

Chapter 10

Process Creation and Management

10.1 Introduction

MPI is primarily concerned with communication rather than process or resource management. However, it is necessary to address these issues to some degree in order to define a useful framework for communication. This chapter presents a set of MPI interfaces that allows for a variety of approaches to process management while placing minimal restrictions on the execution environment.

The MPI model for process creation allows both the creation of an initial set of processes related by their membership in a common MPI_COMM_WORLD and the creation and management of processes after an MPI application has been started. A major impetus for the latter form of process creation comes from the PVM [24] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

The MPI Forum decided not to address resource control because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers. Resource control can encompass a wide range of abilities, including adding and deleting nodes from a virtual parallel machine, reserving and scheduling resources, managing compute partitions of an MPP, and returning information about available resources. MPI assumes that resource control is provided externally — probably by computer vendors, in the case of tightly coupled systems, or by a third party software package when the environment is a cluster of workstations.

The reasons for including process management in MPI are both technical and practical. Important classes of message-passing applications require process control. These include task farms, serial applications with parallel modules, and problems that require a run-time assessment of the number and type of processes that should be started. On the practical side, users of workstation clusters who are migrating from PVM to MPI may be accustomed to using PVM's capabilities for process and resource management. The lack of these features would be a practical stumbling block to migration.

The following goals are central to the design of MPI process management:

- The MPI process model must apply to the vast majority of current parallel environments. These include everything from tightly integrated MPPs to heterogeneous networks of workstations.
- MPI must not take over operating system responsibilities. It should instead provide a

clean interface between an application and system software.

- MPI must guarantee communication determinism in the presense of dynamic processes, i.e., dynamic process management must not introduce unavoidable race conditions.
- MPI must not contain features that compromise performance.

The process management model addresses these issues in two ways. First, MPI remains primarily a communication library. It does not manage the parallel environment in which a parallel program executes, though it provides a minimal interface between an application and external resource and process managers.

Second, MPI maintains a consistent concept of a communicator, regardless of how its members came into existence. A communicator is never changed once created, and it is always created using deterministic collective operations.

10.2 The Dynamic Process Model

The dynamic process model allows for the creation and cooperative termination of processes after an MPI application has started. It provides a mechanism to establish communication between the newly created processes and the existing MPI application. It also provides a mechanism to establish communication between two existing MPI applications, even when one did not “start” the other.

10.2.1 Starting Processes

MPI applications may start new processes through an interface to an external process manager.

MPI_COMM_SPAWN starts MPI processes and establishes communication with them, returning an intercommunicator. MPI_COMM_SPAWN_MULTIPLE starts several different binaries (or the same binary with different arguments), placing them in the same MPI_COMM_WORLD and returning an intercommunicator.

MPI uses the group abstraction to represent processes. A process is identified by a (group, rank) pair.

10.2.2 The Runtime Environment

The MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE routines provide an interface between MPI and the *runtime environment* of an MPI application. The difficulty is that there is an enormous range of runtime environments and application requirements, and MPI must not be tailored to any particular one. Examples of such environments are:

- **MPP managed by a batch queueing system.** Batch queueing systems generally allocate resources before an application begins, enforce limits on resource use (CPU time, memory use, etc.), and do not allow a change in resource allocation after a job begins. Moreover, many MPPs have special limitations or extensions, such as a limit on the number of processes that may run on one processor, or the ability to gang-schedule processes of a parallel application.

- **Network of workstations with PVM.** PVM (Parallel Virtual Machine) allows a user to create a “virtual machine” out of a network of workstations. An application may extend the virtual machine or manage processes (create, kill, redirect output, etc.) through the PVM library. Requests to manage the machine or processes may be intercepted and handled by an external resource manager.
- **Network of workstations managed by a load balancing system.** A load balancing system may choose the location of spawned processes based on dynamic quantities, such as load average. It may transparently migrate processes from one machine to another when a resource becomes unavailable.
- **Large SMP with Unix.** Applications are run directly by the user. They are scheduled at a low level by the operating system. Processes may have special scheduling characteristics (gang-scheduling, processor affinity, deadline scheduling, processor locking, etc.) and be subject to OS resource limits (number of processes, amount of memory, etc.).

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide “operating system” services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.).

Complex interaction of an MPI application with its runtime environment should be done through an environment-specific API. An example of such an API would be the PVM task and machine management routines — `pvm_addhosts`, `pvm_config`, `pvm_tasks`, etc., possibly modified to return an MPI (group, rank) when possible. A Condor or PBS API would be another possibility.

At some low level, obviously, MPI must be able to interact with the runtime system, but the interaction is not visible at the application level and the details of the interaction are not specified by the MPI standard.

In many cases, it is impossible to keep environment-specific information out of the MPI interface without seriously compromising MPI functionality. To permit applications to take advantage of environment-specific functionality, many MPI routines take an `info` argument that allows an application to specify environment-specific information. There is a tradeoff between functionality and portability: applications that make use of `info` are not portable.

MPI does not require the existence of an underlying “virtual machine” model, in which there is a consistent global view of an MPI application and an implicit “operating system” managing resources and processes. For instance, processes spawned by one task may not be visible to another; additional hosts added to the runtime environment by one process may not be visible in another process; tasks spawned by different processes may not be automatically distributed over available resources.

Interaction between MPI and the runtime environment is limited to the following areas:

- A process may start new processes with `MPI_COMM_SPAWN` and `MPI_COMM_SPAWN_MULTIPLE`.
- When a process spawns a child process, it may optionally use an `info` argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.

- An attribute `MPI_UNIVERSE_SIZE` (See Section 10.5.1) on `MPI_COMM_WORLD` tells a program how “large” the initial runtime environment is, namely how many processes can usefully be started in all. One can subtract the size of `MPI_COMM_WORLD` from this value to find out how many processes might usefully be started in addition to those already running.

10.3 Process Manager Interface

10.3.1 Processes in MPI

A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a unique process but a process does not determine a unique (group, rank) pair, since a process may belong to several groups.

10.3.2 Starting Processes and Establishing Communication

The following routine starts a number of MPI processes and establishes communication with them, returning an intercommunicator.

