CHAPTER 6

Grouping Data for Communication

WE MENTIONED IN CHAPTER 3 that with the current generation of parallel systems, sending a message is an expensive operation. A natural consequence of this is that, as a rule of thumb, the fewer messages sent, the better the overall performance of the program. However, in each of our trapezoidal rule programs, when we distributed the input data, we sent a, b, and n in separate messages—whether we used `MPI_Send` and `MPI_Recv` or `MPI_Bcast`. So we should be able to improve the performance of our program by sending the three input values in a single message. MPI provides three mechanisms for grouping individual data items into a single message: the `count` parameter to the various communication routines, derived datatypes, and `MPI_Pack`/`MPI_Unpack`. We examine each of these options in turn.

6.1 The `count` Parameter

Recall that `MPI_Send`, `MPI_Receive`, `MPI_Bcast`, and `MPI_Reduce` all have a `count` and a `datatype` parameter. These two parameters allow the user to group data items having the same basic type into a single message. In order to use this, the grouped data items must be stored in *contiguous* memory locations. Since C guarantees that array elements are stored in contiguous memory locations, if we wish to send the elements of an array, or a subset of an array, we can do so in a single message. In fact, we've already done this in Chapter 3, when we sent an array of char.

As another example, suppose we wish to send the second half of a vector containing 100 floats from process 0 to process 1.

```
float vector[100];
MPI_Status status;
int p;
int my_rank;
        :
/*  Initialize vector and send */
if (my_rank == 0) {
        :
    MPI_Send(vector+50, 50, MPI_FLOAT, 1, 0,
        MPI_COMM_WORLD);
} else { /* my_rank == 1 */
    MPI_Recv(vector+50, 50, MPI_FLOAT, 0, 0,
        MPI_COMM_WORLD, &status);
}
```

Unfortunately, this doesn't help us with the trapezoidal rule program. The data we wish to distribute to the other processes, a, b, and n, are not stored in an array. So even if we declared them one after the other in our program,

```
float a;
float b;
int n;
```

C does not guarantee that they are stored in contiguous memory locations. One might be tempted to store n as a float and put the three values in an array, but this would be poor programming style. In order to solve our problem we need to use one of MPI's other facilities for grouping data.

6.2 Derived Types and MPI_Type_struct

It might seem that another option would be to store a, b, and n in a struct with three members—two floats and an int—since C does guarantee that the members of a struct are stored in contiguous memory locations. However, this solution introduces another problem. Suppose we included the type definition

```
typedef struct {
    float  a;
    float  b;
    int    n;
} INDATA_T;
```

and the variable definition

```
INDATA_T indata;
```

Now suppose we call MPI_Bcast

```
MPI_Bcast(&indata, 1, INDATA_T, 0, MPI_COMM_WORLD);
```

What happens? It won't work. The compiler should scream at you when you try to do this: arguments to functions must be *variables*, not defined types. What we need is a method of defining a type that can be used as a function argument—i.e., a type that can be stored in a variable. Yes, you guessed it, MPI provides just such a type: MPI_Datatype. The problem now is how do we define a variable of type MPI_Datatype that represents two floats and an int?

Let's suppose we've declared a, b, and n in our main program as follows:

```
float  a;
float  b;
int    n;
```

(We could use the struct indata, but it's not necessary.) Also suppose that the user has entered "0.0 1.0 1024" when prompted by the program for input and that on process 0, a, b, and n are stored as follows:

Variable	*Address*	*Contents*
a	24	0.0
b	40	1.0
n	48	1024

In order for the communications subsystem to send a, b, and n in a single message, the following information is required:

1. There are three elements to be transmitted.
2. a. The first element is a float.
   b. The second element is a float.
   c. The third element is an int.
3. a. The first element has address &a.
   b. The second element has address &b.
   c. The third element has address &n.

Looking at this in a somewhat different way, we can compute the **relative addresses** or **displacements** of b and n from a and only provide the address &a. According to our table, a has address &a = 24. The second float, b, is *displaced* 40 − 24 = 16 bytes beyond a. The int, n, is *displaced* 48 − 24 = 24 bytes beyond a. So, alternatively, in order for process 0 to specify completely the data to be transmitted, the following information can be provided to the communications subsystem:

1. There are three elements to be transmitted.
2. a. The first element is a float.

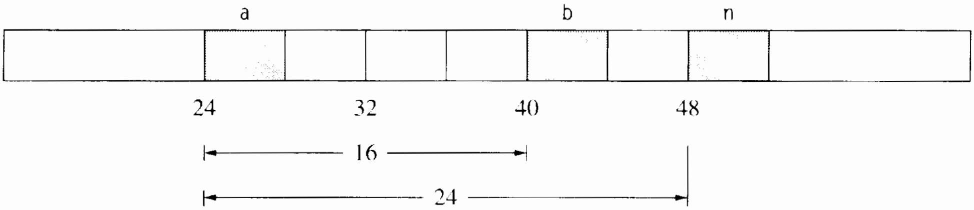

Figure 6.1 Memory layout with displacements

   b. The second element is a `float`.
   c. The third element is an `int`.
3. a. The first element is displaced 0 bytes from the beginning of the message.
   b. The second element is displaced 16 bytes from the beginning of the message.
   c. The third element is displaced 24 bytes from the beginning of the message.
4. The beginning of the message has address `&a`.

See Figure 6.1.

