3.7.3 COMMUNICATION COMPLETION

The functions MPI.WAIT and MPI.TEST are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a synchronous mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive.

The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated).

We shall use the following terminology. A null handle is a handle with value MPI.REQUEST.NULL. A persistent request and the handle to it are inactive if the request is not associated with any ongoing communication (see Section 3.9). A handle is active if it is neither null nor inactive.

**MPI.WAIT(request, status)**

```c
INOUT request request (handle)
OUT status status object (Status)
```

```c
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

**MPI.WAITREQUEST, STATUS, IERROR**

```c
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

A call to MPI.WAIT returns when the operation identified by request is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI.WAIT and the request handle is set to MPI.REQUEST.NULL. MPI.WAIT is a non-local operation.

The call returns, in status, information on the completed operation. The content of the status object for a receive operation can be accessed as described in Section 3.2.5. The status object for a send operation may be queried by a call to MPI.TEST.CANCELLED (see Section 3.8).

One is allowed to call MPI.WAIT with a null or inactive request argument. In this case the operation returns immediately. The status argument is set to return tag = MPI.ANY_TAG, source = MPI.ANY_SOURCE, and is also internally configured so that calls to MPI.GET.COUNT and MPI.GET_ELEMENTS return count = 0.

**Rationale.** This makes MPI.WAIT functionally equivalent to MPI.WAITALL with a list of length one and adds some elegance. Status is set in this way so as to prevent errors due to accesses of stale information.
Successful return of MPI\_WAIT after a MPI\_ISSEND implies that the
user send buffer can be reused—i.e., data has been sent out or copied into
a buffer attached with MPI\_BUFFER\_ATTACH. Note that, at this point, we
can no longer cancel the send (see Section 3.8). If a matching receive is
never posted, then the buffer cannot be freed. This runs somewhat counter
to the stated goal of MPI\_CANCEL (always being able to free program space
that was committed to the communication subsystem). (End of rationale.)

Advice to implementors. In a multi-threaded environment, a call to MPI\_WAIT
should block only the calling thread, allowing the thread scheduler to
schedule another thread for execution. (End of advice to implementors.)

\textbf{MPI\_TEST(request, flag, status)}

\begin{tabular}{ll}
\textbf{INOUT} & \textbf{request} & \textbf{communication request (handle)} \\
\textbf{OUT} & \textbf{flag} & \textbf{true if operation completed (logical)} \\
\textbf{OUT} & \textbf{status} & \textbf{status object (Status)} \\
\end{tabular}

\begin{verbatim}
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

MPI\_TEST\_REQUEST, FLAG, STATUS, IERROR)
    \textbf{LOGICAL} \textbf{FLAG}
    \textbf{INTEGER} \textbf{REQUEST, STATUS(MPI\_STATUS\_SIZE), IERROR}
\end{verbatim}

A call to MPI\_TEST returns flag = true if the operation identified by request
is complete. In such a case, the status object is set to contain information on
the completed operation; if the communication object was created by a non-
blocking send or receive, then it is deallocated and the request handle is set to
MPI\_REQUEST\_NULL. The call returns flag = false, otherwise. In this case, the
value of the status object is undefined. MPI\_TEST is a local operation.

The return status object for a receive operation carries information that can
be accessed as described in Section 3.2.5. The status object for a send operation
carries information that can be accessed by a call to MPI\_TEST\_CANCELL\_ED (see
Section 3.8).

One is allowed to call MPI\_TEST with a null or inactive request argument.
In such a case the operation returns flag = false.

The functions MPI\_WAIT and MPI\_TEST can be used to complete both sends
and receives.

\textbf{Advice to users.} The use of the nonblocking MPI\_TEST call allows the
user to schedule alternative activities within a single thread of execution.
An event-driven thread scheduler can be emulated with periodic calls to
MPI\_TEST. (End of advice to users.)

\textbf{Example 3.10} Simple usage of nonblocking operations and MPI\_WAIT.
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
ELSE
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
END IF

A request object can be deallocated without waiting for the associated communication to complete, by using the following operation.

MPI_REQUEST_FREE(request)

INOUT  request  communication request (handle)

int MPI_Request_free(MPI_Request *request)

MPI_REQUEST_FREE(request, IERROR)

INTEGER REQUEST, IERROR

Mark the request object for deallocation and set request to MPI_REQUEST_NULL. An ongoing communication that is associated with the request will be allowed to complete. The request will be deallocated only after its completion.

Rationale. The MPI_REQUEST_FREE mechanism is provided for reasons of performance and convenience on the sending side. (End of rationale.)

Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not possible to check for the successful completion of the associated communication with calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user—such an error must be treated as fatal. Questions arise as to how one knows when the operations have completed when using MPI_REQUEST_FREE. Depending on the program logic, there may be other ways in which the program knows that certain operations have completed and this makes usage of MPI_REQUEST_FREE practical. For example, an active send request could be freed when the logic of the program is such that the receiver sends a reply to the message sent—the arrival of the reply informs the sender that the send has completed and the send buffer can be reused. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (End of advice to users.)
Example 3.11 An example using MPI_REQUEST_FREE.

CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
IF(rank.EQ.0) THEN
   DO i=1, n
      CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
   END DO
ELSE   ! rank.EQ.1
      CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, req, ierr)
      CALL MPI_WAIT(req, status)
      DO i=1, n-1
         CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, req, ierr)
         CALL MPI_REQUEST_FREE(req, ierr)
         CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, req, ierr)
         CALL MPI_WAIT(req, status, ierr)
      END DO
      CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, req, ierr)
      CALL MPI_WAIT(req, status)
   END IF

3.7.4 SEMANTICS OF NONBLOCKING COMMUNICATIONS

The semantics of nonblocking communication is defined by suitably extending the definitions in Section 3.5.

Order Nonblocking communication operations are ordered according to the execution order of the calls that initiate the communication. The non-overtaking requirement of Section 3.5 is extended to nonblocking communication, with this definition of order being used.

Example 3.12 Message ordering for nonblocking operations.

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
   CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
   CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE   ! rank.EQ.1
   CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
   CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1,status)
CALL MPI_WAIT(r2,status)
The first send of process zero will match the first receive of process one, even if both messages are sent before process one executes either receive.

**Progress** A call to `MPI_WAIT` that completes a receive will eventually terminate and return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is nonblocking, then the receive should complete even if no call is executed by the sender to complete the send. Similarly, a call to `MPI_WAIT` that completes a send will eventually return if a matching receive has been started, unless the receive is satisfied by another send, and even if no call is executed to complete the receive.