Advice to users. It is possible in MPI to start a static SPMD or MPMD application by first starting one process and having that process start its siblings with `MPI_COMM_SPAWN`. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI application. (*End of advice to users.*)

`MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm,
array_of_errcodes)`

IN	command	name of program to be spawned (string, significant only at root)
IN	argv	arguments to <code>command</code> (array of strings, significant only at root)
IN	maxprocs	maximum number of processes to start (integer, significant only at root)
IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root)
IN	root	rank of process in which previous arguments are examined (integer)
IN	comm	intracommunicator containing group of spawning processes (handle)
OUT	intercomm	intercommunicator between original group and the newly spawned group (handle)
OUT	array_of_errcodes	one code per process (array of integer)

```

int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
                   MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
                   int array_of_errcodes[])
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
               array_of_errcodes, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
    INTEGER, INTENT(IN) :: maxprocs, root
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
    INTEGER :: array_of_errcodes(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
               ARRAY_OF_ERRCODES, IERROR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    IERROR

```

MPI_COMM_SPAWN tries to start maxprocs identical copies of the MPI program specified by `command`, establishing communication with them and returning an intercommunicator. The spawned processes are referred to as children. The children have their own MPI_COMM_WORLD, which is separate from that of the parents. MPI_COMM_SPAWN is collective over `comm`, and also may not return until MPI_INIT has been called in the children. Similarly, MPI_INIT in the children may not return until all parents have called MPI_COMM_SPAWN. In this sense, MPI_COMM_SPAWN in the parents and MPI_INIT in the children form a collective operation over the union of parent and child processes. The intercommunicator returned by MPI_COMM_SPAWN contains the parent processes in the local group and the child processes in the remote group. The ordering of processes in the local and remote groups is the same as the ordering of the group of the `comm` in the parents and of MPI_COMM_WORLD of the children, respectively. This intercommunicator can be obtained in the children through the function MPI_COMM_GET_PARENT.

Advice to users. An implementation may automatically establish communication before MPI_INIT is called by the children. Thus, completion of MPI_COMM_SPAWN in the parent does not necessarily mean that MPI_INIT has been called in the children (although the returned intercommunicator can be used immediately). (*End of advice to users.*)

The command argument The `command` argument is a string containing the name of a program to be spawned. The string is null-terminated in C. In Fortran, leading and trailing spaces are stripped. MPI does not specify how to find the executable or how the working directory is determined. These rules are implementation-dependent and should be appropriate for the runtime environment.

Advice to implementors. The implementation should use a natural rule for finding executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning

process, or might search the directories in a PATH environment variable as do Unix shells. An implementation on top of PVM would use PVM's rules for finding executables (usually in \$HOME/pvm3/bin/\$PVM_ARCH). An MPI implementation running under POE on an IBM SP would use POE's method of finding executables. An implementation should document its rules for finding executables and determining working directories, and a high-quality implementation should give the user some control over these rules. (*End of advice to implementors.*)

If the program named in `command` does not call `MPI_INIT`, but instead forks a process that calls `MPI_INIT`, the results are undefined. Implementations may allow this case to work but are not required to.

Advice to users. MPI does not say what happens if the program you start is a shell script and that shell script starts a program that calls `MPI_INIT`. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. (*End of advice to users.*)

The `argv` argument `argv` is an array of strings containing arguments that are passed to the program. The first element of `argv` is the first argument passed to `command`, not, as is conventional in some contexts, the command itself. The argument list is terminated by `NULL` in C and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant `MPI_ARGV_NULL` may be used in C and Fortran to indicate an empty argument list. In C this constant is the same as `NULL`.

Example 10.1 Examples of `argv` in C and Fortran

To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:

```
char command[] = "ocean";
char *argv[] = {"-gridfile", "ocean1.grd", NULL};
MPI_Comm_spawn(command, argv, ...);
```

or, if not everything is known at compile time:

```
char *command;
char **argv;
command = "ocean";
argv=(char **)malloc(3 * sizeof(char *));
argv[0] = "-gridfile";
argv[1] = "ocean1.grd";
argv[2] = NULL;
MPI_Comm_spawn(command, argv, ...);
```

In Fortran:

```

CHARACTER*25 command, argv(3)
command = ' ocean '
argv(1) = ' -gridfile '
argv(2) = ' ocean1.grd'
argv(3) = ' '
call MPI_COMM_SPAWN(command, argv, ...)

```

Arguments are supplied to the program if this is allowed by the operating system. In C, the MPI_COMM_SPAWN argument `argv` differs from the `argv` argument of `main` in two respects. First, it is shifted by one element. Specifically, `argv[0]` of `main` is provided by the implementation and conventionally contains the name of the program (given by `command`). `argv[1]` of `main` corresponds to `argv[0]` in MPI_COMM_SPAWN, `argv[2]` of `main` to `argv[1]` of MPI_COMM_SPAWN, etc. Passing an `argv` of MPI_ARGV_NULL to MPI_COMM_SPAWN results in `main` receiving `argc` of 1 and an `argv` whose element 0 is (conventionally) the name of the program. Second, `argv` of MPI_COMM_SPAWN must be null-terminated, so that its length can be determined.

If a Fortran implementation supplies routines that allow a program to obtain its arguments, the arguments may be available through that mechanism. In C, if the operating system does not support arguments appearing in `argv` of `main()`, the MPI implementation may add the arguments to the `argv` that is passed to MPI_INIT.

The `maxprocs` argument MPI tries to spawn `maxprocs` processes. If it is unable to spawn `maxprocs` processes, it raises an error of class MPI_ERR_SPAWN.

An implementation may allow the `info` argument to change the default behavior, such that if the implementation is unable to spawn all `maxprocs` processes, it may spawn a smaller number of processes instead of raising an error. In principle, the `info` argument may specify an arbitrary set $\{m_i : 0 \leq m_i \leq \text{maxprocs}\}$ of allowed values for the number of processes spawned. The set $\{m_i\}$ does not necessarily include the value `maxprocs`. If an implementation is able to spawn one of these allowed numbers of processes, MPI_COMM_SPAWN returns successfully and the number of spawned processes, m , is given by the size of the remote group of `intercomm`. If m is less than `maxproc`, reasons why the other processes were not spawned are given in `array_of_errcodes` as described below. If it is not possible to spawn one of the allowed numbers of processes, MPI_COMM_SPAWN raises an error of class MPI_ERR_SPAWN.

A spawn call with the default behavior is called *hard*. A spawn call for which fewer than `maxprocs` processes may be returned is called *soft*. See Section 10.3.4 for more information on the *soft* key for `info`.

Advice to users. By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior “spawn as many processes as possible, up to N ,” you should do a soft spawn, where the set of allowed values $\{m_i\}$ is $\{0 \dots N\}$. However, this is not completely portable, as implementations are not required to support soft spawning. (*End of advice to users.*)

The `info` argument The `info` argument to all of the routines in this chapter is an opaque handle of type MPI_Info in C and Fortran with the `mpi_f08` module and INTEGER in Fortran with the `mpi` module or the include file `mpif.h`. It is a container for a

number of user-specified (key,value) pairs. key and value are strings (null-terminated `char*` in C, `character*(*)` in Fortran). Routines to create and manipulate the info argument are described in Chapter 9.

For the SPAWN calls, info provides additional (and possibly implementation-dependent) instructions to MPI and the runtime system on how to start processes. An application may pass `MPI_INFO_NULL` in C or Fortran. Portable programs not requiring detailed control over process locations should use `MPI_INFO_NULL`.

