

D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

June 9, 2009

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"
main( argc, argv )
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();
}
```

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 channel-like constructs and send-receive operations, Nonblocking communication is addressed next, followed by 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 (nonnegative 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(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

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

```
<type> BUF(*)
```

```
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
{ void MPI::Comm::Send(const void* buf, int count, const
                       MPI::Datatype& datatype, int dest, int tag) const (binding
                       deprecated, see section XYZ) }
```

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.*)

MPI datatype	C datatype
MPI_CHAR	signed 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_BYTE	
MPI_PACKED	

Table 4.2: 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,...,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, ... , n-1, 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 (integer)
IN	tag	message tag (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)
```

```

1 MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
2     <type> BUF(*)
3     INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
4     IERROR
5
6 { void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
7     int source, int tag, MPI::Status& status) const (binding
8     deprecated, see section XYZ) }
9
10 { void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
11     int source, int tag) const (binding deprecated, see section XYZ) }

```

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.*)

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, `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.

In C++, the `status` object is handled through the following methods:

```
{ int MPI::Status::Get_source() const (binding deprecated, see section XYZ) }
{ void MPI::Status::Set_source(int source) (binding deprecated, see section XYZ) }
{ int MPI::Status::Get_tag() const (binding deprecated, see section XYZ) }
{ void MPI::Status::Set_tag(int tag) (binding deprecated, see section XYZ) }
{ int MPI::Status::Get_error() const (binding deprecated, see section XYZ) }
{ void MPI::Status::Set_error(int error) (binding deprecated, see section XYZ) }
```

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

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

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

`{ int MPI::Status::Get_count(const MPI::Datatype& datatype) const (binding deprecated, see section XYZ) }`

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. (We shall later see, in Section 4.1.11, that `MPI_GET_COUNT` may return, in certain situations, the value `MPI_UNDEFINED`.)

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, 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_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.

There are no C++ bindings for `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE`. To allow an OUT or INOUT `MPI::Status` argument to be ignored, all MPI C++ bindings that have OUT or INOUT `MPI::Status` parameters are overloaded with a second version that omits the OUT or INOUT `MPI::Status` parameter.

Example 4.1 The C++ bindings for `MPI_PROBE` are:

```

1      void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const
2      void MPI::Comm::Probe(int source, int tag) const
3
4

```

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 4.2 Sender and receiver specify matching types.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
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 4.3 Sender and receiver do not specify matching types.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

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

Example 4.4 Sender and receiver specify communication of untyped values.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

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

1 Type MPI_CHARACTER

2 The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER,
3 rather than the entire character string stored in the variable. Fortran variables of type
4 CHARACTER or substrings are transferred as if they were arrays of characters. This is
5 illustrated in the example below.
6

7 **Example 4.5** Transfer of Fortran CHARACTERS.

```
8
9 CHARACTER*10 a
10 CHARACTER*10 b
11
12 CALL MPI_COMM_RANK(comm, rank, ierr)
13 IF (rank.EQ.0) THEN
14     CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
15 ELSE IF (rank.EQ.1) THEN
16     CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
17 END IF
18
```

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

21
22 *Rationale.* The alternative choice would be for MPI_CHARACTER to match a char-
23 acter of arbitrary length. This runs into problems.

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

32 This problem forces us to provide explicit information on character length with the
33 MPI call. One could add a length parameter to the type MPI_CHARACTER, but this
34 does not add much convenience and the same functionality can be achieved by defining
35 a suitable derived datatype. (*End of rationale.*)
36

37 *Advice to implementors.* Some compilers pass Fortran CHARACTER arguments as a
38 structure with a length and a pointer to the actual string. In such an environment,
39 the MPI call needs to dereference the pointer in order to reach the string. (*End of*
40 *advice to implementors.*)
41

42 4.3.2 Data Conversion

44 One of the goals of MPI is to support parallel computations across heterogeneous environ-
45 nments. Communication in a heterogeneous environment may require data conversions. We
46 use the following terminology.
47

48 **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 or 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, 4.2–4.4. 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 16.6 on page 33.

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 access and overwrite 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(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)
```

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

```
<type> BUF(*)
```

```
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
{ void MPI::Comm::Bsend(const void* buf, int count, const
                        MPI::Datatype& datatype, int dest, int tag) const (binding
                        deprecated, see section XYZ) }
```

Send in buffered mode.

1 MPI_SSEND (buf, count, datatype, dest, tag, comm)

2	IN	buf	initial address of send buffer (choice)
3			
4	IN	count	number of elements in send buffer (non-negative integer)
5			
6	IN	datatype	datatype of each send buffer element (handle)
7	IN	dest	rank of destination (integer)
8			
9	IN	tag	message tag (integer)
10	IN	comm	communicator (handle)

11
12 int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
13 int tag, MPI_Comm comm)

14 MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

15 <type> BUF(*)

16 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

17 { void MPI::Comm::Ssend(const void* buf, int count, const
18 MPI::Datatype& datatype, int dest, int tag) const (*binding*
19 *deprecated, see section XYZ*) }
20
21

22 Send in synchronous mode.

23
24 MPI_RSEND (buf, count, datatype, dest, tag, comm)

25			
26	IN	buf	initial address of send buffer (choice)
27	IN	count	number of elements in send buffer (non-negative integer)
28			
29	IN	datatype	datatype of each send buffer element (handle)
30			
31	IN	dest	rank of destination (integer)
32	IN	tag	message tag (integer)
33	IN	comm	communicator (handle)
34			

35
36 int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
37 int tag, MPI_Comm comm)

38 MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

39 <type> BUF(*)

40 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

41 { void MPI::Comm::Rsend(const void* buf, int count, const
42 MPI::Datatype& datatype, int dest, int tag) const (*binding*
43 *deprecated, see section XYZ*) }
44

45 Send in ready mode.

46 There is only one receive operation, but it matches any of the send modes. The receive
47 operation described in the last section is **blocking**: it returns only after the receive buffer
48

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 multi-threaded 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 access or modify a communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.

Rationale. We prohibit read accesses to a send buffer while it is being used, even though the send operation is not supposed to alter the content of this buffer. This may seem more stringent than necessary, but the additional restriction causes little loss of functionality and allows better performance on some systems — consider the case where data transfer is done by a DMA engine that is not cache-coherent with the main processor. (*End of rationale.*)

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 multi-threaded 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 multi-threaded, 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 4.6 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 4.7 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 multi-threaded 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 signalled 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 4.8 An exchange of messages.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
```

```

1  IF (rank.EQ.0) THEN
2      CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
3      CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
4  ELSE IF (rank.EQ.1) THEN
5      CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
6      CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
7  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 4.9 An errant attempt to exchange messages.

