**UNIT IV**

**DISTRIBUTED MEMORY PROGRAMMING WITH MPI**

MPI program execution – MPI constructs – libraries – MPI send and receive – Point-to-point and Collective communication – MPI derived datatypes – Performance evaluation

**4.Introduction to MPI:**

1. Recall that the world of parallel multiple instruction, multiple data, or MIMD, computers is, for the most part, divided into **distributed-memory** and **shared-memory systems**.
2. From a programmer’s point of view, a **distributed-memory system** consists of a collection of **core-memory pairs** connected by a network, and the memory associated with a core is directly accessible only to that core.
3. On the other hand, from a programmer’s point of view, a **shared-memory system** consists of a collection of cores connected to a globally accessible memory, in which each core can have access to any memory location.
4. In this unit, we’re going to start looking at how to program distributed-memory systems using **message-passin**g.

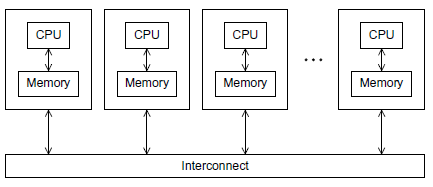


Fig.4.1A distributed-memory system

1. Recall that in message-passing programs,a program running on one core-memory pair is usually called a process, and two processes can communicate by calling functions: one process calls a send function and the other calls a receive function.
2. The implementation of message-passing that we’ll be using is called **MPI**, which is an abbreviation of **Message-Passing Interface**.

**4. 1 GETTING STARTED**

1. Perhaps the ﬁrst program that many of us saw was some variant of the “hello, world” program in Kernighan and Ritchie’s classic text [29]:

#include <stdio.h>

int main(void)

{

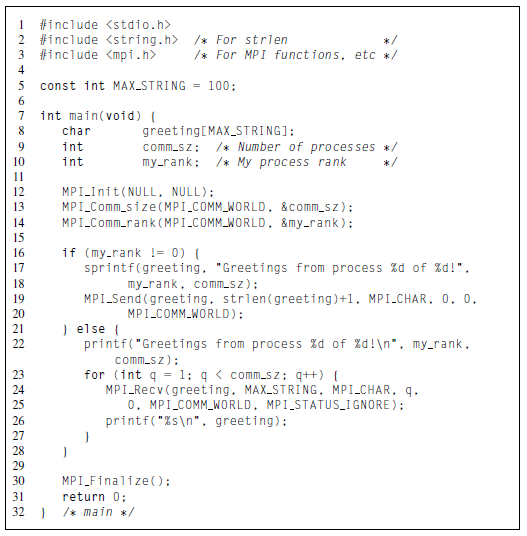
printf("hello, world\n");

return 0;

}

1. Let’s write a program similar to “hello, world” that makes some use of MPI. Instead of having each process simply print a message, we’ll designate one process to do the output, and the other processes will send it messages, which it will print.
2. In parallel programming, it’s common (one might say standard) for the processes to be identiﬁed by nonnegative integer ranks. So if there are p processes, the processes will have ranks **0,1,2,..., p−1**. For our parallel “hello, world,” let’s make **process 0 the designated process, and the other processes will send it messages**. See Program 3.1.

**Compilation and execution**



we’ll assume that we’re using a text editor to write the program source, and the command line to compile and run. Many systems use a command called mpicc for compilation:1

**$ mpicc −o mpii\_hello mpi\_hello.c**

Many systems also support program startup with mpiexec:

**$ mpiexec −n <number of processes> ./mpi\_hello**

So to run the program with one process, we’d type

**$ mpiexec −n 1 ./mpi\_hello**

and to run the program with four processes, we’d type

**$ mpiexec −n 4 ./mpi\_hello**

With one process the program’s output would be

**Greetings from process 0 of 1!**

and with four processes the program’s output would be

**Greetings from process 0 of 4!**

**Greetings from process 1 of 4!**

**Greetings from process 2 of 4!**

**Greetings from process 3 of 4!**

**MPI programs**

1. Line 3 includes the **mpi.h** header ﬁle. This contains prototypes of MPI functions, macro deﬁnitions, type deﬁnitions, and so on; it contains all the deﬁnitions and declarations needed for compiling an MPI program.
2. The second thing to observe is that all of the identiﬁers deﬁned by **MPI\_** start with the string MPI . The ﬁrst letter following the underscore is capitalized for function names and MPI-deﬁned types.

**MPI\_Init and MPI\_Finalize**

1. In Line 12 the call to **MPI\_Init** tells the MPI system to do all of the necessary setup. For example,it might allocate storage for message buffers, and it might decide which process gets which rank**.** As a rule of thumb, no other MPI functions should be called before the program calls MPI\_Init. Its syntax is

int MPI\_Init(

int∗ argc p ,

char∗∗∗ argv p );

1. In Line 30 the call to MPI\_Finalize tells the MPI system that we’re done using MPI, and that any resources allocated for MPI can be freed. The syntax is quite simple:

**int MPI\_Finalize(void);**

In general, no MPI functions should be called after the call to MPI Finalize.

