# Point to Point Chapter Changes in MPI 4.1

MPI Forum Meeting, September 2023, Bristol, UK

On behalf of the Point to Point Chapter Committee

# Point-to-point chapter committee membership

Chair: Dan Holmes

Members: Ken Raffenetti, Ryan Grant, Bill Gropp, Brian Smith

# All PRs that touched the point to point chapter

PR #871 Update to Credits
PR #848 Fix 'in [+the +]case above'
PR #858 Pre-RC4.1 fixes in the point-to-point chapter
PR #736 added MPI_Buffer_attach_comm and MPI_Buffer_detach_comm
PR #808 Issue685 iflush on top of extensive message buffer updates
PR #856 Fix warnings from LaTeX and buildindices
PR #850 Review use of %ALLOWLATEX% and address issues
PR #820 Definition 'noncollective procedure' added, 'nonpersistent' substituted
PR #825 Add definition of 'pending operation' in terms-2.tex
PR #823 'Pending communication' not defined in definition of MPI_Comm_disconnect
PR #750 Add condition stating when MPI_WAIT is nonlocal
PR #799 Add multiple request status funcs
PR #833 Update for getlatex and fixes for explicit font commands
PR #832 Second attempt to apply PR 829 changes
PR #831 Revert "Merge pull request #829"
PR #829 Deprecate X procedures
PR #818 Correct the MPI_TESTANY 'as if' wording
PR #767 add MPI_Status_{set,get} for 3 status fields
PR #203 Fix for issue 188
PR #753 Progress - new section in terms
PR #606 Suggested changes to disambiguate 'rank' in the pt2pt chapter

PR #722 Deprecate mpif.h
PR #748 Add advice to everyone about ambiguity of logically concurrent
PR #667 Clarify progress rule of MPI_REQUEST_GET_STATUS
PR #706 Miscellaneous minor edits
PR #704 Designate main index entry for MPI_REQUEST_NULL
PR #702 Improve primary index entries
PR #698 Editor fixes
PR #697 Improve example index
PR #692 Consistent description usage
PR #689 Fix table formatting and squash overfull boxes
PR #686 Consistent formatting for lists of MPI constants and objects.
PR #685 Replaces use of \const with more specific macros
PR #683 Which hunt - replace with that where appropriate
PR #680 Correct use of paragraph command
PR #677 Change non-xxx to nonxxx in most cases.
PR #675 Review and update index for case, font, missing persistent send and receive entries
PR #631 Implementation of enhanced example index.
PR #656 This ... is solely formatting changes
PR #642 After review by all chapters, merging the new format changes.
PR #605 Remove 'on two processes' from progress rule

```
% git log --first-parent --pretty=format:"%Cred%h%Creset %cs %Cgreen%<(25,trunc)%s%Creset %b" mpi-4.0..mpi-41-rc1 chap-pt2pt
```

# Substantive changes in the point to point chapter

| | | | |
|---|---|---|---|
| Issue #498 | 13/09/2021 | PR #605 | Remove 'on two processes' from progress rule |
| Issue #468 | 09/02/2023 | PR #667 | Clarify progress rule of MPI_REQUEST_GET_STATUS |
| Issue #117 | 21/03/2023 | PR #748 | Add advice to everyone about ambiguity of logically concurrent |
| Issue #472 | 27/03/2023 | PR #606 | Suggested changes to disambiguate 'rank' in the pt2pt chapter |
| Issue #188 | 06/04/2023 | PR #203 | Example 3.10 is "unsafe" but not erroneous |
| Issue #645 | 03/05/2023 | PR #767 | Add MPI_Status_{set,get} for 3 status fields |
| Issue #681 | 03/05/2023 | PR #818 | Correct the MPI_TESTANY 'as if' wording |
| Issue #519 | 12/07/2023 | PR #799 | Add multiple request status funcs |
| Issue #639 | 12/07/2023 | PR #750 | Add condition stating when MPI_WAIT is nonlocal |
| Issue #676 | 13/07/2023 | PR #825 | Add definition of 'pending operation' in terms-2.tex |
| Issue #679 | 13/07/2023 | PR #820 | Definition 'noncollective procedure' added, 'nonpersistent' substituted |
| Issue #710 | 13/07/2023 | PR #823 | 'Pending communication' not defined |
| Issue #685 | 27/07/2023 | PR #808 | Add Iflush procedures to new buffered mode send buffer handling |
| Issue #586 | 07/08/2023 | PR #736 | Added MPI_Buffer_attach_comm and MPI_Buffer_detach_comm |

https://github.com/orgs/mpi-forum/projects/1/views/11?filterQuery=status%3Adone%2C%22Passed+2nd+Vote%22%2C%22Passed+1st+Vote%22%2C%22Passing+final+vote%22+label%3A%22mpi-4.1%22+-reason%3A%22not+planned%22++label%3Achap-p2p

