Intermediate MPI

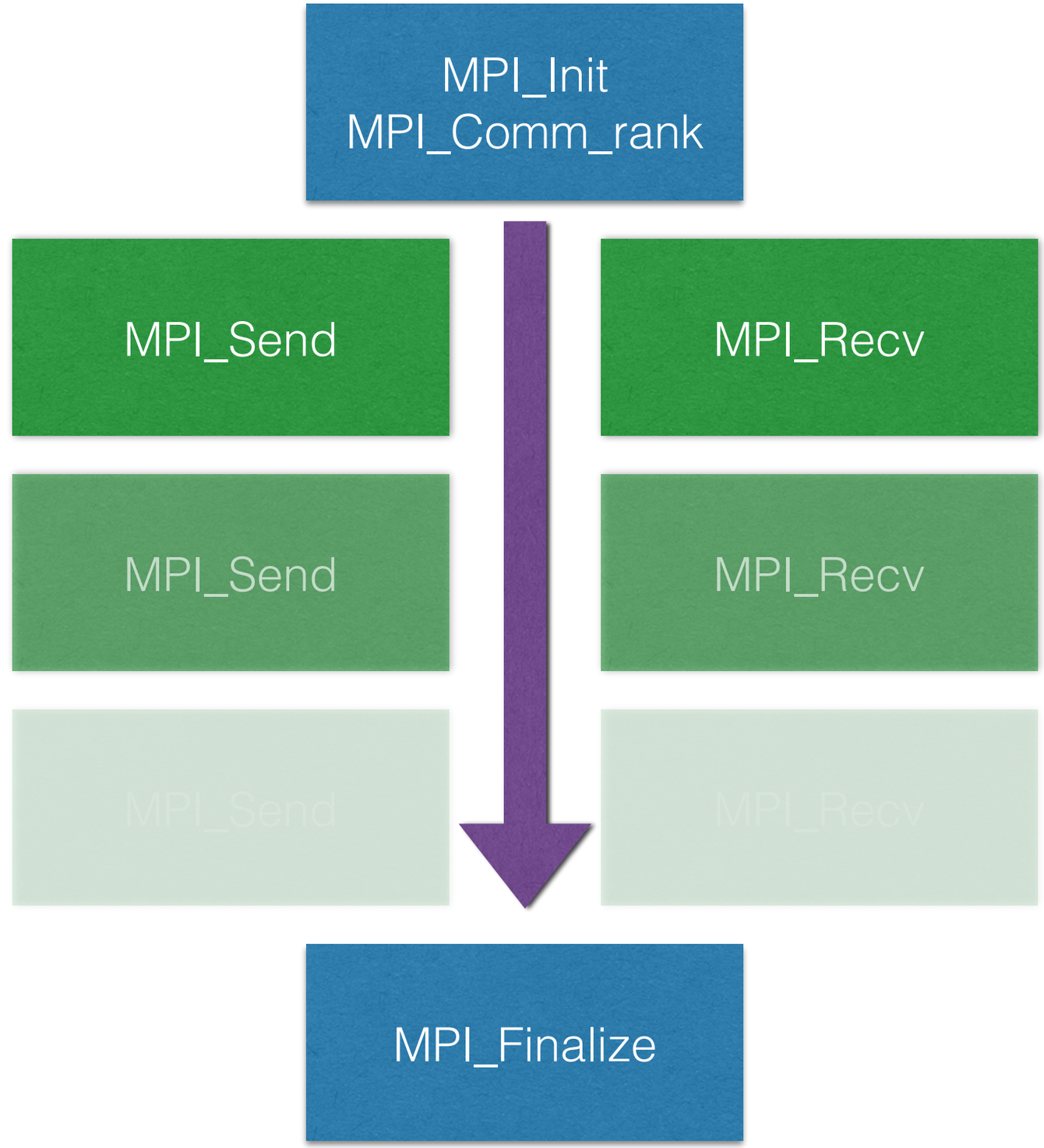
Chris Brady
Heather Ratcliffe

Warwick RSE

19/03/2024

MPI Concepts

- MPI_Init to start MPI
- MPI_Comm_rank to find unique processor rank
- MPI_Send/MPI_Recv pairs
- More Send/Recv
- MPI_Finalize



MPI In Practice

- Real working codes that scale well to the largest computers in the world sometimes use only MPI_Sendrecv and MPI_Allreduce (for their main communications at least)

- What we are teaching here ranges between
  - “helpful and makes code neater” - MPI Types

- Still worth knowing that this stuff is available

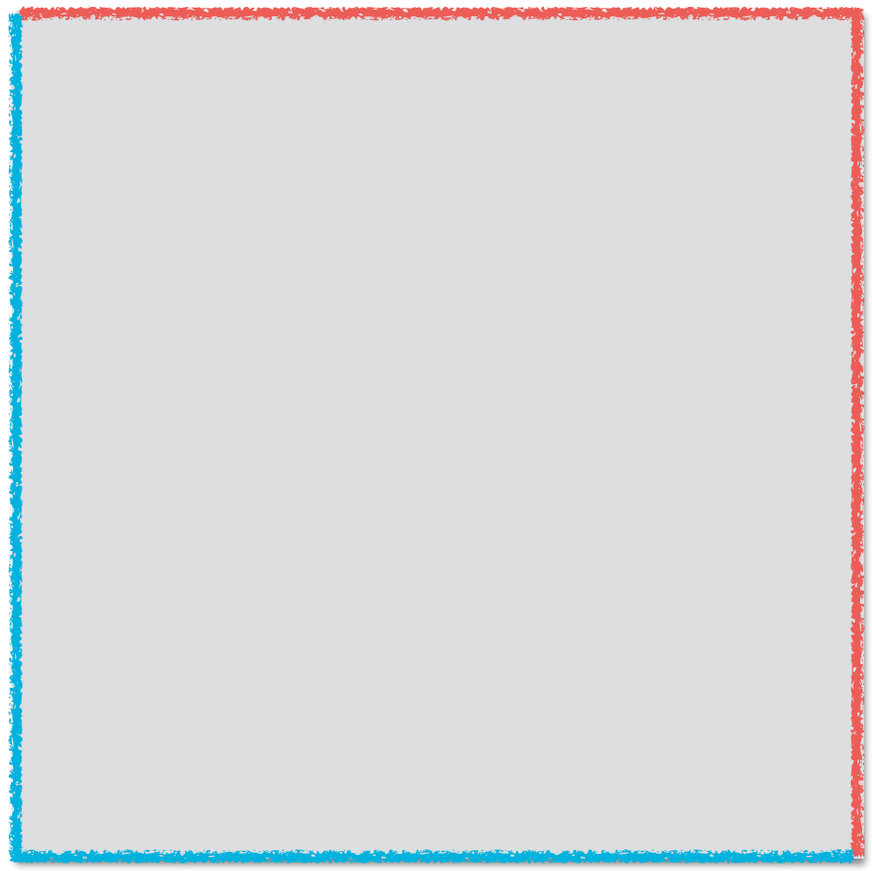
Case study

- To get any further, need a good example of something that can be split up over processors

- Spatial decomposition
    - Want object that has spatial extents

- Sheet of metal
    - Density - Fixed
    - Temperature (?)

Case Study

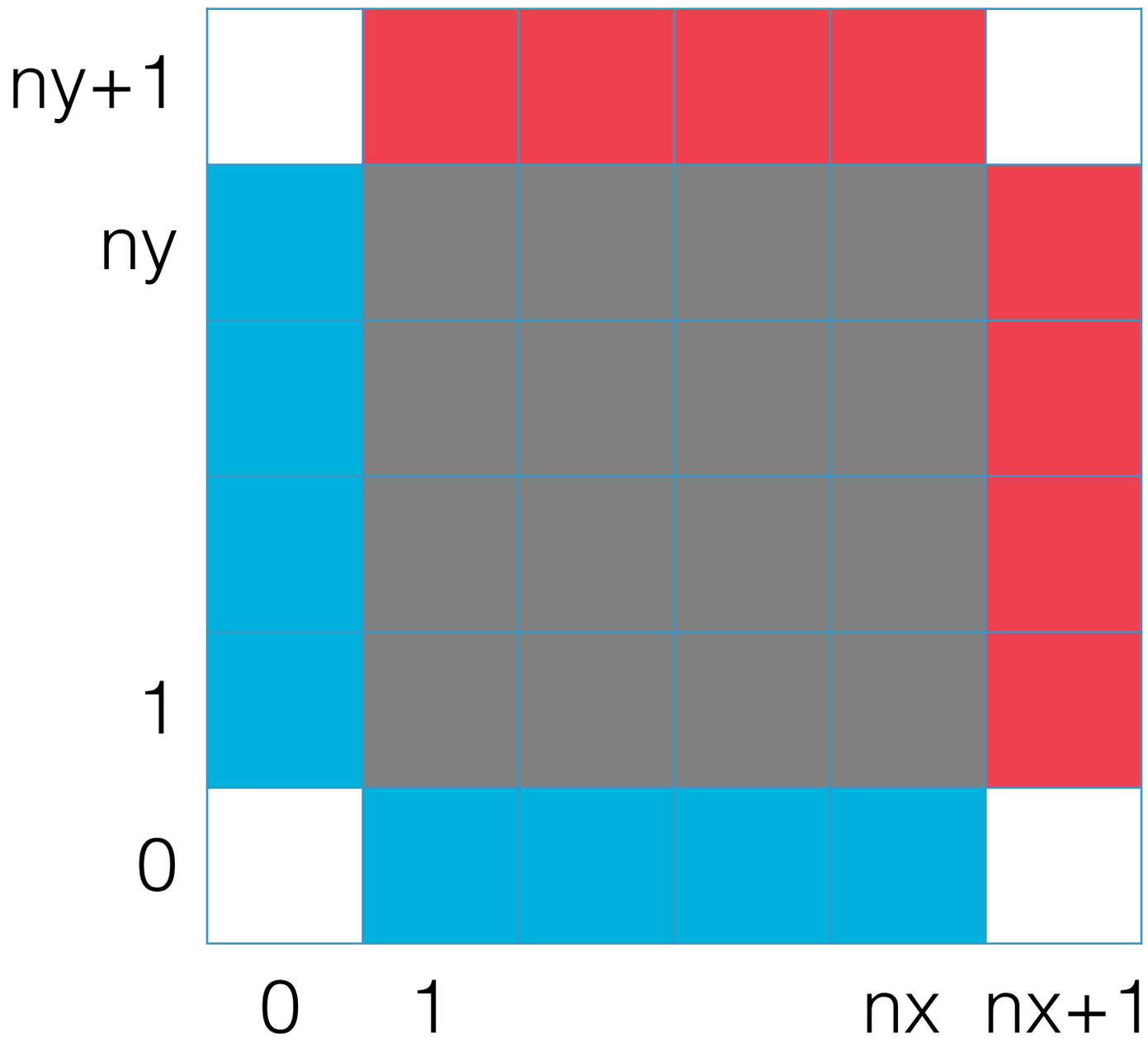


- Temperature of sheet
- Top and right edges kept hot
- Bottom and left edges kept cool
- What is temperature across the sheet?

