Domain Decomposition

Warwick RSE

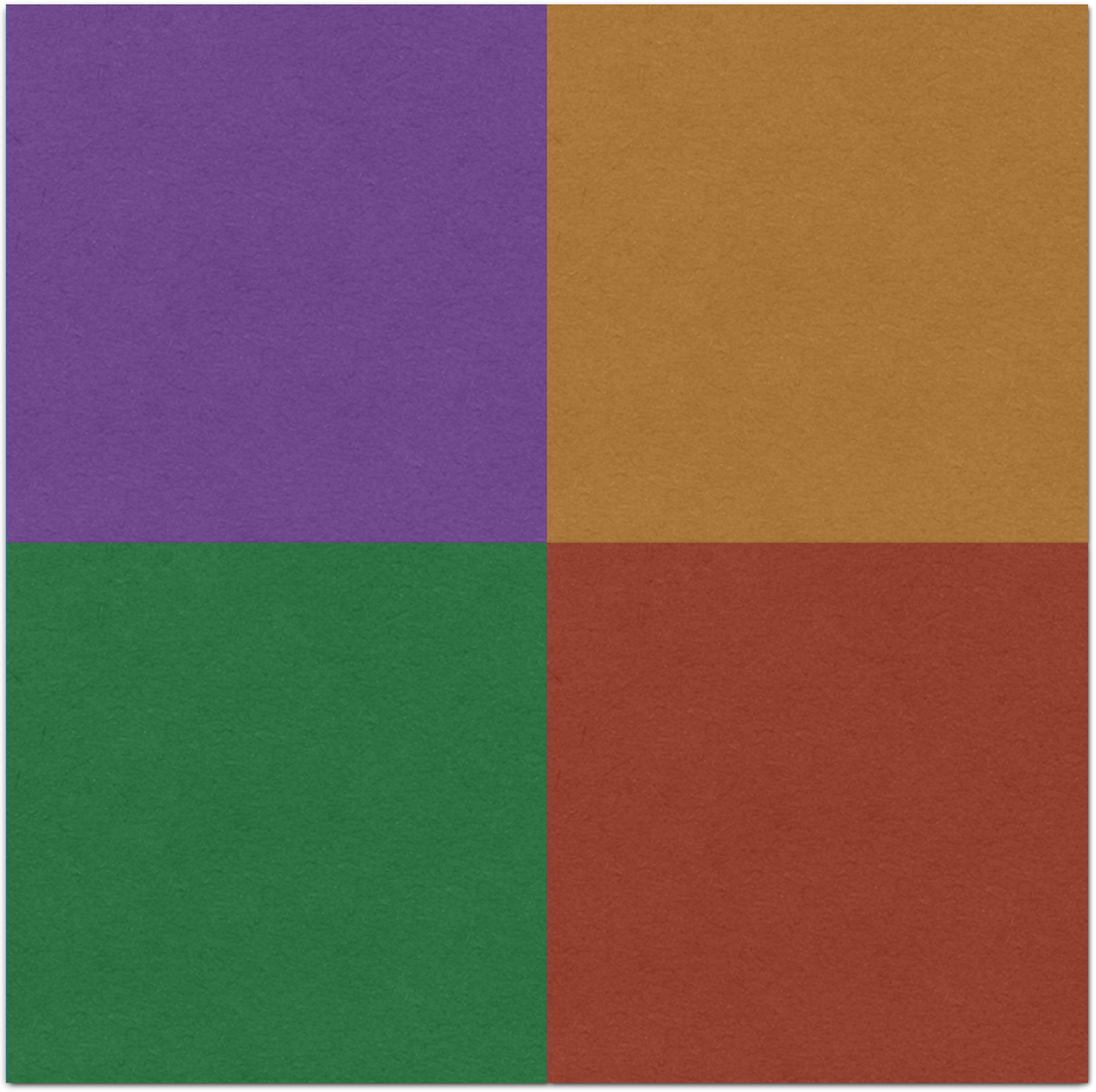
Overview

- That’s most of the MPI commands that you’re going to use
- How do you actually use them?
- Very much depends on your problem
- You need to **decompose** your problem somehow

Overview

- In a working MPI program you effectively want to make your program act as though there was one very powerful processor
- Ignore as far as possible that there's lots of them
- Want to split up the problem to make that as simple as possible
- How depends very much on the problem at hand

Domain decomposition



Domain decomposition

- Split up the “space” of your problem and hand each bit to a separate processor
- Space very often means literal space, but it can mean momentum, energy or any number of other things
- Generally, you have an array and you give different bits to different processors
- You communicate the data that this processor needs from other processors using messages
- So long as your algorithm is **local** this works very well
  - Only use grid-cells near the current one in your algorithm

Domain Decomposition

- Lots of examples of algorithms like this
  - Finite difference/volume/element schemes
  - Iterative schemes for certain types of matrix inversion (Jacobi, Gauss-Seidel (with care)) etc.
- Non local schemes include
  - Direct matrix inversion
  - Iterative inversion of dense matrices
  - Fourier Transforms
- They can be made to work but never scale as well as local algorithms

Domain Decomposition

- Domain decomposition is not restricted to cases where you have quantities residing on a grid
- Codes calculating pairwise interactions between particles are also parallelised
  - Including tree schemes like Barnes-Hutt
- So are mesh free schemes (Mesh Free Galerkin etc.)
- The parallelism is much messier and less illustrative than for grid based schemes
- Grid based schemes are very common in actual use

Domain decomposition

- In practice domain decomposition for grid based schemes is rather simple
- Rather than having one big array (nx, ny) you have N arrays on N processors each (nx_local, ny_local)
- REMEMBER the big array doesn't actually exist anywhere
  - There are parallel schemes called Partitioned Global Address Space schemes that work like this
  - Less common than MPI and often harder to write performant code

