

D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

January 28, 2015

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

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

Chapter 4

Point-to-Point Communication

4.1 Introduction

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are *send* and *receive*. Their use is illustrated in the example below.

```
#include "mpi.h"
int main( int argc, char *argv[])
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0) /* code for process zero */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else if (myrank == 1) /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
    return 0;
}
```

In this example, process zero (`myrank = 0`) sends a message to process one using the *send* operation `MPI_SEND`. The operation specifies a *send buffer* in the sender memory from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable *message* in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition, the send operation associates an *envelope* with the message. This envelope specifies the

message destination and contains distinguishing information that can be used by the *receive* operation to select a particular message. The last three parameters of the send operation, along with the rank of the sender, specify the envelope for the message sent. Process one (`myrank = 1`) receives this message with the *receive* operation `MPI_RECV`. The message to be received is selected according to the value of its envelope, and the message data is stored into the *receive buffer*. In the example above, the receive buffer consists of the storage containing the string `message` in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.

The next sections describe the blocking send and receive operations. We discuss send, receive, blocking communication semantics, type matching requirements, type conversion in heterogeneous environments, and more general communication modes. Nonblocking communication is addressed next, followed by probing and canceling a message, channel-like constructs and send-receive operations, ending with a description of the “dummy” process, `MPI_PROC_NULL`.

4.2 Blocking Send and Receive Operations

4.2.1 Blocking Send

The syntax of the blocking send operation is given below.

`MPI_SEND(buf, count, datatype, dest, tag, comm)`

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

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

The blocking semantics of this call are described in Section 4.4.

4.2.2 Message Data

The send buffer specified by the `MPI_SEND` operation consists of `count` successive entries of the type indicated by `datatype`, starting with the entry at address `buf`. Note that we specify the message length in terms of number of *elements*, not number of *bytes*. The former is machine independent and closer to the application level.

The data part of the message consists of a sequence of `count` values, each of the type indicated by `datatype`. `count` may be zero, in which case the data part of the message is empty. The basic datatypes that can be specified for message data values correspond to the basic datatypes of the host language. Possible values of this argument for Fortran and the corresponding Fortran types are listed in Table 4.1.

MPI datatype	Fortran datatype
<code>MPI_INTEGER</code>	<code>INTEGER</code>
<code>MPI_REAL</code>	<code>REAL</code>
<code>MPI_DOUBLE_PRECISION</code>	<code>DOUBLE PRECISION</code>
<code>MPI_COMPLEX</code>	<code>COMPLEX</code>
<code>MPI_LOGICAL</code>	<code>LOGICAL</code>
<code>MPI_CHARACTER</code>	<code>CHARACTER(1)</code>
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	

Table 4.1: Predefined MPI datatypes corresponding to Fortran datatypes

Possible values for this argument for C and the corresponding C types are listed in Table 4.2.

The datatypes `MPI_BYTE` and `MPI_PACKED` do not correspond to a Fortran or C datatype. A value of type `MPI_BYTE` consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type `MPI_PACKED` is explained in Section 4.2.

MPI requires support of these datatypes, which match the basic datatypes of Fortran and ISO C. Additional MPI datatypes should be provided if the host language has additional data types: `MPI_DOUBLE_COMPLEX` for double precision complex in Fortran declared to be of type `DOUBLE COMPLEX`; `MPI_REAL2`, `MPI_REAL4`, and `MPI_REAL8` for Fortran reals, declared to be of type `REAL*2`, `REAL*4` and `REAL*8`, respectively; `MPI_INTEGER1`, `MPI_INTEGER2`, and `MPI_INTEGER4` for Fortran integers, declared to be of type `INTEGER*1`, `INTEGER*2`, and `INTEGER*4`, respectively; etc.

Rationale. One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit argument. The need for such datatype information will become clear in Section 4.3.2. (*End of rationale.*)

The datatypes `MPI_AINT`, `MPI_OFFSET`, and `MPI_COUNT` correspond to the MPI-defined C types `MPI_Aint`, `MPI_Offset`, and `MPI_Count` and their Fortran equivalents

MPI datatype	C datatype
MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char (treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char (treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t (defined in <stddef.h>) (treated as printable character)
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

Table 4.2: Predefined MPI datatypes corresponding to C datatypes

INTEGER (KIND=MPI_ADDRESS_KIND), INTEGER (KIND=MPI_OFFSET_KIND), and INTEGER (KIND=MPI_COUNT_KIND). This is described in Table 4.3. All predefined datatype handles are available in all language bindings. See Sections 17.2.6 and ?? on page 652 and ?? for information on interlanguage communication with these types.

If there is an accompanying C++ compiler then the datatypes in Table 4.4 are also supported in C and Fortran.

MPI datatype	C datatype	Fortran datatype
MPI_AINT	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
MPI_OFFSET	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)
MPI_COUNT	MPI_Count	INTEGER (KIND=MPI_COUNT_KIND)

Table 4.3: Predefined MPI datatypes corresponding to both C and Fortran datatypes

MPI datatype	C++ datatype
MPI_CXX_BOOL	bool
MPI_CXX_FLOAT_COMPLEX	std::complex<float>
MPI_CXX_DOUBLE_COMPLEX	std::complex<double>
MPI_CXX_LONG_DOUBLE_COMPLEX	std::complex<long double>

Table 4.4: Predefined MPI datatypes corresponding to C++ datatypes

4.2.3 Message Envelope

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the *message envelope*. These fields are

source
destination
tag
communicator

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation.

The message destination is specified by the **dest** argument.

The integer-valued message tag is specified by the **tag** argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is $0, \dots, \text{UB}$, where the value of **UB** is implementation dependent. It can be found by querying the value of the attribute **MPI_TAG_UB**, as described in Chapter 8. MPI requires that **UB** be no less than 32767.

The **comm** argument specifies the *communicator* that is used for the send operation. Communicators are explained in Chapter 6; below is a brief summary of their usage.

A communicator specifies the communication context for a communication operation. Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

The communicator also specifies the set of processes that share this communication context. This *process group* is ordered and processes are identified by their rank within this group. Thus, the range of valid values for **dest** is $0, \dots, n - 1 \cup \{\text{MPI_PROC_NULL}\}$, where n is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 6.)

A predefined communicator **MPI_COMM_WORLD** is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of **MPI_COMM_WORLD**.

Advice to users. Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing communication libraries, need only use the predefined variable `MPI_COMM_WORLD` as the `comm` argument. This will allow communication with all the processes available at initialization time.

Users may define new communicators, as explained in Chapter 6. Communicators provide an important encapsulation mechanism for libraries and modules. They allow modules to have their own disjoint communication universe and their own process numbering scheme. (*End of advice to users.*)

Advice to implementors. The message envelope would normally be encoded by a fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (*End of advice to implementors.*)

4.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

`MPI_RECV (buf, count, datatype, source, tag, comm, status)`

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source or <code>MPI_ANY_SOURCE</code> (integer)
IN	tag	message tag or <code>MPI_ANY_TAG</code> (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
            int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
```

```
TYPE(*), DIMENSION(..) :: buf
INTEGER, INTENT(IN) :: count, source, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
```

```
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
IERROR
```

The blocking semantics of this call are described in Section 4.4.

The receive buffer consists of the storage containing `count` consecutive elements of the type specified by `datatype`, starting at address `buf`. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

If a message that is shorter than the receive buffer arrives, then only those locations corresponding to the (shorter) message are modified.

Advice to users. The `MPI_PROBE` function described in Section 4.8 can be used to receive messages of unknown length. (*End of advice to users.*)

Advice to implementors. Even though no specific behavior is mandated by MPI for erroneous programs, the recommended handling of overflow situations is to return in `status` information about the source and tag of the incoming message. The receive operation will return an error code. A quality implementation will also ensure that no memory that is outside the receive buffer will ever be overwritten.

In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow for some optimizations but this is not allowed. The implementation must be ready to end a copy into the receiver memory exactly at the end of the receive buffer, even if it is an odd address. (*End of advice to implementors.*)

The selection of a message by a receive operation is governed by the value of the message envelope. A message can be received by a receive operation if its envelope matches the `source`, `tag` and `comm` values specified by the receive operation. The receiver may specify a wildcard `MPI_ANY_SOURCE` value for `source`, and/or a wildcard `MPI_ANY_TAG` value for `tag`, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value for `comm`. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching communicator, has matching source unless `source=MPI_ANY_SOURCE` in the pattern, and has a matching tag unless `tag=MPI_ANY_TAG` in the pattern.

The message tag is specified by the `tag` argument of the receive operation. The argument `source`, if different from `MPI_ANY_SOURCE`, is specified as a rank within the process group associated with that same communicator (remote process group, for intercommunicators). Thus, the range of valid values for the `source` argument is $\{0, \dots, n - 1\} \cup \{\text{MPI_ANY_SOURCE}\}$, where n is the number of processes in this group.

Note the asymmetry between send and receive operations: A receive operation may accept messages from an arbitrary sender, on the other hand, a send operation must specify a unique receiver. This matches a “push” communication mechanism, where data transfer is effected by the sender (rather than a “pull” mechanism, where data transfer is effected by the receiver).

Source = destination is allowed, that is, a process can send a message to itself. (However, it is unsafe to do so with the blocking send and receive operations described above, since this may lead to deadlock. See Section 4.5.)

Advice to implementors. Message context and other communicator information can be implemented as an additional tag field. It differs from the regular message tag in that wild card matching is not allowed on this field, and that value setting for

this field is controlled by communicator manipulation functions. (*End of advice to implementors.*)

The use of `dest` or `source=MPI_PROC_NULL` to define a “dummy” destination or source in any send or receive call is described in Section 4.11.

4.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 4.7.5), a distinct error code may need to be returned for each request. The information is returned by the `status` argument of `MPI_RECV`. The type of `status` is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, `status` is a structure that contains three fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`; the structure may contain additional fields. Thus, `status.MPI_SOURCE`, `status.MPI_TAG` and `status.MPI_ERROR` contain the source, tag, and error code, respectively, of the received message.

In Fortran with `USE mpi` or `INCLUDE 'mpif.h'`, `status` is an array of `INTEGER`s of size `MPI_STATUS_SIZE`. The constants `MPI_SOURCE`, `MPI_TAG` and `MPI_ERROR` are the indices of the entries that store the source, tag and error fields. Thus, `status(MPI_SOURCE)`, `status(MPI_TAG)` and `status(MPI_ERROR)` contain, respectively, the source, tag and error code of the received message.

With Fortran `USE mpi_f08`, `status` is defined as the Fortran `BIND(C)` derived type `TYPE(MPI_Status)` containing three public `INTEGER` fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`. `TYPE(MPI_Status)` may contain additional, implementation-specific fields. Thus, `status%MPI_SOURCE`, `status%MPI_TAG` and `status%MPI_ERROR` contain the source, tag, and error code of a received message respectively. Additionally, within both the `mpi` and the `mpi_f08` modules, the constants `MPI_STATUS_SIZE`, `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`, and `TYPE(MPI_Status)` are defined to allow conversion between both status representations. Conversion routines are provided in Section 17.2.5.

Rationale. The Fortran `TYPE(MPI_Status)` is defined as a `BIND(C)` derived type so that it can be used at any location where the status integer array representation can be used, e.g., in user defined common blocks. (*End of rationale.*)

Rationale. It is allowed to have the same name (e.g., `MPI_SOURCE`) defined as a constant (e.g., Fortran parameter) and as a field of a derived type. (*End of rationale.*)

In general, message-passing calls do not modify the value of the error code field of status variables. This field may be updated only by the functions in Section 4.7.5 which return multiple statuses. The field is updated if and only if such function returns with an error code of `MPI_ERR_IN_STATUS`.

Rationale. The error field in status is not needed for calls that return only one status, such as `MPI_WAIT`, since that would only duplicate the information returned by the function itself. The current design avoids the additional overhead of setting it, in such cases. The field is needed for calls that return multiple statuses, since each request may have had a different failure. (*End of rationale.*)

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to `MPI_GET_COUNT` is required to “decode” this information.

`MPI_GET_COUNT(status, datatype, count)`

IN	status	return status of receive operation (Status)
IN	datatype	datatype of each receive buffer entry (handle)
OUT	count	number of received entries (integer)

```
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,
                 int *count)
```

```
MPI_Get_count(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) :: status
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(OUT) :: count
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

Returns the number of entries received. (Again, we count *entries*, each of type *datatype*, not *bytes*.) The *datatype* argument should match the argument provided by the receive call that set the *status* variable. If the number of entries received exceeds the limits of the *count* parameter, then `MPI_GET_COUNT` sets the value of *count* to `MPI_UNDEFINED`. There are other situations where the value of *count* can be set to `MPI_UNDEFINED`; see Section 4.1.11.