MPI does not specify the content of the info argument, except to reserve a number of special key values (see Section 10.3.4). The info argument is quite flexible and could even be used, for example, to specify the executable and its command-line arguments. In this case the command argument to `MPI_COMM_SPAWN` could be empty. The ability to do this follows from the fact that MPI does not specify how an executable is found, and the info argument can tell the runtime system where to “find” the executable “” (empty string). Of course a program that does this will not be portable across MPI implementations.

The root argument All arguments before the root argument are examined only on the process whose rank in `comm` is equal to `root`. The value of these arguments on other processes is ignored.

The array_of_errcodes argument The `array_of_errcodes` is an array of length `maxprocs` in which MPI reports the status of each process that MPI was requested to start. If all `maxprocs` processes were spawned, `array_of_errcodes` is filled in with the value `MPI_SUCCESS`. If only m ($0 \leq m < \text{maxprocs}$) processes are spawned, m of the entries will contain `MPI_SUCCESS` and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class `MPI_ERR_SPAWN` if there was no error in the argument list. In C or Fortran, an application may pass `MPI_ERRCODES_IGNORE` if it is not interested in the error codes.

Advice to implementors. `MPI_ERRCODES_IGNORE` in Fortran is a special type of constant, like `MPI_BOTTOM`. See the discussion in Section 2.5.4. (*End of advice to implementors.*)

MPI_COMM_GET_PARENT(parent)

OUT parent the parent communicator (handle)

```
int MPI_Comm_get_parent(MPI_Comm *parent)

MPI_Comm_get_parent(parent, ierror)
    TYPE(MPI_Comm), INTENT(OUT) :: parent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_COMM_GET_PARENT(PARENT, IERROR)
    INTEGER PARENT, IERROR
```

If a process was started with `MPI_COMM_SPAWN` or `MPI_COMM_SPAWN_MULTIPLE`, `MPI_COMM_GET_PARENT` returns the “parent” intercommunicator of the current process.

This parent intercommunicator is created implicitly inside of `MPI_INIT` and is the same intercommunicator returned by `SPAWN` in the parents.

If the process was not spawned, `MPI_COMM_GET_PARENT` returns `MPI_COMM_NULL`.

After the parent communicator is freed or disconnected, `MPI_COMM_GET_PARENT` returns `MPI_COMM_NULL`.

Advice to users. `MPI_COMM_GET_PARENT` returns a handle to a single intercommunicator. Calling `MPI_COMM_GET_PARENT` a second time returns a handle to the same intercommunicator. Freeing the handle with `MPI_COMM_DISCONNECT` or `MPI_COMM_FREE` will cause other references to the intercommunicator to become invalid (dangling). Note that calling `MPI_COMM_FREE` on the parent communicator is not useful. (*End of advice to users.*)

Rationale. The desire of the Forum was to create a constant `MPI_COMM_PARENT` similar to `MPI_COMM_WORLD`. Unfortunately such a constant cannot be used (syntactically) as an argument to `MPI_COMM_DISCONNECT`, which is explicitly allowed. (*End of rationale.*)

10.3.3 Starting Multiple Executables and Establishing Communication

While `MPI_COMM_SPAWN` is sufficient for most cases, it does not allow the spawning of multiple binaries, or of the same binary with multiple sets of arguments. The following routine spawns multiple binaries or the same binary with multiple sets of arguments, establishing communication with them and placing them in the same `MPI_COMM_WORLD`.

```

1 MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv,
2   array_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes)
3

```

4	IN	count	number of commands (positive integer, significant to MPI only at root — see advice to users)
5			
6	IN	array_of_commands	programs to be executed (array of strings, significant only at root)
7			
8	IN	array_of_argv	arguments for commands (array of array of strings, significant only at root)
9			
10	IN	array_of_maxprocs	maximum number of processes to start for each command (array of integer, significant only at root)
11			
12	IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles, significant only at root)
13			
14	IN	root	rank of process in which previous arguments are examined (integer)
15			
16	IN	comm	intracommunicator containing group of spawning processes (handle)
17			
18	OUT	intercomm	intercommunicator between original group and newly spawned group (handle)
19			
20	OUT	array_of_errcodes	one error code per process (array of integer)
21			

```

22
23 int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
24   char **array_of_argv[], const int array_of_maxprocs[], const
25   MPI_Info array_of_info[], int root, MPI_Comm comm,
26   MPI_Comm *intercomm, int array_of_errcodes[])
27

```

```

28 MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
29   array_of_maxprocs, array_of_info, root, comm, intercomm,
30   array_of_errcodes, ierror)
31

```

```

32 INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
33 CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*)
34 CHARACTER(LEN=*), INTENT(IN) :: array_of_argv(count, *)
35 TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
36 TYPE(MPI_Comm), INTENT(IN) :: comm
37 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
38 INTEGER :: array_of_errcodes(*)
39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40

```

```

41 MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
42   ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
43   ARRAY_OF_ERRCODES, IERROR)
44
45 INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
46 INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
47 CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
48

```

MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there are multiple executable specifications. The first argument, `count`, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_COMM_SPAWN. For the Fortran version of `array_of_argv`, the element `array_of_argv(i,j)` is the j -th argument to command number i .

Rationale. This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI_COMM_SPAWN to sort out arguments. Note that the leading dimension of `array_of_argv` *must* be the same as `count`. Also note that Fortran rules for sequence association allow a different value in the first dimension; in this case, the sequence of array elements is interpreted by MPI_COMM_SPAWN_MULTIPLE as if the sequence is stored in an array defined with the first dimension set to `count`. This Fortran feature allows an implementor to define MPI_ARGVS_NULL (see below) with fixed dimensions, e.g., (1,1), or only with one dimension, e.g., (1). (*End of rationale.*)

Advice to users. The argument `count` is interpreted by MPI only at the root, as is `array_of_argv`. Since the leading dimension of `array_of_argv` is `count`, a non-positive value of `count` at a non-root node could theoretically cause a runtime bounds check error, even though `array_of_argv` should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of `count` on the non-root nodes. (*End of advice to users.*)

In any language, an application may use the constant MPI_ARGVS_NULL (which is likely to be `(char ***)0` in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of `array_of_argv` to MPI_ARGV_NULL is not defined. To specify arguments for some commands but not others, the commands without arguments should have a corresponding `argv` whose first element is null (`(char *)0` in C and empty string in Fortran). In Fortran at non-root processes, the `count` argument must be set to a value that is consistent with the provided `array_of_argv` although the content of these arguments has no meaning for this operation.

All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in MPI_COMM_WORLD correspond directly to the order in which the commands are specified in MPI_COMM_SPAWN_MULTIPLE. Assume that m_1 processes are generated by the first command, m_2 by the second, etc. The processes corresponding to the first command have ranks $0, 1, \dots, m_1 - 1$. The processes in the second command have ranks $m_1, m_1 + 1, \dots, m_1 + m_2 - 1$. The processes in the third have ranks $m_1 + m_2, m_1 + m_2 + 1, \dots, m_1 + m_2 + m_3 - 1$, etc.

