CHAPTER 8

Dealing with I/O

ON SEVERAL OCCASIONS, we've run across the problem of I/O on parallel systems. For example, in Chapter 4 we briefly discussed some of the problems involved with using functions as fundamental as `printf` and `scanf`. Recall that if a parallel program executes a statement such as

```
printf("x = %d\n", x);
```

it isn't immediately clear what should appear on the user's terminal screen. Should there be one value printed from each process? Should only one value appear? If multiple values are printed, how should they be ordered? How will we know which value comes from which process? Similar problems occur with the use of `scanf`. Of course, the problem becomes even more complex when we want to read or print the elements of a composite type.

The problem is that C's I/O functions were not written with parallel systems in mind, and, unfortunately, there isn't a clear consensus about how to carry out I/O on a parallel system. In view of this lack of consensus, the MPI Forum avoided the issue of I/O: the MPI Standard imposes no requirements on the I/O capabilities of an MPI implementation. In spite of this, all implementations of MPI provide *some* support for I/O, and in this chapter we'll take a look at how we can use MPI to leverage very limited support for I/O to the point where we can develop general and powerful programs.

Our main purpose in writing these functions is to provide tools for program development. We are not attempting to develop a high-performance parallel I/O library. As a consequence, we will favor simplicity and reliability over performance.

In the course of developing the I/O functions, we'll learn a lot about MPI and parallel programming. In particular, we'll learn more about attribute

caching, derived datatypes, and data distributions. So even if your system provides very good I/O facilities, you will probably find a lot of useful information in this chapter.

8.1 Dealing with `stdin`, `stdout`, and `stderr`

Since the MPI standard imposes no requirements on the I/O capabilities of an MPI implementation, users have no way of knowing, a priori, what the I/O capabilities of the processes will be. For example, there are many systems that allow each process full access to `stdin`, `stdout`, and `stderr`, while other systems only allow process 0 in `MPI_COMM_WORLD` access to `stdout` and `stderr`. Of course, this creates serious difficulties for the user wishing to write portable programs, since there are few useful programs that do no I/O. This is especially problematic during the coding and debugging phases of program development—it can be very difficult to determine whether a program is functioning correctly, and, if it isn't, what the precise nature of the problem is.

In order to address this problem, we'll develop some functions that will allow programs access to `stdin`, `stdout`, and `stderr` on most systems running MPI. We'll write routines that assume that at least one process has access to `stdin`, `stdout`, and `stderr`. All the implementations with which we are acquainted allow at least one process access to `stdout` and `stderr`. If your implementation doesn't allow access to `stdin`, we'll discuss a few alternatives that may be available to you.

Since our main purpose in writing these functions is to provide ourselves with program development tools, we'll want to use these functions to monitor the state of our programs. As a consequence, we'll want to see output from all the processes, and our I/O functions will turn I/O operations into collective operations with one process designated as the "I/O process." For output, the I/O process will collect data from all the other processes and print it out, and for input, it will read in data and broadcast it to the other processes. Since these are collective operations, each I/O function will take a communicator argument.

Note that making our output functions collective will allow us to organize our output by process rank. Even MPI implementations that allow full access to all of C's I/O functions usually don't organize output very well. For example, if each process executes, more or less simultaneously, the statement

```
printf("Process %d > x = %d\n", my_rank, x);
```

we're liable to get output that looks like this:

```
Process 3 > x = 3
Process 1 > x = 1
Process 0 > x = 2
Process 2 > x = 4
```

and most of us would prefer

```
Process 0 > x = 2
Process 1 > x = 1
Process 2 > x = 4
Process 3 > x = 3
```

So you may want to use our output functions instead of simple `printf`s.

Since each communicator used by the I/O functions should have a designated I/O process, and the rank of the I/O process will depend on the communicator, it will be convenient for us to use MPI's attribute caching facility to store the rank of the I/O process *with the communicator.* So let's begin by discussing attribute caching in MPI.

8.1.1 Attribute Caching

Recall from Chapter 7 that although a basic communicator consists of a group and a context, additional information can be associated with any communicator by a process called *attribute caching.* In that chapter, we were interested in caching topologies with communicators. Since topologies are so important and so generally useful, the functions that are used to create, access, and modify them are specific to topologies. However, MPI provides a completely general interface to attribute caching, so that we can cache attributes of our own with communicators. In particular, we can cache the rank of a process that can carry out I/O.

The content of an attribute in MPI can be a simple scalar or a complex composite datatype. We want our attribute to be a process rank. However, it's not difficult to imagine useful attributes that are much more complex. For example, suppose we want to write our own broadcast function. Recollect (from Chapter 5) that an efficient broadcast will probably use a tree-structured communication pattern, and that the exact characteristics of the optimal tree structure will depend on the underlying parallel system. Thus, we might store with each communicator a composite attribute that provides a readily accessible description of the tree on the system we're using. The content of the attribute might be a struct or array indicating the pairing of processes during the stages of the broadcast, or it might be a function that computes this information.

In order to accommodate this diversity of possible attribute values, an attribute in MPI is a pointer of type `void*`, and we distinguish between an *attribute*, which is a pointer, and the *attribute content*, which is the data referenced by the attribute.

Since any communicator can have multiple different cached attributes, each attribute is identified to the system by a key, which is just a system-defined int. Thus, when we create a new attribute, we call the function `MPI_Keyval_create`, which will generate a new attribute key. Of course, once we've created an attribute and its identifying key, we still need to be able to

access it. For the most part, attributes are accessed with the two functions MPI_Attr_put and MPI_Attr_get. The first assigns a value to an attribute. The second returns a pointer to an attribute.[1]

Here's a short example that makes a copy of MPI_COMM_WORLD and creates an attribute key we can use to identify our I/O process rank attribute. It then caches process rank 0 with the I/O process rank attribute.

```
MPI_Comm  io_comm;      /* Communicator for I/O      */
int       IO_KEY;       /* I/O process attribute key */
int*      io_rank_ptr;  /* Attributes are pointers   */
void*     extra_arg;    /* Unused                    */

/* Get a separate communicator for I/O functions by */
/*     duplicating MPI_COMM_WORLD                   */
MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);

/* Create the attribute key */
MPI_Keyval_create(MPI_DUP_FN, MPI_NULL_DELETE_FN,
    &IO_KEY, extra_arg);

/* Allocate storage for the attribute content */
io_rank_ptr = (int*) malloc(sizeof(int));

/* Set the attribute content */
*io_rank_ptr = 0;

/* Cache the attribute with io_comm */
MPI_Attr_put(io_comm, IO_KEY, io_rank_ptr);
```

The call to MPI_Comm_dup creates a new communicator with the same underlying group as MPI_COMM_WORLD, but a different context. We do this so that the communication in the I/O functions can't be mixed up with the communication in the rest of the program.

Any time we want to print data, we can retrieve the rank we cached with io_comm as follows:

```
int    flag;
int*   io_rank_attr;

/* Retrieve the I/O process rank  */
MPI_Attr_get(io_comm, IO_KEY, &io_rank_attr, &flag);
```

---

1 The syntax of MPI_Attr_get was changed in version 1.1 of the MPI Standard [29]. The attribute parameter formerly had type void**. It now has type void*. This doesn't change its functionality; it simply makes it possible to call the function without casting the corresponding argument.

```
/* If flag == 0, something went wrong: */
/*     there's no attribute cached.    */
if ((flag != 0) && (my_rank == *io_rank_attr))
    printf("Greetings from the I/O Process!\n");
```

Attributes and attribute keys in MPI are process local, as are the functions MPI_Keyval_create, MPI_Attr_put, and MPI_Attr_get. In particular, it's entirely possible that different processes can cache different attributes with the same communicator. Thus, if we wish to create an attribute that has the same content across all the processes in a communicator (as with our I/O process rank), it's up to us to insure that this is the case.

This brief discussion should provide us with enough information about attribute caching so that we can understand how the I/O functions work. So on a first reading, you might want to skip the following subsection.

8.1.2 Callback Functions

Of course, you're wondering about those arguments we used in our call to MPI_Keyval_create: MPI_DUP_FN, MPI_NULL_DELETE_FN, and extra_arg. The syntax of MPI_Keyval_create is

```
int MPI_Keyval_create(
        MPI_Copy_function*    copy_fn    /* in  */,
        MPI_Delete_function*  delete_fn  /* in  */,
        int*                  key_ptr    /* out */,
        void*                 extra_arg  /* in  */)
```

As we already know, the third parameter returns a pointer to a key that we can use to identify a new attribute. The remaining three parameters all have to do with the management of attributes when we create or free communicators.

If we create a communicator with MPI_Comm_dup:

```
MPI_Comm  old_comm;
MPI_Comm  new_comm;

MPI_Comm_dup(old_comm, &new_comm);
```

each attribute cached with old_comm is "copied" to new_comm. For a given attribute key, the function that does the actual copying is specified by the copy_fn parameter of MPI_Keyval_create. In our example, we just used the predefined MPI function, MPI_DUP_FN. It simply copies the attribute (a pointer) in old_comm to the corresponding attribute in new_comm. In general, however, we may want to do something else. So MPI allows for us to define our own functions. For example, we might want to duplicate the memory referenced by the attribute old_comm and make the corresponding attribute in new_comm reference this new block of memory. This, of course, can't be

accomplished by just copying a pointer, and we need to define a function to do this.

Similarly, the second parameter to MPI_Keyval_create, delete_fn, is used by the system when we free a communicator or delete an attribute. We can free a communicator with the collective function MPI_Comm_free, and we can delete an attribute with the function MPI_Attr_delete. So, for example, if we call

```
MPI_Comm_free(&comm);
```

each attribute cached with comm will be "deleted." For each attribute, the actual deletion will be carried out by the delete_fn function specified in the call to MPI_Keyval_create. In our example, we just used the predefined function MPI_NULL_DELETE_FN, which does nothing. However, it's not difficult to imagine cases where we would want to do something more complex. If an attribute references a large block of dynamically allocated memory, we will probably want the delete function to systematically free the block.

