Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Message Passing Interface – Advanced Topics

Lecture 5

István Reguly
reguly.istvan@itk.ppke.hu

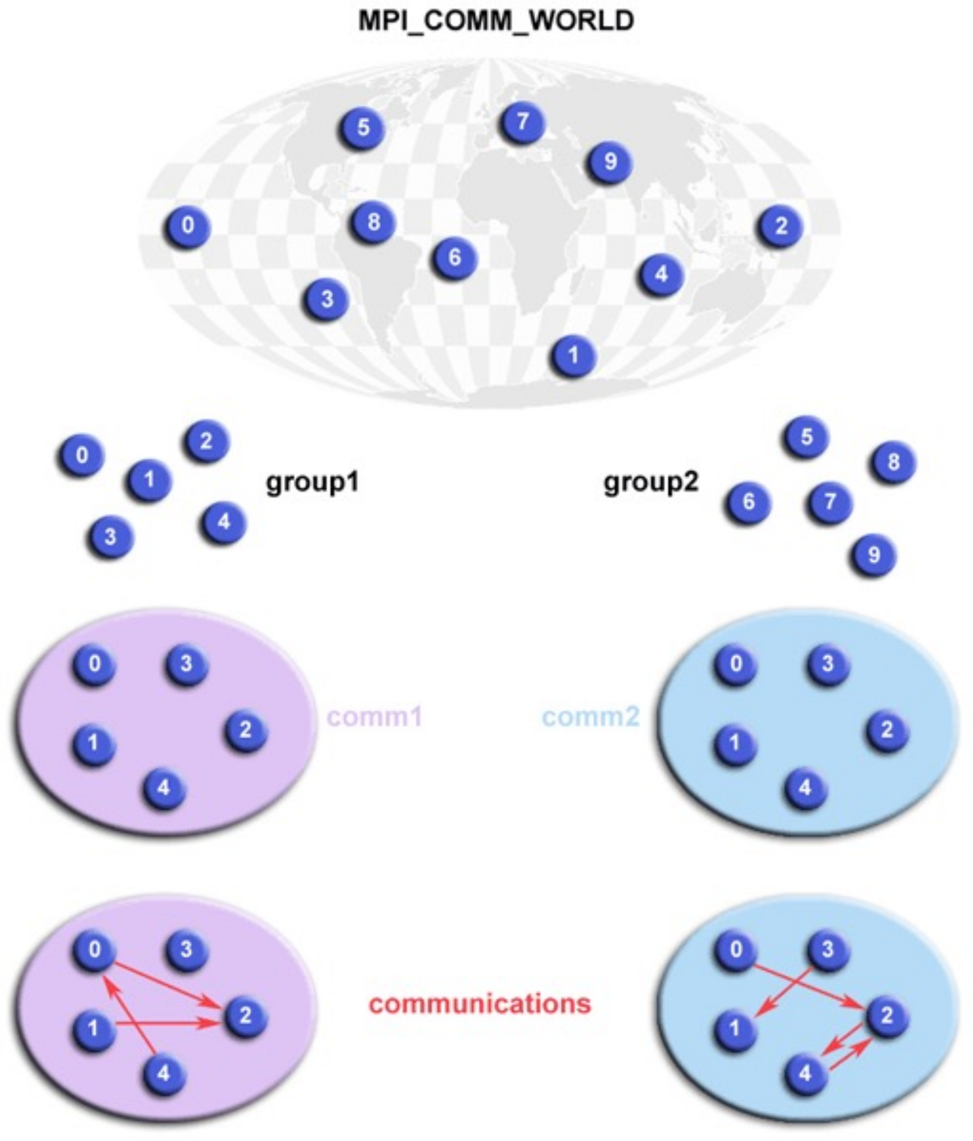
Groups and Communicators

- So far we had MPI_COMM_WORLD, which included all the ranks
- An MPI group is an ordered set of processes.
  - Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, (N is the size of the group)
  - Always associated with a communicator
- An MPI communicator encompasses a group of processes that may communicate with each other
  - All MPI messages use one
  - From the programmer’s perspective, no real difference

Purpose of MPI groups

- Allow you to organize tasks, based upon function, into task groups
- Enable collective communications operations over a subset of related tasks
- Provides basis for implementing user defined virtual topologies
- Safe communications

- Dynamic objects, can be created of freed at any time
- Processes may be in more than one group (they’ll have a unique ID for each)



1. Extract group handle from MPI_COMM_WORLD (MPI_Comm_group)
2. Create subset of global group with MPI_Group_incl
3. Create a new communicator for new group MPI_Comm_create
4. Determine new rank with MPI_Comm_rank
5. Do communications within the group
6. When done, deallocate MPI_Comm_free MPI_Group_free

Virtual topologies

- Describes a mapping/ordering of MPI processes into a geometric shape
- Two main kinds are Cartesian and Graph
- Virtual because there may be no relationship to the physical structure of the machine
  - Some implementations do try to optimize it, especially if communication to “far” nodes is more expensive
- Built upon MPI communicators and groups
- Useful in situations where the application-specific communication pattern matches the MPI topology structure

Cartesian topology

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

- MPI_Cart_create(incomm, numdims, dimsize, periodic, reorder, outcomm)
- MPI_Cart_coords(comm, serialrank, numdims, coords)
- MPI_Cart_shift(comm,dim,offset,&negidx,&posidx)

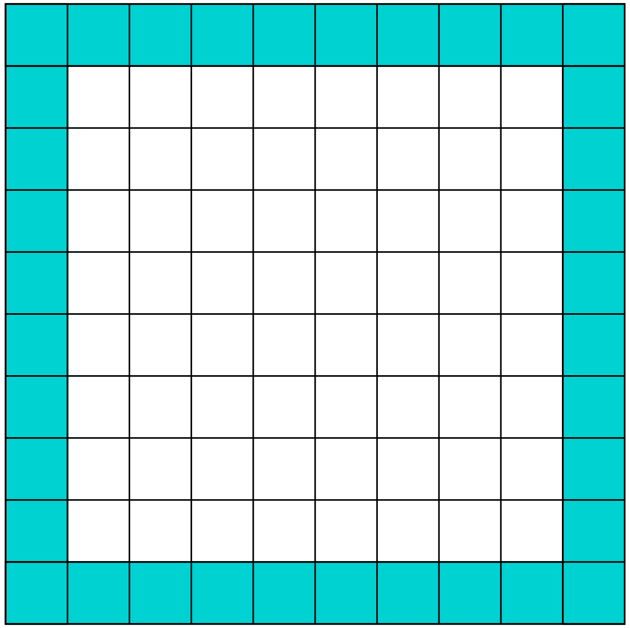
Cartesian topology

- MPI_Cart_create(incomm, numdims, dimsize, periodic, reorder, outcomm)
  - Creates a new communicator *outcomm*, representing a *numdims* dimensional space, with *dimsize[]* processes along each dimension. *periodic[]* specifies whether a rank at the end of a row is neighbours with the rank at the beginning
- MPI_Cart_coords(comm, serialrank, numdims, coords)
  - For a Cartesian communicator *comm*, puts the coordinates of process *rank* in *coords*
- MPI_Cart_shift(comm,dim,offset,&negidx,&posidx)
  - For a Cartesian communicator *comm*, returns the ranks of the neighbours of the current process in dimension *dim* that are *offset* away in the positive and the negative directions

Exercise

- Take a look at mpi_cart.cpp
- Modify it to work with any number of processes
  - Use MPI_Dims_create
  - Evaluate it for 1, 2, 4, 7, 8, 9
  - What sort of decomposition do you get?
- Modify it so each process send a message (its rank) to each of neighbours, receives a message (into inbuf) from each of its neighbours, and prints it

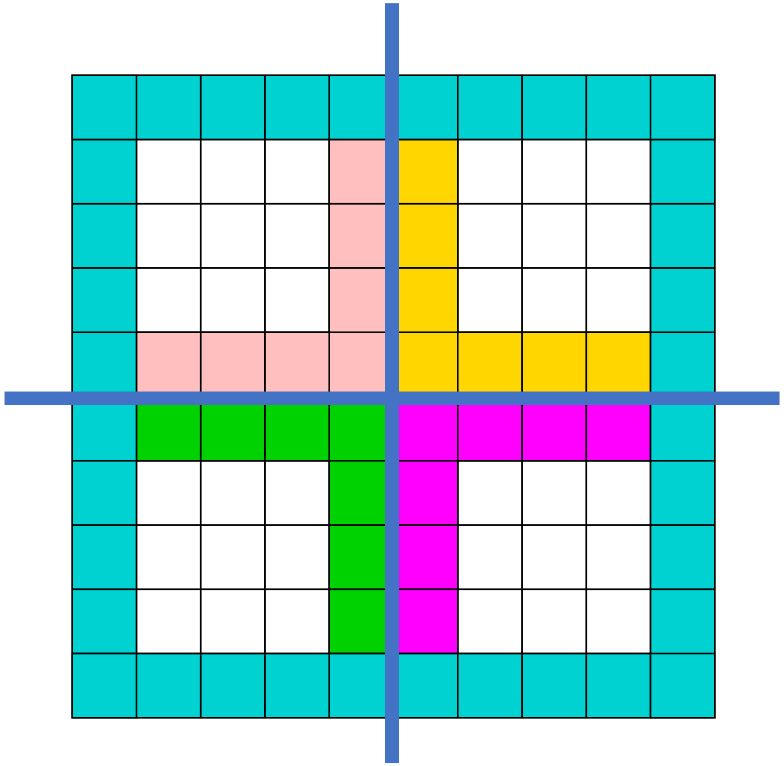
Cartesian grid & PDEs



- Assume grid elements are updated using a 5-point finite-difference stencil
- Grid boundary in cyan
  - Typically constant
- Grid points in interior change in every step

