Chapter 6. Programming Using the Message-Passing Paradigm

Numerous programming languages and libraries have been developed for explicit parallel programming. These differ in their view of the address space that they make available to the programmer, the degree of synchronization imposed on concurrent activities, and the multiplicity of programs. The *message-passing programming paradigm* is one of the oldest and most widely used approaches for programming parallel computers. Its roots can be traced back in the early days of parallel processing and its wide-spread adoption can be attributed to the fact that it imposes minimal requirements on the underlying hardware.

In this chapter, we first describe some of the basic concepts of the message-passing programming paradigm and then explore various message-passing programming techniques using the standard and widely-used Message Passing Interface.

6.1 Principles of Message-Passing Programming

There are two key attributes that characterize the message-passing programming paradigm. The first is that it assumes a partitioned address space and the second is that it supports only explicit parallelization.

The logical view of a machine supporting the message-passing paradigm consists of *p* processes, each with its own exclusive address space. Instances of such a view come naturally from clustered workstations and non-shared address space multicomputers. There are two immediate implications of a partitioned address space. First, each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed. This adds complexity to programming, but encourages locality of access that is critical for achieving high performance on non-UMA architecture, since a processor can access its local data much faster than non-local data on such architectures. The second implication is that all interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data. This requirement for cooperation adds a great deal of complexity for a number of reasons. The process that has the data must participate in the interaction even if it has no logical connection to the events at the requesting process. In certain circumstances, this requirement leads to unnatural programs. In particular, for dynamic and/or unstructured interactions the complexity of the code written for this type of paradigm can be very high for this reason. However, a primary advantage of explicit two-way interactions is that the programmer is fully aware of all the costs of non-local interactions, and is more likely to think about algorithms (and mappings) that minimize interactions. Another major advantage of this type of programming paradigm is that it can be efficiently implemented on a wide variety of architectures.

The message-passing programming paradigm requires that the parallelism is coded explicitly by the programmer. That is, the programmer is responsible for analyzing the underlying serial algorithm/application and identifying ways by which he or she can decompose the computations and extract concurrency. As a result, programming using the message-passing paradigm tends to be hard and intellectually demanding. However, on the other hand, properly written message-passing programs can often achieve very high performance and scale to a very large number of processes.

Structure of Message-Passing Programs Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms. In the asynchronous paradigm, all concurrent tasks execute asynchronously. This makes it possible to implement any parallel algorithm. However, such programs can be harder to reason about, and can have non-deterministic behavior due to race conditions. Loosely synchronous programs are a good compromise between these two extremes. In such programs, tasks or subsets of tasks synchronize to perform interactions. However, between these interactions, tasks execute completely asynchronously. Since the interaction happens synchronously, it is still quite easy to reason about the program. Many of the known parallel algorithms can be naturally implemented using loosely synchronous programs.

In its most general form, the message-passing paradigm supports execution of a different program on each of the *p* processes. This provides the ultimate flexibility in parallel programming, but makes the job of writing parallel programs effectively unscalable. For this reason, most message-passing programs are written using the *single program multiple data* (SPMD) approach. In SPMD programs the code executed by different processes is identical except for a small number of processes (e.g., the "root" process). This does not mean that the

processes work in lock-step. In an extreme case, even in an SPMD program, each process could execute a different code (the program contains a large case statement with code for each process). But except for this degenerate case, most processes execute the same code. SPMD programs can be loosely synchronous or completely asynchronous.

PREVIOUS NEXT

6.2 The Building Blocks: Send and Receive Operations

Since interactions are accomplished by sending and receiving messages, the basic operations in the message-passing programming paradigm are `send` and `receive` . In their simplest form, the prototypes of these operations are defined as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

The `sendbuf` points to a buffer that stores the data to be sent, `recvbuf` points to a buffer that stores the data to be received, `nelems` is the number of data units to be sent and received, `dest` is the identifier of the process that receives the data, and `source` is the identifier of the process that sends the data.

However, to stop at this point would be grossly simplifying the programming and performance ramifications of how these functions are implemented. To motivate the need for further investigation, let us start with a simple example of a process sending a piece of data to another process as illustrated in the following code-fragment:

```
1          P0                                P1
2
3          a = 100;                           receive(&a, 1, 0)
4          send(&a, 1, 1);                    printf("%d\n", a);
5          a=0;
```

In this simple example, process P0 sends a message to process P1 which receives and prints the message. The important thing to note is that process P0 changes the value of `a` to 0 immediately following the `send` . The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0. That is, the value of `a` at the time of the send operation must be the value that is received by process `P1` .

It may seem that it is quite straightforward to ensure the semantics of the send and receive operations. However, based on how the send and receive operations are implemented this may not be the case. Most message passing platforms have additional hardware support for sending and receiving messages. They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware. Network interfaces allow the transfer of messages from buffer memory to desired location without CPU intervention. Similarly, DMA allows copying of data from one memory location to another (e.g., communication buffers) without CPU support (once they have been programmed). As a result, if the send operation programs the communication hardware and returns before the communication operation has been accomplished, process `P1` might receive the value 0 in `a` instead of 100!

While this is undesirable, there are in fact reasons for supporting such send operations for performance reasons. In the rest of this section, we will discuss send and receive operations in the context of such a hardware environment, and motivate various implementation details and message-passing protocols that help in ensuring the semantics of the send and receive operations.

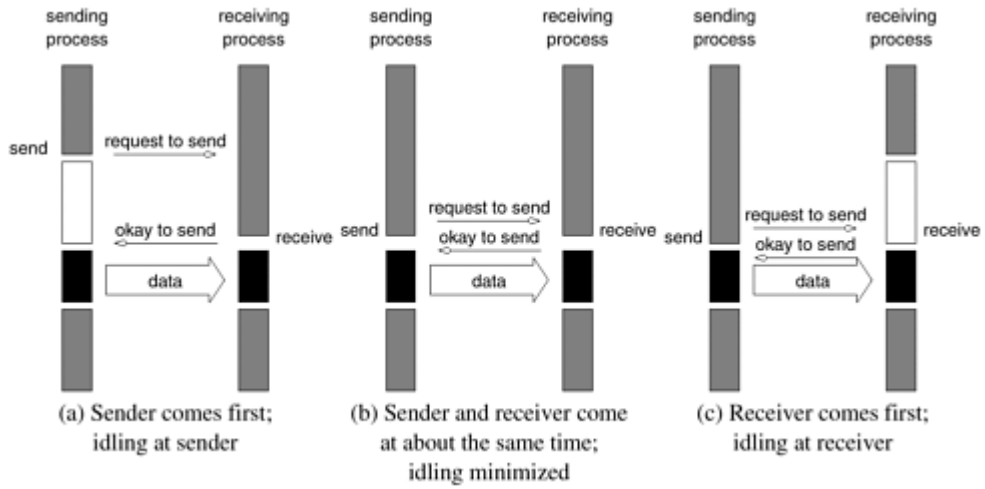
6.2.1 Blocking Message Passing Operations

A simple solution to the dilemma presented in the code fragment above is for the send operation to return only when it is semantically safe to do so. Note that this is not the same as saying that the send operation returns only after the receiver has received the data. It simply means that the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently. There are two mechanisms by which this can be achieved.

Blocking Non-Buffered Send/Receive

In the first case, the send operation does not return until the matching receive has been encountered at the receiving process. When this happens, the message is sent and the send operation returns upon completion of the communication operation. Typically, this process involves a handshake between the sending and receiving processes. The sending process sends a request to communicate to the receiving process. When the receiving process encounters the target receive, it responds to the request. The sending process upon receiving this response initiates a transfer operation. The operation is illustrated in Figure 6.1 . Since there are no buffers used at either sending or receiving ends, this is also referred to as a *non-buffered blocking operation* .

Figure 6.1. Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.



Idling Overheads in Blocking Non-Buffered Operations In Figure 6.1 , we illustrate three scenarios in which the send is reached before the receive is posted, the send and receive are posted around the same time, and the receive is posted before the send is reached. In cases (a) and (c), we notice that there is considerable idling at the sending and receiving process. It is also clear from the figures that a blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time. However, in an asynchronous environment, this may be impossible to predict. This idling overhead is one of the major drawbacks of this protocol.

Deadlocks in Blocking Non-Buffered Operations Consider the following simple exchange of messages that can lead to a deadlock:

```
1          P0                                P1
2
3          send(&a, 1, 1);                   send(&a, 1, 0);
4          receive(&b, 1, 1);                receive(&b, 1, 0);
```

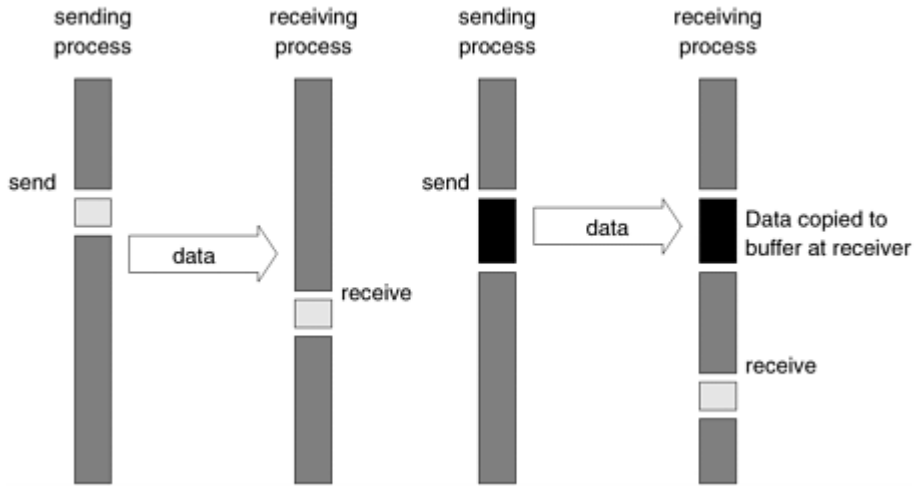
The code fragment makes the values of a available to both processes P0 and P1. However, if the send and receive operations are implemented using a blocking non-buffered protocol, the send at P0 waits for the matching receive at P1 whereas the send at process P1 waits for the corresponding receive at P0, resulting in an infinite wait.

