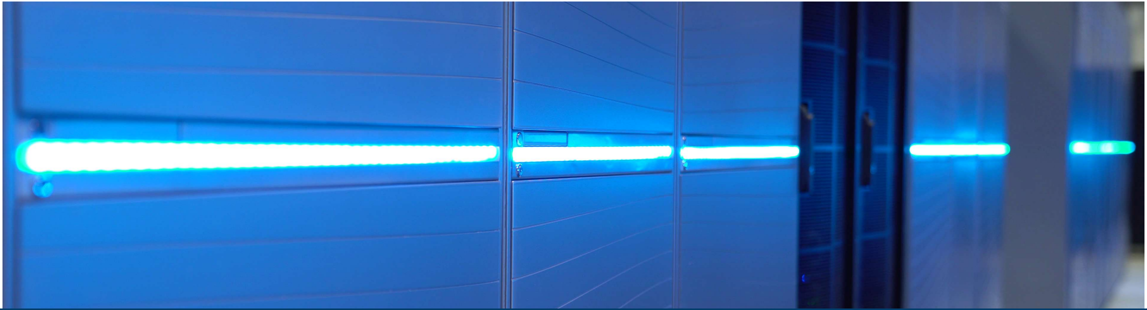


Part XI: Derived Datatypes

MOTIVATION [MPI-4.0, 5.1]

Reminder: Buffer

- Message buffers are defined by a triple (`address`, `count`, `datatype`).
- Basic data types restrict buffers to homogeneous, contiguous sequences of values in memory.

Scenario A

Problem: Want to communicate data describing particles that consists of a position (3 `double`) and a particle species (encoded as an `int`).
Solution(?): Communicate positions and species in two separate operations.

Scenario B

Problem: Have an array `real :: a(:)`, want to communicate only every second entry `a(1:n:2)`.
Solution(?): Copy data to a temporary array.

Derived datatypes are a mechanism for describing arrangements of data in buffers. Gives the MPI library the opportunity to employ the optimal solution.

TYPE MAP & TYPE SIGNATURE [MPI-4.0, 5.1]

Type map

A general datatype is described by its type map, a sequence of pairs of basic datatype and displacement:

$$\mathit{Typemap} = \{(\mathit{type}_0, \mathit{disp}_0), \ldots, (\mathit{type}_{n-1}, \mathit{disp}_{n-1})\}$$

Type signature

A type signature describes the contents of a message read from a buffer with a general datatype:

$$\mathit{Typesig} = \{\mathit{type}_0, \ldots, \mathit{type}_{n-1}\}$$

Type matching is done based on type signatures alone.

EXAMPLE

C

```c
struct heterogeneous {
  int i[4];
  double d[5];
}
```

F08

```fortran
type, bind(C) :: heterogeneous
  integer :: i(4)
  real(real64) :: d(5)
end type
```

	Basic Datatype	
0	MPI_INT	MPI_INTEGER
4	MPI_INT	MPI_INTEGER
8	MPI_INT	MPI_INTEGER
12	MPI_INT	MPI_INTEGER
16	MPI_DOUBLE	MPI_REAL8
24	MPI_DOUBLE	MPI_REAL8
32	MPI_DOUBLE	MPI_REAL8
40	MPI_DOUBLE	MPI_REAL8
48	MPI_DOUBLE	MPI_REAL8

TYPE CONSTRUCTORS [MPI-4.0, 5.1]

A new derived type is constructed from an existing type `oldtype` (basic or derived) using type constructors. In order of increasing generality/complexity:

1. `MPI_Type_contiguous` n consecutive instances of `oldtype`
2. `MPI_Type_vector` n blocks of m instances of `oldtype` with stride s
3. `MPI_Type_create_indexed_block` n blocks of m instances of `oldtype` with displacement d_i for each $i = 1, \ldots, n$
4. `MPI_Type_indexed` n blocks of m_i instances of `oldtype` with displacement d_i for each $i = 1, \ldots, n$
5. `MPI_Type_create_struct` n blocks of m_i instances of $\texttt{oldtype}_i$ with displacement d_i for each $i = 1, \ldots, n$
6. `MPI_Type_create_subarray` n dimensional subarray out of an array with elements of type `oldtype`
7. `MPI_Type_create_darray` distributed array with elements of type `oldtype`

CONTIGUOUS DATA [MPI-4.0, 5.1.2]

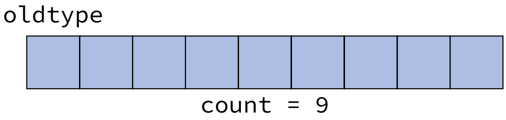
C

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

F08

```fortran
MPI_Type_contiguous(count, oldtype, newtype, ierror)
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: oldtype
type(MPI_Datatype), intent(out) :: newtype
integer, optional, intent(out) :: ierror
```

- Simple concatenation of `oldtype`
- Results in the same access pattern as using `oldtype` and specifying a buffer with `count` greater than one.