Rationale. Some message-passing libraries use `INOUT count`, `tag` and `source` arguments, thus using them both to specify the selection criteria for incoming messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with `INOUT` argument (e.g., using the `MPI_ANY_TAG` constant as the tag in a receive). Some libraries use calls that refer implicitly to the “last message received.” This is not thread safe.

The *datatype* argument is passed to `MPI_GET_COUNT` so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to `MPI_PROBE` or `MPI_IProbe`. With a status from `MPI_PROBE` or `MPI_IProbe`, the same datatypes are allowed as in a call to `MPI_RECV` to receive this message. (*End of rationale.*)

The value returned as the *count* argument of `MPI_GET_COUNT` for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, `MPI_UNDEFINED` is returned.

Rationale. Zero-length datatypes may be created in a number of cases. An important case is `MPI_TYPE_CREATE_DARRAY`, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style

will not check for this special case and may want to use `MPI_GET_COUNT` to check the status. (*End of rationale.*)

Advice to users. The buffer size required for the receive can be affected by data conversions and by the stride of the receive datatype. In most cases, the safest approach is to use the same datatype with `MPI_GET_COUNT` and the receive. (*End of advice to users.*)

All send and receive operations use the `buf`, `count`, `datatype`, `source`, `dest`, `tag`, `comm`, and `status` arguments in the same way as the blocking `MPI_SEND` and `MPI_RECV` operations described in this section.

4.2.6 Passing `MPI_STATUS_IGNORE` for Status

Every call to `MPI_RECV` includes a `status` argument, wherein the system can return details about the message received. There are also a number of other MPI calls where `status` is returned. An object of type `MPI_Status` is not an MPI opaque object; its structure is declared in `mpi.h` and `mpif.h`, and it exists in the user's program. In many cases, application programs are constructed so that it is unnecessary for them to examine the `status` fields. In these cases, it is a waste for the user to allocate a status object, and it is particularly wasteful for the MPI implementation to fill in fields in this object.

To cope with this problem, there are two predefined constants, `MPI_STATUS_IGNORE` and `MPI_STATUSES_IGNORE`, which when passed to a receive, probe, wait, or test function, inform the implementation that the status fields are not to be filled in. Note that `MPI_STATUS_IGNORE` is not a special type of `MPI_Status` object; rather, it is a special value for the argument. In C one would expect it to be `NULL`, not the address of a special `MPI_Status`.

`MPI_STATUS_IGNORE`, and the array version `MPI_STATUSES_IGNORE`, can be used everywhere a status argument is passed to a receive, wait, or test function. `MPI_STATUS_IGNORE` cannot be used when status is an IN argument. Note that in Fortran `MPI_STATUS_IGNORE` and `MPI_STATUSES_IGNORE` are objects like `MPI_BOTTOM` (not usable for initialization or assignment). See Section 2.5.4.

In general, this optimization can apply to all functions for which `status` or an array of `statuses` is an OUT argument. Note that this converts `status` into an INOUT argument. The functions that can be passed `MPI_STATUS_IGNORE` are all the various forms of `MPI_RECV`, `MPI_PROBE`, `MPI_TEST`, and `MPI_WAIT`, as well as `MPI_REQUEST_GET_STATUS`. When an array is passed, as in the `MPI_{TEST|WAIT}{ALL|SOME}` functions, a separate constant, `MPI_STATUSES_IGNORE`, is passed for the array argument. It is possible for an MPI function to return `MPI_ERR_IN_STATUS` even when `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE` has been passed to that function.

`MPI_STATUS_IGNORE` and `MPI_STATUSES_IGNORE` are not required to have the same values in C and Fortran.

It is not allowed to have some of the statuses in an array of statuses for `MPI_{TEST|WAIT}{ALL|SOME}` functions set to `MPI_STATUS_IGNORE`; one either specifies ignoring *all* of the statuses in such a call with `MPI_STATUSES_IGNORE`, or *none* of them by passing normal statuses in all positions in the array of statuses.

4.3 Data Type Matching and Data Conversion

4.3.1 Type Matching Rules

One can think of message transfer as consisting of the following three phases.

1. Data is pulled out of the send buffer and a message is assembled.
2. A message is transferred from sender to receiver.
3. Data is pulled from the incoming message and disassembled into the receive buffer.

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous.

To define type matching more precisely, we need to deal with two issues: matching of types of the host language with types specified in communication operations; and matching of types at sender and receiver.

The types of a send and receive match (phase two) if both operations use identical names. That is, `MPI_INTEGER` matches `MPI_INTEGER`, `MPI_REAL` matches `MPI_REAL`, and so on. There is one exception to this rule, discussed in Section 4.2: the type `MPI_PACKED` can match any other type.

The type of a variable in a host program matches the type specified in the communication operation if the datatype name used by that operation corresponds to the basic type of the host program variable. For example, an entry with type name `MPI_INTEGER` matches a Fortran variable of type `INTEGER`. A table giving this correspondence for Fortran and C appears in Section 4.2.2. There are two exceptions to this last rule: an entry with type name `MPI_BYTE` or `MPI_PACKED` can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte. The type `MPI_PACKED` is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 4.2. The type `MPI_BYTE` allows one to transfer the binary value of a byte in memory unchanged.

To summarize, the type matching rules fall into the three categories below.

- Communication of typed values (e.g., with datatype different from `MPI_BYTE`), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.
- Communication of untyped values (e.g., of datatype `MPI_BYTE`), where both sender and receiver use the datatype `MPI_BYTE`. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.
- Communication involving packed data, where `MPI_PACKED` is used.

The following examples illustrate the first two cases.

Example 4.1 Sender and receiver specify matching types.

```

1  CALL MPI_COMM_RANK(comm, rank, ierr)
2  IF (rank.EQ.0) THEN
3      CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
4  ELSE IF (rank.EQ.1) THEN
5      CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
6  END IF

```

This code is correct if both **a** and **b** are real arrays of size ≥ 10 . (In Fortran, it might be correct to use this code even if **a** or **b** have size < 10 : e.g., when **a**(1) can be equivalenced to an array with ten reals.)

Example 4.2 Sender and receiver do not specify matching types.

```

11 CALL MPI_COMM_RANK(comm, rank, ierr)
12 IF (rank.EQ.0) THEN
13     CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
14 ELSE IF (rank.EQ.1) THEN
15     CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
16 END IF

```

This code is erroneous, since sender and receiver do not provide matching datatype arguments.

Example 4.3 Sender and receiver specify communication of untyped values.

```

24 CALL MPI_COMM_RANK(comm, rank, ierr)
25 IF (rank.EQ.0) THEN
26     CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
27 ELSE IF (rank.EQ.1) THEN
28     CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
29 END IF

```

This code is correct, irrespective of the type and size of **a** and **b** (unless this results in an out of bounds memory access).

Advice to users. If a buffer of type `MPI_BYTE` is passed as an argument to `MPI_SEND`, then MPI will send the data stored at contiguous locations, starting from the address indicated by the `buf` argument. This may have unexpected results when the data layout is not as a casual user would expect it to be. For example, some Fortran compilers implement variables of type `CHARACTER` as a structure that contains the character length and a pointer to the actual string. In such an environment, sending and receiving a Fortran `CHARACTER` variable using the `MPI_BYTE` type will not have the anticipated result of transferring the character string. For this reason, the user is advised to use typed communications whenever possible. (*End of advice to users.*)

Type `MPI_CHARACTER`

The type `MPI_CHARACTER` matches one character of a Fortran variable of type `CHARACTER`, rather than the entire character string stored in the variable. Fortran variables of type `CHARACTER` or substrings are transferred as if they were arrays of characters. This is illustrated in the example below.

Example 4.4

Transfer of Fortran CHARACTERS.

```

CHARACTER*10 a
CHARACTER*10 b

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
END IF

```

The last five characters of string `b` at process 1 are replaced by the first five characters of string `a` at process 0.

Rationale. The alternative choice would be for `MPI_CHARACTER` to match a character of arbitrary length. This runs into problems.

A Fortran character variable is a constant length string, with no special termination symbol. There is no fixed convention on how to represent characters, and how to store their length. Some compilers pass a character argument to a routine as a pair of arguments, one holding the address of the string and the other holding the length of string. Consider the case of an MPI communication call that is passed a communication buffer with type defined by a derived datatype (Section 4.1). If this communicator buffer contains variables of type `CHARACTER` then the information on their length will not be passed to the MPI routine.

This problem forces us to provide explicit information on character length with the MPI call. One could add a length parameter to the type `MPI_CHARACTER`, but this does not add much convenience and the same functionality can be achieved by defining a suitable derived datatype. (*End of rationale.*)

Advice to implementors. Some compilers pass Fortran `CHARACTER` arguments as a structure with a length and a pointer to the actual string. In such an environment, the MPI call needs to dereference the pointer in order to reach the string. (*End of advice to implementors.*)

4.3.2 Data Conversion

One of the goals of MPI is to support parallel computations across heterogeneous environments. Communication in a heterogeneous environment may require data conversions. We use the following terminology.

type conversion changes the datatype of a value, e.g., by rounding a `REAL` to an `INTEGER`.

representation conversion changes the binary representation of a value, e.g., from Hex floating point to IEEE floating point.

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a

typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical and character values, and to convert a floating point value to the nearest value that can be represented on the target system.

Overflow and underflow exceptions may occur during floating point conversions. Conversion of integers or characters may also lead to exceptions when a value that can be represented in one system cannot be represented in the other system. An exception occurring during representation conversion results in a failure of the communication. An error occurs either in the send operation, or the receive operation, or both.

If a value sent in a message is untyped (i.e., of type `MPI_BYTE`), then the binary representation of the byte stored at the receiver is identical to the binary representation of the byte loaded at the sender. This holds true, whether sender and receiver run in the same or in distinct environments. No representation conversion is required. (Note that representation conversion may occur when values of type `MPI_CHARACTER` or `MPI_CHAR` are transferred, for example, from an EBCDIC encoding to an ASCII encoding.)

No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment.