Very simple local algorithm

- `temp[i][j] =`
  `( u[i+1][j] + u[i-1][j]`
  `+ u[i][j+1] + u[i][j-1])/4.0;`

- Copy `temp` back into `u` after iteration finished

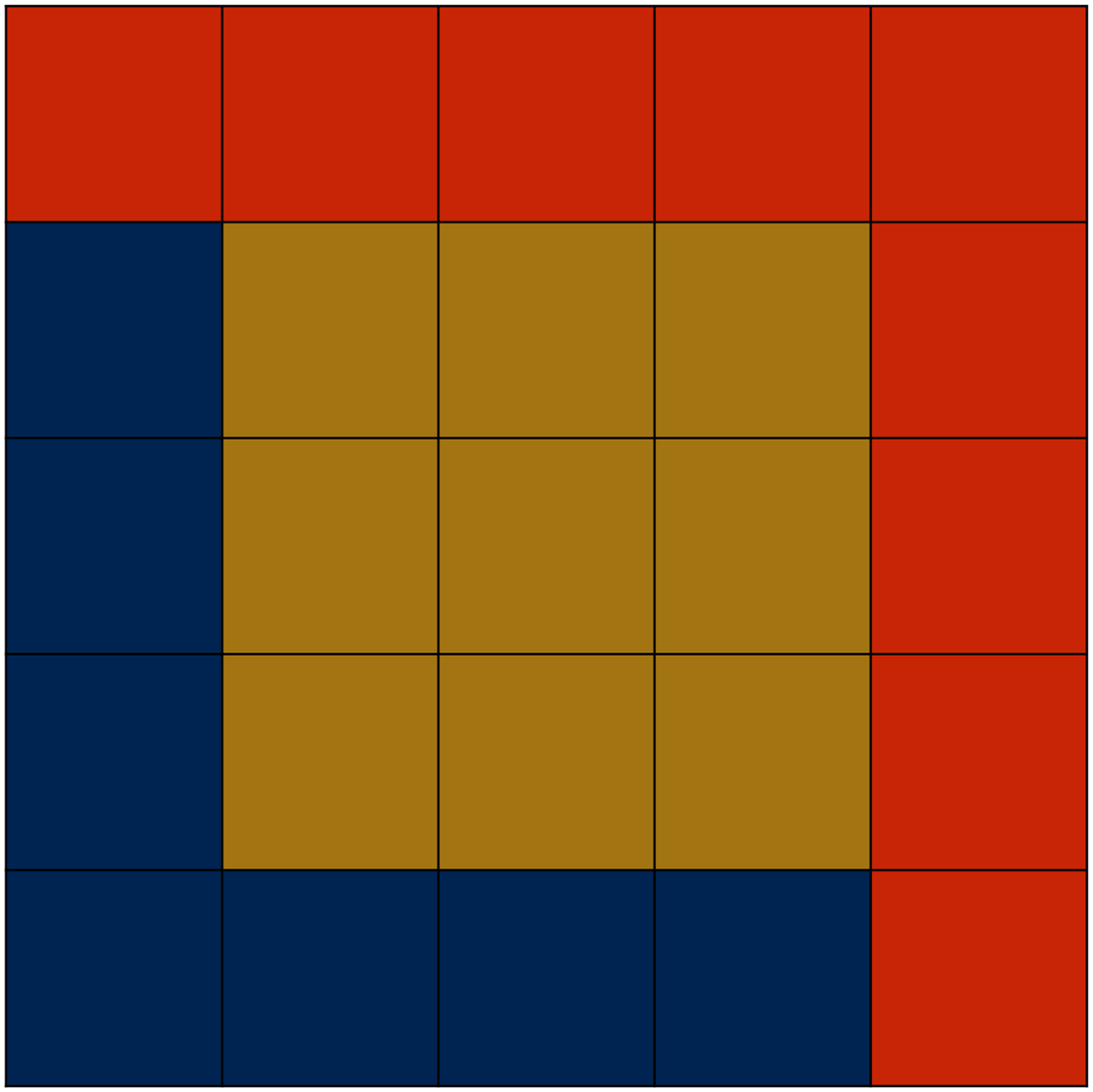
- Simply smooths `u` in space

- Only uses current points and one point to each side of current point

- Actually also solves the heat equation using the Jacobi method

- Going to use as an example

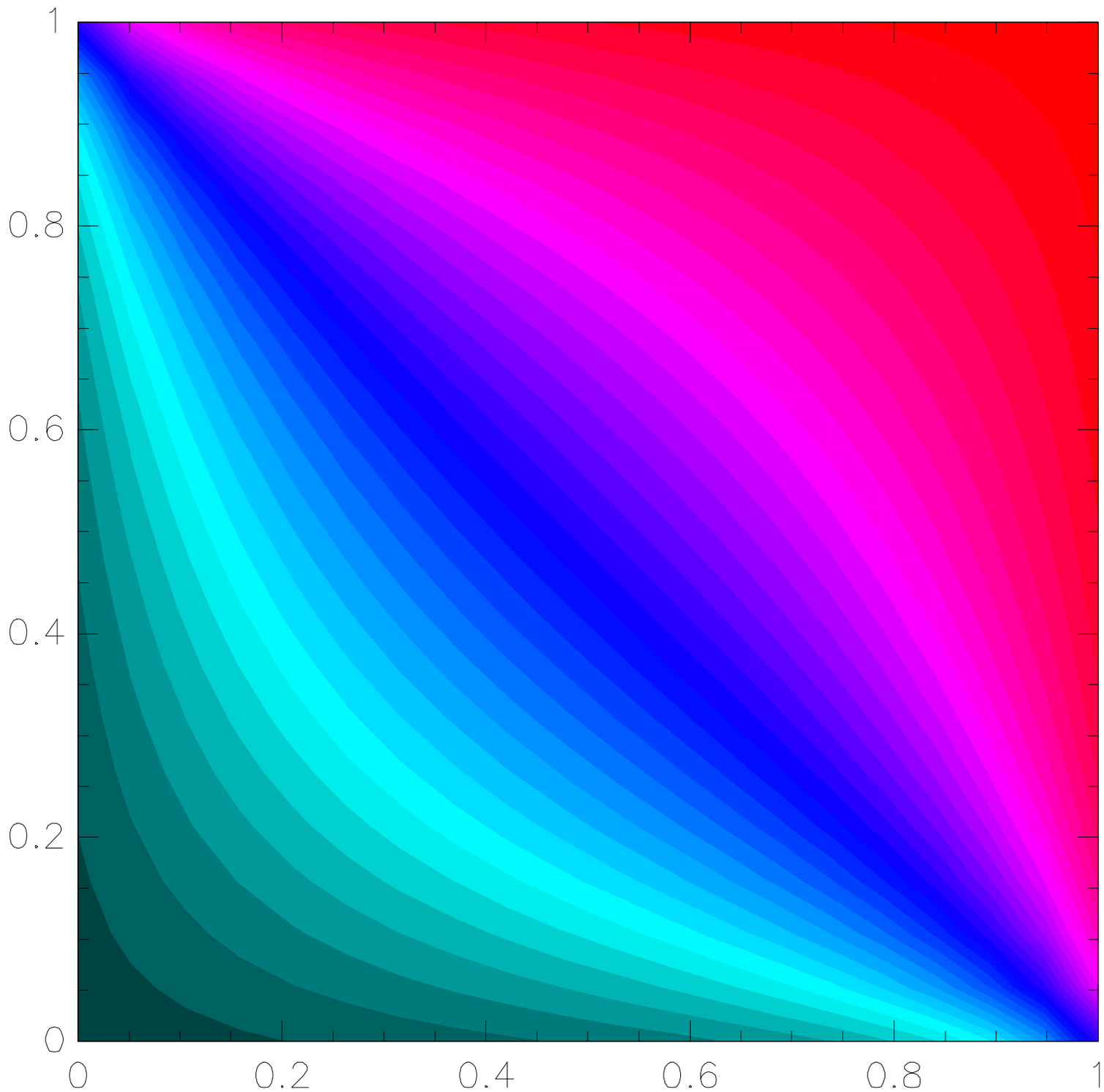
Problem



High value
(hot)

Low value
(cold)

