CHAPTER 9

Debugging Your Program

IN THIS CHAPTER we'll take a look at a somewhat controversial subject: debugging. Why controversial? Well, there are certain individuals who assert that we should design mathematically correct programs and, as a consequence, we should never need to do any debugging. On the other hand, most programmers find that they spend huge amounts of time debugging. So rather than taking the moral high ground (and thereby saving ourselves the trouble of writing this chapter), we will concede that programmers make mistakes and spend some time talking about how to find those mistakes. However, let's stress that the time spent debugging can be minimized by carefully designing and developing the program.

We'll begin with a short review of debugging serial programs. Then we'll continue with a discussion of some additional problems encountered in debugging parallel programs. The bulk of the chapter will be devoted to testing and debugging a small parallel program.

9.1 Quick Review of Serial Debugging

If you do much programming, you probably consider yourself to be an expert on the subject of debugging, but just in case, let's take a few minutes to discuss some of the main issues and techniques in debugging serial programs.

The typical debugging process is cyclical: the program is run. When a bug is encountered, we have a number of options. We'll discuss three that are most frequently used:

1. Examine the source code.
2. Add debugging output statements to the program.

3. Use a symbolic debugger.

After identifying and correcting the bug, we repeat the process. Let's look at each of the methods we listed for locating bugs.

9.1.1 Examine the Source Code

The first approach is most useful if we have a fairly small program and the nature of the errors we're encountering is suggestive of where the problem is located. As an example, let's try to debug the following program:

```
/* bug.c
 * Program that tries to read a list of floats and sort
 *    them in increasing order.
 * Warning!  This program is definitely incorrect!
 *
 * Input:
 *    size of list (int)
 *    list of floats
 *
 * Output:
 *    Sorted list
 *
 * Algorithm:
 *    1. Get list size
 *    2. Get first input float.
 *    3. For each new element
 *        (a) read it in
 *        (b) use linear search to determine where it
 *            should be inserted
 *        (c) insert it by shifting greater elements down
 *            one
 *    4. Print list
 */
#include <stdio.h>

/* Maximum list size */
#define MAX 100

/* Function for printing contents of list */
void Print_x(char* title, float x[], int size);

main() {
    int   num_vals;   /* Input list size                */
    float x[MAX];     /* Storage for the sorted list    */
    float temp;       /* Most recently read input value */
    int   i, j, k;    /* Subscripts.  i:  counts input  */
                      /* values, j:  position to insert */
                      /* new value.                     */
```

```
    printf("How many input values?\n");
    scanf("%d",num_vals);

    printf("Now enter each value.\n");
    /* Get first value */
    scanf("%f", &(x[0]));

    for (i = 1; i < num_vals; i++) {
        scanf("%f", &temp);

        /* Determine where to insert */
        j = i - 1;
        while ((temp < x[j]) && (j > 0))
            j--;

        /* Insert */
        for (k = i; k > j; k++)
            x[k] = x[k-1];
        x[j] = temp;
    }

    Print_x("Contents of x", x, num_vals);
} /* Print_x */

void Print_x(char* title, float x[], int size) {
    int i;

    printf("%s\n", title);
    for (i = 0; i < size; i++)
        printf("%f ", x[i]);
    printf("\n");
} /* Print_x */
```

If we compile the program,

```
% cc -o bug bug.c
```

and we attempt to run it, we get the prompt

```
% bug
How many input values?
```

Then, when we enter a value, we get the dreaded

```
Segmentation fault
```

If you've done much C programming, this is highly suggestive: a segmentation fault just after entering an input value suggests that we made classic error number 1:

```
scanf("%d",num_vals);
```

We forgot to pass a *pointer* to `num_vals`.

It should be noted that some systems aren't nearly so obliging in identifying errors. For example, one system happily reads in the `num_vals` and the first two input floats and crashes later on with the message

```
Bus error
```

This evil system evidently decided that the `int` passed to `scanf` was OK, and the program failed to crash until it hit the inner `for` loop. Unfortunately, this would make it *much* more difficult to diagnose. So, we can't blithely assume that programs will crash when they ought to.

This brings up an important point:

> *It can be virtually impossible to predict the behavior of an erroneous program.*

Many students devote tremendous amounts of energy to trying to decide how their program "ought" to have behaved with a given collection of errors. Don't waste your time. Life is too short.

9.1.2 Add Debugging Output

As you may have already noticed, we haven't quite fixed up our program. Let's see what happens when we try again.

```
% cc -o bug bug.c
% bug
How many input values?
3
Now enter each value.
1 2 3
Segmentation fault
```

The dreaded "Segmentation fault" again! Let's attack this error by adding some diagnostic output. Basically, we would like to get a snapshot of the state of the program at various points during execution. One of the easiest ways to do this is to print out the values of the variables at various strategic points. Let's add a new function that takes care of this for us.

```
void Snapshot(char* title, int num_vals, float x[],
    int i, int j,  int k, float temp) {

    printf("**********************\n");
    printf("%s\n", title);
    printf("num_vals = %d, i = %d, temp = %f\n",
        num_vals, i, temp);
```

```
        printf("j = %d, k = %d\n", j, k);
        Print_x("x = ", x, i);
        printf("**********************\n\n");
        fflush(stdout);
    }
```

The use of `fflush` as the last statement in the function guarantees that everything is printed. On most systems I/O is buffered, so the output won't actually be printed until the output buffer is full. If our program crashes, data in the output buffer may be lost. However, the use of `fflush` will force the system to flush the buffer even if it isn't full, and we're assured that we'll see all our output.

Of course, in a large program, it won't be possible to print the values of all the variables, but the principle is the same: you simply print the values of the variables that are of interest.

Now we'll call `Snapshot` in "strategic" places. Obvious candidates (especially in view of our recent experience) are after the calls to `scanf`. For variables that are uninitialized we'll simply use "0" as the corresponding argument to `Snapshot`. The output (together with our input) is now

```
How many input values?
3
**********************
After getting num_vals
num vals = 3, i = 0, temp = 0.000000
j = 0, k = 0
x =

**********************

Now enter each value.
1 2 3
**********************
After getting first val
num_vals = 3, i = 1, temp = 0.000000
j = 0, k = 0
x =
1.000000
**********************

**********************
After getting next val
num_vals = 3, i = 1, temp = 2.000000
j = 0, k = 0
x =
1.000000
**********************

Segmentation fault
```

Not much help, although we now know that the program is crashing somewhere inside the main `for` loop on its first pass. Let's get rid of these calls to `Snapshot` and add some calls to the body of the loop: one after the `while` loop and one after the inner `for` loop.

```
How many input values?
3
Now enter each value.
1 2 3
**********
After computing j
num_vals = 3, i = 1, temp = 2.000000
j = 0, k = 0
x =
1.000000
**********

Segmentation fault
```

OK. We never got to the second call to `Snapshot`—the one after the inner `for` loop. So the immediate problem must be there. Taking a look, we see

```
/* Insert */
for (k = i; k > j; k++)
    x[k] = x[k-1];
```

Well, we should have caught this a while ago: `k` is being *incremented*. It should be decremented!

9.1.3 Use a Debugger

In theory, the easiest way to locate bugs is with the use of a symbolic debugger. A **symbolic debugger**, or, more often, a **debugger,** is a program that acts as an intermediary between the programmer, the system, and the program. The description of the capabilities of the GNU debugger (`gdb`) in the `gdb` man page are instructive:

> `gdb` can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:
>
> - Start your program, specifying anything that might affect its behavior.
> - Make your program stop on specified conditions.
> - Examine what has happened, when your program has stopped.
> - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Although this excerpt describes the capabilities of `gdb`, it could be applied equally well to most debuggers.

The big problem is that debuggers tend to be difficult to use. Although their principal function is to help the programmer, the commands tend to have somewhat obscure syntax, and, if we fail to type exactly the correct thing, they may either print a cryptic error message or, even worse, completely ignore our message. In spite of this, they can be extremely useful—especially if they have a well-designed graphical user interface.

