**8.1 The MPI Programming Model**

  In the MPI programming model, a computation comprises one or more   processes that communicate by calling library routines to send   and receive messages to other processes. In most MPI implementations,   a fixed set of processes is created at program initialization, and one process is created per processor. However, these processes may execute different programs. Hence, the MPI programming model is sometimes referred to as *multiple program multiple data* (MPMD) to distinguish it from the SPMD model in which every processor   executes the same program.

Because the number of processes in an MPI computation is normally fixed,   our focus in this chapter is on the mechanisms used to communicate   data between processes. Processes can use *point-to-point* communication operations to send a message from one named process to another; these operations can be used to implement local and unstructured communications. A group of processes can call *collective* communication operations to perform commonly used   global operations such as summation and broadcast. MPI's ability to *probe* for messages supports asynchronous communication. Probably MPI's most important feature from a software engineering viewpoint is its support for modular programming. A mechanism called a *communicator* allows the MPI programmer to define modules that encapsulate internal communication structures. In the terminology used in Chapter [4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node39.html#chapmod), these modules can be combined by both sequential and parallel composition.

Most parallel algorithms designed using the techniques of Part I are readily implemented using MPI. Algorithms that create just one task per processor can be implemented directly, with point-to-point or collective communication routines used to meet communication requirements. Algorithms that create tasks in a dynamic fashion or that rely on the concurrent execution of several tasks on a processor must be further refined to permit an MPI implementation. For example, consider the first branch-and-bound search algorithm   developed in Section [2.7](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node21.html#secfloor), which creates a tree of ``search'' tasks dynamically. This algorithm cannot be implemented directly in MPI; however, as discussed in Chapter [2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node14.html#chap2), it can be refined to obtain an algorithm that creates a fixed set of worker processes that exchange messages representing tree nodes to be searched. The resulting SPMD algorithm can be implemented as an MPI program. Algorithms that are not easily modified in this way are better implemented using alternative technologies.

# 8.2 MPI Basics

Although MPI is a complex and multifaceted system, we can solve a wide range of problems using just six of its functions! We introduce MPI   by describing these six functions, which initiate and terminate a computation, identify processes, and send and receive messages:

MPI\_INIT : Initiate an MPI computation.

MPI\_FINALIZE : Terminate a computation.

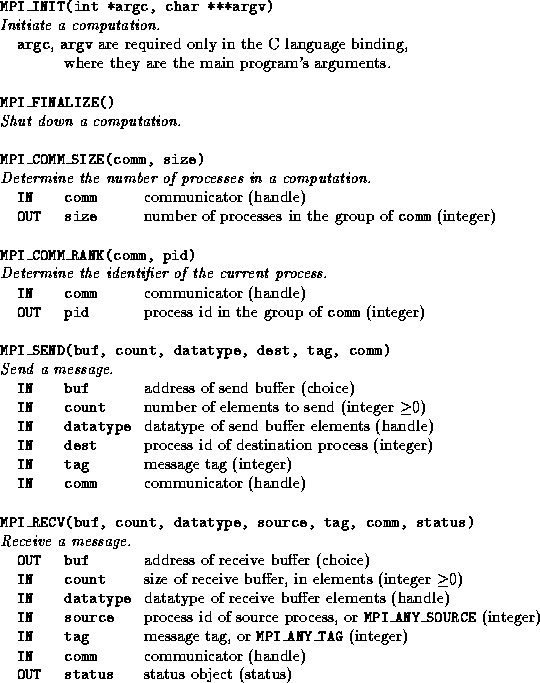
MPI\_COMM\_SIZE : Determine number of processes.

MPI\_COMM\_RANK : Determine my process identifier.

MPI\_SEND : Send a message.

MPI\_RECV : Receive a message.

Function parameters are detailed in Figure [8.1](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#figmpibasics). In this and subsequent figures, the labels IN, OUT, and INOUT indicate whether the function uses but does not modify the parameter ( IN), does not use but may update the parameter ( OUT), or both uses and updates the parameter ( INOUT).

    
**Figure 8.1:** Basic MPI. These six functions suffice to write a wide range of parallel programs. The arguments are characterized as having mode IN or OUT and as having type integer, choice, handle, or status. These terms are explained in the text.

All but the first two calls take a communicator handle as an argument. A communicator identifies the process group and context with respect to which the operation is to be performed. As explained later in this chapter, communicators provide a mechanism for identifying process subsets during development of modular programs and for ensuring that messages intended for different purposes are not confused. For now, it suffices to provide the default value MPI\_COMM\_WORLD, which identifies all processes involved in a computation. Other arguments have type integer, datatype handle, or status. These datatypes are explained in the following.

  The functions MPI\_INIT and MPI\_FINALIZE are used to   initiate and shut down an MPI computation, respectively. MPI\_INIT must be called before any other MPI function and must be called exactly once per process. No further MPI functions can be called after MPI\_FINALIZE.

  The functions MPI\_COMM\_SIZE and MPI\_COMM\_RANK   determine the number of processes in the current computation and the integer identifier assigned to the current process, respectively. (The processes in a process group are identified with unique, contiguous integers numbered from 0.) For example, consider the following program. This is not written in any particular language: we shall see in the next section how to call MPI routines from Fortran and C.

program main

begin

MPI\_INIT() Initiate computation

MPI\_COMM\_SIZE(MPI\_COMM\_WORLD, count) Find # of processes

MPI\_COMM\_RANK(MPI\_COMM\_WORLD, myid) Find my id

print("I am", myid, "of", count) Print message

MPI\_FINALIZE() Shut down

end

The MPI standard does not specify how a parallel computation is   started. However, a typical mechanism could be a command line argument indicating the number of processes that are to be created: for example, myprog -n 4, where myprog is the name of the executable. Additional arguments might be used to specify processor names in a networked environment or executable names in an MPMD computation.

If the above program is executed by four processes, we will obtain something like the following output. The order in which the output appears is not defined; however, we assume here that the output from individual print statements is not interleaved.

I am 1 of 4

I am 3 of 4

I am 0 of 4

I am 2 of 4

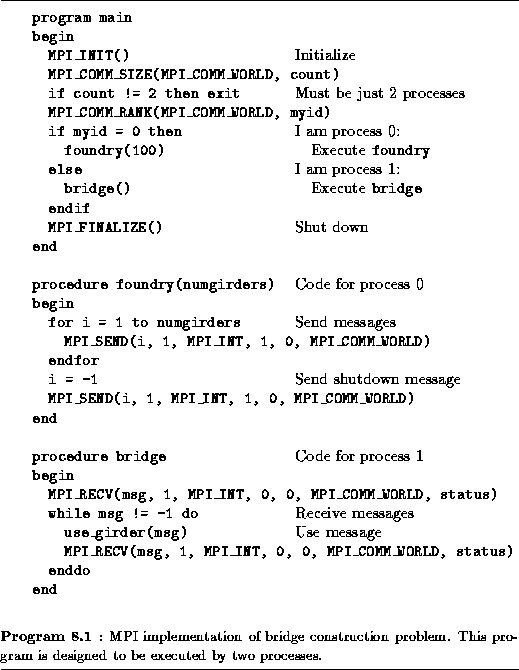
Finally, we consider the functions MPI\_SEND and MPI\_RECV, which are used to send and receive messages,   respectively. A call to MPI\_SEND has the general form   MPI\_SEND(buf, count, datatype, dest, tag, comm)

and specifies that a message containing count elements of the specified datatype starting at address buf is to be sent to the process with identifier dest. As will be explained in greater detail subsequently, this message is associated with an envelope comprising the specified tag, the source process's identifier, and the specified communicator ( comm).

A call to MPI\_RECV has the general form

MPI\_RECV(buf, count, datatype, source, tag, comm, status)

and attempts to receive a message that has an envelope corresponding to the specified tag, source, and comm, blocking until such a message is available. When the message arrives, elements of the specified datatype are placed into the buffer at address buf. This buffer is guaranteed to be large enough to contain at least count elements. The status variable can be used subsequently to inquire about the size, tag, and source of the received message (Section [8.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node98.html#secmpinquire)).

Program [8.1](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#progmpi1) illustrates the use of the six basic calls. This is an implementation of the bridge construction algorithm developed in Example [1.1](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node9.html#exbridge). The program is designed to be   executed by two processes. The first process calls a procedure   foundry and the second calls bridge, effectively creating two different tasks. The first process makes a series of MPI\_SEND calls to communicate 100 integer messages to the second process, terminating the sequence by sending a negative number. The second process receives these messages using MPI\_RECV.

## 8.2.1 Language Bindings

Much of the discussion in this chapter will be language independent; that is, the functions described can be used in C, Fortran, or any other language for which an MPI library has been defined. Only when we present example programs will a particular language be used. In that case, programs will be presented using the syntax of either the Fortran or C language binding. Different language bindings have slightly different syntaxes that reflect a language's peculiarities. Sources of syntactic difference include the function names themselves, the mechanism used for return codes, the representation of the   handles used to access specialized MPI data structures such as   communicators, and the implementation of the status datatype   returned by MPI\_RECV. The use of handles hides the internal representation of MPI data structures.

