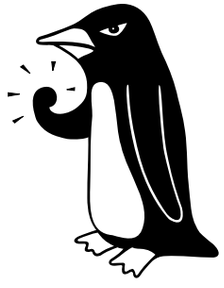
MPI Custom Types

Chris Brady
Heather Ratcliffe



Warwick RSE

Concept

- Primitive MPI datatypes are things like MPI_INT or MPI_FLOAT

- They tell MPI how much data to send for every element

- Obvious extension is to tell it to send more data

- Can do a **lot** more too

Ring pass with *nitems* elements

```
CALL MPI_Sendrecv(values, nitems, MPI_INTEGER, right, tag, values_recv, &
    nitems, MPI_INTEGER, left, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
```

```
MPI_Sendrecv(values, NITEMS, MPI_INT, right, TAG, values_recv, NITEMS,
    MPI_INT, left, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- Very similar to previous ring pass with MPI_Sendrecv
- Now send nitems elements rather than 1
- In C, no longer have "`&values`" and "`&values_recv`" since they are now arrays

MPI_Type_contiguous

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

- `count` - Number of elements of `oldtype` in `newtype`
- `oldtype` - Datatype to base this type off. Usually, but not necessarily a primitive type
- `newtype` - Output containing the newly created type
- Creates a type representing `count` contiguous elements of `oldtype`

Can use my new type?

- No
- Oddly, it might work but you can't rely on that behaviour
- Need to *commit* the type before using it

MPI_Type_commit

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

- `datatype` - Datatype created with a type creation call

- Once committed a custom datatype can be used the same as a normal datatype

  - Slight caveat about MPI_Reduce and MPI_Allreduce

- Opposite of commit is **free**

MPI_Type_free

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

- `datatype` - Datatype created with a type creation call that has been committed

- After a type has been freed it can no longer be used in MPI calls

Ring pass with *nitems* elements

```
CALL MPI_Type_contiguous(nitems, MPI_INTEGER, contig_type, ierr)
CALL MPI_Type_commit(contig_type, ierr)
CALL MPI_Sendrecv(values, 1, contig_type, right, tag, values_recv, &
    1, contig_type, left, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
```

```
MPI_Type_contiguous(NITEMS, MPI_INT, &contig_type);
MPI_Type_commit(&contig_type);
MPI_Sendrecv(values, 1, contig_type, right, TAG, values_recv, 1,
    contig_type, left, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- Create type and then commit it
- Same Sendrecv as before
- Note that **`count`** parameters are back to being 1
- Sending 1 unit of the contiguous type

Why do it?

- Not many reasons for doing it with MPI_Type_contiguous
- Might be faster
  - I've never observed it to be
- In theory needed if you want to send > 4GB (sometimes 2GB) of data
- Notice that size parameters are of type `int`?
- But can send up to 19EB (Exabytes) (probably more by the time needed) by sending types

More useful ones?

- What about if you want to send the red cells above?

- You can create several types that would let you do that

- Simplest is `MPI_Type_indexed`

Indexed array blocks

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

- `count` - number of blocks
- `array_of_blocklengths` - array of length of blocks in multiples of `oldtype`
- `array_of_displacements` - array of starts of blocks in multiples of `oldtype`
- `oldtype` - Type to base custom type on. Usually, but not necessarily a primitive type
- `newtype` - Return value of new indexed type

More useful ones?

```
lengths[0] = 2; lengths[1] = 3;
displacements[0] = 0; displacements[1] = 4;
MPI_Type_indexed(2, lengths, displacements, MPI_INT, &index_type);
MPI_Type_commit(&index_type);
```

```
lengths = (/2, 3/)
displacements = (/0, 4/)
CALL MPI_Type_indexed(2, lengths, displacements, MPI_INTEGER, index_type, &
    ierr)
CALL MPI_Type_commit(index_type, ierr)
```

Mild warning

- There is a more general version of MPI_Type_indexed
- It allows you to specify the offsets in bytes rather than in units of oldtype
- So you can deal with structures of mixed data if you want to
- It's called MPI_Type_create_hindexed
- There *used* to be a function MPI_Type_hindexed
  - It is one of the few MPI **deprecated** functions**.** Do not use it in new code

