D R A F T
Document for a Standard Message-Passing Interface

Message Passing Interface Forum

July 20, 2009

This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 4

Datatypes

Basic datatypes were introduced in Section 3.2.2 Message Data on page 27 and in Section 3.3 Data Type Matching and Data Conversion on page 34. In this chapter, this model is extended to describe any data layout. We consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

4.1 Derived Datatypes

Up to here, all point to point communication have involved only buffers containing a sequence of identical basic datatypes. This is too constraining on two accounts. One often wants to pass messages that contain values with different datatypes (e.g., an integer count, followed by a sequence of real numbers); and one often wants to send noncontiguous data (e.g., a sub-block of a matrix). One solution is to pack noncontiguous data into a contiguous buffer at the sender site and unpack it at the receiver site. This has the disadvantage of requiring additional memory-to-memory copy operations at both sites, even when the communication subsystem has scatter-gather capabilities. Instead, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shape and size. It is not assumed that the MPI library is cognizant of the objects declared in the host language. Thus, if one wants to transfer a structure, or an array section, it will be necessary to provide in MPI a definition of a communication buffer that mimics the definition of the structure or array section in question. These facilities can be used by library designers to define communication functions that can transfer objects defined in the host language — by decoding their definitions as available in a symbol table or a dope vector. Such higher-level communication functions are not part of MPI.

More general communication buffers are specified by replacing the basic datatypes that have been used so far with derived datatypes that are constructed from basic datatypes using the constructors described in this section. These methods of constructing derived datatypes can be applied recursively.

A **general datatype** is an opaque object that specifies two things:

- A sequence of basic datatypes

- A sequence of integer (byte) displacements

The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. We call such a pair of sequences (or sequence of pairs) a **type map**. The sequence of basic datatypes (displacements ignored) is the **type signature** of the datatype.

Let

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

be such a type map, where $type_i$ are basic types, and $disp_i$ are displacements. Let

$$Typesig = \{type_0, ..., type_{n-1}\}$$

be the associated type signature. This type map, together with a base address *buf*, specifies a communication buffer: the communication buffer that consists of n entries, where the i-th entry is at address $buf + disp_i$ and has type $type_i$. A message assembled from such a communication buffer will consist of n values, of the types defined by $Typesig$.

Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation MPI_SEND(buf, 1, datatype,...) will use the send buffer defined by the base address buf and the general datatype associated with datatype; it will generate a message with the type signature determined by the datatype argument. MPI_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base address buf and the general datatype associated with datatype.

General datatypes can be used in all send and receive operations. We discuss, in Section 4.1.11, the case where the second argument count has value > 1.

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, MPI_INT is a predefined handle to a datatype with type map $\{(\mathsf{int}, 0)\}$, with one entry of type int and displacement zero. The other basic datatypes are similar.

The **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

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

then

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + sizeof(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{4.1}$$

If $type_i$ requires alignment to a byte address that is a multiple of k_i, then ϵ is the least non-negative increment needed to round $extent(Typemap)$ to the next multiple of $\max_i k_i$. The complete definition of **extent** is given on page 20.

Example 4.1 Assume that $Type = \{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$ (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

> *Rationale.* The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. (*End of rationale.*)

4.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI_TYPE_CREATE_HVECTOR, MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_STRUCT, and MPI_GET_ADDRESS accept arguments of type `INTEGER(KIND=MPI_ADDRESS_KIND)`, wherever arguments of type MPI_Aint and MPI::Aint are used in C and C++. On Fortran 77 systems that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs are 32 bits, these arguments will be of type INTEGER*8.

4.1.2 Datatype Constructors

Contiguous The simplest datatype constructor is MPI_TYPE_CONTIGUOUS which allows replication of a datatype into contiguous locations.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count (non-negative integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

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

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_contiguous(int count) const
```

newtype is the datatype obtained by concatenating count copies of oldtype. Concatenation is defined using *extent* as the size of the concatenated copies.

Example 4.2 Let oldtype have type map $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, with extent 16, and let $\mathsf{count} = 3$. The type map of the datatype returned by newtype is

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{double}, 32), (\mathsf{char}, 40)\};$$

i.e., alternating double and char elements, with displacements $0, 8, 16, 24, 32, 40$.

In general, assume that the type map of oldtype is

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

with extent ex. Then newtype has a type map with $\mathsf{count} \cdot \mathsf{n}$ entries defined by:

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

$$..., (type_0, disp_0 + ex \cdot (\mathsf{count} - 1)), ..., (type_{n-1}, disp_{n-1} + ex \cdot (\mathsf{count} - 1))\}.$$

Vector The function MPI_TYPE_VECTOR is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	number of blocks (non-negative integer)
IN	blocklength	number of elements in each block (non-negative integer)
IN	stride	number of elements between start of each block (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_vector(int count, int blocklength, int stride,
              MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_vector(int count, int blocklength,
              int stride) const
```

Example 4.3 Assume, again, that oldtype has type map $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, with extent 16. A call to MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with type map,

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{double}, 32), (\mathsf{char}, 40),$$

$$(\mathsf{double}, 64), (\mathsf{char}, 72), (\mathsf{double}, 80), (\mathsf{char}, 88), (\mathsf{double}, 96), (\mathsf{char}, 104)\}.$$

That is, two blocks with three copies each of the old type, with a stride of 4 elements ($4 \cdot 16$ bytes) between the blocks.

Example 4.4 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, -32), (\mathsf{char}, -24), (\mathsf{double}, -64), (\mathsf{char}, -56)\}.$$

In general, assume that oldtype has type map,

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

with extent ex. Let bl be the blocklength. The newly created datatype has a type map with $\mathsf{count} \cdot \mathsf{bl} \cdot n$ entries:

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

$$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$$

$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$

$$(type_0, disp_0 + \mathsf{stride} \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex)\}.$$

A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), n arbitrary.

Hvector The function MPI_TYPE_CREATE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Section 4.1.14. (H stands for "heterogeneous").

