### 3.12.3 LOWER-BOUND AND UPPER-BOUND MARKERS

It is often convenient to define explicitly the lower bound and upper bound of a type map, and override the definition given by Equation 3.1 on page 233. This allows one to define a datatype that has "holes" at its beginning or its end, or a datatype with entries that extend above the upper bound or below the lower bound. Examples of such usage are provided in Section 3.12.7. To achieve this, we add two additional "pseudo-datatypes," MPI_LB and MPI_UB, that can be used, respectively, to mark the lower bound or the upper bound of a datatype. These pseudo-datatypes occupy no space ($extent$(MPI_LB) = $extent$(MPI_UB) = 0). They do not affect the size or count of a datatype, and do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

**Example 3.25** Let D = (−3, 0, 6); T = (MPI_LB, MPI_INT, MPI_UB), and B = (1, 1, 1). Then a call to MPI_TYPE_STRUCT(3, B, D, T, type1) creates a new datatype that has an extent of 9 (from −3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the sequence {(lb, −3), (int, 0), (ub, 6)} . If this type is replicated twice by a call to MPI_TYPE_CONTIGUOUS(2, type1, type2) then the newly created type can be described by the sequence

{(lb, −3), (int, 0), (int,9), (ub, 15)} . (Entries of type lb or ub can be deleted if they are not at the end-points of the datatype.)

In general, if

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},$$

then the **lower bound** of $Typemap$ is defined to be

$$lb(Typemap) = \begin{cases} \min_j disp_j & \text{if no entry} \\ & \text{has basic type lb} \\ \min_j\{disp_j \text{ such that } type_j = \text{lb}\} & \text{otherwise} \end{cases}$$

Similarly, the **upper bound** of $Typemap$ is defined to be

$$ub(Typemap) = \begin{cases} \max_j disp_j + sizeof(type_j) & \text{if no entry} \\ & \text{has basic type ub} \\ \max_j\{disp_j \text{ such that } type_j = \text{ub}\} & \text{otherwise} \end{cases}$$

Then

$$extent(Typemap) = ub(Typemap) - lb(Typemap) + \epsilon$$

If $type_i$ requires alignment to a byte address that is a multiple of $k_i$, then $\epsilon$ is the least nonnegative increment needed to round $extent(Typemap)$ to the next multiple of $\max_i k_i$.

The formal definitions given for the various datatype constructors apply now, with the amended definition of **extent**.

The two functions below can be used for finding the lower bound and the upper bound of a datatype.

MPI_TYPE_LB( datatype, displacement)

| | | |
|---|---|---|
| IN | datatype | datatype (handle) |
| OUT | displacement | displacement of lower bound from origin, in bytes (integer) |

```
int MPI_Type_lb(MPI_Datatype datatype, int* MPI_Aint displacement)

MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

MPI_TYPE_UB( datatype, displacement)

| | | |
|---|---|---|
| IN | datatype | datatype (handle) |
| OUT | displacement | displacement of upper bound from origin, in bytes (integer) |

```
int MPI_Type_ub(MPI_Datatype datatype, int* MPI_Aint displacement)

MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

*Rationale.* Note that the rules given in Section 3.12.6 imply that it is erroneous to call MPI_TYPE_EXTENT, MPI_TYPE_LB, and MPI_TYPE_UB with a datatype argument that contains absolute addresses, unless all these addresses are within the same sequential storage. For this reason, the displacement for the C binding in MPI_TYPE_UB is an `int` and not of type `MPI_Aint`. (*End of rationale.*)

### 3.12.4 COMMIT AND FREE

A datatype object has to be **committed** before it can be used in a communication. A committed datatype can still be used as a argument in datatype constructors. There is no need to commit basic datatypes. They are "pre-committed."

MPI_TYPE_COMMIT(datatype)

| | | |
|---|---|---|
| INOUT | datatype | datatype that is committed (handle) |

```
int MPI_Type_commit(MPI_Datatype *datatype)

MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

*Advice to implementors.* The system may "compile" at commit time an internal representation for the datatype that facilitates communication, e.g., change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (*End of advice to implementors.*)

MPI_TYPE_FREE(datatype)

| | | |
|---|---|---|
| INOUT | datatype | datatype that is freed (handle) |

```
int MPI_Type_free(MPI_Datatype *datatype)

MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

Marks the datatype object associated with datatype for deallocation and sets datatype to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will complete normally. Derived datatypes that were defined from the freed datatype are not affected.

