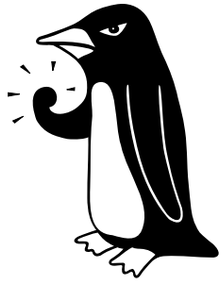
Case Study in MPI

Chris Brady
Heather Ratcliffe

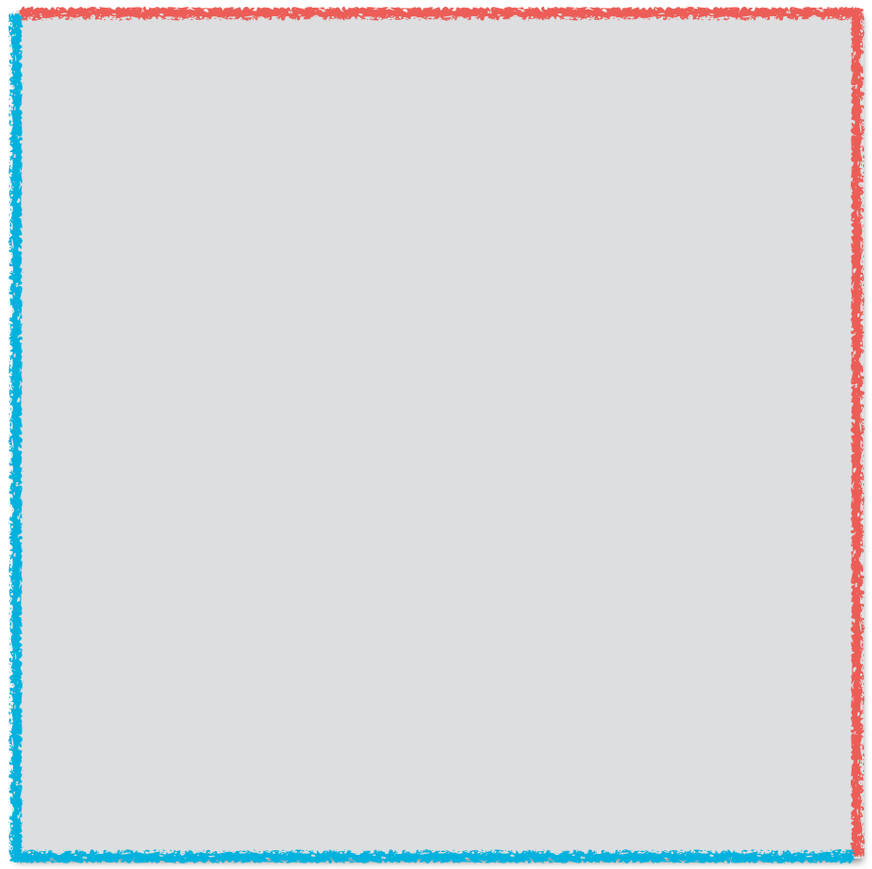


Warwick RSE

Case study

- 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

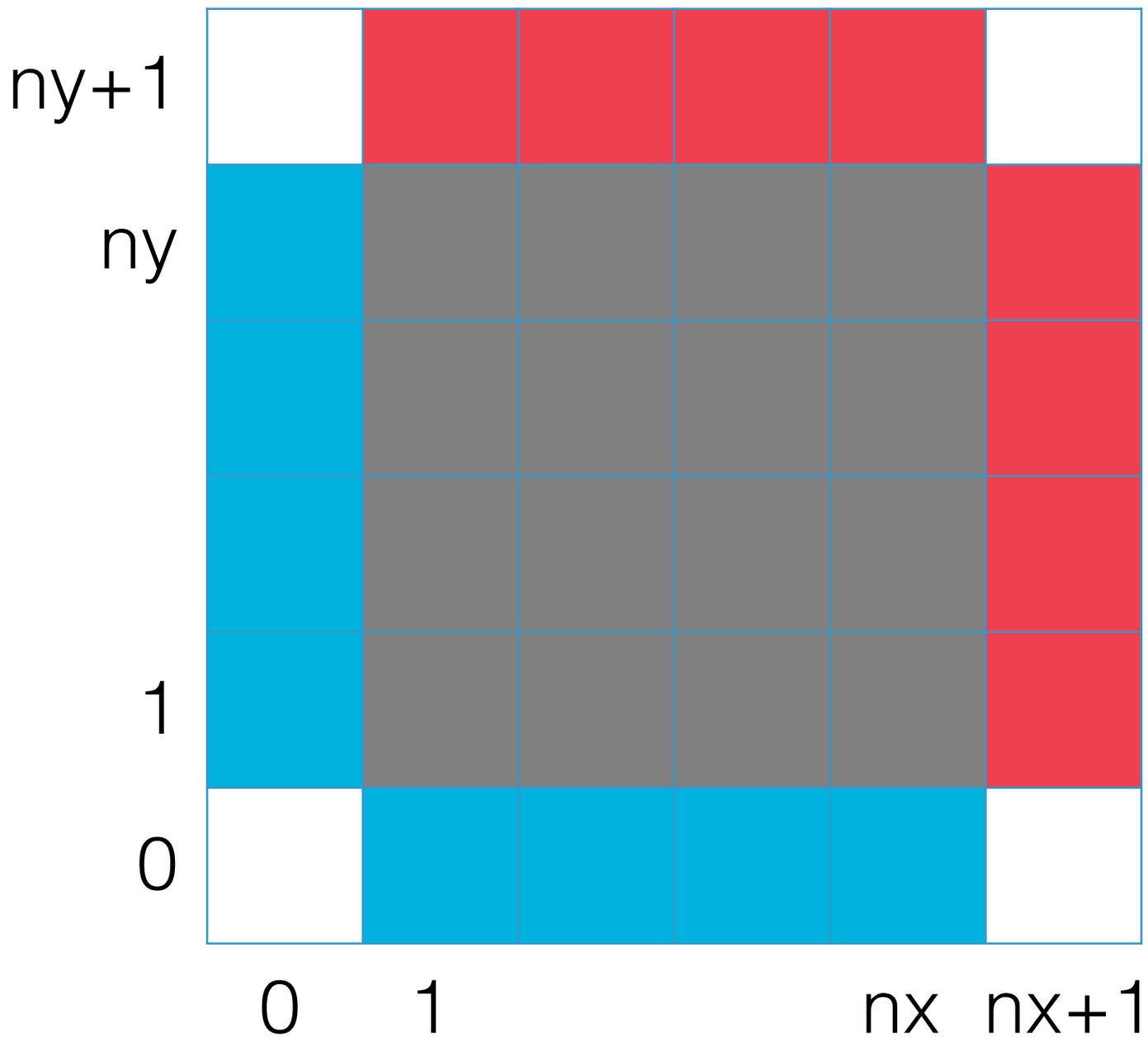


- Temperature of sheet
- Top and right edges kept hot
- Bottom and left edges kept cool
- What is temperature across the sheet?