```

13 CALL MPI_COMM_RANK(comm, rank, ierr)
14 IF (rank.EQ.0) THEN
15     CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
16     CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
17 ELSE IF (rank.EQ.1) THEN
18     CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
19     CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
20 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 4.10 An exchange that relies on buffering.

```

29 CALL MPI_COMM_RANK(comm, rank, ierr)
30 IF (rank.EQ.0) THEN
31     CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
32     CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
33 ELSE IF (rank.EQ.1) THEN
34     CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
35     CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
36 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 4.10. 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> BUFFER(*)`

`INTEGER SIZE, IERROR`

`{ void MPI::Attach_buffer(void* buffer, int size) (binding deprecated, see section XYZ) }`

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.

```
1 MPI_BUFFER_DETACH(buffer_addr, size)
```

```
2     OUT     buffer_addr           initial buffer address (choice)
```

```
3     OUT     size                  buffer size, in bytes (non-negative integer)
```

```
5  
6 int MPI_Buffer_detach(void* buffer_addr, int* size)
```

```
7 MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
```

```
8     <type> BUFFER_ADDR(*)
```

```
9     INTEGER SIZE, IERROR
```

```
10  
11 { int MPI::Detach_buffer(void*& buffer) (binding deprecated, see section XYZ) }
```

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 4.11 Calls to attach and detach buffers.

```
18  
19 #define BUFFSIZE 10000
```

```
20 int size
```

```
21 char *buff;
```

```
22 MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
```

```
23 /* a buffer of 10000 bytes can now be used by MPI_Bsend */
```

```
24 MPI_Buffer_detach( &buff, &size);
```

```
25 /* Buffer size reduced to zero */
```

```
26 MPI_Buffer_attach( buff, size);
```

```
27 /* 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. (*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 send-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. A nonblocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of nonblocking sends is advantageous in these cases only if data copying can be concurrent 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)

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)
OUT	request	communication request (handle)

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

```
1 MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
```

```
2     <type> BUF(*)
```

```
3     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
4 { MPI::Request MPI::Comm::Isend(const void* buf, int count, const
5     MPI::Datatype& datatype, int dest, int tag) const (binding
6     deprecated, see section XYZ) }
```

```
7
8     Start a standard mode, nonblocking send.
```

```
9
10 MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)
```

```
12     IN          buf          initial address of send buffer (choice)
```

```
13     IN          count        number of elements in send buffer (non-negative inte-
14                                ger)
```

```
15
16     IN          datatype      datatype of each send buffer element (handle)
```

```
17     IN          dest          rank of destination (integer)
```

```
18     IN          tag           message tag (integer)
```

```
19     IN          comm          communicator (handle)
```

```
20     OUT         request       communication request (handle)
```

```
21
22
23 int MPI_IbSend(void* buf, int count, MPI_Datatype datatype, int dest,
24     int tag, MPI_Comm comm, MPI_Request *request)
```

```
25
26 MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
```

```
27     <type> BUF(*)
```

```
28     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
29 { MPI::Request MPI::Comm::IbSend(const void* buf, int count, const
30     MPI::Datatype& datatype, int dest, int tag) const (binding
31     deprecated, see section XYZ) }
```

```
32
33     Start a buffered mode, nonblocking send.
```

```
34
35 MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
```

```
36     IN          buf          initial address of send buffer (choice)
```

```
37     IN          count        number of elements in send buffer (non-negative inte-
38                                ger)
```

```
39
40     IN          datatype      datatype of each send buffer element (handle)
```

```
41     IN          dest          rank of destination (integer)
```

```
42     IN          tag           message tag (integer)
```

```
43     IN          comm          communicator (handle)
```

```
44     OUT         request       communication request (handle)
```

```
45
46
47
48
```

```

int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request *request)
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{ MPI::Request MPI::Comm::Issend(const void* buf, int count, const
    MPI::Datatype& datatype, int dest, int tag) const (binding
    deprecated, see section XYZ) }
Start a synchronous mode, nonblocking send.

```

```

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

int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request *request)
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{ MPI::Request MPI::Comm::Irsend(const void* buf, int count, const
    MPI::Datatype& datatype, int dest, int tag) const (binding
    deprecated, see section XYZ) }
Start a ready mode nonblocking send.