Thus, a typical MPI program has the following basic outline:

. . . #include <mpi.h>

. . . int main(int argc, char∗ argv[])

{

. . .

/∗ No MPI calls before this ∗/

**MPI\_Init(&argc, &argv);**

. . .

**MPI\_Finalize();**

/∗ No MPI calls after this ∗/

. . .

return 0;

}

However,we’ve already seen that it’s not necessary to pass pointers to argc and argv to MPI\_Init. It’s also not necessary that the calls to MPI\_Init and MPI\_Finalize be in main.

**Communicators, MPI\_Comm\_size and MPI\_Comm\_rank**

1. In MPI a **communicator** is a collection of processes that can send messages to each other. One of the purposes of MPI\_Init is to deﬁne a communicator that consists of all of the processes started by the user when she started the program.
2. This communicator is called MPI\_COMM\_WORLD. The function calls in Lines 13 and 14 are getting information about MPI\_COMM\_WORLD.

**int MPI\_Comm\_size( MPI\_Comm comm , int∗ comm\_sz\_p )**

**int MPI\_Comm\_rank( MPI\_Comm comm, int∗ my\_rank\_p);**

* + 1. For both functions, the ﬁrst argument is a communicator and has the special type deﬁned by MPI for communicators,
    2. MPI\_Comm. MPI\_Comm\_size returns in its second argument the number of processes in the communicator, and MPI\_Comm\_rank returns in its second argument the calling process’ rank in the communicator. We’ll often use the variable comm\_sz for the number of processes in MPI\_COMM\_WORLD, and the variable my\_rank for the process rank.

**SPMD programs**

1. Notice that we compiled a single program—we didn’t compile a different program for each process—and we did this inspite of the fact that process 0 is doing something fundamentally different from the other processes: **it’s receiving a series of messages and printing them, while each of the other processes is creating and sending a message.**
2. Recall that this approach to parallel programming is called **single program, multiple data,or SPMD.**The if−else statemet in Lines16 through 28 makes our program SPMD.

**Communication**

1. In Lines 17 and 18, each process, other than process 0, creates a message it will send to process 0. (The function sprintf is very similar to printf, except that instead of writing to stdout, it writes to a string.)
2. Lines 19–20 actually send the message to process 0. Process 0, on the other hand, simply prints its message using printf, and then uses a for loop to receive and print the messages sent by processes 1,2,...,comm sz−1.
3. Lines 24–25 receive the message sent by process q, for q=1,2,...,comm sz−1.

**4.2 MPI\_Send**

1. The sends executed by processes 1,2,...,comm sz−1 are fairly complex, so let’s take a closer look at them. Each of the sends is carried out by a call to MPI Send, whose syntax is

**int MPI\_Send( void∗ msg\_buf\_p ,**

**int msg\_size ,**

**MPI\_Datatype msg\_type ,**

**int dest ,**

**int tag ,**

**MPI\_Comm communicator );**

1. The ﬁrst three arguments, msg\_buf\_p, msg\_size, and msg\_type, determine the contents of the message. The remaining arguments, dest, tag, and communicator, determine the destination of the message.
2. The ﬁrst argument, msg\_buf\_p, is a pointer to the block of memory containing the contents of the message. In our program, this is just the string containing the message, greeting. (Remember that in C an array, such as a string, is a pointer.) The second and third arguments, msg \_size and msg\_type, determine the amount of data to be sent.
3. In our program,the msg\_size argument is the number of characters in the message plus one character for the ‘\0’ character that terminates C strings. The msg\_type argument is MPI\_CHAR. These two arguments together tell the system that the message contains strlen(greeting)+1 chars.
4. Since C types (int, char, and so on.) can’t be passed as arguments to functions, MPI deﬁnes a special type, MPI\_Datatype, that is used for the msg\_type argument. MPI also deﬁnes a number of constant values for this type.
5. The fourth argument, dest, speciﬁes the rank of the process that should receive the message. The ﬁfth argument, tag, is a nonnegative int. It can be used to distinguish messages that are otherwise identical.
6. For example, suppose process 1 is sending ﬂoats to process 0. Some of the ﬂoats should be printed, while others should be used in a computation. Then the ﬁrst four arguments to MPI\_Send provide no information regarding which ﬂoats should be printed and which should be used in a computation.
7. So process 1 can use, say, a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation. The ﬁnal argument toMPI\_Send is a communicator. All MPI functions that involve communication have a communicator argument.

**4.3 MPI\_Recv**

1. The ﬁrst six arguments to MPI\_Recv correspond to the ﬁrst six arguments of MPI\_Send:

int MPI\_Recv( void∗ msg\_buf\_p ,

int buf\_size ,

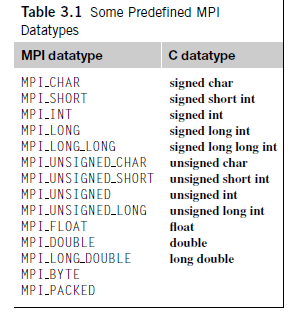
MPI\_Datatype buf\_type,

int source ,

int tag ,

MPI\_Comm communicator ,

MPI\_Status∗ status p );



1. Thus, the ﬁrst three arguments specify the memory available for receiving the message: msg\_buf \_p points to the block of memory, buf\_size determines the number of objects that can be stored in the block, and buf\_type indicates the type of the objects.
2. The next three arguments identify the message. The source argument speciﬁes the process from which the message should be received.
3. The tag argument should match the tag argument of the message being sent, and the communicator argument must match the communicator used by the sending process. The status\_p argument, in many cases it won’t be used by the calling function, and, as in our “greetings” program, the special MPI constan MPI\_STATUS\_IGNORE can be passed.