In MPI, these attribute copy and delete functions are called **callback** functions. Their type definitions are

```
typedef int MPI_Copy_function(
                MPI_Comm  old_comm       /* in  */,
                int       keyval         /* in  */,
                void*     extra_arg      /* in  */,
                void*     attribute_in   /* in  */,
                void*     attribute_out  /* out */,
                int*      flag           /* out */)

typedef int MPI_Delete_function(
                MPI_Comm  comm          /* in  */,
                int       keyval        /* in  */,
                void*     attribute     /* in  */,
                void*     extra_arg     /* in  */)
```

Our example of a copy_fn that copies the block of memory referenced by an attribute can be used to explain the purpose of the mysterious extra_arg parameter in both MPI_Keyval_create and the type definitions of the copy and delete functions: suppose that the content of the attribute is a struct with one or more dynamically allocated array members. Then, unless the sizes of the arrays are stored in the struct itself, the copy_fn won't have any way of knowing how much memory to allocate for them. The extra_arg can be used for this by passing in the sizes of the arrays when the attribute key is created.

8.1.3 Identifying the I/O Process Rank

We still don't know how to identify a process that can carry out I/O. In our example, we just used process 0 in MPI_COMM_WORLD. While this is probably

a safe bet, there may be an alternative. Each implementation of MPI is supposed to provide several predefined attributes cached with MPI_COMM_WORLD. Among these is the attribute with key MPI_IO. The content of this attribute is the rank of a process that can carry out language standard I/O (e.g., fopen, fprintf). However, there are numerous caveats:

1. If no process can carry out I/O, the attribute content will be the predefined constant MPI_PROC_NULL.
2. If every process can carry out I/O, the attribute content will be the predefined constant MPI_ANY_SOURCE.
3. If some processes can carry out I/O, but others cannot, different processes may have different attribute content. Processes that can carry out I/O are required to have attribute content equal to their rank. Processes that cannot should have attribute content equal to the rank of some process that can.
4. The MPI_IO attribute does not indicate which processes can provide input. So even if the content of the MPI_IO attribute is MPI_ANY_SOURCE, it may be the case that the implementation doesn't provide any access to stdin.

Thus, we cannot rely on the MPI_IO attribute to provide a unique process rank, nor can we be sure that it will provide information about input. However, we can use it to try to define a unique I/O process rank as follows:

```
int* mpi_io_ptr;
int  io_rank;
int  flag;

MPI_Attr_get(MPI_COMM_WORLD, MPI_IO, &mpi_io_ptr,
    &flag);

if (flag == 0) {
    /* Attribute not cached.  Not MPI compliant */
    io_rank = MPI_PROC_NULL;
} else if (*mpi_io_ptr == MPI_PROC_NULL) {
    io_rank = MPI_PROC_NULL;
} else if (*mpi_io_ptr == MPI_ANY_SOURCE) {
    /* Any process can carry out I/O.  */
    /* Use process 0 */
    io_rank = 0;
} else {
    /* Different ranks may have been returned */
    /* on different processes.  Get min       */
    MPI_Allreduce(mpi_io_ptr, &io_rank, 1, MPI_INT,
        MPI_MIN, MPI_COMM_WORLD);
}
```

The code attempts to retrieve the attribute corresponding to MPI_IO and copy its contents into io_rank. If the flag argument is 0, the MPI_IO attribute

hasn't been cached. Otherwise, we consider the first three possibilities enumerated above. If the attribute content is neither MPI_PROC_NULL nor MPI_ANY_SOURCE, we must perform some global operation in order to insure a unique rank across all the processes: we chose to take the minimum rank.

8.1.4 Caching an I/O Process Rank

Armed with this information on attributes, we can build a communicator for use in our I/O functions and attempt to cache the rank of a process with it. We should build a separate communicator so that the communication we do in the I/O functions can't be confused with other communication. In order to build it, we can use any of the communicator construction functions; e.g., MPI_Comm_split or MPI_Comm_dup. Then we can call the first function in our "I/O library," Cache_io_rank, to try to cache an I/O process rank with the new communicator.

In an external definition, we define the key that we'll use to identify the I/O process rank attribute: IO_KEY. It is initialized with MPI_KEYVAL_INVALID, a predefined MPI identifier. We also define a constant, NO_IO_ATTR, which we'll use as a return value for the functions in the I/O library if they were unable to find an I/O process rank attribute. In the actual source code this constant will go into a header file.

The function takes two parameters: the original communicator from which the I/O communicator was created, and the I/O communicator, io_comm. It begins by checking to see whether IO_KEY has been initialized by MPI; i.e., whether it no longer has the value MPI_KEYVAL_INVALID. If it hasn't been initialized by MPI, the function calls MPI_Keyval_create, as discussed above.

Once we're sure IO_KEY has been initialized, we first check to see whether either orig_comm or io_comm already has an I/O process rank attribute cached. If so, we use this rank. If not, we see if an MPI_IO attribute has been cached with either communicator, and use the method outlined above to try determine an I/O process rank from the value of the MPI_IO attribute. If all the attempts to find a valid process rank fail, we cache MPI_PROC_NULL as the content of the I/O process rank attribute and return NO_IO_ATTR.

The work of actually retrieving attributes and caching values is done by the function Copy_attr.

```
/* Key identifying I/O process rank attribute */
int     IO_KEY = MPI_KEYVAL_INVALID;

/* Unused */
void*  extra_arg;

#define NO_IO_ATTR -1
```

```
int Cache_io_rank(
        MPI_Comm    orig_comm       /* in     */,
        MPI_Comm    io_comm         /* in/out */) {

    int retval;   /* 0 or NO_IO_ATTR */

    /* Check whether IO_KEY is defined.  If not, define */
    if (IO_KEY == MPI_KEYVAL_INVALID) {
        MPI_Keyval_create(MPI_DUP_FN,
            MPI_NULL_DELETE_FN, &IO_KEY, extra_arg);
    } else if ((retval = Copy_attr(io_comm, io_comm,
                IO_KEY)) != NO_IO_ATTR) {
        /* Value cached */
        return retval;
    } else if ((retval = Copy_attr(orig_comm, io_comm,
                IO_KEY)) != NO_IO_ATTR) {
        /* Value cached */
        return retval;
    }

    /* Now see if we can find a value cached for MPI_IO */
    if ((retval = Copy_attr(orig_comm, io_comm,
                MPI_IO)) != NO_IO_ATTR) {
        /* Value cached */
        return retval;
    } else if ((retval = Copy_attr(io_comm, io_comm,
                MPI_IO)) != NO_IO_ATTR) {
        /* Value cached */
        return retval;
    }

    /* Couldn't find process that could carry out I/O */
    /* Copy_attr has cached MPI_PROC_NULL             */
    return NO_IO_ATTR;

}  /* Cache_io_rank */
```

The function Copy_attr attempts to retrieve the contents of the attribute identified by the parameter KEY from the communicator comm1 and cache it with the IO_KEY attribute on comm2. KEY can be either MPI_IO or IO_KEY. If it finds a valid process rank to cache, it returns 0. Otherwise, it caches MPI_PROC_NULL and returns NO_IO_ATTR.

Copy_attr uses a new MPI function, MPI_Comm_compare. This can be used to determine whether two communicators are identical (same group and context), whether they are congruent (same group, different contexts), or whether they are similar (underlying process sets are the same, but ranks are different). In the first case it returns MPI_IDENT, in the second it returns

MPI_CONGRUENT, and in the third it returns MPI_SIMILAR. If the communicators are neither identical, similar, nor congruent, it returns MPI_UNEQUAL.

Copy_attr also calls a function in our I/O library, Get_corresp_rank. If comm1 and comm2 are distinct communicators, Get_corresp_rank attempts to determine whether a process in comm1 also belongs to comm2. If it does, it returns the rank in comm2. If it doesn't, it returns MPI_UNDEFINED, a predefined MPI constant. Get_corresp_rank also uses a new MPI function, MPI_Group_translate_ranks, which takes a list of process ranks in one group and attempts to determine the corresponding ranks in a second group. If a process in the first group doesn't belong to the second, the corresponding rank is MPI_UNDEFINED.

```
/* All process ranks are < HUGE */
#define HUGE 32768

int Copy_attr(
        MPI_Comm   comm1    /* in     */,
        MPI_Comm   comm2    /* in/out */,
        int        KEY      /* in     */) {

    int   io_rank;
    int   temp_rank;
    int*  io_rank_ptr;
    int   equal_comm;
    int   flag;

    MPI_Attr_get(comm1, KEY, &io_rank_ptr, &flag);

    if (flag == 0) {
        /* Attribute not cached with comm1 */
        io_rank_ptr = (int*) malloc(sizeof(int));
        *io_rank_ptr = MPI_PROC_NULL;
        MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);
        return NO_IO_ATTR;
    } else if (*io_rank_ptr == MPI_PROC_NULL) {
        MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);
        return NO_IO_ATTR;
    } else if (*io_rank_ptr == MPI_ANY_SOURCE) {
        /* Any process can carry out I/O.  Use */
        /* process 0                          */
        io_rank_ptr = (int*) malloc(sizeof(int));
        *io_rank_ptr = 0;
        MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);
        return 0;
    }
```

```
        /* Value in *io_rank_ptr is a valid process  */
        /* rank in comm1.  Action depends on whether */
        /* comm1 == comm2.                           */
        MPI_Comm_compare(comm1, comm2, &equal_comm);

        if (equal_comm == MPI_IDENT) {
            /* comm1 == comm2.  Valid value already */
            /* cached.  Do nothing.                 */
            return 0;
        } else {
            /* Check whether rank returned is valid */
            /* process rank in comm2                */
            Get_corresp_rank(comm1, *io_rank_ptr,
                comm2, &temp_rank);

            /* Different ranks may have been returned */
            /* on different processes.  Get min       */
            if (temp_rank == MPI_UNDEFINED) temp_rank = HUGE;
            MPI_Allreduce(&temp_rank, &io_rank, 1, MPI_INT,
                MPI_MIN, comm2);

            io_rank_ptr = (int*) malloc(sizeof(int));
            if (io_rank < HUGE) {
                *io_rank_ptr = io_rank;
                MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);
                return 0;
            } else {
                /* No process got a valid rank in comm2 */
                /* from Get_corresp_rank                */
                *io_rank_ptr = MPI_PROC_NULL;
                MPI_Attr_put(comm2, IO_KEY, io_rank_ptr);
                return NO_IO_ATTR;
            }
        }
    }  /* Copy_attr  */


    /*****************************************************/
    void Get_corresp_rank(
            MPI_Comm   comm1       /* in  */,
            int        rank1       /* in  */,
            MPI_Comm   comm2       /* in  */,
            int*       rank2_ptr   /* out */) {

        MPI_Group  group1;
        MPI_Group  group2;

        MPI_Comm_group(comm1, &group1);
        MPI_Comm_group(comm2, &group2);
```

```
    MPI_Group_translate_ranks(group1, 1,
            &rank1, group2, rank2_ptr);

}  /* Get_corresp_rank */
```

Note that `Copy_attr` is a collective function since it may call `MPI_Allreduce`, and since `Copy_attr` is collective, `Cache_io_rank` is also collective.

