# $D \ R \ A \ F \ T$ Document for a Standard Message-Passing Interface

Message Passing Interface Forum

July 30, 2019

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**

#### Introduction 4.1

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"
                                                                                     20
int main(int argc, char *argv[])
                                                                                     21
{
                                                                                     22
  char message[20];
                                                                                     23
                                                                                     24
  int myrank;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                                                                                     27
  if (myrank == 0)
                       /* code for process zero */
                                                                                     28
  {
                                                                                     29
      strcpy(message,"Hello, there");
                                                                                     30
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
                                                                                     31
  }
  else if (myrank == 1) /* code for process one */
                                                                                     33
                                                                                     34
  {
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
                                                                                     35
      printf("received :%s:\n", message);
                                                                                     36
  }
                                                                                     37
 MPI_Finalize();
                                                                                     38
                                                                                     39
  return 0;
}
```

In this example, process zero (myrank = 0) sends a message to process one using the 42send operation MPI\_SEND. The operation specifies a send buffer in the sender memory 43from which the message data is taken. In the example above, the send buffer consists of 44 the storage containing the variable **message** in the memory of process zero. The location, 45size and type of the send buffer are specified by the first three parameters of the send 46 operation. The message sent will contain the 13 characters of this variable. In addition, 47the send operation associates an **envelope** with the message. This envelope specifies the 48

1516

17

18

19

25

26

32

message destination and contains distinguishing information that can be used by the **receive** 1 operation to select a particular message. The last three parameters of the send operation,  $^{2}$ along with the rank of the sender, specify the envelope for the message sent. Process one 3 (myrank = 1) receives this message with the receive operation MPI\_RECV. The message to 4 be received is selected according to the value of its envelope, and the message data is stored  $\mathbf{5}$ into the **receive buffer**. In the example above, the receive buffer consists of the storage 6 containing the string message in the memory of process one. The first three parameters 7 of the receive operation specify the location, size and type of the receive buffer. The next 8 three parameters are used for selecting the incoming message. The last parameter is used 9 to return information on the message just received. 10

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

- 4.2 Blocking Send and Receive Operations
- 4.2.1 Blocking Send

17 18

19 20

21

22 23 24

- The syntax of the blocking send operation is given below.
- MPI\_SEND(buf, count, datatype, dest, tag, comm)

26	IN	buf	initial address of send buffer (choice)
27 28	IN	count	number of elements in send buffer (non-negative integer)
29 30	IN	datatype	datatype of each send buffer element (handle)
31	IN	dest	rank of destination (integer)
32	IN	tag	message tag (integer)
33 34	IN	comm	communicator (handle)
35 36 37	int MPI	_Send(const void *h int tag, MPL	ouf, int count, MPI_Datatype datatype, int dest, _Comm comm)
38 39 40	int MPI		ouf, MPI_Count count, MPI_Datatype datatype, t tag, MPI_Comm comm)
40 41 42	int MPI		<pre>*buf, MPI_Count count, MPI_Datatype datatype, t tag, MPI_Comm comm)</pre>
43 44 45 46 47 48	TYP INT TYP	E(*), DIMENSION() EGER, INTENT(IN) ::	type, dest, tag, comm, ierror) ), INTENT(IN) :: buf : count, dest, tag NTENT(IN) :: datatype F(IN) :: comm

INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_Send(buf, count, datatype, dest, tag, comm, ierror) TYPE(\*), DIMENSION(..), INTENT(IN) :: buf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: dest, tag TYPE(MPI\_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(\*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

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

#### 4.2.2 Message Data

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

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

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

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

1	MPI datatype	C datatype
2	MPI_CHAR	char
3		(treated as printable character)
4	MPI_SHORT	signed short int
5	MPI_INT	signed int
6	MPI_LONG	signed long int
7	MPI_LONG_LONG_INT	signed long long int
8	MPI_LONG_LONG (as a synonym)	signed long long int
9	MPI_SIGNED_CHAR	signed char
10		(treated as integral value)
11	MPI_UNSIGNED_CHAR	unsigned char
12		(treated as integral value)
13	MPI_UNSIGNED_SHORT	unsigned short int
14	MPI_UNSIGNED	unsigned int
15	MPI_UNSIGNED_LONG	unsigned long int
16	MPI_UNSIGNED_LONG_LONG	unsigned long long int
17	MPI_FLOAT	float
18	MPI_DOUBLE	double
19	MPI_LONG_DOUBLE	long double
20	MPI_WCHAR	wchar_t
21		(defined in <stddef.h>)</stddef.h>
22		(treated as printable character)
23	MPI_C_BOOL	_Bool
24	MPI_INT8_T	int8_t
25	MPI_INT16_T	int16_t
26	MPI_INT32_T	int32_t
27	MPI_INT64_T	int64_t
28	MPI_UINT8_T	uint8_t
29	MPI_UINT16_T	uint16_t
30	MPI_UINT32_T	uint32_t
31	MPI_UINT64_T	uint64_t
32	MPI_C_COMPLEX	float _Complex
33	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
34	MPI_C_DOUBLE_COMPLEX	double _Complex
35	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
36	MPI_BYTE	
37	MPI_PACKED	
38		1
39		
40	Table 4.2: Predefined MPI datatypes co	prresponding to C datatypes
41		

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.

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

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

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

The datatypes MPI\_AINT, MPI\_OFFSET, and MPI\_COUNT correspond to the MPIdefined C types MPI\_Aint, MPI\_Offset, and MPI\_Count and their Fortran equivalents INTEGER (KIND=MPI\_ADDRESS\_KIND), INTEGER (KIND=MPI\_OFFSET\_KIND), and INTEGER (KIND=MPI\_COUNT\_KIND). This is described in Table 4.3. All predefined datatype handles are available in all language bindings. See Sections 17.2.6 and ?? on page 656 and ?? for information on interlanguage communication with these types.

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

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

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

#### 4.2.3 Message Envelope

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

source	
destination	
$\operatorname{tag}$	
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 48

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  $0, \ldots, n-1 \cup \{\text{MPI}\_\text{PROC}\_\text{NULL}\}$ , where *n* is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 6.)

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

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

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

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

#### 4.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

38 39

18

19

20

21

22

23

28

29

30 31

32

33 34 35

36

37

- 40
- 41 42

- 44
- 45
- 46
- 47 48

MPI_RE	CV(buf, count, datatyp	be, source, tag, comm, status)	1
OUT	buf	initial address of receive buffer (choice)	2
IN	count	number of elements in receive buffer (non-negative in-teger)	3 4 5
IN	datatype	datatype of each receive buffer element (handle)	6
IN	source	rank of source or MPI_ANY_SOURCE (integer)	7
IN	tag	message tag or MPI_ANY_TAG (integer)	8 9
IN	comm	communicator (handle)	9 10
OUT	status	status object (Status)	11
001	Status	status object (Status)	12
int MPI		nt count, MPI_Datatype datatype, int source, E_Comm comm, MPI_Status *status)	13 14
int MDT	Becu(woid *huf M	PI_Count count, MPI_Datatype datatype, int source,	15 16
IIIC MFI		C_Comm comm, MPI_Status *status)	17
int MDT	-		18
IIIC MPI		MPI_Count count, MPI_Datatype datatype, int tag, MPI_Comm comm, MPI_Status *status)	19
		-	20
	v(buf, count, data E(*), DIMENSION(	type, source, tag, comm, status, ierror)	21 22
		: count, source, tag	23
		NTENT(IN) :: datatype	24
	E(MPI_Comm), INTEN	• -	25
	E(MPI_Status) ::		26
INT	EGER, OPTIONAL, IN	TENT(OUT) :: ierror	27
MPI_Rec	v(buf, count, data	type, source, tag, comm, status, ierror)	28 29
	E(*), DIMENSION(		30
		T_KIND), INTENT(IN) :: count	31
		NTENT(IN) :: datatype	32
	EGER, INTENT(IN) : E(MPI_Comm), INTEN	-	33
	E(MPI_Status) ::		34
	EGER, OPTIONAL, IN		35
			36 37
	v(BUF, CUUNI, DAIA pe> BUF(*)	TYPE, SOURCE, TAG, COMM, STATUS, IERROR)	38
U	-	PE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),	39
	ROR		40
The	blocking computies	f this call are described in Section 4.4.	41
THE	biocking semantics 0.	T THIS CAN ALE DESCRIPED IN DECUON 4.4.	42

MDL DECV//huf assurt de `

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.*)

14 The selection of a message by a receive operation is governed by the value of the 15message envelope. A message can be received by a receive operation if its envelope matches 16the source, tag and comm values specified by the receive operation. The receiver may 17specify a wildcard MPI\_ANY\_SOURCE value for source, and/or a wildcard MPI\_ANY\_TAG 18 value for tag, indicating that any source and/or tag are acceptable. It cannot specify a 19 wildcard value for comm. Thus, a message can be received by a receive operation only 20if it is addressed to the receiving process, has a matching communicator, has matching 21source unless source=MPI\_ANY\_SOURCE in the pattern, and has a matching tag unless 22tag=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  $\{0, ..., n-1\} \cup$ {MPI\_ANY\_SOURCE} $\cup$ {MPI\_PROC\_NULL}, where *n* is the number of processes in this group.

<sup>28</sup> Note the asymmetry between send and receive operations: A receive operation may <sup>29</sup> accept messages from an arbitrary sender, on the other hand, a send operation must specify <sup>30</sup> a unique receiver. This matches a "push" communication mechanism, where data transfer <sup>31</sup> is effected by the sender (rather than a "pull" mechanism, where data transfer is effected <sup>32</sup> by the receiver).

<sup>33</sup> Source = destination is allowed, that is, a process can send a message to itself. (How-<sup>34</sup> ever, it is unsafe to do so with the blocking send and receive operations described above, <sup>35</sup> since this may lead to deadlock. See Section 4.5.)

Advice to implementors. Message context and other communicator information can be implemented as an additional tag field. It differs from the regular message tag in that wild card matching is not allowed on this field, and that value setting for this field is controlled by communicator manipulation functions. (*End of advice to implementors.*)

The use of dest or source=MPI\_PROC\_NULL to define a "dummy" destination or source in any send or receive call is described in Section 4.11.

#### 4.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function

#### **Unofficial Draft for Comment Only**

1

2 3

4

5

6

7

8

9

10

11

12

13

37

38

39

40

41 42

43

44 45

(see Section 4.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI\_RECV. The type of status is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, status is a structure that contains three fields named MPI\_SOURCE, MPI\_TAG, and MPI\_ERROR; the structure may contain additional fields. Thus,

status.MPI\_SOURCE, status.MPI\_TAG and status.MPI\_ERROR contain the source, tag, and error code, respectively, of the received message.

In Fortran with USE mpi or INCLUDE 'mpif.h', status is an array of INTEGERS of size MPI\_STATUS\_SIZE. The constants MPI\_SOURCE, MPI\_TAG and MPI\_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI\_SOURCE), status(MPI\_TAG) and status(MPI\_ERROR) contain, respectively, the source, tag and error code of the received message.

With Fortran USE mpi\_f08, status is defined as the Fortran BIND(C) derived type TYPE(MPI\_Status) containing three public INTEGER fields named MPI\_SOURCE, MPI\_TAG, and MPI\_ERROR. TYPE(MPI\_Status) may contain additional, implementation-specific fields. Thus, status%MPI\_SOURCE, status%MPI\_TAG and status%MPI\_ERROR contain the source, tag, and error code of a received message respectively. Additionally, within both the mpi and the mpi\_f08 modules, the constants MPI\_STATUS\_SIZE, MPI\_SOURCE, MPI\_TAG, MPI\_ERROR, and TYPE(MPI\_Status) are defined to allow conversion between both status representations. Conversion routines are provided in Section 17.2.5.

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

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

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

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

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to MPI\_GET\_COUNT is required to "decode" this information.

 $\overline{7}$ 

MPI\_GET\_COUNT(status, datatype, count) 1  $\mathbf{2}$ IN status return status of receive operation (Status) 3 IN datatype of each receive buffer entry (handle) datatype 4 OUT number of received entries (non-negative integer) 5count 6 7 int MPI\_Get\_count(const MPI\_Status \*status, MPI\_Datatype datatype, 8 int \*count) 9 int MPI\_Get\_count(const MPI\_Status \*status, MPI\_Datatype datatype, 10 MPI\_Count \*count) 11 12int MPI\_Get\_count\_x(const MPI\_Status \*status, MPI\_Datatype datatype, 13 MPI\_Count \*count) 14 MPI\_Get\_count(status, datatype, count, ierror) 15 TYPE(MPI\_Status), INTENT(IN) :: status 16 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 17INTEGER, INTENT(OUT) :: count 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 20MPI\_Get\_count(status, datatype, count, ierror) 21TYPE(MPI\_Status), INTENT(IN) :: status 22 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 23 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: count 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25MPI\_GET\_COUNT(STATUS, DATATYPE, COUNT, IERROR) 26INTEGER STATUS(MPI\_STATUS\_SIZE), DATATYPE, COUNT, IERROR 2728Returns the number of entries received. (Again, we count *entries*, each of type *datatype*, 29 not bytes.) The datatype argument should match the argument provided by the receive call 30 that set the status variable. If the number of entries received exceeds the limits of the count 31 parameter, then MPI\_GET\_COUNT sets the value of count to MPI\_UNDEFINED. There are 32 other situations where the value of count can be set to MPI\_UNDEFINED; see Section 4.1.11. 33 34 Rationale. Some message-passing libraries use INOUT count, tag and 35 source arguments, thus using them both to specify the selection criteria for incoming 36 messages and return the actual envelope values of the received message. The use of a 37 separate status argument prevents errors that are often attached with INOUT argument 38 (e.g., using the MPI\_ANY\_TAG constant as the tag in a receive). Some libraries use 39 calls that refer implicitly to the "last message received." This is not thread safe. 40 The datatype argument is passed to MPI\_GET\_COUNT so as to improve performance. 41A message might be received without counting the number of elements it contains, 42 and the count value is often not needed. Also, this allows the same function to be 43used after a call to MPI\_PROBE or MPI\_IPROBE. With a status from MPI\_PROBE 44 or MPI\_IPROBE, the same datatypes are allowed as in a call to MPI\_RECV to receive 45 this message. (End of rationale.) 46 47The value returned as the count argument of MPI\_GET\_COUNT for a datatype of length 48

zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI\_UNDEFINED is returned.

*Rationale.* Zero-length datatypes may be created in a number of cases. An important case is MPI\_TYPE\_CREATE\_DARRAY, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style will not check for this special case and may want to use MPI\_GET\_COUNT to check the status. (*End of rationale.*)

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

All send and receive operations use the buf, count, datatype, source, dest, tag, comm, and status arguments in the same way as the blocking MPI\_SEND and MPI\_RECV operations described in this section.

### 4.2.6 Passing MPI\_STATUS\_IGNORE for Status

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

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

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

In general, this optimization can apply to all functions for which status or an array of statuses is an OUT argument. Note that this converts status into an INOUT argument. The functions that can be passed MPI\_STATUS\_IGNORE are all the various forms of MPI\_RECV, MPI\_PROBE, MPI\_TEST, and MPI\_WAIT, as well as MPI\_REQUEST\_GET\_STATUS. When an array is passed, as in the MPI\_{TEST|WAIT}{ALL|SOME} functions, a separate constant, MPI\_STATUSES\_IGNORE, is passed for the array argument. It is possible for an MPI function to return MPI\_ERR\_IN\_STATUS even when MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE has been passed to that function.

MPI\_STATUS\_IGNORE and MPI\_STATUSES\_IGNORE are not required to have the same values in C and Fortran.

It is not allowed to have some of the statuses in an array of statuses for MPI\_{TEST|WAIT}{ALL|SOME} functions set to MPI\_STATUS\_IGNORE; one either specifies  $^{2}$ ignoring all of the statuses in such a call with MPI\_STATUSES\_IGNORE, or *none* of them by 3 passing normal statuses in all positions in the array of statuses.

#### 4.3 Data Type Matching and Data Conversion

4.3.1 Type Matching Rules

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

1. Data is pulled out of the send buffer and a message is assembled.

2. A message is transferred from sender to receiver.

3. Data is pulled from the incoming message and disassembled into the receive buffer.

16Type matching has to be observed at each of these three phases: The type of each 17variable in the sender buffer has to match the type specified for that entry by the send 18 operation; the type specified by the send operation has to match the type specified by the 19 receive operation; and the type of each variable in the receive buffer has to match the type 20specified for that entry by the receive operation. A program that fails to observe these three 21rules is erroneous. 22

To define type matching more precisely, we need to deal with two issues: matching of 23types of the host language with types specified in communication operations; and matching 24of types at sender and receiver.

25The types of a send and receive match (phase two) if both operations use identical 26 names. That is, MPI\_INTEGER matches MPI\_INTEGER, MPI\_REAL matches MPI\_REAL, 27and so on. There is one exception to this rule, discussed in Section 4.2: the type 28 MPI\_PACKED can match any other type. 29

The type of a variable in a host program matches the type specified in the commu-30 nication operation if the datatype name used by that operation corresponds to the basic 31 type of the host program variable. For example, an entry with type name MPI\_INTEGER 32 matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran 33 and C appears in Section 4.2.2. There are two exceptions to this last rule: an entry with 34 type name MPI\_BYTE or MPI\_PACKED can be used to match any byte of storage (on a 35 byte-addressable machine), irrespective of the datatype of the variable that contains this 36 byte. The type MPI\_PACKED is used to send data that has been explicitly packed, or 37 receive data that will be explicitly unpacked, see Section 4.2. The type MPI\_BYTE allows 38 one to transfer the binary value of a byte in memory unchanged. 39

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.

1

4  $\mathbf{5}$ 6

7 8

9

10 11

1213

14

15

40  $^{41}$ 

42

43

44

45

46

47