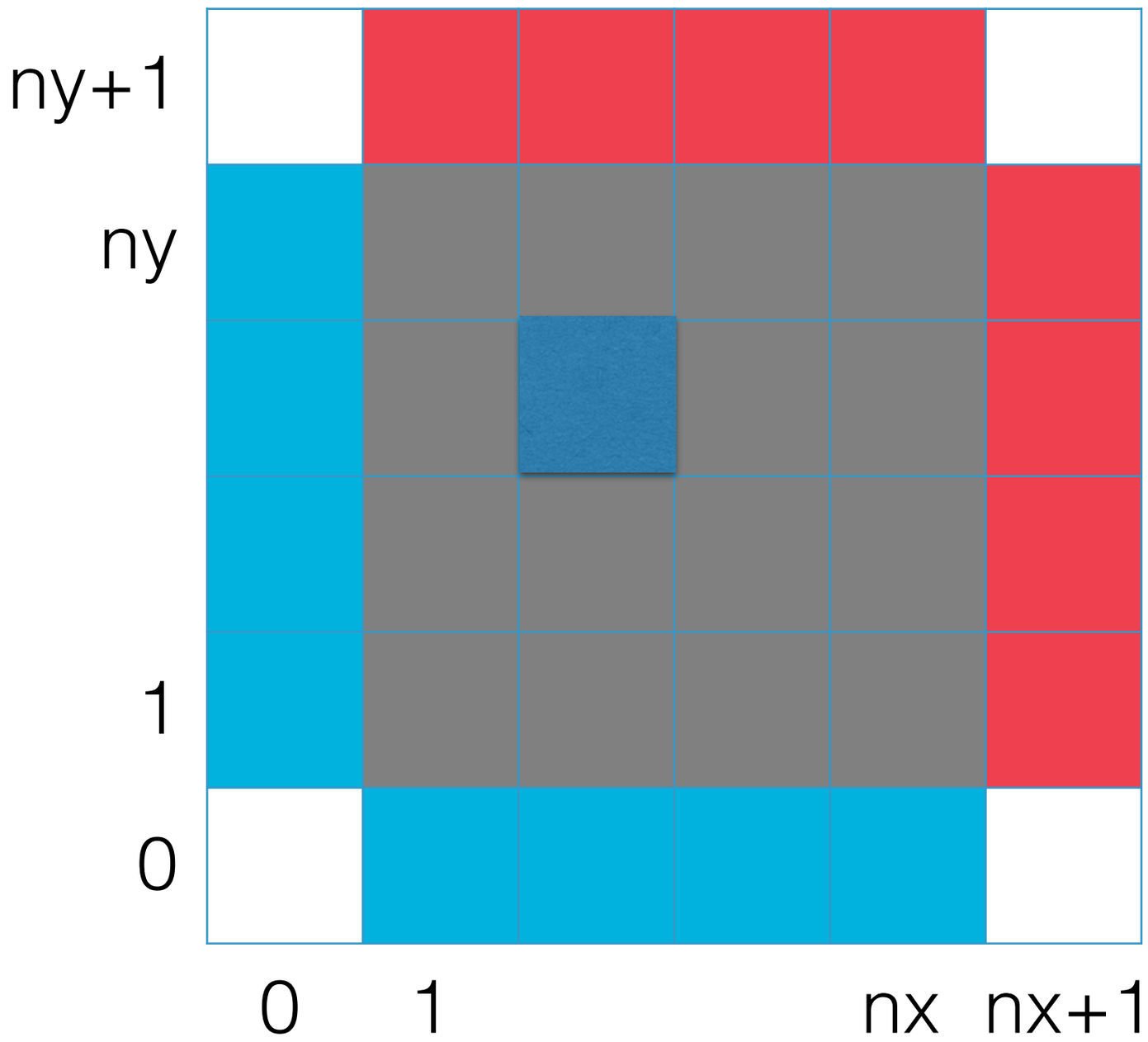
Brief details

- 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”
- Iterative solution of final answer - not change in time
- Hot and cold edges are implemented as a “halo” of “ghost” or “guard” cells