```

1	MPI_Irecv (buf, count, datatype, source, tag, comm, request)		
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	IN	source	rank of source (integer)
8			
9	IN	tag	message tag (integer)
10	IN	comm	communicator (handle)
11	OUT	request	communication request (handle)
12			

```

13 int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
14             int tag, MPI_Comm comm, MPI_Request *request)
15 
```

```

16 MPI_Irecv(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)

```

```

17 <type> BUF(*)
18 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
19 { MPI::Request MPI::Comm::Irecv(void* buf, int count, const
20                               MPI::Datatype& datatype, int source, int tag) const (binding
21                               deprecated, see section XYZ) }
22 
```

23 Start a nonblocking receive.

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

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

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

33
 34 *Advice to users.* To prevent problems with the argument copying and register opti-
 35 mization done by Fortran compilers, please note the hints in subsections “Problems
 36 Due to Data Copying and Sequence Association,” and “A Problem with Register
 37 Optimization” in Section 16.5.2 on pages 18 and 21. (*End of advice to users.*)

40 4.7.3 Communication Completion

41 The functions `MPI_WAIT` and `MPI_TEST` are used to complete a nonblocking communica-
 42 tion. The completion of a send operation indicates that the sender is now free to update the
 43 locations in the send buffer (the send operation itself leaves the content of the send buffer
 44 unchanged). It does not indicate that the message has been received, rather, it may have
 45 been buffered by the communication subsystem. However, if a **synchronous** mode send was
 46 used, the completion of the send operation indicates that a matching receive was initiated,
 47 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` and `MPI_GET_ELEMENTS` 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 (Status)

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

`MPI_WAIT(REQUEST, STATUS, IERROR)`

`INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR`

`{ void MPI::Request::Wait(MPI::Status& status) (binding deprecated, see section XYZ) }`

`{ void MPI::Request::Wait() (binding deprecated, see section XYZ) }`

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

The call returns, in **status**, information on the completed operation. The content of the status object for a receive operation can be accessed as described in Section 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 multi-threaded environment, a call to `MPI_WAIT` should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (*End of advice to implementors.*)

`MPI_TEST(request, flag, status)`

INOUT	request	communication request (handle)
OUT	flag	true if operation completed (logical)
OUT	status	status object (Status)

`int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`

`MPI_TEST(REQUEST, FLAG, STATUS, IERROR)`

LOGICAL FLAG

INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

`{ bool MPI::Request::Test(MPI::Status& status) (binding deprecated, see section XYZ) }`

`{ bool MPI::Request::Test() (binding deprecated, see section XYZ) }`

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

The return status object for a receive operation carries information that can be accessed as described in Section 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.*)

Rationale. The function `MPI_TEST` returns with `flag = true` exactly in those situations where the function `MPI_WAIT` returns; both functions return in such case the same value in `status`. Thus, a blocking `Wait` can be easily replaced by a nonblocking `Test`. (*End of rationale.*)

Example 4.12 Simple usage of nonblocking operations and `MPI_WAIT`.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
END IF
```

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

```
MPI_REQUEST_FREE(request)
    INOUT      request          communication request (handle)

int MPI_Request_free(MPI_Request *request)

MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

{ void MPI::Request::Free() (binding deprecated, see section XYZ) }
```

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

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

Advice to users. Once a request is freed by a call to `MPI_REQUEST_FREE`, it is not possible to check for the successful completion of the associated communication with calls to `MPI_WAIT` or `MPI_TEST`. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user — such an error must be treated as fatal. Questions arise as to how one knows when the operations have completed when using `MPI_REQUEST_FREE`. Depending on the program logic, there may be other ways in which the program knows that certain operations have completed and this makes usage of `MPI_REQUEST_FREE` practical. For example, an active send request could be freed when the logic of the program is such that the receiver sends a reply to the message sent — the arrival of the reply informs the sender that the send has completed and the send buffer can be reused. An active

receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (*End of advice to users.*)

Example 4.13 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
  CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
  CALL MPI_WAIT(req, status, ierr)
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 4.14 Message ordering for nonblocking operations.

```

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
  CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
  CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE IF (rank.EQ.1) THEN
  CALL MPI_Irecv(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
  CALL MPI_Irecv(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1, status, ierr)
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 4.15 An illustration of progress semantics.

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

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

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

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)
```

```

1  MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
2      INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
3      IERROR
4
5  { static int MPI::Request::Waitany(int count,
6      MPI::Request array_of_requests[], MPI::Status& status) (binding
7      deprecated, see section XYZ) }
8
9  { static int MPI::Request::Waitany(int count,
10     MPI::Request array_of_requests[]) (binding deprecated, see section
11     XYZ) }

```

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 communication. (The array is indexed from zero in C, and from one in Fortran.) If the request was allocated by a nonblocking communication operation, then it is deallocated and the request handle is set to `MPI_REQUEST_NULL`.

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

```

26 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 MPI_UNDEFINED if none completed (integer)
OUT	flag	true if one of the operations is complete (logical)
OUT	status	status object (Status)

```

36 int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
37     int *flag, MPI_Status *status)
38
39 MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
40     LOGICAL FLAG
41     INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
42     IERROR
43
44 { static bool MPI::Request::Testany(int count,
45     MPI::Request array_of_requests[], int& index,
46     MPI::Status& status) (binding deprecated, see section XYZ) }
47
48 { static bool MPI::Request::Testany(int count,
49     MPI::Request array_of_requests[], int& index) (binding

```

deprecated, see section XYZ) }

Tests for completion of either one or none of the operations associated with active handles. In the former case, it returns `flag = true`, returns in `index` the index of this request in the array, and returns in `status` the status of that operation; if the request was allocated by a nonblocking communication call then the request is deallocated and the handle is set to `MPI_REQUEST_NULL`. (The array is indexed from zero in C, and from one in Fortran.) In the latter case (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`.