**Message matching**

Suppose process *q* calls MPI\_Send with

**MPI\_Send(send\_buf\_p, send\_buf\_sz, send\_type, dest, send\_tag, send\_comm);**

Also suppose that process *r* calls MPI\_Recv with

**MPI\_Recv(recv\_buf\_p, recv\_buf\_sz, recv\_type, src, recv\_tag, recv\_comm, &status);**

Then the message sent by *q* with the above call to MPI Send can be received by *r* with the call to MPI \_Recv if

* + - * + recv\_comm = send\_comm,
        + recv\_tag = send\_tag,
        + dest = r, and
        + src = q.

1. These conditions aren’t quite enough for the message to be successfully received, however.
2. The parameters speciﬁed by the ﬁrst three pairs of arguments, send\_buf\_p/recv\_buf\_p, send\_buf\_sz/recv\_buf\_sz, and send\_type/recv\_type, must specify compatible buffers.
3. For detailed rules, see the MPI-1 speciﬁcation [39]. Most of the time, the following rule will sufﬁce: . If recv\_type = send\_type and recv\_buf\_sz ≥ send\_buf\_sz, then the message sent by *q* can be successfully received by *r*.

for (i = 1; i < comm sz; i++)

{

MPI\_Recv(result, result\_sz, result\_type, MPI\_ANY\_SOURCE, result\_tag, comm, MPI\_STATUS\_IGNORE);

Process\_result(result);

}

**The status\_p argument**

1. If you think about these rules for a minute, you’ll notice that a receiver can receive a message without knowing
   * 1. the amount of data in the message,
     2. the sender of the message, or
     3. the tag of the message.
2. So how can the receiver ﬁnd out these values? Recall that the last argument to MPI\_Recv has type MPI\_Status∗. The MPI type MPI\_Status is a struct with at least the three members MPI\_SOURCE, MPI\_TAG, and MPI\_ERROR. Suppose our program contains the deﬁnition

MPI\_Status status;

Then, after a call to MPI\_Recv in which &status is passed as the last argument, we can determine the sender and tag by examining the two members

status.MPI\_SOURCE

status.MPI\_TAG

**4.4 THE TRAPEZOIDAL RULE IN MPI**

Printing messages from processes is all well and good, but we’re probably not taking the trouble to learn to write MPI programs just to print messages. Let’s take a look at a somewhat more useful program—let’s write a program that implements the trapezoidal rule for numerical integration.

**The trapezoidal rule**

Recall that we can use the trapezoidal rule to approximate the area between the graph of a function, y=f(x), two vertical lines, and the x-axis. See Figure 3.3. The basic idea is to divide the interval on the x-axis into n equal subintervals.

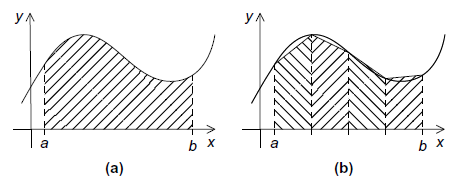
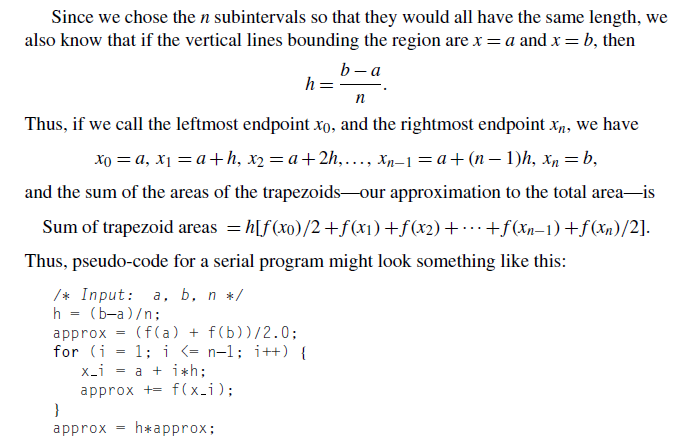
 

Fig. The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids



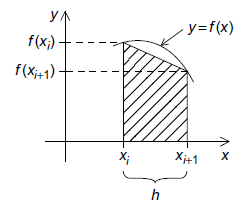


Fig. One trapezoid

**Parallelizing the trapezoidal rule**

Recall that we can design a parallel program using four basic steps:

1. Partition the problem solution into tasks.
2. Identify the communication channels between the tasks.
3. Aggregate the tasks into composite tasks.
4. Map the composite tasks to cores.