Note also that with this information (displacements computed according to local data layouts), each of the receiving processes can determine exactly where the data should be received. The principle behind MPI's derived datatypes is to provide all of the information *except* the address of the beginning of the message in a new *MPI* datatype. Then, when a program calls `MPI_Send`, `MPI_Recv`, etc., it simply provides the address of the first element, and the communications subsystem can determine exactly what needs to be sent or received. In effect, we are defining a struct during execution, rather than at compile time.

More precisely, a **general MPI datatype** or **derived datatype** is a sequence of pairs

$$\{(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1})\},$$

where each t_i is a basic MPI datatype and each d_i is a displacement in bytes. Recall that the basic MPI datatypes correspond for the most part to the predefined types in C: `MPI_INT`, `MPI_CHAR`, `MPI_FLOAT`, etc.—see Table 3.1. A displacement d_i is the number of bytes the element of type t_i lies from the start of a message using the derived type. In order to send `a`, `b`, and `n` in a single message, we would like to build the following derived datatype.

$$\{(\texttt{MPI\_FLOAT}, 0), (\texttt{MPI\_FLOAT}, 16), (\texttt{MPI\_INT}, 24)\}$$

There are several mechanisms for building derived datatypes. Let's take a look at an example of one of them: this is how we might build a derived datatype that can be used to incorporate `a`, `b`, and `n` into a single message.

```
void Build_derived_type(
         float*          a_ptr            /* in   */,
         float*          b_ptr            /* in   */,
         int*            n_ptr            /* in   */,
         MPI_Datatype*   mesg_mpi_t_ptr   /* out  */) {
                         /* pointer to new MPI type */

    /* The number of elements in each "block" of the    */
    /*     new type.  For us, 1 each.                   */
    int block_lengths[3];

    /* Displacement of each element from start of new   */
    /*     type.  The "d_i's."                          */
    /* MPI_Aint ("address int") is an MPI defined C     */
    /*     type.  Usually an int or a long int.         */
    MPI_Aint displacements[3];

    /* MPI types of the elements.  The "t_i's."         */
    MPI_Datatype typelist[3];

    /* Use for calculating displacements                */
    MPI_Aint start_address;
    MPI_Aint address;

    block_lengths[0] = block_lengths[1]
                     = block_lengths[2] = 1;

    /* Build a derived datatype consisting of  */
    /* two floats and an int                   */
    typelist[0] = MPI_FLOAT;
    typelist[1] = MPI_FLOAT;
    typelist[2] = MPI_INT;

    /* First element, a, is at displacement 0      */
    displacements[0] = 0;

    /* Calculate other displacements relative to a */
    MPI_Address(a_ptr, &start_address);

    /* Find address of b and displacement from a   */
    MPI_Address(b_ptr, &address);
    displacements[1] = address - start_address;

    /* Find address of n and displacement from a   */
    MPI_Address(n_ptr, &address);
    displacements[2] = address - start_address;

    /* Build the derived datatype */
    MPI_Type_struct(3, block_lengths, displacements,
        typelist, mesg_mpi_t_ptr);
```

```
    /* Commit it--tell system we'll be using it for */
    /* communication.                                */
    MPI_Type_commit(mesg_mpi_t_ptr);
} /* Build_derived_type */

void Get_data3(
         float*      a_ptr     /* out */,
         float*      b_ptr     /* out */,
         int*        n_ptr     /* in  */,
         int         my_rank   /* in  */) {
    MPI_Datatype  mesg_mpi_t; /* MPI type corresponding */
                              /* to a, b, and n         */

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }

    Build_derived_type(a_ptr, b_ptr, n_ptr, &mesg_mpi_t);
    MPI_Bcast(a_ptr, 1, mesg_mpi_t, 0, MPI_COMM_WORLD);
} /* Get_data3 */
```

Observe that in the call to `MPI_Type_struct` we provide all but one of the items we listed as necessary for correct identification of the data to be sent (or received):

1. The first argument (3) is the number of elements (or more generally blocks of elements) in the new MPI type.
2. The fourth argument (`typelist`) contains a list of the types of the elements to be sent.
3. The third argument (`displacements`) contains a list of the displacements of the elements from the beginning of the message.

The beginning of the message (`&a`) is omitted to allow for the possibility that the derived datatype represents a data layout that occurs often in the program. For example, it might represent a frequently used user-defined struct. If this is the case, the derived type can be used for *any* variable having the frequently used type.

The remaining argument to `MPI_Type_struct`, `block_lengths`, allows for the possibility that an element is an array. For example, if the second element of the derived type consisted of ten floats rather than one, we would have initialized `block_lengths` as follows:

```
    block_lengths[0] = block_lengths[2] = 1;
    block_lengths[1] = 10;
```

Note that in this case, the first argument would still be 3 (not 12).

To summarize, then, we can build general derived datatypes by calling MPI_Type_struct. The syntax is

```
int MPI_Type_struct(
        int             count               /* in  */,
        int             block_lengths[]     /* in  */,
        MPI_Aint        displacements[]     /* in  */,
        MPI_Datatype    typelist[]          /* in  */,
        MPI_Datatype*   new_mpi_t           /* out */);
```

The parameter count is the number of blocks of elements in the derived type. It is also the size of the three arrays, block_lengths, displacements, and typelist. The array block_lengths contains the number of entries in each element of the type. So if an element of the type is an array of m values, then the corresponding entry in block_lengths is m. The array displacements contains the displacement of each element from the beginning of the message, and the array typelist contains the MPI datatype of each entry. The parameter new_mpi_t returns a pointer to the MPI datatype created by the call to MPI_Type_struct.

