Advice to implementors. Implementors should document which routines are implemented as macros. (*End of advice to implementors.*)

Advice to users. If these routines are implemented as macros, they will not work with the MPI profiling interface. (*End of advice to users.*)

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. ticket310. [Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.]Multiple MPI processes can execute within a single address space. ticket310.

Advice to implementors. An implementation that supports multiple MPI processes within the same address space must maintain the association of user-created threads to MPI processes. For example, it can store, in a thread-private location, a pointer to the data structure associated with this MPI process. (*End of advice to implementors.*)

This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of communication, I/O, and process management is not specified. Unless otherwise stated in the specification of the standard, MPI places no requirements on the result of its interaction with external mechanisms that provide similar or equivalent functionality. This includes, but is not limited to, interactions with external mechanisms for process control, shared and remote memory access, file system access and control, interprocess communication, process signaling, and terminal I/O. High quality implementations should strive to make the results of such interactions intuitive to users, and attempt to document restrictions where deemed necessary.

Advice to implementors. Implementations that support such additional mechanisms for functionality supported within MPI are expected to document how these interact with MPI. (*End of advice to implementors.*)

The interaction of MPI and threads is defined in Section 12.4.

2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.

Of course, MPI programs may still be erroneous. A **program error** can occur when an MPI call is made with an incorrect argument (non-existing destination in a send operation, buffer too small in a receive operation, etc.). This type of error would occur in any

```
        /* Determine my color */
        MPI_Comm_rank ( multiple_server_comm, &rank );
        color = rank % num_servers;

        /* Split the intercommunicator */
        MPI_Comm_split ( multiple_server_comm, color, rank,
                         &single_server_comm );
```

The following is the corresponding server code:

```
        /* Server code */
        MPI_Comm  multiple_client_comm;
        MPI_Comm  single_server_comm;
        int       rank;

        /* Create intercommunicator with clients and servers:
           multiple_client_comm */
        ...

        /* Split the intercommunicator for a single server per group
           of clients */
        MPI_Comm_rank ( multiple_client_comm, &rank );
        MPI_Comm_split ( multiple_client_comm, rank, 0,
                         &single_server_comm );
```

MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)

IN	comm	communicator (handle)
IN	split_type	type of processes to be grouped together (integer)
IN	key	control of rank assignment (integer)
IN	info	info argument (handle)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info
             info, MPI_Comm *newcomm)
```

```
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
    INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
```

This function partitions the group associated with comm into disjoint subgroups, based on the type specified by split_type. Each subgroup contains all processes of the same type. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. This is a collective call; all processes must provide the same split_type, but each process is permitted to provide different values for key. An exception to this rule is that a process may supply the type value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL.

The following type is predefined by MPI:

MPI_COMM_TYPE_SHARED — this type splits the communicator into subcommunicators, each of which can create a shared memory region.

ticket310.

MPI_COMM_TYPE_ADDRESS_SPACE — this type splits the communicator into subcommunicators, in which all processes share an address space.

Advice to implementors. Implementations can define their own types, or use the info argument, to assist in creating communicators that help expose platform-specific information to the application. (*End of advice to implementors.*)

ticket310.

Example 6.3 The following example illustrates how MPI processes within the same address space can share global data.

```
int *buf;

int main(int argc, char **argv)
{
  int me, size;
  MPI_Comm comm_address_space;

  MPI_Init(&argc, &argv);

  MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_ADDRESS_SPACE,
                      0, MPI_INFO_NULL, &comm_address_space);

  MPI_Comm_rank(comm_address_space, &me);
  if (me == 0) {
     buf = (int *) malloc(10000);
     /* initialize buffer */
  }
  MPI_Barrier(comm_address_space);

  /* All processes within the address space share the same 'buf' */
  x = buf[0];
  y = buf[1];

  MPI_Comm_free(&comm_address_space);
  MPI_Finalize();
}
```

6.4.3 Communicator Destructors

MPI_COMM_FREE(comm)

INOUT	comm	communicator to be destroyed (handle)

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 must contain exactly one call to an MPI initialization routine: MPI_INIT or MPI_INIT_THREAD.] Each MPI process must call an MPI initialization routine, MPI_INIT or MPI_INIT_THREAD, exactly once. 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.

ticket310.

If multiple MPI processes run in the same address space, then the first call to an initialization routine by a thread will initialize all the MPI processes in this address space. Subsequent initialization calls executed by other MPI processes are not erroneous, but have no effect.

Rationale. In an environment such as OpenMP, the execution starts at one thread. Code is simplified if MPI is initialized by that thread, even if the openMP program contains multiple MPI processes.

On the other hand, in an environment as provided by UPC or CAF, we may want to associate one MPI process with each UPC thread or CAF image, even when multiple UPC threads (resp. CAF images are in the same address space. In such a case, it is preferable to have an initialization call on each MPI process. The current design supports both models. (*End of rationale.*)

Advice to implementors. An implementation should check whether MPI is already initialized before executing the initialization code. (*End of advice to implementors.*)

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. ticket313. [and C++. In C++, there is an alternative ticket313. binding for MPI::Init that does not have these arguments at all.

> *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

ticket310. file Value is the name of a file in which additional information is specified.

asp Number of MPI processes in local address space.

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 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.*)

ticket310.]

If multiple MPI processes run in the same address space then MPI_FINALIZE must be invoked on each MPI process that invoked MPI_INIT() or MPI_INIT_THREAD().

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);
```