**Example 3.26** The following code fragment gives examples of using MPI_TYPE_COMMIT.

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
                ! new type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
                ! now type1 can be used for communication
type2 = type1
                ! type2 can be used for communication
                ! (it is a handle to same object as type1)
CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
                ! new uncommitted type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
                ! now type1 can be used anew for communication
```

Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

> *Advice to implementors.* The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather then copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (*End of advice to implementors.*)

### 3.12.5 USE OF GENERAL DATATYPES IN COMMUNICATION

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI_SEND(buf, count, datatype , ...), where count > 1, is interpreted as if the call was passed a new datatype which is the concatenation of count copies of datatype. Thus, MPI_SEND(buf, count, datatype, dest, tag, comm) is equivalent to,

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm).
```

Similar statements apply to all other communication functions that have a count and datatype argument.

Suppose that a send operation MPI_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

$$\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},$$

and extent *extent*. (Empty entries of "pseudo-type" MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) The send operation

sends $n \cdot$ count entries, where entry $i \cdot n + j$ is at location $addr_{i,j} =$ buf $+ extent \cdot i + disp_j$ and has type $type_j$, for $i = 0, \ldots,$ count $- 1$ and $j = 0, \ldots, n - 1$. These entries need not be contiguous, nor distinct; their order can be arbitrary.

The variable stored at address $addr_{i,j}$ in the calling program should be of a type that matches $type_j$, where type matching is defined as in Section 3.3.1. The message sent contains $n \cdot$ count entries, where entry $i \cdot n + j$ has type $type_j$.

Similarly, suppose that a receive operation MPI_RECV(buf, count, datatype, source, tag, comm, status) is executed, where datatype has type map,

$$\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},$$

with extent *extent*. (Again, empty entries of "pseudo-type" MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) This receive operation receives $n \cdot$ count entries, where entry $i \cdot n + j$ is at location buf $+ extent \cdot i + disp_j$ and has type $type_j$. If the incoming message consists of $k$ elements, then we must have $k \leq n \cdot$ count; the $i \cdot n + j$-th element of the message should have a type that matches $type_j$.

Type matching is defined according to the type signature of the corresponding datatypes, that is, the sequence of basic type components. Type matching does not depend on some aspects of the datatype definition, such as the displacements (layout in memory) or the intermediate types used.

**Example 3.27** This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
...
CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
...
CALL MPI_SEND( a, 4, MPI_REAL, ...)
CALL MPI_SEND( a, 2, type2, ...)
CALL MPI_SEND( a, 1, type22, ...)
CALL MPI_SEND( a, 1, type4, ...)
...
CALL MPI_RECV( a, 4, MPI_REAL, ...)
CALL MPI_RECV( a, 2, type2, ...)
CALL MPI_RECV( a, 1, type22, ...)
CALL MPI_RECV( a, 1, type4, ...)
```

Each of the sends matches any of the receives.

A datatype may specify overlapping entries. If such a datatype is used in a receive operation, that is, if some part of the receive buffer is written more than once by the receive operation, then the call is erroneous.

Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed, where datatype has type map,

$$\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}.$$

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of $n$. Any number, $k$, of basic elements can be received, where $0 \leq k \leq \text{count} \cdot n$. The number of basic elements received can be retrieved from status using the query function MPI_GET_ELEMENTS.

MPI_GET_ELEMENTS( status, datatype, count)

| | | |
|---|---|---|
| IN | status | return status of receive operation (Status) |
| IN | datatype | datatype used by receive operation (handle) |
| OUT | count | number of received basic elements (integer) |

```
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)

MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

The previously defined function, MPI_GET_COUNT (Section 3.2.5), has a different behavior. It returns the number of "top-level elements" received. In the previous example, MPI_GET_COUNT may return any integer value $k$, where $0 \leq k \leq$ count. If MPI_GET_COUNT returns $k$, then the number of basic elements received (and the value returned by MPI_GET_ELEMENTS) is $n \cdot k$. If the number of basic elements received is not a multiple of $n$, that is, if the receive operation has not received an integral number of datatype "copies," then MPI_GET_COUNT returns the value MPI_UNDEFINED.