Perhaps the most widely used debugger is `dbx`—most UNIX systems provide some version of it. So let's use it to find the remaining bugs in our program.

In order to use `dbx` (and most debuggers) we need the compiler to generate a full symbol table: a mapping between the internal representation of the program objects and our source representation. For example, if the variable `x` begins in memory location 804,780,616, then we need to include in our table the correspondence $\texttt{x} \longleftrightarrow 804,780,616$. With this information, when we request the value of, say, `x[0]`, the debugger can translate that into something like "the value stored at address 804,780,616." On most systems, the full symbol table is generated with the `-g` option to the compiler. For example,

```
% cc -g -o bug bug.c
```

It should be noted that compiler optimizations can seriously confuse a debugger. So you shouldn't use optimization flags (e.g., `-O`) when you're debugging and you probably shouldn't use the `-g` option on "production" codes.

Having compiled the program with the `-g` option, we can now run it under the control of the debugger.

```
% dbx bug
dbx Version 3.1
Type 'help' for help.
reading symbolic information ...
(dbx)
```

The exact details of the information printed during startup vary from system to system. Eventually, however, we'll get the `(dbx)` prompt. Now we can run our program by typing `run` followed by any command line arguments we might use to start our program directly from the shell prompt.

```
(dbx) run
How many input values?
3
Now enter each value.
1 2 3
Contents of x
2.000000 3.000000 1.000000

execution completed (exit code 1)
(dbx)
```

Except for a few comments generated by the debugger, the program runs just the way it runs when we start it from the system prompt. Notice that when our program wants to read input from the keyboard, we can type the input just as we would when we run our program directly from the shell.

OK. Not surprisingly, there are still bugs in our program. But we don't really have any information about where they are. We need to get `dbx` to give us some information on what the program is doing during execution. In order to do this, we can stop the program during execution and look at what has happened, and in order to do this, we can add some **breakpoints.** That is, we can stop the program at "interesting points" during execution and check things out. Since the elements of the list are incorrectly sorted, it would seem that one interesting point is after the computation of `j`—the position to insert the new value. How do we find out where to put the breakpoint? We can get `dbx` to list segments of the source code with the `list` command:

```
(dbx) list 35,45
   35       for (i = 1; i < num_vals; i++) {
   36           scanf("%f", &temp);
   37
   38           /* Determine where to insert */
   39           j = i - 1;
   40           while ((temp < x[j]) && (j > 0))
   41               j--;
   42
   43           /* Insert */
   44           for (k = i; k > j; k--)
   45               x[k] = x[k-1];
```

Its syntax is

```
list <firstline>, <lastline>
```

and it prints the source code starting at line `firstline` and ending at line `lastline`. As it can be rather tedious to search through the source program using `list`, it's usually much more convenient to open a window containing the source code, or use something like C-shell job control to switch back and forth between the source listing and `dbx`.

We can only insert breakpoints at the beginning of functions or on lines where an executable statement begins. So if we try putting a breakpoint at line 41, 42, or 43, `dbx` will simply ignore it. Our breakpoint should be at line 44. We can add it by typing

```
(dbx) stop at 44
[2] stop at 44
```

The numeral printed before the echoing of our command is a reference number—`dbx` has identified this breakpoint with the numeral 2. So if, for

example, we want to remove the breakpoint later on, we can reference it with the number 2. We don't have to remember this: we can get a list of active breakpoints with the `status` command.

```
(dbx) status
[2] stop at 44
```

Incidentally, we also don't need to memorize all of the commands in `dbx`. Each implementation has a help command. Its details vary from system to system, but you can always get started with just

```
(dbx) help
```

Getting back to our program, we have completed one execution, and we've inserted a breakpoint after the computation of `j`. Let's run the program and see what happens.

```
(dbx) run
How many input values?
3
Now enter each value.
1 2 3
[2] stopped in main at line 44
   44            for (k = i; k > j; k--)
```

Now let's see what's going on.

```
(dbx) print i
1
(dbx) print x[0]
1.0
(dbx) print temp
2.0
(dbx) print j
0
```

Aha! The program is going to try to insert 2.0 in the wrong place. So whatever else is wrong with our program, it's clear that it's not computing `j` correctly.

Now we have basically two options: we can continue to run the program under the control of the debugger—trying to analyze under what conditions it fails—or we can go back to the drawing board. For this program, I think we should go back to the drawing board . . . . Whatever, *you* decide, you should correct the program as a matter of conscience. But for the moment, we're done with `dbx`. So let's

```
(dbx) quit
```

9.2 More on Serial Debugging

Of course, we've barely scratched the surface of debugging. There are a number of other approaches that can be quite useful. It would be very helpful to have a catalog of debugging techniques, common programming errors, and tentative diagnoses of common failures. Fortunately, there are a number of books that address these issues. See the references at the end of the chapter.

9.3 Parallel Debugging

If you've been writing the programs at the ends of the chapters, you've probably already discovered that debugging a parallel program is a good deal more challenging than debugging a serial program. This isn't surprising: we would expect that communication among the processes would increase complexity. Unfortunately, and not surprisingly, the state of the art in parallel debugging is not as far advanced as it is in serial debugging: it remains an active research topic. As we'll discuss below, the development of parallel debuggers remains an active area for research and, as we've already seen, generating output from a parallel program can be extremely difficult.

9.4 Nondeterminism

In addition to all these problems, parallel programs can exhibit **nondeterministic** behavior—the exact sequence of computations and communications performed on each of the processes may vary from execution to execution. As a simple example, consider the following program fragment. Its purpose is to perform a global reduction, where the reduction operation is 2×2 matrix multiplication.

```
/* mat_mult.c
 * Multiply a sequence of 2x2 matrices--1 factor from
 *     each process.  Erroneous.
 *
 * Input: none
 *
 * Output: product of a sequence of 2x2 matrices
 *
 * Algorithm
 *    1. Generate local matrix
 *    2. Send local matrix to process 0
 *    3  if (my_rank == 0)
 *    3a.    for each process, receive matrix and
 *              multiply by product
 *    3b.    print product
```

```
 *
 * Notes:
 *    1. The matrices are stored as linear arrays.  The
 *       correspondence is row major: Matrix[i][j] <->
 *       Array[2*i + j]
 *    2. Local matrices have the form
 *            [my_rank    my_rank+1]
 *            [my_rank+2  my_rank  ]
 */
#include <stdio.h>
#include "mpi.h"

#define MATRIX_ORDER 2
#define ARRAY_ORDER 4

void Initialize(float my_matrix[], int my_rank);
void Mult(float product[], float factor[]);
void Print_matrix(char* title, float matrix[]);

main(int argc, char* argv[]) {
    float        my_matrix[ARRAY_ORDER];
    float        temp[ARRAY_ORDER];
    float        product[ARRAY_ORDER] = {1, 0, 0, 1};
                     /* product is the identity matrix */
    int          p;
    int          my_rank;
    MPI_Status   status;
    int          i;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    Initialize(my_matrix, my_rank);

    MPI_Send(my_matrix, ARRAY_ORDER, MPI_FLOAT, 0, 0,
        MPI_COMM_WORLD);

    if (my_rank == 0) {
        for (i = 0; i < p; i++) {
            MPI_Recv(temp, ARRAY_ORDER, MPI_FLOAT,
                MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
            Mult(product, temp);
        }
        Print_matrix("The product is", product);
    }

    MPI_Finalize();
} /* main */
```

The first time we run the program with three processes, the output is

```
The product is
10.000000 11.000000
20.000000 14.000000
```

However, when we run the program later, the output is

```
The product is
10.000000 10.000000
22.000000 14.000000
```

What's going on? The local matrices are

$$\text{Process 0:} \quad A_0 = \begin{pmatrix} 0 & 1 \\ 2 & 0 \end{pmatrix}$$

$$\text{Process 1:} \quad A_1 = \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$$

$$\text{Process 2:} \quad A_2 = \begin{pmatrix} 2 & 3 \\ 4 & 2 \end{pmatrix}$$

So one might guess that the product should be

$$A_0A_1A_2 = \begin{pmatrix} 10 & 11 \\ 20 & 14 \end{pmatrix}.$$

How did it come up with

$$\begin{pmatrix} 10 & 10 \\ 22 & 14 \end{pmatrix}?$$

You may have guessed that the key is that matrix multiplication is not commutative, and the program is multiplying the matrices in different orders. In fact,

$$A_0A_2A_1 = \begin{pmatrix} 10 & 10 \\ 22 & 14 \end{pmatrix},$$

and this is exactly what's happening. The problem is that our calls to `MPI_Recv` have source argument `MPI_ANY_SOURCE`. So the first time we ran the program, process 0 received the matrices in the order A_0, A_1, A_2; during the second run, the matrices were received in the order A_0, A_2, A_1. This situation is sometimes called a **race**: the processes are in a race to get their matrices sent off to process 0. This time the problem is easy to solve—just replace the source argument to `MPI_Recv`, `MPI_ANY_SOURCE`, with the loop index `i`.

