MPI_COMM_DUP Duplicates the existing communicator comm with associated key values. 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. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 492 for further discussion about the C++ bindings for Dup() and Clone().

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

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

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

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

ticket150.

{MPI::Intercomm MPI::Intercomm::Create(const MPI::Group& group) const *(binding deprecated, see Section 15.2)* }

ticket150.
ticket150.

{MPI::Intracomm MPI::Intracomm::Create(const MPI::Group& group) const *(binding deprecated, see Section 15.2)* }

ticket150.
ticket66.

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

ticket66.

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 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 sometimes to avoid communication related to context creation.
>
> (*End of rationale.*)

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

> *Advice to implementors.* [Since all processes calling MPI_COMM_DUP or ticket66. MPI_COMM_CREATE provide the same group argument, it is theoretically possible to agree on a group-wide unique context with no communication.] When calling ticket66. 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 should be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (*End of advice to implementors.*)

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. ticket66. 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.*)

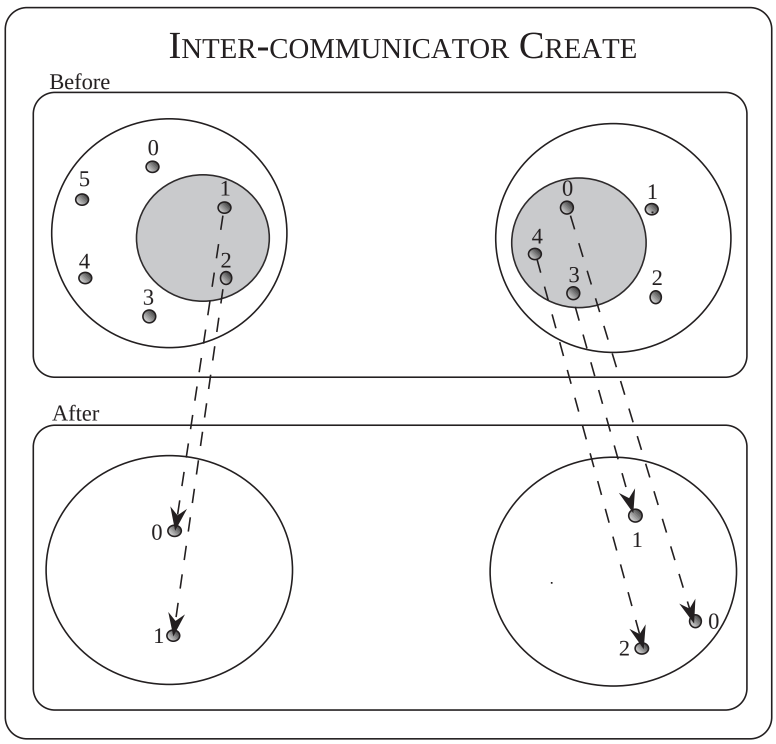

Figure 6.1: Intercommunicator create using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

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_SPLIT(comm, color, key, newcomm)

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

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

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

ticket150.

{`MPI::Intercomm MPI::Intercomm::Split(int color, int key) const` *(binding deprecated, see Section 15.2)* } ticket150.

ticket150.

{`MPI::Intracomm MPI::Intracomm::Split(int color, int key) const` *(binding deprecated, see Section 15.2)* } ticket150.

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. ticket66.

[A call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where all members of group provide color $= 0$ and key $=$ rank in group, and all processes that are not members of group provide color $=$ MPI_UNDEFINED. The function MPI_COMM_SPLIT allows more general partitioning of a group into one or more subgroups with optional reordering.] 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 ticket66.

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.

ticket74. The value of color must be non-negative.

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. ticket66. 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.

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 doesn't really care about the rank-order of the processes in the new communicator. (*End of advice to users.*)

ticket74. *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.

ticket66. *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.*)

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.

6. Section 3.7 on page 50.
   The Advice to users for IBSEND and IRSEND was slightly changed. ticket143.

7. Section 3.7.3 on page 55.
   The advice to free an active request was removed in the Advice to users for MPI_REQUEST_FREE. ticket137.

8. Section 3.7.6 on page 67.
   MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input. ticket31.

9. Section 5.8 on page 161.
   "In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and MPI_ALLTOALLW for intracommunicators. ticket64.

10. Section 5.9.2 on page 169.
    Predefined parameterized datatypes (e.g., returned by MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g. MPI_REAL8) have been added to the list of valid datatypes in reduction operations. ticket18.

11. Section 5.9.2 on page 169.
    MPI_(U)INT{8,16,32,64}_T are all considered C integer types for the purposes of the predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran integer types. MPI_C_BOOL is considered a Logical type. MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types. ticket24.

12. Section 5.9.7 on page 180.
    The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been added. ticket27.

13. Section 5.10.1 on page 182.
    The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI standard. ticket94.

14. Section 5.11.2 on page 185.
    Added in place argument to MPI_EXSCAN. ticket19.

15. Section 6.4.2 on page 204, and Section 6.6 on page 224.
    Implementations that did not implement MPI_COMM_CREATE on intercommunicators will need to add that functionality. As the standard described the behavior of this operation on intercommunicators, it is believed that most implementations already provide this functionality. Note also that the C++ binding for both MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms. ticket66.

16. Section 6.4.2 on page 204.
    MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm is an intracommunicator. If comm is an intercommunicator it was clarified that all processes in the same local group of comm must specify the same value for group. ticket33.

17. Section 7.5.4 on page 268.
    New functions for a scalable distributed graph topology interface has been added. In this section, the functions MPI_DIST_GRAPH_CREATE_ADJACENT and