

*D R A F T*

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

March 6, 2012

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 12

## External Interfaces

### 12.1 Introduction

This chapter begins with calls used to create **generalized requests**, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. This can be used to layer new functionality on top of MPI. Next, Section 12.3 deals with setting the information found in `status`. [This is] This functionality is needed for generalized requests.

The chapter continues, in Section 12.4, with a discussion of how threads are to be handled in MPI. Although thread compliance is not required, the standard specifies how threads are to work if they are provided.

### 12.2 Generalized Requests

The goal of generalized requests is to allow users to define new nonblocking operations. Such an outstanding nonblocking operation is represented by a (generalized) request. A fundamental property of nonblocking operations is that progress toward the completion of this operation occurs asynchronously, i.e., concurrently with normal program execution. Typically, this requires execution of code concurrently with the execution of the user code, e.g., in a separate thread or in a signal handler. Operating systems provide a variety of mechanisms in support of concurrent execution. MPI does not attempt to standardize or replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as `MPI_WAIT` or `MPI_CANCEL` when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

*Rationale.* It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is very difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the application. For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI when the operation completes. This is done by making a call to `MPI_GREQUEST_COMPLETE`. MPI maintains the “completion” status of generalized requests. Any other request state has to be maintained by the user.

A new generalized request is started with

```

MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)

    IN      query_fn      callback function invoked when request status is queried
                                (function)
    IN      free_fn       callback function invoked when request is freed (function)
    IN      cancel_fn     callback function invoked when request is cancelled
                                (function)
    IN      extra_state   extra state
    OUT     request       generalized request (handle)

int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
                      MPI_Grequest_free_function *free_fn,
                      MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
                      MPI_Request *request)

MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
                  IERROR)
    INTEGER REQUEST, IERROR
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE

{static MPI::Grequest
    MPI::Grequest::Start(const MPI::Grequest::Query_function*
                        query_fn, const MPI::Grequest::Free_function* free_fn,
                        const MPI::Grequest::Cancel_function* cancel_fn,
                        void *extra_state) (binding deprecated, see Section ??) }

```

*Advice to users.* Note that a generalized request belongs, in C++, to the class `MPI::Grequest`, which is a derived class of `MPI::Request`. It is of the same type as regular requests, in C and Fortran. (*End of advice to users.*)

The call starts a generalized request and returns a handle to it in `request`.

The syntax and meaning of the callback functions are listed below. All callback functions are passed the `extra_state` argument that was associated with the request by the starting call `MPI_GREQUEST_START`[. [This can](#)]; `extra_state` [can](#) be used to maintain user-defined state for the request.

In C, the query function is

```
typedef int MPI_Grequest_query_function(void *extra_state,
                                         MPI_Status *status);
```

in Fortran

```
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
  INTEGER STATUS(MPI_STATUS_SIZE), IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

and in C++

```
{typedef int MPI::Grequest::Query_function(void* extra_state,
                                             MPI::Status& status); (binding deprecated, see Section ??)}
```

**[query\_fn]** The **query\_fn** function computes the status that should be returned for the generalized request. The status also includes information about successful/unsuccessful cancellation of the request (result to be returned by `MPI_TEST_CANCELLED`).

**[query\_fn]** The **query\_fn** callback is invoked by the `MPI_{WAIT|TEST}{ANY|SOME|ALL}` call that completed the generalized request associated with this callback. The callback function is also invoked by calls to `MPI_REQUEST_GET_STATUS`, if the request is complete when the call occurs. In both cases, the callback is passed a reference to the corresponding status variable passed by the user to the MPI call; the status set by the callback function is returned by the MPI call. If the user provided `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE` to the MPI function that causes **query\_fn** to be called, then MPI will pass a valid status object to **query\_fn**, and this status will be ignored upon return of the callback function. Note that **query\_fn** is invoked only after `MPI_GREQUEST_COMPLETE` is called on the request; it may be invoked several times for the same generalized request, e.g., if the user calls `MPI_REQUEST_GET_STATUS` several times for this request. Note also that a call to `MPI_{WAIT|TEST}{SOME|ALL}` may cause multiple invocations of **query\_fn** callback functions, one for each generalized request that is completed by the MPI call. The order of these invocations is not specified by MPI.

In C, the free function is

```
typedef int MPI_Grequest_free_function(void *extra_state);
```

and in Fortran

```
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
  INTEGER IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

and in C++

```
{typedef int MPI::Grequest::Free_function(void* extra_state); (binding deprecated, see Section ??)}
```

**[free\_fn]** The **free\_fn** function is invoked to clean up user-allocated resources when the generalized request is freed.

**[free\_fn]** The **free\_fn** callback is invoked by the `MPI_{WAIT|TEST}{ANY|SOME|ALL}` call that completed the generalized request associated with this callback. **free\_fn** is invoked after the call to **query\_fn** for the same request. However, if the MPI call completed multiple generalized requests, the order in which **free\_fn** callback functions are invoked is not specified by MPI.

[free\_fn]The **free\_fn** callback is also invoked for generalized requests that are freed by a call to `MPI_REQUEST_FREE` (no call to `WAIT_{WAIT|TEST}{ANY|SOME|ALL}` will occur for such a request). In this case, the callback function will be called either in the MPI call `MPI_REQUEST_FREE(request)`, or in the MPI call `MPI_GREQUEST_COMPLETE(request)`, whichever happens last, i.e., in this case the actual freeing code is executed as soon as both calls `MPI_REQUEST_FREE` and `MPI_GREQUEST_COMPLETE` have occurred. The request is not deallocated until after `free_fn` completes. Note that `free_fn` will be invoked only once per request by a correct program.