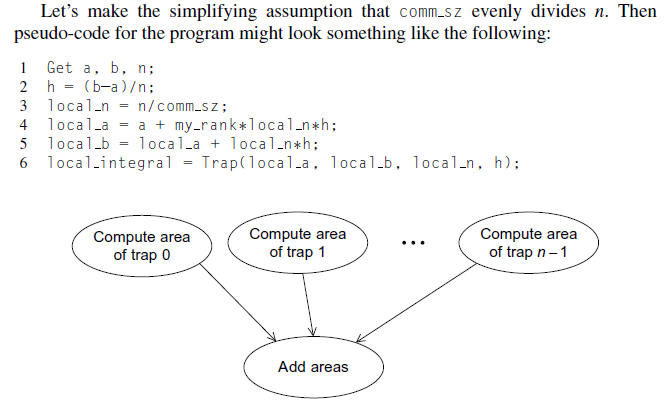
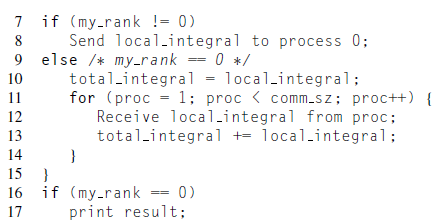
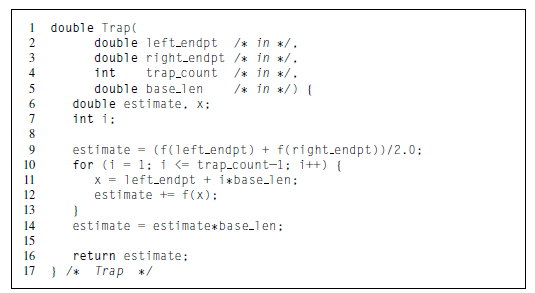
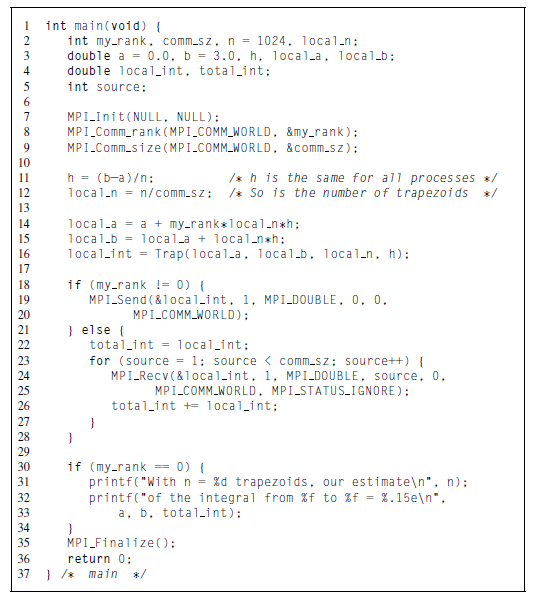


Fig. Tasks and communications for the trapezoidal rule



1. A natural way to do this is to split the interval [a,b] up into comm.\_sz subintervals. If comm\_sz evenly divides n, the number of trapezoids, we can simply apply the trapezoidal rule with n/comm\_sz trapezoids to each of the comm\_sz subintervals. To ﬁnish, we can have one of the processes, say process 0, add the estimates. Let’s make the simplifying assumption that comm\_sz evenly divides n.
2. Let’s defer, for the moment, the issue of input and just “hardwire” the values for a, b,and n.When we do this, we get the MPI program shown. The Trap function is just an implementation of the serial trapezoidal rule.
3. Notice that in our choice of identiﬁers, we try to differentiate between local and global variables. Local variables are variables whose contents are signiﬁcant only on the process that’s using them. Some examples from the trapezoidal rule program are local a, local b, and local n. Variables whose contents are signiﬁcant to all the processes are sometimes called global variables



First version of the MPI trapezoidal rule

**4.5 COLLECTIVE COMMUNICATION**

1. One of the most obvious is that the “global sum” after each process has computed its part of the integral.
2. If we hire eight workers to, say, build a house, we might feel that we weren’t getting our money’s worth if seven of the workers told the ﬁrst what to do, and then the seven collected their pay and went home.
3. But this is very similar to what we’re doing in our global sum. Each process with rank greater than 0 is “telling process 0 what to do” and then quitting.

**Tree-structured communication**

1. As we already saw in Unit 1 we might use a “binary tree structure” like that illustrated in Figure 3.6. In this diagram, initially students or processes 1, 3, 5, and 7 send their values to processes 0, 2, 4, and 6, respectively.
2. Then processes 0, 2, 4, and 6 add the received values to their original values, and the process is repeated twice:

1. a. Processes 2 and 6 send their new values to processes 0 and 4, respectively.

b. Processes 0 and 4 add the received values into their new values.

2. a. Process 4 sends its newest value to process 0.

b. Process 0 adds the received value to its newest value.