A few observations are in order. Note that the type of displacements is MPI_Aint—not int. This is a special C type in MPI. It allows for the possibility that addresses are too large to be stored in an int. Note also that new_mpi_t and the entries in typelist all have type MPI_Datatype. So MPI_Type_struct can be called recursively to build more complex derived datatypes.

In order to compute addresses, we used the function

```
MPI_Address(
    void*       location  /* in  */
    MPI_Aint*   address   /* out */)
```

It returns the byte address of location in address. We use it instead of C's & operator to insure portability. Although many implementations of C allow arithmetic on pointers, it is technically legal to do this only when the pointers refer to elements of the same array.

After the call to MPI_Type_struct, we can't use new_mpi_t in communication functions until we call MPI_Type_commit. Its syntax is simply

```
int MPI_Type_commit(
        MPI_Datatype*  new_mpi_t  /* in/out */)
```

This is a mechanism for the system to make internal changes in the representation of new_mpi_t that may improve the communication performance. These changes won't be needed if the new type is only going to be used as a building block for another, more complex type. Hence MPI makes it a separate function.

6.3 Other Derived Datatype Constructors

MPI_Type_struct is the most general datatype constructor in MPI, and as a consequence, the user must provide a *complete* description of each element of the type. If the data to be transmitted consists of a subset of the entries in an array, we shouldn't need to provide such detailed information since all the elements have the same basic type. MPI provides three derived datatype constructors for dealing with this situation: MPI_Type_contiguous, MPI_Type_vector, and MPI_Type_indexed. The first constructor builds a derived type whose elements are contiguous entries in an array. The second builds a type whose elements are equally spaced entries of an array, and the third builds a type whose elements are arbitrary entries of an array.

As an example, we'll use MPI_Type_vector to send a column of a two-dimensional array. So suppose that a program contains the following definition:

```
float A[10][10];
```

Recall that C stores two-dimensional arrays in *row-major* order. This means, for example, that in memory A[2][3] is preceded by A[2][2] and followed by A[2][4].[1] So if we wish to send, say, the third row of A from process 0 to process 1, we can simply use the following code:

```
if (my_rank == 0) {
    MPI_Send(&(A[2][0]), 10, MPI_FLOAT, 1, 0,
        MPI_COMM_WORLD);
} else { /* my_rank = 1 */
    MPI_Recv(&(A[2][0]), 10, MPI_FLOAT, 0, 0,
        MPI_COMM_WORLD, &status);
}
```

The reason this works is that the 10 memory locations starting at A[2][0] are A[2][0], A[2][1], A[2][2], . . . , A[2][9]—the third row of A.

If we wish to send the third column of A, this won't work, since A[0][2], A[1][2], . . . , A[9][2] aren't stored in contiguous memory locations. However, we can use MPI_Type_vector to create a derived datatype, since the displacement of successive elements of the derived type is constant—A[1][2] is displaced 10 floats beyond A[0][2], A[2][2] is 10 floats beyond A[1][2], etc. The syntax is

```
int MPI_Type_vector(
        int              count             /* in  */,
```

1 Fortran stores two-dimensional arrays in *column-major* order. So A(2,3) is preceded by A(1,3) and followed by A(3,3). Thus, an equivalent problem in Fortran would be sending a *row* of A.

```
            int             block_length   /* in  */,
            int             stride         /* in  */,
            MPI_Datatype    element_type   /* in  */,
            MPI_Datatype*   new_mpi_t      /* out */)
```

The parameter `count` is the number of elements in the type. `Block_length` is the number of entries in each element. `Stride` is the number of elements of type `element_type` between successive elements of `new_mpi_t`. `Element_type` is the type of the elements composing the derived type, and `new_mpi_t` is the MPI type of the new derived type.

So in order to send the third column of A from process 0 to process 1, we can use the following code:

```
/* column_mpi_t is declared to have type */
/* MPI_Datatype                          */
MPI_Type_vector(10, 1, 10, MPI_FLOAT, &column_mpi_t);
MPI_Type_commit(&column_mpi_t);
if (my_rank == 0)
    MPI_Send(&(A[0][2]), 1, column_mpi_t, 1, 0,
        MPI_COMM_WORLD);
else
    MPI_Recv(&(A[0][2]), 1, column_mpi_t, 0, 0,
        MPI_COMM_WORLD, &status);
```

Note that `column_mpi_t` can be used to send any column of A. If we want to send the jth column of A, we simply call the communication routine with first argument `&(A[0][j])`. Also note that in fact `column_mpi_t` can be used to send any column of any 10×10 matrix of floats, since the stride and element type will be the same.

This last point is important. In general, it is fairly expensive to build a derived datatype. So applications that make use of derived datatypes typically use the types many times.

