4 Grouping Data for Communication

With current generation machines sending a message is an expensive operation. So as a rule of thumb, the fewer messages sent, the better the overall performance of the program. However, in each of our trapezoid 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 the 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.

4.1 The Count Parameter

Recall that MPI_Send, MPI_Receive, MPI_Bcast, and MPI_Reduce all have a count and a datatype argument. 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 section 2, 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];
int tag, count, dest, source;
MPI_Status status;
int p;
int my_rank;
     ⋮
if (my_rank == 0) {
    /*  Initialize vector and send */
     ⋮
    tag = 47;
    count = 50;
```

```
        dest = 1;
        MPI_Send(vector + 50, count, MPI_FLOAT, dest, tag,
            MPI_COMM_WORLD);
    } else { /* my_rank == 1 */
        tag = 47;
        count = 50;
        source = 0;
        MPI_Recv(vector+50, count, MPI_FLOAT, source, tag,
            MPI_COMM_WORLD, &status);
    }
```

Unfortunately, this doesn't help us with the trapezoid 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 and it wouldn't address the fundamental issue. In order to solve the problem we need to use one of MPI's other facilities for grouping data.

4.2 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 — and try to use the datatype argument to MPI_Bcast. The difficulty here is that the type of datatype is MPI_Datatype, which is an actual type itself — not the same thing as a user-defined type in C. For example, suppose we included the type definition

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

and the variable definition

```
INDATA_TYPE indata
```

Now if we call MPI_Bcast

```
MPI_Bcast(&indata, 1, INDATA_TYPE, 0, MPI_COMM_WORLD)
```

the program will fail. The details depend on the implementation of MPI that you're using. If you have an ANSI C compiler, it will flag an error in the call to MPI_Bcast, since INDATA_TYPE does not have type MPI_Datatype. The problem here is that MPI is a *pre-existing* library of functions. That is, the MPI functions were written without knowledge of the datatypes that you define in your program. In particular, none of the MPI functions "knows" about INDATA_TYPE.

MPI provides a partial solution to this problem, by allowing the user to build MPI datatypes at execution time. In order to build an MPI datatype, one essentially specifies the layout of the data in the type — the member types and their relative locations in memory. Such a type is called a *derived datatype*. In order to see how this works, let's write a function that will build a derived type that corresponds to INDATA_TYPE.

```
void Build_derived_type(INDATA_TYPE* indata,
    MPI_Datatype* message_type_ptr){

    int block_lengths[3];
    MPI_Aint displacements[3];
    MPI_Aint addresses[4];
    MPI_Datatype typelist[3];

    /*  Build a derived datatype consisting of
     *  two floats and an int */

    /* First specify the types */
    typelist[0] = MPI_FLOAT;
    typelist[1] = MPI_FLOAT;
    typelist[2] = MPI_INT;

    /* Specify the number of elements of each type */
```

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

    /* Calculate the displacements of the members
     * relative to indata */
    MPI_Address(indata, &addresses[0]);
    MPI_Address(&(indata->a), &addresses[1]);
    MPI_Address(&(indata->b), &addresses[2]);
    MPI_Address(&(indata->n), &addresses[3]);
    displacements[0] = addresses[1] - addresses[0];
    displacements[1] = addresses[2] - addresses[0];
    displacements[2] = addresses[3] - addresses[0];

    /* Create the derived type */
    MPI_Type_struct(3, block_lengths, displacements, typelist,
         message_type_ptr);

    /* Commit it so that it can be used */
    MPI_Type_commit(message_type_ptr);
} /* Build_derived_type */
```

The first three statements specify the types of the members of the derived type, and the next specifies the number of elements of each type. The next four calculate the addresses of the three members of indata. The next three statements use the calculated addresses to determine the *displacements* of the three members relative to the address of the first — which is given displacement 0. With this information, we know the types, sizes and relative locations of the members of a variable having C type INDATA_TYPE, and hence we can define a derived data type that corresponds to the C type. This is done by calling the functions MPI_Type_struct and MPI_Type_commit.

