

associated with this key in the new communicator; one particular action that a copy call-back 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 473 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	<code>comm</code>	communicator (handle)
IN	<code>group</code>	Group, which is a subset of the group of <code>comm</code> (handle)
OUT	<code>newcomm</code>	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

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

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

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. 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 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`. 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.

ticket66.
ticket66.

ticket66.

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.*)

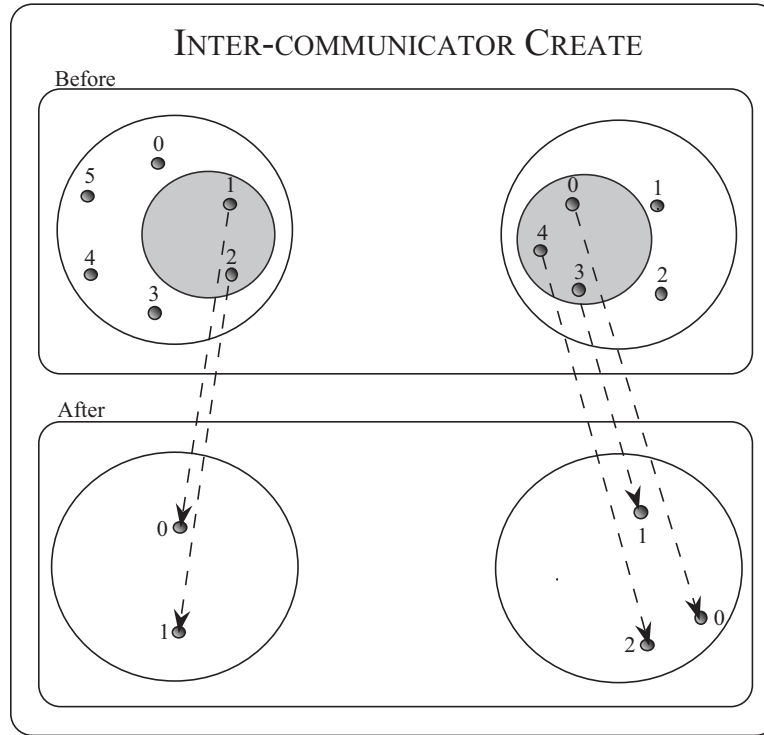


Figure 6.1: Intercommunicator create using `MPI_COMM_CREATE` extended to intercommunicators. The input groups are those in the grey circle.

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 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)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
{MPI::Intercomm MPI::Intercomm::Split(int color, int key) const (binding
    deprecated, see Section 15.2) }
{MPI::Intracomm MPI::Intracomm::Split(int color, int key) const (binding
    deprecated, see Section 15.2) }

```

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*.

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.

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 conflict 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.

```

/* 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 );

```