*Advice to users.* Calling `MPI_REQUEST_FREE(request)` will cause the request handle to be set to `MPI_REQUEST_NULL`. This handle to the generalized request is no longer valid. However, user copies of this handle are valid until after `free_fn` completes since MPI does not deallocate the object until then. Since `free_fn` is not called until after `MPI_GREQUEST_COMPLETE`, the user copy of the handle can be used to make this call. Users should note that MPI will deallocate the object after `free_fn` executes. At this point, user copies of the request handle no longer point to a valid request. MPI will not set user copies to `MPI_REQUEST_NULL` in this case, so it is up to the user to avoid accessing this stale handle. This is a special case [where]in which MPI defers deallocating the object until a later time that is known by the user. (*End of advice to users.*)

In C, the cancel function is

```
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
```

in Fortran

```
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
  INTEGER IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
  LOGICAL COMPLETE
```

and in C++

```
{typedef int MPI::Grequest::Cancel_function(void* extra_state,
      bool complete); (binding deprecated, see Section ??)}
```

[cancel\_fn]The **cancel\_fn** function is invoked to start the cancelation of a generalized request. It is called by `MPI_CANCEL(request)`. MPI passes [to the callback function `complete=true`]complete=true to the callback function if `MPI_GREQUEST_COMPLETE` was already called on the request, and `complete=false` otherwise.

All callback functions return an error code. The code is passed back and dealt with as appropriate for the error code by the MPI function that invoked the callback function. For example, if error codes are returned then the error code returned by the callback function will be returned by the MPI function that invoked the callback function. In the case of an `MPI_{WAIT|TEST}{ANY}` call that invokes both `query_fn` and `free_fn`, the MPI call will return the error code returned by the last callback, namely `free_fn`. If one or more of the requests in a call to `MPI_{WAIT|TEST}{SOME|ALL}` failed, then the MPI call will return `MPI_ERR_IN_STATUS`. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its `free_fn` callback function.

However, if the MPI function was passed MPI\_STATUSES\_IGNORE, then the individual error codes returned by each callback functions will be lost.

*Advice to users.* query\_fn must **not** set the error field of status since query\_fn may be called by MPI\_WAIT or MPI\_TEST, in which case the error field of status should not change. The MPI library knows the “context” in which query\_fn is invoked and can decide correctly when to put in the error field of status the returned error code. (End of advice to users.)

MPI\_GREQUEST\_COMPLETE(request)

INOUT request generalized request (handle)

int MPI\_Grequest\_complete(MPI\_Request request)

MPI\_GREQUEST\_COMPLETE(REQUEST, IERROR)

INTEGER REQUEST, IERROR

{void MPI::Grequest::Complete() (binding deprecated, see Section ??) }

The call informs MPI that the operations represented by the generalized request request are complete (see definitions in Section ??). A call to MPI\_WAIT(request, status) will return and a call to MPI\_TEST(request, flag, status) will return flag=true only after a call to MPI\_GREQUEST\_COMPLETE has declared that these operations are complete.

MPI imposes no restrictions on the code executed by the callback functions. However, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as MPI\_TEST, MPI\_REQUEST\_FREE, or MPI\_CANCEL still hold. For example, all these calls are supposed to be local and nonblocking. Therefore, the callback functions query\_fn, free\_fn, or cancel\_fn should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once MPI\_CANCEL is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired “local” semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

*Advice to implementors.* A call to MPI\_GREQUEST\_COMPLETE may unblock a blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (End of advice to implementors.)

### 12.2.1 Examples

**Example 12.1** This example shows the code for a user-defined reduce operation on an int using a binary tree: each non-root node receives two messages, sums them, and sends them up. We assume that no status is returned and that the operation cannot be cancelled.

```

1  typedef struct {
2      MPI_Comm comm;
3      int tag;
4      int root;
5      int valin;
6      int *valout;
7      MPI_Request request;
8      } ARGS;
9
10
11 int myreduce(MPI_Comm comm, int tag, int root,
12             int valin, int *valout, MPI_Request *request)
13 {
14     ARGS *args;
15     pthread_t thread;
16
17     /* start request */
18     MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);
19
20     args = (ARGS*)malloc(sizeof(ARGS));
21     args->comm = comm;
22     args->tag = tag;
23     args->root = root;
24     args->valin = valin;
25     args->valout = valout;
26     args->request = *request;
27
28     /* spawn thread to handle request */
29     /* The availability of the pthread_create call is system dependent */
30     pthread_create(&thread, NULL, reduce_thread, args);
31
32     return MPI_SUCCESS;
33 }
34
35 /* thread code */
36 void* reduce_thread(void *ptr)
37 {
38     int lchild, rchild, parent, lval, rval, val;
39     MPI_Request req[2];
40     ARGS *args;
41
42     args = (ARGS*)ptr;
43
44     /* compute left,right child and parent in tree; set
45        to MPI_PROC_NULL if does not exist */
46     /* code not shown */
47     ...
48