As can be inferred, deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined. In the above example, this can be corrected by replacing the operation sequence of one of the processes by a `receive` and a `send` as opposed to the other way around. This often makes the code more cumbersome and buggy.

Blocking Buffered Send/Receive

A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends. We start with a simple case in which the sender has a buffer pre-allocated for communicating messages. On encountering a send operation, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The sender process can now continue with the program knowing that any changes to the data will not impact program semantics. The actual communication can be accomplished in many ways depending on the available hardware resources. If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer. Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics. Instead, the data is copied into a buffer at the receiver as well. When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer. If so, the data is copied into the target location. This operation is illustrated in Figure 6.2(a) .

Figure 6.2. Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.



In the protocol illustrated above, buffers are used at both sender and receiver and communication is handled by dedicated hardware. Sometimes machines do not have such communication hardware. In this case, some of the overhead can be saved by buffering only on

one side. For example, on encountering a send operation, the sender interrupts the receiver, both processes participate in a communication operation and the message is deposited in a buffer at the receiver end. When the receiver eventually encounters a receive operation, the message is copied from the buffer into the target location. This protocol is illustrated in Figure 6.2(b) . It is not difficult to conceive a protocol in which the buffering is done only at the sender and the receiver initiates a transfer by interrupting the sender.

It is easy to see that buffered protocols alleviate idling overheads at the cost of adding buffer management overheads. In general, if the parallel program is highly synchronous (i.e., sends and receives are posted around the same time), non-buffered sends may perform better than buffered sends. However, in general applications, this is not the case and buffered sends are desirable unless buffer capacity becomes an issue.

Example 6.1 Impact of finite buffers in message passing

Consider the following code fragment:

```
1          P0                                    P1
2
3          for (i = 0; i < 1000; i++) {          for (i = 0; i < 1000; i++) {
4            produce_data(&a);                     receive(&a, 1, 0);
5            send(&a, 1, 1);                       consume_data(&a);
6          }                                     }
```

In this code fragment, process P0 produces 1000 data items and process P1 consumes them. However, if process P1 was slow getting to this loop, process P0 might have sent all of its data. If there is enough buffer space, then both processes can proceed; however, if the buffer is not sufficient (i.e., buffer overflow), the sender would have to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space. This can often lead to unforeseen overheads and performance degradation. In general, it is a good idea to write programs that have bounded buffer requirements. ■

Deadlocks in Buffered Send and Receive Operations While buffering alleviates many of the deadlock situations, it is still possible to write code that deadlocks. This is due to the fact that as in the non-buffered case, receive calls are always blocking (to ensure semantic consistency). Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

```
1          P0                                    P1
2
3          receive(&a, 1, 1);                    receive(&a, 1, 0);
4          send(&b, 1, 1);                       send(&b, 1, 0);
```

Once again, such circular waits have to be broken. However, deadlocks are caused only by waits on receive operations in this case.

6.2.2 Non-Blocking Message Passing Operations

In blocking protocols, the overhead of guaranteeing semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered). Often, it is possible to require the

programmer to ensure semantic correctness and provide a fast send/receive operation that incurs little overhead. This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so. Consequently, the user must be careful not to alter data that may be potentially participating in a communication operation. Non-blocking operations are generally accompanied by a `check-status` operation, which indicates whether the semantics of a previously initiated transfer may be violated or not. Upon return from a non-blocking send or receive operation, the process is free to perform any computation that does not depend upon the completion of the operation. Later in the program, the process can check whether or not the non-blocking operation has completed, and, if necessary, wait for its completion.

As illustrated in Figure 6.3 , non-blocking operations can themselves be buffered or non-buffered. In the non-buffered case, a process wishing to send data to another simply posts a pending message and returns to the user program. The program can then do other useful work. At some point in the future, when the corresponding receive is posted, the communication operation is initiated. When this operation is completed, the check-status operation indicates that it is safe for the programmer to touch this data. This transfer is indicated in Figure 6.4(a) .

Figure 6.3. Space of possible protocols for send and receive operations.

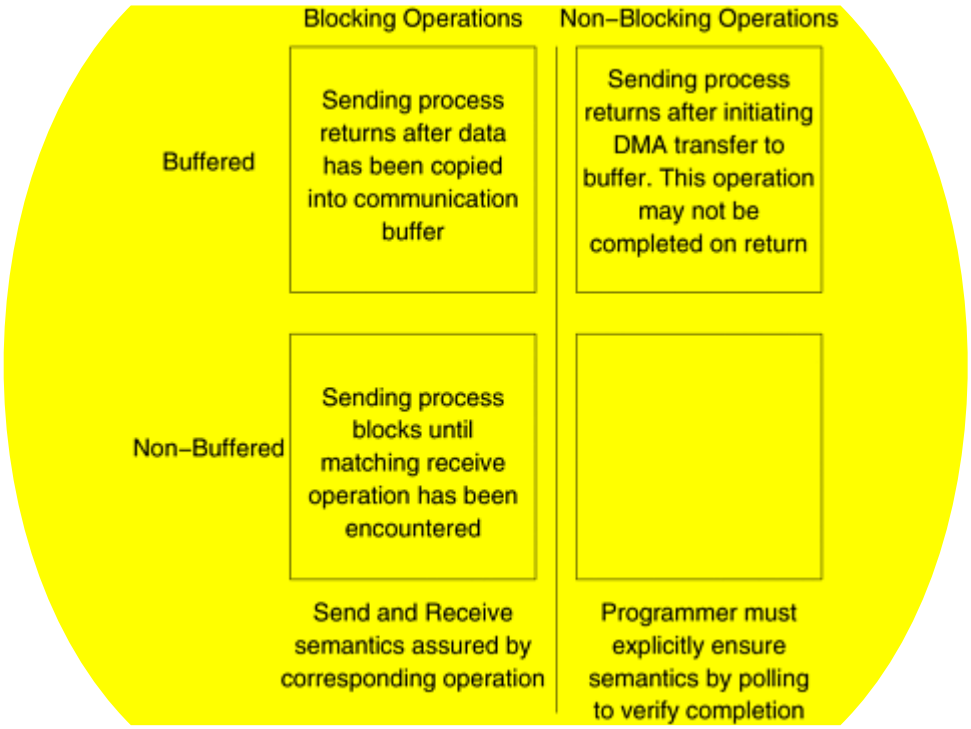
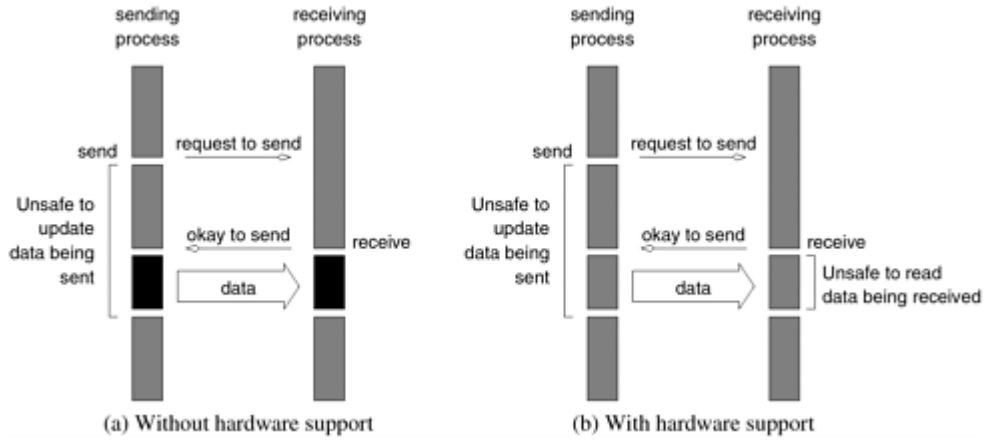


Figure 6.4. Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.



Comparing Figures 6.4(a) and 6.1(a) , it is easy to see that the idling time when the process is waiting for the corresponding receive in a blocking operation can now be utilized for computation, provided it does not update the data being sent. This alleviates the major bottleneck associated with the former at the expense of some program restructuring. The benefits of non-blocking operations are further enhanced by the presence of dedicated communication hardware. This is illustrated in Figure 6.4(b) . In this case, the communication overhead can be almost entirely masked by non-blocking operations. In this case, however, the data being received is unsafe for the duration of the receive operation.

Non-blocking operations can also be used with a buffered protocol. In this case, the sender initiates a DMA operation and returns immediately. The data becomes safe the moment the DMA operation has been completed. At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location. Using buffers with non-blocking operation has the effect of reducing the time during which the data is unsafe.