# Items needed for point to point chapter to be ready

Page 33 line 22 (of Draftable diff between mpi-4.0 and mpi-41-rc):
“returned in the detach procedure. The value returned”
->“returned in the detach procedure and the value returned”

Page 33 line 30 (of Draftable diff between mpi-4.0 and mpi-41-rc):
“These procedure will block until” -> “These procedures will delay their return until”

Page 36 line 15-33 (of Draftable diff between mpi-4.0 and mpi-41-rc):
Example 3.12 exhibits incorrect formatting

Page 35 line 48 (of Draftable diff between mpi-4.0 and mpi-41-rc):
Widow/orphan control for footnote from page 34

# Detail of changes – full chapter PDF diff

Comparison of single-chapter PDF files (between mpi-4.0 tag and mpi-41-rc tag):

https://draftable.com/compare/fLjitoDCyeqF

```
$ git diff -U0 mpi-4.0 mpi-41-rc1 chap-pt2pt
```

Produces 1928 lines of output :(

# Remove 'on two processes' from progress rule

**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.

# Clarify progress rule of MPI_REQUEST_GET_STATUS

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

The *progress* rule for MPI_TEST, as described in Section 4.7.4, also applies to MPI_REQUEST_GET_STATUS.

# Add advice to everyone about ambiguity of logically concurrent

*Advice to users.* The MPI Forum believes the following paragraph is ambiguous and may clarify the meaning in a future version of the MPI Standard. (*End of advice to users.*)

On the other hand, if the MPI 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.

*Advice to implementors.* The MPI Forum believes the previous paragraph is ambiguous and may clarify the meaning in a future version of the MPI Standard. (*End of advice to implementors.*)

# Disambiguate "rank"



of the sender, specify the *envelope* for the message sent.

Process one (`myrank = 1`, strictly 'the MPI process with rank 1 in communicator MPI_COMM_WORLD') receives this message with the *receive* operation MPI_RECV. The message to be received is selected according to the value of its *envelope*, and the *message data* is stored into the *receive buffer*. In the example above, the receive ...ssage in the memory of process one. ...cify the location, size and type of the ... selecting the incoming message. The last parameter is used to return information on the message just received.

> *Advice to users.* Colloquial usage commonly permits references to "rank 0" or "process 0", which are strictly ambiguous and ideally should be qualified by including the relevant context, for example, the MPI communicator in the case above. (*End of advice to users.*)

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 cancelling a message, channel-like constructs and send-receive operations, ending with a description of the "dummy" MPI process, MPI_PROC_NULL.

## 4.2 Blocking Send and Receive Operations

group. See Chapter 7.)

An MPI process may have a different rank in each group in which it is a member.

When using the World Model (see Section 11.2), a predefined communicator

# Example 3.10 is "unsafe" but not erroneous

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

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

Many programmers prefer to have more leeway and opt to use the "unsafe" programming style shown in Example 4.10. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that "common practice" programs will not *deadlock*. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

*Advice to users.* If standard mode send operations are used as in Example 4.10, then a deadlock situation may occur where both MPI processes are blocked because sufficient 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 portable program using standard mode send operations should not rely on message buffering for the program to complete without *deadlock*. All sends in such a portable program can be replaced with synchronous mode sends and the program will still run correctly. The buffered send mode can be used for programs that require buffering.

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

## 4.6 Buffer Allocation and Usage

# Add MPI_Status_{set,get} for 3 status fields

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 procedures described in this section.

While the MPI_SOURCE, MPI_TAG, and MPI_ERROR status values are directly accessible by the user, for convenience in some contexts, users can also access them via procedure calls, as described below.