MPI_TYPE_CREATE_HVECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	number of blocks (non-negative integer)
IN	blocklength	number of elements in each block (non-negative integer)
IN	stride	number of bytes between start of each block (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
             IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
```

```
MPI::Datatype MPI::Datatype::Create_hvector(int count, int blocklength,
             MPI::Aint stride) const
```

This function replaces MPI_TYPE_HVECTOR, whose use is deprecated. See also Chapter 15.

Assume that oldtype has type map,

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

with extent ex. Let bl be the blocklength. The newly created datatype has a type map with $\mathsf{count} \cdot \mathsf{bl} \cdot n$ entries:

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

$$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$$

$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$

$$(type_0, disp_0 + \mathsf{stride}), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride}), ...,$$

$$(type_0, disp_0 + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1)), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1)), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex)\}.$$

Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	number of blocks – also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)
IN	array_of_blocklengths	number of elements per block (array of non-negative integers)
IN	array_of_displacements	displacement for each block, in multiples of oldtype extent (array of integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_indexed(int count, int *array_of_blocklengths,
             int *array_of_displacements, MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_indexed(int count,
             const int array_of_blocklengths[],
             const int array_of_displacements[]) const
```

Example 4.5 Let oldtype have type map $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, with extent 16. Let $\mathsf{B} = (3,\ 1)$ and let $\mathsf{D} = (4,\ 0)$. A call to MPI_TYPE_INDEXED(2, B, D, oldtype, newtype) returns a datatype with type map,

$$\{(\mathsf{double}, 64), (\mathsf{char}, 72), (\mathsf{double}, 80), (\mathsf{char}, 88), (\mathsf{double}, 96), (\mathsf{char}, 104),$$

$$(\mathsf{double}, 0), (\mathsf{char}, 8)\}.$$

That is, three copies of the old type starting at displacement 64, and one copy starting at displacement 0.

In general, assume that oldtype has type map,

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

with extent *ex*. Let B be the array_of_blocklength argument and D be the array_of_displacements argument. The newly created datatype has $n \cdot \sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]}$ entries:

$$\{(type_0, disp_0 + \mathsf{D[0]} \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[0]} \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{D[0]} + \mathsf{B[0]} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{D[0]} + \mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{D[count} - 1] \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[count} - 1] \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{D[count}-1] + \mathsf{B[count}-1] - 1)\cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + (\mathsf{D[count}-1] + \mathsf{B[count}-1] - 1)\cdot ex)\}.$$

A call to MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent to a call to MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) where

$$\mathsf{D[j]} = j \cdot \mathsf{stride},\ j = 0, ..., \mathsf{count} - 1,$$

and

$$\mathsf{B[j]} = \mathsf{blocklength},\ j = 0, ..., \mathsf{count} - 1.$$

Hindexed The function MPI_TYPE_CREATE_HINDEXED is identical to MPI_TYPE_INDEXED, except that block displacements in array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.

MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	number of blocks — also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)
IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_hindexed(int count, int array_of_blocklengths[],
             MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
             ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

```
MPI::Datatype MPI::Datatype::Create_hindexed(int count,
             const int array_of_blocklengths[],
             const MPI::Aint array_of_displacements[]) const
```

This function replaces MPI_TYPE_HINDEXED, whose use is deprecated. See also Chapter 15.

Assume that oldtype has type map,

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

with extent ex. Let B be the array_of_blocklength argument and D be the array_of_displacements argument. The newly created datatype has a type map with $n \cdot \sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]}$ entries:

$$\{(type_0, disp_0 + \mathsf{D[0]}), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[0]}), ...,$$

$$(type_0, disp_0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{D[count - 1]}), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[count - 1]}), ...,$$

$$(type_0, disp_0 + \mathsf{D[count - 1]} + (\mathsf{B[count - 1]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[count - 1]} + (\mathsf{B[count - 1]} - 1) \cdot ex)\}.$$

Indexed_block This function is the same as MPI_TYPE_INDEXED except that the blocklength is the same for all blocks. There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). The following convenience function allows for constant blocksize and arbitrary displacements.

MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements, oldtype, newtype)

IN	count	length of array of displacements (non-negative integer)
IN	blocklength	size of block (non-negative integer)
IN	array_of_displacements	array of displacements (array of integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_indexed_block(int count, int blocklength,
             int array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
    NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_indexed_block(int count,
             int blocklength, const int array_of_displacements[]) const
```

Struct MPI_TYPE_STRUCT is the most general type constructor. It further generalizes MPI_TYPE_CREATE_HINDEXED in that it allows each block to consist of replications of different datatypes.

MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks (non-negative integer) — also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths
IN	array_of_blocklength	number of elements in each block (array of non-negative integer)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	array_of_types	type of elements in each block (array of handles to datatype objects)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
             MPI_Aint array_of_displacements[],
             MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
             ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
    IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