Advice to users. Calling MPI_COMM_SPAWN multiple times would create many sets of children with different MPI_COMM_WORLDS whereas MPI_COMM_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD, so the two methods are not completely equivalent. There are also two performance-related reasons why, if you need to spawn multiple executables, you may want to use MPI_COMM_SPAWN_MULTIPLE instead of calling MPI_COMM_SPAWN several times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes spawned at the same time may be faster than communication between processes spawned separately. (*End of advice to users.*)

The `array_of_errcodes` argument is a 1-dimensional array of size $\sum_{i=1}^{count} n_i$, where n_i is the i -th element of `array_of_maxprocs`. Command number i corresponds to the n_i contiguous slots in this array from element $\sum_{j=1}^{i-1} n_j$ to $\left[\sum_{j=1}^i n_j\right] - 1$. Error codes are treated as for `MPI_COMM_SPAWN`.

Example 10.2 Examples of `array_of_argv` in C and Fortran

To run the program “ocean” with arguments “-gridfile” and “ocean1.grd” and the program “atmos” with argument “atmos.grd” in C:

```
char *array_of_commands[2] = {"ocean", "atmos"};
char **array_of_argv[2];
char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
char *argv1[] = {"atmos.grd", (char *)0};
array_of_argv[0] = argv0;
array_of_argv[1] = argv1;
MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
```

Here is how you do it in Fortran:

```
CHARACTER*25 commands(2), array_of_argv(2, 3)
commands(1) = ' ocean '
array_of_argv(1, 1) = ' -gridfile '
array_of_argv(1, 2) = ' ocean1.grd'
array_of_argv(1, 3) = ' '

commands(2) = ' atmos '
array_of_argv(2, 1) = ' atmos.grd '
array_of_argv(2, 2) = ' '

call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
```

10.3.4 Reserved Keys

The following keys are reserved. An implementation is not required to interpret these keys, but if it does interpret the key, it must provide the functionality described.

host Value is a hostname. The format of the hostname is determined by the implementation.

arch Value is an architecture name. Valid architecture names and what they mean are determined by the implementation.

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 $a, a + 1, a + 2, \dots, b$
3. $a:b:c$ means $a, a + c, a + 2c, \dots, a + ck$, 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 a power-of-two number of processes.
4. $2:10000:2$ allows an even number of processes.
5. $2:10:2,7$ allows 2, 4, 6, 7, 8, or 10 processes.

10.3.5 Spawn Example

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);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_size != 1)    error("Top heavy with management");

    MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
                      &universe_sizep, &flag);
    if (!flag) {
        printf("This MPI does not support UNIVERSE_SIZE. How many\n\
processes total?");
        scanf("%d", &universe_size);
    } else universe_size = *universe_sizep;
```

```

1   if (universe_size == 1) error("No room to start workers");
2
3   /*
4    * Now spawn the workers. Note that there is a run-time determination
5    * of what type of worker to spawn, and presumably this calculation must
6    * be done at run time and cannot be calculated before starting
7    * the program. If everything is known when the application is
8    * first started, it is generally better to start them all at once
9    * in a single MPI_COMM_WORLD.
10   */
11
12   choose_worker_program(worker_program);
13   MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
14                 MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
15                 MPI_ERRCODES_IGNORE);
16   /*
17    * Parallel code here. The communicator "everyone" can be used
18    * to communicate with the spawned processes, which have ranks 0,..
19    * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
20    * "everyone".
21    */
22
23   MPI_Finalize();
24   return 0;
25 }
26
27 /* worker */
28
29 #include "mpi.h"
30 int main(int argc, char *argv[])
31 {
32     int size;
33     MPI_Comm parent;
34     MPI_Init(&argc, &argv);
35     MPI_Comm_get_parent(&parent);
36     if (parent == MPI_COMM_NULL) error("No parent!");
37     MPI_Comm_remote_size(parent, &size);
38     if (size != 1) error("Something's wrong with the parent");
39
40     /*
41      * Parallel code here.
42      * The manager is represented as the process with rank 0 in (the remote
43      * group of) the parent communicator. If the workers need to communicate
44      * among themselves, they can use MPI_COMM_WORLD.
45      */
46
47     MPI_Finalize();
48     return 0;

```

}

10.4 Establishing Communication

This section provides functions that establish communication between two sets of MPI processes that do not share a communicator.

Some situations in which these functions are useful are:

1. Two parts of an application that are started independently need to communicate.
2. A visualization tool wants to attach to a running process.
3. A server wants to accept connections from multiple clients. Both clients and server may be parallel programs.

In each of these situations, MPI must establish communication channels where none existed before, and there is no parent/child relationship. The routines described in this section establish communication between the two sets of processes by creating an MPI intercommunicator, where the two groups of the intercommunicator are the original sets of processes.

Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. One group of processes indicates its willingness to accept connections from other groups of processes. We will call this group the (parallel) *server*, even if this is not a client/server type of application. The other group connects to the server; we will call it the *client*.

Advice to users. While the names *client* and *server* are used throughout this section, MPI does not guarantee the traditional robustness of client/server systems. The functionality described in this section is intended to allow two cooperating parts of the same application to communicate with one another. For instance, a client that gets a segmentation fault and dies, or one that does not participate in a collective operation may cause a server to crash or hang. (*End of advice to users.*)

10.4.1 Names, Addresses, Ports, and All That

Almost all of the complexity in MPI client/server routines addresses the question “how does the client find out how to contact the server?” The difficulty, of course, is that there is no existing communication channel between them, yet they must somehow agree on a rendezvous point where they will establish communication.

Agreeing on a rendezvous point always involves a third party. The third party may itself provide the rendezvous point or may communicate rendezvous information from server to client. Complicating matters might be the fact that a client does not really care what server it contacts, only that it be able to get in touch with one that can handle its request.

Ideally, MPI can accommodate a wide variety of run-time systems while retaining the ability to write simple, portable code. The following should be compatible with MPI:

- The server resides at a well-known internet address host:port.
- The server prints out an address to the terminal; the user gives this address to the client program.

- The server places the address information on a nameserver, where it can be retrieved with an agreed-upon name.
- The server to which the client connects is actually a broker, acting as a middleman between the client and the real server.

MPI does not require a nameserver, so not all implementations will be able to support all of the above scenarios. However, MPI provides an optional nameserver interface, and is compatible with external name servers.

A `port_name` is a *system-supplied* string that encodes a low-level network address at which a server can be contacted. Typically this is an IP address and a port number, but an implementation is free to use any protocol. The server establishes a `port_name` with the `MPI_OPEN_PORT` routine. It accepts a connection to a given port with `MPI_COMM_ACCEPT`. A client uses `port_name` to connect to the server.

By itself, the `port_name` mechanism is completely portable, but it may be clumsy to use because of the necessity to communicate `port_name` to the client. It would be more convenient if a server could specify that it be known by an *application-supplied* `service_name` so that the client could connect to that `service_name` without knowing the `port_name`.

An MPI implementation may allow the server to publish a (`port_name`, `service_name`) pair with `MPI_PUBLISH_NAME` and the client to retrieve the port name from the service name with `MPI_LOOKUP_NAME`. This allows three levels of portability, with increasing levels of functionality.

1. Applications that do not rely on the ability to publish names are the most portable. Typically the `port_name` must be transferred “by hand” from server to client.
2. Applications that use the `MPI_PUBLISH_NAME` mechanism are completely portable among implementations that provide this service. To be portable among all implementations, these applications should have a fall-back mechanism that can be used when names are not published.
3. Applications may ignore MPI’s name publishing functionality and use their own mechanism (possibly system-supplied) to publish names. This allows arbitrary flexibility but is not portable.

10.4.2 Server Routines

A server makes itself available with two routines. First it must call `MPI_OPEN_PORT` to establish a port at which it may be contacted. Secondly it must call `MPI_COMM_ACCEPT` to accept connections from clients.

`MPI_OPEN_PORT`(info, port_name)

IN	info	implementation-specific information on how to establish an address (handle)
OUT	port_name	newly established port (string)

`int MPI_Open_port(MPI_Info info, char *port_name)`

`MPI_Open_port`(info, port_name, ierror)


