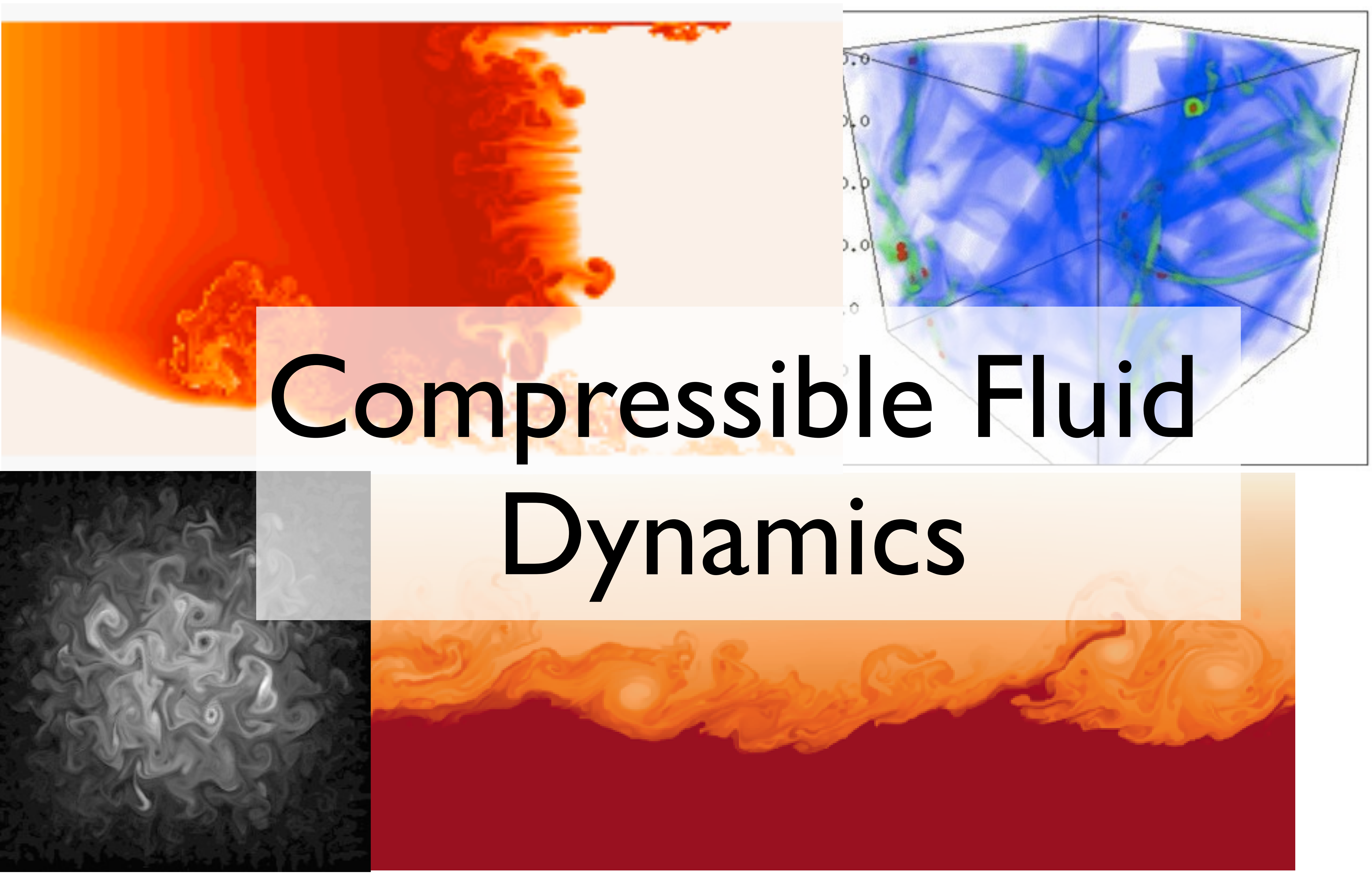


Compressible Fluid Dynamics

Fluids: Almost Everything

- 99% of the visible matter in the Universe is in the form of fluids
- Most of the astrophysical systems we don't fully understand, it's the fluid dynamics tripping us up

M42 - Orion Nebula

Credit: NASA, ESA, M. Robberto (STScI/ESA) and the Hubble Space Telescope Orion Treasury Project Team
http://antwrp.gsfc.nasa.gov/apod/ap060119.html

Equations of Hydrodynamics

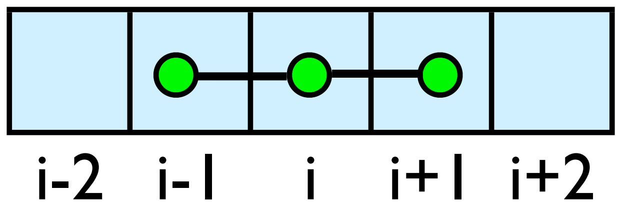
- Density, momentum, and energy equations
- Supplemented by an equation of state - pressure as a function of dens, energy

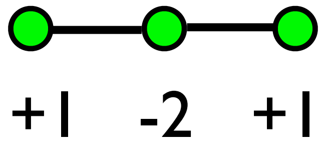
$$
\begin{aligned}
\frac{\partial}{\partial t}\rho + \nabla\cdot(\rho\mathbf{v}) &= 0\\
\frac{\partial}{\partial t}(\rho\mathbf{v}) + \nabla\cdot(\rho\mathbf{v}\mathbf{v}) &= -\nabla p\\
\frac{\partial}{\partial t}(\rho E) + \nabla\cdot((\rho E + p)\,\mathbf{v}) &= 0
\end{aligned}
$$

Discretizing Derivatives

$$\left.\frac{d^2Q}{dx^2}\right|_i \approx \frac{Q_{i+1} - 2Q_i + Q_{i-1}}{\Delta x^2}$$

- Done by finite differencing the discretized values
- Implicitly or explicitly involves interpolating data and taking derivative of the interpolant
- More accuracy - larger ‘stencils’

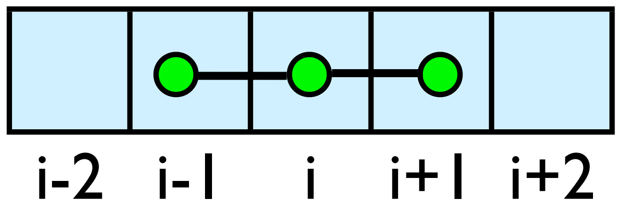


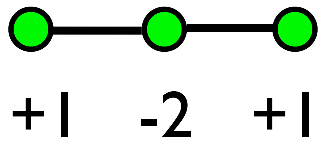


Discretizing Derivatives

- Explicit hydrodynamics: only need information from as far away as the stencil reaches
- Nearest few neighbors
- Locality galore!

$$
\begin{aligned}
\frac{\partial Q}{\partial t} &= f\left(\frac{\partial Q}{\partial x}\right) \\
\left.\frac{\partial Q^{(n)}}{\partial t}\right|_i &\approx \frac{Q_i^{(n+1)} - Q_i^{(n)}}{\Delta t} \\
\left.\frac{dQ^{(n)}}{dx}\right|_i &\approx \frac{Q_{i+1}^{(n)} - Q_{i-1}^{(n)}}{\Delta x} \\
Q_i^{(n+1)} &= Q_i^{(n)} + \Delta t f\left(\frac{Q_i^{(n+1)} - Q_i^{(n)}}{\Delta t}\right)
\end{aligned}
$$

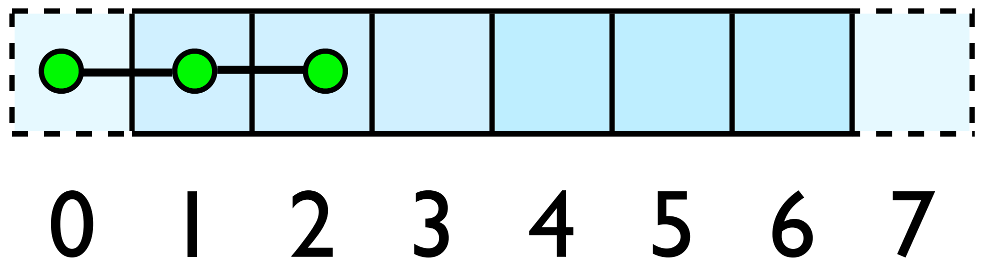




Guardcells

- How to deal with boundaries?
- Because stencil juts out, need information on cells beyond those you are updating
- Pad domain with ‘guard cells’ so that stencil works even for the 0th point in domain
- Fill guard cells with values such that the required boundary conditions are met