Unless we intentionally introduce randomness into a serial program, it will execute exactly the same statements every time we run it on a fixed data set. So how does this nondeterminism occur in parallel programs? There are several possible reasons for this: they generally have to do with "inequalities"

among the physical processors and communication links and the order and rate at which processes are started. For example, recall that in distributed-memory systems, there may be no global clock. So the processors may run at different rates. In our example, if, say, process 1 is loaded onto a relatively fast processor during the first execution, and a relatively slow processor during the second, while the situation is reversed for process 2, we shouldn't be surprised that the order in which they complete their setups is reversed, and, as a consequence, the order in which their matrices arrive at process 0 is reversed. On a network of workstations or any parallel system that allows multitasking on the processors, this effect will be exacerbated by the unpredictable fluctuations in the system and network loads.

In our example, nondeterminism is a bug. However, there are many cases where it is a feature. As an example, suppose we try to search a large tree (e.g., a game tree) using a simple "client-server" program. Process 0 is the server: it keeps a store of subtrees that need to be searched. The other processes are the clients: they request subtrees from process 0, which they, in turn, search. As subtrees are searched, new subtrees will be generated, and "surplus" subtrees can be sent back to process 0. When a client completes a search of a subtree, it can request a new subtree from process 0. It is not difficult to see that the subtrees searched by, say, process 1 may vary from execution to execution, even if the main tree is the same and the number of processes is the same. In this case, there is no reason to regard this as a bug: the output of the program (if it's properly coded) should be equally correct regardless of the actual sequence of subtrees examined by a given process.

9.5 An Example

In order to get a better idea of the problems and methods of debugging parallel programs, let's take a look at an extended example. The program is supposed to find the time that it takes to forward messages of different sizes around a ring of processes. For each message size, it will repeatedly forward a message around a ring and compute the average, maximum, and minimum times for the circuit.

The heart of the program is forwarding the message once around the ring: each process receives the message from its predecessor in the ring and then sends it on to its successor. Although, in general, the sequence defined by process ranks in `MPI_COMM_WORLD` won't correspond to a ring of underlying physical processors, we'll not worry about this in order to simplify the code. It should be easy to use MPI's topology functions to reorder the processes so that they form a one-dimensional ring.

In order to take the actual timings, we use the MPI function

```
double MPI_Wtime()
```

This function returns the "wall clock" time in seconds since some time in the past.

9.5.1 The Program?

Simplicity itself. So here is the *not entirely bug-free* source code.

```
/* comm_time.c
 * Time communication around a ring of processes.
 *     Guaranteed to have bugs.
 *
 * Input: None (see notes).
 *
 * Output:  Average, minimum, and maximum time for messages
 *    of varying sizes to be forwarded around a ring of
 *    processes.
 *
 * Algorithm:
 *    1.  Allocate and initialize storage for messages
 *        and communication times
 *    2.  Compute ranks of neighbors in ring.
 *    3.  For each message size
 *    3a.     For each test
 *    3b.         Start clock
 *    3c.         Send message around loop
 *    3d.         Add elapsed time to running sum
 *    3e.         Update max/min elapsed time
 *    4.  Print times.
 *
 * Functions:
 *    Initialize:  Allocate and initialize arrays
 *    Print_results:  Send results to IO_process
 *        and print.
 *
 * Notes:
 *    1. Due to difficulties some MPI implementations
 *       have with input, the number of tests, the max
 *       message size, the min message size, and the size
 *       increment are hardwired.
 *    2. We assume that the size increment evenly divides
 *       the difference max_size - min_size
 */
#include <stdio.h>
#include "mpi.h"
#include "cio.h"

void Initialize(int max_size, int min_size, int size_incr,
    int my_rank, float** x_ptr, double** times_ptr,
    double** max_times_ptr, double** min_times_ptr,
    int* order_ptr);
```

```
void Print_results(MPI_Comm io_comm, int my_rank,
    int min_size, int max_size, int size_incr,
    int time_array_order, int test_count,
    double* times, double* max_times, double* min_times);

main(int argc, char* argv[]) {
    int         test_count = 1000; /* Number of tests  */
    int         max_size = 10000;  /* Max msg. length  */
    int         min_size = 0;      /* Min msg. length  */
    int         size_incr = 1000;  /* Increment for    */
                                   /*     msg. sizes   */
    float*      x;                 /* Message buffer   */
    double*     times;             /* Elapsed times    */
    double*     max_times;         /* Max times        */
    double*     min_times;         /* Min times        */
    int         time_array_order;  /* Size of timing   */
                                   /*     arrays.      */
    double      start;             /* Start time       */
    double      elapsed;           /* Elapsed time     */
    int         i, test, size;     /* Loop variables   */
    int         p, my_rank, source, dest;
    MPI_Comm    io_comm;
    MPI_Status  status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);
    Cache_io_rank(MPI_COMM_WORLD, io_comm);

    Initialize(max_size, min_size, size_incr, my_rank,
        &x, &times, &max_times, &min_times,
        &time_array_order);

    source = (my_rank - 1) % p;
    dest = (my_rank + 1) % p;

    /* For each message size, find average circuit time */
    /*     Loop var size = message size                 */
    /*     Loop var i = index into arrays for timings   */
    for (size = min_size, i = 0; size <= max_size;
            size = size + size_incr, i++) {
        times[i] = 0.0;
        max_times[i] = 0.0;
        min_times[i] = 1000000.0;
        for (test = 0; test < test_count; test++) {
            start = MPI_Wtime();
            MPI_Recv(x, size, MPI_FLOAT, source, 0,
                 MPI_COMM_WORLD, &status);
            MPI_Send(x, size, MPI_FLOAT, dest, 0,
                 MPI_COMM_WORLD);
```

```
                elapsed = MPI_Wtime() - start;
                times[i] = times[i] + elapsed;
                if (elapsed > max_times[i])
                    max_times[i] = elapsed;
                if (elapsed < min_times[i])
                    min_times[i] = elapsed;
            }
        } /* for size . . . */

        Print_results(io_comm, my_rank, min_size, max_size,
            size_incr, time_array_order, test_count, times,
            max_times, min_times);

        MPI_Finalize();
    } /* main */

    /*********************************************************/
    void Initialize(int max_size, int min_size, int size_incr,
        int my_rank, float** x_ptr, double** times_ptr,
        double** max_times_ptr, double** min_times_ptr,
        int* order_ptr) {
        int i;

        *x_ptr = (float *) malloc(max_size*sizeof(float));

        *order_ptr = (max_size - min_size)/size_incr;
        *times_ptr =
           (double *) malloc((*order_ptr)*sizeof(double));
        *max_times_ptr =
           (double *) malloc((*order_ptr)*sizeof(double));
        *min_times_ptr =
           (double *) malloc((*order_ptr)*sizeof(double));

        /* Initialize buffer--why this? */
        for (i = 0; i < max_size; i++)
            (*x_ptr)[i] = (float) my_rank;
    } /* Initialize */


    /*********************************************************/
    /* Send results from process 0 in MPI_COMM_WORLD to      */
    /* I/O process in io_comm, which prints the results.     */
    void Print_results(MPI_Comm io_comm, int my_rank,
        int min_size, int max_size, int size_incr,
        int time_array_order, int test_count, double* times,
        double* max_times, double* min_times) {
        int        i;
        int        size;
        MPI_Status status;
```

```
        int         io_process;
        int         io_rank;

        Get_io_rank(io_comm, &io_process);
        MPI_Comm_rank(io_comm, &io_rank);

        if (my_rank == 0) {
            MPI_Send(times, time_array_order, MPI_DOUBLE,
                io_rank, 0, io_comm);
            MPI_Send(max_times, time_array_order, MPI_DOUBLE,
                io_process, 0, io_comm);
            MPI_Send(min_times, time_array_order, MPI_DOUBLE,
                io_process, 0, io_comm);
        }
        if (io_rank == io_process) {
            MPI_Recv(times, time_array_order, MPI_DOUBLE,
                MPI_ANY_SOURCE, 0, io_comm, &status);
            MPI_Recv(max_times, time_array_order, MPI_DOUBLE,
                MPI_ANY_SOURCE, 0, io_comm, &status);
            MPI_Recv(min_times, time_array_order, MPI_DOUBLE,
                MPI_ANY_SOURCE, 0, io_comm, &status);

            printf("Message size (floats):  ");
            for (size = min_size;
                 size <= max_size; size += size_incr)
                printf("%10d ", size);
            printf("\n");
            printf("Avg circuit time (ms):  ");
            for (i = 0; i < time_array_order; i++)
                printf("%10f ",1000.0*times[i]/test_count);
            printf("\n");
            printf("Max circuit time (ms):  ");
            for (i = 0; i < time_array_order; i++)
                printf("%10f ",1000.0*max_times[i]);
            printf("\n");
            printf("Min circuit time (ms):  ");
            for (i = 0; i < time_array_order; i++)
                printf("%10f ",1000.0*min_times[i]);
            printf("\n\n");
            fflush(stdout);
        }
    } /* Print_results */
```