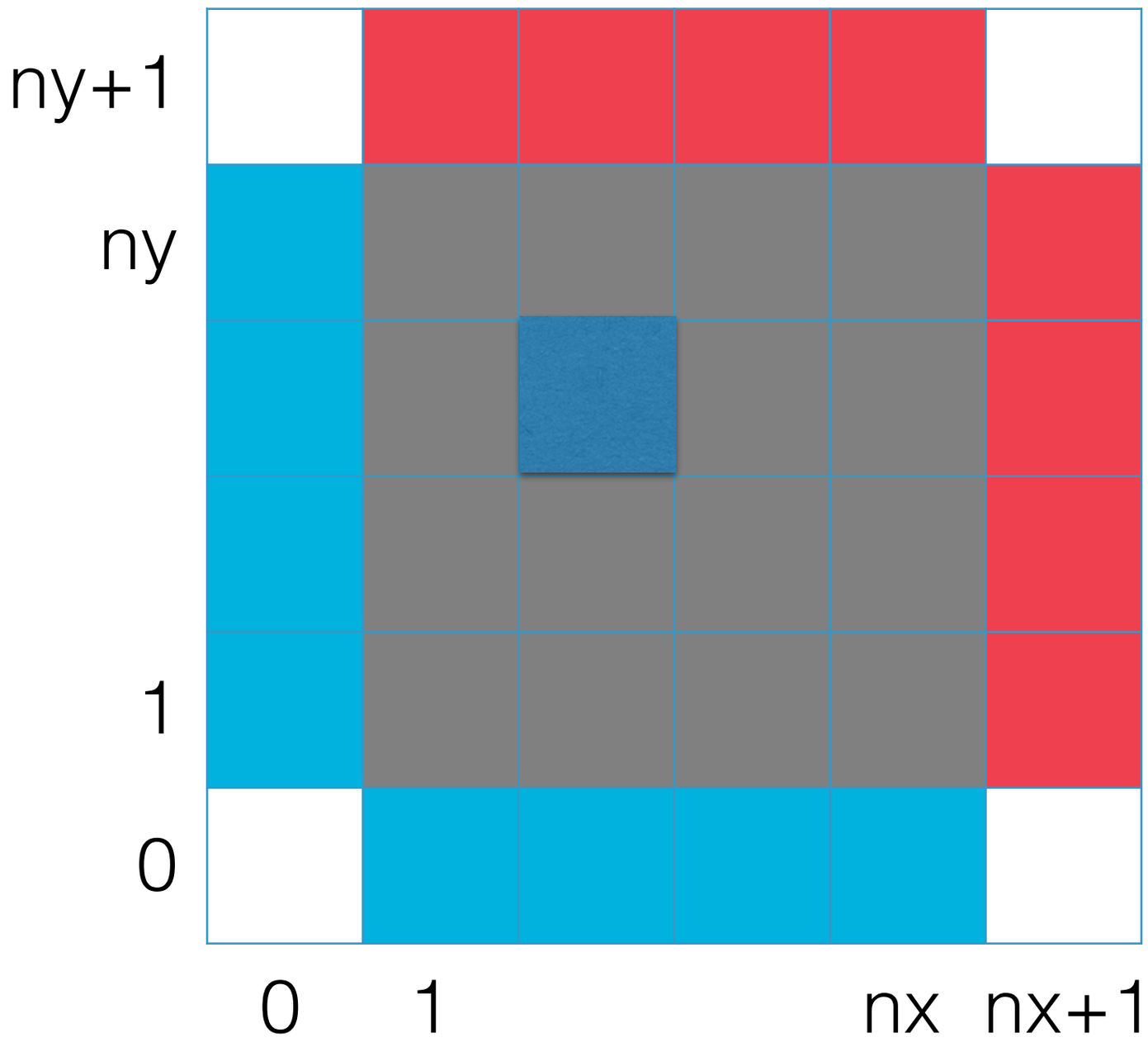
Brief details

- Solve heat equation (https://en.wikipedia.org/wiki/Heat_equation) for steady state
- Split space up into a set of discrete grid points
- Lots of details, but can reduce calculation to simply averaging four cardinally adjacent cells (for some specific values)
- Keep going until solution is “finished” (i.e. converged to a given level of correctness)
- Iterative solution of final answer - not change in time
- Hot and cold edges are implemented as a “halo” of “ghost” or “guard” cells