```



```

MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]); 1
MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]); 2
MPI_Waitall(2, req, MPI_STATUSES_IGNORE); 3
val = lval + args->valin + rval; 4
MPI_Send( &val, 1, MPI_INT, parent, args->tag, args->comm ); 5
if (parent == MPI_PROC_NULL) *(args->valout) = val; 6
MPI_Grequest_complete((args->request)); 7
free(ptr); 8
return(NULL); 9
} 10
11
int query_fn(void *extra_state, MPI_Status *status) 12
{ 13
    /* always send just one int */ 14
    MPI_Status_set_elements(status, MPI_INT, 1); 15
    /* can never cancel so always true */ 16
    MPI_Status_set_cancelled(status, 0); 17
    /* choose not to return a value for this */ 18
    status->MPI_SOURCE = MPI_UNDEFINED; 19
    /* tag has no meaning for this generalized request */ 20
    status->MPI_TAG = MPI_UNDEFINED; 21
    /* this generalized request never fails */ 22
    return MPI_SUCCESS; 23
} 24
25
26
int free_fn(void *extra_state) 27
{ 28
    /* this generalized request does not need to do any freeing */ 29
    /* as a result it never fails here */ 30
    return MPI_SUCCESS; 31
} 32
33
34
int cancel_fn(void *extra_state, int complete) 35
{ 36
    /* This generalized request does not support cancelling. 37
       Abort if not already done. If done then treat as if cancel failed.*/ 38
    if (!complete) { 39
        fprintf(stderr, 40
            "Cannot cancel generalized request - aborting program\n"); 41
        MPI_Abort(MPI_COMM_WORLD, 99); 42
    } 43
    return MPI_SUCCESS; 44
} 45
46
47
48

```

## 12.3 Associating Information with Status

MPI supports several different types of requests besides those for point-to-point operations. These range from MPI calls for I/O to generalized requests. It is desirable to allow these calls *[use]to use* the same request *[mechanism. This]mechanism, which* allows one to wait or test on different types of requests. However, `MPI_{TEST|WAIT}{ANY|SOME|ALL}` returns a status with information about the request. With the generalization of requests, one needs to define what information will be returned in the status object.

Each MPI call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to `MPI_{TEST|WAIT}{ANY|SOME|ALL}` can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful *[value]values* for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, these calls are provided:

`MPI_STATUS_SET_ELEMENTS(status, datatype, count)`

INOUT	status	status with which to associate count (Status)
IN	datatype	datatype associated with count (handle)
IN	count	number of elements to associate with status (integer)

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

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

```
{void MPI::Status::Set_elements(const MPI::Datatype& datatype, int
                                count) (binding deprecated, see Section ??) }
```

`MPI_STATUS_SET_ELEMENTS_X(status, datatype, count)`

INOUT	status	status with which to associate count (Status)
IN	datatype	datatype associated with count (handle)
IN	count	number of elements to associate with status (integer)

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

```
MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
    INTEGER (KIND=MPI_COUNT_KIND) COUNT
```

[This call modifies] These functions modify the opaque part of status so that a call to MPI\_GET\_ELEMENTS or MPI\_GET\_ELEMENTS\_X will return count. MPI\_GET\_COUNT will return a compatible value.

*Rationale.* The number of elements is set instead of the count because the former can deal with a nonintegral number of datatypes. (*End of rationale.*)

A subsequent call to MPI\_GET\_COUNT(status, datatype, count) [ or to ], MPI\_GET\_ELEMENTS(status, datatype, count) , or MPI\_GET\_ELEMENTS\_X(status, datatype, count) must use a datatype argument that has the same type signature as the datatype argument that was used in the call to MPI\_STATUS\_SET\_ELEMENTS or MPI\_STATUS\_SET\_ELEMENTS\_X.

*Rationale.* [This] The requirement of matching type signatures for these calls is similar to the restriction that holds when count is set by a receive operation: in that case, the calls to MPI\_GET\_COUNT [ and ], MPI\_GET\_ELEMENTS, and MPI\_GET\_ELEMENTS\_X must use a datatype with the same signature as the datatype used in the receive call. (*End of rationale.*)

MPI\_STATUS\_SET\_CANCELLED(status, flag)

INOUT	status	status with which to associate cancel flag (Status)
IN	flag	if true indicates request was cancelled (logical)

int MPI\_Status\_set\_cancelled(MPI\_Status \*status, int flag)

MPI\_STATUS\_SET\_CANCELLED(STATUS, FLAG, IERROR)

INTEGER STATUS(MPI\_STATUS\_SIZE), IERROR

LOGICAL FLAG

{void MPI::Status::Set\_cancelled(bool flag) (*binding deprecated, see Section ??*) }

If flag is set to true then a subsequent call to MPI\_TEST\_CANCELLED(status, flag) will also return flag = true, otherwise it will return false.

*Advice to users.* Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI\_GET\_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra\_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI\_RECV, may lead to unpredictable results and is strongly discouraged. (*End of advice to users.*)

## 12.4 MPI and Threads

This section specifies the interaction between MPI calls and threads. The section lists minimal requirements for **thread compliant** MPI implementations and defines functions

that can be used for initializing the thread environment. MPI may be implemented in environments where threads are not supported or perform poorly. Therefore, it is not required that all MPI implementations fulfill all the requirements specified in this section.

This section generally assumes a thread package similar to POSIX threads [?], but the syntax and semantics of thread calls are not specified here — these are beyond the scope of this document.

### 12.4.1 General

In a thread-compliant implementation, an MPI process is a process that may be multi-threaded. Each thread can issue MPI calls; however, threads are not separately addressable: a rank in a send or receive call identifies a process, not a thread. A message sent to a process can be received by any thread in this process.