Notice that we hardwired the parameters that specify the message sizes and the number of messages. This was done in order to avoid some of the problems associated with I/O on some implementations of MPI.

Also notice that this program is definitely broken. What? You didn't notice? Well, OK. Let's try to debug it.

9.5.2 Debugging The Program

Perhaps the first thing we should be aware of is that the behavior of a buggy program depends strongly on the system on which it's run. As a consequence, when you run the program on your system, the errors that appear may be different from those that occurred when we ran the program. We'll look at several different systems: an nCUBE 2 running the `mpich` implementation of MPI, a network of workstations running `mpich`, and a network of workstations running the `LAM` implementation of MPI.

In general, it's a good idea to make initial tests relatively small. If the program is broken, but it doesn't crash, it could run for a long time before we even realize there's a problem, and we'll have wasted a lot of time and compute power. With the given "input" data,

```
int         test_count = 1000; /* Number of tests  */
int         max_size = 10000;  /* Max msg. length  */
int         min size = 0;      /* Min msg. length  */
int         size_incr = 1000;  /* Increment for    */
                               /*      msg. sizes  */
```

our program *should* generate 11,000 full circuits before it completes. Let's change this to something minimal.

```
int         test_count = 2;    /* Number of tests  */
int         max_size = 1000;   /* Max msg. length  */
int         min_size = 1000;   /* Min msg. length  */
int         size_incr = 1000;  /* Increment for    */
                               /*      msg. sizes  */
```

This should only run through two circuits. Of course, this won't test different sizes. So before we're through testing, we will need to run the program with `min_size` < `max_size`.

Let's compile it and see what happens with just one process .... (What does the program do with just one process? What is a one-process "circuit"?) On the nCUBE and on a network running `LAM`, it just hangs. On a network running `mpich`, it prints

```
p0_361: Bailing out
```

and quits. (The "p0" indicates process 0, and the "361" is the process id assigned to the UNIX process started by the program.) So there can't be any doubt that the program is broken.

9.5.3 A Brief Discussion of Parallel Debuggers

The failures didn't give us much information about what's wrong. A seemingly simple solution to the problem of finding out where the program is hanging (or crashing) would be to run it under the control of a debugger. (To check

where a program is hanging, we can just send an interrupt when the program hangs, and then do a backtrace with the command `where`.) Unfortunately, the development of debuggers for parallel systems remains an active research subject, and, as a consequence, there isn't much standardization and it's difficult to provide general guidance in their use.

In this case, however, since we are running under a single process, it shouldn't be too difficult to get things going. The first job is to figure out what command the *system* uses to start your job. For example, suppose you're using the `LAM` implementation on a network of workstations. Then you probably started your job with something like

```
% mpirun -v n0 comm0
```

Here, `comm0` is the name of the executable created by the compiler. The obvious thing to do is to start up `dbx`:

```
% dbx comm0
dbx version 3.19 Nov  3 1994 19:59:46
Executable /usr/people/peter/mpi.test/comm0
(dbx)
```

and then try starting your program

```
(dbx) mpirun -v n0 comm0

"mpirun" is not a valid command name.

Apparent syntax error in examine command. expected / ? ,
(dbx)
```

The problem is that `dbx` doesn't know about `mpirun`. So we need to figure out what `mpirun` does to start the process, then we can try executing this inside `dbx`. To do this, we can start the program using `mpirun` and use the `ps` command to see what is actually being run.

```
% mpirun -np 1 comm0
14554 comm0 running on n0 (o)
% ps -ef
              :
peter 14554 14523  0 14:40:58 ?         0:00 comm0
              :
%
```

We started the program as usual. The command `ps -ef` generated a list of all processes on the system. The details of this command will be different on other systems—check with your local expert. We've deleted the processes we're not interested in. The important point is that `mpirun` simply executes the command

```
comm0
```

So we can just type

```
(dbx) run
```

to start our program in dbx. Be careful here: different systems will be substantially different in the details of how this is done; as usual, check with your local expert.

Let's see what happens when we start the program under the control of dbx.

```
% dbx comm0
dbx version 3.19 Nov  3 1994 19:59:46
Executable /usr/people/peter/mpi.lam/comm0
(dbx) run
Process 14576 (comm0) started
^C
Interrupt
Process 14576 (comm0) stopped on signal SIGINT:
Interrupt (handler cipc_catastrophe)
[_recvfrom:12 +0x8,0xfad5b44]
Source (of recvfrom.s) not available for Process 14576
(dbx) where
>  0 _recvfrom(0x6, 0x7fffabec, 0x28, 0x0)
["recvfrom.s":12, 0xfad5b44]
   1 _cio_recvfrom(0x6, 0x7fffabec, 0x28, 0x0)
["../../../otb/t/kreq/clientio2.udp.c":391, 0x41897c]
   2 _cio_kreqback(0x6, 0x7fffabec, 0x28, 0x0)
["../../../otb/t/kreq/clientio.udp.c":244, 0x414e7c]
   3 _cipc_ksr(0x7fffac14, 0x7fffabec, 0x28, 0x0)
["../../../otb/t/kreq/couter.c":301, 0x41b0fc]
   4 ksr(0x7fffac9c, 0x7fffabec, 0x28, 0x0)
["../../../../share/kreq/ksr.c":63, 0x41b870]
   5 dsfr(0x6, 0x7fffabec, 0x7fffacf4, 0x0)
["../../../../share/nreq/dsfr.c":67, 0x418f24]
   6 bfrecvsql(0x7fffadc4, 0x7fffabec, 0x28, 0x0)
["../../../../share/nreq/bfrecvsql.c":112, 0x41604c]
   7 dorecv(0x7fffadc4, 0x7fffae1c, 0x28, 0x0)
["../../../share/mpi/lamrecv.c":252, 0x40c950]
   8 __lam_recv_(0x10004cf0, 0x3e8, 0x10002788, 0x0)
["../../../share/mpi/lamrecv.c":160, 0x40c684]
   9 MPI_Recv(0x10004cf0, 0x3e8, 0x10002788, 0x0)
["../../../share/mpi/recv.c":48, 0x406c84]
   10 main(argc = 1, argv = 0x7fffaf24)
["/usr/people/peter/mpi.lam/comm_time0.c":103, 0x404844]
   11 __start() ["crt1text.s":133, 0x40459c]
(dbx)
```

We started the program as planned; when it hung, we interrupted it with `Ctrl-C`. Then we checked to see where it was by typing `where`. Most of the output is from functions that are part of the `LAM` implementation. It doesn't get to our code until the line numbered `10`. Evidently the code hung at line `103` in `main` in the call to `MPI_Recv`. This is highly suggestive of what the problem is, but before proceeding with our debugging, let's talk a little more about parallel debuggers.