Typical message-passing libraries such as Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) implement both blocking and non-blocking operations. Blocking operations facilitate safe and easier programming and non-blocking operations are useful for performance optimization by masking communication overhead. One must, however, be careful using non-blocking protocols since errors can result from unsafe access to data that is in the process of being communicated.

PREVIOUS NEXT

6.3 MPI: the Message Passing Interface

Many early generation commercial parallel computers were based on the message-passing architecture due to its lower cost relative to shared-address-space architectures. Since message-passing is the natural programming paradigm for these machines, this resulted in the development of many different message-passing libraries. In fact, message-passing became the modern-age form of assembly language, in which every hardware vendor provided its own library, that performed very well on its own hardware, but was incompatible with the parallel computers offered by other vendors. Many of the differences between the various vendor-specific message-passing libraries were only syntactic; however, often enough there were some serious semantic differences that required significant re-engineering to port a message-passing program from one library to another.

The message-passing interface, or MPI as it is commonly known, was created to essentially solve this problem. MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran. The MPI standard defines both the syntax as well as the semantics of a core set of library routines that are very useful in writing message-passing programs. MPI was developed by a group of researchers from academia and industry, and has enjoyed wide support by almost all the hardware vendors. Vendor implementations of MPI are available on almost all commercial parallel computers.

The MPI library contains over 125 routines, but the number of key concepts is much smaller. In fact, it is possible to write fully-functional message-passing programs by using only the six routines shown in Table 6.1 . These routines are used to initialize and terminate the MPI library, to get information about the parallel computing environment, and to send and receive messages.

In this section we describe these routines as well as some basic concepts that are essential in writing correct and efficient message-passing programs using MPI.

`MPI_Init`

Initializes MPI.

`MPI_Finalize`

Terminates MPI.

`MPI_Comm_size`

Determines the number of processes.

`MPI_Comm_rank`

Determines the label of the calling process.

`MPI_Send`

Sends a message.

`MPI_Recv`

Receives a message.

Table 6.1. The minimal set of MPI routines.

6.3.1 Starting and Terminating the MPI Library

`MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment. Calling `MPI_Init` more than once during the execution of a program will lead to an error. `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment. No MPI calls may be performed after `MPI_Finalize` has been called, not even `MPI_Init` . Both `MPI_Init` and `MPI_Finalize` must be called by all the processes, otherwise MPI's behavior will be undefined. The exact calling sequences of these two routines for C are as follows:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

The arguments `argc` and `argv` of `MPI_Init` are the command-line arguments of the C program. An MPI implementation is expected to remove from the `argv` array any command-line arguments that should be processed by the implementation before returning back to the program, and to decrement `argc` accordingly. Thus, command-line processing should be performed only after `MPI_Init` has been called. Upon successful execution, `MPI_Init` and `MPI_Finalize` return `MPI_SUCCESS` ; otherwise they return an implementation-defined error code.

The bindings and calling sequences of these two functions are illustrative of the naming practices and argument conventions followed by MPI. All MPI routines, data-types, and constants are prefixed by "`MPI_` ". The return code for successful completion is `MPI_SUCCESS` . This and other MPI constants and data-structures are defined for C in the file `"mpi.h"` . This header file must be included in each MPI program.

6.3.2 Communicators

A key concept used throughout MPI is that of the *communication domain* . A communication domain is a set of processes that are allowed to communicate with each other. Information about communication domains is stored in variables of type `MPI_Comm` ,that are called *communicators* . These communicators are used as arguments to all message transfer MPI routines and they uniquely identify the processes participating in the message transfer operation. Note that each process can belong to many different (possibly overlapping) communication domains.

The communicator is used to define a set of processes that can communicate with each other. This set of processes form a *communication domain* . In general, all the processes may need to communicate with each other. For this reason, MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes involved in the parallel execution. However, in many cases we want to perform communication only within (possibly overlapping) groups of processes. By using a different communicator for each such group, we can ensure that no messages will ever interfere with messages destined to any other group. How to create and use such communicators is described at a later point in this chapter. For now, it suffices to use `MPI_COMM_WORLD` as the communicator argument to all the MPI functions that require a communicator.

6.3.3 Getting Information

The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively. The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

The function `MPI_Comm_size` returns in the variable `size` the number of processes that belong to the communicator `comm` . So, when there is a single process per processor, the call `MPI_Comm_size(MPI_COMM_WORLD, &size)` will return in `size` the number of processors used by the program. Every process that belongs to a communicator is uniquely identified by its *rank* . The rank of a process is an integer that ranges from zero up to the size of the communicator minus one. A process can determine its rank in a communicator by using the `MPI_Comm_rank` function that takes two arguments: the communicator and an integer variable `rank` . Up on return, the variable `rank` stores the rank of the process. Note that each process that calls either one of these functions must belong in the supplied communicator, otherwise an error will occur.

Example 6.2 Hello World

We can use the four MPI functions just described to write a program that prints out a "Hello World" message from each processor.

```
1   #include <mpi.h>
2
3   main(int argc, char *argv[])
4   {
5     int npes, myrank;
6
7     MPI_Init(&argc, &argv);
8      MPI_Comm_size(MPI_COMM_WORLD, &npes);
9      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10    printf("From process %d out of %d, Hello World!\n",
11            myrank, npes);
12    MPI_Finalize();
13  }
```

■

6.3.4 Sending and Receiving Messages

The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv` , respectively. The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
         int dest, int tag, MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
         int source, int tag, MPI_Comm comm, MPI_Status *status)
```

MPI_Send sends the data stored in the buffer pointed by buf . This buffer consists of consecutive entries of the type specified by the parameter datatype . The number of entries in the buffer is given by the parameter count . The correspondence between MPI datatypes and those provided by C is shown in Table 6.2 . Note that for all C datatypes, an equivalent MPI datatype is provided. However, MPI allows two additional datatypes that are not part of the C language. These are MPI_BYTE and MPI_PACKED .

MPI_BYTE corresponds to a byte (8 bits) and MPI_PACKED corresponds to a collection of data items that has been created by packing non-contiguous data. Note that the length of the message in MPI_Send , as well as in other MPI routines, is specified in terms of the number of entries being sent and not in terms of the number of bytes. Specifying the length in terms of the number of entries has the advantage of making the MPI code portable, since the number of bytes used to store various datatypes can be different for different architectures.

The destination of the message sent by MPI_Send is uniquely specified by the dest and comm arguments. The dest argument is the rank of the destination process in the communication domain specified by the communicator comm . Each message has an integer-valued tag associated with it. This is used to distinguish different types of messages. The message-tag can take values ranging from zero up to the MPI defined constant MPI_TAG_UB . Even though the value of MPI_TAG_UB is implementation specific, it is at least 32,767.

MPI_Recv receives a message sent by a process whose rank is given by the source in the communication domain specified by the comm argument. The tag of the sent message must be that specified by the tag argument. If there are many messages with identical tag from the same process, then any one of these messages is received. MPI allows specification of wildcard arguments for both source and tag . If source is set to MPI_ANY_SOURCE , then any process of the communication domain can be the source of the message. Similarly, if tag is set to MPI_ANY_TAG , then messages with any tag are accepted. The received message is stored in continuous locations in the buffer pointed to by buf . The count and datatype arguments of MPI_Recv are used to specify the length of the supplied buffer. The received message should be of length equal to or less than this length. This allows the receiving process to not know the exact size of the message being sent. If the received message is larger than the supplied buffer, then an overflow error will occur, and the routine will return the error MPI_ERR_TRUNCATE .

MPI_CHAR

signed char

MPI_SHORT

signed short int

MPI_INT

signed int

MPI_LONG

signed long int

MPI_UNSIGNED_CHAR

unsigned char

MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Table 6.2. Correspondence between the datatypes supported by MPI and those supported by C.

After a message has been received, the `status` variable can be used to get information about the `MPI_Recv` operation. In C, `status` is stored using the `MPI_Status` data-structure. This is implemented as a structure with three fields, as follows:

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
};
```

`MPI_SOURCE` and `MPI_TAG` store the source and the tag of the received message. They are particularly useful when `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are used for the `source` and `tag` arguments. `MPI_ERROR` stores the error-code of the received message.

The status argument also returns information about the length of the received message. This information is not directly accessible from the `status` variable, but it can be retrieved by calling the `MPI_Get_count` function. The calling sequence of this function is as follows:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
        int *count)
```

`MPI_Get_count` takes as arguments the `status` returned by `MPI_Recv` and the type of the received data in `datatype` , and returns the number of entries that were actually received in the `count` variable.

The `MPI_Recv` returns only after the requested message has been received and copied into the buffer. That is, `MPI_Recv` is a blocking receive operation. However, MPI allows two different implementations for `MPI_Send` . In the first implementation, `MPI_Send` returns only after the corresponding `MPI_Recv` have been issued and the message has been sent to the receiver. In the second implementation, `MPI_Send` first copies the message into a buffer and then returns, without waiting for the corresponding `MPI_Recv` to be executed. In either implementation, the buffer that is pointed by the `buf` argument of `MPI_Send` can be safely reused and overwritten. MPI programs must be able to run correctly regardless of which of the two methods is used for implementing `MPI_Send` . Such programs are called *safe* . In writing safe MPI programs, sometimes it is helpful to forget about the alternate implementation of `MPI_Send` and just think of it as being a blocking send operation.

Avoiding Deadlocks The semantics of `MPI_Send` and `MPI_Recv` place some restrictions on how we can mix and match send and receive operations. For example, consider the following piece of code in which process 0 sends two messages with different tags to process 1, and process 1 receives them in the reverse order.