Consider the three examples, 4.1–4.3. The first program is correct, assuming that `a` and `b` are `REAL` arrays of size ≥ 10 . If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

The second program is erroneous, and its behavior is undefined.

The third program is correct. The exact same sequence of forty bytes that were loaded from the send buffer will be stored in the receive buffer, even if sender and receiver run in a different environment. The message sent has exactly the same length (in bytes) and the same binary representation as the message received. If `a` and `b` are of different types, or if they are of the same type but different data representations are used, then the bits stored in the receive buffer may encode values that are different from the values they encoded in the send buffer.

Data representation conversion also applies to the envelope of a message: source, destination and tag are all integers that may need to be converted.

Advice to implementors. The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion.

Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be particularly useful in a slower but safer debug mode. (*End of advice to implementors.*)

MPI requires support for inter-language communication, i.e., if messages are sent by a C or C++ process and received by a Fortran process, or vice-versa. The behavior is defined in Section 17.2.

4.4 Communication Modes

The send call described in Section 4.2.1 is *blocking*: it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

Message buffering decouples the send and receive operations. A blocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. On the other hand, message buffering can be expensive, as it entails additional memory-to-memory copying, and it requires the allocation of memory for buffering. MPI offers the choice of several communication modes that allow one to control the choice of the communication protocol.

The send call described in Section 4.2.1 uses the *standard* communication mode. In this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

Thus, a send in standard mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. The standard mode send is *non-local*: successful completion of the send operation may depend on the occurrence of a matching receive.

Rationale. The reluctance of MPI to mandate whether standard sends are buffering or not stems from the desire to achieve portable programs. Since any system will run out of buffer resources as message sizes are increased, and some implementations may want to provide little buffering, MPI takes the position that correct (and therefore, portable) programs do not rely on system buffering in standard mode. Buffering may improve the performance of a correct program, but it doesn't affect the result of the program. If the user wishes to guarantee a certain amount of buffering, the user-provided buffer system of Section 4.6 should be used, along with the buffered-mode send. (*End of rationale.*)

There are three additional communication modes.

A *buffered* mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is *local*, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. An error will occur if there is insufficient buffer space. The amount of available buffer space is controlled by the user — see Section 4.6. Buffer allocation by the user may be required for the buffered mode to be effective.

A send that uses the *synchronous* mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its

execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is *non-local*.

A send that uses the *ready* communication mode may be started *only* if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

Three additional send functions are provided for the three additional communication modes. The communication mode is indicated by a one letter prefix: B for buffered, S for synchronous, and R for ready.

MPI_BSEND (buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Bsend(const void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)
```

```
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
```

```
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

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

Send in buffered mode.

MPI_SSEND (buf, count, datatype, dest, tag, comm)	1
IN buf initial address of send buffer (choice)	2
IN count number of elements in send buffer (non-negative integer)	3
	4
	5
IN datatype datatype of each send buffer element (handle)	6
IN dest rank of destination (integer)	7
	8
IN tag message tag (integer)	9
IN comm communicator (handle)	10
	11
int MPI_Ssend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	12
	13
	14
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)	15
TYPE(*), DIMENSION(..), INTENT(IN) :: buf	16
INTEGER, INTENT(IN) :: count, dest, tag	17
TYPE(MPI_Datatype), INTENT(IN) :: datatype	18
TYPE(MPI_Comm), INTENT(IN) :: comm	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	21
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	22
<type> BUF(*)	23
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	24
Send in synchronous mode.	25
	26
	27
MPI_RSEND (buf, count, datatype, dest, tag, comm)	28
IN buf initial address of send buffer (choice)	29
IN count number of elements in send buffer (non-negative integer)	30
	31
	32
IN datatype datatype of each send buffer element (handle)	33
IN dest rank of destination (integer)	34
	35
IN tag message tag (integer)	36
IN comm communicator (handle)	37
	38
int MPI_Rsend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	39
	40
	41
MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)	42
TYPE(*), DIMENSION(..), INTENT(IN) :: buf	43
INTEGER, INTENT(IN) :: count, dest, tag	44
TYPE(MPI_Datatype), INTENT(IN) :: datatype	45
TYPE(MPI_Comm), INTENT(IN) :: comm	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
	48
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	

```

1  <type> BUF(*)
2  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

```

Send in ready mode.

There is only one receive operation, but it matches any of the send modes. The receive operation described in the last section is *blocking*: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

In a multithreaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. In such a case it is the user's responsibility not to modify a communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.

Advice to implementors. Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation.

It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the sender until a matching receive occurs by using the synchronous send mode.

A possible communication protocol for the various communication modes is outlined below.

ready send: The message is sent as soon as possible.

synchronous send: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.

standard send: First protocol may be used for short messages, and second protocol for long messages.

buffered send: The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).

Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.

Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.

A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, users may expect some buffering.

In a multithreaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. (*End of advice to implementors.*)

4.5 Semantics of Point-to-Point Communication

A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.

Order Messages are *non-overtaking*: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard `MPI_ANY_SOURCE` is not used in receives. (Some of the calls described later, such as `MPI_CANCEL` or `MPI_WAITANY`, are additional sources of nondeterminism.)

If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multithreaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

Example 4.5 An example of non-overtaking messages.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by the first send must be received by the first receive, and the message sent by the second send must be received by the second receive.

Progress If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process.

Example 4.6 An example of two, intertwined matching pairs.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
    CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At that point, a matching pair of send and receive operation is enabled, and both operations must complete. Process one next invokes its second receive call, which will be satisfied by the buffered message. Note that process one received the messages in the reverse order they were sent.

Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multithreaded process. Then it is possible that messages that match this receive are repeatedly received, yet the receive is never satisfied, because it is overtaken by other receives posted at this node (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations.

Resource limitations Any pending communication operation consumes system resources that are limited. Errors may occur when lack of resources prevent the execution of an MPI call. A quality implementation will use a (small) fixed amount of resources for each pending send in the ready or synchronous mode and for each pending receive. However, buffer space may be consumed to store messages sent in standard mode, and must be consumed to store messages sent in buffered mode, when no matching receive is available. The amount of space available for buffering will be much smaller than program data memory on many systems. Then, it will be easy to write programs that overrun available buffer space.

MPI allows the user to provide buffer memory for messages sent in the buffered mode. Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI implementation is required to do no worse than implied by this model. This allows users to avoid buffer overflows when they use buffered sends. Buffer allocation and use is described in Section 4.6.

A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signaled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values and sends them to a consumer. Assume that the producer produces new values faster than the consumer can consume them. If buffered sends are used, then a buffer overflow will result. Additional synchronization has to be added to the program so as to prevent this from occurring. If standard sends are used, then the producer will be automatically throttled, as its send operations will block when buffer space is unavailable.

In some situations, a lack of buffer space leads to deadlock situations. This is illustrated by the examples below.

Example 4.7 An exchange of messages.

```

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF

```

This program will succeed even if no buffer space for data is available. The standard send operation can be replaced, in this example, with a synchronous send.

Example 4.8 An errant attempt to exchange messages.

```

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF

```

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second processor is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock. The same holds for any other send mode.

Example 4.9 An exchange that relies on buffering.

```

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF

```

The message sent by each process has to be copied out before the send operation returns and the receive operation starts. For the program to complete, it is necessary that at least one of the two messages sent be buffered. Thus, this program can succeed only if the communication system can buffer at least count words of data.

Advice to users. When standard send operations are used, then a deadlock situation may occur where both processes are blocked because buffer space is not available. The same will certainly happen, if the synchronous mode is used. If the buffered mode is used, and not enough buffer space is available, then the program will not complete either. However, rather than a deadlock situation, we shall have a buffer overflow error.

A program is “safe” if no message buffering is required for the program to complete. One can replace all sends in such program with synchronous sends, and the program will still run correctly. This conservative programming style provides the best portability, since program completion does not depend on the amount of buffer space available or on the communication protocol used.

Many programmers prefer to have more leeway and opt to use the “unsafe” programming style shown in Example 4.9. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that “common practice” programs will not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

Nonblocking message-passing operations, as described in Section 4.7, can be used to avoid the need for buffering outgoing messages. This prevents deadlocks due to lack of buffer space, and improves performance, by allowing overlap of computation and communication, and avoiding the overheads of allocating buffers and copying messages into buffers. (*End of advice to users.*)

4.6 Buffer Allocation and Usage

A user may specify a buffer to be used for buffering messages sent in buffered mode. Buffering is done by the sender.

`MPI_BUFFER_ATTACH(buffer, size)`

IN	buffer	initial buffer address (choice)
IN	size	buffer size, in bytes (non-negative integer)

`int MPI_Buffer_attach(void* buffer, int size)`

`MPI_Buffer_attach(buffer, size, ierror)`

TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer

INTEGER, INTENT(IN) :: size

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

`MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)`

<type> BUFFER(*)

INTEGER SIZE, IERROR

Provides to MPI a buffer in the user’s memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time. In C, `buffer` is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be ‘simply contiguous’ (for ‘simply contiguous,’ see also Section 17.1.12).

```

MPI_BUFFER_DETACH(buffer_addr, size)
    OUT    buffer_addr          initial buffer address (choice)
    OUT    size                 buffer size, in bytes (non-negative integer)

```

```

int MPI_Buffer_detach(void* buffer_addr, int* size)

```

```

MPI_Buffer_detach(buffer_addr, size, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    TYPE(C_PTR), INTENT(OUT) :: buffer_addr
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
    <type> BUFFER_ADDR(*)
    INTEGER SIZE, IERROR

```

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This operation will block until all messages currently in the buffer have been transmitted. Upon return of this function, the user may reuse or deallocate the space taken by the buffer.

Example 4.10 Calls to attach and detach buffers.

```

#define BUFFSIZE 10000
int size;
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach( &buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */

```

Advice to users. Even though the C functions `MPI_Buffer_attach` and `MPI_Buffer_detach` both have a first argument of type `void*`, these arguments are used differently: A pointer to the buffer is passed to `MPI_Buffer_attach`; the address of the pointer is passed to `MPI_Buffer_detach`, so that this call can return the pointer value. In Fortran with the `mpi` module or `mpif.h`, the type of the `buffer_addr` argument is wrongly defined and the argument is therefore unused. In Fortran with the `mpi_f08` module, the address of the buffer is returned as `TYPE(C_PTR)`, see also Example 8.1 about the use of `C_PTR` pointers. (*End of advice to users.*)

Rationale. Both arguments are defined to be of type `void*` (rather than `void*` and `void**`, respectively), so as to avoid complex type casts. E.g., in the last example, `&buff`, which is of type `char**`, can be passed as argument to `MPI_Buffer_detach` without type casting. If the formal parameter had type `void**` then we would need a type cast before and after the call. (*End of rationale.*)

The statements made in this section describe the behavior of MPI for buffered-mode sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is associated with the process.

MPI must provide as much buffering for outgoing messages *as if* outgoing message data were buffered by the sending process, in the specified buffer space, using a circular, contiguous-space allocation policy. We outline below a model implementation that defines this policy. MPI may provide more buffering, and may use a better buffer allocation algorithm than described below. On the other hand, MPI may signal an error whenever the simple buffering allocator described below would run out of space. In particular, if no buffer is explicitly associated with the process, then any buffered send may cause an error.

MPI does not provide mechanisms for querying or controlling buffering done by standard mode sends. It is expected that vendors will provide such information for their implementations.