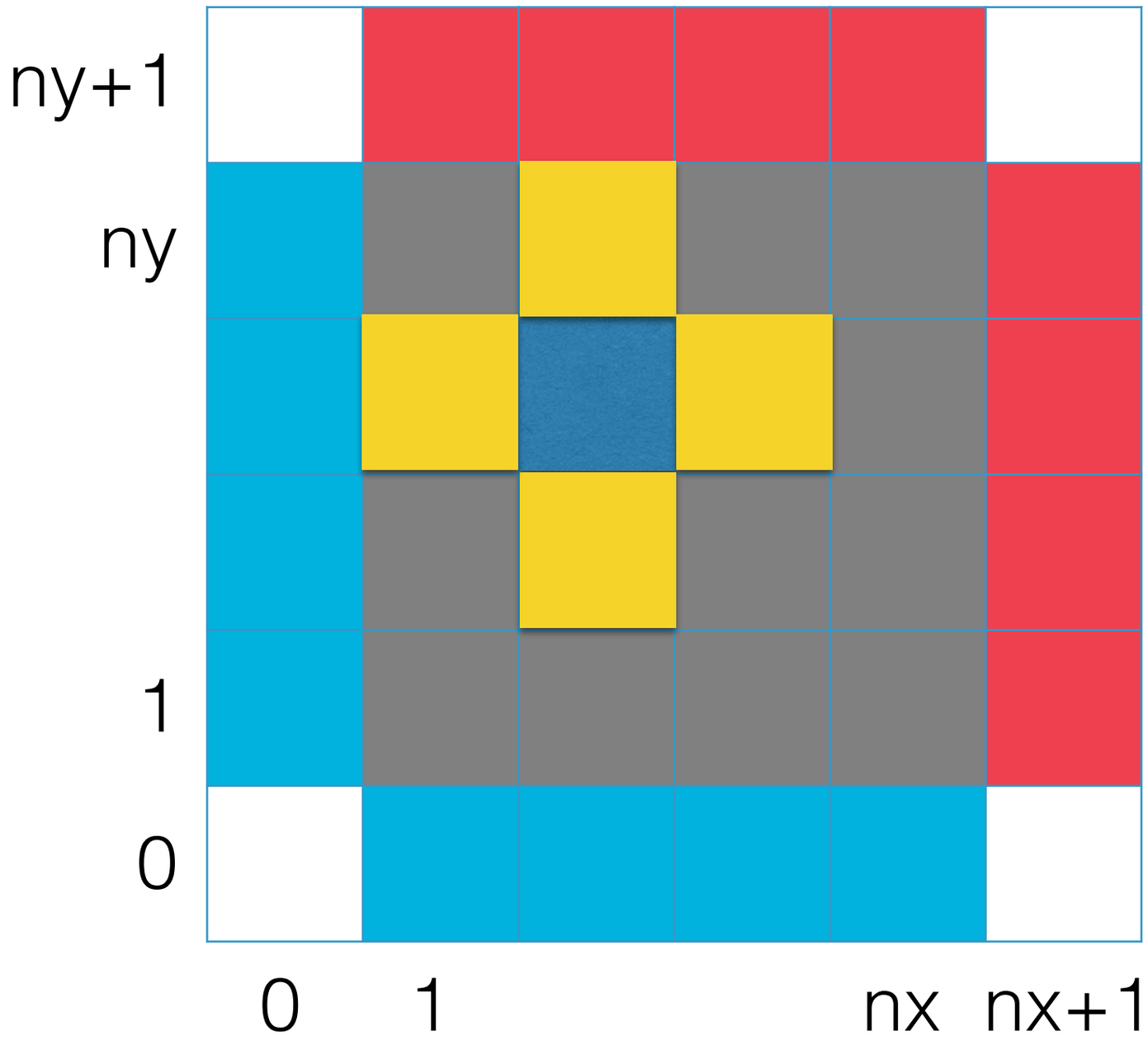
Diagram



Diagram



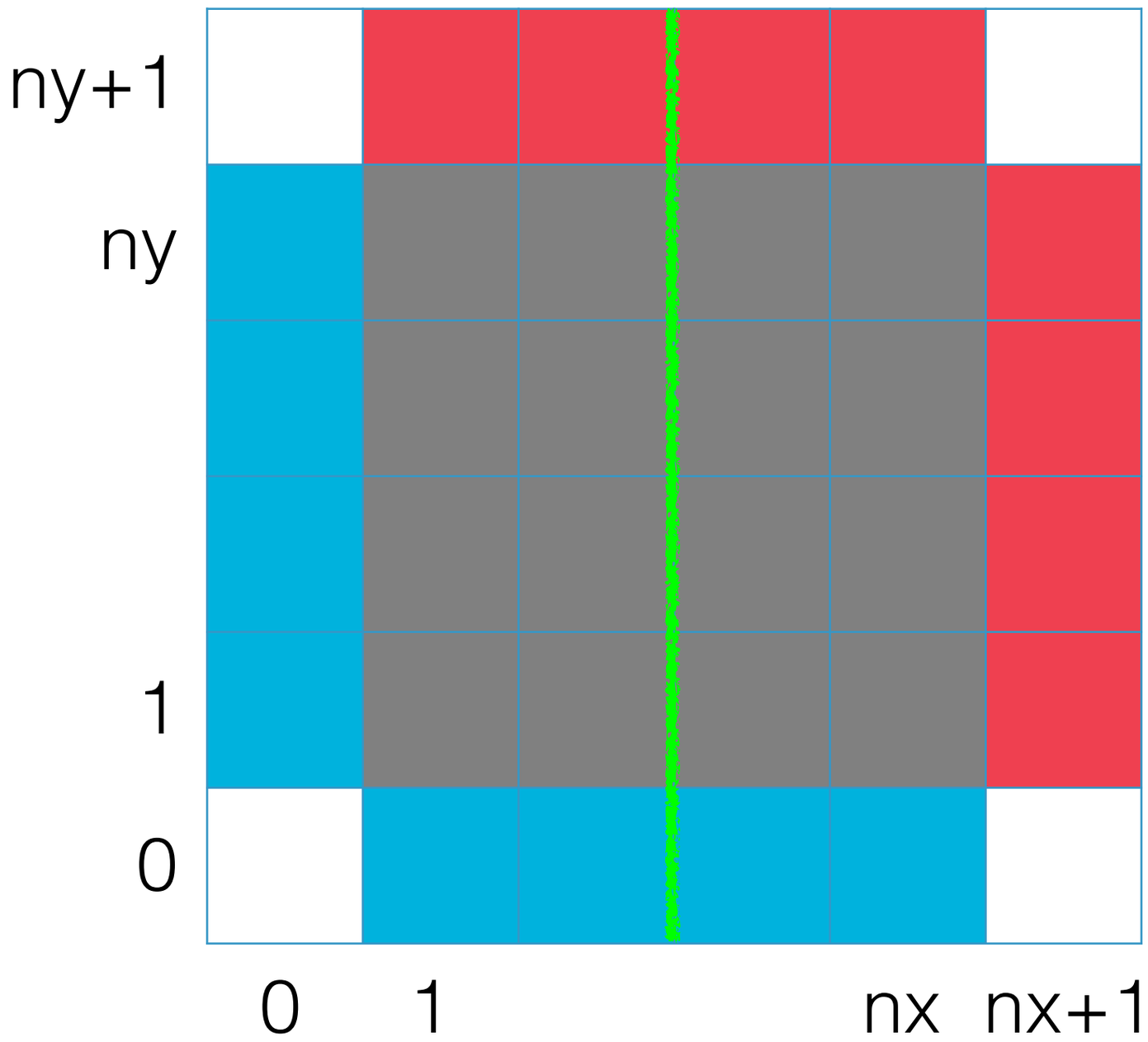
Diagram



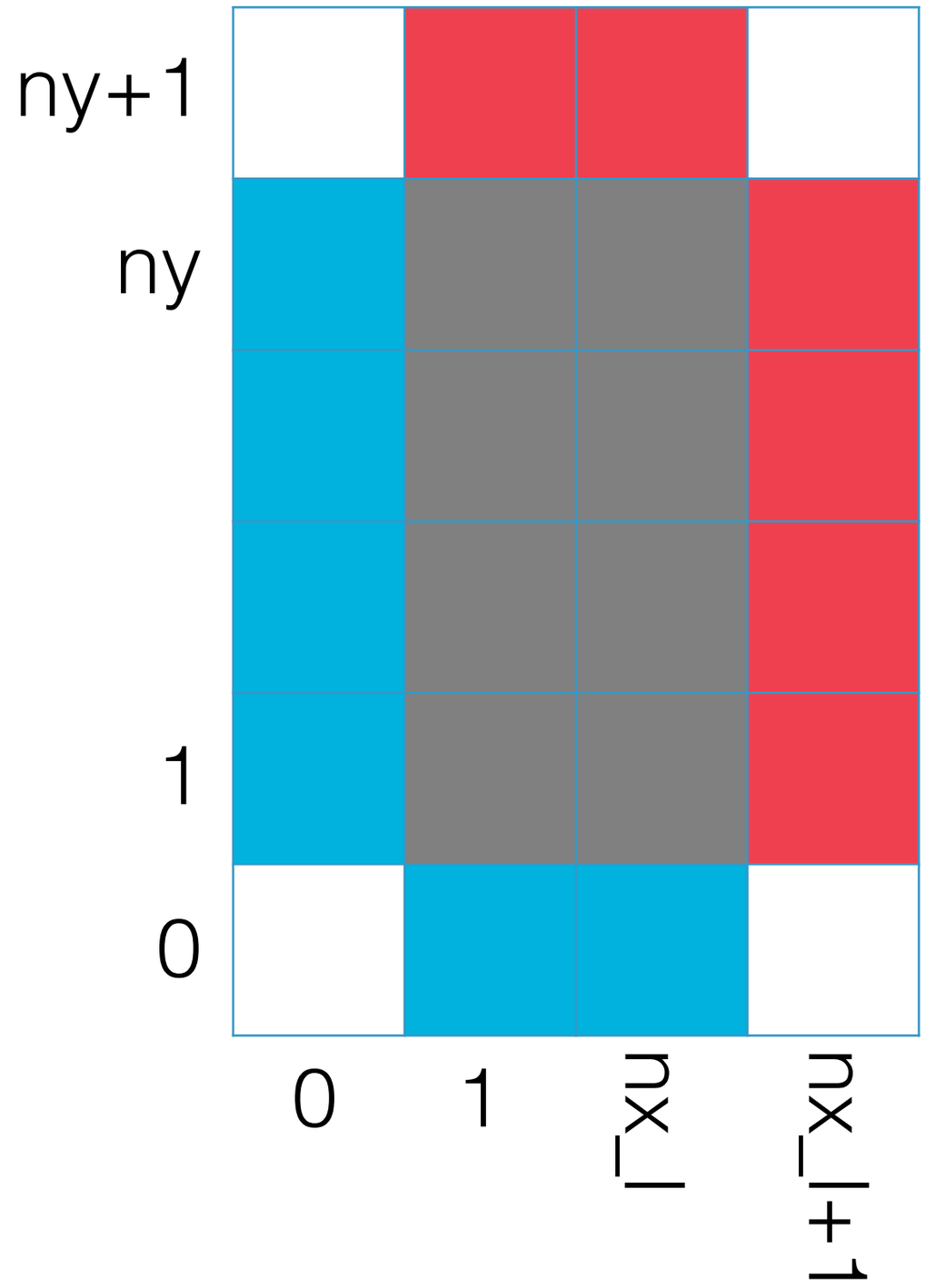
Implementation Details

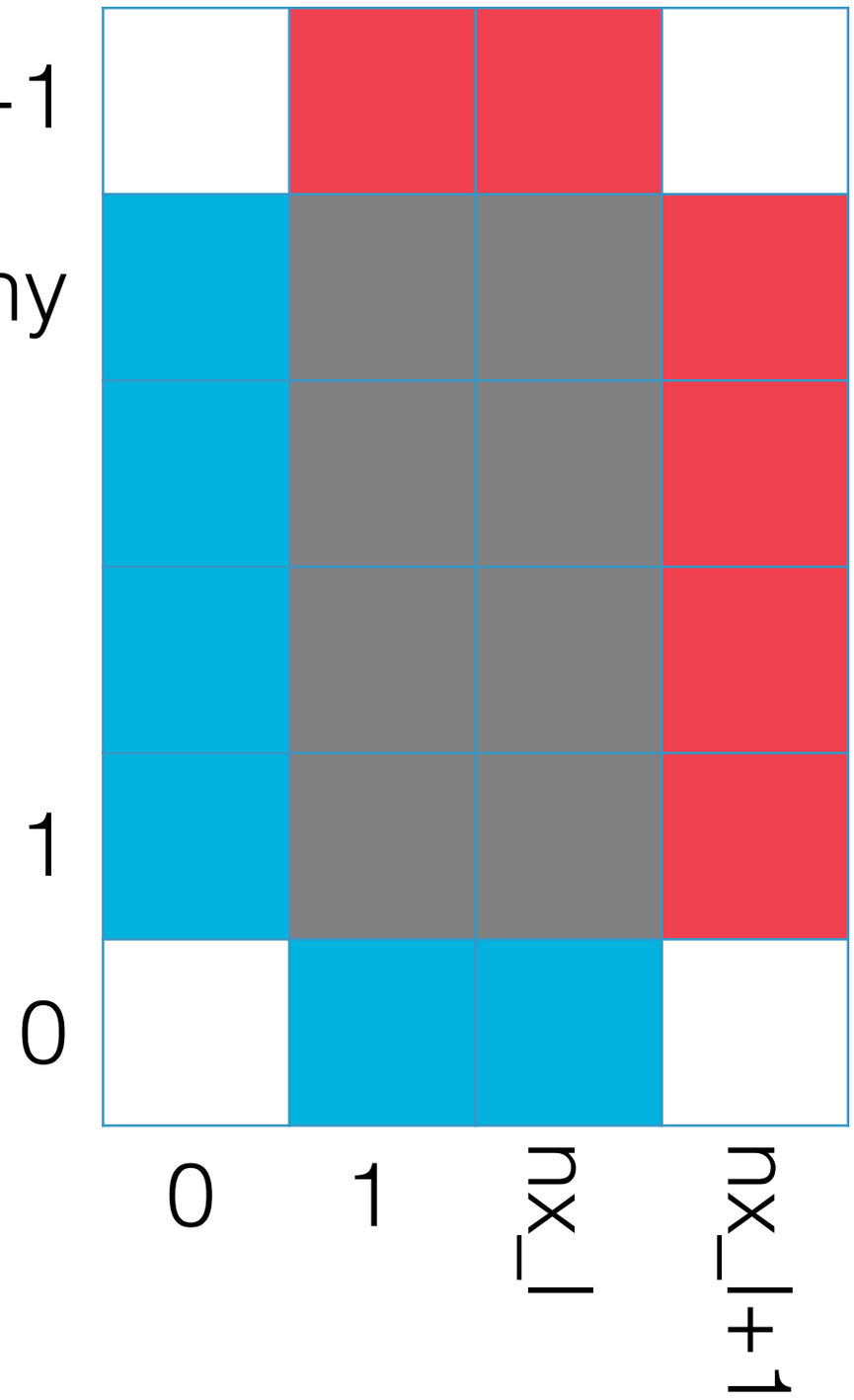
- Fortran code using F95
  - Allocatable multidimensional arrays with explicit upper and lower bounds
  - Copy and assign operations on array sections
- C code using C99
  - Equivalent arrays created in our code (include Fortran array ordering!)
  - See support/array.c for details
  - See code to see implementation