```
1    int a[10], b[10], myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
5    if (myrank == 0) {
6      MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
7      MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
8    }
9    else if (myrank == 1) {
10     MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
11     MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
12   }
13   ...
```

If `MPI_Send` is implemented using buffering, then this code will run correctly provided that sufficient buffer space is available. However, if `MPI_Send` is implemented by blocking until the matching receive has been issued, then neither of the two processes will be able to proceed. This is because process zero (i.e., `myrank == 0`) will wait until process one issues the matching `MPI_Recv` (i.e., the one with `tag` equal to 1), and at the same time process one will wait until process zero performs the matching `MPI_Send` (i.e., the one with `tag` equal to 2). This code fragment is not safe, as its behavior is implementation dependent. It is up to the programmer to ensure that his or her program will run correctly on any MPI implementation. The problem in this program can be corrected by *matching the order in which the send and receive operations are issued* . Similar deadlock situations can also occur when a process sends a message to itself. Even though this is legal, its behavior is implementation dependent and must be avoided.

Improper use of `MPI_Send` and `MPI_Recv` can also lead to deadlocks in situations when each processor needs to send and receive a message in a circular fashion. Consider the following piece of code, in which process *i* sends a message to process *i* + 1 (modulo the number of processes) and receives a message from process *i* - 1 (module the number of processes).

```
1    int a[10], b[10], npes, myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_size(MPI_COMM_WORLD, &npes);
5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
7    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
8    ...
```

When MPI_Send is implemented using buffering, the program will work correctly, since every call to MPI_Send will get buffered, allowing the call of the MPI_Recv to be performed, which will transfer the required data. However, if MPI_Send blocks until the matching receive has been issued, all processes will enter an infinite wait state, waiting for the neighboring process to issue a MPI_Recv operation. Note that the deadlock still remains even when we have only two processes. Thus, when pairs of processes need to exchange data, the above method leads to an unsafe program. The above example can be made safe, by rewriting it as follows:

```
1    int a[10], b[10], npes, myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_size(MPI_COMM_WORLD, &npes);
5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6    if (myrank%2 == 1) {
7       MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
8       MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
9    }
10   else {
11      MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
12      MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
13   }
14   ...
```

This new implementation partitions the processes into two groups. One consists of the odd-numbered processes and the other of the even-numbered processes. The odd-numbered processes perform a send followed by a receive, and the even-numbered processes perform a receive followed by a send. Thus, when an odd-numbered process calls MPI_Send ,the target process (which has an even number) will call MPI_Recv to receive that message, before attempting to send its own message.

Sending and Receiving Messages Simultaneously The above communication pattern appears frequently in many message-passing programs, and for this reason MPI provides the MPI_Sendrecv function that both sends and receives a message.

MPI_Sendrecv does not suffer from the circular deadlock problems of MPI_Send and MPI_Recv . You can think of MPI_Sendrecv as allowing data to travel for both send and receive simultaneously. The calling sequence of MPI_Sendrecv is the following:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
          MPI_Datatype senddatatype, int dest, int sendtag,
          void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
          int source, int recvtag, MPI_Comm comm,
          MPI_Status *status)
```

The arguments of MPI_Sendrecv are essentially the combination of the arguments of MPI_Send and MPI_Recv . The send and receive buffers must be disjoint, and the source and destination of

the messages can be the same or different. The safe version of our earlier example using MPI_Sendrecv is as follows.

```
1    int a[10], b[10], npes, myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_size(MPI_COMM_WORLD, &npes);
5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6    MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
7                  b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
8                 MPI_COMM_WORLD, &status);
9    ...
```

In many programs, the requirement for the send and receive buffers of MPI_Sendrecv be disjoint may force us to use a temporary buffer. This increases the amount of memory required by the program and also increases the overall run time due to the extra copy. This problem can be solved by using that MPI_Sendrecv_replace MPI function. This function performs a blocking send and receive, but it uses a single buffer for both the send and receive operation. That is, the received data replaces the data that was sent out of the buffer. The calling sequence of this function is the following:

```
int MPI_Sendrecv_replace(void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

Note that both the send and receive operations must transfer data of the same datatype.

6.3.5 Example: Odd-Even Sort

We will now use the MPI functions described in the previous sections to write a complete message-passing program that will sort a list of numbers using the odd-even sorting algorithm. Recall from Section 9.3.1 that the odd-even sorting algorithm sorts a sequence of n elements using p processes in a total of p phases. During each of these phases, the odd-or even-numbered processes perform a compare-split step with their right neighbors. The MPI program for performing the odd-even sort in parallel is shown in Program 6.1 . To simplify the presentation, this program assumes that n is divisible by p .

Program 6.1 Odd-Even Sorting

[View full width]

```
 1  #include <stdlib.h>
 2  #include <mpi.h> /* Include MPI's header file */
 3
 4  main(int argc, char *argv[])
 5  {
 6    int n;         /* The total number of elements to be sorted */
 7    int npes;      /* The total number of processes */
 8    int myrank;    /* The rank of the calling process */
 9    int nlocal;    /* The local number of elements, and the array that stores th
10    int *elmnts;   /* The array that stores the local elements */
11    int *relmnts;  /* The array that stores the received elements */
```

```
12    int oddrank;   /* The rank of the process during odd-phase communication */
13    int evenrank;  /* The rank of the process during even-phase communication */
14    int *wspace;   /* Working space during the compare-split operation */
15    int i;
16    MPI_Status status;
17
18    /* Initialize MPI and get system information */
19    MPI_Init(&argc, &argv);
20    MPI_Comm_size(MPI_COMM_WORLD, &npes);
21    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
22
23    n = atoi(argv[1]);
24    nlocal = n/npes; /* Compute the number of elements to be stored locally. */
25
26    /* Allocate memory for the various arrays */
27    elmnts  = (int *)malloc(nlocal*sizeof(int));
28    relmnts = (int *)malloc(nlocal*sizeof(int));
29    wspace  = (int *)malloc(nlocal*sizeof(int));
30
31    /* Fill-in the elmnts array with random elements */
32    srandom(myrank);
33    for (i=0; i<nlocal; i++)
34      elmnts[i] = random();
35
36    /* Sort the local elements using the built-in quicksort routine */
37    qsort(elmnts, nlocal, sizeof(int), IncOrder);
38
39    /* Determine the rank of the processors that myrank needs to communicate dur
➥ the */
40    /* odd and even phases of the algorithm */
41    if (myrank%2 == 0) {
42      oddrank  = myrank-1;
43      evenrank = myrank+1;
44    }
45    else {
46      oddrank  = myrank+1;
47      evenrank = myrank-1;
48    }
49
50    /* Set the ranks of the processors at the end of the linear */
51    if (oddrank == -1 || oddrank == npes)
52      oddrank = MPI_PROC_NULL;
53    if (evenrank == -1 || evenrank == npes)
54      evenrank = MPI_PROC_NULL;
55
56    /* Get into the main loop of the odd-even sorting algorithm */
57    for (i=0; i<npes-1; i++) {
58      if (i%2 == 1) /* Odd phase */
59        MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
60            nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
61      else /* Even phase */
62        MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
63            nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);
64
65      CompareSplit(nlocal, elmnts, relmnts, wspace,
```

```
66                    myrank < status.MPI_SOURCE);
67    }
68
69     free(elmnts); free(relmnts); free(wspace);
70    MPI_Finalize();
71  }
72
73  /* This is the CompareSplit function */
74   CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
75                int keepsmall)
76  {
77    int i, j, k;
78
79    for (i=0; i<nlocal; i++)
80      wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */
81
82    if (keepsmall) { /* Keep the nlocal smaller elements */
83      for (i=j=k=0; k<nlocal; k++) {
84        if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
85          elmnts[k] = wspace[i++];
86        else
87          elmnts[k] = relmnts[j++];
88      }
89    }
90    else { /* Keep the nlocal larger elements */
91       for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
92         if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
93           elmnts[k] = wspace[i--];
94         else
95           elmnts[k] = relmnts[j--];
96      }
97    }
98  }
99
100 /* The IncOrder function that is called by qsort is defined as follows */
101 int IncOrder(const void *e1, const void *e2)
102 {
103   return (*((int *)e1) - *((int *)e2));
104 }
```


PREVIOUS NEXT

6.6 Collective Communication and Computation Operations

MPI provides an extensive set of functions for performing many commonly used collective communication operations. In particular, the majority of the basic communication operations described in Chapter 4 are supported by MPI. All of the collective communication functions provided by MPI take as an argument a communicator that defines the group of processes that participate in the collective operation. All the processes that belong to this communicator participate in the operation, and all of them must call the collective communication function. Even though collective communication operations do not act like barriers (i.e., it is possible for a processor to go past its call for the collective communication operation even before other processes have reached it), it acts like a *virtual* synchronization step in the following sense: the parallel program should be written such that it behaves correctly even if a global synchronization is performed before and after the collective call. Since the operations are virtually synchronous, they do not require tags. In some of the collective functions data is required to be sent from a single process (source-process) or to be received by a single process (target-process). In these functions, the source- or target-process is one of the arguments supplied to the routines. All the processes in the group (i.e., communicator) must specify the same source- or target-process. For most collective communication operations, MPI provides two different variants. The first transfers equal-size data to or from each process, and the second transfers data that can be of different sizes.

6.6.1 Barrier

The barrier synchronization operation is performed in MPI using the `MPI_Barrier` function.

