Derived Datatypes

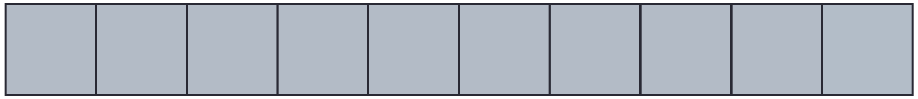
Reusing this material

MPI Datatypes

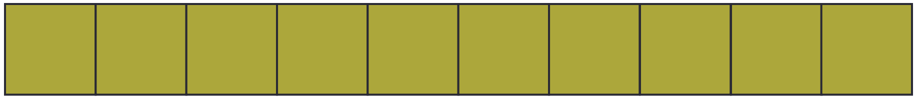
- Basic types
- Derived types
  - vectors
  - structs
  - others

Basic datatypes

```
int x[10];
INTEGER:: x(10);
```

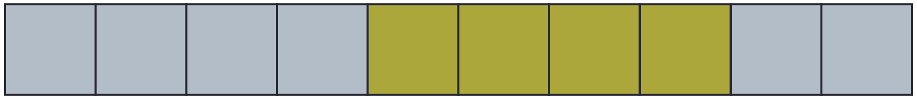


```
// send all 10 values
MPI_Send(x, 10, MPI_INT, …);
MPI_SEND(x, 10, MPI_INTEGER, …)
```

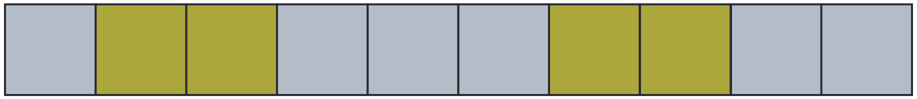


```
// send first 4 values
MPI_Send(&x[0], 4, …);
MPI_SEND( x(1), 4, …)
```

```
// send 5th, 6th, 7th, 8th
MPI_Send(&x[4], 4, …);
MPI_SEND( x(5), 4, …)
```



```
// ??
struct mystruct x[10];
type(mytype) :: x(10)
```



Motivation

- Send / Recv calls need a datatype argument
  - pre-defined values exist for pre-defined language types
  - e.g. **`real`** `<->` **`MPI_REAL`**; **`int`** `<->` **`MPI_INT`**

- What about types defined by a program?
  - e.g. structures (in C) or user-defined types (Fortran)

- **`Send`** / **`Recv`** calls take a count parameter
  - what about data that isn’t contiguous in memory?
  - e.g. subsections of 2D arrays

Approach

- Can define new types in MPI
  - user calls setup routines to describe new data type to MPI
    - remember, MPI is a library and NOT a compiler!
  - MPI returns a new data type handle
  - store this value in a variable, e.g. **`MPI_MY_NEWTYPE`**

- Derived types have same status as pre-defined
  - can use in any message-passing call

- Some care needed for reduction operations
  - user must also define a new **`MPI_Op`** appropriate to the new data type to tell MPI how to combine them

Defining types

- All derived types stored by MPI as a list of basic types and displacements (in bytes)
  - for a structure, types may be different
  - for an array subsection, types will be the same

- User can define new derived types in terms of both basic types and other derived types

Derived Data types - Type

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

Contiguous Data

- The simplest derived datatype consists of a number of contiguous items of the same datatype.
- C:

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

- Fortran:

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

INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

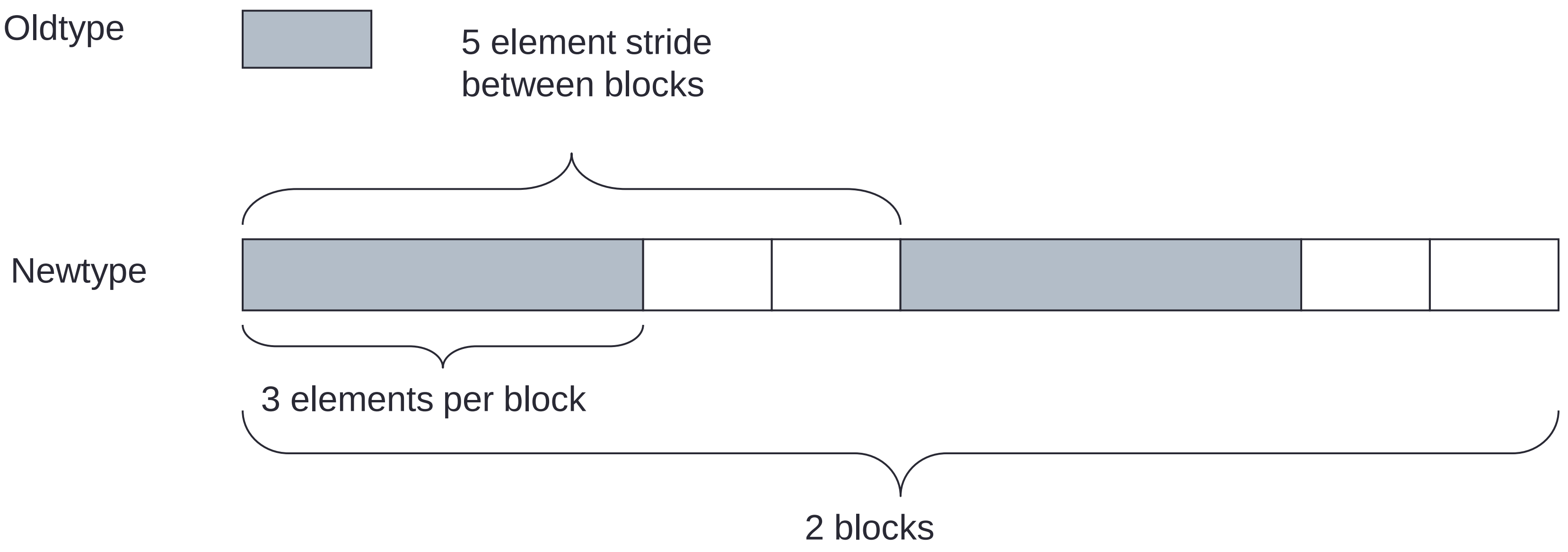
Use of contiguous

- May make program clearer to read
- Imagine sending a block of 4 integers
  - use **`MPI_Ssend`** with **`MPI_INT`** / **`MPI_INTEGER`** and **`count = 4`**

- Or …
  - define a new contiguous type of 4 integers called **`BLOCK4`**
  - use **`MPI_Ssend`** with **`type=BLOCK4`** and **`count = 1`**

- May also be useful intermediate stage in building more complicated types
  - i.e. later used in definition of another derived type

Vector Datatype Example



- count = 2
- stride = 5
- blocklength = 3

What is a vector type?

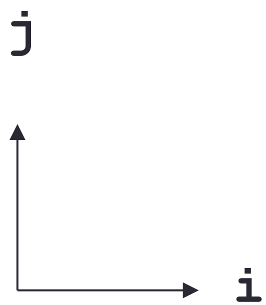
- Why is a pattern with blocks and gaps useful?

A vector type corresponds to a subsection of a 2D array

- Think about how arrays are stored in memory
  - unfortunately, different conventions for C and Fortran!
  - must use statically allocated arrays in C because dynamically allocated arrays (using **`malloc`**) have no defined storage format
  - In Fortran, can use either static or allocatable arrays

Coordinate System (how I draw arrays)

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(i,j)

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)

Arrray Layout in Memory

C: `x[16]`

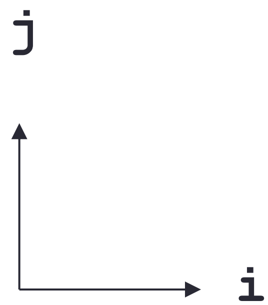
F: `x(16)`

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

C: `x[4][4]`

F: `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 for mapping 2D to 1D arrays in C and Fortran

Memory Layout

- You can choose to draw arrays however you like – how you draw them does not change reality!

