

Advanced Parallel Programming

Derived Datatypes

Dr David Henty
HPC Training and Support Manager
d.henty@epcc.ed.ac.uk
+44 131 650 5960

- Lecture will cover
 - derived datatypes
 - memory layouts
 - vector datatypes
 - floating vs fixed datatypes
 - subarray datatypes

$x[i][j]$

j



i

$x(i, j)$

| | | | |
|-----------|-----------|-----------|-----------|
| $x[0][3]$ | $x[1][3]$ | $x[2][3]$ | $x[3][3]$ |
| $x[0][2]$ | $x[1][2]$ | $x[2][2]$ | $x[3][2]$ |
| $x[0][1]$ | $x[1][1]$ | $x[2][1]$ | $x[3][1]$ |
| $x[0][0]$ | $x[1][0]$ | $x[2][0]$ | $x[3][0]$ |

| | | | |
|-----------|-----------|-----------|-----------|
| $x(1, 4)$ | $x(2, 4)$ | $x(3, 4)$ | $x(4, 4)$ |
| $x(1, 3)$ | $x(2, 3)$ | $x(3, 3)$ | $x(4, 3)$ |
| $x(1, 2)$ | $x(2, 2)$ | $x(3, 2)$ | $x(4, 2)$ |
| $x(1, 1)$ | $x(2, 1)$ | $x(3, 1)$ | $x(4, 1)$ |

- MPI has a number of pre-defined datatypes
 - eg **MPI_INT / MPI_INTEGER, MPI_FLOAT / MPI_REAL**
 - user passes them to send and receive operations
- For example, to send 4 integers from an array x

C: `int[10];`

F: `INTEGER x(10)`



`MPI_Send(x, 4, MPI_INT, ...);`

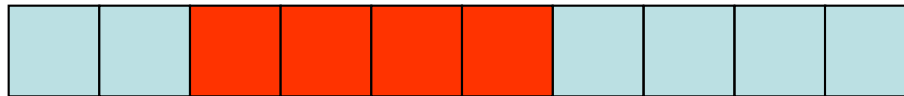
`MPI_SEND(x, 4, MPI_INTEGER, ...)`



- Can send different data by specifying different buffer

```
MPI_Send (&x [2] , 4 , MPI_INT , ... ) ;
```

```
MPI_SEND (x (3) , 4 , MPI_INTEGER , ... )
```



- but can only send a single block of contiguous data

- Can define new datatypes called *derived types*
 - various different options in MPI
 - we will use them to send data with gaps in it: a **vector type**
 - other MPI derived types correspond to, for example, C structs

- Contiguous type

```
MPI Datatype my_new_type;  
MPI_Type_contiguous(count=4, oldtype=MPI_INT, newtype=&my_new_type);  
MPI_Type_commit(&my_new_type);
```

```
INTEGER MY_NEW_TYPE  
CALL MPI_TYPE_CONTIGUOUS(4, MPI_INTEGER, MY_NEW_TYPE, IERROR)  
CALL MPI_TYPE_COMMIT(MY_NEW_TYPE, IERROR)
```

```
MPI_Send(x, 1, my_new_type, ...);
```

```
MPI_SEND(x, 1, MY_NEW_TYPE, ...)
```