MPI_STATUS_GET_SOURCE(status, source)

| | | |
|---|---|---|
| IN | status | status from which to retrieve source rank (status) |
| OUT | source | rank set in the MPI_SOURCE field (integer) |

**C binding**

```
int MPI_Status_get_source(MPI_Status *status, int *sour
```

**Fortran 2008 binding**

```
MPI_Status_get_source(status, source, ierror)
    TYPE(MPI_Status), INTENT(IN) :: status
    INTEGER, INTENT(OUT) :: source
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**Fortran binding**

```
MPI_STATUS_GET_SOURCE(STATUS, SOURCE, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), SOURCE, IERROR
```

Returns in **source** the value of the MPI_SOURCE field in the **status** object.

MPI_STATUS_GET_TAG(status, tag)

| | | |
|---|---|---|
| IN | status | status from which to retrieve tag (status) |
| OUT | tag | tag set in the MPI_TAG field (integer) |

**C binding**

```
int MPI_Status_get_tag(MPI_Status *status, int *tag)
```

**Fortran 2008 binding**

```
MPI_Status_get_tag(status, tag, ierror)
    TYPE(MPI_Status), INTENT(IN) :: status
    INTEGER, INTENT(OUT) :: tag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**Fortran binding**

```
MPI_STATUS_GET_TAG(STATUS, TAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), TAG, IERROR
```

Returns in **tag** the value in the MPI_TAG field of the **status** object.

MPI_STATUS_GET_ERROR(status, err)

| | | |
|---|---|---|
| IN | status | status from which to retrieve error (status) |
| OUT | err | error set in the MPI_ERROR field (integer) |

**C binding**

```
int MPI_Status_get_error(MPI_Status *status, int *err)
```

**Fortran 2008 binding**

```
MPI_Status_get_error(status, err, ierror)
    TYPE(MPI_Status), INTENT(IN) :: status
    INTEGER, INTENT(OUT) :: err
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**Fortran binding**

```
MPI_STATUS_GET_ERROR(STATUS, ERR, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), ERR, IERROR
```

Returns in **err** the value in the MPI_ERROR field of the **status** object.
Procedures for setting these fields in a status object are defined in Section 13.3.

# Correct the MPI_TESTANY 'as if' wording

If the array of requests contains *active* handles then the execution of MPI_TESTANY has the same effect as the execution of MPI_TEST with each of the *active* handles in the array in some arbitrary order, until one call returns flag = true, or all return flag = false. In the former case, index is set to indicate which array element returned flag = true 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.

*ıctive* handles then the execution of MPI_TESTANY
MPI_TEST with each of the array elements in some
flag = true, or all fail. In the former case, index is
turned flag = true and in the latter case, it is set to
ı an array containing one *active* entry is equivalent

# Add multiple request status funcs

MPI_REQUEST_GET_STATUS_ANY(count, array_of_requests, index, flag, status)

| | | |
|---|---|---|
| IN | count | list length (non-negative integer) |
| IN | array_of_requests | array of requests (array of handles) |
| OUT | index | index of operation that completed or MPI_UNDEFINED if none completed (integer) |
| OUT | flag | true if one of the operations is complete (logical) |
| OUT | status | status object if flag is true (status) |

**C binding**

```
int MPI_Request_get_status_any(int count,
            const MPI_Request array_of_requests[], int *index, int *flag,
            MPI_Status *status)
```

**Fortran 2008 binding**

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

**Fortran binding**

```
MPI_REQUEST_GET_STATUS_ANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS,
            IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
```

Tests for *completion* of either one or none of the operations associated with *active* handles. In the former case, it returns flag = true, returns in index the index of this request in the array, and returns in status the status of that operation. (The array is indexed from zero in C, and from one in Fortran.) In the latter case (no operation *completed*), it returns flag = false, returns a value of MPI_UNDEFINED in index and status is undefined.





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

If the array of requests contains active handles then the execution of MPI_REQUEST_GET_STATUS_ANY has the same effect as the execution of MPI_REQUEST_GET_STATUS with each of the active array elements in some arbitrary order, until one call returns flag = true, or all return flag = false. In the former case, index is set to indicate which array element returned flag = true and in the latter case, it is set to MPI_UNDEFINED. MPI_REQUEST_GET_STATUS_ANY with an array containing one request is equivalent to MPI_REQUEST_GET_STATUS.

MPI_REQUEST_GET_STATUS_ALL(count, array_of_requests, flag, array_of_statuses)

| | | |
|---|---|---|
| IN | count | list length (non-negative integer) |
| IN | array_of_requests | array of requests (array of handles) |
| OUT | flag | true if all of the operations are complete (logical) |
| OUT | array_of_statuses | array of status objects (array of status) |

**C binding**