First index i	Second index j	Format
right	up	coordinates
down	right	matrix
right	down	graphics (scan lines)

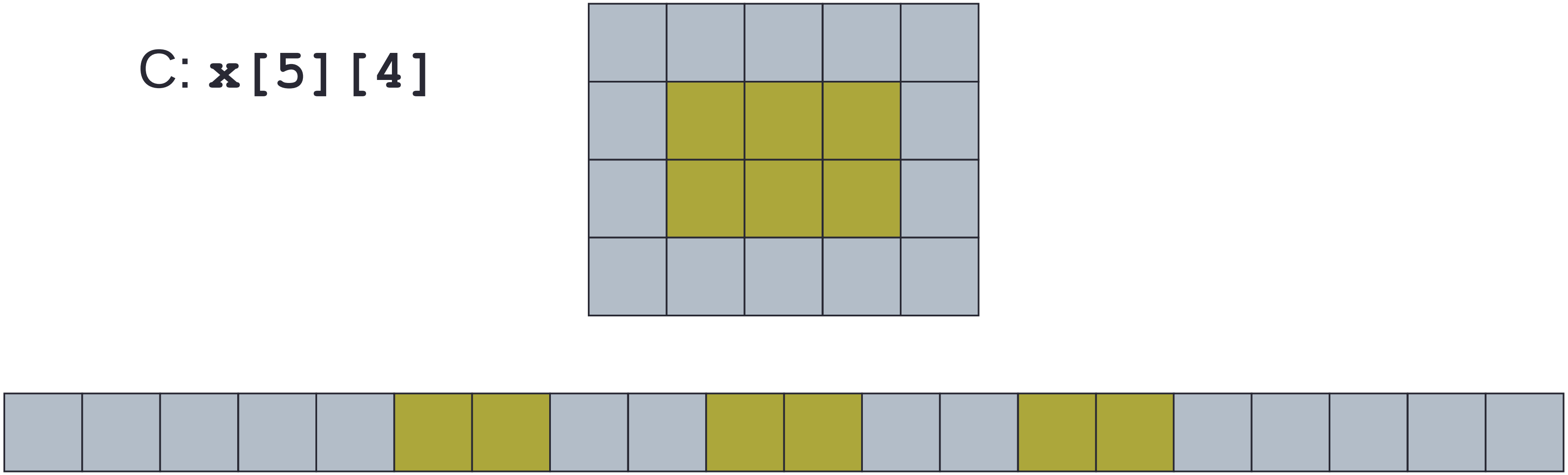
- Regardless of how you draw them, the layout in memory is:
  - `x[i][j]` is followed by `x[i][j+1]` (in C)
  - `x(i,j)` is followed by `x(i+1,j)` (in Fortran)

  - if you create arrays with malloc in C/C++ things are more complicated ...

- Depending on how you draw them, this can appear “row major” or “column major”

C example

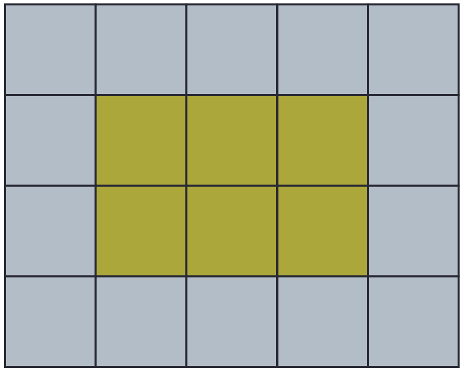
C: **`x[5][4]`**

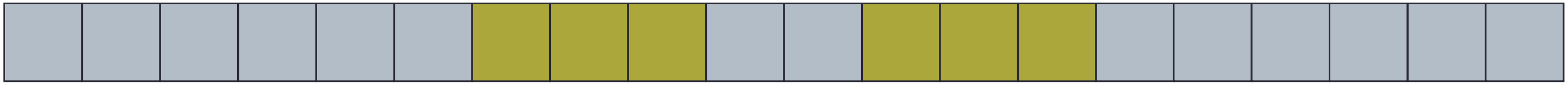


- A 3 x 2 subsection of a 5 x 4 array
  - three blocks of two elements separated by gaps of two

