5.6.3 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.

```c
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);
    }
}
```

![Fig. 5.1 Three-group pipeline.](image)
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();
}

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.

main(int argc, char **argv)
{
    MPI_Comm  myComm;     /* intra-communicator of local sub-group */
    MPI_Comm  myFirstComm; /* inter-communicators */
    MPI_Comm  mySecondComm;
    MPI_Status status;
    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();
Example 3: Building Name Service for Inter-communication

The following procedures exemplify the process by which a user could create name service for building inter-communicators via a rendezvous involving a server communicator, and a tag name selected by both groups.

After all MPI processes execute MPI_INIT, every process calls the example function, Init_server(), defined below. Then, if the new_world returned is NULL, the process getting NULL is required to implement a server function, in a reactive loop, Do_server(). Everyone else just does their prescribed computation, using new_world as the new effective “global” communicator. One designated process calls Undo_Server() to get rid of the server when it is not needed any longer.

Features of this approach include:

- Support for multiple name servers
- Ability to scope the name servers to specific processes
- Ability to make such servers come and go as desired.

#define INIT_SERVER_TAG_1 666
#define UNDO_SERVER_TAG_1 777

static int server_key_val;

/* for attribute management for server_comm, copy callback: */
void handle_copy_fn(MPI_Comm *oldcomm, int *keyval, void *extra_state, void *attribute_val_in, void **attribute_val_out, int *flag)
{
    /* copy the handle */
    *attribute_val_out = attribute_val_in;
    *flag = 1; /* indicate that copy to happen */
}

int Init_server(peer_comm, rank_of_server, server_comm, new_world)
MPI_Comm peer_comm;
int rank_of_server;
MPI_Comm *server_comm;
MPI_Comm *new_world; /* new effective world, sans server */
{
    MPI_Comm tempcomm, lone_comm;
    MPI_Group peer_group, temp_group;
    int rank_in_peer_comm, size, color, key = 0;
    int peer_leader, peer_leader_rank_in_temp_comm;

    MPI_Comm_rank(peer_comm, &rank_in_peer_comm);
    MPI_Comm_size(peer_comm, &size);
if ((size < 2) || (0 > rank_of_server) || (rank_of_server >= size))
    return (MPI_ERR_OTHER);

/* create two communicators, by splitting peer_comm
   into the server process, and everyone else */
peer_leader = (rank_of_server + 1) % size; /* arbitrary choice */

if (((color = (rank_in_peer_comm == rank_of_server))
{
    MPI_Comm_split(peer_comm, color, key, &lone_comm);

    MPI_Intercomm_create(lone_comm, 0, peer_comm, peer_leader,
                         INIT_SERVER_TAG_1, server_comm);

    MPI_Comm_free(&lone_comm);
    *new_world = (MPI_Comm) 0;
}
else
{
    MPI_Comm_split(peer_comm, color, key, &temp_comm);

    MPI_Comm_group(peer_comm, &peer_group);
    MPI_Comm_group(temp_comm, &temp_group);
    MPI_Group_translate_ranks(peer_group, 1, &peer_leader,
                              temp_group, &peer_leader_rank_in_temp_comm);

    MPI_Intercomm_create(temp_comm, peer_leader_rank_in_temp_comm,
                          peer_comm, rank_of_server,
                          INIT_SERVER_TAG_1, server_comm);

    // attach new_world communication attribute to server_comm: */

    /* CRITICAL SECTION FOR MULTITHREADING */
    if(server_keyval == MPI_KEYVAL_INVALID)
    {
        /* acquire the process-local name for the server keyval */
        MPI_Attr_keyval_create(handle_copy_fn, NULL,
                                &server_keyval, NULL);
    }

    *new_world = temp_comm;

    /* Cache handle of intra-communicator on inter-communicator: */
MPI_Attr_put(server_comm, server_keyval, (void *)(*new_world));
}