Mixing types

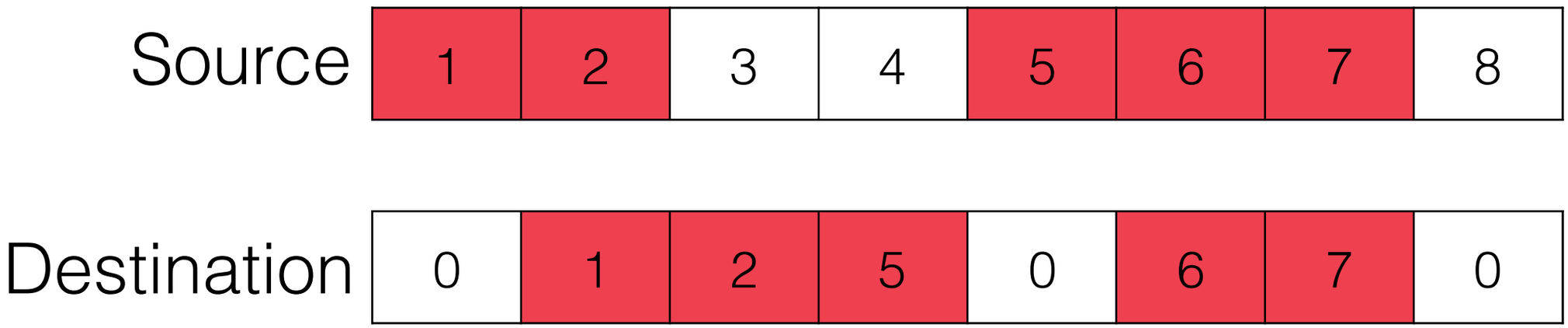
- You can get a lot of power from mixing MPI types
- You can use different types for sending and receiving
- Allows you to remap data during the send/receive process

Mixing types

```
lengths[0] = 2; lengths[1] = 3;
displacements[0] = 0; displacements[1] = 4;
MPI_Type_indexed(2, lengths, displacements, MPI_INT, &index_type);
MPI_Type_commit(&index_type);

lengths[0] = 3; lengths[1] = 2;
displacements[0] = 1; displacements[1] = 5;
MPI_Type_indexed(2, lengths, displacements, MPI_INT, &index_type_recv);
MPI_Type_commit(&index_type_recv);
```

```
lengths = (/2, 3/)
displacements = (/0, 4/)
CALL MPI_Type_indexed(2, lengths, displacements, MPI_INTEGER, index_type, &
    ierr)
CALL MPI_Type_commit(index_type, ierr)

lengths = (/3, 2/)
displacements = (/1, 5/)
CALL MPI_Type_indexed(2, lengths, displacements, MPI_INTEGER, &
    index_type_recv, ierr)
CALL MPI_Type_commit(index_type_recv, ierr)
```

Mixing types



- Source array is set to be 1-8
- Destination starts as all 0
- After comms, destination reads

Back to Case Study

Most useful MPI_Type routine

- The most broadly useful MPI type creation routine is one that allows you to create a type representing a subsection of an array
- Use it in the case study in place of the array temporaries
- Makes the C code much more readable