8.1.5 Retrieving the I/O Process Rank

Before writing the actual I/O functions, we need one more attribute management function: a function that will retrieve the rank of the I/O process. If we were sure that a valid rank had been cached, we could simply call `MPI_Attr_get`:

```
int* io_rank_ptr;
int  flag;

MPI_Attr_get(io_comm, IO_KEY, &io_rank_ptr, &flag);
```

However, if our program has mistakenly failed to cache an attribute or if the attribute content is `MPI_PROC_NULL`, we won't be able to carry out any I/O. So let's try to add a little security by having the attribute retrieval function do a little checking. First it will check to see if `IO_KEY` has been initialized by MPI. If it isn't, it will try to initialize it. If it is, it will try to retrieve the I/O process rank attribute from `io_comm`. If this fails, either because the attribute hasn't been cached or because it's `MPI_PROC_NULL`, the function will try to get an I/O process rank from `MPI_COMM_WORLD` by calling `Copy_attr`. The return values are 0 if a valid rank has been found, and `NO_IO_ATTR` otherwise. Because it may call `Copy_attr`, `Get_io_rank` is also a collective function.

```
int Get_io_rank(
        MPI_Comm io_comm      /* in  */,
        int*     io_rank_ptr  /* out */) {

        void*    extra_arg;
        int*     temp_ptr;
        int      flag;

    if (IO_KEY == MPI_KEYVAL_INVALID) {
        MPI_Keyval_create(MPI_DUP_FN,
            MPI_NULL_DELETE_FN, &IO_KEY, extra_arg);
    } else {
        MPI_Attr_get(io_comm, IO_KEY, &temp_ptr, &flag);
        if ((flag != 0) && (*temp_ptr != MPI_PROC_NULL)) {
            *io_rank_ptr = *temp_ptr;
```

```
            return 0;
        }
    }
    if (Copy_attr(MPI_COMM_WORLD, io_comm, MPI_IO)
            == NO_IO_ATTR) {
        return NO_IO_ATTR;
    } else {
        MPI_Attr_get(io_comm, IO_KEY, &temp_ptr, &flag);
        *io_rank_ptr = *temp_ptr;
        return 0;
    }

}  /* Get_io_rank */
```

8.1.6 Reading from stdin

As you've probably already seen, any time we try to design an I/O routine for parallel processors, we need to make some decisions about where the data is going to or coming from. For example, in an input routine, do we want every process to receive all the input data? Or would we prefer to distribute different parts of the data to different processes? For our input routine, we're going to assume that all the data is needed on all the processes, or else that the amount of data is so small that it will not be terribly expensive to send it to all the processes even if they don't need it. The I/O process will just read in a line of data as a string and broadcast it to all the processes. Each process will then make use of the `stdarg` macros to parse the input line. Some implementations of C provide a function, `vsscanf`, that can read data from a string into a variable length argument list. However, this is not standard, and it may be necessary to write your own. In our code, we assume the existence of this function.

Since all of our I/O functions are collective, we'll use the familiar C names `printf` and `scanf` with a "C" prefix. The `Cscanf` function will be similar to the `scanf` function: it will take a format string and a list of variables in its parameter list. However, it will also take a communicator (`io_comm`) and a string that can be used to prompt the user for input. Like the other I/O functions, its return values will be 0 and `NO_IO_ATTR`.

```
/* Used by all the I/O functions.  BUFSIZ is defined  */
/*     in stdio.h                                     */
char io_buf[BUFSIZ];

int Cscanf(
        MPI_Comm  io_comm  /* in  */,
        char*     prompt   /* in  */,
        char*     format   /* in  */,
                  ...      /* out */) {
```

```
    va_list  args;          /* Must include stdarg.h */
    int      my_io_rank;    /* My_rank in io_comm    */
    int      root;          /* The I/O process       */

    /* Try to get the rank of a process that can carry */
    /*     out I/O                                     */
    if (Get_io_rank(io_comm, &root) == NO_IO_ATTR)
        return NO_IO_ATTR;
    MPI_Comm_rank(io_comm, &my_io_rank);

    /* Read in data on root */
    if (my_io_rank == root) {
        printf("%s\n", prompt);
        gets(io_buf);
    }  /* my_io_rank == root */

    /* Broadcast the input data */
    MPI_Bcast(io_buf, BUFSIZ, MPI_CHAR, root, io_comm);

    /* Copy the input data into the parameters */
    va_start(args, format);
    vsscanf(io_buf, format, args);
    va_end(args);

    return 0;
} /* Cscanf */
```

8.1.7 Writing to `stdout`

Writing to `stdout` reverses the steps of reading from `stdin`: we copy the output data into a string and gather each process's string onto the I/O process, which prints them. Once again we use the `stdarg` package, since we don't know what data will be printed. The parameter list includes the same format string and variable list we use with `printf`. However, it also contains a communicator and a title. The analog of `vsscanf`, `vsprintf`, is a standard C function. So we don't need to worry about writing our own function to copy the data into the string.

In order to gather the data onto the I/O process, we can either use `MPI_Gather` or a loop of receives. If we use `MPI_Gather`, we'll have to allocate a very long array on the I/O process in which to collect the data. As a consequence, we've opted to use the simple loop of receives.

As usual, the return value is either 0 or `NO_IO_ATTR`.

```
int Cprintf(
        MPI_Comm  io_comm  /* in */,
        char*     title    /* in */,
        char*     format   /* in */,
                  ...      /* in */) {
```

```
int         q;
int         my_io_rank;
int         io_p;
int         root;
MPI_Status  status;
va_list     args;

if (Get_io_rank(io_comm, &root) == NO_IO_ATTR)
    return NO_IO_ATTR;
MPI_Comm_rank(io_comm, &my_io_rank);
MPI_Comm_size(io_comm, &io_p);

/* Send output data to root */
if (my_io_rank != root) {
    /* Copy the output data into io_buf */
    va_start(args, format);
    vsprintf(io_buf, format, args);
    va_end(args);

    MPI_Send(io_buf, strlen(io_buf) + 1, MPI_CHAR,
        root, 0, io_comm);
} else { /* my_io_rank == root */
    printf("%s\n",title);
    fflush(stdout);
    for (q = 0; q < root; q++) {
        MPI_Recv(io_buf, BUFSIZ, MPI_CHAR, q,
            0, io_comm, &status);
        printf("Process %d > %s\n",q, io_buf);
        fflush(stdout);
    }

    /* Copy the output data into io buf */
    va_start(args, format);
    vsprintf(io_buf, format, args);
    va_end(args);
    printf("Process %d > %s\n",root, io_buf);
    fflush(stdout);

    for (q = root+1; q < io_p; q++) {
        MPI_Recv(io_buf, BUFSIZ, MPI_CHAR, q,
            0, io_comm, &status);
        printf("Process %d > %s\n",q, io_buf);
        fflush(stdout);
    }
    printf("\n");
    fflush(stdout);
}
```

```
    return 0;
} /* Cprintf */
```

8.1.8 Writing to `stderr` and Error Checking

We want to write to `stderr` and end execution if we detect an error in our program. For example, if we attempt to allocate a large array and the call to `malloc` fails, we would like to inform the user and shut the program down. It would seem that a "natural" way to do this would be for the process on which the `malloc` failed to print a message and shut down the other processes. However, given the restrictions on which processes can carry out I/O, unless the `malloc` fails on the I/O process, we won't be able to print a message. Alternatively, we might like to interrupt the I/O process and have it print a message, but the current version of MPI provides no mechanism for us to do this.

Thus, it seems that we have two options: we can dedicate the I/O process to I/O exclusively (i.e., it spends all its time waiting for and printing messages from other processes) or we can make error checking a collective operation. We're probably reluctant to just give up a process so that we can print error messages. So let's take a look at setting up a collective operation.

The idea is that we're running an SPMD-style program, and every process is doing more or less the same thing. For example, in order to check whether a `malloc` failed, we assume that every process called `malloc` at about the same point in the program. We can then carry out a collective operation to check whether any process encountered an error. So if each process is executing a completely different sequence of statements, this approach won't work.

In this setting we can check for an error by performing an allgather on the error code generated on each process. Then each process can check the list of error codes; the I/O process can print a list of the processes that failed, and, if this list is nonempty, we can stop the program. We'll use negative error codes to indicate that the program should be stopped.