Result

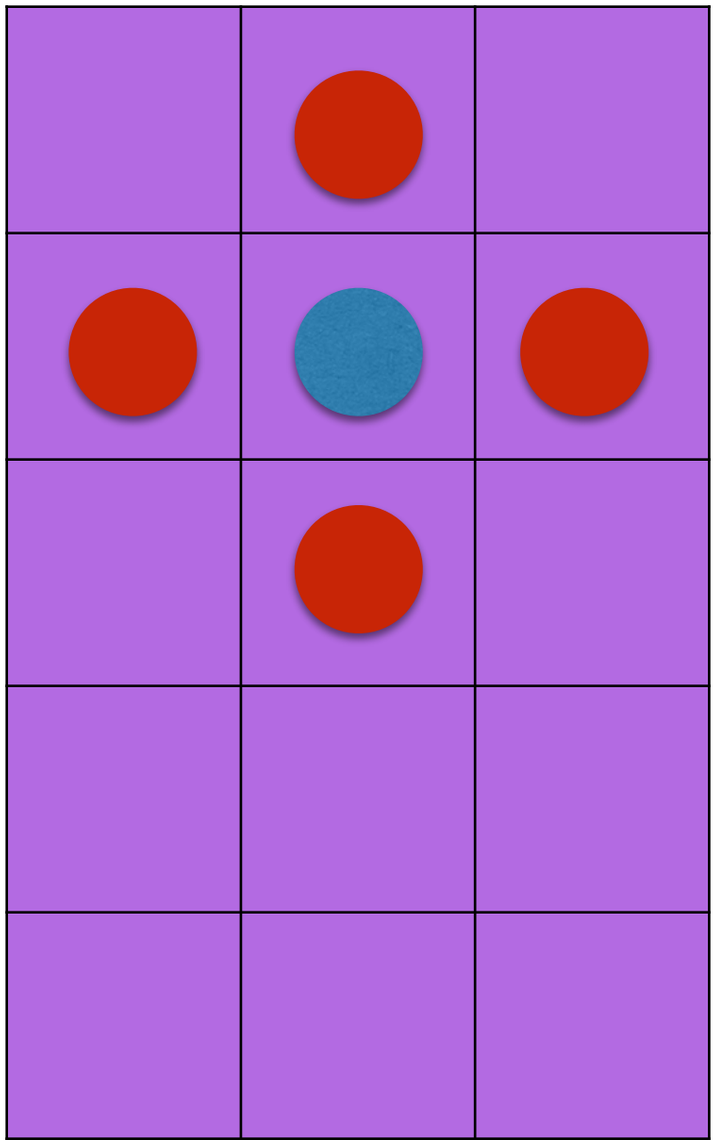


Hot end

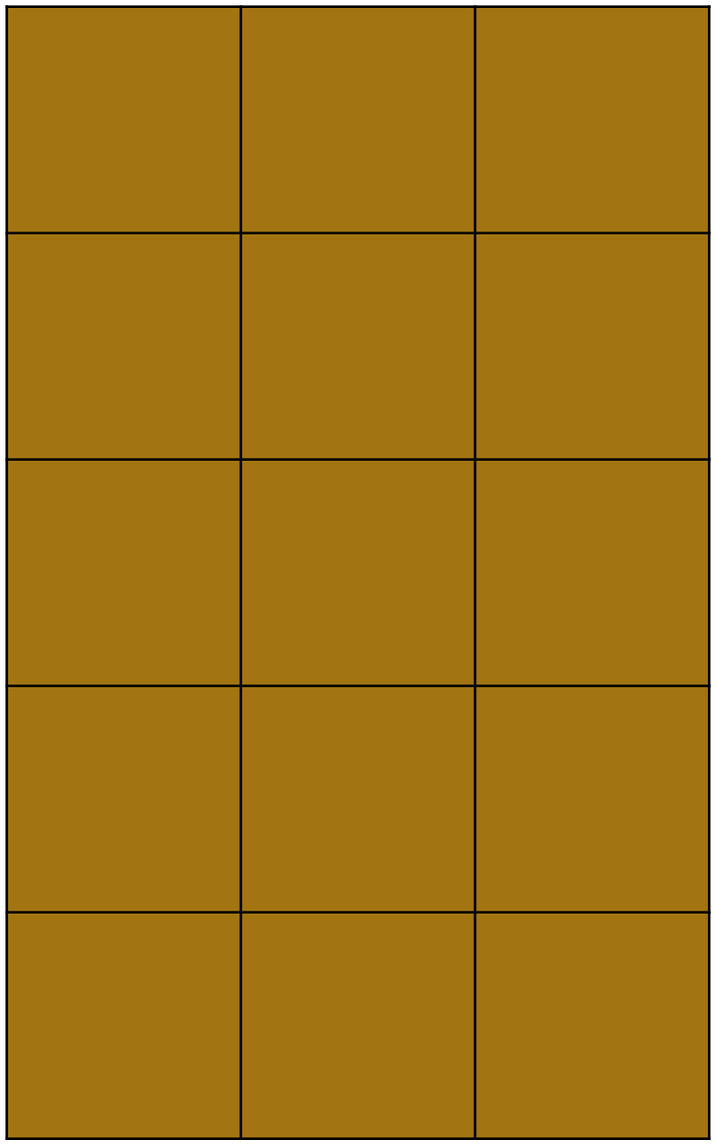
Cold End

Implementation

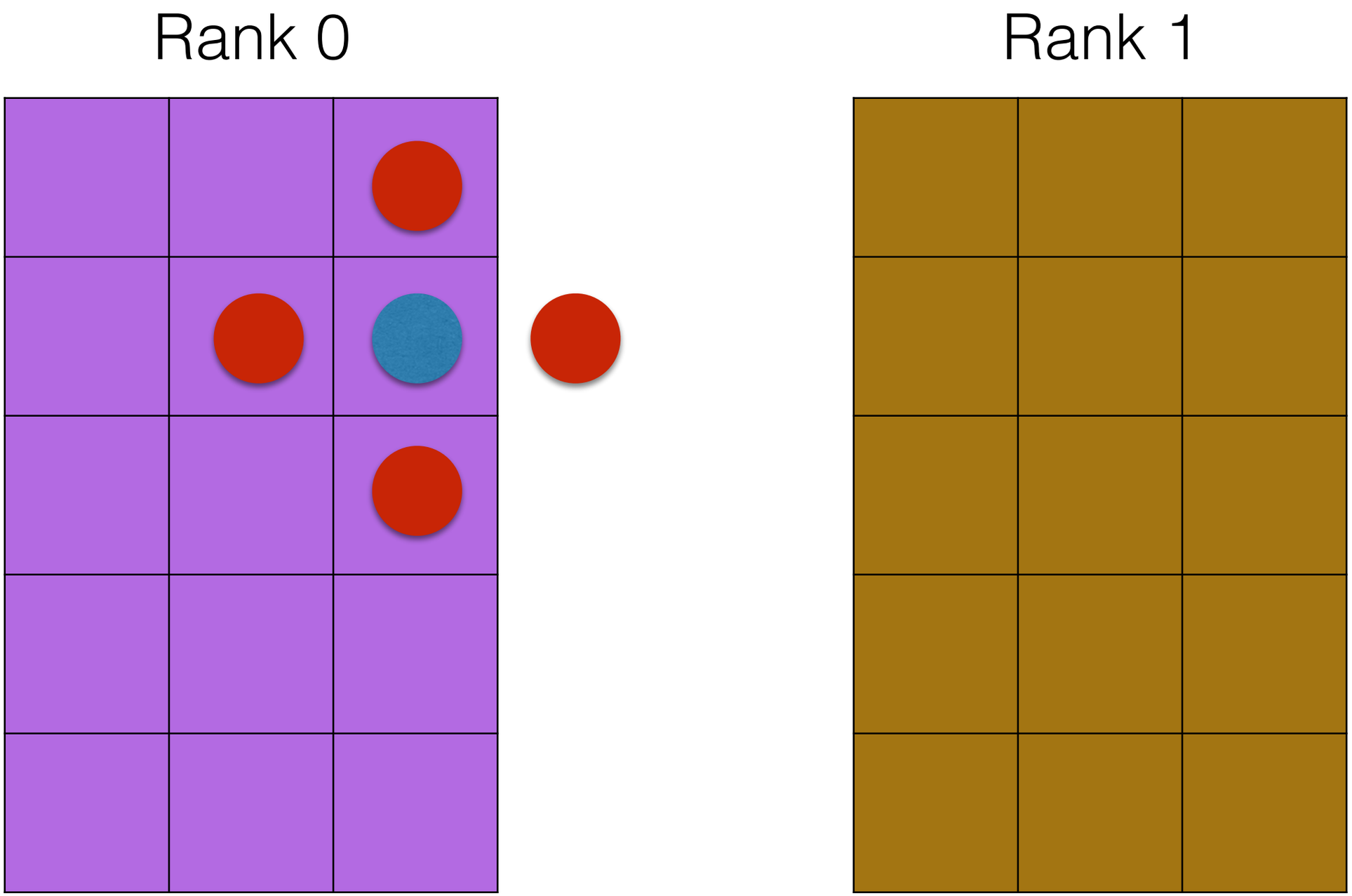
Rank 0



Rank 1



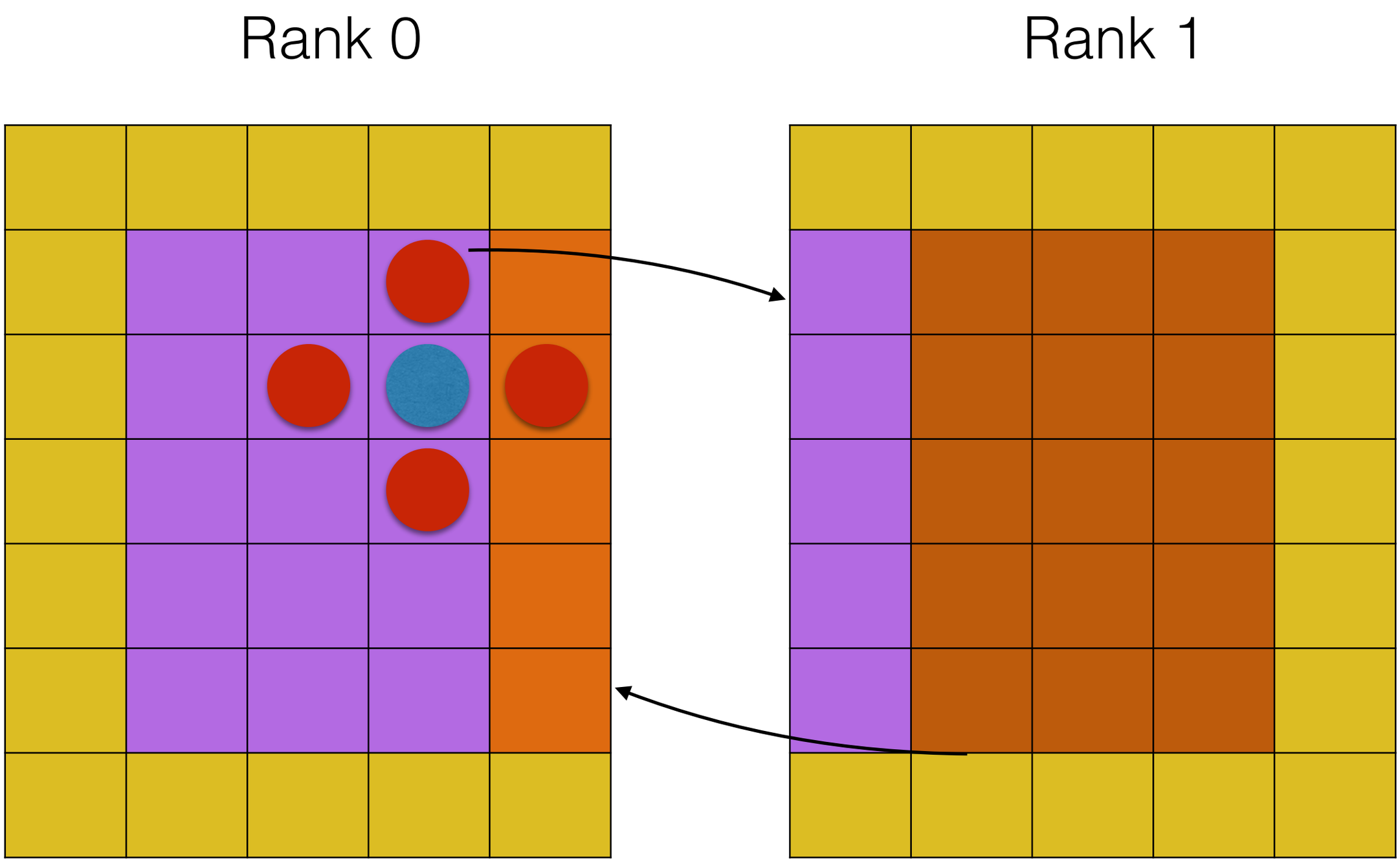
Implementation



Ghost cells

- Also called guard cells
- Cells around the edge of the domain that contain boundary values
- Very often how you just implement normal boundary conditions
- Effectively interprocessor boundaries are just boundaries
- Just send messages to update the ghost cells at the end of each iteration

Implementation



Implementation

- Perfect use for MPI_Sendrecv
- You do one Sendrecv to send the ghost cells along each edge
  - 2 in 1D
  - 4 in 2D
  - etc. etc.

Decomposition

- Before you can actually write a code you have to divide your domain up over processors

- In this course we will assume that every cell of your domain causes as much work as every other cell so you just want every processor to have as many cells as every other processor

  - If this isn't true then you have a problem of **load balancing**. They are not fun

- In 1D decomposition is easy, you just divide your number of cells by your number of processors

Decomposition

- If this isn't an integer then you can
  - Refuse to allow the code to run (easy option, but limiting)
  - Allow some domains to have more cells than others (limits performance a bit but is usually acceptable)
- In 2D it gets rather harder

Decomposition

- You have N processors but you want to split up a 2D array
  - Could do it simply by splitting in only one direction
  - This generally is not the best performing approach
  - Want to map N processors onto an (L, M) grid of processors
- Equivalent in 3D ((K, L, M) grid is wanted)