The syntax for the other two constructors is

```
int MPI_Type_contiguous(
        int             count      /* in  */,
        MPI_Datatype    old_type   /* in  */,
        MPI_Datatype*   new_mpi_t  /* out */)

int MPI_Type_indexed(
        int             count,
        int             block_lengths[],
        int             displacements[],
        MPI_Datatype    old_type,
        MPI_Datatype*   new_mpi_t)
```

In `MPI_Type_contiguous`, one simply specifies that the derived type `new_mpi_t` will consist of `count` contiguous elements, each of which has type old_type. In MPI_Type_indexed, the derived type consists of count elements of type old_type. The ith element consists of block_lengths[i] entries, and it is displaced displacements[i] units of old_type from the beginning (displacement 0) of the type. Note that displacements are not measured in bytes.[2]

As an example of the use of MPI_Type_indexed, let's send the upper triangular portion of a square matrix stored on process 0 to process 1:

```
float          A[n][n];           /* Complete Matrix */
float          T[n][n];           /* Upper Triangle  */
int            displacements[n];
int            block_lengths[n];
MPI_Datatype   index_mpi_t;

for (i = 0; i < n; i++) {
    block_lengths[i] = n-i;
    displacements[i] = (n+1)*i;
}

MPI_Type_indexed(n, block_lengths, displacements,
    MPI_FLOAT, &index_mpi_t);
MPI_Type_commit(&index_mpi_t);

if (my_rank == 0)
    MPI_Send(A, 1, index_mpi_t, 1, 0, MPI_COMM_WORLD);
else /* my_rank == 1 */
    MPI_Recv(T, 1, index_mpi_t, 0, 0, MPI_COMM_WORLD,
        &status);
```

Note that even though the blocks are uniformly spaced in memory ($n + 1$ floats apart), we couldn't use MPI_Type_vector here because the block lengths are different for each row.

6.4 Type Matching

At this point it is natural to ask, What are the rules for matching MPI datatypes? For example, suppose a program contains the following code:

```
if (my_rank == 0)
    MPI_Send(message, send_count, send_mpi_t, 1, 0,
            MPI_COMM_WORLD);
else if (my_rank == 1)
    MPI_Recv(message, recv_count, recv_mpi_t, 0, 0,
            MPI_COMM_WORLD, &status);
```

2 MPI does provide functions where the stride or displacements are measured in bytes: MPI_Type_hvector and MPI_Type_hindexed. For details see [28, 29].

Must `send_mpi_t` be identical to `recv_mpi_t`? What about `send_count` and `recv_count`?

In order to answer this question, recall that a derived datatype is a sequence of pairs. The first element of a pair is a basic MPI type; the second, a displacement. That is, a general datatype has the form

$$\{(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1})\},$$

where each t_i is a basic MPI type and each d_i is a displacement in bytes. The sequence of basic types,

$$\{t_0, t_1, \ldots, t_{n-1}\},$$

is called the **type signature** of the type. The fundamental rule for type matching in MPI is that the type signatures specified by the sender and the receiver must be compatible. That is, suppose the type signature specified by the arguments passed to `MPI_Send` is

$$\{t_0, t_1, \ldots, t_{n-1}\},$$

and the type signature specified by the arguments to `MPI_Recv` is

$$\{u_0, u_1, \ldots, u_{m-1}\}.$$

Then n must be less than or equal to m and t_i must equal u_i for $i = 0, \ldots, n-1$. So displacements do not affect type matching.

In order to fully understand this rule, keep in mind that if, for example, `send_count` is greater than 1, then the type signature is obtained by simply concatenating `send_count` copies of the type signature of `send_mpi_t`.

Also keep in mind that for collective communication functions (unlike `MPI_Send` and `MPI_Recv`), the type signatures specified by all the processes must be *identical.*

Let's take a look at a short example. Recall that in section 6.3 we created a type `column_mpi_t` that corresponded to a column of a 10×10 array of floats. Thus the type is

```
{(MPI_FLOAT, 0), (MPI_FLOAT, 10*sizeof(float)),
     (MPI_FLOAT,20*sizeof(float)), . . . ,
     (MPI_FLOAT,90*sizeof(float))}
```

and its type signature is

```
{MPI_FLOAT, MPI_FLOAT, . . . , MPI_FLOAT}
```

(repeated 10 times). So if we use `MPI_Send` to send a message consisting of one copy of `column_mpi_t`, it can be received by a call to `MPI_Recv` provided the type signature specified by the receive consists of at least 10 floats. Thus, we can receive a column of a 10×10 matrix into a row of a 10×10 matrix as follows:

```
float A[10][10];

if (my_rank == 0)
    MPI_Send(&(A[0][0]), 1, column_mpi_t, 1, 0,
             MPI_COMM_WORLD);
else if (my_rank == 1)
    MPI_Recv(&(A[0][0]), 10, MPI_FLOAT, 0, 0,
             MPI_COMM_WORLD, &status);
```

This will send the first column of the matrix A on process 0 to the first row of the matrix A on process 1.