MPI_Type_create_subarray

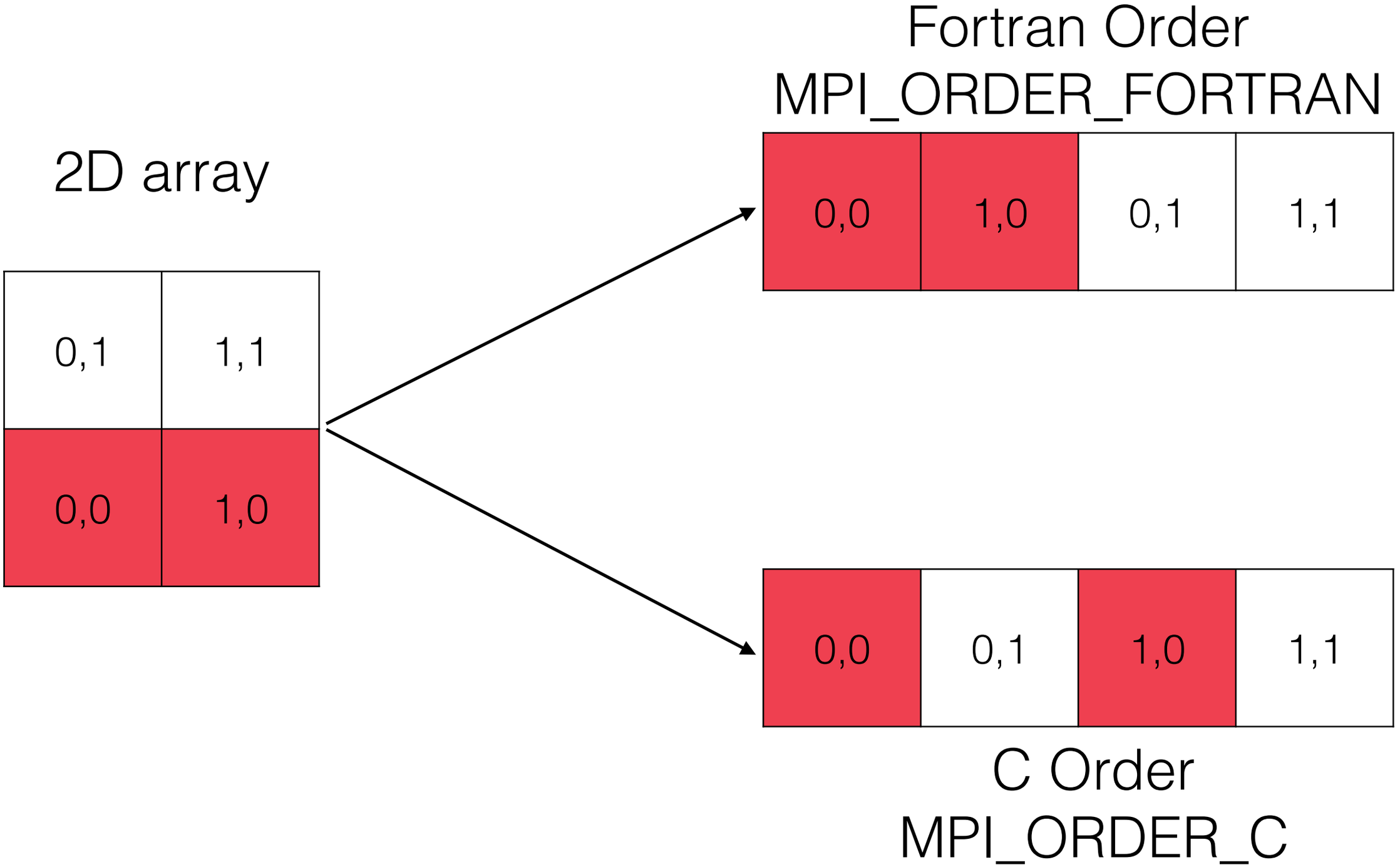
```
int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],
const int array_of_subsizes[], const int array_of_starts[], int order,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- `ndims` - Dimensions of array
- `array_of_sizes` - Sizes of outer array