Decomposition

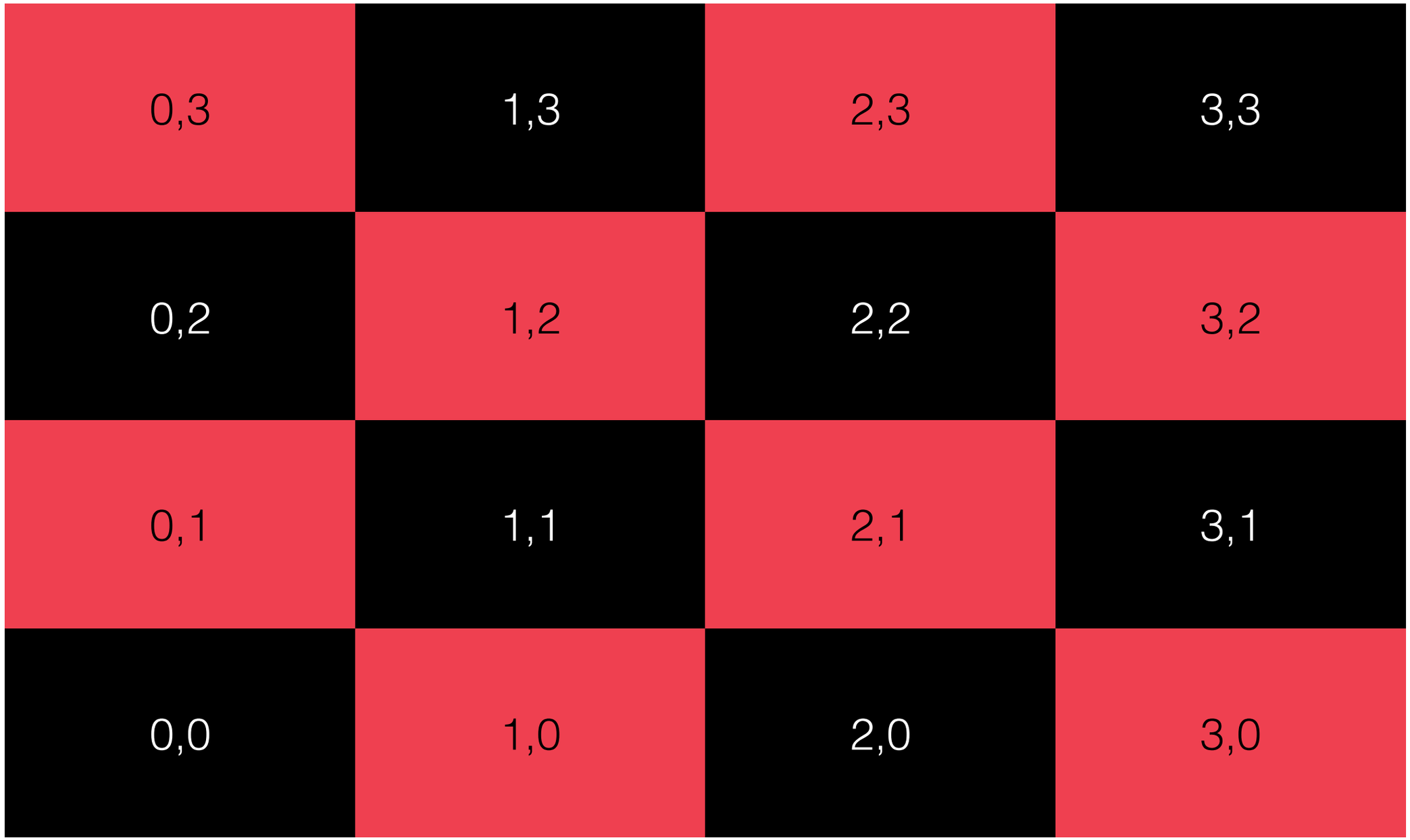
```
MPI_Dims_create(int nnodes, int ndims,
int dims[])
```

- nnodes - Number of processors (from MPI_Comm_size)
- ndims - Number of dimensions that you want to decompose in (2D, 3D etc.)
- dims - Returned array containing numbers of processors in that dimension

Decomposition

- MPI_Dims_create will give you an (L, M) decomposition of your N processors
- It will try to give as close to a square decomposition (SQRT(N), SQRT(N)) as it can
- It isn't (and can't be) guaranteed to be optimal but it will give you numbers without you worrying about it

Cartesian topology



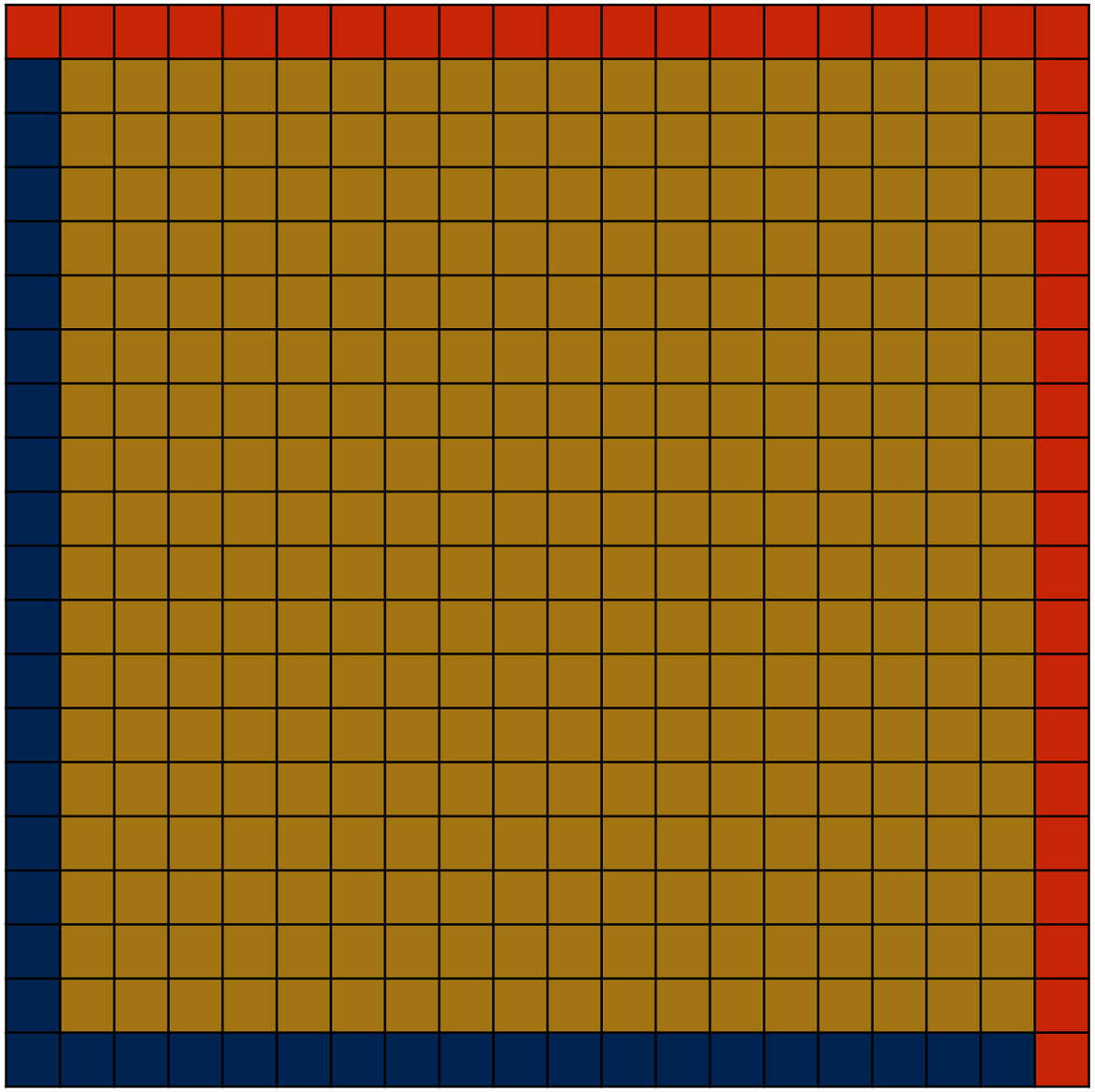
Why Decomposition Matters

- In any distributed system you want to spend as much time computing as possible and as little time communicating
- That means that you want the lowest possible surface area to volume ratio
  - You compute over the volume
  - You exchange ghost cells along edges
- Also, if you have enough processors you will run out of grid points to split in just 1D but you might still want your code to go faster