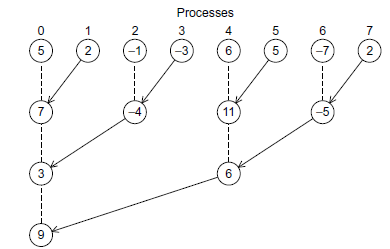


Fig. A tree-structured global sum

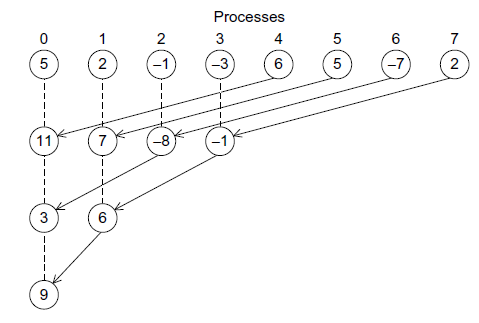
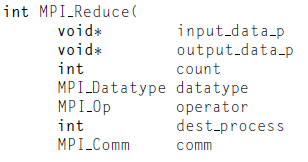


Fig.An alternative tree-structured global sum

**MPI\_Reduce**

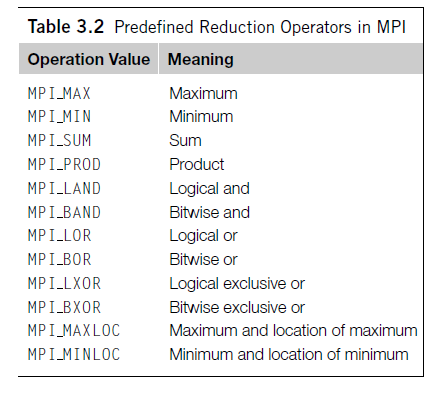
1. Now, a “global-sum function” will obviously require communication. However, unlike the MPI\_Send-MPI\_Recv pair, the global-sum function may involve more than two processes.
2. Infact,in our trapezoidal rule program it will involve all the processes in MPI\_COMM\_WORLD. In MPI parlance, communication functions that involve all the processes in a communicator are called collective communications.
3. To distinguish between collective communications and functions such as MPI\_Send and MPI\_Recv, MPI\_Send and MPI\_Recv are often called point-to-point communications.
4. In fact, global sum is just a special case of an entire class of collective communications.For example,it might happen that instead of ﬁnding the sum of a collection of comm\_sz numbers distributed among the processes.
5. MPI generalized the global-sum function so that any one of these possibilities can be implemented with a single function:

)

The key to the generalization is the ﬁfth argument, operator. It has type MPI Op, which is a predeﬁned MPI type like MPI Data type and MPI\_Comm.There are a number of predeﬁned values in this type. See Table 3.2. It’s also possible to deﬁne your own operators; for details, see the MPI-1 Standard [39].

The operator we want is MPI\_SUM.Using this value for the operator argument,we can replace the code in Lines 18 through 28 of Program 3.2 with the single function call

**MPI\_Reduce(&local\_int, &total\_int, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);**

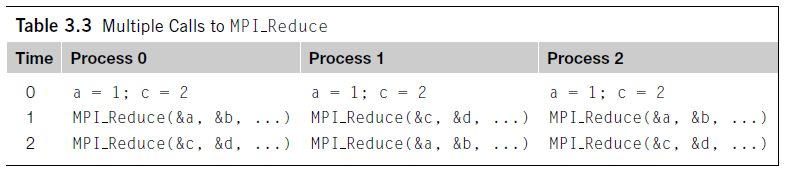


**Collective vs. point-to-point communications**

It’s important to remember that collective communications differ in several ways

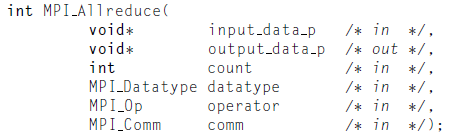
from point-to-point communications:

1. *All the processes in the communicator must call the same collective function. For* example, a program that attempts to match a call to MPI\_Reduce on one process with a call to MPI\_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.
2. The arguments passed by each process to an MPI collective communication must be “compatible.” For example, if one process passes in 0 as the dest\_process and another passes in 1, then the outcome of a call to MPI\_Reduce is erroneous, and, once again, the program is likely to hang or crash.
3. The output data p argument is only used on dest process. However, all of the processes still need to pass in an actual argument corresponding to output data p, even if it’s just NULL.
4. Point-to-point communications are matched on the basis of tags and communicators. Collective communications don’t use tags, so they’re matched solely on the basis of the communicator and the order in which they’re called.
5. The order of the calls will determine the matching, so the value stored in b will be 1+2+1=4, and the value stored in d will be 2+1+2=5.



**MPI\_Allreduce**

1. In our trapezoidal rule program, we just print the result, so it’s perfectly natural for only one process to get the result of the global sum. However, it’s not difﬁcult to imagine a situation in which all of the processes need the result of a global sum in order to complete some larger computation.
2. In this situation, we encounter some of the same problems we encountered with our original global sum. For example, if we use a tree to compute a global sum, we might “reverse” the branches to distribute the global sum.
3. Alternatively, we might have the processes exchange partial results instead of using one-way communications. Such a communication pattern is sometimes called a butterﬂy.



