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 15.2)* }

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

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

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 419).

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 ticket311. will be provided by MPI. It can be one of [the four values listed above.]the predefined values for levels of thread support.

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 ticket313. = 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.]

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.

A call to MPI_INIT has the same effect as a call to MPI_INIT_THREAD with a required ticket310. = MPI_THREAD_SINGLE.

In an environment where multiple MPI processes are in the same address space, MPI must be initialized by calling MPI_INIT_THREAD. All MPI processes in the same address space will have the same level of thread support. The level of thread support provided must be at least MPI_THREAD_FUNNELED. If the level of thread support is MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED or MPI_THREAD_MULTIPLE then the association of threads to MPI proceses is controlled by the system and does not change during the lifetime of a thread. At least one (main) thread is associated with each MPI ticket311. process.

Two additional levels of thread support are defined for such an environment:

MPI_THREAD_ATTACH Threads must be explicitly attached to an MPI process, in order to execute MPI calls. The association of a thread to an MPI process does not change during the lifetime of the thread.

MPI_THREAD_REATTACH Threads must be explicitly attached to an MPI process, in order to execute MPI calls. The association of a thread to an MPI process may be changed during execution.

An MPI process may not have any thread attached to it during some of the program execution. The behavior of such a process is the same as a behavior of a process where no thread invokes MPI functions.

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)* }

ticket0. This function can be called by a thread to [find out whether]determine if it is the main
ticket311. thread [(the thread that called MPI_INIT or MPI_INIT_THREAD).] If the MPI process was initialized by a call to MPI_INIT or MPI_INIT_THREAD on that process than the main thread is the thread that performed this call. This thread should call MPI_FINALIZE for this process. If the MPI process was initialized by other means, than the main thread is designated by the runtime. This thread must continue execution until the MPI process is finalized.

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
ticket0. [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
ticket313. 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 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.*)

ticket310.

12.5 Multiple MPI Processes Within the Same Address Space

When multiple MPI processes are in the same address space, it is not immediately obvious whether a thread belongs to an MPI process and, if so, which.

A thread can find whether it belongs to an MPI process by calling MPI_INITIALIZED; the call will return `true` if such is the case.

A thread that belongs to an MPI process can find its rank with MPI_COMM_WORLD by calling MPI_COMM_RANK. It can find how many MPI processes are running in the same address space by extracting the value associated with the key asp in the info object MPI_INFO_ENV. It can establish MPI communication with these processes by calling MPI_COMM_SPLIT_TYPE with a type argument MPI_COMM_TYPE_ADDRESS_SPACE.MPI processes that belong to the same address space have contiguous ranks in
ticket311. MPI_COMM_WORLD.

If the level of thread support is MPI_THREAD_ATTACH or MPI_THREAD_REATTACH then a thread needs to be attached to an MPI process by calling MPI_THREAD_ATTACH before it can execute MPI calls (other than those allowed before MPI is initialized).

MPI_THREAD_ATTACH(index)

IN	index	index of MPI process within address space (integer)

```
int MPI_Thread_attach(int index)
```

```
MPI_thread_attach(index, IERROR)
    INTEGER INDEX, IERROR
```

The thread performing the call will be attached to the MPI process specified by the index argument. MPI processes within an address space are numbered from 0 to asp-1. The call is erroneous of index is out of range or the level of thread support is MPI_THREAD_MULTIPLE or less.

If the level of thread support is MPI_THREAD_ATTACH then a thread can attach to an MPI process only once; the thread belongs to the process it attached to until it terminates. If the level of thread support is MPI_THREAD_REATTACH then a thread can invoke the function MPI_THREAD_ATTACH even if it already belongs to an MPI process; the invoking thread will detach from its current MPI process and attach to the one specified by index. ticket310.

12.6 Interoperability

We present in this section several examples that illustrate how MPI can be used in conjunction with OpenMP, a Pthread library or a PGAS program, both with one MPI process per address space, and with multiple processes. We use the same running example: A library, such as DPLASMA [15] that executes a static dataflow graph. The graph tasks are allocated statically to compute nodes and are scheduled dynamically when their inputs are available.

12.6.1 OpenMP

Example 12.3 The following example shows an OpenMP program running within an MPI process. One task acts as the communication master, receiving messages and dispatching computation slave tasks to work on these messages. The tasks are untied, so could be migrated from one thread to another. The call to MPI_Init_thread is not necessary, but harmless if MPI is already initialized.

```
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
...
int main() {
  ...
  MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, *provided);
  if(provided != MPI_THREAD_MULTIPLE) MPI_Abort(MPI_COMM_WORLD,0);
  while (notdone) {
```

```
    item = (Work_item*) malloc(sizeof(Work_item));
    MPI_Recv(item, 1, Work_packet_type, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
    Make_runnable_list(item, rlist);
    for(p = rlist; p != NULL; p = p.next)
      #pragma omp task
      {
        compute(p);
        Make_output_list(p, olist);
        for (q=olist; q != null; q = q.next)
          #pragma omp task
             MPI_Send(q.item, 1, Work_packet_type, q.dest, 0,
                      MPI_COMM_WORLD);
      }
  }
  MPI_Finalize();
}
```