As we have noted in several places, the details of how to start a parallel program under the control of a debugger will vary considerably from system to system. For example, on the same network running the `mpich` implementation, we would need to create a `procgroup` file and start the program inside the debugger with

```
(dbx) run -p4pg <procgroup file name>
```

To make matters worse, on some systems it is difficult, or impossible, to start the program from inside the debugger. Under these circumstances, the most common approach is to start the program, get its process number, and then start the debugger, "attaching" it to the running process. A final solution to starting the debugger is to invoke it when the program crashes. This may be done automatically by the system or it may be done by hand after a core dump. In either case, it should be possible to figure out where the program crashed by executing `where`. Once again, check with your local expert for details on whether these options are available and, if so, how to use them on your system.

Unfortunately, it may be difficult to get all this sorted out, and even if we do, our troubles may be just beginning. On a network of workstations, if we're running multiple processes on distinct hosts, we'll probably want to start a debugger on each host in a separate window. When we do this, we need to worry about such things as how setting a breakpoint or source stepping one command at a time on one host will affect the processes on the other hosts. The situation with parallel machines is somewhat better, but as we noted earlier, there is little standardization. As a consequence, we'll leave it to you, the ever-interested reader, to explore (in your vast amount of leisure time) the debuggers on your system.

9.5.4 The Old Standby: `printf/fflush`

Rather than developing a solution using debuggers that probably won't work on your system, let's use the old standby for debugging: `printf/fflush`.

Before going on, we should note that adding I/O statements to a parallel program comes with no guarantees. Indeed, it may cause more problems than it solves. As we'll see, adding I/O statements to a program can introduce new bugs and hide existing bugs. It isn't scalable: it frequently generates unmanageably large amounts of output; if we're using many processes, I/O

statements can dramatically reduce performance. Finally, some systems may have serious problems if too many processes are simultaneously trying to print. In spite of all these drawbacks, it is frequently the only method available. So let's proceed—with caution.

We would like the program to keep a record of what it does so that when it hangs or crashes we'll have a pretty clear idea of where things are going wrong. So let's begin by printing out where we are at a few critical points in the program's execution. Let's print out our location and the values of a few variables before the calls to `Initialize`, `MPI_Recv`, and `MPI_Send`. Since some of us may not be able to use `printf/fflush` statements, let's use `Cprintf` statements on all the processes from the minimal I/O library we developed in Chapter 8. We'll put the following calls in the indicated locations in `main`:

```
Cprintf(io_comm,"","Before Initialize, p = %d,
    my_rank = %d", p, my_rank);
          ⋮
Cprintf(io_comm,"","Before MPI_Recv, source = %d,
    my_rank = %d", source, my_rank);
          ⋮
Cprintf(io_comm,"","Before MPI_Send, dest = %d,
    my_rank = %d", dest, my_rank);
```

Now our output looks like this:

```
Process 0 > Before Initialize, p = 1, my_rank = 0

Process 0 > Before MPI_Recv, source = 0, my_rank = 0
```

and, as before, the program hangs or crashes.

9.5.5 The Classical Bugs in Parallel Programs

The fact that the program hangs in a call to `MPI_Recv` is highly suggestive. Since `MPI_Recv` blocks until it finds data that matches its requirements, it seems likely that the program is hanging for one (or both) of the following reasons:

1. **Trying to receive data before sending in an exchange, or trying to receive data when there has been no send.** The problem here is that since `MPI_Recv` blocks, it will wait forever if there is no data to receive. For example, if process *A* and process *B* are trying to exchange data, then the sends must be carried out before the receives. Consider the code

```
if (my_rank == A) {
    MPI_Recv(&x, count1, datatype1, B, tag1, comm,
        &status);
    MPI_Send(&y, count2, datatype2, B, tag2, comm);
} else { /* my_rank == B */
    MPI_Recv(&y, count2, datatype2, A, tag2, comm,
        &status);
    MPI Send(&x, count1, datatype1, A, tag1, comm);
}
```

It will almost certainly hang: both *A* and *B* may wait forever to receive messages that will never be sent. Note that this problem applies to situations more general than simple exchanges between two processes—e.g., each process in a ring tries to receive a message from its neighbor.

We've encountered this problem in Chapter 5. It is a very simple example of what is commonly called **deadlock.** In general, a deadlocked program is one in which each process is waiting for something that is supposed to occur on some other process. Since all the processes are waiting, the program just hangs.

2. **Incorrect parameters to send/receive.** How the program fails will depend on the system and which parameters are incorrect. For example, in the code

```
if (my_rank == A)
    MPI Send(&x, count, datatype, B, 1, comm);
else /* my_rank == B */
    MPI_Recv(&x, count, datatype, A, 2, comm,
        &status);
```

process *B* will probably hang, and *A* will continue. However, in the code

```
if (my_rank == A)
    MPI_Send(&x, count, datatype, B, 0, comm);
else /* my rank == B.  Should receive from A */
    MPI_Recv(&x, count, datatype, C, 0, comm,
        &status);
```

or the code

```
if (my rank == A) /* Should send to B */
    MPI Send(&x, count, datatype, C, 0, comm);
else /* my rank == B */
    MPI_Recv(&x, count, datatype, A, 0, comm,
        &status);
```

the program may hang or crash.

These are the two classical errors in message-passing programs.

It's probably pretty clear which one is causing us problems: process 0 is trying to receive a message that has never been sent. Indeed, it's pretty clear that the program will hang even if there are more processes: they'll all just sit and wait, since the first message is never sent.

9.5.6 First Fix

In order to fix this, we need to get the message started on process 0. So we should rewrite the code so that process 0 is the unique process that first sends the data. Let's try replacing the nested pair of `for` loops with the following code:

```
for (size = min_size, i = 0; size <= max_size;
        size = size + size_incr, i++) {
    if (my_rank == 0) {
        times[i] =0.0;
        max_times[i] = 0.0;
        min_times[i] = 1000000.0;
        for (test = 0; test < test_count; test++) {
            start = MPI_Wtime();
            MPI_Send(x, size, MPI_FLOAT, dest, 0,
                MPI_COMM_WORLD);
            MPI_Recv(x, size, MPI_FLOAT, source, 0,
                MPI_COMM_WORLD, &status);
            elapsed = MPI_Wtime() - start;
            times[i] = times[i] + elapsed;
            if (elapsed > max_times[i])
                max_times[i] = elapsed;
            if (elapsed < min_times[i])
                min_times[i] = elapsed;
        }
    } else { /* my_rank != 0 */
        for (test = 0; test < test_count; test++) {
            MPI_Recv(x, size, MPI_FLOAT, source, 0,
                 MPI_COMM_WORLD, &status);
            MPI_Send(x, size, MPI_FLOAT, dest, 0,
                 MPI_COMM_WORLD);
        }
    }
} /* for size . . . */
```

We have just rewritten the code so that process 0 starts the circuit by sending and ends the circuit by receiving. The behavior of the other processes is unchanged (except that they no longer take timing data).