- Vector types correspond to patterns such as



C: $\mathbf{x}[16]$

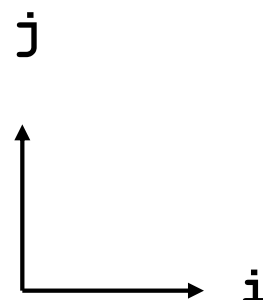
F: $\mathbf{x}(16)$

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|

C: $\mathbf{x}[4][4]$

F: $\mathbf{x}(4,4)$

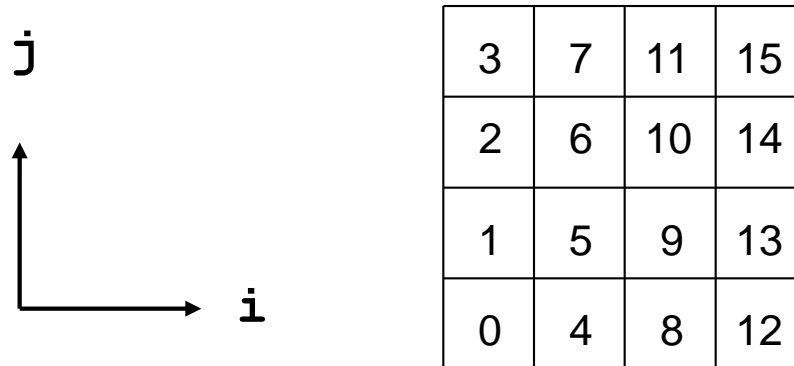
| | | | |
|---|---|----|----|
| 4 | 8 | 12 | 16 |
| 3 | 7 | 11 | 15 |
| 2 | 6 | 10 | 14 |
| 1 | 5 | 9 | 13 |



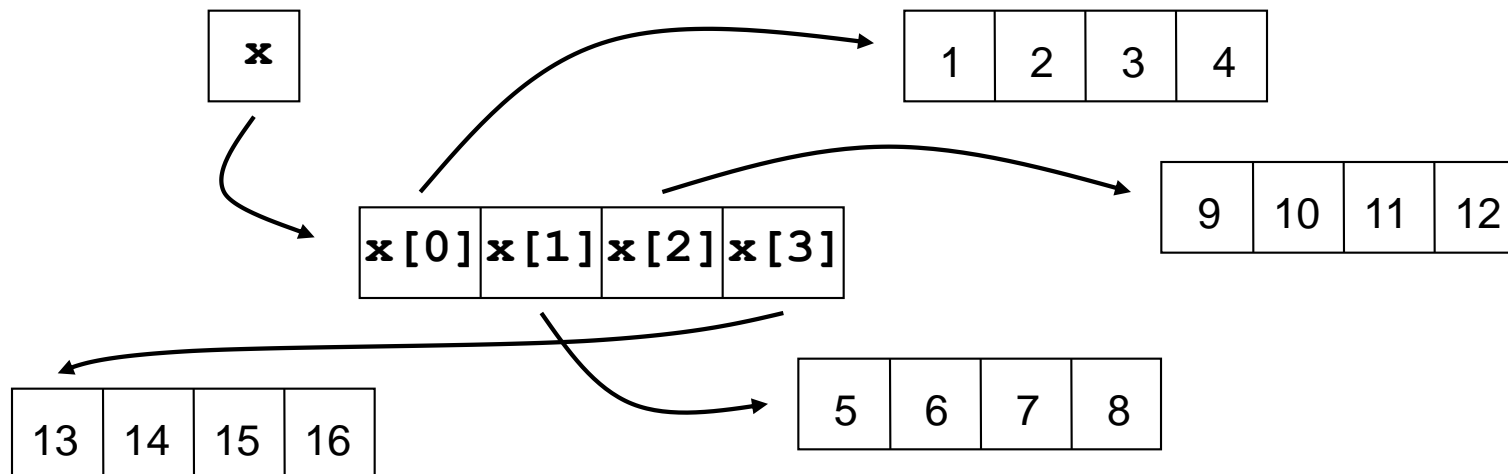
| | | | |
|----|----|----|----|
| 13 | 14 | 15 | 16 |
| 9 | 10 | 11 | 12 |
| 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 |

- Data is contiguous in memory
 - different conventions in C and Fortran
 - for statically allocated C arrays $\mathbf{x} == \&\mathbf{x}[0][0]$

- I use C convention for *process* coordinates, even in Fortran
 - ie processes always ordered as for C arrays
 - and array indices also start from 0
- Why?
 - this is what is returned by MPI for cartesian topologies
 - turns out to be convenient for future exercises
- Example: *process* rank layout on a 4x4 *process* grid
 - rank 6 is at position (1,2), ie $i = 1$ and $j = 2$, for C and Fortran