Rationale. There is a wide spectrum of possible implementations of buffered communication: buffering can be done at sender, at receiver, or both; buffers can be dedicated to one sender-receiver pair, or be shared by all communications; buffering can be done in real or in virtual memory; it can use dedicated memory, or memory shared by other processes; buffer space may be allocated statically or be changed dynamically; etc. It does not seem feasible to provide a portable mechanism for querying or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (*End of rationale.*)

4.6.1 Model Implementation of Buffered Mode

The model implementation uses the packing and unpacking functions described in Section 4.2 and the nonblocking communication functions described in Section 4.7.

We assume that a circular queue of pending message entries (PME) is maintained. Each entry contains a communication request handle that identifies a pending nonblocking send, a pointer to the next entry and the packed message data. The entries are stored in successive locations in the buffer. Free space is available between the queue tail and the queue head.

A buffered send call results in the execution of the following code.

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
- Compute the number, n , of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function `MPI_PACK_SIZE(count, datatype, comm, size)`, with the `count`, `datatype` and `comm` arguments used in the `MPI_BSEND` call, returns an upper bound on the amount of space needed to buffer the message data (see Section 4.2). The MPI constant `MPI_BSEND_OVERHEAD` provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).
- Find the next contiguous empty space of n bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; `MPI_PACK` is used to pack data.

- Post nonblocking send (standard mode) for packed data.
- Return

4.7 Nonblocking Communication

One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use *nonblocking communication*. A nonblocking *send start* call initiates the send operation, but does not complete it. The send start call can return before the message was copied out of the send buffer. A separate *send complete* call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking *receive start call* initiates the receive operation, but does not complete it. The call can return before a message is stored into the receive buffer. A separate *receive complete* call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

Nonblocking send start calls can use the same four modes as blocking sends: *standard*, *buffered*, *synchronous* and *ready*. These carry the same meaning. Sends of all modes, *ready* excepted, can be started whether a matching receive has been posted or not; a nonblocking *ready* send can be started only if a matching receive is posted. In all cases, the send start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in “pathological” cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

The send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode.

If the send mode is *synchronous*, then the send can complete only if a matching receive has started. That is, a receive has been posted, and has been matched with the send. In this case, the send-complete call is non-local. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the sender “knows” the transfer will complete, but before the receiver “knows” the transfer will complete.)

If the send mode is *buffered* then the message must be buffered if there is no pending receive. In this case, the send-complete call is local, and must succeed irrespective of the status of a matching receive.

If the send mode is *standard* then the send-complete call may return before a matching receive is posted, if the message is buffered. On the other hand, the receive-complete may not complete until a matching receive is posted, and the message was copied into the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa.

Advice to users. The completion of a send operation may be delayed, for standard mode, and must be delayed, for synchronous mode, until a matching receive is posted. The use of nonblocking sends in these two cases allows the sender to proceed ahead of the receiver, so that the computation is more tolerant of fluctuations in the speeds of the two processes.

Nonblocking sends in the buffered and ready modes have a more limited impact, e.g., the blocking version of buffered send is capable of completing regardless of when a matching receive call is made. However, separating the start from the completion of these sends still gives some opportunity for optimization within the MPI library. For example, starting a buffered send gives an implementation more flexibility in determining if and how the message is buffered. There are also advantages for both nonblocking buffered and ready modes when data copying can be done concurrently with computation.

The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of nonblocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send. (*End of advice to users.*)

4.7.1 Communication Request Objects

Nonblocking communications use opaque *request* objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

4.7.2 Communication Initiation

We use the same naming conventions as for blocking communication: a prefix of B, S, or R is used for *buffered*, *synchronous* or *ready* mode. In addition a prefix of I (for *immediate*) indicates that the call is nonblocking.

```

MPI_ISEND(buf, count, datatype, dest, tag, comm, request) 1
    IN      buf      initial address of send buffer (choice) 2
    IN      count     number of elements in send buffer (non-negative inte- 3
                        ger) 4
    IN      datatype  datatype of each send buffer element (handle) 5
    IN      dest      rank of destination (integer) 6
    IN      tag       message tag (integer) 7
    IN      comm      communicator (handle) 8
    OUT     request   communication request (handle) 9
    10
    11
    12
    13
int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest, 14
              int tag, MPI_Comm comm, MPI_Request *request) 15
    16
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) 17
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 18
    INTEGER, INTENT(IN) :: count, dest, tag 19
    TYPE(MPI_Datatype), INTENT(IN) :: datatype 20
    TYPE(MPI_Comm), INTENT(IN) :: comm 21
    TYPE(MPI_Request), INTENT(OUT) :: request 22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23
    24
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 25
    <type> BUF(*) 26
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 27
    28
    Start a standard mode, nonblocking send. 29
    30
MPI_IBSEND(buf, count, datatype, dest, tag, comm, request) 31
    IN      buf      initial address of send buffer (choice) 32
    IN      count     number of elements in send buffer (non-negative inte- 33
                        ger) 34
    IN      datatype  datatype of each send buffer element (handle) 35
    IN      dest      rank of destination (integer) 36
    IN      tag       message tag (integer) 37
    IN      comm      communicator (handle) 38
    OUT     request   communication request (handle) 39
    40
    41
    42
int MPI_Ibsend(const void* buf, int count, MPI_Datatype datatype, int dest, 43
               int tag, MPI_Comm comm, MPI_Request *request) 44
    45
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) 46
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 47
    INTEGER, INTENT(IN) :: count, dest, tag 48
    TYPE(MPI_Datatype), INTENT(IN) :: datatype

```

```

1      TYPE(MPI_Comm), INTENT(IN) :: comm
2      TYPE(MPI_Request), INTENT(OUT) :: request
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5      MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
6      <type> BUF(*)
7      INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
8
9      Start a buffered mode, nonblocking send.
10
11     MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
12
13     IN          buf          initial address of send buffer (choice)
14
15     IN          count        number of elements in send buffer (non-negative inte-
16                               ger)
17
18     IN          datatype     datatype of each send buffer element (handle)
19
20     IN          dest         rank of destination (integer)
21
22     IN          tag          message tag (integer)
23
24     IN          comm         communicator (handle)
25
26     OUT         request      communication request (handle)
27
28     int MPI_Issend(const void* buf, int count, MPI_Datatype datatype, int dest,
29                   int tag, MPI_Comm comm, MPI_Request *request)
30
31     MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
32     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
33     INTEGER, INTENT(IN) :: count, dest, tag
34     TYPE(MPI_Datatype), INTENT(IN) :: datatype
35     TYPE(MPI_Comm), INTENT(IN) :: comm
36     TYPE(MPI_Request), INTENT(OUT) :: request
37     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39     MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
40     <type> BUF(*)
41     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
42
43     Start a synchronous mode, nonblocking send.
44
45
46
47
48

```

MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)			1
IN	buf	initial address of send buffer (choice)	2
IN	count	number of elements in send buffer (non-negative integer)	3
			4
			5
IN	datatype	datatype of each send buffer element (handle)	6
IN	dest	rank of destination (integer)	7
			8
IN	tag	message tag (integer)	9
IN	comm	communicator (handle)	10
OUT	request	communication request (handle)	11
			12
			13
int MPI_Irsend(const void* buf, int count, MPI_Datatype datatype, int dest,			14
int tag, MPI_Comm comm, MPI_Request *request)			15
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)			16
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf			17
INTEGER, INTENT(IN) :: count, dest, tag			18
TYPE(MPI_Datatype), INTENT(IN) :: datatype			19
TYPE(MPI_Comm), INTENT(IN) :: comm			20
TYPE(MPI_Request), INTENT(OUT) :: request			21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			22
			23
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)			24
<type> BUF(*)			25
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR			26
Start a ready mode nonblocking send.			27
			28
			29
MPI_IRECV (buf, count, datatype, source, tag, comm, request)			30
OUT	buf	initial address of receive buffer (choice)	31
IN	count	number of elements in receive buffer (non-negative integer)	32
			33
			34
IN	datatype	datatype of each receive buffer element (handle)	35
IN	source	rank of source or MPI_ANY_SOURCE (integer)	36
IN	tag	message tag or MPI_ANY_TAG (integer)	37
			38
IN	comm	communicator (handle)	39
OUT	request	communication request (handle)	40
			41
			42
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,			43
int tag, MPI_Comm comm, MPI_Request *request)			44
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)			45
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf			46
INTEGER, INTENT(IN) :: count, source, tag			47
TYPE(MPI_Datatype), INTENT(IN) :: datatype			48

```

1      TYPE(MPI_Comm), INTENT(IN) :: comm
2      TYPE(MPI_Request), INTENT(OUT) :: request
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5      MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
6      <type> buf(*)
7      INTEGER count, datatype, source, tag, comm, request, ierror

```

Start a nonblocking receive.

These calls allocate a communication request object and associate it with the request handle (the argument `request`). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10–??.
(*End of advice to users.*)

4.7.3 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 4.9). A handle is *active* if it is neither null nor inactive. An *empty* status is a status which is set to return `tag = MPI_ANY_TAG`, `source = MPI_ANY_SOURCE`, `error = MPI_SUCCESS`, and is also internally configured so that calls to `MPI_GET_COUNT`, `MPI_GET_ELEMENTS`, and `MPI_GET_ELEMENTS_X` return `count = 0` and `MPI_TEST_CANCELLED` returns false. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

The fields in a *status* object returned by a call to `MPI_WAIT`, `MPI_TEST`, or any of the other derived functions (`MPI_{TEST|WAIT}{ALL|SOME|ANY}`), where the *request* corresponds to a send call, are undefined, with two exceptions: The error status field will

contain valid information if the wait or test call returned with `MPI_ERR_IN_STATUS`; and the returned status can be queried by the call `MPI_TEST_CANCELLED`.

Error codes belonging to the error class `MPI_ERR_IN_STATUS` should be returned only by the MPI completion functions that take arrays of `MPI_Status`. For the functions `MPI_TEST`, `MPI_TESTANY`, `MPI_WAIT`, and `MPI_WAITANY`, which return a single `MPI_Status` value, the normal MPI error return process should be used (not the `MPI_ERROR` field in the `MPI_Status` argument).

`MPI_WAIT(request, status)`

INOUT	request	request (handle)
OUT	status	status object (<code>Status</code>)

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

```

MPI_Wait(request, status, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

```

A call to `MPI_WAIT` returns when the operation identified by `request` is complete. If the request is an active persistent request, it is marked inactive. Any other type of request is 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 4.2.5. The status object for a send operation may be queried by a call to `MPI_TEST_CANCELLED` (see Section 4.8).

One is allowed to call `MPI_WAIT` with a null or inactive `request` argument. In this case the operation returns immediately with empty `status`.

Advice to users. Successful return of `MPI_WAIT` after a `MPI_IBSEND` 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 4.8). 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 advice to users.*)

Advice to implementors. In a multithreaded 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.*)

```

1  MPI_TEST(request, flag, status)
2      INOUT    request          communication request (handle)
3      OUT      flag             true if operation completed (logical)
4      OUT      status           status object (Status)
5
6
7  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
8
9  MPI_Test(request, flag, status, ierror)
10     TYPE(MPI_Request), INTENT(INOUT) :: request
11     LOGICAL, INTENT(OUT) :: flag
12     TYPE(MPI_Status) :: status
13     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15  MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
16     LOGICAL FLAG
17     INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

```

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 request is an active persistent request, it is marked as inactive. Any other type of request is deallocated and the request handle is set to `MPI_REQUEST_NULL`. The call returns `flag = false` if the operation identified by `request` is not complete. 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 4.2.5. The status object for a send operation carries information that can be accessed by a call to `MPI_TEST_CANCELLED` (see Section 4.8).