1. The argument list is identical to that for MPI\_Reduce except that there is no dest\_process since all the processes should get the result.

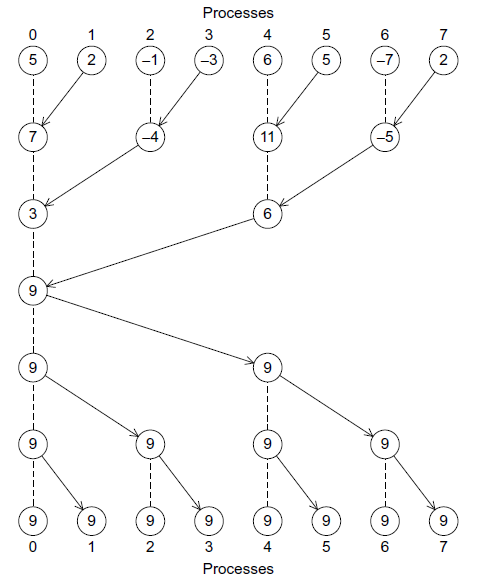
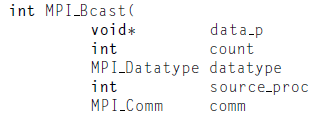


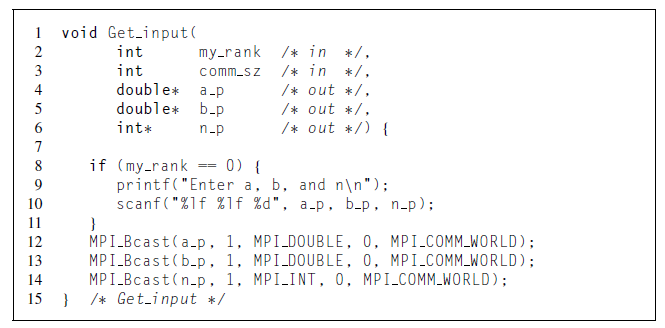
Fig. A global sum followed by distribution of the result

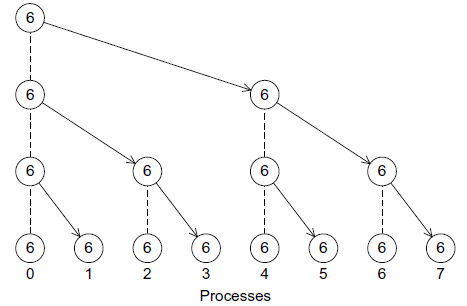
**Broadcast**

A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a **broadcast,** and you’ve probably guessed that MPI provides a broadcast function:

The process with rank source\_proc sends the contents of the memory referenced by data\_p to all the processes in the communicator comm.

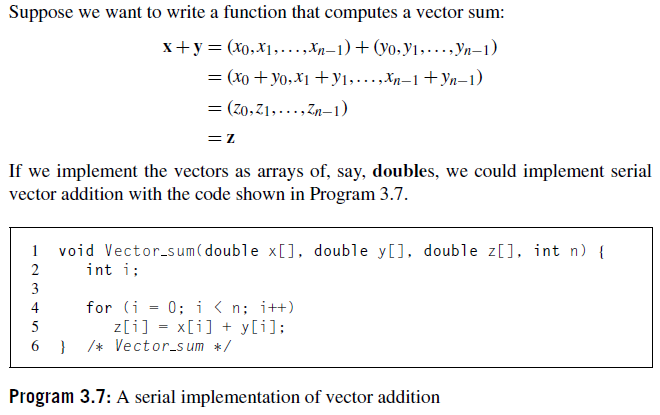
)

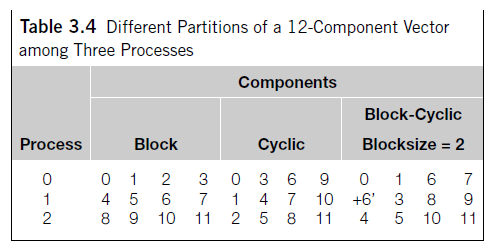




1. The program shows that to modify the Get input function shown in Program. so that it uses MPI Bcast instead of MPI\_Send and MPI\_Recv. Recall that in serial programs, an in/out argument is one whose value is both used and changed by the function.
2. For MPI\_Bcast, however, the data\_p argument is an input argument on the process with rank source proc and an output argument on the other processes.
3. Thus, when an argument to a collective communication is labeled in/out, it’s possible that it’s an input argument on some processes and an output argument on other processes.

