D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

February 4, 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 ticket0. requests.

The chapter continues, in Section 12.4, with a discussion of how threads are to be handled in MPI, including interoperability with threads and helper thread functionality ticket0. to share threads between the application and the MPI implementation. Although thread compliance is not required, the standard specifies how threads are to work if they are provided. ticket0.

Section 12.5 discusses MPI functionality to create and free shared memory regions.

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]The memory location to which this argument

ticket0.

points 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 ??)*}

ticket0.

[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). ticket0.

[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 ??)*}

ticket0.

[free_fn]The free_fn function is invoked to clean up user-allocated resources when the generalized request is freed. ticket0.

[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 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
> ticket0. 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 ??)}
```

ticket0.

[cancel_fn]The cancel_fn function is invoked to start the cancelation of a generalized
ticket0. 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.

```
typedef struct {
   MPI_Comm comm;
   int tag;
   int root;
   int valin;
   int *valout;
   MPI_Request request;
   } ARGS;


int myreduce(MPI_Comm comm, int tag, int root,
              int valin, int *valout, MPI_Request *request)
{
   ARGS *args;
   pthread_t thread;

   /* start request */
   MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);

   args = (ARGS*)malloc(sizeof(ARGS));
   args->comm = comm;
   args->tag = tag;
   args->root = root;
   args->valin = valin;
   args->valout = valout;
   args->request = *request;

   /* spawn thread to handle request */
   /* The availability of the pthread_create call is system dependent */
   pthread_create(&thread, NULL, reduce_thread, args);

   return MPI_SUCCESS;
}

/* thread code */
void* reduce_thread(void *ptr)
{
   int lchild, rchild, parent, lval, rval, val;
   MPI_Request req[2];
   ARGS *args;

   args = (ARGS*)ptr;

   /* compute left,right child and parent in tree; set
      to MPI_PROC_NULL if does not exist  */
   /* code not shown */
   ...
```

```
   MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
   MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
   MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
   val = lval + args->valin + rval;
   MPI_Send( &val, 1, MPI_INT, parent, args->tag, args->comm );
   if (parent == MPI_PROC_NULL) *(args->valout) = val;
   MPI_Grequest_complete((args->request));
   free(ptr);
   return(NULL);
}

int query_fn(void *extra_state, MPI_Status *status)
{
   /* always send just one int */
   MPI_Status_set_elements(status, MPI_INT, 1);
   /* can never cancel so always true */
   MPI_Status_set_cancelled(status, 0);
   /* choose not to return a value for this */
   status->MPI_SOURCE = MPI_UNDEFINED;
   /* tag has no meaning for this generalized request */
   status->MPI_TAG = MPI_UNDEFINED;
   /* this generalized request never fails */
   return MPI_SUCCESS;
}


int free_fn(void *extra_state)
{
   /* this generalized request does not need to do any freeing */
   /* as a result it never fails here */
   return MPI_SUCCESS;
}


int cancel_fn(void *extra_state, int complete)
{
   /* This generalized request does not support cancelling.
      Abort if not already done.  If done then treat as if cancel failed.*/
   if (!complete) {
     fprintf(stderr,
             "Cannot cancel generalized request - aborting program\n");
     MPI_Abort(MPI_COMM_WORLD, 99);
     }
   return MPI_SUCCESS;
}
```

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 ticket0. calls [use]to use the same request [mechanism. This]mechanism, which allows one to wait or ticket0. 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 ticket0. 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* **??***)* }

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.

> ticket0. *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 ??)* }

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 and functions to allow an application to share threads with the MPI library. 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. ticket0. ticket0.

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.*) ticket0.

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. ticket0.

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.*) ticket0.

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.*) ticket0.

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." (*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.] the appropriate null pointer 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 cover these two cases. (*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 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 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 ??)* }

ticket0. 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). ticket0.

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.*) ticket0.

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.*)

ticket0.

12.4.4 Sharing Helper Threads with the MPI Implementation

The following functions may be used for applications to temporarily hand-over control of its threads for the MPI implementation to use. These functions allow the application to create teams of threads, and use these teams to perform the processing required by the MPI implementation for MPI calls made by one or more of the threads in the team.

MPI_HELPER_TEAM_CREATE(team_size, info, team)

IN	team_size	total number of members in team (integer)
IN	info	info argument (handle)
OUT	team	handle describing team (handle)

```
int MPI_Helper_team_create(int team_size, MPI_Info info,
             MPI_Helper_team *team)
```