**Example 3.13** An illustration of progress semantics.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
    CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE     ! rank.EQ.1
    CALL MPI_IRecv(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
    CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, ierr)
    CALL MPI_WAIT(r, status, ierr)
END IF
```

This code should not deadlock in a correct MPI implementation. The first synchronous send of process zero must complete after process one posts the matching (nonblocking) receive even if process one has not yet reached the completing wait call. Thus, process zero will continue and execute the second send, allowing process one to complete execution.

If an `MPI_TEST` that completes a receive is repeatedly called with the same arguments, and a matching send has been started, then the call will eventually return `flag = true`, unless the send is satisfied by another receive. If an `MPI_TEST` that completes a send is repeatedly called with the same arguments, and a matching receive has been started, then the call will eventually return `flag = true`, unless the receive is satisfied by another send.

### 3.7.5 Multiple Completions

It is convenient to be able to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to `MPI_WAITANY` or `MPI_TESTANY` can be used to wait for the completion of one out of several operations. A call to `MPI_WAITALL` or `MPI_TESTALL` can be used to wait for all pending operations in a list. A call to `MPI_WAITSAME` or `MPI_TESTSOME` can be used to complete all enabled operations in a list.
MPI.WAITANY (count, array_of_requests, index, status)

IN count list length (integer)
INOUT array_of_requests array of requests (array of handles)
OUT index index of handle for operation that completed (integer)
OUT status status object (Status)

int MPI.Waitany(int count, MPI_Request *array_of_requests, int *index,
                 MPI_Status *status)

Blocks until one of the operations associated with the active requests in
the array has completed. If more then one operation is enabled and can terminate,
one is arbitrarily chosen. Returns in index the index of that request in the array
and returns in status the status of the completing communication. (The array is
indexed from zero in C, and from one in Fortran.) If the request was allocated by
a nonblocking communication operation, then it is deallocated and the request
handle is set to MPI_REQUEST_NULL.

The array_of_requests list may contain null or inactive handles. If the list
contains no active handles (list has length zero or all entries are null or inactive),
then the call returns immediately with index = MPI.UNDEFINED.

The execution of MPI.WAITANY(count, array_of_requests, index, status) has
the same effect as the execution of MPI.WAIT(array_of_requests[i], status),
where i is the value returned by index. MPI.WAITANY with an array containing
one active entry is equivalent to MPI.WAIT.

MPI.TESTANY (count, array_of_requests, index, flag, status)

IN count list length (integer)
INOUT array_of_requests array of requests (array of handles)
OUT index index of operation that completed, or MPI.UNDEFINED if none completed (integer)
OUT flag true if one of the operations is complete (logical)
OUT status status object (Status)

int MPI.Testany(int count, MPI_Request *array_of_requests, int *index,
                int *flag, MPI_Status *status)

MPI.TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)

LOGICAL FLAG
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI.STATUS_SIZE),
IERROR
Tests for completion of either one or none of the operations associated with active handles. In the former case, it returns flag = true, returns in index the index of this request in the array, and returns in status the status of that operation; if the request was allocated by a nonblocking communication call then the request is deallocated and the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from one in Fortran.) In the latter case, it returns flag = false, returns a value of MPI_UNDEFINED in index and status is undefined. The array may contain null or inactive handles. If the array contains no active handles then the call returns immediately with flag = false, index = MPI_UNDEFINED, and status undefined.

The execution of MPI.TESTANY(count, array_of_requests, index, status) has the same effect as the execution of MPI.TEST(array_of_requests[i], flag, status), for i=0, 1, ..., count-1, in some arbitrary order, until one call returns flag = true, or all fail. In the former case, index is set to the last value of i, and in the latter case, it is set to MPI_UNDEFINED. MPI.TESTANY with an array containing one active entry is equivalent to MPI.TEST.

MPI.WAITALL( count, array_of_requests, array_of_statuses)

IN  count  lists length (integer)
INOUT array_of_requests  array of requests (array of handles)
OUT  array_of_statuses  array of status objects (array of Status)

int MPI.Waitall(int count, MPI.Request *array_of_requests,
                MPI.Status *array_of_statuses)

MPI.WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)

INTEGER COUNT, ARRAY_OF_REQUESTS(*)
INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The i-th entry in array_of_statuses is set to the return status of the i-th operation. Requests that were created by nonblocking communication operations are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. The list may contain null or inactive handles. The call returns in the status of each such entry tag = MPI.Any_Tag, source = MPI.Any_Source, and each status entry is also configured so that calls to MPI.GET.COUNT and MPI.GET.ELEMENTS return count = 0.

The execution of MPI.WAITALL(count, array_of_requests, array_of_statuses) has the same effect as the execution of MPI.WAIT(array_of_requests[i], array_of_statuses[i]), for i=0, ..., count-1, in some arbitrary order. MPI.WAITALL with an array of length one is equivalent to MPI.WAIT.
MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)

IN count lists length (integer)
INOUT array_of_requests array of requests (array of handles)
OUT flag (logical)
OUT array_of_statuses array of status objects (array of Status)

int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
                MPI_Status *array_of_statuses)

MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
LOGICAL FLAG
INTEGER COUNT, ARRAY_OF_REQUESTS(*),
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

Returns flag = true if all communications associated with active handles in
the array have completed (this includes the case where no handle in the list is ac-
tive). In this case, each status entry that corresponds to an active handle request
is set to the status of the corresponding communication; if the request was allo-
cated by a nonblocking communication call then it is deallocated, and the handle
is set to MPI_REQUEST_NULL. Each status entry that corresponds to a null or in-
active handle is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE,
and is also configured so that calls to MPI_GET_COUNT and MPI_GET_ELEMENTS
return count = 0.

Otherwise, flag = false is returned, no request is modified and the values of
the status entries are undefined. This is a local operation.

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN incount length of array_of_requests (integer)
INOUT array_of_requests array of requests (array of handles)
OUT outcount number of completed requests (integer)
OUT array_of_indices array of indices of operations that completed
              (array of integers)
OUT array_of_statuses array of status objects for operations that
              completed (array of Status)

int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount,
                 int *array_of_indices, MPI_Status *array_of_statuses)

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
              ARRAY_OF_STATUSES, IERROR)
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
Waits until at least one of the operations associated with active handles in the list have completed. Returns in `outcount` the number of requests from the list `array_of_requests` that have completed. Returns in the first `outcount` locations of the array `array_of_indices` the indices of these operations (index within the array `array_of_requests`; the array is indexed from zero in C and from one in Fortran). Returns in the first `outcount` locations of the array `array_of_statuses` the status for these completed operations. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to `MPI_REQUEST_NULL`.

If the list contains no active handles, then the call returns immediately with `outcount = 0`.

`MPI_TESTSOME` (incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

- **IN** `incount` length of `array_of_requests` (integer)
- **INOUT** `array_of_requests` array of requests (array of handles)
- **OUT** `outcount` number of completed requests (integer)
- **OUT** `array_of_indices` array of indices of operations that completed (array of integers)
- **OUT** `array_of_statuses` array of status objects for operations that completed (array of Status)

