CHAPTER 3

Greetings!

WE'LL BEGIN OUR STUDY of *practical* parallel computing by using `MPI_Send` and `MPI_Recv` to write a simple program. This chapter is shorter than most of the other chapters because we want you to start writing parallel programs as soon as possible.

3.1 The Program

The first C program that most of us saw was the "hello, world" program in Kernighan and Ritchie's classic text, *The C Programming Language* [24]. It simply prints the message "hello, world." A variant that makes some use of multiple processes is to have each process send a greeting to another process.

On most parallel systems, the processes involved in the execution of a parallel program are identified by a sequence of nonnegative integers. If there are p processes executing a program, they will have ranks $0, 1, \ldots, p-1$. So one possibility here is for each process other than 0 to send a message to process 0. Of course, we want to know that process 0 received the messages. So we'll have it print them out. Here is an MPI program that does this.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    int         my_rank;       /* rank of process      */
    int         p;             /* number of processes  */
    int         source;        /* rank of sender       */
    int         dest;          /* rank of receiver     */
```

```
    int         tag = 0;        /* tag for messages     */
    char        message[100];   /* storage for message  */
    MPI_Status  status;         /* return status for    */
                                /* receive              */

    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank  */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR,
            dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    /* Shut down MPI */
    MPI_Finalize();
} /* main */
```

3.2 Execution

The details of compiling and executing this program depend on the system you're using. Compiling may be as simple as

```
% cc -o greetings greetings.c -lmpi
```

However, there may also be a special script or makefile for compiling. So ask your local expert how to compile and run a parallel program that uses MPI. When the program is compiled and run with two processes, the output should be

```
Greetings from process 1!
```

If it's run with four processes, the output should be

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
```

Although the details of what happens when the program is executed vary from system to system, the essentials are the same on all systems, provided we run one process on each processor.

1. The user issues a directive to the operating system that has the effect of placing a copy of the executable program on each processor.
2. Each processor begins execution of its copy of the executable.
3. *Different processes can execute different statements by branching within the program based on their process ranks.*

This last point is very important. In the most general form of MIMD programming, each process runs a different program. However, in practice, this generality is usually not needed, and the appearance of "each process running a different program" is obtained by putting branching statements within a single program. So in the "Greetings!" program, even though the statements executed by process 0 are essentially different from those executed by the other processes, we avoid writing several distinct programs by including the branching statement

```
if (my_rank != 0)
      ⋮
else
      ⋮
```

This form of MIMD programming is frequently called **single-program multiple-data (SPMD)** programming. Don't confuse it with SIMD programming, since it is a form of MIMD programming. All of the programs in this book will use the SPMD paradigm.

3.3 MPI

Notice that the program consists entirely of conventional C statements and preprocessor directives. MPI is not a new programming language. It's simply a library of definitions and functions that can be used in C (and Fortran) programs. So in order to understand MPI, we just need to learn about a collection of special definitions and functions.

3.3.1 General MPI Programs

Let's begin at the beginning and discuss the MPI statements in the program. Every MPI program must contain the preprocessor directive

```
#include "mpi.h"
```

This file, `mpi.h`, contains the definitions and declarations necessary for compiling an MPI program.

MPI uses a consistent scheme for MPI-defined identifiers. All MPI identifiers begin with the string "`MPI_`." The remaining characters of most MPI constants are in capitals (e.g., `MPI_CHAR`). The first character of the remainder of the name of each MPI function is capitalized and subsequent characters are lowercase (e.g., `MPI_Init`).