Global Domain



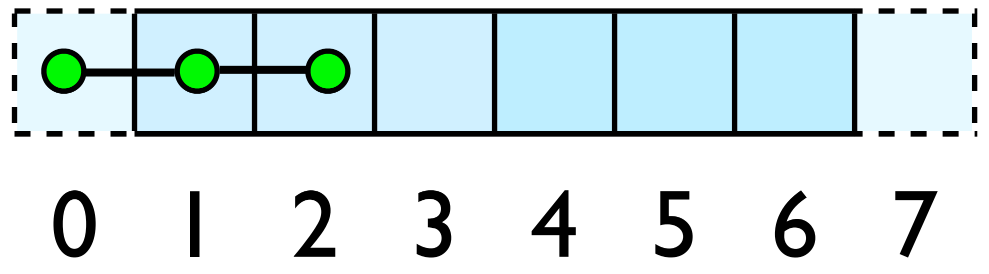
ng = 1

loop from ng, N - 2 ng

Guardcells

- Impose BCs before each timestep
- Our hydro code - 3 common boundary conditions
- ‘outflow’, reflect, and periodic
- Outflow (-1)- cell 0 just gets value from 1
- Reflect (-2); mirror the values
- Periodic(-3); copy values from other side (cell 0 gets values from cell 6)

Global Domain



ng = 1

loop from ng, N - 2 ng

Equations of Hydrodynamics

- Density, momentum, and energy equations
- Supplemented by an equation of state - pressure & temperature as a function of dens, energy

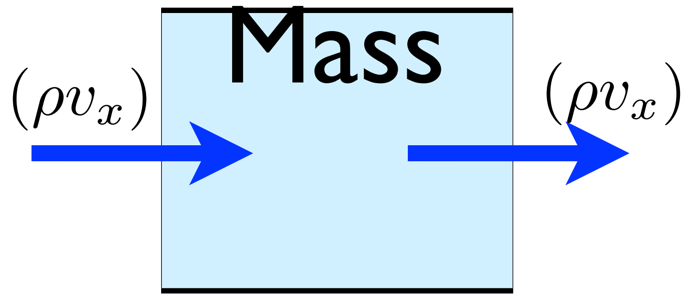
$$\begin{aligned}
\frac{\partial}{\partial t}\rho + \nabla \cdot (\rho \mathbf{v}) &= 0 \\
\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v}\mathbf{v}) &= -\nabla p \\
\frac{\partial}{\partial t}(\rho E) + \nabla \cdot ((\rho E + p)\,\mathbf{v}) &= 0
\end{aligned}$$

Conservation Law form

- Conservation of mass, momentum, energy
- These are important properties, want numerical solver to maintain them

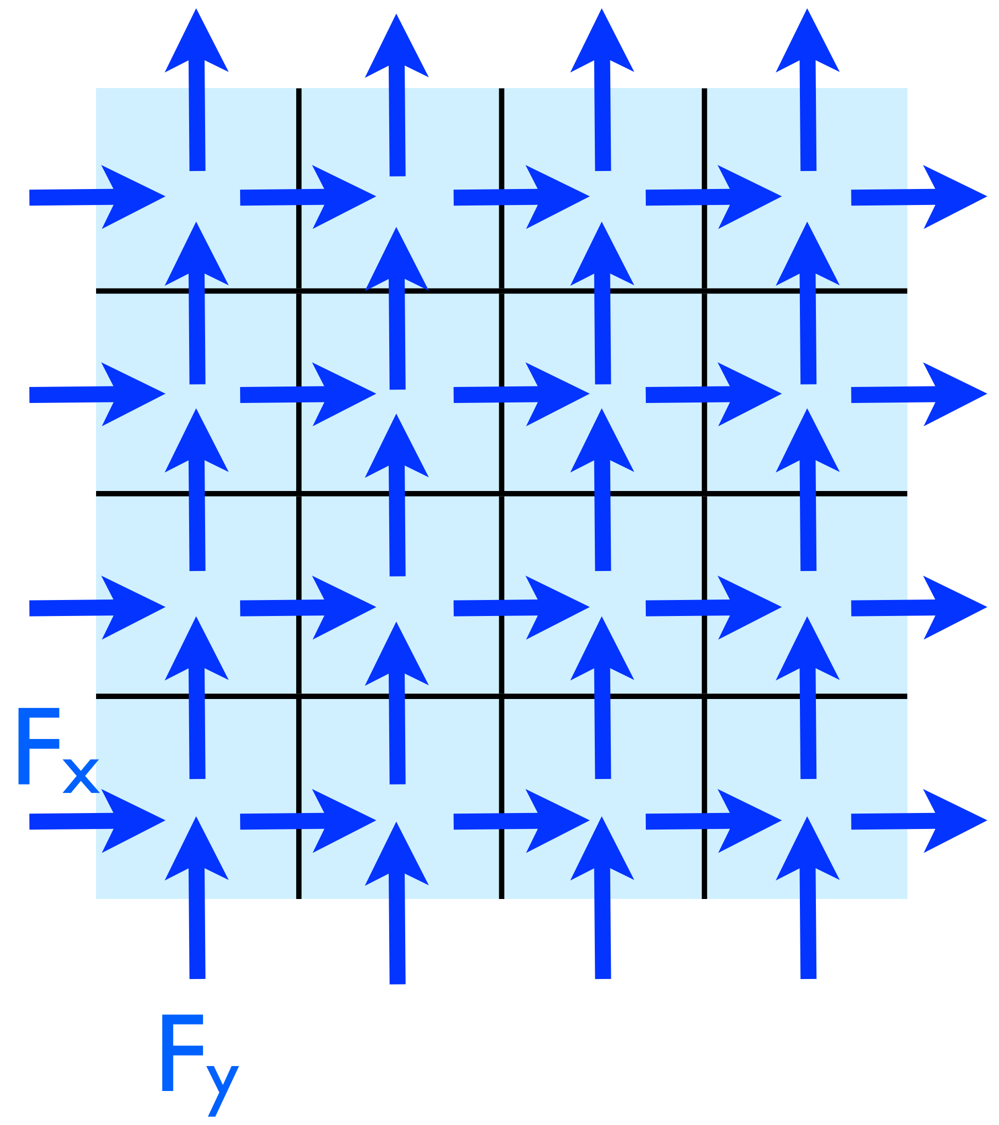
$$
\begin{aligned}
\frac{\partial}{\partial t}\rho + \nabla\cdot(\rho\mathbf{v}) &= 0\\
\frac{\partial}{\partial t}\rho + \frac{\partial}{\partial x}(\rho v_x) &= 0\\
\int_{x_L}^{x_R}\frac{\partial}{\partial t}\rho dx &= -\int_{x_L}^{x_R}\frac{\partial}{\partial x}(\rho v_x)\\
\frac{\partial}{\partial t}\mathrm{Mass} &= -(\rho v_x)_R + (\rho v_x)_L
\end{aligned}
$$