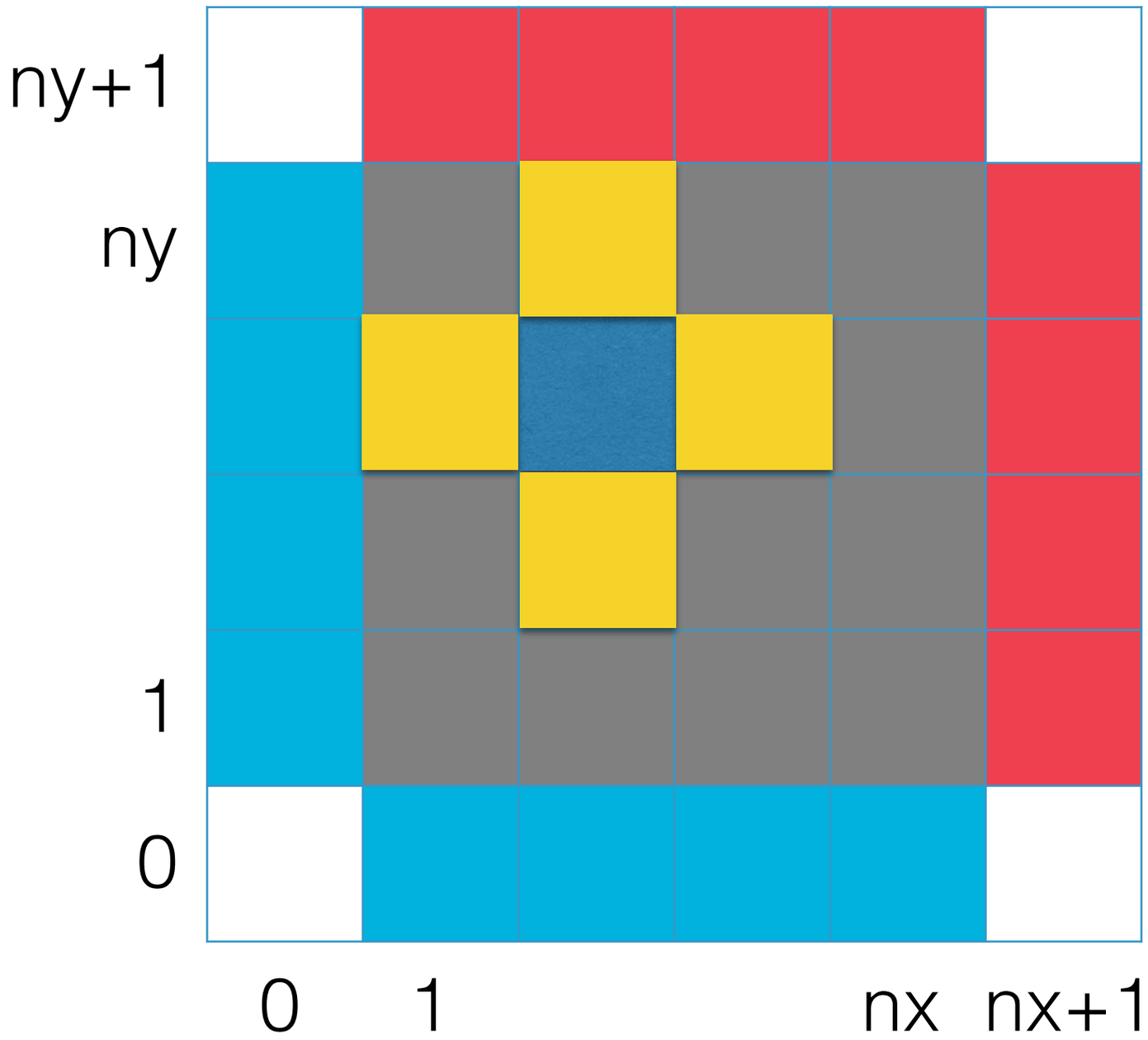
Diagram



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

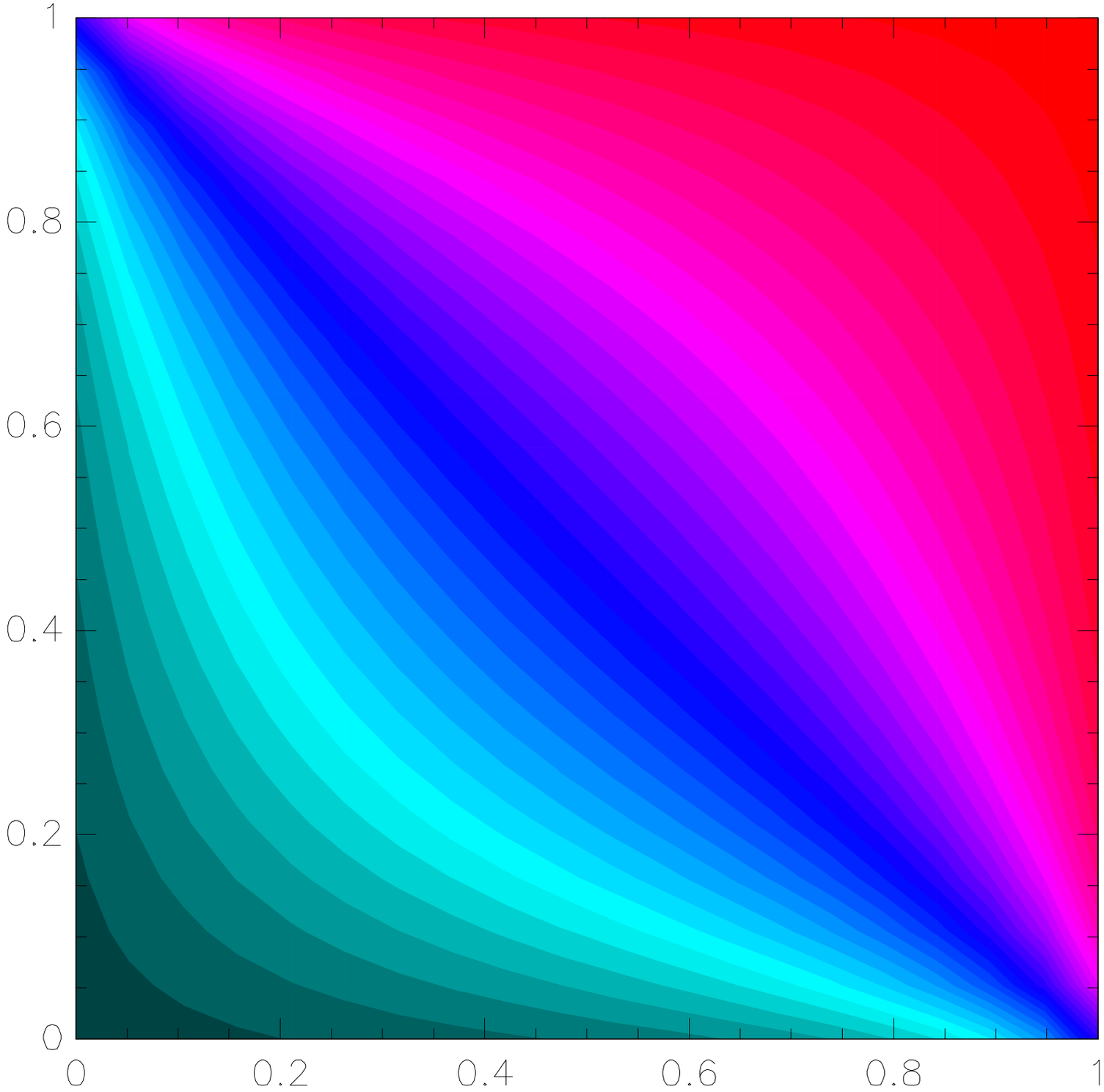
Fortran Code

```
INTEGER, PARAMETER :: nx = 20, ny = 20
REAL, DIMENSION(0:nx+1, 0:ny+1) :: values, temp_values
INTEGER :: ix, iy, icycle

values = 5.5
values(0,:) = 1.0
values(nx+1,:) = 10.0
values(:,0) = 1.0
values(:, ny+1) = 10.0

CALL display_result(values)
PRINT *,'Please press a key to advance'
READ(*,*)
DO icycle = 1, 500
  DO iy = 1, ny
    DO ix = 1, nx
      temp_values(ix,iy) = 0.25 * (values(ix+1,iy) + &
           values(ix,iy+1) + values(ix-1,iy) + values(ix,iy-1))
    END DO
  END DO
  values(1:nx,1:ny) = temp_values(1:nx,1:ny)
  IF (MOD(icycle,50) == 0) THEN
    CALL display_result(values)
    PRINT *,'Please press a key to advance'
    READ (*,*)
  ENDIF
END DO
```

C Code

```c
grid_type values, temp_values;
int ix, iy, icycle;
//Allocate a 2D array with indices that run 0->nx+1 and 0->ny+1
//This replicates Fortran's arrays with variable starts and ends
allocate_grid(&values, 0, nx+1, 0, ny+1);
allocate_grid(&temp_values, 0, nx+1, 0, ny+1);
//Assign the value 5.5 to the whole grid
assign_grid(&values, 0, nx+1, 0, ny+1, 5.5);
//Assign the boundary conditions. 1.0 along the left and bottom
//10.0 along the right and top
assign_grid(&values, 0, 0, 0, ny+1, 1.0);
assign_grid(&values, nx+1, nx+1, 0, ny+1, 10.0);
assign_grid(&values, 0, nx+1, 0, 0, 1.0);
assign_grid(&values, 0, nx+1, ny+1, ny+1, 10.0);

//To a C programmer, this looks backwards, but the array is using
//Fortran ordering deliberately
for (icycle=0;icycle<500;++icycle){
  for (iy=1;iy<=ny;++iy){
    for (ix=1;ix<=nx;++ix){
      *(access_grid(&temp_values, ix, iy)) = 0.25 * (
          *(access_grid(&values, ix+1, iy  )) +
          *(access_grid(&values, ix  , iy+1)) +
          *(access_grid(&values, ix-1, iy  )) +
          *(access_grid(&values, ix  , iy-1)));
    }
  }
  copy_grid(&values, &temp_values, 1, nx, 1, ny);
  if(icycle%50==0){
    display_result(&values);
    getchar();
  }
}
```