MPI provides a function, `MPI_Abort`, that can be called to terminate execution. Its syntax is

```
int MPI_Abort(
        MPI_Comm  comm        /* in */,
        int       error_code  /* in */)
```

It tries to shut down all the processes in `comm`. However, its behavior is implementation dependent, and an implementation is only required to try to shut down all the processes in `MPI_COMM_WORLD`. In a UNIX environment, the error code is returned as if the main program returned with

```
return error_code;
```

As usual the return value of `Cerror_test` is either 0 or `NO_IO_ATTR`.

```
char* error_buf;
int   error_bufsiz = 0;

int Cerror_test(
        MPI_Comm  io_comm       /* in */,
        char*     routine_name  /* in */,
        int       error         /* in */) {

    int q;
    int io_p;
    int error_count = 0;
    int io_process;
    int my_io_rank;

    if (Get_io_rank(io_comm, &io_process) == NO_IO_ATTR)
        return NO_IO_ATTR;
    MPI_Comm_size(io_comm, &io_p);
    MPI_Comm_rank(io_comm, &my_io_rank);

    /*  If necessary increase the size of error_buf */
    if (error_bufsiz == 0) {
        error_buf = (int*) malloc(io_p*sizeof(int));
        error_bufsiz = io_p;
    } else if (error_bufsiz < io_p) {
        realloc(error_buf, io_p);
        error_bufsiz = io_p;
    }

    MPI_Allgather(&error, 1, MPI_INT, error_buf, 1,
        MPI_INT, io_comm);
    for (q = 0; q < io_p; q++) {
        if (error_buf[q] < 0) {
            error_count++;
            if (my_io_rank == io_process) {
                fprintf(stderr,"Error in %s on process %d\n",
                    routine_name, q);
                fflush(stderr);
            }
        }
    }
    if (error_count > 0)
        MPI_Abort(MPI_COMM_WORLD, -1);
    return 0;
} /* Cerror_test */
```

We can use this function to check for errors in both MPI and non-MPI functions. However, in order to use it with MPI functions, we need to change MPI's default behavior when it encounters an error. See section 9.6 for a discussion of how to do this.

8.2 Limited Access to stdin

As we mentioned earlier, there are MPI implementations that provide no access to stdin, and the MPI_IO attribute says nothing about whether processes can read input. Thus, before you can safely use any input functions, you need to test your implementation of MPI. Fortunately, this should be pretty simple. If you run the following program on one process, you should be able to determine whether your implementation allows any process access to stdin.

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

main(int argc, char* argv[]) {
    int x;

    MPI_Init(&argc, &argv);

    printf("Enter an integer\n");
    fflush(stdout);
    scanf("%d", &x);
    printf("We read x = %d\n", x);
    fflush(stdout);

    MPI_Finalize();
}
```

If the program hangs or crashes after "Enter an integer," you can be fairly sure that your implementation won't let you read from stdin. Don't despair. We have several options: we may be able to use command line arguments, self-initialization, or files other than stdin. We'll discuss command line arguments and self-initialization in this section. We'll discuss file I/O in the next section.

Many implementations of MPI allow each process full access to any command line arguments the user may have typed. This can be quite useful if your program doesn't take much input. In order to determine what the capabilities of your implementation are, try running the following program with two processes.

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

/* Header file for our I/O library */
#include "cio.h"

main(int argc, char* argv[]) {
    MPI_Comm  io_comm;
    int       i;

    MPI_Init(&argc, &argv);
```

```
    /* Set up communicator for I/O */
    MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);
    Cache_io_rank(MPI_COMM_WORLD, io_comm);

    for (i = 0; i < argc; i++)
        Cprintf(io_comm,"","argv[%d] = %s", i, argv[i]);

    MPI_Finalize();
}
```

If we run this program with some command line arguments, it should print the arguments available to each process. For example, if we run the program with two processes on a network of workstations using the mpich implementation of MPI, we get

```
% mpirun -np 2 argtest hello, world
Process 0 > argv[0] = /home/peter/argtest
Process 1 > argv[0] = /home/peter/argtest

Process 0 > argv[1] = hello,
Process 1 > argv[1] = hello,

Process 0 > argv[2] = world
Process 1 > argv[2] = world
```

Beware that if your implementation doesn't give each process access to the command line arguments, the program may crash or hang. If this is the case, you may want to try running the program with only one process to see if process 0 is passed the command line arguments. If it is, you can have it broadcast them to the other processes.

Our second alternative, self-initialization, is really quite simple. Just create a C source file with the necessary data stored in static variables, and write a short function that assigns the values in the static variables to parameters. For example, suppose we are writing a program that reads in the order of two vectors, and then reads in the two vectors.

```
/* data.c
 * A separate file containing the input data.
 */

static int n = 4;

static float a[] = {1.0, 2.0, 3.0, 4.0};

static float b[] = {4.0, 3.0, 2.0, 1.0};
```

```
void Initialize(
         int*   n_ptr,
         float  vector_a[],
         float  vector_b[]) {
    int i;

    *n_ptr = n;

     for (i = 0; i < n; i++) {
         vector_a[i] = a[i];
         vector_b[i] = b[i];
     }
} /* Initialize */
```

Now, if you want to run your program with different input data, you simply edit the file `data.c`, recompile it, and link it with your already compiled source code. This approach has the virtue that it is guaranteed to work. However, it has the liability that you do need to recompile every time you want to change the input.

8.3 File I/O

Thus far all of our attention has been focussed on `stdin`, `stdout`, and `stderr`. While the use of these streams is adequate for many applications, access to only these streams can be a severe limitation in others. Fortunately, many implementations of MPI allow all processes to read and write files other than `stdin`, `stdout`, and `stderr`. So we may have more flexibility in dealing with file I/O than with `stdin`, `stdout`, and `stderr`.

Of course, increased flexibility usually implies increased complexity, and dealing with file I/O is definitely more complex than dealing with `stdin`, `stdout`, and `stderr`. The reason for this is that when we deal with `stdin`, `stdout`, and `stderr`, we have essentially a single input stream and a single output stream, while with file I/O, we have the possibility of multiple input and multiple output streams. Furthermore, in a parallel system, the underlying hardware may make it possible to access multiple streams simultaneously. Thus, a key issue in parallel I/O is data mapping. For example, suppose a large array has been distributed across d disks, and we wish to access this array in a program with p processes. In general, p won't be the same as d, and we'll have to redistribute the data when we read it in. The details of implementing such a mapping and the design of language interface continue to be the subject of intensive research. As a consequence we'll limit ourselves to the simplest cases: $d = 1$ and $d = p$.

In the case where $d = 1$, we can simply modify our collective I/O functions so that they take a file parameter. For example, the `Cprintf` function might become `Cfprintf` and its declaration might be

```
int Cfprintf(
        FILE*     fp        /* in/out */,
        MPI_Comm  io_comm   /* in     */,
        char*     title     /* in     */,
        char*     format    /* in     */,
                  ...       /* in     */)
```

Of course, we'll need to define open and close functions analogous to the C functions. For example, to open a file, we might use the following function.

```
FILE* Cfopen(
        char*     filename  /* in */,
        char*     mode      /* in */,
        MPI_Comm  io_comm   /* in */) {

    int    root;
    int    my_io_rank;
    FILE*  fp;

    Get_io_rank(io_comm, &root);

    MPI_Comm_rank(io_comm, &my_io_rank);

    if (my_io_rank == root) {
        fp = fopen(filename, mode);
        return fp;
    } else {
        return NULL;
    }
}  /* Cfopen */
```

This function simply has the I/O process open the file and return a pointer to it, while the other processes return NULL.

Note that if all our processes can read files other than `stdin`, we can avoid the problems we discussed in the previous section by arbitrarily designating a process to be the I/O process (e.g., process 0), caching this rank with `io_comm`, putting the input data into a file, and using `Cfscanf` instead of `Cscanf`.

If each process is reading or writing a file, we can simply use the standard C file manipulation functions. For example, if each process is writing to a separate file, we might include something like this:

```
FILE*  my_fp;
int    my_rank;
char   filename[100];

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
sprintf(filename,"file.%d", my_rank);
my_fp = fopen(filename, "w");
```

```
        ⋮
fprintf(my_fp, "Greetings from Process %d!\n",
    my_rank);
        ⋮
fclose(my_fp);
MPI_Finalize();
```

A couple of points should be mentioned here. First, we were careful to make sure that each process opened a file with a different filename. We did this by appending the process rank to the filename. The reason for this is that although it may appear to us that each process is accessing a different disk, the I/O subsystem may in fact be directing all I/O to a single disk. For example, if we're using a network of workstations, it's highly likely that our home directory resides on a single disk that is NFS mounted on the other systems.

Second, `MPI_Init` and `MPI_Finalize` may change the program's view of the file system. For example, on a network of workstations, the pathname of the current working directory may be changed. Hence, files shouldn't be opened until after calling `MPI_Init`, and they should be closed before calling `MPI_Finalize`.

8.4 Array I/O

Unfortunately, our simple, collective I/O functions don't generalize to arrays very well. For example, suppose we have a distributed linear array of floats that we would like to print. Say each process has 100 floats, and the first 100 are on process 0, the next on process 1, etc. Since the `Cprintf` function expects to receive data from all the processes, together with a format string, it would be necessary to copy the 100 floats into a string using `sprintf` and send the string to `Cprintf`. If the floats are distributed in "cyclic" order—the first float in the global ordering is on process 0, the next on process 1, etc.—then it would be necessary to call `Cprintf` 100 times from each process, and it could be a *long* time before the data were printed. Finally, the data would be printed with the ordinarily useful, but in this case probably useless and probably annoying, information on the process from which the data came. So we would like to develop some more useful functions for array I/O.

Before proceeding, note that the key issue here, data distribution or mapping, has been encountered many times already. For example, we devoted section 2.2.5 to a brief discussion, and we discussed it in several connections in Chapters 4, 5, and 7. Thus, this is not simply an I/O issue: the problem of distributing composite data among processes is a central one in parallel computing. We're returning to it in this chapter because this is the first application we've discussed in which it's necessary to give it a somewhat more comprehensive treatment.