One is allowed to call `MPI_TEST` with a null or inactive `request` argument. In such a case the operation returns with `flag = true` and empty `status`.

The functions `MPI_WAIT` and `MPI_TEST` can be used to complete both sends and receives.

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.*)

Example 4.11 Simple usage of nonblocking operations and `MPI_WAIT`.

```

39  CALL MPI_COMM_RANK(comm, rank, ierr)
40  IF (rank.EQ.0) THEN
41      CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
42      **** do some computation to mask latency ****
43      CALL MPI_WAIT(request, status, ierr)
44  ELSE IF (rank.EQ.1) THEN
45      CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
46      **** do some computation to mask latency ****
47      CALL MPI_WAIT(request, status, ierr)
48  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)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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. 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 4.12 An example using `MPI_REQUEST_FREE`.

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

```

1      END DO
2      CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
3      CALL MPI_WAIT(req, status, ierr)
4  END IF

```

4.7.4 Semantics of Nonblocking Communications

The semantics of nonblocking communication is defined by suitably extending the definitions in Section 4.5.

Order Nonblocking communication operations are ordered according to the execution order of the calls that initiate the communication. The non-overtaking requirement of Section 4.5 is extended to nonblocking communication, with this definition of order being used.

Example 4.13 Message ordering for nonblocking operations.

```

16 CALL MPI_COMM_RANK(comm, rank, ierr)
17 IF (RANK.EQ.0) THEN
18     CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
19     CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
20 ELSE IF (rank.EQ.1) THEN
21     CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
22     CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
23 END IF
24 CALL MPI_WAIT(r1, status, ierr)
25 CALL MPI_WAIT(r2, status, ierr)

```

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 4.14 An illustration of progress semantics.

```

38 CALL MPI_COMM_RANK(comm, rank, ierr)
39 IF (RANK.EQ.0) THEN
40     CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
41     CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
42 ELSE IF (rank.EQ.1) THEN
43     CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
44     CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
45     CALL MPI_WAIT(r, status, ierr)
46 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.

4.7.5 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_WAITSOME` 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 (non-negative 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)
```

```
MPI_Waitany(count, array_of_requests, index, status, ierror)
  INTEGER, INTENT(IN) :: count
  TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
  INTEGER, INTENT(OUT) :: index
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
  INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
  IERROR
```

Blocks until one of the operations associated with the active requests in the array has completed. If more than 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 operation. (The array is indexed from zero in C, and from one in Fortran.) If the request is an active persistent request, it is marked inactive. Any other type of request 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`, and an empty `status`.

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` (unless the value of `index` is `MPI_UNDEFINED`). `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 (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	index	index of operation that completed, or <code>MPI_UNDEFINED</code> 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)
```

```
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, INTENT(OUT) :: index
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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 is an active persistent request, it is marked as inactive. Any other type of 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 (no operation completed), 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 = true`, `index = MPI_UNDEFINED`, and an empty `status`.

If the array of requests contains active handles then 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 (non-negative 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, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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. Active persistent requests are marked inactive. Requests of any other type 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 sets to empty the status of each such entry.

The error-free execution of `MPI_WAITALL(count, array_of_requests, array_of_statuses)` has the same effect as the execution of `MPI_WAIT(&array_of_request[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`.

When one or more of the communications completed by a call to `MPI_WAITALL` fail, it is desirable to return specific information on each communication. The function `MPI_WAITALL` will return in such case the error code `MPI_ERR_IN_STATUS` and will set the error field of each status to a specific error code. This code will be `MPI_SUCCESS`, if the specific communication completed; it will be another specific error code, if it failed; or it can be `MPI_ERR_PENDING` if it has neither failed nor completed. The function `MPI_WAITALL` will return `MPI_SUCCESS` if no request had an error, or will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

Rationale. This design streamlines error handling in the application. The application code need only test the (single) function result to determine if an error has occurred. It needs to check each individual status only when an error occurred. (*End of rationale.*)

```
1 MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)
```

```
2     IN          count                lists length (non-negative integer)
3
4     INOUT      array_of_requests     array of requests (array of handles)
5
6     OUT        flag                  (logical)
7
8     OUT        array_of_statuses     array of status objects (array of Status)
```

```
8
9 int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag,
10                MPI_Status array_of_statuses[])
```

```
11 MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
12     INTEGER, INTENT(IN) :: count
13     TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
14     LOGICAL, INTENT(OUT) :: flag
15     TYPE(MPI_Status) :: array_of_statuses(*)
16     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
17 MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
18     LOGICAL FLAG
19     INTEGER COUNT, ARRAY_OF_REQUESTS(*),
20     ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

22 Returns `flag = true` if all communications associated with active handles in the array
23 have completed (this includes the case where no handle in the list is active). In this case, each
24 status entry that corresponds to an active request is set to the status of the corresponding
25 operation. Active persistent requests are marked inactive. Requests of any other type are
26 deallocated and the corresponding handles in the array are set to `MPI_REQUEST_NULL`.
27 Each status entry that corresponds to a null or inactive handle is set to empty.

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

30 Errors that occurred during the execution of `MPI_TESTALL` are handled in the same
31 manner as errors in `MPI_WAITALL`.

```
32
33
34 MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)
```

```
35
36     IN          incount                length of array_of_requests (non-negative integer)
37
38     INOUT      array_of_requests     array of requests (array of handles)
39
40     OUT        outcount                number of completed requests (integer)
41
42     OUT        array_of_indices       array of indices of operations that completed (array of
43                                         integers)
44
45     OUT        array_of_statuses     array of status objects for operations that completed
46                                         (array of Status)
```

```
45
46 int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
47                int *outcount, int array_of_indices[],
48                MPI_Status array_of_statuses[])
```

```

MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
              array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WAIT SOME(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_status` the status for these completed operations. Completed active persistent requests are marked as inactive. Any other type or request that completed 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 = MPI_UNDEFINED`.

When one or more of the communications completed by `MPI_WAIT SOME` fails, then it is desirable to return specific information on each communication. The arguments `outcount`, `array_of_indices` and `array_of_statuses` will be adjusted to indicate completion of all communications that have succeeded or failed. The call will return the error code `MPI_ERR_IN_STATUS` and the error field of each status returned will be set to indicate success or to indicate the specific error that occurred. The call will return `MPI_SUCCESS` if no request resulted in an error, and will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

```

MPI_TEST SOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

```

IN	<code>incount</code>	length of <code>array_of_requests</code> (non-negative integer)
INOUT	<code>array_of_requests</code>	array of requests (array of handles)
OUT	<code>outcount</code>	number of completed requests (integer)
OUT	<code>array_of_indices</code>	array of indices of operations that completed (array of integers)
OUT	<code>array_of_statuses</code>	array of status objects for operations that completed (array of Status)

```

int MPI_Testsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[])

```

```

1  MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
2              array_of_statuses, ierror)
3      INTEGER, INTENT(IN) :: incount
4      TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
5      INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
6      TYPE(MPI_Status) :: array_of_statuses(*)
7      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9  MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
10             ARRAY_OF_STATUSES, IERROR)
11      INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
12      ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

```

Behaves like `MPI_WAITSSOME`, except that it returns immediately. If no operation has completed it returns `outcount = 0`. If there is no active handle in the list it returns `outcount = MPI_UNDEFINED`.

`MPI_TESTSSOME` is a local operation, which returns immediately, whereas `MPI_WAITSSOME` 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_WAITSSOME` or `MPI_TESTSSOME`, 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.

Errors that occur during the execution of `MPI_TESTSSOME` are handled as for `MPI_WAITSSOME`.

Advice to users. The use of `MPI_TESTSSOME` 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_WAITSSOME` so as not to starve any client. Clients send messages to the server with service requests. The server calls `MPI_WAITSSOME` 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_TESTSSOME` should complete as many pending communications as possible. (*End of advice to implementors.*)

Example 4.15 Client-server code (starvation can occur).

```

40 CALL MPI_COMM_SIZE(comm, size, ierr)
41 CALL MPI_COMM_RANK(comm, rank, ierr)
42 IF(rank .GT. 0) THEN          ! client code
43     DO WHILE(.TRUE.)
44         CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
45         CALL MPI_WAIT(request, status, ierr)
46     END DO
47 ELSE                          ! rank=0 -- server code
48     DO i=1, size-1

```

```

        CALL MPI_Irecv(a(1,i), n, MPI_REAL, i, 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(1,index)) ! handle one message
        CALL MPI_Irecv(a(1, index), n, MPI_REAL, index, tag,
                        comm, request_list(index), ierr)
    END DO
END IF

```

Example 4.16 Same code, using MPI_WAITSSOME.

```

CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 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(1,i), n, MPI_REAL, i, tag,
                        comm, request_list(i), ierr)
    END DO
    DO WHILE(.TRUE.)
        CALL MPI_WAITSSOME(size, request_list, numdone,
                            indices, statuses, ierr)
        DO i=1, numdone
            CALL DO_SERVICE(a(1, indices(i)))
            CALL MPI_Irecv(a(1, indices(i)), n, MPI_REAL, 0, tag,
                            comm, request_list(indices(i)), ierr)
        END DO
    END DO
END IF

```

4.7.6 Non-destructive Test of status

This call is useful for accessing the information associated with a request, without freeing the request (in case the user is expected to access it later). It allows one to layer libraries more conveniently, since multiple layers of software may access the same completed request and extract from it the status information.

```

1 MPI_REQUEST_GET_STATUS( request, flag, status )
2     IN      request      request (handle)
3     OUT     flag          boolean flag, same as from MPI_TEST (logical)
4     OUT     status        status object if flag is true (Status)
5
6

```

```

7 int MPI_Request_get_status(MPI_Request request, int *flag,
8     MPI_Status *status)
9

```

```

10 MPI_Request_get_status(request, flag, status, ierror)
11     TYPE(MPI_Request), INTENT(IN) :: request
12     LOGICAL, INTENT(OUT) :: flag
13     TYPE(MPI_Status) :: status
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15

```

```

16 MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
17     INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
18     LOGICAL FLAG
19

```

Sets `flag=true` if the operation is complete, and, if so, returns in `status` the request status. However, unlike `test` or `wait`, it does not deallocate or inactivate the request; a subsequent call to `test`, `wait` or `free` should be executed with that request. It sets `flag=false` if the operation is not complete.

One is allowed to call `MPI_REQUEST_GET_STATUS` with a null or inactive request argument. In such a case the operation returns with `flag=true` and empty `status`.

4.8 Probe and Cancel

The `MPI_PROBE`, `MPI_IPROBE`, `MPI_MPROBE`, and `MPI_IMPROBE` 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 cancelled. 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.

4.8.1 Probe

```

40 MPI_IPROBE(source, tag, comm, flag, status)
41     IN      source      rank of source or MPI_ANY_SOURCE (integer)
42     IN      tag          message tag or MPI_ANY_TAG (integer)
43     IN      comm         communicator (handle)
44     OUT     flag          (logical)
45     OUT     status        status object (Status)
46
47
48

```

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

```
MPI_Iprobe(source, tag, comm, flag, status, ierror)
```

```
    INTEGER, INTENT(IN) :: source, tag
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
    LOGICAL, INTENT(OUT) :: flag
```

```
    TYPE(MPI_Status) :: status
```

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

```
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
```

```
    LOGICAL FLAG
```

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

MPI_IPROBE(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_IPROBE returns flag = true, then the content of the status object can be subsequently accessed as described in Section 4.2.5 to find the source, tag and length of the probed message.

A subsequent receive executed with the same communicator, and the source and tag returned in status by MPI_IPROBE will receive the message that was matched by the probe, if no other intervening receive occurs after the probe, and the send is not successfully cancelled before the receive. If the receiving process is multithreaded, 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.

A probe with MPI_PROC_NULL as source returns flag = true, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0; see Section 4.11.

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

```
IN          source          rank of source or MPI_ANY_SOURCE (integer)
```

```
IN          tag             message tag 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, INTENT(IN) :: source, tag
```

```

1      TYPE(MPI_Comm), INTENT(IN) :: comm
2      TYPE(MPI_Status) :: status
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

4      MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
5      INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
6

```

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_IPROBE and a matching message has been issued, then the call to MPI_IPROBE will eventually return `flag = true` unless the message is received by another concurrent receive operation or matched by a concurrent matched probe.

Example 4.17

Use blocking probe to wait for an incoming message.