```c
int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount,
                  int *array_of_indices, MPI_Status *array_of_statuses)
```

Behaves like `MPI_Waitsome`, except that it returns immediately. If no operation has completed it returns `outcount = 0`.

`MPI_TESTSOME` is a local operation, which returns immediately, whereas `MPI_Waitsome` will block until a communication completes, if it was passed a list that contains at least one active handle. Both calls fulfill a fairness requirement: If a request for a receive repeatedly appears in a list of requests passed to `MPI_Waitsome` or `MPI_Testsome`, and a matching send has been posted, then the receive will eventually succeed, unless the send is satisfied by another receive; and similarly for send requests.

Advice to users. The use of `MPI_TESTSOME` is likely to be more efficient than the use of `MPI_TESTANY`. The former returns information on all completed communications, with the latter, a new call is required for each communication that completes.
A server with multiple clients can use MPI_WAITSOME so as not to starve any client. Clients send messages to the server with service requests. The server calls MPI_WAITSOME with one receive request for each client, and then handles all receives that completed. If a call to MPI_WAITANY is used instead, then one client could starve while requests from another client always sneak in first. (End of advice to users.)

Advice to implementors. MPI_TESTSOME should complete as many pending communications as possible. (End of advice to implementors.)

Example 3.14 Client-server code (starvation can occur).

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank > 0) THEN     ! client code
   DO WHILE(.TRUE.)
      CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
      CALL MPI_WAIT(request, status, ierr)
   END DO
ELSE                  ! rank=0 -- server code
   DO i=1, size-1
      CALL MPI_IRECV(a(i), n, MPI_REAL, 0, tag, comm, request_list(i), ierr)
   END DO
   DO WHILE(.TRUE.)
      CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
      CALL DO_SERVICE(a(index)) ! handle one message
      CALL MPI_IRECV(a(1, index), n, MPI_REAL, 0, tag, comm, request_list(index), ierr)
   END DO
END IF
```

Example 3.15 Same code, using MPI_WAITSOME.

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank > 0) THEN     ! client code
   DO WHILE(.TRUE.)
      CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
      CALL MPI_WAIT(request, status, ierr)
   END DO
ELSE                  ! rank=0 -- server code
   DO i=1, size-1
```
CALL MPI_RECV(a(1,i), n, MPI_REAL, 0, tag,  
    comm, request_list(i), ierr)
END DO
END DO

3.8 Probe and Cancel

The MPI_PROBE and MPI_RECV operations allow incoming messages to be  
checked for, without actually receiving them. The user can then decide how to  
receive them, based on the information returned by the probe (basically, the  
information returned by status). In particular, the user may allocate memory  
for the receive buffer, according to the length of the probed message.

The MPI_CANCEL operation allows pending communications to be canceled.  
This is required for cleanup. Posting a send or a receive ties up user resources  
(send or receive buffers), and a cancel may be needed to free these resources  
gracefully.

MPI_PROBE(source, tag, comm, flag, status)

IN    source    source rank, or MPI_ANY_SOURCE (integer)
IN    tag       tag value or MPI_ANY_TAG (integer)
IN    comm      communicator (handle)
OUT   flag      (logical)
OUT   status    status object (Status)

int MPIprobe(int source, int tag, MPI_Comm comm, int *flag,  
    MPI_Status *status)

MPI_PROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)

LOGICAL FLAG

INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_PROBE(source, tag, comm, flag, status) returns flag = true if there is  
a message that can be received and that matches the pattern specified by the  
arguments source, tag, and comm. The call matches the same message that  
would have been received by a call to MPI_RECV(…, source, tag, comm, status)  
executed at the same point in the program, and returns in status the same value
that would have been returned by `MPI_RECV()`. Otherwise, the call returns `flag = false`, and leaves `status` undefined.

If `MPI_PROBE` returns `flag = true`, then the content of the status object can be subsequently accessed as described in Section 3.2.5 to find the source, tag and length of the probed message.

A subsequent receive executed with the same context, and the source and tag returned in `status` by `MPI_PROBE` will receive the message that was matched by the probe, if no other intervening receive occurs after the probe. If the receiving process is multi-threaded, it is the user’s responsibility to ensure that the last condition holds.

The source argument of `MPI_PROBE` can be `MPI_ANY_SOURCE`, and the tag argument can be `MPI_ANY_TAG`, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the `comm` argument.

It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.

```
MPI_PROBE(source, tag, comm, status)

IN source source rank, or MPI_ANY_SOURCE (integer)
IN tag tag value, or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT status status object (Status)
```

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

`MPI_PROBE` behaves like `MPI_IPROBE` except that it is a blocking call that returns only after a matching message has been found.

The MPI implementation of `MPI_PROBE` and `MPI_IPROBE` needs to guarantee progress: if a call to `MPI_PROBE` has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to `MPI_PROBE` will return, unless the message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with `MPI_PROBE` and a matching message has been issued, then the call to `MPI_PROBE` will eventually return `flag = true` unless the message is received by another concurrent receive operation.

**Example 3.16** Use blocking probe to wait for an incoming message.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
```
ELSE IF(rank.EQ.1) THEN
  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE  ! rank.EQ.2
  DO i=1, 2
    CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                 comm, status, ierr)
    IF (status(MPI_SOURCE) = 0) THEN
      CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, status, ierr)
    ELSE
      CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, status, ierr)
    END IF
  END DO
END IF

Each message is received with the right type.

Example 3.17 A similar program to the previous example, but now it has a problem.

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
ELSE IF(rank.EQ.1) THEN
  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE
  DO i=1, 2
    CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                   comm, status, ierr)
    IF (status(MPI_SOURCE) = 0) THEN
      CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
                    0, status, ierr)
    ELSE
      CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
                    0, status, ierr)
    END IF
  END DO
END IF

We slightly modified example 3.16, using MPI_ANY_SOURCE as the source argument in the two receive calls in statements labeled 100 and 200. The program is now incorrect: the receive operation may receive a message that is distinct from the message probed by the preceding call to MPI_PROBE.