Change in mass = -outflux + influx



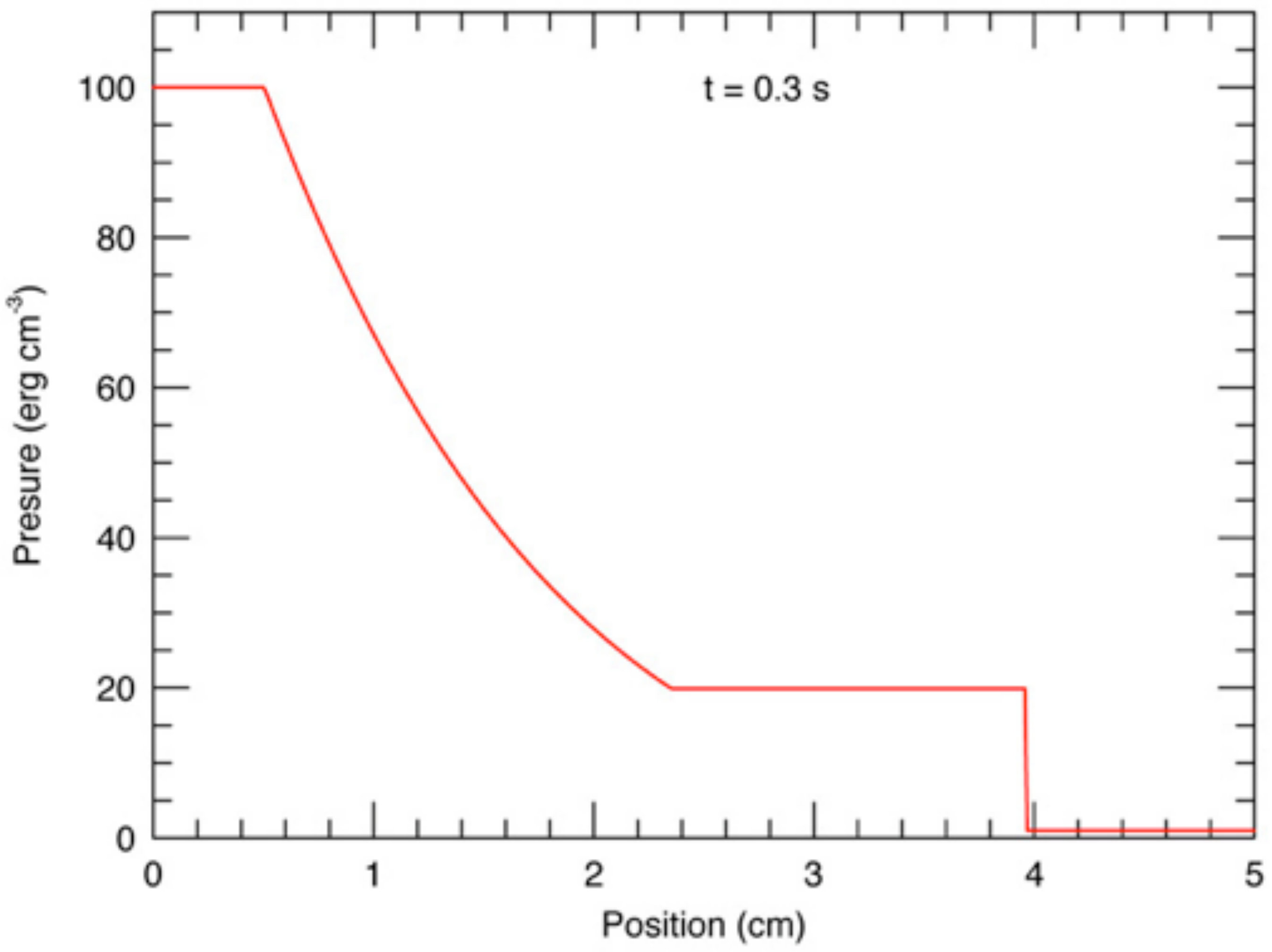
Finite Volume Method

- Conservative; very well suited to high-speed flows with shocks
- At each timestep, calculate fluxes using interpolation/finite differences, and update cell quantities.
- Use conserved variables -- eg, momentum, not velocity.



Flux Calculations

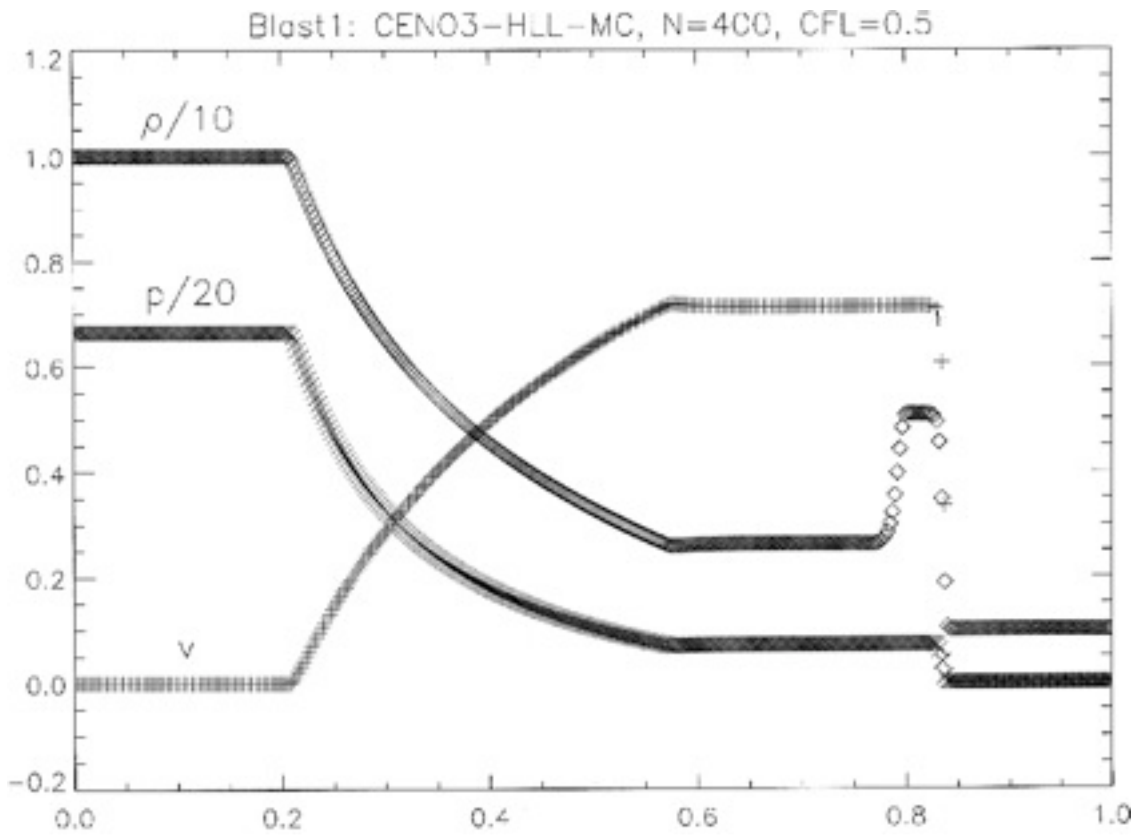
- Compressible flows: common to use Godunov-based schemes
- At cell interfaces, a Riemann problem is solved -- exact solution to a fluid jump
- Expensive, but does a great job of dealing with shocks