```
int MPI_Request_get_status_all(int count,
            const MPI_Request array_of_requests[], int *flag,
            MPI_Status array_of_statuses[])
```

**Fortran 2008 binding**

```
MPI_Request_get_status_all(count, array_of_requests, flag, array_of_statuses,
            ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(IN) :: array_of_requests(count)
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**Fortran binding**

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

MPI_REQUEST_GET_STATUS_ALL returns flag = true if all communication operations associated with *active* handles in the array have *completed* (this includes the case where all handles in the list are *inactive* or MPI_REQUEST_NULL). In this case, each status entry that corresponds to an *active* request is set to the status of the corresponding operation. Unlike test or wait, it does not deallocate or *inactivate* the requests; a subsequent call to test, wait or free should be executed with each of those requests.

Each status entry that corresponds to a *null* or *inactive* handle is set to *empty*. Otherwise, flag = false is returned and the values of the status entries are undefined.





The *progress* rule for MPI_TEST, as described in Section 4.7.4, also applies to MPI_REQUEST_GET_STATUS_ALL.

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

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

**C binding**

```
int MPI_Request_get_status_some(int incount,
            const MPI_Request array_of_requests[], int *outcount,
            int array_of_indices[], MPI_Status array_of_statuses[])
```

**Fortran 2008 binding**

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

**Fortran binding**

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

MPI_REQUEST_GET_STATUS_SOME returns in outcount the number of requests from the list array_of_requests that have *completed*. Returns in the first outcount locations of the array array_of_indices the indices of these operations within the array array_of_requests; the array is indexed from zero in C and from one in Fortran. Returns in the first outcount locations of the array array_of_statuses the status for these *completed* operations. However, unlike test or wait, it does not deallocate or *inactivate* any requests in array_of_requests; a subsequent call to test, wait or free should be executed with each completed request. If no operation in array_of_requests is complete, it returns outcount = 0. If all operations in array_of_requests are either MPI_REQUEST_NULL or *inactive*, outcount will be set to MPI_UNDEFINED. The *progress* rule for MPI_TEST, as described in Section 4.7.4, also applies to MPI_REQUEST_GET_STATUS_SOME.

Like MPI_WAITSOME and MPI_TESTSOME, MPI_REQUEST_GET_STATUS_SOME fulfills a **fairness requirement**: If a request for a receive repeatedly appears in a list of requests passed to MPI_REQUEST_GET_STATUS_SOME, MPI_WAITSOME, or





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

Errors that occur during the execution of MPI_REQUEST_GET_STATUS_SOME are handled as for MPI_WAITSOME.

*Advice to implementors.* MPI_REQUEST_GET_STATUS_SOME should *complete* as many pending communication operations as possible. (*End of advice to implementors.*)

*Advice to users.* MPI_REQUEST_GET_STATUS_ANY, MPI_REQUEST_GET_STATUS_SOME, and MPI_REQUEST_GET_STATUS_ALL offer tradeoffs between precision and speed, as do the corresponding TEST and WAIT functions. The ANY variants are fast, but imprecise and unfair. The ALL variants will provide all-or-nothing information and/or completion, which can limit their applicability. The SOME variants, because of their precision and fairness guarantee, will typically be the slowest on a per-call basis. (*End of advice to users.*)

# Add condition stating when MPI_WAIT is nonlocal

**Fortran binding**

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

A call to MPI_WAIT returns when the operation identified by request is *complete*. If the request is an *active persistent communication request*, it is marked *inactive*. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is in general a *nonlocal* procedure. When the operation represented by the request is *enabled* then a call to MPI_WAIT is a *local* procedure call.

The call returns, in status, information on the completed operation. The content of the status object for a receive operation can be accessed as described in Section 4.2.5. The status object for a send operation may be queried by a call to MPI_TEST_CANCELLED (see Section 4.8).