**Example 3.28** Usage of MPI_GET_COUNT and MPI_GET_ELEMENT.

```
...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)
...
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
      CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)      ! returns i=1
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=2
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)      ! returns i=MPI_UNDEFINED
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=3
END IF
```

The function MPI_GET_ELEMENTS can also be used after a probe to find the number of elements in the probed message. Note that the two functions

MPI_GET_COUNT and MPI_GET_ELEMENTS return the same values when they are used with basic datatypes.

*Rationale.* The extension given to the definition of MPI_GET_COUNT seems natural: one would expect this function to return the value of the count argument, when the receive buffer is filled. Sometimes datatype represents a basic unit of data one wants to transfer, for example, a record in an array of records (structures). One should be able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI_GET_ELEMENTS. (*End of rationale.*)

*Advice to implementors.* The definition implies that a receive cannot change the value of storage outside the entries defined to compose the communication buffer. In particular, the definition implies that padding space in a structure should not be modified when such a structure is copied from one process to another. This would prevent the obvious optimization of copying the structure, together with the padding, as one contiguous block. The implementation is free to do this optimization when it does not impact the outcome of the computation. The user can "force" this optimization by explicitly including padding as part of the message. (*End of advice to implementors.*)

### 3.12.6 CORRECT USE OF ADDRESSES

Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, the use of **addresses**, that is, displacements relative to the start address MPI_BOTTOM, has to be restricted.

Variables belong to the same **sequential storage** if they belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are defined recursively as follows:

1. The function MPI_ADDRESS returns a valid address, when passed as argument a variable of the calling program.
2. The buf argument of a communication function evaluates to a valid address, when passed as argument a variable of the calling program.
3. If v is a valid address, and i is an integer, then v+i is a valid address, provided v and v+i are in the same sequential storage.
4. If v is a valid address then MPI_BOTTOM + v is a valid address.

A correct program uses only valid addresses to identify the locations of entries in communication buffers. Furthermore, if u and v are two valid addresses, then

the (integer) difference u − v can be computed only if both u and v are in the same sequential storage. No other arithmetic operations can be meaningfully executed on addresses.

The rules above impose no constraints on the use of derived datatypes, as long as they are used to define a communication buffer that is wholly contained within the same sequential storage. However, the construction of a communication buffer that contains variables that are not within the same sequential storage must obey certain restrictions. Basically, a communication buffer with variables that are not within the same sequential storage can be used only by specifying in the communication call buf = MPI_BOTTOM, count = 1, and using a datatype argument where all displacements are valid (absolute) addresses.

*Advice to users.* It is not expected that MPI implementations will be able to detect erroneous, "out of bound" displacements—unless those overflow the user address space—since the MPI call may not know the extent of the arrays and records in the host program. (*End of advice to users.*)

*Advice to implementors.* There is no need to distinguish (absolute) addresses and (relative) displacements on a machine with contiguous address space: MPI_BOTTOM is zero, and both addresses and displacements are integers. On machines where the distinction is required, addresses are recognized as expressions that involve MPI_BOTTOM. (*End of advice to implementors.*)

### 3.12.7 EXAMPLES

The following examples illustrate the use of derived datatypes.

**Example 3.29** Send and receive a section of a 3D array.

```
      REAL a(100,100,100), e(9,9,9)
      INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C      extract the section a(1:17:2, 3:11, 2:10)
C      and store it in e(:,:,:).

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for a 1D section
      CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)

C     create datatype for a 2D section
      CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)

C     create datatype for the entire section
```

```
      CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice, 1,
                             threeslice, ierr)

      CALL MPI_TYPE_COMMIT( threeslice, ierr)
      CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9,
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

**Example 3.30** Copy the (strictly) lower triangular part of a matrix.

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(100), blocklen(100), ltype, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     copy lower triangular part of array a
C     onto lower triangular part of array b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

C     compute start and size of each column
      DO i=1, 100
        disp(i) = 100*(i-1) + i
        block(i) = 100-i
      END DO

C     create datatype for lower triangular part
      CALL MPI_TYPE_INDEXED( 100, block, disp, MPI_REAL, ltype, ierr)

      CALL MPI_TYPE_COMMIT(ltype, ierr)
      CALL MPI_SENDRECV( a, 1, ltype, myrank, 0, b, 1,
                         ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

**Example 3.31** Transpose a matrix.

