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 421).

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 predefined values for levels of thread support. ticket311.

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.]

ticket313.

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.

ticket310.

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 must request the same level of thread support, and all are provided the same level of thread support, which must be at least MPI_THREAD_FUNNELED. The association of threads to MPI proceses is controlled by the system and does not change during the lifetime of a thread. The main thread of an MPI process is the thread associated with this process that invoked the MPI initialization call, or a thread chosen by the system, if no such call occurred on that process.

ticket311.

An additional levels of thread support is defined for such an environment:

MPI_THREAD_MOBILE Threads can move from one MPI process to another within the same address space.

This thread support level is described in Section 12.5. We have MPI_THREAD_MULTIPLE < MPI_THREAD_MOBILE.

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.

```
    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)].

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 establish MPI communication with the other MPI processes in the same address space by calling MPI_COMM_SPLIT_TYPE with a type argument
ticket311. MPI_COMM_TYPE_ADDRESS_SPACE.

If the level of thread support is MPI_THREAD_MOBILE and multiple MPI processes share the same address space, then threads can change the MPI process they are associated with by calling MPI_THREAD_ATTACH.

MPI_THREAD_ATTACH(int rank, MPI_Comm comm)

IN	rank	rank of MPI process the thread attaches to (integer)
IN	comm	communicator (handle)

```
int MPI_Thread_attach(int rank, MPI_Comm comm))
```

```
MPI_THREAD_ATTACH(RANK, COMM,IERROR)
    INTEGER RANK, COMM, IERROR
```

The thread performing the call will be detached from the MPI process it currently belongs to and attached to the MPI process with rank rank in the communicator comm. The call is erroneous if the specified MPI process is not in the same address space as the calling thread. The call has no effect if the tread attaches to the MPI process it is currently attached to.

Every thread is associated to an MPI process once MPI is initialized. The association mechanism is implementation dependent. The function MPI_THREAD_ATTACH allows one to overide this default.

When a main thread detaches from its current MPI process, it ceases to be main. A new thread that is currently attached to the process is selected to become main.

An MPI process may have no attached threads, for a period of time. The behavior of such process is as if no MPI call is executed at that process for that period of time.

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.

```
#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. The program uses `asp` communication threads and at least `MINWORKERS` computation threads.

```
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#define MINWORKERS 10
...
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();
}
```

ticket311.

Example 12.5 In this example, we use `size` dedicated receiver threads, `size` dedicated sender threads, and at least `MINWORKERS` dedicated worker threads. We assume in this example that the master OpenMP thread belongs to the first MPI process in the address space.

```
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#define MINWORKERS 10
...
int main() {
  ...
  MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, *provided);
  if(provided < MPI_THREAD_MOBILE) MPI_Abort(MPI_COMM_WORLD,0);
  #pragma omp parallel
  {
    MPI_Comm_split_type(MPI_COMM_WORLD,
                        MPI_COMM_TYPE_ADDRESS_SPACE,0,NULL,&lcomm);
    MPI_Comm_size(lcomm, &size);
    if(omp_get_numthreads() < 2*size + MINWORKERS) exit(0);
    if ((id =omp_get_threadnum())<size) {
       /* receiver thread */
       MPI_Thread_attach(id, lcomm);
       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 (id < 2*size) {
        /* sender thread */
        MPI_Thread_attach(id-size, lcomm);
        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 as in 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_MOBILE, &provided)
  if(provided < MPI_THREAD_MOBILE) 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 multiple 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 multiple UPC threads per address space and one MPI process per UPC thread.

```
#include <upc.h>
#include <mpi.h>
...
MPI_Init_thread(MPI_THREAD_ATTACH, &provided);
if (provided < MPI_THREAD_MOBILE) exit(0);
MPI_Thread_attach(MYTHRAD);
... /*UPC code */
... /* Library using MPI can be invoked */
PDEGESV(...);
...

MPI_Finalize();
```

Advice to users. When multiple C/C++ threads run in the same address space, they share one copy of the static variables in the program. If one wants a library using MPI behave identically, whether it runs with one or with multiple MPI processes in the same address space, then such variables must be made thread-local (e.g., with the __thread specifier in gcc). The code must be thread safe. (*End of advice to users.*)