Frank Timmes,
http://cococubed.asu.edu/code_pages/exact_riemann.shtml

Flux Calculations

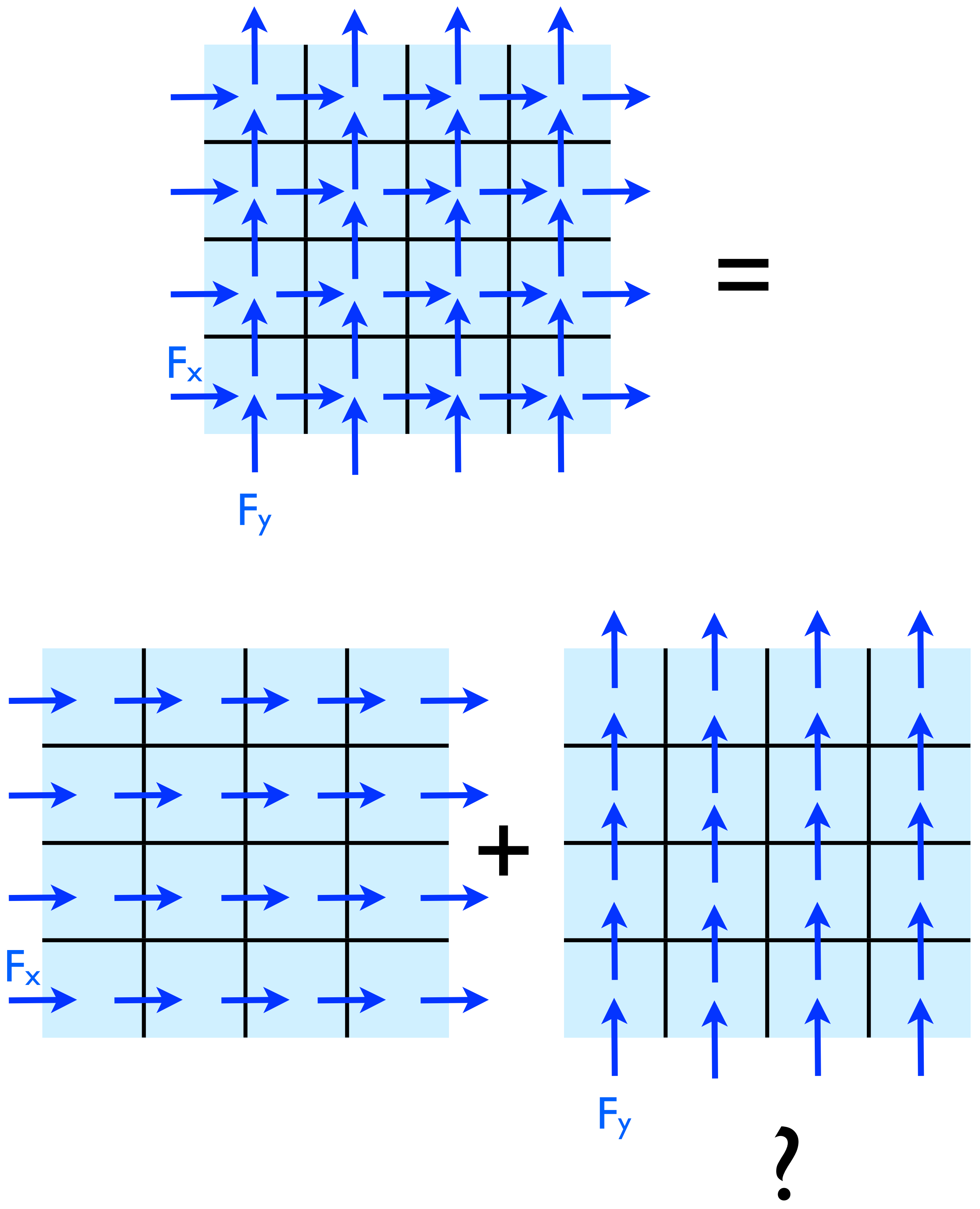
- We're using a 'central scheme' or 'Kurganov scheme'
- No Riemann solve; average over possible waves
- Averaging means shocks are smeared out compared to Riemann solvers; but much faster, simpler to code (particularly for RHD, MHD)

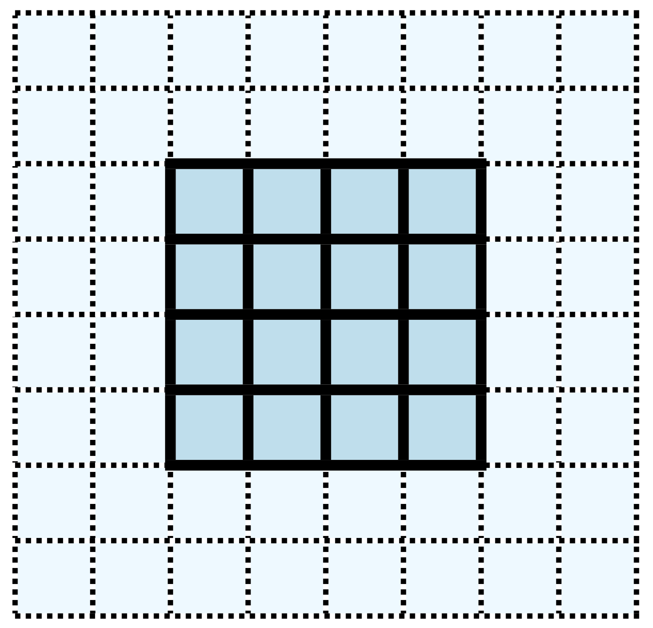


Del Zanna, Bucciantini (2002) A&A **390**:1177

Dimensional Splitting

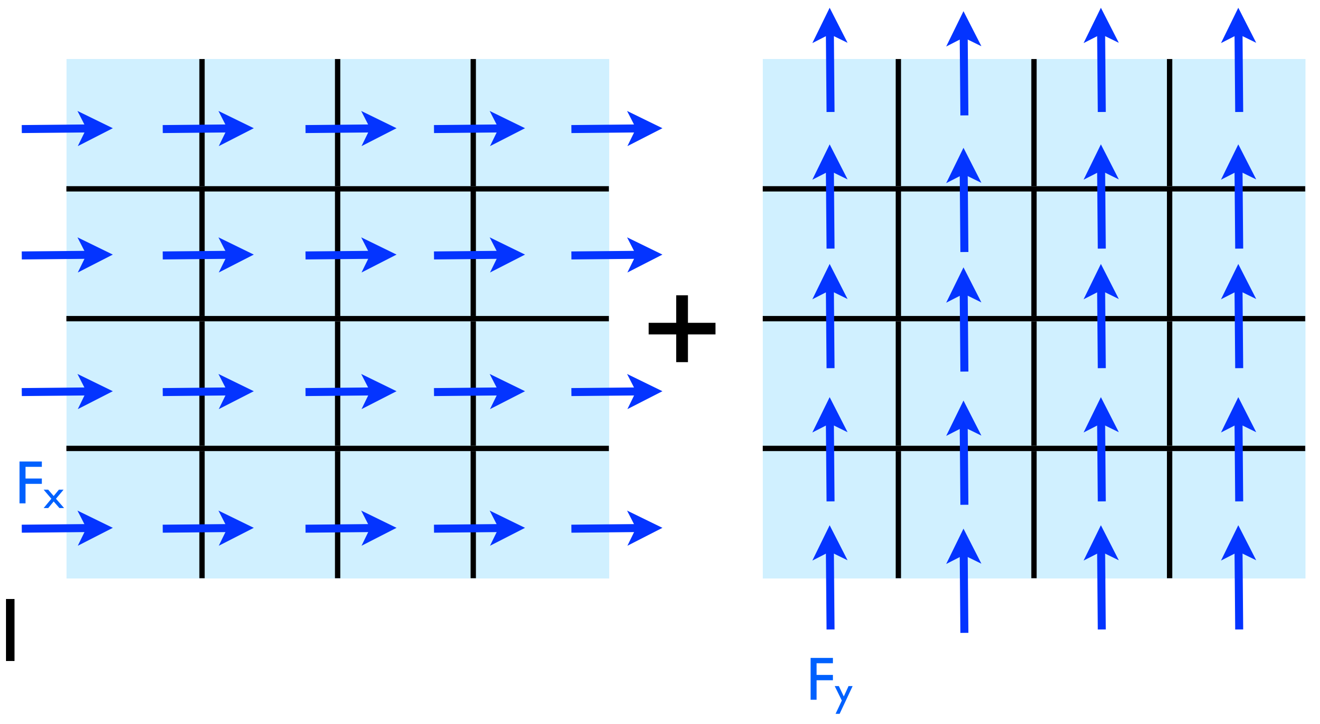
- Strang Splitting: Operators (including X and Y hydro operators) can be done separately, at cost of limiting time accuracy to Δt^2.
- Not at all obvious that should work as well as it does.
- Makes code much easier - get a 1d solver working, build 3d solver trivially





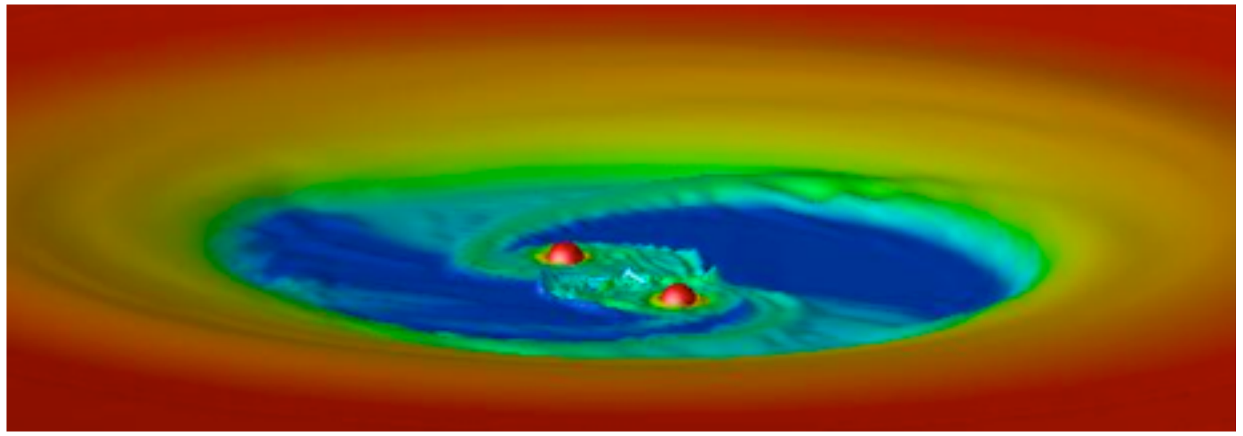
Hydrodynamics

- Finite volume dimensionally split central scheme
- Need only local info (+/- 2 zones in each dimension)
- Implemented with dimensional splitting; sweep in x, then y (then y, then x)