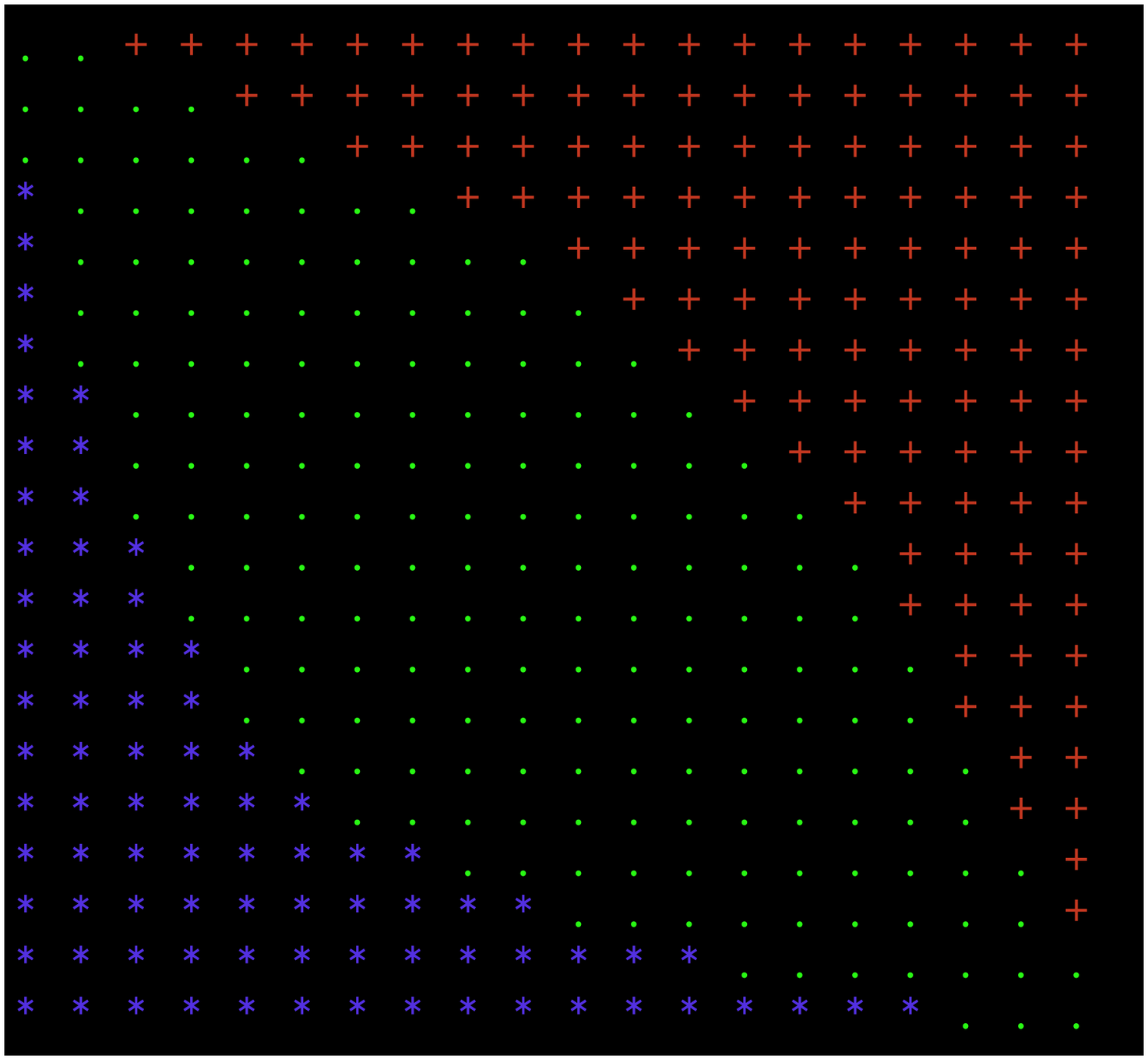
Result



Hot end

Cold End

Result



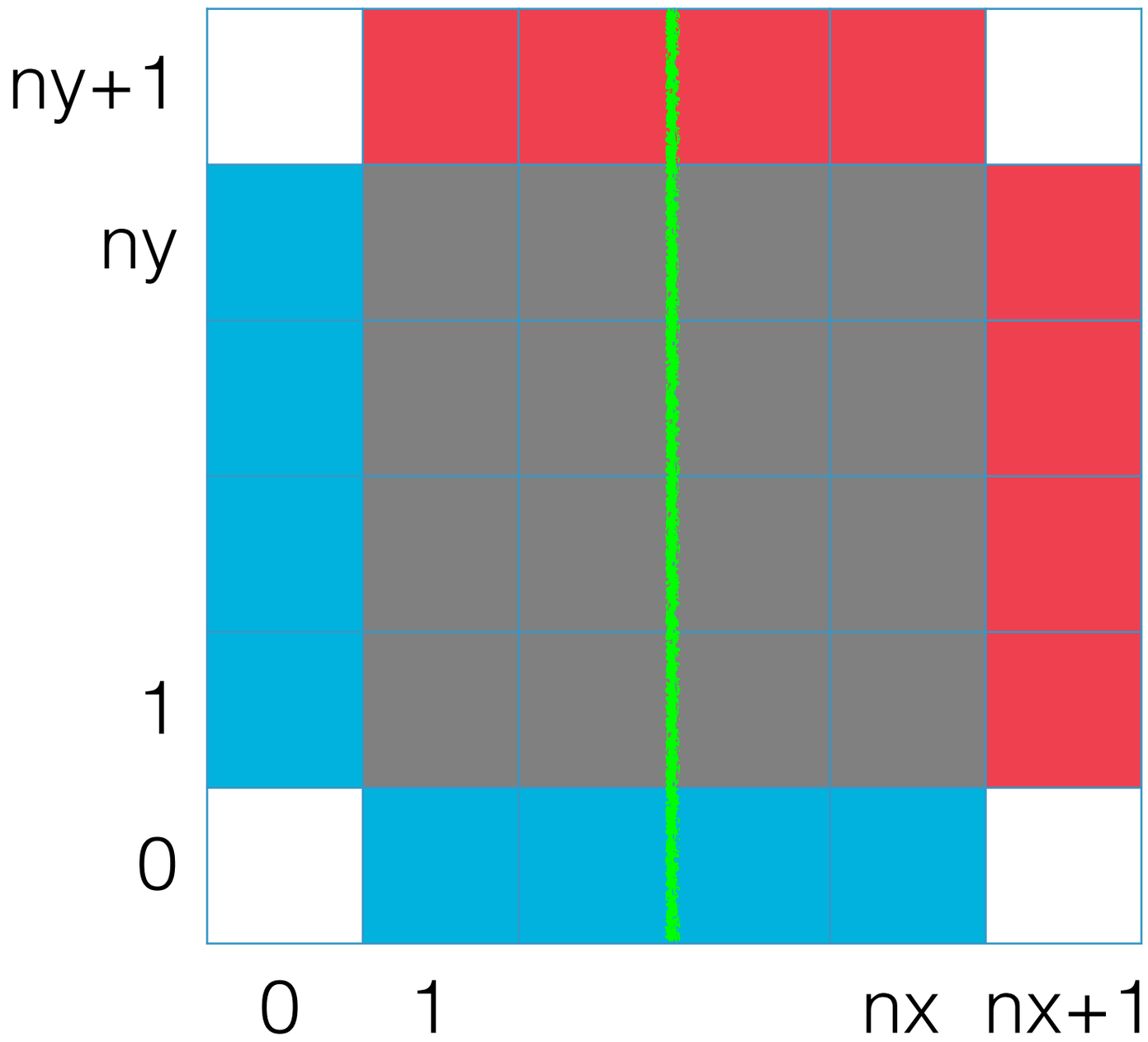
Hot end

Cold End

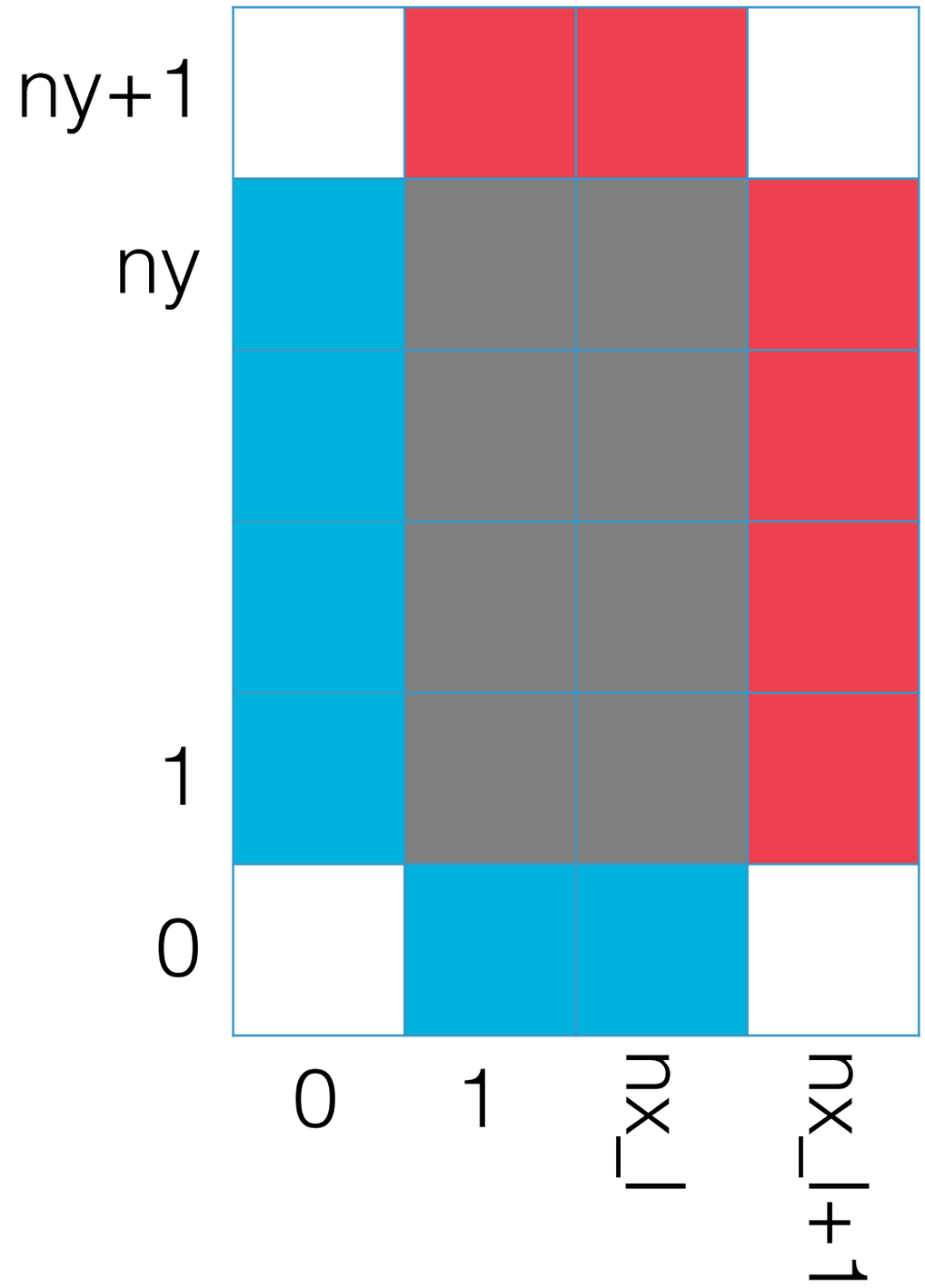
So, serial version works