Advice to implementors. A call to MPI_PROBE(source, tag, comm, status) will match the message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point. Suppose that this
message has source $s$, tag $t$ and communicator $c$. If the tag argument in the probe call has value MPI.ANY.TAG then the message probed will be the earliest pending message from source $s$ with communicator $c$ and any tag; in any case, the message probed will be the earliest pending message from source $s$ with tag $t$ and communicator $c$ (this is the message that would have been received, so as to preserve message order). This message continues as the earliest pending message from source $s$ with tag $t$ and communicator $c$, until it is received. A receive operation subsequent to the probe that uses the same communicator as the probe and uses the tag and source values returned by the probe, must receive this message, unless it has already been received by another receive operation. (End of advice to implementors.)

MPI.CANCEL(request)

IN request communication request (handle)

int MPI_Cancel(MPI_Request *request)

MPI.CANCEL REQUEST, IERROR
INTEGER REQUEST, IERROR

A call to MPI.CANCEL marks for cancellation a pending, nonblocking communication operation (send or receive). The cancel call is local. It returns immediately, possibly before the communication is actually canceled. It is still necessary to complete a communication that has been marked for cancellation, using a call to MPI.REQUEST.FREE, MPI.WAIT or MPI.TEST (or any of the derived operations).

If a communication is marked for cancellation, then a MPI.WAIT call for that communication is guaranteed to return, irrespective of the activities of other processes (i.e., MPI.WAIT behaves as a local function); similarly if MPI.TEST is repeatedly called in a busy wait loop for a canceled communication, then MPI.TEST will eventually be successful.

MPI.CANCEL can be used to cancel a communication that uses a persistent request (see Section 3.9), in the same way it is used for nonpersistent requests. A successful cancellation cancels the active communication, but not the request itself. After the call to MPI.CANCEL and the subsequent call to MPI.WAIT or MPI.TEST, the request becomes inactive and can be activated for a new communication.

The successful cancellation of a buffered send frees the buffer space occupied by the pending message.

Either the cancellation succeeds, or the communication succeeds, but not both. If a send is marked for cancellation, then it must be the case that either the send completes normally, in which case the message sent was received at the destination process, or that the send is successfully canceled, in which case no part of the message was received at the destination. Then, any matching receive has to be satisfied by another send. If a receive is marked for cancellation, then
it must be the case that either the receive completes normally, or that the receive is successfully canceled, in which case no part of the receive buffer is altered. Then, any matching send has to be satisfied by another receive.

If the operation has been canceled, then information to that effect will be returned in the status argument of the operation that completes the communication.

```c
MPI_TEST_CANCELED(status, flag)

IN    status     status object (Status)
OUT   flag        (logical)
```

```c
int MPI.Test.cancelled(MPI.Status *status, int *flag)

MPI_TEST_CANCELED(STATUS, FLAG, IERROR)
  LOGICAL FLAG
  INTEGER STATUS(MPI.STATUS_SIZE), IERROR
```

Returns `flag = true` if the communication associated with the status object was canceled successfully. In such a case, all other fields of `status` (such as `count` or `tag`) are undefined. Returns `flag = false`, otherwise. If a receive operation might be canceled then one should call `MPI.TEST_CANCELED` first, to check whether the operation was canceled, before checking on the other fields of the return status.

Advice to users. Cancel can be an expensive operation that should be used only exceptionally. (End of advice to users.)

Advice to implementors. If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement `MPI.CANCEL`, this is still a local operation, since its completion does not depend on the code executed by other processes. If processing is required on another process, this should be transparent to the application (hence the need for an interrupt and an interrupt handler). (End of advice to implementors.)

### 3.9 Persistent Communication Requests

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a persistent communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a "half-channel." It does
not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another. It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

A persistent communication request is created using one of the four following calls. These calls involve no communication.

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)

| IN   | buf            | initial address of send buffer (choice) |
| IN   | count          | number of elements sent (integer)      |
| IN   | datatype       | type of each element (handle)          |
| IN   | dest           | rank of destination (integer)          |
| IN   | tag            | message tag (integer)                  |
| IN   | comm           | communicator (handle)                  |
| OUT  | request        | communication request (handle)         |

int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
                   int tag, MPI_Comm comm, MPI_Request *request)

MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

Creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.

MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)

| IN   | buf            | initial address of send buffer (choice) |
| IN   | count          | number of elements sent (integer)      |
| IN   | datatype       | type of each element (handle)          |
| IN   | dest           | rank of destination (integer)          |
| IN   | tag            | message tag (integer)                  |
| IN   | comm           | communicator (handle)                  |
| OUT  | request        | communication request (handle)         |

int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
                    int tag, MPI_Comm comm, MPI_Request *request)

MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
Creates a persistent communication request for a buffered mode send.

**MPI.SSEND_INIT**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>buf</td>
</tr>
<tr>
<td>IN</td>
<td>count</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
</tr>
<tr>
<td>IN</td>
<td>dest</td>
</tr>
<tr>
<td>IN</td>
<td>tag</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
</tr>
<tr>
<td>OUT</td>
<td>request</td>
</tr>
</tbody>
</table>

```c
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
                    int tag, MPI_Comm comm, MPI_Request *request)
```

**MPI.SSEND_INIT**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUF</td>
<td>(choice)</td>
</tr>
<tr>
<td>IN</td>
<td>COUNT</td>
</tr>
<tr>
<td>IN</td>
<td>DATATYPE</td>
</tr>
<tr>
<td>IN</td>
<td>DEST</td>
</tr>
<tr>
<td>IN</td>
<td>TAG</td>
</tr>
<tr>
<td>IN</td>
<td>COMM</td>
</tr>
<tr>
<td>OUT</td>
<td>REQUEST</td>
</tr>
<tr>
<td>IERROR</td>
<td></td>
</tr>
</tbody>
</table>

Creates a persistent communication object for a synchronous mode send operation.

**MPI.RSEND_INIT**

```c
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
                    int tag, MPI_Comm comm, MPI_Request *request)
```

**MPI.RSEND_INIT**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUF</td>
<td>(choice)</td>
</tr>
<tr>
<td>IN</td>
<td>COUNT</td>
</tr>
<tr>
<td>IN</td>
<td>DATATYPE</td>
</tr>
<tr>
<td>IN</td>
<td>DEST</td>
</tr>
<tr>
<td>IN</td>
<td>TAG</td>
</tr>
<tr>
<td>IN</td>
<td>COMM</td>
</tr>
<tr>
<td>OUT</td>
<td>REQUEST</td>
</tr>
<tr>
<td>IERROR</td>
<td></td>
</tr>
</tbody>
</table>

Creates a persistent communication object for a ready mode send operation.