```
int MPI_Barrier(MPI_Comm comm)
```

The only argument of `MPI_Barrier` is the communicator that defines the group of processes that are synchronized. The call to `MPI_Barrier` returns only after all the processes in the group have called this function.

6.6.2 Broadcast

The one-to-all broadcast operation described in Section 4.1 is performed in MPI using the `MPI_Bcast` function.

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int source, MPI_Comm comm)
```

`MPI_Bcast` sends the data stored in the buffer `buf` of process `source` to all the other processes in the group. The data received by each process is stored in the buffer `buf` . The data that is broadcast consist of `count` entries of type `datatype` . The amount of data sent by the `source` process must be equal to the amount of data that is being received by each process; i.e., the `count` and `datatype` fields must match on all processes.

6.6.3 Reduction

The all-to-one reduction operation described in Section 4.1 is performed in MPI using the `MPI_Reduce` function.

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
         MPI_Datatype datatype, MPI_Op op, int target,
         MPI_Comm comm)
```

`MPI_Reduce` combines the elements stored in the buffer `sendbuf` of each process in the group, using the operation specified in `op` , and returns the combined values in the buffer `recvbuf` of the process with rank `target` . Both the `sendbuf` and `recvbuf` must have the same number of `count` items of type `datatype` . Note that all processes must provide a `recvbuf` array, even if they are not the *target* of the reduction operation. When `count` is more than one, then the combine operation is applied element-wise on each entry of the sequence. All the processes must call `MPI_Reduce` with the same value for `count` , `datatype` , `op` , `target` , and `comm` .

MPI provides a list of predefined operations that can be used to combine the elements stored in `sendbuf` . MPI also allows programmers to define their own operations, which is not covered in this book. The predefined operations are shown in Table 6.3 . For example, in order to compute the maximum of the elements stored in `sendbuf` , the `MPI_MAX` value must be used for the `op` argument. Not all of these operations can be applied to all possible data-types supported by MPI. For example, a bit-wise OR operation (i.e., *op* = `MPI_BOR`) is not defined for real-valued data-types such as `MPI_FLOAT` and `MPI_REAL` . The last column of Table 6.3 shows the various data-types that can be used with each operation.

`MPI_MAX`

Maximum

C integers and floating point

`MPI_MIN`

Minimum

C integers and floating point

`MPI_SUM`

Sum

C integers and floating point

`MPI_PROD`

Product

C integers and floating point

`MPI_LAND`

Logical AND

C integers

MPI_BAND

Bit-wise AND

C integers and byte

MPI_LOR

Logical OR

C integers

MPI_BOR

Bit-wise OR

C integers and byte

MPI_LXOR

Logical XOR

C integers

MPI_BXOR

Bit-wise XOR

C integers and byte

MPI_MAXLOC

max-min value-location

Data-pairs

MPI_MINLOC

min-min value-location

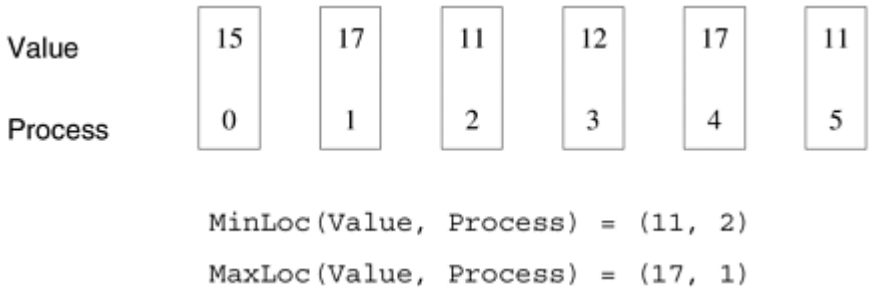
Data-pairs

Table 6.3. Predefined reduction operations.

Operation	Meaning	Datatypes

The operation MPI_MAXLOC combines pairs of values (v_i , l_i) and returns the pair (v , l) such that v is the maximum among all v_i 's and l is the smallest among all l_i 's such that $v = v_i$. Similarly, MPI_MINLOC combines pairs of values and returns the pair (v , l) such that v is the minimum among all v_i 's and l is the smallest among all l_i 's such that $v = v_i$. One possible application of MPI_MAXLOC or MPI_MINLOC is to compute the maximum or minimum of a list of numbers each residing on a different process and also the rank of the first process that stores this maximum or minimum, as illustrated in Figure 6.6 . Since both MPI_MAXLOC and MPI_MINLOC require datatypes that correspond to pairs of values, a new set of MPI datatypes have been defined as shown in Table 6.4 . In C, these datatypes are implemented as structures containing the corresponding types.

Figure 6.6. An example use of the **MPI_MINLOC** and **MPI_MAXLOC** operators.



When the result of the reduction operation is needed by all the processes, MPI provides the `MPI_Allreduce` operation that returns the result to all the processes. This function provides the functionality of the all-reduce operation described in Section 4.3 .

`MPI_2INT`

pair of `int` s

`MPI_SHORT_INT`

`short` and `int`

`MPI_LONG_INT`

`long` and `int`

`MPI_LONG_DOUBLE_INT`

`long double` and `int`

`MPI_FLOAT_INT`

`float` and `int`

`MPI_DOUBLE_INT`

`double` and `int`

Table 6.4. MPI datatypes for data-pairs used with the **MPI_MAXLOC** and **MPI_MINLOC** reduction operations.

MPI Datatype	C Datatype

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Note that there is no `target` argument since all processes receive the result of the operation.

6.6.4 Prefix

The prefix-sum operation described in Section 4.3 is performed in MPI using the `MPI_Scan` function.

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI_Scan performs a prefix reduction of the data stored in the buffer sendbuf at each process and returns the result in the buffer recvbuf . The receive buffer of the process with rank *i* will store, at the end of the operation, the reduction of the send buffers of the processes whose ranks range from 0 up to and including *i* . The type of supported operations (i.e., op) as well as the restrictions on the various arguments of MPI_Scan are the same as those for the reduction operation MPI_Reduce .

6.6.5 Gather

The gather operation described in Section 4.4 is performed in MPI using the MPI_Gather function.

```
int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)
```

Each process, including the target process, sends the data stored in the array sendbuf to the target process. As a result, if *p* is the number of processors in the communication comm , the target process receives a total of *p* buffers. The data is stored in the array recvbuf of the target process, in a rank order. That is, the data from process with rank *i* are stored in the recvbuf starting at location *i* * sendcount (assuming that the array recvbuf is of the same type as recvdatatype).

The data sent by each process must be of the same size and type. That is, MPI_Gather must be called with the sendcount and senddatatype arguments having the same values at each process. The information about the receive buffer, its length and type applies only for the target process and is ignored for all the other processes. The argument recvcount specifies the number of elements received by each process and not the total number of elements it receives. So, recvcount must be the same as sendcount and their datatypes must be matching.

MPI also provides the MPI_Allgather function in which the data are gathered to all the processes and not only at the target process.

```
int MPI_Allgather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

The meanings of the various parameters are similar to those for MPI_Gather ; however, each process must now supply a recvbuf array that will store the gathered data.

In addition to the above versions of the gather operation, in which the sizes of the arrays sent by each process are the same, MPI also provides versions in which the size of the arrays can be different. MPI refers to these operations as the *vector* variants. The vector variants of the MPI_Gather and MPI_Allgather operations are provided by the functions MPI_Gatherv and MPI_Allgatherv , respectively.

```
int MPI_Gatherv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int *recvcounts, int *displs,
```

```
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)

int MPI_Allgatherv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int *recvcounts, int *displs, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```

These functions allow a different number of data elements to be sent by each process by replacing the `recvcount` parameter with the array `recvcounts` . The amount of data sent by process *i* is equal to `recvcounts[i]` . Note that the size of `recvcounts` is equal to the size of the communicator `comm` . The array parameter `displs` , which is also of the same size, is used to determine where in `recvbuf` the data sent by each process will be stored. In particular, the data sent by process *i* are stored in `recvbuf` starting at location `displs[i]` . Note that, as opposed to the non-vector variants, the `sendcount` parameter can be different for different processes.

6.6.6 Scatter

The scatter operation described in Section 4.4 is performed in MPI using the `MPI_Scatter` function.

```
int MPI_Scatter(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int source, MPI_Comm comm)
```

The `source` process sends a different part of the send buffer `sendbuf` to each processes, including itself. The data that are received are stored in `recvbuf` . Process *i* receives `sendcount` contiguous elements of type `senddatatype` starting from the *i* * `sendcount` location of the `sendbuf` of the source process (assuming that `sendbuf` is of the same type as `senddatatype`). `MPI_Scatter` must be called by all the processes with the same values for the `sendcount` , `senddatatype` , `recvcount` , `recvdatatype` , `source` , and `comm` arguments. Note again that `sendcount` is the number of elements sent to each individual process.

Similarly to the gather operation, MPI provides a vector variant of the scatter operation, called `MPI_Scatterv` , that allows different amounts of data to be sent to different processes.

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int source, MPI_Comm comm)
```

As we can see, the parameter `sendcount` has been replaced by the array `sendcounts` that determines the number of elements to be sent to each process. In particular, the `target` process sends `sendcounts[i]` elements to process *i* . Also, the array `displs` is used to determine where in `sendbuf` these elements will be sent from. In particular, if `sendbuf` is of the same type is `senddatatype` , the data sent to process *i* start at location `displs[i]` of array `sendbuf` . Both the `sendcounts` and `displs` arrays are of size equal to the number of processes in the communicator. Note that by appropriately setting the `displs` array we can use `MPI_Scatterv` to send overlapping regions of `sendbuf` .