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

# Add definition of 'pending operation' in terms-2.tex

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

The implementation of MPI_MPROBE and MPI_IMPROBE needs to guarantee *progress* in the same way as in the case of MPI_PROBE and MPI_IPROBE. See also Section 2.9 on *progress*.

# Definition 'noncollective procedure' added, 'nonpersistent' substituted



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 communication request* (see Section 4.9), in the same way it is used for nonpersistent requests.



= true.

MPI_CANCEL can be used to *cancel* a communication that uses a *persistent communication request* (see Section 4.9), in the same way as it is described above for nonblocking operations. *Cancelling* a persistent 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

# 'Pending communication' not defined

**Create (Start Complete)$^*$ Free**

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

*Inactive persistent requests* are not automatically *freed* when the associated communicator is disconnected (via MPI_COMM_DISCONNECT, see 11.10.4) or the associated World Model or Sessions Model is finalized (via MPI_FNIALIZE, see 11.2.2, or MPI_SESSION_FINALIZE, see 11.3.1). In these situations, any further use of the request handle is erroneous. In particular, freeing associated inactive request handles after such a communicator disconnect or finalization is then impossible.

*Advice to users.* Persistent request handles may bind internal resources such as MPI buffers in shared memory for providing efficient communication. Therefore, it is highly recommended to explicitly free inactive request handles, using MPI_REQUEST_FREE, when they are no longer in use, and in particular before freeing or disconnecting the associated communicator with MPI_COMM_FREE or MPI_COMM_DISCONNECT or finalizing the associated session with MPI_SESSION_FINALIZE. (*End of advice to users.*)

A send operation *started* with MPI_START can be *matched* with any receive operation and, likewise, a receive operation *started* with MPI_START can receive messages generated by any send operation.

# Add new Flush and Iflush procedures to new buffered mode send buffer handling

MPI_COMM_FLUSH_BUFFER(comm)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |

**C binding**

```
int MPI_Comm_flush_buffer(MPI_Comm comm)
```





**Fortran 2008 binding**

```
MPI_Comm_flush_buffer(comm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**Fortran binding**

```
MPI_COMM_FLUSH_BUFFER(COMM, IERROR)
    INTEGER COMM, IERROR
```

MPI_COMM_FLUSH_BUFFER will not return until all messages currently in the communicator-specific buffer of the calling MPI process have been transmitted.

MPI_SESSION_FLUSH_BUFFER(session)

| | | |
|---|---|---|
| IN | session | session (handle) |

**C binding**

```
int MPI_Session_flush_buffer(MPI_Session session)
```

**Fortran 2008 binding**

```
MPI_Session_flush_buffer(session, ierror)
    TYPE(MPI_Session), INTENT(IN) :: session
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**Fortran binding**

```
MPI_SESSION_FLUSH_BUFFER(SESSION, IERROR)
    INTEGER SESSION, IERROR
```

MPI_SESSION_FLUSH_BUFFER will not return until all messages currently in the session-specific buffer of the calling MPI process have been transmitted.

MPI_BUFFER_FLUSH()

**C binding**

```
int MPI_Buffer_flush(void)
```

**Fortran 2008 binding**

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

**Fortran binding**

```
MPI_BUFFER_FLUSH(IERROR)
    INTEGER IERROR
```

MPI_BUFFER_FLUSH will not return until all messages currently in the MPI process-specific buffer of the calling MPI process have been transmitted.

For all MPI_XXX_FLUSH_BUFFER procedures, there also exist the following nonblocking variants, which start the respective flush operation. These operations will not complete until all messages currently in the respective buffer of the calling MPI process have been transmitted.

MPI_COMM_IFLUSH_BUFFER(comm, request)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Comm_iflush_buffer(MPI_Comm comm, MPI_Request *request)
```

**Fortran 2008 binding**

```
MPI_Comm_iflush_buffer(comm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**Fortran binding**

```
MPI_COMM_IFLUSH_BUFFER(COMM, REQUEST, IERROR)
    INTEGER COMM, REQUEST, IERROR
```

MPI_SESSION_IFLUSH_BUFFER(session, request)

| | | |
|---|---|---|
| IN | session | session (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Session_iflush_buffer(MPI_Session session, MPI_Request *request)
```

**Fortran 2008 binding**