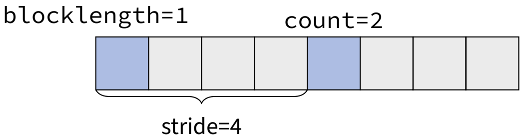
VECTOR DATA [MPI-4.0, 5.1.2]

C

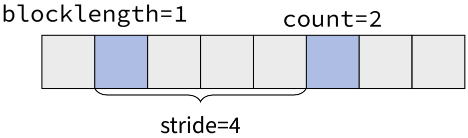
```c
int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype
  ↪ oldtype, MPI_Datatype *newtype)
```

F08

```fortran
MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
integer, intent(in) :: count, blocklength, stride
type(MPI_Datatype), intent(in) :: oldtype
type(MPI_Datatype), intent(out) :: newtype
integer, optional, intent(out) :: ierror
```



QUIZ



How does one send/broadcast elements shown above, using exactly the same previously defined `MPI_Type_vector` datatype?

JÜLICH Forschungszentrum

EXERCISES

Exercise 14 – Derived Datatypes

14.1 Matrix Access – Diagonal

In the file `matrix_access.{c|cxx|f90|py}` implement the function/subroutine `get_diagonal` that extracts the elements on the diagonal of an $N \times N$ matrix into a vector:

$$\texttt{vector}_i = \texttt{matrix}_{i,i}, \quad i = 1 \ldots N.$$

Do not access the elements of either the matrix or the vector directly. Rather, use MPI datatypes for accessing your data. Assume that the matrix elements are stored in row-major order in C (all elements of the first row, followed by all elements of the second row, etc.), column-major order in Fortran.

Hint: `MPI_Sendrecv` on the `MPI_COMM_SELF` communicator can be used for copying the data.

Use: `MPI_Type_vector`

COMMIT & FREE [MPI-4.0, 5.1.9]

Before using a derived datatype in communication it needs to be committed

C

```c
int MPI_Type_commit(MPI_Datatype* datatype)
```

F08

```fortran
MPI_Type_commit(datatype, ierror)
type(MPI_Datatype), intent(inout) :: datatype
integer, optional, intent(out) :: ierror
```

Marking derived datatypes for deallocation

C

```c
int MPI_Type_free(MPI_Datatype *datatype)
```

F08

```fortran
MPI_Type_free(datatype, ierror)
type(MPI_Datatype), intent(inout) :: datatype
integer, optional, intent(out) :: ierror
```

STRUCT DATA [MPI-4.0, 5.1.2]

C

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

F08

```fortran
MPI_Type_create_struct(count, array_of_blocklengths,
  ↪  array_of_displacements, array_of_types, newtype, ierror)
integer, intent(in) :: count, array_of_blocklengths(count)
integer(kind=MPI_ADDRESS_KIND), intent(in) :: array_of_displacements(count)
type(MPI_Datatype), intent(in) :: array_of_types(count)
type(MPI_Datatype), intent(out) :: newtype
integer, optional, intent(out) :: ierror
```

Fortran derived data types also supports **sequence** or **bind**(C) declaration, see [MPI-4.0, 19.1.15].
The difference is in the padding default behaviour between Fortran or C compiler.

JÜLICH Forschungszentrum

EXAMPLE

```
struct heterogeneous {
  int i[4];
  double d[5];
}

count = 2;
array_of_blocklengths[0] = 4;
array_of_displacements[0] = 0;
array_of_types[0] = MPI_INT;
array_of_blocklengths[1] = 5;
array_of_displacements[1] = 16;
array_of_types[1] = MPI_DOUBLE;
```

EXAMPLE

F08

```fortran
type, bind(C) :: heterogeneous
  integer :: i(4)
  real(real64) :: d(5)
end type

count = 2;
array_of_blocklengths(1) = 4
array_of_displacements(1) = 0
array_of_types(1) = MPI_INTEGER
array_of_blocklengths(2) = 5
array_of_displacements(2) = 16
array_of_types(2) = MPI_REAL8
```

ALIGNMENT & PADDING

```c
struct heterogeneous {
  int i[3];
  double d[5];
}

count = 2;
array_of_blocklengths[0] = 3;
array_of_displacements[0] = 0;
array_of_types[0] = MPI_INT;
array_of_blocklengths[1] = 5;
array_of_displacements[1] = 16;
array_of_types[1] = MPI_DOUBLE;
```

ALIGNMENT & PADDING

F08

```
type, bind(C) :: heterogeneous
  integer :: i(3)
  real(real64) :: d(5)
end type

count = 2;
array_of_blocklengths(1) = 3
array_of_displacements(1) = 0
array_of_types(1) = MPI_INTEGER
array_of_blocklengths(2) = 5
array_of_displacements(2) = 16
array_of_types(2) = MPI_REAL8
```