```
      REAL a(100,100), b(100,100)
      INTEGER row, xpose, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     transpose matrix a onto b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
```

```
C     create datatype for matrix in row-major order
      CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)

      CALL MPI_TYPE_COMMIT( xpose, ierr)

C     send matrix in row-major order and receive in column major order
      CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,
                 MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

**Example 3.32** Another approach to the transpose problem:

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(2), blocklen(2), type(2), row, row1, sizeofreal
      INTEGER  myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

C     transpose matrix a onto b

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for one row, with the extent of one real number
      disp(1) = 0
      disp(2) = sizeofreal
      type(1)  = row
      type(2)  = MPI_UB
      blocklen(1)  = 1
      blocklen(2)  = 1
      CALL MPI_TYPE_STRUCT( 2, blocklen, disp, type, row1, ierr)

      CALL MPI_TYPE_COMMIT( row1, ierr)

C     send 100 rows and receive in column major order
      CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
                 MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

**Example 3.33** We manipulate an array of structures.

```
struct Partstruct
   {
   int    class;  /* particle class */
   double d[6];   /* particle coordinates */
```

```
    char   b[7];   /* some additional information */
    };

struct Partstruct    particle[1000];

int                  i, dest, rank;
MPI_Comm     comm;


/* build datatype describing structure */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          blocklen[3] = {1, 6, 7};
MPI_Aint     disp[3];
int          base;

/* compute displacements of structure components */

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i <3; i++) disp[i] -= base;

MPI_Type_struct( 3, blocklen, disp, type, &Particletype);

   /* If compiler does padding in mysterious ways,
      the following may be safer */

MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
int          blocklen1[4] = {1, 6, 7, 1};
MPI_Aint     disp1[4];

/* compute displacements of structure components */

MPI_Address( particle, disp1);
MPI_Address( particle[0].d, disp1+1);
MPI_Address( particle[0].b, disp1+2);
MPI_Address( particle+1, disp1+3);
base = disp1[0];
for (i=0; i <4; i++) disp1[i] -= base;
```

```
/* build datatype describing structure */

MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);


            /* 4.1:
          send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);


            /* 4.2:
          send only the entries of class zero particles,
          preceded by the number of such entries */

MPI_Datatype Zparticles;   /* datatype describing all particles
                              with class zero (needs to be recomputed
                              if classes change) */
MPI_Datatype Ztype;

MPI_Aint     zdisp[1000];
int zblock[1000], j, k;
int zzblock[2] = {1,1};
MPI_Aint     zzdisp[2];
MPI_Datatype zztype[2];

/* compute displacements of class zero particles */
j = 0;
for(i=0; i < 1000; i++)
  if (particle[i].class==0)
     {
     zdisp[j] = i;
     zblock[j] = 1;
     j++;
     }

/* create datatype for class zero particles  */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);

/* prepend particle count */
MPI_Address(&j, zzdisp);
MPI_Address(particle, zzdisp+1);
zztype[0] = MPI_INT;
```

```
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);


      /* A probably more efficient way of defining Zparticles */

/* consecutive particles with index zero are handled as one block */
j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
    zdisp[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);


             /* 4.3:
        send the first two coordinates of all entries */

MPI_Datatype Allpairs;       /* datatype for all pairs of coordinates */

MPI_Aint sizeofentry;

MPI_Type_extent( Particletype, &sizeofentry);

    /* sizeofentry can also be computed by subtracting the address
       of particle[0] from the address of particle[1] */

MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
MPI_Type_commit( &Allpairs);
MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);

     /* an alternative solution to 4.3 */

MPI_Datatype Onepair;   /* datatype for one pair of coordinates, with
                           the extent of one particle entry */
MPI_Aint disp2[3];
```

```
MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int blocklen2[3] = {1, 2, 1};

MPI_Address( particle, disp2);
MPI_Address( particle[0].d, disp2+1);
MPI_Address( particle+1, disp2+2);
base = disp2[0];
for (i=0; i<2; i++) disp2[i] -= base;

MPI_Type_struct( 3, blocklen2, disp2, type2, &Onepair);
MPI_Type_commit( &Onepair);
MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);
```