```
• Communication involving packed data, where MPI_PACKED is used.
                                                                                           1
                                                                                           2
    The following examples illustrate the first two cases.
                                                                                           3
                                                                                           4
Example 4.1
                Sender and receiver specify matching types.
                                                                                           5
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                           6
IF (rank.EQ.0) THEN
                                                                                           7
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
                                                                                           8
ELSE IF (rank.EQ.1) THEN
                                                                                           9
    CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
                                                                                           10
                                                                                           11
END IF
                                                                                           12
    This code is correct if both a and b are real arrays of size \geq 10. (In Fortran, it might be
                                                                                           13
correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced
                                                                                           14
to an array with ten reals.)
                                                                                           15
                                                                                           16
Example 4.2
                Sender and receiver do not specify matching types.
                                                                                           17
                                                                                           18
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                           19
IF (rank.EQ.0) THEN
                                                                                           20
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
                                                                                           21
ELSE IF (rank.EQ.1) THEN
                                                                                           22
    CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
                                                                                           23
END IF
                                                                                           24
                                                                                           25
    This code is erroneous, since sender and receiver do not provide matching datatype
                                                                                           26
arguments.
                                                                                           27
                Sender and receiver specify communication of untyped values.
                                                                                           28
Example 4.3
                                                                                           29
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                           30
IF (rank.EQ.0) THEN
                                                                                           31
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
                                                                                           32
ELSE IF (rank.EQ.1) THEN
                                                                                           33
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
                                                                                           34
END IF
                                                                                           35
                                                                                           36
    This code is correct, irrespective of the type and size of a and b (unless this results in
                                                                                           37
an out of bounds memory access).
                                                                                           38
                                                                                           39
     Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND,
                                                                                           40
     then MPI will send the data stored at contiguous locations, starting from the address
                                                                                           41
     indicated by the buf argument. This may have unexpected results when the data
                                                                                           42
     layout is not as a casual user would expect it to be. For example, some Fortran
                                                                                           43
     compilers implement variables of type CHARACTER as a structure that contains the
                                                                                           44
     character length and a pointer to the actual string. In such an environment, sending
                                                                                           45
```

```
Unofficial Draft for Comment Only
```

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.)

46

47

1	Type MPI_CHARACTER
2 3 4 5 6	The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER, rather than the entire character string stored in the variable. Fortran variables of type CHARACTER or substrings are transferred as if they were arrays of characters. This is illustrated in the example below.
7 8 9	<b>Example 4.4</b> Transfer of Fortran CHARACTERs.
10 11 12	CHARACTER*10 a CHARACTER*10 b
13 14 15 16 17	CALL MPI_COMM_RANK(comm, rank, ierr) IF (rank.EQ.0) THEN CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr) ELSE IF (rank.EQ.1) THEN CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
18 19 20 21 22	END IF The last five characters of string <b>b</b> at process 1 are replaced by the first five characters of string <b>a</b> at process 0.
23 24	<i>Rationale.</i> The alternative choice would be for MPI_CHARACTER to match a character of arbitrary length. This runs into problems.
25 26 27 28 29 30 31 32	A Fortran character variable is a constant length string, with no special termina- tion symbol. There is no fixed convention on how to represent characters, and how to store their length. Some compilers pass a character argument to a routine as a pair of arguments, one holding the address of the string and the other holding the length of string. Consider the case of an MPI communication call that is passed a communication buffer with type defined by a derived datatype (Section 4.1). If this communicator buffer contains variables of type CHARACTER then the information on their length will not be passed to the MPI routine.
33 34 35 36 37	This problem forces us to provide explicit information on character length with the MPI call. One could add a length parameter to the type MPI_CHARACTER, but this does not add much convenience and the same functionality can be achieved by defining a suitable derived datatype. ( <i>End of rationale.</i> )
38 39 40 41 42	Advice to implementors. Some compilers pass Fortran CHARACTER arguments as a structure with a length and a pointer to the actual string. In such an environment, the MPI call needs to dereference the pointer in order to reach the string. (End of advice to implementors.)
43 44	4.3.2 Data Conversion
45 46 47 48	One of the goals of MPI is to support parallel computations across heterogeneous environ- ments. Communication in a heterogeneous environment may require data conversions. We use the following terminology.

14

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

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

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical and character values, and to convert a floating point value to the nearest value that can be represented on the target system.

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

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

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

Consider the three examples, 4.1-4.3. The first program is correct, assuming that a and b are REAL arrays of size  $\geq 10$ . 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.

 $\overline{7}$ 

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

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

## 4.4 Communication Modes

<sup>11</sup> The send call described in Section 4.2.1 is **blocking**: it does not return until the message <sup>12</sup> data and envelope have been safely stored away so that the sender is free to modify the <sup>13</sup> send buffer. The message might be copied directly into the matching receive buffer, or it <sup>14</sup> might be copied into a temporary system buffer.

<sup>15</sup> Message buffering decouples the send and receive operations. A blocking send can com-<sup>16</sup> plete as soon as the message was buffered, even if no matching receive has been executed by <sup>17</sup> the receiver. On the other hand, message buffering can be expensive, as it entails additional <sup>18</sup> memory-to-memory copying, and it requires the allocation of memory for buffering. MPI <sup>19</sup> offers the choice of several communication modes that allow one to control the choice of the <sup>20</sup> 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.

32 33

34

35

36

37

38 39 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 userprovided buffer system of Section 4.6 should be used, along with the buffered-mode send. (End of rationale.)

41 42 43

40

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

1

2 3

4

 $\mathbf{5}$ 

6

7 8 9

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 4receive 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 10 communication semantics: a communication does not complete at either end before both 11 processes rendezvous at the communication. A send executed in this mode is non-local. 12

A send that uses the **ready** communication mode may be started *only* if the matching 13 receive is already posted. Otherwise, the operation is erroneous and its outcome is unde-14 fined. On some systems, this allows the removal of a hand-shake operation that is otherwise 15required and results in improved performance. The completion of the send operation does 16 not depend on the status of a matching receive, and merely indicates that the send buffer 17can be reused. A send operation that uses the ready mode has the same semantics as a 18 standard send operation, or a synchronous send operation; it is merely that the sender 19 provides additional information to the system (namely that a matching receive is already 20posted), that can save some overhead. In a correct program, therefore, a ready send could 21be replaced by a standard send with no effect on the behavior of the program other than 22performance. 23

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)	29 30
IN	count	number of elements in send buffer (non-negative inte-	31
		ger)	32
IN	datatype	datatype of each send buffer element (handle)	33 34
IN	dest	rank of destination (integer)	35
IN	tag	message tag (integer)	36
IN	comm	communicator (handle)	37 38
			39
<pre>int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>			40
	int tag, MPI_Comm con	nm)	41
int MPI B	send(const void *buf. MP]	_Count count, MPI_Datatype datatype,	42
	int dest, int tag, M		43
	C		44
int MPI_B		<pre>IPI_Count count, MPI_Datatype datatype,</pre>	45
	int dest, int tag, M	PI_Comm comm)	46
MPI Bsend	(buf, count, datatype, de	est. tag. comm. ierror)	47
	(,, <b>u</b> uuujpo, u	,,,	48

MDI DCENID(L... (many detations deat to a comme 1

2

3

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

24

25

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

-	IYPE(MPI_Datatype), INTENT(IN)         INTEGER, INTENT(IN) ::         IYPE(MPI_Comm), INTENT(IN) ::         INTEGER, OPTIONAL, INTENT(OUT)	tag comm	1 2 3 4
MPI_S	SSEND(BUF, COUNT, DATATYPE, DI <type> BUF(*)</type>		5 6
	INTEGER COUNT, DATATYPE, DEST	, TAG, COMM, IERROR	7 8
Ş	Send in synchronous mode.		9
			10 11
MPI_	RSEND(buf, count, datatype, dest, t	tag, comm)	12
IN	buf	initial address of send buffer (choice)	13
IN	count	number of elements in send buffer (non-negative integer)	14 15
IN	datatype	datatype of each send buffer element (handle)	16 17
IN	dest	rank of destination (integer)	18
IN	tag	message tag (integer)	19
IN	comm	communicator (handle)	20 21
int 1	MPI_Rsend(const void *buf, int int tag, MPI_Comm co	t count, MPI_Datatype datatype, int dest, mm)	22 23 24
int 1	MPI_Rsend(const void *buf, MPI int dest, int tag, M	I_Count count, MPI_Datatype datatype, PI_Comm comm)	25 26
int 1	MPI_Rsend_x(const void *buf, M int dest, int tag, M	MPI_Count count, MPI_Datatype datatype, PI_Comm comm)	27 28 29
- - - - - - -	Rsend(buf, count, datatype, de TYPE(*), DIMENSION(), INTENT INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) TYPE(MPI_Comm), INTENT(IN) :: INTEGER, OPTIONAL, INTENT(OUT)	<pre>F(IN) :: buf , dest, tag ) :: datatype     comm</pre>	30 31 32 33 34 35 36
-	Rsend(buf, count, datatype, de TYPE(*), DIMENSION(), INTENT INTEGER(KIND=MPI_COUNT_KIND), TYPE(MPI_Datatype), INTENT(IN) INTEGER, INTENT(IN) :: dest, TYPE(MPI_Comm), INTENT(IN) :: INTEGER, OPTIONAL, INTENT(OUT)	<pre>F(IN) :: buf INTENT(IN) :: count ) :: datatype tag comm</pre>	37 38 39 40 41 42 43
	RSEND(BUF, COUNT, DATATYPE, DI <type> BUF(*) INTEGER COUNT, DATATYPE, DEST</type>		44 45 46 47
	Send in ready mode.		48

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

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

10 11

12

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.

- <sup>16</sup> A possible communication protocol for the various communication modes is outlined <sup>17</sup> below.
- <sup>19</sup> *ready send*: The message is sent as soon as possible.
- synchronous send: The sender sends a request-to-send message. The receiver stores
   this request. When a matching receive is posted, the receiver sends back a permission to-send message, and the sender now sends the message.
- standard send: First protocol may be used for short messages, and second protocol
   for long messages.
- *buffered send*: The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).
- Additional control messages might be needed for flow control and error recovery. Of
   course, there are many other possible protocols.
- Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.
- A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, users may expect some buffering.

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

37 38 39

40 41

42

43 44

35

36

## 4.5 Semantics of Point-to-Point Communication

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

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

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

**Example 4.5** An example of non-overtaking messages.

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

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

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

**Example 4.6** An example of two, intertwined matching pairs.

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

Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At

#### **Unofficial Draft for Comment Only**

 $\overline{7}$ 

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.

5

6 Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose 7 that a send is posted. Then it is possible that the destination process repeatedly posts a 8 receive that matches this send, yet the message is never received, because it is each time 9 overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multithreaded process. Then it is possible that messages that match this 10 11receive are repeatedly received, yet the receive is never satisfied, because it is overtaken 12by other receives posted at this node (by other executing threads). It is the programmer's 13 responsibility to prevent starvation in such situations.

14

41

15Resource limitations Any pending communication operation consumes system resources 16that are limited. Errors may occur when lack of resources prevent the execution of an MPI 17call. A quality implementation will use a (small) fixed amount of resources for each pending 18 send in the ready or synchronous mode and for each pending receive. However, buffer space 19 may be consumed to store messages sent in standard mode, and must be consumed to store 20messages sent in buffered mode, when no matching receive is available. The amount of space 21available for buffering will be much smaller than program data memory on many systems. 22Then, it will be easy to write programs that overrun available buffer space.

<sup>23</sup> MPI allows the user to provide buffer memory for messages sent in the buffered mode. <sup>24</sup> Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI <sup>25</sup> implementation is required to do no worse than implied by this model. This allows users to <sup>26</sup> avoid buffer overflows when they use buffered sends. Buffer allocation and use is described <sup>27</sup> in Section 4.6.

28 A buffered send operation that cannot complete because of a lack of buffer space is 29 erroneous. When such a situation is detected, an error is signaled that may cause the 30 program to terminate abnormally. On the other hand, a standard send operation that 31 cannot complete because of lack of buffer space will merely block, waiting for buffer space 32 to become available or for a matching receive to be posted. This behavior is preferable in 33 many situations. Consider a situation where a producer repeatedly produces new values 34 and sends them to a consumer. Assume that the producer produces new values faster 35 than the consumer can consume them. If buffered sends are used, then a buffer overflow 36 will result. Additional synchronization has to be added to the program so as to prevent 37 this from occurring. If standard sends are used, then the producer will be automatically 38 throttled, as its send operations will block when buffer space is unavailable.

<sup>39</sup> In some situations, a lack of buffer space leads to deadlock situations. This is illustrated
 <sup>40</sup> by the examples below.

Example 4.7 An exchange of messages.
 CALL MPI\_COMM\_RANK(comm, rank, ierr)
 IF (rank.EQ.0) THEN
 CALL MPI\_SEND(sendbuf, count, MPI\_REAL, 1, tag, comm, ierr)
 CALL MPI\_RECV(recvbuf, count, MPI\_REAL, 1, tag, comm, status, ierr)
 ELSE IF (rank.EQ.1) THEN

```
CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

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

**Example 4.8** An errant attempt to exchange messages.

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

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

**Example 4.9** An exchange that relies on buffering.

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

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

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

A program is "safe" if no message buffering is required for the program to complete. One can replace all sends in such program with synchronous sends, and the program will still run correctly. This conservative programming style provides the best portability, since program completion does not depend on the amount of buffer space available or on the communication protocol used. 2

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

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

#### 4.6 Buffer Allocation and Usage

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

```
MPI_BUFFER_ATTACH(buffer, size)
```

```
IN
                buffer
25
                                           initial buffer address (choice)
26
       IN
                size
                                           buffer size, in bytes (non-negative integer)
27
28
     int MPI_Buffer_attach(void *buffer, int size)
29
     int MPI_Buffer_attach(void *buffer, MPI_Count size)
30
31
     int MPI_Buffer_attach_x(void *buffer, MPI_Count size)
32
33
     MPI_Buffer_attach(buffer, size, ierror)
34
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
35
         INTEGER, INTENT(IN) :: size
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_Buffer_attach(buffer, size, ierror)
38
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
39
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) ::
                                                           size
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
43
          <type> BUFFER(*)
44
         INTEGER SIZE, IERROR
45
         Provides to MPI a buffer in the user's memory to be used for buffering outgoing mes-
```

46 sages. The buffer is used only by messages sent in buffered mode. Only one buffer can be 47attached to a process at a time. In C, buffer is the starting address of a memory region. In 48

1

3

4

5

6

7

8

9

10

11

12

13

14

15

16 1718

19

20

2122 23

MPI\_Buffer\_attach(buff, size);

/\* Buffer of 10000 bytes available again \*/

Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 17.1.12).