6.5 Pack/Unpack

An alternative approach to grouping data is provided by the MPI functions `MPI_Pack` and `MPI_Unpack`. `MPI_Pack` allows one to explicitly store noncontiguous data in contiguous memory locations, and `MPI_Unpack` can be used to copy data from a contiguous buffer into noncontiguous memory locations. In order to see how they are used, let's rewrite `Get_data` one last time.

```
void Get_data4(
         float*  a_ptr    /* out */,
         float*  b_ptr    /* out */,
         int*    n_ptr    /* out */,
         int     my_rank  /* in  */) {

    char  buffer[100];  /* Store data in buffer        */
    int   position;     /* Keep track of where data is */
                        /*     in the buffer           */

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);

        /* Now pack the data into buffer.  Position = 0 */
        /* says start at beginning of buffer.           */
        position = 0;

        /* Position is in/out */
        MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100,
            &position, MPI_COMM_WORLD);
        /* Position has been incremented: it now refer- */
        /* ences the first free location in buffer.     */

        MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100,
            &position, MPI_COMM_WORLD);
        /* Position has been incremented again. */
```

```
            MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100,
                &position, MPI_COMM_WORLD);
            /* Position has been incremented again. */

            /* Now broadcast contents of buffer */
            MPI_Bcast(buffer, 100, MPI_PACKED, 0,
                MPI_COMM_WORLD);
        } else {
            MPI_Bcast(buffer, 100, MPI_PACKED, 0,
                MPI_COMM_WORLD);

            /* Now unpack the contents of buffer */
            position = 0;
            MPI_Unpack(buffer, 100, &position, a_ptr, 1,
                MPI_FLOAT, MPI_COMM_WORLD);
            /* Once again position has been incremented: */
            /* it now references the beginning of b.     */

            MPI_Unpack(buffer, 100, &position, b_ptr, 1,
                MPI_FLOAT, MPI_COMM_WORLD);
            MPI_Unpack(buffer, 100, &position, n_ptr, 1,
                MPI_INT, MPI_COMM_WORLD);
        }
    } /* Get_data4 */
```

In this version of `Get_data`, process 0 uses `MPI_Pack` to copy `a` to `buffer` and then successively append `b` and `n`. After the broadcast of `buffer`, the remaining processes use `MPI_Unpack` to successively extract `a`, `b`, and `n` from `buffer`. Note that the datatype for the calls to `MPI_Bcast` is `MPI_PACKED`.

The syntax of `MPI_Pack` is

```
int MPI_Pack(
        void*          pack_data      /* in      */,
        int            in_count       /* in      */,
        MPI_Datatype   datatype       /* in      */,
        void*          buffer         /* out     */,
        int            buffer_size    /* in      */,
        int*           position       /* in/out */,
        MPI_Comm       comm           /* in      */)
```

The parameter `pack_data` references the data to be buffered. It should consist of `in_count` elements, each having type `datatype`. The parameter `position` is an `in/out` parameter. On input, the data referenced by `pack_data` is copied into memory starting at address `buffer + *position`. On return, `*position` references the first location in `buffer` *after* the data that was copied. The parameter `buffer_size` contains the size in bytes of the memory referenced by `buffer`, and `comm` is the communicator that will be using `buffer`. See Figure 6.2.

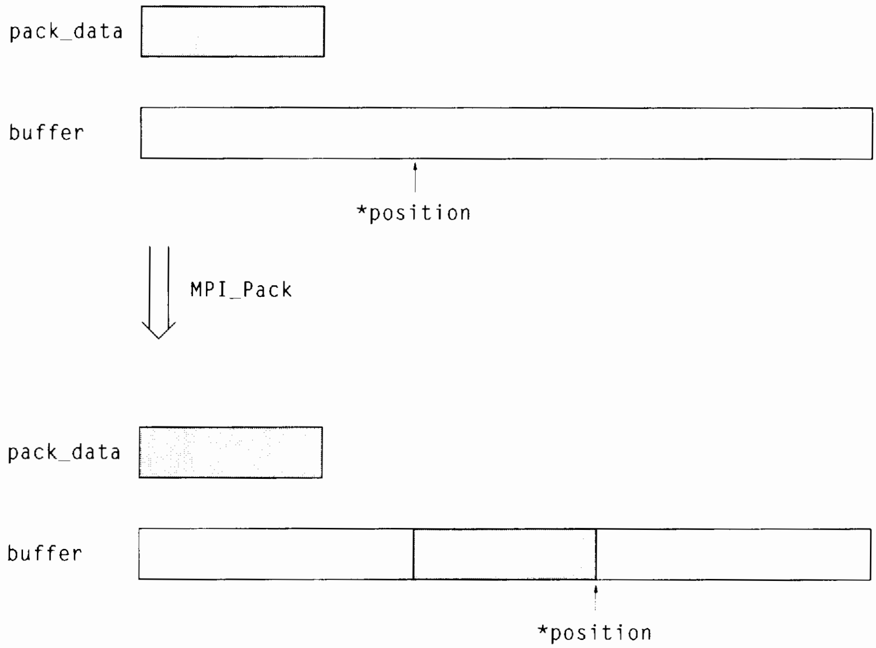

Figure 6.2 MPI_Pack

The syntax of MPI_Unpack is

```
int MPI_Unpack(
        void*         buffer        /* in     */,
        int           size          /* in     */,
        int*          position      /* in/out */,
        void*         unpack_data   /* out    */,
        int           count         /* in     */,
        MPI_Datatype  datatype      /* in     */,
        MPI_comm      comm          /* in     */)
```

The parameter buffer references the data to be unpacked. It consists of size bytes. The parameter position is once again an in/out parameter. When MPI_Unpack is called, the data starting at address buffer + *position is copied into the memory referenced by unpack_data. On return, *position references the first location in buffer after the data that was just copied. MPI_Unpack will copy count elements having type datatype into unpack_data. The communicator associated with buffer is comm. See Figure 6.3.