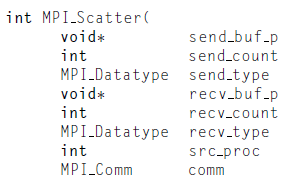
**Data distributions**



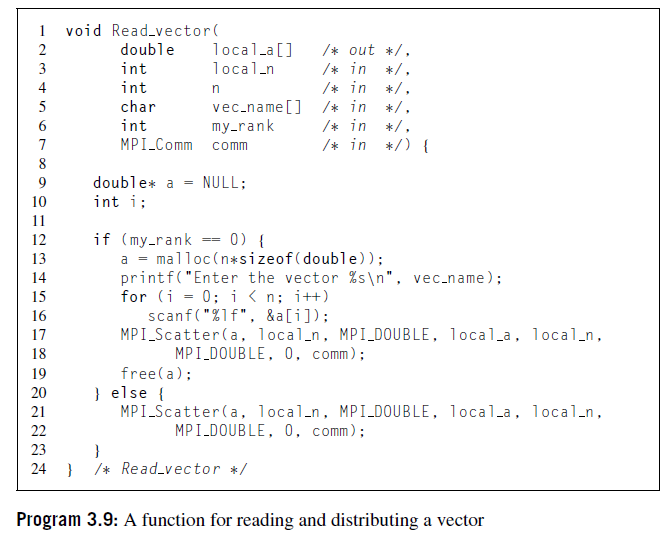


**Scatter**

1. We might try something similar with the vectors: process 0 could read them in and broadcast them to the other processes. However, this could be very wasteful. If there are 10 processes and the vectors have 10,000 components, then each process will need to allocate storage for vectors with 10,000 components, when it is only operating on subvectors with 1000 components.
2. If, for example, we use a block distribution, it would be better if process 0 sent only components 1000 to 1999 to process 1, components 2000 to 2999 to process 2, and so on. Using this approach, processes 1 to 9 would only need to allocate storage for the components they’re actually using.
3. Thus, we might try writing a function that reads in an entire vector that is on process 0 but only sends the needed components to each of the other processes. For the communication MPI provides just such a function:

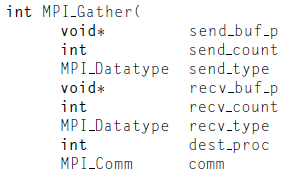
)

1. If the communicator comm contains comm\_sz processes, then MPI Scatter divides the data referenced by send\_buf\_p into comm\_sz pieces—the first piece goes to process0, the second to process 1, the third to process 2, and so on.
2. For example, suppose we’re using a block distribution and process 0 has read in all of an *n-component vector into* Send\_buf\_p. Then, process 0 will get the first local\_n =*n/comm\_sz components,* process 1 will get the next local n components, and so on.
3. Each process should pass its local vector as the recv\_buf\_p argument and the recv\_count argument should be local n. Both send\_type and recv\_type should be MPI\_DOUBLE and src\_proc should be 0.
4. Perhaps surprisingly, send\_count should also be local\_n—send\_count is the *amount of data going to each process; it’s not the amount of data in the memory* referred to by send\_buf\_p.

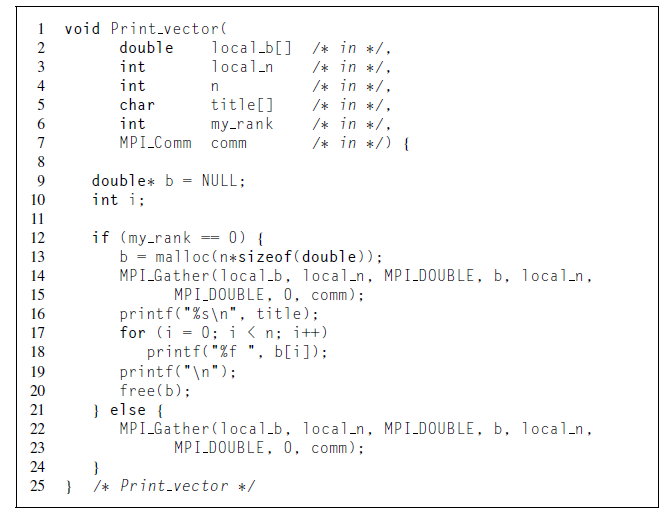


**Gather**

1. Of course, our test program will be useless unless we can see the result of our vector addition, so we need to write a function for printing out a distributed vector.
2. Our function can collect all of the components of the vector onto process 0, and then process 0 can print all of the components. The communication in this function can be carried out by MPI Gather,

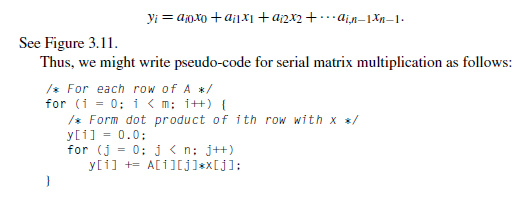
)

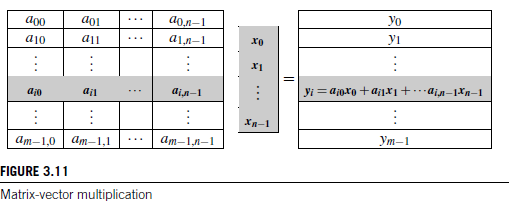
1. The data stored in the memory referred to by send\_buf\_p on process 0 is stored in the ﬁrst block in recv\_buf\_p, the data stored in the memory referred to by send\_buf\_p on process 1 is stored in the second block referred to by recv\_buf\_p, and so on.
2. So, if we’re using a block distribution, we can implement our distributed vector print function as shown in Program 3.10. Note that recv\_count is the number of data items received from each process, not the total number of data items received.