```
MPI_HELPER_TEAM_CREATE(TEAM_SIZE, INFO, TEAM, IERROR)
    INTEGER TEAM_SIZE, INFO, TEAM, IERROR
```

This call creates a team of helper threads to be used with subsequent JOIN calls. This call must be made by only one thread. It is not required for the thread creating a team to join the team. A thread can be a part of any number of teams.

MPI_HELPER_TEAM_JOIN(team)

IN	team	handle describing team (handle)

```
int MPI_Helper_team_join(MPI_Helper_team team)
```

```
MPI_HELPER_TEAM_JOIN(TEAM, IERROR)
    INTEGER TEAM, IERROR
```

This call registers the calling thread as an active participant in the team. A team has to be first created using the MPI_HELPER_THREAD_CREATE before a thread can join it as an active participant. The caller threads resources may now be used by communications started by other members of the team. A thread may only be active in one team at a time.

MPI_HELPER_TEAM_LEAVE(team)

IN	team	handle describing team (handle)

```
int MPI_Helper_team_leave(MPI_Helper_team team)
```

```
MPI_HELPER_TEAM_LEAVE(TEAM, IERROR)
    INTEGER TEAM, IERROR
```

This call deregisters the calling thread from being an active participant in the team. This call must be made by all members of the team.

Non-blocking operations cannot span JOIN-LEAVE boundaries. That is, all non-blocking operations initiated within the JOIN-LEAVE boundary have to complete within the boundary.

Discussion Item: Is this restriction required?

Advice to users. The MPI implementation can use any of the resources available in the entire team for any MPI calls made between MPI_HELPER_TEAM_JOIN and MPI_HELPER_TEAM_LEAVE by any thread in the team. The MPI implementation may choose to make MPI_HELPER_TEAM_JOIN, MPI_HELPER_TEAM_LEAVE or both blocking to achieve this. The MPI implementation might treat the MPI_HELPER_TEAM_JOIN call as a “promise” that this thread is available to help MPI operations initiated by other members of the team (including itself), while maintaining the local/non-local semantics of the MPI operations (that is, the completion of local MPI operations depends only on the local executing process and does not require communication occurring with another user process). (*End of advice to users.*)

MPI_HELPER_TEAM_FREE(team)

INOUT	team	handle describing team (handle)

```
int MPI_Helper_team_free(MPI_Helper_team *team)
```

```
MPI_HELPER_TEAM_FREE(TEAM, IERROR)
    INTEGER TEAM, IERROR
```

This call frees the team object team and returns a null handle (equal to MPI_TEAM_NULL). This call must be made by only one thread. It is not required for the same thread that created this team to free it. MPI_TEAM_FREE(team) can be invoked by a thread only after it has completed its involvement in MPI communications initiated while it had joined the team team: i.e., the thread has called MPI_TEAM_LEAVE on the team, before it can free the team.

MPI_HELPER_TEAM_FENCE(team)

IN	team	handle describing team (handle)

```
int MPI_Helper_team_fence(MPI_Helper_team team)
```

```
MPI_HELPER_TEAM_FENCE(TEAM, IERROR)
    INTEGER TEAM, IERROR
```

This call is similar to MPI_HELPER_TEAM_LEAVE with respect to allowing threads to completing any outstanding MPI operations within the team. However, it does not cause the threads to leave the team. MPI_HELPER_TEAM_FENCE is conceptually identical to calling MPI_HELPER_TEAM_LEAVE followed by MPI_HELPER_TEAM_JOIN on the same team.

NOTE: This call was suggested at one of the previous MPI Forums, but no one in the working group is convinced of it. We are planning to drop it.

12.4.5 Examples

Example 12.3 The following example shows an OpenMP code that uses multiple threads to help MPI communication using MPI_ALLREDUCE initiated by one thread.

```
...
MPI_Helper_team team;
MPI_Helper_team_create(0, omp_get_num_threads(), MPI_INFO_NULL, &team);

#pragma omp parallel num_threads(N) {
   ...
   t = omp_get_thread_num();

   /* some computation and/or communication */

   MPI_Helper_team_join(team);

   if (t == 0) {
      MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm);
   }
   else {
      /* The remaining threads directly go to MPI_Helper_team_leave */
   }
```

```
   MPI_Helper_team_leave(team);

   /* more computation and/or communication */
}

MPI_Helper_team_free(&team);
```

Example 12.4 The following example shows an OpenMP code that uses multiple threads to help MPI communication initiated by some threads.