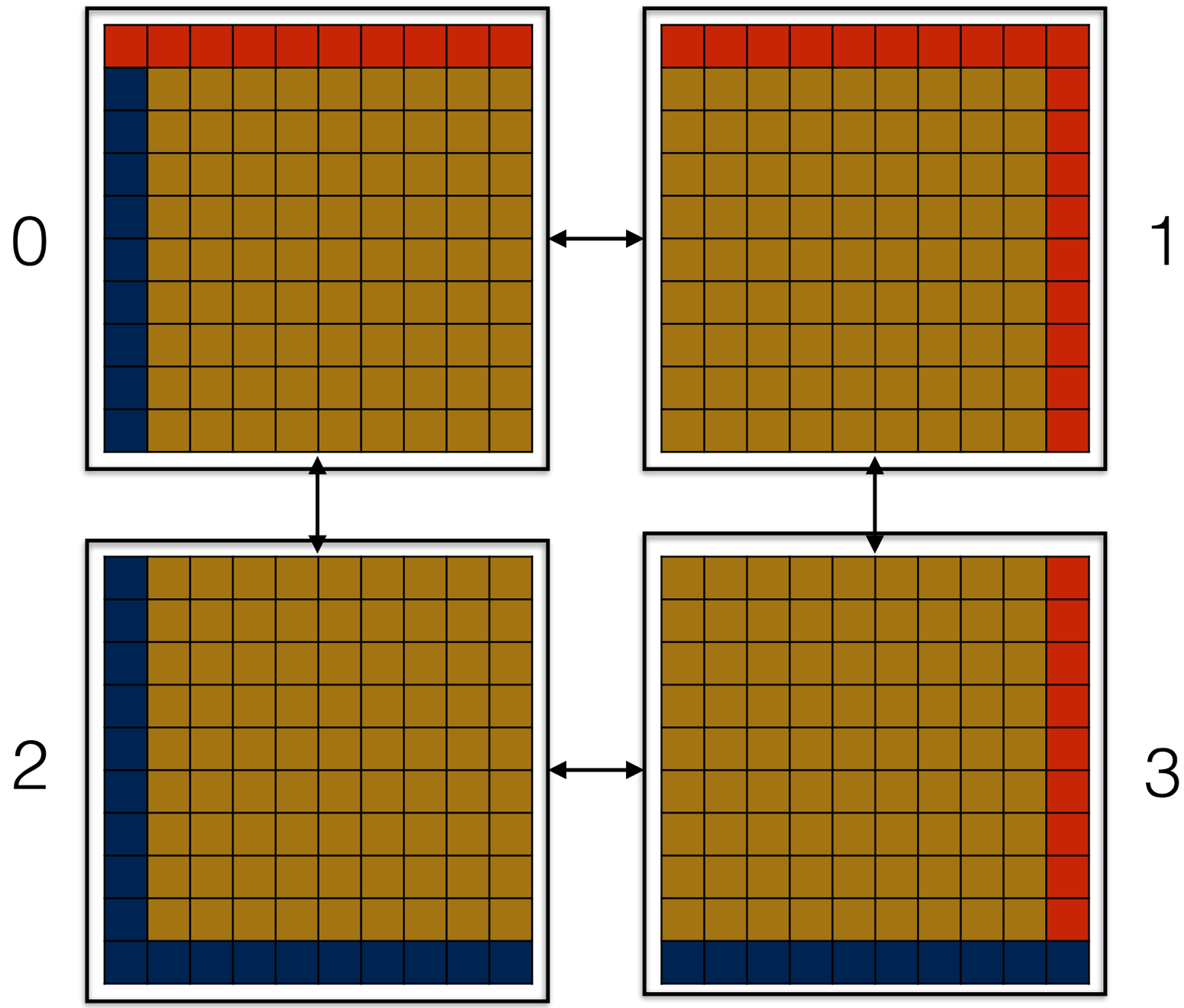
Topology

- You now have a logical breakdown of your processors into an L by M grid
- You need to break up your array onto this grid in a way that is consistent with that breakdown

Topology

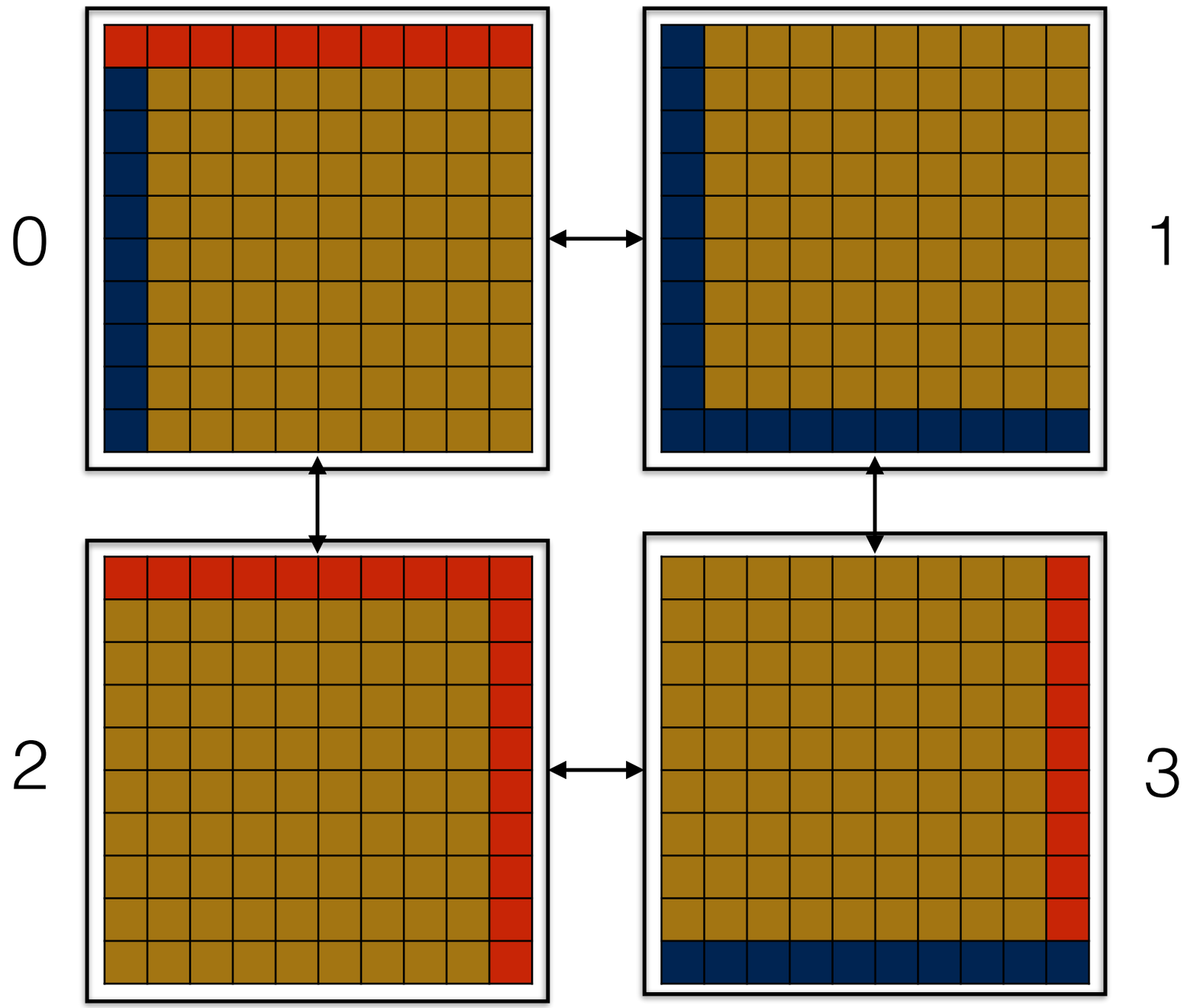


Topology



Works

Topology



Gives nonsense

Topology

- There are also questions about how the actual computer works
- In some computers certain processors are “nearer” (in some way) to your current processor than other processors in the communication network
- Since you are trying to only communicate with your neighbours it makes sense to chose those “near” processors to be your neighbours
- MPI provides functions to help with this

Topology

- They are two families
  - MPI_Cart_* - Several routines for dealing with Cartesian topology (what I just described)
  - MPI_Graph_* - Several routines for dealing with arbitrary topologies described as graphs
- They are quite useful but add 3 or 4 new commands and quite a bit of complexity so we're going to ignore them
- They are covered in our intermediate MPI course