```
static MPI::Datatype MPI::Datatype::Create_struct(int count,
             const int array_of_blocklengths[], const MPI::Aint
             array_of_displacements[],
             const MPI::Datatype array_of_types[])
```

This function replaces MPI_TYPE_STRUCT, whose use is deprecated. See also Chapter 15.

Example 4.6 Let type1 have type map,

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8)\},$$

with extent 16. Let B = (2, 1, 3), D = (0, 16, 26), and T = (MPI_FLOAT, type1, MPI_CHAR). Then a call to MPI_TYPE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,

$$\{(\mathsf{float}, 0), (\mathsf{float}, 4), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{char}, 26), (\mathsf{char}, 27), (\mathsf{char}, 28)\}.$$

That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies four bytes.)

In general, let T be the array_of_types argument, where T[i] is a handle to,

$$typemap_i = \{(type_0^i, disp_0^i), ..., (type_{n_i-1}^i, disp_{n_i-1}^i)\},$$

with extent ex_i. Let B be the array_of_blocklength argument and D be the array_of_displacements argument. Let c be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{\mathsf{c}-1} \mathsf{B[i]} \cdot n_i$ entries:

$$\{(type_0^0, disp_0^0 + \mathsf{D[0]}), ..., (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D[0]}), ...,$$

$$(type_0^0, disp_0^0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex_0), ..., (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex_0), ...,$$

$$(type_0^{\mathsf{c}-1}, disp_0^{\mathsf{c}-1} + \mathsf{D[c-1]}), ..., (type_{n_{\mathsf{c}-1}-1}^{\mathsf{c}-1}, disp_{n_{\mathsf{c}-1}-1}^{\mathsf{c}-1} + \mathsf{D[c-1]}), ...,$$

$$(type_0^{\mathsf{c}-1}, disp_0^{\mathsf{c}-1} + \mathsf{D[c-1]} + (\mathsf{B[c-1]} - 1) \cdot ex_{\mathsf{c}-1}), ...,$$

$$(type_{n_{\mathsf{c}-1}-1}^{\mathsf{c}-1}, disp_{n_{\mathsf{c}-1}-1}^{\mathsf{c}-1} + \mathsf{D[c-1]} + (\mathsf{B[c-1]} - 1) \cdot ex_{\mathsf{c}-1})\}.$$

A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T is equal to oldtype.

4.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN	ndims	number of array dimensions (positive integer)
IN	array_of_sizes	number of elements of type oldtype in each dimension of the full array (array of positive integers)
IN	array_of_subsizes	number of elements of type oldtype in each dimension of the subarray (array of positive integers)
IN	array_of_starts	starting coordinates of the subarray in each dimension (array of non-negative integers)
IN	order	array storage order flag (state)
IN	oldtype	array element datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_subarray(int ndims, int array_of_sizes[],
             int array_of_subsizes[], int array_of_starts[], int order,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
             ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
    INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
    ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_subarray(int ndims,
             const int array_of_sizes[], const int array_of_subsizes[],
             const int array_of_starts[], int order) const
```

The subarray type constructor creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array. This type constructor facilitates creating filetypes to access arrays distributed in blocks among processes to a single file that contains the global array, see MPI I/O, especially Section 13.1.1 on page 373.

This type constructor can handle arrays with an arbitrary number of dimensions and works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note that a C program may use Fortran order and a Fortran program may use C order.

The ndims parameter specifies the number of dimensions in the full data array and gives the number of elements in array_of_sizes, array_of_subsizes, and array_of_starts.

The number of elements of type oldtype in each dimension of the n-dimensional array and the requested subarray are specified by array_of_sizes and array_of_subsizes, respectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] <1 or array_of_subsizes[i] $>$ array_of_sizes[i].

The array_of_starts contains the starting coordinates of each dimension of the subarray. Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to specify array_of_starts[i] <0 or array_of_starts[i] $>$ (array_of_sizes[i] $-$ array_of_subsizes[i]).

Advice to users. In a Fortran program with arrays indexed starting from 1, if the starting coordinate of a particular dimension of the subarray is n, then the entry in array_of_starts for that dimension is n-1. (*End of advice to users.*)

The order argument specifies the storage order for the subarray as well as the full array. It must be set to one of the following:

MPI_ORDER_C The ordering used by C arrays, (i.e., row-major order)

MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-major order)

A ndims-dimensional subarray (newtype) with no extra padding can be defined by the function Subarray() as follows:

$$\begin{aligned}\mathsf{newtype} \;=\; & \mathrm{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\},\\ & \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\\ & \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype})\end{aligned}$$

Let the typemap of oldtype have the form:

$$\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$$

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype. Then we define the Subarray() function recursively using the following three equations. Equation 4.2 defines the base step. Equation 4.3 defines the recursion step when order = MPI_ORDER_FORTRAN, and Equation 4.4 defines the recursion step when order = MPI_ORDER_C.

$$\begin{aligned}
&\text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \\
&\qquad \{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}) \\
&= \{(\mathsf{MPI\_LB}, 0), \\
&\quad (type_0, disp_0 + start_0 \times ex), \ldots, (type_{n-1}, disp_{n-1} + start_0 \times ex), \\
&\quad (type_0, disp_0 + (start_0 + 1) \times ex), \ldots, (type_{n-1}, \\
&\qquad\quad disp_{n-1} + (start_0 + 1) \times ex), \ldots \\
&\quad (type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots, \\
&\qquad\quad (type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex), \\
&\quad (\mathsf{MPI\_UB}, size_0 \times ex)\}
\end{aligned} \tag{4.2}$$

$$\begin{aligned}
&\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \\
&= \text{Subarray}(ndims - 1, \{size_1, size_2, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_1, subsize_2, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_1, start_2, \ldots, start_{ndims-1}\}, \\
&\qquad\quad \text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \mathsf{oldtype}))
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
&\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \\
&= \text{Subarray}(ndims - 1, \{size_0, size_1, \ldots, size_{ndims-2}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-2}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-2}\}, \\
&\qquad\quad \text{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, \mathsf{oldtype}))
\end{aligned} \tag{4.4}$$

For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 13.9.2.

4.1.4 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [1] data distributions. However, unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

> *Advice to users.* One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call this constructor with identical arguments (with the exception of rank which should be set appropriately). These filetypes (along with identical disp and etype) are then used to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 13.1.1 on page 373 and Section 13.3 on page 385. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs, array_of_dargs, array_of_psizes, order, oldtype, newtype)