return (MPI_SUCCESS);
}

The actual server process would commit to running the following code:

```
int Do_server(server_comm)
MPI_Comm server_comm;
{
  void init_queue();
  int en_queue(), de_queue(); /* keep triplets of integers
     for later matching (fns not shown) */

  MPI_Comm comm;
  MPI_Status status;
  int client_tag, client_source;
  int client_rank_in_new_world, pairs_rank_in_new_world;
  int buffer[10], count = 1;

  void *queue;
  init_queue(&queue);

  for (;;) {
    MPI_Recv(buffer, count, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
              server_comm, &status); /* accept from any client */

    /* determine client: */
    client_tag = status.MPI_TAG;
    client_source = status.MPI_SOURCE;
    client_rank_in_new_world = buffer[0];

    if (client_tag == UNDO_SERVER_TAG)     /* client that
        terminates server */
    {
      while (de_queue(queue, MPI_ANY_TAG, &pairs_rank_in_new_world,
                      &pairs_rank_in_server));

      MPI_Intercomm_free(&server_comm);
      break;
    }
```
if (de_queue(queue, client_tag, &pairs_rank_in_new_world,
             &pairs_rank_in_server))
{
    /* matched pair with same tag, tell them
       about each other! */
    buffer[0] = pairs_rank_in_new_world;
    MPI_Send(buffer, 1, MPI_INT, client_src, client_tag,
              server_comm);

    buffer[0] = client_rank_in_new_world;
    MPI_Send(buffer, 1, MPI_INT, pairs_rank_in_server, client_tag,
              server_comm);
}
else
    en_queue(queue, client_tag, client_source,
              client_rank_in_new_world);

} }

A particular process would be responsible for ending the server when it is
no longer needed. Its call to Undo_server would terminate server function.

int Undo_server(server_comm)  /* example client that ends server */
MPI_Comm *server_comm;
{
    int buffer = 0;
    MPI_Send(&buffer, 1, MPI_INT, 0, UNDO_SERVER_TAG_1, *server_comm);
    MPI_Intercomm_free(server_comm);
}

The following is a blocking name-service for inter-communication, with the
same semantic restrictions as MPI_Intercomm_create, but simplified syntax. It
uses the functionality just defined to create the name-service.

int Intercomm_name_create(local_comm, server_comm, tag, comm)
MPI_Comm local_comm, server_comm;
int tag;
MPI_Comm *comm;
{
    int error;
    int found;  /* attribute acquisition mgnt for new_world */
    /* comm in server_comm */
    void *val;
MPI_Comm new_world;

int buffer[10], rank;
int local_leader = 0;

MPI_Attr_get(server_comm, server_keyval, &val, &found);
new_world = (MPI_Comm)val;  /* retrieve cached handle */

MPI_Comm_rank(server_comm, &rank);  /* rank in local group */

if (rank == local_leader)
{
    buffer[0] = rank;
    MPI_Send(&buffer, 1, MPI_INT, 0, tag, server_comm);
    MPI_Recv(&buffer, 1, MPI_INT, 0, tag, server_comm);
}

error = MPI_Intercomm_create(local_leader, local_comm, buffer[0],
                              new_world, tag, comm);

return(error);


5.7 Caching

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called attributes, to communicators. 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,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the communicator 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 attributing-caching mechanism.
(End of advice to users.)
5.7.1 FUNCTIONALITY

Attributes are attached to communicators. 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 (and even then the application must give specific permission through callback functions for the attribute to be copied).

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 represents that attributes be stored by MPI opaquely within a communicator. 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.

**MPI.KEYVAL.CREATE(copy_fn, delete_fn, keyval, extra_state)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>copy_fn</td>
</tr>
<tr>
<td>IN</td>
<td>delete_fn</td>
</tr>
<tr>
<td>OUT</td>
<td>keyval</td>
</tr>
<tr>
<td>IN</td>
<td>extra_state</td>
</tr>
</tbody>
</table>
int MPI_keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function *delete_fn, int *keyval, void* extra_state)

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
EXTERNAL COPY_FN, DELETE_FN
INTEGER KEYVAL, EXTRA_STATE, IERROR

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 copy_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP. copy_fn should be of type MPI_Copy_function, which is defined as follows:

typedef int MPI_Copy_function(MPI_Comm *oldcomm, int *keyval,
void *extra_state, void *attribute_val_in,
void **attribute_val_out, int *flag)

A Fortran declaration for such a function is as follows:
FUNCTION COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG)
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
LOGICAL FLAG

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, then the attribute is deleted in the duplicated communicator. Otherwise (flag = 1), the new attribute value is set via attribute_val_out. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_DUP will fail).

copy_fn may be specified as MPI_NULL_FN from either C or FORTRAN, in which case no copy callback occurs for keyval; MPI_NULL_FN is a function that does nothing other than returning flag = 0. In C, the NULL function pointer has the same behavior as using MPI_NULL_FN. As a further convenience, MPI_DUP_FN is a simple-minded copy callback available from C and FORTRAN; it sets flag = 1, and returns the value of attribute_val_in in attribute_val_out.