**Example 3.34** The same manipulations as in the previous example, but use absolute addresses in datatypes.

```
struct Partstruct
   {
   int class;
   double d[6];
   char b[7];
   };

struct Partstruct particle[1000];

          /* build datatype describing first array entry */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          block[3] = {1, 6, 7};
MPI_Aint     disp[3];

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
MPI_Type_struct( 3, block, disp, type, &Particletype);

/* Particletype describes first array entry -- using absolute
   addresses */

                /* 5.1:
           send the entire array */

MPI_Type_commit( &Particletype);
```

```
MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);

                /* 5.2:
        send the entries of class zero,
        preceded by the number of such entries */

MPI_Datatype Zparticles, Ztype;

MPI_Aint zdisp[1000]
int zblock[1000], i, j, k;
int zzblock[2] = {1,1};
MPI_Datatype zztype[2];
MPI_Aint     zzdisp[2];

j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index = 0) ; k++);
    zdisp[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with class zero, using
   their absolute addresses*/

/* prepend particle count */
MPI_Address(&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
```

**Example 3.35** Handling of unions.

```
union {
   int    ival;
   float  fval;
     } u[1000]
```

```
int     utype;

/* All entries of u have identical type; variable
   utype keeps track of their current type */

MPI_Datatype   type[2];
int            blocklen[2] = {1,1};
MPI_Aint       disp[2];
MPI_Datatype   mpi_utype[2];
MPI_Aint       i,j;

/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */

MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-i;
type[1] = MPI_UB;

type[0] = MPI_INT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);

type[0] = MPI_FLOAT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);

for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);

/* actual communication */

MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
```

## 3.13 Pack and Unpack

Some existing communication libraries provide pack/unpack functions for sending noncontiguous data. In these, the user explicitly packs data into a contiguous buffer before sending it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are described in Section 3.12, allow one, in most cases, to avoid explicit packing and unpacking. The user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer. The pack/unpack routines are provided for compatibility with previous libraries. Also, they provide some functionality that is not otherwise available in MPI. For instance, a message can be received in several parts, where the receive operation done on a later part may depend on the con-

tent of a former part. Another use is that outgoing messages may be explicitly buffered in user supplied space, thus overriding the system buffering policy. Finally, the availability of pack and unpack operations facilitates the development of additional communication libraries layered on top of MPI.

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

| | | |
|---|---|---|
| IN | inbuf | input buffer start (choice) |
| IN | incount | number of input data items (integer) |
| IN | datatype | datatype of each input data item (handle) |
| OUT | outbuf | output buffer start (choice) |
| IN | outsize | output buffer size, in bytes (integer) |
| INOUT | position | current position in buffer, in bytes (integer) |
| IN | comm | communicator for packed message (handle) |

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
             int outsize, int *position, MPI_Comm comm)

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer space specified by outbuf and outsize. The input buffer can be any communication buffer allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were a communication buffer for a message of type MPI_PACKED).

The input value of position is the first location in the output buffer to be used for packing. position is incremented by the size of the packed message, and the output value of position is the first location in the output buffer following the locations occupied by the packed message. The comm argument is the communicator that will be subsequently used for sending the packed message.

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

| | | |
|---|---|---|
| IN | inbuf | input buffer start (choice) |
| IN | insize | size of input buffer, in bytes (integer) |
| INOUT | position | current position in bytes (integer) |
| OUT | outbuf | output buffer start (choice) |
| IN | outcount | number of items to be unpacked (integer) |
| IN | datatype | datatype of each output data item (handle) |
| IN | comm | communicator for packed message (handle) |

```
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
               int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