#### C Language Binding.

In the C language binding, function names are as in the MPI   definition but with only the MPI prefix and the first letter of the function name in upper case. Status values are returned as integer return codes. The return code for successful completion is MPI\_SUCCESS; a set of error codes is also defined. Compile-time constants are all in upper case and are defined in the file mpi.h, which must be included in any program that makes MPI calls. Handles are represented by special defined types, defined in mpi.h. These will be introduced as needed in the following discussion. Function parameters with type IN are passed by value, while parameters with type OUT and INOUT are passed by reference (that is, as pointers). A status variable has type MPI\_Status and is a structure with fields status.MPI\_SOURCE and status.MPI\_TAG containing source and tag information. Finally, an MPI datatype is defined for each C datatype: MPI\_CHAR, MPI\_INT, MPI\_LONG, MPI\_UNSIGNED\_CHAR, MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_FLOAT, MPI\_DOUBLE, MPI\_LONG\_DOUBLE, etc.

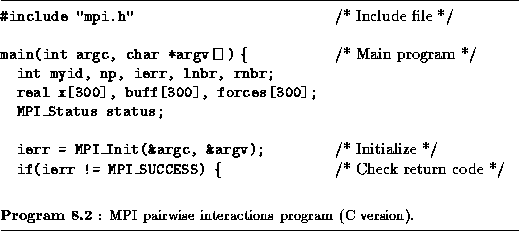
#### Fortran Language Binding.

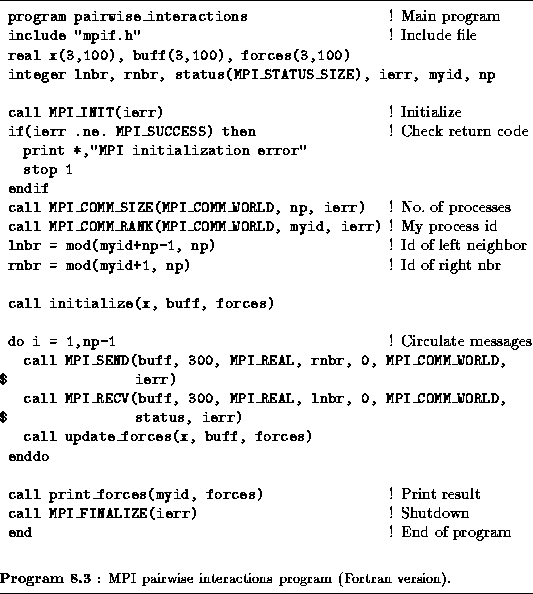
In the Fortran language binding, function names are in upper case. Function return codes are represented by an additional integer   argument. The return code for successful completion is MPI\_SUCCESS; a set of error codes is also defined. Compile-time constants are all in upper case and are defined in the file mpif.h, which must be included in any program that makes MPI calls. All handles have type INTEGER. A status variable is an array of integers of size MPI\_STATUS\_SIZE, with the constants MPI\_SOURCE and MPI\_TAG indexing the source and tag fields, respectively. Finally, an MPI datatype is defined for each Fortran datatype: MPI\_INTEGER, MPI\_REAL, MPI\_DOUBLE\_PRECISION, MPI\_COMPLEX, MPI\_LOGICAL, MPI\_CHARACTER, etc.

**Example http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1008.gif.http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1007.gif Pairwise Interactions**:

The pairwise interactions algorithm of Section [1.4.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node10.html#exinteractions) illustrate the two language bindings.   Recall that in this algorithm, T tasks ( T an odd number)   are connected in a ring. Each task is responsible for computing interactions involving N data. Data are circulated around the ring in T-1 phases, with interactions computed at each phase. Programs [8.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#progmp1) and [8.3](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#progmp2) are C and Fortran versions of an MPI implementation, respectively.

The number of processes created is specified when the program is invoked. Each process is responsible for 100 objects, and each object is represented by three floating-point values, so the various work arrays have size 300. As each process executes the same program, the first few lines are used to determine the total number of processes involved in the computation ( np), the process's identifier ( myid), and the identify of the process's neighbors in the ring ( lnbr, rnbr). The computation then proceeds as described in Section [1.4.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node10.html#exinteractions) but with messages sent to numbered processes rather than on channels.

## 8.2.2 Determinism

Before proceeding to more sophisticated aspects of MPI, we consider the important topic of determinism. Message-passing programming   models are by default nondeterministic: the arrival order of messages   sent from two processes, A and B, to a third process, C, is not defined. (However, MPI does guarantee that two messages sent from one process, A, to another process, B, will arrive in the order sent.) It is the programmer's responsibility to ensure that a computation is deterministic when (as is usually the case) this is required.

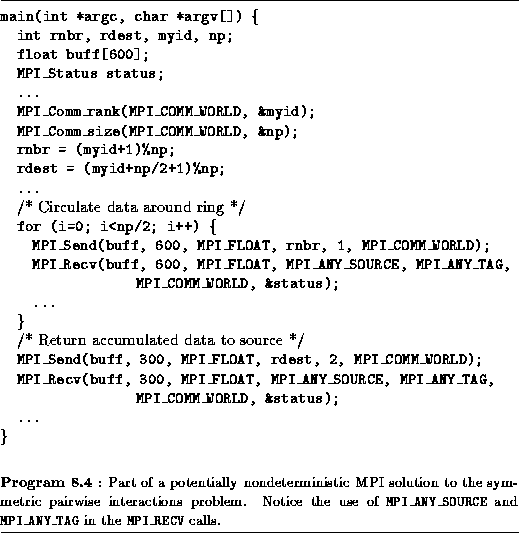
  In the task/channel programming model, determinism is guaranteed by defining separate channels for different communications and by ensuring that each channel has a single writer and a single reader. Hence, a process C can distinguish messages received from A or B as they arrive on separate channels. MPI does not support channels directly, but it does provide similar mechanisms. In particular, it allows a receive operation to specify a source, tag, and/or context. (Recall that these data constitute a message's envelope.) We consider the first two of these mechanisms in this section.

The source specifier in the MPI\_RECV function allows the programmer to specify that a message is to be received either from a single named process (specified by its integer process identifier) or from any process (specified by the special value MPI\_ANY\_SOURCE). The latter option allows a process to receive data from any source; this is sometimes useful. However, the former is preferable because it eliminates errors due to messages arriving in time-dependent order.

Message tags provide a further mechanism for distinguishing between different messages. A sending process must associate an integer tag with a message. This is achieved via the tag field in the   MPI\_SEND call. (This tag has always been set to 0 in the examples presented so far.) A receiving process can then specify that it wishes to receive messages either with a specified tag or with any tag ( MPI\_ANY\_TAG). Again, the former option is preferable because it reduces the possibility of error.

**Example http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1013.gif.http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1011.gif Nondeterministic Program**:

To illustrate the importance of source specifiers and tags, we examine a program that fails to use them and that, consequently, suffers from   nondeterminism. Program [8.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#progmpnondet) is part of an MPI   implementation of the symmetric pairwise interaction algorithm of Section [1.4.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node10.html#exinteractions). Recall that in this algorithm, messages are communicated only half way around the ring (in T/2-1 steps, if the number of tasks T is odd), with interactions accumulated both in processes and in messages. As in Example [8.1](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#expio), we assume 100 objects, so the arrays to be communicated in this phase have size 100.3.2=600. In a final step, each message (with size 100.3=300) is returned to its originating process. Hence, each process sends and receives N/2-1 data messages and one result message.

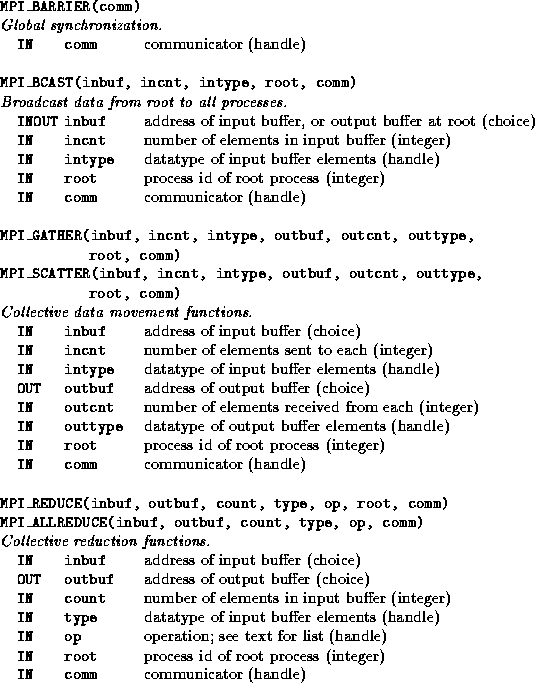
Program [8.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#progmpnondet) specifies neither sources nor tags in its MPI\_RECV calls. Consequently, a result message arriving before the final data message may be received as if it were a data message, thereby resulting in an incorrect computation. Determinism can be achieved by specifying either a source processor or a tag in the receive calls. It is good practice to use both mechanisms. In effect, each ``channel'' in the original design is then represented by a unique (source, destination, tag) triple.

