

D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

January 8, 2011

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.*)

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

7 A new generalized request is started with

```
8
9
10 MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)
11     IN          query_fn          callback function invoked when request status is queried
12                                     (function)
13     IN          free_fn           callback function invoked when request is freed (func-
14                                     tion)
15     IN          cancel_fn        callback function invoked when request is cancelled
16                                     (function)
17     IN          extra_state       extra state
18     OUT         request          generalized request (handle)
```

```
19
20
21 int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
22                       MPI_Grequest_free_function *free_fn,
23                       MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
24                       MPI_Request *request)
```

```
25
26 MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
27                   IERROR)
28     INTEGER REQUEST, IERROR
29     EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
30     INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
31 {static MPI::Grequest
32     MPI::Grequest::Start(const MPI::Grequest::Query_function*
33                         query_fn, const MPI::Grequest::Free_function* free_fn,
34                         const MPI::Grequest::Cancel_function* cancel_fn,
35                         void *extra_state) (binding deprecated, see Section 15.2) }
```

36
 37
 38 *Advice to users.* Note that a generalized request belongs, in C++, to the class
 39 `MPI::Grequest`, which is a derived class of `MPI::Request`. It is of the same type as
 40 regular requests, in C and Fortran. (*End of advice to users.*)

41
 42 The call starts a generalized request and returns a handle to it in `request`.

43 The syntax and meaning of the callback functions are listed below. All callback func-
 44 tions are passed the `extra_state` argument that was associated with the request by the start-
 45 **ticket0.** ing call `MPI_GREQUEST_START`. [This can]The memory location to which this argument
 46 **points can** be used to maintain user-defined state for the request.

47 In C, the query function is

48

```
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 15.2)}
```

[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 15.2)}
```

[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

1 by MPI.

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

10
 11 *Advice to users.* Calling `MPI_REQUEST_FREE(request)` will cause the request handle
 12 to be set to `MPI_REQUEST_NULL`. This handle to the generalized request is no longer
 13 valid. However, user copies of this handle are valid until after `free_fn` completes since
 14 MPI does not deallocate the object until then. Since `free_fn` is not called until after
 15 `MPI_GREQUEST_COMPLETE`, the user copy of the handle can be used to make this
 16 call. Users should note that MPI will deallocate the object after `free_fn` executes. At
 17 this point, user copies of the request handle no longer point to a valid request. MPI
 18 will not set user copies to `MPI_REQUEST_NULL` in this case, so it is up to the user to
 19 avoid accessing this stale handle. This is a special case [where]in which MPI defers
 20 deallocating the object until a later time that is known by the user. (*End of advice*
 21 *to users.*)

22
 23 In C, the cancel function is

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

25
 26 in Fortran

27 `SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)`

28 `INTEGER IERROR`

29 `INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE`

30 `LOGICAL COMPLETE`

31
 32 and in C++

33 `{typedef int MPI::Grequest::Cancel_function(void* extra_state,`
 34 `bool complete); (binding deprecated, see Section 15.2)}`

35
 36 [cancel_fn]The `cancel_fn` function is invoked to start the cancelation of a generalized
 37 request. It is called by `MPI_CANCEL(request)`. MPI passes [to the callback function
 38 `complete=true`]complete=true to the callback function if `MPI_GREQUEST_COMPLETE` was
 39 already called on the request, and `complete=false` otherwise.

40 All callback functions return an error code. The code is passed back and dealt with as
 41 appropriate for the error code by the MPI function that invoked the callback function. For
 42 example, if error codes are returned then the error code returned by the callback function
 43 will be returned by the MPI function that invoked the callback function. In the case of
 44 an `MPI_{WAIT|TEST}{ANY}` call that invokes both `query_fn` and `free_fn`, the MPI call will
 45 return the error code returned by the last callback, namely `free_fn`. If one or more of the
 46 requests in a call to `MPI_{WAIT|TEST}{SOME|ALL}` failed, then the MPI call will return
 47 `MPI_ERR_IN_STATUS`. In such a case, if the MPI call was passed an array of statuses, then
 48 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 15.2*) }

The call informs MPI that the operations represented by the generalized request `request` are complete (see definitions in Section 2.4). 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 15.2) }
```

This call modifies the opaque part of status so that a call to `MPI_GET_ELEMENTS` 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)` 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`.