Note that the C version of this MPI_COMM_DUP assumes that the callback functions follow the C prototype, while the corresponding FORTRAN version assumes the FORTRAN prototype.

Advice to users. 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.)
Analogous to `copy_fn` is a callback deletion function, defined as follows. The `delete_fn` function is invoked when a communicator is deleted by `MPI_COMM_FREE` or when a call is made explicitly to `MPI_ATTR_DELETE`. `delete_fn` should be of type `MPI_Delete_function`, which is defined as follows:

```c
typedef int MPI_Delete_function(MPI_Comm *comm, int *keyval,
                               void *attribute_val, void *extra_state);
```

A Fortran declaration for such a function is as follows:

```fortran
FUNCTION DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE)
   INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE

   ...  
END FUNCTION DELETE_FUNCTION
```

This function is called by `MPI_COMM_FREE` and `MPI_ATTR_DELETE` to do whatever is needed to remove an attribute. It may be specified as the null function pointer in C or as `MPI_NULL_FN` from either C or FORTRAN, in which case no delete callback occurs for `keyval`.

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

### `MPI_KEYVAL_FREE(keyval)`

```c
INOUT   keyval
```

Frees the integer key value (integer)

```c
int MPI_Keyval_free(int *keyval)
```

### `MPI_KEYVAL_FREE(ATTR, IERROR)`

```c
INTEGER 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 explicitly freed by the program, either via calls to `MPI_ATTR_DELETE` that free one attribute instance, or by calls to `MPI_COMM_FREE` that free all attribute instances associated with the freed communicator.

**Advice to implementors.** The function `MPI_NULL_FN` need not be aliased to `(void (*)*)0` in C, though this is fine. It could be a legitimately callable function that profiles and so on. For FORTRAN, it is most convenient to have `MPI_NULL_FN` be a legitimate do-nothing function call. *End of advice to implementors.*

### `MPI_ATTR_PUT(comm, keyval, attribute_val)`

```c
IN   comm
     communicator to which attribute will be attached (handle)

IN   keyval
     key value, as returned by `MPI_KEYVAL_CREATE` (integer)