Rationale. The function `MPI_TESTANY` returns with `flag = true` exactly in those situations where the function `MPI_WAITANY` returns; both functions return in that case the same values in the remaining parameters. Thus, a blocking `MPI_WAITANY` can be easily replaced by a nonblocking `MPI_TESTANY`. The same relation holds for the other pairs of Wait and Test functions defined in this section. (*End of rationale.*)

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

```
{ static void MPI::Request::Waitall(int count,
    MPI::Request array_of_requests[],
    MPI::Status array_of_statuses[]) (binding deprecated, see section
    XYZ) }
```

```
{ static void MPI::Request::Waitall(int count,
    MPI::Request array_of_requests[]) (binding deprecated, see section
    XYZ) }
```

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no

handle in the list is active). Both arrays have the same number of valid entries. The i -th entry in `array_of_statuses` is set to the return status of the i -th operation. Requests that were created by nonblocking communication operations are deallocated and the corresponding handles in the array are set to `MPI_REQUEST_NULL`. The list may contain null or inactive handles. The call 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 $i=0, \dots, \text{count}-1$, 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.*)

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

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

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

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

```
{ static bool MPI::Request::Testall(int count,
    MPI::Request array_of_requests[],
    MPI::Status array_of_statuses[]) (binding deprecated, see section
    XYZ) }
```

```
{ static bool MPI::Request::Testall(int count,
    MPI::Request array_of_requests[]) (binding deprecated, see section
    XYZ) }
```

Returns `flag = true` if all communications associated with active handles in the array have completed (this includes the case where no handle in the list is active). In this case, each status entry that corresponds to an active handle request is set to the status of the corresponding communication; if the request was allocated by a nonblocking communication call then it is deallocated, and the handle is set to `MPI_REQUEST_NULL`. Each status entry that corresponds to a null or inactive handle is set to empty.

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

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

`MPI_WAITSOME(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 <code>Status</code>)

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

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

{ static int MPI::Request::Waitsome(int incount,
    MPI::Request array_of_requests[], int array_of_indices[],
    MPI::Status array_of_statuses[]) (binding deprecated, see section
    XYZ) }

{ static int MPI::Request::Waitsome(int incount,
    MPI::Request array_of_requests[], int array_of_indices[])
    (binding deprecated, see section XYZ) }
```

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. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to `MPI_REQUEST_NULL`.

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

When one or more of the communications completed by `MPI_WAITSSOME` 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_TESTSSOME(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)

MPI_TESTSSOME(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

{ static int MPI::Request::Testsome(int incount,
    MPI::Request array_of_requests[], int array_of_indices[],
    MPI::Status array_of_statuses[]) (binding deprecated, see section
    XYZ) }

{ static int MPI::Request::Testsome(int incount,
    MPI::Request array_of_requests[], int array_of_indices[])
    (binding deprecated, see section XYZ) }

```

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_TESTSOME, and a matching send has been posted, then the receive will eventually succeed, unless the send is satisfied by another receive; and similarly for send requests.

Errors that occur during the execution of MPI_TESTSOME are handled as for MPI_WAITSUME.

Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use of MPI_TESTANY. The former returns information on all completed communications, with the latter, a new call is required for each communication that completes.

A server with multiple clients can use MPI_WAITSUME so as not to starve any client. Clients send messages to the server with service requests. The server calls MPI_WAITSUME with one receive request for each client, and then handles all receives that completed. If a call to MPI_WAITANY is used instead, then one client could starve while requests from another client always sneak in first. (*End of advice to users.*)

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

Example 4.16 Client-server code (starvation can occur).

```
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_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 4.17 Same code, using MPI_WAITSUME.

```
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)
```

```

1      CALL MPI_WAIT(request, status, ierr)
2  END DO
3  ELSE      ! rank=0 -- server code
4      DO i=1, size-1
5          CALL MPI_Irecv(a(1,i), n, MPI_REAL, i, tag,
6              comm, request_list(i), ierr)
7      END DO
8      DO WHILE(.TRUE.)
9          CALL MPI_WAITsome(size, request_list, numdone,
10              indices, statuses, ierr)
11         DO i=1, numdone
12             CALL DO_SERVICE(a(1, indices(i)))
13             CALL MPI_Irecv(a(1, indices(i)), n, MPI_REAL, 0, tag,
14                 comm, request_list(indices(i)), ierr)
15         END DO
16     END DO
17 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.

MPI_REQUEST_GET_STATUS(request, flag, status)

IN	request	request (handle)
OUT	flag	boolean flag, same as from MPI_TEST (logical)
OUT	status	MPI_STATUS object if flag is true (Status)

```

int MPI_Request_get_status(MPI_Request request, int *flag,
    MPI_Status *status)

```

```

MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

```

```

{ bool MPI::Request::Get_status(MPI::Status& status) const (binding deprecated,
    see section XYZ) }

```

```

{ bool MPI::Request::Get_status() const (binding deprecated, see section XYZ) }

```

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.