Other Hydrodynamic approaches

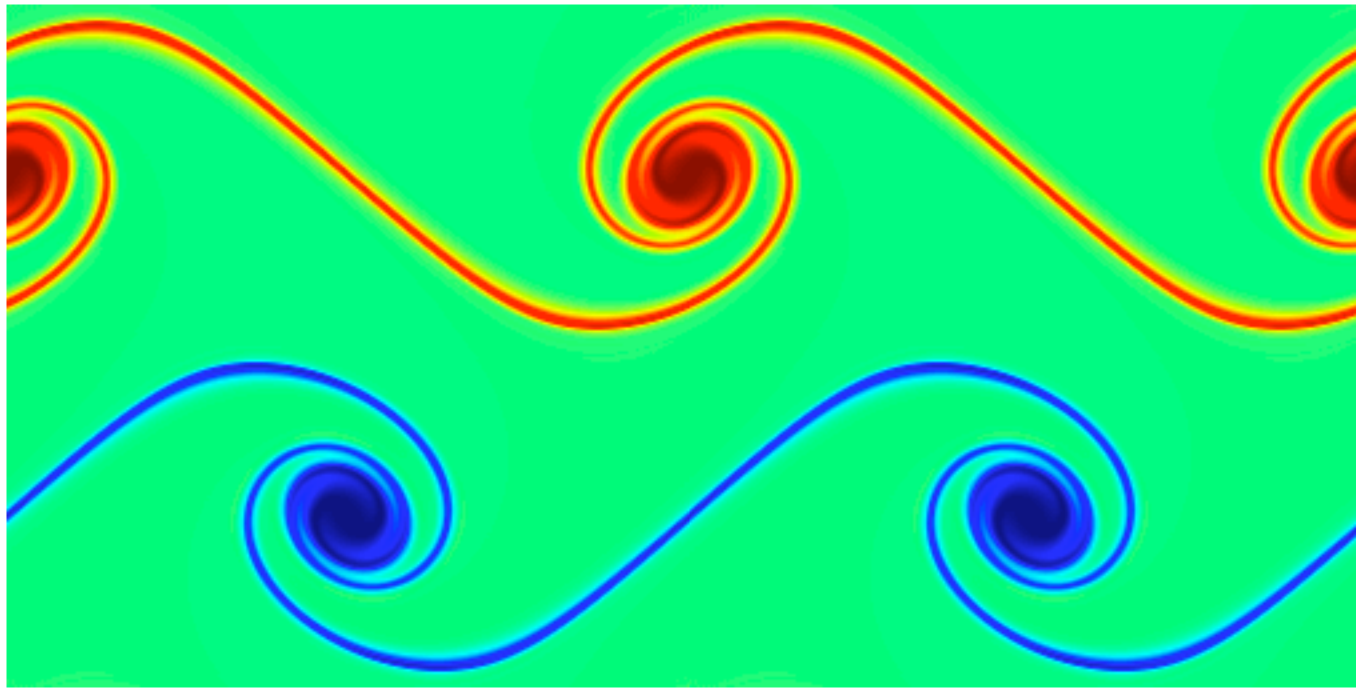
- Finite difference approaches; don’t work in fluxes. Easier to incorporate some types of physics with high time accuracy.
- Parallelization issues same as finite volume codes.



Richard Günther, University of Tübingen.
http://www.tat.physik.uni-tuebingen.de/~rguenth/

Other Hydrodynamic approaches

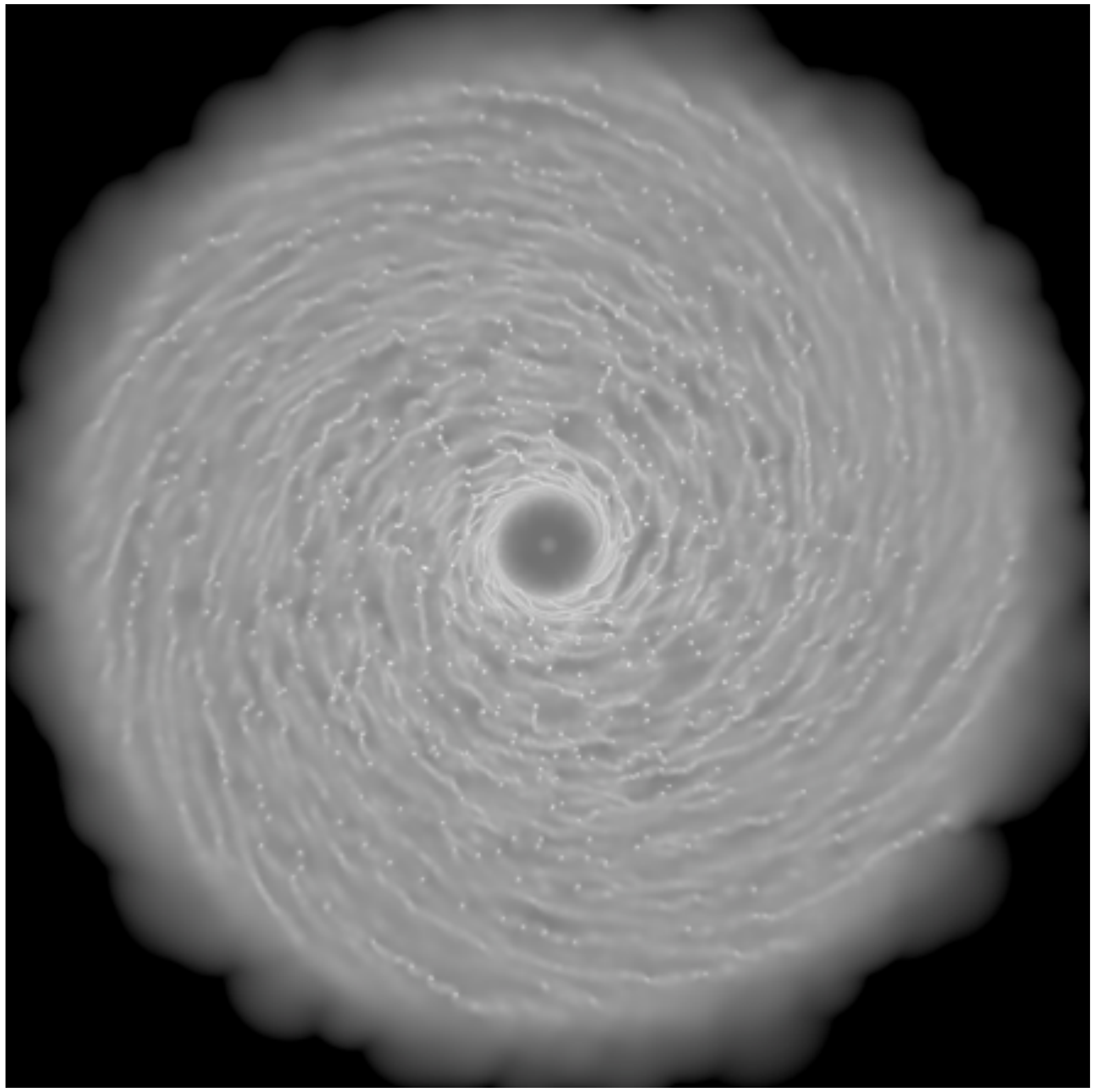
- Incompressible flows
- Additional complexity: elliptical solver (implicit scheme)
- What we have here + linear solvers
- Or Multigrid: also mostly guardcell filling