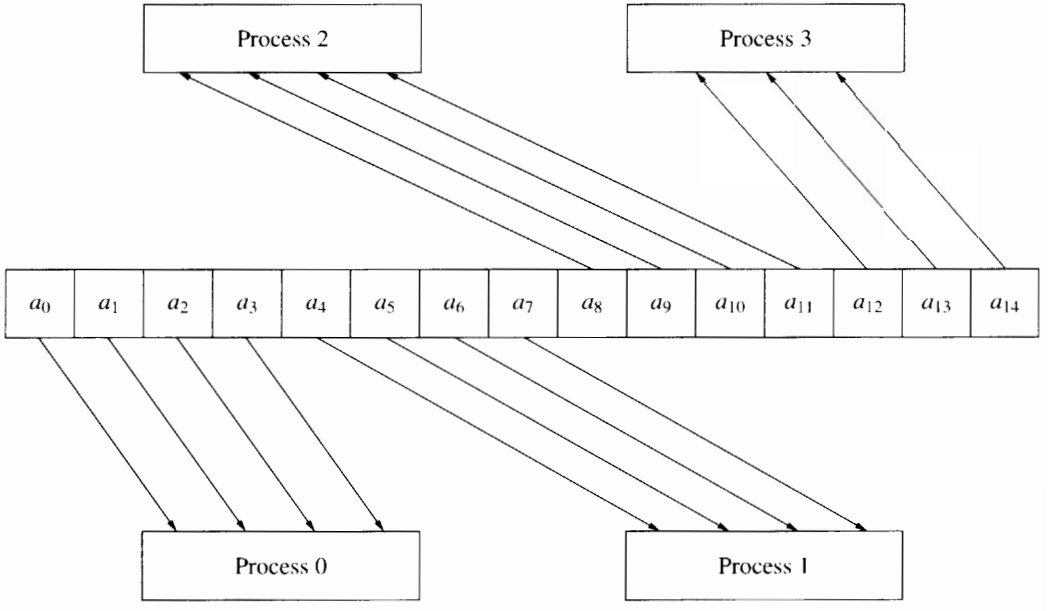

Figure 8.1 Block distribution of 15 elements among four processes

8.4.1 Data Distributions

Suppose that the array contains n elements, and we're running our program with p processes. For the sake of explicitness, suppose that the array is $A = (a_0, a_1, \ldots, a_{n-1})$. Since one of the goals of a well-designed parallel program is to equalize the amount of work done by the processes, we'll usually want to distribute the arrays so that the number of elements is the same (or nearly the same) on each process. That is, we would like to have approximately n/p elements assigned to each process. There are three basic methods that are commonly used for doing this. Here's a brief description.

1. **Block distribution.** If p evenly divides n, then the first n/p elements are assigned to process 0, the next n/p to process 1, etc. If p doesn't divide n evenly, suppose that $n = qp + r$, where $0 \leq r < p$. Then we can assign $\lceil n/p \rceil$ elements to the first r processes, and $\lfloor n/p \rfloor$ elements to the remaining $p - r$ processes.[2] See Figure 8.1.
2. **Cyclic distribution.** The first element is assigned to process 0, the next to process 1, etc., until we've assigned one element to each process. Then we assign the pth element to process 0, the $(p + 1)$st to process 1, etc.

2 Recall that $\lceil x \rceil$ is the smallest integer greater than or equal to x, and $\lfloor x \rfloor$ is the largest integer less than or equal to x.

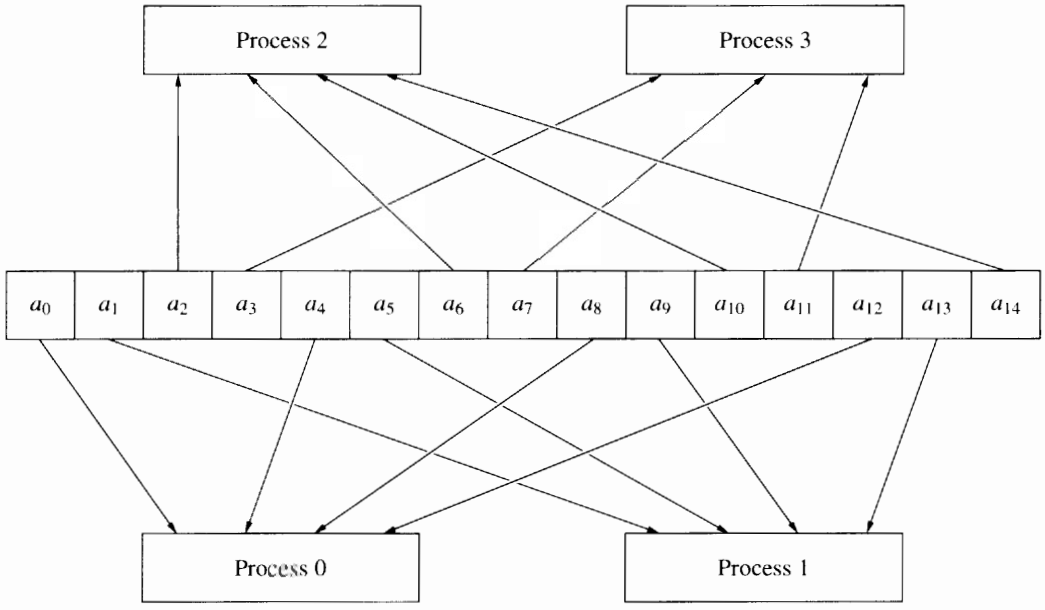

Figure 8.2 Cyclic distribution of 15 elements among four processes

If p doesn't divide n, and, as before, the remainder is r, we will have assigned $\lceil n/p \rceil$ elements to the first r processes and $\lfloor n/p \rfloor$ elements to the remaining $p - r$ processes. See Figure 8.2.

3. **Block-cyclic distribution.** In a block-cyclic distribution, there is a third parameter, the blocksize, b. The idea is to form a hybrid between the block and the cyclic distributions. We assign the first block of b elements to process 0, the next b elements to process 1, etc. If $pb < n$, we won't have exhausted all the elements after we've assigned b elements to process $p - 1$. So, as with the cyclic distribution, we go back to process 0, and continue. See Figure 8.3.

Clearly the cyclic distribution is a special case of the block-cyclic distribution: it is the block-cyclic distribution with blocksize 1. However, whether the block distribution is a special case of the block-cyclic depends on how the block-cyclic deals with remainders. We'll explore this issue further in the exercises.

As far as code goes, it would seem that we should develop functions for block-cyclic I/O. However, the complexity of these functions tends to obscure the main issues. On the other hand, coding of block I/O tends to miss some of the subtleties. So let's develop some functions for cyclic I/O.

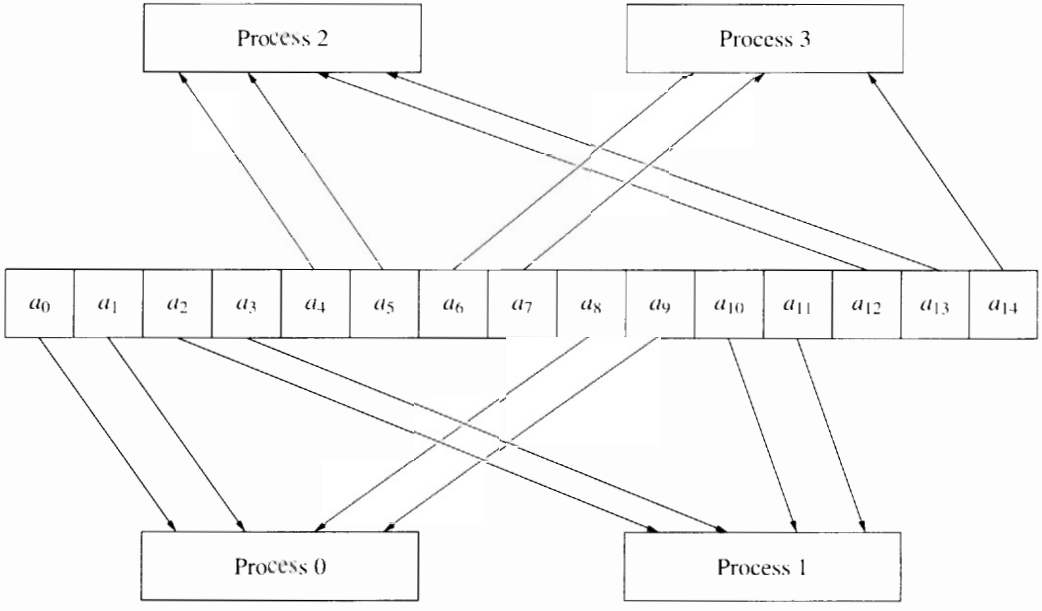

Figure 8.3 Block cyclic distribution of 15 elements among four processes with a blocksize of 2

8.4.2 Model Problem

In the process of designing the input and output functions, it will help to have a model problem in mind. With such a model in mind, it usually won't be too difficult to decide between several alternative solutions to subproblems we encounter in the process of developing our functions. Of course, the choices we make based on the model problem won't be suitable for all applications. However, if we review the design process we used, it shouldn't be too difficult to choose other alternatives and construct more suitable functions.

For our model problem, let's suppose our functions will be used in the solution of a linear system by an iterative method. Jacobi's method, which we'll discuss in Chapter 10, is an especially simple form of such a method. In this setting, we can think of our input array as the right-hand side of the linear system, $A\mathbf{x} = \mathbf{b}$, and the output array as the solution. Typical iterative methods for solving linear systems involve repeated use of the following operations:

1. Scalar multiplication and vector addition
2. Dot product
3. Matrix-vector multiplication

If we distribute the matrix of coefficients by rows among the processes and distribute the corresponding entries of the arrays in the same way, then the

first operation is perfectly parallel and the second (as we've already seen) will require a local dot product followed by an `MPI_Allreduce`. Recollect that for the third operation we form the dot product of each row of the matrix with the vector. Thus, if the vector is distributed, it will be necessary to perform an `MPI_Allgather` before we form the dot products of the rows of the matrix with the vector.

8.4.3 Distribution of the Input

We'll assume that the input will be stored in a single file that can be read by the I/O process. We'll also assume that the input array is preceded by n, the order of the array, and the actual array entries are floats. The I/O process will read in n and broadcast it to the other processes—we can use `Cscanf` to do this. We'll also read in the array entries on the I/O process. In order to distribute them, we encounter the usual "speed vs. memory usage" trade-off. We can read all the entries into a single array and use `MPI_Scatter` to distribute them, or we can read a block of, say, p entries, scatter them, read a second block, scatter, etc. Since we cannot use the same argument as both the input and output buffer to `MPI_Scatter`, in order to use the first approach we'll need two arrays on process 0, while we'll only need one array and a scalar if we use the second approach. However, in our model problem, we will want to gather the entire array onto each process in order to carry out a matrix-vector multiplication. So we will need the extra storage anyway, and we might as well use the faster first approach.