MPI_INITIALIZED(flag)

OUT	flag	Flag is true if MPI_INIT has been called and false otherwise.

```
int MPI_Initialized(int *flag)
```

```
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

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

This routine may be used to determine whether MPI_INIT has been called. MPI_INITIALIZED returns `true` if the calling process has called MPI_INIT. Whether MPI_FINALIZE has been called does not affect the behavior of MPI_INITIALIZED. It is one of the few routines that may be called before MPI_INIT is called.

suite script that runs hundreds of programs can be a portable script if it is written using such a standard starup mechanism. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead of `mpirun` MPI specifies `mpiexec`.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an `mpiexec` startup command and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called `mpiexec`, it must be of the form described below.

It is suggested that

```
mpiexec -n <numprocs> <program>
```

be at least one way to start `<program>` with an initial MPI_COMM_WORLD whose group contains `<numprocs>` processes. Other arguments to `mpiexec` may be implementation-dependent.

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that `mpiexec` be able to be viewed as a command-line version of MPI_COMM_SPAWN (See Section 10.3.4).

Analogous to MPI_COMM_SPAWN, we have ticket310.

[

```
    mpiexec -n    <maxprocs>
           -soft  <        >
           -host  <        >
           -arch  <        >
           -wdir  <        >
           -path  <        >
           -file  <        >
            ...
           <command line>
```

] ticket310.

```
    mpiexec -n    <maxprocs>
           -soft  <        >
           -host  <        >
           -arch  <        >
           -wdir  <        >
           -path  <        >
           -file  <        >
           -asp   <        >
            ...
           <command line>
```

for the case where a single command line for the application program and its arguments will suffice. See Section 10.3.4 for the meanings of these arguments. For the case corresponding to MPI_COMM_SPAWN_MULTIPLE there are two possible formats:

Form A:

```
    mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }
```

As with MPI_COMM_SPAWN, all the arguments are optional. (Even the `-n x` argument is optional; the default is implementation dependent. It might be `1`, it might be taken from an environment variable, or it might be specified at compile time.) The names and meanings of the arguments are taken from the keys in the `info` argument to MPI_COMM_SPAWN. There may be other, implementation-dependent arguments as well.

Note that Form A, though convenient to type, prevents colons from being program arguments. Therefore an alternate, file-based form is allowed:

Form B:

```
    mpiexec -configfile <filename>
```

where the lines of <`filename`> are of the form separated by the colons in Form A. Lines beginning with '`#`' are comments, and lines may be continued by terminating the partial line with '\'.

Example 8.15 Start 16 instances of `myprog` on the current or default machine:

```
    mpiexec -n 16 myprog
```

Example 8.16 Start 10 processes on the machine called `ferrari`:

```
    mpiexec -n 10 -host ferrari myprog
```

Example 8.17 Start three copies of the same program with different command-line arguments:

```
    mpiexec myprog infile1 : myprog infile2 : myprog infile3
```

Example 8.18 Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000's:

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

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

Example 8.19 Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000's (Form B):

```
    mpiexec -configfile myfile
```

where `myfile` contains

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

ticket310.

Example 8.20 Start 12 MPI processes of the `foo` program, with 4 MPI processes in each address space:

```
    mpiexec -asp 4 -n 12 foo
```

(*End of advice to implementors.*)

wdir Value is the name of a directory on a machine on which the spawned process(es) execute(s). This directory is made the working directory of the executing process(es). The format of the directory name is determined by the implementation.

path Value is a directory or set of directories where the implementation should look for the executable. The format of path is determined by the implementation.

file Value is the name of a file in which additional information is specified. The format of the filename and internal format of the file are determined by the implementation.

soft Value specifies a set of numbers which are allowed values for the number of processes that MPI_COMM_SPAWN (et al.) may create. The format of the value is a comma-separated list of Fortran-90 triplets each of which specifies a set of integers and which together specify the set formed by the union of these sets. Negative values in this set and values greater than maxprocs are ignored. MPI will spawn the largest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant.

By Fortran-90 triplets, we mean:

1. `a` means a
2. `a:b` means $a, a+1, a+2, \ldots, b$
3. `a:b:c` means $a, a+c, a+2c, \ldots, a+ck$, where for $c > 0$, k is the largest integer for which $a+ck \leq b$ and for $c < 0$, k is the largest integer for which $a+ck \geq b$. If $b > a$ then c must be positive. If $b < a$ then c must be negative.

Examples:

1. `a:b` gives a range between a and b
2. `0:N` gives full "soft" functionality
3. `1,2,4,8,16,32,64,128,256,512,1024,2048,4096` allows power-of-two number of processes.
4. `2:10000:2` allows even number of processes.
5. `2:10:2,7` allows 2, 4, 6, 7, 8, or 10 processes.

ticket310.

asp Value is the number of MPI processes to use within an address space.

10.3.5 Spawn Example

ticket0. Manager-worker Example [,] Using MPI_COMM_SPAWN.

```
/* manager */
#include "mpi.h"
int main(int argc, char *argv[])
{
   int world_size, universe_size, *universe_sizep, flag;
   MPI_Comm everyone;           /* intercommunicator */
   char worker_program[100];

   MPI_Init(&argc, &argv);
```

Chapter 12

External Interfaces

12.1 Introduction

This chapter begins with calls used to create **generalized requests**, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. This can be used to layer new functionality on top of MPI. Next, Section 12.3 deals with setting the information found in status. [This is]This functionality is needed for generalized requests. ticket0.

The chapter continues, in Section 12.4, with a discussion of how threads are to be handled in MPI. Although thread compliance is not required, the standard specifies how threads are to work if they are provided. ticket310.

Section 12.5 describes how threads are associated to MPI processes in an environment where multiple such processes can run in the same address space. Finally Section 12.6 provides examples for the interoperability of MPI with OpenMP and with PGAS languages.

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

12.4 MPI and Threads

This section specifies the interaction between MPI calls and threads. The section lists minimal requirements for **thread compliant** MPI implementations and defines functions that can be used for initializing the thread environment. MPI may be implemented in environments where threads are not supported or perform poorly. Therefore, it is not required that all MPI implementations fulfill all the requirements specified in this section.

This section generally assumes a thread package similar to POSIX threads [34], but the syntax and semantics of thread calls are not specified here — these are beyond the scope of this document.

12.4.1 General

ticket310.

In a thread-compliant implementation, an MPI process 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

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”]MPI processes are single-threaded). (*End of rationale.*)

ticket0. 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 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. 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.

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++

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 MPI_COMM_TYPE_ADDRESS_SPACE.

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();
}
```

12.6.2 PGAS Languages

Example 12.5 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 12.6 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();
```

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.*)

Datatypes for reduction functions (C and C++)

C type: `MPI_Datatype` Fortran type: `INTEGER`	C++ type: `MPI::Datatype`
MPI_FLOAT_INT	MPI::FLOAT_INT
MPI_DOUBLE_INT	MPI::DOUBLE_INT
MPI_LONG_INT	MPI::LONG_INT
MPI_2INT	MPI::TWOINT
MPI_SHORT_INT	MPI::SHORT_INT
MPI_LONG_DOUBLE_INT	MPI::LONG_DOUBLE_INT

Datatypes for reduction functions (Fortran)

C type: `MPI_Datatype` Fortran type: `INTEGER`	C++ type: `MPI::Datatype`
MPI_2REAL	MPI::TWOREAL
MPI_2DOUBLE_PRECISION	MPI::TWODOUBLE_PRECISION
MPI_2INTEGER	MPI::TWOINTEGER

Special datatypes for constructing derived datatypes

C type: `MPI_Datatype` Fortran type: `INTEGER`	C++ type: `MPI::Datatype`
MPI_UB	MPI::UB
MPI_LB	MPI::LB

Reserved communicators

C type: `MPI_Comm` Fortran type: `INTEGER`	C++ type: `MPI::Intracomm`
MPI_COMM_WORLD	MPI::COMM_WORLD
MPI_COMM_SELF	MPI::COMM_SELF

ticket310.

Communicator split type constants

C type: const int (or unnamed enum) Fortran type: `INTEGER`
MPI_COMM_TYPE_ADDRESS_SPACE

Results of communicator and group comparisons

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_IDENT	MPI::IDENT
MPI_CONGRUENT	MPI::CONGRUENT
MPI_SIMILAR	MPI::SIMILAR
MPI_UNEQUAL	MPI::UNEQUAL