IN   attribute_val
     attribute value
```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)

MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

This function stores the stipulated attribute value attribute_val for subsequent retrieval by MPI_ATTR_GET. If the value is already present, then the outcome is as if MPI_ATTR_DELETE was first called to delete the previous value (and the callback function delete_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.

MPI_ATTR_GET(comm, keyval, attribute_val, flag)

IN      comm        communicator to which attribute is attached (handle)
IN      keyval      key value (integer)
OUT     attribute_val attribute value, unless flag = false
OUT     flag        true if an attribute value was extracted; false if no attribute is associated with the key

int MPI_Attr_get(MPI_Comm comm, int keyval, void **attribute_val, int *flag)

MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

    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.

MPI_ATTR_DELETE(comm, keyval)

IN      comm        communicator to which attribute is attached (handle)
IN      keyval      The key value of the deleted attribute (integer)

int MPI_Attr_delete(MPI_Comm comm, int keyval)

MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR

Delete attribute from cache by key. This function invokes the attribute delete function delete_fn specified when the keyval was created.

Whenever a communicator is replicated using the function MPI_COMM_DUP, 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.

5.7.2 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. The
coding style assumes that MPI function results return only error statuses.
(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;

Efficient_Collective_Op (comm, ...)
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_Attr_keyval_create( gop_stuff_copyer,
                                     gop_stuff_destructor,
                                     &gop_key, (void *)0));
        /* get the key while assigning its copy and delete callback
         behavior. */

        MPI_Abort("Insufficient keys available");
    }

    MPI_Attr_get (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_Attr_put ( 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 */

gop_stuff_destructor (comm, keyval, gop_stuff, extra)
MPI_Comm comm;
int keyval;
gop_stuff_type *gop_stuff;
void *extra;
{ 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);
    }
}

/* The following routine is called by MPI when a group is copied */
gop_stuff_copyer (comm, keyval, gop_stuff, extra)
MPI_Comm comm;
int keyval;
gop_stuff_type *gop_stuff;
void *extra;
{
  if (keyval != gop_key) /* abort -- programming error */

  /* The new group adds one reference to this gop_stuff */
gop_stuff -> ref_count += 1;
}

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

5.8.1 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.

5.8.2 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, then communicator creation be properly coordinated.
6.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in chapter 5, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are proposed in this chapter deal only with machine-independent mapping.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [20]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping
is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [10, 9].

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with tremendous benefits for program readability and notational power in message-passing programming. (*End of rationale.*)

### 6.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes stand for the processes, and the edges connect processes that communicate with each other. MPI provides message passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping. Edges in the communication graph are not weighted, so that processes are either simply connected or not connected at all.

*Rationale.* Experience with similar techniques in PARMACS [5, 8] show that this information is usually sufficient for a good mapping. Additionally, a more precise specification is more difficult for the user to set up, and it would make the interface functions substantially more complicated. (*End of rationale.*)

Specifying the virtual topology in terms of a graph is sufficient for all applications. However, in many applications the graph structure is regular, and the detailed set-up of the graph would be inconvenient for the user and might be less efficient at run time. A large fraction of all parallel applications use process topologies like rings, two- or higher-dimensional grids, or tori. These structures are completely defined by the number of dimensions and the numbers of processes in each coordinate direction. Also, the mapping of grids and tori is generally an easier problem than that of general graphs. Thus, it is desirable to address these cases explicitly.

Process coordinates in a cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a cartesian structure.
This means that, for example, the relation between group rank and coordinates for four processes in a $2 \times 2$ grid is as follows.

\[
\begin{align*}
\text{coord (0,0)} &: \text{ rank 0} \\
\text{coord (0,1)} &: \text{ rank 1} \\
\text{coord (1,0)} &: \text{ rank 2} \\
\text{coord (1,1)} &: \text{ rank 3}
\end{align*}
\]

### 6.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 5.

### 6.4 Overview of the Functions

The functions `MPI.GRAPH_CREATE` and `MPI.CART_CREATE` are used to create general (graph) virtual topologies and cartesian topologies, respectively. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator `comm.old`, which defines the set of processes on which the topology is to be mapped. A new communicator `comm.topol` is created that carries the topological structure as cached information (see Chapter 5). In analogy to function `MPI.COMM_CREATE`, no cached information propagates from `comm.old` to `comm.topol`.

`MPI.CART_CREATE` can be used to describe cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an $n$-dimensional hypercube is an $n$-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function `MPI.DIMS_CREATE` can be used to compute a balanced distribution of processes among a given number of dimensions.

*Rationale.* Similar functions are contained in EXPRESS [22] and PARMACS. (End of rationale.)

The function `MPI.TOPO_TEST` can be used to inquire about the topology associated with a communicator. The topological information can be extracted from the communicator using the functions `MPI.GRAHNDIMS_GET` and `MPI.GRAPH.GET`, for general graphs, and `MPI.CARTDIM_GET` and `MPI.CART.GET`, for cartesian topologies. Several additional functions are provided to manipulate cartesian topologies: the functions `MPI.CART.RANK` and `MPI.CART.COORDS` translate cartesian coordinates into a group rank, and vice versa; the function `MPI.CART.SUB` can be used to extract a cartesian subspace (analogous
to MPI.COMM.SPLIT). The function MPI.CARTSHIFT provides the information needed to communicate with neighbors in a cartesian dimension. The two functions MPI.GRAPH.NEIGHBORS.COUNT and MPI.GRAPH.NEIGHBORS can be used to extract the neighbors of a node in a graph. The function MPI.CART.SUB is collective over the input communicator's group; all other functions are local.

Two additional functions, MPI.GRAPH.MAP and MPI.CART.MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 5, they are sufficient to implement all other topology functions. Section 6.5.7 outlines such an implementation.

### 6.5 Topology Constructors

#### 6.5.1 Cartesian Constructor

MPI.CART.CREATE(comm.old, ndims, dims, periods, reorder, comm.cart)

<table>
<thead>
<tr>
<th>IN</th>
<th>comm.old</th>
<th>input communicator (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>ndims</td>
<td>number of dimensions of cartesian grid (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>dims</td>
<td>integer array of size ndims specifying the number of processes in each dimension</td>
</tr>
<tr>
<td>IN</td>
<td>periods</td>
<td>logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension</td>
</tr>
<tr>
<td>IN</td>
<td>reorder</td>
<td>ranking may be reordered (true) or not (false) (logical)</td>
</tr>
<tr>
<td>OUT</td>
<td>comm.cart</td>
<td>communicator with new cartesian topology (handle)</td>
</tr>
</tbody>
</table>

```
int MPI.Cart.create(MPI.Comm comm.old, int ndims, int *dims, int *periods,
                    int reorder, MPI.Comm *comm.cart)
```

```
MPI.CART.CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

MPI.CART.CREATE returns a handle to a new communicator to which the cartesian topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the cartesian grid is smaller than the size of the group of comm, then some processes are returned MPI.COMM.NULL, in analogy to MPI.COMM.SPLIT. The call is erroneous if it specifies a grid that is larger than the group size.
6.5.2 CARTESIAN CONVENIENCE FUNCTION: MPI.DIMS.CREATE

For cartesian topologies, the function MPI.DIMS.CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI.COMM.WORLD's group) into an $n$-dimensional topology.