```

    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERROR

```

This function establishes a network address, encoded in the `port_name` string, at which the server will be able to accept connections from clients. `port_name` is supplied by the system, possibly using information in the `info` argument.

MPI copies a system-supplied port name into `port_name`. `port_name` identifies the newly opened port and can be used by a client to contact the server. The maximum size string that may be supplied by the system is `MPI_MAX_PORT_NAME`.

Advice to users. The system copies the port name into `port_name`. The application must pass a buffer of sufficient size to hold this value. (*End of advice to users.*)

`port_name` is essentially a network address. It is unique within the communication universe to which it belongs (determined by the implementation), and may be used by any client within that communication universe. For instance, if it is an internet (host:port) address, it will be unique on the internet. If it is a low level switch address on an IBM SP, it will be unique to that SP.

Advice to implementors. These examples are not meant to constrain implementations. A `port_name` could, for instance, contain a user name or the name of a batch job, as long as it is unique within some well-defined communication domain. The larger the communication domain, the more useful MPI's client/server functionality will be. (*End of advice to implementors.*)

The precise form of the address is implementation-defined. For instance, an internet address may be a host name or IP address, or anything that the implementation can decode into an IP address. A port name may be reused after it is freed with `MPI_CLOSE_PORT` and released by the system.

Advice to implementors. Since the user may type in `port_name` by hand, it is useful to choose a form that is easily readable and does not have embedded spaces. (*End of advice to implementors.*)

`info` may be used to tell the implementation how to establish the address. It may, and usually will, be `MPI_INFO_NULL` in order to get the implementation defaults.

```

MPI_CLOSE_PORT(port_name)
    IN          port_name          a port (string)

int MPI_Close_port(const char *port_name)

MPI_Close_port(port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```
1 MPI_CLOSE_PORT(PORT_NAME, IERROR)
```

```
2     CHARACTER*(*) PORT_NAME
```

```
3     INTEGER IERROR
```

4 This function releases the network address represented by `port_name`.

```
7 MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
```

```
9     IN      port_name      port name (string, used only on root)
```

```
10    IN      info           implementation-dependent information (handle, used
11                                only on root)
```

```
12    IN      root           rank in comm of root node (integer)
```

```
13    IN      comm           intracommunicator over which call is collective (han-
14                                dle)
```

```
15    OUT     newcomm        intercommunicator with client as remote group (han-
16                                dle)
```

```
19 int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
20                     MPI_Comm comm, MPI_Comm *newcomm)
```

```
21 MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
```

```
22     CHARACTER(LEN=*), INTENT(IN) :: port_name
```

```
23     TYPE(MPI_Info), INTENT(IN) :: info
```

```
24     INTEGER, INTENT(IN) :: root
```

```
25     TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
26     TYPE(MPI_Comm), INTENT(OUT) :: newcomm
```

```
27     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
29 MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
```

```
30     CHARACTER*(*) PORT_NAME
```

```
31     INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
```

32 `MPI_COMM_ACCEPT` establishes communication with a client. It is collective over the
33 calling communicator. It returns an intercommunicator that allows communication with the
34 client.

35 The `port_name` must have been established through a call to `MPI_OPEN_PORT`.

36 `info` can be used to provide directives that may influence the behavior of the `ACCEPT`
37 call.

39 10.4.3 Client Routines

40 There is only one routine on the client side.

MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)			1
IN	port_name	network address (string, used only on root)	2
IN	info	implementation-dependent information (handle, used only on root)	3
IN	root	rank in comm of root node (integer)	4
IN	comm	intracommunicator over which call is collective (handle)	5
OUT	newcomm	intercommunicator with server as remote group (handle)	6

```

int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
                    MPI_Comm comm, MPI_Comm *newcomm)

```

```

MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
    CHARACTER(LEN=*) INTENT(IN) :: port_name
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(IN) :: root
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

```

This routine establishes communication with a server specified by `port_name`. It is collective over the calling communicator and returns an intercommunicator in which the remote group participated in an `MPI_COMM_ACCEPT`.

If the named port does not exist (or has been closed), `MPI_COMM_CONNECT` raises an error of class `MPI_ERR_PORT`.

If the port exists, but does not have a pending `MPI_COMM_ACCEPT`, the connection attempt will eventually time out after an implementation-defined time, or succeed when the server calls `MPI_COMM_ACCEPT`. In the case of a time out, `MPI_COMM_CONNECT` raises an error of class `MPI_ERR_PORT`.

Advice to implementors. The time out period may be arbitrarily short or long. However, a high-quality implementation will try to queue connection attempts so that a server can handle simultaneous requests from several clients. A high-quality implementation may also provide a mechanism, through the `info` arguments to `MPI_OPEN_PORT`, `MPI_COMM_ACCEPT`, and/or `MPI_COMM_CONNECT`, for the user to control timeout and queuing behavior. (*End of advice to implementors.*)

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order they were initiated and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

`port_name` is the address of the server. It must be the same as the name returned by `MPI_OPEN_PORT` on the server. Some freedom is allowed here. If there are equivalent

forms of `port_name`, an implementation may accept them as well. For instance, if `port_name` is `(hostname:port)`, an implementation may accept `(ip_address:port)` as well.

10.4.4 Name Publishing

The routines in this section provide a mechanism for publishing names. A `(service_name, port_name)` pair is published by the server, and may be retrieved by a client using the `service_name` only. An MPI implementation defines the *scope* of the `service_name`, that is, the domain over which the `service_name` can be retrieved. If the domain is the empty set, that is, if no client can retrieve the information, then we say that name publishing is not supported. Implementations should document how the scope is determined. High-quality implementations will give some control to users through the `info` arguments to name publishing functions. Examples are given in the descriptions of individual functions.

`MPI_PUBLISH_NAME(service_name, info, port_name)`

IN	<code>service_name</code>	a service name to associate with the port (string)
IN	<code>info</code>	implementation-specific information (handle)
IN	<code>port_name</code>	a port name (string)

```
int MPI_Publish_name(const char *service_name, MPI_Info info, const
                    char *port_name)
```

```
MPI_Publish_name(service_name, info, port_name, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

This routine publishes the pair `(port_name, service_name)` so that an application may retrieve a system-supplied `port_name` using a well-known `service_name`.