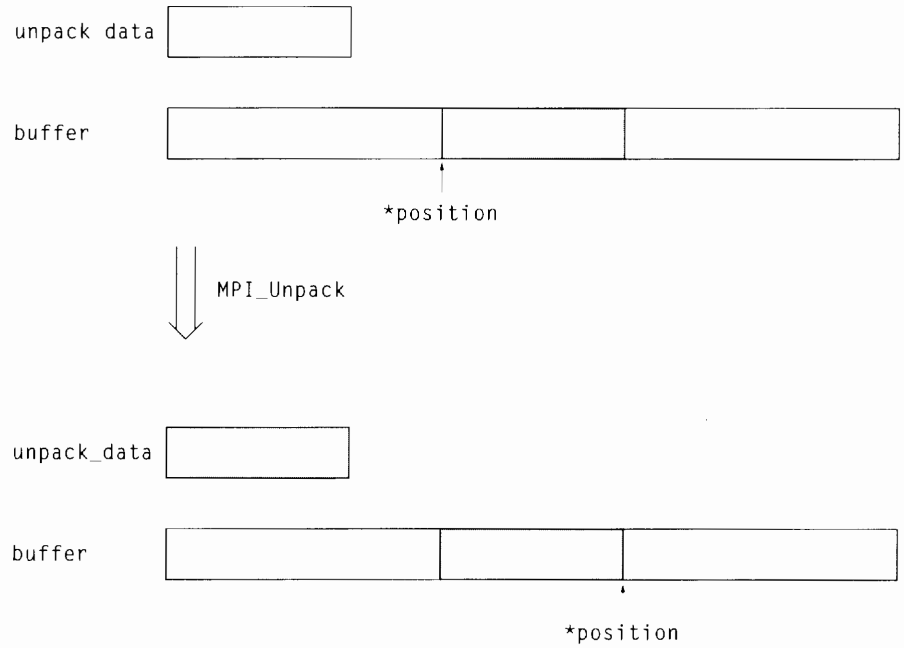

Figure 6.3 MPI_Unpack

6.6 Deciding Which Method to Use

If the data to be sent is stored in consecutive entries of an array, then one should simply use the count and datatype parameters to the communication function(s). This approach involves no additional overhead in the form of calls to derived datatype creation functions or calls to MPI_Pack/MPI_Unpack.

If there are a large number of elements that are not in contiguous memory locations, then building a derived type will probably involve less overhead than a large number of calls to MPI_Pack/MPI_Unpack.

If the data all have the same type and are stored at regular intervals in memory (e.g., a column of a matrix), then it will almost certainly be much easier and faster to use a derived datatype than it will be to use MPI_Pack/MPI_Unpack. Furthermore, if the data all have the same type, but are stored in irregularly spaced locations in memory, it will still probably be easier and more efficient to create a derived type using MPI_Type_indexed. Finally, if the data are heterogeneous and you are repeatedly sending the same collection of data (e.g., row number, column number, matrix entry), then it will be better to use a derived type, since the overhead of creating the derived type is incurred only once, while the overhead of calling MPI_Pack/MPI_Unpack must be incurred every time the data is communicated.

This leaves the case where you are sending heterogeneous data only once, or very few times. In this case, it may be a good idea to collect some data on the cost of derived type creation and packing/unpacking the data. For example, on an nCUBE 2 running the `mpich` implementation of MPI, it takes about 12 milliseconds to create the derived type used in `Get_data3`, while it only takes about 2 milliseconds to pack or unpack the data in `Get_data4`. Of course, the saving isn't as great as it seems because of the asymmetry in the pack/unpack procedure. That is, while process 0 packs the data, the other processes are idle, and the entire function won't complete until both the pack and unpack are executed. So the cost ratio is probably more like 3:1 than 6:1.

There are also a couple of situations in which the use of `MPI_Pack`/`MPI_Unpack` is preferred. Note first that it may be possible to avoid the use of *system* buffering with pack, since the data is explicitly stored in a user-defined buffer. The system can exploit this by noting that the message datatype is `MPI_PACKED`. Also note that the user can send "variable length" messages by packing the number of elements at the beginning of the buffer. For example, suppose we want to send rows of a sparse matrix. If we have stored each row as a pair of arrays—one containing the column subscripts, and one containing the corresponding matrix entries—we could send a row from process 0 to process 1 as follows:

```
float*      entries;
int*        column_subscripts;
int         nonzeroes;
int         position;
int         row_number;
char        buffer[HUGE]; /* HUGE is a constant     */
                          /* defined in the program */
MPI_Status  status;
    .
    .
    .
if (my_rank == 0) {
    /* Get the number of nonzeroes in the row.  */
    /* Allocate storage for the row.            */
    /* Initialize entries and column_subscripts */
        .
        .
        .
    /* Now pack the data and send */
    position = 0;
    MPI_Pack(&nonzeroes, 1, MPI_INT, buffer, HUGE,
        &position, MPI_COMM_WORLD);
    MPI_Pack(&row_number, 1, MPI_INT, buffer, HUGE,
        &position, MPI_COMM_WORLD);
    MPI_Pack(entries, nonzeroes, MPI_FLOAT, buffer,
        HUGE, &position, MPI_COMM_WORLD);
    MPI_Pack(column_subscripts, nonzeroes, MPI_INT,
        buffer, HUGE, &position, MPI_COMM_WORLD);
    MPI_Send(buffer, position, MPI_PACKED, 1, 0,
        MPI_COMM_WORLD);
} else { /* my_rank == 1 */
```

```
        MPI_Recv(buffer, HUGE, MPI_PACKED, 0, 0,
            MPI_COMM_WORLD, &status);
        position = 0;
        MPI_Unpack(buffer, HUGE, &position, &nonzeroes,
            1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, HUGE, &position, &row_number,
            1, MPI_INT, MPI_COMM_WORLD);
        /* Allocate storage for entries */
        /* and column_subscripts       */
        entries = (float *) malloc(nonzeroes*sizeof(float));
        column_subscripts =
            (int *) malloc(nonzeroes*sizeof(int));
        MPI_Unpack(buffer,HUGE, &position, entries,
            nonzeroes, MPI_FLOAT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, HUGE, &position,
            column_subscripts, nonzeroes, MPI_INT,
            MPI_COMM_WORLD);
    }
```