Before any other MPI functions can be called, the function `MPI_Init` must be called, and it should only be called once. Its parameters are pointers to the `main` function's parameters—`argc` and `argv`. It allows systems to do any special setup so that the MPI library can be used. After a program has finished using the MPI library, it must call `MPI_Finalize`. This cleans up any "unfinished business" left by MPI—e.g., it frees memory allocated by MPI. So a typical MPI program has the following layout:

```
          .
          .
          .
#include "mpi.h"
          .
          .
          .
main(int argc, char* argv[]) {
          .
          .
          .
    /* No MPI functions called before this */
    MPI_Init(&argc, &argv);
          .
          .
          .
    MPI_Finalize();
    /* No MPI functions called after this */
          .
          .
          .
} /* main */
          .
          .
          .
```

Note that it is not necessary to call `MPI_Init` as the first executable statement in the program, or even in `main`: it must be called *before any other MPI function is called.* Similarly, it is not necessary to call `MPI_Finalize` as the last executable statement or even in `main`: it must be called *at some point following the last call to any other MPI function.*

3.3.2 Finding Out about the Rest of the World

Since the flow of control in an SPMD program depends on the rank of a process, MPI provides the function `MPI_Comm_rank`, which returns the rank

of a process in its second parameter. The first parameter is a **communicator**. Essentially a communicator is a collection of processes that can send messages to each other. For now, the only communicator we'll need is `MPI_COMM_WORLD`. It is predefined in MPI and consists of all the processes running when program execution begins.

Many of the constructs in our programs also depend on the number of processes executing the program. So MPI provides the function `MPI_Comm_size` for determining this. Its first parameter is a communicator. It returns the number of processes in a communicator in its second parameter.

3.3.3 Message: Data + Envelope

The actual message passing in our program is carried out by the MPI functions `MPI_Send` and `MPI_Recv`. The first command sends a message to a designated process. The second receives a message from a process. These are the most basic message-passing commands in MPI.

Before we discuss the details of their parameter lists, let's look at some of the problems involved in message passing. Suppose process *A* wants to send a message to process *B*, and there is some type of physical connection between *A* and *B*—e.g., a wire. Consider an analogous situation: Amy wants to send a message to Bob. In this case the physical connection is the route the postal service uses when it collects and delivers the letter. Amy proceeds by composing the letter, putting it in an envelope, addressing and stamping the envelope, and dropping the letter in a mailbox. The postal service collects the mail in the mailbox and delivers the letter to Bob's house. Bob checks his mail, finds the letter, opens it, and reads Amy's message.

The analogy with Amy's composing the message is clear: *A* must compose the message; i.e., put it in a buffer. *A* must "drop the message in a mailbox" by calling `MPI_Send`. In order for the postal service, or message-passing system, to know where to deliver the message, it must be addressed. This is done by "enclosing the message in an envelope" and adding an address. Physically this corresponds to adding some information to the actual data that *A* wishes to send. But just the address isn't enough. Since the physical message is just a sequence of electrical signals, the system needs to be able to determine where the message ends. One solution is to also add the size of the message. Another solution is to mark the end of the message with a special symbol. In either case, the number of elements in the message and their type can be used to identify the end of the message. Thus, typical message-passing systems "enclose" messages in **envelopes**. Among other things, the envelope contains the destination of the message and information identifying the size or end of the message. Certainly this information, destination and size of message, should be enough for *B* to receive the message, and in order to receive the message, it calls `MPI_Recv`.

However, let's see if we can make some trouble for Bob and our processes. Bob receives three types of correspondence at his office: junk mail, personal

mail from acquaintances, and personal mail from strangers, and he wants his secretary to sort his mail into these three categories. Bob is planning to reply to the mail from the acquaintances, read the personal mail from strangers, and toss the junk mail. An analogy for the processes might be that *A* sends messages to *B* that ask for data, while *C* sends messages to *B* that contain information *B* needs in order to do calculations, and *D* sends data that should be printed. Bob's secretary will sort his mail by looking at the return addresses on the envelopes. The obvious analogy for message-passing computing is to add the address of the source process to the envelope so that *B* can take appropriate action.

