before other MPI routines may be called. To provide for this, MPI includes an initialization routine MPI_INIT.

MPI_INIT()

```
int MPI_Init(int *argc, char ***argv)
```

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

{`void MPI::Init(int& argc, char**& argv)` *(binding deprecated, see Section 15.2)* }

{`void MPI::Init()` *(binding deprecated, see Section 15.2)* }

ticket313.

[All MPI programs] Each MPI process must contain exactly one call to an MPI initialization routine: MPI_INIT or MPI_INIT_THREAD. Subsequent calls to any initialization routines are erroneous. The only MPI functions that may be invoked before the MPI initialization routines are called are MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED.

The version for ISO C accepts the argc and argv that are provided by the arguments to main or NULL:

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    /* parse arguments */
    /* main program    */

    MPI_Finalize();     /* see below */
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc e argv arguments of main in C. [and C++. In C++, there is an alternative binding for MPI::Init that does not have these arguments at all.

ticket313. ticket313.

Rationale. In some applications, libraries may be making the call to MPI_Init, and may not have access to argc and argv from main. It is anticipated that applications requiring special information about the environment or information supplied by mpiexec can get that information from environment variables. (*End of rationale.*)

]

ticket313.

After MPI is initialized, the application can access information about the execution environment by querying the predefined info object MPI_INFO_ENV. The following keys are predefined for this object, corresponding to the arguments of MPI_COMM_SPAWN or of mpiexec:

command name of program executed

argv (space separated) arguments to command maxprocs Maximum number of MPI processes to start.

soft Allowed values for number of processors

host Hostname.

arch Architecture name.

wdir Working directory of the MPI process

file Value is the name of a file in which additional information is specified.

thread_level Requested level of thread support (if requested before the program started execution)

The info object MPI_INFO_ENV need not contain a (key,value) pair for each of these predefined keys; the set of (key,value) pairs provided is implementation-dependent. Implementations may provide additional, implementation specific, (key,value) pairs.

In case where the MPI processes where started with MPI_COMM_SPAWN_MULTIPLE or, equivalently, with a startup mechanism that supports multiple process specifications, then the values stored in the info object MPI_INFO_KEY at a process are those values that affect the local MPI process.

Example 8.3 If MPI is started with a call to

```
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
```

Then the first 5 processes will have have in their MPI_INFO_ENV object the pairs (command, ocean), (maxprocs,5), and (arch, sun). The next 10 processes will have in MPI_INFO_KEY (command, atmos), (maxprocs,10), and (arch, rs600)

Advice to users. The values passed in MPI_INFO_KEY are the values of the arguments passed to the mechanism that started the MPI execution – not the actual value provided. Thus, the value associated with maxprocs is the number of MPI processes requested; it can be larger than the actual number of processes obtained, if the soft option was used. (*End of advice to users.*)

Advice to implementors. Good quality implementations will provide a (key,value) pair for each parameter that can be passed to the command that starts an MPI program. (*End of advice to implementors.*)

MPI_FINALIZE()

```
int MPI_Finalize(void)

MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

{void MPI::Finalize() *(binding deprecated, see Section 15.2)* }

ticket313. This routine cleans up all MPI state. [Each process must call MPI_FINALIZE before it exits. Unless there has been a call to MPI_ABORT, before each process exits process must ensure that all pending nonblocking communications are (locally) complete before calling MPI_FINALIZE. Further, at the instant at which the last process calls MPI_FINALIZE, all pending sends must be matched by a receive, and all pending receives must be matched by a send.

For example, the following program is correct

] If an MPI program terminates normally (i.e., not due to a call to MPI_ABORT or an unrecoverable error) then MPI must be finalized at each MPI process by a call to MPI_FINALIZE on this process.

Before MPI is finalized at an MPI process, the process must locally complete all MPI calls. When the last process calls MPI_FINALIZE, all non-local MPI calls at each process must be matched by MPI calls at the other processes that are needed to complete the relevant operation: For example, for each send, the matching receive has occurred, and for each receive, a marching send has occurred; each collective operation has been invoked at all involved processes, etc.

The call to MPI_FINALIZE does not free objects created by MPI commands – i.e., objects that the user can free using MPI calls.

MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4 on page 362.

The following examples illustrates these rules

Example 8.4 The following code is correct