Diagram

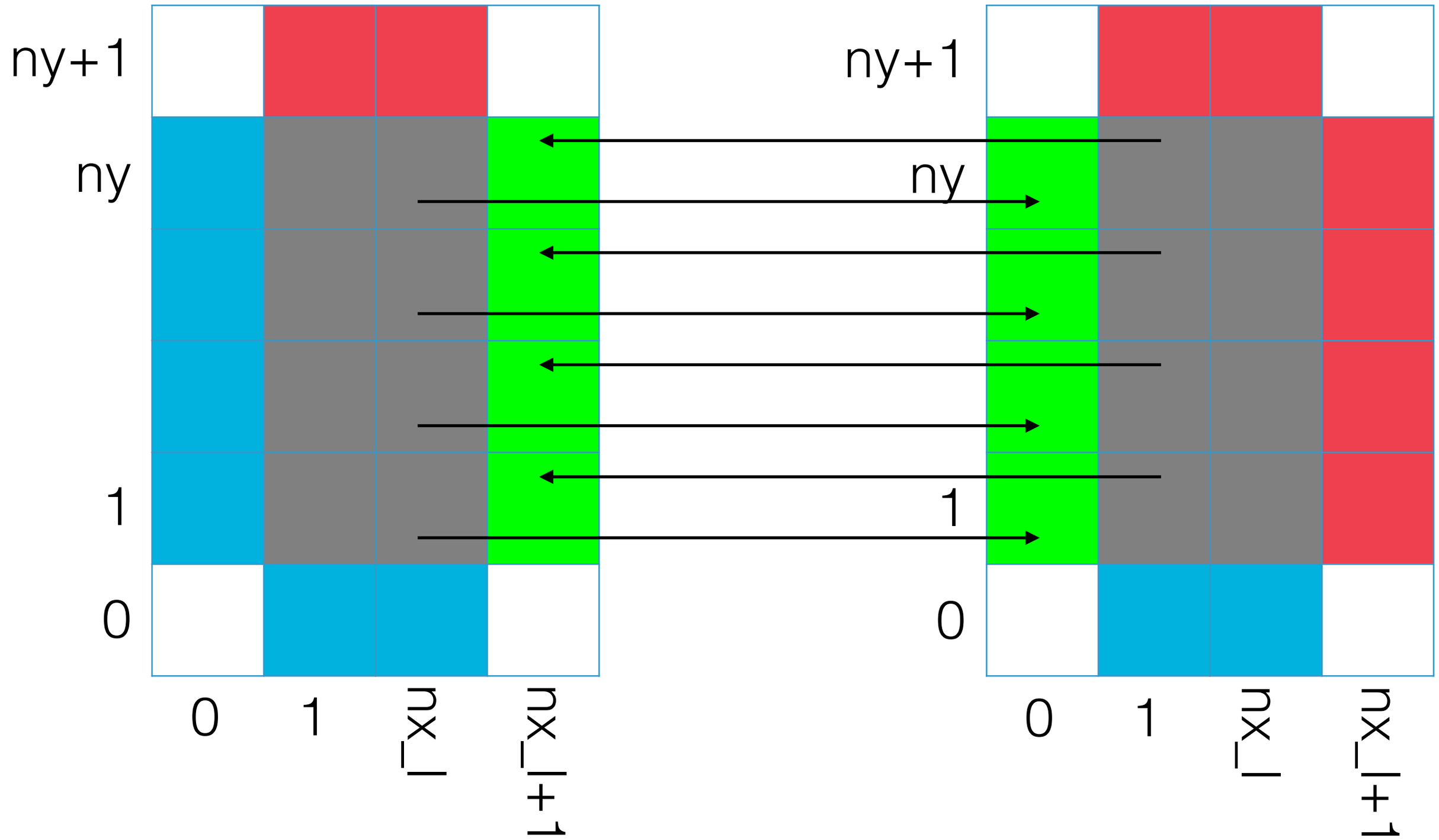


Domain decomposition





Domain decomposition



Parallelising

- The concept of parallelising this code is simple
    - Create additional “virtual” boundaries between spatial domains
- Good practice is to have a special boundary conditions routine
    - Includes both virtual boundaries and real boundaries
    - Real boundaries not needed in this case because they are fixed
    - Set once and leave

Ranges

- Global domain
  - (0:nx+1) x (0:ny+1) total cells
  - (1:nx) x (1:ny) simulation domain
- Local domain
  - (0:nx_l + 1) x (0:ny_l + 1)
  - (1:nx_l) x (1:ny_l)
- We call nx_l and ny_l “**nx_local**” and “**ny_local**” in code because otherwise hard to see

Selecting local sizes

- nx and ny are usually defined by the problem (at least you have to have enough points)
- nx_l and ny_l can only be defined at runtime
  - Depend on the number of processors and how you choose to split the processors up
  - In general, this is a hard problem
    - Load balancing
  - Here just want to minimize perimeter to area ratio

MPI topologies

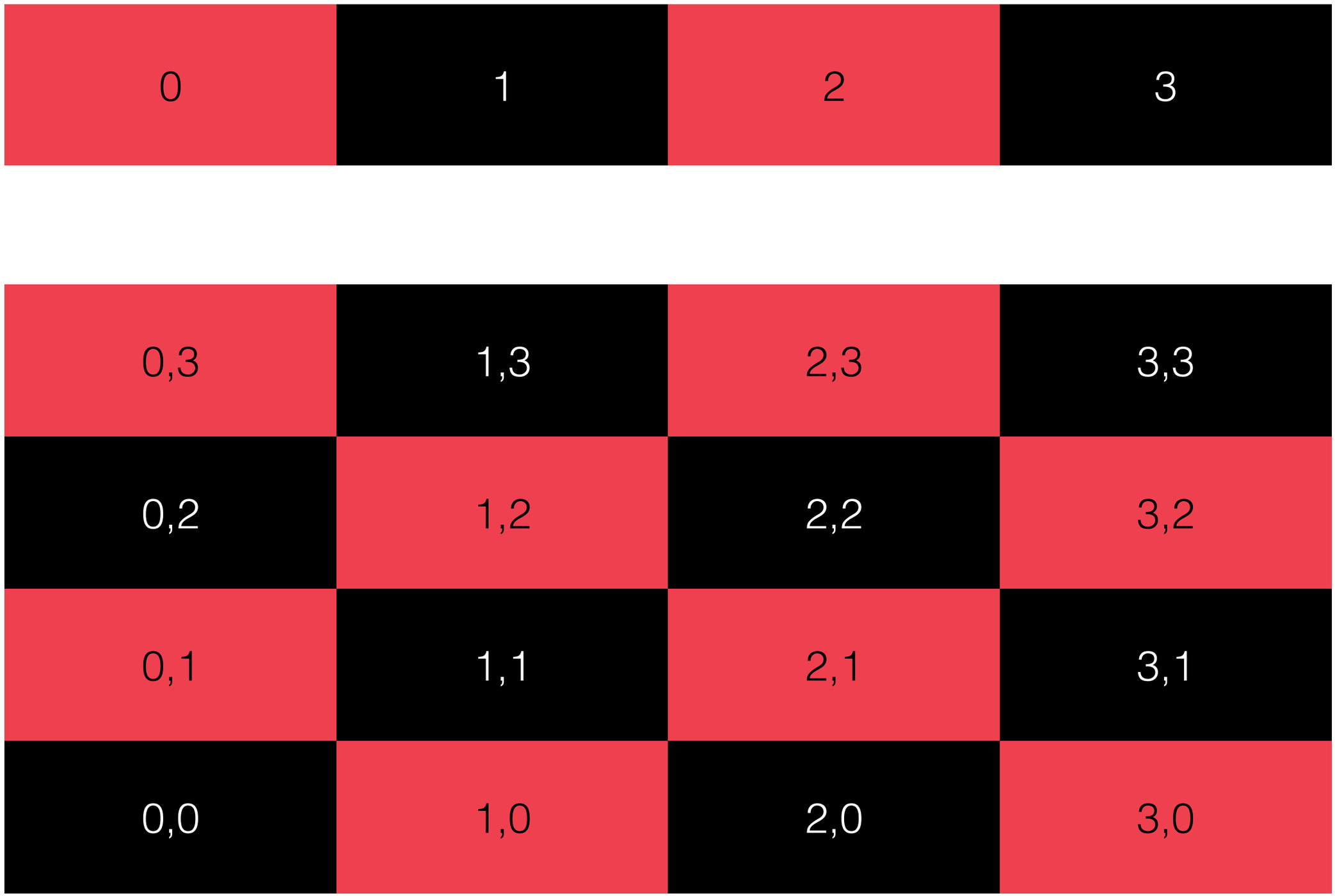
MPI Topologies

- You can tell MPI how your domains are connected together
    - Create new communicator
- Tries to keep connected nodes “close” in the physical hardware of the machine
- Two kinds
    - MPI_Graph_create
    - MPI_Cart_create
- Only going to talk about MPI_Cart_create here
- The created communicator can be used wherever MPI_COMM_WORLD is normally
- Will in general have different rank in different communicator

MPI Topologies

- We use another helper routine **MPI_Dims_create**
- This routine gives you a possible decomposition of N processors into a grid of N dimensions
- It isn't necessarily optimal for a given problem, but it is an easy way of doing this decomposition

Cartesian topology



Coordinates

- Once you have a grid of processors you now have the concept of a processors coordinate in this grid as well as its rank

- You can get a processor's rank in the Cartesian communicator by using **MPI_Comm_rank** with the new communicator rather than **MPI_COMM_WORLD**

- **MPI_Cart_coords** lets you get the coordinate of a processor from this rank

- **MPI_Cart_rank** lets you get a rank from coordinates

- You still use rank for communication, not coordinates

Getting neighbours

- As well as your own position in the grid, you will in general also want to know the rank of your neighbours

- You can get this by just offsetting your coordinates and using MPI_Cart_rank to get the position

- There is a useful function **MPI_Cart_shift** that tells you about your neighbours which is demonstrated in the example code

- It introduces one last value that ranks can have

Getting neighbours

- When you call **MPI_Cart_create** you can specify if boundaries are periodic or not - if they are then processors on an edge of the grid have neighbours on the other edge

- If not then when you use **MPI_Cart_shift** to get each neighbour you get a special value **MPI_PROC_NULL**