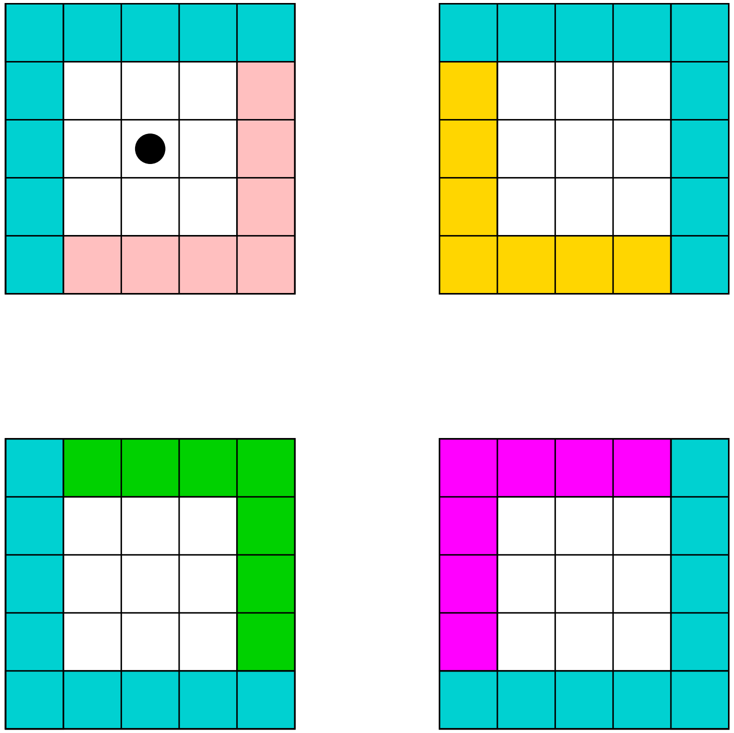
Partitioning the grid

- Suppose we want to partition into four subdomains
- Could be done only vertically or only horizontally
- We do both



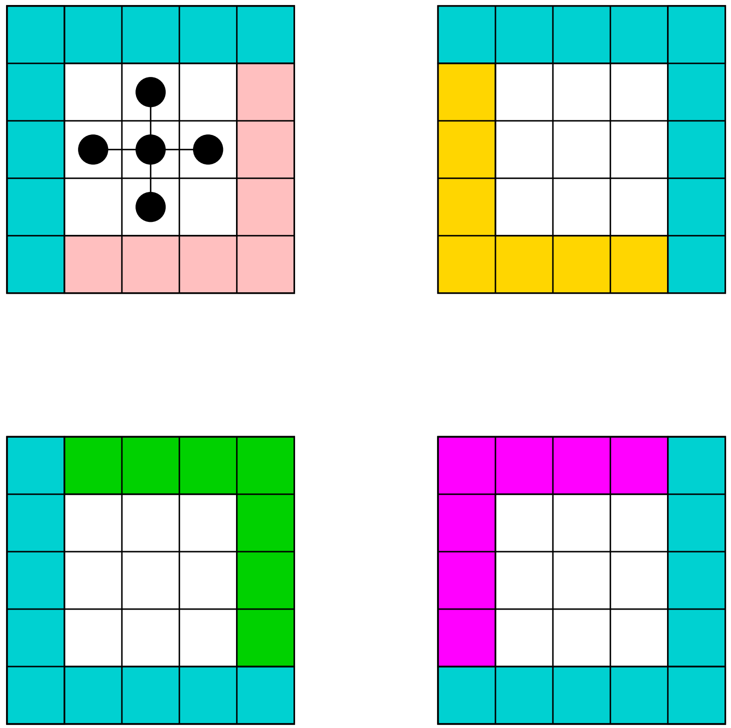
Computing

- To update the grid point marked, information is needed from the four neighbours
  - Homework’s lbm_d2q9.cpp “Gather neighbour values” loop



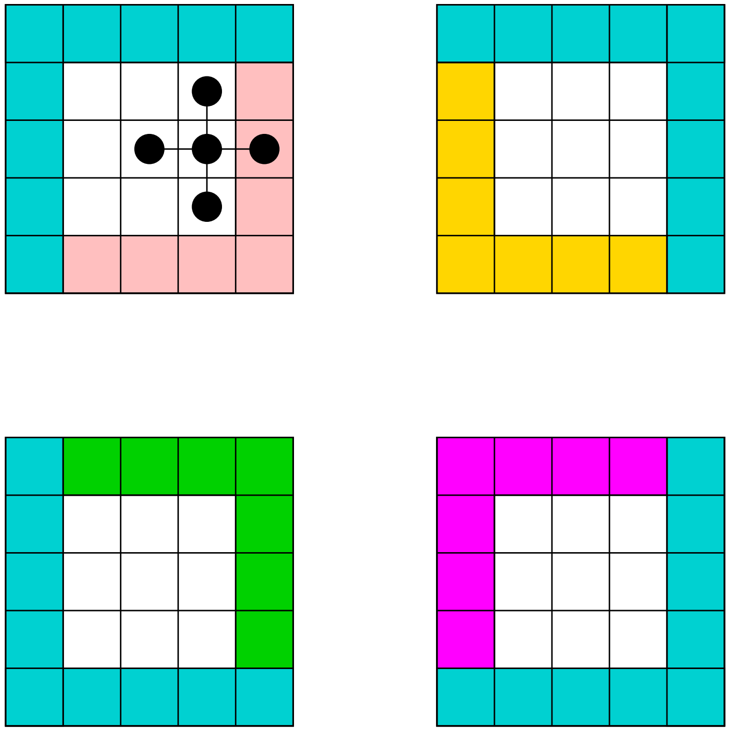
Computing

- To update the grid point marked, information is needed from the four neighbours
- The stencil shows the grid points needed for the update
- All accessed points are in the subdomain



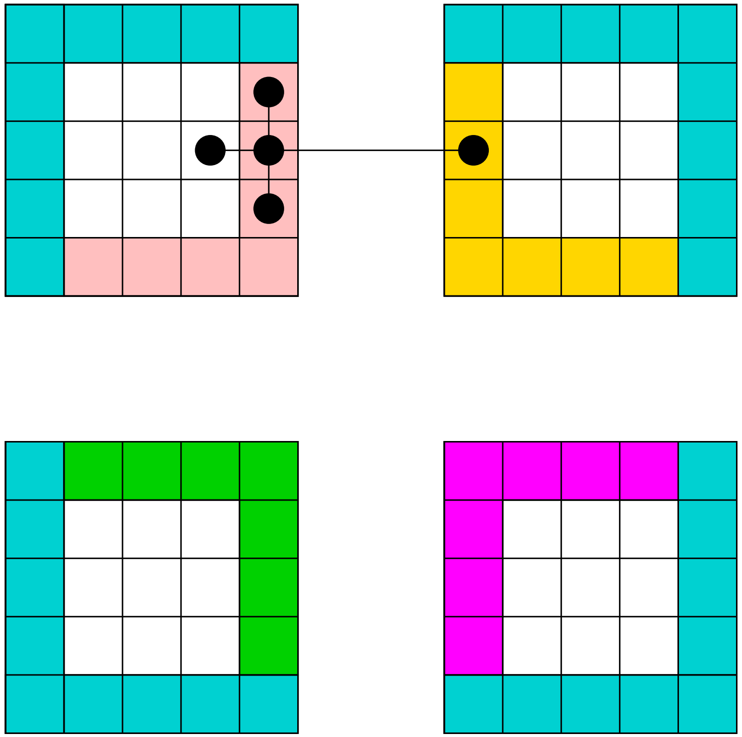
Computing

- To update an adjacent grid point, we are still okay



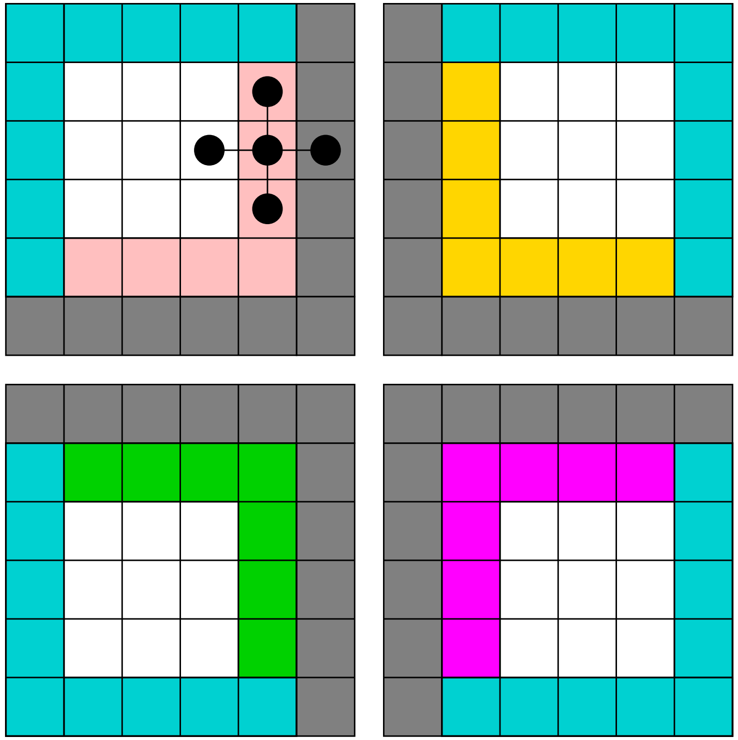
Computing

- To update an adjacent grid point, we are still okay
- But moving over one more, we now need information from the adjacent subdomain