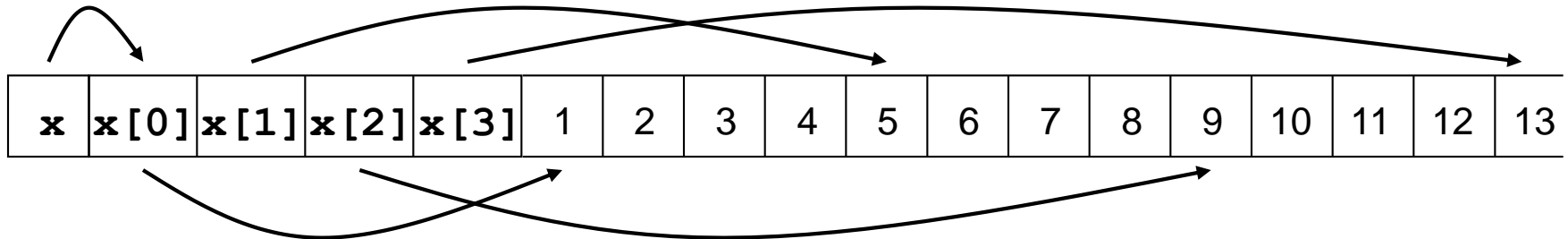



```
float **x = (float **) malloc(4, sizeof(float *));  
  
for (i=0; i < 4; i++)  
{  
    x[i] = (float *) malloc(4, sizeof(float));  
}
```



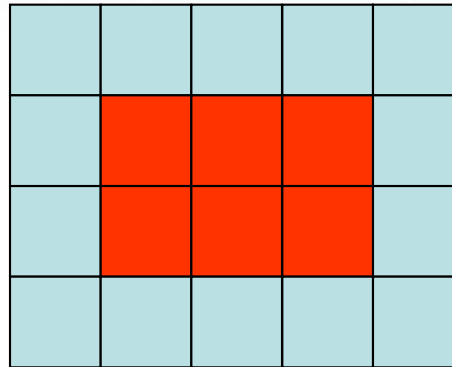
- Data non-contiguous, and `x != &x[0][0]`
 - cannot use regular templates such as vector datatypes
 - cannot pass `x` to any MPI routine

```
float **x = (float **) arralloc(sizeof(float), 2, 4, 4);
/* do some work */
free((void *) x);
```

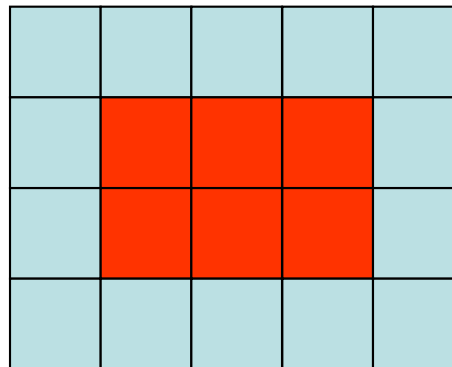


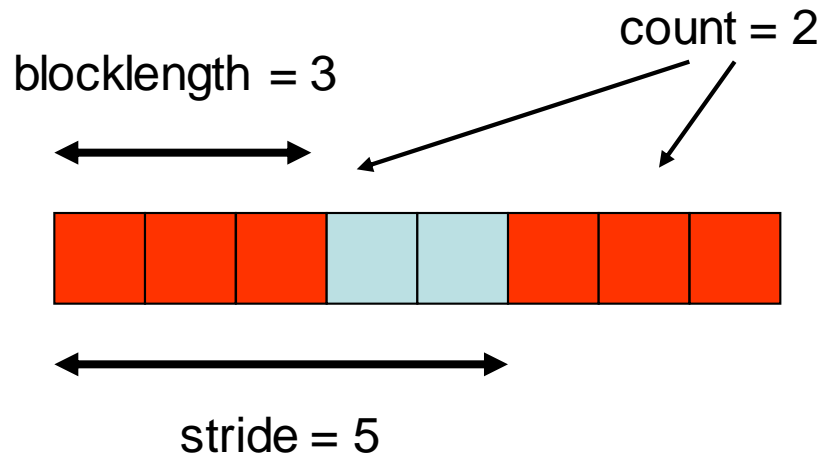
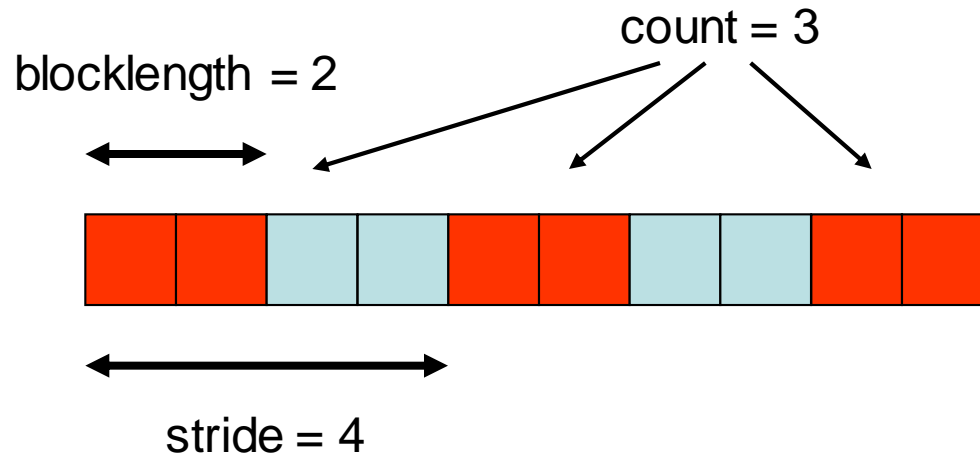
- Data is now contiguous, but still `x != &x[0][0]`
 - can now use regular template such as vector datatype
 - must pass `&x[0][0]` (start of contiguous data) to MPI routines
 - see `PSMA-arralloc.tar` for example of use in practice
- Will illustrate all calls using `&x[i][j]` syntax
 - correct for both static and (contiguously allocated) dynamic arrays