**MPI.RECV_INIT**

```c
int MPI_RECV_INIT(void* buf, int count, MPI_Datatype datatype, int source,
                   int tag, MPI_Comm comm, MPI_Request *request)
```
int MPI_Comm_create(MPI_Comm parent, const char name[], MPI_Comm *comm)

MPI_COMM_CREATE(CHARACTER NAME, COMM, IERROR)

An alternative communicator creation routine.

Creates a new communicator based on the properties of
another communicator.

IN parent communicator or handle

INOUT comm new communicator handle

OUT name name of new communicator (string)

int MPI_Comm_set_attribute(MPI_Comm comm, MPI_Aint offset, const void *value, int tag)

MPI_COMM_SET_ATTRIBUTE(ADDRESS OFFSET, VALUE, TAG, COMM, IERROR)

Sets a communication attribute.

IN comm communicator handle

IN offset communication attribute offset

IN value communication attribute value

IN tag communication attribute tag

int MPI_Comm_free(MPI_Comm comm)

MPI_COMM_FREE(COMM, IERROR)

Frees a communicator.

IN comm communicator handle

int MPI_Comm_get_size(MPI_Comm comm, int *size)

MPI_COMM_GET_SIZE(COMM, SIZE, IERROR)

Returns the size of a communicator.

IN comm communicator handle

OUT size size of communicator (integer)

int MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI_COMM_RANK(COMM, RANK, IERROR)

Returns the rank of an individual communicator.

IN comm communicator handle

OUT rank communicator rank (integer)

int MPI_Comm_size(MPI_Comm comm, int *size)

MPI_COMM_SIZE(COMM, SIZE, IERROR)

Returns the size of a communicator.

IN comm communicator handle

OUT size size of communicator (integer)
MPI.STARTALL(count, array_of_requests, ierror)

INTEGER count, array_of_requests(*), ierror

Start all communications associated with requests in array_of_requests. A call to MPI.STARTALL(count, array_of_requests) has the same effect as calls to MPI.START(array_of_requests[i]), executed for i=0,..., count-1, in some arbitrary order.

A communication started with a call to MPI.START or MPI.STARTALL is completed by a call to MPI.WAIT, MPI.TEST, or one of the derived functions described in Section 3.7.5. The request becomes inactive after successful completion of such call. The request is not deallocated and it can be activated anew by an MPI.START or MPI.STARTALL call.

A persistent request is deallocated by a call to MPI.REQUEST_FREE (Section 3.7.3).

The call to MPI.REQUEST_FREE can occur at any point in the program after the persistent request was created. However, the request will be deallocated only after it becomes inactive. Active receive requests should not be freed. Otherwise, it will not be possible to check that the receive has completed. It is preferable, in general, to free requests when they are inactive. If this rule is followed, then the functions described in this section will be invoked in a sequence of the form, Create(Start Complete)* Free, where * indicates zero or more repetitions. If the same communication object is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is obeyed.

A send operation initiated with MPI.START can be matched with any receive operation and, likewise, a receive operation initiated with MPI.START can receive messages generated by any send operation.

3.10 Send-receive

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 6 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.
MPI_SENDRECV (sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

| IN    | sendbuf      | initial address of send buffer (choice) |
| IN    | sendcount    | number of elements in send buffer (integer) |
| IN    | sendtype     | type of elements in send buffer (handle) |
| IN    | dest         | rank of destination (integer) |
| IN    | sendtag      | send tag (integer) |
| OUT   | recvbuf      | initial address of receive buffer (choice) |
| IN    | recvcount    | number of elements in receive buffer (integer) |
| IN    | recvtype     | type of elements in receive buffer (handle) |
| IN    | source       | rank of source (integer) |
| IN    | recvtag      | receive tag (integer) |
| IN    | comm         | communicator (handle) |
| OUT   | status       | status object (Status) |

int MPI_Ssendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                        int dest, int sendtag, void *recvbuf, int recvcount,
                        MPI_Datatype recvtype, int source, MPI_Datatype recvtag,
                        MPI_Comm comm, MPI_Status *status)

MPI_SENDRECV (SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
                  RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE,
              RECVTAG, COMM, STATUS(MPI_STATUS.SIZE), IERROR

Execute a blocking send and receive operation. Both send and receive use
the same communicator, but possibly different tags. The send buffer and receive
buffers must be disjoint, and may have different lengths and datatypes.

MPI_SENDRECV_REPLACE (buf, count, datatype, dest, sendtag, source, recvtag,
                       comm, status)

| INOUT | buf          | initial address of send and receive buffer (choice) |
| IN    | count        | number of elements in send and receive buffer (integer) |
| IN    | datatype     | type of elements in send and receive buffer (handle) |
| IN    | dest         | rank of destination (integer) |
| IN    | sendtag      | send message tag (integer) |
| IN    | source       | rank of source (integer) |
| IN    | recvtag      | receive message tag (integer) |
| IN    | comm         | communicator (handle) |
| OUT   | status       | status object (Status) |
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
     int dest, int sendtag, int source, int recvtag,
     MPI_Comm comm, MPI_Status *status)

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
     COMM, STATUS, IERROR)

<type> BUF(*)

INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
     STATUS(MPI_STATUS_SIZE), IERROR

Execute a blocking send and receive. The same buffer is used both for the
send and for the receive, so that the message sent is replaced by the message
received.

The semantics of a send-receive operation is what would be obtained if the
caller forked two concurrent threads, one to execute the send, and one to ex-
cute the receive, followed by a join of these two threads.

Advice to implementors. Additional intermediate buffering is needed for the
"replace" variant. (End of advice to implementors.)

3.11 Null Processes

In many instances, it is convenient to specify a “dummy” source or destination
for communication. This simplifies the code that is needed for dealing with
boundaries, for example, in the case of a non-circular shift done with calls to
send-receive.

The special value MPI_PROC_NULL can be used instead of a rank wherever a
source or a destination argument is required in a call. A communication with
process MPI_PROC_NULL has no effect. A send to MPI_PROC_NULL succeeds and
returns as soon as possible. A receive from MPI_PROC_NULL succeeds and returns
as soon as possible with no modifications to the receive buffer. When a receive
with source = MPI_PROC_NULL is executed then the status object returns source =
MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0.