MPI_BUFF	- ER_DETACH(buffer_addr, size	2)	4
OUT	buffer_addr	initial buffer address	5 6
	_		7
OUT	size	buffer size, in bytes (non-negative integer)	8
			9
int MPI_E	Buffer_detach(void *buffe:	r_addr, int *size)	10
int MPI_E	Buffer_detach(void *buffe:	r_addr, MPI_Count *size)	11
	)	for other MDT Count waited	12
int MPI_E	Suffer_detach_x(void *bui)	fer_addr, MPI_Count *size)	13
MPI_Buffe	er_detach(buffer_addr, siz	ze, ierror)	14
-	INTRINSIC :: ISO_C_BIND	-	15
	(C_PTR), INTENT(OUT) :: 1	puffer_addr	16
	SER, INTENT(OUT) :: size		17
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	18
MPI_Buffe	er_detach(buffer_addr, siz	ze, ierror)	19 20
USE,	INTRINSIC :: ISO_C_BIND	ING, ONLY : C_PTR	20
TYPE (	(C_PTR), INTENT(OUT) :: 1	puffer_addr	22
INTEG	<pre>SER(KIND=MPI_COUNT_KIND),</pre>	INTENT(OUT) :: size	23
INTEC	ER, OPTIONAL, INTENT(OUT)	) :: ierror	24
MPT BUFFF	ER_DETACH(BUFFER_ADDR, SI	ZE. TERROR)	25
	ER(KIND=MPI_ADDRESS_KIND)		26
	ER SIZE, IERROR	_	27
Datas	h the buffer currently age ist	ad with MDI. The call nations the address and the	28
	e e	tion will block until all messages currently in the	29
	-	rn of this function, the user may reuse or deallocate	30
	taken by the buffer.	In or this function, the user may reuse or deallocate	31
the space	taken by the buller.		32
Example	4.10 Calls to attach and de	tach buffers.	33
			34 35
	BUFFSIZE 10000		36
int size;			37
char *buf	er_attach(malloc(BUFFSIZE)	BIFFSTZF)	38
	fer of 10000 bytes can not		39
	er_detach(&buff, &size);		40
	size reduced to zero */		41
	· · · ·		

Advice to users.Even though the C functions MPI\_Buffer\_attach and45MPI\_Buffer\_detach both have a first argument of type void\*, these arguments are used46differently: A pointer to the buffer is passed to MPI\_Buffer\_attach; the address of the47pointer is passed to MPI\_Buffer\_detach, so that this call can return the pointer value.48

#### Unofficial Draft for Comment Only

1

2 3

42

43

In Fortran with the mpi module or mpif.h, the type of the buffer\_addr argument is wrongly defined and the argument is therefore unused. In Fortran with the mpi\_f08 module, the address of the buffer is returned as TYPE(C\_PTR), see also Example 8.1 about the use of C\_PTR pointers. (*End of advice to users.*)

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

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

<sup>15</sup> MPI must provide as much buffering for outgoing messages *as if* outgoing message <sup>16</sup> data were buffered by the sending process, in the specified buffer space, using a circular, <sup>17</sup> contiguous-space allocation policy. We outline below a model implementation that defines <sup>18</sup> this policy. MPI may provide more buffering, and may use a better buffer allocation algo-<sup>19</sup> rithm than described below. On the other hand, MPI may signal an error whenever the <sup>20</sup> simple buffering allocator described below would run out of space. In particular, if no buffer <sup>21</sup> 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.
- 47 48

1

2 3

 $\frac{4}{5}$ 

6

7

8

9

10 11

12

13

14

25

26

27

28

29

30

31

32

33 34

35

39

40

41

42

43

44

45

- 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 com-putation. This is especially true on systems where communication can be executed au-tonomously by an intelligent communication controller. Light-weight threads are one mech-anism for achieving such overlap. An alternative mechanism that often leads to better performance is to use **nonblocking communication**. A nonblocking **send start** call ini-tiates 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.  $\mathbf{2}$ 

If the send mode is **synchronous**, then the send can complete only if a matching receive 3 has started. That is, a receive has been posted, and has been matched with the send. In 4 this case, the send-complete call is non-local. Note that a synchronous, nonblocking send 5may complete, if matched by a nonblocking receive, before the receive complete call occurs. 6 (It can complete as soon as the sender "knows" the transfer will complete, but before the 7 receiver "knows" the transfer will complete.) 8

If the send mode is **buffered** then the message must be buffered if there is no pending 9 receive. In this case, the send-complete call is local, and must succeed irrespective of the 10 status of a matching receive. 11

If the send mode is **standard** then the send-complete call may return before a matching 12receive is posted, if the message is buffered. On the other hand, the receive-complete may 13 not complete until a matching receive is posted, and the message was copied into the receive 14 buffer. 15

- 16
- 17

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 18 mode, and must be delayed, for synchronous mode, until a matching receive is posted. 19 The use of nonblocking sends in these two cases allows the sender to proceed ahead 20of the receiver, so that the computation is more tolerant of fluctuations in the speeds 21of the two processes. 22

23 Nonblocking sends in the buffered and ready modes have a more limited impact, e.g., 24 the blocking version of buffered send is capable of completing regardless of when a 25matching receive call is made. However, separating the start from the completion 26 of these sends still gives some opportunity for optimization within the MPI library. 27 For example, starting a buffered send gives an implementation more flexibility in 28 determining if and how the message is buffered. There are also advantages for both 29 nonblocking buffered and ready modes when data copying can be done concurrently 30 with computation.

31 The message-passing model implies that communication is initiated by the sender. 32 The communication will generally have lower overhead if a receive is already posted 33 when the sender initiates the communication (data can be moved directly to the 34 receive buffer, and there is no need to queue a pending send request). However, a 35receive operation can complete only after the matching send has occurred. The use 36 of nonblocking receives allows one to achieve lower communication overheads without 37 blocking the receiver while it waits for the send. (End of advice to users.)

38 39

40

#### **Communication Request Objects** 4.7.1

41Nonblocking communications use opaque **request** objects to identify communication oper-42ations and match the operation that initiates the communication with the operation that 43terminates it. These are system objects that are accessed via a handle. A request object 44 identifies various properties of a communication operation, such as the send mode, the com-45munication buffer that is associated with it, its context, the tag and destination arguments 46 to be used for a send, or the tag and source arguments to be used for a receive. In addition, 47this object stores information about the status of the pending communication operation. 48

28

#### 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)

			8
IN	buf	initial address of send buffer (choice)	9
IN	count	number of elements in send buffer (non-negative inte-	10
		ger)	11
IN	datatype	datatype of each send buffer element (handle)	12
IN	dest	rank of destination (integer)	13
		rum or doormation (mooger)	14
IN	tag	message tag (integer)	15
IN	comm	communicator (handle)	16
OUT	request	communication request (handle)	17
001	request	communication request (nandie)	18
	_		19
int MPI_I:	-	count, MPI_Datatype datatype, int dest,	20
	int tag, MPI_Comm cor	nm, MPI_Request *request)	21
int MPI Is	send(const void *buf, MPI	_Count count, MPI_Datatype datatype,	22
_		PI_Comm comm, MPI_Request *request)	23
	,,,,,		24
int MPI_Is	send_x(const void *buf, M	<pre>IPI_Count count, MPI_Datatype datatype,</pre>	25
	int dest, int tag, MI	PI_Comm comm, MPI_Request *request)	26
MPI Isend	(buf. count. datatype. de	est, tag, comm, request, ierror)	27
	• •	(IN). ASYNCHRONOUS :: buf	28

- TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI\_Datatype), INTENT(IN) :: datatype TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror
- MPI\_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
   TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
   INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count
   TYPE(MPI\_Datatype), INTENT(IN) :: datatype
   INTEGER, INTENT(IN) :: dest, tag
   TYPE(MPI\_Comm), INTENT(IN) :: comm
   TYPE(MPI\_Comm), INTENT(IN) :: request
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI\_IBSEND(buf, count, datatype, dest, tag, comm, request) 1  $\mathbf{2}$ IN buf initial address of send buffer (choice) 3 IN number of elements in send buffer (non-negative intecount 4 ger) 5IN datatype datatype of each send buffer element (handle) 6 7 rank of destination (integer) IN dest 8 IN message tag (integer) tag 9 communicator (handle) IN comm 10 11OUT communication request (handle) request 1213 int MPI\_Ibsend(const void \*buf, int count, MPI\_Datatype datatype, int dest, 14 int tag, MPI\_Comm comm, MPI\_Request \*request) 1516int MPI\_Ibsend(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 17int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 18 int MPI\_Ibsend\_x(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 19 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 2021MPI\_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) 22TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag 23 24 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 25TYPE(MPI\_Comm), INTENT(IN) :: comm 26 TYPE(MPI\_Request), INTENT(OUT) :: request 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28MPI\_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) 29 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 30 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 31 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 32 INTEGER, INTENT(IN) :: dest, tag 33 TYPE(MPI\_Comm), INTENT(IN) :: comm 34TYPE(MPI\_Request), INTENT(OUT) :: request 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI\_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 38 <type> BUF(\*) 39 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 40 Start a buffered mode, nonblocking send. 41 42 4344 4546 4748

MPI_ISS	END(buf, count, datatype,	dest, tag, comm, request)	1					
IN	buf	initial address of send buffer (choice)	2					
IN	count	number of elements in send buffer (non-negative inte-	3 4					
		ger)	4 5					
IN	datatype	datatype of each send buffer element (handle) (handle)	6					
IN	dest	rank of destination (integer)	7					
IN	tag	message tag (integer)	8					
IN	comm	communicator (handle)	9 10					
OUT			11					
001	request	communication request (handle)	12					
int MPT	Issend(const void *bu	f, int count, MPI_Datatype datatype, int dest,	13					
1110 111 1		m comm, MPI_Request *request)	14					
·+ MDT	-		15 16					
int MPI		f, MPI_Count count, MPI_Datatype datatype, ag, MPI_Comm comm, MPI_Request *request)	10					
			18					
int MPI		buf, MPI_Count count, MPI_Datatype datatype,	19					
	int dest, int ta	ag, MPI_Comm comm, MPI_Request *request)	20					
MPI_Iss	MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)							
		NTENT(IN), ASYNCHRONOUS :: buf	22					
	EGER, INTENT(IN) :: c	-	23					
	E(MPI_Datatype), INTEN		24 25					
	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request							
	EGER, OPTIONAL, INTENT	-	27					
	•	pe, dest, tag, comm, request, ierror)	29					
		NTENT(IN), ASYNCHRONOUS :: buf ND), INTENT(IN) :: count	30					
	E(MPI_Datatype), INTEN		31					
	EGER, INTENT(IN) :: d	• =	32 33					
TYP	TYPE(MPI_Comm), INTENT(IN) :: comm							
TYP	E(MPI_Request), INTENT	(OUT) :: request	34 35					
INT	EGER, OPTIONAL, INTENT	(OUT) :: ierror	36					
MPI_ISS	END(BUF, COUNT, DATATY	PE, DEST, TAG, COMM, REQUEST, IERROR)	37					
	pe> BUF(*)		38					
INT	EGER COUNT, DATATYPE,	DEST, TAG, COMM, REQUEST, IERROR	39 40					
Start a synchronous mode, nonblocking send.								
			41 42					
			42					

MPI\_IRSEND(buf, count, datatype, dest, tag, comm, request) 1  $\mathbf{2}$ IN buf initial address of send buffer (choice) 3 IN number of elements in send buffer (non-negative intecount 4 ger) 5IN datatype datatype of each send buffer element (handle) 6 7 rank of destination (integer) IN dest 8 IN message tag (integer) tag 9 communicator (handle) IN comm 10 11OUT communication request (handle) request 1213 int MPI\_Irsend(const void \*buf, int count, MPI\_Datatype datatype, int dest, 14 int tag, MPI\_Comm comm, MPI\_Request \*request) 1516int MPI\_Irsend(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 17int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 18 int MPI\_Irsend\_x(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 19 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 2021MPI\_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) 22TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag 23 24 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 25TYPE(MPI\_Comm), INTENT(IN) :: comm 26 TYPE(MPI\_Request), INTENT(OUT) :: request 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28MPI\_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) 29 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 30 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 31 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 32 INTEGER, INTENT(IN) :: dest, tag 33 TYPE(MPI\_Comm), INTENT(IN) :: comm 34TYPE(MPI\_Request), INTENT(OUT) :: request 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI\_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 38 <type> BUF(\*) 39 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 40 Start a ready mode nonblocking send. 41 42 4344 45 46 4748

MPI_IRECV(buf, count, datatype, source, tag, comm, request)							
OUT	buf	initial address of receive buffer (choice)	2				
IN	count	number of elements in receive buffer (non-negative in-teger)	3 4 5				
IN	datatype	datatype of each receive buffer element (handle)	6				
IN	source	rank of source or MPI_ANY_SOURCE (integer)	7				
IN	tag	message tag or MPI_ANY_TAG (integer)	8 9				
IN	comm	communicator (handle)	9 10				
OUT	request	communication request (handle)	11				
001	request	communication request (nanule)	12				
int MDI Incov(void thuf int count MDI Deteture deteture int course							
1110 III 1	<pre>int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,</pre>						
	-		15				
int MPI		t count, MPI_Datatype datatype,	16				
	int source, int tag,	MPI_Comm comm, MPI_Request *request)	17 18				
<pre>int MPI_Irecv_x(void *buf, MPI_Count count, MPI_Datatype datatype,</pre>							
<pre>int source, int tag, MPI_Comm comm, MPI_Request *request)</pre>							
MPI_Irec	/(buf, count, datatype, s	ource, tag, comm, request, ierror)	21				
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag							
							TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm							
	TYPE(MPI_Request), INTENT(OUT) :: request						
INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
	• =	ource, tag, comm, request, ierror)	28 29				
	(*), DIMENSION(), ASYNCI		30				
	GER(KIND=MPI_COUNT_KIND),		31				
	TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: source, tag TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request						
	GER, OPTIONAL, INTENT(OUT)	-	35				
			36				
		DURCE, TAG, COMM, REQUEST, IERROR)	37 38				
• -	<type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR</type>						
	JER GOUNT, DATATIFE, SUUR	CE, ING, CUTIT, REQUEST, LERRUR	39 40				
	a nonblocking receive.		41				
These calls allocate a communication request object and associate it with the request							
handle (the argument request). The request can be used later to query the status of the							

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

44

45

46

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

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

#### 4.7.3 Communication Completion

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

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

We shall use the following terminology: A **null handle** is a handle with value MPI\_REQUEST\_NULL. A persistent request and the handle to it are **inactive** if the request is not associated with any ongoing communication (see Section 4.9). A handle is **active** if it is neither null nor inactive. An **empty** status is a status which is set to return **tag** = MPI\_ANY\_TAG, source = MPI\_ANY\_SOURCE, error = MPI\_SUCCESS, and is also internally configured so that calls to MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and

<sup>21</sup> MPI\_GET\_ELEMENTS\_X return count = 0 and MPI\_TEST\_CANCELLED returns false. We <sup>29</sup> set a status variable to empty when the value returned by it is not significant. Status is set <sup>30</sup> 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 that take a single MPI\_Status argument, the error code is returned by the function, and the value of the MPI\_ERROR field in the MPI\_Status argument is undefined (see 4.2.5).