C: $\mathbf{x}[5][4]$



F: $\mathbf{x}(5, 4)$





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

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE,  
                OLDTYPE, NEWTYPE, IERR)
```

```
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE  
INTEGER NEWTYPE, IERR
```

```
MPI_Datatype vector3x2;
```

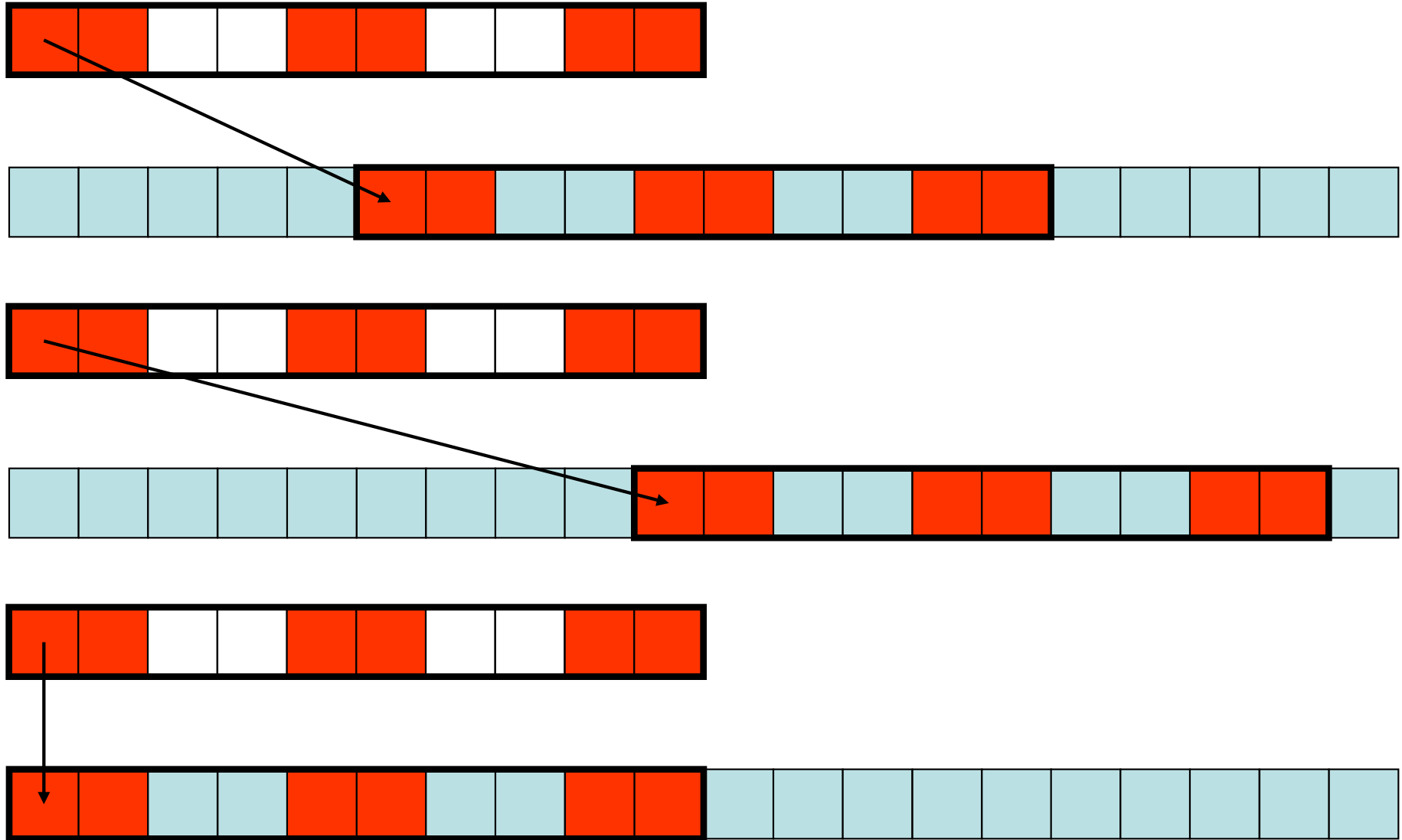
```
MPI_Type_vector(3, 2, 4, MPI_FLOAT, &vector3x2)
```