The implementation must define the *scope* of a published service name, that is, the domain over which the service name is unique, and conversely, the domain over which the `(port name, service name)` pair may be retrieved. For instance, a service name may be unique to a job (where job is defined by a distributed operating system or batch scheduler), unique to a machine, or unique to a Kerberos realm. The scope may depend on the `info` argument to `MPI_PUBLISH_NAME`.

MPI permits publishing more than one `service_name` for a single `port_name`. On the other hand, if `service_name` has already been published within the scope determined by `info`, the behavior of `MPI_PUBLISH_NAME` is undefined. An MPI implementation may, through a mechanism in the `info` argument to `MPI_PUBLISH_NAME`, provide a way to allow multiple servers with the same service in the same scope. In this case, an implementation-defined policy will determine which of several port names is returned by `MPI_LOOKUP_NAME`.

Note that while `service_name` has a limited scope, determined by the implementation, `port_name` always has global scope within the communication universe used by the imple-

mentation (i.e., it is globally unique).

`port_name` should be the name of a port established by `MPI_OPEN_PORT` and not yet released by `MPI_CLOSE_PORT`. If it is not, the result is undefined.

Advice to implementors. In some cases, an MPI implementation may use a name service that a user can also access directly. In this case, a name published by MPI could easily conflict with a name published by a user. In order to avoid such conflicts, MPI implementations should mangle service names so that they are unlikely to conflict with user code that makes use of the same service. Such name mangling will of course be completely transparent to the user.

The following situation is problematic but unavoidable, if we want to allow implementations to use nameservers. Suppose there are multiple instances of “ocean” running on a machine. If the scope of a service name is confined to a job, then multiple oceans can coexist. If an implementation provides site-wide scope, however, multiple instances are not possible as all calls to `MPI_PUBLISH_NAME` after the first may fail. There is no universal solution to this.

To handle these situations, a high-quality implementation should make it possible to limit the domain over which names are published. (*End of advice to implementors.*)

`MPI_UNPUBLISH_NAME(service_name, info, port_name)`

IN	<code>service_name</code>	a service name (string)
IN	<code>info</code>	implementation-specific information (handle)
IN	<code>port_name</code>	a port name (string)

```
int MPI_Unpublish_name(const char *service_name, MPI_Info info, const
                      char *port_name)
```

```
MPI_Unpublish_name(service_name, info, port_name, ierror)
  CHARACTER(LEN=*) INTENT(IN) :: service_name, port_name
  TYPE(MPI_Info), INTENT(IN) :: info
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
  INTEGER INFO, IERROR
  CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

This routine unpublishes a service name that has been previously published. Attempting to unpublish a name that has not been published or has already been unpublished is erroneous and is indicated by the error class `MPI_ERR_SERVICE`.

All published names must be unpublished before the corresponding port is closed and before the publishing process exits. The behavior of `MPI_UNPUBLISH_NAME` is implementation dependent when a process tries to unpublish a name that it did not publish.

If the `info` argument was used with `MPI_PUBLISH_NAME` to tell the implementation how to publish names, the implementation may require that `info` passed to `MPI_UNPUBLISH_NAME` contain information to tell the implementation how to unpublish a name.

```

1 MPI_LOOKUP_NAME(service_name, info, port_name)
2     IN          service_name          a service name (string)
3
4     IN          info                  implementation-specific information (handle)
5
6     OUT         port_name             a port name (string)

```

```

7 int MPI_Lookup_name(const char *service_name, MPI_Info info,
8                     char *port_name)
9
10 MPI_Lookup_name(service_name, info, port_name, ierror)
11     CHARACTER(LEN=*), INTENT(IN) :: service_name
12     TYPE(MPI_Info), INTENT(IN) :: info
13     CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16 MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
17     CHARACTER*(*) SERVICE_NAME, PORT_NAME
18     INTEGER INFO, IERROR

```

This function retrieves a `port_name` published by `MPI_PUBLISH_NAME` with `service_name`. If `service_name` has not been published, it raises an error in the error class `MPI_ERR_NAME`. The application must supply a `port_name` buffer large enough to hold the largest possible port name (see discussion above under `MPI_OPEN_PORT`).

If an implementation allows multiple entries with the same `service_name` within the same scope, a particular `port_name` is chosen in a way determined by the implementation.

If the `info` argument was used with `MPI_PUBLISH_NAME` to tell the implementation how to publish names, a similar `info` argument may be required for `MPI_LOOKUP_NAME`.

10.4.5 Reserved Key Values

The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.

`ip_port` Value contains IP port number at which to establish a `port`. (Reserved for `MPI_OPEN_PORT` only).

`ip_address` Value contains IP address at which to establish a `port`. If the address is not a valid IP address of the host on which the `MPI_OPEN_PORT` call is made, the results are undefined. (Reserved for `MPI_OPEN_PORT` only).

10.4.6 Client/Server Examples

Simplest Example — Completely Portable.

The following example shows the simplest way to use the client/server interface. It does not use service names at all.

On the server side:

```

47 char myport[MPI_MAX_PORT_NAME];
48 MPI_Comm intercomm;

```

```

/* ... */
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);

MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */

```

The server prints out the port name to the terminal and the user must type it in when starting up the client (assuming the MPI implementation supports `stdin` such that this works). On the client side:

```

MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);

```

Ocean/Atmosphere — Relies on Name Publishing

In this example, the “ocean” application is the “server” side of a coupled ocean-atmosphere climate model. It assumes that the MPI implementation publishes names.

```

MPI_Open_port(MPI_INFO_NULL, port_name);
MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);

MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */
MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);

```

On the client side:

```

MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
                  &intercomm);

```

Simple Client-Server Example

This is a simple example; the server accepts only a single connection at a time and serves that connection until the client requests to be disconnected. The server is a single process.

Here is the server. It accepts a single connection and then processes data until it receives a message with tag 1. A message with tag 0 tells the server to exit.