*Rationale.* This model corresponds to the POSIX model of interprocess communication: the fact that a process is multi-threaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations [where]in which MPI ‘processes’ are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their “processes” are single-threaded). (*End of rationale.*)

*Advice to users.* It is the user’s responsibility to prevent races when threads within the same application post conflicting communication calls. The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. (*End of advice to users.*)

The two main requirements for a thread-compliant implementation are listed below.

1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

**Example 12.2** Process 0 consists of two threads. The first thread executes a blocking send call `MPI_Send(buff1, count, type, 0, 0, comm)`, whereas the second thread executes a blocking receive call `MPI_Recv(buff2, count, type, 0, 0, comm, &status)`, i.e., the first thread sends a message that is received by the second thread. This communication should always succeed. According to the first requirement, the execution will correspond to some interleaving of the two calls. According to the second requirement, a call can only block the calling thread and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receive call will occur. Once both calls occur, the communication is enabled and both calls will complete. On the other hand, a single-threaded process that posts a send, followed by a matching receive, may deadlock.

The progress requirement for multithreaded implementations is stronger, as a blocked call cannot prevent progress in other threads.

*Advice to implementors.* MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process-global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is released in between. Finer locks can provide more concurrency, at the expense of higher locking overheads. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads. (*End of advice to implementors.*)

## 12.4.2 Clarifications

**Initialization and Completion** The call to MPI\_FINALIZE should occur on the same thread that initialized MPI. We call this thread the **main thread**. The call should occur only after all the process threads have completed their MPI calls, and have no pending communications or I/O operations.

*Rationale.* This constraint simplifies implementation. (*End of rationale.*)

**Multiple threads completing the same request.** A program where two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent MPI\_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a request can only be completed once. Any combination of wait or test [which]that violates this rule is erroneous.

*Rationale.* [This]This restriction is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is possible to post a second wait on the same handle. With threads, an MPI\_WAIT{ANY|SOME|ALL} may be blocked without having nullified its request(s) so it becomes the user's responsibility to avoid using the same request in an MPI\_WAIT on another thread. This constraint also simplifies implementation, as only one thread will be blocked on any communication or I/O event. (*End of rationale.*)

**Probe** A receive call that uses source and tag values returned by a preceding call to MPI\_PROBE or MPI\_IProbe will receive the message matched by the probe call only if there was no other matching receive after the probe and before that receive. In a multithreaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process.

**Collective calls** Matching of collective calls on a communicator, window, or file handle is done according to the order in which the calls are issued at each process. If concurrent threads issue such calls on the same communicator, window or file handle, it is up to the user to make sure the calls are correctly ordered, using interthread synchronization.

*Advice to users.* With three concurrent threads in each MPI process of a communicator `comm`, it is allowed that thread A in each MPI process calls a collective operation on `comm`, thread B calls a file operation on an existing filehandle that was formerly opened on `comm`, and thread C invokes one-sided operations on an existing window handle that was also formerly created on `comm`. (*End of advice to users.*)

*Rationale.* As already specified in `MPI_FILE_OPEN` and `MPI_WIN_CREATE`, a file handle and a window handle inherit only the group of processes of the underlying communicator, but not the communicator itself. Accesses to communicators, window handles and file handles cannot affect one another. (*End of rationale.*)

*Advice to implementors.* [Advice to implementors.] If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (*End of advice to implementors.*)

**Exception handlers** An exception handler does not necessarily execute in the context of the thread that made the exception-raising MPI call; the exception handler may be executed by a thread that is distinct from the thread that will return the error code.

*Rationale.* The MPI implementation may be multithreaded, so that part of the communication protocol may execute on a thread that is distinct from the thread that made the MPI call. The design allows the exception handler to be executed on the thread where the exception occurred. (*End of rationale.*)

**Interaction with signals and cancellations** The outcome is undefined if a thread that executes an MPI call is cancelled (by another thread), or if a thread catches a signal while executing an MPI call. However, a thread of an MPI process may terminate, and may catch signals or be cancelled by another thread when not executing MPI calls.

*Rationale.* Few C library functions are signal safe, and many have cancellation points — points [where]at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be “async-cancel-safe” or [“async-signal-safe.”] “async-signal-safe”). (*End of rationale.*)

*Advice to users.* Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to `sigwait` for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

*Advice to implementors.* The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (*End of advice to implementors.*)

### 12.4.3 Initialization

The following function may be used to initialize MPI, and initialize the MPI thread environment, instead of `MPI_INIT`.

MPI\_INIT\_THREAD(required, provided)

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)

```
int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
                   int *provided)
```

```
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
INTEGER REQUIRED, PROVIDED, IERROR
```

```
{int MPI::Init_thread(int& argc, char**& argv, int required) (binding
                        deprecated, see Section ??) }
```

```
{int MPI::Init_thread(int required) (binding deprecated, see Section ??) }
```

*Advice to users.* In C and C++, the passing of `argc` and `argv` is [optional.]optional, as with `MPI_INIT` as discussed in Section ?? . In C, [this is accomplished by passing the appropriate null pointer.] null pointers may be passed in their place. In C++, [this is accomplished with two separate bindings to cover these two cases. This is as with `MPI_INIT` as discussed in Section ??.]two separate bindings support this choice. (End of advice to users.)

This call initializes MPI in the same way that a call to `MPI_INIT` would. In addition, it initializes the thread environment. The argument `required` is used to specify the desired level of thread support. The possible values are listed in increasing order of thread support.

**MPI\_THREAD\_SINGLE** Only one thread will execute.

**MPI\_THREAD\_FUNNELED** The process may be multi-threaded, but the application must ensure that only the main thread makes MPI calls (for the definition of main thread, see `MPI_IS_THREAD_MAIN` on page 15).