MPI.DIMS.CREATE(nnodes, ndims, dims)

IN nnodes number of nodes in a grid (integer)
IN ndims number of cartesian dimensions (integer)
INOUT dims integer array of size ndims specifying the number of nodes in each dimension

int MPI.Dims.create(int nnodes, int ndims, int *dims)

MPI.DIMS.CREATE(NNODES, NDIMS, DIMS, IERROR)

INTEGER NNODES, NDIMS, DIMS(+), IERROR

The entries in the array dims are set to describe a cartesian grid with ndims dimensions and a total of nnodes nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array dims. If dims[i] is set to a positive number, the routine will not modify the number of nodes in dimension $i$; only those entries where dims[i] = 0 are modified by the call.

Negative input values of dims[i] are erroneous. An error will occur if nnodes is not a multiple of $\prod_{i \in [0, ndims)} \text{dims}[i]$.

For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array dims is suitable for use as input to routine MPI.CART_CREATE. MPI.DIMS.CREATE is local.

Example 6.1

<table>
<thead>
<tr>
<th>dims before call</th>
<th>function call</th>
<th>dims on return</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>MPI.DIMS.CREATE(6, 2, dims)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>MPI.DIMS.CREATE(7, 2, dims)</td>
<td>(7,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI.DIMS.CREATE(6, 3, dims)</td>
<td>(2,3,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI.DIMS.CREATE(7, 3, dims)</td>
<td>erroneous call</td>
</tr>
</tbody>
</table>

6.5.3 GENERAL (GRAPH) CONSTRUCTOR

MPI.GRAPHCREATE(comm.old, nnodes, index, edges, reorder, comm.graph)

IN comm_old input communicator without topology (handle)
IN nnodes number of nodes in graph (integer)
IN  index  array of integers describing node degrees (see below)
IN  edges  array of integers describing graph edges (see below)
IN  reorder  ranking may be reordered (true) or not (false) (logical)
OUT comm_graph  communicator with graph topology added (handle)

int MPI.Graph.create(MPI.Comm comm.old, int nnodes, int *index, int *edges,
                     int reorder, MPI.Comm *comm.graph)

MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH, IERROR)
   INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
   LOGICAL REORDER

MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes. If the size, nnodes, of the graph is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE and MPI_COMM_SPLIT. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

Example 6.2 Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

<table>
<thead>
<tr>
<th>process</th>
<th>neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0, 2</td>
</tr>
</tbody>
</table>
Then, the input arguments are:

- nnodes = 4
- index = 2, 3, 4, 6
- edges = 1, 3, 0, 3, 0, 2

Thus, in C, index[0] is the degree of node zero, and index[i] - index[i-1] is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges[j], for 0 ≤ j ≤ index[0] - 1 and the list of neighbors of node i, i > 0, is stored in edges[j], index[i - 1] ≤ j ≤ index[i] - 1.

In Fortran, index(1) is the degree of node zero, and index(i+1) - index(i) is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges(j), for 1 ≤ j ≤ index(1) and the list of neighbors of node i, i > 0, is stored in edges(j), index(i) + 1 ≤ j ≤ index(i + 1).

Advice to implementors. The following topology information is likely to be stored with a communicator:

- Type of topology (cartesian/graph),
- For a cartesian topology:
  1. adim (number of dimensions),
  2. dim (numbers of processes per coordinate direction),
  3. period (periodicity information),
  4. own_position (own position in grid, could also be computed from rank and dims)
- For a graph topology:
  1. index,
  2. edges,

which are the vectors defining the graph structure.

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array index simplifies access to the topology information. (End of advice to implementors.)

### 6.5.4 TOPOLOGY INQUIRY FUNCTIONS

If a topology has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls.