```

#include "mpi.h"
int main(int argc, char *argv[])
{
    MPI_Comm client;
    MPI_Status status;
    char port_name[MPI_MAX_PORT_NAME];

```

```

1      double buf[MAX_DATA];
2      int    size, again;
3
4      MPI_Init(&argc, &argv);
5      MPI_Comm_size(MPI_COMM_WORLD, &size);
6      if (size != 1) error(FATAL, "Server too big");
7      MPI_Open_port(MPI_INFO_NULL, port_name);
8      printf("server available at %s\n", port_name);
9      while (1) {
10         MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
11                         &client);
12         again = 1;
13         while (again) {
14             MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
15                     MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
16             switch (status.MPI_TAG) {
17                 case 0: MPI_Comm_free(&client);
18                         MPI_Close_port(port_name);
19                         MPI_Finalize();
20                         return 0;
21                 case 1: MPI_Comm_disconnect(&client);
22                         again = 0;
23                         break;
24                 case 2: /* do something */
25                         ...
26                 default:
27                     /* Unexpected message type */
28                     MPI_Abort(MPI_COMM_WORLD, 1);
29             }
30         }
31     }
32 }
33
34     Here is the client.
35
36 #include "mpi.h"
37 int main( int argc, char **argv )
38 {
39     MPI_Comm server;
40     double buf[MAX_DATA];
41     char port_name[MPI_MAX_PORT_NAME];
42
43     MPI_Init( &argc, &argv );
44     strcpy( port_name, argv[1] ); /* assume server's name is cmd-line arg */
45
46     MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
47                     &server );
48

```



```

while (!done) {
    tag = 2; /* Action to perform */
    MPI_Send( buf, n, MPI_DOUBLE, 0, tag, server );
    /* etc */
}
MPI_Send( buf, 0, MPI_DOUBLE, 0, 1, server );
MPI_Comm_disconnect( &server );
MPI_Finalize();
return 0;
}

```

10.5 Other Functionality

10.5.1 Universe Size

Many “dynamic” MPI applications are expected to exist in a static runtime environment, in which resources have been allocated before the application is run. When a user (or possibly a batch system) runs one of these quasi-static applications, she will usually specify a number of processes to start and a total number of processes that are expected. An application simply needs to know how many slots there are, i.e., how many processes it should spawn.

MPI provides an attribute on `MPI_COMM_WORLD`, `MPI_UNIVERSE_SIZE`, that allows the application to obtain this information in a portable manner. This attribute indicates the total number of processes that are expected. In Fortran, the attribute is the integer value. In C, the attribute is a pointer to the integer value. An application typically subtracts the size of `MPI_COMM_WORLD` from `MPI_UNIVERSE_SIZE` to find out how many processes it should spawn. `MPI_UNIVERSE_SIZE` is initialized in `MPI_INIT` and is not changed by MPI. If defined, it has the same value on all processes of `MPI_COMM_WORLD`. `MPI_UNIVERSE_SIZE` is determined by the application startup mechanism in a way not specified by MPI. (The size of `MPI_COMM_WORLD` is another example of such a parameter.)

Possibilities for how `MPI_UNIVERSE_SIZE` might be set include

- A `-universe_size` argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.
- An environment variable set by the user.
- Extra information passed to `MPI_COMM_SPAWN` through the `info` argument.

An implementation must document how `MPI_UNIVERSE_SIZE` is set. An implementation may not support the ability to set `MPI_UNIVERSE_SIZE`, in which case the attribute `MPI_UNIVERSE_SIZE` is not set.

`MPI_UNIVERSE_SIZE` is a recommendation, not necessarily a hard limit. For instance, some implementations may allow an application to spawn 50 processes per processor, if they are requested. However, it is likely that the user only wants to spawn one process per processor.

`MPI_UNIVERSE_SIZE` is assumed to have been specified when an application was started, and is in essence a portable mechanism to allow the user to pass to the application (through

the MPI process startup mechanism, such as `mpiexec`) a piece of critical runtime information. Note that no interaction with the runtime environment is required. If the runtime environment changes size while an application is running, `MPI_UNIVERSE_SIZE` is not updated, and the application must find out about the change through direct communication with the runtime system.

10.5.2 Singleton MPI_INIT

A high-quality implementation will allow any process (including those not started with a “parallel application” mechanism) to become an MPI process by calling `MPI_INIT`. Such a process can then connect to other MPI processes using the `MPI_COMM_ACCEPT` and `MPI_COMM_CONNECT` routines, or spawn other MPI processes. MPI does not mandate this behavior, but strongly encourages it where technically feasible.

Advice to implementors. To start MPI processes belonging to the same `MPI_COMM_WORLD` requires some special coordination. The processes must be started at the “same” time, they must have a mechanism to establish communication, etc. Either the user or the operating system must take special steps beyond simply starting processes.

When an application enters `MPI_INIT`, clearly it must be able to determine if these special steps were taken. If a process enters `MPI_INIT` and determines that no special steps were taken (i.e., it has not been given the information to form an `MPI_COMM_WORLD` with other processes) it succeeds and forms a singleton MPI program, that is, one in which `MPI_COMM_WORLD` has size 1.

In some implementations, MPI may not be able to function without an “MPI environment.” For example, MPI may require that daemons be running or MPI may not be able to work at all on the front-end of an MPP. In this case, an MPI implementation may either

1. Create the environment (e.g., start a daemon) or
2. Raise an error if it cannot create the environment and the environment has not been started independently.

A high-quality implementation will try to create a singleton MPI process and not raise an error.

(End of advice to implementors.)

10.5.3 MPI_APPNUM

There is a predefined attribute `MPI_APPNUM` of `MPI_COMM_WORLD`. In Fortran, the attribute is an integer value. In C, the attribute is a pointer to an integer value. If a process was spawned with `MPI_COMM_SPAWN_MULTIPLE`, `MPI_APPNUM` is the command number that generated the current process. Numbering starts from zero. If a process was spawned with `MPI_COMM_SPAWN`, it will have `MPI_APPNUM` equal to zero.

Additionally, if the process was not started by a spawn call, but by an implementation-specific startup mechanism that can handle multiple process specifications, `MPI_APPNUM` should be set to the number of the corresponding process specification. In particular, if it is started with