Fortran example

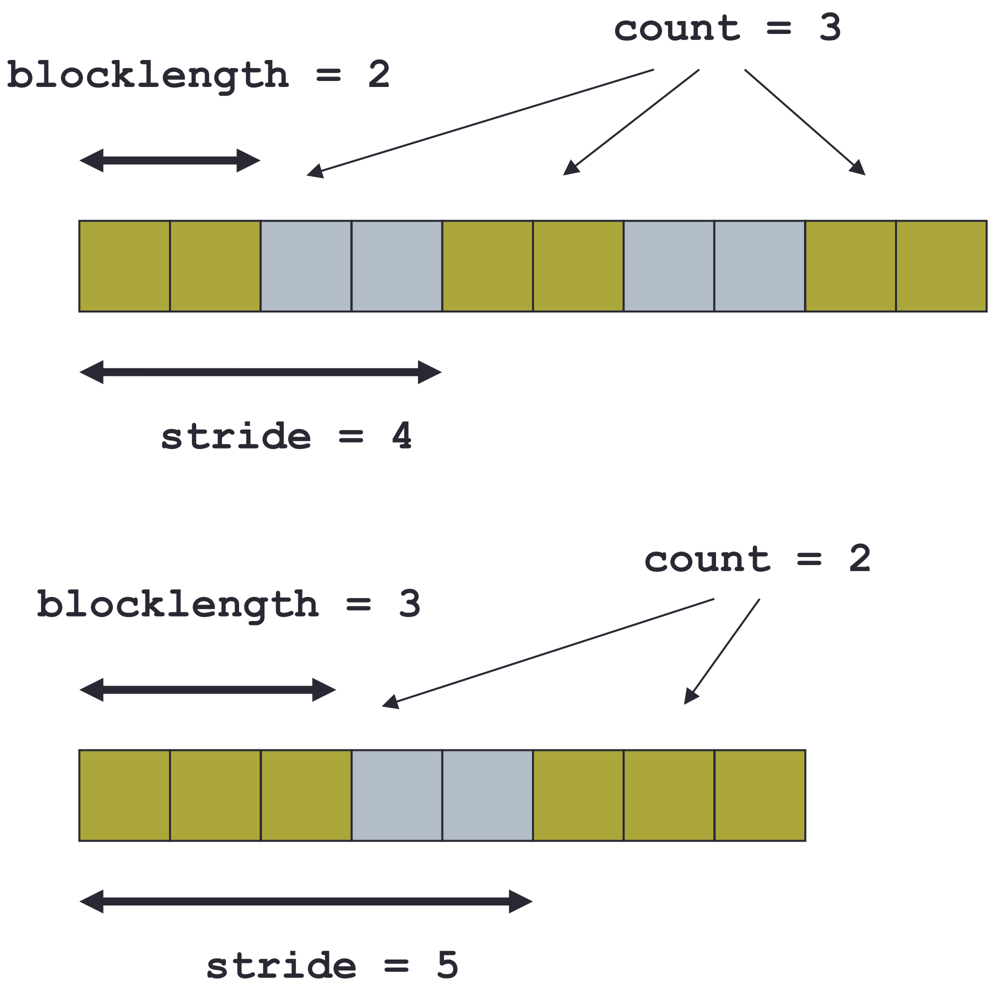
F: **`x(5,4)`**





- A 3 x 2 subsection of a 5 x 4 array
  - two blocks of three elements separated by gaps of two