**MPI\_THREAD\_SERIALIZED** The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are “serialized”).

**MPI\_THREAD\_MULTIPLE** Multiple threads may call MPI, with no restrictions.

These values are monotonic; i.e., `MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE`.

Different processes in `MPI_COMM_WORLD` may require different levels of thread support.

The call returns in `provided` information about the actual level of thread support that will be provided by MPI. It can be one of the four values listed above.

The level(s) of thread support that can be provided by `MPI_INIT_THREAD` will depend on the implementation, and may depend on information provided by the user before the program started to execute (e.g., with arguments to `mpiexec`). If possible, the call will return `provided = required`. Failing this, the call will return the least supported level such that `provided > required` (thus providing a stronger level of support than required by the

user). Finally, if the user requirement cannot be satisfied, then the call will return in provided the highest supported level.

A **thread compliant** MPI implementation will be able to return provided = MPI\_THREAD\_MULTIPLE. Such an implementation may always return provided = MPI\_THREAD\_MULTIPLE, irrespective of the value of required. At the other extreme, an MPI library that is not thread compliant may always return provided = MPI\_THREAD\_SINGLE, irrespective of the value of required.

A call to MPI\_INIT has the same effect as a call to MPI\_INIT\_THREAD with a required = MPI\_THREAD\_SINGLE.

Vendors may provide (implementation dependent) means to specify the level(s) of thread support available when the MPI program is started, e.g., with arguments to `mpiexec`. This will affect the outcome of calls to MPI\_INIT and MPI\_INIT\_THREAD. Suppose, for example, that an MPI program has been started so that only MPI\_THREAD\_MULTIPLE is available. Then MPI\_INIT\_THREAD will return provided = MPI\_THREAD\_MULTIPLE, irrespective of the value of required; a call to MPI\_INIT will also initialize the MPI thread support level to MPI\_THREAD\_MULTIPLE. Suppose, on the other hand, that an MPI program has been started so that all four levels of thread support are available. Then, a call to MPI\_INIT\_THREAD will return provided = required; on the other hand, a call to MPI\_INIT will initialize the MPI thread support level to MPI\_THREAD\_SINGLE.

*Rationale.* Various optimizations are possible when MPI code is executed single-threaded, or is executed on multiple threads, but not concurrently: mutual exclusion code may be omitted. Furthermore, if only one thread executes, then the MPI library can use library functions that are not thread safe, without risking conflicts with user threads. Also, the model of one communication thread, multiple computation threads fits many applications well, e.g., if the process code is a sequential Fortran/C/C++ program with MPI calls that has been parallelized by a compiler for execution on an SMP node, in a cluster of SMPs, then the process computation is multi-threaded, but MPI calls will likely execute on a single thread.

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multi-threaded MPI codes. (*End of rationale.*)

*Advice to implementors.* If provided is not MPI\_THREAD\_SINGLE then the MPI library should not invoke C/ C++/Fortran library calls that are not thread safe, e.g., in an environment where `malloc` is not thread safe, then `malloc` should not be used by the MPI library.

Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI\_INIT\_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time. (*End of advice to implementors.*)

The following function can be used to query the current level of thread support.



**MPI\_QUERY\_THREAD(provided)**

OUT      provided      provided level of thread support (integer)

int MPI\_Query\_thread(int \*provided)

MPI\_QUERY\_THREAD(PROVIDED, IERROR)

INTEGER PROVIDED, IERROR

{int MPI::Query\_thread() (*binding deprecated, see Section ??*) }

The call returns in provided the current level of thread [support. This]support, which will be the value returned in provided by MPI\_INIT\_THREAD, if MPI was initialized by a call to MPI\_INIT\_THREAD().

**MPI\_IS\_THREAD\_MAIN(flag)**

OUT      flag      true if calling thread is main thread, false otherwise  
(logical)

int MPI\_Is\_thread\_main(int \*flag)

MPI\_IS\_THREAD\_MAIN(FLAG, IERROR)

LOGICAL FLAG

INTEGER IERROR

{bool MPI::Is\_thread\_main() (*binding deprecated, see Section ??*) }

This function can be called by a thread to [find out whether]determine if it is the main thread (the thread that called MPI\_INIT or MPI\_INIT\_THREAD).

All routines listed in this section must be supported by all MPI implementations.

*Rationale.* MPI libraries are required to provide these calls even if they do not support threads, so that portable code that contains invocations to these functions [be able to]can link correctly. MPI\_INIT continues to be supported so as to provide compatibility with current MPI codes. (*End of rationale.*)

*Advice to users.* It is possible to spawn threads before MPI is initialized, but no MPI call other than MPI\_INITIALIZED should be executed by these threads, until MPI\_INIT\_THREAD is invoked by one thread (which, thereby, becomes the main thread). In particular, it is possible to enter the MPI execution with a multi-threaded process.

The level of thread support provided is a global property of the MPI process that can be specified only once, when MPI is initialized on that process (or before). Portable third party libraries have to be written so as to accommodate any provided level of thread support. Otherwise, their usage will be restricted to specific level(s) of thread support. If such a library can run only with specific level(s) of thread support, e.g., only with MPI\_THREAD\_MULTIPLE, then MPI\_QUERY\_THREAD can be used to check whether the user initialized MPI to the correct level of thread support and, if not, raise an exception. (*End of advice to users.*)

#### 12.4.4 Helper Team Functionality

The following functions provide a mechanism to allow the MPI implementation to parallelize its internal processing using threads provided by the user. The application temporarily hands over control of its threads to the MPI implementation. These functions allow the application to create teams of threads, and use these teams to perform the processing required by the MPI operations initiated by one or more of the threads in the team. These functions do not alter the outcome of a program, they only allow an MPI program to communicate the availability of threads to the MPI implementation.