8.4.4 Derived Datatypes

Note that since the elements to be sent to a single process are not stored in contiguous entries after we read in the data on process 0, we'll need to use a derived datatype when we use `MPI_Scatter`. For example, if $p = 3, n = 12$, and our input array is

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12),$$

we'll send

$$\begin{aligned}(1, 4, 7, 10) &\longrightarrow \text{process 0,}\\ (2, 5, 8, 11) &\longrightarrow \text{process 1, and}\\ (3, 6, 9, 12) &\longrightarrow \text{process 2.}\end{aligned}$$

So the elements going to a single process are spaced a constant number of components apart, and for this array we could build a derived datatype using `MPI_Type_vector`. Recall its syntax:

```
int MPI_Type_vector(
        int             count          /* in  */,
        int             block_length   /* in  */,
        int             stride         /* in  */,
        MPI_Datatype    old_type       /* in  */,
        MPI_Datatype*   new_type       /* out */)
```

The parameter `count` is the number of blocks of elements in the type. In our example, it would be 4. The parameter `blocklength` is the number of contiguous elements in each block. In any cyclic mapping it should just be 1. The `stride` is the number of elements between the start of consecutive blocks. In our example, it's 3. Thus, we could build the derived type for our example with

```
MPI_Type_vector(4, 1, 3, MPI_FLOAT, &vector_mpi_t);
```

More generally, it would seem that the following code could be used to build the type:

```
count = n/p;
stride = p;

MPI_Type_vector(count, 1, stride, MPI_FLOAT,
    &vector_mpi_t);
```

However, you're probably wondering what happens if n doesn't evenly divide p. For example, suppose $n = 14$ and $p = 3$. Then if we're distributing the array

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14),$$

we should send

$$\begin{aligned}(1, 4, 7, 10, 13) &\longrightarrow \text{process 0,}\\ (2, 5, 8, 11, 14) &\longrightarrow \text{process 1, and}\\ (3, 6, 9, 12) &\longrightarrow \text{process 2.}\end{aligned}$$

But now the count parameter for the data sent to process 0 and process 1 should be 5, while it's still 4 for the data sent to process 3. If you recall that a derived datatype is a sequence of basic datatypes together with displacements, it becomes apparent that a single derived datatype cannot be used to represent both four floats and five floats. Thus, we either need to pad our arrays with fake data, or we need to reconsider our approach to data distribution. Since padding the arrays will increase their size by at most $p - 1$, and we expect n to be much larger than p in most cases of interest, let's take the easy alternative of padding the arrays.

The next subsection probes rather deeply into the details of MPI derived datatypes, so you may want to skip it on a first reading. However, since it

contains some very important methods for dealing with derived datatypes that contain noncontiguous data, you should return to it before continuing with the rest of the book.

8.4.5 The Extent of a Derived Datatype

OK, so let's take a look at the call to `MPI_Scatter`. Recall its syntax:

```
int MPI_Scatter(
        void*          send_buf    /* in  */,
        int            send_count  /* in  */,
        MPI_Datatype   send_type   /* in  */,
        void*          recv buf    /* out */,
        int            recv_count  /* in  */,
        MPI_Datatype   recv_type   /* in  */,
        int            root        /* in  */,
        MPI_Comm       comm        /* in  */)
```

In our input function we would call it as follows:

```
MPI_Scatter(input_data, 1, vector_mpi_t,
    local_data, padded_size/p, MPI_FLOAT,
    io_process, io_comm);
```

It might seem that there could be a problem with receiving noncontiguous data into contiguous locations, but this isn't a problem. Recall that we can receive data as long as the type signatures match, and that a derived datatype is just a list of pairs: the first element of each pair is a basic MPI type, the second is a displacement in bytes. So if we have 4-byte floats, `vector_mpi_t` can be represented as

$$\{(\texttt{MPI\_FLOAT}, 0), (\texttt{MPI\_FLOAT}, 4p), \ldots, (\texttt{MPI\_FLOAT}, 4(\texttt{padded\_size}/p - 1))\}.$$

A type signature is just a list of the basic types in a derived datatype, and hence the type signature of `vector_mpi_t` is just a list of `padded_size/p` `MPI_FLOAT`s.

Unfortunately, we've still got problems. The problem isn't at all obvious, but a careful reading of the MPI Standard shows that the data sent to process q in our example will begin

```
q * extent(vector_mpi_t)
```

bytes beyond the beginning of the `input_data` array. For example, the data going to process 1 will begin `extent(vector_mpi_t)` bytes beyond the beginning of the array. So if `extent(vector_mpi_t)` is different from `sizeof(float)`, we're in trouble. The **extent** of a derived datatype is, roughly speaking, the distance, in bytes, from the beginning to the end of the type. So in our

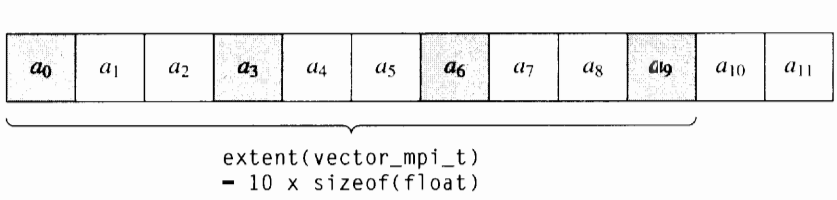

Figure 8.4 Extent of `vector_mpi_t`

example of distributing an array of order 12 among three processes, if floats are 4 bytes long, our derived datatype consists of four equally spaced floats:

$$\{(\texttt{MPI\_FLOAT}, 0), (\texttt{MPI\_FLOAT}, 12), (\texttt{MPI\_FLOAT}, 24), (\texttt{MPI\_FLOAT}, 36)\}.$$

So the extent of our type isn't 4 bytes; it's 40 bytes! See Figure 8.4. Thus, the first four floats should arrive on process 0, but the floats going to process 1 will start with the 10th element of the input array, and we'll run past the end of the array when we try to send the data to processes 1 and 2.

All is not lost, however. MPI provides a method for us to artificially change the extent of a type. The basic idea is that we add an "upper bound" marker to the type. This marker overrides the natural extent calculation, and when the system calculates the extent of the type, instead of computing the distance from the beginning to the end of the entire type, it only computes the distance from the beginning of the type to the marker. All the other properties of the type are unchanged. In particular, the actual content—`padded_size`/p floats—is unchanged. In order to do this, we need to use `MPI_Type_struct`.

```
int           block_lengths[2];
MPI_Aint      displacements[2];
MPI_Datatype  types[2];
MPI_Datatype  vector_mpi_t;
MPI_Datatype  ub_mpi_t;

/* First create vector_mpi_t                          */
count = padded_size/p;
stride = p;
MPI_Type_vector(count, 1, stride, MPI_FLOAT,
    &vector_mpi_t);

/* The first type is vector_mpi_t                     */
types[0] = vector_mpi_t;

/* The second type is MPI_UB                          */
types[1] = MPI_UB;

/* vector_mpi_t starts at displacement 0              */
displacements[0] = 0;
```

```
/* MPI_UB starts at displacement sizeof(float)  */
displacements[1] = sizeof(float);

/* The derived type will have 1 element of type */
/*     vector_mpi_t and 1 element of type       */
/*     MPI_UB                                   */
block_lengths[0] = block_lengths[1] = 1;

/* Now create the full type */
MPI_Type_struct(2, block_lengths, displacements,
    types, &ub_mpi_t);
MPI_Type_commit(&ub_mpi_t);
```

A nontrivial exercise, but an extremely useful one!

Now `ub_mpi_t` is the same as the original `vector_mpi_type`, except for the `MPI_UB` marker. Thus, if floats are 4 bytes, our type is

$$\{(\texttt{MPI\_FLOAT}, 0), (\texttt{MPI\_UB}, 4), (\texttt{MPI\_FLOAT}, 4p), \ldots, (\texttt{MPI\_FLOAT}, 4(\texttt{padded\_size}/p - 1))\},$$

and its extent is computed by taking the difference between the displacement of the `MPI_UB` and the start of the type; i.e, $4 - 0 = 4$ bytes.

8.4.6 The Input Code

OK. Now we're ready to write some of the code. A program that uses the input function should first read in n using `Cscanf`. Before reading in the array entries, we'll build a data structure that can be used for storing the array. Rather than just storing the entries in an array, it will be extremely useful to group such information as the global array size, the padded array size, the local array size, etc., with the array itself. Thus, a header file might include the following definitions:

```
typedef struct {
    MPI_Comm* comm;           /* Comm for collective ops */
    #define Comm_ptr(array)   ((array)->comm)
    #define Comm(array)       (*((array)->comm))

    int       p;              /* Size of array_comm      */
    #define Comm_size(array)  ((array)->p)

    int       my_rank;        /* My rank in array_comm   */
    #define Comm_rank(array)  ((array)->my_rank)

    int       global_order;   /* Global size of array    */
    #define Order(array)      ((array)->global_order)
```

```
    int       padded_size;   /* Padded array size       */
    #define Padded_size(array) ((array)->padded_size)

    float     entries[MAX];  /* Elements of the array   */
    #define Entries(array)   ((array)->entries)
    #define Entry(array,i)   ((array)->entries[i])

    int       local_size;    /* = n/p or n/p+1          */
    #define Local_size(array) ((array)->local_size)

    float     local_entries[LOCAL_MAX];
                             /* Local entries of array  */
    #define Local_entries(array) ((array)->local_entries)
    #define Local_entry(array,i) ((array)->local_entries[i])

    MPI_Datatype ub_mpi_t;   /* The derived datatype    */
    #define Type(array)      ((array)->ub_mpi_t)
} CYCLIC_ARRAY_STRUCT;

typedef CYCLIC_ARRAY_STRUCT* CYCLIC_ARRAY_T;
```

The macros are defined for member access. Although this muddies up the type definition, it has a couple of advantages that make it well worthwhile. First it makes the rest of the code more readable: `Entry(A,i)` is almost certainly easier to understand than `A->entries[i]` or `*(A->(entries+i))`. It also makes it easy to change the member definition. If, for example, we decide to group the scalars in a substructure, we can simply change the definition of the macro, and the rest of the program won't need modification.