```

20      CALL MPI_COMM_RANK(comm, rank, ierr)
21      IF (rank.EQ.0) THEN
22          CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
23      ELSE IF (rank.EQ.1) THEN
24          CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
25      ELSE IF (rank.EQ.2) THEN
26          DO i=1, 2
27              CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
28                             comm, status, ierr)
29              IF (status(MPI_SOURCE) .EQ. 0) THEN
30                  100      CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
31              ELSE
32                  200      CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
33              END IF
34          END DO
35      END IF
36

```

Each message is received with the right type.

Example 4.18 A similar program to the previous example, but now it has a problem.

```

41      CALL MPI_COMM_RANK(comm, rank, ierr)
42      IF (rank.EQ.0) THEN
43          CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
44      ELSE IF (rank.EQ.1) THEN
45          CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
46      ELSE IF (rank.EQ.2) THEN
47          DO i=1, 2
48              CALL MPI_PROBE(MPI_ANY_SOURCE, 0,

```

```

                                comm, status, ierr)          1
    IF (status(MPI_SOURCE) .EQ. 0) THEN                      2
100      CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,    3
                                0, comm, status, ierr)        4
    ELSE                                                      5
200      CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,       6
                                0, comm, status, ierr)        7
    END IF                                                    8
  END DO                                                      9
END IF                                                       10

```

In Example 4.18, the two receive calls in statements labeled 100 and 200 in Example 4.17 slightly modified, using `MPI_ANY_SOURCE` as the `source` argument. 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 users. In a multithreaded MPI program, `MPI_PROBE` and `MPI_Iprobe` might need special care. If a thread probes for a message and then immediately posts a matching receive, the receive may match a message other than that found by the probe since another thread could concurrently receive that original message [2]. `MPI_Mprobe` and `MPI_improbe` solve this problem by matching the incoming message so that it may only be received with `MPI_Mrecv` or `MPI_imrecv` on the corresponding message handle. (*End of advice to users.*)

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.*)

4.8.2 Matching Probe

The function `MPI_PROBE` checks for incoming messages without receiving them. Since the list of incoming messages is global among the threads of each MPI process, it can be hard to use this functionality in threaded environments [2, 1].

Like `MPI_PROBE` and `MPI_Iprobe`, the `MPI_Mprobe` and `MPI_improbe` operations allow incoming messages to be queried without actually receiving them, except that `MPI_Mprobe` and `MPI_improbe` provide a mechanism to receive the specific message that was matched regardless of other intervening probe or receive operations. This gives the application an opportunity to decide how to receive the message, based on the information returned by the probe. In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

```

1 MPI_IMPROBE(source, tag, comm, flag, message, status)
2     IN          source          rank of source or MPI_ANY_SOURCE (integer)
3     IN          tag             message tag or MPI_ANY_TAG (integer)
4     IN          comm            communicator (handle)
5     OUT         flag            flag (logical)
6     OUT         message         returned message (handle)
7     OUT         status          status object (Status)

```

```

11 int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,
12                MPI_Message *message, MPI_Status *status)

```

```

13 MPI_Improbe(source, tag, comm, flag, message, status, ierror)
14     INTEGER, INTENT(IN) :: source, tag
15     TYPE(MPI_Comm), INTENT(IN) :: comm
16     LOGICAL, INTENT(OUT) :: flag
17     TYPE(MPI_Message), INTENT(OUT) :: message
18     TYPE(MPI_Status) :: status
19     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

21 MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
22     INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
23     LOGICAL FLAG

```

MPI_IMPROBE(source, tag, comm, flag, message, 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. In addition, it returns in message a handle to the matched message. Otherwise, the call returns flag = false, and leaves status and message undefined.

A matched receive (MPI_MRECV or MPI_IMRECV) executed with the message handle will receive the message that was matched by the probe. Unlike MPI_IPROBE, no other probe or receive operation may match the message returned by MPI_IMPROBE. Each message returned by MPI_IMPROBE must be received with either MPI_MRECV or MPI_IMRECV.

The source argument of MPI_IMPROBE 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.

A synchronous send operation that is matched with MPI_IMPROBE or MPI_MPROBE will complete successfully only if both a matching receive is posted with MPI_MRECV or MPI_IMRECV, and the receive operation has started to receive the message sent by the synchronous send.

There is a special predefined message: MPI_MESSAGE_NO_PROC, which is a message which has MPI_PROC_NULL as its source process. The predefined constant MPI_MESSAGE_NULL is the value used for invalid message handles.

A matching probe with `MPI_PROC_NULL` as source returns `flag = true`, `message = MPI_MESSAGE_NO_PROC`, and the status object returns `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG`, and `count = 0`; see Section 4.11. It is not necessary to call `MPI_MRECV` or `MPI_IMRECV` with `MPI_MESSAGE_NO_PROC`, but it is not erroneous to do so.

Rationale. `MPI_MESSAGE_NO_PROC` was chosen instead of `MPI_MESSAGE_PROC_NULL` to avoid possible confusion as another null handle constant. (*End of rationale.*)

`MPI_MPROBE(source, tag, comm, message, status)`

IN	source	rank of source or <code>MPI_ANY_SOURCE</code> (integer)
IN	tag	message tag or <code>MPI_ANY_TAG</code> (integer)
IN	comm	communicator (handle)
OUT	message	returned message (handle)
OUT	status	status object (Status)

```
int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message,
               MPI_Status *status)
```

```
MPI_Mprobe(source, tag, comm, message, status, ierror)
```

```
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Message), INTENT(OUT) :: message
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)
```

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

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

The implementation of `MPI_MPROBE` and `MPI_IMPROBE` needs to guarantee progress in the same way as in the case of `MPI_PROBE` and `MPI_IPROBE`.

4.8.3 Matched Receives

The functions `MPI_MRECV` and `MPI_IMRECV` receive messages that have been previously matched by a matching probe (Section 4.8.2).

1 MPI_MRECV(buf, count, datatype, message, status)

2	OUT	buf	initial address of receive buffer (choice)
3			
4	IN	count	number of elements in receive buffer (non-negative integer)
5			
6	IN	datatype	datatype of each receive buffer element (handle)
7	INOUT	message	message (handle)
8			
9	OUT	status	status object (Status)

10
11 int MPI_Mrecv(void* buf, int count, MPI_Datatype datatype,
12 MPI_Message *message, MPI_Status *status)

13 MPI_Mrecv(buf, count, datatype, message, status, ierror)

14 TYPE(*), DIMENSION(..) :: buf
15 INTEGER, INTENT(IN) :: count
16 TYPE(MPI_Datatype), INTENT(IN) :: datatype
17 TYPE(MPI_Message), INTENT(INOUT) :: message
18 TYPE(MPI_Status) :: status
19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20

21 MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)

22 <type> BUF(*)
23 INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR

24 This call receives a message matched by a matching probe operation (Section 4.8.2).

25 The receive buffer consists of the storage containing count consecutive elements of the
26 type specified by datatype, starting at address buf. The length of the received message must
27 be less than or equal to the length of the receive buffer. An overflow error occurs if all
28 incoming data does not fit, without truncation, into the receive buffer.

29 If the message is shorter than the receive buffer, then only those locations corresponding
30 to the (shorter) message are modified.

31 On return from this function, the message handle is set to MPI_MESSAGE_NULL. All
32 errors that occur during the execution of this operation are handled according to the error
33 handler set for the communicator used in the matching probe call that produced the message
34 handle.

35 If MPI_MRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the
36 call returns immediately with the status object set to source = MPI_PROC_NULL, tag =
37 MPI_ANY_TAG, and count = 0, as if a receive from MPI_PROC_NULL was issued (see Sec-
38 tion 4.11). A call to MPI_MRECV with MPI_MESSAGE_NULL is erroneous.
39

`MPI_IMRECV(buf, count, datatype, message, request)`

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
INOUT	message	message (handle)
OUT	request	communication request (handle)

```
int MPI_Imrecv(void* buf, int count, MPI_Datatype datatype,
               MPI_Message *message, MPI_Request *request)
```

```
MPI_Imrecv(buf, count, datatype, message, request, ierror)
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Message), INTENT(INOUT) :: message
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)
```

```
<type> BUF(*)
INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR
```

`MPI_IMRECV` is the nonblocking variant of `MPI_MRECV` and starts a nonblocking receive of a matched message. Completion semantics are similar to `MPI_IRECV` as described in Section 4.7.2. On return from this function, the message handle is set to `MPI_MESSAGE_NULL`.

If `MPI_IMRECV` is called with `MPI_MESSAGE_NO_PROC` as the message argument, the call returns immediately with a request object which, when completed, will yield a status object set to source = `MPI_PROC_NULL`, tag = `MPI_ANY_TAG`, and count = 0, as if a receive from `MPI_PROC_NULL` was issued (see Section 4.11). A call to `MPI_IMRECV` with `MPI_MESSAGE_NULL` is erroneous.

Advice to implementors. If reception of a matched message is started with `MPI_IMRECV`, then it is possible to cancel the returned request with `MPI_CANCEL`. If `MPI_CANCEL` succeeds, the matched message must be found by a subsequent message probe (`MPI_PROBE`, `MPI_IPROBE`, `MPI_MPROBE`, or `MPI_ImPROBE`), received by a subsequent receive operation or cancelled by the sender. See Section 4.8.4 for details about `MPI_CANCEL`. The cancellation of operations initiated with `MPI_IMRECV` may fail. (*End of advice to implementors.*)

4.8.4 Cancel

`MPI_CANCEL(request)`

IN	request	communication request (handle)
----	---------	--------------------------------

```

1  int MPI_Cancel(MPI_Request *request)
2
3  MPI_Cancel(request, ierror)
4      TYPE(MPI_Request), INTENT(IN) :: request
5      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7  MPI_CANCEL(REQUEST, IERROR)
8      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 cancelled. It is still necessary to call `MPI_REQUEST_FREE`, `MPI_WAIT` or `MPI_TEST` (or any of the derived operations) with the cancelled request as argument after the call to `MPI_CANCEL`. 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 cancelled communication, then `MPI_TEST` will eventually be successful.

`MPI_CANCEL` can be used to cancel a communication that uses a persistent request (see Section 4.9), 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 cancelled, 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 cancelled, 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 cancelled, then information to that effect will be returned in the status argument of the operation that completes the communication.