A team initially has no association with any threads. A thread's membership in a team is established when that thread calls `MPI_TEAM_JOIN`. A thread's membership in a team is dissolved (and optionally reestablished) when that thread calls `MPI_TEAM_LEAVE`. A team epoch begins when the first member calls `MPI_TEAM_JOIN`, and ends when the first member exits `MPI_TEAM_LEAVE` (except for `MPI_TEAM_BREAK`), on the team. It is erroneous for a thread to call `MPI_TEAM_JOIN` when it is already involved in any team epoch, including on the same team. Unless the option `MPI_TEAM_BREAK` is used, a thread can exit from the `MPI_TEAM_LEAVE` call only after all threads participating in the team epoch have called `MPI_TEAM_LEAVE`.

`MPI_TEAM_CREATE(teammaxsize, info, team)`

IN	teammaxsize	the maximum number of threads that will join the team (integer)
IN	info	info describing team usage (handle)
OUT	team	the team (handle)

`int MPI_Team_create(int teammaxsize, MPI_Info info, MPI_Team *team)`

`MPI_TEAM_CREATE(TEAMMAXSIZE, INFO, TEAM, IERROR)`

`INTEGER TEAMMAXSIZE, INFO, TEAM, IERROR`

This call creates a team for helper threads to be used with subsequent `MPI_TEAM_JOIN` and `MPI_TEAM_LEAVE` calls. This call is made exactly once per team. It is not required for the thread creating a team to ever join the team.

The `info` argument provides assertions to the runtime about how the threads team will be used. The following info keys are predefined:

`nobreak` if set to “true”, then no threads in the team (as defined by the `teamsize` parameter to `MPI_TEAM_JOIN`) will call `MPI_TEAM_LEAVE` with the `MPI_TEAM_BREAK` option.

The MPI implementation may synchronize between all the threads in an `MPI_TEAM_JOIN` or `MPI_TEAM_LEAVE` call during a team epoch, and assume that no thread in the team would call `MPI_TEAM_LEAVE` with the `MPI_TEAM_BREAK` option, on that team. The MPI implementation may expect that `teamsize` threads will be available to participate in communications performed during a team epoch.

`MPI_TEAM_FREE(team)`

INOUT      `team`                      The team (handle)

`int MPI_Team_free(MPI_Team *team)`

`MPI_TEAM_FREE(TEAM, IERROR)`

INTEGER `TEAM`, `IERROR`

This call marks `team` for deallocation. The handle is set to `MPI_TEAM_NULL`. Any pending operations that use this team will complete normally; the object is actually deallocated only if there are no other active references to it.

*Advice to implementors.* Implementors should be careful not to free team resources before all the threads in the team have called `MPI_TEAM_LEAVE`, possibly by internally reference counting on the handle. (*End of advice to implementors.*)

`MPI_TEAM_JOIN(teamsize, team)`

IN              `teamsize`                      The number of threads that are currently joining the team (integer)

IN              `team`                      The team (handle)

`int MPI_Team_join(int teamsizes, MPI_Team team)`

`MPI_TEAM_JOIN(TEAMSIZE, TEAM, IERROR)`

INTEGER `TEAMSIZES`, `TEAM`, `IERROR`

This call registers the calling thread as a participant in a team epoch on `team`, indicating that the thread will eventually call `MPI_TEAM_LEAVE`. A thread may only participate in one team epoch at a time. All threads participating in a team epoch must specify the same value for `teamsizes`. It is erroneous to call `MPI_TEAM_JOIN` with `teamsizes` larger than the `teammaxsize` value associated with `team`. Exactly `teamsizes` distinct threads must participate in `team` at each team epoch.

*Rationale.* The parameter `teamsizes` specifies the actual number of threads that will participate in the current epoch on `team`. This number may not be known when the call to `MPI_TEAM_CREATE` is made. For example, OpenMP does not guarantee that the number of threads requested will be the actual number of threads that execute a given set of tasks. (*End of rationale.*)

*Advice to users.* If the team is created with the info key `nobreak` set to “true”, the MPI implementation might treat the `MPI_TEAM_JOIN` call as a “contract” that this thread will be available to help MPI operations initiated by any members of the team (including itself), while maintaining the local/non-local semantics of the MPI operations. (*End of advice to users.*)

1 `MPI_TEAM_LEAVE(option, team)`

2     IN         option                     Options controlling behavior

3     IN         team                     The team (handle)

5  
6 `int MPI_Team_leave(int option, MPI_Team team)`

7 `MPI_TEAM_LEAVE(OPTION, TEAM, IERROR)`

8     INTEGER OPTION, TEAM, IERROR

9  
10     This call deregisters the calling thread from being a participant in the team epoch, and  
11     optionally reregisters the calling thread in a new team epoch.