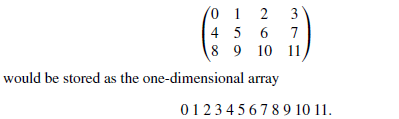
**Allgather**

1. As a ﬁnal example, let’s look at how we might write an MPI function that multiplies a matrix by a vector. Recall that if A=(aij) is an m×n matrix and x is a vector with n components, then y=Ax is a vector with m components and we can ﬁnd the ith component of y by forming the dot product of the ith row of A with x:



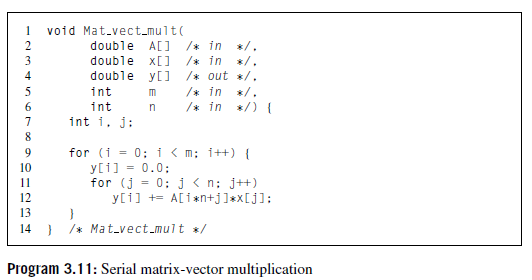


1. In fact, this could be actual C code. However, there are some peculiarities in the way that C programs deal with two-dimensional arrays (see Exercise 3.14), so C programmers frequently use one-dimensional arrays to “simulate” two-dimensional arrays. The most common way to do this is to list the rows one after another.
2. For example, the two-dimensional array



1. In this example, if we start counting rows and columns from 0,then the element stored in row 2 and column 1 in the two-dimensional array (the 9), is located in position **2×4+1=9** in the one-dimensional array.
2. More generally, if our array has n columns, when we use this scheme, we see that the element stored in row i and column j is located in position **i×n+j** in the one-dimensional array.
3. Now let’s see how we might parallelize this function. An individual task can be the multiplication of an element of A by a component of x and the addition of this product into a component of y. That is, each execution of the statement

**y[i] += A[i∗n+j]∗x[j];**

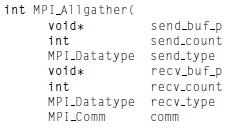


1. So if x has a block distribution, how can we arrange that each process has access to all the components of x before we execute the following loop?

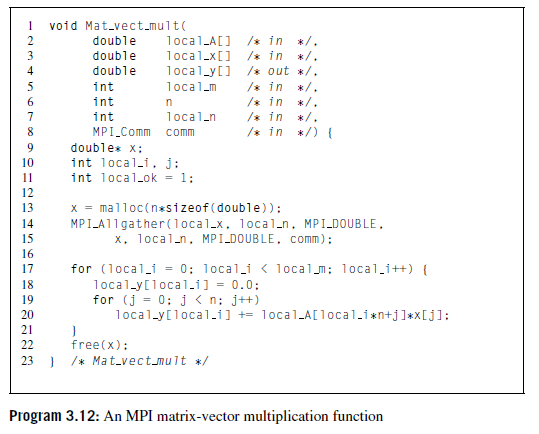
**for (j = 0; j < n; j++)**

**y[i] += A[i∗n+j]∗x[j];**

1. Using the collective communications we’re already familiar with, we could execute a call to MPI\_Gather followed by a call to MPI\_Bcast. This would, in all likelihood, involve two tree-structured communications, and we may be able to do better by using a butterﬂy. So, once again, MPI provides a single function:

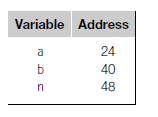
)

1. This function concatenates the contents of each process’ send\_buf\_p and stores this in each process’ recv\_buf\_p. As usual, recv\_count is the amount of data being received from each process, so in most cases, recv \_count will be the same as send count.



**4.6 MPI DERIVED DATATYPES**

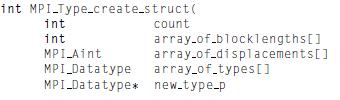
1. In MPI, a derived datatype can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory. The idea here is that if a function that sends data knows the types and the relative locations in memory of a collection of data items,it can collect the items from memory before they are sent.
2. As an example, in our trapezoidal rule program we needed to call MPI\_Bcast three times: once for the **left endpoint a**, once for the **right endpoint b**, and once for the **number of trapezoids n**. As an alternative, we could build a single derived datatype that consists of two doubles and one int. If we do this, we’ll only need one call to MPI\_Bcast. On process 0, a,b, and n will be sent with the one call, while on the other processes, the values will be received with the call.
3. Formally, a derived datatype consists of a sequence of basic MPI datatypes together with a displacement for each of the datatypes. In our trapezoidal rule example,suppose that on process0 the variables a,b,and n are stored in memory locations with the following addresses:



Then the following derived datatype could represent these data items:

**{(MPI\_DOUBLE,0),(MPI\_DOUBLE,16),(MPI\_INT,24)}.**

The ﬁrst element of each pair corresponds to the type of the data, and the second element of each pair is the displacement of the data element from the beginning of the type. We’ve assumed that the type begins with a, so it has displacement 0, and the other elements have displacements measured, in bytes, from a: b is 40−24=16 bytes beyond the start of a, and n is 48−24=24 bytes beyond the start of a. We can use MPI\_Type\_create\_struct to build a derived datatype that consists of individual elements that have different basic types:

)