Now suppose the Presto Computer Corporation is sending some advertising information to Bob, and an employee of Presto, whom Bob doesn't know, is also sending Bob some information that Bob requested on Presto's new superchip. What should Bob's secretary do? He's received junk mail and personal mail from a single source—the Presto Computer Corp. His secretary will probably be able to distinguish between the junk mail and the personal mail by looking at the size of the envelope or the amount of postage. An analogy for processes might be that *B* receives floats from several processes. Some of these floats should be printed, while others should be stored in an array, and a single process can send both floats to be printed and floats to be stored. How is *B* to distinguish between the two different types? Neither the source nor the size of the message is sufficient. Two possible solutions come to mind:

- Each process sends two messages. The first specifies whether the float is to be printed or stored, and the second contains the actual float.
- Each process can send a single message, a string, that contains both the float and whether the float is to be printed or stored.

There are problems with both. The biggest problem with the first is that it is very expensive on current generation systems to send messages—the general rule of thumb is to send as few as possible. Another problem with the first is that if *B* is receiving lots of messages, the first message of a pair may get "separated" from the second by other messages, and it may be difficult, or impossible, to decide which messages should be paired. If we use the second approach, the sending process must "encode" the data into a string before sending, and the receiving process must "decode" the data from the string after receiving it. This is time-consuming. Furthermore, there may be a loss of precision when the float is encoded as a string and then decoded.

The solution to this problem that has become standard on message-passing systems is the use of **tags** or **message types.** A tag or message type is just an int specified by the programmer that the system adds to the message envelope. In our setting, for example, the programmer might decide that floats to be printed should have tag 0, and floats to be stored should have tag 1. When *B* receives the message, it checks the tag on the envelope and acts accordingly. There is nothing special about these numbers. MPI guarantees that the integers

0–32767 can be used as tags. Most implementations allow much larger values. Since message type can be easily confused with datatype, in order to avoid confusion, in the remainder of the book we'll always use tag rather than type.

A final issue for Bob: Bob has two occupations. To his coworkers he's known as a mild-mannered programmer, but he is also a secret agent, and as part of his work as a secret agent, he routinely conducts huge financial transactions with some disreputable characters. Of course he doesn't want his perfectly respectable secretary reading mail from these characters. So he rents a post office box, and all of his secret correspondence is sent to the post office box.

An analogous problem for the processes might occur in this situation: Amy has written a large program that solves a system of differential equations. As part of her solution, she needs to solve systems of linear equations, but she doesn't feel that she has the expertise necessary to write code that will do this efficiently. So she acquires a library of functions that solves systems of linear equations, and her program simply calls functions from this library. If the routines in the library need to do message passing (and they will in this case), how can Amy's program distinguish between messages it sends and messages sent by routines in the library? They might accidentally use the same tags. The solution adopted by MPI is to add a further piece of information to the message envelope: a *communicator*. We mentioned before that a communicator is a collection of processes that can send messages to each other. Further, two processes using distinct communicators cannot receive messages from each other. So Amy can get her program to distinguish between its messages and the library's messages by passing a communicator to the library that is different from any communicator(s) she uses in the parts of the program that she has written. Thus, a final piece of information for the envelope is a communicator.

In summary, then, the message envelope contains at least the following information:

1. The rank of the receiver
2. The rank of the sender
3. A tag
4. A communicator

3.3.4 Sending Messages

OK. Now it should be fairly clear what most of the parameters of `MPI_Send` and `MPI_Recv` are. The exact syntax for the two functions is

```
int MPI_Send(
        void*         message    /* in */,
        int           count      /* in */,
        MPI_Datatype  datatype   /* in */,
        int           dest       /* in */,
```

Table 3.1 Predefined MPI datatypes

MPI datatype	*C datatype*
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

```
        int             tag         /* in */,
        MPI_Comm        comm        /* in */)

int MPI_Recv(
        void*           message     /* out */,
        int             count       /* in  */,
        MPI_Datatype    datatype    /* in  */,
        int             source      /* in  */,
        int             tag         /* in  */,
        MPI_Comm        comm        /* in  */,
        MPI_Status*     status      /* out */)
```