3.12 Derived Datatypes

Up to here, all point-to-point communications have involved only contiguous
buffers containing a sequence of elements of the same type. This is too con-
straining on two accounts. One often wants to pass messages that contain values
with different datatypes (e.g., an integer count, followed by a sequence of real
numbers); and one often wants to send noncontiguous data (e.g., a sub-block
of a matrix). One solution is to pack noncontiguous data into a contiguous
buffer at the sender site and unpack it back at the receiver site. This has the dis-
advantage of requiring additional memory-to-memory copy operations at both
sites, even when the communication subsystem has scatter-gather capabilities.
Instead, MPI provides mechanisms to specify more general, mixed, and non-contiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shape and size. It is not assumed that the MPI library is cognizant of the objects declared in the host language. Thus, if one wants to transfer a structure, or an array section, it will be necessary to provide in MPI a definition of a communication buffer that mimics the definition of the structure or array section in question. These facilities can be used by library designers to define communication functions that can transfer objects defined in the host language—by decoding their definitions as available in a symbol table or a dope vector. Such higher-level communication functions are not part of MPI.

More general communication buffers are specified by replacing the basic datatypes that have been used so far with derived datatypes that are constructed from basic datatypes using the constructors described in this section. These methods of constructing derived datatypes can be applied recursively.

A general datatype is an opaque object that specifies two things:

- A sequence of basic datatypes
- A sequence of integer (byte) displacements

The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. We call such a pair of sequences (or sequence of pairs) a type map. The sequence of basic datatypes (displacements ignored) is the type signature of the datatype.

Let

\[
Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}
\]

be such a type map, where \(type_i\) are basic types, and \(disp_i\) are displacements. Let

\[
Typesig = \{type_0, \ldots, type_{n-1}\}
\]

be the associated type signature. This type map, together with a base address \(buf\), specifies a communication buffer: the communication buffer that consists of \(n\) entries, where the \(i\)-th entry is at address \(buf + disp_i\) and has type \(type_i\). A message assembled from such a communication buffer will consist of \(n\) values, of the types defined by \(Typesig\).

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation \(MPI\_SEND(buf, 1, datatype, \ldots)\) will use the send buffer defined by the base address \(buf\) and the general datatype associated with \(datatype\); it will generate a message with the type signature determined by the \(datatype\) argument. \(MPI\_RECV(buf, 1, \ldots)\)
datatype, ...) will use the receive buffer defined by the base address buf and the general datatypewriter associated with datatype.

General datatypes can be used in all send and receive operations. We discuss, in Section 3.12.5, the case where the second argument count has value > 1.

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, MPLINT is a predefined handle to a datatype with type map {(int, 0)}, with one entry of type int and displacement zero. The other basic datatypes are similar.

The extent of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

\[ T_{\text{typemap}} = \{(\text{type}_0, \text{disp}_0), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1})\}. \]

then

\[
\begin{align*}
\text{lb}(T_{\text{typemap}}) &= \min_j \text{disp}_j, \\
\text{ub}(T_{\text{typemap}}) &= \max_j (\text{disp}_j + \text{sizeof}((\text{type}_j)), \text{and} \\
\text{extent}(T_{\text{typemap}}) &= \text{ub}(T_{\text{typemap}}) - \text{lb}(T_{\text{typemap}}) + \epsilon. 
\end{align*}
\]

If type requires alignment to a byte address that is is a multiple of \( k_i \), then \( \epsilon \) is the least nonnegative increment needed to round extent\( (T_{\text{typemap}}) \) to the next multiple of \( \max_i k_i \).

**Example 3.18** Assume that \( T_{\text{ype}} = \{(\text{double}, 0), (\text{char}, 8)\} \) (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

**Rationale.** The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 3.12.3. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. (End of rationale.)

### 3.12.1 DATATYPE CONSTRUCTORS

Contiguous. The simplest datatype constructor is `MPI_TYPE_CONTIGUOUS` which allows replication of a datatype into contiguous locations.
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
  IN  count  replication count (nonnegative integer)
  IN  oldtype old datatype (handle)
  OUT newtype new datatype (handle)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                         MPI_Datatype *newtype)

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
  INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

newtype is the datatype obtained by concatenating count copies of oldtype. Concatenation is defined using extent as the size of the concatenated copies.

Example 3.19 Let oldtype have type map \{\{(double, 0), (char, 8)\}\}, with extent 16, and let count = 3. The type map of the datatype returned by newtype is

\{\{(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40)\}\};

i.e., alternating double and char elements, with displacements 0, 8, 16, 24, 32, 40.

In general, assume that the type map of oldtype is

\{\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}\},

with extent ex. Then newtype has a type map with count \cdot n entries defined by:

\{\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), \ldots, (type_{n-1}, disp_{n-1} + ex), \ldots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1)), \ldots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}\}.

Vector The function MPI_TYPE_VECTOR is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.
MPI_TYPE_VECTOR( count, blocklength, stride, oldtype, newtype)

IN    count          number of blocks (nonnegative integer)
IN    blocklength    number of elements in each block (nonnegative integer)
IN    stride         number of elements between start of each block (integer)
IN    oldtype        old datatype (handle)
OUT   newtype        new datatype (handle)

int MPI_Type_vector(int count, int blocklength, int stride,
                     MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)

INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

Example 3.20 Assume, again, that oldtype has type map \{\text{(double, 0)}, \text{(char, 8)}\}, with extent 16. A call to MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with type map,

\{\text{(double, 0)}, \text{(char, 8)}, \text{(double, 16)}, \text{(char, 24)}, \text{(double, 32)}, \text{(char, 40)},
\text{(double, 64)}, \text{(char, 72)}, \text{(double, 80)}, \text{(char, 88)}, \text{(double, 96)},
\text{(char, 104)}\}.

That is, two blocks with three copies each of the old type, with a stride of 4 elements (4 \times 16 \text{ bytes}) between the blocks.

Example 3.21 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,

\{\text{(double, 0)}, \text{(char, 8)}, \text{(double, -32)}, \text{(char, -24)}, \text{(double, -64)},
\text{(char, -56)}\}.