12  
13     *Advice to users.* The MPI implementation may choose to synchronize all threads in  
14     the team during the `MPI_TEAM_LEAVE` call to effectively utilize all resources for MPI  
15     operations initiated by the team members. An MPI implementation may assign work  
16     from outside the team to a thread in `MPI_TEAM_LEAVE`, however any such work  
17     will not indefinitely delay a thread's exit from `MPI_TEAM_LEAVE`. (*End of advice to*  
18     *users.*)

19     The following values for the `option` argument are supported.

20  
21     **MPI\_TEAM\_DEFAULT** The thread is deregistered from being a participant in the team  
22     epoch. A thread can exit from the `MPI_TEAM_LEAVE` call only after all threads  
23     participating in the team epoch have called `MPI_TEAM_LEAVE`. The calling thread  
24     may be used by the MPI implementation to do work while waiting for the remaining  
25     team members to call `MPI_TEAM_LEAVE`. The current team epoch ends.

26  
27     **MPI\_TEAM\_REJOIN** A thread can exit from the `MPI_TEAM_LEAVE` call only after all  
28     threads participating in the team epoch have called `MPI_TEAM_LEAVE`. The calling  
29     thread may be used by the MPI implementation to do work while waiting for the re-  
30     maining team members to call `MPI_TEAM_LEAVE`. The [current]current team epoch  
31     ends and a new team epoch begins. The new team epoch uses the same `teamsize`  
32     value as the previous team epoch. This option may avoid the overhead of releasing  
33     and re-acquiring resources associated with `MPI_TEAM_LEAVE(MPI_TEAM_DEFAULT)`  
34     followed by `MPI_TEAM_JOIN`.

35  
36     **MPI\_TEAM\_BREAK** The thread is deregistered from being a participant in the team epoch  
37     without synchronizing with other threads in the team. [ The net effect is to reduce the  
38     `teamsize` value associated with `team` by 1 for the remainder of the current team epoch.  
39     ] The calling thread provides no further assistance to remaining members of `team`. It  
40     is erroneous to use `MPI_TEAM_BREAK` on a `team` for which the info key `nobreak` is set  
41     to "true".

42  
43     *Advice to users.* It is permitted for a team epoch to mix `MPI_TEAM_DEFAULT`,  
44     `MPI_TEAM_REJOIN`, and `MPI_TEAM_BREAK` in the calls to `MPI_TEAM_LEAVE`. Note  
45     the restrictions on `teamsize` when using `MPI_TEAM_REJOIN`. (*End of advice to users.*)  
46  
47  
48

### 12.4.5 Helper Team Examples

#### **Example 12.3**

The following example shows OpenMP code that uses multiple threads to help MPI communication using `MPI_ALLREDUCE` initiated by one thread. It also demonstrates use of the info argument key `nobreak`.

The value `COUNT` is a constant specifying the size of the arrays and subsequent loops. It is defined outside this example.

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48

```

1  MPI_Team team;
2  MPI_Info info;
3  double oldval = 0.0, newval = 9.9e99;
4  double tolerance = 1.0e-6;
5  double sendbuf[COUNT] = { 0.0 };
6  double recvbuf[COUNT] = { 0.0 };
7  MPI_Info_create(&info);
8  MPI_Info_set(info, "nobreak", "true");
9  MPI_Team_create(omp_get_thread_limit(), info, &team);
10 MPI_Info_free(&info);
11 #pragma omp parallel num_threads(omp_get_thread_limit()) \
12     shared(newval, oldval, sendbuf, team)
13 {
14     while (abs(newval - oldval) > tolerance) {
15         double myval = 0.0;
16         int i;
17         oldval = newval;
18
19         #pragma omp for
20         for (i = 0; i < COUNT; i++) {
21             myval += do_work(i, sendbuf);
22         }
23
24         #pragma omp critical
25         {
26             newval += myval;
27         }
28
29         MPI_Team_join(omp_get_num_threads(), team);
30         /* this barrier is not required but helps ensure
31          * all threads arrive before the MPI_Allreduce begins */
32         #pragma omp barrier
33         #pragma omp master
34         {
35             MPI_Allreduce(sendbuf, recvbuf, COUNT, MPI_DOUBLE,
36                           MPI_SUM, MPI_COMM_WORLD);
37         }
38         /* The remaining threads directly go to MPI_Team_leave.
39          * This is where the threads help on the MPI_ALLREDUCE */
40         MPI_Team_leave(MPI_TEAM_DEFAULT, team);
41
42         #pragma omp for
43         for (i = 0; i < COUNT; i++) {
44             sendbuf[i] = recvbuf[i];
45         }
46     }
47 }
48 MPI_Team_free(&team);

```

[This text and extra page work around a bug in the MPIupdate macros. Otherwise the last page is not colored red (marked as a change).]