The contents of the message are stored in a block of memory referenced by the parameter `message`. The next two parameters, `count` and `datatype`, allow the system to determine how much storage is needed for the message: the message contains a sequence of `count` values, each having *MPI* type `datatype`. This type is not a C type, although most of the predefined MPI datatypes correspond to C types. The predefined MPI types and the corresponding C types (if they exist) are listed in Table 3.1. The last two types, `MPI_BYTE` and `MPI_PACKED`, don't correspond to standard C types—we'll discuss them later. It should be noted that there may be additional MPI types if the system supports additional C types. For example, if the system has type `long long int`, then there should be an MPI type `MPI_LONG_LONG_INT`.

Note that the amount of space allocated for the receiving buffer does not have to match the exact amount of space in the message being received. For example, when our program is run, the size of the message that process 1 sends, `strlen(message) + 1`, is 26 chars, but process 0 receives the message in a buffer that has storage for 100 characters. This makes sense. In general,

the receiving process may not know the exact size of the message being sent. So MPI allows a message to be received as long as there is sufficient storage allocated. If there isn't sufficient storage, an overflow error occurs.

The parameters `dest` and `source` are, respectively, the ranks of the receiving and the sending processes. MPI allows `source` to be a wildcard. There is a predefined constant `MPI_ANY_SOURCE` that can be used if a process is ready to receive a message from any sending process rather than a particular sending process. There is not a wildcard for `dest`.

As we noted earlier, MPI has two mechanisms specifically designed for partitioning the message space: tags and communicators. The parameters `tag` and `comm` are, respectively, the tag and communicator. The `tag` is an `int`, and, for now, our only communicator is `MPI_COMM_WORLD`, which is predefined on all MPI systems and consists of all the processes running when execution of the program begins. There is a wildcard, `MPI_ANY_TAG`, that `MPI_Recv` can use for the tag. There is no wildcard for the communicator. In other words, in order for process *A* to send a message to process *B*, the argument `comm` that *A* uses in its call to `MPI_Send` must be identical to the argument that *B* uses in its call to `MPI_Recv`, while *A* must use a tag and *B* can receive with either an identical tag or `MPI_ANY_TAG`.

Note that the possible use of wildcards for the arguments `source` and `tag` by `MPI_Recv`, but not by `MPI_Send` "matches a 'push' communication mechanism, where data transfer is effected by the sender (rather than a 'pull' mechanism, where data transfer is effected by the receiver)" [28, 29].

The last parameter of `MPI_Recv`, `status`, returns information on the data that was actually received. It references a struct with at least three members—one for the source, one for the tag, and one for an error code. Their names are

```
status -> MPI_SOURCE
status -> MPI_TAG
status -> MPI_ERROR
```

So if, for example, the source of the receive was `MPI_ANY_SOURCE`, then `status -> MPI_SOURCE` will contain the rank of the process that sent the message. There may be additional, implementation-specific members.

The `status` parameter also returns information on the size of the message received. However, this is not directly accessible to the user as a member. In order to determine the size of the message received, we can call

```
int MPI_Get_count(
        MPI_Status*   status      /* in  */,
        MPI_Datatype  datatype    /* in  */,
        int*          count_ptr   /* out */)
```

The status doesn't contain a member for the count since this may involve an unnecessary computation. For example, if the system records the number

of bytes received, then every call to MPI_Recv would have to carry out a division to convert bytes to the number of elements received. In most cases, this information probably won't be needed. So in the relatively rare instances when it is needed, the program can call MPI_Get_count.

As with almost all other MPI functions, both MPI_Send and MPI_Recv have integer return values. These return values are error codes: if the function detected an error, it can return an int indicating the nature of the error. However, the default behavior of MPI implementations is to abort execution of the program if an MPI function detects an error. In Chapter 9 we'll discuss how to change this default behavior, but for the time being we'll ignore the return values.