6.6.7 All-to-All

The all-to-all personalized communication operation described in Section 4.5 is performed in MPI by using the `MPI_Alltoall` function.

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

Each process sends a different portion of the `sendbuf` array to each other process, including itself. Each process sends to process *i* `sendcount` contiguous elements of type `senddatatype` starting from the *i* * `sendcount` location of its `sendbuf` array. The data that are received are stored in the `recvbuf` array. Each process receives from process *i* `recvcount` elements of type `recvdatatype` and stores them in its `recvbuf` array starting at location *i* * `recvcount` . `MPI_Alltoall` must be called by all the processes with the same values for the `sendcount` , `senddatatype` , `recvcount` , `recvdatatype` , and `comm` arguments. Note that `sendcount` and `recvcount` are the number of elements sent to, and received from, each individual process.

MPI also provides a vector variant of the all-to-all personalized communication operation called `MPI_Alltoallv` that allows different amounts of data to be sent to and received from each process.

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls
        MPI_Datatype senddatatype, void *recvbuf, int *recvcounts,
        int *rdispls, MPI_Datatype recvdatatype, MPI_Comm comm)
```

The parameter `sendcounts` is used to specify the number of elements sent to each process, and the parameter `sdispls` is used to specify the location in `sendbuf` in which these elements are stored. In particular, each process sends to process *i* , starting at location `sdispls[i]` of the array `sendbuf` , `sendcounts[i]` contiguous elements. The parameter `recvcounts` is used to specify the number of elements received by each process, and the parameter `rdispls` is used to specify the location in `recvbuf` in which these elements are stored. In particular, each process receives from process *i* `recvcounts[i]` elements that are stored in contiguous locations of `recvbuf` starting at location `rdispls[i]` . `MPI_Alltoallv` must be called by all the processes with the same values for the `senddatatype` , `recvdatatype` , and `comm` arguments.

6.6.8 Example: One-Dimensional Matrix-Vector Multiplication

Our first message-passing program using collective communications will be to multiply a dense *n* x *n* matrix *A* with a vector *b* , i.e., *x* = *Ab* . As discussed in Section 8.1 , one way of performing this multiplication in parallel is to have each process compute different portions of the product-vector *x* . In particular, each one of the *p* processes is responsible for computing *n* /*p* consecutive elements of *x* . This algorithm can be implemented in MPI by distributing the matrix *A* in a row-wise fashion, such that each process receives the *n* /*p* rows that correspond to the portion of the product-vector *x* it computes. Vector *b* is distributed in a fashion similar to *x* .

Program 6.4 shows the MPI program that uses a row-wise distribution of matrix *A* . The dimension of the matrices is supplied in the parameter `n` , the parameters `a` and `b` point to the locally stored portions of matrix *A* and vector *b* , respectively, and the parameter `x` points to the local portion of the output matrix-vector product. This program assumes that *n* is a multiple of the number of processors.

Program 6.4 Row-wise Matrix-Vector Multiplication

```
1   RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
2                           MPI_Comm comm)
```

```
{
  int i, j;
  int nlocal;         /* Number of locally stored rows of A */
  double *fb;         /* Will point to a buffer that stores the entire vector b
  int npes, myrank;
  MPI_Status status;

  /* Get information about the communicator */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Allocate the memory that will store the entire vector b */
  fb = (double *)malloc(n*sizeof(double));

  nlocal = n/npes;

  /* Gather the entire vector b on each processor using MPI's ALLGATHER operat
  MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
      comm);

  /* Perform the matrix-vector multiplication involving the locally stored sub
  for (i=0; i<nlocal; i++) {
    x[i] = 0.0;
    for (j=0; j<n; j++)
      x[i] += a[i*n+j]*fb[j];
  }

  free(fb);
}
```

An alternate way of computing x is to parallelize the task of performing the dot-product for each element of x . That is, for each element x_i , of vector x , all the processes will compute a part of it, and the result will be obtained by adding up these partial dot-products. This algorithm can be implemented in MPI by distributing matrix A in a column-wise fashion. Each process gets n/p consecutive columns of A , and the elements of vector b that correspond to these columns. Furthermore, at the end of the computation we want the product-vector x to be distributed in a fashion similar to vector b . Program 6.5 shows the MPI program that implements this column-wise distribution of the matrix.

Program 6.5 Column-wise Matrix-Vector Multiplication

```
ColMatrixVectorMultiply(int n, double *a, double *b, double *x,
                        MPI_Comm comm)
{
  int i, j;
  int nlocal;
  double *px;
  double *fx;
  int npes, myrank;
  MPI_Status status;

  /* Get identity and size information from the communicator */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);
```

```
14
15      nlocal = n/npes;
16
17      /* Allocate memory for arrays storing intermediate results. */
18      px = (double *)malloc(n*sizeof(double));
19      fx = (double *)malloc(n*sizeof(double));
20
21      /* Compute the partial-dot products that correspond to the local columns of
22      for (i=0; i<n; i++) {
23        px[i] = 0.0;
24        for (j=0; j<nlocal; j++)
25          px[i] += a[i*nlocal+j]*b[j];
26      }
27
28      /* Sum-up the results by performing an element-wise reduction operation */
29      MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);
30
31      /* Redistribute fx in a fashion similar to that of vector b */
32      MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0,
33          comm);
34
35      free(px); free(fx);
36    }
```

Comparing these two programs for performing matrix-vector multiplication we see that the row-wise version needs to perform only a `MPI_Allgather` operation whereas the column-wise program needs to perform a `MPI_Reduce` and a `MPI_Scatter` operation. In general, a row-wise distribution is preferable as it leads to small communication overhead (see Problem 6.6). However, many times, an application needs to compute not only Ax but also $A^T x$. In that case, the row-wise distribution can be used to compute Ax , but the computation of $A^T x$ requires the column-wise distribution (a row-wise distribution of A is a column-wise distribution of its transpose A^T). It is much cheaper to use the program for the column-wise distribution than to transpose the matrix and then use the row-wise program. We must also note that using a dual of the all-gather operation, it is possible to develop a parallel formulation for column-wise distribution that is as fast as the program using row-wise distribution (see Problem 6.7). However, this dual operation is not available in MPI.

6.6.9 Example: Single-Source Shortest-Path

Our second message-passing program that uses collective communication operations computes the shortest paths from a source-vertex s to all the other vertices in a graph using Dijkstra's single-source shortest-path algorithm described in Section 10.3 . This program is shown in Program 6.6.

The parameter `n` stores the total number of vertices in the graph, and the parameter `source` stores the vertex from which we want to compute the single-source shortest path. The parameter `wgt` points to the locally stored portion of the weighted adjacency matrix of the graph. The parameter `lengths` points to a vector that will store the length of the shortest paths from `source` to the locally stored vertices. Finally, the parameter `comm` is the communicator to be used by the MPI routines. Note that this routine assumes that the number of vertices is a multiple of the number of processors.

Program 6.6 Dijkstra's Single-Source Shortest-Path

[View full width]

```
 1   SingleSource(int n, int source, int *wgt, int *lengths, MPI_Comm comm)
 2   {
 3     int i, j;
 4     int nlocal;  /* The number of vertices stored locally */
 5     int *marker;  /* Used to mark the vertices belonging to V_o */
 6     int firstvtx;  /* The index number of the first vertex that is stored locall
 7     int lastvtx;  /* The index number of the last vertex that is stored locally
 8     int u, udist;
 9     int lminpair[2], gminpair[2];
10     int npes, myrank;
11     MPI_Status status;
12
13     MPI_Comm_size(comm, &npes);
14     MPI_Comm_rank(comm, &myrank);
15
16     nlocal   = n/npes;
17     firstvtx = myrank*nlocal;
18     lastvtx  = firstvtx+nlocal-1;
19
20     /* Set the initial distances from source to all the other vertices */
21     for (j=0; j<nlocal; j++)
22       lengths[j] = wgt[source*nlocal + j];
23
24     /* This array is used to indicate if the shortest part to a vertex has been
   or not. */
25     /* if marker [v] is one, then the shortest path to v has been found, otherwi
   has not. */
26     marker = (int *)malloc(nlocal*sizeof(int));
27     for (j=0; j<nlocal; j++)
28       marker[j] = 1;
29
30     /* The process that stores the source vertex, marks it as being seen */
31     if (source >= firstvtx && source <= lastvtx)
32       marker[source-firstvtx] = 0;
33
34     /* The main loop of Dijkstra's algorithm */
35     for (i=1; i<n; i++) {
36       /* Step 1: Find the local vertex that is at the smallest distance from sou
37       lminpair[0] = MAXINT; /* set it to an architecture dependent large number
38       lminpair[1] = -1;
39       for (j=0; j<nlocal; j++) {
40         if (marker[j] && lengths[j] < lminpair[0]) {
41           lminpair[0] = lengths[j];
42           lminpair[1] = firstvtx+j;
43         }
44       }
45
46       /* Step 2: Compute the global minimum vertex, and insert it into V_c */
47        MPI_Allreduce(lminpair, gminpair, 1, MPI_2INT, MPI_MINLOC,
48           comm);
49       udist = gminpair[0];
50       u = gminpair[1];
51
52       /* The process that stores the minimum vertex, marks it as being seen */
```