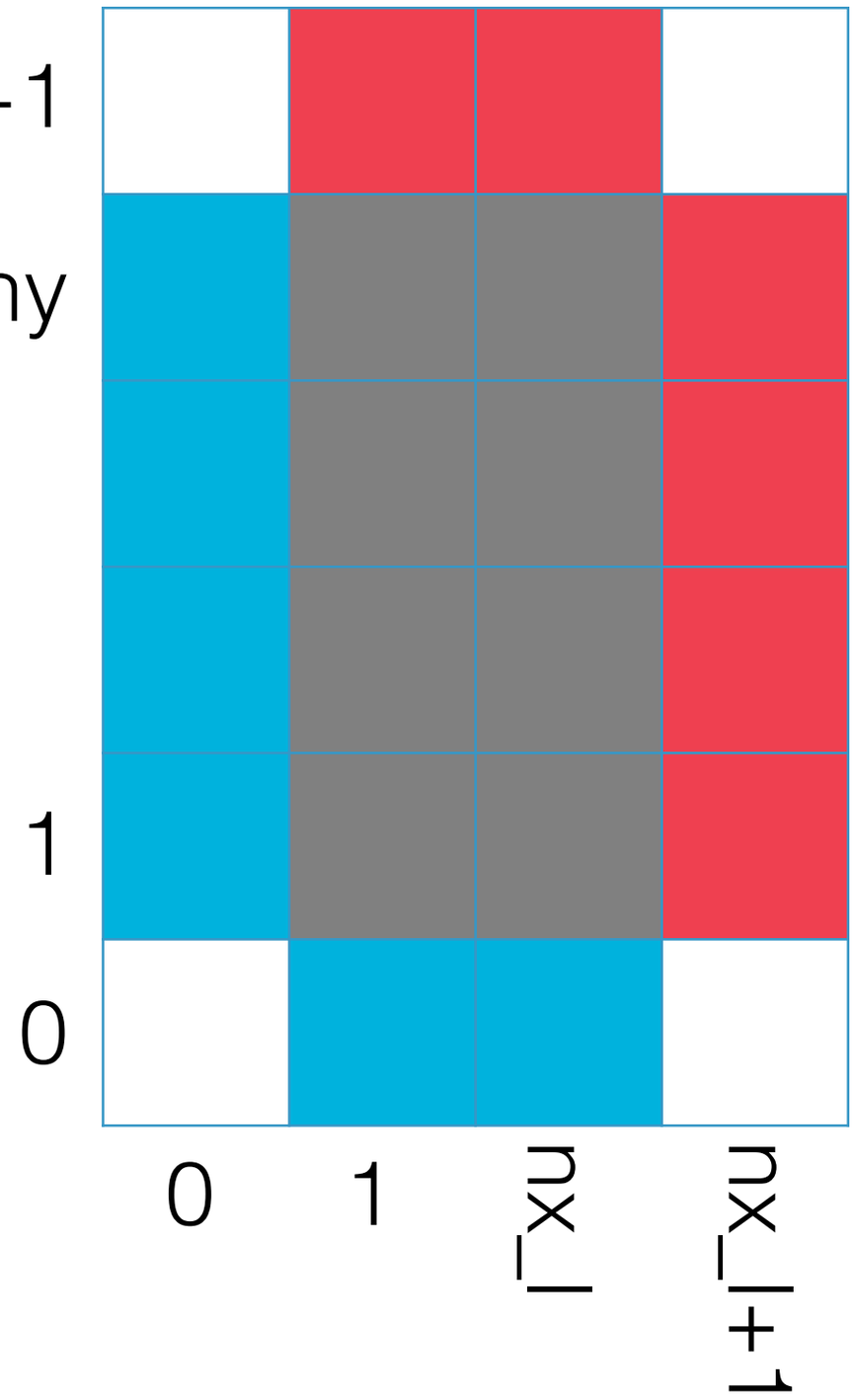
- Result is smooth gradient across the box
- Doesn’t matter though for this course
- Only need to get *the same* answer in serial and parallel
- So how do we do it in parallel?