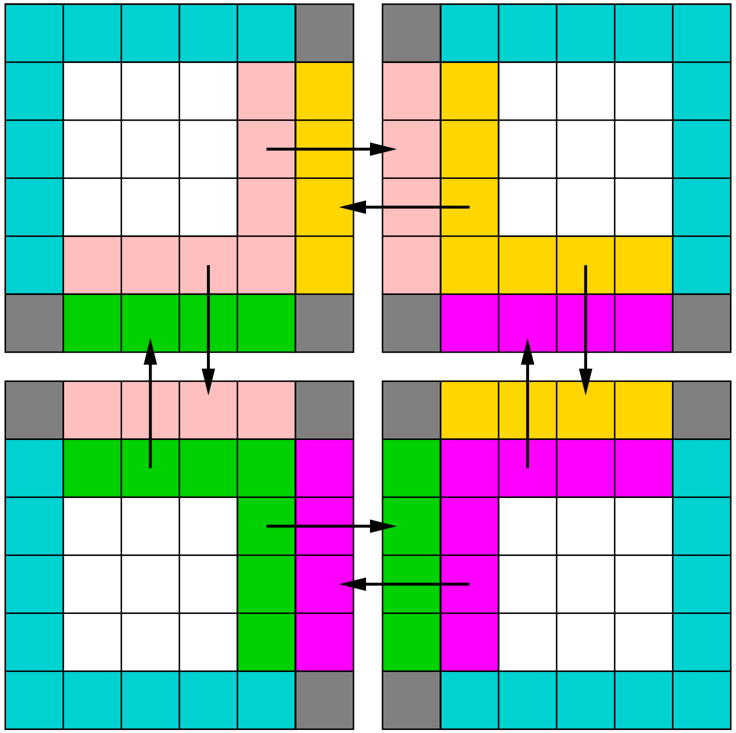
Communication

- Assuming message passing, we want to minimize communication
- We create additional grid locations to hold the copy of data from neighbours: *Ghost cells*
- Transfer can happen in blocks



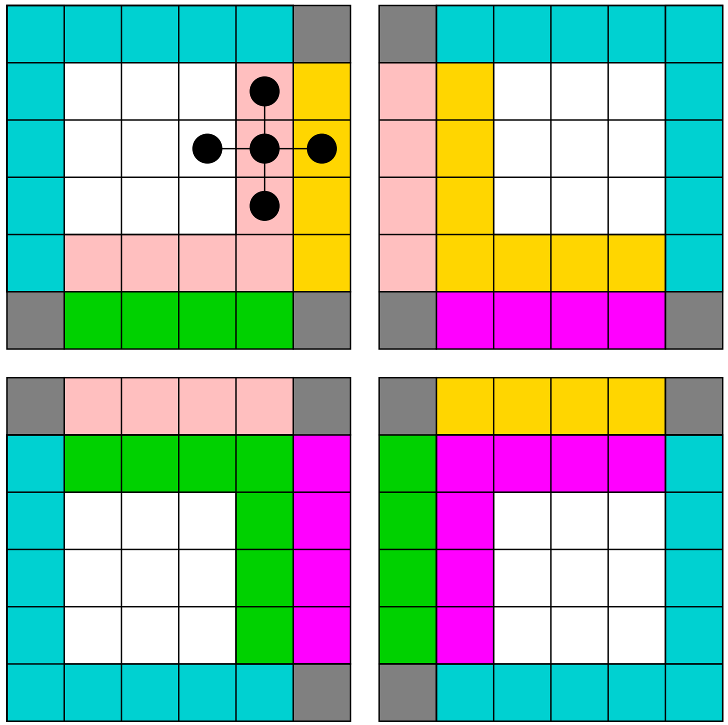
Communication

- Before each new set of updates...
- Interior boundary data is sent to processes working on adjacent subdomains



Computations

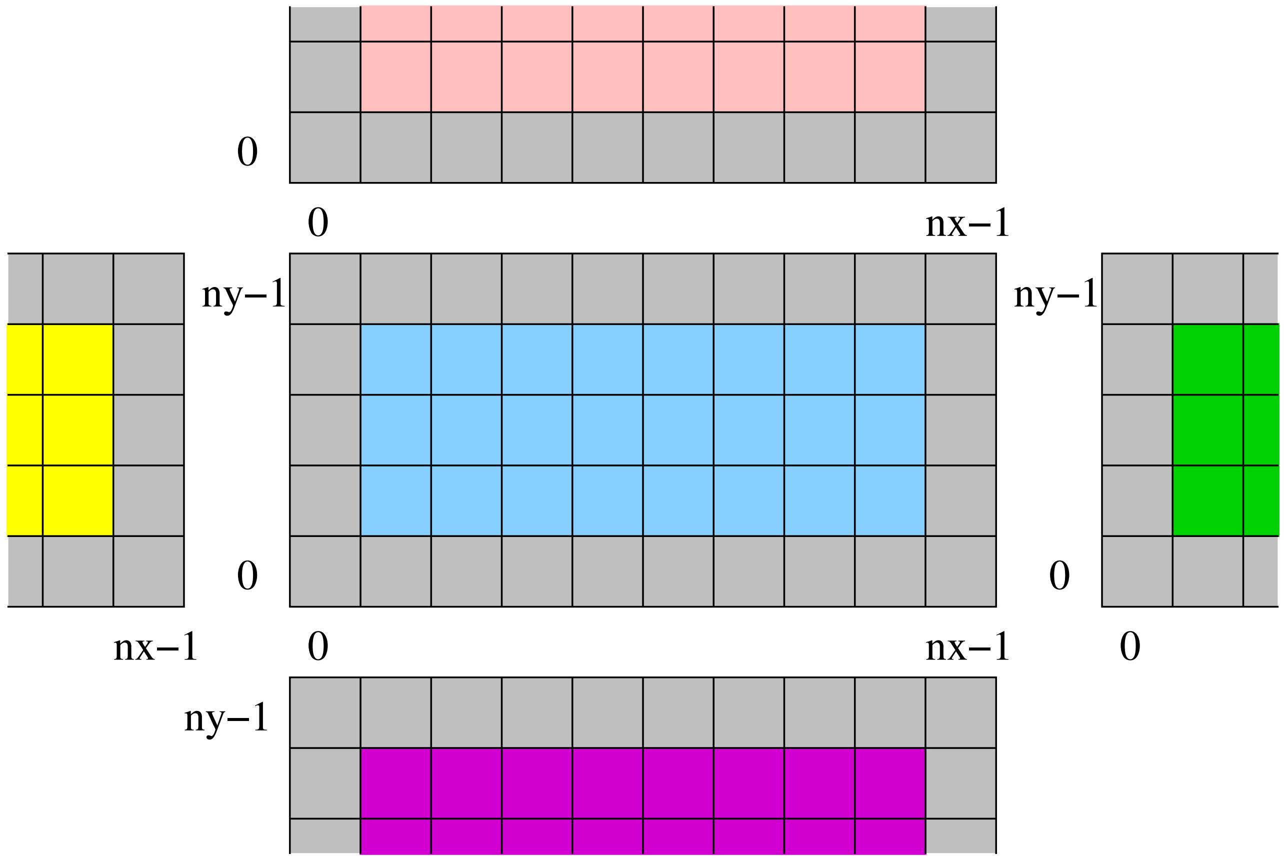
- Before each new set of updates...
- Interior boundary data is sent to processes working on adjacent subdomains
- Now accesses are once again limited to local subdomain