Taking the pessimistic (or realistic) view that the program is probably not completely bug free, let's continue to monitor its progress with `Cprintf` statements. Note, however, that it's no longer clear where we should place the

`Cprintf` statements, since processes are doing essentially different things. We could try matching the `Cprintf` associated with the send on 0 with `Cprintfs` associated with receives on the other processes, and vice versa, but this may cause some problems (see below). A less confusing, albeit not so informative, solution is to just put a single `Cprintf` before the `if--then--else` branch.

For the time being, let's take the easy way out and just put the following statement before the `if`:

```
Cprintf(io_comm,"",
    "Before if, my_rank = %d, source = %d, dest = %d",
    my_rank, source, dest);
```

When we run the program with a single process, the behavior is different on different systems. Two systems (nCUBE running `mpich`, network of workstations running `LAM`) produce

```
Process 0 > Before Initialize, p = 1, my_rank = 0

Process 0 > Before if, my_rank = 0, source = 0, dest = 0
Message size (floats):        1000
Avg circuit time (ms):
Max circuit time (ms):
Min circuit time (ms):
```

Let's look at the network running `mpich` later.

9.5.7 Many Parallel Programming Bugs Are Really Serial Programming Bugs

Clearly something is wrong in `Print_results`. Why is the message size correctly printed, but none of the times? Recall how `Print_results` works:

1. The process with rank 0 in `MPI_COMM_WORLD` sends its timing data to the I/O process. (Since we're only using one process, these are, of course, the same process.)
2. The I/O process receives the data and prints it.

It uses four `for` loops to generate the output. The first loops over the message sizes, while the other three loop over the order of the arrays storing the timing data. Explicitly, the first `for` is

```
for (size = min_size; size <= max_size;
    size += size_incr)
```

and the others are

```
for (i = 0; i < time_array_order; i++)
```

So we should check to see what value is being stored in `time_array_order`. Since we're only interested in its value on the I/O process, we can just use `printf/fflush` statements instead of `Cprintf`. Let's check its value just before the first `for` that prints out timing data:

```
printf("io_process = %d, time_array_order = %d\n",
    io_process, time_array_order);
fflush(stdout);
```

Now the output is

```
Process 0 > Before Initialize, p = 1, my_rank = 0

Process 0 > Before if, my_rank = 0, source = 0, dest = 0
Message size (floats):        1000
io_process = 0, time_array order = 0
Avg circuit time (ms):
Max circuit time (ms):
Min circuit time (ms):
```

As we suspected, `time_array_order` is 0. It should be 1—we're checking circuit times for a single message size (1000 floats). There are several possibilities here:

1. Our argument list in the call to `Print_results` may not correspond properly to the function's parameter list.
2. We may have inadvertently modified `time_array_order` at some point in the program.
3. The argument list in the call to `Initialize` may not correspond properly to the function's parameter list.
4. We computed its value incorrectly in `Initialize`.

Notice that none of these possibilities has anything to do with the message passing in our program. The variable `time_array_order` is completely local on each process: it is initialized locally, and it is never sent to another process. In other words, it appears that this error has nothing to do with parallel computing. Rather, it seems that we have just made an error in C programming. This brings up one of the most important points to keep in mind when you're debugging a parallel program:

> *Many (if not most) parallel program bugs have nothing to do with the fact that the program is a parallel program. Many (if not most) parallel program bugs are caused by the same mistakes that cause serial program bugs.*

Enough moralizing (at least for the time being). Let's search through our source code and see if we can find the error .... Sure enough! We've incorrectly calculated `time_array_order` in the first place. Take a look at the initialization in `Initialize`:

```
*order_ptr = (max_size - min_size)/size_incr;
```

Here `order_ptr` is the formal parameter that corresponds to the argument `&time_array_order` in the call to `Initialize`, and since `max_size` and `min_size` have both been initialized to 1000,

$$(\texttt{max\_size} - \texttt{min\_size})/\texttt{size\_incr} = 0.$$

We want `time_array_order` to be 1! So our basic formula is wrong. A little thought and a few example calculations will convince you that the correct formula should be

```
*order_ptr = (max_size - min_size)/size_incr + 1;
```

9.5.8 Different Systems, Different Errors

Before continuing with our debugging, let's take a look at what happens on another system (a network of workstations running `mpich`). When we run the program on this system with the incorrect value for `time_array_order`, the output is

```
Process 0 > Before Initialize, p = 1, my_rank = 0

Process 0 > Before if, my_rank = 0, source = 0, dest = 0
p0_468:  p4_error: interrupt SIGSEGV: 11
```

That is, the program crashed with a segmentation violation. This suggests that we have either failed to allocate storage correctly, or that a subscript has been calculated incorrectly. So this error also points to a possible problem with `time_array_order`: it suggests that we may not have allocated the correct amount of space for the arrays we're using to store the timings. Indeed, this is what happens: when we try to write values to the timing arrays (which have length 0), a segmentation violation results and the system aborts the program.

You may well ask why the program didn't crash when we ran it on the other systems. This brings up a point we've noted before:

> *Different systems respond in different ways if a program contains a bug. In particular, it's entirely possible that a program that has been "fully debugged" on one system will crash on another. The exact behavior of erroneous programs is impossible to predict.*

9.5.9 Moving to Multiple Processes

Let's move on. After fixing the computation of `time_array_order`, the output of the program run with one process is

```
Process 0 > Before Initialize, p = 1, my_rank = 0

Process 0 > Before if, my_rank = 0, source = 0, dest = 0
Message size (floats):        1000
io_process = 0, time_array_order = 1
Avg circuit time (ms):    1.493990
Max circuit time (ms):    1.518011
Min circuit time (ms):    1.469970
```

Of course, the reported times are different on different systems, but the basic structure of the output is the same.

OK! We've got the program to pass our first test. The next step is to try to get it to run with different size messages. Let's change `min_size` to 0. Now it should send two messages containing 0 floats and two messages containing 1000 floats. (What is actually sent in the messages containing 0 floats?) The output is

```
Process 0 > Before Initialize, p = 1, my_rank = 0

Process 0 > Before if, my_rank = 0, source = 0, dest = 0

Process 0 > Before if, my_rank = 0, source = 0, dest = 0
Message size (floats):           0       1000
io_process = 0, time_array_order = 2
Avg circuit time (ms):    0.533491   1.492471
Max circuit time (ms):    0.533998   1.514971
Min circuit time (ms):    0.532985   1.469970
```

and it seems that we've got a program that works with one process.

So we're finally ready to run the program with more than one process. Before doing so, however, let's remove that `printf` statement from `Print_results` that's messing up our output. With this removed, our output with two processes is

```
Process 0 > Before Initialize, p = 2, my_rank = 0
Process 1 > Before Initialize, p = 2, my_rank = 1

Process 0 > Before if, my_rank = 0, source = -1, dest = 1
Process 1 > Before if, my_rank = 1, source = 0, dest = 0

Process 0 > Before if, my_rank = 0, source = -1, dest = 1
Process 1 > Before if, my_rank = 1, source = 0, dest = 0
Message size (floats):           0       1000
```

```
Avg circuit time (ms):     0.384986    0.599504
Max circuit time (ms):     0.500977    0.606954
Min circuit time (ms):     0.268996    0.592053
```

At first glance, it looks OK. However, closer examination reveals an oddity: process 0 is reporting that the source for its receives is process -1! Of course, there's no process -1, and we've miscalculated the source. Recall that the source is calculated using the formula:

```
source = (my_rank - 1) % p
```

This is fine as long as `my_rank` is greater than 0. However, when `my rank` equals 0, the formula produces a negative value.[1] This can be changed easily enough: guarantee that the dividend is positive by adding in p. That is, a formula that will always produce a nonnegative source is

```
source = (my_rank + p - 1) % p
```

Adding in p doesn't change the congruence class, but it does guarantee that the "representative" of the congruence class is positive.