IN	size	size of process group (positive integer)
IN	rank	rank in process group (non-negative integer)
IN	ndims	number of array dimensions as well as process grid dimensions (positive integer)
IN	array_of_gsizes	number of elements of type oldtype in each dimension of global array (array of positive integers)
IN	array_of_distribs	distribution of array in each dimension (array of state)
IN	array_of_dargs	distribution argument in each dimension (array of positive integers)
IN	array_of_psizes	size of process grid in each dimension (array of positive integers)
IN	order	array storage order flag (state)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_darray(int size, int rank, int ndims,
             int array_of_gsizes[], int array_of_distribs[], int
             array_of_dargs[], int array_of_psizes[], int order,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
             ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
    ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_darray(int size, int rank, int ndims,
             const int array_of_gsizes[], const int array_of_distribs[],
             const int array_of_dargs[], const int array_of_psizes[],
             int order) const
```

MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding to the distribution of an ndims-dimensional array of oldtype elements onto an ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be set to 1. (See Example 4.7, page 17.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the equation $\prod_{i=0}^{ndims-1} array\_of\_psizes[i] = size$ must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies .

Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 7 on page 241. (*End of advice to users.*)

Each dimension of the array can be distributed in one of three ways:

- MPI_DISTRIBUTE_BLOCK - Block distribution
- MPI_DISTRIBUTE_CYCLIC - Cyclic distribution
- MPI_DISTRIBUTE_NONE - Dimension not distributed.

The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify array_of_dargs[i] $*$ array_of_psizes[i] $<$ array_of_gsizes[i].

For example, the HPF layout `ARRAY(CYCLIC(15))` corresponds to MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout ARRAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of MPI_DISTRIBUTE_DFLT_DARG.

The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the storage order. Therefore, arrays described by this type constructor may be stored in Fortran (column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN and MPI_ORDER_C.

This routine creates a new MPI datatype with a typemap defined in terms of a function called "cyclic()" (see below).

Without loss of generality, it suffices to define the typemap for the MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.

MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.

MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to

$$(\text{array\_of\_gsizes[i]} + \text{array\_of\_psizes[i]} - 1)/\text{array\_of\_psizes[i]}.$$

If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_CYCLIC are equivalent.

MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to array_of_gsizes[i].

Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to 1.

For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined by the following code fragment:

```
oldtype[0] = oldtype;
for ( i = 0; i < ndims; i++ ) {
   oldtype[i+1] = cyclic(array_of_dargs[i],
                         array_of_gsizes[i],
                         r[i],
                         array_of_psizes[i],
                         oldtype[i]);
}
newtype = oldtype[ndims];
```

For MPI_ORDER_C, the code is:

```
oldtype[0] = oldtype;
for ( i = 0; i < ndims; i++ ) {
   oldtype[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
                           array_of_gsizes[ndims - i - 1],
                           r[ndims - i - 1],
                           array_of_psizes[ndims - i - 1],
                           oldtype[i]);
}
newtype = oldtype[ndims];
```

where $r[i]$ is the position of the process (with rank rank) in the process grid at dimension i. The values of $r[i]$ are given by the following code fragment:

```
t_rank = rank;
t_size = 1;
for (i = 0; i < ndims; i++)
        t_size *= array_of_psizes[i];
for (i = 0; i < ndims; i++) {
    t_size = t_size / array_of_psizes[i];
    r[i] = t_rank / t_size;
    t_rank = t_rank % t_size;
}
```

Let the typemap of oldtype have the form:

$$\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$$

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype.

Given the above, the function cyclic() is defined as follows:

$$\begin{aligned}
&\text{cyclic}(darg, gsize, r, psize, \textsf{oldtype}) \\
&\quad = \{(\textsf{MPI\_LB}, 0), \\
&\qquad (type_0, disp_0 + r \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex), \\
&\qquad (type_0, disp_0 + (r \times darg + 1) \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex), \\
&\qquad \ldots \\
&\qquad (type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex), \\
&\\
&\qquad (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex), \\
&\qquad (type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),
\end{aligned}$$

$$\begin{aligned}
&\ldots\\
&(type_0, disp_0 + ((r+1)\times darg - 1)\times ex + psize \times darg \times ex), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + ((r+1)\times darg - 1)\times ex + psize \times darg \times ex),\\
&\qquad \vdots\\
&(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),\\
&(type_0, disp_0 + (r \times darg + 1)\times ex + psize \times darg \times ex \times (count - 1)), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1)\times ex\\
&\qquad\qquad + psize \times darg \times ex \times (count - 1)),\\
&\ldots\\
&(type_0, disp_0 + (r \times darg + darg_{last} - 1)\times ex\\
&\qquad\qquad + psize \times darg \times ex \times (count - 1)), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1)\times ex\\
&\qquad\qquad + psize \times darg \times ex \times (count - 1)),\\
&(\mathsf{MPI\_UB}, gsize * ex)\}
\end{aligned}$$

where *count* is defined by this code fragment:

```
nblocks = (gsize + (darg - 1)) / darg;
count = nblocks / psize;
left_over = nblocks - count * psize;
if (r < left_over)
    count = count + 1;
```

Here, *nblocks* is the number of blocks that must be distributed among the processors. Finally, $darg_{last}$ is defined by this code fragment:

```
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
     darg_last = darg;
else
     darg_last = num_in_last_cyclic - darg * r;
     if (darg_last > darg)
            darg_last = darg;
     if (darg_last <= 0)
            darg_last = darg;
```

Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:

```
      <oldtype> FILEARRAY(100, 200, 300)
!HPF$ PROCESSORS PROCESSES(2, 3)
!HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
```

This can be achieved by the following Fortran code, assuming there will be six processes attached to the run:

```
    ndims = 3
    array_of_gsizes(1) = 100
    array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
    array_of_dargs(1) = 10
    array_of_gsizes(2) = 200
    array_of_distribs(2) = MPI_DISTRIBUTE_NONE
    array_of_dargs(2) = 0
    array_of_gsizes(3) = 300
    array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
    array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_ARG
    array_of_psizes(1) = 2
    array_of_psizes(2) = 1
    array_of_psizes(3) = 3
    call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
    call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
         array_of_distribs, array_of_dargs, array_of_psizes,        &
         MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
```

4.1.5 Address and Size Functions

The displacements in a general datatype are relative to some initial buffer address. **Absolute addresses** can be substituted for these displacements: we treat them as displacements relative to “address zero,” the start of the address space. This initial address zero is indicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI_BOTTOM.

The address of a location in memory can be found by invoking the function MPI_GET_ADDRESS.

MPI_GET_ADDRESS(location, address)

IN	location	location in caller memory (choice)
OUT	address	address of location (integer)

```
int MPI_Get_address(void *location, MPI_Aint *address)
```

```
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
```

```
MPI::Aint MPI::Get_address(void* location)
```

This function replaces MPI_ADDRESS, whose use is deprecated. See also Chapter 15.
Returns the (byte) address of location.

Advice to users. Current Fortran MPI codes will run unmodified, and will port to any system. However, they may fail if addresses larger than $2^{32} - 1$ are used in the program. New codes should be written so that they use the new functions. This provides compatibility with C/C++ and avoids errors on 64 bit architectures.

However, such newly written codes may need to be (slightly) rewritten to port to old Fortran 77 environments that do not support KIND declarations. (*End of advice to users.*)

Example 4.8 Using MPI_GET_ADDRESS for an array.