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` 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 15.2)
    }

```

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 [1], 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.*)

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

5
 6 The two main requirements for a thread-compliant implementation are listed below.

- 7 1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI
 8 calls and the outcome will be as if the calls executed in some order, even if their
 9 execution is interleaved.
- 10
 11 2. Blocking MPI calls will block the calling thread only, allowing another thread to
 12 execute, if available. The calling thread will be blocked until the event on which it
 13 is waiting occurs. Once the blocked communication is enabled and can proceed, then
 14 the call will complete and the thread will be marked runnable, within a finite time.
 15 A blocked thread will not prevent progress of other runnable threads on the same
 16 process, and will not prevent them from executing MPI calls.

17
 18 **Example 12.2** Process 0 consists of two threads. The first thread executes a blocking
 19 send call `MPI_Send(buff1, count, type, 0, 0, comm)`, whereas the second thread executes
 20 a blocking receive call `MPI_Recv(buff2, count, type, 0, 0, comm, &status)`, i.e., the first
 21 thread sends a message that is received by the second thread. This communication should
 22 always succeed. According to the first requirement, the execution will correspond to some
 23 interleaving of the two calls. According to the second requirement, a call can only block
 24 the calling thread and cannot prevent progress of the other thread. If the send call went
 25 ahead of the receive call, then the sending thread may block, but this will not prevent
 26 the receiving thread from executing. Thus, the receive call will occur. Once both calls
 27 occur, the communication is enabled and both calls will complete. On the other hand, a
 28 single-threaded process that posts a send, followed by a matching receive, may deadlock.
 29 The progress requirement for multithreaded implementations is stronger, as a blocked call
 30 cannot prevent progress in other threads.

31
 32 *Advice to implementors.* MPI calls can be made thread-safe by executing only one at
 33 a time, e.g., by protecting MPI code with one process-global lock. However, blocked
 34 operations cannot hold the lock, as this would prevent progress of other threads in
 35 the process. The lock is held only for the duration of an atomic, locally-completing
 36 suboperation such as posting a send or completing a send, and is released in between.
 37 Finer locks can provide more concurrency, at the expense of higher locking overheads.
 38 Concurrency can also be achieved by having some of the MPI protocol executed by
 39 separate server threads. (*End of advice to implementors.*)

40 41 12.4.2 Clarifications

42
 43 **Initialization and Completion** The call to `MPI_FINALIZE` should occur on the same thread
 44 that initialized MPI. We call this thread the **main thread**. The call should occur only after
 45 all the process threads have completed their MPI calls, and have no pending communications
 46 or I/O operations.

47
 48 *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.*)

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

5
 6 *Rationale.* Few C library functions are signal safe, and many have cancellation points
 ticket0. 7 — points [where]at which the thread executing them may be cancelled. The above
 8 restriction simplifies implementation (no need for the MPI library to be “async-cancel-
 ticket0. 9 safe” or [“async-signal-safe.”]“async-signal-safe”). (*End of rationale.*)

10
 11 *Advice to users.* Users can catch signals in separate, non-MPI threads (e.g., by
 12 masking signals on MPI calling threads, and unmasking them in one or more non-MPI
 13 threads). A good programming practice is to have a distinct thread blocked in a
 14 call to `sigwait` for each user expected signal that may occur. Users must not catch
 15 signals used by the MPI implementation; as each MPI implementation is required to
 16 document the signals used internally, users can avoid these signals. (*End of advice to*
 17 *users.*)

18
 19 *Advice to implementors.* The MPI library should not invoke library calls that are
 20 not thread safe, if multiple threads execute. (*End of advice to implementors.*)

21 12.4.3 Initialization

22
 23 The following function may be used to initialize MPI, and initialize the MPI thread envi-
 24 ronment, instead of `MPI_INIT`.

25
 26
 27 `MPI_INIT_THREAD(required, provided)`

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

30
 31
 32 `int MPI_Init_thread(int *argc, char *((*argv)[]), int required,`
 33 `int *provided)`

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

36
 37 `{int MPI::Init_thread(int& argc, char**& argv, int required) (binding`
 38 `deprecated, see Section 15.2) }`

39