**MPI.TOPO.TEST(comm, status)**

- IN comm communicator (handle)
- OUT status topology type of communicator comm (choice)
int MPI_Topo_test(MPI_Comm comm, int *status)

MPI_Topo_test(comm, status, IERROR)
    INTEGER comm, status, IERROR

The function MPI_Topo_test returns the type of topology that is assigned to a communicator.
The output value status is one of the following:

MPI_GRAPH  graph topology
MPI_CART  cartesian topology
MPI_UNDEFINED  no topology

MPI_Graphdims_get(comm, nnodes, nedges)
IN    comm    communicator for group with graph structure (handle)
OUT    nnodes    number of nodes in graph (integer) (same as number of processes in the group)
OUT    nedges    number of edges in graph (integer)

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

MPI_Graphdims_get(comm, nnodes, nedges, IERROR)
    INTEGER comm, nnodes, nedges, IERROR

Functions MPI_Graphdims_get and MPI_Graph_get retrieve the graph-topology information that was associated with a communicator by MPI_Graph_create.
The information provided by MPI_Graphdims_get can be used to dimension the vectors index and edges correctly for the following call to MPI_Graph_get.

MPI_Graph_get(comm, maxindex, maxedges, index, edges)
IN    comm    communicator with graph structure (handle)
IN    maxindex    length of vector index in the calling program (integer)
IN    maxedges    length of vector edges in the calling program (integer)
OUT    index    array of integers containing the graph structure
    (for details see the definition of MPI_Graph_create)
OUT    edges    array of integers containing the graph structure

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
                   int *edges)
MPI.GRAPH.GET(comm, maxindex, maxedges, index, edges, ierror)
INTEGER comm, maxindex, maxedges, index( * ), edges( * ), ierror

MPI.CARTDIM.GET(comm, ndims)
IN comm communicator with cartesian structure (handle)
OUT ndims number of dimensions of the cartesian structure (integer)
int MPI.Cartdim.get(MPI_Comm comm, int *ndims)

MPI.CARTDIM.GET(comm, ndims, ierror)
INTEGER comm, ndims, ierror

The functions MPI.CARTDIM.GET and MPI.CART.GET return the cartesian topology information that was associated with a communicator by MPI.CART_CREATE.

MPI.CART.GET(comm, maxdims, dims, periods, coords)
IN comm communicator with cartesian structure (handle)
IN maxdims length of vectors dims, periods, and coords in the calling program (integer)
OUT dims number of processes for each cartesian dimension (array of integer)
OUT periods periodicity (true/false) for each cartesian dimension (array of logical)
OUT coords coordinates of calling process in cartesian structure (array of integer)
int MPI.Cart.get(MPI_Comm comm, int maxdims, int *dims, int *periods,
int *coords)

MPI.CART.GET(comm, maxdims, dims, periods, coords, ierror)
INTEGER comm, maxdims, dims( * ), coords( * ), ierror
LOGICAL periods( * )

MPI.CART.RANK(comm, coords, rank)
IN comm communicator with cartesian structure (handle)
IN coords integer array (of size ndims) specifying the cartesian coordinates of a process
OUT rank rank of specified process (integer)
int MPI.Cart.rank(MPI_Comm comm, int *coords, int *rank)

MPI.CART.RANK(comm, coords, rank, ierror)
INTEGER comm, coords( * ), rank, ierror
For a process group with cartesian structure, the function MPI.CART.RANK translates the logical process coordinates to process ranks as they are used by the point-to-point routines.

For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that is, coords(i) < 0 or coords(i) ≥ dime(i), it is shifted back to the interval 0 ≤ coords(i) < dime(i) automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

**MPI.CART.COORDS(comm, rank, maxdims, coords)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator with cartesian structure (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>rank</td>
<td>rank of a process within group of comm (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>maxdims</td>
<td>length of vector coord in the calling program (integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>coords</td>
<td>integer array (of size ndims) containing the cartesian coordinates of specified process (integer)</td>
</tr>
</tbody>
</table>

```c
int MPI.Cart.coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

**MPI.Cart.COORDS(comm, rank, maxdims, coords, ierror)**

```c
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

The inverse mapping, rank-to-coordinates translation is provided by MPI.CART.COORDS.