4.8 Probe and Cancel

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

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

`MPI_IPROBE(source, tag, comm, flag, status)`

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

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

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

```
LOGICAL FLAG
```

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

```
{ bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status) const
  (binding deprecated, see section XYZ) }
```

```
{ bool MPI::Comm::Iprobe(int source, int tag) const (binding deprecated, see
  section XYZ) }
```

`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 multi-threaded, it is the user's responsibility to ensure that the last condition holds.

The `source` argument of `MPI_PROBE` can be `MPI_ANY_SOURCE`, and the `tag` argument can be `MPI_ANY_TAG`, so that one can probe for messages from an arbitrary source and/or

with an arbitrary tag. However, a specific communication context must be provided with the `comm` argument.

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

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

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

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

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

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

`{ void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const`
(binding deprecated, see section XYZ) }

`{ void MPI::Comm::Probe(int source, int tag) const` *(binding deprecated, see*
section XYZ) }

`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.

Example 4.18 Use blocking probe to wait for an incoming message.

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

```

END DO
END IF

```

Each message is received with the right type.

Example 4.19 A similar program to the previous example, but now it has a problem.

```

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

```

We slightly modified example 4.18, using `MPI_ANY_SOURCE` as the `source` argument in the two receive calls in statements labeled 100 and 200. The program is now incorrect: the receive operation may receive a message that is distinct from the message probed by the preceding call to `MPI_PROBE`.

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

MPI_CANCEL(request)

```
IN      request      communication request (handle)
```

```
1 int MPI_Cancel(MPI_Request *request)
```

```
2 MPI_CANCEL(REQUEST, IERROR)
```

```
3     INTEGER REQUEST, IERROR
```

```
4 { void MPI::Request::Cancel() const (binding deprecated, see section XYZ) }
```

A call to `MPI_CANCEL` marks for cancellation a pending, nonblocking communication operation (send or receive). The cancel call is local. It returns immediately, possibly before the communication is actually canceled. It is still necessary to complete a communication that has been marked for cancellation, using a call to `MPI_REQUEST_FREE`, `MPI_WAIT` or `MPI_TEST` (or any of the derived operations).

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

`MPI_CANCEL` can be used to cancel a communication that uses a persistent request (see Section 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 canceled, in which case no part of the message was received at the destination. Then, any matching receive has to be satisfied by another send. If a receive is marked for cancellation, then it must be the case that either the receive completes normally, or that the receive is successfully canceled, in which case no part of the receive buffer is altered. Then, any matching send has to be satisfied by another receive.

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

```
35 MPI_TEST_CANCELLED(status, flag)
```

```
36     IN      status          status object (Status)
```

```
37     OUT    flag            (logical)
```

```
40 int MPI_Test_cancelled(MPI_Status *status, int *flag)
```

```
41 MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
```

```
42     LOGICAL FLAG
```

```
43     INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```

```
44 { bool MPI::Status::Is_cancelled() const (binding deprecated, see section XYZ) }
```

Returns `flag = true` if the communication associated with the status object was canceled successfully. In such a case, all other fields of `status` (such as `count` or `tag`) are undefined.

Returns `flag = false`, otherwise. If a receive operation might be canceled then one should call `MPI_TEST_CANCELLED` first, to check whether the operation was canceled, before checking on the other fields of the return status.

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

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

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.

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

IN	<code>buf</code>	initial address of send buffer (choice)
IN	<code>count</code>	number of elements sent (non-negative integer)
IN	<code>datatype</code>	type of each element (handle)
IN	<code>dest</code>	rank of destination (integer)
IN	<code>tag</code>	message tag (integer)
IN	<code>comm</code>	communicator (handle)
OUT	<code>request</code>	communication request (handle)

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

```

1 MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
2     <type> BUF(*)
3     INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
4 { MPI::Prequest MPI::Comm::Send_init(const void* buf, int count, const
5     MPI::Datatype& datatype, int dest, int tag) const (binding
6     deprecated, see section XYZ) }
7

```

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

```

11 MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)
12

```

13	IN	buf	initial address of send buffer (choice)
14	IN	count	number of elements sent (non-negative integer)
15	IN	datatype	type of each element (handle)
16	IN	dest	rank of destination (integer)
17	IN	tag	message tag (integer)
18	IN	comm	communicator (handle)
19	IN	request	communication request (handle)

```

23 int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
24     int tag, MPI_Comm comm, MPI_Request *request)
25

```

```

26 MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
27     <type> BUF(*)
28     INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
29 { MPI::Prequest MPI::Comm::Bsend_init(const void* buf, int count, const
30     MPI::Datatype& datatype, int dest, int tag) const (binding
31     deprecated, see section XYZ) }
32

```

Creates a persistent communication request for a buffered mode send.

