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 );
```

ticket287.

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)* }

ticket310.

[All MPI processes 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.]

ticket310.

An MPI process is initialized by a call to an initialization routine, MPI_INIT or MPI_INIT_THREAD on that process. Subsequent calls by the process to any initialization routine have no effect. The only MPI functions that may be invoked by a process before the MPI initialization is complete are MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED.

If multiple MPI processes run in the same address space, then the first call to an initialization routine will initialize all the MPI processes in this address space. All MPI processes running within the same address space will have the same level of thread support.

Implementations may also choose to initialize MPI before the user code starts executing.

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

When a language such as UPC or CAF is used, then its parallel environment is initialized before the user code starts running. If MPI is used in such an environment, it may be advantageous to initialize MPI before the user code starts running, at the same time that UPC or CAF are initialized.

We maintain compatibility with current codesby allowing the MPI initialization routine to be called on each MPI process, even if several processes run in the same address space. (*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 binding for MPI::Init that does not have these arguments at all. 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:

ticket310.

MPI_ENV_N	Total number of MPI processes.
MPI_ENV_ASP	Number of MPI processes in local address space.
MPI_ENV_THREAD_LEVEL	Level of thread support.
MPI_ENV_HOST	Hostname.
MPI_ENV_ARCH	Architecture name.
MPI_ENV_WDIR	Working directory of the MPI process
MPI_ENV_FILE	Value is the name of a file in which additional information is specified.

Implementations may provide additional, implementation specific, keys.

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 unrecovered error) then MPI must be finalized at each MPI process. MPI is finalized at a 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 and free all objects created by MPI calls on that process.

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 each send, the matching receive has occurred, each collective operation has been invoked at all involved processes, etc.

The following examples illustrates these rules

Example 8.3 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 8.4 Without a matching receive, the program is erroneous

```
Process 0                Process 1
-----------              -----------
MPI_Init();              MPI_Init();
MPI_Send (dest=1);
MPI_Finalize();          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 8.5 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 8.6 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 8.7 This program is erroneous: The MPI_Isend call is not guaranteed to be complete even if the send request was freed and the matching receive completed; the call MPI_Finalize cannot be used to complete the call.

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

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 8.8 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 8.9 This program is erroneous; the program must call MPI_Buffer_detach before the call to MPI_FINALIZE.

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

[ticket313.

Example 8.10 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 8.11 This program is correct. The cancel operation must succeed, since the send cannot complete normally.

```
Process 0                     Process 1
---------                     ---------
MPI_Isend();                  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. 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.*)

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

ticket310. ticket310. 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.]

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.

ticket310.

Advice to implementors. An implementation should check whether MPI is already finalized, before executing the finalization code.

High-quality MPI implementations will free MPI resources not freed by the user before the finalize call, when MPI_FINALIZE executes. (*End of advice to implementors.*)

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 than no MPI process in this address space may call MPI_FINALIZE before all processes in this address space are ready to call MPI_FINALIZE. The first call to MPI_FINALIZE in this address space will finalize MPI for all MPI processes in that address space.

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 8.12 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.

MPI_ABORT(comm, errorcode)

IN	comm	communicator of tasks to abort
IN	errorcode	error code to return to invoking environment

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

{`void MPI::Comm::Abort(int errorcode)` *(binding deprecated, see Section 15.2)* }

This routine makes a "best attempt" to abort all tasks in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a `return errorcode` from the main program.

It may not be possible for an MPI implementation to abort only the processes represented by comm if this is a subset of the processes. In this case, the MPI implementation should attempt to abort all the connected processes but should not abort any unconnected processes. If no processes were spawned, accepted or connected then this has the effect of aborting all the processes associated with MPI_COMM_WORLD.

> *Rationale.* The communicator argument is provided to allow for future extensions of MPI to environments with, for example, dynamic process management. In particular, it allows but does not require an MPI implementation to abort a subset of MPI_COMM_WORLD. (*End of rationale.*)

> *Advice to users.* Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., `mpiexec`), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

> *Advice to implementors.* Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. `mpiexec` or singleton init). (*End of advice to implementors.*)

8.7.1 Allowing User Functions at Process Termination

There are times in which it would be convenient to have actions happen when an MPI process finishes. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that being terminated in the case of dynamically created processes) is finished. This can be accomplished in MPI by attaching an attribute to MPI_COMM_SELF with a callback function. When MPI_FINALIZE is called, it will first execute the equivalent of an MPI_COMM_FREE on MPI_COMM_SELF. This will cause the delete callback function to be executed on all keys associated with MPI_COMM_SELF, in the reverse order that they were set on MPI_COMM_SELF. If no key has been attached to MPI_COMM_SELF, then no callback is invoked. The "freeing" of MPI_COMM_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.

> *Advice to implementors.* Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. Implementations that use the attribute delete callback on MPI_COMM_SELF internally should register their internal callbacks before returning from MPI_INIT / MPI_INIT_THREAD, so that libraries or applications will not have portions of the MPI implementation shut down before the application-level callbacks are made. (*End of advice to implementors.*)

8.7.2 Determining Whether MPI Has Finished

One of the goals of MPI was to allow for layered libraries. In order for a library to do this cleanly, it needs to know if MPI is active. In MPI the function MPI_INITIALIZED was provided to tell if MPI had been initialized. The problem arises in knowing if MPI has been finalized. Once MPI has been finalized it is no longer active and cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this the following function is needed:

MPI_FINALIZED(flag)

OUT	flag	true if MPI was finalized (logical)

```
int MPI_Finalized(int *flag)
```

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

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

This routine returns `true` if MPI_FINALIZE has completed. It is legal to call MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE.

> *Advice to users.* MPI is "active" and it is thus safe to call MPI functions if MPI_INIT *has* completed and MPI_FINALIZE *has not* completed. If a library has no other way of knowing whether MPI is active or not, then it can use MPI_INITIALIZED and MPI_FINALIZED to determine this. For example, MPI is "active" in callback functions that are invoked during MPI_FINALIZE. (*End of advice to users.*)

8.8 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

```
    mpirun <mpirun arguments> <program> <program arguments>
```

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the MPI program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation 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).

ticket310. Analogous to MPI_COMM_SPAWN, we have

[

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

```
mpiexec -n 16 myprog
```

Example 8.14 Start 10 processes on the machine called `ferrari`:

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

Example 8.15 Start three copies of the same program with different command-line arguments:

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

Example 8.16 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 8.17 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 8.18 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 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

ticket310.

[

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.*)

ticket0.
> *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.*)

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.

ticket0.
ticket0.
> *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."]"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 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 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. 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.

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 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 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++ 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 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.

ticket313.

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 MPI_COMM_WORLD.

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();
```

ticket310.

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