```
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
           IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
```

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the output buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of position is the first location in the output buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

> *Advice to users.* Note the difference between MPI_RECV and MPI_UNPACK: in MPI_RECV, the count argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In MPI_UNPACK, the count argument specifies the actual number of items that are unpacked; the "size" of the corresponding message is the increment in position. The reason for this change is that the "incoming message size" is not predetermined since the user decides how much to unpack; nor is it easy to determine the "message size" from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined *a priori*. (*End of advice to users.*)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or sscanf in C, for a similar function.)

Several messages can be successively packed into one **packing unit**. This is effected by several successive **related** calls to MPI_PACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for outbuf, outcount and comm. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the "concatenation" of the individual send buffers.

A packing unit can be sent using type MPI_PACKED. Any point-to-point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI_PACKED.

A message sent with any type (including MPI_PACKED) can be received using the type MPI_PACKED. Such a message can then be unpacked by calls to MPI_UNPACK.

A packing unit (or a message created by a regular, "typed" send) can be unpacked into several successive messages. This is effected by several successive related calls to MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for inbuf, insize, and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two packing units and then unpack the result as one packing unit; nor can one unpack a substring of a packing unit as a separate packing unit. Each packing unit, that was created by a related sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of related unpack calls.

> *Rationale.* The restriction on "atomic" packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment). (*End of rationale.*)

The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.

MPI_PACK_SIZE(incount, datatype, comm, size)

| | | |
|---|---|---|
| IN | incount | count argument to packing call (integer) |
| IN | datatype | datatype argument to packing call (handle) |
| IN | comm | communicator argument to packing call (handle) |
| OUT | size | upper bound on size of packed message, in bytes (integer) |

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
                  int *size)
```

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm).

> *Rationale.* The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). (*End of rationale.*)

**Example 3.36** An example using MPI_PACK.

```
int position, i, j, a[2];
char buff[1000];

....

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
   / * SENDER CODE */

  position = 0;
  MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else  /* RECEIVER CODE */
  MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD)

}
```

**Example 3.37** A elaborate example.

```
int position, i;
float a[1000];
char buff[1000]

....

MPI_Comm_rank(MPI_Comm_world, &myrank);
if (myrank == 0)
{
  / * SENDER CODE */

  int len[2];
  MPI_Aint disp[2];
  MPI_Datatype type[2], newtype;

  /* build datatype for i followed by a[0]...a[i-1] */

  len[0] = 1;
  len[1] = i;
  MPI_Address( &i, disp);
  MPI_Address( a, disp+1);
```

```
  type[0] = MPI_INT;
  type[1] = MPI_FLOAT;
  MPI_Type_struct( 2, len, disp, type, &newtype);
  MPI_Type_commit( &newtype);

  /* Pack i followed by a[0]...a[i-1]*/

  position = 0;
  MPI_Pack( MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);

  /* Send */

  MPI_Send( buff, position, MPI_PACKED, 1, 0,
            MPI_COMM_WORLD)

/* *****
   One can replace the last three lines with
   MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   ***** */
}
else /* myrank == 1 */
{
   /* RECEIVER CODE */

  MPI_Status status;

  /* Receive */

  MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, &status);

  /* Unpack i */

 position = 0;
 MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);

 /* Unpack a[0]...a[i-1] */
 MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
```

**Example 3.38** Each process sends a count, followed by count characters to the root; the root concatenates all characters into one string.

```
int count, gsize, counts[64], totalcount, k1, k2, k,
    displs[64], position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;
```

```
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

      /* allocate local pack buffer */
MPI_Pack_size(1, MPI_INT, comm, &k1);
MPI_Pack_size(count, MPI_CHAR, &k2);
k = k1+k2;
lbuf = (char *)malloc(k);

      /* pack count, followed by count characters */
position = 0;
MPI_Pack(&count, 1, MPI_INT, &lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, &lbuf, k, &position, comm);

if (myrank != root)
      /* gather at root sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, NULL, NULL,
               NULL, root, comm);

      /* gather at root packed messages */
  MPI_Gatherv( &buf, position, MPI_PACKED, NULL,
            NULL, NULL, NULL, root, comm);

else {   /* root code */
      /* gather sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, counts, 1,
               MPI_INT, root, comm);

      /* gather all packed messages */
  displs[0] = 0;
  for (i=1; i < gsize; i++)
    displs[i] = displs[i-1] + counts[i-1];
  totalcount = dipls[gsize-1] + counts[gsize-1];
  rbuf = (char *)malloc(totalcount);
  cbuf = (char *)malloc(totalcount);
  MPI_Gatherv( lbuf, position, MPI_PACKED, rbuf,
            counts, displs, MPI_PACKED, root, comm);

      /* unpack all messages and concatenate strings */
  concat_pos = 0;
  for (i=0; i < gsize; i++) {
    position = 0;
    MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
```

```
        &position, &count, 1, MPI_INT, comm);
    MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
        &position, cbuf+concat_pos, count, MPI_CHAR, comm);
    concat_pos += count;
    }
cbuf[concat_pos] = '\0';
}
```