Mike Zingale, SUNY Stony Brook
http://www.astro.sunysb.edu/mzingale/pyro/

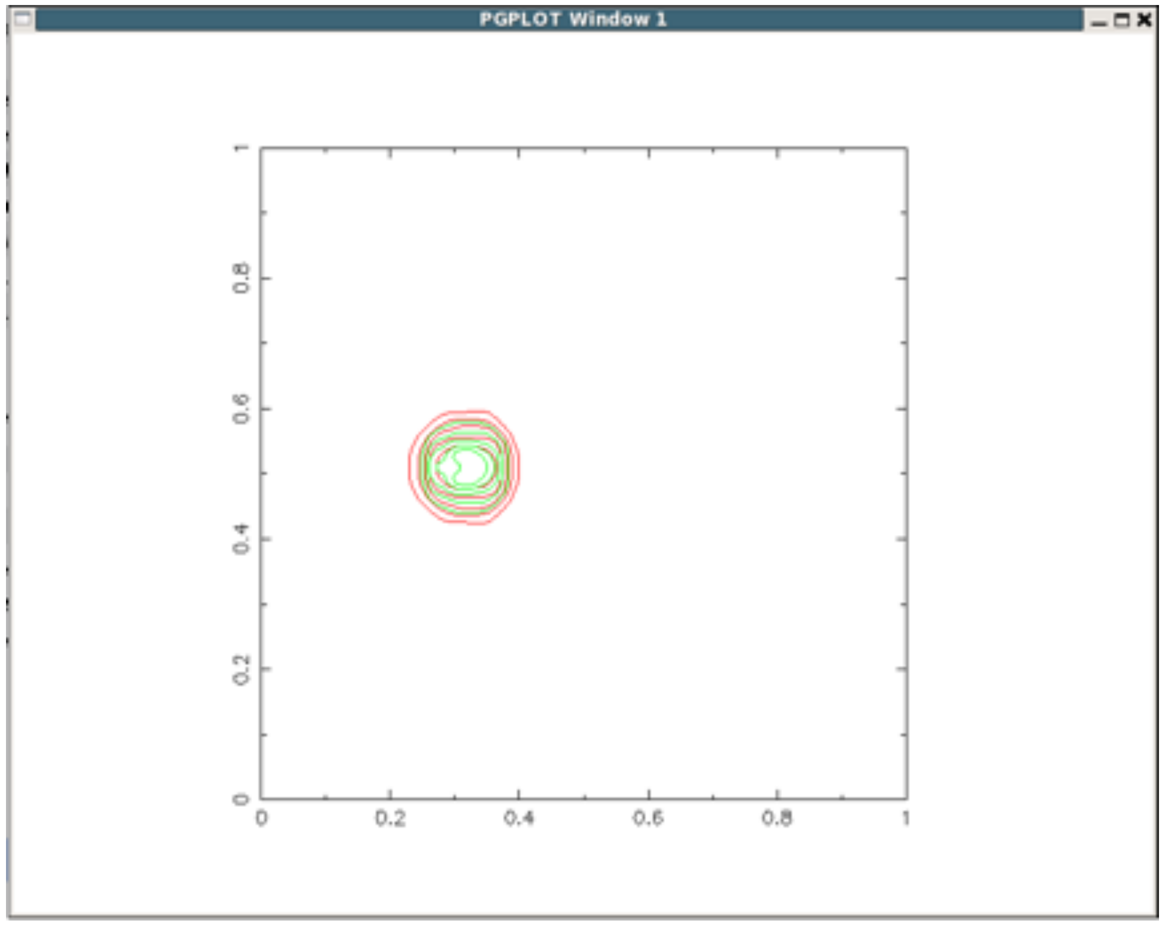
Other Hydrodynamic approaches

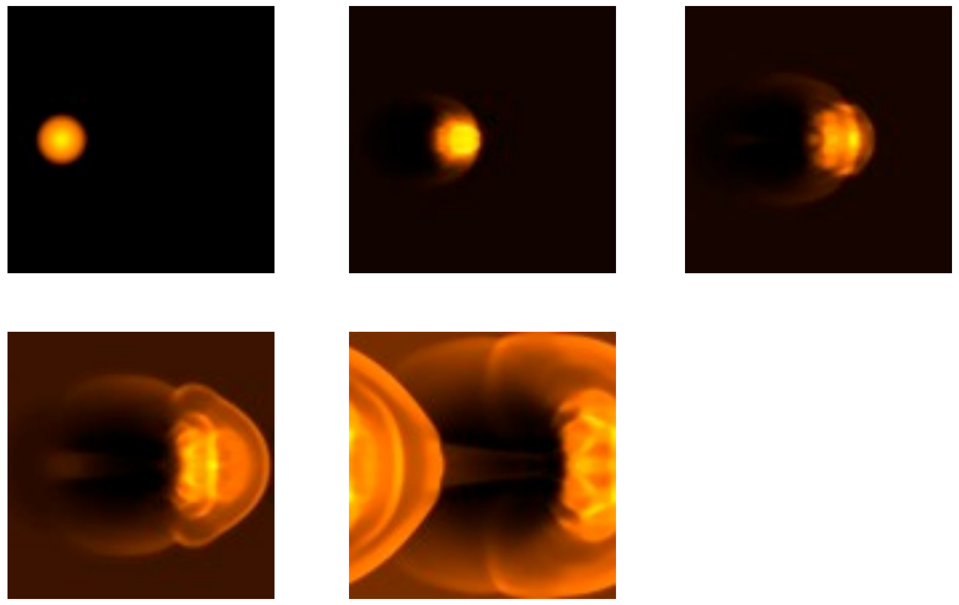
- SPH: no grid at all.   Fluid parcels.
- Hard to do highly accurate schemes, but arguably better suited for some problems.
- Gadget-2
- Some of the same parallelization issues as N-body gravity



Single-Processor hydro code

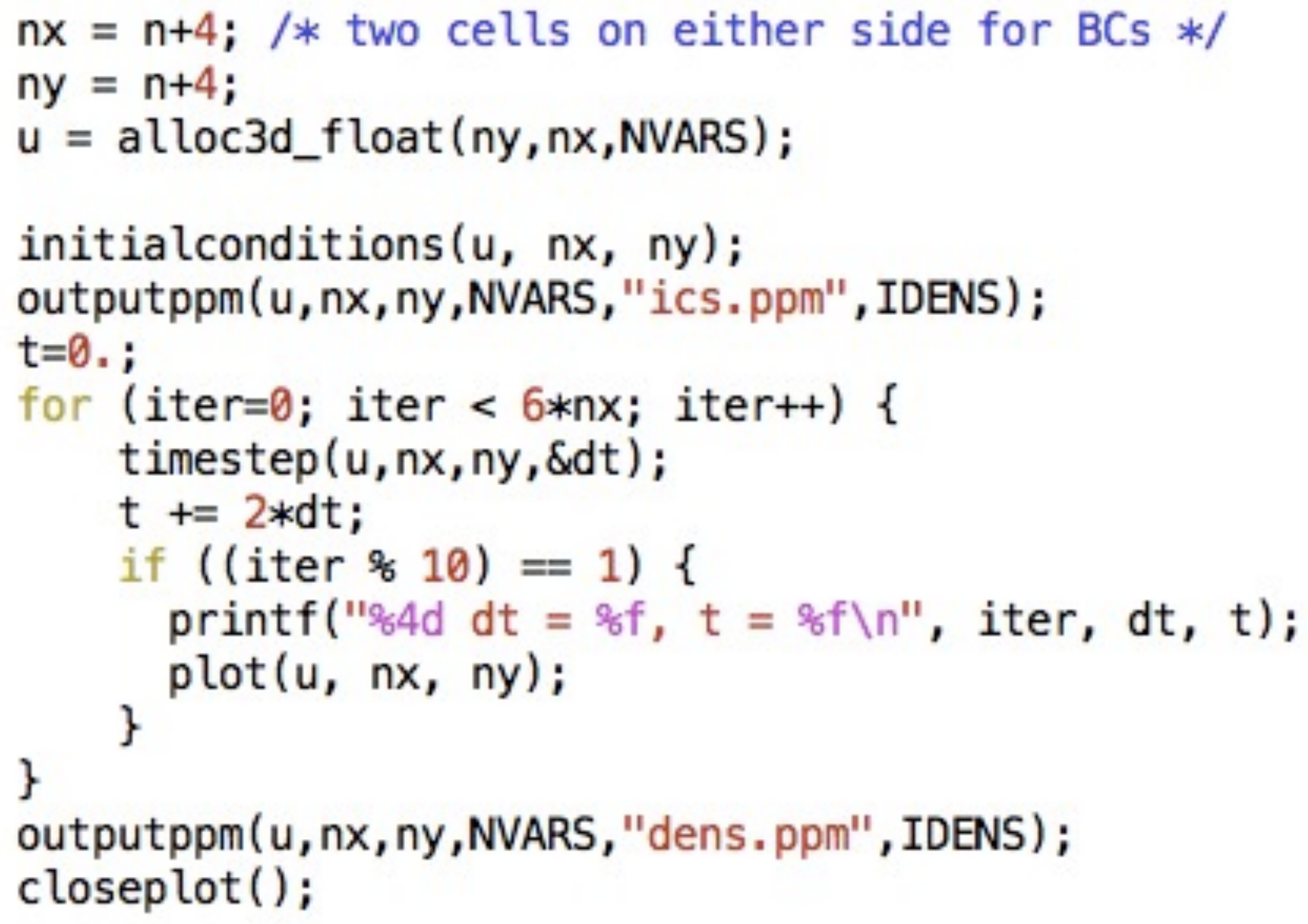
- `cd hydro{c,f};  make`
- `./hydro 100`
- Takes options:
  - number of points to write
- Outputs image (ppm) of initial conditions, final state (plots density)
- display ics.ppm
- display dens.ppm