# 8.3 Global Operations

As explained in Chapter [2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node14.html#chap2), parallel algorithms often call for coordinated communication operations involving multiple processes. For example, all processes may need to cooperate to transpose a distributed matrix or to sum a set of numbers distributed one per process. Clearly, these global operations can be implemented by a programmer using the send and receive functions introduced in Section [8.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#secmpibasics). For convenience, and to permit optimized implementations, MPI also   provides a suite of specialized   collective communication functions that perform commonly used   operations of this type. These functions include the following.

* Barrier: Synchronizes all processes.
* Broadcast: Sends data from one process to all processes.
* Gather: Gathers data from all processes to one process.
* Scatter: Scatters data from one process to all processes.
* Reduction operations: Sums, multiplies, etc., distributed data.

These operations are summarized in Figure [8.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node97.html#figmpiglobal). All are executed collectively, meaning that each process in a process group calls the communication routine with the same parameters.

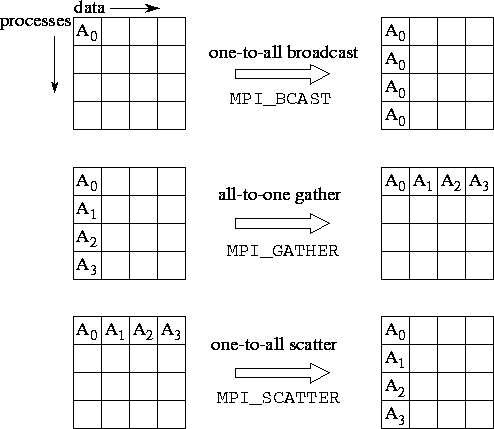
    
**Figure 8.2:** MPI global communication functions.

## 8.3.1 Barrier

MPI\_BARRIER is used to synchronize execution of a group of   processes. No process returns from this function until all processes   have called it. A barrier is a simple way of separating two phases of   a computation to ensure that messages generated in the two phases do not intermingle. For example, a call to MPI\_BARRIER could be inserted before the second send operation in Program [8.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#progmpnondet) to ensure deterministic execution. Of course, in this example as in many others, the need for an explicit barrier can be avoided by the appropriate use of tags, source specifiers, and/or contexts.

## 8.3.2 Data Movement

MPI\_BCAST, MPI\_GATHER, and MPI\_SCATTER are collective data movement routines, in which all processes interact with a distinguished root process to broadcast, gather, or scatter data, respectively. The operation of these functions is illustrated in Figure [8.3](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node97.html#figmpdata). In each case, the first three arguments specify the location ( inbuf) and type ( intype) of the data to be communicated and the number of elements to be sent to each destination ( incnt). Other arguments specify the location and type of the result ( outbuf, outtype) and the number of elements to be received from each source ( outcnt).

    
**Figure 8.3:** MPI collective data movement functions, illustrated for a group of 4 processes. In each set of 16 boxes, each row represents data locations in a different process. Thus, in the one-to-all broadcast, the data *Ahttp://www.mcs.anl.gov/%7Eitf/dbpp/text/img1016.gif* is initially located just in process 0; after the call, it is replicated in all processes. In each case, both *incnt* and *outcnt* are 1, meaning that each message comprises a single data element.

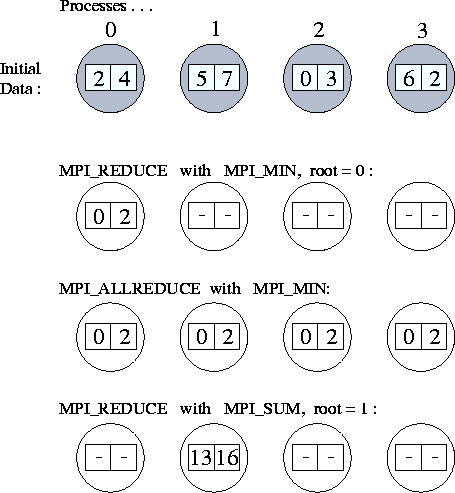
MPI\_BCAST implements a one-to-all broadcast   operation whereby a single named process ( root) sends the same   data to all other processes; each process receives this data from the root process. At the time of call, the data are located in inbuf in process root and consists of incnt data items of a specified intype. After the call, the data are replicated in inbuf in all processes. As inbuf is used for input at the root and for output in other processes, it has type INOUT.

MPI\_GATHER implements an all-to-one gather   operation. All processes (including the root process) send data   located in inbuf to root. This process places the data in contiguous nonoverlapping locations in outbuf, with the data from process i preceding that from process i+1 . Hence, the outbuf in the root process must be P times larger than inbuf, where P is the number of processes participating in the operation. The outbuf in processes other than the root is ignored.

MPI\_SCATTER implements a one-to-all scatter   operation; it is the reverse of MPI\_GATHER. A specified   root process sends data to all processes, sending the i th portion of its inbuf to process i ; each process receives data from root in outbuf. Hence, the inbuf in the root process must be P times larger than outbuf. Notice the subtle difference between this function and MPI\_BCAST: while in MPI\_BCAST every process receives the same value from the root process, in MPI\_SCATTER every process receives a different value.

## 8.3.3 Reduction Operations

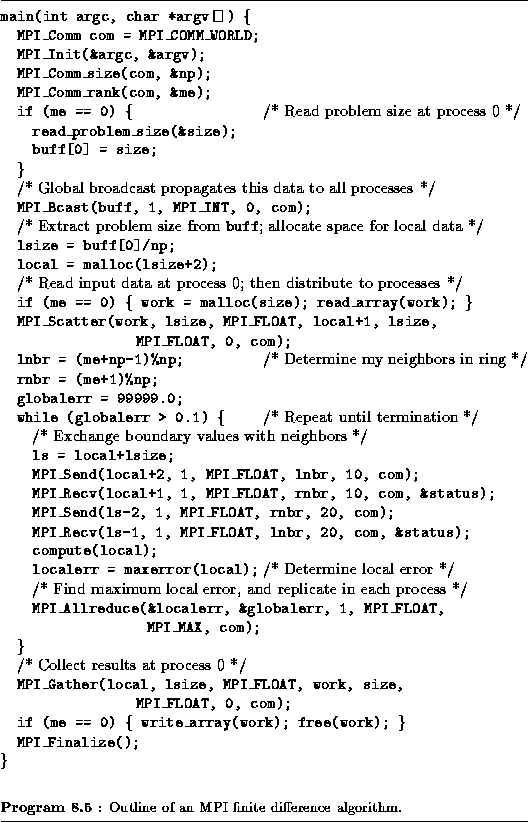
  The functions MPI\_REDUCE and MPI\_ALLREDUCE implement reduction operations. They combine the values provided in the input buffer of each process, using a specified operation op, and   return the combined value either to the output buffer of the single   root process (in the case of MPI\_REDUCE) or to the output   buffer of all processes ( MPI\_ALLREDUCE). The operation is applied pointwise to each of the count values provided by each process. All operations return count values with the same datatype as the operands. Valid operations include maximum and minimum ( MPI\_MAX and MPI\_MIN); sum and product ( MPI\_SUM and MPI\_PROD); logical and, or, and exclusive or ( MPI\_LAND, MPI\_LOR, and MPI\_LXOR); and bitwise and, or, and exclusive or ( MPI\_BAND, MPI\_BOR, and MPI\_BXOR).

    
**Figure:** Applications of *MPI\_REDUCE* and *MPI\_ALLREDUCE*. The first line shows the send buffers (of size 2) in each of four processes; subsequent lines show the output buffers in each process following four different global communication calls.

As an example, the following call would be used to compute the minimum of two sets of P values, where P is the number of processes involved in the reduction.

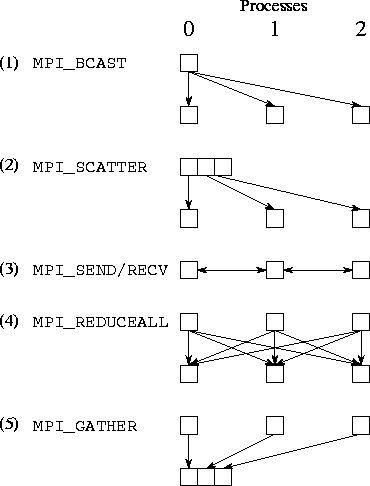
MPI\_REDUCE(inbuf, outbuf, 2, MPI\_INT, MPI\_MIN, 0, MPI\_COMM\_WORLD)

After the reduction, outbuf[0] in process 0 contains the minimum of the first element in each input buffer ( min(inbuf[0])); similarly, outbuf[1] contains min(inbuf[1]). The operation of this and other calls to MPI reduction functions are illustrated in Figure [8.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node97.html#figmpreduce).