- `array_of_subsizes` - Size of subarray in outer array
- `array_of_starts` - Offset in each dimension of subarray
- `order` - Whether the array is C or Fortran ordered
- `oldtype` - Type of the underlying data in the array
- `newtype` - Return of new type

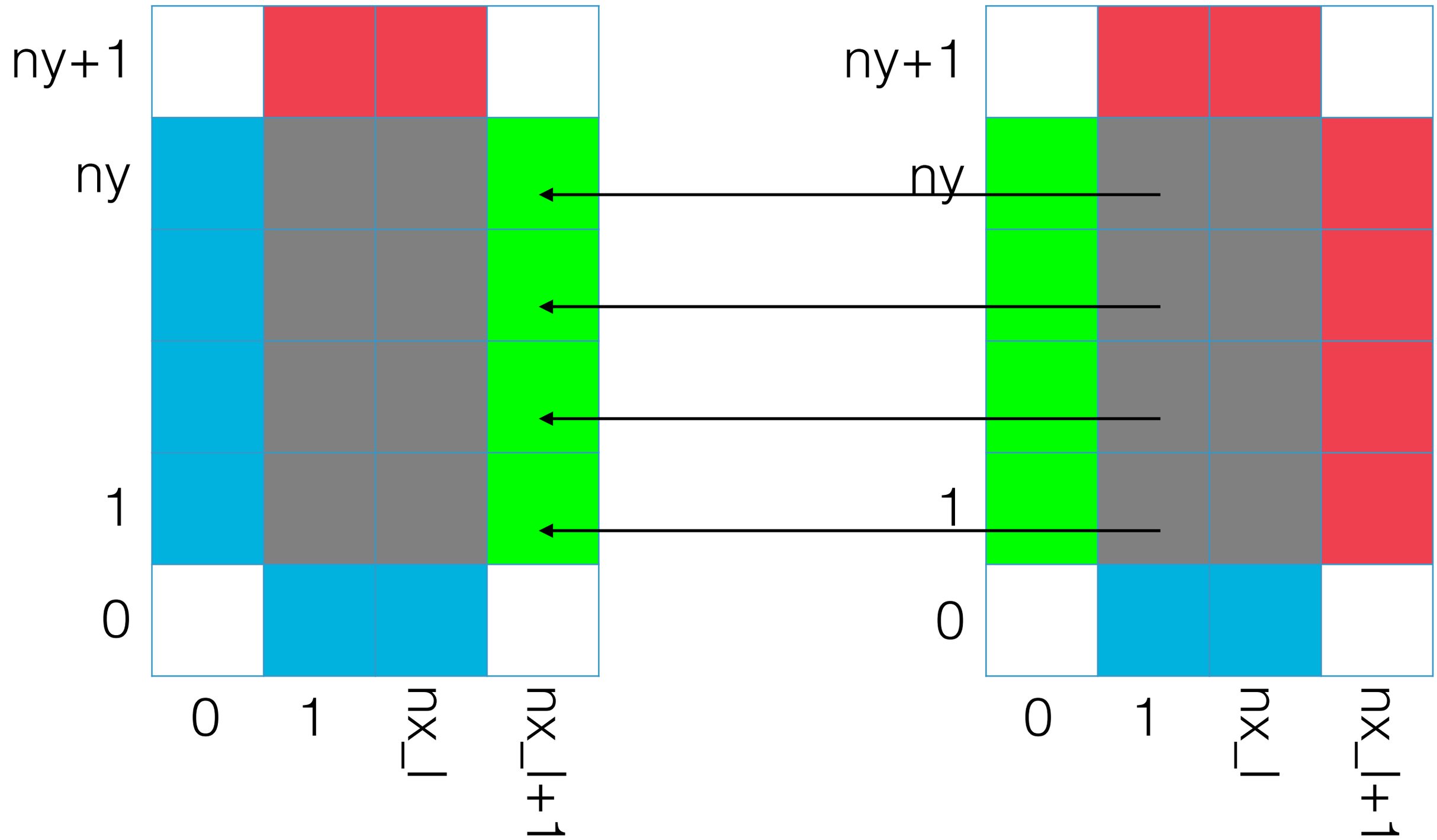
Array order

- By default
  - C is column major order