```
MPI_Type_commit(&vector3x2)
```

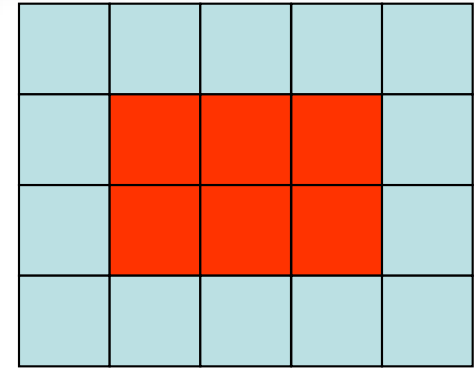
```
integer vector3x2
```

```
call MPI_TYPE_VECTOR(2, 3, 5, MPI_REAL, vector3x2, ierr)
```

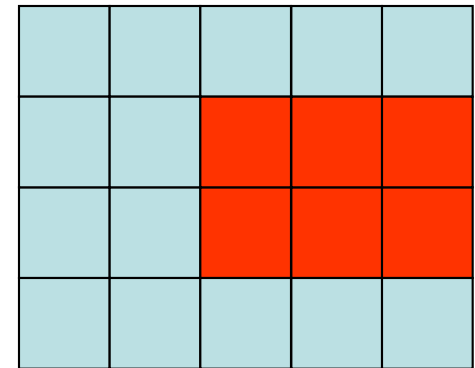
```
call MPI_TYPE_COMMIT(vector3x2, ierr)
```

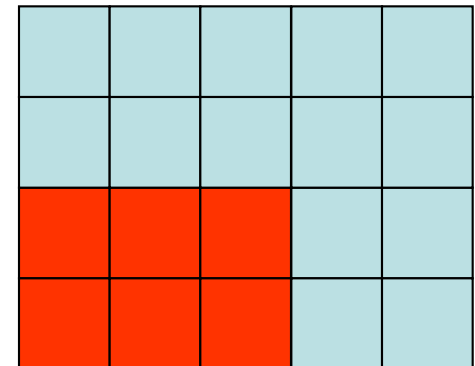
```
MPI_Send(&x[1][1], 1, vector3x2, ...);  
MPI_SEND(x(2,2), 1, vector3x2, ...)
```



```
MPI_Send(&x[2][1], 1, vector3x2, ...);  
MPI_SEND(x(3,2), 1, vector3x2, ...)
```



```
MPI_Send(&x[0][0], 1, vector3x2, ...);  
MPI_SEND(x(1,1), 1, vector3x2, ...)
```



- When sending multiple datatypes
 - datatypes are read from memory separated by their extent
 - for basic datatypes, extent is the size of the object
 - for vector datatypes, extent is distance from first to last data



extent = 10*extent(basic type)

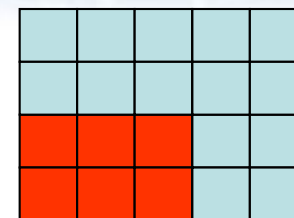


extent = 8*extent(basic type)

- Extent *does not* include trailing spaces