Diagram

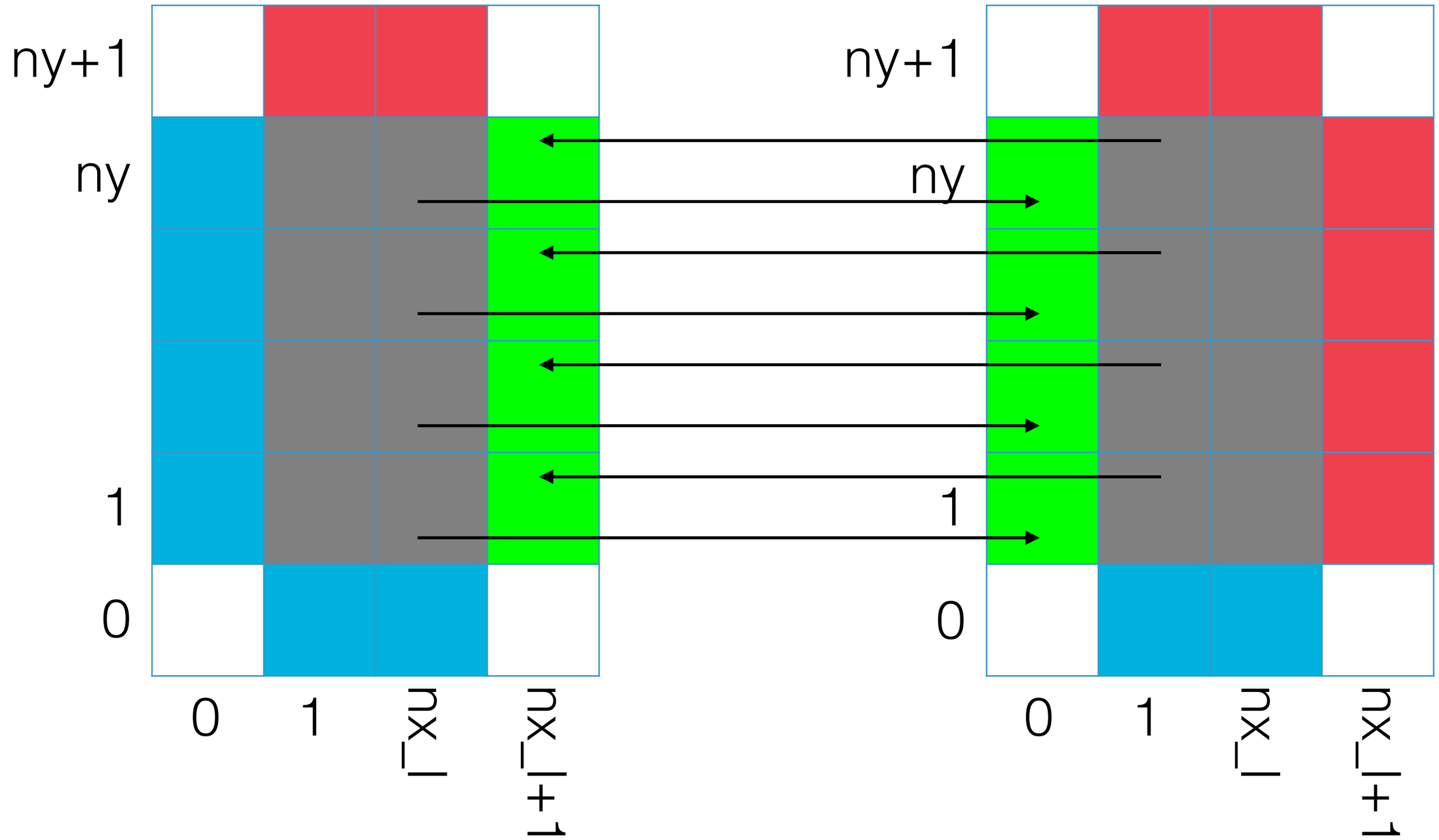


Domain decomposition





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)
- Going to 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
- Used wherever MPI_COMM_WORLD is normally
- Will in general have different rank in different communicator

MPI Topologies

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const int
periods[], int reorder, MPI_Comm *comm_cart)
```

- `comm_old` - Usually MPI_COMM_WORLD
- `ndims` - Number of dimensions in new communicator. Usually same as problem
- `dims` - Number of processors in each direction
- `periods` - Should processor topology wrap round at edges
- `reorder` - Should processor rank be changed to allow better map to physical hardware
- `comm_cart` - Cartesian partitioned communicator

Creating ndims

```
int MPI_Dims_create(int nnodes, int ndims, int dims[])
```

- `nnodes` - Generally the number of processors
- `ndims` - Number of dimensions wanted
- `dims` - Output array of number of processors in each dimension
  - Is “sensible” not necessarily optimal

Cartesian topology

0	1	2	3

0,3	1,3	2,3	3,3
0,2	1,2	2,2	3,2
0,1	1,1	2,1	3,1
0,0	1,0	2,0	3,0

Getting coordinates

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int
coords[])
```

- **`comm`** - should be a Cartesian communicator from **`MPI_Cart_create`**
- **`rank`** - rank of processor that you want to know the coordinates of. Rank should be in **`comm`**, not **`MPI_COMM_WORLD`**
- **`maxdims`** - effectively length of **`coords`** array
- **`coords`** - output array containing coordinates of **`rank`**

Getting rank from coordinates

```
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
```

- `comm` - should be a Cartesian communicator from `MPI_Cart_create`
- `coords` - coordinates in `coords`
- `rank` - output rank for specified `coords`
- `coords` must be within range or an error occurs
  - Does NOT return `MPI_PROC_NULL`

Getting neighbours

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
```

- `comm` - should be a Cartesian communicator from `MPI_Cart_create`
- `direction` - 0 based integer saying which direction to get neighbours in
- `disp` - nth neighbour away. 1 gets nearest neighbours, 2 second nearest etc.
- `rank_source` - rank of processor 1 lower in `direction`-th coordinate
- `rank_dest` - rank of processor 1 higher in `direction`-th coordinate

Getting neighbours

- Processors at edges of domain are MPI_PROC_NULL unless you specify periodic boundaries
- If you do then processor on opposite edge is automatically mapped round by MPI
- “free” periodic boundary conditions
- To get corner neighbours must use MPI_Cart_rank

Back to case study

Cartesian communicator

```
CALL MPI_Init(ierr)

CALL MPI_Dims_create(nproc, 2, nprocs, ierr)
CALL MPI_Cart_create(MPI_COMM_WORLD, 2, nprocs, periods, .TRUE., &
    cart_comm, ierr)
!Rank in new communicator
CALL MPI_Comm_rank(cart_comm, rank, ierr)

!Get the rank of the neighbouring processors in Cartesian communicator
CALL MPI_Cart_shift(cart_comm, 0, 1, x_min_rank, x_max_rank, ierr)
CALL MPI_Cart_shift(cart_comm, 1, 1, y_min_rank, y_max_rank, ierr)
!Get my coordinates in Cartesian communicator
CALL MPI_Cart_coords(cart_comm, rank, 2, coordinates, ierr)

!Divide the global size (nx x ny) per processor
!Note that MPI_Dims_create works backwards
nx_local = nx / nprocs(1)
ny_local = ny / nprocs(2)

