```
        /* 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.

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.*)

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)

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

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>
```

```
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 [33], 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.*)
>
> *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.*)

ticket0.

ticket0.

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

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 predefined predefined values for levels of thread support. ticket310.

The level(s) of thread support that can be provided by MPI_INIT_THREAD will depend on the implementation, and may depend on information provided by the user before the program started to execute (e.g., with arguments to `mpiexec`). If possible, the call will return provided = required. Failing this, the call will return the least supported level such that provided > required (thus providing a stronger level of support than required by the user). Finally, if the user requirement cannot be satisfied, then the call will return in provided the highest supported level.

A **thread compliant** MPI implementation will be able to return provided = MPI_THREAD_MULTIPLE. Such an implementation may always return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required. [At the other extreme, an MPI library that is not thread compliant may always return provided = MPI_THREAD_SINGLE, irrespective of the value of required.]

An MPI library that is not thread compliant must always return provided=MPI_THREAD_SINGLE, even if MPI_THREAD_INIT 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.

In an environment where multiple MPI processes are in the same address space, MPI must be initialized by calling MPI_THREAD_INIT. 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.

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

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

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

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

Vendors may provide (implementation dependent) means to specify the level(s) of thread support available when the MPI program is started, e.g., with arguments to `mpiexec`. This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for example, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is available. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required; a call to MPI_INIT will also initialize the MPI thread support level to MPI_THREAD_MULTIPLE. Suppose, on the other hand, that an MPI program has been started so that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD will return provided = required; on the other hand, a call to MPI_INIT will initialize the MPI thread support level to MPI_THREAD_SINGLE.

```
    INTEGER IERROR
```

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

ticket0. This function can be called by a thread to [find out whether]determine if it is the main
ticket310. thread [(the thread that called MPI_INIT or MPI_INIT_THREAD).] If the MPI process was initialized by a call to MPI_INIT or MPI_INIT_THREAD on that process than the main thread is the thread that performed this call. This thread should call MPI_FINALIZE for this process. If the MPI process was initialized by other means, then main thread is designated by the runtime. This thread must continue execution until the MPI process is finalized. All routines listed in this section must be supported by all MPI implementations.

Rationale. MPI libraries are required to provide these calls even if they do not support threads, so that portable code that contains invocations to these functions
ticket0. [be able to]can link correctly. MPI_INIT continues to be supported so as to provide compatibility with current MPI codes. (*End of rationale.*)

Advice to users. It is possible to spawn threads before MPI is initialized, but no
ticket313. MPI call other than [MPI_INITIALIZED] MPI_GET_VERSION, MPI_INITIALIZED, or MPI_FINALIZED should be executed by these threads, until MPI_INIT_THREAD is invoked by one thread (which, thereby, becomes the main thread). In particular, it is possible to enter the MPI execution with a multi-threaded process.

The level of thread support provided is a global property of the MPI process that can be specified only once, when MPI is initialized on that process (or before). Portable third party libraries have to be written so as to accommodate any provided level of thread support. Otherwise, their usage will be restricted to specific level(s) of thread support. If such a library can run only with specific level(s) of thread support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be used to check whether the user initialized MPI to the correct level of thread support and, if not, raise an exception. (*End of advice to users.*)

ticket310.

12.5 Multiple MPI Processes Within the Same Address Space

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

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

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

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

MPI_THREAD_ATTACH(index)

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

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

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

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

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

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 cide, such as DPLASMA [**?**] 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, "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, "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(0;
  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();
```

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

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

MPI_Finalize();
```

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