Assigning values to the scalar entries in the struct is straightforward. We'll also need to use the code from section 8.4.5 to define `ub_mpi_t`. But after we have taken care of these matters, the actual input function is surprisingly simple. First get the I/O process rank cached with the communicator member of the array struct, and have the I/O process read the entire array into its global entries member. After reading in the array, the "fake" entries should be assigned some value—e.g., 0. Then we can call `MPI_Scatter`. For the scatter, we'll use our derived type on the I/O process, but we'll receive the local entries into a contiguous block. So we'll just use `recv_count = padded_size/p` and `recv_type = MPI_FLOAT`.

```
void Read_entries(
         char*           prompt  /* in     */,
         CYCLIC_ARRAY_T  array   /* in/out */) {

    int root;
    int i;
    int c;
    int recv_count;
```

```
    Get_io_rank(Comm(array), &root);

    if (Comm_rank(array) == root) {
        printf("%s\n",prompt);
        for (i = 0; i < Order(array); i++)
            scanf("%f", &Entry(array,i));
        /* Skip to end of line */
        while ((c = getchar()) != '\n');

        /* Fill padding with 0's */
        for (i = Order(array); i < Padded_size(array); i++)
            Entry(array,i) = 0.0;
    }

    /* Receive array element into contiguous block of */
    /*     Local_entries(array)                       */
    recv_count = Padded_size(array)/Comm_size(array);
    MPI_Scatter(Entries(array), 1, Type(array),
        Local_entries(array), recv_count,
        MPI_FLOAT, root, Comm(array));

} /* Read_entries */
```

8.4.7 Printing the Array

Printing the array is also quite easy. We can simply get the I/O process rank of the array communicator, gather the array onto the I/O process, and have the I/O process print the entries. The only complexity occurs if we want a clear indication of which entries belong to which process and, at the same time, a clear picture of the sequential structure of the array. We can arrange for this by printing p entries of the array in each line of output. When we do this, a single column of output will contain the entries assigned to a single process, but the sequential structure of the array will be preserved as we read from left to right and from top to bottom. For example, if the array

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)$$

is scattered among three processes, we might print it as follows:

```
    Processes
    0        1        2
-------------------------
   1.0      2.0      3.0
   4.0      5.0      6.0
   7.0      8.0      9.0
  10.0     11.0     12.0
  13.0     14.0
```

In order to do this, we can compute the integer quotient $q = n/p$, and the remainder $r = n \bmod p$. Then we should print q lines consisting of p entries and 1 line consisting of r entries.

```
void Print_entries(
         char*            title  /* in */,
         CYCLIC_ARRAY_T   array  /* in */) {
    int root;
    int q;
    int quotient;
    int remainder;
    int i, j, k;
    int send_count;

    Get_io_rank(Comm(array), &root);

    send_count = Padded_size(array)/Comm_size(array);
    MPI_Gather(Local_entries(array), send_count, MPI_FLOAT,
        Entries(array), 1, Type(array), root, Comm(array));

    if (Comm_rank(array) == root) {
        printf("%s\n", title);
        printf("    Processes\n");
        for (q = 0; q < Comm_size(array); q++)
            printf("%4d    ",q);
        printf("\n");
        for (q = 0; q < Comm_size(array); q++)
            printf("--------");
        printf("\n");

        quotient = Order(array)/Comm_size(array);
        remainder = Order(array) % Comm_size(array);
        k = 0;
        for (i = 0; i < quotient; i++) {
            for (j = 0; j < Comm size(array); j++) {
                printf("%7.3f ",  Entry(array,k));
                k++;
            }
            printf("\n");
            fflush(stdout);
        }
        for (j = 0; j < remainder; j++) {
            printf("%7.3f ",  Entry(array,k));
            k++;
        }
        printf("\n");
        fflush(stdout);
    }
} /* Print_entries */
```

8.4.8 An Example

In order to clarify the ideas, let's write a short program that uses our various I/O functions. It will simply read in two vectors (x and y), distribute them among the processes, find their sum (z), and print the result.

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

/* Header file for the basic I/O functions */
#include "cio.h"

/* Header file for the cyclic array I/O functions */
#include "cyclic_io.h"

main(int argc, char* argv[]) {
    CYCLIC_ARRAY_STRUCT  x;
    CYCLIC_ARRAY_STRUCT  y;
    CYCLIC_ARRAY_STRUCT  z;
    int                  n;
    MPI_Comm             io_comm;
    int                  i;

    MPI_Init(&argc, &argv);

    /* Build communicator for I/O */
    MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);
    if (Cache_io_rank(MPI_COMM_WORLD, io_comm) ==
            NO_IO_ATTR)
        MPI_Abort(MPI_COMM_WORLD, -1);

    /* Get n */
    Cscanf(io_comm, "Enter the array order", "%d", &n);

    /* Allocate storage and initialize scalar    */
    /*     members.  Calls function for building */
    /*     derived type                          */
    Initialize_params(&io_comm, n, &x);
    Initialize_params(&io_comm, n, &y);
    Initialize_params(&io_comm, n, &z);

    /* Get vector elements */
    Read_entries("Enter elements of x", &x);
    Read_entries("Enter elements of y", &y);

    /* Add local entries */
    for (i = 0; i < Local_size(&x); i++)
        Local_entry(&z,i) =
            Local_entry(&x,i) + Local_entry(&y,i);
```

```
        /* Print z */
        Print_entries("x + y =", &z);

        MPI_Finalize();
}
```

If we run this with four processes and input

```
14
1 2 3 4 5 6 7 8 9 10 11 12 13 14
14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

the output will be

```
Enter the array order
Enter elements of x
Enter elements of y
x + y =
    Processes
    0        1        2        3
-----------------------------------
  15.0     15.0     15.0     15.0
  15.0     15.0     15.0     15.0
  15.0     15.0     15.0     15.0
  15.0     15.0
```

8.5 Summary

We've covered a lot of ground in this chapter. Our goal was to write some functions that we can use for convenient I/O on parallel systems. However, in order to write these functions, we learned a lot about MPI and took a brief look at data distributions and a few of the issues involved in parallel I/O.

We started the chapter with a discussion of the problem of I/O on parallel systems. The most important point was that since there isn't a consensus on the mechanics of I/O on parallel systems, the MPI Standard imposes no requirements on the I/O capabilities of an MPI implementation. An unfortunate consequence of this is that programmers have no guaranteed I/O interface, and the main purpose of this chapter was to develop some simple I/O functions. We assume that there is at least one process that can carry out basic I/O—e.g., `printf` and `scanf`—and turn our I/O functions into *collective* operations: data to be read is read in by the process with I/O capabilities and distributed to the other processes, and data to be printed is collected onto the I/O process and printed.

Since I/O is collective, our I/O functions all take a communicator argument. Since the rank of the process with I/O capabilities is communicator

dependent, it would be convenient if we could somehow attach this information to a communicator. MPI provides just such a facility: it's called *attribute caching*. Since, in general, we may want to attach much more complex sets of information with communicators, attributes in MPI are just pointers. So we distinguish between the attribute and the *attribute content*. For our I/O functions the attribute was a pointer to an int, and the attribute content was the rank of the process that can carry out I/O. Since a communicator may have many attributes cached, each attribute has a system-defined *key*.

Attributes and keys are process local: each process in a communicator may have different keys identifying an attribute and different attribute content associated with the key. Indeed, all the attribute access functions are purely local operations, so it's entirely possible that one process may have a given attribute defined while another doesn't.

We can create a key with the MPI function `MPI_Keyval_create`:

```
int MPI_Keyval_create(
        MPI_Copy_function*    copy_fn    /* in  */,
        MPI_Delete_function*  delete_fn  /* in  */,
        int*                  key_ptr    /* out */,
        void*                 extra_arg  /* in  */)
```

The main significance of this function for our purposes was that it returned a pointer to a key that we could use to identify our I/O process rank attribute. The parameters `copy_fn` and `delete_fn` are called *callback* functions. The `copy_fn` parameter tells MPI what should be done with the attribute when a communicator is duplicated with `MPI_Comm_dup`,

```
int MPI_Comm_dup(
        MPI_Comm  old_comm  /* in  */,
        MPI_Comm* new_comm  /* out */)
```

The `delete_fn` parameter tells MPI what should be done when a communicator is freed with the collective function

```
int MPI_Comm_free(
        MPI_Comm*  comm  /* in/out */)
```

or when an attribute is deleted with

```
int MPI_Attr_delete(
        MPI_Comm   comm    /* in */
        int        keyval  /* in */)
```

Both `copy_fn` and `delete_fn` can, in general, be user defined. However, for our I/O process rank attribute, we just used the predefined MPI functions, `MPI_DUP_FN` and `MPI_NULL_DELETE_FN`. The first simply copies the attribute

(pointer) from the old communicator to the new communicator, and the second does nothing. The purpose of the `extra_arg` parameter is to provide additional information to the `copy_fn` and the `delete_fn`; we made no use of it.

After creating a key for the I/O process rank attribute, we needed to devise a method for determining the rank of a process that could carry out I/O. We did this by accessing the predefined MPI attribute with key, `MPI_IO`. This attribute should be cached with `MPI_COMM_WORLD` in all MPI implementations. Its content can either be `MPI_PROC_NULL` (no process can carry out I/O), `MPI_ANY_SOURCE` (all processes can carry out I/O), or a process rank. We can determine the value of this and any other attribute by calling

```
int MPI_Attr_get(
        MPI_Comm  comm      /* in  */,
        int       key       /* in  */,
        void*     attr_ptr  /* out */,
        int*      flag      /* out */)
```

If the attribute identified by `key` has been cached with `comm`, on return the value referenced by `flag` will be nonzero, and `attr_ptr` will reference the attribute. Be careful here: `attr_ptr` points to the actual attribute, which in turn points to the content of the attribute.