The real question is, Why didn't the program crash with `source` $= -1$? We accidentally stumbled across a value that matches one of MPI's special possibilities for a source in `MPI_Recv`. That is, if we examined the include files for MPI, we might find

```
#define MPI_ANY_SOURCE -1
```

or perhaps

```
#define MPI_PROC_NULL -1
```

If the first definition is used, then the program will actually do what it's supposed to do: process 0 will receive data from any process that attempts to send to it. Since process 1 is the only process that's trying to send to it, the program does exactly what it's supposed to do. However, if the second definition is used, then `MPI_Recv` simply returns as soon as possible without changing the contents of the receive buffer and the timing results for our program are meaningless. (For a discussion of the meaning and use of `MPI_PROC_NULL`, see section 13.4.)

This brings up an extremely important point:

During testing and debugging, you must make certain that communications are behaving as they should. In some cases, this can be

1 The ANSI standard for C is not explicit about this: it only insists that the absolute value of the remainder be less than the absolute value of the divisor.

determined simply by examining the output of the program. In other cases, however, this may involve printing out the arguments passed to and returned from communication functions.

9.5.10 Confusion about I/O

Since the output of the program doesn't tell us whether the communications functions are behaving as we intended, we should examine the contents of the arguments. As we noted earlier, this can cause problems if we use `Cprintf`: the program could actually hang if we're not careful. If each process can print, then there's no difficulty. But let's look at what happens to the less fortunate. Suppose we want to examine the arguments being passed into `MPI_Send` and the arguments being returned from `MPI_Recv`. Then we might insert `Cprintf`s as indicated.

```
if (my_rank == 0) {
      .
      .
    for (test = 0; test < test_count; test++) {
        start = MPI_Wtime();
/* 1 */ Cprintf(io_comm,"","Before send, x[0] = . . .);
        /* dest = 1 */
        MPI_Send(x, size, MPI_FLOAT, dest, . . .);
        /* source should be p - 1 */
        MPI_Recv(x, size, MPI_FLOAT, source, . . .);
/* 2 */ Cprintf(io_comm,"","After recv, x[0] = . . .);
            .
            .
    } /* for test */
} else { /* my_rank != 0 */
    for (test = 0; test < test_count; test++) {
        MPI_Recv(x, size, MPI_FLOAT, source, . . .);
/* 1 */ Cprintf(io_comm,"","After recv, x[0] = . . .);
/* 2 */ Cprintf(io_comm,"","Before send, x[0] = . . .);
        MPI_Send(x, size, MPI_FLOAT, dest, . . .);
    } /* for test */
}
```

When we run this program with two processes, the output is

```
Process 0 > Before Initialize, p = 2, my_rank = 0
Process 1 > Before Initialize, p = 2, my_rank = 1

Process 0 > Before if, my_rank = 0, source = 1, dest = 1
Process 1 > Before if, my_rank = 1, source = 0, dest = 0

Process 0 > Before send, x[0] = 0.0, size = 0, dest = 1
```

The program hangs! What's happened? Recollect that `Cprintf` prints a message from *every* process in `io_comm`, and the underlying group of `io_comm` is the same as the underlying group of `MPI_COMM_WORLD`. So the `Cprintf`s that are marked with a "1" will be paired, and the `Cprintf`s marked by a "2" will be paired. Thus, since process 0 in `io_comm` is the same as process 0 in `MPI_COMM_WORLD`, it will block in `Cprintf` until it receives data from *every* process in `io_comm`. So it will never get to `MPI_Send`, since it will never receive a message from process 1 inside the call to `Cprintf`. Is this confusing, or what?

We have another form of deadlock: Process 0 is waiting inside `Cprintf` for a message from process 1, but process 1 is waiting in `MPI_Recv` called from `main` for a message from process 0, and process 0 can't get to `MPI_Send` because it's hung in `Cprintf`.

The moral here is that since `Cprintf` is a collective operation, we must be careful about interlacing calls to it with other communication operations, and we should keep in mind:

> *The addition of debugging output can change the behavior of our program.*

One solution in this case is just to make sure that all processes call `Cprintf` before and after each send and each receive. Another is to have each process call `Cprintf` before any sends or receives are started and after all sends and receives (in a single circuit) are completed. We'll opt for the second alternative. Now our code should look something like this:

```
if (my_rank == 0) {
        ⋮
    for (test = 0; test < test_count; test++) {
        start = MPI_Wtime();
/* 1 */ Cprintf(io_comm,"","Before send, x[0] = . . .");
        /* dest = 1 */
        MPI_Send(x, size, MPI_FLOAT, dest, . . .);
        /* source should be p - 1 */
        MPI_Recv(x, size, MPI_FLOAT, source, . . .);
/* 2 */ Cprintf(io_comm,"","After recv, x[0] = . . .");
            ⋮
    } /* for test */
} else { /* my_rank != 0 */
    for (test = 0; test < test_count; test++) {
/* 1 */ Cprintf(io_comm,"","Before recv, x[0] = . . .");
        MPI_Recv(x, size, MPI_FLOAT, source, . . .);
        MPI_Send(x, size, MPI_FLOAT, dest, . . .);
/* 2 */ Cprintf(io_comm,"","After send, x[0] = . . .");
    } /* for test */
}
```

The output is OK. However, as it is fairly long, we'll omit it. Incidentally, it's easy to get flooded by the output of a parallel program. So we should be careful to consider how much output our program will generate before we run it.

If you prefer not to use `Cprintf`, and you can't use `printf`, you may want to consider having each process open its own output file. This can be confusing: it may be difficult to get a comprehensive view of what is happening on all the processes at a given instant, but it does avoid the complexities associated with the use of collective operations. See Chapter 8 for a brief discussion of the mechanics of this.

Finally, it should be noted that in addition to the problems associated with collective output operations, the addition of any output statements may change a race condition, and hence change the details of an error.

9.5.11 Finishing Up

In order to complete the debugging, we still need to test the program with

1. large values of `test_count`
2. a large range of sizes
3. different numbers of processes

In order to do this and not be swamped by output, we can first test the program with different numbers of processes, but use small values for `test_count` and only a few sizes. Then we can finish up by removing most of the output statements and testing the program on large values of `test_count` and many different sizes of messages.

When we run the program with different numbers of processes but with small values of `test_count` and only a few sizes, it seems to be OK. Once again, we'll omit the output of these runs, since it is fairly extensive.

The "production" output of the program running with two processes on an nCUBE 2 and using only three different message sizes was

```
Message size (floats):              0         500        1000
Avg circuit time (ms):       0.355654    2.665530    4.961305
Max circuit time (ms):       0.364006    2.679050    4.966021
Min circuit time (ms):       0.346959    2.651989    4.944980
```

The `test_count` was 1000.

We're done! That wasn't so painful ... was it?

9.6 Error Handling in MPI

Our last topic is an option built into MPI that can be used to help analyze program bugs.

An **error handler** is basically a function to which control is transferred when an error condition is detected by a program. Every implementation of MPI is required to make two error handlers available to the user. The default error handler, MPI_ERRORS_ARE_FATAL, causes programs to abort when errors are detected. The other standard error handler is MPI_ERRORS_RETURN. When an error is detected, this handler returns an error code to the function that called the MPI function. Implementations are free to define additional error handlers. For example, the mpich implementation has defined the error handlers MPE_Errors_call_dbx_in_xterm and MPE_Signals_call_debugger. The first will open a window and start dbx when an error occurs. The second will start a debugger if it catches a signal (e.g., SIGSEGV).

Error handlers are associated with communicators. In order to associate a new error handler with a communicator, one can simply call the function MPI_Errhandler_set:

```
int MPI_Errhandler_set(
        MPI_Comm        comm        /* in */,
        MPI_Errhandler  errhandler  /* in */)
```

This has the effect of associating the error handler errhandler with the communicator comm on the calling process. After calling this function, when an error is detected in an MPI function that is using the communicator comm, the MPI function will use the error handler errhandler.