Single-Processor hydro code

- Set initial conditions
- Loop, calling *timestep()* and maybe some output routines (*plot()* - contours)
- At beginning and end, save an image file with *outputppm()*
- All data stored in array *u*.

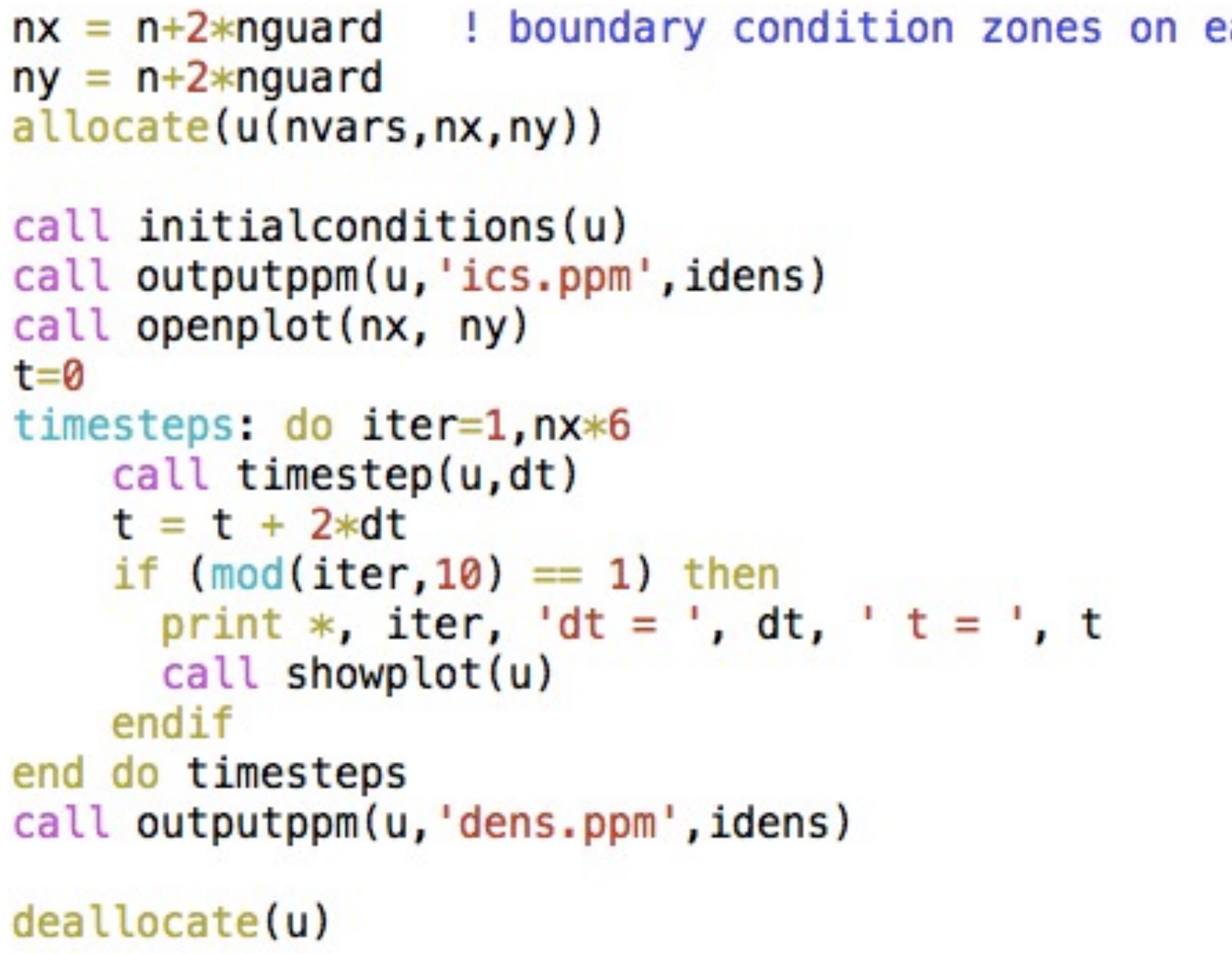


```c
nx = n+4; /* two cells on either side for BCs */
ny = n+4;
u = alloc3d_float(ny,nx,NVARS);

initialconditions(u, nx, ny);
outputppm(u,nx,ny,NVARS,"ics.ppm",IDENS);
t=0.;
for (iter=0; iter < 6*nx; iter++) {
    timestep(u,nx,ny,&dt);
    t += 2*dt;
    if ((iter % 10) == 1) {
      printf("%4d dt = %f, t = %f\n", iter, dt, t);
      plot(u, nx, ny);
    }
}
outputppm(u,nx,ny,NVARS,"dens.ppm",IDENS);
closeplot();
```

hydro.c

Single-Processor hydro code

- Set initial conditions
- Loop, calling *timestep()* and maybe some output routines (*plot()* - contours)
- At beginning and end, save an image file with *outputppm()*
- All data stored in array *u*.