!Calculate what fraction of the global array this processor has
x_cell_min_local = nx_local * coordinates(1) + 1
x_cell_max_local = nx_local * (coordinates(1) + 1)
y_cell_min_local = ny_local * coordinates(2) + 1
y_cell_max_local = ny_local * (coordinates(2) + 1)
```

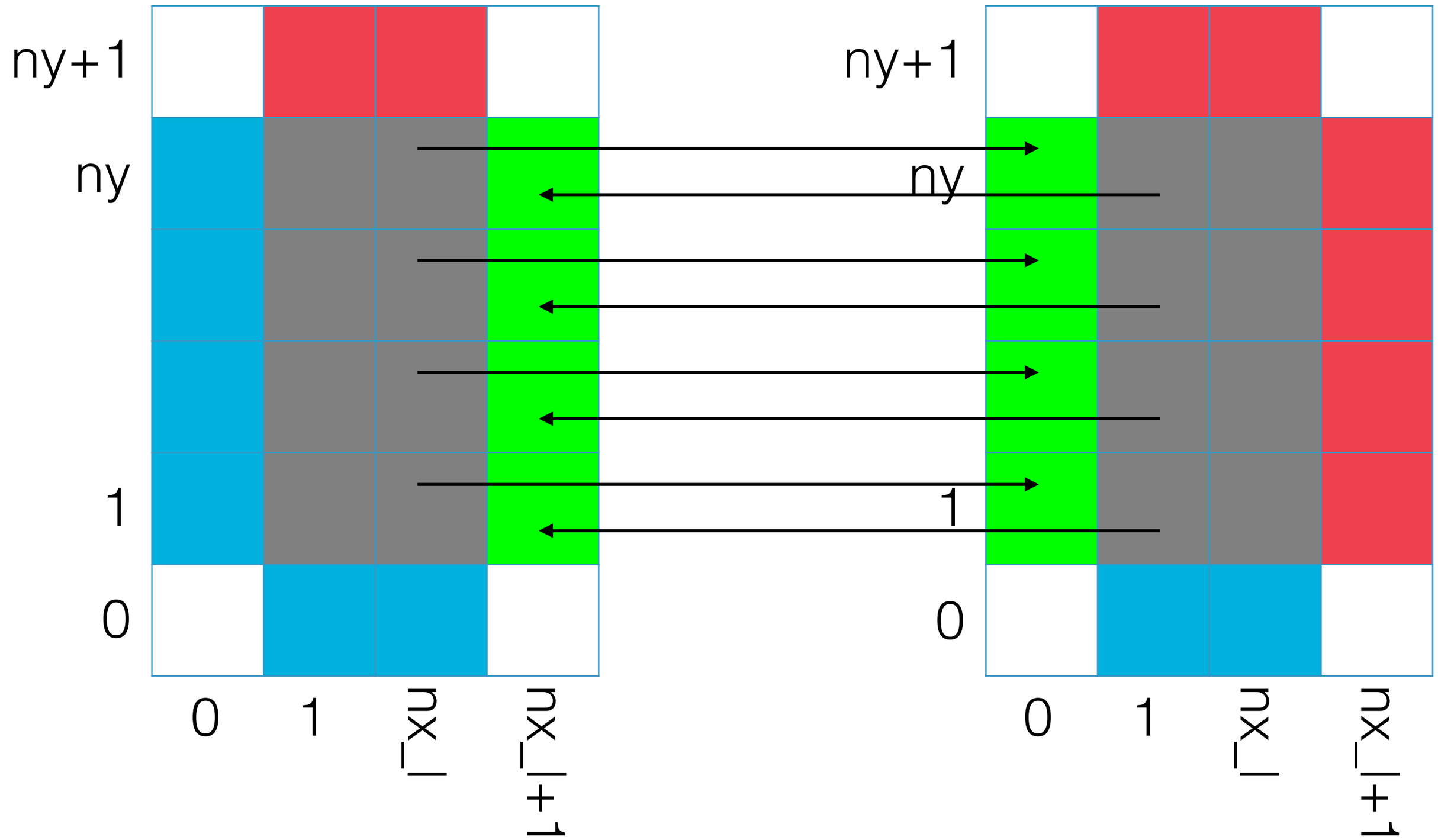
Cartesian communicator

```
MPI_Init(argc, argv);
MPI_Dims_create(nproc, 2, nprocs);
MPI_Cart_create(MPI_COMM_WORLD, 2, nprocs, periods, 1,
    &cart_comm);
//Rank in new communicator might be different
MPI_Comm_rank(cart_comm, &rank);

//Get the rank of the neighbouring processors in Cartesian communicator
MPI_Cart_shift(cart_comm, 0, 1, &x_min_rank, &x_max_rank);
MPI_Cart_shift(cart_comm, 1, 1, &y_min_rank, &y_max_rank);

//Get my coordinates in Cartesian communicator
MPI_Cart_coords(cart_comm, rank, 2, coordinates);

//Divide the global size (nx x ny) per processor
nx_local = nx / nprocs[0];
ny_local = ny / nprocs[1];

//Calculate what fraction of the global array this processor has
x_cell_min_local = nx_local * coordinates[0] + 1;
x_cell_max_local = nx_local * (coordinates[0] + 1);
y_cell_min_local = ny_local * coordinates[1] + 1;
y_cell_max_local = ny_local * (coordinates[1] + 1);
```

Current state

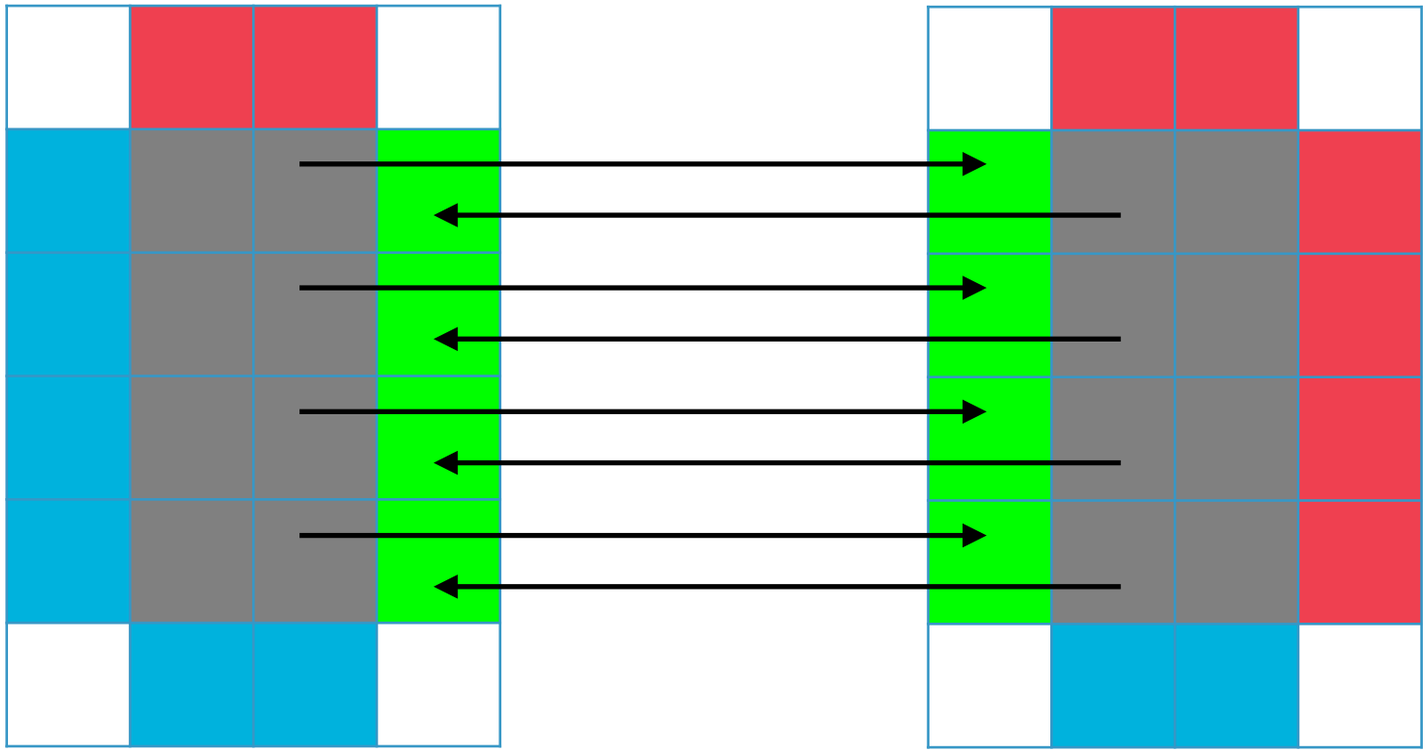
- Now have code to create MPI communicator
- Create a version of nx_l and ny_l (remember called **nx_local** and **ny_local** in the code to avoid l looking like 1)
- Serial core of the code is just rewritten to use **nx_local** and **ny_local** as the size of arrays
- Now need to write the boundary conditions

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)

Fortran boundary conditions

```
!Send left most strip of cells left and receive into right guard cells
CALL MPI_Sendrecv(array(1,1:ny_local), ny_local, MPI_REAL, x_min_rank, &
    tag, array(nx_local+1,1:ny_local), ny_local, MPI_REAL, 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(nx_local, 1:ny_local), ny_local, MPI_REAL, &
    x_max_rank, tag, array(0,1:ny_local), ny_local, MPI_REAL, x_min_rank, &
    tag, cart_comm, MPI_STATUS_IGNORE, ierr)