```
MPI_Session_iflush_buffer(session, request, ierror)
    TYPE(MPI_Session), INTENT(IN) :: session
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**Fortran binding**

```
MPI_SESSION_IFLUSH_BUFFER(SESSION, REQUEST, IERROR)
    INTEGER SESSION, REQUEST, IERROR
```

MPI_BUFFER_IFLUSH(request)

| | | |
|---|---|---|
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Buffer_iflush(MPI_Request *request)
```

**Fortran 2008 binding**

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

**Fortran binding**

```
MPI_BUFFER_IFLUSH(REQUEST, IERROR)
```





```
    INTEGER REQUEST, IERROR
```

# Add new buffer attach/detach procedures scoped by communicator and by session

### 4.6 Buffer Allocation and Usage

A user may specify up to one buffer per communicator, up to one buffer per session, and up to one buffer per MPI process to be used for buffering messages sent in buffered mode. Buffering is done by the sender.

MPI_COMM_ATTACH_BUFFER(comm, buffer, size)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| IN | buffer | initial buffer address (choice) |
| IN | size | buffer size, in bytes (non-negative integer) |

**C binding**
```
int MPI_Comm_attach_buffer(MPI_Comm comm, void *buffer, int size)
int MPI_Comm_attach_buffer_c(MPI_Comm comm, void *buffer, MPI_Count size)
```
**Fortran 2008 binding**
```
MPI_Comm_attach_buffer(comm, buffer, size, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: size
```





```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Comm_attach_buffer(comm, buffer, size, ierror) !(_c)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```
**Fortran binding**
```
MPI_COMM_ATTACH_BUFFER(COMM, BUFFER, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
    <type> BUFFER(*)
```

Provides to MPI a communicator-specific buffer in memory. This is to be used for buffering outgoing messages sent when a buffered mode send is started that uses the communicator comm.

If MPI_BUFFER_AUTOMATIC is passed as the argument buffer, no explicit buffer is attached; rather, automatic buffering is enabled for all buffered mode communication associated with the communicator comm (see Section 4.6). Further, if MPI_BUFFER_AUTOMATIC is passed as the argument buffer, the value of size is irrelevant. Note that in Fortran MPI_BUFFER_AUTOMATIC is an object like MPI_BOTTOM (not usable for initialization or assignment), see Section 2.5.4.

MPI_SESSION_ATTACH_BUFFER(session, buffer, size)

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

**C binding**
```
int MPI_Session_attach_buffer(MPI_Session session, void *buffer, int size)
int MPI_Session_attach_buffer_c(MPI_Session session, void *buffer,
            MPI_Count size)
```
**Fortran 2008 binding**
```
MPI_Session_attach_buffer(session, buffer, size, ierror)
    TYPE(MPI_Session), INTENT(IN) :: session
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Session_attach_buffer(session, buffer, size, ierror) !(_c)
    TYPE(MPI_Session), INTENT(IN) :: session
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```
**Fortran binding**
```
MPI_SESSION_ATTACH_BUFFER(SESSION, BUFFER, SIZE, IERROR)
    INTEGER SESSION, SIZE, IERROR
```



```
    <type> BUFFER(*)
```

Provides to MPI a session-specific buffer in memory. This buffer is to be used for buffering outgoing messages sent when using a communicator that is created from a group that is derived from the session session. However, if there is a communicator-specific buffer attached to the particular communicator at the time of the buffered mode send is started, that buffer is used.

If MPI_BUFFER_AUTOMATIC is passed as the argument buffer, no explicit buffer is attached; rather, automatic buffering is enabled for all buffered mode communication associated with the session session that is not explicitly covered by a buffer provided at communicator level (see Section 4.6). Further, if MPI_BUFFER_AUTOMATIC is passed as the argument buffer, the value of size is irrelevant. Note that in Fortran MPI_BUFFER_AUTOMATIC is an object like MPI_BOTTOM (not usable for initialization or assignment), see Section 2.5.4.

MPI_BUFFER_ATTACH(buffer, size)

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

**C binding**
```
int MPI_Buffer_attach(void *buffer, int size)
int MPI_Buffer_attach_c(void *buffer, MPI_Count size)
```
**Fortran 2008 binding**
```
MPI_Buffer_attach(buffer, size, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Buffer_attach(buffer, size, ierror) !(_c)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```
**Fortran binding**
```
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR
```

Provides to MPI an MPI process-specific buffer in memory. This buffer is to be used for buffering outgoing messages sent when using a communicator to which no communicator-specific buffer is attached or which is derived from a session to which no session-specific buffer is attached at the time the buffered mode send is started.