MPI\_WAIT(request, status)

10		,			
42 43	INOUT	request		request (handl	e)
44	OUT	status		status object (	(Status)
45					
46	int MPI_Wa	ait(MPI_Request	<pre>*request,</pre>	MPI_Status	*status)
47			· )		
48	mpi_wait()	request, status	, rerror)		

1

2 3

4

 $\mathbf{5}$ 

6

7 8

9

36

37

38

TYPE	MPI_Request), INTENT(INO	JT) :: request	1				
	MPI_Status) :: status		2				
INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
MPI_WAIT(REQUEST, STATUS, IERROR)							
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR							
			6 7				
A call to MPI_WAIT returns when the operation identified by request is complete. If							
-	t is marked inactive. Any other type of request is	8 9					
deallocated and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-loc operation.							
-	all returns in status informa	tion on the completed operation. The content of	10 11				
		can be accessed as described in Section 4.2.5. The	12				
status object for a send operation may be queried by a call to MPI_TEST_CANCELLED							
(see Sectio			14				
One is allowed to call MPI_WAIT with a null or inactive request argument. In this case							
the operat	ion returns immediately with	empty status.	16				
			17				
		n of MPI_WAIT after a MPI_IBSEND implies that	18				
		ed — i.e., data has been sent out or copied into ER_ATTACH. Note that, at this point, we can no	19				
		4.8). If a matching receive is never posted, then the	20 21				
0		omewhat counter to the stated goal of MPI_CANCEL	21				
		a space that was committed to the communication	23				
	ystem). (End of advice to use	-	24				
·		,	25				
	-	tithreaded environment, a call to MPI_WAIT should	26				
	, e	ing the thread scheduler to schedule another thread	27				
for e	xecution. (End of advice to in	nplementors.)	28				
			29				
			30				
MPI_TEST	(request, flag, status)		31				
INOUT	request	communication request (handle)	32				
OUT	flag		33 34				
	-	true if operation completed (logical)	35				
OUT	status	status object (Status)	36				
			37				
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status) 3							
MDI Toat (request flog stotus jerrer)							

MPI_Test(request, flag, status, ierror)	39
TYPE(MPI_Request), INTENT(INOUT) :: request	40
LOGICAL, INTENT(OUT) :: flag	41
TYPE(MPI_Status) :: status	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
	44
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)	45
INTEGER REQUEST	46
LOGICAL FLAG	47
INTEGER STATUS(MPI_STATUS_SIZE), IERROR	48

```
A call to MPI_TEST returns flag = true if the operation identified by request is complete.
1
     In such a case, the status object is set to contain information on the completed operation.
^{2}
     If the request is an active persistent request, it is marked as inactive. Any other type of
3
     request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns
4
     flag = false if the operation identified by request is not complete. In this case, the value of
5
     the status object is undefined. MPI_TEST is a local operation.
6
          The return status object for a receive operation carries information that can be accessed
7
     as described in Section 4.2.5. The status object for a send operation carries information
8
     that can be accessed by a call to MPI_TEST_CANCELLED (see Section 4.8).
9
          One is allowed to call MPI_TEST with a null or inactive request argument. In such a
10
     case the operation returns with flag = true and empty status.
11
          The functions MPI_WAIT and MPI_TEST can be used to complete both sends and
12
     receives.
13
14
           Advice to users.
                              The use of the nonblocking MPI_TEST call allows the user to
15
           schedule alternative activities within a single thread of execution. An event-driven
16
           thread scheduler can be emulated with periodic calls to MPI_TEST. (End of advice to
17
           users.)
18
19
20
                       Simple usage of nonblocking operations and MPI_WAIT.
     Example 4.11
21
22
     CALL MPI_COMM_RANK(comm, rank, ierr)
23
     IF (rank.EQ.0) THEN
24
          CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
25
          **** do some computation to mask latency ****
26
          CALL MPI_WAIT(request, status, ierr)
27
     ELSE IF (rank.EQ.1) THEN
28
          CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
29
          **** do some computation to mask latency ****
30
          CALL MPI_WAIT(request, status, ierr)
31
     END IF
32
          A request object can be deallocated by using the following operation.
33
34
35
     MPI_REQUEST_FREE(request)
36
       INOUT
                 request
                                              communication request (handle)
37
38
39
     int MPI_Request_free(MPI_Request *request)
40
     MPI_Request_free(request, ierror)
41
          TYPE(MPI_Request), INTENT(INOUT) :: request
42
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_REQUEST_FREE(REQUEST, IERROR)
45
          INTEGER REQUEST, IERROR
46
          MPI_REQUEST_FREE is a local operation that marks the request object for deallo-
47
     cation and sets request to MPI_REQUEST_NULL. Ongoing communication, if any, that is
48
```

associated with the request will be allowed to complete. The request will be deallocated only after its completion. Classes of operations described later in the standard, such as nonblocking collective and persistent collective (see Chapters 5 and 7), also use request objects. In the case of nonblocking collective operations and persistent collective operations, it is erroneous to call MPI\_REQUEST\_FREE unless the request is inactive.

*Rationale.* For point-to-point operations, the MPI\_REQUEST\_FREE mechanism is provided for reasons of performance and convenience on the sending side. (*End of rationale.*)

Advice to users. Once a request is freed by a call to MPI\_REQUEST\_FREE, it is not possible to check for the successful completion of the associated communication with calls to MPI\_WAIT or MPI\_TEST. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user — such an error must be treated as fatal. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (*End of advice to users.*)

```
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.

OrderNonblocking communication operations are ordered according to the execution order46of the calls that initiate the communication. The non-overtaking requirement of Section 4.547is extended to nonblocking communication, with this definition of order being used.48

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9 10

11

12

13

14

15

16

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40 41

42

43

Message ordering for nonblocking operations. Example 4.13 1  $\mathbf{2}$ CALL MPI\_COMM\_RANK(comm, rank, ierr) 3 IF (RANK.EQ.0) THEN 4 CALL MPI\_ISEND(a, 1, MPI\_REAL, 1, 0, comm, r1, ierr)  $\mathbf{5}$ CALL MPI\_ISEND(b, 1, MPI\_REAL, 1, 0, comm, r2, ierr) 6 ELSE IF (rank.EQ.1) THEN 7 CALL MPI\_IRECV(a, 1, MPI\_REAL, 0, MPI\_ANY\_TAG, comm, r1, ierr) 8 CALL MPI\_IRECV(b, 1, MPI\_REAL, 0, 0, comm, r2, ierr) 9 END IF 10 CALL MPI\_WAIT(r1, status, ierr) 11CALL MPI\_WAIT(r2, status, ierr) 1213 The first send of process zero will match the first receive of process one, even if both messages 14are sent before process one executes either receive. 1516**Progress** A call to MPI\_WAIT that completes a receive will eventually terminate and return 17if a matching send has been started, unless the send is satisfied by another receive. In 18 particular, if the matching send is nonblocking, then the receive should complete even if no 19 call is executed by the sender to complete the send. Similarly, a call to MPI\_WAIT that 20completes a send will eventually return if a matching receive has been started, unless the 21receive is satisfied by another send, and even if no call is executed to complete the receive. 2223Example 4.14 An illustration of progress semantics. 24CALL MPI\_COMM\_RANK(comm, rank, ierr) 25IF (RANK.EQ.O) THEN 26 CALL MPI\_SSEND(a, 1, MPI\_REAL, 1, 0, comm, ierr) 27CALL MPI\_SEND(b, 1, MPI\_REAL, 1, 1, comm, ierr) 28ELSE IF (rank.EQ.1) THEN 29 CALL MPI\_IRECV(a, 1, MPI\_REAL, 0, 0, comm, r, ierr) 30 CALL MPI\_RECV(b, 1, MPI\_REAL, 0, 1, comm, status, ierr) 31 CALL MPI\_WAIT(r, status, ierr) 32END IF 33 34 This code should not deadlock in a correct MPI implementation. The first synchronous 35send of process zero must complete after process one posts the matching (nonblocking) 36 receive even if process one has not yet reached the completing wait call. Thus, process zero 37

<sup>38</sup> If an MPI\_TEST that completes a receive is repeatedly called with the same arguments, <sup>39</sup> and a matching send has been started, then the call will eventually return flag = true, unless <sup>40</sup> the send is satisfied by another receive. If an MPI\_TEST that completes a send is repeatedly <sup>41</sup> called with the same arguments, and a matching receive has been started, then the call will <sup>42</sup> eventually return flag = true, unless the receive is satisfied by another send.

will continue and execute the second send, allowing process one to complete execution.

44 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.

IN co	rray_of_requests ndex	list length (non-negative integer) array of requests (array of handles) index of handle for operation that completed (integer)	6 7 8
	rray_of_requests ndex	array of requests (array of handles)	8
INOUT ar	ndex	· - ( · / /	
		index of handle for operation that completed (integer)	
OUT in			9 10
OUT st	tatus	status object (Status)	11
			12
int MPI_Wait	tany(int count, MPI_Requ MPI_Status *status)	<pre>nest array_of_requests[], int *index,</pre>	$13 \\ 14$
			15
int MPI_Wait	v	PI_Request array_of_requests[],	16
	int *index, MPI_Statu	s *status)	17
int MPI_Wait	<pre>tany_x(MPI_Count count,</pre>	<pre>MPI_Request array_of_requests[],</pre>	18
	int *index, MPI_Statu	s *status)	19
MPT Waitany	(count array of request	ts, index, status, ierror)	20
•	, INTENT(IN) :: count	S, Index, Status, Ierrory	21 22
		<pre>C) :: array_of_requests(count)</pre>	22
	, INTENT(OUT) :: index		20
TYPE(MP)	I_Status) :: status		25
INTEGER	, OPTIONAL, INTENT(OUT)	:: ierror	26
MPT Waitany	(count array of request	ts, index, status, ierror)	27
•	(KIND=MPI_COUNT_KIND), 1		28
		T) :: array_of_requests(count)	29
	, INTENT(OUT) :: index		30
TYPE(MP)	I_Status) :: status		31
INTEGER	, OPTIONAL, INTENT(OUT)	:: ierror	32
ΜΡΤ ΜΔΤΤΔΝΥ	COUNT ARRAY OF REGUEST	TS, INDEX, STATUS, IERROR)	33
		TS, INDEX, STATUS, TERROR, TERROR, TS, INDEX, STATUS, TERROR, TS, TATUS,	34 35
IERROR		,,,,	35 36
		• • • • • • • • • • • •	37

Blocks until one of the operations associated with the active requests in the array has completed. If more than one operation is enabled and can terminate, one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing operation. (The array is indexed from zero in C, and from one in Fortran.) If the request is an active persistent request, it is marked inactive. Any other type of request is deallocated and the request handle is set to MPI\_REQUEST\_NULL.

The array\_of\_requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns immediately with index = MPI\_UNDEFINED, and an empty status.

The execution of MPI\_WAITANY(count, array\_of\_requests, index, status) has the same effect as the execution of MPI\_WAIT(&array\_of\_requests[i], status), where i is the value

1

 $\mathbf{2}$ 

3 4

38

39

40

 $^{41}$ 

42

43

44

45

46

47

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.