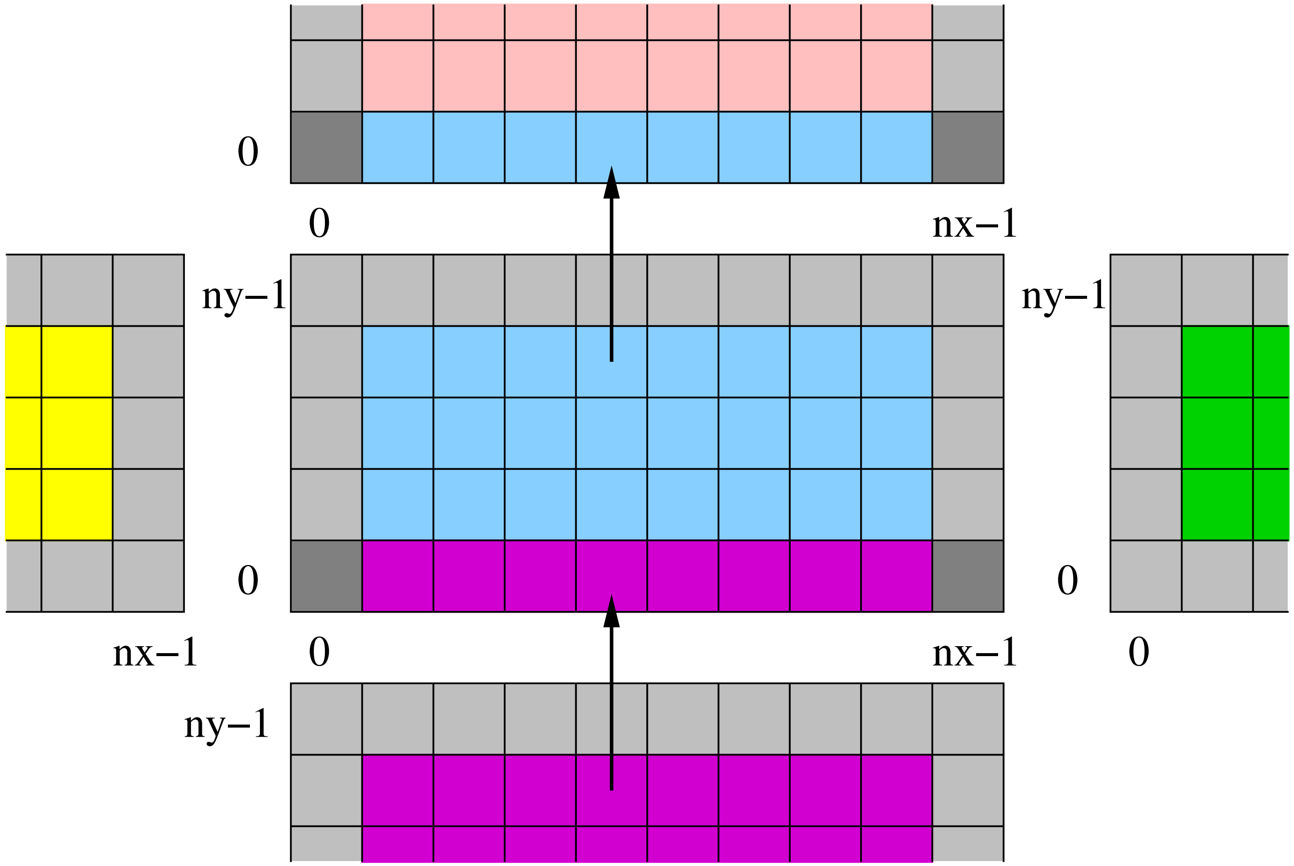
Typical Application

- An important class of applications that uses this data access patterns is the numerical solution of partial differential equations
- Programs are iterative and repeatedly update grid points with various ”sweeps”
- Between each sweep, interior boundary data must be communicated
  - By dimension: first in x, then in y, etc...
  - Either non-blocking, or circular pattern of MPI_Sendrecv