**Example http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1022.gif.http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1020.gif Finite Difference**:

Once again we consider a finite difference problem, this time to   illustrate the use of global operations. The algorithm   considered requires both nearest-neighbor communication (to exchange boundary values) and global communication (to detect termination). Similar problems have previously been discussed in Chapter [2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node14.html#chap2). The MPI implementation given in Program [8.5](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node97.html#progmp4) is for a one-dimensional decomposition of a one-dimensional problem in which each process has two neighbors. It uses MPI\_SEND and MPI\_RECV for nearest-neighbor communication and four MPI global communication routines, for a total of five distinct communication operations. These are summarized as follows and are illustrated in Figure [8.5](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node97.html#figmpfdcomm):

    
**Figure 8.5:** Communication performed in the finite difference program, assuming three processes. Each column represents a processor; each subfigure shows data movement in a single phase. The five phases illustrated are (1) broadcast, (2) scatter, (3) nearest-neighbor exchange, (4) reduction, and (5) gather.

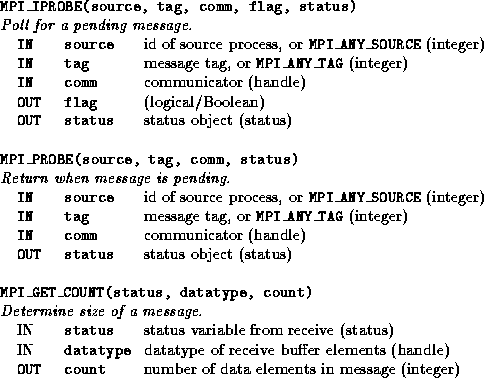
1. MPI\_BCAST to broadcast the problem size parameter ( size) from process 0 to all np processes;
2. MPI\_SCATTER to distribute an input array ( work) from process 0 to other processes, so that each process receives size/np elements;
3. MPI\_SEND and MPI\_RECV for exchange of data (a single floating-point number) with neighbors;
4. MPI\_ALLREDUCE to determine the maximum of a set of localerr values computed at the different processes and to distribute this maximum value to each process; and
5. MPI\_GATHER to accumulate an output array at process 0.

The use of scatter and gather operations to transfer input and output data is particularly simple and convenient. Note, however, that their use in this example is inherently nonscalable. As we solve larger problems, storage limitations will eventually prevent us from accumulating all input and output data in a single process. In addition, the associated communication costs may be prohibitive.

**8.4 Asynchronous Communication**

  Recall from Chapter [2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node14.html#chap2) that the need for asynchronous   communication can arise when a computation must access elements of a   shared data structure in an unstructured manner. One implementation approach is to encapsulate the data structure in a set of specialized data tasks to which read and write requests can be directed. This approach is not typically efficient in MPI, however, because of its MPMD programming model.

As noted in Section [2.3.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node17.html#seccommas), an alternative implementation approach is to distribute the shared data structure among the   computational processes, which must then poll periodically for pending read and write requests. This technique is supported by the MPI\_IPROBE function, which is described in this section along with the related functions MPI\_PROBE and MPI\_GET\_COUNT. The three functions are summarized in   Figure [8.6](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node98.html#figmpiinquiry).

    
**Figure 8.6:** MPI inquiry and probe operations.

The MPI\_IPROBE function checks for the existence of pending messages without receiving them, thereby allowing us to write programs that interleave local computation with the processing of incoming messages. A call to MPI\_IPROBE has the general form MPI\_IPROBE(source, tag, comm, flag, status)

and sets a Boolean argument flag to indicate whether a message that matches the specified source, tag, and communicator is available. If an appropriate message is available, flag is set to true; otherwise, it is set to false. The message can then be received by using MPI\_RECV. The receive call must specify the same source, tag, and communicator; otherwise, a different message may be received.

Related to MPI\_IPROBE is the function MPI\_PROBE, which   blocks until a message of the specified source, tag, and communicator is available and then returns and sets its status argument. The MPI\_PROBE function is used to receive messages for which we have incomplete information.

The status argument constructed by an MPI\_RECV call, an MPI\_PROBE call, or a successful MPI\_IPROBE call can be used to determine the (pending) message's source, tag, and size. The inquiry function MPI\_GET\_COUNT yields the length of a message just received. Its first two (input) parameters are a status object set by a previous probe or MPI\_RECV call and the datatype of the elements to be received, while the third (output) parameter is an integer used to return the number of elements received (Figure [8.6](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node98.html#figmpiinquiry)). Other information about the received message can be obtained directly from the status object. In the C language binding, this object is a structure with fields MPI\_SOURCE and MPI\_TAG. Thus, status.MPI\_SOURCE and status.MPI\_TAG contain the source and tag of the message just received. In Fortran, the status object is an array of size MPI\_STATUS\_SIZE, and the constants MPI\_SOURCE and MPI\_TAG are the indices of the array elements containing the source and tag information. Thus, status(MPI\_SOURCE) and status(MPI\_TAG) contain the source and tag of the message just received.

The following code fragment use these functions to receive a message from an unknown source and containing an unknown number of integers. It first detects arrival of the message using MPI\_PROBE. Then, it determines the message source and uses MPI\_GET\_COUNT to determine the message size. Finally, it allocates a buffer of the appropriate size and receives the message.

int count, \*buf, source;

MPI\_Probe(MPI\_ANY\_SOURCE, 0, comm, &status);

source = status.MPI\_SOURCE;

MPI\_Get\_count(status, MPI\_INT, &count);

buf = malloc(count\*sizeof(int));

MPI\_Recv(buf, count, MPI\_INT, source, 0, comm, &status);

**Example http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1025.gif.http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1024.gif Fock Matrix Construction**:

The Fock matrix construction algorithm of Section [2.8](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node22.html#secchem)   allocates to each processor a data task, which manages part of the   D and F matrices, and a computation task, which generates requests for matrix elements. The two tasks execute concurrently, with the data task responding to requests for data and the computation   task performing computation. Briefly, the two tasks are defined as follows.

/\* Data task \*/

/\* Computation task \*/

while(done != TRUE) { while(done != TRUE) {

receive(request); identify\_next\_task();

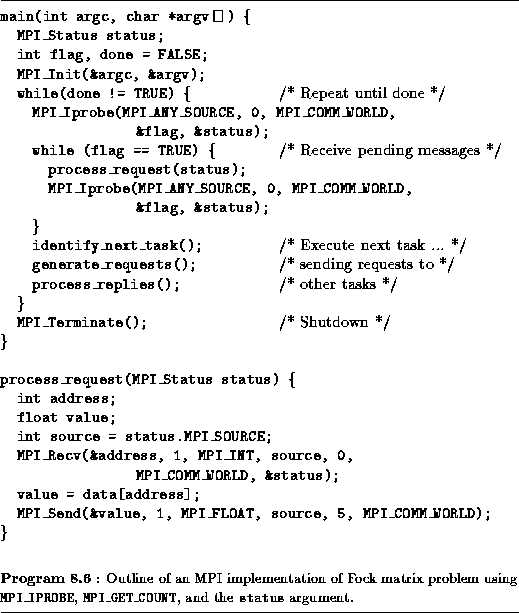
reply\_to(request); generate\_requests();

} process\_replies();

}

A polling version of this program integrates the functions of the database and computation tasks into a single process, which alternates between checking for pending data requests and performing computation. This integration can be achieved as in Program [8.6](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node98.html#progmpfock). The program uses the MPI\_IPROBE function to determine whether database messages are pending. If they are, these messages are processed before further computation is performed.

For simplicity, the procedure process\_request deals with a single type of request: a read operation on a single array element. A process receiving such a request determines the source of the message, retrieves the requested value, and returns the value to the source   process.

# 8.5 Modularity

In Chapter [4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node39.html#chapmod), we distinguished three general forms of composition that can be used for the modular construction of   parallel programs: sequential, parallel, and concurrent. Recall that   in sequential composition, two program components execute in sequence on the same set of processors. In parallel composition, two program components execute concurrently on disjoint sets of processors. In concurrent composition, two program components execute on potentially nondisjoint sets of processors.