For example, if we want to examine the error codes generated by MPI functions instead of aborting, we can associate MPI_ERRORS_RETURN with MPI_COMM_WORLD by making the following call on all processes:

```
int error_code;
error_code = MPI_Errhandler_set(MPI_COMM_WORLD,
                MPI_ERRORS_RETURN);
```

Now we can examine errors occurring within MPI functions by testing the error code after calling an MPI function. For example,

```
char error_message[MPI_MAX_ERROR_STRING];
int message_length;

error_code = MPI_Bcast(&x, 1, MPI_INT, 0, comm);
if (error_code != MPI_SUCCESS) {
    MPI_Error_string(error_code, error_message,
        &message_length);
    fprintf(stderr, "Error in call to MPI_Bcast = %s\n",
        error_message);
    fprintf(stderr, "Exiting from function XXX\n");
    MPI_Abort(MPI_COMM_WORLD, -1);
}
```

If, for example, we fail to properly initialize comm, then after the call to MPI_Bcast each process will print something like

```
Error in call to MPI_Bcast = Invalid communicator
Exiting from function XXX
```

and the program will abort.

MPI_SUCCESS and MPI_MAX_ERROR_STRING are predefined constants in MPI. In order to be consistent with C usage, MPI_SUCCESS is always 0. MPI_MAX_ERROR_STRING is defined by the implementation. The function MPI_Error_String returns the error message (and the length of the error message) associated with the value in error_code. Both the code and the message are implementation dependent. The second parameter of MPI_Abort is an error code that is returned to the program that started the MPI program (e.g., the shell). It is a good idea to abort after a call that returns an error code: MPI makes no guarantees that it will recover correctly. So it's up to us to clean things up before we try to proceed.

Of course, this code won't work if you can't print from every process. See Chapter 8 for a partial solution.

Error handlers are inherited by newly created communicators. Thus, if the preceding call to MPI_Errhandler_set were placed immediately after the call to MPI_Init, and there were no subsequent calls to MPI_Errhandler_set, then all of the MPI functions would return error codes. Note that since error handlers are associated with communicators, it's entirely possible to have multiple different error handlers simultaneously active in a program. Also note that since the association of error handlers is local, it's possible to have different handlers associated to the same communicator on different processes.

MPI does provide facilities for users to write their own error handlers. See the references for pointers to discussions of this.

Finally, we note that we can use MPI's profiling interface to have our programs generate traces. We'll discuss the profiling interface in section 12.6.1.

9.7 Summary

The most important observation in this chapter has to do with design and development: a carefully designed and developed program should require minimal debugging. This point cannot be overstressed. If, however, our program does contain errors, we discussed some methods for debugging. Typically, the debugging process for both serial and parallel programs is cyclic: The program is run; when an error is detected, one of several methods is used for identifying the cause of the error. After the error is corrected, the process is repeated until no further errors are found. Not surprisingly, debugging parallel programs tends to be much more complex than debugging serial programs. In addition to the complexity added by communication, parallel programs can exhibit nondeterminism because of race conditions in which processes effectively compete against each other to complete tasks. Nondeterminism may not be a bug: it

is perfectly natural for some programs to be designed so that faster processes complete more work than slower processes.

The cyclic process on serial programs typically progresses from simple data sets to more complex data sets. That is, after bugs have been eliminated from the program when it is run on the simplest possible input, progressively more complex data is used to exercise all aspects of the program. With parallel programs, this cyclic process should be repeated with varying numbers of processes: first the program is fully debugged with just one process; then the program is debugged with two processes; etc.

We discussed the three most commonly used methods for identifying bugs:

1. Examine the source code.
2. Add debugging output to the source code.
3. Use a symbolic debugger.

All three methods are applicable to both serial and parallel programs. Simply examining the source code is most useful when the program is fairly small and the errors suggest the nature of the problem.

The use of debugging output is the mainstay of both serial and parallel debugging: it is typically used to provide snapshots of the state of the program at various points during execution. When we're debugging parallel programs, the introduction of output statements can change the behavior of the program. Collective output operations (such as `Cprintf`) can cause deadlock if they are interlaced with other communication operations. Any output statements may change race conditions, and hence the apparent nature of bugs. If all processes don't have access to `stdout` or `stderr`, then you may want to consider writing the output of each process to a separate file rather than using a collective output function.

Debuggers can be used to start a program, stop execution of a program at various points, examine the state of a program during execution, and make changes to a program. The development of parallel debuggers remains an active area of research, and, as a consequence, there is little standardization. Currently, the only universally available debuggers are just serial debuggers attached to the various processes of a parallel execution. Starting a debugger for a parallel program is highly system dependent. It may be necessary to start the program and then attach a debugger to the processes started by the program. If the debugger is just a serial debugger, it can be very difficult to monitor and control the execution of a program running multiple processes.

Much of the chapter was devoted to warnings of pitfalls, examples of common bugs, and sage advice:

- Be aware that the manifestation of a bug can change from system to system. Indeed, an apparently bug-free program may crash ignominiously when it is run on a new system.
- The classical bugs in message-passing programs are

—A receive that has no matching send will usually cause the receiving process to hang. If all processes are trying to receive and there is no send, the program will deadlock. A deadlock occurs when each process is waiting on another process.

—A receive or a send with an incorrect argument will usually cause a process to hang or crash.

- Don't be misled into thinking that all your bugs are the result of the fact that the program is parallel. Many, if not most, parallel program bugs are really just serial program bugs.
- Be very careful to check the arguments being passed in and out of communication functions: it can be very dangerous to make assumptions about what is sent or received in a communication.
- It's easy when debugging parallel programs to get flooded by output. Be careful to take into consideration that all your output will typically be multiplied by the number of processes.

We didn't learn a lot of new MPI functions. All the new MPI functions that we studied had to do with MPI's error handling facilities. An error handler is basically a function that is called when an error condition arises. In MPI, error handlers are local to a process and associated with communicators. All implementations of MPI have two error handlers: `MPI_ERRORS_ARE_FATAL`, the default error handler, and `MPI_ERRORS_RETURN`, which can be used to get MPI functions to return error codes if they detect errors. In order to associate an error handler with a communicator, we can use the local function

```
int MPI_Errhandler_set(
        MPI_Comm        comm        /* in */,
        MPI_Errhandler  errhandler  /* in */)
```

In order to find out what an error code means, we can obtain a string by calling

```
int MPI_Error_string(
        int    error_code      /* in  */,
        char*  error_message   /* out */,
        int*   message_length  /* out */)
```

Except for the predefined constant `MPI_SUCCESS`, which is always 0, error codes are implementation dependent. The amount of storage needed for an error string is also implementation dependent. However, it will always be less than `MPI_MAX_ERROR_STRING`. When we use an error handler that does not automatically abort a program, we must be very careful about continuing execution. It is usually a good idea to just print a message and then abort the program whenever MPI detects an error in one of its functions. Recollect that we can abort a program with the function `MPI_Abort`.

9.8 References

The quotation from the `gdb` man page can be found in [36]. For an excellent source of information on debugging and testing serial programs, see [35].

See [21], [29], and [34] for discussions of user-defined error handlers.

Both `mpich` and `LAM` provide more powerful debugging facilities than we've outlined here. Consult the `mpich` user's guide [20] or the `LAM` MPI primer [30] for further information.

9.9 Exercises

1. Fully debug the serial insertion sort program.
2. Modify `comm_time.c` so that it uses a communicator with a one-dimensional ring topology for its message passing.
3. Although we didn't worry about it in our example, the calls to `MPI_Wtime` may add a significant overhead to the elapsed time. Modify `comm_time.c` so that this overhead is subtracted from elapsed time. How does this affect the results of your timings?

9.10 Programming Assignments

1. Write a function that will print out a matrix that uses a block-checkerboard distribution. Assume that the process grid is square and that $\sqrt{p}$ evenly divides n. The output should present a unified picture of the matrix; it shouldn't simply list the blocks from each process.
2. Write a matrix-vector multiplication program that uses a block-checkerboard distribution of the matrix and a block distribution of the vectors along the *processes* on the diagonal. Assume that the processes form a square virtual grid. Use the output function you wrote in programming assignment 1.
3. Take a look at the programming assignments in the previous chapters. Now's a good time to go back and try to write the ones that looked too formidable before.