```
...
MPI_Helper_team team;
MPI_Helper_team_create(0, omp_get_num_threads(), MPI_INFO_NULL, &team);
#pragma omp parallel num_threads(N) {
   ...
   t = omp_get_thread_num();

   /* some computation and/or communication */

   MPI_Helper_team_join(team);

   if (t == 0) {
      MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm1);
   }
   else if (t == 1) {
      MPI_Bcast(buffer, count, datatype, root, comm2);
   }
   else if (t == 2) {
      MPI_Send(buf, count, datatype, dest, tag, comm3);
   }
   else {
      /* The remaining threads directly go to MPI_Helper_team_leave */
   }

   MPI_Helper_team_leave();

   /* more computation and/or communication */
}

MPI_Helper_team_free(&team);
```

12.5 MPI and Shared Memory

This section specifies methods in MPI to portably create and free shared memory regions. Shared memory regions created using these calls are usable for load/store operations and MPI operations.

MPI_COMM_SHM_ALLOC(comm, size, info, baseptr, shm)

IN	comm	communicator (handle)
IN	size	size of the shared memory region
IN	info	info argument (handle)
OUT	baseptr	pointer to beginning of memory segment allocated
OUT	shm	handle to the shared memory allocation

```
int MPI_Comm_shm_alloc(MPI_Comm comm, MPI_Aint size, MPI_Info info,
             void *baseptr, MPI_Shm shm)
```

```
MPI_COMM_SHM_ALLOC(COMM, SIZE, INFO, BASEPTR, SHM, IERROR)
    INTEGER COMM, SIZE, INFO, SHM, IERROR
    <type> BASEPTR(*)
```

This is a collective call that allocates a region of shared memory accessible by the ranks in an input communicator. The semantics of this call are similar to that of MPI_ALLOC_MEM. An error code of MPI_ERR_COMM is returned if no shared memory is possible.

The info argument provides optimization hints to the runtime. The following info key is predefined:

symm_alloc — if set to true, then the implementation can try to return a symmetric base pointer baseptr to all processes in the communicator.

Advice to users. Users cannot assume that the base pointers returned on all processes are symmetric, even if the info argument is set to symm_alloc. Users can perform an MPI_Allreduce on the base pointers to verify if the allocation was symmetric or not.

Symmetric allocation might be expensive and/or limited, as the implementation might have to move data to satisfy the request. So, the users should limit how much shared memory they allocate as symmetric. (*End of advice to users.*)

MPI_COMM_SHM_FREE(shm)

IN	shm	shared memory handle

```
int MPI_Comm_shm_free(MPI_Shm *shm)
```

```
MPI_COMM_SHM_FREE(SHM, IERROR)
    INTEGER COMM, IERROR
    <type> BASEPTR(*)
```

This is a collective call that frees a region of shared memory allocated with MPI_COMM_SHM_ALLOC and sets the shm handle to MPI_SHM_NULL. MPI_COMM_SHM_FREE can be invoked by a process only after it has completed the involvement of the shared memory region in all outstanding MPI operations.

Advice to implementors. MPI_COMM_SHM_FREE requires a barrier synchronization: no process can return from free until all processes in the group of comm called free. This, to ensure that no process will attempt to access a shared memory region after it was freed. (*End of advice to implementors.*)

MPI_COMM_SHM_SYNC(shm)

IN	shm	shared memory handle

```
int MPI_Comm_shm_sync(MPI_Shm shm)
```

```
MPI_COMM_SHM_SYNC(SHM, IERROR)
    INTEGER SHM, IERROR
```

The MPI_COMM_SHM_SYNC call ensures that stores to the shared memory region shm are visible to other processes in the communicator comm.

Discussion item: what should be the syntax of this call? Address + size? Does the address need to be a base pointer?

12.5.1 Examples

Example 12.5 The following example shows a code that uses shared memory allocated by two processes.

```
...
MPI_Shm shm;
ret = MPI_Comm_alloc_shm(comm, size, MPI_INFO_NULL, &baseptr, &shm);
if (ret == MPI_SUCCESS) {
   int *ptr = (int *) baseptr;
   int rank;

   MPI_Comm_rank(comm, &rank);
   ptr[rank] = rank;

   MPI_Comm_shm_sync(shm);

   if (rank == 0) {
      int sum = 0;
      int i;

      for (i = 0; i < size; i++) {
         sum += ptr[i];
      }

      printf("sum = %d\n", sum);
   }
```

```
    MPI_Comm_free_shm(&shm);
}
```

Bibliography

Index