If MPI_BUFFER_AUTOMATIC is passed as the argument buffer, no explicit buffer is attached; rather, automatic buffering is enabled for all buffered mode communication not explicitly covered by a buffer provided at session or communicator level (see Section 4.6). Further, if MPI_BUFFER_AUTOMATIC is passed as the argument buffer, the value of size is irrelevant. Note that in Fortran MPI_BUFFER_AUTOMATIC is an object like MPI_BOTTOM (not usable for initialization or assignment), see Section 2.5.4.





*Advice to users.* The use of a global shared buffer can be problematic when used for communication in different libraries, as the buffer represents a shared resource used for all buffered mode communication. Further, with the introduction of the Sessions Model, the use of a single shared buffer violates the concept of resource isolation that is intended with MPI Sessions. It is therefore recommended, especially for libraries and programs using the Sessions Model, to use only communicator-specific or session-specific buffers. (*End of advice to users.*)

Any of these buffers are used only for messages sent in buffered mode. Only one MPI process-specific buffer can be attached to an MPI process at a time, only one session-specific buffer can be attached to a session at a time and only one communicator-specific buffer can be attached to a communicator at a time.

If automatic buffering is enabled at any level, no other buffer can be attached at that level.

A particular memory region can only be used in one buffer; reusing buffer space for multiple sessions, communicators and/or the global buffer is erroneous. Further, only one buffer is used for any one communication following the rules above; buffer space is not combined, even if two buffers are directly or indirectly provided to a communicator to be used for buffered sends.

In C, buffer is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 19.1.12).

MPI_COMM_DETACH_BUFFER(comm, buffer_addr, size)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | buffer_addr | initial buffer address (choice) |
| OUT | size | buffer size, in bytes (integer) |

**C binding**
```
int MPI_Comm_detach_buffer(MPI_Comm comm, void *buffer_addr, int *size)
int MPI_Comm_detach_buffer_c(MPI_Comm comm, void *buffer_addr, MPI_Count *size)
```
**Fortran 2008 binding**
```
MPI_Comm_detach_buffer(comm, buffer_addr, size, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(C_PTR), INTENT(OUT) :: buffer_addr
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Comm_detach_buffer(comm, buffer_addr, size, ierror) !(_c)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(C_PTR), INTENT(OUT) :: buffer_addr
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```





**Fortran binding**
```
MPI_COMM_DETACH_BUFFER(COMM, BUFFER_ADDR, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
    <type> BUFFER_ADDR(*)
```

Detach the communicator-specific buffer currently attached to the communicator.

MPI_SESSION_DETACH_BUFFER(session, buffer_addr, size)

| | | |
|---|---|---|
| IN | session | session (handle) |
| OUT | buffer_addr | initial buffer address (choice) |
| OUT | size | buffer size, in bytes (integer) |

**C binding**
```
int MPI_Session_detach_buffer(MPI_Session session, void *buffer_addr,
            int *size)
int MPI_Session_detach_buffer_c(MPI_Session session, void *buffer_addr,
            MPI_Count *size)
```
**Fortran 2008 binding**
```
MPI_Session_detach_buffer(session, buffer_addr, size, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    TYPE(MPI_Session), INTENT(IN) :: session
    TYPE(C_PTR), INTENT(OUT) :: buffer_addr
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Session_detach_buffer(session, buffer_addr, size, ierror) !(_c)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    TYPE(MPI_Session), INTENT(IN) :: session
    TYPE(C_PTR), INTENT(OUT) :: buffer_addr
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```
**Fortran binding**
```
MPI_SESSION_DETACH_BUFFER(SESSION, BUFFER_ADDR, SIZE, IERROR)
    INTEGER SESSION, SIZE, IERROR
    <type> BUFFER_ADDR(*)
```

Detach the session-specific buffer currently attached to the session.

MPI_BUFFER_DETACH(buffer_addr, size)

| | | |
|---|---|---|
| OUT | buffer_addr | initial buffer address (choice) |
| OUT | size | buffer size, in bytes (integer) |

**C binding**
```
int MPI_Buffer_detach(void *buffer_addr, int *size)
```