```
53        if (u == lminpair[1])
54          marker[u-firstvtx] = 0;
55
56        /* Step 3: Update the distances given that u got inserted */
57        for (j=0; j<nlocal; j++) {
58          if (marker[j] && udist + wgt[u*nlocal+j] < lengths[j])
59            lengths[j] = udist + wgt[u*nlocal+j];
60        }
61      }
62
63      free(marker);
64    }
```

The main computational loop of Dijkstra's parallel single-source shortest path algorithm performs three steps. First, each process finds the locally stored vertex in V_o that has the smallest distance from the source. Second, the vertex that has the smallest distance over all processes is determined, and it is included in V_c . Third, all processes update their distance arrays to reflect the inclusion of the new vertex in V_c .

The first step is performed by scanning the locally stored vertices in V_o and determining the one vertex *v* with the smaller *lengths* [*v*] value. The result of this computation is stored in the array *lminpair* . In particular, *lminpair* [0] stores the distance of the vertex, and *lminpair* [1] stores the vertex itself. The reason for using this storage scheme will become clear when we consider the next step, in which we must compute the vertex that has the smallest overall distance from the source. We can find the overall shortest distance by performing a min-reduction on the distance values stored in *lminpair* [0]. However, in addition to the shortest distance, we also need to know the vertex that is at that shortest distance. For this reason, the appropriate reduction operation is the `MPI_MINLOC` which returns both the minimum as well as an index value associated with that minimum. Because of `MPI_MINLOC` we use the two-element array *lminpair* to store the distance as well as the vertex that achieves this distance. Also, because the result of the reduction operation is needed by all the processes to perform the third step, we use the `MPI_Allreduce` operation to perform the reduction. The result of the reduction operation is returned in the *gminpair* array. The third and final step during each iteration is performed by scanning the local vertices that belong in V_o and updating their shortest distances from the source vertex.

Avoiding Load Imbalances Program 6.6 assigns *n* /*p* consecutive columns of *W* to each processor and in each iteration it uses the `MPI_MINLOC` reduction operation to select the vertex v to be included in V_c . Recall that the `MPI_MINLOC` operation for the pairs (*a* , *i*) and (*a* , *j*) will return the one that has the smaller index (since both of them have the same value). Consequently, among the vertices that are equally close to the source vertex, it favors the smaller numbered vertices. This may lead to load imbalances, because vertices stored in lower-ranked processes will tend to be included in V_c faster than vertices in higher-ranked processes (especially when many vertices in V_o are at the same minimum distance from the source). Consequently, the size of the set V_o will be larger in higher-ranked processes, dominating the overall runtime.

One way of correcting this problem is to distribute the columns of *W* using a cyclic distribution. In this distribution process *i* gets every *p* th vertex starting from vertex *i* . This scheme also assigns *n* /*p* vertices to each process but these vertices have indices that span almost the entire graph. Consequently, the preference given to lower-numbered vertices by `MPI_MINLOC` does not lead to load-imbalance problems.

6.6.10 Example: Sample Sort

The last problem requiring collective communications that we will consider is that of sorting a

sequence A of n elements using the sample sort algorithm described in Section 9.5 . The program is shown in Program 6.7 .

The `SampleSort` function takes as input the sequence of elements stored at each process and returns a pointer to an array that stores the sorted sequence as well as the number of elements in this sequence. The elements of this `SampleSort` function are integers and they are sorted in increasing order. The total number of elements to be sorted is specified by the parameter `n` and a pointer to the array that stores the local portion of these elements is specified by `elmnts` . On return, the parameter `nsorted` will store the number of elements in the returned sorted array. This routine assumes that `n` is a multiple of the number of processes.

Program 6.7 Samplesort

[View full width]

```
 1   int *SampleSort(int n, int *elmnts, int *nsorted, MPI_Comm comm)
 2   {
 3     int i, j, nlocal, npes, myrank;
 4     int *sorted_elmnts, *splitters, *allpicks;
 5     int *scounts, *sdispls, *rcounts, *rdispls;
 6
 7     /* Get communicator-related information */
 8     MPI_Comm_size(comm, &npes);
 9     MPI_Comm_rank(comm, &myrank);
10
11     nlocal = n/npes;
12
13     /* Allocate memory for the arrays that will store the splitters */
14     splitters = (int *)malloc(npes*sizeof(int));
15     allpicks = (int *)malloc(npes*(npes-1)*sizeof(int));
16
17     /* Sort local array */
18     qsort(elmnts, nlocal, sizeof(int), IncOrder);
19
20     /* Select local npes-1 equally spaced elements */
21     for (i=1; i<npes; i++)
22       splitters[i-1] = elmnts[i*nlocal/npes];
23
24     /* Gather the samples in the processors */
25     MPI_Allgather(splitters, npes-1, MPI_INT, allpicks, npes-1,
26         MPI_INT, comm);
27
28     /* sort these samples */
29     qsort(allpicks, npes*(npes-1), sizeof(int), IncOrder);
30
31     /* Select splitters */
32     for (i=1; i<npes; i++)
33       splitters[i-1] = allpicks[i*npes];
34     splitters[npes-1] = MAXINT;
35
36     /* Compute the number of elements that belong to each bucket */
37     scounts = (int *)malloc(npes*sizeof(int));
38     for (i=0; i<npes; i++)
39       scounts[i] = 0;
40
```

```
41      for (j=i=0; i<nlocal; i++) {
42        if (elmnts[i] < splitters[j])
43          scounts[j]++;
44        else
45          scounts[++j]++;
46      }
47
48      /* Determine the starting location of each bucket's elements in the elmnts a
49       sdispls = (int *)malloc(npes*sizeof(int));
50      sdispls[0] = 0;
51      for (i=1; i<npes; i++)
52         sdispls[i] = sdispls[i-1]+scounts[i-1];
53
54      /* Perform an all-to-all to inform the corresponding processes of the number
    elements */
55      /* they are going to receive. This information is stored in rcounts array */
56       rcounts = (int *)malloc(npes*sizeof(int));
57       MPI_Alltoall(scounts, 1, MPI_INT, rcounts, 1, MPI_INT, comm);
58
59      /* Based on rcounts determine where in the local array the data from each
    processor */
60      /* will be stored. This array will store the received elements as well as th
    final */
61      /* sorted sequence */
62       rdispls = (int *)malloc(npes*sizeof(int));
63      rdispls[0] = 0;
64      for (i=1; i<npes; i++)
65         rdispls[i] = rdispls[i-1]+rcounts[i-1];
66
67       *nsorted = rdispls[npes-1]+rcounts[i-1];
68       sorted_elmnts = (int *)malloc((*nsorted)*sizeof(int));
69
70      /* Each process sends and receives the corresponding elements, using the
    MPI_Alltoallv */
71      /* operation. The arrays scounts and sdispls are used to specify the number
    elements */
72      /* to be sent and where these elements are stored, respectively. The arrays
    rcounts */
73      /* and rdispls are used to specify the number of elements to be received, ar
    where these */
74      /* elements will be stored, respectively. */
75       MPI_Alltoallv(elmnts, scounts, sdispls, MPI_INT, sorted_elmnts,
76           rcounts, rdispls, MPI_INT, comm);
77
78      /* Perform the final local sort */
79       qsort(sorted_elmnts, *nsorted, sizeof(int), IncOrder);
80
81       free(splitters); free(allpicks); free(scounts); free(sdispls);
82       free(rcounts); free(rdispls);
83
84       return sorted_elmnts;
85    }
```

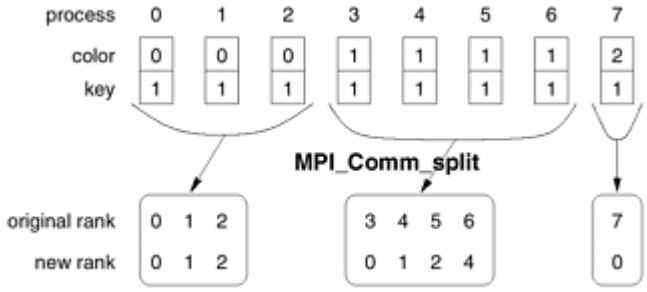
6.7 Groups and Communicators

In many parallel algorithms, communication operations need to be restricted to certain subsets of processes. MPI provides several mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator. A general method for partitioning a graph of processes is to use `MPI_Comm_split` that is defined as follows:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
        MPI_Comm *newcomm)
```

This function is a collective operation, and thus needs to be called by all the processes in the communicator `comm` . The function takes `color` and `key` as input parameters in addition to the communicator, and partitions the group of processes in the communicator `comm` into disjoint subgroups. Each subgroup contains all processes that have supplied the same value for the `color` parameter. Within each subgroup, the processes are ranked in the order defined by the value of the `key` parameter, with ties broken according to their rank in the old communicator (i.e., `comm`). A new communicator for each subgroup is returned in the `newcomm` parameter. Figure 6.7 shows an example of splitting a communicator using the `MPI_Comm_split` function. If each process called `MPI_Comm_split` using the values of parameters `color` and `key` as shown in Figure 6.7 , then three communicators will be created, containing processes {0, 1, 2}, {3, 4, 5, 6}, and {7}, respectively.

Figure 6.7. Using **`MPI_Comm_split`** to split a group of processes in a communicator into subgroups.