```
   REAL A(100,100)
   INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
   CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
   CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
   DIFF = I2 - I1
! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
! implementation dependent.
```

Advice to users. C users may be tempted to avoid the usage of MPI_GET_ADDRESS and rely on the availability of the address operator &. Note, however, that & *cast-expression* is a pointer, not an address. ISO C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at — although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI_GET_ADDRESS to "reference" C variables guarantees portability to such machines as well. (*End of advice to users.*)

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.2 on pages 463 and 466. (*End of advice to users.*)

The following auxiliary function provides useful information on derived datatypes.

MPI_TYPE_SIZE(datatype, size)

IN	datatype	datatype (handle)
OUT	size	datatype size (integer)

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

```
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

```
int MPI::Datatype::Get_size() const
```

MPI_TYPE_SIZE returns the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity.

4.1.6 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 on page 20. 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 4.1.14. Also, the user may want to overide the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to overide default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

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 ($extent(\mathsf{MPI\_LB}) = extent(\mathsf{MPI\_UB}) = 0$). 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 4.9 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)} . (An entry of type ub can be deleted if there is another entry of type ub with a higher displacement; an entry of type lb can be deleted if there is another entry of type lb with a lower displacement.)

In general, if

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

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

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

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

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

Then

$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

If $type_i$ requires alignment to a byte address that is a multiple of k_i, then ϵ is the least non-negative 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**.

4.1.7 Extent and Bounds of Datatypes

The following function replaces the three functions MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT. It also returns address sized integers, in the Fortran binding. The use of MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT is deprecated.

MPI_TYPE_GET_EXTENT(datatype, lb, extent)

IN	datatype	datatype to get information on (handle)
OUT	lb	lower bound of datatype (integer)
OUT	extent	extent of datatype (integer)

```
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
             MPI_Aint *extent)
```

```
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT
```

```
void MPI::Datatype::Get_extent(MPI::Aint& lb, MPI::Aint& extent) const
```

Returns the lower bound and the extent of datatype (as defined in Section 4.1.6 on page 20).

MPI allows one to change the extent of a datatype, using lower bound and upper bound markers (MPI_LB and MPI_UB). This is useful, as it allows to control the stride of successive datatypes that are replicated by datatype constructors, or are replicated by the count argument in a send or receive call. However, the current mechanism for achieving it is painful; also it is restrictive. MPI_LB and MPI_UB are “sticky”: once present in a datatype, they cannot be overridden (e.g., the upper bound can be moved up, by adding a new MPI_UB marker, but cannot be moved down below an existing MPI_UB marker). A new type constructor is provided to facilitate these changes. The use of MPI_LB and MPI_UB is deprecated.

MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)

IN	oldtype	input datatype (handle)
IN	lb	new lower bound of datatype (integer)
IN	extent	new extent of datatype (integer)
OUT	newtype	output datatype (handle)

```
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint
             extent, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
```

```
MPI::Datatype MPI::Datatype::Create_resized(const MPI::Aint lb,
             const MPI::Aint extent) const
```

Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb + extent. Any previous **lb** and **ub** markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and extent arguments. This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

> *Advice to users.* It is strongly recommended that users use these two new functions, rather than the old MPI-1 functions to set and access lower bound, upper bound and extent of datatypes. (*End of advice to users.*)

4.1.8 True Extent of Datatypes

Suppose we implement gather (see also Section 5.5 on page 137) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent using the MPI_UB and MPI_LB values. A function is provided which returns the true extent of the datatype.

MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)

IN	datatype	datatype to get information on (handle)
OUT	true_lb	true lower bound of datatype (integer)
OUT	true_extent	true size of datatype (integer)

```
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
             MPI_Aint *true_extent)
```

```
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
```

```
void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb,
             MPI::Aint& true_extent) const
```

true_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring MPI_LB markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring MPI_LB and MPI_UB markers, and performing no rounding for alignment. If the typemap associated with datatype is

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

Then

$$true\_lb(Typemap) = min_j\{disp_j \;:\; type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

$$true\_ub(Typemap) = max_j\{disp_j + sizeof(type_j) \;:\; type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

and

$$true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).$$

(Readers should compare this with the definitions in Section 4.1.6 on page 20 and Section 4.1.7 on page 21, which describe the function MPI_TYPE_GET_EXTENT.)

The true_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed.

4.1.9 Commit and Free

A datatype object has to be **committed** before it can be used in a communication. As an argument in datatype constructors, uncommitted and also committed datatypes can be used. 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
```

```
void MPI::Datatype::Commit()
```

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_COMMIT will accept a committed datatype; in this case, it is equivalent to a no-op.

Example 4.10 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
```

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
```

```
void MPI::Datatype::Free()
```

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. 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.*)

4.1.10 Duplicating a Datatype

MPI_TYPE_DUP(type, newtype)

IN	type	datatype (handle)
OUT	newtype	copy of type (handle)

```
int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)
```

```
MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)
    INTEGER TYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Dup() const
```

MPI_TYPE_DUP is a type constructor which duplicates the existing type with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as type and any copied cached information, see Section 6.7.4 on page 230. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 4.1.13. The newtype has the same committed state as the old type.

4.1.11 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,

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

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 $n \cdot \mathsf{count}$ entries, where entry $i \cdot n + j$ is at location $addr_{i,j} = \mathsf{buf} + extent \cdot i + disp_j$ and has type $type_j$, for $i = 0, ..., \mathsf{count} - 1$ and $j = 0, ..., n - 1$. 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 $n \cdot \mathsf{count}$ 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,

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

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 $n \cdot \mathsf{count}$ entries, where entry $i \cdot n + j$ is at location $\mathsf{buf} + extent \cdot i + disp_j$ and has type $type_j$. If the incoming message consists of k elements, then we must have $k \leq n \cdot \mathsf{count}$; 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 4.11 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. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)

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

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

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 $0 \leq k \leq \mathsf{count} \cdot n$. 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
```

```
int MPI::Status::Get_elements(const MPI::Datatype& datatype) const
```

The previously defined function, MPI_GET_COUNT (Section 3.2.5), has a different behavior. It returns the number of “top-level entries” received, i.e. the number of “copies” of type datatype. In the previous example, MPI_GET_COUNT may return any integer value k, where $0 \leq k \leq \mathsf{count}$. 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. The datatype argument should match the argument provided by the receive call that set the status variable.

Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.

```
...
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 IF (rank.EQ.1) THEN
      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.*)

4.1.12 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_GET_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.*)

4.1.13 Decoding a Datatype

MPI datatype objects allow users to specify an arbitrary layout of data in memory. There are several cases where accessing the layout information in opaque datatype objects would be useful. The opaque datatype object has found a number of uses outside MPI. Furthermore, a number of tools wish to display internal information about a datatype. To achieve this, datatype decoding functions are provided. The two functions in this section are used together to decode datatypes to recreate the calling sequence used in their initial definition. These can be used to allow a user to determine the type map and type signature of a datatype.

MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, combiner)

IN	datatype	datatype to access (handle)
OUT	num_integers	number of input integers used in the call constructing combiner (non-negative integer)
OUT	num_addresses	number of input addresses used in the call constructing combiner (non-negative integer)
OUT	num_datatypes	number of input datatypes used in the call constructing combiner (non-negative integer)
OUT	combiner	combiner (state)

```
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
             int *num_addresses, int *num_datatypes, int *combiner)
```

```
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
             COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR
```

```
void MPI::Datatype::Get_envelope(int& num_integers, int& num_addresses,
             int& num_datatypes, int& combiner) const
```

For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-of-arguments values returned can be used to provide sufficiently large arrays in the decoding routine MPI_TYPE_GET_CONTENTS. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

Rationale. By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. One call is effectively the same as another when the information obtained from MPI_TYPE_GET_CONTENTS may be used with either to produce the same outcome. C calls MPI_Type_hindexed and MPI_Type_create_hindexed are always effectively the same while the Fortran call MPI_TYPE_HINDEXED will be different than either of these in some MPI implementations. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

The decoded information keeps track of datatype duplications. This is important as one needs to distinguish between a predefined datatype and a dup of a predefined datatype. The former is a constant object that cannot be freed, while the latter is a derived datatype that can be freed. (*End of rationale.*)

The list below has the values that can be returned in combiner on the left and the call associated with them on the right.

If combiner is MPI_COMBINER_NAMED then datatype is a named predefined datatype.

MPI_COMBINER_NAMED	a named predefined datatype
MPI_COMBINER_DUP	MPI_TYPE_DUP
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR
MPI_COMBINER_HVECTOR_INTEGER	MPI_TYPE_HVECTOR from Fortran
MPI_COMBINER_HVECTOR	MPI_TYPE_HVECTOR from C or C++ and in some case Fortran or MPI_TYPE_CREATE_HVECTOR
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED
MPI_COMBINER_HINDEXED_INTEGER	MPI_TYPE_HINDEXED from Fortran
MPI_COMBINER_HINDEXED	MPI_TYPE_HINDEXED from C or C++ and in some case Fortran or MPI_TYPE_CREATE_HINDEXED
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK
MPI_COMBINER_STRUCT_INTEGER	MPI_TYPE_STRUCT from Fortran
MPI_COMBINER_STRUCT	MPI_TYPE_STRUCT from C or C++ and in some case Fortran or MPI_TYPE_CREATE_STRUCT
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED

Table 4.1: combiner values returned from MPI_TYPE_GET_ENVELOPE

For deprecated calls with address arguments, we sometimes need to differentiate whether the call used an integer or an address size argument. For example, there are two combiners for hvector: MPI_COMBINER_HVECTOR_INTEGER and MPI_COMBINER_HVECTOR. The former is used if it was the MPI-1 call from Fortran, and the latter is used if it was the MPI-1 call from C or C++. However, on systems where MPI_ADDRESS_KIND = MPI_INTEGER_KIND (i.e., where integer arguments and address size arguments are the same), the combiner MPI_COMBINER_HVECTOR may be returned for a datatype constructed by a call to MPI_TYPE_HVECTOR from Fortran. Similarly, MPI_COMBINER_HINDEXED may be returned for a datatype constructed by a call to MPI_TYPE_HINDEXED from Fortran, and MPI_COMBINER_STRUCT may be returned for a datatype constructed by a call to MPI_TYPE_STRUCT from Fortran. On such systems, one need not differentiate constructors that take address size arguments from constructors that take integer arguments, since these are the same. The preferred calls all use address sized arguments so two combiners are not required for them.

Rationale. For recreating the original call, it is important to know if address information may have been truncated. The deprecated calls from Fortran for a few routines could be subject to truncation in the case where the default `INTEGER` size is smaller than the size of an address. (*End of rationale.*)

The actual arguments used in the creation call for a datatype can be obtained from the call:

MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes, array_of_integers, array_of_addresses, array_of_datatypes)

IN	datatype	datatype to access (handle)
IN	max_integers	number of elements in array_of_integers (non-negative integer)
IN	max_addresses	number of elements in array_of_addresses (non-negative integer)
IN	max_datatypes	number of elements in array_of_datatypes (non-negative integer)
OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)
OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)
OUT	array_of_datatypes	contains datatype arguments used in constructing datatype (array of handles)

```
int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
             int max_addresses, int max_datatypes, int array_of_integers[],
             MPI_Aint array_of_addresses[],
             MPI_Datatype array_of_datatypes[])
```

```
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
             ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
             IERROR)
    INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
    ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
```