```

35 MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)
36

```

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

```

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

```



```

MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{ MPI::Prerequest MPI::Comm::Ssend_init(const void* buf, int count, const
    MPI::Datatype& datatype, int dest, int tag) const (binding
    deprecated, see section XYZ) }

```

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)

```

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

```

```

MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{ MPI::Prerequest MPI::Comm::Rsend_init(const void* buf, int count, const
    MPI::Datatype& datatype, int dest, int tag) const (binding
    deprecated, see section XYZ) }

```

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

```

MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)

```

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements received (non-negative integer)
IN	datatype	type of each element (handle)
IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```

int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm, MPI_Request *request)

```

```

1 MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
2     <type> BUF(*)
3     INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

```

```

4 { MPI::Prequest MPI::Comm::Recv_init(void* buf, int count, const
5     MPI::Datatype& datatype, int source, int tag) const (binding
6     deprecated, see section XYZ) }
7

```

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`.

```

16 MPI_START(request)

```

```

18     INOUT    request                communication request (handle)
19

```

```

20 int MPI_Start(MPI_Request *request)
21

```

```

22 MPI_START(REQUEST, IERROR)
23     INTEGER REQUEST, IERROR

```

```

24 { void MPI::Prequest::Start() (binding deprecated, see section XYZ) }
25

```

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 accessed 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.

```

37 MPI_STARTALL(count, array_of_requests)

```

```

39     IN        count                list length (non-negative integer)
40     INOUT    array_of_requests     array of requests (array of handle)
41

```

```

42 int MPI_Startall(int count, MPI_Request *array_of_requests)
43

```

```

44 MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
45     INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR

```

```

46 { static void MPI::Prequest::Startall(int count,
47     MPI::Prequest array_of_requests[]) (binding deprecated, see section
48

```

XYZ) }

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 subsections “Problems Due to Data Copying and Sequence Association,” and “A Problem with Register Optimization” in Section 16.5.2 on pages 18 and 21. (*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.

```

1 MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype,
2 source, recvtag, comm, status)

```

3	IN	sendbuf	initial address of send buffer (choice)
4			
5	IN	sendcount	number of elements in send buffer (non-negative integer)
6			
7	IN	sendtype	type of elements in send buffer (handle)
8	IN	dest	rank of destination (integer)
9			
10	IN	sendtag	send tag (integer)
11	OUT	recvbuf	initial address of receive buffer (choice)
12	IN	recvcount	number of elements in receive buffer (non-negative integer)
13			
14	IN	recvtype	type of elements in receive buffer (handle)
15			
16	IN	source	rank of source (integer)
17	IN	recvtag	receive tag (integer)
18			
19	IN	comm	communicator (handle)
20	OUT	status	status object (Status)

```

21
22 int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
23                 int dest, int sendtag, void *recvbuf, int recvcount,
24                 MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
25                 MPI_Status *status)

```

```

26
27 MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
28             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
29 <type> SENDBUF(*), RECVBUF(*)
30 INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, REVCOUNT, RECVTYPE,
31 SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

```

```

32 { void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const
33                             MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,
34                             int recvcount, const MPI::Datatype& recvtype, int source,
35                             int recvtag, MPI::Status& status) const (binding deprecated, see
36                             section XYZ) }

```

```

37
38 { void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const
39                             MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,
40                             int recvcount, const MPI::Datatype& recvtype, int source,
41                             int recvtag) const (binding deprecated, see section XYZ) }

```

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.

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

INOUT	buf	initial address of send and receive buffer (choice)
IN	count	number of elements in send and receive buffer (non-negative integer)
IN	datatype	type of elements in send and receive buffer (handle)
IN	dest	rank of destination (integer)
IN	sendtag	send message tag (integer)
IN	source	rank of source (integer)
IN	recvtag	receive message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
                        int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
                        MPI_Status *status)
```

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

```
<type> BUF(*)
```

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

```
{ void MPI::Comm::Sendrecv_replace(void* buf, int count, const
    MPI::Datatype& datatype, int dest, int sendtag, int source,
    int recvtag, MPI::Status& status) const (binding deprecated, see
    section XYZ) }
```