ADDRESS CALCULATION [MPI-4.0, 5.1.5]

Displacements are calculated as the difference between the addresses at the start of a buffer and at a particular piece of data in the buffer. The address of a location in memory is found using:

C

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

F08

```fortran
MPI_Get_address(location, address, ierror)
type(*), dimension(..), asynchronous :: location
integer(kind=MPI_ADDRESS_KIND), intent(out) :: address
integer, optional, intent(out) :: ierror
```

ADDRESS ARITHMETIC [MPI-4.0, 5.1.5]

Addition

C
```c
MPI_Aint MPI_Aint_add(MPI_Aint a, MPI_Aint b)
```

F08
```fortran
integer(kind=MPI_ADDRESS_KIND) MPI_Aint_add(a, b)
integer(kind=MPI_ADDRESS_KIND), intent(in) :: a, b
```

Subtraction

C
```c
MPI_Aint MPI_Aint_diff(MPI_Aint a, MPI_Aint b)
```

F08
```fortran
integer(kind=MPI_ADDRESS_KIND) MPI_Aint_diff(a, b)
integer(kind=MPI_ADDRESS_KIND), intent(in) :: a, b
```

EXAMPLE

```c
struct heterogeneous h;
MPI_Aint base, displ[2];
MPI_Datatype newtype;
MPI_Datatype types[2] = { MPI_INT, MPI_DOUBLE };
int blocklen[2] = { 3, 5 };

MPI_Get_address(&h, &base);
MPI_Get_address(&h.i, &displ[0]);
displ[0] = MPI_Aint_diff(displ[0], base);
MPI_Get_address(&h.d, &displ[1]);
displ[1] = MPI_Aint_diff(displ[1], base);

MPI_Type_create_struct(2, blocklen, displ, types, &newtype);
MPI_Type_commit(&newtype);
```

EXAMPLE

F08

```
type(heterogeneous) :: h
integer(kind=MPI_ADDRESS_KIND) :: base, displ(2)
type(MPI_Datatype) :: types(2), newtype
integer :: blocklen(2)

types = (/ MPI_INTEGER, MPI_REAL8 /)
blocklen = (/ 3, 5 /)

call MPI_Get_address(h, base)
call MPI_Get_address(h%i, displ(1))
displ(1) = MPI_Aint_diff(displ(1), base)
call MPI_Get_address(h%d, displ(2))
displ(2) = MPI_Aint_diff(displ(2), base)

call MPI_Type_create_struct(2, blocklen, displ, types, newtype)
call MPI_Type_commit(newtype)
```

EXERCISES

Exercise 14 – Derived Datatypes

14.2 Structs

Given a definition of a datatype that represents a point in three-dimensional space with additional properties:

- 3 color values (`rgb`, integers)
- 3 coordinates (`xyz`, double precision)
- 1 tag (1 character)

write a function `point_datatype` in `struct.{c|cxx|f90}` or `struct_.py` that returns a committed MPI Datatype that describes the data layout. Your function will be tested by using the datatype you construct for copying an instance of the point type.

Modification: Change the order of the components of the point structure. Does your program still produce correct results?

Use: `MPI_Get_address`, `MPI_Aint_diff`, `MPI_Type_create_struct`, `MPI_Type_commit`

SUBARRAY DATA [MPI-4.0, 5.1.3]

C

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

F08

```fortran
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
    array_of_starts, order, oldtype, newtype, ierror)
integer, intent(in) :: ndims, array_of_sizes(ndims),
    array_of_subsizes(ndims), array_of_starts(ndims), order
type(MPI_Datatype), intent(in) :: oldtype
type(MPI_Datatype), intent(out) :: newtype
integer, optional, intent(out) :: ierror
```

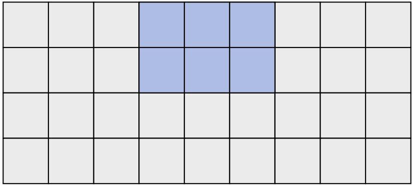
2D dynamic allocation will introduce additional padding in between the ‘rows’, which causes unforeseen offsets if not carefully treated.