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48

# Bibliography



# Index

- cancel\_fn, [4](#), [5](#)
- CONST:MPI::Grequest, [2](#), [2](#)
- CONST:MPI::Request, [2](#), [5](#)
- CONST:MPI::Status, [2](#), [8](#), [9](#)
- CONST:MPI\_COMM\_WORLD, [13](#)
- CONST:MPI\_ERR\_IN\_STATUS, [4](#)
- CONST:MPI\_Request, [2](#), [5](#)
- CONST:MPI\_REQUEST\_NULL, [4](#)
- CONST:MPI\_Status, [2](#), [8](#), [9](#)
- CONST:MPI\_STATUS\_IGNORE, [3](#)
- CONST:MPI\_STATUSES\_IGNORE, [3](#), [5](#)
- CONST:MPI\_TEAM\_BREAK, [16](#), [18](#)
- CONST:MPI\_TEAM\_DEFAULT, [18](#)
- CONST:MPI\_TEAM\_NULL, [17](#)
- CONST:MPI\_TEAM\_REJOIN, [18](#)
- CONST:MPI\_THREAD\_FUNNELED, [13](#)
- CONST:MPI\_THREAD\_MULTIPLE, [13–15](#)
- CONST:MPI\_THREAD\_SERIALIZED, [13](#)
- CONST:MPI\_THREAD\_SINGLE, [13](#), [14](#)
- CONST:nobreak, [16–19](#)
- EXAMPLES:MPI\_Grequest\_complete, [5](#)
- EXAMPLES:MPI\_Grequest\_start, [5](#)
- EXAMPLES:MPI\_Info:nobreak, [19](#)
- EXAMPLES:MPI\_Team\_create, [19](#)
- EXAMPLES:MPI\_TEAM\_DEFAULT, [19](#)
- EXAMPLES:MPI\_Team\_free, [19](#)
- EXAMPLES:MPI\_Team\_join, [19](#)
- EXAMPLES:MPI\_Team\_leave, [19](#)
- EXAMPLES:Threads and MPI, [10](#)
- free\_fn, [3–5](#)
- MPI\_ALLREDUCE, [19](#)
- MPI\_CANCEL, [1](#), [5](#)
- MPI\_CANCEL(request), [4](#)
- MPI\_FILE\_OPEN, [12](#)
- MPI\_FINALIZE, [11](#)
- MPI\_GET\_COUNT, [9](#)
- MPI\_GET\_COUNT(status, datatype, count), [9](#)
- MPI\_GET\_ELEMENTS, [9](#)
- MPI\_GET\_ELEMENTS(status, datatype, count), [9](#)
- MPI\_GET\_ELEMENTS\_X, [9](#)
- MPI\_GET\_ELEMENTS\_X(status, datatype, count), [9](#)
- MPI\_GREQUEST\_COMPLETE, [2–5](#)
- MPI\_GREQUEST\_COMPLETE(request), [4](#), [5](#)
- MPI\_GREQUEST\_START, [2](#)
- MPI\_GREQUEST\_START(query\_fn, free\_fn, cancel\_fn, extra\_state, request), [2](#)
- MPI\_INIT, [12–15](#)
- MPI\_INIT\_THREAD, [13–15](#)
- MPI\_INIT\_THREAD(), [15](#)
- MPI\_INIT\_THREAD(required, provided), [13](#)
- MPI\_INITIALIZED, [15](#)
- MPI\_IPROBE, [11](#)
- MPI\_IS\_THREAD\_MAIN, [13](#)
- MPI\_IS\_THREAD\_MAIN(flag), [15](#)
- MPI\_PROBE, [11](#)
- MPI\_QUERY\_THREAD, [15](#)
- MPI\_QUERY\_THREAD(provided), [15](#)
- MPI\_RECV, [9](#)
- MPI\_REQUEST\_FREE, [4](#), [5](#)
- MPI\_REQUEST\_FREE(request), [4](#)
- MPI\_REQUEST\_GET\_STATUS, [3](#)
- MPI\_STATUS\_SET\_CANCELLED(status, flag), [9](#)
- MPI\_STATUS\_SET\_ELEMENTS, [9](#)
- MPI\_STATUS\_SET\_ELEMENTS(status, datatype, count), [8](#)
- MPI\_STATUS\_SET\_ELEMENTS\_X, [9](#)
- MPI\_STATUS\_SET\_ELEMENTS\_X(status, datatype, count), [8](#)
- MPI\_TEAM\_CREATE, [17](#)
- MPI\_TEAM\_CREATE(teammaxsize, info, team), [16](#)
- MPI\_TEAM\_FREE(team), [17](#)
- MPI\_TEAM\_JOIN, [16–18](#)

1 MPI\_TEAM\_JOIN(teamsize, team), [17](#)  
 2 MPI\_TEAM\_LEAVE, [16–18](#)  
 3 MPI\_TEAM\_LEAVE(option, team), [18](#)  
 4 MPI\_TEST, [5](#)  
 5 MPI\_TEST(request, flag, status), [5](#)  
 6 MPI\_TEST\_CANCELLED, [3](#)  
 7 MPI\_TEST\_CANCELLED(status, flag), [9](#)  
 8 MPI\_TESTALL, [3, 4, 8, 11](#)  
 9 MPI\_TESTANY, [3, 4, 8, 11](#)  
 10 MPI\_TESTSOME, [3, 4, 8, 11](#)  
 11 MPI\_WAIT, [1, 5, 11](#)  
 12 MPI\_WAIT(request, status), [5](#)  
 13 MPI\_WAITALL, [3, 4, 8, 11](#)  
 14 MPI\_WAITANY, [3, 4, 8, 11](#)  
 15 MPI\_WAITSOME, [3, 4, 8, 11](#)  
 16 MPI\_WIN\_CREATE, [12](#)  
 17 mpiexec, [13, 14](#)  
 18  
 19 query\_fn, [3–5](#)  
 20  
 21 TYPEDEF:MPI\_Grequest\_cancel\_function(void  
 22     \*extra\_state, int complete), [4](#)  
 23 TYPEDEF:MPI\_Grequest\_free\_function(void  
 24     \*extra\_state), [3](#)  
 25 TYPEDEF:MPI\_Grequest\_query\_function(void  
 26     \*extra\_state, MPI\_Status \*status),  
 27     [3](#)  
 28  
 29  
 30  
 31  
 32  
 33  
 34  
 35  
 36  
 37  
 38  
 39  
 40  
 41  
 42  
 43  
 44  
 45  
 46  
 47  
 48