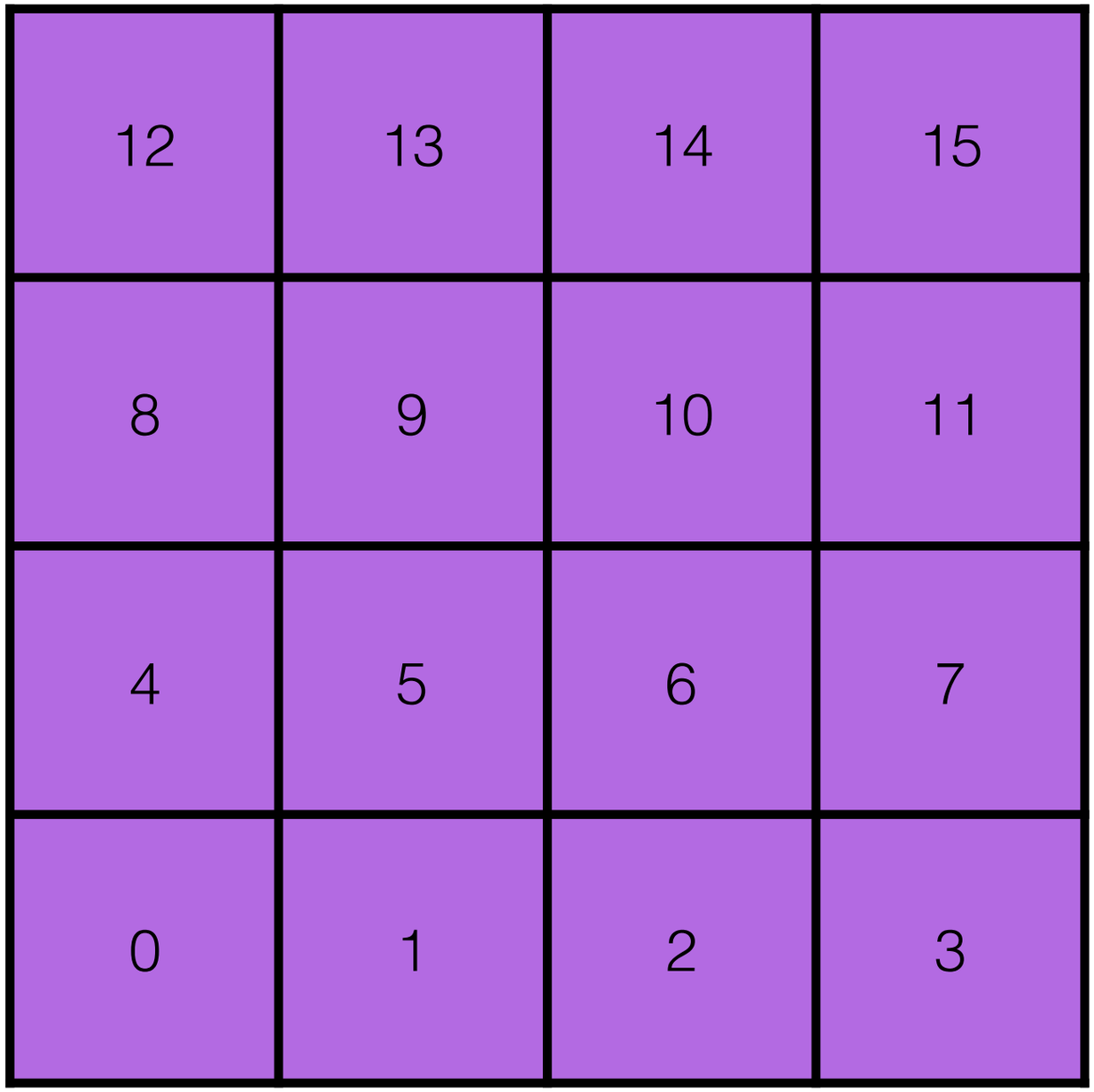
Topology

- We are just going to pretend that ranks increase by going through our first index from MPI_Dims_create first
- And then increment our second index when we run out of x processors
  - coords_y = FLOOR(rank / nprocx)
  - coords_x = rank - coords_y * nprocx

Topology

- You then need to know your neighbouring processors' ranks so that you can communicate with them
- Not too hard

PROCESSORS!!



- Processor to left have rank 1 lower
- Processor to right has rank 1 higher
- Processor upwards has rank nproc_x higher
- Processor downwards has rank nproc_x lower

Topology

- Then have to deal with the edges of the 2D processor decomposition
- If your problem has periodic boundaries then you can do this at this stage
  - Periodic boundaries simply become additional processor boundaries and are “free”
- Otherwise, there is a special rank called MPI_PROC_NULL

MPI_PROC_NULL

- If passed to any MPI send or receive command turns that command into a null operation
  - Does nothing
- If passed to one half of a Sendrecv command (either rank_recv, or rank_send) then that half becomes a null operation while the second half operates as normal
- Not valid for root process rank in Gather/Reduce etc. will cause **run time** failure

Local part of processor

- The last thing is to calculate which bit of the "virtual" global array this processor has

- That's pretty easy

- ```
  nx_local = nx / nproc_x
  x_cell_min_local = nx_local * coords_x
  x_cell_max_local = nx_local *
  (coords_x + 1) - 1
  ```