Rationale. Although the IN request handle parameter should not need to be passed by reference, the C binding has listed the argument type as `MPI_Request*` since MPI-1.0. This function signature therefore cannot be changed without breaking existing MPI applications. (*End of rationale.*)

```

41 MPI_TEST_CANCELLED(status, flag)
42
43     IN          status          status object (Status)
44     OUT         flag            (logical)
45
46 int MPI_Test_cancelled(const MPI_Status *status, int *flag)
47
48 MPI_Test_cancelled(status, flag, ierror)

```

```

    TYPE(MPI_Status), INTENT(IN) :: status
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    LOGICAL FLAG
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR

```

Returns `flag = true` if the communication associated with the status object was cancelled 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 cancelled then one should call `MPI_TEST_CANCELLED` first, to check whether the operation was cancelled, 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.*)

4.9 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 five following calls. These calls involve no communication.

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

```
2     IN      buf                initial address of send buffer (choice)
3
4     IN      count              number of elements sent (non-negative integer)
5
6     IN      datatype           type of each element (handle)
7
8     IN      dest               rank of destination (integer)
9
10    IN      tag                 message tag (integer)
11
12    IN      comm                communicator (handle)
13
14    OUT     request             communication request (handle)
```

```
12 int MPI_Send_init(const void* buf, int count, MPI_Datatype datatype,
13                  int dest, int tag, MPI_Comm comm, MPI_Request *request)
14
```

```
15 MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
16     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
17     INTEGER, INTENT(IN) :: count, dest, tag
18     TYPE(MPI_Datatype), INTENT(IN) :: datatype
19     TYPE(MPI_Comm), INTENT(IN) :: comm
20     TYPE(MPI_Request), INTENT(OUT) :: request
21     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

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

```
29
30 MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)
```

```
31     IN      buf                initial address of send buffer (choice)
32
33     IN      count              number of elements sent (non-negative integer)
34
35     IN      datatype           type of each element (handle)
36
37     IN      dest               rank of destination (integer)
38
39     IN      tag                 message tag (integer)
40
41     IN      comm                communicator (handle)
42
43     OUT     request             communication request (handle)
```

```
41 int MPI_Bsend_init(const void* buf, int count, MPI_Datatype datatype,
42                  int dest, int tag, MPI_Comm comm, MPI_Request *request)
43
```

```
44 MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
45     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
46     INTEGER, INTENT(IN) :: count, dest, tag
47     TYPE(MPI_Datatype), INTENT(IN) :: datatype
48     TYPE(MPI_Comm), INTENT(IN) :: comm
```

```

    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

    Creates a persistent communication request for a buffered mode send.

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

    IN      buf                initial address of send buffer (choice)
    IN      count              number of elements sent (non-negative 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_Ssend_init(const void* buf, int count, MPI_Datatype datatype,
                  int dest, int tag, MPI_Comm comm, MPI_Request *request)

MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

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

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

    IN      buf                initial address of send buffer (choice)
    IN      count              number of elements sent (non-negative 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)

```

```

1  int MPI_Rsend_init(const void* buf, int count, MPI_Datatype datatype,
2                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
3
4  MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
5      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
6      INTEGER, INTENT(IN) :: count, dest, tag
7      TYPE(MPI_Datatype), INTENT(IN) :: datatype
8      TYPE(MPI_Comm), INTENT(IN) :: comm
9      TYPE(MPI_Request), INTENT(OUT) :: request
10     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12 MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
13     <type> BUF(*)
14     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

```

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

```

17 MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)
18
19     OUT      buf                initial address of receive buffer (choice)
20     IN       count              number of elements received (non-negative integer)
21     IN       datatype           type of each element (handle)
22     IN       source             rank of source or MPI_ANY_SOURCE (integer)
23     IN       tag                message tag or MPI_ANY_TAG (integer)
24     IN       comm               communicator (handle)
25     OUT      request            communication request (handle)
26
27

```

```

28 int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
29                  int tag, MPI_Comm comm, MPI_Request *request)
30
31 MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
32     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
33     INTEGER, INTENT(IN) :: count, source, tag
34     TYPE(MPI_Datatype), INTENT(IN) :: datatype
35     TYPE(MPI_Comm), INTENT(IN) :: comm
36     TYPE(MPI_Request), INTENT(OUT) :: request
37     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39 MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
40     <type> BUF(*)
41     INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

```

Creates a persistent communication request for a receive operation. The argument `buf` is marked as OUT because the user gives permission to write on the receive buffer by passing the argument to `MPI_RECV_INIT`.

A persistent communication request is inactive after it was created — no active communication is attached to the request.

A communication (send or receive) that uses a persistent request is initiated by the function `MPI_START`.

`MPI_START(request)`

INOUT request communication request (handle)

`int MPI_Start(MPI_Request *request)`

`MPI_Start(request, ierror)`

`TYPE(MPI_Request), INTENT(INOUT) :: request`

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

`MPI_START(REQUEST, IERROR)`

`INTEGER REQUEST, IERROR`

The argument, `request`, is a handle returned by one of the previous five calls. The associated request should be inactive. The request becomes active once the call is made.

If the request is for a send with ready mode, then a matching receive should be posted before the call is made. The communication buffer should not be modified after the call, and until the operation completes.

The call is local, with similar semantics to the nonblocking communication operations described in Section 4.7. That is, a call to `MPI_START` with a request created by `MPI_SEND_INIT` starts a communication in the same manner as a call to `MPI_ISEND`; a call to `MPI_START` with a request created by `MPI_BSEND_INIT` starts a communication in the same manner as a call to `MPI_IBSEND`; and so on.

`MPI_STARTALL(count, array_of_requests)`

IN count list length (non-negative integer)

INOUT array_of_requests array of requests (array of handle)

`int MPI_Startall(int count, MPI_Request array_of_requests[])`

`MPI_Startall(count, array_of_requests, ierror)`

`INTEGER, INTENT(IN) :: count`

`TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)`

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

`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 $i=0, \dots, \text{count}-1$, 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 4.7.5. 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 4.7.3).

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.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10–??.
(*End of advice to users.*)

4.10 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 7 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 (non-negative 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 (non-negative integer)
IN	recvtype	type of elements in receive buffer (handle)
IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	recvtag	receive tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```

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

```

```

MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
              recvcount, recvtype, source, recvtag, comm, status, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  TYPE(*), DIMENSION(..) :: recvbuf
  INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
  recvtag
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

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.

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 execute the receive, followed by a join of these two threads.