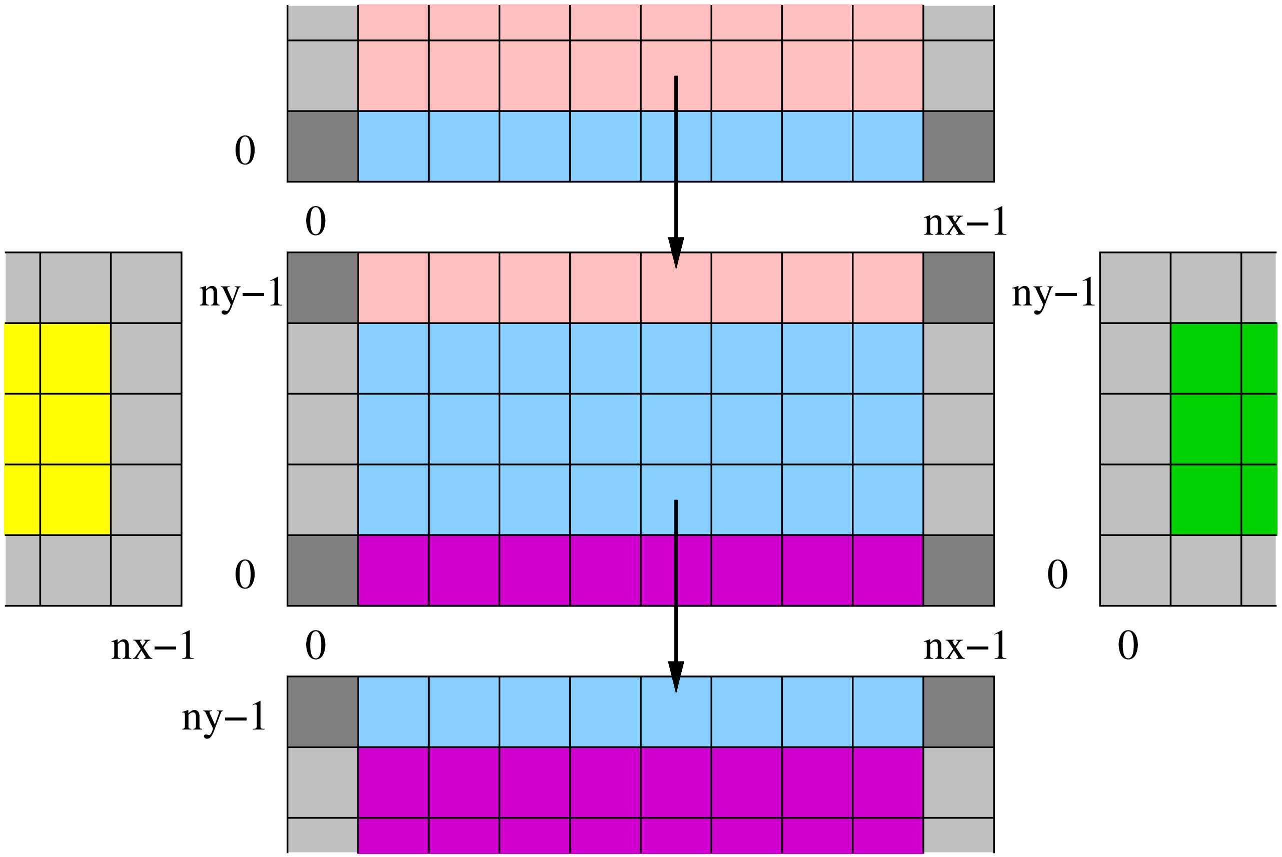
Ghost exchange



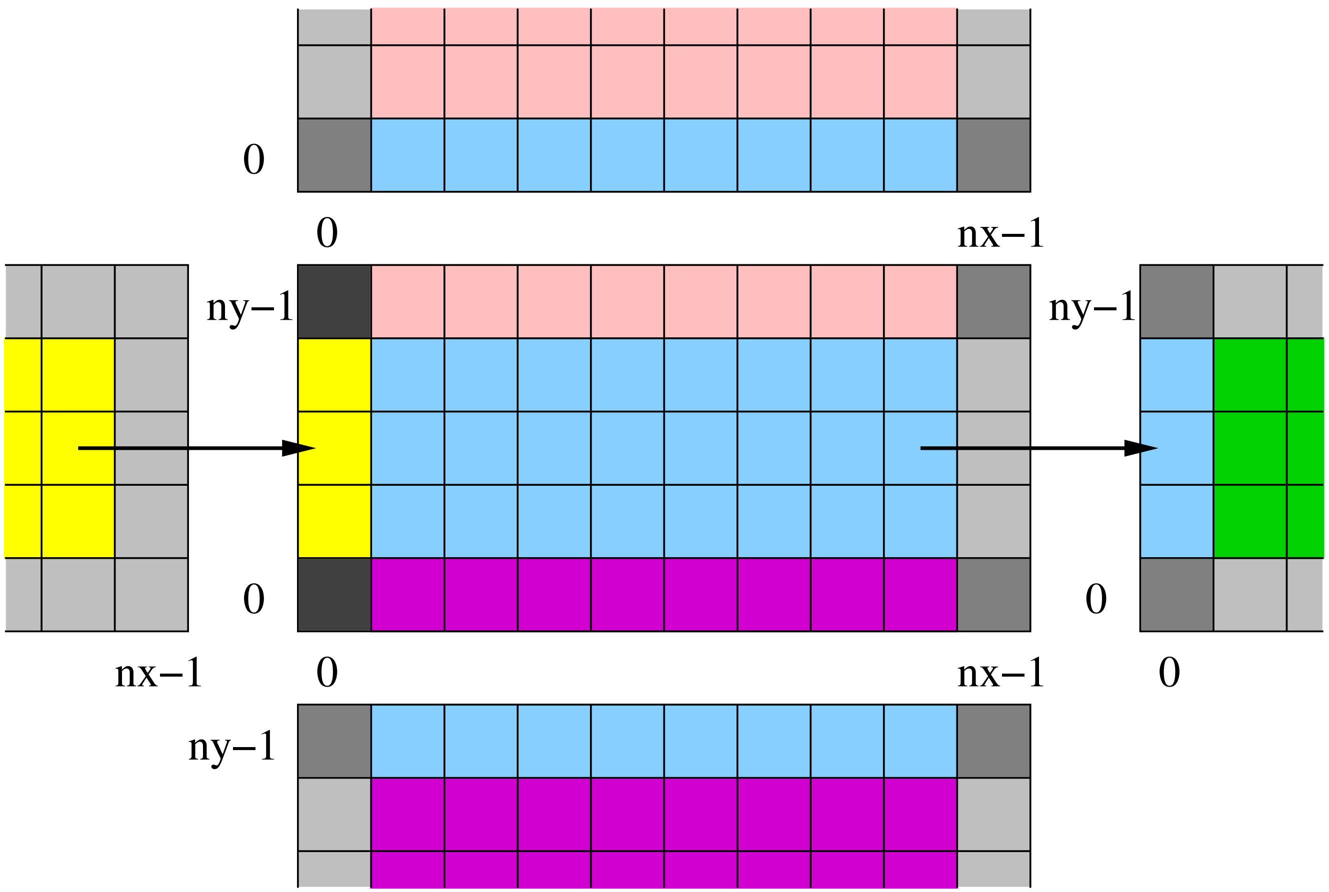
Ghost exchange



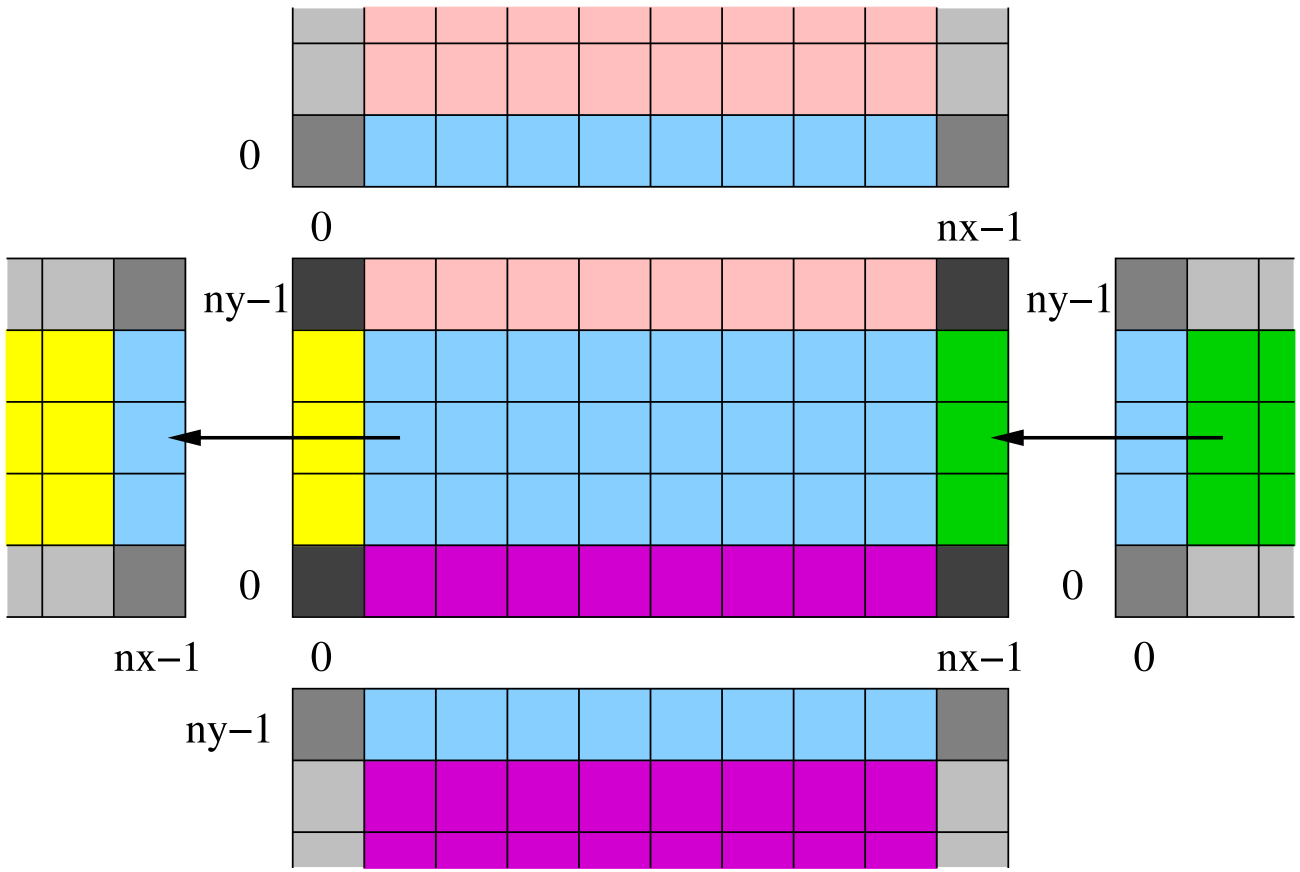
Ghost exchange



Ghost exchange



Ghost exchange

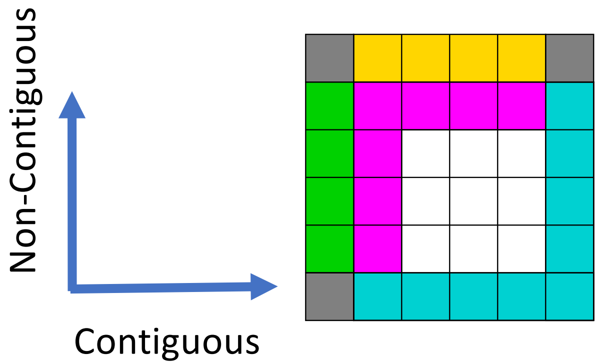


Ghost exchange

- Extends trivially to diagonal dependencies
- Extends trivially to periodic boundaries
- MPI_Sendrecv can be used with non-existent processes: MPI_PROC_NULL
  - If this is the destination, nothing is sent

Derived datatypes

- The MPI standard defines many built-in datatypes, mostly mirroring C/C++ and Fortran datatypes
- These are sufficient for sending single values or contiguous chunks
- Sometimes however we want to send non-contiguous data
  - E.g. 2D grid boundary



Derived Datatypes

- MPI provides ways of constructing datatypes to handle a wide variety of situations:
  - Contiguous
  - Vector, Hvector
  - Indexed, Hindexed
  - Indexed_block
  - Struct
- Routines marked with “H” differ from the others in that strides and block displacements are specified in bytes

Creating & using a new datatype

- Two steps are necessary to create and use a new datatype in MPI:
  - *Create* the type using one of MPI's type construction routines
  - *Commit* the type using MPI_Type_commit()
- Once a type has been committed, it may be used in send, receive and other buffer operations
- Can be released with MPI_Type_free()

Contiguous type

- The contiguous datatype allows for a single type to refer to multiple contiguous elements of an existing datatype

```
int MPI_Type_contiguous(
  int count,                // replication count
  MPI_Datatype oldtype,     // old datatype
  MPI_Datatype* newtype)    // new datatype
```

- Essentially an array of `count` elements having `oldtype`. The following two are equivalent:

```
MPI_Send(a, n, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);

MPI_Datatype rowtype;
MPI_Type_contiguous(n, MPI_DOUBLE, &rowtype);
MPI_Type_commit(&rowtype);
MPI_Send(a, 1, rowtype, dest, tag, MPI_COMM_WORLD);
```

Vector type

- Similar to the contiguous, but allows for a constant non-unit stride between elements

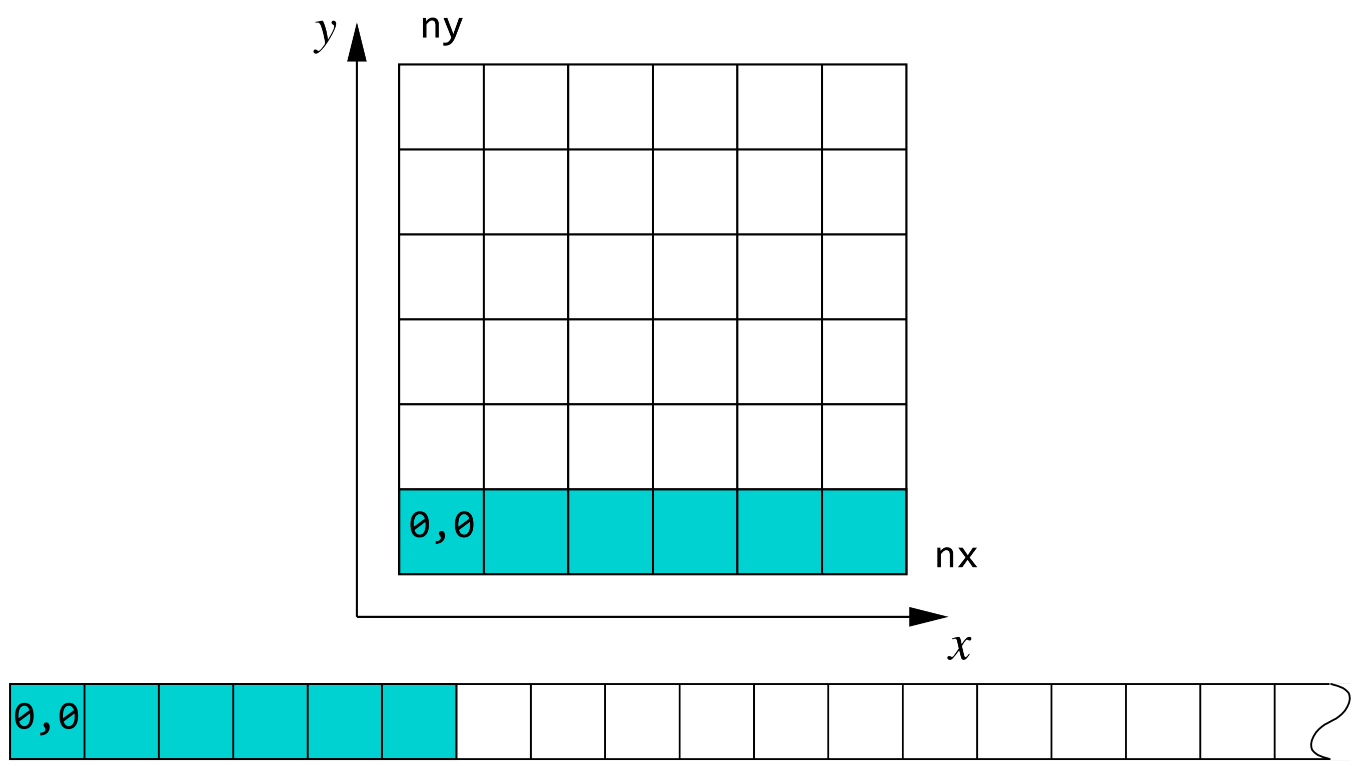
```
int MPI_Type_vector(
  int count,            // number of blocks
  int blocklength,      // number of elements in each block
  int stride,           // number of elements between each block
  MPI_Datatype oldtype,   // old datatype
  MPI_Datatype* newtype) // new datatype
```