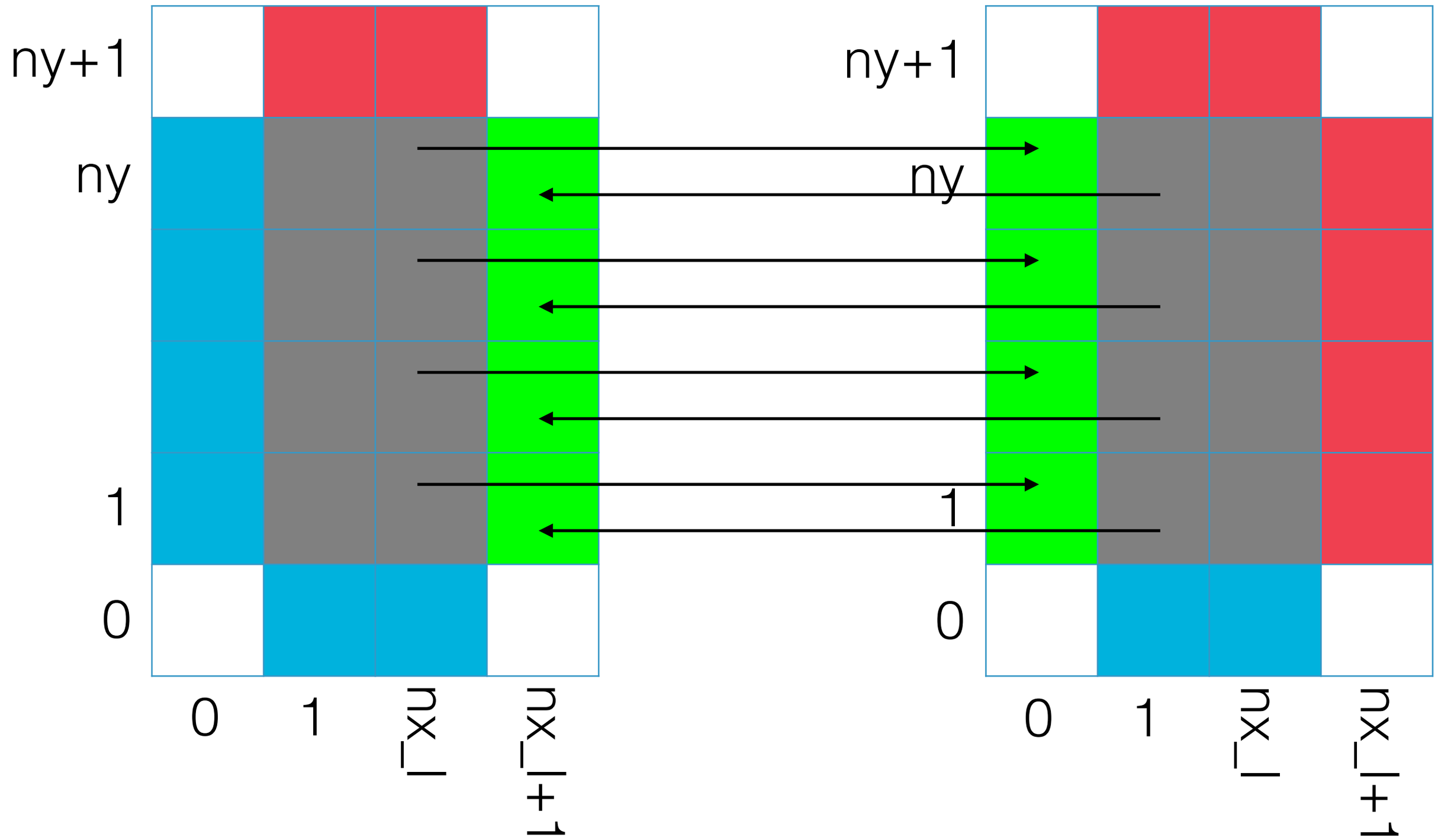
- **MPI_PROC_NULL** turns any send or receive operation where it is passed in as the rank into a null operation - no communication is done

- You can use **MPI_PROC_NULL** to avoid having to have your own code to check if you are at the edge of a domain - just use MPI_PROC_NULL to indicate that nothing should happen

- If you use **MPI_PROC_NULL** with **MPI_Sendrecv** then the send and receive bits are inactivated separately depending on if their rank is **MPI_PROC_NULL**

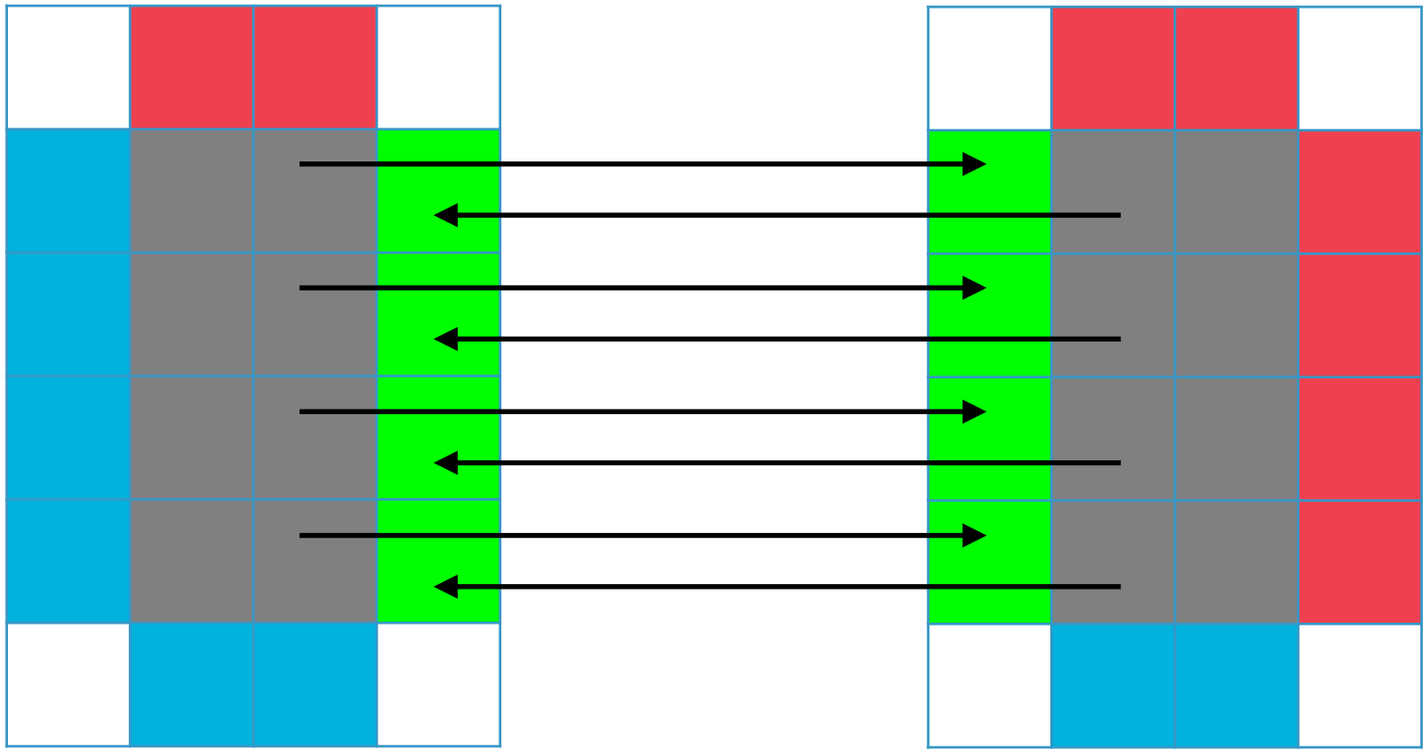
Back to case study

Mapping



Mapping

1



2

- (nx_l,1:ny_l) on 1 ⇒ (0, 1:ny_l) on 2
- (1,1:ny_l) on 2 ⇒ (nx_l+1, 1:ny_l) on 1

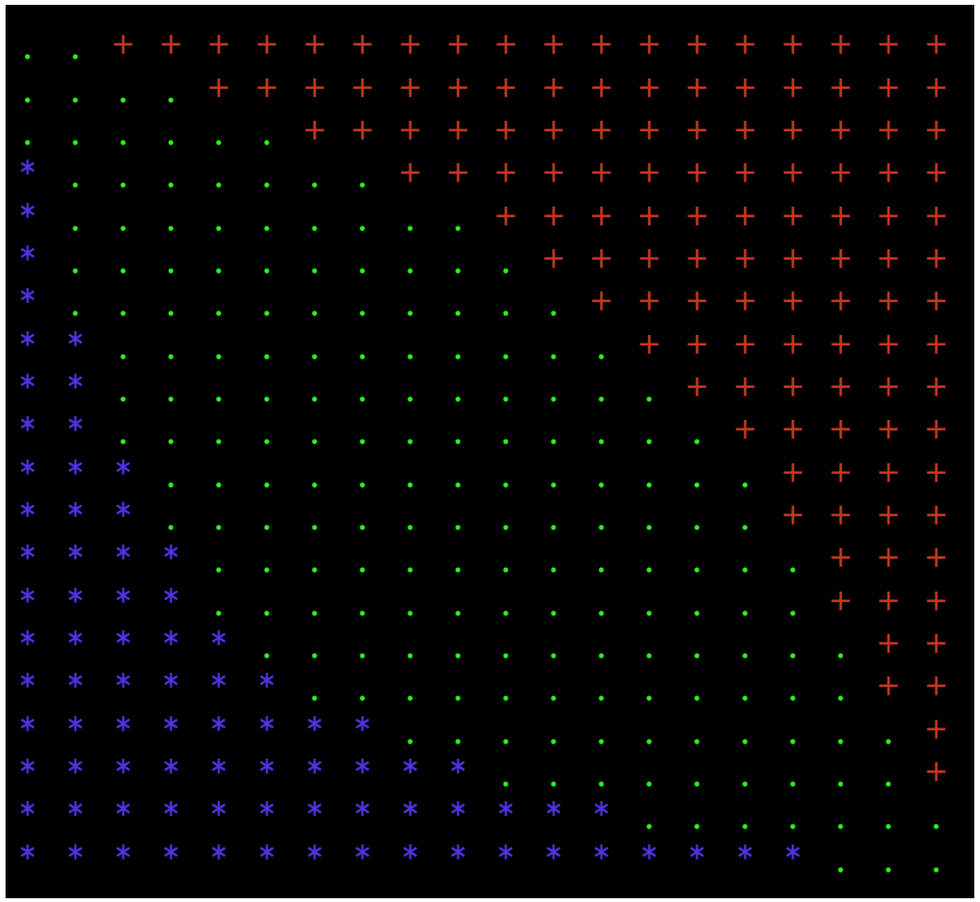
Mapping

- To send right (nx_l,1:ny_l) ⇒ (0, 1:ny_l)
- To send left (1,1:ny_l) ⇒ (nx_l+1, 1:ny_l)
- To send up (1:nx_l, ny_l) ⇒ (1:nx_l, 0)
- To send down (1:nx_l, 1) ⇒ (1:nx_l, ny_l+1)