In general, assume that oldtype has type map,

\{\text{(type\textsubscript{0}, disp\textsubscript{0})}, \ldots, \text{(type\textsubscript{n-1}, disp\textsubscript{n-1})}\},

with extent ex. Let bl be the blocklength. The newly created datatype has a type
map with count · bl · n entries:

\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1}),

(type_0, disp_0 + ex), \ldots, (type_{n-1}, disp_{n-1} + ex), \ldots,

(type_0, disp_0 + (bl - 1) · ex), \ldots, (type_{n-1}, disp_{n-1} + (bl - 1) · ex),

(type_0, disp_0 + stride · ex), \ldots, (type_{n-1}, disp_{n-1} + stride · ex), \ldots,

(type_0, disp_0 + (stride + bl - 1) · ex), \ldots,

(type_{n-1}, disp_{n-1} + (stride + bl - 1) · ex), \ldots,

(type_0, disp_0 + stride · (count - 1) · ex), \ldots,

(type_{n-1}, disp_{n-1} + stride · (count - 1) · ex), \ldots,

(type_0, disp_0 + (stride · (count - 1) + bl - 1) · ex), \ldots,

(type_{n-1}, disp_{n-1} + (stride · (count - 1) + bl - 1) · ex)\}.

A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), n arbitrary.

Mvvector The function MPI_TYPE_MVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Section 3.12.7. (H stands for "heterogeneous").

MPI_TYPE_MVECTOR( count, blocklength, stride, oldtype, newtype)

| IN   | count            | number of blocks (nonnegative integer) |
| IN   | blocklength      | number of elements in each block (nonnegative integer) |
| IN   | stride           | number of bytes between start of each block (integer) |
| IN   | oldtype          | old datatype (handle)                       |
| OUT  | newtype          | new datatype (handle)                      |

int MPI_Type_mvector(int count, int blocklength, MPI_Aint stride,
                     MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)

INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

Assume that OLDTYPE has type map,

\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},

with extent ex. Let bl be the blocklength. The newly created datatype has a type map with \(\text{count} \cdot \text{bl} \cdot n\) entries:

\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},

\{(type_0, disp_0 + ex), \ldots, (type_{n-1}, disp_{n-1} + ex)\}, \ldots,

\{(type_0, disp_0 + (bl - 1) \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex)\}, \ldots,

\{(type_0, disp_0 + \text{stride}), \ldots, (type_{n-1}, disp_{n-1} + \text{stride})\}, \ldots,

\{(type_0, disp_0 + \text{stride} + (bl - 1) \cdot ex), \ldots, \}

\{(type_{n-1}, disp_{n-1} + \text{stride} + (bl - 1) \cdot ex), \ldots, \}

\{(type_0, disp_0 + \text{stride} \cdot (\text{count} - 1)), \ldots, \}

\{(type_{n-1}, disp_{n-1} + \text{stride} \cdot (\text{count} - 1)), \ldots, \}

\{(type_0, disp_0 + \text{stride} \cdot (\text{count} - 1) + (bl - 1) \cdot ex), \ldots, \}

\{(type_{n-1}, disp_{n-1} + \text{stride} \cdot (\text{count} - 1) + (bl - 1) \cdot ex)\}.

Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

MPI_TYPE_INDEXED( count, array_of_blocklengths, array_of_displacements, OLDTYPE, NEWTYPE)

IN count number of blocks – also number of entries in array_of_displacements and array_of_blocklengths (nonnegative integer)

IN array_of_blocklengths number of elements per block (array of nonnegative integers)
IN        array_of_displacements  displacement for each block, in multiples of old-type extent (array of integer)
IN        oldtype                old datatype (handle)
OUT       newtype                new datatype (handle)

int MPI_Type_indexed(int count, int *array_of_blocklengths,
                      int *array_of_displacements, MPI_Datatype oldtype,
                      MPI_Datatype *newtype)

MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
                 OLDTYPE, NEWTYPE, IERROR)

INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
        OLDTYPE, NEWTYPE, IERROR

Example 3.22 Let oldtype have type map \{(double, 0), (char, 8)\}, with extent 16. Let \(B = (3, 1)\) and let \(D = (4, 0)\). A call to MPI_TYPE_INDEXED(2, B, D, oldtype, newtype) returns a datatype with type map,

\[
\{(double, 64), (char, 72), (double, 80), (char, 88), (double, 96),
  (char, 104), (double, 0), (char, 8)\}.
\]

That is, three copies of the old type starting at displacement 64, and one copy starting at displacement 0.

In general, assume that oldtype has type map,

\[
\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},
\]

with extent \(ex\). Let \(B\) be the array_of_blocklength argument and \(D\) be the array_of_displacements argument. The newly created datatype has \(n \cdot \sum_{i=0}^{count-1} B[i]\) entries:

\[
\{(type_0, disp_0 + D[0] \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + D[0] \cdot ex), \ldots,
  (type_0, disp_0 + (D[0] + B[0] - 1) \cdot ex), \ldots,
  (type_{n-1}, disp_{n-1} + (D[0] + B[0] - 1) \cdot ex), \ldots,
  (type_0, disp_0 + D[count - 1] \cdot ex), \ldots,
  (type_{n-1}, disp_{n-1} + D[count - 1] \cdot ex), \ldots,
  (type_0, disp_0 + (D[count - 1] + B[count - 1] - 1) \cdot ex), \ldots,
  (type_{n-1}, disp_{n-1} + (D[count - 1] + B[count - 1] - 1) \cdot ex)\}.
\]
A call to MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent to a call to MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) where
\[ D[j] = j \cdot \text{stride}, \quad j = 0, \ldots, \text{count} - 1, \]
and
\[ B[j] = \text{blocklength}, \quad j = 0, \ldots, \text{count} - 1. \]

Hindexed The function MPI_TYPE_INDEXED is identical to MPI_TYPE_INDEXED, except that block displacements in array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.

**MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)**

- **IN** count number of blocks—also number of entries in array_of_displacements and array_of_blocklengths (integer)
- **IN** array_of_blocklengths number of elements in each block (array of non-negative integers)
- **IN** array_of_displacements byte displacement of each block (array of integer)
- **IN** oldtype old datatype (handle)
- **OUT** newtype new datatype (handle)

```c
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
            MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
            MPI_Datatype *newtype)
```

**MPI_TYPE_INDEXED**

```c
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
                 OLDTYPE, NEWTYPE, IERROR)
```

INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
OLDTYPE, NEWTYPE, IERROR

Assume that oldtype has type map,
\[ \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}, \]
with extent \(ex\). Let \(B\) be the array_of_blocklength argument and \(D\) be the array_of_displacements argument. The newly created datatype has a type map with \(n \cdot \sum_{i=0}^{\text{count}-1} B[i]\) entries:
\[ \{(type_0, disp_0 + D[0]), \ldots, (type_{n-1}, disp_{n-1} + D[0]), \ldots, \]
\[ (type_0, disp_0 + D[0] + (B[0] - 1) \cdot ex), \ldots, \]
```
\[(\text{type}_{n-1}, \text{disp}_{n-1} + D[0] + (B[0] - 1) \cdot ex), \ldots,\]
\[(\text{type}_0, \text{disp}_0 + D[\text{count} - 1]), \ldots, (\text{type}_{x-1}, \text{disp}_{x-1} + D[\text{count} - 1]), \ldots,\]
\[(\text{type}_0, \text{disp}_0 + D[\text{count} - 1] + (B[\text{count} - 1] - 1) \cdot ex), \ldots,\]
\[(\text{type}_{n-1}, \text{disp}_{n-1} + D[\text{count} - 1] + (B[\text{count} - 1] - 1) \cdot ex)\}.

Struct MPI_TYPE_STRUCT is the most general type constructor. It further generalizes the previous one in that it allows each block to consist of replications of different datatypes.

MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN count number of blocks (integer) – also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths

IN array_of_blocklengths number of elements in each block (array of integer)

IN array_of_displacements byte displacement of each block (array of integer)

IN array_of_types type of elements in each block (array of handles to datatype objects)

OUT newtype new datatype (handle)

int MPI_Type_struct(int count, int *array_of_blocklengths,
                      MPI_Aint *array_of_displacements,
                      MPI_Datatype *array_of_types, MPI_Datatype *newtype)

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
                     ARRAY_OF_TYPES, NEWTYPE, IERROR)

INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
ARRAY_OF_TYPES(*), NEWTYPE, IERROR

Example 3.23 Let type1 have type map,
\{((\text{double}, 0), (\text{char}, 8))\},
with extent 16. Let B = (2, 1, 3), D = (0, 16, 26), and T = (MPI_FLOAT, type1, MPI_CHAR). Then a call to MPI_TYPE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,
\{((\text{float}, 0), (\text{float}, 4), (\text{double}, 16), (\text{char}, 24), (\text{char}, 26),
(\text{char}, 27), (\text{char}, 28))\}.
That is, two copies of MPI.FLOAT starting at 0, followed by one copy of type 1 starting at 16, followed by three copies of MPI.CHAR, starting at 26. (We assume that a float occupies four bytes.)

In general, let \( T \) be the \texttt{array.of.types} argument, where \( T[i] \) is a handle to,

\[
\textit{typemap}_i = \{(\textit{type}_0^i, \textit{disp}_0^i), \ldots, (\textit{type}_{n-1}^i, \textit{disp}_{n-1}^i)\},
\]

with extent \( e_0 \). Let \( B \) be the \texttt{array.of.blocklength} argument and \( D \) be the \texttt{array.of.displacements} argument. Let \( c \) be the count argument. Then the newly created datatype has a type map with \( \sum_{i=0}^{c-1} B[i] \cdot n_i \) entries:

\[
\{(\textit{type}_0^0, \textit{disp}_0^0 + D[0]), \ldots, (\textit{type}_{n_0}^0, \textit{disp}_{n_0}^0 + D[0]), \ldots, \\
(\textit{type}_0^c, \textit{disp}_0^c + D[0] + (B[0] - 1) \cdot e_0), \ldots, \\
(\textit{type}_{n_0}^c, \textit{disp}_{n_0}^c + D[0] + (B[0] - 1) \cdot e_0), \ldots, \\
(\textit{type}_0^{c-1}, \textit{disp}_0^{c-1} + D[c - 1]), \ldots, (\textit{type}_{n_{c-1}}^{c-1}, \textit{disp}_{n_{c-1}}^{c-1} + D[c - 1]), \ldots, \\
(\textit{type}_0^{c-1}, \textit{disp}_0^{c-1} + D[c - 1] + (B[c] - 1) \cdot e_{c-1}), \ldots, \\
(\textit{type}_{n_{c-1}}^{c-1}, \textit{disp}_{n_{c-1}}^{c-1} + D[c - 1] + (B[c] - 1) \cdot e_{c-1})\}. 
\]

A call to MPI\_TYPE\_HINDEXED( count, B, D, oldtype, newtype) is equivalent to a call to MPI\_TYPE\_STRUCT( count, B, D, T, newtype), where each entry of \( T \) is equal to oldtype.

### 3.12.2 ADDRESS AND EXTENT FUNCTIONS

The displacements in a general datatype are relative to some initial buffer address. **Absolute addresses** can be substituted for these displacements: we treat them as displacements relative to “address zero,” the start of the address space. This initial address zero is indicated by the constant MPI\_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI\_BOTTOM.

The address of a location in memory can be found by invoking the function MPI\_ADDRESS.

**MPI\_ADDRESS** (location, address)

- **IN** location: location in caller memory (choice)
- **OUT** address: address of location (integer)
int MPI_Address(void* location, MPI_Aint *address)

MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
<type> LOCATION(*)
INTEGER ADDRESS, IERROR

Returns the (byte) address of location.

Example 3.24 Using MPI_ADDRESS for an array.

REAL A(100,100)
INTEGER I1, I2, DIFF
CALL MPI_ADDRESS(A(1,1), I1, IERROR)
CALL MPI_ADDRESS(A(10,10), I2, IERROR)
DIFF = I2 - I1
! The value of DIFF is 909*sizeof(real); the values of I1 and I2 are
! implementation dependent.

Advice to users. C users may be tempted to avoid the usage of MPI_ADDRESS and rely on the availability of the address operator &. Note, however, that & cast-expression is a pointer, not an address. ANSI C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at—although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI_ADDRESS to “reference” C variables guarantees portability to such machines as well. (End of advice to users.)

The following auxiliary functions provide useful information on derived datatypes.

MPI_TYPE_EXTENT(datatype, extent)

IN  datatype                  datatype (handle)
OUT extent                    datatype extent (integer)

int MPI_Type_extent(MPI_Datatype datatype, int MPI_Aint *extent)

MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR

Returns the extent of a datatype, where extent is as defined in Eq. 3.1 on page 233.
MPI_TYPE_SIZE(datatype, size)

IN    datatype   datatype (handle)
OUT   size       datatype size (integer)

int MPI_Type_size(MPI_Datatype datatype, int MPI_Aint *size)

MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)

INTEGER DATATYPE, SIZE, IERROR

MPI_TYPE_SIZE returns the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity.

MPI_TYPE_COUNT(datatype, count)

IN    datatype   datatype (handle)
OUT   count      datatype count (integer)

int MPI_Type_count(MPI_Datatype datatype, int *count)

MPI_TYPE_COUNT(DATATYPE, COUNT, IERROR)

INTEGER DATATYPE, COUNT, IERROR

Returns the number of "top-level" entries in the datatype.