    - Last index varies fastest
  - Fortran is row major order
    - First index varies fastest
- Both languages can “mock up” arrays ordered the other way round

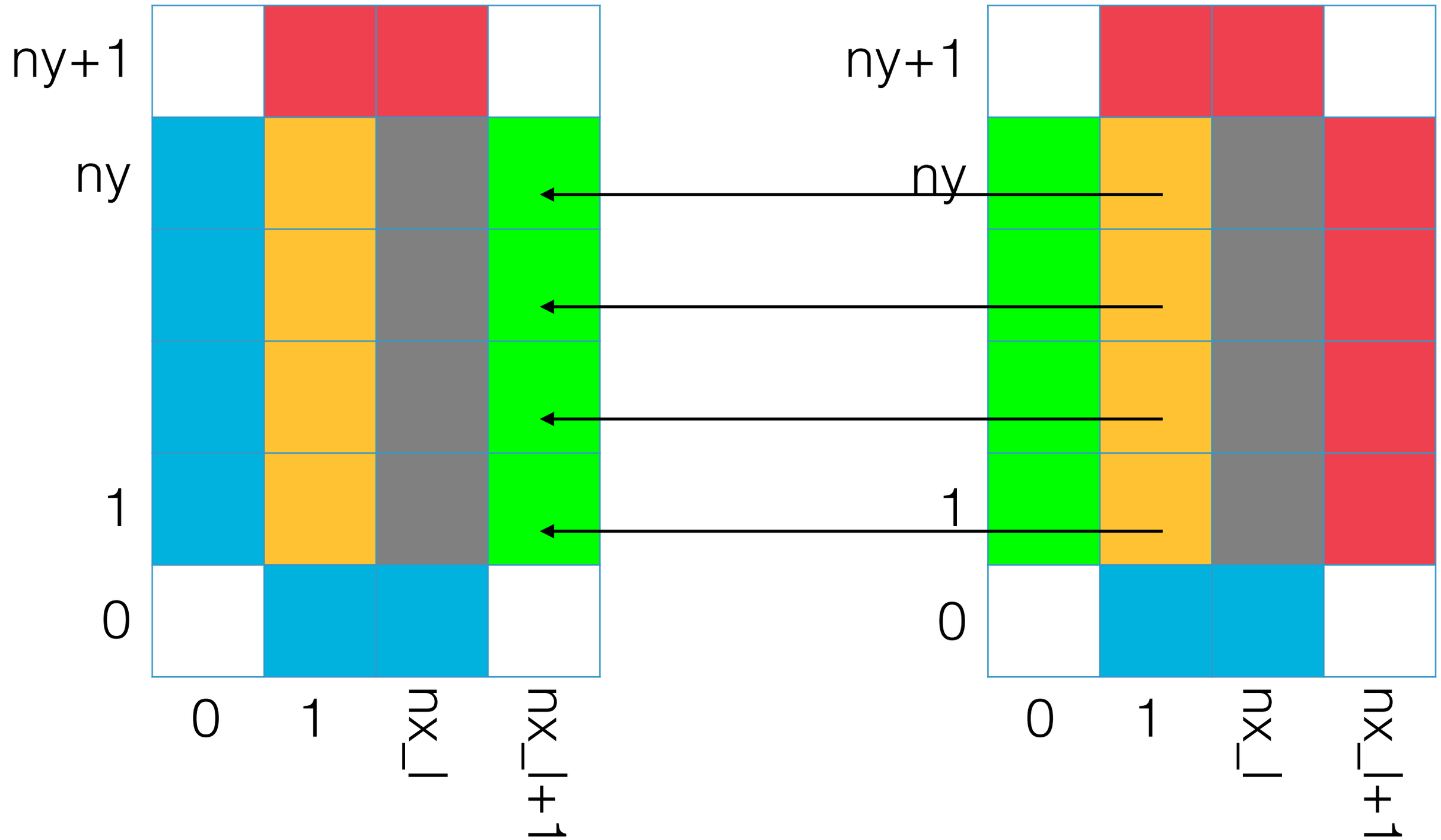
Array order


2D array
0,1
1,1
0,0
1,0
Fortran Order
MPI_ORDER_FORTRAN
0,0
1,0
0,1
1,1
C Order
MPI_ORDER_C
0,0
0,1
1,0
1,1