Once we determine the rank of a process that can carry out I/O, we can cache with the communicator by calling

```
int MPI_Attr_put(
        MPI_comm   comm       /* in */,
        int        key        /* in */,
        void*      attribute  /* in */)
```

This will cache `attribute` with `comm`. The attribute is identified by `key`. Note that unlike `MPI_Attr_get` we pass the actual attribute to `MPI_Attr_put`. This distinction is natural: in `MPI_Attr_put`, the system copies the address of the attribute content to system-defined memory, while in `MPI_Attr_get` we want to *return* the attribute, and the only way we can do this in C is to return a pointer to the attribute. The confusion arises because both parameters have the same type—`void*`—and the reason for this is that `void*` indicates a pointer to anything, including another pointer.

We created a separate communicator for I/O so that the communications in the I/O functions couldn't be confused with other communication. However, this meant that after the new I/O communicator was created, we might need to get the I/O rank from another communicator. This led to our using two new MPI functions:

```
int MPI_Comm_compare(
        MPI_Comm   comm1   /* in  */,
        MPI_Comm   comm2   /* in  */,
        int*       result  /* out */)
```

```
int MPI_Group_translate_ranks(
        MPI_Group  group1         /* in  */,
        int        array_size     /* in  */,
        int        ranks_in_1[]   /* in  */,
        MPI_Group  group2         /* in  */,
        int        ranks_in_2[]   /* out */)
```

The first function compares the groups and contexts of comm1 and comm2. If they're both identical, it returns MPI_IDENT. If the groups are identical but the contexts are different, it returns MPI_CONGRUENT. If the groups contain the same processes but the order of the processes is different, it returns MPI_SIMILAR. Otherwise it returns MPI_UNEQUAL.

The second function, MPI_Group_translate_ranks, takes two arrays of size array_size. The array ranks_in_1 contains a list of process ranks in group1. The second array returns the corresponding process ranks in group2—if they exist. If a process listed in ranks_in_1 doesn't belong to group2, the corresponding entry in ranks_in_2 will be MPI_UNDEFINED.

Most of our work in section 8.1 was spent on learning about attributes and communicators. Once we had this material, our actual communicator access functions and our I/O functions were fairly simple to write. The input function Cscanf gets the rank of the I/O process, reads in the data on this process, and broadcasts it to the other processes. The output function Cprintf reverses the sequence: after getting the rank of the I/O process, it gathers the data from all the processes onto process 0 and prints the data from each. Probably the most complicated of the functions was our Cerror test function. In it we gathered error codes from all the processes onto *all* the processes, and then each process systematically checked the codes to see if any process had encountered a problem. If this was the case, the I/O process printed the ranks of the processes that had encountered errors, and all the processes called

```
MPI_Abort(
    MPI_Comm   comm        /* in */
    int        error_code  /* in */)
```

It tries to shut down all the processes in comm. However, its behavior is implementation dependent, and an implementation is only required to try to shut down all the processes in MPI_COMM_WORLD. In a UNIX environment, the error code is returned as if the main program returned with

```
return error_code;
```

After writing the basic I/O functions we considered some problems users might encounter and possible solutions. The main issue here was access to stdin. All MPI implementations with which we're familiar allow at least one process access to stdout and stderr. However, it is not unusual for an implementation to provide no access to stdin. If this is the case, there are

basically three additional options. The simplest may be to use command line arguments. If there isn't much input to our program, we can type the input on the command line, and many MPI implementations send the command line arguments to all the processes. A second alternative is to put the input in a separate C source file. Then changing input will involve recompiling the separate source file and relinking it with the rest of the code. The final option was to get input from a file other than `stdin`.

This last option brought up the more general problem of I/O to files other than `stdin`, `stdout`, and `stderr`. In general parallel systems there may be multiple I/O devices attached to multiple processes. So general parallel I/O can be extremely complex. We only briefly considered the case of a single input or output file, and the case where each process opens its own input or output file. In the first case, we saw that we could modify our functions for accessing `stdin`, `stdout`, and `stderr` so that they take file parameters. In the second case, we can simply use C's file I/O functions. The main problem here is to make sure that each process is writing a different file from the other processes.

The functions we developed in the first part of the chapter were designed for the I/O of small amounts of data. For large amounts of data they will be extremely slow and cumbersome. So the last section of the chapter dealt with array I/O. In order to design array I/O functions, it was necessary to consider how the arrays would be distributed among the processes. So we briefly recalled the three most common distributions: block, cyclic, and block-cyclic. We decided to develop functions for the I/O of cyclic arrays. The functions used the same model that we used for I/O of scalar data: a single process carried out the actual I/O, and the I/O functions were collective operations. Thus, the input function, `Read_entries`, retrieved the rank of the I/O process, had it read in the data into a single array, and scattered the data among the processes using `MPI_Scatter`. The output function, `Print_entries`, retrieved the I/O rank and gathered the data onto the I/O process, which printed the array.

Since we were using a cyclic distribution of the data, the data sent to a single process by `Read_entries` came from noncontiguous locations in the array used by the I/O process. So we had to use `MPI_Type_vector` to create a derived datatype, `vector_mpi_t`, for the distribution. However, we discovered that `MPI_Scatter` doesn't choose the data being sent to the different processes the way we wanted it to. For example, in our distribution, the data being sent to process 1 should begin one float past the point where the data sent to process 0 begins. However, `MPI_Scatter` will send the data beginning `extent(vector_mpi_type)` bytes past the point where the data sent to process 0 begins, and the extent of `vector_mpi_type` was much larger than a single float. This led to a discussion of the extent of an MPI datatype, and a method for artificially redefining a type's extent.

Roughly speaking, the extent of an MPI datatype is the distance in bytes from the beginning to the end of the type. So if the input array consisted of n

floats and we were using p processes, the extent of `vector_mpi_t` was

$$(n - p + 1) * \texttt{sizeof(float)}$$

bytes. However, MPI provides a special type, `MPI_UB`, that can be used to artificially change the extent of a type. We did this by first building `vector_mpi_t` using `MPI_Type_vector` as before, but then we used `MPI_Type_struct` to build a type from `vector_mpi_t` starting at displacement 0, and `MPI_UB` starting at displacement `sizeof(float)` bytes.

8.6 References

The main reference for this chapter is, as usual, the MPI Standard [28, 29]. Further discussion of attributes and derived datatypes can be found in both [21] and [34].

Kernighan and Ritchie [24] discuss the use of the `stdarg` package. They also provide a small example of how to write a "minimal" `printf` function. This can be used as a model for writing your own `vsscanf` function. For detailed discussions of issues in general parallel I/O, see [9] and [10].

8.7 Exercises

1. Create two files `cio.h` and `cio.c`. The first file should contain the declarations, definitions, etc., needed so that when a source program includes it, it can use our collective I/O functions. The second should contain the actual source code for the collective I/O functions. Make sure that you've set things up correctly by writing a short program that reads in different types of data, modifies them, and prints out the modified data.
2. Modify the functions in `cio.c` so that they take a file argument. Create analogs of `fopen` and `fclose`.
3. Repeat exercise 1 for the cyclic array I/O functions. In this case you'll have to write some of the functions yourself—e.g., `Initialize_params`. Name your files `cyclic_io.h` and `cyclic_io.c`. Be sure to test them. In order to reduce memory usage, you may want to delete the `entries` member and use local storage instead.
4. Modify the array I/O functions so that they take a file parameter.
5. Modify the array I/O functions so that the processes store the local array data in the same form in which it is stored globally. For example, if $p = 3$ and the input array is

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$$

then the local arrays should look something like this:

Process 0: $(1, -, -, 4, -, -, 7, -, -, 10)$

Process 1: $(-, 2, -, -, 5, -, -, 8, -, -)$

Process 2: $(-, -, 3, -, -, 6, -, -, 9, -)$

The "$-$" indicates an undefined entry. Modify the vector sum program so that it uses this new format.

6. If you store a matrix as a linear array, then a block-row distribution of the matrix corresponds to a block distribution of the array, while a block-column distribution corresponds to a block-cyclic distribution of the array. Discuss the MPI derived datatype declarations that would be needed for the I/O of a matrix that uses a block-column distribution. Can you devise MPI derived datatypes that could be used for other distributions? Consider cyclic row distributions, cyclic column distributions, and block-checkerboard distributions.
7. Programs whose performance is limited by I/O speed rather than processor speed are said to be **I/O bound.** List some application programs that are probably I/O bound.

8.8 Programming Assignments

1. Write a function that will broadcast an int from process 0 in `MPI_COMM_WORLD` to the other processes in `MPI_COMM_WORLD`. The function should use a tree-structured broadcast. The first time the function is called, each process should cache information on the structure of the broadcast with `MPI_COMM_WORLD`. For example, if there are 8 processes, process 0 should store the information that it will first send the data to process 4, then process 2, and finally process 1. Process 2 should store the information that it will be idle during the first stage of the broadcast, during the second it will receive from 0, and during the third it will send to 3. This information can be stored in an array with one element for each stage. Each element can consist of the operation to be carried out at the stage (send, receive, idle) and the rank of a process if the operation is send or receive. Use the callback functions `MPI_DUP_FN` and `MPI_NULL_DELETE_FN`.
2. If you have an implementation of MPI that runs on a network of workstations and the workstations are not all using the same disk for their home directories, use the file I/O functions to write an `rcp` function. It should take as input the name of a file and two process ranks. The first process has access to the file. The second process should create a copy of the file on its local disk under the same name.
3. Modify your broadcast function so that it uses callback functions that you've defined. The copy function should allocate enough memory to

store the attribute content and copy the attribute content from the old communicator to the new. The delete function should free the memory referenced by the attribute.

4. Write functions for the I/O of arrays that use a block distribution.
5. Write functions for the I/O of arrays that use a block-cyclic distribution. If p is the number of processes, b is the blocksize, and n is the array order, assume that pb evenly divides n.