, INTENT(OUT) ::  resultlen
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
    INTEGER WIN, RESULTLEN, IERROR
```

```
CHARACTER*(*) WIN_NAME
```

6.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

6.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that communicator must be free of side effects throughout execution of the subprogram: there should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work "safely." For libraries so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communications will interfere. Since we permit good implementations to create new communicators without synchronization (such as by preallocated contexts on communicators), this does not impose a significant overhead.

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

6.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by having each executing process invoke the procedure. The invocation is a collective operation: it is executed by all processes in the execution group, and invocations are similarly ordered at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

Static Communicator Allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

Dynamic Communicator Allocation

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in a subset of a group executing the same parallel procedure. Thus, processes that execute the same parallel procedure have the same execution stack.

In such a case, a new communicator needs to be dynamically allocated for each new invocation of a parallel procedure. The allocation is done by the caller. A new communicator can be generated by a call to MPI_COMM_DUP, if the callee execution group is identical to the caller execution group, or by a call to MPI_COMM_SPLIT if the caller execution group is split into several subgroups executing distinct parallel routines. The new communicator is passed as an argument to the invoked routine.

The need for generating a new communicator at each invocation can be alleviated or avoided altogether in some cases: If the execution group is not split, then one can allocate a stack of communicators in a preamble, and next manage the stack in a way that mimics the stack of recursive calls.

One can also take advantage of the well-ordering property of communication to avoid confusing caller and callee communication, even if both use the same communicator. To do so, one needs to abide by the following two rules:

- messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;
- messages are always selected by source (no use is made of MPI_ANY_SOURCE).

The General Case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, communicator creation is properly coordinated.

Bibliography

[1] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994. 6.1

[2] Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors. *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*. Springer, 2011. 3, 5

[3] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective communicator creation in MPI. In Cotronis et al. [2], pages 282–291. 6.4.2

[4] D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991. 6.1.2

[5] Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI — common practice, issues, and extensions. In Cotronis et al. [2], pages 345–355. 6.4.2

[6] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990. 6.1.2

[7] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993. 6.1

[8] 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. 6.1.2, 6.5.6

Index