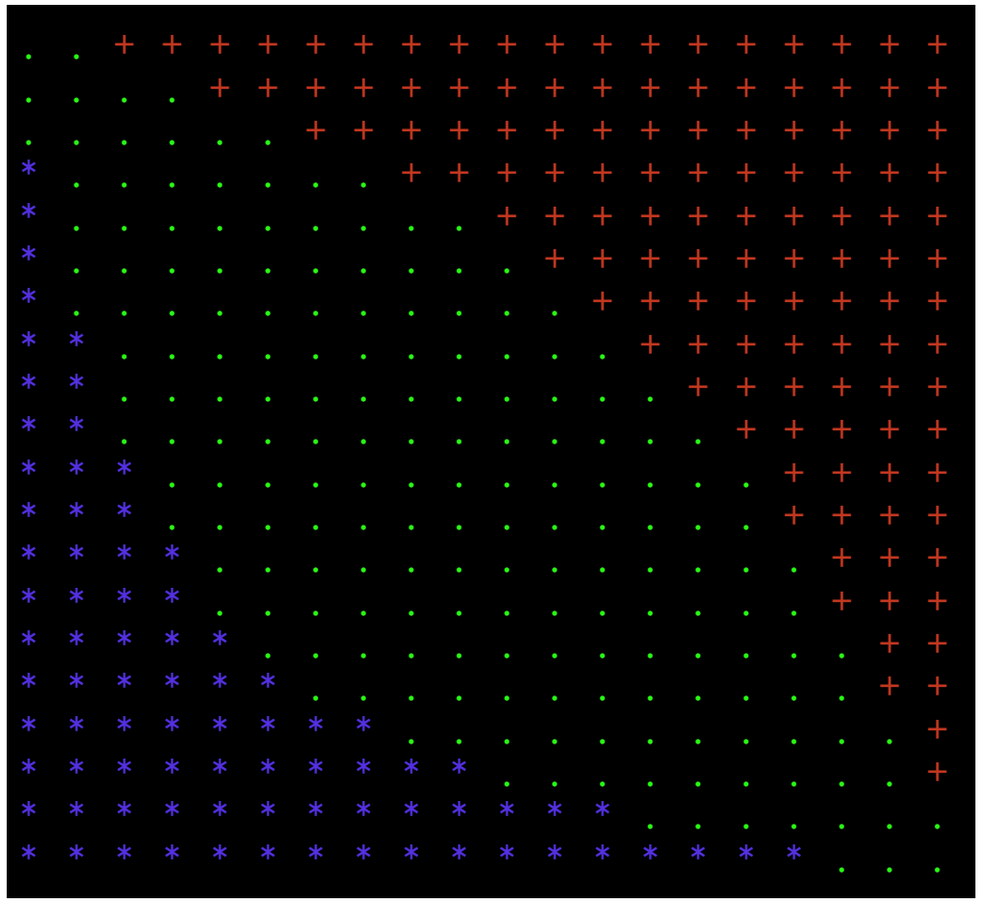
Communicating

- This is easy enough, just write MPI_Sendrecv calls matching these
- For each direction send to the processor at higher coordinate and receive from the processor at lower coordinate
- Then reverse and send lower and receive higher
- You want to send just the strip of data that is needed to populate the ghost cells
- Easy to send an array subsection in Fortran, has to be copied into a temporary in C/C++

Results



1 Processor



16 Processors

MPI Custom Types

MPI Types

- Remember that in all of these communication routines we have been passing one parameter which is the type of the data to be sent or received

- Usually **MPI_DOUBLE** or **MPI_DOUBLE_PRECISION**

- Why do we need to do that?

  - The compiler knows the type of the variable that we are passing to be sent or to have data received into

- We can create custom types for a variety of jobs

  - Including removing those temporary copies in C/C++

MPI Types

- There are a variety of MPI routines for creating custom datatypes, ranging from the simple to the quite complex

- Important things to note before looking at any of them

- **Creating** the type does not make it usable for communication you have to pass it to the function **MPI_Type_commit** first

- Once you have finished with a type, free it with **MPI_Type_free**

  - Only so many slots for types are available, so don't forget this

Most basic custom type

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

- You can just send them by using a primitive datatype and saying how many you want to send

- You can create a custom type using **MPI_Type_contiguous**

- Note that when you use your new contiguous type you now only send **one** of it even though you are now sending multiple items

  - Sending two of this type would send 16 items in total

More useful ones?

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

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

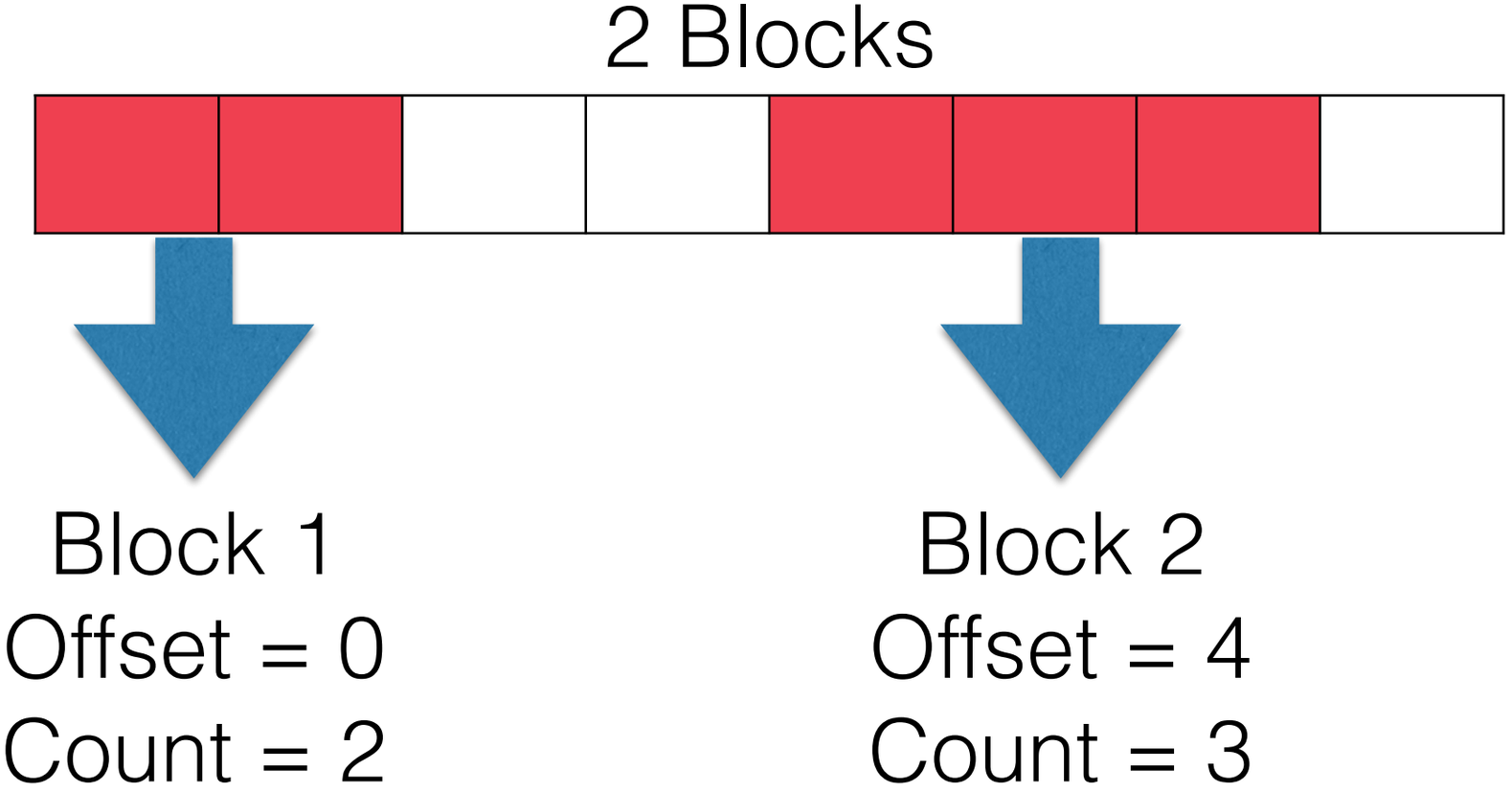
- Simplest is **`MPI_Type_indexed`**

More useful ones?

- Many MPI type creation routines work by specifying block lengths and block offsets

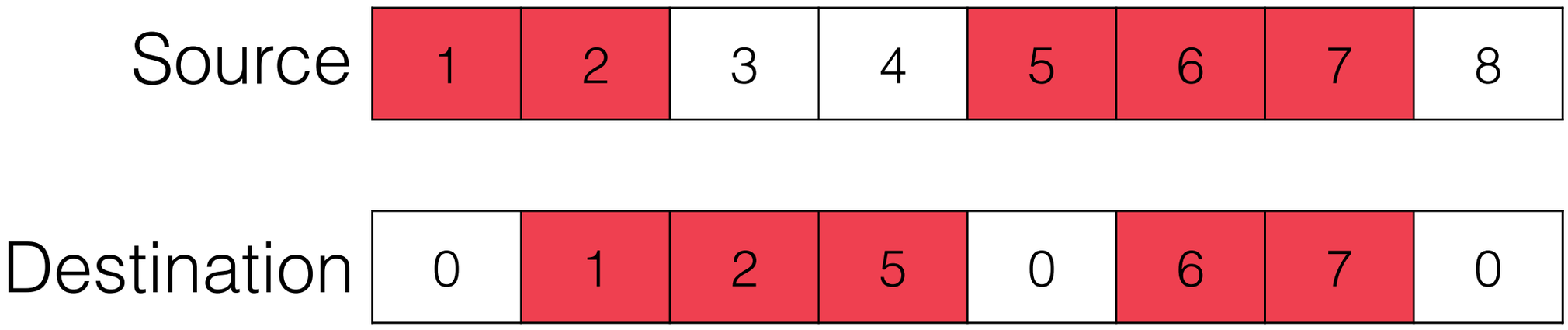
- MPI_Type_Indexed works like this

More useful ones?