**MPI.GRAPH.NEIGHBORS.COUNT(comm, rank, nneighbors)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator with graph topology (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>rank</td>
<td>rank of process in group of comm (integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>nneighbors</td>
<td>number of neighbors of specified process (integer)</td>
</tr>
</tbody>
</table>

```c
int MPI.Graph.neighbors.count(MPI_Comm comm, int rank, int *nneighbors)
```

**MPI.GRAPH.NEIGHBORS.COUNT(comm, rank, nneighbors, ierror)**

```c
INTEGER COMM, RANK, NNEIGHBORS, IERROR
```

**MPI.GRAPH.NEIGHBORS.COUNT and MPI.GRAPH.NEIGHBORS** provide adjacency information for a general, graph topology.

**MPI.GRAPH.NEIGHBORS(comm, rank, maxneighbors, neighbors)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator with graph topology (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>rank</td>
<td>rank of process in group of comm (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>maxneighbors</td>
<td>size of array neighbors (integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>neighbors</td>
<td>ranks of processes that are neighbors to specified process (array of integer)</td>
</tr>
</tbody>
</table>
Example 6.3 Suppose that \( \text{comm} \) is a communicator with a shuffle-exchange topology. The group has \( 2^n \) members. Each process is labeled by \( a_1, \ldots, a_n \) with \( a_i \in \{0, 1\} \), and has three neighbors: exchange \( (a_1, \ldots, a_n) = (a_1, \ldots, a_{n-1}, \bar{a}_n) \), shuffle \( (a_1, \ldots, a_n) = (a_2, \ldots, a_n, a_1) \), and unshuffle \( (a_1, \ldots, a_n) = (a_n, a_1, \ldots, a_{n-1}) \). The graph adjacency list is illustrated below for \( n = 3 \).

<table>
<thead>
<tr>
<th>node</th>
<th>exchange neighbors(1)</th>
<th>shuffle neighbors(2)</th>
<th>unshuffle neighbors(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (000)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 (001)</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2 (010)</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3 (011)</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4 (100)</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5 (101)</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>6 (110)</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>7 (111)</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Suppose that the communicator \( \text{comm} \) has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```c
C assume: each process has stored a real number A.
C extract neighborhood information
    CALL MPI_COMM_RANK(comm, myrank, ierr)
    CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
C perform exchange permutation
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
                              + neighbors(1), 0, comm, status, ierr)
C perform shuffle permutation
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
                              + neighbors(3), 0, comm, status, ierr)
C perform unshuffle permutation
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
                              + neighbors(2), 0, comm, status, ierr)
```

### 6.5.5 CARTESIAN SHIFT COORDINATES

If the process topology is a cartesian structure, a MPI_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and
the rank of a destination process for the send. If the function `MPI_CART_SHIFT` is called for a cartesian process group, it provides the calling process with the above identifiers, which then can be passed to `MPI_SENDRECV`. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

```c
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
```

### Example 6.4
The communicator, `comm`, has a two-dimensional, periodic, cartesian topology associated with it. A two-dimensional array of `REALS` is stored one element per process, in variable `A`. One wishes to skew this array, by shifting column `i` (vertically, i.e., along the column) by `i` steps.

```c
....
C find process rank
    CALL MPI_COMM_RANK(comm, rank, ierr))
C find cartesian coordinates
    CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
    CALL MPI_CART_SHIFT(comm, 1, coords(2), source, dest, ierr)
C skew array
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &status, ierr)
```

### 6.5.6 Partitioning of Cartesian Structures

- `MPI_CART_SUB(comm, remain_dims, newcomm)`