```
3
4
     MPI_TESTANY(count, array_of_requests, index, flag, status)
5
       IN
6
                count
                                            list length) (non-negative integer)
7
       INOUT
                array_of_requests
                                            array of requests (array of handles)
8
       OUT
                index
                                            index of operation that completed or
9
                                            MPI_UNDEFINED if none completed (integer)
10
11
       OUT
                flag
                                            true if one of the operations is complete (logical)
12
       OUT
                status
                                            status object (Status)
13
14
     int MPI_Testany(int count, MPI_Request array_of_requests[], int *index,
15
                    int *flag, MPI_Status *status)
16
     int MPI_Testany(MPI_Count count, MPI_Request array_of_requests[],
17
18
                    int *index, int *flag, MPI_Status *status)
19
     int MPI_Testany_x(MPI_Count count, MPI_Request array_of_requests[],
20
                    int *index, int *flag, MPI_Status *status)
21
22
     MPI_Testany(count, array_of_requests, index, flag, status, ierror)
23
         INTEGER, INTENT(IN) :: count
24
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
25
         INTEGER, INTENT(OUT) :: index
26
         LOGICAL, INTENT(OUT) ::
                                     flag
27
         TYPE(MPI_Status) :: status
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Testany(count, array_of_requests, index, flag, status, ierror)
30
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
31
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
32
         INTEGER, INTENT(OUT) ::
                                     index
33
         LOGICAL, INTENT(OUT) ::
                                     flag
34
         TYPE(MPI_Status) :: status
35
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
36
37
     MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
38
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX
39
         LOGICAL FLAG
40
         INTEGER STATUS(MPI_STATUS_SIZE), IERROR
41
         Tests for completion of either one or none of the operations associated with active
42
     handles. In the former case, it returns flag = true, returns in index the index of this request
43
     in the array, and returns in status the status of that operation. If the request is an active
44
```

<sup>45</sup> persistent request, it is marked as inactive. Any other type of request is deallocated and
<sup>46</sup> the handle is set to MPI\_REQUEST\_NULL. (The array is indexed from zero in C, and from
<sup>47</sup> one in Fortran.) In the latter case (no operation completed), it returns flag = false, returns
<sup>48</sup> a value of MPI\_UNDEFINED in index and status is undefined.

The array may contain null or inactive handles. If the array contains no active handles then the call returns immediately with flag = true,  $index = MPI_UNDEFINED$ , and an empty status.

If the array of requests contains active handles then the execution of MPI\_TESTANY(count, array\_of\_requests, index, status) has the same effect as the execution of MPI\_TEST( &array\_of\_requests[i], flag, status), for i=0, 1, ..., count-1, in some arbitrary order, until one call returns flag = true, or all fail. In the former case, index is set to the last value of i, and in the latter case, it is set to MPI\_UNDEFINED. MPI\_TESTANY with an array containing one active entry is equivalent to MPI\_TEST.

MPI WAIT	ALL(count, array_of_request	s array of statuses)	11
	<b>`</b>	- , ,	12
IN	count	lists length (non-negative integer)	13
INOUT	array_of_requests	array of requests (array of handles)	14
OUT	array_of_statuses	array of status objects (array of Status)	15
	,	J J ( J / J	16
int MPT W	aitall(int count MPT B	equest array_of_requests[],	17
1110 111 1	MPI_Status array_of		18 19
	·		19 20
int MPI_W		<pre>MPI_Request array_of_requests[],</pre>	20 21
	MPI_Status array_of	_statuses[])	21
int MPT W	aitall x(MPT Count coun	t, MPI_Request array_of_requests[],	22
	MPI_Status array_of		23
	Ů		25
		ests, array_of_statuses, ierror)	26
	ER, INTENT(IN) :: coun		27
	1 .	OUT) :: array_of_requests(count)	28
	MPI_Status) :: array_o		29
INTEG	ER, OPTIONAL, INTENT(OU	T) :: ierror	30
MPI_Waita	ll(count, array_of_requ	ests, array_of_statuses, ierror)	31
	ER(KIND=MPI_COUNT_KIND)		32
		OUT) :: array_of_requests(count)	33
TYPE(	MPI_Status) :: array_o	f_statuses(*)	34
INTEG	ER, OPTIONAL, INTENT(OU	T) :: ierror	35
			36
		ESTS, ARRAY_OF_STATUSES, IERROR)	37
	ER COUNT, ARRAY_OF_REQU		38
АККАҮ	_OF_STATUSES(MPI_STATUS	_DILE,*/, IEKKUK	39
D1 1			

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The i-th entry in array\_of\_statuses is set to the return status of the i-th operation. Active persistent requests are marked inactive. Requests of any other type are deallocated and the corresponding handles in the array are set to MPI\_REQUEST\_NULL. The list may contain null or inactive handles. The call sets to empty the status of each such entry.

The error-free execution of MPI\_WAITALL(count, array\_of\_requests, array\_of\_statuses) has the same effect as the execution of

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9 10

40

41

42

43

44

45

46

47

MPI\_WAIT(&array\_of\_request[i], &array\_of\_statuses[i]), for i=0,..., count-1, in some arbi-1 trary order. MPI\_WAITALL with an array of length one is equivalent to MPI\_WAIT. 2 When one or more of the communications completed by a call to MPI\_WAITALL fail, 3 it is desirable to return specific information on each communication. The function 4 MPI\_WAITALL will return in such case the error code MPI\_ERR\_IN\_STATUS and will set the 5error field of each status to a specific error code. This code will be MPI\_SUCCESS, if the 6 specific communication completed; it will be another specific error code, if it failed; or it can 7 be MPI\_ERR\_PENDING if it has neither failed nor completed. The function MPI\_WAITALL 8 will return MPI\_SUCCESS if no request had an error, or will return another error code if it 9 failed for other reasons (such as invalid arguments). In such cases, it will not update the 10 error fields of the statuses. 11 12Rationale. This design streamlines error handling in the application. The application 13 code need only test the (single) function result to determine if an error has occurred. It 14 needs to check each individual status only when an error occurred. (End of rationale.) 15 16 1718 MPI\_TESTALL(count, array\_of\_requests, flag, array\_of\_statuses) 19 IN count lists length (non-negative integer) 2021INOUT array\_of\_requests array of requests (array of handles) 22 OUT flag (logical) 23 OUT array\_of\_statuses array of status objects (array of Status) 24 25int MPI\_Testall(int count, MPI\_Request array\_of\_requests[], int \*flag, 26MPI\_Status array\_of\_statuses[]) 2728 int MPI\_Testall(MPI\_Count count, MPI\_Request array\_of\_requests[], 29 int \*flag, MPI\_Status array\_of\_statuses[]) 30 31 int MPI\_Testall\_x(MPI\_Count count, MPI\_Request array\_of\_requests[], int \*flag, MPI\_Status array\_of\_statuses[]) 32 33 MPI\_Testall(count, array\_of\_requests, flag, array\_of\_statuses, ierror) 34 INTEGER, INTENT(IN) :: count 35 TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count) 36 LOGICAL, INTENT(OUT) :: flag 37 TYPE(MPI\_Status) :: array\_of\_statuses(\*) 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40 MPI\_Testall(count, array\_of\_requests, flag, array\_of\_statuses, ierror) 41 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 42 TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count) 43LOGICAL, INTENT(OUT) :: flag TYPE(MPI\_Status) :: array\_of\_statuses(\*) 44 45 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46 MPI\_TESTALL(COUNT, ARRAY\_OF\_REQUESTS, FLAG, ARRAY\_OF\_STATUSES, IERROR) 47INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*) 48

LOGICAL FLAG 1 INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR 2											
INTEGER ARRAY_OF_STATUSES(MPI_STATUS_STZE,*), TERROR											
Retur	Returns $flag = true$ if all communications associated with active handles in the array										
have comp	have completed (this includes the case where no handle in the list is active). In this case, each										
status entr	ry that corresponds to an ac	tive request is set to the status of the corresponding	5								
operation.	Active persistent requests	are marked inactive. Requests of any other type are	6								
-		andles in the array are set to MPI_REQUEST_NULL.	7								
Each statu	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	s that occurred during the o	execution of MPI_TESTALL are handled in the same	11								
	errors in MPI_WAITALL.		12								
			13								
			14								
MPI_WAI	FSOME(incount, array_of_re	quests, outcount, array_of_indices, array_of_statuses)	15								
			16								
IN	incount	length of arry_of_requests (non-negative integer)	17 18								
INOUT	array_of_requests	array of requests (array of handles)	19								
OUT	outcount	number of completed requests (non-negative integer)	20								
OUT	array_of_indices	array of indices of operations that completed (array of	21								
	5	integers)	22 23								
OUT	array_of_statuses	array of status objects for operations that completed	24								
	5	(array of Status)	25								
			26								
int MPT W	Jaitsome(int incount MI	PI_Request array_of_requests[],	27								
1110 111 1_1		<pre>array_of_indices[],</pre>	28								
	MPI_Status array_o	•	29								
	-		30								
int MPI_V		<pre>int, MPI_Request array_of_requests[],</pre>	31								
		t, int array_of_indices[],	32								
	MPI_Status array_o	f_statuses[])	33								
int MPI W	Waitsome x(MPI Count inc	<pre>count, MPI_Request array_of_requests[],</pre>	34								
_		t, int array_of_indices[],	35								
	_ MPI_Status array_o	•	36								
	C C		37								
MPI_Waits	•	requests, outcount, array_of_indices,	38								
	array_of_statuses,		39								
	GER, INTENT(IN) :: inco		40								
	-	<pre>IOUT) :: array_of_requests(count)</pre>	41								
	GER, INTENT(OUT) :: out		42								
	GER, INTENT(OUT) :: ari	•	43								
	TYPE(MPI_Status) :: array_of_statuses(*)										
TNLE	GER, OPTIONAL, INTENT(OU	)); :: lerror	45								
MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, 46											
	array_of_statuses, ierror) 47										
	-		48								

1 2 3 4 5 6 7	<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count) INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: outcount INTEGER, INTENT(OUT) :: array_of_indices(*) TYPE(MPI_Status) :: array_of_statuses(*) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,</pre>											
8 9 10 11	ARRAY_OF_STATUSES, IERROR) INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR											
12 13 14 15 16 17 18 19	completed have comp indices of from zero array array requests at	. Returns in outcount the number of these operations (index with in C and from one in Fortr _of_status the status for these re marked as inactive. Any c	ations associated with active handles in the list have mber of requests from the list array_of_requests that outcount locations of the array array_of_indices the in the array array_of_requests; the array is indexed an). Returns in the first outcount locations of the e completed operations. Completed active persistent other type or request that completed is deallocated, REQUEST_NULL									
20 21 22 23 24 25 26	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_WAITSOME 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											
27 28 29 30 31	if no reque	est resulted in an error, and	r that occurred. The call will return MPI_SUCCESS will return another error code if it failed for other such cases, it will not update the error fields of the									
32 33 34	MPI_TEST	SOME(incount, array_of_req	uests, outcount, array_of_indices, array_of_statuses)									
35	IN	incount	length of array_of_requests (non-negative integer)									
36	INOUT	array_of_requests	array of requests (array of handles)									
37 38	OUT	outcount	number of completed requests (non-negative integer)									
39 40	OUT	array_of_indices	array of indices of operations that completed (array of integers)									
41 42 43	OUT	array_of_statuses	array of status objects for operations that completed (array of Status)									
44 45 46	<pre>int MPI_Testsome(int incount, MPI_Request array_of_requests[],</pre>											
47 48	<pre>int MPI_Testsome(MPI_Count incount, MPI_Request array_of_requests[],</pre>											

MPI\_Count \*outcount, int array\_of\_indices[], 1 MPI\_Status array\_of\_statuses[]) 2 3 int MPI\_Testsome\_x(MPI\_Count incount, MPI\_Request array\_of\_requests[], MPI\_Count \*outcount, int array\_of\_indices[], 5MPI\_Status array\_of\_statuses[]) 6 MPI\_Testsome(incount, array\_of\_requests, outcount, array\_of\_indices, 7 array\_of\_statuses, ierror) 8 INTEGER, INTENT(IN) :: incount 9 TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count) 10 INTEGER, INTENT(OUT) :: outcount 11 INTEGER, INTENT(OUT) :: array\_of\_indices(\*) 12 TYPE(MPI\_Status) :: array\_of\_statuses(\*) 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 14 15MPI\_Testsome(incount, array\_of\_requests, outcount, array\_of\_indices, 16 array\_of\_statuses, ierror) 17INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: incount 18 TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count) 19 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: outcount 20INTEGER, INTENT(OUT) :: array\_of\_indices(\*) 21TYPE(MPI\_Status) :: array\_of\_statuses(\*) 22INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23 MPI\_TESTSOME(INCOUNT, ARRAY\_OF\_REQUESTS, OUTCOUNT, ARRAY\_OF\_INDICES, 24ARRAY\_OF\_STATUSES, IERROR) 25INTEGER INCOUNT, ARRAY\_OF\_REQUESTS(\*), OUTCOUNT, ARRAY\_OF\_INDICES(\*), 26ARRAY\_OF\_STATUSES(MPI\_STATUS\_SIZE,\*), IERROR 2728Behaves like MPI\_WAITSOME, except that it returns immediately. If no operation has 29 completed it returns outcount = 0. If there is no active handle in the list it returns outcount30 = MPI\_UNDEFINED. 31 MPI\_TESTSOME is a local operation, which returns immediately, whereas 32 MPI\_WAITSOME will block until a communication completes, if it was passed a list that 33 contains at least one active handle. Both calls fulfill a **fairness** requirement: If a request 34 for a receive repeatedly appears in a list of requests passed to MPI\_WAITSOME or 35MPI\_TESTSOME, and a matching send has been posted, then the receive will eventually 36 succeed, unless the send is satisfied by another receive; and similarly for send requests. 37 Errors that occur during the execution of MPI\_TESTSOME are handled as for 38 MPI\_WAITSOME. 39 40 Advice to users. The use of MPI\_TESTSOME is likely to be more efficient than the use 41 of MPI\_TESTANY. The former returns information on all completed communications, 42 with the latter, a new call is required for each communication that completes. 43A server with multiple clients can use MPI\_WAITSOME so as not to starve any client. 44 Clients send messages to the server with service requests. The server calls 45MPI\_WAITSOME with one receive request for each client, and then handles all receives 46that completed. If a call to MPI\_WAITANY is used instead, then one client could starve 47while requests from another client always sneak in first. (End of advice to users.) 48

```
MPI_TESTSOME should complete as many pending com-
          Advice to implementors.
1
          munications as possible. (End of advice to implementors.)
2
3
4
     Example 4.15
                      Client-server code (starvation can occur).
5
6
7
     CALL MPI_COMM_SIZE(comm, size, ierr)
8
     CALL MPI_COMM_RANK(comm, rank, ierr)
9
     IF(rank .GT. 0) THEN
                                    ! client code
10
         DO WHILE(.TRUE.)
11
             CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
             CALL MPI_WAIT(request, status, ierr)
12
         END DO
13
14
     ELSE
                   ! rank=0 -- server code
15
            DO i=1, size-1
16
                CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                         comm, request_list(i), ierr)
17
18
            END DO
19
            DO WHILE(.TRUE.)
20
                CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
21
                CALL DO_SERVICE(a(1,index)) ! handle one message
22
                CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag,
                           comm, request_list(index), ierr)
23
24
            END DO
25
     END IF
26
27
     Example 4.16
                      Same code, using MPI_WAITSOME.
28
29
     CALL MPI_COMM_SIZE(comm, size, ierr)
30
     CALL MPI_COMM_RANK(comm, rank, ierr)
31
     IF(rank .GT. 0) THEN
                                    ! client code
32
         DO WHILE(.TRUE.)
33
             CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
34
             CALL MPI_WAIT(request, status, ierr)
35
         END DO
36
     ELSE
                   ! rank=0 -- server code
37
         DO i=1, size-1
38
             CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
39
                             comm, request_list(i), ierr)
40
         END DO
41
42
         DO WHILE(.TRUE.)
            CALL MPI_WAITSOME(size, request_list, numdone,
43
                               indices, statuses, ierr)
44
            DO i=1, numdone
45
                CALL DO_SERVICE(a(1, indices(i)))
46
                CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag,
47
                              comm, request_list(indices(i)), ierr)
48
```

	END	DO	)															
	END DO																	
END	IF																	

## 4.7.6 Non-destructive Test of status

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

### MPI\_REQUEST\_GET\_STATUS(request, flag, status)

INTEGER STATUS(MPI\_STATUS\_SIZE), IERROR

IN	request	request (handle)						
OUT	flag	boolean flag, same as from $MPI\_TEST$ (logical)						
OUT	status	status object if flag is true (Status)						
<pre>int MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status)</pre>								
<pre>MPI_Request_get_status(request, flag, status, ierror)    TYPE(MPI_Request), INTENT(IN) :: request    LOGICAL, INTENT(OUT) :: flag    TYPE(MPI_Status) :: status    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>								
INTEG	ST_GET_STATUS(REQUEST, FL ER REQUEST AL FLAG	AG, STATUS, IERROR)						

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

One is allowed to call MPI\_REQUEST\_GET\_STATUS with a null or inactive request argument. In such a case the operation returns with flag=true and empty status.

## 4.8 Probe and Cancel

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

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

#### **Unofficial Draft for Comment Only**