- Add one to both for Fortran style base 1 arrays

- Same in Y direction
  ```

Local part of processor

- It's worth noting that you only need to know about the position of the local array in the global array so you can
  - Set initial conditions
  - Bring it all back together for output
- The actual computation is still strictly local and makes no use of that at all
- The communication is just nearest neighbour processors so as long as you know which processors are next to you it will work

To date

- Domain is decomposed onto processors
- You know how the “virtual global” array maps onto your original array
- You know which processors are neighbours to your current processor
- Now need to actually do the sends and receives
- Not too hard

Points to note

- My local array runs (0:nx_local+1, 0:ny_local+1)
- The 0 and nX_local+1 strips are ghost cells, either filled from actual boundary conditions or from MPI calls
- I have stored the rank of my neighbouring processors in variables called
  - x_min_rank, x_max_rank, y_min_rank, y_max_rank

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));

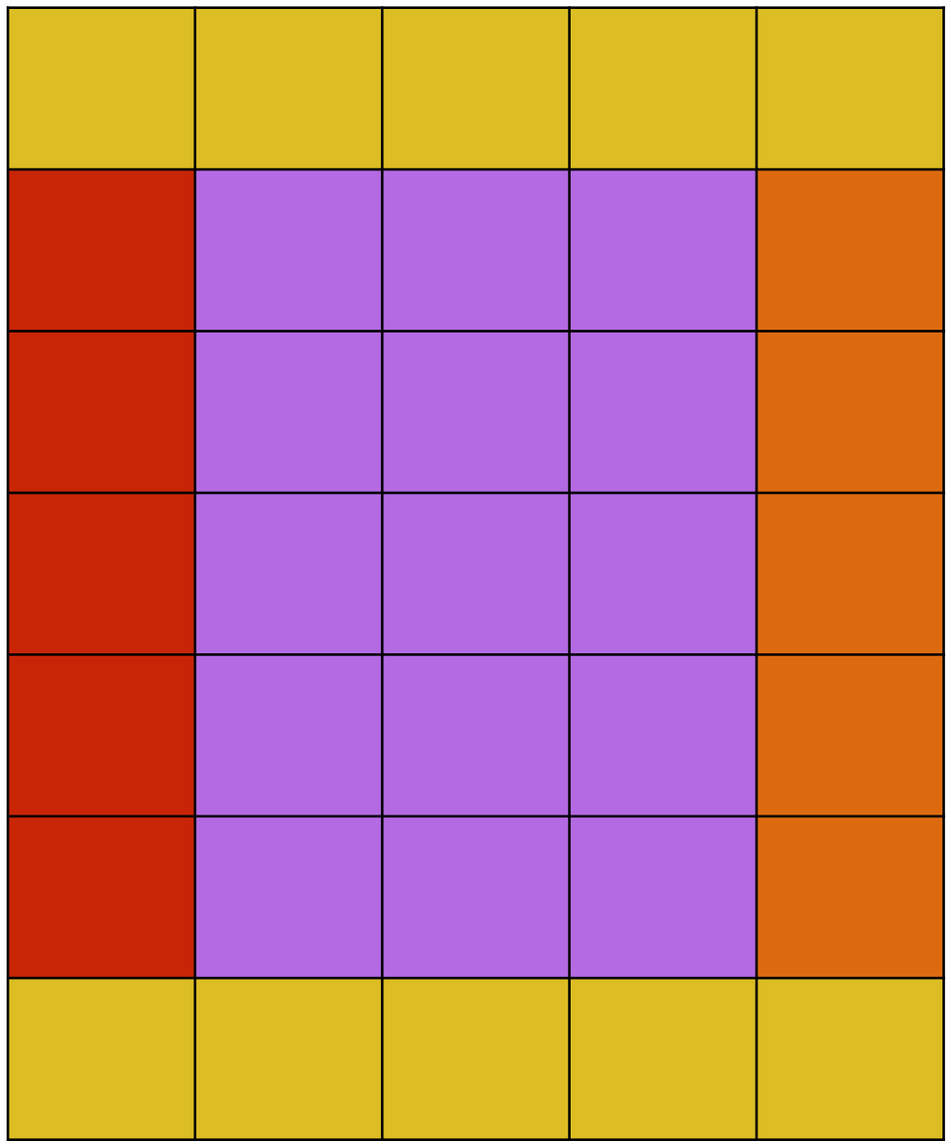
//Send left most strip of cells left and receive into right guard cells
for (index = 1; index<=ny_local; ++index){
  src[index-1] = *(access_grid(data, 1, index ));
  //Copy existing numbers into dest because MPI_Sendrecv is a no-op if
  //one of the other ranks is MPI_PROC_NULL
  dest[index-1] = *(access_grid(data,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

What those Sendrecvs do

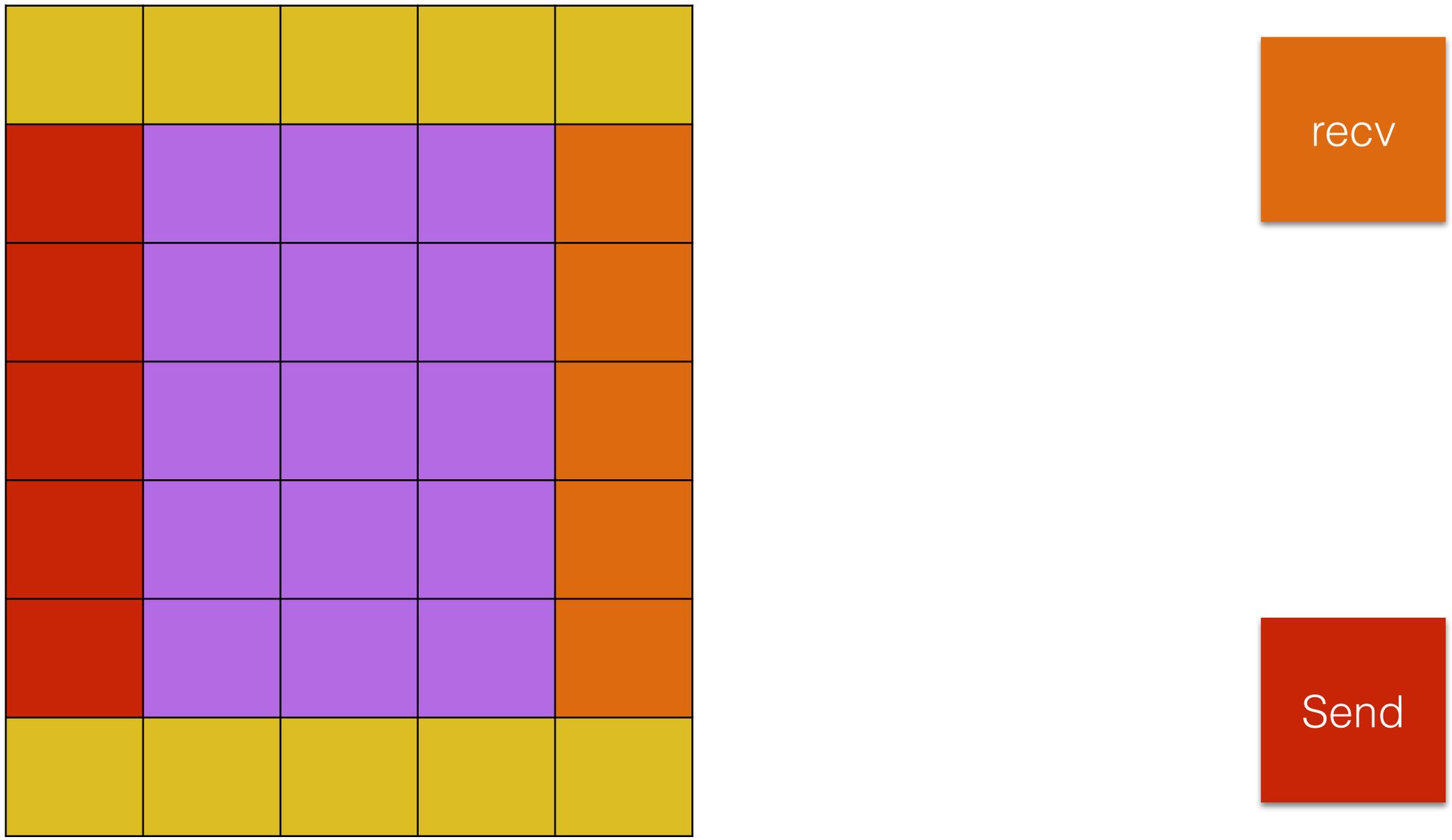


Send

All on one processor!

recv

What those Sendrecvs do



Now see the example

- The fully working example is supplied in the software pack that you've been provided with
- Have a look at it, play around with it and try to make a few changes
- Suggestions are
  - ....

Note to C programmers

- The C code includes a non standard array library that we wrote
- It allows you to assign values quickly and easily to sections of an array
  - Makes the code much less messy since there aren't loops everywhere
- Welcome to use it in your own code if you want (BSD license), but we haven't tested it much
- This is all built into Fortran making it much easier

Note to C programmers

- allocate_grid(ptr_to_grid_type, x_min_index, x_max_index, y_min_index, y_max_index) - Allocate a 2D grid with the specified index ranges (in this case 0:nx+1, 0:ny+1)

- assign_grid(ptr_to_grid_type, x_min_index, x_max_index, y_min_index, y_max_index, value) - Assign value to all cells within the specified index ranges

- access_grid(grid_type, ix, iy) - Returns a pointer to element [ix][iy]

- copy_grid(ptr_to_dest, ptr_to_src, x_min_index, x_max_index, y_min_index, y_max_index) - Copy values in specified range from src to dest