The newly created MPI datatype can be used in any of the MPI communication functions. In order to use it, we simply use the starting address of a variable of type INDATA_TYPE as the first argument, and the derived type in the datatype argument. For example, we could rewrite the Get_data function as follows.

```
void Get_data3(INDATA_TYPE* indata, int my_rank){
    MPI_Datatype message_type;  /* Arguments to */
```

```
    int root = 0;                /* MPI_Bcast */
    int count = 1;

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d",
            &(indata->a), &(indata->b), &(indata->n));
    }

    Build_derived_type(indata, &message_type);
    MPI_Bcast(indata, count, message_type, root,
        MPI_COMM_WORLD);
} /* Get_data3 */
```

A few observations are in order. Note that we calculated the addresses of the members of indata with MPI_Address rather than C's & operator. The reason for this is that ANSI C does not require that a pointer be an int (although this is commonly the case). See [4], for a more detailed discussion of this point. Note also that the type of array_of_displacements is MPI_Aint — not int. This is a special type in MPI. It allows for the possibility that addresses are too large to be stored in an int.

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

```
    int MPI_Type_Struct(int count,
        int* array_of_block_lengths,
        MPI_Aint* array_of_displacements,
        MPI_Datatype* array_of_types,
        MPI_Datatype* newtype)
```

The argument count is the number of elements in the derived type. It is also the size of the three arrays, array_of_block_lengths, array_of_displacements, and array_of_types. The array array_of_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 array_of_block_lengths is m. The array array_of_displacements contains the displacement of each element from the beginning of the message, and the array array_of_types contains the MPI datatype of each entry. The argument newtype returns a pointer to the MPI datatype created by the call to MPI_Type_struct.

Note also that newtype and the entries in array_of_types all have type MPI_Datatype. So MPI_Type_struct can be called recursively to build more complex derived datatypes.

4.3 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. Note that before any derived type can be used in communication it must be *committed* with a call to MPI_Type_commit.

Details of the syntax of the additional type constructors follow.

- ```
  int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
      MPI_Datatype* newtype)
  ```

  MPI_Type_contiguous creates a derived datatype consisting of count elements of type oldtype. The elements belong to contiguous memory locations.

- ```
  int MPI_Type_vector(int count, int block_length,
          int stride, MPI_Datatype element_type,
          MPI_Datatype* newtype)
  ```

  MPI_Type_vector creates a derived type consisting of count elements. Each element contains block_length entries of type element_type. Stride is the number of elements of type element_type between successive elements of new_type.

- ```
  int MPI_Type_indexed(int count,
      int* array_of_block_lengths,
      int* array_of_displacements,
      MPI_Datatype element_type,
      MPI_Datatype* newtype)
  ```

  MPI_Type_indexed creates a derived type consisting of count elements. The ith element ($i = 0, 1, \ldots, \text{count} - 1$), consists of array_of_block_lengths[i] entries of type element_type, and it is displaced array_of_displacements[i] units of type element_type from the beginning of newtype.

4.4 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(int my_rank, float* a_ptr, float* b_ptr,
    int* n_ptr) {
    int root = 0;      /* Argument to MPI_Bcast */
    char buffer[100];  /* Arguments to MPI_Pack/Unpack */
    int position;      /* and MPI_Bcast*/

    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;  /* Start at beginning of buffer */
        MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100,
            &position, MPI_COMM_WORLD);
        /* Position has been incremented by */
        /* sizeof(float) bytes */
        MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100,
```
```

```
            &position, MPI_COMM_WORLD);
        MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100,
            &position, MPI_COMM_WORLD);

        /* Now broadcast contents of buffer */
        MPI_Bcast(buffer, 100, MPI_PACKED, root,
            MPI_COMM_WORLD);
    } else {
        MPI_Bcast(buffer, 100, MPI_PACKED, root,
            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 */
        /* by sizeof(float) bytes */
        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 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, int in_count,
    MPI_Datatype datatype, void* buffer,
    int size, int* position_ptr, MPI_Comm comm)
```

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

The syntax of MPI_Unpack is

```
int MPI_Unpack(void* buffer, int size,
    int* position_ptr, void* unpack_data, int count,
    MPI_Datatype datatype, MPI_comm comm)
```

The parameter buffer references the data to be unpacked. It consists of size bytes. The parameter position_ptr is once again an *in/out* parameter. When MPI_Unpack is called, the data starting at address buffer + *position_ptr is copied into the memory referenced by unpack_data. On return, *position_ptr 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.

4.5 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 arguments 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 one is 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 one is sending heterogeneous data only once, or very few times. In this case, it may be a good idea to collect some information 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 and 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 a 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;  /* number of nonzeroes in row */
int position;
int row_number;
char* buffer[HUGE]; /* HUGE is a predefined constant */
MPI_Status status;
      ⋮
if (my_rank == 0) {
    /* Get the number of nonzeros 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, 193,
        MPI_COMM_WORLD);
} else { /* my_rank == 1 */
    MPI_Recv(buffer, HUGE, MPI_PACKED, 0, 193,
        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);
}
```