```
Cancelling a send request by calling MPI_CANCEL is deprecated.
1
\mathbf{2}
3
     4.8.1 Probe
4
\mathbf{5}
6
      MPI_IPROBE(source, tag, comm, flag, status)
7
       IN
                 source
                                               rank of source or MPI_ANY_SOURCE (integer)
8
       IN
                                               message tag or MPI_ANY_TAG (integer)
9
                  tag
10
       IN
                 comm
                                               communicator (handle)
11
        OUT
                 flag
                                               (logical)
12
       OUT
                 status
                                               status object (Status)
13
14
15
      int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
16
                     MPI_Status *status)
17
     MPI_Iprobe(source, tag, comm, flag, status, ierror)
18
          INTEGER, INTENT(IN) :: source, tag
19
          TYPE(MPI_Comm), INTENT(IN) :: comm
20
          LOGICAL, INTENT(OUT) ::
                                        flag
21
          TYPE(MPI_Status) :: status
22
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
25
          INTEGER SOURCE, TAG, COMM
26
          LOGICAL FLAG
27
          INTEGER STATUS(MPI_STATUS_SIZE), IERROR
28
          MPI_IPROBE(source, tag, comm, flag, status) returns flag = true if there is a message
29
      that can be received and that matches the pattern specified by the arguments source, tag,
30
      and comm. The call matches the same message that would have been received by a call to
31
      MPI_RECV(..., source, tag, comm, status) executed at the same point in the program, and
32
      returns in status the same value that would have been returned by MPI_RECV(). Otherwise,
33
      the call returns flag = false, and leaves status undefined.
34
          If MPI_IPROBE returns flag = true, then the content of the status object can be sub-
35
      sequently accessed as described in Section 4.2.5 to find the source, tag and length of the
36
      probed message.
37
          A subsequent receive executed with the same communicator, and the source and tag re-
38
      turned in status by MPI_IPROBE will receive the message that was matched by the probe, if
39
      no other intervening receive occurs after the probe, and the send is not successfully cancelled
40
      before the receive. If the receiving process is multithreaded, it is the user's responsibility
41
      to ensure that the last condition holds.
42
          The source argument of MPI_PROBE can be MPI_ANY_SOURCE, and the tag argument
43
      can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and/or
44
      with an arbitrary tag. However, a specific communication context must be provided with
45
      the comm argument.
46
          It is not necessary to receive a message immediately after it has been probed for, and
47
```

A probe with MPI\_PROC\_NULL as source returns flag = true, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0; see Section 4.11.

MPI\_PROBE(source, tag, comm, status)

IN	source	rank of source or $MPI\_ANY\_SOURCE$ (integer)
IN	tag	message tag or $MPI\_ANY\_TAG$ (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

int MPI\_Probe(int source, int tag, MPI\_Comm comm, MPI\_Status \*status)

MPI\_Probe(source, tag, comm, status, ierror)
 INTEGER, INTENT(IN) :: source, tag
 TYPE(MPI\_Comm), INTENT(IN) :: comm
 TYPE(MPI\_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI\_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
INTEGER SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR

MPI\_PROBE behaves like MPI\_IPROBE except that it is a blocking call that returns only after a matching message has been found.

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

```
Example 4.17
```

Use blocking probe to wait for an incoming message.

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 CALL MPI\_RECV(i, 1, MPI\_INTEGER, 0, 0, comm, status, ierr) ELSE CALL MPI\_RECV(x, 1, MPI\_REAL, 1, 0, comm, status, ierr) 

**Unofficial Draft for Comment Only** 

1	END IF
2	END DO
3	END IF
4 5	Each message is received with the right type.
6	Encourse 1, 4, 18 A similar means to the annuing second but more it has a making
7	<b>Example 4.18</b> A similar program to the previous example, but now it has a problem.
8	CALL MPI_COMM_RANK(comm, rank, ierr)
9 10	IF (rank.EQ.0) THEN
10	CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
12	ELSE IF (rank.EQ.1) THEN
13	CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
14	ELSE IF (rank.EQ.2) THEN
15	DO i=1, 2
16	CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
17	comm, status, ierr)
18	IF (status(MPI_SOURCE) .EQ. 0) THEN
19	100 CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
20	0, comm, status, ierr)
21	ELSE 200 CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
22	200 CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, 0, comm, status, ierr)
23	END IF
24	END DO
25	END IF
26	
27	In Example 4.18, the two receive calls in statements labeled 100 and 200 in Example 4.17
28	slightly modified, using MPI_ANY_SOURCE as the source argument. The program is now
29	incorrect: the receive operation may receive a message that is distinct from the message
30	probed by the preceding call to MPI_PROBE.
31	
32	Advice to users. In a multithreaded MPI program, MPI_PROBE and
33 34	MPI_IPROBE might need special care. If a thread probes for a message and then
35	immediately posts a matching receive, the receive may match a message other than
36	that found by the probe since another thread could concurrently receive that original message [2]. MPI_MPROBE and MPI_IMPROBE solve this problem by matching the
37	incoming message so that it may only be received with MPI_MRECV or MPI_IMRECV
38	on the corresponding message handle. ( <i>End of advice to users.</i> )
39	on the corresponding message handle. (End of dubice to users.)
40	Advice to implementors. A call to MPI_PROBE(source, tag, comm, status) will match
41	the message that would have been received by a call to MPI_RECV(, source, tag,
42	comm, status) executed at the same point. Suppose that this message has source s,
43	tag t and communicator c. If the tag argument in the probe call has value
44	MPI_ANY_TAG then the message probed will be the earliest pending message from
45	source s with communicator c and any tag; in any case, the message probed will be
46	the earliest pending message from source $s$ with tag $t$ and communicator $c$ (this is the
47	message that would have been received, so as to preserve message order). This message
48	continues as the earliest pending message from source $\boldsymbol{s}$ with tag $\boldsymbol{t}$ and communicator

 $\frac{4}{5}$ 

c, until it is received. A receive operation subsequent to the probe that uses the same communicator as the probe and uses the tag and source values returned by the probe, must receive this message, unless it has already been received by another receive operation. (*End of advice to implementors.*)

## 4.8.2 Matching Probe

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

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

MPI\_IMPROBE(source, tag, comm, flag, message, status)

IN	source	rank of source or $MPI\_ANY\_SOURCE$ (integer)
IN	tag	message tag or $MPI\_ANY\_TAG$ (integer)
IN	comm	communicator (handle)
OUT	flag	(logical)
OUT	message	returned message (handle)
OUT	status	status object (Status)

- MPI\_Improbe(source, tag, comm, flag, message, status, ierror)
   INTEGER, INTENT(IN) :: source, tag
   TYPE(MPI\_Comm), INTENT(IN) :: comm
   LOGICAL, INTENT(OUT) :: flag
   TYPE(MPI\_Message), INTENT(OUT) :: message
   TYPE(MPI\_Status) :: status
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
  MPI\_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
   INTEGER SOURCE, TAG, COMM

LOGICAL FLAG INTEGER MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR MPI\_IMPROBE(source, tag, comm, flag, message, status) returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received

by a call to MPI\_RECV(..., source, tag, comm, status) executed at the same point in the

program and returns in status the same value that would have been returned by MPI\_RECV.

In addition, it returns in message a handle to the matched message. Otherwise, the call 1 returns flag = false, and leaves status and message undefined.  $^{2}$ 

A matched receive (MPI\_MRECV or MPI\_IMRECV) executed with the message han-3 dle will receive the message that was matched by the probe. Unlike MPI\_IPROBE, no 4 other probe or receive operation may match the message returned by MPI\_IMPROBE. 5Each message returned by MPI\_IMPROBE must be received with either MPI\_MRECV or 6 MPI\_IMRECV. 7

The source argument of MPI\_IMPROBE can be MPI\_ANY\_SOURCE, and the tag argu-8 ment can be MPI\_ANY\_TAG, so that one can probe for messages from an arbitrary source 9 and/or with an arbitrary tag. However, a specific communication context must be provided 10 with the comm argument. 11

A synchronous send operation that is matched with MPI\_IMPROBE or MPI\_MPROBE 12will complete successfully only if both a matching receive is posted with MPI\_MRECV or 13 MPI\_IMRECV, and the receive operation has started to receive the message sent by the 14 synchronous send. 15

There is a special predefined message: MPI\_MESSAGE\_NO\_PROC, which is a message 16 which has MPI\_PROC\_NULL as its source process. The predefined constant 1718

MPI\_MESSAGE\_NULL is the value used for invalid message handles.

A matching probe with MPI\_PROC\_NULL as source returns flag = true, message = 19 MPI\_MESSAGE\_NO\_PROC, and the status object returns source = MPI\_PROC\_NULL, tag 20= MPI\_ANY\_TAG, and count = 0; see Section 4.11. It is not necessary to call MPI\_MRECV 21or MPI\_IMRECV with MPI\_MESSAGE\_NO\_PROC, but it is not erroneous to do so. 22

Rationale. MPI\_MESSAGE\_NO\_PROC was chosen instead of MPI\_MESSAGE\_PROC\_NULL to avoid possible confusion as another null handle constant. (End of rationale.)

```
29
     MPI_MPROBE(source, tag, comm, message, status)
```

```
30
       IN
                  source
                                               rank of source or MPI_ANY_SOURCE (integer)
31
       IN
                                               message tag or MPI_ANY_TAG (integer)
                  tag
32
33
       IN
                                               communicator (handle)
                 comm
34
        OUT
                                               returned message (handle)
                  message
35
       OUT
                 status
                                               status object (Status)
36
37
      int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message,
38
                     MPI_Status *status)
39
```

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

```
44
         TYPE(MPI_Status) :: status
45
```

```
INTEGER, OPTIONAL, INTENT(OUT) ::
                                              ierror
46
```

```
MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)
47
         INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
48
```

23

24

25

26 27 28

40

41

42

48

MPI\_MPROBE behaves like MPI\_IMPROBE except that it is a blocking call that returns 1 only after a matching message has been found. 2 The implementation of MPI\_MPROBE and MPI\_IMPROBE needs to guarantee progress 3 in the same way as in the case of MPI\_PROBE and MPI\_IPROBE. 4 54.8.3 Matched Receives 6 7 The functions MPI\_MRECV and MPI\_IMRECV receive messages that have been previously 8 matched by a matching probe (Section 4.8.2). 9 10 11 MPI\_MRECV(buf, count, datatype, message, status) 12 OUT buf initial address of receive buffer (choice) 13 IN number of elements in receive buffer (non-negative incount 14 teger) 1516IN datatype datatype of each receive buffer element (handle) 17INOUT message message (handle) 18 status object (Status) OUT status 19 2021int MPI\_Mrecv(void \*buf, int count, MPI\_Datatype datatype, MPI\_Message \*message, MPI\_Status \*status) 22 23 int MPI\_Mrecv(void \*buf, MPI\_Count count, MPI\_Datatype datatype, 24MPI\_Message \*message, MPI\_Status \*status) 2526int MPI\_Mrecv\_x(void \*buf, MPI\_Count count, MPI\_Datatype datatype, 27MPI\_Message \*message, MPI\_Status \*status) 28MPI\_Mrecv(buf, count, datatype, message, status, ierror) 29 TYPE(\*), DIMENSION(..) :: buf 30 INTEGER, INTENT(IN) :: count 31 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 32 TYPE(MPI\_Message), INTENT(INOUT) :: message 33 TYPE(MPI\_Status) :: status 34INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3536 MPI\_Mrecv(buf, count, datatype, message, status, ierror) 37 TYPE(\*), DIMENSION(..) :: buf 38 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 39 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 40 TYPE(MPI\_Message), INTENT(INOUT) :: message 41 TYPE(MPI\_Status) :: status 42 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43MPI\_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR) 44 <type> BUF(\*) 45INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 46 47This call receives a message matched by a matching probe operation (Section 4.8.2).

The receive buffer consists of the storage containing **count** consecutive elements of the 1 type specified by datatype, starting at address buf. The length of the received message must 2 be less than or equal to the length of the receive buffer. An overflow error occurs if all 3 incoming data does not fit, without truncation, into the receive buffer. 4 If the message is shorter than the receive buffer, then only those locations corresponding  $\mathbf{5}$ to the (shorter) message are modified. 6 On return from this function, the message handle is set to MPI\_MESSAGE\_NULL. All 7 errors that occur during the execution of this operation are handled according to the error 8 handler set for the communicator used in the matching probe call that produced the message 9 handle. 10 If MPI\_MRECV is called with MPI\_MESSAGE\_NO\_PROC as the message argument, the 11 call returns immediately with the status object set to source = MPI\_PROC\_NULL, tag =12 MPI\_ANY\_TAG, and count = 0, as if a receive from MPI\_PROC\_NULL was issued (see Sec-13 tion 4.11). A call to MPI\_MRECV with MPI\_MESSAGE\_NULL is erroneous. 14 15 16 MPI\_IMRECV(buf, count, datatype, message, request) 17OUT 18 buf initial address of receive buffer (choice) 19 IN count number of elements in receive buffer (non-negative in-20teger) 21IN datatype datatype of each receive buffer element (handle) 22INOUT message message (handle) 23 24OUT request communication request (handle) 2526 int MPI\_Imrecv(void \*buf, int count, MPI\_Datatype datatype, 27MPI\_Message \*message, MPI\_Request \*request) 2829 int MPI\_Imrecv(void \*buf, MPI\_Count count, MPI\_Datatype datatype, MPI\_Message \*message, MPI\_Request \*request) 30 31 int MPI\_Imrecv\_x(void \*buf, MPI\_Count count, MPI\_Datatype datatype, 32MPI\_Message \*message, MPI\_Request \*request) 33 34MPI\_Imrecv(buf, count, datatype, message, request, ierror) 35 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 36 INTEGER, INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: datatype 37 38 TYPE(MPI\_Message), INTENT(INOUT) :: message 39 TYPE(MPI\_Request), INTENT(OUT) :: request 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 MPI\_Imrecv(buf, count, datatype, message, request, ierror) 42 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 43INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 44 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 45 TYPE(MPI\_Message), INTENT(INOUT) :: message 46 TYPE(MPI\_Request), INTENT(OUT) :: request 47INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

MPI_IMRECV(BUF,	COUNT, DATATY	PE, MESSAGE, R	EQUEST, IERF	OR)
<type> BUF(*</type>	<)			
INTEGER COUN	IT, DATATYPE, N	MESSAGE, REQUE	ST, IERROR	

MPI\_IMRECV is the nonblocking variant of MPI\_MRECV and starts a nonblocking receive of a matched message. Completion semantics are similar to MPI\_IRECV as described in Section 4.7.2. On return from this function, the message handle is set to MPI\_MESSAGE\_NULL.

If MPI\_IMRECV is called with MPI\_MESSAGE\_NO\_PROC as the message argument, the call returns immediately with a request object which, when completed, will yield a status object set to source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0, as if a receive from MPI\_PROC\_NULL was issued (see Section 4.11). A call to MPI\_IMRECV with MPI\_MESSAGE\_NULL is erroneous.

Advice to implementors. If reception of a matched message is started with MPI\_IMRECV, then it is possible to cancel the returned request with MPI\_CANCEL. If MPI\_CANCEL succeeds, the matched message must be found by a subsequent message probe (MPI\_PROBE, MPI\_IPROBE, MPI\_MPROBE, or MPI\_IMPROBE), received by a subsequent receive operation or cancelled by the sender. See Section 4.8.4 for details about MPI\_CANCEL. The cancellation of operations initiated with MPI\_IMRECV may fail. (End of advice to implementors.)

4.8.4 Cancel

```
MPI_CANCEL(request)
 IN
           request
                                    communication request (handle)
int MPI_Cancel(MPI_Request *request)
MPI_Cancel(request, ierror)
    TYPE(MPI_Request), INTENT(IN) ::
                                        request
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                         ierror
MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

A call to MPI\_CANCEL marks for cancellation a pending, nonblocking communica-37 tion operation (send or receive). Cancelling a send request by calling MPI\_CANCEL is 38 deprecated. The cancel call is local. It returns immediately, possibly before the communication is actually cancelled. It is still necessary to call MPI\_REQUEST\_FREE, MPI\_WAIT or MPI\_TEST (or any of the derived operations) with the cancelled request as argument after the call to MPI\_CANCEL. If a communication is marked for cancellation, then a MPI\_WAIT call for that communication is guaranteed to return, irrespective of the activities of other processes (i.e., MPI\_WAIT behaves as a local function); similarly if MPI\_TEST is repeatedly called in a busy wait loop for a cancelled communication, then MPI\_TEST will eventually be successful.