3.4 Summary

MPI is not a new programming language. It is a collection of functions and macros, or a *library* that can be used in C programs.

The programs that we will write in this text use the *single-program multiple-data (SPMD)* model. In this model, each process runs the same executable program. However, the processes execute different statements by taking different branches in the program: the branches are determined by the process rank.

Every MPI program must include the preprocessor directive

```
#include "mpi.h"
```

This includes the declarations and definitions necessary for compiling an MPI program. MPI uses a consistent scheme for MPI-defined identifiers. All MPI identifiers begin with the string "MPI_." The remaining characters of most MPI constants are capital letters. The first character of the remainder of the name of each MPI function is capitalized and subsequent characters are lowercase (e.g., MPI_Init).

Before any other MPI function is called, our program must call

```
int MPI_Init(
        int*    argc    /* in/out */,
        char**  argv[]  /* in/out */)
```

After our program is finished using MPI, it must call

```
int MPI_Finalize(void)
```

In order for a process to find out how many processes are involved in the execution of a program, it can call

```
int MPI_Comm_size(
        MPI_Comm  comm                  /* in  */,
        int*      number_of_processes   /* out */)
```

A *communicator* is a collection of processes that can send messages to each other. MPI_COMM_WORLD is a predefined communicator: it consists of all the processes running when program execution begins. In order for a process to find out its rank, it can call

```
int MPI_Comm_rank(
        MPI_Comm  comm      /* in  */,
        int*      my_rank   /* out */)
```

Actual message passing is accomplished using the two functions

```
int MPI_Send(
        void*         message    /* in */,
        int           count      /* in */,
        MPI_Datatype  datatype   /* in */,
        int           dest       /* in */,
        int           tag        /* in */,
        MPI_Comm      comm       /* in */)

int MPI_Recv(
        void*         message    /* out */,
        int           count      /* in  */,
        MPI_Datatype  datatype   /* in  */,
        int           source     /* in  */,
        int           tag        /* in  */,
        MPI_Comm      comm       /* in  */,
        MPI_Status*   status     /* out */)
```

The parameter message refers to the actual data being transmitted. The parameters count and datatype determine the size of the message. MPI_Recv doesn't need to know the exact size of the message being received, but it must have at least as much space as the size of the message it's trying to receive.

The tag and comm are used to make sure that messages don't get mixed up. Since MPI_Recv can use wildcards for source and tag, the status parameter returns the source and tag of the message that was actually received.

Each message consists of two parts: the data being transmitted and the envelope. The envelope of a message contains

1. the rank of the receiver
2. the rank of the sender
3. a tag
4. a communicator

3.5 References

A detailed discussion of the syntax and semantics of the various MPI functions is contained in the MPI Standard [28, 29]. There are also very complete dis-

cussions in [34] and [21]. Both [21] and [17] provide introductory discussions of MPI.

3.6 Exercises

1. Create a C source file containing the "Greetings!" program. Find out how to compile it and run it on different numbers of processors. What is the output if the program is run with only one process? How many processors can you use?
2. Modify the "Greetings!" program so that it uses wildcards in the receives for both `source` and `tag`. Is there any difference in the output of the program?
3. Try modifying some of the parameters to `MPI_Send` and `MPI_Recv` (e.g., `count`, `datatype`, `source`, `dest`). What happens when you run your program? Does it crash? Does it hang (i.e., stop in the middle of execution without crashing)?
4. Modify the "Greetings!" program so that all the processes send a message to process $p-1$. On many parallel systems, every process can print to the screen of the terminal from which the program was started. Have process $p-1$ print the messages it receives. What happens? Can process $p-1$ print to the screen?

3.7 Programming Assignment

1. Write a program in which process i sends a greeting to process $(i+1)\%p$. (Be careful of how i calculates from whom it should receive!) Should process i send its message to process $i+1$ first and then receive the message from process $i-1$? Should it first receive and then send? Does it matter? What happens when the program is run on one processor?