 40 `{int MPI::Init_thread(int required) (binding deprecated, see Section 15.2) }`

41
 ticket0. 42 *Advice to users.* In C and C++, the passing of `argc` and `argv` is [optional.]optional,
 ticket0. 43 as with `MPI_INIT` as discussed in Section 8.7. In C, [this is accomplished by passing
 44 the appropriate null pointer.] the appropriate null pointers may be passed in their
 ticket0. 45 place. In C++, [this is accomplished with two separate bindings to cover these two
 46 cases. This is as with `MPI_INIT` as discussed in Section 8.7.]two separate bindings
 47 cover these two cases. (*End of advice to users.*)
 48

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 14).

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

1 can use library functions that are not thread safe, without risking conflicts with user
 2 threads. Also, the model of one communication thread, multiple computation threads
 3 fits many applications well, e.g., if the process code is a sequential Fortran/C/C++
 4 program with MPI calls that has been parallelized by a compiler for execution on an
 5 SMP node, in a cluster of SMPs, then the process computation is multi-threaded, but
 6 MPI calls will likely execute on a single thread.

7 The design accommodates a static specification of the thread support level, for en-
 8 vironments that require static binding of libraries, and for compatibility for current
 9 multi-threaded MPI codes. (*End of rationale.*)
 10

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

15 Some implementors may want to use different MPI libraries for different levels of thread
 16 support. They can do so using dynamic linking and selecting which library will be
 17 linked when `MPI_INIT_THREAD` is invoked. If this is not possible, then optimizations
 18 for lower levels of thread support will occur only when the level of thread support
 19 required is specified at link time. (*End of advice to implementors.*)
 20

21 The following function can be used to query the current level of thread support.
 22

23
 24 `MPI_QUERY_THREAD(provided)`

25 OUT provided provided level of thread support (integer)
 26

27 `int MPI_Query_thread(int *provided)`

28 `MPI_QUERY_THREAD(PROVIDED, IERROR)`

29 INTEGER PROVIDED, IERROR

30 {`int MPI::Query_thread()` (*binding deprecated, see Section 15.2*) }

ticket0. 31
 32
 33 The call returns in `provided` the current level of thread `[support. This]support, which`
 34 will be the value returned in `provided` by `MPI_INIT_THREAD`, if MPI was initialized by a
 35 call to `MPI_INIT_THREAD()`.
 36

37
 38 `MPI_IS_THREAD_MAIN(flag)`

39 OUT flag true if calling thread is main thread, false otherwise
 40 (logical)
 41

42 `int MPI_Is_thread_main(int *flag)`

43 `MPI_IS_THREAD_MAIN(FLAG, IERROR)`

44 LOGICAL FLAG

45 INTEGER IERROR

46 {`bool MPI::Is_thread_main()` (*binding deprecated, see Section 15.2*) }

47
 48

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.*)

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

- [1] International Organization for Standardization, Geneva. *Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, December 1996. [12.4](#)

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_THREAD_FUNNELED, [13](#)
- CONST:MPI_THREAD_MULTIPLE, [13](#), [15](#)
- CONST:MPI_THREAD_SERIALIZED, [13](#)
- CONST:MPI_THREAD_SINGLE, [13](#), [14](#)

- EXAMPLES:MPI_Grequest_complete, [5](#)
- EXAMPLES:MPI_Grequest_start, [5](#)
- EXAMPLES:Threads and MPI, [10](#)

- free_fn, [3-5](#)

- MPI_CANCEL, [1](#), [5](#)
- MPI_CANCEL(request), [4](#)
- MPI_FILE_OPEN, [11](#)
- MPI_FINALIZE, [10](#)
- MPI_GET_COUNT, [8](#)
- MPI_GET_COUNT(status, datatype, count), [8](#)
- MPI_GET_ELEMENTS, [8](#), [9](#)
- MPI_GET_ELEMENTS(status, datatype, count), [8](#)
- 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](#), [13](#), [15](#)
- MPI_INIT_THREAD, [13-15](#)
- MPI_INIT_THREAD(), [14](#)
- MPI_INIT_THREAD(required, provided), [12](#)
- MPI_INITIALIZED, [15](#)
- MPI_IPROBE, [11](#)
- MPI_IS_THREAD_MAIN, [13](#)
- MPI_IS_THREAD_MAIN(flag), [14](#)
- MPI_PROBE, [11](#)
- MPI_QUERY_THREAD, [15](#)
- MPI_QUERY_THREAD(provided), [14](#)
- 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, [8](#)
- MPI_STATUS_SET_ELEMENTS(status, datatype, count), [8](#)
- MPI_TEST, [5](#)
- MPI_TEST(request, flag, status), [5](#)
- MPI_TEST_CANCELLED, [3](#)
- MPI_TEST_CANCELLED(status, flag), [9](#)
- MPI_TESTALL, [3](#), [4](#), [8](#), [11](#)
- MPI_TESTANY, [3](#), [4](#), [8](#), [11](#)
- MPI_TESTSOME, [3](#), [4](#), [8](#), [11](#)
- MPI_WAIT, [1](#), [5](#), [11](#)
- MPI_WAIT(request, status), [5](#)
- MPI_WAITALL, [3](#), [4](#), [8](#), [11](#)
- MPI_WAITANY, [3](#), [4](#), [8](#), [11](#)
- MPI_WAITSOME, [3](#), [4](#), [8](#), [11](#)
- MPI_WIN_CREATE, [11](#)
- mpiexec, [13](#)
- query_fn, [3-5](#)
- TYPEDEF:MPI_Grequest_cancel_function(void *extra_state, int complete), [4](#)
- TYPEDEF:MPI_Grequest_free_function(void *extra_state), [3](#)
- TYPEDEF:MPI_Grequest_query_function(void *extra_state, MPI_Status *status), [3](#)