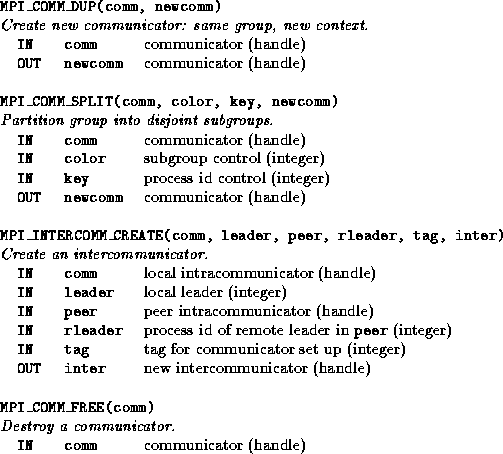
  MPI supports modular programming via its communicator mechanism, which provides the information hiding needed when building modular programs, by allowing the specification of program components that encapsulate internal communication operations and provide a local name space for processes. In this section, we show how communicators can be used to implement various forms of sequential and parallel composition. MPI's MPMD programming model means that the full generality of concurrent composition is not generally available.

An MPI communication operation always specifies a communicator. This identifies the process group that is engaged in the communication operation and the context in which the communication occurs. As we shall see, process groups allow a subset of processes to communicate among themselves using local process identifiers and to perform collective communication operations without involving other processes. The context forms part of the envelope associated with a message. A receive operation can receive a message only if the message was sent in the same context. Hence, if two routines use different contexts for their internal communication, there can be no danger of their communications being confused.

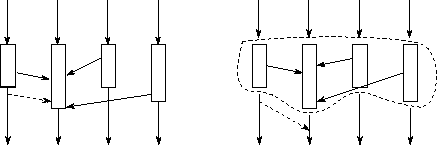
In preceding sections, all communication operations have used the default communicator MPI\_COMM\_WORLD, which incorporates all processes involved in an MPI computation and defines a default   context. We now describe four functions that allow communicators to be used in more flexible ways. These functions, and their roles in modular design, are as follows.

1. MPI\_COMM\_DUP. A program may create a new communicator   comprising the same process group but a new context to ensure that communications performed for different purposes are not confused.   This mechanism supports sequential composition.
2. MPI\_COMM\_SPLIT. A program may create a new communicator   comprising just a subset of a given group of processes. These processes can then communicate among themselves without fear of conflict with other concurrent computations. This mechanism supports   parallel composition.
3. MPI\_INTERCOMM\_CREATE. A program may construct an   intercommunicator , which links processes in two groups. This mechanism supports parallel composition.
4. MPI\_COMM\_FREE. This function can be used to release a communicator created using the preceding three functions.

The four functions are summarized in Figure [8.7](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node99.html#figmpicommun); their arguments and the ways they are called are described next.

    
**Figure 8.7:** MPI communicator functions.

## 8.5.1 Creating Communicators

    
**Figure 8.8:** Errors can occur in a sequential composition of two parallel program components (e.g., an application program and a parallel library) if the two components use the same message tags. The figure on the left shows how this can occur. Each of the four vertical lines represents a single thread of control (process) in an SPMD program. All call an SPMD library, which are represented by the boxes. One process finishes sooner than the others, and a message that this process generates during subsequent computation (the dashed arrow) is intercepted by the library. The figure on the right shows how this problem is avoided by using contexts: the library communicates using a distinct tag space, which cannot be penetrated by other messages.

As discussed in Section [8.2.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#sectags), message tags provide a mechanism for distinguishing between messages used for different purposes. However, they do not provide a sufficient basis for modular design. For example, consider an application that calls a library routine implementing (for example) an array transpose operation. It is important to ensure that the message tags used in the library are distinct from those used in the rest of the application (Figure [8.8](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node99.html#figmpconflict)). Yet the user of a library routine may not know the tags the library uses; indeed, tag values may be computed on the fly.

  Communicators provide a solution to this problem. A call of the form   MPI\_COMM\_DUP(comm, newcomm)

  creates a new communicator newcomm comprising the same processes as comm but with a new context. This new communicator can be passed as an argument to the library routine, as in the following code, which calls transpose to transpose an array A.

integer comm, newcomm, ierr ! Handles are integers

...

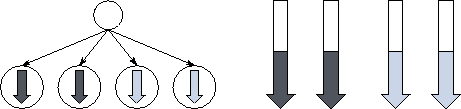
call MPI\_COMM\_DUP(comm, newcomm, ierr) ! Create new context

call transpose(newcomm, A) ! Pass to library

call MPI\_COMM\_FREE(newcomm, ierr) ! Free new context

The transpose routine itself will be defined to use the communicator newcomm in all communication operations, thereby ensuring that communications performed within this routine cannot be confused with communications performed outside.

## 8.5.2 Partitioning Processes

    
**Figure 8.9:** Different views of parallel composition. On the left is the task-parallel view, in which new tasks are created dynamically to execute two different program components. Four tasks are created: two perform one computation (dark shading) and two another (light shading). On the right is the MPMD view. Here, a fixed set of processes (represented by vertical arrows) change character, for example, by calling different subroutines.

Recall that we use the term parallel composition to denote   the parallel execution of two or more program components on disjoint   sets of processors (Section [4.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node41.html#secmodpar)). One approach to the implementation of parallel composition is to create tasks dynamically and to place newly created tasks on different processors. This task-parallel approach is taken in CC++ and Fortran M, for example. In MPMD programs, parallel composition is implemented differently. As illustrated in Figure [8.9](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node99.html#figmpview), available processes are partitioned into disjoint sets, with each set executing the appropriate program. This partitioning is achieved by using the function MPI\_COMM\_SPLIT. A call of the form

MPI\_COMM\_SPLIT(comm, color, key, newcomm)

creates one or more new communicators. This function is a collective communication operation, meaning that it must be executed by each process in the process group associated with comm. A new communicator is created for each unique value of color other than the defined constant MPI\_UNDEFINED. Each new communicator comprises those processes that specified its value of color in the MPI\_COMM\_SPLIT call. These processes are assigned identifiers within the new communicator starting from zero, with order determined by the value of key or, in the event of ties, by the identifier in the old communicator. Thus, a call of the form MPI\_COMM\_SPLIT(comm, 0, 0, newcomm)

in which all processes specify the same color and key, is equivalent to a call MPI\_COMM\_DUP(comm, newcomm)

That is, both calls create a new communicator containing all the processes in the old communicator comm. In contrast, the following code creates three new communicators if comm contains at least three processes.

MPI\_Comm comm, newcomm;

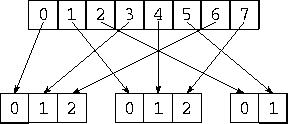
int myid, color;

MPI\_Comm\_rank(comm, &myid);

color = myid%3;

MPI\_Comm\_split(comm, color, myid, &newcomm);

For example, if comm contains eight processes, then processes 0, 3, and 6 form a new communicator of size three, as do processes 1, 4, and 7, while processes 2 and 5 form a new communicator of size two (Figure [8.10](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node99.html#figmpcomm)).

    
**Figure:** Using *MPI\_COMM\_SPLIT* to form new communicators. The first communicator is a group of eight processes. Setting color to *myid%3* and calling *MPI\_COMM\_SPLIT(comm, color, myid, newcomm)* split this into three disjoint process groups.

As a final example, the following code fragment creates a new communicator ( newcomm) containing at most eight processes. Processes with identifiers greater than eight in communicator comm call MPI\_COMM\_SPLIT with newid=MPI\_UNDEFINED and hence are not part of the new communicator.

MPI\_Comm comm, newcomm;

int myid, color;

MPI\_Comm\_rank(comm, &myid);

if (myid < 8) /\* Select first 8 processes \*/

color = 1;

else /\* Others are not in group \*/

color = MPI\_UNDEFINED;

MPI\_Comm\_split(comm, color, myid, &newcomm);

## 8.5.3 Communicating between Groups

A communicator returned by MPI\_COMM\_SPLIT can be used to communicate within a group of processes. Hence, it is called an intracommunicator. (The default communicator, MPI\_COMM\_WORLD, is an intracommunicator.) It is also possible to   create an intercommunicator that can be used to communicate   between process groups. An intercommunicator that connects two groups A and B containing http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1031.gifand http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1032.gifprocesses, respectively, allows processes in group A to communicate with processes 0..http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1033.gif in group B by using MPI send and receive calls (collective operations are not supported). Similarly, processes in group B can communicate with processes 0..http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1034.gif in group A .