- For an nx * ny Cartesian grid is stored in a row-contiguous way, the we can define the following two:

```
MPI_Datatype row, column;
MPI_Type_vector(nx, 1, 1, MPI_DOUBLE, &row);
MPI_Type_vector(ny, 1, nx, MPI_DOUBLE, &column);
MPI_Type_commit(&row);
MPI_Type_commit(&column);
```

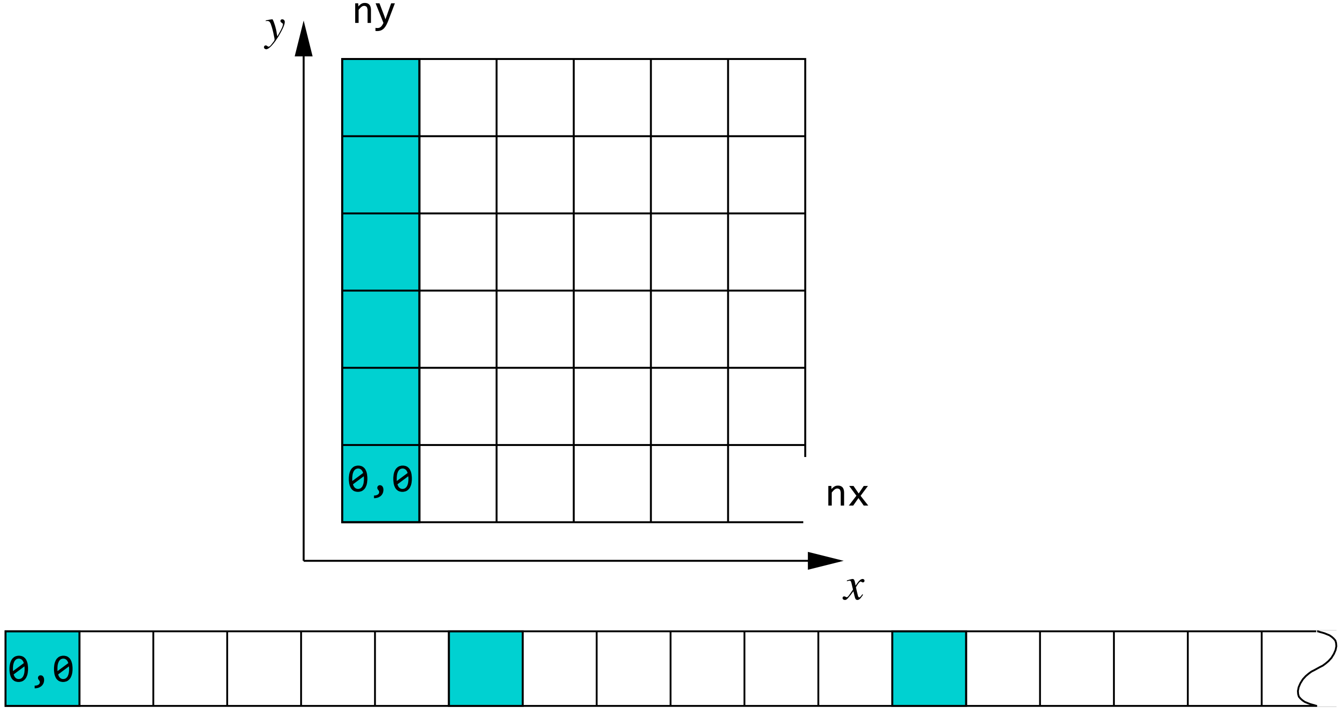
Vector type

```
MPI_Type_vector(nx, 1, 1, MPI_DOUBLE, &row);
```



Vector type

```
MPI_Type_vector(ny, 1, nx, MPI_DOUBLE, &column);
```



Indexed & Struct type

- Indexed datatype provides for varying strides between elements

```
int MPI_Type_indexed(
  int count,                // number of blocks
  int* blocklengths,        // number of elements per block
  int* displacements,       // displacement for each block
  MPI_Datatype oldtype,     // old datatype
  MPI_Datatype* newtype)    // new datatype
```

- Most general constructor allows for the creation of types representing general C/C++ structs/classes

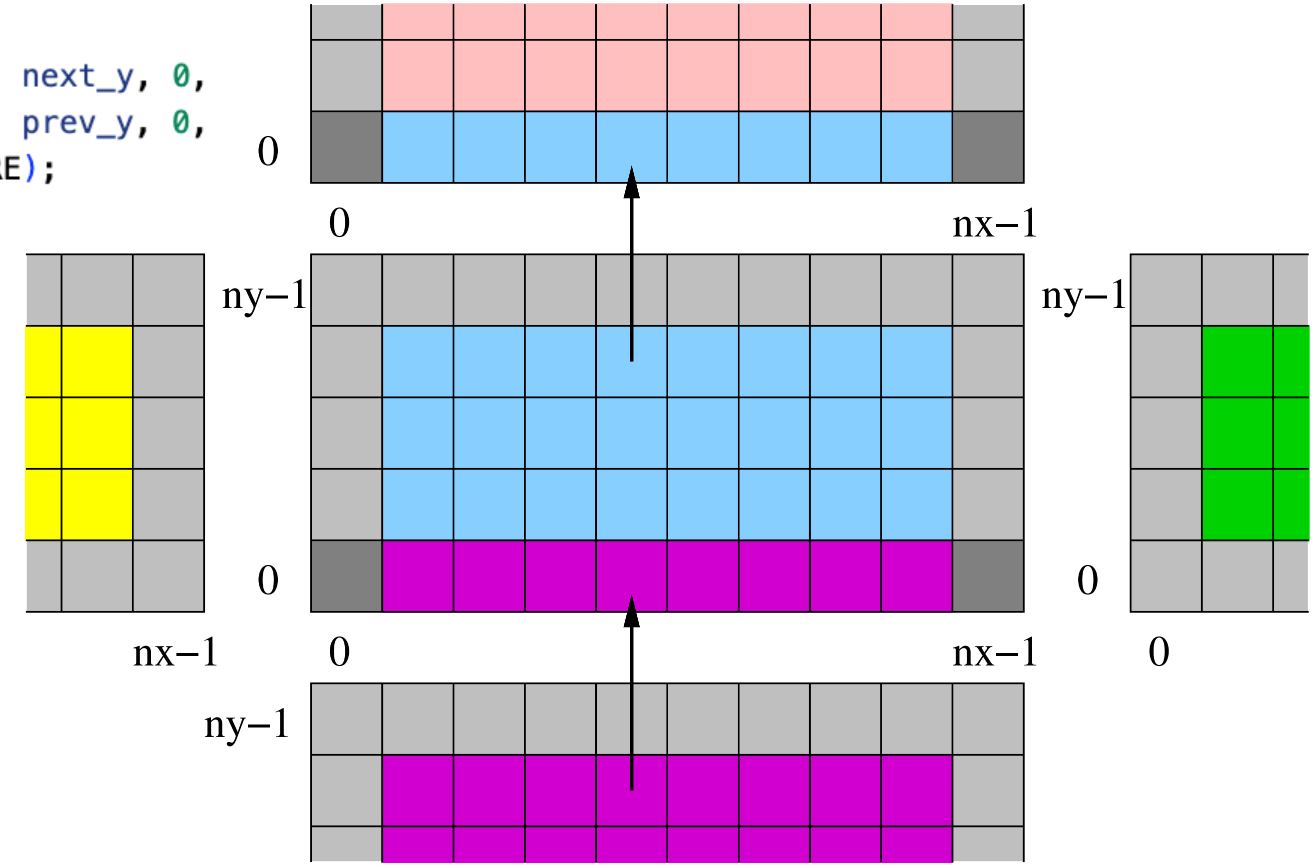
```
int MPI_Type_create_struct(
  int count,                 // number of blocks
  int* blocklengths,         // number of elements per block
  MPI_Aint* displacements,   // byte displacement of each block
  MPI_Datatype* datatypes,   // type of elements in each block
  MPI_Datatype* newtype)     // new datatype
```

Ghost exchange

```
MPI_Sendrecv(&arr[(ny-2)*(nx)],  1, row,    next_y, 0,
             &arr[0],            1, row,    prev_y, 0,
             MPI_COMM_CART, MPI_STATUS_IGNORE);
```