MPI\_CANCEL can be used to cancel a communication that uses a persistent request (see 47Section 4.9), in the same way it is used for nonpersistent requests. Cancelling a persistent 48

5

6

7

8

9

10

11

12

13 14

15

16

17

18

19

20

2122

23 2425

26

27

2829

30

31

32

33 34

35

36

39

40

41

42

43

44

45

send request by calling MPI\_CANCEL is deprecated. 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. Cancelling a buffered send request by calling MPI\_CANCEL is deprecated. Either the cancellation succeeds, or the communication succeeds, but not both. If a

send is marked for cancellation, which is deprecated, then it must be the case that either the send completes normally, in which case the message sent was received at the destination process, or that the send is successfully cancelled, in which case no part of the message was received at the destination. Then, any matching receive has to be satisfied by another send. If a receive is marked for cancellation, then it must be the case that either the receive completes normally, or that the receive is successfully cancelled, in which case no part of the receive buffer is altered. Then, any matching send has to be satisfied by another receive.

<sup>15</sup> If the operation has been cancelled, then information to that effect will be returned in <sup>16</sup> the status argument of the operation that completes the communication.

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

```
21
22
23
```

17

18

19

20

```
24
```

MPI\_TEST\_CANCELLED(status, flag)

25IN status status object (Status) 2627OUT (logical) flag 2829 int MPI\_Test\_cancelled(const MPI\_Status \*status, int \*flag) 30 MPI\_Test\_cancelled(status, flag, ierror) 31 TYPE(MPI\_Status), INTENT(IN) :: status 32LOGICAL, INTENT(OUT) :: flag 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35MPI\_TEST\_CANCELLED(STATUS, FLAG, IERROR) 36 INTEGER STATUS (MPI\_STATUS\_SIZE) 37 LOGICAL FLAG 38 INTEGER IERROR 39

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

46 Advice to users. Cancel can be an expensive operation that should be used only 47 exceptionally. (End of advice to users.)

48

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 (with the exception of the buffer contents) 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 operations. In the case of point-to-point communication, 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 point-to-point request be received by a receive operation using a persistent point-to-point request, or vice versa.

There are also collective communication persistent operations defined in Section 5.13 and Section 7.8. The remainder of this section covers the point-to-point persistent initialization operations and the start routines, which are used for both point-to-point and collective persistent communication.

A persistent point-to-point communication request is created using one of the five following calls. These point-to-point persistent calls involve no communication.

MPI\_SEND\_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)	34
IN	count	number of elements in send buffer (non-negative inte-	35 36
		ger)	37
IN	datatype	type of each element (handle)	38
IN	dest	rank of destination (integer)	39
IN	tag	message tag (integer)	40 41
IN	comm	communicator (handle)	42
OUT	request	communication request (handle)	43
		- 、 ,	44
			45

```
int MPI_Send_init(const void *buf, MPI_Count count, MPI_Datatype datatype,
1
                   int dest, int tag, MPI_Comm comm, MPI_Request *request)
^{2}
3
     int MPI_Send_init_x(const void *buf, MPI_Count count,
4
                   MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
5
                   MPI_Request *request)
6
7
     MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
8
         INTEGER, INTENT(IN) :: count, dest, tag
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
15
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) ::
                                                          count
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         INTEGER, INTENT(IN) :: dest, tag
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
24
         <type> BUF(*)
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
25
26
         Creates a persistent communication request for a standard mode send operation, and
27
     binds to it all the arguments of a send operation.
28
29
30
     MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)
31
       IN
                buf
                                           initial address of send buffer (choice)
32
       IN
                count
                                           number of elements sent (non-negative integer)
33
34
       IN
                datatype
                                           type of each element (handle)
35
       IN
                dest
                                           rank of destination (integer)
36
       IN
                tag
                                           message tag (integer)
37
38
       IN
                comm
                                           communicator (handle)
39
       OUT
                                           communication request (handle)
                request
40
41
     int MPI_Bsend_init(const void *buf, int count, MPI_Datatype datatype,
42
                   int dest, int tag, MPI_Comm comm, MPI_Request *request)
43
44
     int MPI_Bsend_init(const void *buf, MPI_Count count, MPI_Datatype datatype,
45
                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
46
     int MPI_Bsend_init_x(const void *buf, MPI_Count count,
47
                   MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
48
```

MPI\_Request \*request) 1 2 MPI\_Bsend\_init(buf, count, datatype, dest, tag, comm, request, ierror) 3 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 4 INTEGER, INTENT(IN) :: count, dest, tag 5TYPE(MPI\_Datatype), INTENT(IN) :: datatype 6 TYPE(MPI\_Comm), INTENT(IN) :: comm 7 TYPE(MPI\_Request), INTENT(OUT) :: request 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 MPI\_Bsend\_init(buf, count, datatype, dest, tag, comm, request, ierror) 10 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 11 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 12 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 13INTEGER, INTENT(IN) :: dest, tag 14 TYPE(MPI\_Comm), INTENT(IN) :: comm 15TYPE(MPI\_Request), INTENT(OUT) :: request 16 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1718 MPI\_BSEND\_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 19 <type> BUF(\*) 20INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 21Creates a persistent communication request for a buffered mode send. 22 23 24MPI\_SSEND\_INIT(buf, count, datatype, dest, tag, comm, request) 2526IN buf initial address of send buffer (choice) 27IN number of elements sent (non-negative integer) count 28 IN type of each element (handle) datatype 29 IN dest rank of destination (integer) 30 31 IN message tag (integer) tag 32 communicator (handle) IN comm 33 OUT 34communication request (handle) request 3536 int MPI\_Ssend\_init(const void \*buf, int count, MPI\_Datatype datatype, 37 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 38 int MPI\_Ssend\_init(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 39 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 40 41int MPI\_Ssend\_init\_x(const void \*buf, MPI\_Count count, 42 MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm, 43MPI\_Request \*request) 44

MPI\_Ssend\_init(buf, count, datatype, dest, tag, comm, request, ierror) 45
TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 46
INTEGER, INTENT(IN) :: count, dest, tag 47
TYPE(MPI\_Datatype), INTENT(IN) :: datatype 48

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

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

12		_INIT(BUF, >> BUF(*)	COUNT, D	ATATYPE	, DESI	I, TAG	, COM	MM, R	EQUEST	, IERF	ROR)	
3	INTEG	GER COUNT,	DATATYPE	, DEST,	TAG,	COMM,	REQU	JEST,	IERRC	)R		
4 5 6 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 com-											
8 9	-	n is attached		-								
10 11	A communication (send or receive) that uses a persistent request is initiated by the function MPI_START.											
12 13	MPI_STAF	RT(request)										
14 15	INOUT	request			comm	unicatio	on requ	uest (l	nandle)			
16 17	int MPI_S	Start(MPI_H	Request *	request	)							
18		(request,										
19 20		(MPI_Reques ER, OPTION				requ ierror						
21		-	-	NI(UUI)	:: 1	error						
22 23	MPI_START(REQUEST, IERROR) INTEGER REQUEST, IERROR											
24 25 26 27 28	The argument, request, is a handle returned by one of the previous five calls. The associated request should be inactive. The request becomes active once the call is made. If the request is for a send with ready mode, then a matching receive should be posted before the call is made. The communication buffer should not be modified after the call,											
29 30 31 32	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											
$33 \\ 34$	in the same manner as a call to MPI_IBSEND; and so on.											
35												
36	MPI_STAF	RTALL(count	t, array_of_	requests	)							
37 38	IN	count			list ler	ngth (no	on-neg	ative	integer)			
39	INOUT	array_of_r	equests		array	of reque	ests (a	rray o	f handle	es)		
40 41	int MPI_S	Startall(in	nt count,	MPI_Red	quest	array	_of_r	reque	sts[])	i i		
42	int MPI_S	Startall(MI	PI_Count	count, N	MPI_Re	equest	arra	ay_of	_reque	ests[])	)	
$43 \\ 44$	int MPI_S	Startall_x	(MPI_Coun	t count	, MPI_	Reque	st ar	ray_	of_req	uests	[])	
45 46 47	INTEC	Call(count) GER, INTEN (MPI_Reques	Γ(IN) ::	count				requ	ests (c	ount)		
48		<b></b> 9400	,		,		J	14				

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
MPI_Startall(count, array_of_requests, ierror)	
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR) INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR	

Start all communications associated with requests in array\_of\_requests. A call to MPI\_STARTALL(count, array\_of\_requests) has the same effect as calls to MPI\_START (&array\_of\_requests[i]), executed for i=0, ..., 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. Collective operation requests (defined in Section 5.12 and Section 7.7 for nonblocking collective operations, and Section 5.13 and Section 7.8 for persistent collective operations) must not be freed while active. It is preferable, in general, to free requests when they are inactive. If this rule is followed, then the functions described in this section will be invoked in a sequence of the form,

### Create (Start Complete)\* Free

where \* indicates zero or more repetitions. If the same communication object is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is obeyed.

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

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

## 4.10 Send-Receive

The **send-receive** operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic

## 

 $46 \\ 47$ 

dependencies that may lead to deadlock. When a send-receive operation is used, the com-1 munication subsystem takes care of these issues. The send-receive operation can be used 2 in conjunction with the functions described in Chapter 7 in order to perform shifts on var-3 ious logical topologies. Also, a send-receive operation is useful for implementing remote 4 procedure calls. 5A message sent by a send-receive operation can be received by a regular receive oper-6 ation or probed by a probe operation; a send-receive operation can receive a message sent 7 by a regular send operation. 8 9 10 MPI\_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, 11 source, recvtag, comm, status) 12IN sendbuf initial address of send buffer (choice) 1314 IN sendcount number of elements in send buffer (non-negative inte-15 ger) 16 IN sendtype type of elements in send buffer (handle) 17 dest rank of destination (integer) 18 IN 19 IN sendtag send tag (integer) 20OUT recvbuf initial address of receive buffer (choice) 21IN number of elements in receive buffer (non-negative in-22recvcount teger) 23 24IN recvtype type of elements receive buffer element (handle) 25rank of source or MPI\_ANY\_SOURCE (integer) IN source 26 27IN recvtag receive tag or MPI\_ANY\_TAG (integer) 28IN comm communicator (handle) 29 OUT status object (Status) status 30 31 int MPI\_Sendrecv(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 32int dest, int sendtag, void \*recvbuf, int recvcount, 33 MPI\_Datatype recvtype, int source, int recvtag, MPI\_Comm comm, 34 MPI\_Status \*status) 3536 int MPI\_Sendrecv(const void \*sendbuf, MPI\_Count sendcount, 37 MPI\_Datatype sendtype, int dest, int sendtag, void \*recvbuf, 38 MPI\_Count recvcount, MPI\_Datatype recvtype, int source, 39 int recvtag, MPI\_Comm comm, MPI\_Status \*status) 40 int MPI\_Sendrecv\_x(const void \*sendbuf, MPI\_Count sendcount, 41 MPI\_Datatype sendtype, int dest, int sendtag, void \*recvbuf, 42 MPI\_Count recvcount, MPI\_Datatype recvtype, int source, 43int recvtag, MPI\_Comm comm, MPI\_Status \*status) 44 45MPI\_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, 46recvcount, recvtype, source, recvtag, comm, status, ierror) 47TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 48

```
INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
                                                                                  1
    recvtag
                                                                                  2
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  3
    TYPE(*), DIMENSION(..) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  5
    TYPE(MPI_Status) :: status
                                                                                  6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  8
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
             recvcount, recvtype, source, recvtag, comm, status, ierror)
                                                                                  10
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  11
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  12
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  13
    INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
                                                                                  14
    TYPE(*), DIMENSION(..)
                            :: recvbuf
                                                                                  15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  16
    TYPE(MPI_Status) :: status
                                                                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
                                                                                  19
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
                                                                                  20
    <type> SENDBUF(*)
                                                                                  21
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG
                                                                                  22
    <type> RECVBUF(*)
                                                                                  23
    INTEGER RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM,
                                                                                  24
    STATUS(MPI_STATUS_SIZE), IERROR
                                                                                  25
                                                                                  26
    Execute a blocking send and receive operation. Both send and receive use the same
                                                                                  27
```

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.

28

29

30

31

MPI\_SENDRECV\_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, sta-1 tus)  $^{2}$ 3 INOUT buf initial address of send and receive buffer (choice) 4 IN count number of elements in send and receive buffer (non-5negative integer) 6 IN 7 datatype type of elements in send and receive buffer (handle) 8 IN dest rank of destination (integer) 9 IN sendtag send message tag (integer) 10 11 IN source rank of source or MPI\_ANY\_SOURCE (integer) 12receive message tag or MPI\_ANY\_TAG (integer) IN recvtag 13IN communicator (handle) comm 14 OUT status status object (Status) 151617int MPI\_Sendrecv\_replace(void \*buf, int count, MPI\_Datatype datatype, 18 int dest, int sendtag, int source, int recvtag, MPI\_Comm comm, 19 MPI\_Status \*status) 20int MPI\_Sendrecv\_replace(void \*buf, MPI\_Count count, MPI\_Datatype datatype, 21int dest, int sendtag, int source, int recvtag, MPI\_Comm comm, 22MPI\_Status \*status) 23 24int MPI\_Sendrecv\_replace\_x(void \*buf, MPI\_Count count, 25MPI\_Datatype datatype, int dest, int sendtag, int source, 26int recvtag, MPI\_Comm comm, MPI\_Status \*status) 27MPI\_Sendrecv\_replace(buf, count, datatype, dest, sendtag, source, recvtag, 28comm, status, ierror) 29 TYPE(\*), DIMENSION(..) :: buf 30 INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag 31 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 32 TYPE(MPI\_Comm), INTENT(IN) :: comm 33 TYPE(MPI\_Status) :: status 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 36 MPI\_Sendrecv\_replace(buf, count, datatype, dest, sendtag, source, recvtag, 37 comm, status, ierror) 38 TYPE(\*), DIMENSION(..) :: buf 39 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 40 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 41INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag 42 TYPE(MPI\_Comm), INTENT(IN) :: comm 43TYPE(MPI\_Status) :: status 44 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 45MPI\_SENDRECV\_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, 46 COMM, STATUS, IERROR) 47<type> BUF(\*) 48

INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR

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

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

#### 4.11 Null Processes

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

The special value MPI\_PROC\_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI\_PROC\_NULL has no effect. A send to MPI\_PROC\_NULL succeeds and returns as soon as possible. A receive from MPI\_PROC\_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with  $source = MPI_PROC_NULL$  is executed then the status object returns source = MPI\_PROC\_NULL,  $tag = MPI_ANY_TAG$  and count = 0. A probe or matching probe with source  $= MPI_PROC_NULL$  succeeds and returns as soon as possible, and the status object returns source  $= MPI_PROC_NULL$ , tag  $= MPI_ANY_TAG$  and count = 0. A matching probe (cf. Section 4.8.2) with MPI\_PROC\_NULL as source returns  $flag = true, message = MPI_MESSAGE_NO_PROC, and the status object returns source =$ MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0.