!Now equivalently in y
CALL MPI_Sendrecv(array(1:nx_local,1), nx_local, MPI_REAL, y_min_rank, &
    tag, array(1:nx_local,ny_local+1), nx_local, MPI_REAL, y_max_rank, &
    tag, cart_comm, MPI_STATUS_IGNORE, ierr)

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

- Use array subsections to tell MPI what data to send and receive

- Do nothing at the real domain edges because not using periodic domain, so MPI_Sendrecv is a null operation

C boundary conditions

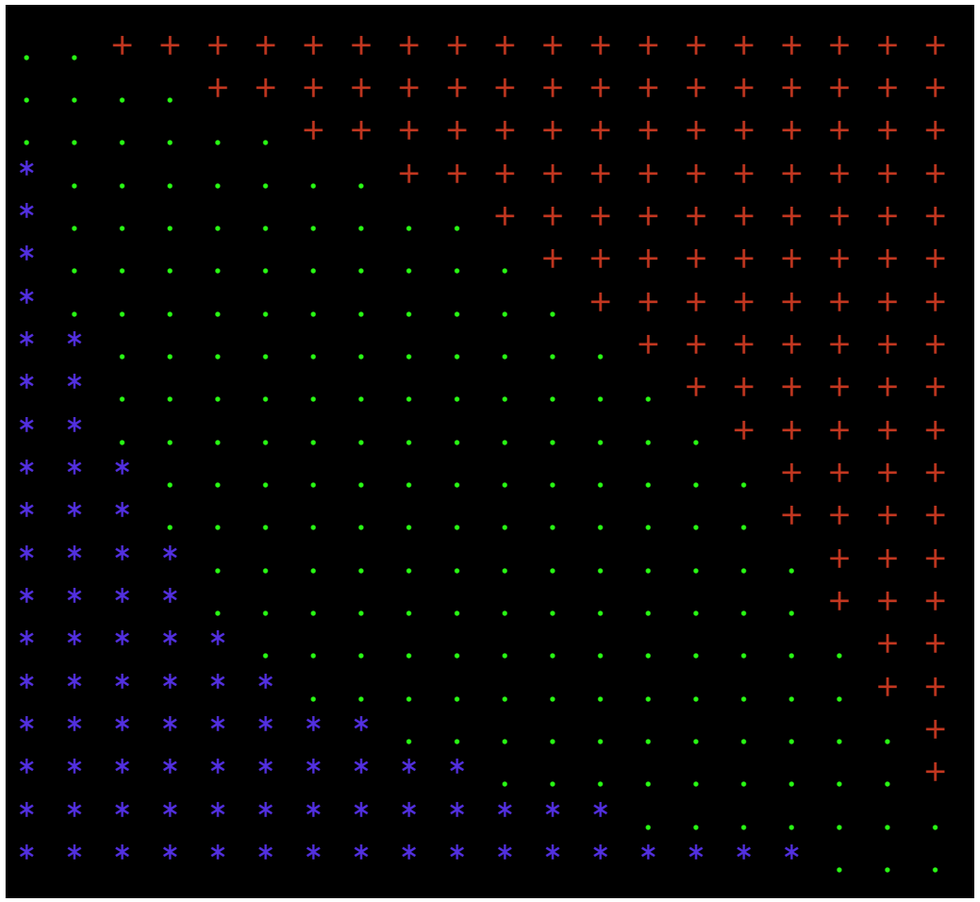
```
//Unlike in Fortran, can't use array subsections. Have to copy to temporaries
src = (float*) malloc(sizeof(float)*(ny_local));
dest = (float*) malloc(sizeof(float)*(ny_local));

//Send left most strip of cells left and receive into right guard cells
for (index = 1; index<=ny_local; ++index){
  src[index-1] = *(access_grid(data, 1, index ));
  //Copy existing numbers into dest because MPI_Sendrecv is a no-op if
  //one of the other ranks is MPI_PROC_NULL
  dest[index-1] = *(access_grid(data, nx_local + 1, index ));
}

MPI_Sendrecv(src, ny_local, MPI_FLOAT, x_min_rank,
    TAG, dest, ny_local, MPI_FLOAT, x_max_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);

for (index = 1; index<=ny_local; ++index){
  *(access_grid(data, nx_local + 1, index )) = dest[index-1];
}
```

- Code for single exchange
- Now have to explicitly copy data into temporary arrays since no array subsections in C

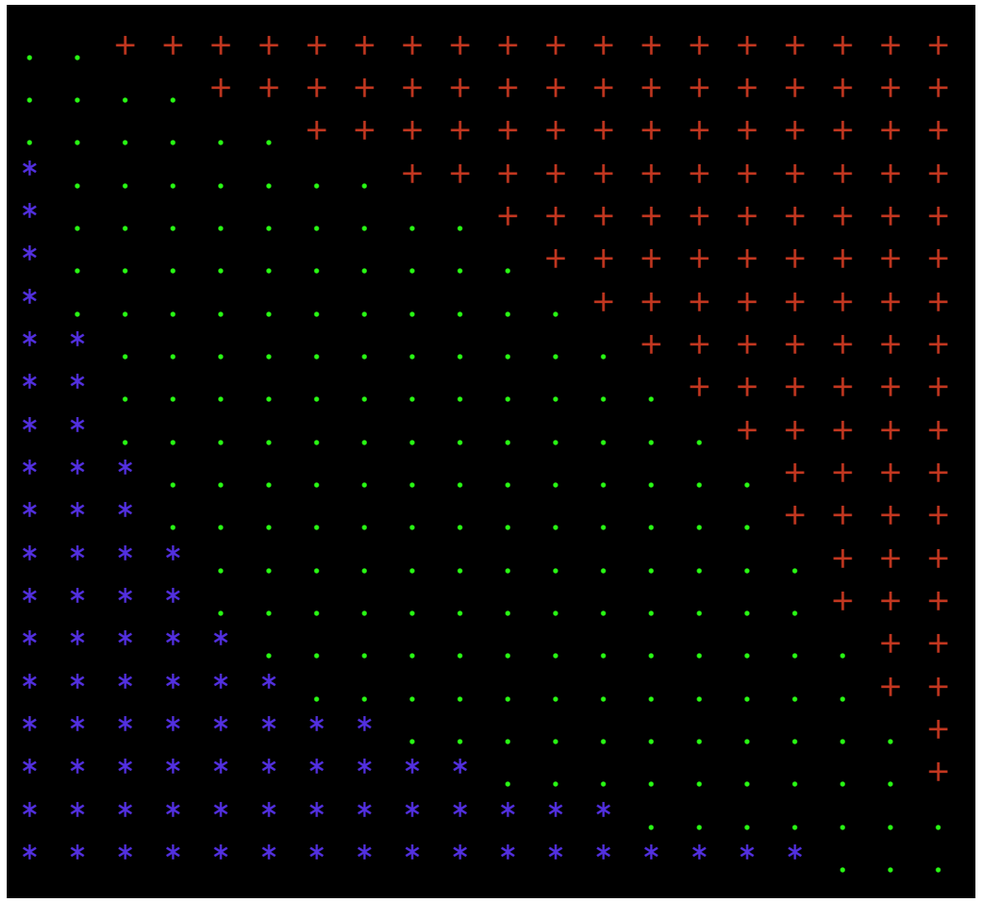
Boundary conditions

- In Fortran, using array subsections works pretty well
- Stack space problems with some compilers
- In C, using temporary arrays is very ugly
- But it does work perfectly fine

Results



1 Processor



16 Processors