Send last-1 row up

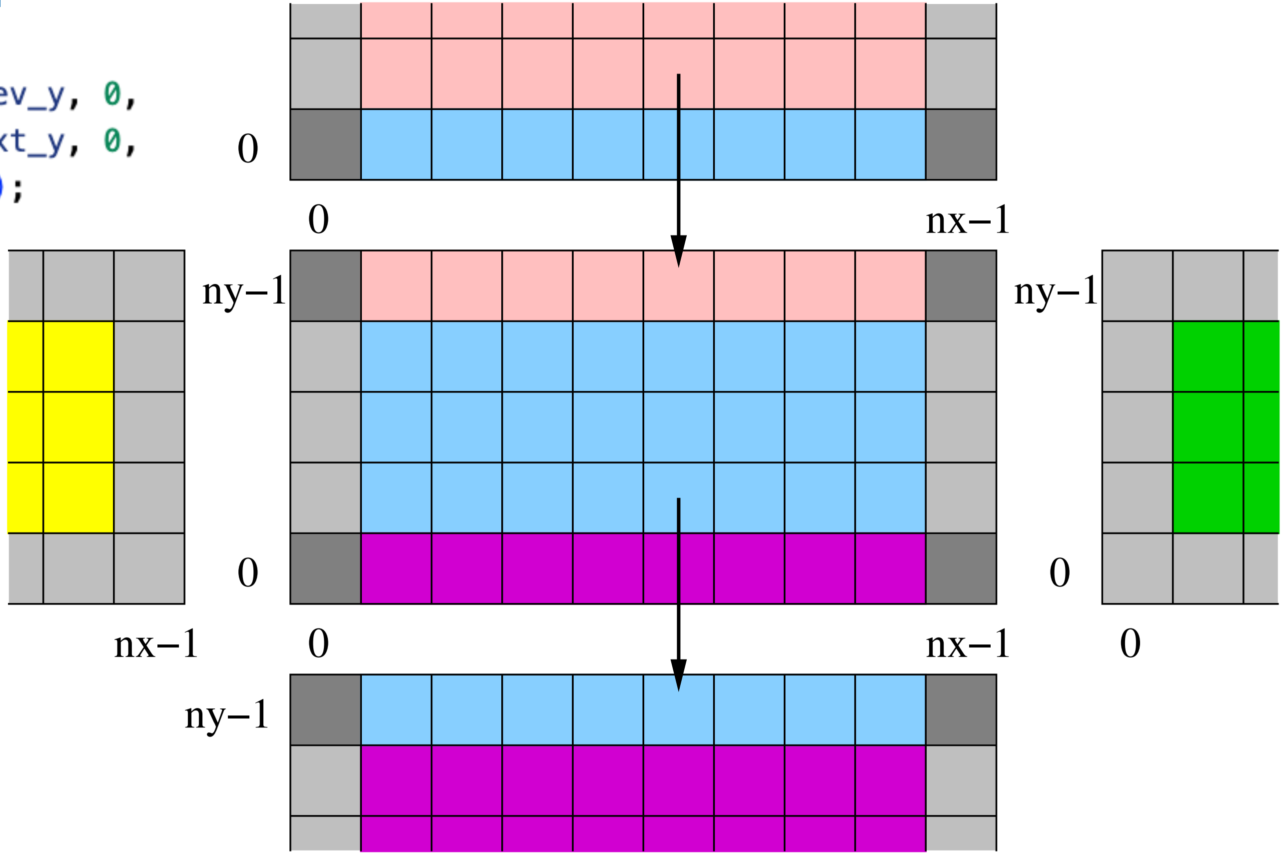
Receive into first row



Ghost exchange

```
MPI_Sendrecv(&arr[0+1*(nx)],    1, row,     prev_y, 0,
             &arr[(ny-1)*(nx)], 1, row,     next_y, 0,
             MPI_COMM_CART, MPI_STATUS_IGNORE);
```

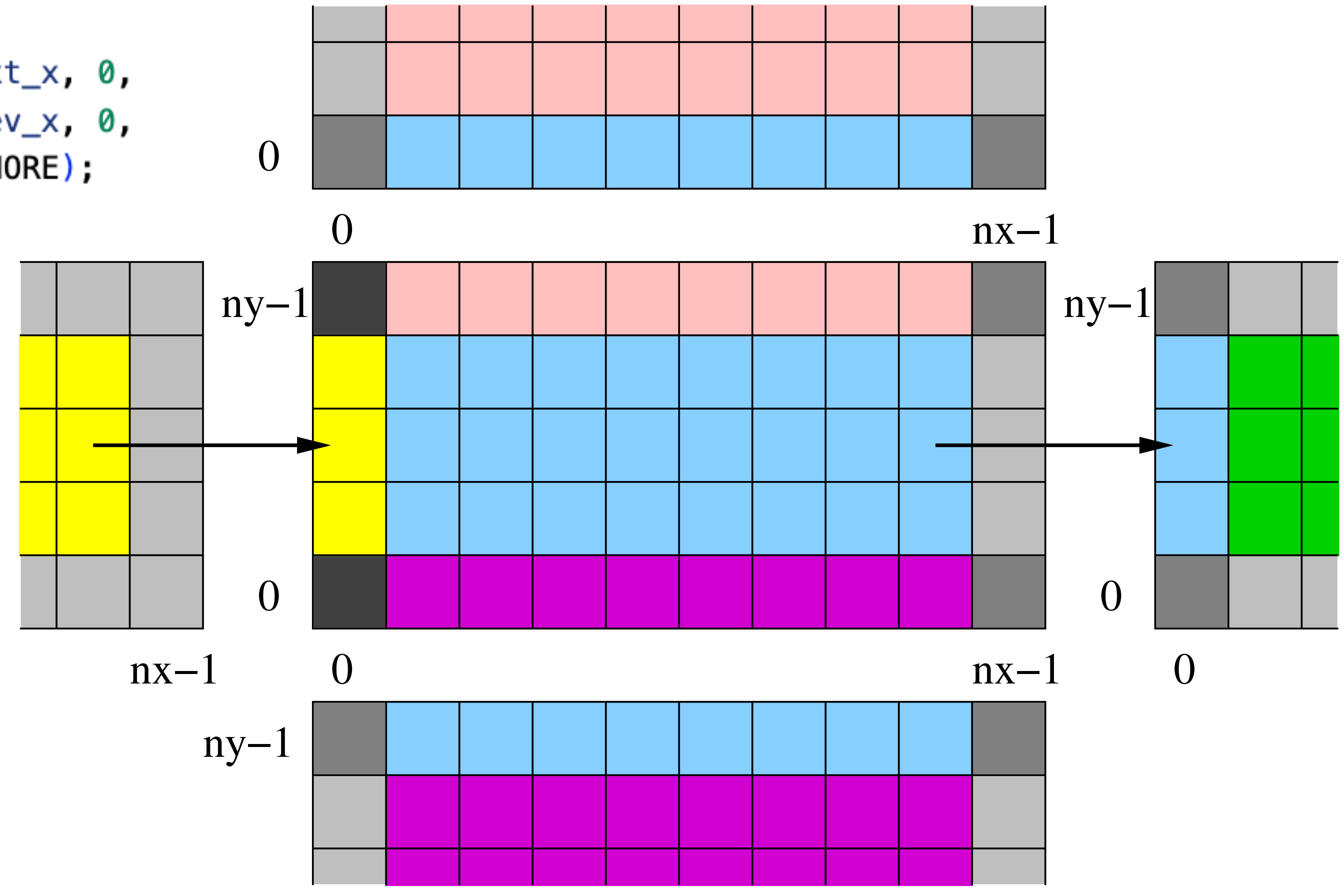
Send first row down
Receive into last row



Ghost exchange

```
MPI_Sendrecv(&arr[nx-2],    1, column, next_x, 0,
             &arr[0],       1, column, prev_x, 0,
             MPI_COMM_CART, MPI_STATUS_IGNORE);
```

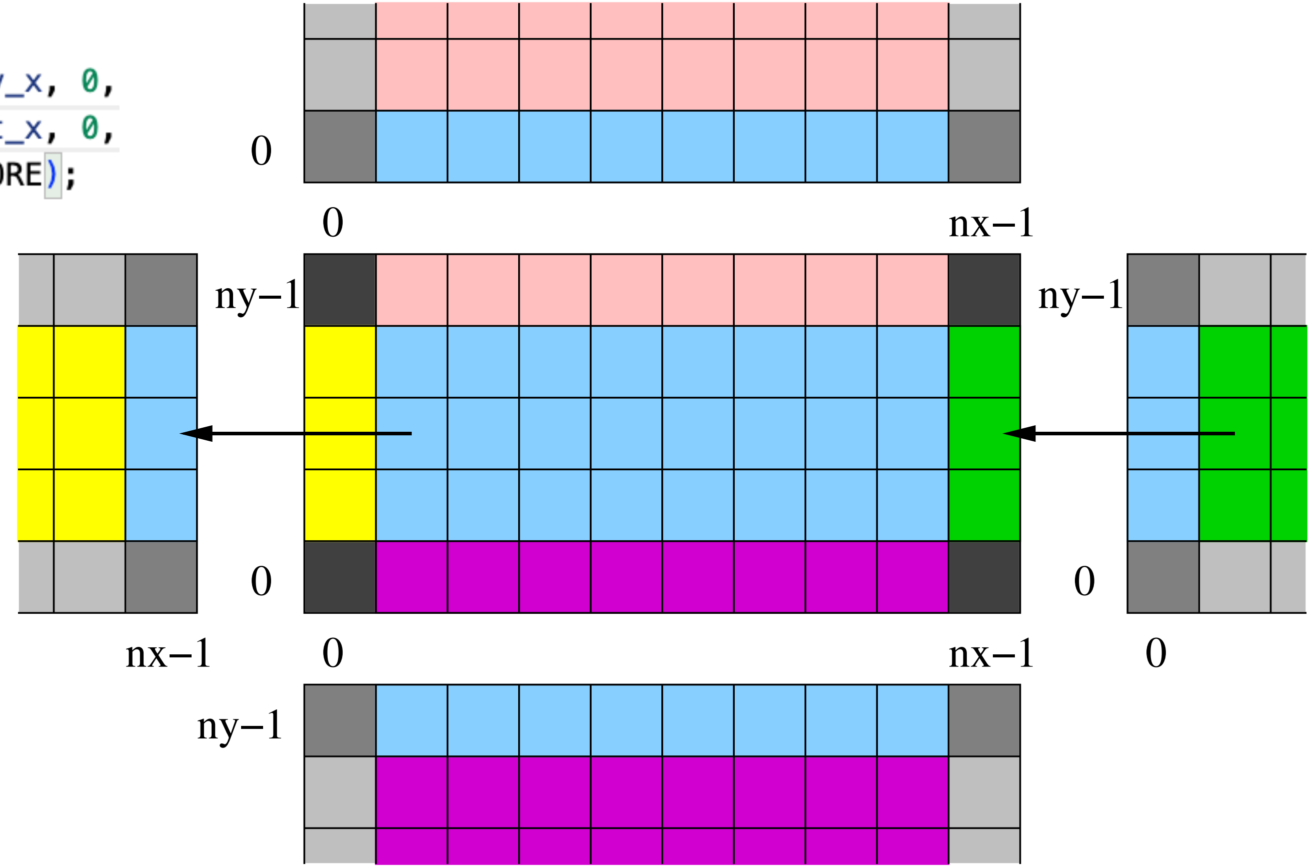
Send last-1 column right
Receive into first column