```
Process 0                Process 1
---------                ---------
MPI_Init();              MPI_Init();
MPI_Send(dest=1);        MPI_Recv(src=0);
MPI_Finalize();          MPI_Finalize();
```

Example 8.5 Without a matching receive, the program is erroneous

```
Process 0                Process 1
-----------              -----------
MPI_Init();              MPI_Init();
MPI_Send (dest=1);
MPI_Finalize();          MPI_Finalize();
```

ticket313.

[deleted in April Since MPI_FINALIZE is a collective call, a correct MPI program will naturally ensure that all participants in pending collective operations have made the call before calling MPI_FINALIZE.

A successful return from a blocking communication operation or from MPI_WAIT or MPI_TEST tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from MPI_REQUEST_FREE with a request handle generated by an MPI_ISEND nullifies the handle but provides no assurance of operation completion. The MPI_ISEND is complete only when it is known by some means that a matching receive has completed. MPI_FINALIZE guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

MPI_FINALIZE guarantees nothing about pending communications that have not been completed (completion is assured only by MPI_WAIT, MPI_TEST, or MPI_REQUEST_FREE combined with some other verification of completion).]

ticket313.

[

Example 8.6 This program is correct HEADER SKIP ENDHEADER

```
rank 0                          rank 1
=====================================================
...                             ...
MPI_Isend();                    MPI_Recv();
MPI_Request_free();             MPI_Barrier();
MPI_Barrier();                  MPI_Finalize();
MPI_Finalize();                 exit();
exit();
```

Example 8.7 This program is erroneous and its behavior is undefined: HEADER SKIP ENDHEADER

```
rank 0                          rank 1
=====================================================
...                             ...
MPI_Isend();                    MPI_Recv();
MPI_Request_free();             MPI_Finalize();
MPI_Finalize();                 exit();
exit();
```

ticket313.]

Example 8.8 This program is correct: The send operation on process 0 is locally complete when MPI_Finalize is called: the local buffer can be reused and no further MPI calls are required on the sender side.

```
   Process 0                   Process 1
   ---------                   ---------
   MPI_Init();                  MPI_Init();
   MPI_Isend();                MPI_Recv();
   MPI_Request_free();         MPI_Barrier();
   MPI_Barrier();              MPI_Finalize();
   MPI_Finalize();
```

Example 8.9 This program is erroneous: The send operation on process 0 is not locally complete when MPI_Finalize is called

```
  Process 0                    Proces 1
  --------                     --------
  MPI_Init();                  MPI_Init();
  MPI_Isend();                 MPI_Recv();
  MPI_Request_free();          MPI_Finalize();
  MPI_Finalize();              exit();
exit();
```

ticket313.

[If no MPI_BUFFER_DETACH occurs between an MPI_BSEND (or other buffered send) and MPI_FINALIZE, the MPI_FINALIZE implicitly supplies the MPI_BUFFER_DETACH.

Example 8.10 This program is correct, and after the MPI_Finalize, it is as if the buffer had been detached. HEADER SKIP ENDHEADER

```
rank 0                         rank 1
=====================================================
...                            ...
buffer = malloc(1000000);      MPI_Recv();
MPI_Buffer_attach();           MPI_Finalize();
MPI_Bsend();                   exit();
MPI_Finalize();
free(buffer);
exit();
```

]

ticket313.

Example 8.11 This program is correct. The attached buffer is a resource allocated by the user, not by MPI; it is availble to the user after MPI is finalized.