```

1  MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, sta-
2      tus)
3      INOUT    buf                initial address of send and receive buffer (choice)
4      IN       count              number of elements in send and receive buffer (non-
5                                  negative integer)
6
7      IN       datatype           type of elements in send and receive buffer (handle)
8      IN       dest               rank of destination (integer)
9      IN       sendtag            send message tag (integer)
10     IN       source              rank of source or MPI_ANY_SOURCE (integer)
11     IN       recvtag            receive message tag or MPI_ANY_TAG (integer)
12     IN       comm                communicator (handle)
13     OUT      status              status object (Status)
14
15
16
17  int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
18                          int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
19                          MPI_Status *status)
20
21  MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
22                      comm, status, ierror)
23      TYPE(*), DIMENSION(..) :: buf
24      INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
25      TYPE(MPI_Datatype), INTENT(IN) :: datatype
26      TYPE(MPI_Comm), INTENT(IN) :: comm
27      TYPE(MPI_Status) :: status
28      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30  MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
31                      COMM, STATUS, IERROR)
32
33  <type> BUF(*)
34  INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
35  STATUS(MPI_STATUS_SIZE), IERROR
36

```

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.

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

4.11 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`. A probe or matching probe with `source = MPI_PROC_NULL` succeeds and returns as soon as possible, and the status object returns `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG` and `count = 0`. A matching probe (cf. Section 4.8.2) with `MPI_PROC_NULL` as source returns `flag = true`, `message = MPI_MESSAGE_NO_PROC`, and the status object returns `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG`, and `count = 0`.

Bibliography

- [1] D. Gregor, T. Hoefer, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009. [4.8.2](#)
- [2] T. Hoefer, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. [4.8.1](#), [4.8.2](#)

Index

- active, [30](#)
- blocking, [15](#), [18](#)
- buffered, [15](#), [25](#), [26](#)
- buffered send, [18](#)
- communicator, [5](#)
- CONST:a, [13](#)
- CONST:b, [13](#)
- CONST:bool, [5](#)
- CONST:char**, [23](#)
- CONST:CHARACTER, [12](#), [13](#)
- CONST:false, [30](#)
- CONST:message = MPI_MESSAGE_NO_PROC, [47](#), [59](#)
- CONST:MPI_ADDRESS_KIND, [4](#)
- CONST:MPI_AINT, [3](#), [5](#)
- CONST:MPI_Aint, [3](#)
- CONST:MPI_ANY_SOURCE, [6](#), [7](#), [19](#), [29](#), [30](#), [42](#), [43](#), [45–47](#), [54](#), [57](#), [58](#)
- CONST:MPI_ANY_TAG, [6](#), [7](#), [9](#), [29](#), [30](#), [42](#), [43](#), [45–49](#), [54](#), [57–59](#)
- CONST:MPI_BOTTOM, [10](#)
- CONST:MPI_BSEND_OVERHEAD, [24](#)
- CONST:MPI_BYTE, [3](#), [4](#), [11](#), [12](#), [14](#)
- CONST:MPI_C_BOOL, [4](#)
- CONST:MPI_C_COMPLEX, [4](#)
- CONST:MPI_C_DOUBLE_COMPLEX, [4](#)
- CONST:MPI_C_FLOAT_COMPLEX (as a synonym), [4](#)
- CONST:MPI_C_LONG_DOUBLE_COMPLEX, [4](#)
- CONST:MPI_CHAR, [4](#), [14](#)
- CONST:MPI_CHARACTER, [3](#), [12–14](#)
- CONST:MPI_Comm, [2](#)
- CONST:MPI_COMM_WORLD, [5](#), [6](#)
- CONST:MPI_COMPLEX, [3](#)
- CONST:MPI_COUNT, [3](#), [5](#)
- CONST:MPI_Count, [3](#)
- CONST:MPI_COUNT_KIND, [4](#)
- CONST:MPI_CXX_BOOL, [5](#)
- CONST:MPI_CXX_DOUBLE_COMPLEX, [5](#)
- CONST:MPI_CXX_FLOAT_COMPLEX, [5](#)
- CONST:MPI_CXX_LONG_DOUBLE_COMPLEX, [5](#)
- CONST:MPI_DOUBLE, [4](#)
- CONST:MPI_DOUBLE_COMPLEX, [3](#)
- CONST:MPI_DOUBLE_PRECISION, [3](#)
- CONST:MPI_ERR_IN_STATUS, [8](#), [10](#), [31](#), [37](#), [39](#)
- CONST:MPI_ERR_PENDING, [37](#)
- CONST:MPI_ERROR, [8](#), [31](#)
- CONST:MPI_FLOAT, [4](#)
- CONST:MPI_INT, [4](#)
- CONST:MPI_INT16_T, [4](#)
- CONST:MPI_INT32_T, [4](#)
- CONST:MPI_INT64_T, [4](#)
- CONST:MPI_INT8_T, [4](#)
- CONST:MPI_INTEGER, [3](#), [11](#)
- CONST:MPI_INTEGER1, [3](#)
- CONST:MPI_INTEGER2, [3](#)
- CONST:MPI_INTEGER4, [3](#)
- CONST:MPI_LOGICAL, [3](#)
- CONST:MPI_LONG, [4](#)
- CONST:MPI_LONG_DOUBLE, [4](#)
- CONST:MPI_LONG_LONG, [4](#)
- CONST:MPI_LONG_LONG_INT, [4](#)
- CONST:MPI_Message, [46](#)
- CONST:MPI_MESSAGE_NO_PROC, [46](#), [48](#), [49](#)
- CONST:MPI_MESSAGE_NULL, [46](#), [48](#), [49](#)
- CONST:MPI_OFFSET, [3](#)
- CONST:MPI_Offset, [3](#)
- CONST:MPI_OFFSET , [5](#)
- CONST:MPI_OFFSET_KIND, [4](#)
- CONST:MPI_PACKED, [3](#), [4](#), [11](#)
- CONST:MPI_PROC_NULL, [2](#), [5](#), [7](#), [8](#), [43](#), [46–49](#), [58](#), [59](#)
- CONST:MPI_REAL, [3](#), [11](#)
- CONST:MPI_REAL2, [3](#)

- CONST:MPI_REAL4, [3](#)
- CONST:MPI_REAL8, [3](#)
- CONST:MPI_Request, [27–29](#), [31](#), [32](#), [33](#), [35–39](#), [42](#), [49](#), [52–55](#)
- CONST:MPI_REQUEST_NULL, [30–33](#), [35–39](#)
- CONST:MPI_SHORT, [4](#)
- CONST:MPI_SIGNED_CHAR, [4](#)
- CONST:MPI_SOURCE, [8](#)
- CONST:MPI_Status, [6](#), [8–10](#), [31](#), [32](#), [35–39](#), [42](#), [43](#), [46–48](#), [50](#), [57](#), [58](#)
- CONST:MPI_STATUS_IGNORE, [10](#)
- CONST:MPI_STATUS_SIZE, [8](#)
- CONST:MPI_STATUSES_IGNORE, [10](#)
- CONST:MPI_SUCCESS, [30](#), [37](#), [39](#)
- CONST:MPI_TAG, [8](#)
- CONST:MPI_TAG_UB, [5](#)
- CONST:MPI_UINT16_T, [4](#)
- CONST:MPI_UINT32_T, [4](#)
- CONST:MPI_UINT64_T, [4](#)
- CONST:MPI_UINT8_T, [4](#)
- CONST:MPI_UNDEFINED, [9](#), [36](#), [39](#), [40](#)
- CONST:MPI_UNSIGNED, [4](#)
- CONST:MPI_UNSIGNED_CHAR, [4](#)
- CONST:MPI_UNSIGNED_LONG, [4](#)
- CONST:MPI_UNSIGNED_LONG_LONG, [4](#)
- CONST:MPI_UNSIGNED_SHORT, [4](#)
- CONST:MPI_WCHAR, [4](#)
- CONST:NULL, [10](#)
- CONST:std::complex<double>, [5](#)
- CONST:std::complex<float>, [5](#)
- CONST:std::complex<long double>, [5](#)
- CONST:true, [36](#)
- CONST:void*, [23](#)
- CONST:void**, [23](#)
- empty, [30](#)
- envelope, [1](#)
- EXAMPLES:Client-server code, [40](#), [41](#)
 - with blocking probe, [44](#)
 - with blocking probe, wrong, [44](#)
- EXAMPLES:Datatypes
 - matching, [11](#)
 - not matching, [12](#)
 - untyped, [12](#)
- EXAMPLES:Deadlock
 - if not buffered, [21](#)
 - wrong message exchange, [21](#)
- EXAMPLES:Intertwined matching pairs, [19](#)
- EXAMPLES:Message exchange (ping-pong), [20](#)
- EXAMPLES:MPI_BSEND, [19](#)
- EXAMPLES:MPI_Buffer_attach, [23](#)
- EXAMPLES:MPI_Buffer_detach, [23](#)
- EXAMPLES:MPI_BYTE, [12](#)
- EXAMPLES:MPI_CHARACTER, [13](#)
- EXAMPLES:MPI_IRECV, [32–34](#), [40](#), [41](#)
- EXAMPLES:MPI_ISEND, [32–34](#), [40](#), [41](#)
- EXAMPLES:MPI_PROBE, [44](#)
- EXAMPLES:MPI_RECV, [11–13](#), [19–21](#), [34](#), [44](#)
- EXAMPLES:MPI_REQUEST_FREE, [33](#)
- EXAMPLES:MPI_SEND, [11–13](#), [20](#), [21](#), [34](#), [44](#)
- EXAMPLES:MPI_SSEND, [19](#), [34](#)
- EXAMPLES:MPI_WAIT, [32–34](#), [40](#), [41](#)
- EXAMPLES:MPI_WAITANY, [40](#)
- EXAMPLES:MPI_WAIT SOME, [41](#)
- EXAMPLES:Non-overtaking messages, [19](#)
- EXAMPLES:Nonblocking operations, [32](#), [33](#)
 - message ordering, [34](#)
 - progress, [34](#)
- fairness, [40](#)
- immediate, [26](#)
- inactive, [30](#)
- local, [15](#)
- message, [1](#)
- message envelope, [5](#)
- MPI_BSEND, [24](#)
- MPI_BSEND (buf, count, datatype, dest, tag, comm), [16](#)
- MPI_BSEND_INIT, [55](#)
- MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request), [52](#)
- MPI_BUFFER_ATTACH, [31](#)
- MPI_BUFFER_ATTACH(buffer, size), [22](#)
- MPI_BUFFER_DETACH(buffer_addr, size), [23](#)
- MPI_CANCEL, [19](#), [31](#), [42](#), [49–51](#)
- MPI_CANCEL(request), [49](#)
- MPI_GET_COUNT, [9](#), [10](#), [30](#)
- MPI_GET_COUNT(status, datatype, count), [9](#)

MPI_GET_ELEMENTS, 30	MPI_REQUEST_GET_STATUS(request, flag, status), 42	1
MPI_GET_ELEMENTS_X, 30		2
MPI_IBSEND, 31 , 55	MPI_RSEND (buf, count, datatype, dest, tag, comm), 17	3
MPI_IBSEND(buf, count, datatype, dest, tag, comm, request), 27		4
MPI_IMPROBE, 42 , 45–47 , 49	MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request), 53	5
MPI_IMPROBE(source, tag, comm, flag, message, status), 46 , 46		6
MPI_IMRECV, 45–47 , 49	MPI_SEND, 1 , 3 , 10 , 12	7
MPI_IMRECV(buf, count, datatype, message, request), 49	MPI_SEND(buf, count, datatype, dest, tag, comm), 2	8
MPI_IPROBE, 9 , 42–47 , 49	MPI_SEND_INIT, 55	9
MPI_IPROBE(source, tag, comm, flag, status), 42 , 43	MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request), 52	10
MPI_IRECV, 49	MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status), 57	11
MPI_IRECV (buf, count, datatype, source, tag, comm, request), 29		12
MPI_IRSEND(buf, count, datatype, dest, tag, comm, request), 29	MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status), 58	13
MPI_ISEND, 55		14
MPI_ISEND(buf, count, datatype, dest, tag, comm, request), 27	MPI_SSEND (buf, count, datatype, dest, tag, comm), 17	15
MPI_ISSEND(buf, count, datatype, dest, tag, comm, request), 28		16
MPI_MPROBE, 42 , 45–47 , 49	MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request), 53	17
MPI_MPROBE(source, tag, comm, message, status), 47		18
MPI_MRECV, 45–49	MPI_START, 54–56	19
MPI_MRECV(buf, count, datatype, message, status), 48	MPI_START(request), 55	20
MPI_PACK, 24	MPI_STARTALL, 55	21
MPI_PACK_SIZE(count, datatype, comm, size), 24	MPI_STARTALL(count, array_of_requests), 55 , 55	22
MPI_PROBE, 7 , 9 , 10 , 42–45 , 47 , 49	MPI_TEST, 10 , 30–33 , 35 , 36 , 50 , 55	23
MPI_PROBE(source, tag, comm, status), 43 , 45	MPI_TEST(&array_of_requests[i], flag, status), 36	24
MPI_RECV, 2 , 8–10 , 46		25
MPI_RECV (buf, count, datatype, source, tag, comm, status), 6	MPI_TEST(request, flag, status), 32	26
MPI_RECV(..., source, tag, comm, status), 43 , 45 , 46	MPI_TEST_CANCELLED, 30–32 , 51	27
MPI_RECV(), 43	MPI_TEST_CANCELLED(status, flag), 50	28
MPI_RECV_INIT, 54	MPI_TESTALL, 35 , 38	29
MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request), 54	MPI_TESTALL(count, array_of_requests, flag, array_of_statuses), 38	30
MPI_REQUEST_FREE, 33 , 50 , 55		31
MPI_REQUEST_FREE(request), 33	MPI_TESTANY, 31 , 35 , 36 , 40	32
MPI_REQUEST_GET_STATUS, 10 , 42	MPI_TESTANY(count, array_of_requests, index, flag, status), 36	33
	MPI_TESTANY(count, array_of_requests, index, status), 36	34
	MPI_TESTSOME, 35 , 40	35
	MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses), 39	36
		37
	MPI_TYPE_CREATE_DARRAY, 9	38
	MPI_WAIT, 8 , 10 , 30–34 , 36 , 37 , 50 , 55	39
		40
		41
		42
		43
		44
		45
		46
		47
		48

1 MPI_WAIT(&array_of_request[i], &array_of_statuses[i]),
 2 [37](#)
 3 MPI_WAIT(&array_of_requests[i], status), [36](#)
 4 MPI_WAIT(request, status), [31](#)
 5 MPI_WAITALL, [35](#), [37](#), [38](#)
 6 MPI_WAITALL(count, array_of_requests,
 7 array_of_statuses), [37](#)
 8 MPI_WAITALL(count, array_of_requests, ar-
 9 ray_of_statuses), [37](#)
 10 MPI_WAITANY, [19](#), [31](#), [35](#), [36](#), [40](#)
 11 MPI_WAITANY (count, array_of_requests,
 12 index, status), [35](#)
 13 MPI_WAITANY(count, array_of_requests, in-
 14 dex, status), [36](#)
 15 MPI_WAITSOME, [35](#), [39–41](#)
 16 MPI_WAITSOME(incount, array_of_requests,
 17 outcount, array_of_indices, array_of_statuses),
 18 [38](#)
 19
 20 non-local, [15](#), [16](#)
 21 nonblocking communication, [25](#)
 22 null, [30](#)
 23
 24 persistent, [51](#)
 25 process group, [5](#)
 26
 27 ready, [16](#), [25](#), [26](#)
 28 ready send, [18](#)
 29 receive, [1](#), [2](#)
 30 receive buffer, [2](#)
 31 receive complete, [25](#)
 32 receive start call, [25](#)
 33 request, [26](#)
 34
 35 send, [1](#)
 36 send buffer, [1](#)
 37 send complete, [25](#)
 38 send start, [25](#)
 39 send-receive, [56](#)
 40 standard, [15](#), [25](#)
 41 standard send, [18](#)
 42 synchronous, [15](#), [25](#), [26](#), [30](#)
 43 synchronous send, [18](#)
 44
 45
 46
 47
 48