```
MPI_Send(&x[0][0], 1, vector3x2, ...);
```

```
MPI_SEND(x(1,1), 1, vector3x2, ...)
```



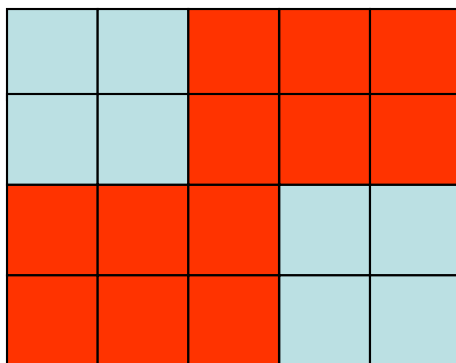
```
MPI_Send(&x[0][0], 2, vector3x2, ...);
```



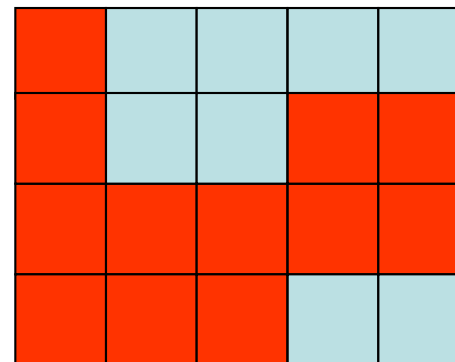
```
MPI_SEND(x(1,1), 2, vector3x2, ...)
```



C

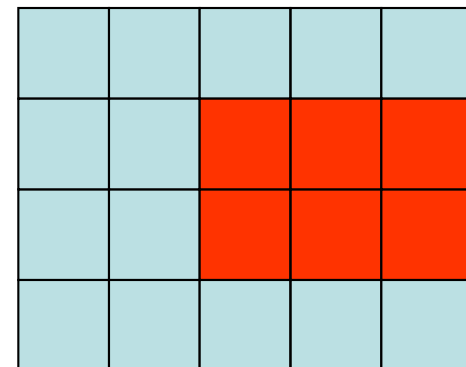


F



- Sending multiple vectors is not often useful
 - extents are not defined as you might expect for 2D arrays
- A 3D array subsection is not a vector
 - but cannot easily use 2D vectors as building blocks due to extents
 - becomes even harder for higher-dimensional arrays
- It is possible to set the extent manually
 - routine is called `MPI_Type_create_resized`
 - this is not a very elegant solution

- Vectors are floating datatypes
 - this may have some advantages, eg define a single halo datatype and use for both up and down halos
 - actual location is selected by passing address of appropriate element
 - equivalent in MPI-IO is specifying a displacement into the file
 - this will turn out to be rather clumsy
- Fixed datatype
 - always pass starting address of array
 - datatype encodes both the shape and position of the subarray
- How do we define a fixed datatype?
 - requires a datatype with leading spaces
 - difficult to do with vectors



- A single call that defines multi-dimensional subsections
 - much easier than vector types for 3D arrays
 - datatypes are fixed
 - pass the starting address of the array to all MPI calls

```
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, IERR)
```

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

```
#define NDIMS 2
MPI_Datatype subarray3x2;
int array_of_sizes[NDIMS], array_of_subsizes[NDIMS],
    arrays_of_starts[NDIMS];

array_of_sizes[0]      = 5; array_of_sizes[1]      = 4;
array_of_subsizes[0]  = 3; array_of_subsizes[1]    = 2;
array_of_starts[0]    = 2; array_of_starts[1]      = 1;

order = MPI_ORDER_C;