- Create arrays holding the offsets and counts
- Pass to **MPI_Type_indexed**
- Then pass the resulting type to **MPI_Type_commit**

Mixing types



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

Structures

- You can send C structs and some kinds of Fortran TYPE using MPI types

- Have to specify

  - The offset of each member of the struct from the start of the struct in **BYTES** (this is not something that you work out, because due to compiler padding of datatypes you have to be careful)

  - The MPI datatype of each member of the struct

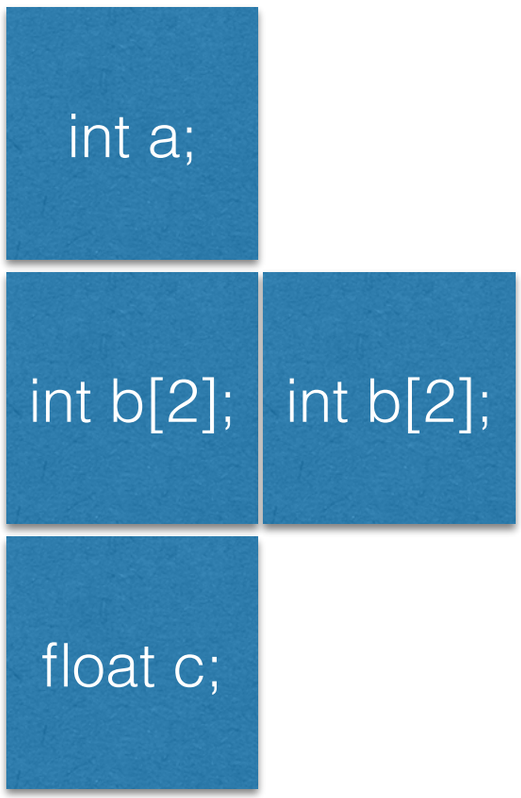
  - The number of elements of the MPI datatype in each member of the struct

Structures

```
typedef struct {
  int a;
  int b[2];
  float c;} mystruct;

mystruct struct_instance;
```

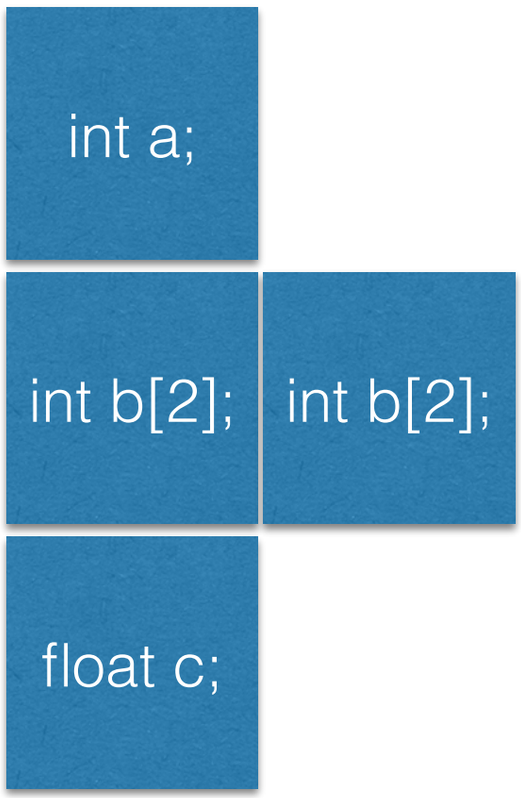
```
TYPE mytype
  SEQUENCE
  INTEGER :: a
  INTEGER, DIMENSION(2) :: b
  REAL(KIND(1.0)) :: c
END TYPE mytype

TYPE(mytype) :: type_instance
```

Structures



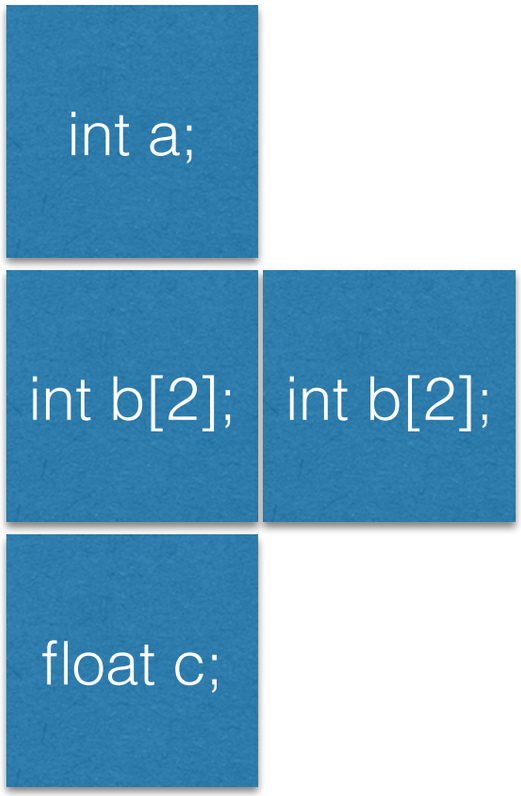
- Types are easy
  - MPI_INT
  - MPI_INT
  - MPI_FLOAT

Structures



- Blocklengths are easy
  - 1
  - 2
  - 1

Structures



- Offsets are a bit harder
- Need to know how far from the start of your type you are in bytes
- Not going to labour the point here but in C there is a standard function **offsetof** and in Fortran there are MPI routines to help do this (**MPI_Get_address** and **MPI_Aint_diff**)

Structures

- Once you have created and committed the type representing your structure you can send and receive all of the data in a structure or in arrays of structures in a single message

- This can be much quicker than using one message for each component of a structure

- Note that in Fortran only **BIND(C)** or **SEQUENCE** types can be sent and received like this because normal types are not guaranteed to have any particular memory layout