JÜLICH Forschungszentrum

EXAMPLE

```
ndims = 2;
array_of_sizes[] = { 4, 9 };
array_of_subsizes[] = { 2, 3 };
array_of_starts[] = { 0, 3 };
order = MPI_ORDER_C;
oldtype = MPI_INT;
```

An array with global size 4×9 containing a subarray of size 2×3 at offsets $0, 3$:

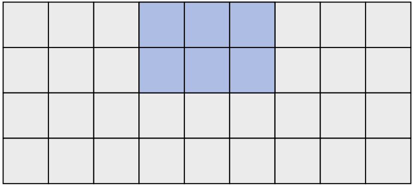


EXAMPLE

F08

```
ndims = 2
array_of_sizes(:) = (/ 4, 9 /)
array_of_subsizes(:) = (/ 2, 3 /)
array_of_starts(:) = (/ 0, 3 /)
order = MPI_ORDER_FORTRAN
oldtype = MPI_INTEGER
```

An array with global size 4×9 containing a subarray of size 2×3 at offsets $0, 3$:



QUIZ

How do MPI_ORDER_{C | FORTRAN} typically corresponds to "row-major" and "column-major" orders?

1. Both correspond to "row-major"
2. MPI_ORDER_C corresponds to "row-major", MPI_ORDER_FORTRAN corresponds to "column-major"
3. MPI_ORDER_C corresponds to "column-major", MPI_ORDER_FORTRAN corresponds to "row-major"
4. Both correspond to "column-major"

TYPE EXTENT [MPI-4.0, 5.1]

Extent

The extent of a type is determined from its lower bounds and upper bounds:

$$
\begin{aligned}
\mathit{Typemap} &= \{(\mathit{type}_0, \mathit{disp}_0), \dots, (\mathit{type}_{n-1}, \mathit{disp}_{n-1})\} \\
\mathrm{lb}\ \mathit{Typemap} &= \min_j \mathit{disp}_j \\
\mathrm{ub}\ \mathit{Typemap} &= \max_j (\mathit{disp}_j + \mathrm{sizeof}\ \mathit{type}_j) + \epsilon \\
\mathrm{extent}\ \mathit{Typemap} &= \mathrm{ub}\ \mathit{Typemap} - \mathrm{lb}\ \mathit{Typemap}
\end{aligned}
$$

Extent and spacing

Let `t` be a type with type map `{(MPI_CHAR, 1)}` and `b` an array of **char**, `b = { 'a', 'b', 'c', 'd', 'e', 'f' }`, then `MPI_Send(b, 3, t, ...)` will result in a message `{'b', 'c', 'd'}` and not `{'b', 'd', 'f'}`.

Explicit padding can be added by resizing the type.

RESIZE [MPI-4.0, 5.1.7]

C

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

F08

```fortran
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
integer(kind=MPI_ADDRESS_KIND), intent(in) :: lb, extent
type(MPI_Datatype), intent(in) :: oldtype
type(MPI_Datatype), intent(out) :: newtype
integer, optional, intent(out) :: ierror
```

Creates a new derived type `newtype` with the same type map as `oldtype` but explicit lower bound `lb` and explicit upper bound `lb + extent`.

Extent and true extent of a type can be queried using `MPI_Type_get_extent` and `MPI_Type_get_true_extent`. The size of resulting messages can be queried with `MPI_Type_size`.

MESSAGE ASSEMBLY

Buffer	0	1	2	3	4	5	6	7	8	9	...

```
MPI_Send(buffer, 4, {(MPI_INT, 0), (ub, 8)}, ...)
```

Message	0	2	4	6

```
MPI_Recv(buffer, 4, {(MPI_INT, 0)}, ...)
```

Buffer	0	2	4	6	?	?	?	?	...

LARGE COUNT EXAMPLE

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

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

LARGE COUNT EXAMPLE

F08

```
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
 ↪  ierror)
integer, intent(in) :: count, blocklength
integer(kind=MPI_ADDRESS_KIND), intent(in) :: stride
type(MPI_Datatype), intent(in) :: oldtype
type(MPI_Datatype), intent(out) :: newtype
integer, optional, intent(out) :: ierror

MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
 ↪  ierror)
integer(kind=MPI_COUNT_KIND), intent(in) :: count, blocklength, stride
type(MPI_Datatype), intent(in) :: oldtype
type(MPI_Datatype), intent(out) :: newtype
integer, optional, intent(out) :: ierror
```

EXERCISES

Exercise 14 – Derived Datatypes

14.3 Matrix Access – Upper Triangle

In the file `matrix_access.{c|cxx|f90|py}` implement the function/subroutine `get_upper` that copies all elements on or above the diagonal of an $N \times N$ matrix to a second matrix and leaves all other elements untouched.

$$\texttt{upper}_{i,j} = \texttt{matrix}_{i,j}, \quad i = 1 \ldots N, j = i \ldots N$$

As in the previous exercise, do not access the matrix elements directly and assume row-major layout of the matrices in C, column-major order in Fortran. Make sure to un-comment the call to `test_get_upper()` to have your solution tested.

Hint: `MPI_Sendrecv` on the `MPI_COMM_SELF` communicator can be used for copying the data.

Use: `MPI_Type_indexed`

JÜLICH Forschungszentrum