```
{ void MPI::Comm::Sendrecv_replace(void* buf, int count, const
    MPI::Datatype& datatype, int dest, int sendtag, int source,
    int recvtag) const (binding deprecated, see section XYZ) }
```

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

1 from `MPI_PROC_NULL` succeeds and returns as soon as possible with no modifications to
2 the receive buffer. When a receive with `source = MPI_PROC_NULL` is executed then the
3 status object returns `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG` and `count = 0`.
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

Index

([&array_of_requests\[i\]](#)), [49](#)

[MPI_WAIT](#), [30](#)

[CONST: MPI_ANY_TAG](#), [6](#)

[CONST: MPI_BYTE](#), [10](#)

[CONST: MPI_PROC_NULL](#), [51](#), [52](#)

[CONST:a](#), [12](#)

[CONST:b](#), [12](#)

[CONST:char**](#), [22](#)

[CONST:CHARACTER](#), [11](#), [12](#)

[CONST:DOUBLE COMPLEX](#), [3](#)

[CONST:false](#), [29](#)

[CONST:INTEGER*1](#), [3](#)

[CONST:INTEGER*2](#), [3](#)

[CONST:INTEGER*4](#), [3](#)

[CONST:MPI::Comm](#), [2](#)

[CONST:MPI::Prequest](#), [46](#)

[CONST:MPI::Request](#), [25–28](#), [29](#), [30](#), [31](#), [33–38](#), [40](#), [43](#), [45–48](#)

[CONST:MPI::Status](#), [5](#), [8](#), [29](#), [30](#), [33–38](#), [40–42](#), [44](#), [50](#), [51](#)

[CONST:MPI_ANY_SOURCE](#), [6](#), [18](#), [41–43](#), [47](#)

[CONST:MPI_ANY_TAG](#), [6](#), [8](#), [41–43](#), [47](#), [52](#)

[CONST:MPI_BOTTOM](#), [9](#)

[CONST:MPI_BSEND_OVERHEAD](#), [23](#)

[CONST:MPI_BYTE](#), [3](#), [4](#), [10](#), [11](#), [13](#)

[CONST:MPI_CHAR](#), [4](#), [13](#)

[CONST:MPI_CHARACTER](#), [3](#), [12](#), [13](#)

[CONST:MPI_Comm](#), [2](#)

[CONST:MPI_COMM_WORLD](#), [5](#)

[CONST:MPI_COMPLEX](#), [3](#)

[CONST:MPI_DOUBLE](#), [4](#)

[CONST:MPI_DOUBLE_COMPLEX](#), [3](#)

[CONST:MPI_DOUBLE_PRECISION](#), [3](#)

[CONST:MPI_ERR_IN_STATUS](#), [7](#), [9](#), [29](#), [36](#), [38](#)

[CONST:MPI_ERR_PENDING](#), [36](#)

[CONST:MPI_ERROR](#), [7](#), [29](#)

[CONST:MPI_FLOAT](#), [4](#)

[CONST:MPI_INT](#), [4](#)

[CONST:MPI_INTEGER](#), [3](#), [10](#)

[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_PACKED](#), [3](#), [4](#), [10](#)

[CONST:MPI_PROC_NULL](#), [2](#), [52](#)

[CONST:MPI_REAL](#), [3](#), [10](#)

[CONST:MPI_REAL2](#), [3](#)

[CONST:MPI_REAL4](#), [3](#)

[CONST:MPI_REAL8](#), [3](#)

[CONST:MPI_Request](#), [25–28](#), [29](#), [30](#), [31](#), [33–38](#), [40](#), [43](#), [45–48](#)

[CONST:MPI_REQUEST_NULL](#), [29–31](#), [34–37](#)

[CONST:MPI_SHORT](#), [4](#)

[CONST:MPI_SIGNED_CHAR](#), [4](#)

[CONST:MPI_SOURCE](#), [7](#)

[CONST:MPI_STATUS](#), [9](#), [29](#)

[CONST:MPI_Status](#), [5](#), [8](#), [29](#), [30](#), [33–38](#), [40–42](#), [44](#), [50](#), [51](#)

[CONST:MPI_STATUS_IGNORE](#), [9](#)

[CONST:MPI_STATUS_SIZE](#), [7](#)

[CONST:MPI_STATUSES_IGNORE](#), [9](#)

[CONST:MPI_SUCCESS](#), [36](#), [38](#)

[CONST:MPI_TAG](#), [7](#)

[CONST:MPI_TAG_UB](#), [4](#)

[CONST:MPI_UNDEFINED](#), [8](#), [34](#), [35](#)

[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](#)

- 1 CONST:NULL, [9](#)
- 2 CONST:REAL*2, [3](#)
- 3 CONST:REAL*4, [3](#)
- 4 CONST:REAL*8, [3](#)
- 5 CONST:true, [34](#)
- 6 CONST:void*, [22](#)
- 7 CONST:void**, [22](#)
- 8
- 9 EXAMPLES:Client-server code, [39](#)
- 10 EXAMPLES:Client-server code with block-
- 11 ing probe, [42](#)
- 12 EXAMPLES:Client-server code with block-
- 13 ing probe, wrong, [43](#)
- 14 EXAMPLES:Datypes - matching, [11](#)
- 15 EXAMPLES:Datypes - not matching, [11](#)
- 16 EXAMPLES:Datypes - untyped, [11](#)
- 17 EXAMPLES:Deadlock, if not buffered, [20](#)
- 18 EXAMPLES:Deadlock, wrong message ex-
- 19 change, [20](#)
- 20 EXAMPLES:Intertwined matching pairs, [18](#)
- 21 EXAMPLES:Message exchange (ping-pong),
- 22 [19](#)
- 23 EXAMPLES:MPI::Comm::Probe, [9](#)
- 24 EXAMPLES:MPI_BSEND, [18](#)
- 25 EXAMPLES:MPI_Buffer_attach, [22](#)
- 26 EXAMPLES:MPI_Buffer_detach, [22](#)
- 27 EXAMPLES:MPI_BYTE, [11](#)
- 28 EXAMPLES:MPI_CHARACTER, [12](#)
- 29 EXAMPLES:MPI_IRECV, [31–33](#), [39](#)
- 30 EXAMPLES:MPI_ISEND, [31](#), [32](#), [39](#)
- 31 EXAMPLES:MPI_PROBE, [42](#), [43](#)
- 32 EXAMPLES:MPI_RECV, [11](#), [12](#), [18–20](#), [33](#),