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

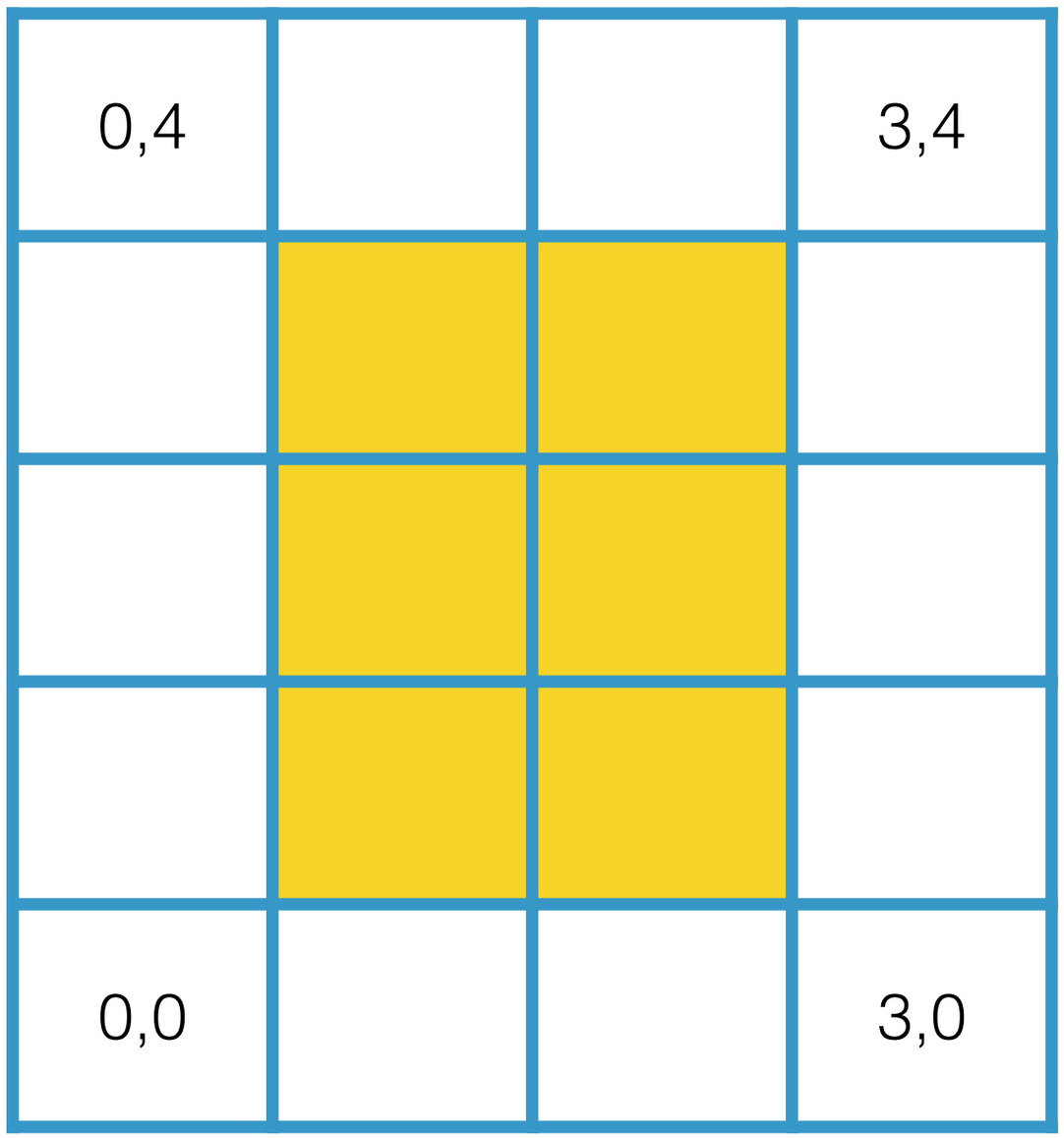
- Makes the C code much more readable

- Routine is **MPI_Type_create_subarray**

MPI_Type_create_subarray

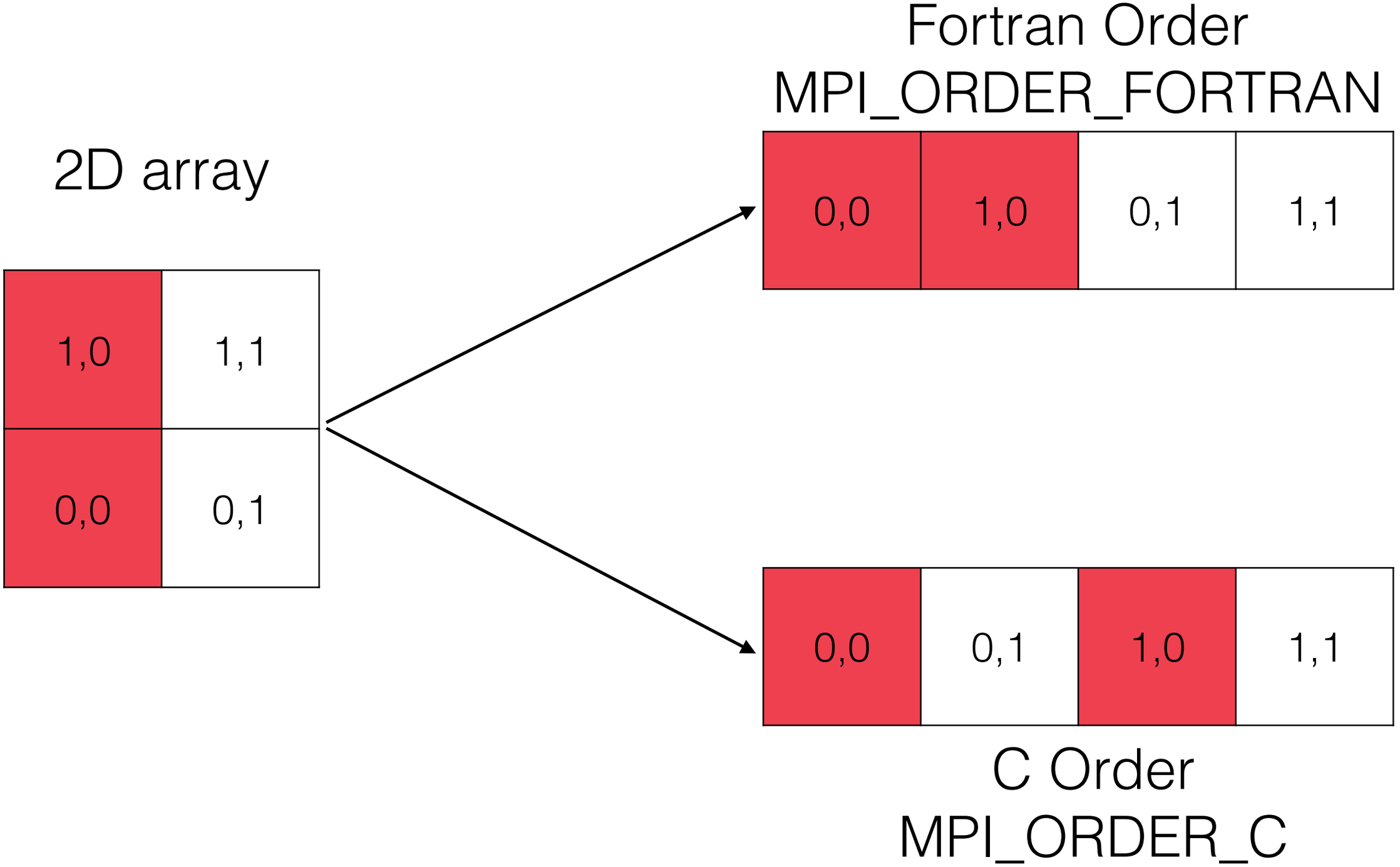
- To create a subarray you have to specify
  - The size of the whole array in each dimension
  - The size of the subarray in each dimension
  - The offset of the subarray into the whole array in each dimension (called **starts** by **MPI_Type_create_subarray**)
  - The ordering in memory of the array

MPI_Type_create_subarray



Sizes = [4,5]
Subsizes = [2,3]
Starts = [1,1]

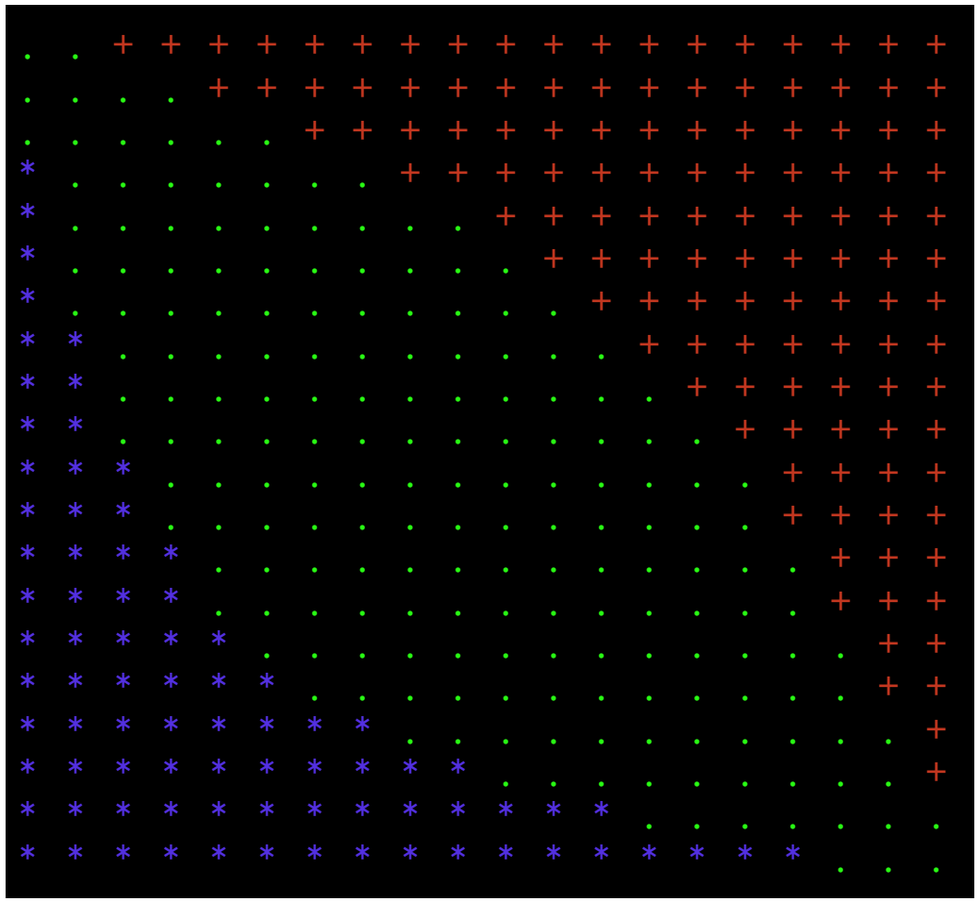
Array order


2D array
1,0
1,1
0,0
0,1
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

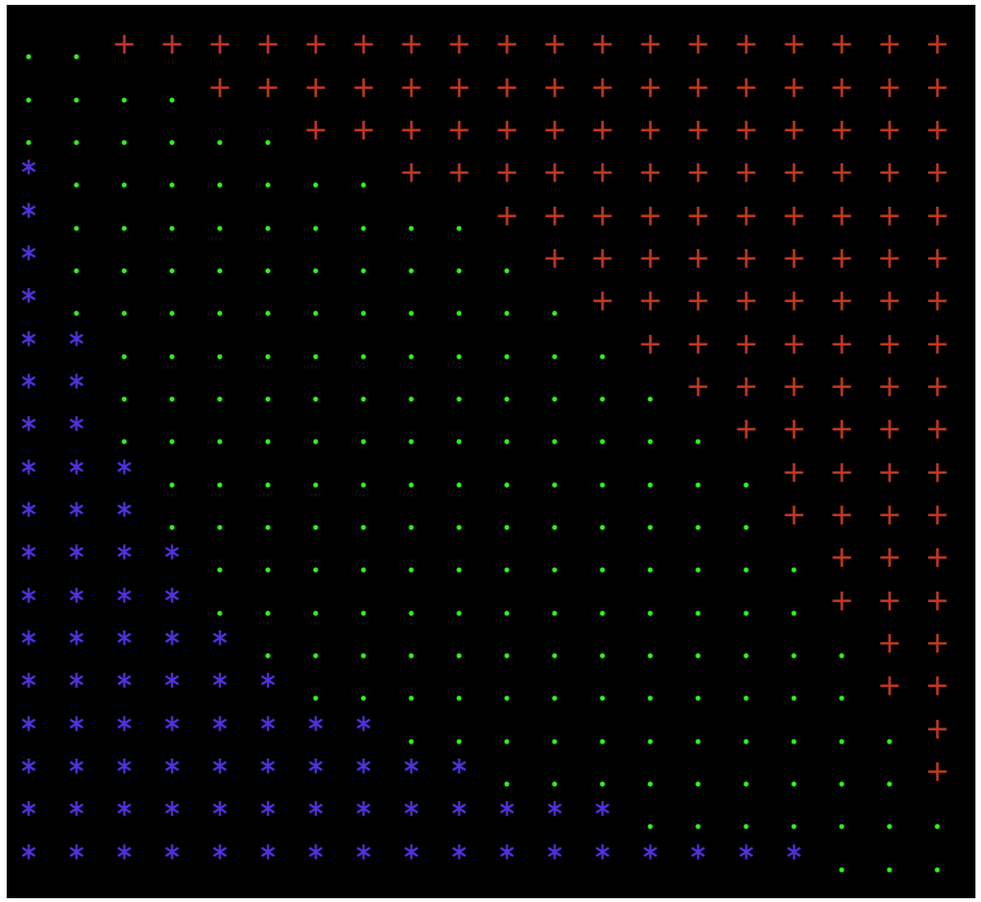
Array order

- By default
  - C is row major order
    - Last index varies fastest
  - Fortran is column major order
    - First index varies fastest
- Both languages can “mock up” arrays ordered the other way round
- You pass in either **MPI_ORDER_FORTRAN** or **MPI_ORDER_C** to **MPI_TYPE_CREATE_SUBARRAY**
- You **always** specify starts, sizes and subsizes by index order regardless of the order in memory

Results



1 Processor



16 Processors

Non-blocking in brief

Concept

- We replaced matched MPI_Send and MPI_Recv calls with MPI_Sendrecv so that you could do a single send and a single receive at once
    - What about if you wanted to do more sends and receives at once?
- Non blocking communication
    - Can send or receive data and then immediately go on to do other things while waiting for the communications to happen

Why?

- Not many reasons
  - Don’t need synchronisation but need to get data away to another processor
    - Posting results back to a master process for example
- Can do other work while the communication happens
  - “Over lap compute and communicate”

Very similar

- The non blocking send and receive commands are MPI_Isend and MPI_Irecv

- Almost the same as MPI_Send and MPI_Recv

    - You add one new parameter and one new concept the **communication handle**

- Communication handles tell you about an inflight communication

- It also adds MPI_Test and MPI_Wait to either test for completion or wait for completion

- Note that non-blocking is just another way of sending messages - you can send a message with MPI_Isend and receive it with MPI_Recv

Why not always

- It does add complexity to your code
- You have to work out exactly when you want data
- You have to guarantee that you won't alter the data in a send buffer until a send has completed
- There is an overhead to setting non-blocking calls up so you need to have a real advantage to latency hiding
- There are **persistent communication** versions if you regularly send or receive from the same buffers which almost eliminate this overhead