```
   Process 0                     Process 1
   ---------                     ---------
   MPI_Init(0;                  MPI_Init();
   buffer = malloc(1000000);    MPI_Recv();
   MPI_Buffer_attach();         MPI_Finalize();
   MPI_Bsend();                 exit();
   MPI_Finalize();
   free(buffer);
   exit();
```

ticket313.

[

Example 8.12 In this example, MPI_Iprobe() must return a FALSE flag. MPI_Test_cancelled() must return a TRUE flag, independent of the relative order of execution of MPI_Cancel() in process 0 and MPI_Finalize() in process 1.

The MPI_Iprobe() call is there to make sure the implementation knows that the "tag1" message exists at the destination, without being able to claim that the user knows about it.

HEADER SKIP ENDHEADER

```
rank 0                          rank 1
=======================================================
MPI_Init();                     MPI_Init();
MPI_Isend(tag1);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Iprobe(tag2);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Finalize();
                                exit();
MPI_Cancel();
MPI_Wait();
MPI_Test_cancelled();
MPI_Finalize();
exit();
```

ticket313.]

Example 8.13 This program is correct. The cancel operation must succeed, since the send cannot complete normally.

```
   Process 0                     Process 1
   ---------                     ---------
   MPI_Issend();                  MPI_Finalize();
   MPI_Cancel();
   MPI_Wait();
   MPI_Finalize();
```

ticket313. [

Advice to implementors. An implementation may need to delay the return from MPI_FINALIZE until all potential future message cancellations have been processed. One possible solution is to place a barrier inside MPI_FINALIZE (*End of advice to implementors.*)

ticket313.]

Advice to implementors.

Even though a process has completed all the communications it initiated, such communication may not yet be completed from the viewpoint of the underlying MPI system. E.g., a blocking send may have returned, even though the data is still buffered at the sender. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication to fail.

The MPI implementation should also complete freeing all objects marked for deletion by MPI calls that freed them.

An implementation may need to delay the return from MPI_FINALIZE on a process even if all communications related to MPI calls by that process have completed; the process may still receive cancel requests for messages it has completed receiving. One possible solution is to place a barrier inside MPI_FINALIZE.

(*End of advice to implementors.*)

ticket313.

Advice to users. If a process continues execution after the call to MPI_FINALIZE then it is recommended that the user explitly free all the objects allocated by MPI calls before the call to MPI_FINALIZE. (*End of advice to users.*)

Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, except for MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED.

ticket313.

[Each process must complete any pending communication it initiated before it calls MPI_FINALIZE. If the call returns, each process may continue local computations, or exit, without participating in further MPI communication with other processes.]

ticket313.

[MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4 on page 362.]

ticket313.

[

Advice to implementors. Even though a process has completed all the communication it initiated, such communication may not yet be completed from the viewpoint of the underlying MPI system. E.g., a blocking send may have completed, even though the data is still buffered at the sender. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication to fail. (*End of advice to implementors.*)

]

Although it is not required that all processes return from MPI_FINALIZE, it is required that at least process 0 in MPI_COMM_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, they may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Example 8.14 The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
    resultfile = fopen("outfile","w");
    dump_results(resultfile);
    fclose(resultfile);
}
exit(0);
```

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. ticket313. [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.]

An MPI library that is not thread compliant must always return provided=MPI_THREAD_SINGLE, even if MPI_INIT_THREAD is called on a multithreaded process. The library should also return correct values for the MPI calls that can be executed before initialization, even if multiple threads have been spawned.

Rationale. Such code is erroneous, but the error cannot be detected until MPI_INIT_THREAD is called. The requirements in the previous paragraph ensure that the error can be properly detected. (*End of rationale.*)

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 15.2)* }

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(). ticket0.

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 15.2)* }

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] MPI_GET_VERSION, MPI_INITIALIZED, or MPI_FINALIZED should be executed by these threads, until MPI_INIT_THREAD is ticket313.