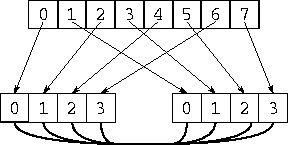
An intercommunicator is created by a collective call executed in the two groups that are to be connected. In making this call, the processes in the two groups must each supply a local intracommunicator that identifies the processes involved in their group. They must also agree on the identifier of a ``leader'' process in each group and a parent communicator that contains all the processes in both groups, via which the connection can be established. The default communicator MPI\_COMM\_WORLD can always be used for this purpose. The collective call has the general form

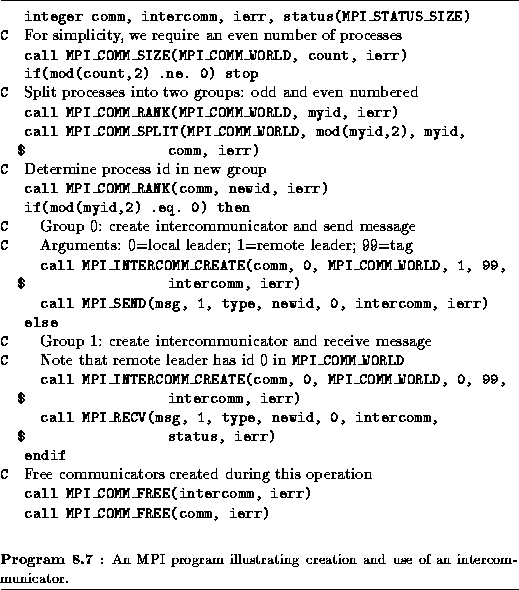
MPI\_INTERCOMM\_CREATE(comm, local\_leader, peercomm,

remote\_leader, tag, intercomm)

where comm is an intracommunicator in the local group and local\_leader is the identifier of the nominated leader process within this group. (It does not matter which process is chosen as the leader; however, all participants in the collective operation must nominate the same process.) The parent communicator is specified by peercomm, while remote\_leader is the identifier of the other group's leader process within the parent communicator. The two other arguments are (1) a ``safe'' tag that the two groups' leader processes can use to communicate within the parent communicator's context without confusion with other communications and (2) the new intercommunicator intercomm.

Program [8.7](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node99.html#progmpic) illustrates these ideas. It first uses MPI\_COMM\_SPLIT to split available processes into two disjoint groups. Even-numbered processes are in one group; odd-numbered processes are in a second. Calls to MPI\_COMM\_RANK are used to determine the values of the variables myid and   newid, which represent each process's identifier in the original communicator and the appropriate new communicator, respectively. In this example, newid=myid/2. Then, the MPI\_INTERCOMM\_CREATE call defines an intercommunicator that links the two groups (Figure [8.11](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node99.html#figmpcomm3)). Process 0 within each group are selected as the two leaders; these processes correspond to processes 0 and 1 within the original group, respectively. Once the intercommunicator is created, each process in the first group sends a message to the corresponding process in the second group. Finally, the new communicators created by the program are deleted.

    
**Figure:** Establishing an intercommunicator between two process groups. At the top is an original group of eight processes; this is *MPI\_COMM\_WORLD*. An *MPI\_COMM\_SPLIT* call creates two process groups, each containing four processes. Then, an *MPI\_INTERCOMM\_CREATE* call creates an intercommunicator between the two groups.

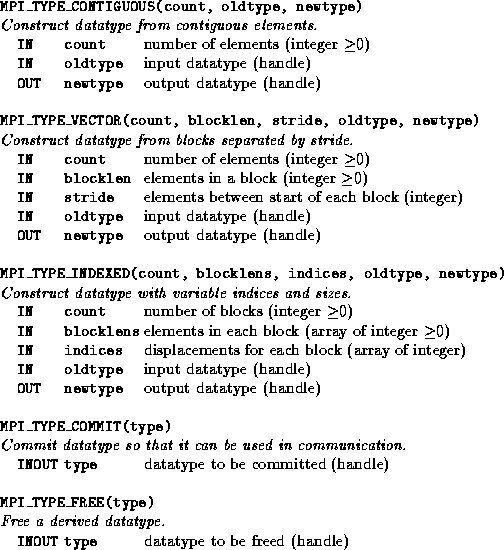
 

# 8.6 Other MPI Features

In this section, we discuss MPI's derived datatype mechanism. We also list MPI features not covered in this book.

## 8.6.1 Derived Datatypes

  In earlier sections of this chapter, MPI routines have been used to communicate simple datatypes, such as integers and reals, or arrays of these types. The final set of MPI functions that we describe implements derived types, a mechanism allowing noncontiguous data elements to be grouped together in a message. This mechanism permits us to avoid data copy operations. Without it, the sending of a row of a two-dimensional array stored by columns would require that these noncontiguous elements be copied into a buffer before being sent.

    
**Figure 8.12:** MPI derived datatype functions.

Three sets of functions are applied for manipulating derived types. Derived datatypes are constructed by applying *constructor* functions to simple or derived types; we describe three constructor functions MPI\_TYPE\_CONTIGUOUS, MPI\_TYPE\_VECTOR, and MPI\_TYPE\_INDEXED. The commit function, MPI\_TYPE\_COMMIT, must be applied to a derived type before it can be used in a communication operation. Finally, the free function, MPI\_TYPE\_FREE, should be applied to a derived type after use, in order to reclaim storage. These functions are summarized in Figure [8.12](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node100.html#figmpiderived).

The constructor MPI\_TYPE\_CONTIGUOUS is used to define a type   comprising one or more contiguous data elements. A call of the form MPI\_TYPE\_CONTIGUOUS(count, oldtype, newtype)

defines a derived type newtype comprising count consecutive occurrences of datatype oldtype. For example, the sequence of calls

call MPI\_TYPE\_CONTIGUOUS(10, MPI\_REAL, tenrealtype, ierr)

call MPI\_TYPE\_COMMIT(tenrealtype, ierr)

call MPI\_SEND(data, 1, tenrealtype, dest, tag,

$ MPI\_COMM\_WORLD, ierr)

CALL MPI\_TYPE\_FREE(tenrealtype, ierr)

is equivalent to the following single call.

call MPI\_SEND(data, 10, MPI\_REAL, dest, tag,

$ MPI\_COMM\_WORLD, ierr)

Both code fragments send a sequence of ten contiguous real values at location data to process dest.

  The constructor MPI\_TYPE\_VECTOR is used to define a type comprising one or more blocks of data elements separated by a constant stride in an array. A call of the form

MPI\_TYPE\_VECTOR(count, blocklen, stride, oldtype, newtype)

defines a derived type newtype comprising count consecutive blocks of data elements with datatype oldtype, with each block containing blocklen data elements, and the start of successive blocks separated by stride data elements. For example, the sequence of calls

float data[1024];

MPI\_Datatype floattype;

MPI\_Type\_vector(10, 1, 32, MPI\_FLOAT, &floattype);

MPI\_Type\_commit(&floattype);

MPI\_Send(data, 1, floattype, dest, tag, MPI\_COMM\_WORLD);

MPI\_Type\_free(&floattype);

is equivalent to the following code.

float data[1024], buff[10];

for (i=0; i<10; i++) buff[i] = data[i\*32];

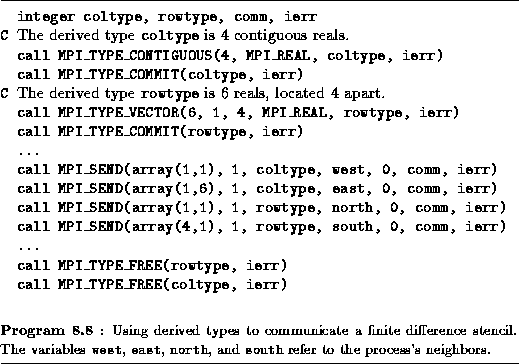
MPI\_Send(buff, 10, MPI\_FLOAT, dest, tag, MPI\_COMM\_WORLD);

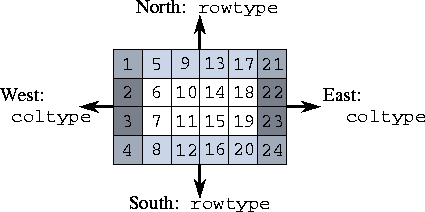
Both send ten floating-point numbers from locations data[0], data[32],..., data[288].

**Example http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1040.gif.http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1038.gif Finite Difference Stencil**:

Program [8.8](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node100.html#progmpfd) uses derived types to communicate the north and south rows and the west and east columns of a http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1039.gifFortran array.   As illustrated in Figure [8.13](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node100.html#figbounds), a column of this array is stored in contiguous locations and can be accessed by using a contiguous derived type. On the other hand, row i of this array (comprising elements array( i ,1), ( i ,2), ... , ( i ,6)) is located in elements i , i +4, ..., i +20. As these elements are not stored in contiguous locations, a call to MPI\_TYPE\_VECTOR is used to define the appropriate type, rowtype.

Program [8.8](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node100.html#progmpfd) frees the derived types that it defines immediately after they are used. In practice, a type might be reused many times before being freed.

    
**Figure 8.13:** A *http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1043.gif*finite difference grid. Areas to be sent to west, east, north, and south neighbors are highlighted.

The third constructor, MPI\_TYPE\_INDEXED, is used to define a type comprising one or more blocks of a primitive or   previously defined datatype, where block lengths and the displacements between blocks are specified in arrays. A call of the form

MPI\_TYPE\_INDEXED(count, lengths, indices, oldtype, newtype)

defines a derived type newtype comprising count consecutive blocks of data elements with datatype oldtype, with block i having a displacement of indices( i ) data elements and containing lengths( i ) data elements.