Ghost exchange

```
MPI_Sendrecv(&arr[1],      1, column, prev_x, 0,
             &arr[nx-1],   1, column, next_x, 0,
             MPI_COMM_CART, MPI_STATUS_IGNORE);
```

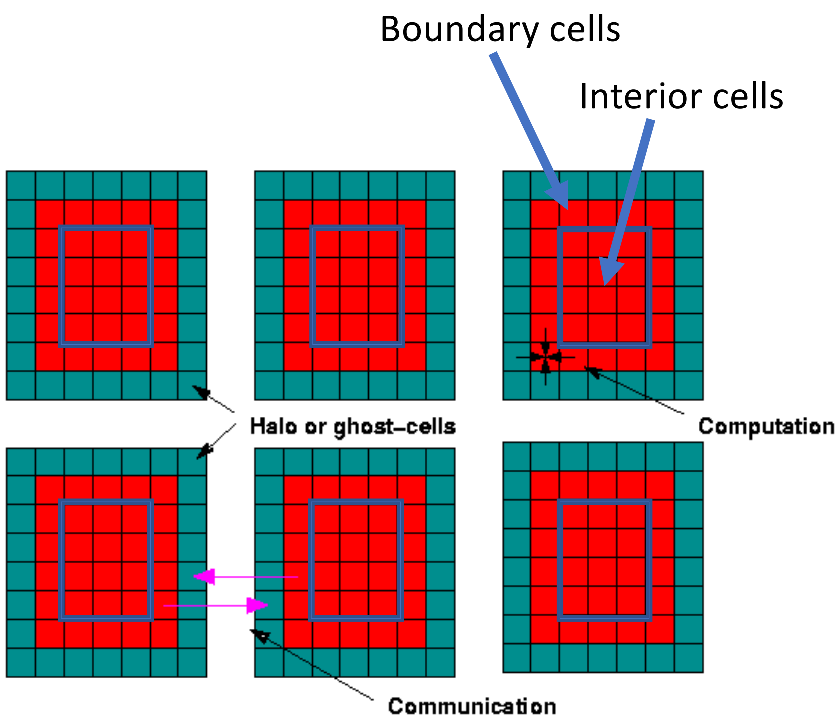
Send second column left
Receive into last column



Non-blocking operations

- Some interior cells do not need the presence of updated ghost cells – let's use them to hide the latency of communications

```
For niters:
  start exchange ghost cells
  for each interior cell
    compute new values
      ...
  wait for exchange
  for each boundary cell
    compute new values
      ...
```



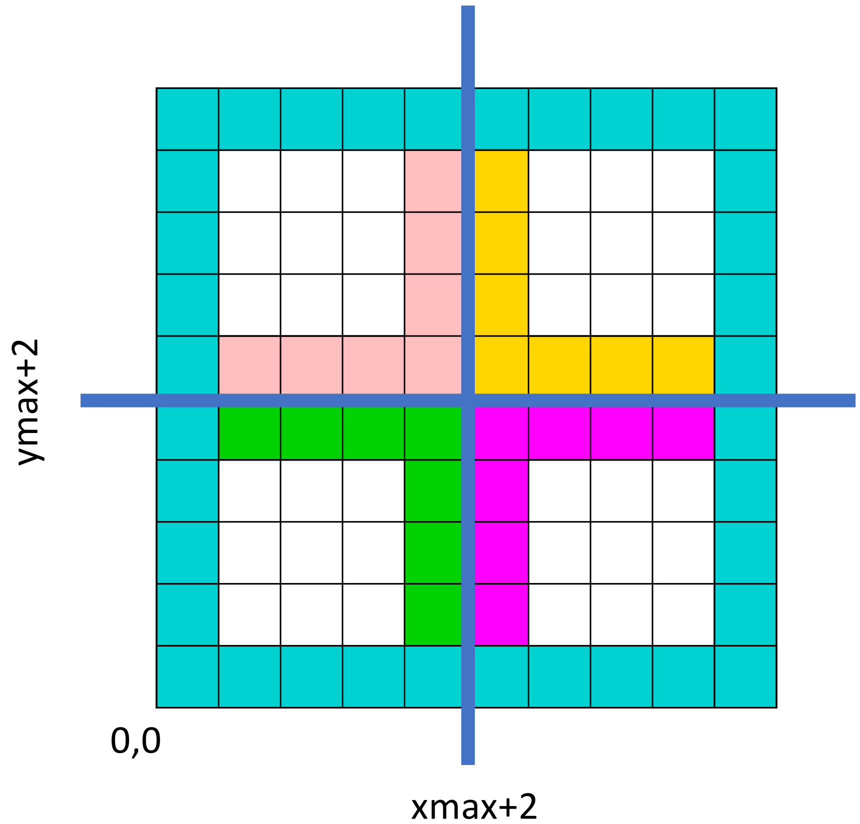
Outlook – MPI 2 and 3

- MPI 2 standard
  - Dynamic processes – can remove and create new processes at runtime
  - One-sided communications – one-directional comms
  - More collectives
  - Parallel I/O
- MPI 3 standard
  - Non-blocking collective operations
  - More one-sided communications
  - Neighbourhood collectives (for Cartesian and graph topologies)
  - MPIT Tool Interface for profiling

Exercise

- Laplace 2D convert the laplace.cpp example to use MPI, with any arbitrary number of processes
  - Use the mpi_laplace.h and mpi_laplace.cpp files to perform MPI initialization and communications

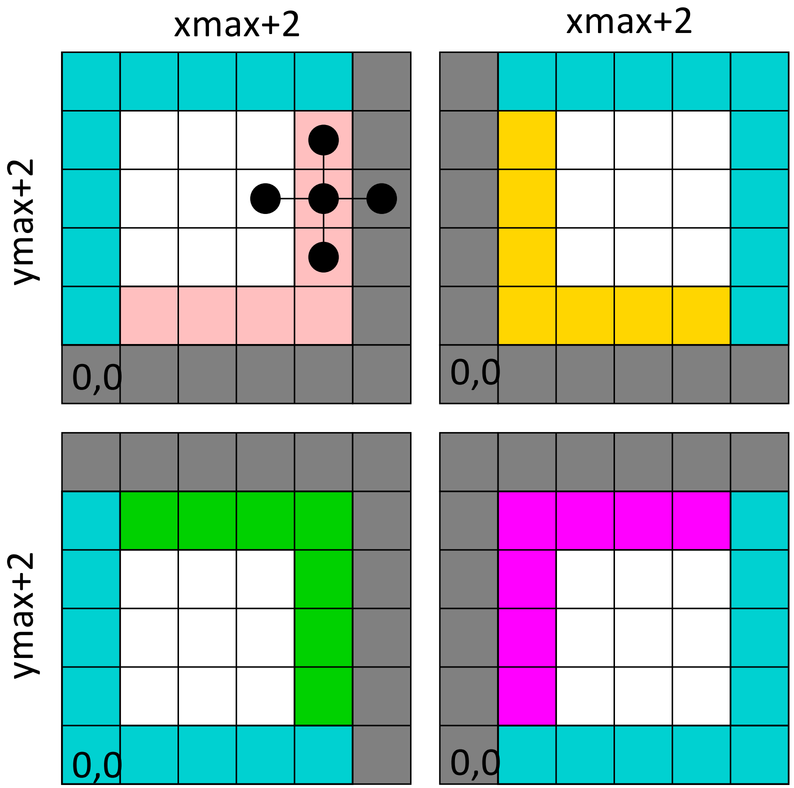
Exercise - solution



Exercise - solution

gbl_x_begin=0
gbl_y_begin=0

gbl_x_begin=xmax_full/2
gbl_y_begin=0



xmax_full = xmax
xmax = xmax/2
ymax_full = ymax
ymax = ymax/2

gbl_x_begin=xmax_full/2
gbl_y_begin=ymax_full/2

Homework – due apr 14 midnight

- Modify your homework to support MPI
  - Use lbm_d2q9.cpp from previous homework, commit to this lecture's repo
- Re-use our mpi_laplace.cpp and mpi_laplace.h
- **Push results to assignment-5 repo!**
- Things to keep in mind
  - **Only need to work for square number of processes**
  - See which loops require ghost cells, add the call to exchange right before
  - We use periodic boundaries here (rightmost process' right neighbour is the leftmost one)
  - Reduction…