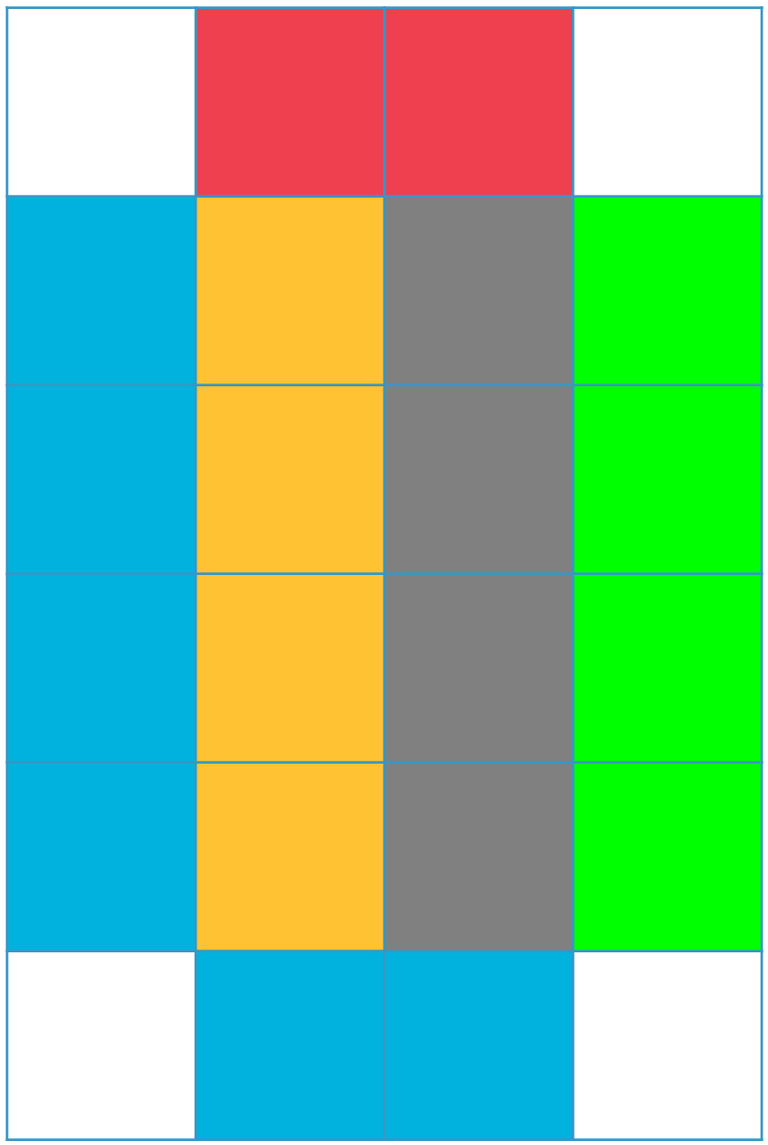
Mapping



Mapping

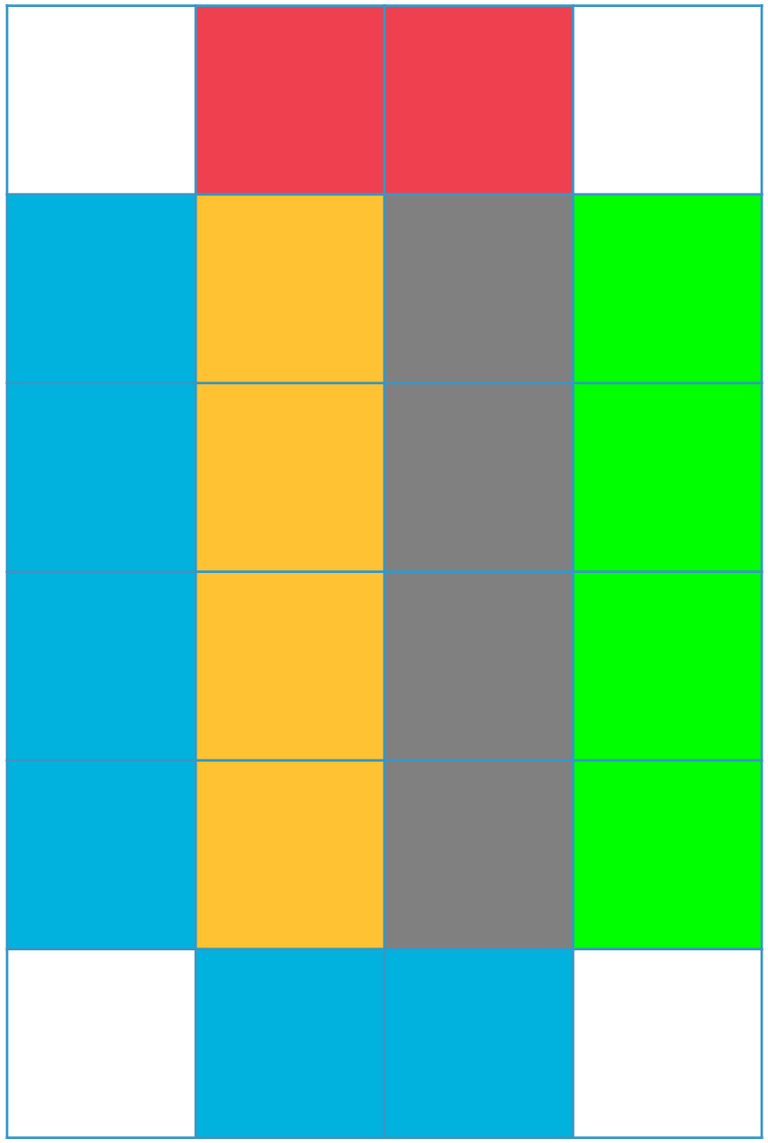


Cells to send and receive



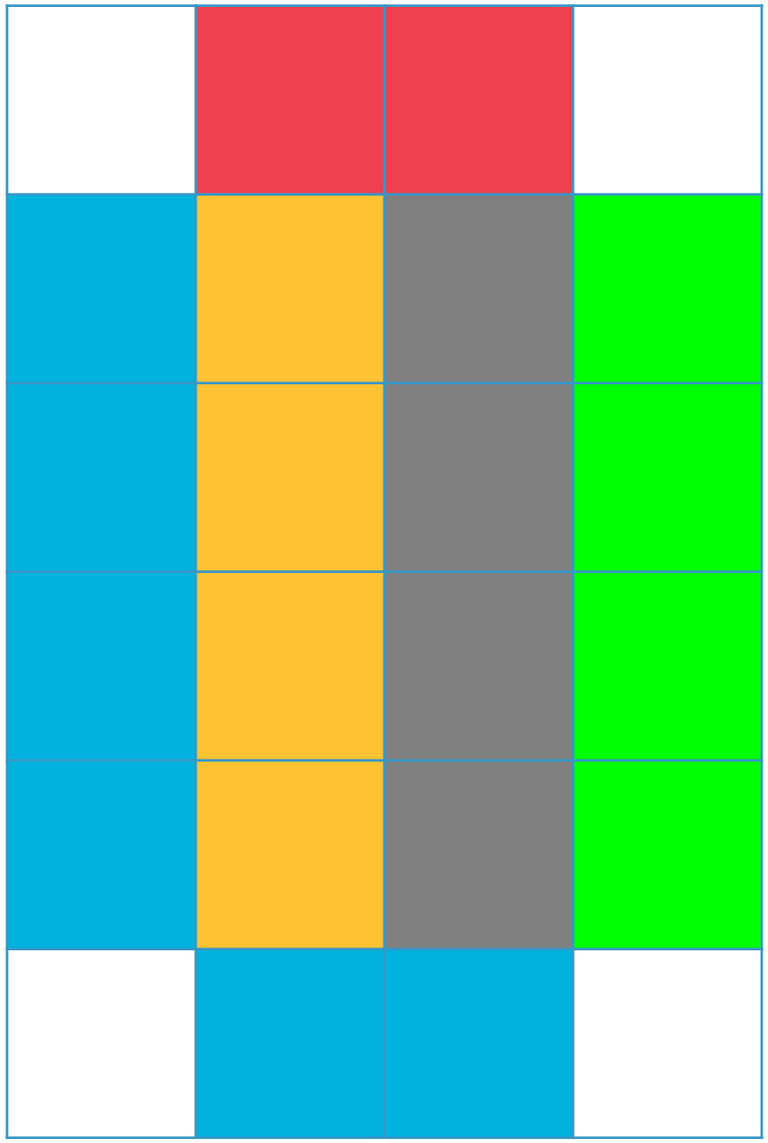
- So to complete **one** send/ receive pair need to
- Send yellow cells on current processor to left
- Receive green cells from processor on right

Represent green cells