**Example http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1046.gif.http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1045.gif Fock Matrix Problem**:

In Example [8.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node98.html#exmpfock1) and Program [8.6](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node98.html#progmpfock), we   developed an implementation for a Fock matrix task that receives read   requests containing the address of a single data value. A more realistic program might support messages comprising len/2 indices followed by len/2 block lengths. The MPI\_TYPE\_INDEXED constructor can then be used to return the required values, as follows.

call MPI\_TYPE\_INDEXED(len/2, inbuf(len/2+1), inbuf(1),

$ MPI\_INTEGER, focktype, ierr)

call MPI\_TYPE\_COMMIT(focktype, ierr);

call MPI\_SEND(data, 1, focktype, source, MPI\_COMM\_WORLD,

ierr)

call MPI\_TYPE\_FREE(focktype, ierr)

An alternative approach that does not use the constructor is to accumulate the values that are to be returned in a buffer. The relative efficiency of the two approaches depends on several factors, including the amount of data to be transferred and the capabilities of the computer used to execute the program.

## 8.6.2 MPI Features Not Covered

For simplicity, we have focused on a subset of MPI in this chapter. Of necessity, numerous subtleties have been omitted in this brief description. Also, the following MPI features have not been covered.

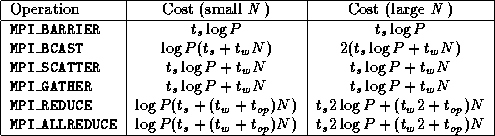
1. Heterogeneous types. Different datatypes can be encapsulated in a single derived type, thereby allowing communication of heterogeneous messages. In addition, data can be sent in ``raw'' format, without   data conversion in heterogeneous networks.
2. Environmental inquiry. A program can obtain information about the environment in which it is running, including information that can be used to tune algorithm performance.
3. Additional global operations. These operations support all-to-all communication and variable-sized contributions from different processes. Additional reduction operations can be used to determine the location of minimum and maximum values and to perform reductions with user-defined functions.
4. Specialized communication modes. These modes include   synchronous communication, which causes the sender to block until the corresponding receive operation has begun; buffered communication, which allows the programmer to allocate buffers so as to ensure that system resources are not exhausted during communications; and   nonblocking communication, which can be more efficient on computers that allow user computation to be overlapped with some of the sending of a message.

# 8.7 Performance Issues

The performance analysis techniques developed in Chapter [3](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node26.html#chapperf) can be applied directly to MPI programs. We discuss some relevant costs here.

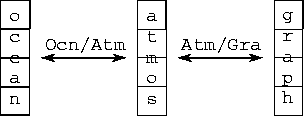
An MPI\_SEND/ MPI\_RECV pair communicates a single message. The cost of this communication can be modeled with Equation [3.1](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node29.html#eq88). The cost of the blocking MPI\_PROBE operation will normally be similar to that of an ordinary receive. The cost of the nonblocking MPI\_IPROBE operation can vary significantly according to implementation: in some implementations it may be negligible, while in others it can be higher than an ordinary receive.

The performance of global operations is less straightforward to analyze, as their execution can involve considerable internal communication. Implementations of these functions can normally be expected to use efficient communication algorithms, such as the hypercube algorithms described in Chapter [11](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node123.html#chapcube). In the absence of bandwidth limitations, these allow a barrier to complete in http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1047.gifsteps on P processors, a broadcast of N words to proceed in time http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1048.gifif N is small and in time http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1049.gifif N is large, and so on. The costs associated with these algorithms are summarized in Table [8.1](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node101.html#tabmpicomm). Remember that on many architectures, bandwidth limitations can increase actual costs, especially for larger messages (Section[3.7](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node33.html#secperfrefine)).

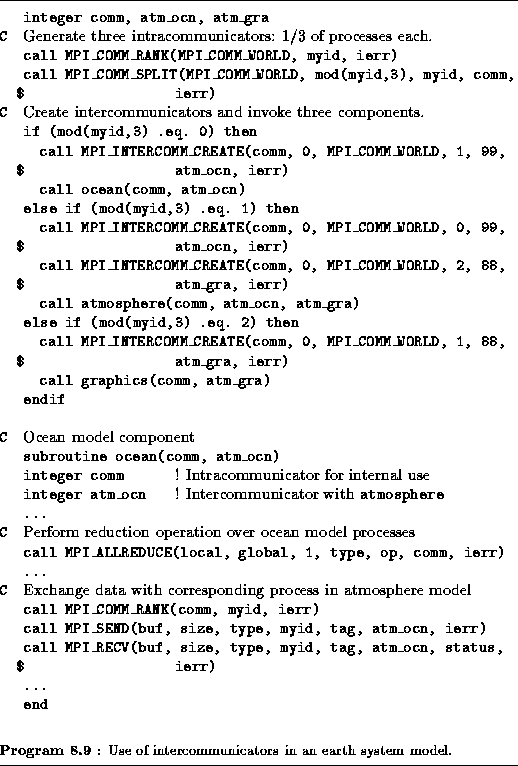
    
**Table 8.1:** Communication costs associated with various MPI global   operations when implemented using hypercube communication algorithms on the idealized multicomputer architecture. The term http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1051.gifrepresents the cost of a single reduction operation.

The MPI\_COMM\_DUP, MPI\_COMM\_SPLIT, and MPI\_COMM\_FREE operations should not normally involve communication. The cost of the MPI\_INTERCOMM\_CREATE operation is implementation dependent, but will normally involve at least one communication operation. The cost of the MPI\_INIT and MPI\_FINALIZE operations is implementation dependent and can be high. However, these functions are called once only in a program execution. Other functions can normally be expected to execute without communication and with little local computation.

# 8.8 Case Study: Earth System Model

    
**Figure 8.14:** Communicators and intercommunicators in an earth system model. Available processes are partitioned into three disjoint groups, each with its own communicator used for internal communication. Intercommunicators allow the atmosphere model to communicate with the ocean model and graphics model.

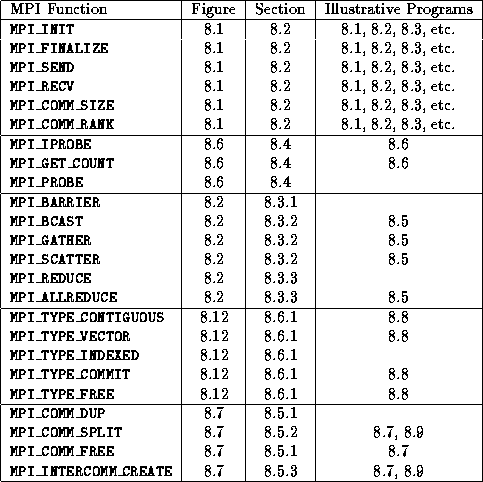
We conclude by showing how the earth system model introduced in Chapter [2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node14.html#chap2) can be constructed in a modular fashion by using MPI communicators. In particular, we consider a hypothetical model constructed   as a parallel composition of atmosphere, ocean, and graphics   components. The atmosphere and ocean models execute concurrently and exchange data periodically; in addition, the atmosphere model sends data periodically to the graphics component, which performs data reduction and rendering functions and outputs high-quality graphics. We allocate the same number of processes to each component; this approach is unlikely to be efficient in practice, but simplifies the presentation.

Program [8.9](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node102.html#progmpes) implements this modular structure (Figure [8.14](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node102.html#figmp4)). The first two lines partition available processes into the three equal-sized, disjoint process groups that will be used to execute the three components. The code following the ``if'' statement establishes intercommunicators that link the atmosphere model with the ocean and graphics components, and initiates execution of the three components. Part of the ocean model code is shown also. This performs a reduction within the ocean model processes by using MPI\_ALLREDUCE and the intracommunicator comm. Then, it exchanges data with corresponding processes in the atmosphere model by using the intercommunicator atm\_ocn.