```
mpirexec spec0 [: spec1 : spec2 : ...]
```

MPI_APPNUM should be set to the number of the corresponding specification.

If an application was not spawned with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, and MPI_APPNUM does not make sense in the context of the implementation-specific startup mechanism, MPI_APPNUM is not set.

MPI implementations may optionally provide a mechanism to override the value of MPI_APPNUM through the info argument. MPI reserves the following key for all SPAWN calls.

appnum Value contains an integer that overrides the default value for MPI_APPNUM in the child.

Rationale. When a single application is started, it is able to figure out how many processes there are by looking at the size of MPI_COMM_WORLD. An application consisting of multiple SPMD sub-applications has no way to find out how many sub-applications there are and to which sub-application the process belongs. While there are ways to figure it out in special cases, there is no general mechanism. MPI_APPNUM provides such a general mechanism. (*End of rationale.*)

10.5.4 Releasing Connections

Before a client and server connect, they are independent MPI applications. An error in one does not affect the other. After establishing a connection with MPI_COMM_CONNECT and MPI_COMM_ACCEPT, an error in one may affect the other. It is desirable for a client and server to be able to disconnect, so that an error in one will not affect the other. Similarly, it might be desirable for a parent and child to disconnect, so that errors in the child do not affect the parent, or vice-versa.

- Two processes are *connected* if there is a communication path (direct or indirect) between them. More precisely:
 1. Two processes are connected if
 - (a) they both belong to the same communicator (inter- or intra-, including MPI_COMM_WORLD) *or*
 - (b) they have previously belonged to a communicator that was freed with MPI_COMM_FREE instead of MPI_COMM_DISCONNECT *or*
 - (c) they both belong to the group of the same window or filehandle.
 2. If A is connected to B and B to C, then A is connected to C.
- Two processes are *disconnected* (also *independent*) if they are not connected.
- By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.
- Processes which are connected, but do not share the same MPI_COMM_WORLD, may become disconnected (independent) if the communication path between them is broken by using MPI_COMM_DISCONNECT.

The following additional rules apply to MPI routines in other chapters:

- `MPI_FINALIZE` is collective over a set of connected processes.
- `MPI_ABORT` does not abort independent processes. It may abort all processes in the caller's `MPI_COMM_WORLD` (ignoring its `comm` argument). Additionally, it may abort connected processes as well, though it makes a “best attempt” to abort only the processes in `comm`.
- If a process terminates without calling `MPI_FINALIZE`, independent processes are not affected but the effect on connected processes is not defined.

`MPI_COMM_DISCONNECT(comm)`

INOUT `comm` communicator (handle)

```
int MPI_Comm_disconnect(MPI_Comm *comm)
```

```
MPI_Comm_disconnect(comm, ierror)
```

```
    TYPE(MPI_Comm), INTENT(INOUT) :: comm
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_COMM_DISCONNECT(COMM, IERROR)
```

```
    INTEGER COMM, IERROR
```

This function waits for all pending communication on `comm` to complete internally, deallocates the communicator object, and sets the handle to `MPI_COMM_NULL`. It is a collective operation.

It may not be called with the communicator `MPI_COMM_WORLD` or `MPI_COMM_SELF`.

`MPI_COMM_DISCONNECT` may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for `MPI_FINALIZE`.

`MPI_COMM_DISCONNECT` has the same action as `MPI_COMM_FREE`, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.

Advice to users. To disconnect two processes you may need to call `MPI_COMM_DISCONNECT`, `MPI_WIN_FREE`, and `MPI_FILE_CLOSE` to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (*End of advice to users.*)

Rationale. It would be nice to be able to use `MPI_COMM_FREE` instead, but that function explicitly does not wait for pending communication to complete. (*End of rationale.*)

10.5.5 Another Way to Establish MPI Communication

`MPI_COMM_JOIN(fd, intercomm)`

IN	<code>fd</code>	socket file descriptor
OUT	<code>intercomm</code>	new intercommunicator (handle)

`int MPI_Comm_join(int fd, MPI_Comm *intercomm)`

`MPI_Comm_join(fd, intercomm, ierror)`

INTEGER, INTENT(IN) ::	<code>fd</code>
TYPE(MPI_Comm), INTENT(OUT) ::	<code>intercomm</code>
INTEGER, OPTIONAL, INTENT(OUT) ::	<code>ierror</code>

`MPI_COMM_JOIN(FD, INTERCOMM, IERROR)`

INTEGER FD, INTERCOMM, IERROR

`MPI_COMM_JOIN` is intended for MPI implementations that exist in an environment supporting the Berkeley Socket interface [45, 49]. Implementations that exist in an environment not supporting Berkeley Sockets should provide the entry point for `MPI_COMM_JOIN` and should return `MPI_COMM_NULL`.

This call creates an intercommunicator from the union of two MPI processes which are connected by a socket. `MPI_COMM_JOIN` should normally succeed if the local and remote processes have access to the same implementation-defined MPI communication universe.

Advice to users. An MPI implementation may require a specific communication medium for MPI communication, such as a shared memory segment or a special switch. In this case, it may not be possible for two processes to successfully join even if there is a socket connecting them and they are using the same MPI implementation. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to establish communication over a slow medium if its preferred one is not available. If implementations do not do this, they must document why they cannot do MPI communication over the medium used by the socket (especially if the socket is a TCP connection). (*End of advice to implementors.*)

`fd` is a file descriptor representing a socket of type `SOCK_STREAM` (a two-way reliable byte-stream connection). Nonblocking I/O and asynchronous notification via `SIGIO` must not be enabled for the socket. The socket must be in a connected state. The socket must be quiescent when `MPI_COMM_JOIN` is called (see below). It is the responsibility of the application to create the socket using standard socket API calls.

`MPI_COMM_JOIN` must be called by the process at each end of the socket. It does not return until both processes have called `MPI_COMM_JOIN`. The two processes are referred to as the local and remote processes.

MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing else. Upon return from `MPI_COMM_JOIN`, the file descriptor will be open and quiescent (see below).

1 If MPI is unable to create an intercommunicator, but is able to leave the socket in its
2 original state, with no pending communication, it succeeds and sets `intercomm` to
3 `MPI_COMM_NULL`.

4 The socket must be quiescent before `MPI_COMM_JOIN` is called and after
5 `MPI_COMM_JOIN` returns. More specifically, on entry to `MPI_COMM_JOIN`, a `read` on the
6 socket will not read any data that was written to the socket before the remote process called
7 `MPI_COMM_JOIN`. On exit from `MPI_COMM_JOIN`, a `read` will not read any data that was
8 written to the socket before the remote process returned from `MPI_COMM_JOIN`. It is the
9 responsibility of the application to ensure the first condition, and the responsibility of the
10 MPI implementation to ensure the second. In a multithreaded application, the application
11 must ensure that one thread does not access the socket while another is calling
12 `MPI_COMM_JOIN`, or call `MPI_COMM_JOIN` concurrently.

13
14 *Advice to implementors.* MPI is free to use any available communication path(s)
15 for MPI messages in the new communicator; the socket is only used for the initial
16 handshaking. (*End of advice to implementors.*)

17
18 `MPI_COMM_JOIN` uses non-MPI communication to do its work. The interaction of
19 non-MPI communication with pending MPI communication is not defined. Therefore, the
20 result of calling `MPI_COMM_JOIN` on two connected processes (see Section 10.5.4 for the
21 definition of connected) is undefined.

22 The returned communicator may be used to establish MPI communication with addi-
23 tional processes, through the usual MPI communicator creation mechanisms.

24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48