# Bibliography

## [1] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009. 4.8.2

## [2] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. 4.8.1, 4.8.2

## Index

CONST:a, 14 CONST:MPI\_DOUBLE\_PRECISION, 3 CONST:b, 14 CONST:MPI\_ERR\_IN\_STATUS, 9, 11, 34, CONST:bool, 5 42, 44 CONST:char\*\*, 26 CONST:MPI\_ERR\_PENDING, 42 CONST:CHARACTER, 13, 14 CONST:MPI\_ERROR, 9, 34 CONST: false, 34 CONST:MPI\_FLOAT, 4  $CONST:message = MPI_MESSAGE_NO_PROCONST:MPI_INT, 4$ CONST:MPI\_INT16\_T, 4 52.67 CONST:MPI\_ADDRESS\_KIND, 5 CONST:MPI\_INT32\_T, 4 CONST:MPI\_AINT, 5 CONST:MPI\_INT64\_T, 4 CONST:MPI\_Aint, 5 CONST:MPI\_INT8\_T, 4 CONST: MPI\_ANY\_SOURCE, 7, 8, 20, 33, CONST:MPI\_INTEGER, 3, 12 34, 48–52, 61, 64, 65 CONST:MPI\_INTEGER1, 4 CONST:MPI\_ANY\_TAG, 7, 8, 10, 33, 34, CONST:MPI\_INTEGER2, 4 48-52, 54, 55, 61, 64, 65, 67 CONST:MPI\_INTEGER4, 4 CONST:MPI\_BOTTOM, 11 CONST:MPI\_LOGICAL, 3 CONST:MPI\_BSEND\_OVERHEAD, 27 CONST:MPI\_LONG, 4 CONST:MPI\_BYTE, 3, 4, 12, 13, 15 CONST:MPI\_LONG\_DOUBLE, 4 CONST:MPI\_C\_BOOL, 4 CONST:MPI\_LONG\_LONG, 4 CONST:MPI\_C\_COMPLEX, 4 CONST:MPI\_LONG\_LONG\_INT, 4 CONST:MPI\_C\_DOUBLE\_COMPLEX, 4 CONST:MPI\_Message, 51 CONST:MPI\_MESSAGE\_NO\_PROC, 52, 54, CONST:MPI\_C\_FLOAT\_COMPLEX (as a synonym), 4 55 CONST:MPI\_C\_LONG\_DOUBLE\_COMPLE&ONST:MPI\_MESSAGE\_NULL, 52, 54, 55 4 CONST:MPI\_OFFSET, 5 CONST:MPI\_CHAR, 4, 15 CONST:MPI\_Offset, 5 CONST:MPI\_CHARACTER, 3, 13-15 CONST:MPI\_OFFSET, 5 CONST:MPI\_Comm, 2 CONST:MPI\_OFFSET\_KIND, 5 CONST:MPI\_COMM\_WORLD, 6 CONST:MPI\_PACKED, 3, 4, 12 CONST:MPI\_PROC\_NULL, 2, 6, 8, 49, 52, CONST:MPI\_COMPLEX, 3 CONST:MPI\_COUNT, 5 54, 55, 66, 67 CONST:MPI\_Count, 5 CONST:MPI\_REAL, 3, 12 CONST:MPI\_COUNT\_KIND, 5 CONST:MPI\_REAL2, 4 CONST:MPI\_CXX\_BOOL, 5 CONST:MPI\_REAL4, 4 CONST:MPI\_CXX\_DOUBLE\_COMPLEX, 5 CONST:MPI\_REAL8, 4 CONST:MPI\_CXX\_FLOAT\_COMPLEX, 5 CONST:MPI\_Request, 28-32, 34, 35, 36, 39-CONST:MPI\_CXX\_LONG\_DOUBLE\_COMPLEX, 44, 47, 55, 57-62 CONST:MPI\_REQUEST\_NULL, 34-36, 39-5 CONST:MPI\_DOUBLE, 4 41, 43, 44 CONST:MPI\_DOUBLE\_COMPLEX, 4 CONST:MPI\_SHORT, 4

CONST:MPI\_SIGNED\_CHAR, 4 1 CONST:MPI\_SOURCE, 9  $^{2}$ CONST:MPI\_Status, 6, 9, 11, 34, 35, 39-44, 3 47-49, 51-53, 56, 64, 65 4 CONST:MPI\_STATUS\_IGNORE, 11  $\mathbf{5}$ CONST:MPI\_STATUS\_SIZE, 9 6 CONST:MPI\_STATUSES\_IGNORE, 11 7 CONST:MPI\_SUCCESS, 34, 42, 44 8 CONST:MPI\_TAG, 9 9 CONST:MPI\_TAG\_UB, 6 10 CONST:MPI\_UINT16\_T, 4 11CONST:MPI\_UINT32\_T, 4 12CONST:MPI\_UINT64\_T, 4 13 CONST:MPI\_UINT8\_T, 4 14 CONST:MPI\_UNDEFINED, 10, 39-41, 44, 1516 45 CONST:MPI\_UNSIGNED, 4 17CONST:MPI\_UNSIGNED\_CHAR, 4 18 CONST:MPI\_UNSIGNED\_LONG, 4 19 CONST:MPI\_UNSIGNED\_LONG\_LONG, 4 20CONST:MPI\_UNSIGNED\_SHORT, 4 21CONST:MPI\_WCHAR, 4 22CONST:NULL, 11 23CONST:std::complex<double>, 5 24CONST:std::complex<float>, 5 25CONST:std::complex<long double>, 5 26CONST: true, 40 27CONST:void\*, 26 28CONST:void\*\*, 26 29 30 EXAMPLES: Client-server code, 46 31 with blocking probe, 49 32with blocking probe, wrong, 5033 EXAMPLES: Datatypes 34 matching, 12 35 not matching, 13 36 untyped, 1337 EXAMPLES:Deadlock 38 if not buffered, 23 39 wrong message exchange, 22 40 EXAMPLES: Intertwined matching pairs, 21 41 EXAMPLES:Message exchange (ping-pong), 42 2243EXAMPLES:MPI\_BSEND, 21 44 EXAMPLES:MPI\_Buffer\_attach, 25 45 EXAMPLES:MPI\_Buffer\_detach, 25 46EXAMPLES:MPI\_BYTE, 13 47EXAMPLES:MPI\_CHARACTER, 13 48

EXAMPLES:MPI\_IRECV, 36–38, 46 EXAMPLES:MPI\_ISEND, 36-38, 46 EXAMPLES:MPI\_PROBE, 49, 50 EXAMPLES:MPI\_RECV, 12, 13, 21-23, 38, 49.50 EXAMPLES:MPI\_REQUEST\_FREE, 37 EXAMPLES: MPI\_SEND, 12, 13, 22, 23, 38, 49, 50 EXAMPLES:MPI\_SSEND, 21, 38 EXAMPLES:MPI\_WAIT, 36-38, 46 EXAMPLES:MPI\_WAITANY, 46 EXAMPLES:MPI\_WAITSOME, 46 EXAMPLES:Non-overtaking messages, 21 EXAMPLES:Nonblocking operations, 36, 37 message ordering, 38progress, 38 MPI\_BSEND, 26 MPI\_BSEND(buf, count, datatype, dest, tag,  $\operatorname{comm}$ ), <u>17</u> MPI\_BSEND\_INIT, 62 MPI\_BSEND\_INIT(buf, count, datatype, dest, tag, comm, request), 58 MPI\_BUFFER\_ATTACH, 35 MPI\_BUFFER\_ATTACH(buffer, size), 24 MPI\_BUFFER\_DETACH(buffer\_addr, size), 25MPI\_CANCEL, 20, 35, 47, 48, 55–57 MPI\_CANCEL(request), 55 MPI\_GET\_COUNT, 9-11, 34 MPI\_GET\_COUNT(status, datatype, count), 9 MPI\_GET\_ELEMENTS, 34 MPI\_GET\_ELEMENTS\_X, 34 MPI\_IBSEND, 35, 62 MPI\_IBSEND(buf, count, datatype, dest, tag, comm, request), 30MPI\_IMPROBE, 47, 50–53, 55 MPI\_IMPROBE(source, tag, comm, flag, message, status), 51, 51MPI\_IMRECV, 50, 52, 53, 55 MPI\_IMRECV(buf, count, datatype, message, request), 54 MPI\_IPROBE, 10, 47-53, 55 MPI\_IPROBE(source, tag, comm, flag, status), 48, 48 MPI\_IRECV, 55

MPI_IRECV(buf, count, datatype, source,	status), $\underline{64}$	1
tag, comm, request), $\underline{33}$	MPI_SENDRECV_REPLACE(buf, count, data	tyzpe,
MPI_IRSEND(buf, count, datatype, dest, tag,	dest, sendtag, source, recvtag, comm,	3
comm, request), $\underline{32}$	status), $\underline{65}$	4
MPI_ISEND, 62	MPI_SSEND(buf, count, datatype, dest, tag,	5
MPI_ISEND(buf, count, datatype, dest, tag,	comm), $\underline{18}$	6
comm, request), $\underline{29}$	MPI_SSEND_INIT(buf, count, datatype, dest,	7
MPI_ISSEND(buf, count, datatype, dest, tag,	tag, comm, request), $\underline{59}$	8
comm, request), $\underline{31}$	MPI_START, 62, 63	9
MPI_MPROBE, 47, 50–53, 55	$MPI\_START(request), \underline{62}$	10
MPI_MPROBE(source, tag, comm, message,	MPI_STARTALL, 63	11
status), $\underline{52}$	MPI_STARTALL(count, array_of_requests),	12
MPI_MRECV, 50, 52–55	$\underline{62}, 63$	13
MPI_MRECV(buf, count, datatype, message,	MPI_TEST, 11, 34, 36–38, 41, 47, 55, 56, 63	14
status), $53$	MPI_TEST( &array_of_requests[i], flag, sta-	15
MPI_PACK, 27	tus), 41	16
MPI_PACK_SIZE(count, datatype, comm, siz	eMPI_TEST(request, flag, status), <u>35</u>	17
26	MPI_TEST_CANCELLED, 34–36, 56	18
MPI_PROBE, 8, 10, 11, 47–51, 53, 55	MPI_TEST_CANCELLED(status, flag), <u>56</u>	19
	MPI_TESTALL, 39, 43	20
50	MPI_TESTALL(count, array_of_requests, flag,	21
MPI_RECV, 2, 9–11, 51	$array_of_statuses), \frac{42}{2}$	22
MPI_RECV(, source, tag, comm, status),	MPI_TESTANY, 38, 41, 45	23
, , , , , , , , , , , , , , , , , , , ,	MPI_TESTANY(count, array_of_requests, in-	24
MPI_RECV(), 48	dex, flag, status), $40$	25
	MPI_TESTANY(count, array_of_requests, in-	26
comm, status), <u>7</u>	dex, status), 41	27
MPI_RECV_INIT, 62	MPI_TESTSOME, 39, 45, 46	28
,	MPI_TESTSOME(incount, array_of_requests,	29
tag, comm, request), $\underline{61}$	outcount, array_of_indices, array_of_st	
MPI_REQUEST_FREE, 36, 37, 55, 63	44	31
MPI_REQUEST_FREE(request), <u>36</u>	MPI_TYPE_CREATE_DARRAY, 10	32
MPI_REQUEST_GET_STATUS, 11, 47	MPI_WAIT, 9, 11, 34–38, 40, 42, 55, 56, 63	33
	g,MPI_WAIT(&array_of_request[i], &array_of_st	
status), <u>47</u>	42	35
,,	MPI_WAIT(&array_of_requests[i], status), 39	36
comm), <u>19</u>	MPI_WAIT(request, status), <u>34</u>	37
MPI_RSEND_INIT(buf, count, datatype, dest		38
tag, comm, request), $\underline{60}$	MPI_WAITALL(count, array_of_requests, ar-	39
MPI_SEND, 1, 3, 11, 13	ray_of_statuses), <u>41</u> , 41	40
MPI_SEND(buf, count, datatype, dest, tag,	MPI_WAITANY, 20, 38, 40, 45	40
comm), <u>2</u>	MPI_WAITANY(count, array_of_requests, in-	41
$MPI_SEND_INIT, 62$	dex, status), <u>39</u> , 39	42
MPI_SEND_INIT(buf, count, datatype, dest,		43
tag, comm, request), <u>57</u>	MPI_WAITSOME(incount, array_of_requests,	44 45
MPI_SENDRECV(sendbuf, sendcount, send-	outcount, array_of_indices, array_of_st	
type, dest, sendtag, recvbuf, recv-	$\underline{43}$	47

count, recvtype, source, recvtag, comm,

TERM: active, 34 1 TERM:blocking, <u>16</u>, 19  $^{2}$ 3 TERM: buffer allocation, 24 TERM:buffered, <u>16</u>, 27, <u>28</u> 4 nonblocking, 28  $\mathbf{5}$ TERM: buffered send, 206 TERM:cancel, 35, 47, 55  $\overline{7}$ **TERM**:communication 8 modes, **16** 9 point-to-point, 1 10 TERM:communicator, 6 11 TERM:completion, 34 12multiple, 38 13 TERM:conversion, 14 14 TERM:data, 3 15TERM:data conversion, 14 16TERM:empty, <u>34</u> 17TERM:envelope, 1, 5 18 TERM: fairness, 21, 45 19 TERM: immediate, 28 20TERM:inactive, 34 21TERM: initiation, 28 22TERM:local, 16 23 TERM:matched receives, 53 24TERM:matching 25type, **11** 26 TERM:matching probe, **51** 27TERM:message, 1 28 data, 3 29 envelope, 5 30 TERM:modes, 16 31 TERM:multiple completions, 38 32TERM:non-local, 16, 17 33 TERM:nonblocking, 27 34 communication, 2735 completion, 34 36 initiation, 28 37 request objects, 28 38 TERM:null handle, 34 39 TERM:null processes, 66 40 TERM: persistent communication requests, 37, 41 42 57 TERM:point-to-point communication, 1 43TERM:probe, 48 44 TERM:probe, matching, **51** 45TERM: process group, 6 46TERM:ready, <u>17</u>, 27, <u>28</u> 47

48 nonblocking,  $\underline{27}$ 

TERM:ready send, 20 TERM:receive, 1, 2, 6 buffer, 2 complete, 27start call, 27 TERM:request objects, 28 **TERM**:semantics nonblocking communications, 37 point-to-point communication, 20 TERM:send, 1, 2 buffer, 1 complete, 27start, 27 TERM:send-receive, 63 TERM:standard, 16, 27 nonblocking, 28 TERM:standard send, 20 TERM: status, 8 ignore, 11 test, **47** TERM:synchronous, <u>16</u>, 27, <u>28</u>, <u>34</u> nonblocking, 27TERM:synchronous send, 20 TERM:type matching, 11 TERMnoindex:buffered, 28 TERMnoindex:message envelope, 5 TERMnoindex:nonblocking communication, 27TERMnoindex:persistent, 57 TERMnoindex:ready, 27TERMnoindex:receive buffer, 2 TERMnoindex:receive complete, 27TERMnoindex: receive start call, 27 TERMnoindex:request, 28 TERMnoindex:send buffer, 1 TERMnoindex:send complete, 27 TERMnoindex:send start, 27 TERMnoindex:send-receive, 63 TERMnoindex:standard, 28 TERMnoindex:synchronous, 27