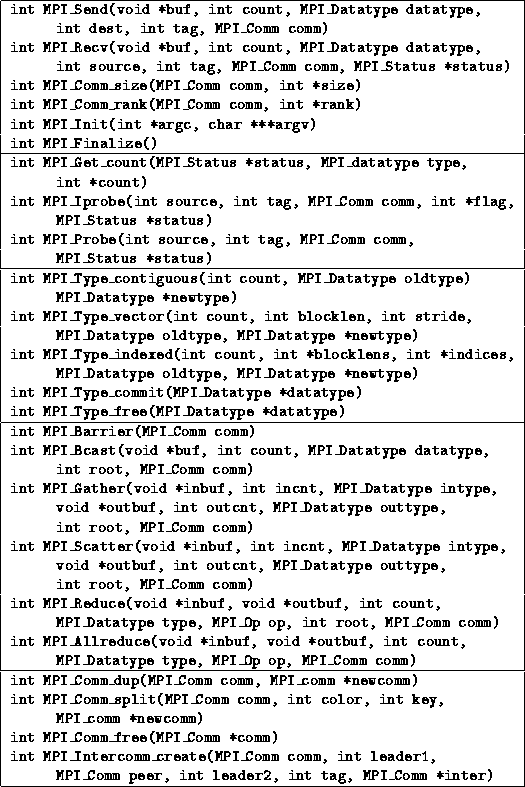
**8.9 Summary**

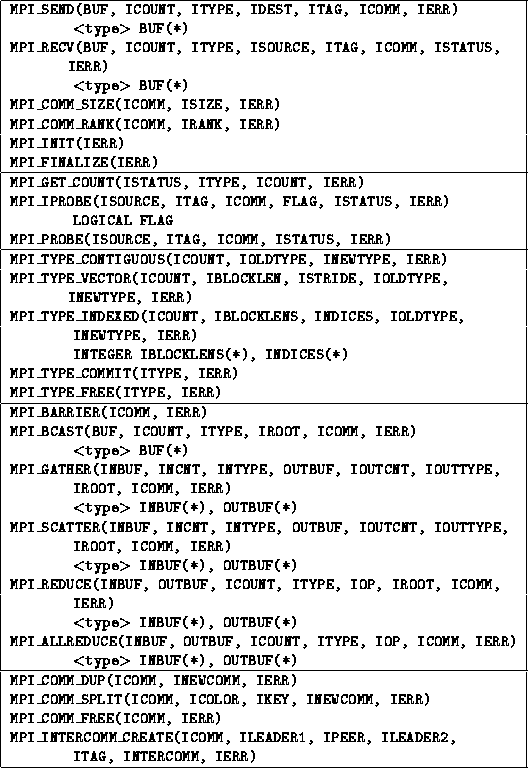
This chapter has described the message-passing library approach to parallel programming and has shown how parallel algorithm designs developed using the techniques from Part I can be translated into message-passing programs. It has also provided a tutorial introduction to the MPI message-passing standard. Table [8.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node103.html#tabmpsum) summarizes the MPI routines described in this chapter; Tables [8.3](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node103.html#tabmpc) and [8.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node103.html#tabmpf) summarize the C and Fortran language bindings, respectively, for these functions and give the types of each function's arguments.

    
**Table 8.2:** MPI quick reference: the functions included in the MPI subset, the figure in which each is defined, the section in which each is described, and the programs that illustrate their use.

The principal features of the message-passing programming model as realized in MPI are as follows.

1. A computation consists of a (typically fixed) set of heavyweight processes, each with a unique identifier (integers 0..P--1).
2. Processes interact by exchanging typed messages, by engaging in collective communication operations, or by probing for pending messages.
3. Modularity is supported via communicators, which allow subprograms to encapsulate communication operations and to be combined in sequential and parallel compositions.
4. Algorithms developed using the techniques set out in Part I can be expressed directly if they do not create tasks dynamically or place multiple tasks on a processor.
5. Algorithms that *do* create tasks dynamically or place multiple tasks on a processor can require substantial refinement before they can be implemented in MPI.
6. Determinism is not guaranteed but can be achieved with careful programming.

    
**Table 8.3:** MPI quick reference: C language binding.

    
**Table:** MPI quick reference: Fortran language binding. For brevity, we adopt the convention that arguments with an I prefix have type INTEGER unless specified otherwise. The ISTATUS argument is always an integer array of size MPI\_STATUS\_SIZE.

**Exercises**

1. Devise an execution sequence for five processes such that Program [8.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node96.html#progmpnondet) yields an incorrect result because of an out-of-order message.
2. Write an MPI program in which two processes exchange a message of size *N* words a large number of times. Use this program to measure communication bandwidth as a function of *N* on one or more networked or parallel computers, and hence obtain estimates for http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1058.gifand http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1059.gif.
3. Compare the performance of the program developed in Exercise [2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node104.html#exmpi1) with an equivalent CC++ or FM program.
4. Implement a two-dimensional finite difference algorithm using MPI. Measure performance on one or more parallel computers, and use performance models to explain your results.
5. Compare the performance of the program developed in Exercise [4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node104.html#exmpi2) with an equivalent CC++ , FM, or HPF programs. Account for any differences.
6. Study the performance of the MPI global operations for different data sizes and numbers of processes. What can you infer from your results about the algorithms used to implement these operations?
7. Implement the vector reduction algorithm of Section [11.2](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node125.html#secvecred) by using MPI point-to-point communication algorithms. Compare the performance of your implementation with that of MPI\_ALLREDUCE for a range of processor counts and problem sizes. Explain any differences.
8. Use MPI to implement a two-dimensional array transpose in which an array of size *N* http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1060.gif*N* is decomposed over *P* processes ( *P* dividing *N* ), with each process having *N/P* rows before the transpose and *N/P* columns after. Compare its performance with that predicted by the performance models presented in Chapter [3](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node26.html#chapperf).
9. Use MPI to implement a three-dimensional array transpose in which an array of size *N* http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1061.gif*N* http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1062.gif*N* is decomposed over http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1063.gifprocesses. Each processor has *(N/P)* http://www.mcs.anl.gov/%7Eitf/dbpp/text/img1064.gif*(N/P)* *x/y* columns before the transpose, the same number of *x/z* columns after the first transpose, and the same number of *y/z* columns after the second transpose. Use an algorithm similar to that developed in Exercise [8](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node104.html#exmpi5) as a building block.
10. Construct an MPI implementation of the parallel parameter study algorithm described in Section [1.4.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node10.html#exdatabase). Use a single manager process to both allocate tasks and collect results. Represent tasks by integers and results by real numbers, and have each worker perform a random amount of computation per task.
11. Study the performance of the program developed in Exercise [10](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node104.html#exmplb) for a variety of processor counts and problem costs. At what point does the central manager become a bottleneck?
12. Modify the program developed in Exercise [10](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node104.html#exmplb) to use a decentralized scheduling structure. Design and carry out experiments to determine when this code is more efficient.
13. Construct an MPI implementation of the parallel/transpose and parallel/pipeline convolution algorithms of Section [4.4](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node43.html#eximage), using intercommunicators to structure the program. Compare the performance of the two algorithms, and account for any differences.
14. Develop a variant of Program [8.8](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node100.html#progmpfd) that implements the nine-point finite difference stencil of Figure [2.22](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node20.html#figsten).
15. Complete Program [8.6](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node98.html#progmpfock), adding support for an accumulate operation and incorporating dummy implementations of routines such as identify\_next\_task.
16. Use MPI to implement a hypercube communication template (see Chapter [11](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node123.html#chapcube)). Use this template to implement simple reduction, vector reduction, and broadcast algorithms.

# Chapter Notes

Message-passing functions were incorporated in specialized libraries developed for early distributed-memory computers such as the Cosmic   Cube [[254](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#Cosmic)], iPSC [[227](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#NX2)], and nCUBE [[211](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#Vertex)].   Subsequent developments emphasized portability across different   computers and explored the functionality required in message-passing   systems. Systems such as Express [[219](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#express)],   p4 [[44](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#p4article),[194](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#p4book)], PICL [[118](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#PICL)],   PARMACS [[143](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#parmacs1),[144](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#parmacs2)], and PVM [[275](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#PVM1)] all run on a   variety of homogeneous and heterogeneous systems. Each focused on a   different set of issues, with the commercially supported Express and PARMACS systems providing the most extensive functionality, p4 integrating shared-memory support, PICL incorporating instrumentation, and PVM permitting dynamic process creation. A special issue of Parallel Computing includes articles on many of these systems [[196](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#PCissue)].

An unfortunate consequence of this exploration was that although various vendor-supplied and portable systems provided similar functionality, syntactic differences and numerous minor incompatibilities made it difficult to port applications from one computer to another. This situation was resolved in 1993 with the   formation of the Message Passing Interface Forum, a consortium of industrial, academic, and governmental organizations interested in standardization [[203](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#mpi1)]. This group produced the MPI specification in early 1994. MPI incorporates ideas developed   previously in a range of systems, notably p4, Express, PICL, and PARMACS. An important innovation is the use of communicators to support modular design. This feature builds on ideas previously   explored in Zipcode [[266](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#zipcode)], CHIMP [[90](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#chimp1),[91](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#chimp2)], and   research systems at IBM Yorktown [[24](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#IBMreport1),[25](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#IBMreport2)].

The presentation of MPI provided in this chapter is intended to be self-contained. Nevertheless, space constraints have prevented inclusion of its more complex features. The MPI standard provides a detailed technical description [[202](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#mpi)]. Gropp, Lusk, and Skjellum [[126](http://www.mcs.anl.gov/%7Eitf/dbpp/text/node132.html#GLS94)] provide an excellent, more accessible tutorial text that includes not only a description of MPI but also material on the development of SPMD libraries and on MPI implementation.

Here is a [Web Tour](http://www.mcs.anl.gov/dbpp/web-tours/mpi.html) providing access to additional information on programming in MPI, including public domain implementations, a tutorial, and example programs.