- 33 [42](#), [43](#)
- 34 EXAMPLES:MPI_REQUEST_FREE, [32](#)
- 35 EXAMPLES:MPI_SEND, [11](#), [12](#), [19](#), [20](#), [33](#),
- 36 [42](#), [43](#)
- 37 EXAMPLES:MPI_SSEND, [18](#), [33](#)
- 38 EXAMPLES:MPI_WAIT, [31–33](#), [39](#)
- 39 EXAMPLES:MPI_WAITANY, [39](#)
- 40 EXAMPLES:MPI_WAIT SOME, [39](#)
- 41 EXAMPLES:Non-overtaking messages, [18](#)
- 42 EXAMPLES:Nonblocking operations, [31](#), [32](#)
- 43 EXAMPLES:Nonblocking operations - mes-
- 44 sage ordering, [32](#)
- 45 EXAMPLES:Nonblocking operations - progress,
- 46 [33](#)
- 47 MPI_BSEND, [23](#)
- 48 MPI_BSEND (buf, count, datatype, dest, tag,
- comm), [15](#)
- MPI_BSEND_INIT, [48](#)
- MPI_BSEND_INIT(buf, count, datatype, dest,
- tag, comm, request), [46](#)
- MPI_BUFFER_ATTACH, [30](#)
- MPI_BUFFER_ATTACH(buffer, size), [21](#)
- MPI_BUFFER_DETACH(buffer_addr, size),
- [22](#)
- MPI_CANCEL, [18](#), [30](#), [41](#), [44](#), [45](#)
- MPI_CANCEL(request), [43](#)
- MPI_GET_COUNT, [8](#), [9](#), [29](#)
- MPI_GET_COUNT(status, datatype, count),
- [8](#)
- MPI_GET_ELEMENTS, [29](#)
- MPI_IBSEND, [30](#), [48](#)
- MPI_IBSEND(buf, count, datatype, dest, tag,
- comm, request), [26](#)
- MPI_IPROBE, [8](#), [41](#), [42](#)
- MPI_IPROBE(source, tag, comm, flag, sta-
- tus), [41](#), [41](#)
- MPI_Irecv (buf, count, datatype, source,
- tag, comm, request), [28](#)
- MPI_IRSEND(buf, count, datatype, dest, tag,
- comm, request), [27](#)
- MPI_ISEND, [48](#)
- MPI_ISEND(buf, count, datatype, dest, tag,
- comm, request), [25](#)
- MPI_ISSEND(buf, count, datatype, dest, tag,
- comm, request), [26](#)
- MPI_PACK, [23](#)
- MPI_PACK_SIZE(count, datatype, comm, size),
- [23](#)
- MPI_PROBE, [6](#), [8](#), [9](#), [41–43](#)
- MPI_PROBE(source, tag, comm, status), [42](#),
- [43](#)
- MPI_RECV, [2](#), [7–9](#)
- MPI_RECV (buf, count, datatype, source,
- tag, comm, status), [5](#)
- MPI_RECV(), [41](#)
- MPI_RECV(..., source, tag, comm, status),
- [41](#), [43](#)
- MPI_RECV_INIT, [48](#)
- MPI_RECV_INIT(buf, count, datatype, source,
- tag, comm, request), [47](#)
- MPI_REQUEST_FREE, [31](#), [32](#), [44](#), [49](#)
- MPI_REQUEST_FREE(request), [31](#)
- MPI_REQUEST_GET_STATUS, [9](#)

MPI_REQUEST_GET_STATUS(request, flag,	MPI_WAIT(&array_of_request[i], &array_of_statuses[i],	
status),	40	36
MPI_RSEND (buf, count, datatype, dest, tag,	MPI_WAIT(&array_of_requests[i], status),	34
comm),	16	MPI_WAIT(request, status), 29
MPI_RSEND_INIT(buf, count, datatype, dest,	MPI_WAITALL, 33 , 36 , 37	
tag, comm, request),	47	MPI_WAITALL(count, array_of_requests,
MPI_SEND, 1 , 3 , 9 , 11		array_of_statuses), 35
MPI_SEND(buf, count, datatype, dest, tag,	MPI_WAITALL(count, array_of_requests, ar-	
comm),	2	ray_of_statuses), 36
MPI_SEND_INIT, 48		MPI_WAITANY, 18 , 29 , 33–35 , 39
MPI_SEND_INIT(buf, count, datatype, dest,	MPI_WAITANY (count, array_of_requests,	
tag, comm, request),	45	index, status), 33
MPI_SENDRECV(sendbuf, sendcount, send-	MPI_WAITANY(count, array_of_requests, in-	
type, dest, sendtag, recvbuf, recv-	dex, status), 34	
count, recvtype, source, recvtage, comm,	MPI_WAIT SOME, 33 , 38 , 39	
status),	50	MPI_WAIT SOME(incount, array_of_requests,
MPI_SENDRECV_REPLACE(buf, count, datatype,	outcount, array_of_indices, array_of_statuses),	
dest, sendtag, source, recvtage, comm,	37	
status),	51	
MPI_SSEND (buf, count, datatype, dest, tag,		
comm),	16	
MPI_SSEND_INIT(buf, count, datatype, dest,		
tag, comm, request),	46	
MPI_START, 48 , 49		
MPI_START(request),	48	
MPI_STARTALL, 49		
MPI_STARTALL(count, array_of_requests),		
48 , 49		
MPI_TEST, 9 , 28–31 , 33 , 35 , 44 , 49		
MPI_TEST(&array_of_requests[i], flag, sta-		
tus),	35	
MPI_TEST(request, flag, status),	30	
MPI_TEST_CANCELLED, 29 , 30 , 45		
MPI_TEST_CANCELLED(status, flag),	44	
MPI_TESTALL, 33 , 37		
MPI_TESTALL(count, array_of_requests, flag,		
array_of_statuses),	36	
MPI_TESTANY, 29 , 33 , 35 , 39		
MPI_TESTANY(count, array_of_requests, in-		
dex, flag, status),	34	
MPI_TESTANY(count, array_of_requests, in-		
dex, status),	35	
MPI_TEST SOME, 33 , 38 , 39		
MPI_TEST SOME(incount, array_of_requests,		
outcount, array_of_indices, array_of_statuses),	38	
MPI_TYPE_CREATE_DARRAY, 8		
MPI_WAIT, 7 , 9 , 28–31 , 33 , 34 , 36 , 44 , 49		