6.7 Summary

MPI provides three methods for sending messages consisting of more than one scalar element. The simplest method can be used for sending consecutive entries in arrays: call the appropriate communication function with the `count` parameter set equal to the number of entries to be sent and the `datatype` parameter set equal to the basic type of the array elements. For more complex messages, one can either build a *derived* datatype, or one can use the two functions `MPI_Pack` and `MPI_Unpack`.

A derived datatype is essentially a struct that is built *during execution* of the program and can be passed as the datatype argument to MPI communication functions. In order to build one, the user must specify

1. the number of elements in the type
2. the types of the elements
3. the relative locations, or *displacements*, of the elements in memory

MPI provides a number of functions for building derived types. The simplest to use are `MPI_Type_contiguous` and `MPI_Type_vector`. The first can be used to construct a type containing a subset of consecutive entries in an array. The second can be used to construct a type consisting of array elements that are uniformly spaced in memory. `MPI_Type_indexed` can be used to construct a type consisting of array elements that may not be uniformly spaced in memory. The most general constructor is `MPI_Type_struct`. It can be used to build derived types whose elements have different types and arbitrary locations in memory. Their syntax is

```
int MPI_Type_contiguous(
        int             count           /* in  */,
        MPI_Datatype    old_type        /* in  */,
        MPI_Datatype*   new_mpi_t       /* out */)

int MPI_Type_vector(
        int             count           /* in  */,
        int             block_length    /* in  */,
        int             stride          /* in  */,
        MPI_Datatype    element_type    /* in  */,
        MPI_Datatype*   new_mpi_t       /* out */)

int MPI_Type_indexed(
        int             count           /* in  */,
        int             block_lengths[] /* in  */,
        int             displacements[] /* in  */,
        MPI_Datatype    old_type        /* in  */,
        MPI_Datatype*   new_mpi_t       /* out */)

int MPI_Type_struct(
        int             count           /* in  */,
        int             block_lengths[] /* in  */,
        MPI_Aint        displacements[] /* in  */,
        MPI_Datatype    typelist[]      /* in  */,
        MPI_Datatype*   new_mpi_t       /* out */);
```

Before a derived type can be used by a comunication function, it must be *committed* with a call to MPI_Type_commit. Its syntax is simply

```
int MPI_Type_commit(
        MPI_Datatype*   new_mpi_t   /* in/out */)
```

Formally, a derived datatype is a sequence of pairs:

$$\{(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1})\}.$$

The first element of each pair is a basic MPI datatype—MPI_INT, MPI_CHAR, etc. The second element is a displacement in bytes. The *type signature* is just the sequence of types specified by a derived datatype:

$$\{t_0, t_1, \ldots, t_{n-1}\}.$$

In order for a message to be received, the type signatures specified by the sender and the receiver must be compatible. Suppose the type signature specified by the sender is

$$\{s_0, s_1, \ldots, s_{m-1}\}.$$

and the type signature specified by the receiver is

$$\{t_0, t_1, \ldots, t_{n-1}\}.$$

Then if the communication is carried out using `MPI_Send` and `MPI_Recv`, m must be less than or equal to n, and s_i must be the same as t_i for $i = 0, 1, \ldots,$ $m - 1$. If the communication is carried out using a collective communication function (`MPI_Bcast`, `MPI_Reduce`, etc.), then the type signatures must be identical.

`MPI_Pack` can be used to explicitly store data in a *user-defined* buffer. `MPI_Unpack` can be used to extract data from a buffer that was constructed using `MPI_Pack`. Messages that have been constructed using `MPI_Pack` should be communicated with datatype argument `MPI_PACKED`. Their syntax is

```
int MPI_Pack(
        void*          pack_data      /* in      */,
        int            in_count       /* in      */,
        MPI_Datatype   datatype       /* in      */,
        void*          buffer         /* out     */,
        int            buffer_size    /* in      */,
        int*           position       /* in/out */,
        MPI_Comm       comm           /* in      */)

int MPI_Unpack(
        void*          buffer        /* in      */,
        int            size          /* in      */,
        int*           position      /* in/out */,
        void*          unpack_data   /* out     */,
        int            count         /* in      */,
        MPI_Datatype   datatype      /* in      */,
        MPI_comm       comm          /* in      */)
```

In general, if a message consists of an array of scalar types, it's a good idea to just use the `count` and `datatype` parameters to the communications routines. For more complicated messages, it's usually better to use derived types. The most important exceptions are the following:

1. The type would only be used a very few times, and the overhead associated with building the derived type is greater than the overhead associated with using pack and unpack.
2. You wish to buffer messages in user memory instead of system memory.
3. You wish to specify *in the message* the number of items it contains.