```c
IN comm communicator with cartesian structure (handle)
IN remain_dims the ith entry of remain_dims specifies whether
```
int MPI.Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)

If a cartesian topology has been created with MPI_CART_CREATE, the function MPI_CART_SUB can be used to partition the communicator group into subgroups that form lower-dimensional cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid cartesian topology. (This function is closely related to MPI_COMM_SPLIT.)

Example 6.5 Assume that MPI_CART_CREATE(…, comm) has defined a (2 x 3 x 4) grid. Let remain_dims = (true, false, true). Then a call to,

    MPI_CART_SUB(comm, remain_dims, comm_new),

will create three communicators each with eight processes in a 2 x 4 cartesian topology. If remain_dims = (false, false, true) then the call to MPI_CART_SUB(comm, remain_dims, comm_new) will create six non-overlapping communicators, each with four processes, in a one-dimensional cartesian topology.

### 6.5.7 LOW-LEVEL TOPOLOGY FUNCTIONS

The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, unless he or she is creating additional virtual topology capability other than that provided by MPI.

MPI_CART_MAP(comm, ndims, dims, periods, newrank)

IN comm input communicator (handle)
IN ndims number of dimensions of cartesian structure (integer)
IN dims integer array of size ndims specifying the number of processes in each coordinate direction
IN periods logical array of size ndims specifying the periodicity specification in each coordinate direction
OUT newrank reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)
int MPI.Cart.map(MPI.Comm comm, int ndims, int *dims, int *periods,
int *newrank)

MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
LOGICAL PERIODS(*)

MPI_CART_MAP computes an "optimal" placement for the calling process on the physical machine. A possible implementation of this function is to always return the rank of the calling process, that is, not to perform any reordering.

Advice to implementors. The function MPI_CART_CREATE(comm, ndims, dims, periods, reorder, comm.cart), with reorder = true can be implemented by calling MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling MPI.COMM.SPLIT(comm, color, key, comm.cart), with color = 0 if newrank ≠ MPI.UNDEFINED, color = MPI.UNDEFINED otherwise, and key = newrank.

The function MPI_CART_SUB(comm, remaindims, comm.new) can be implemented by a call to MPI.COMM.SPLIT(comm, color, key, comm.new), using a single number encoding of the lost dimensions as color and a single number encoding of the preserved dimensions as key.

All other cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (End of advice to implementors.)

The corresponding new function for general graph structures is as follows.

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)

IN comm input communicator (handle)
IN nnodes number of graph nodes (integer)
IN index integer array specifying the graph structure, see MPI_GRAPH_CREATE
IN edges integer array specifying the graph structure
OUT newrank reordered rank of the calling process; MPI.UNDEFINED if the calling process does not belong to graph (integer)

int MPI.Graph.map(MPI.Comm comm, int nnodes, int *index, int *edges,
int *newrank)

MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR

Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm.graph), with reorder = true can be implemented by calling MPI_GRAPH_MAP(comm, nnodes, index, edges,
integer ndims, num_neigh
logical reorder
parameter (ndims=2, num_neigh=4, reorder=.true.)
integer comm, comm.cart, dims(ndims), neigh.def(ndims), ierr
integer neigh_rank(num_neigh), own.position(ndims), i, j
logical periods(ndims)
real*8 u(0:101,0:101), f(0:101,0:101)
data dims / ndims * 0 /
comm = MPI.COMM_WORLD

C Set process grid size and periodicity
call MPI.DIMS.CREATE(comm, ndims, dims)ierr
periods(1) = .TRUE.
periods(2) = .TRUE.

C Create a grid structure in WORLD group and inquire about own position
call MPI.CART.CREATE (comm, ndims, dims, periods, reorder, comm.cart,ierr)
call MPI.CART.GET (comm.cart, ndims, dims, periods, own.position,ierr)

C Look up the ranks for the neighbors. Own process coordinates are (i, j).
C Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1)
i = own.position(1)
j = own.position(2)
neigh.def(1) = i-1
neigh.def(2) = j
call MPI.CART.RANK (comm.cart, neigh.def, neigh_rank(1),ierr)
neigh.def(1) = i+1
neigh.def(2) = j
call MPI.CART.RANK (comm.cart, neigh.def, neigh_rank(2),ierr)
neigh.def(1) = i
neigh.def(2) = j-1
call MPI.CART.RANK (comm.cart, neigh.def, neigh_rank(3),ierr)
neigh.def(1) = i
neigh.def(2) = j+1
call MPI.CART.RANK (comm.cart, neigh.def, neigh_rank(4),ierr)

C Initialize the grid functions and start the iteration
call init (u, f)
do 10 it=1,100
   call relax (u, f)
C Exchange data with neighbor processes
call exchange (u, comm.cart, neigh.rank, num.neigh)
10 continue
call output (u)
end

---

Fig. 6.1 Set-up of process structure for two-dimensional parallel Poisson solver.
newrank), then calling MPI.COMM.Split(comm, color, key, comm.graph),
with color = 0 if newrank ≠ MPI.UNDEFINED, color = MPI.UNDEFINED otherwise,
and key = newrank.

All other graph topology functions can be implemented locally, using
the topology information that is cached with the communicator. (End of
advice to implementors.)

6.6 An Application Example

Example 6.6 The example in figure 6.1 shows how the grid definition and in-
quiry functions can be used in an application program. A partial differential
equation, for instance the Poisson equation, is to be solved on a rectangular do-
main. First, the processes organize themselves in a two-dimensional structure.
Each process then inquires about the ranks of its neighbors in the four directions
(up, down, right, left). The numerical problem is solved by an iterative method,
the details of which are hidden in the subroutine relax.

In each relaxation step each process computes new values for the solution
grid function at all points owned by the process. Then the values at inter-process
boundaries have to be exchanged with neighboring processes. For example, the
exchange subroutine might contain a call like MPI.SEND(...,neigh_rank(1),...)
to send updated values to the left-hand neighbor (i-1,j).