MPI_type_create_subarray(NDIMS, array_of_sizes,
    array_of_subsizes, array_of_starts, order,
    MPI_FLOAT, &subarray3x2);
MPI_TYPE_COMMIT(&subarray3x2);
```

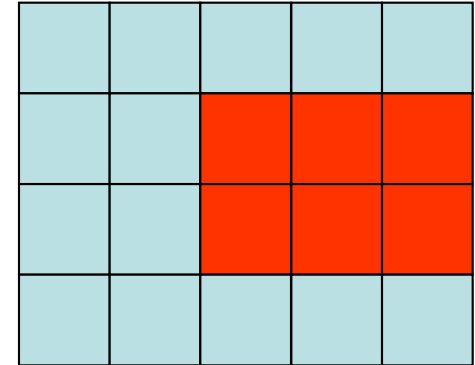
```
integer, parameter :: ndims = 2
integer subarray3x2
integer, dimension(ndims) :: array_of_sizes, array_of_subsizes,
                             arrays_of_starts
! Indices start at 0 as in C !

array_of_sizes(1)      = 5; array_of_sizes(2)      = 4
array_of_subsizes(1)  = 3; array_of_subsizes(2)    = 2
array_of_starts(1)    = 2; array_of_starts(2)      = 1

order = MPI_ORDER_FORTRAN

call MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes,
                              array_of_subsizes, array_of_starts, order,
                              MPI_REAL, subarray3x2, ierr)
call MPI_TYPE_COMMIT(subarray3x2, ierr)
```

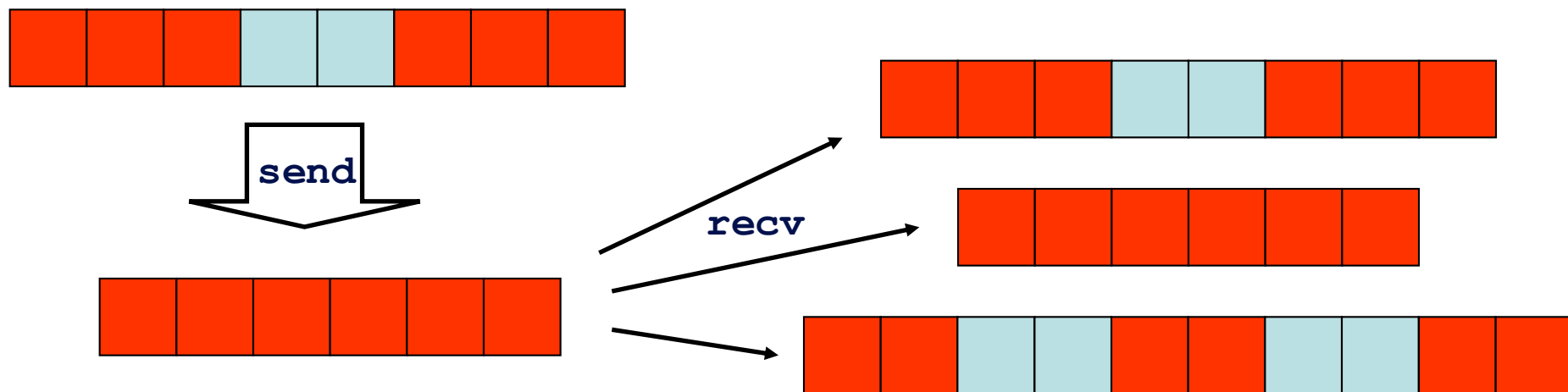
```
MPI_Send (&x[0][0], 1, subarray3x2, ...);  
MPI_SEND (x, 1, subarray3x2, ...)  
MPI_SEND (x(1,1), 1, subarray3x2, ...)
```



- Generalisation to IO

- each process counts from the start of the file
- actual displacements from file origin depend on the position of the process in the process array
- this is all already encoded in the datatype

- A datatype is defined by two attributes:
 - type signature: a list of the basic datatypes in order
 - type map: the locations (displacements) of each basic datatype
- For a receive to match a send only signatures need to match
 - type map is defined by the receiving datatype
- Think of messages being packed for transmission by sender
 - and independently unpacked by the receiver



- There is an overhead to defining a derived type
 - a real code may have many calls to the IO routines
 - no need to re-define the data types every time
 - array sizes unlikely to change: define types once at start of program
- If you do create lots of derived types in a program ...
 - they take up memory!
 - clear up the memory using `MPI_Type_free` whenever possible
- But try and avoid:
 - do loop = 1, 1000000
 - do stuff
 - define type
 - use type
 - free type
 - end do