6.8 References

Details on rules for using derived datatypes and `MPI_Pack`/`MPI_Unpack` can be found in both the MPI Standard [28, 29] and [34]. Examples of their use can be found in these references and [21]. A discussion of legal operations on pointers can be found in [24].

6.9 Exercises

1. Edit the trapezoidal rule program so that it uses Get_data3.
2. Edit the trapezoidal rule program so that it uses Get_data4.
3. Write a function that creates a derived type representing a sparse matrix entry. A matrix entry is a struct consisting of a float and two ints. The ints represent the row and column number of an entry whose value is given by the float. Test your derived type by using it in a short program that sends a matrix entry from one process to another.
4. In view of the type matching rule (section 6.4), it's possible to have many different types specified by a sender correspond to a given type specified by the receiver. Consider the following definitions:

```
float         B[5][5];
float         x[5];
MPI_Datatype  first_mpi_t;
MPI_Datatype  second_mpi_t;
MPI_Datatype  third_mpi_t;
int           blocklengths[5] = {1,1,1,1,1};
int           displacements[5];

MPI_Type_contiguous(5, MPI_FLOAT, &first_mpi_t);
MPI_Type_vector(5, 1, 5, MPI_FLOAT,
                    &second_mpi_t);
for (i = 0; i < 5; i++)
    displacements[i] = 6*i;
MPI_Type_indexed(5, block_lengths, displacements,
                     MPI_FLOAT, &third_mpi_t);
```

Suppose a program contains these definitions, and the following sends and receives (in no particular order):

```
Process 0:
    MPI_Send(x, 5, MPI_FLOAT, 1, 0,
             MPI_COMM_WORLD);
    MPI_Send(&(B[1][0]), 5, MPI_FLOAT, 1, 0,
             MPI_COMM_WORLD);
    MPI_Send(x, 1, first_mpi_t, 1, 0,
             MPI_COMM_WORLD);
    MPI_Send(&(B[0][3]), 1, second_mpi_t, 1, 0,
             MPI_COMM_WORLD);
    MPI_Send(&(B[0][0]), 1, third_mpi_t, 1, 0,
             MPI_COMM_WORLD);
Process 1:
    MPI_Recv(x, 5, MPI_FLOAT, 0, 0,
             MPI_COMM_WORLD, &status);
```

```
MPI_Recv(&(B[1][0]), 5, MPI_FLOAT, 0, 0,
        MPI_COMM_WORLD, &status);
MPI_Recv(x, 1, first_mpi_t, 0, 0,
        MPI_COMM_WORLD, &status);
MPI_Recv(&(B[0][1]), 1, second_mpi_t, 0, 0,
        MPI_COMM_WORLD, &status);
MPI_Recv(&(B[0][0]), 1, third_mpi_t, 0, 0,
        MPI_COMM_WORLD, &status);
```

Briefly describe the memory on process 0 and process 1 referenced by each send/receive (e.g., "first row of B, second column of B, x"). Which receives could match which sends?

6.10 Programming Assignments

1. We can use derived datatypes to write functions for (dense) matrix I/O when we store the matrix by block *columns*.
   a. Write a function that prints a square matrix distributed by block columns among the processes. Suppose that the order of the matrix is n and the number of processes is p, and assume that n is evenly divisible by p. The function should successively gather blocks of n/p rows to process 0, and process 0 should print each block of n/p rows immediately after it has been received. For each gather of n/p rows, each process should send (using `MPI_Send`) a block of order $n/p \times n/p$ to process 0. Process 0 should carry out the gather using a sequence of calls to `MPI_Recv`. The datatype argument should be a derived datatype created with `MPI_Type_vector`. (Although it may be tempting to use `MPI_Gather` for this function, there are some technical problems this introduces that we aren't quite ready to deal with. See section 8.4 for details.)
   b. Write a function that reads in a square matrix stored in row-major order in a single file. Process 0 should read in the number of rows and broadcast this information to the other processes. Assume that n, the number of rows, is evenly divisible by p, the number of processes. Process 0 should then read in a block of n/p rows and distribute blocks of n/p columns to each of the processes: the first n/p columns go to 0, the next n/p to 1, etc. Process 0 should then repeat this process for each block of n/p rows. Use a derived type created with `MPI_Type_vector` so that the data sent to each process can be sent with a single call to `MPI_Send`. (Use `MPI_Send` and `MPI_Recv` rather than `MPI_Scatter`. See section 8.4 for details.)
2. Use your matrix I/O functions in a matrix-vector multiplication program. Read and distribute the coefficient matrix and the vector. Multiply them and print the result.

3. Write a dense matrix transpose function: Suppose a dense $n \times n$ matrix A is stored on process 0. Create a derived datatype representing a single column of A. Send each column of A to process 1, but have process 1 receive each column into a row. When the function returns, A should be stored on process 0 and A^{T} on process 1.

4. Repeat the preceding exercise for a sparse matrix. Suppose that a sparse matrix has been stored as an array of rows. Each row is represented by a struct consisting of three members: the number of nonzero entries in the row, the entries in the row, and the column numbers of the entries in the row. Write a function that identifies the entries in a column of the matrix. Also write a function that uses `MPI_Pack` to store the entries in a user-defined buffer, and a function that uses `MPI_Unpack` to extract the entries from the buffer and store them in the same fashion as a row.

   Use your functions in a program that stores the matrix A on process 0 and its transpose on process 1.