```
int sizes[2], subsizes[2], starts[2];

//Whole array is nx_local+2 x ny_local+2
sizes[0] = nx_local + 2;
sizes[1] = ny_local + 2;
//Want a strip 1 x ny_local
subsizes[0] = 1;
subsizes[1] = ny_local;

//Start 1 cell up, nx_local + 1 in
starts[0] = nx_local + 1;
starts[1] = 1;

MPI_Type_create_subarray(2, sizes, subsizes,
starts, MPI_ORDER_FORTRAN, MPI_FLOAT,
&newtype);

MPI_Type_commit(&newtype);
```

Represent yellow cells



```
int sizes[2], subsizes[2], starts[2];

//Whole array is nx_local+2 x ny_local+2
sizes[0] = nx_local + 2;
sizes[1] = ny_local + 2;
//Want a strip 1 x ny_local
subsizes[0] = 1;
subsizes[1] = ny_local;

//Start 1 cell up, 1 cell in
starts[0] = 1;
starts[1] = 1;

MPI_Type_create_subarray(2, sizes, subsizes,
starts, MPI_ORDER_FORTRAN, MPI_FLOAT,
&newtype);

MPI_Type_commit(&newtype);
```

In case study

- Create one type for each strip of cells sent or received
- In practice, have a routine to create and commit types quickly

```
create_single_type(sizes, subsizes, starts, &newtype);
```

- Use types instead of temporary arrays in MPI_Sendrecv calls
- Makes little difference to Fortran
- Makes C much simpler

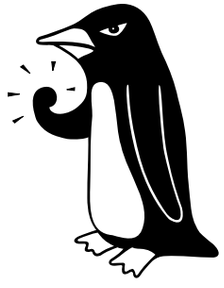
C Code

```
MPI_Sendrecv(data->data, 1, type_l_s, x_min_rank,
    TAG, data->data, 1, type_r_r, x_max_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);

MPI_Sendrecv(data->data, 1, type_r_s,
    x_max_rank, TAG, data->data, 1, type_l_r, x_min_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);

MPI_Sendrecv(data->data, 1, type_d_s, y_min_rank,
    TAG, data->data, 1, type_u_r, y_max_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);

MPI_Sendrecv(data->data, 1,type_u_s, y_max_rank,
    TAG, data->data, 1, type_d_r, y_min_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);
```

Fortran Code