```
void MPI::Datatype::Get_contents(int max_integers, int max_addresses,
             int max_datatypes, int array_of_integers[],
             MPI::Aint array_of_addresses[],
             MPI::Datatype array_of_datatypes[]) const
```

datatype must be a predefined unnamed or a derived datatype; the call is erroneous if datatype is a predefined named datatype.

The values given for max_integers, max_addresses, and max_datatypes must be at least as large as the value returned in num_integers, num_addresses, and num_datatypes, respectively, in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.

Rationale. The arguments max_integers, max_addresses, and max_datatypes allow for error checking in the call. (*End of rationale.*)

The datatypes returned in array_of_datatypes are handles to datatype objects that are equivalent to the datatypes used in the original construction call. If these were derived datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then the returned datatype is equal to that (constant) predefined datatype and cannot be freed.

The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is undefined.

Note that MPI_TYPE_GET_CONTENTS can be invoked with a datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed predefined datatype). In such a case, an empty array_of_datatypes is returned.

> *Rationale.* The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, it is possible to use the `==` or `.EQ.` comparison operator to determine the datatype involved. (*End of rationale.*)

> *Advice to implementors.* The datatypes returned in array_of_datatypes must appear to the user as if each is an equivalent copy of the datatype used in the type constructor call. Whether this is done by creating a new datatype or via another mechanism such as a reference count mechanism is up to the implementation as long as the semantics are preserved. (*End of advice to implementors.*)

> *Rationale.* The committed state and attributes of the returned datatype is deliberately left vague. The datatype used in the original construction may have been modified since its use in the constructor call. Attributes can be added, removed, or modified as well as having the datatype committed. The semantics given allow for a reference count implementation without having to track these changes. (*End of rationale.*)

In the deprecated datatype constructor calls, the address arguments in Fortran are of type `INTEGER`. In the preferred calls, the address arguments are of type `INTEGER(KIND=MPI_ADDRESS_KIND)`. The call MPI_TYPE_GET_CONTENTS returns all addresses in an argument of type `INTEGER(KIND=MPI_ADDRESS_KIND)`. This is true even if the deprecated calls were used. Thus, the location of values returned can be thought of as being returned by the C bindings. It can also be determined by examining the preferred calls for datatype constructors for the deprecated calls that involve addresses.

> *Rationale.* By having all address arguments returned in the array_of_addresses argument, the result from a C and Fortran decoding of a datatype gives the result in the same argument. It is assumed that an integer of type `INTEGER(KIND=MPI_ADDRESS_KIND)` will be at least as large as the `INTEGER` argument used in datatype construction with the old MPI-1 calls so no loss of information will occur. (*End of rationale.*)

The following defines what values are placed in each entry of the returned arrays depending on the datatype constructor used for datatype. It also specifies the size of the arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, the following calls were made:

```
      PARAMETER (LARGE = 1000)
      INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) A(LARGE)
!     CONSTRUCT DATATYPE TYPE (NOT SHOWN)
      CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)
      IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
        WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, &
        " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
        CALL MPI_ABORT(MPI_COMM_WORLD, 99)
      ENDIF
      CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR)
```

or in C the analogous calls of:

```
#define LARGE 1000
int ni, na, nd, combiner, i[LARGE];
MPI_Aint a[LARGE];
MPI_Datatype type, d[LARGE];
/* construct datatype type (not shown) */
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
  fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
  fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
          LARGE);
  MPI_Abort(MPI_COMM_WORLD, 99);
};
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
```

The C++ code is in analogy to the C code above with the same values returned.

In the descriptions that follow, the lower case name of arguments is used.

If combiner is MPI_COMBINER_NAMED then it is erroneous to call MPI_TYPE_GET_CONTENTS.

If combiner is MPI_COMBINER_DUP then

Constructor argument	C & C++ location	Fortran location
oldtype	d[0]	D(1)

and ni = 0, na = 0, nd = 1.

If combiner is MPI_COMBINER_CONTIGUOUS then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
oldtype	d[0]	D(1)

and ni = 1, na = 0, nd = 1.

If combiner is MPI_COMBINER_VECTOR then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	i[2]	I(3)
oldtype	d[0]	D(1)

and ni = 3, na = 0, nd = 1.

If combiner is MPI_COMBINER_HVECTOR_INTEGER or MPI_COMBINER_HVECTOR then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	a[0]	A(1)
oldtype	d[0]	D(1)

and ni = 2, na = 1, nd = 1.

If combiner is MPI_COMBINER_INDEXED then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
oldtype	d[0]	D(1)

and ni = 2*count+1, na = 0, nd = 1.

If combiner is MPI_COMBINER_HINDEXED_INTEGER or MPI_COMBINER_HINDEXED then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
oldtype	d[0]	D(1)

and ni = count+1, na = count, nd = 1.

If combiner is MPI_COMBINER_INDEXED_BLOCK then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
array_of_displacements	i[2] to i[i[0]+1]	I(3) to I(I(1)+2)
oldtype	d[0]	D(1)

and ni = count+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_STRUCT_INTEGER or MPI_COMBINER_STRUCT then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
array_of_types	d[0] to d[i[0]-1]	D(1) to D(I(1))

and ni = count+1, na = count, nd = count.

If combiner is MPI_COMBINER_SUBARRAY then

Constructor argument	C & C++ location	Fortran location
ndims	i[0]	I(1)
array_of_sizes	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_subsizes	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
array_of_starts	i[2*i[0]+1] to i[3*i[0]]	I(2*I(1)+2) to I(3*I(1)+1)
order	i[3*i[0]+1]	I(3*I(1)+2]
oldtype	d[0]	D(1)

and ni = 3*ndims+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_DARRAY then

Constructor argument	C & C++ location	Fortran location
size	i[0]	I(1)
rank	i[1]	I(2)
ndims	i[2]	I(3)
array_of_gsizes	i[3] to i[i[2]+2]	I(4) to I(I(3)+3)
array_of_distribs	i[i[2]+3] to i[2*i[2]+2]	I(I(3)+4) to I(2*I(3)+3)
array_of_dargs	i[2*i[2]+3] to i[3*i[2]+2]	I(2*I(3)+4) to I(3*I(3)+3)
array_of_psizes	i[3*i[2]+3] to i[4*i[2]+2]	I(3*I(3)+4) to I(4*I(3)+3)
order	i[4*i[2]+3]	I(4*I(3)+4)
oldtype	d[0]	D(1)

and ni = 4*ndims+4, na = 0, nd = 1.

If combiner is MPI_COMBINER_F90_REAL then

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_COMPLEX then

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_INTEGER then

Constructor argument	C & C++ location	Fortran location
r	i[0]	I(1)

and ni = 1, na = 0, nd = 0.

If combiner is MPI_COMBINER_RESIZED then

Constructor argument	C & C++ location	Fortran location
lb	a[0]	A(1)
extent	a[1]	A(2)
oldtype	d[0]	D(1)

and ni = 0, na = 2, nd = 1.

4.1.14 Examples

The following examples illustrate the use of derived datatypes.

Example 4.13 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, ierr)

      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,
                             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 4.14 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, ierr)

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 4.15 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, ierr)

      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 4.16 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, ierr)

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 4.17 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];
MPI_Aint     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 4.18 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 4.19 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);
```