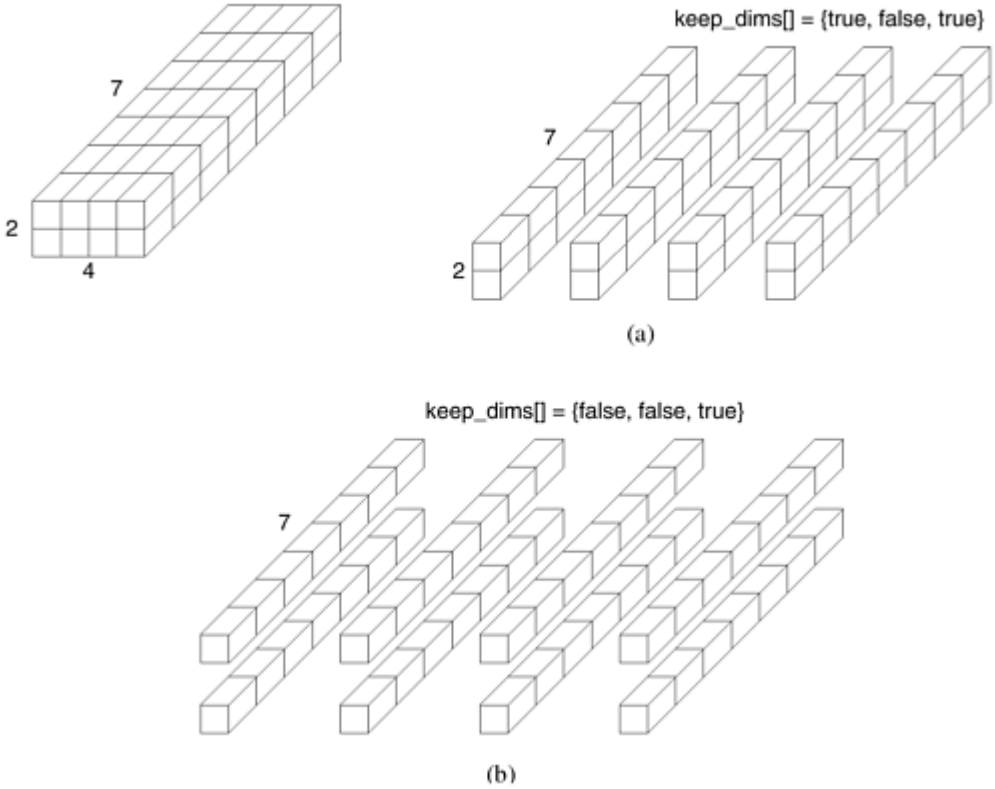
Splitting Cartesian Topologies In many parallel algorithms, processes are arranged in a virtual grid, and in different steps of the algorithm, communication needs to be restricted to a different subset of the grid. MPI provides a convenient way to partition a Cartesian topology to form lower-dimensional grids.

MPI provides the `MPI_Cart_sub` function that allows us to partition a Cartesian topology into sub-topologies that form lower-dimensional grids. For example, we can partition a two-dimensional topology into groups, each consisting of the processes along the row or column of the topology. The calling sequence of `MPI_Cart_sub` is the following:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
        MPI_Comm *comm_subcart)
```

The array `keep_dims` is used to specify how the Cartesian topology is partitioned. In particular, if `keep_dims[i]` is true (non-zero value in C) then the `i` th dimension is retained in the new sub-topology. For example, consider a three-dimensional topology of size 2 x 4 x 7. If `keep_dims` is {true, false, true}, then the original topology is split into four two-dimensional sub-topologies of size 2 x 7, as illustrated in Figure 6.8(a) . If `keep_dims` is {false, false, true}, then the original topology is split into eight one-dimensional topologies of size seven, illustrated in Figure 6.8(b) . Note that the number of sub-topologies created is equal to the product of the number of processes along the dimensions that are not being retained. The original topology is specified by the communicator `comm_cart` , and the returned communicator `comm_subcart` stores information about the created sub-topology. Only a single communicator is returned to each process, and for processes that do not belong to the same sub-topology, the group specified by the returned communicator is different.

Figure 6.8. Splitting a Cartesian topology of size 2 x 4 x 7 into (a) four subgroups of size 2 x 1 x 7, and (b) eight subgroups of size 1 x 1 x 7.



The processes belonging to a given sub-topology can be determined as follows. Consider a three-dimensional topology of size d_1 x d_2 x d_3 , and assume that `keep_dims` is set to {true, false, true}. The group of processes that belong to the same sub-topology as the process with coordinates (x , y , z) is given by (*, y , *), where a '*' in a coordinate denotes all the possible values for this coordinate. Note also that since the second coordinate can take d_2 values, a total of d_2 sub-topologies are created.

Also, the coordinate of a process in a sub-topology created by `MPI_Cart_sub` can be obtained from its coordinate in the original topology by disregarding the coordinates that correspond to the dimensions that were not retained. For example, the coordinate of a process in the column-based sub-topology is equal to its row-coordinate in the two-dimensional topology. For instance, the process with coordinates (2, 3) has a coordinate of (2) in the sub-topology that corresponds to the third column of the grid.

6.7.1 Example: Two-Dimensional Matrix-Vector Multiplication

In Section 6.6.8 , we presented two programs for performing the matrix-vector multiplication $x = Ab$ using a row- and column-wise distribution of the matrix. As discussed in Section 8.1.2 , an alternative way of distributing matrix A is to use a two-dimensional distribution, giving rise to the two-dimensional parallel formulations of the matrix-vector multiplication algorithm.

Program 6.8 shows how these topologies and their partitioning are used to implement the two-dimensional matrix-vector multiplication. The dimension of the matrix is supplied in the parameter n , the parameters a and b point to the locally stored portions of matrix A and vector b , respectively, and the parameter x points to the local portion of the output matrix-vector product. Note that only the processes along the first column of the process grid will store b initially, and that upon return, the same set of processes will store the result x . For simplicity, the program assumes that the number of processes p is a perfect square and that n is a multiple of $\sqrt{p}$.

Program 6.8 Two-Dimensional Matrix-Vector Multiplication

[View full width]

```
 1   MatrixVectorMultiply_2D(int n, double *a, double *b, double *x,
 2                           MPI_Comm comm)
 3   {
 4     int ROW=0, COL=1; /* Improve readability */
 5     int i, j, nlocal;
 6     double *px; /* Will store partial dot products */
 7     int npes, dims[2], periods[2], keep_dims[2];
 8     int myrank, my2drank, mycoords[2];
 9     int other_rank, coords[2];
10     MPI_Status status;
11     MPI_Comm comm_2d, comm_row, comm_col;
12
13     /* Get information about the communicator */
14     MPI_Comm_size(comm, &npes);
15     MPI_Comm_rank(comm, &myrank);
16
17     /* Compute the size of the square grid */
18     dims[ROW] = dims[COL] = sqrt(npes);
19
20     nlocal = n/dims[ROW];
21
22     /* Allocate memory for the array that will hold the partial dot-products */
23     px = malloc(nlocal*sizeof(double));
24
25     /* Set up the Cartesian topology and get the rank & coordinates of the proce
   this topology */
26     periods[ROW] = periods[COL] = 1; /* Set the periods for wrap-around connecti
27
28     MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);
29
30     MPI_Comm_rank(comm_2d, &my2drank); /* Get my rank in the new topology */
31     MPI_Cart_coords(comm_2d, my2drank, 2, mycoords); /* Get my coordinates */
32
33     /* Create the row-based sub-topology */
```

```
34      keep_dims[ROW] = 0;
35      keep_dims[COL] = 1;
36      MPI_Cart_sub(comm_2d, keep_dims, &comm_row);
37
38     /* Create the column-based sub-topology */
39      keep_dims[ROW] = 1;
40      keep_dims[COL] = 0;
41      MPI_Cart_sub(comm_2d, keep_dims, &comm_col);
42
43     /* Redistribute the b vector. */
44     /* Step 1. The processors along the 0th column send their data to the diagor
processors */
45      if (mycoords[COL] == 0 && mycoords[ROW] != 0) { /* I'm in the first column *
46        coords[ROW] = mycoords[ROW];
47        coords[COL] = mycoords[ROW];
48        MPI_Cart_rank(comm_2d, coords, &other_rank);
49        MPI_Send(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d);
50      }
51      if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
52        coords[ROW] = mycoords[ROW];
53        coords[COL] = 0;
54        MPI_Cart_rank(comm_2d, coords, &other_rank);
55        MPI_Recv(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d,
56            &status);
57      }
58
59     /* Step 2. The diagonal processors perform a column-wise broadcast */
60      coords[0] = mycoords[COL];
61      MPI_Cart_rank(comm_col, coords, &other_rank);
62      MPI_Bcast(b, nlocal, MPI_DOUBLE, other_rank, comm_col);
63
64     /* Get into the main computational loop */
65      for (i=0; i<nlocal; i++) {
66        px[i] = 0.0;
67        for (j=0; j<nlocal; j++)
68          px[i] += a[i*nlocal+j]*b[j];
69      }
70
71     /* Perform the sum-reduction along the rows to add up the partial dot-produc
72     coords[0] = 0;
73      MPI_Cart_rank(comm_row, coords, &other_rank);
74      MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, other_rank,
75          comm_row);
76
77      MPI_Comm_free(&comm_2d); /* Free up communicator */
78      MPI_Comm_free(&comm_row); /* Free up communicator */
79      MPI_Comm_free(&comm_col); /* Free up communicator */
80
81      free(px);
82    }
```