```
    !Send left most strip of cells left and receive into right guard cells
    CALL MPI_Sendrecv(array, 1, type_l_s, x_min_rank, &
        tag, array, 1, type_r_r, x_max_rank, &
        tag, cart_comm, MPI_STATUS_IGNORE, ierr)

    !Send right most strip of cells right and receive into left guard cells
    CALL MPI_Sendrecv(array, 1, type_r_s, x_max_rank, &
        tag, array, 1, type_l_r, x_min_rank, &
        tag, cart_comm, MPI_STATUS_IGNORE, ierr)

    !Now equivalently in y
    CALL MPI_Sendrecv(array, 1, type_d_s, y_min_rank, &
        tag, array, 1, type_u_r, y_max_rank, &
        tag, cart_comm, MPI_STATUS_IGNORE, ierr)

    CALL MPI_Sendrecv(array, 1, type_u_s, y_max_rank, &
        tag, array, 1, type_d_r, y_min_rank, &
        tag, cart_comm, MPI_STATUS_IGNORE, ierr)
```

MPI Types



“The Angry Penguin“, used under creative commons licence from Swantje Hess and Jannis Pohlmann.

Warwick RSE

Useful trick

Using fewer types

- All of the types are very similar
- They represent the same shaped subarray
  - Different starting points
- MPI guarantees that it will not write anywhere other than the specified elements of the type
- Can shift the start of the array by handing MPI_Sendrecv a pointer to later in the array

Using fewer types

- Create a single type for the x direction and the y direction.

```
//Always the same sizes. This is local to each CPU
//So use nx_local, not nx
sizes[0] = nx_local+2; sizes[1] = ny_local+2;
//Same subsizes for all sends and receives in x direction
subsizes[0] = 1; subsizes[1] = ny_local;
//All operations in x direction
starts[0] = 0; starts[1] = 1;
```

- Then use it with an offset in MPI_Sendrecv

Using fewer types

```
MPI_Sendrecv(access_grid(data, 1, 0 ), 1, type_x_dir, x_min_rank,
    TAG, access_grid(data, nx_local + 1, 0 ), 1, type_x_dir, x_max_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);

MPI_Sendrecv(access_grid(data, nx_local, 0 ), 1, type_x_dir,
    x_max_rank, TAG, access_grid(data, 0, 0 ), 1, type_x_dir, x_min_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);

MPI_Sendrecv(access_grid(data, 0, 1 ), 1, type_y_dir, y_min_rank,
    TAG, access_grid(data, 0, ny_local + 1 ), 1, type_y_dir, y_max_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);

MPI_Sendrecv(access_grid(data, 0, ny_local ), 1,type_y_dir, y_max_rank,
    TAG, access_grid(data, 0, 0 ), 1, type_y_dir, y_min_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);
```

- Note that the data sources does not point to the start of the array
- Points to the start of the section to be sent/received

Using fewer types

```
    CALL MPI_Sendrecv(array(1, 0), 1, type_x_dir, x_min_rank, &
        tag, array(nx_local + 1, 0), 1, type_x_dir, x_max_rank, &
        tag, cart_comm, MPI_STATUS_IGNORE, ierr)

    CALL MPI_Sendrecv(array(nx_local, 0), 1, type_x_dir, x_max_rank, &
        tag, array(0, 0), 1, type_x_dir, x_min_rank, &
        tag, cart_comm, MPI_STATUS_IGNORE, ierr)

    CALL MPI_Sendrecv(array(0, 1), 1, type_y_dir, y_min_rank, &
        tag, array(0, ny_local + 1), 1, type_y_dir, y_max_rank, &
        tag, cart_comm, MPI_STATUS_IGNORE, ierr)

    CALL MPI_Sendrecv(array(0, ny_local), 1, type_y_dir, y_max_rank, &
        tag, array(0, 0), 1, type_y_dir, y_min_rank, &
        tag, cart_comm, MPI_STATUS_IGNORE, ierr)
```

- Fortran always implicitly passes arguments by reference
- Effectively pointer
- Same approach, just give index of first array element
- Note that array subsections are in general copies

Using fewer types

- This works because you are sending the same amount of data in each direction

- MPI will just go to where you specify in memory and send the type relative to that point

- The MPI Type now extends past the end of your array

- But doesn’t matter because those cells aren’t used

- If you’ve designed the type right

- Can cause weird, weird bugs

- Do once everything else is working