Example 4.20 This example shows how a datatype can be decoded. The routine printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for datatypes that are not predefined.

```
/*
  Example of decoding a datatype.

  Returns 0 if the datatype is predefined, 1 otherwise
 */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int printdatatype( MPI_Datatype datatype )
{
    int *array_of_ints;
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
    int i;

    MPI_Type_get_envelope( datatype,
                           &num_ints, &num_adds, &num_dtypes, &combiner );
    switch (combiner) {
    case MPI_COMBINER_NAMED:
        printf( "Datatype is named:" );
        /* To print the specific type, we can match against the
           predefined forms. We can NOT use a switch statement here
```

```
              We could also use MPI_TYPE_GET_NAME if we prefered to use
              names that the user may have changed.
            */
           if      (datatype == MPI_INT)    printf( "MPI_INT\n" );
           else if (datatype == MPI_DOUBLE) printf( "MPI_DOUBLE\n" );
           ... else test for other types ...
           return 0;
           break;
       case MPI_COMBINER_STRUCT:
       case MPI_COMBINER_STRUCT_INTEGER:
           printf( "Datatype is struct containing" );
           array_of_ints   = (int *)malloc( num_ints * sizeof(int) );
           array_of_adds   =
                      (MPI_Aint *) malloc( num_adds * sizeof(MPI_Aint) );
           array_of_dtypes = (MPI_Datatype *)
               malloc( num_dtypes * sizeof(MPI_Datatype) );
           MPI_Type_get_contents( datatype, num_ints, num_adds, num_dtypes,
                            array_of_ints, array_of_adds, array_of_dtypes );
           printf( " %d datatypes:\n", array_of_ints[0] );
           for (i=0; i<array_of_ints[0]; i++) {
               printf( "blocklength %d, displacement %ld, type:\n",
                       array_of_ints[i+1], array_of_adds[i] );
               if (printdatatype( array_of_dtypes[i] )) {
                   /* Note that we free the type ONLY if it
                      is not predefined */
                   MPI_Type_free( &array_of_dtypes[i] );
               }
           }
           free( array_of_ints );
           free( array_of_adds );
           free( array_of_dtypes );
           break;
           ... other combiner values ...
       default:
           printf( "Unrecognized combiner type\n" );
       }
       return 1;
   }
```

4.2 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 4.1, 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 content 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 (non-negative integer)
IN	datatype	datatype of each input data item (handle)
OUT	outbuf	output buffer start (choice)
IN	outsize	output buffer size, in bytes (non-negative 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
```

```
void MPI::Datatype::Pack(const void* inbuf, int incount, void *outbuf,
             int outsize, int& position, const MPI::Comm &comm) const
```

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 (non-negative 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
```

```
void MPI::Datatype::Unpack(const void* inbuf, int insize, void *outbuf,
             int outcount, int& position, const MPI::Comm& comm) const
```

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 input 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 input 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 (non-negative 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 (non-negative 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
```

```
int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm) const
```

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 4.21 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 4.22 An 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 if (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 4.23 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, comm, &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';
}
```

4.3 Canonical MPI_PACK and MPI_UNPACK

These functions read/write data to/from the buffer in the "external32" data format specified in Section 13.5.2, and calculate the size needed for packing. Their first arguments specify the data format, for future extensibility, but currently the only valid value of the datarep argument is "external32."

Advice to users. These functions could be used, for example, to send typed data in a portable format from one MPI implementation to another. (*End of advice to users.*)

The buffer will contain exactly the packed data, without headers. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.

Rationale. MPI_PACK_EXTERNAL specifies that there is no header on the message and further specifies the exact format of the data. Since MPI_PACK may (and is allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed with MPI_PACK_EXTERNAL. (*End of rationale.*)

MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)

IN	datarep	data representation (string)
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)

```
int MPI_Pack_external(char *datarep, void *inbuf, int incount,
             MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
             MPI_Aint *position)
```

```
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
             POSITION, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
```

```
void MPI::Datatype::Pack_external(const char* datarep, const void* inbuf,
             int incount, void* outbuf, MPI::Aint outsize,
             MPI::Aint& position) const
```

MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outsize, position)

IN	datarep	data representation (string)
IN	inbuf	input buffer start (choice)
IN	insize	input buffer size, in bytes (integer)
INOUT	position	current position in buffer, in bytes (integer)
OUT	outbuf	output buffer start (choice)
IN	outcount	number of output data items (integer)
IN	datatype	datatype of output data item (handle)

```
int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize,
             MPI_Aint *position, void *outbuf, int outcount,
             MPI_Datatype datatype)
```

```
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
             DATATYPE, IERROR)
    INTEGER OUTCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
```

```
void MPI::Datatype::Unpack_external(const char* datarep, const void* inbuf,
             MPI::Aint insize, MPI::Aint& position, void* outbuf,
             int outcount) const
```

MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)

IN	datarep	data representation (string)
IN	incount	number of input data items (integer)
IN	datatype	datatype of each input data item (handle)
OUT	size	output buffer size, in bytes (integer)

```
int MPI_Pack_external_size(char *datarep, int incount,
             MPI_Datatype datatype, MPI_Aint *size)
```

```
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    CHARACTER*(*) DATAREP
```

```
MPI::Aint MPI::Datatype::Pack_external_size(const char* datarep,
             int incount) const
```

Bibliography

[1] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1993. 4.1.4

Index