Equivalent Vector Datatypes



Constructing a Vector Datatype

- C:

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

- Fortran:

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

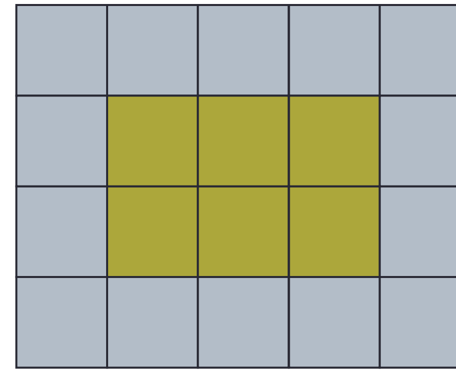
Sending a vector

- Have defined a **`3x2`** subsection of a **`5x4`** array
  - but not defined WHICH subsection
  - is it the bottom left-hand corner? top-right?

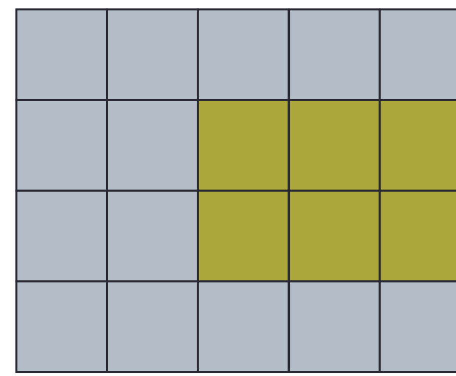
- Data that is sent depends on what buffer you pass to the send routines
  - pass the address of the first element that should be sent

Vectors in send routines

```
MPI_Ssend(&x[1][1], 1, vector3x2, ...);
MPI_SSEND(x(2,2)   , 1, vector3x2, ...)
```



```
MPI_Ssend(&x[2][1], 1, vector3x2, ...);
MPI_SSEND(x(3,2)   , 1, vector3x2, ...)
```



Extent of a Datatatype

- May be useful to find out how big a derived type is
  - extent is distance from start of first to end of last data entry
  - can use these routines to compute extents of basic types too
  - answer is returned in bytes

- C:

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

- Fortran:

```
MPI_TYPE_GET_EXTENT( DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR
```

Structures

- Can define compound objects in C and Fortran

```
struct compound {
  int    ival;
  double dval[3];
};
```

```
type compound
  integer          :: ival
  double precision :: dval(3)
end type compound
```

- Storage format NOT defined by the language
  - different compilers do different things
  - e.g. insert arbitrary padding between successive elements
  - need to tell MPI the byte displacements of every element

Constructing a Struct Datatype

- C:

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

- Fortran:

```
MPI_TYPE_CREATE_STRUCT (COUNT,
     ARRAY_OF_BLOCKLENGTHS,
     ARRAY_OF_DISPLACEMENTS,
     ARRAY_OF_TYPES, NEWTYPE, IERROR)
```

Struct Datatype Example

- count = 2
- array_of_blocklengths[0] = 1
- array_of_types[0] = MPI_INT
- array_of_blocklengths[1] = 3
- array_of_types[1] = MPI_DOUBLE

- But how do we compute the displacements?
  - need to create a compound variable in our program
  - explicitly compute memory addresses of every member
  - subtract addresses to get displacements from origin

Address of a Variable

- C:

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

- Fortran:

```
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)

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

Committing a datatype

- Once a datatype has been constructed, it needs to be committed before it is used in a message-passing call
- This is done using **MPI_TYPE_COMMIT**

- C:

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

- Fortran:

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

Exercise

- See Exercise 7.1 on the sheet
- Modify the passing-around-a-ring exercise.
- Calculate two separate sums:
  - rank integer sum, as before
  - rank floating point sum
- Use a **`struct datatype`** for this.
- If you are a Fortran programmer unfamiliar with Fortran derived types then jump to exercise 7.2
  - illustrates the use of **`MPI_Type_vector`**