Example 12.4 This example is similar to the previous one, except that the code uses multiple communication master threads, each with a different rank within MPI_COMM_WORLD.

```
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
...
int main() {
  ...
  MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, *provided);
  if(provided != MPI_THREAD_MULTIPLE) MPI_Abort(MPI_COMM_WORLD,0);
  MPI_Info_get(MPI_INFO_ENV, "MPI_ENV_ASP", vlen, val, *flag);
  asp = atoi(val);
  #pragma omp parallel
  {
    if(omp_get_numthreads() < asp + MINWORKERS) exit(0);
    if ((MPI_Initialized(init), init) && (MPI_Is_thread_main(main),
                                          main)) {
      /* communication master thread */
      while (notdone) {
        ...
        item = (Work_item*) malloc(sizeof(Work_item));
        MPI_Recv(item, 1, Work_packet_type, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
                         MPI_STATUS_IGNORE);
        Make_runnable_list(item, rlist);
        for(p = rlist; p != NULL; p = p.next)
        #pragma omp task
```

```
      {
        compute(p);
        Make_output_list(p, olist);
        for (q=olist; q != null; q = q.next)
        #pragma omp task
          MPI_Send(q.item, 1, Work_packet_type, q.dest, 0,
                   MPI_COMM_WORLD);
      }
    }
  }
}
MPI_Finalize();
}
```

Example 12.5 In this example, we use dedicated receiver threads, dedicated sender threads, and dedicated worker threads.

```
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
...
int main() {
  ...
  MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, *provided);
  if(provided < MPI_THREAD_ATTACH) MPI_Abort(MPI_COMM_WORLD,0);
  MPI_Info_get(MPI_INFO_ENV, "MPI_ENV_ASP", vlen, val, *flag);
  asp = atoi(val);
  #pragma omp parallel
  {
    if(omp_get_numthreads() < 2*asp + MINWORKERS) exit(0);
    if (omp_get_threadnum()<asp)
       /* receiver thread */
       while (notdone) {
        item = (Work_item*) malloc(sizeof(Work_item));
        MPI_Recv(item, 1, Work_packet_type, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
                          MPI_STATUS_IGNORE);
        Make_runnable_list(item, rlist);
        #pragma omp critical (workqueue)
           Engueue(workqueue, rlist);
       ]
      else if (omp_get_threadnum < 2*asp)
        /* sender thread */
        while (notdone) {
          #pragma omp critical (sendqueue)
             Dequeue(sendqueue, item);
          if (item != NULL)
```

```
            MPI_Send(q.item, 1, Work_packet_type, q.dest, 0,
                        MPI_COMM_WORLD);
        }
        else
          /* worker thread */
          while (notdone) {
            #pragma omp critical (workqueue)
               Dequeue(workqueue, item);
               if (item != NULL) {
                 Compute(item, slist);
                 #pragam omp critical (sendqueue)
                    Enqueue(sendqueue, slist);
                }
          }
  }
  MPI_Finalize();
}
```

12.6.2 The Pthread Library

Example 12.6 We show the same code of the previous examples, written using the Pthread library.

```
#include <mpi.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

pthread_t thread[NUM_THREADS];
pthread_attr_t attr;
int t;
void *status;
char notdone = 1;
Queue queue;

int main() {
  MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, *provided)
  if(provided < MPI_THREAD_ATTACH) MPI_Abort(MPI_COMM_WORLD,0);

  /* Initialize and set thread detached attribute */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  /* start compute threads */
  for (t=0; t<asp; t++)
    pthread_create(&thread[t], &attr, Receiver, NULL);
  for (t=0; t< asp; t++)
```

```
    pthread_create(&thread[t], &attr, Sender, NULL);
  for (t=0; t < NUMWORKERS; t++)
    pthread_create(&thread[t], &attr, Worker, NULL);
  /* wait for all compute slaves */
  for(t=0; t<NUM_THREADS; t++)
    pthread_join(thread[t], &status);
  MPI_Finalize();
  pthread_exit(NULL);
}
```

12.6.3 PGAS Languages

Example 12.7 UPC code running with one UPC thread per address space and one MPI process per UPC thread. (UPC_THREAD_PER_PROC=1) .

```
#include <upc.h>
#include <mpi.h>
...
MPI_Init()
... /*UPC code */
... /* Library using MPI can be invoked */
PDEGESV(...)
...
MPI_Finalize();
```

Example 12.8 UPC code running with multiplee UPC threads per address space and one MPI process per address space. (UPC_THREAD_PER_PROC ¿ 1) .

```
#include <upc.h>
#include <mpi.h>
...
MPI_Init_thread(MPI_THREAD_FUNNELED, *provided);
if (provided < MPI_THREAD_FUNNELED) exit(0);
... /*UPC code */
... /* Library using MPI can be invoked */
if (MPI_Is_main_thread(*main), main) PDEGESV(...);
...

MPI_Finalize();
```

ticket311.

Example 12.9 UPC code running with multiplee UPC threads per address space and one MPI process per UPC thread.