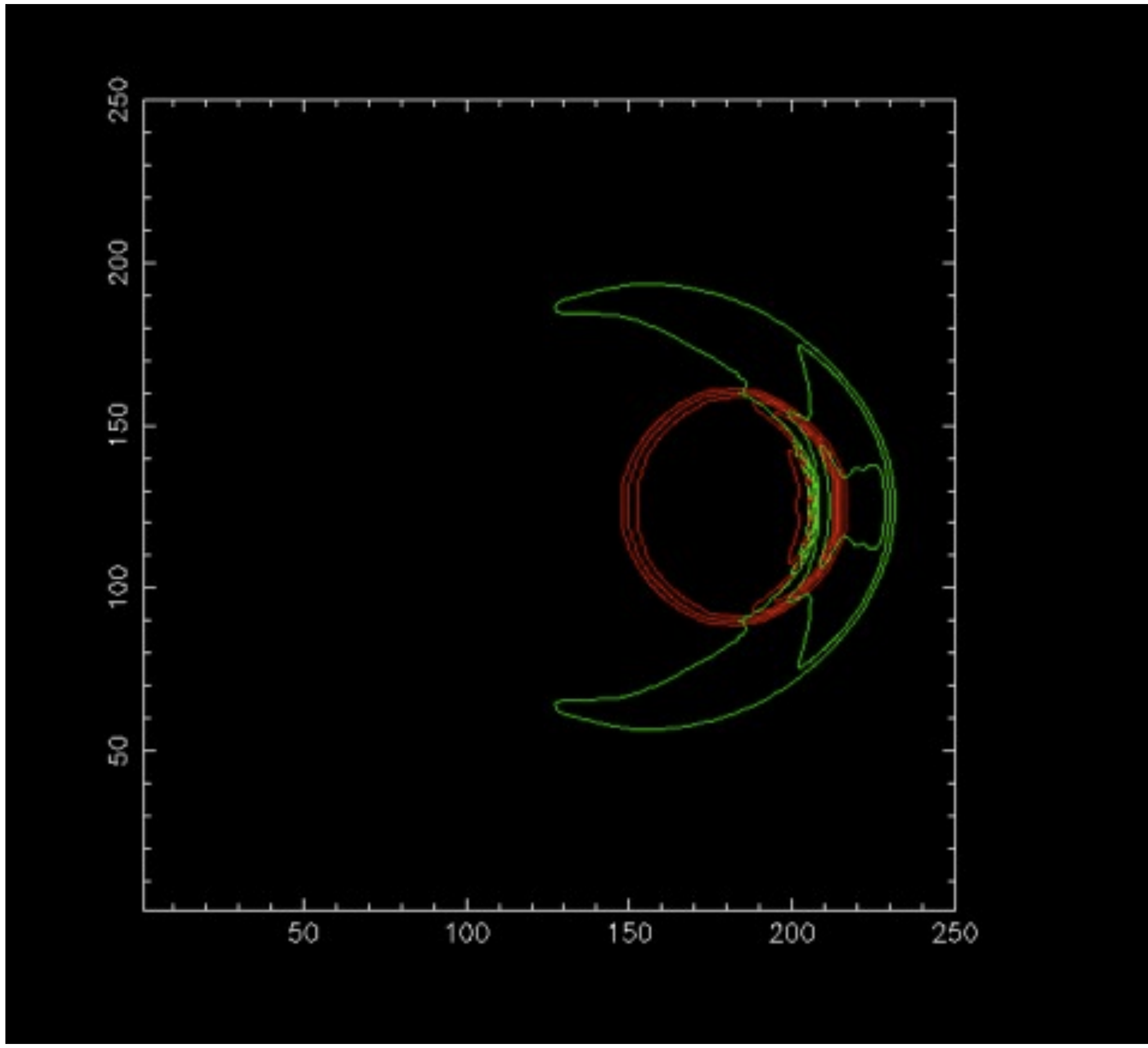


```
nx = n+2*nguard   ! boundary condition zones on e
ny = n+2*nguard
allocate(u(nvars,nx,ny))

call initialconditions(u)
call outputppm(u,'ics.ppm',idens)
call openplot(nx, ny)
t=0
timesteps: do iter=1,nx*6
    call timestep(u,dt)
    t = t + 2*dt
    if (mod(iter,10) == 1) then
      print *, iter, 'dt = ', dt, ' t = ', t
      call showplot(u)
    endif
end do timesteps
call outputppm(u,'dens.ppm',idens)

deallocate(u)
```

hydro.f90

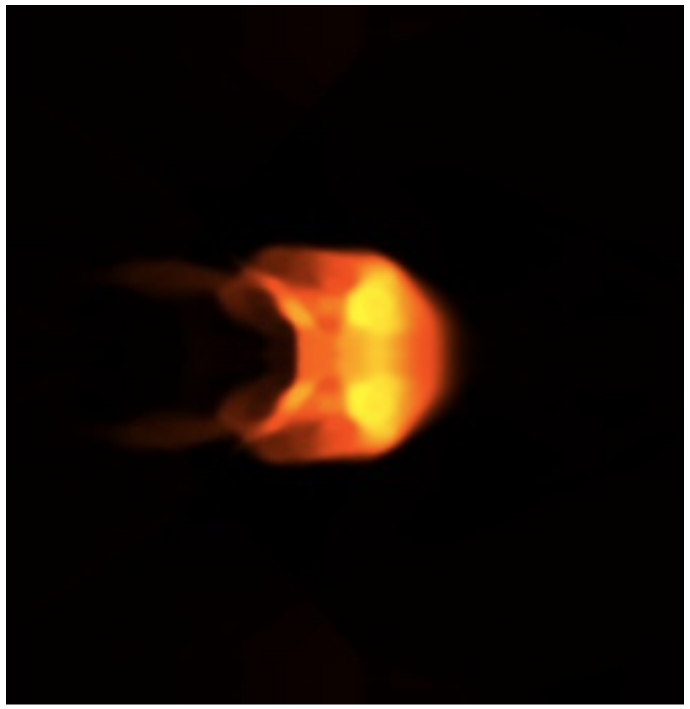
Plotting to screen

- plot.c, plot.f90
- Every 10 timesteps
- Find min, max of pressure, density
- Plot 5 contours of density (red) and pressure (green)
- pgplot library (old, but works).



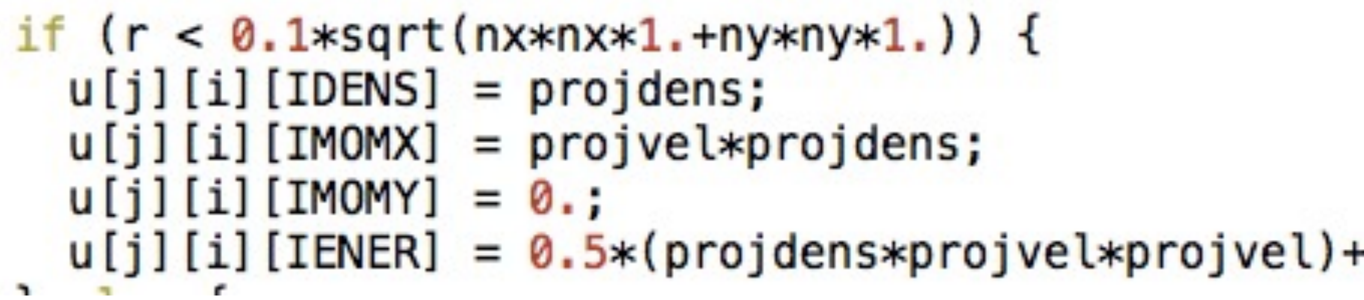
Plotting to file

- ppm.c, ppm.f90
- PPM format -- binary (w/ ascii header)
- Find min, max of density
- Calculate r,g,b values for scaled density (black = min, yellow = max)
- Write header, then data.



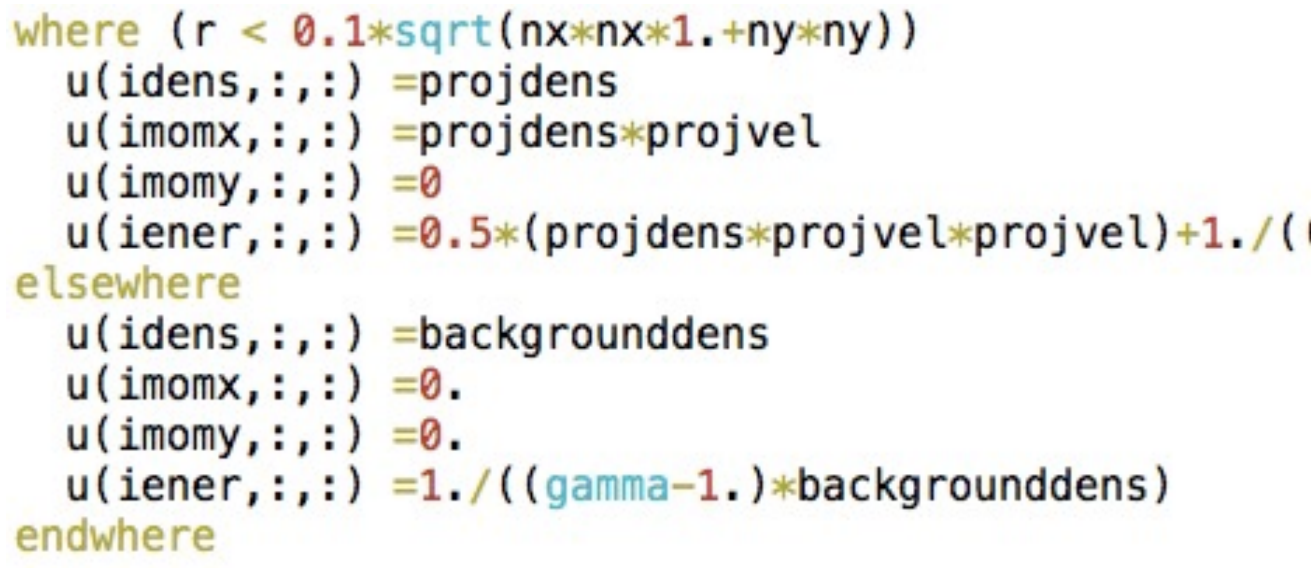
Data structure

- *u* : 3 dimensional array containing each variable in 2d space
- eg, u[j][i][IDENS]
- or u(idens, i, j)



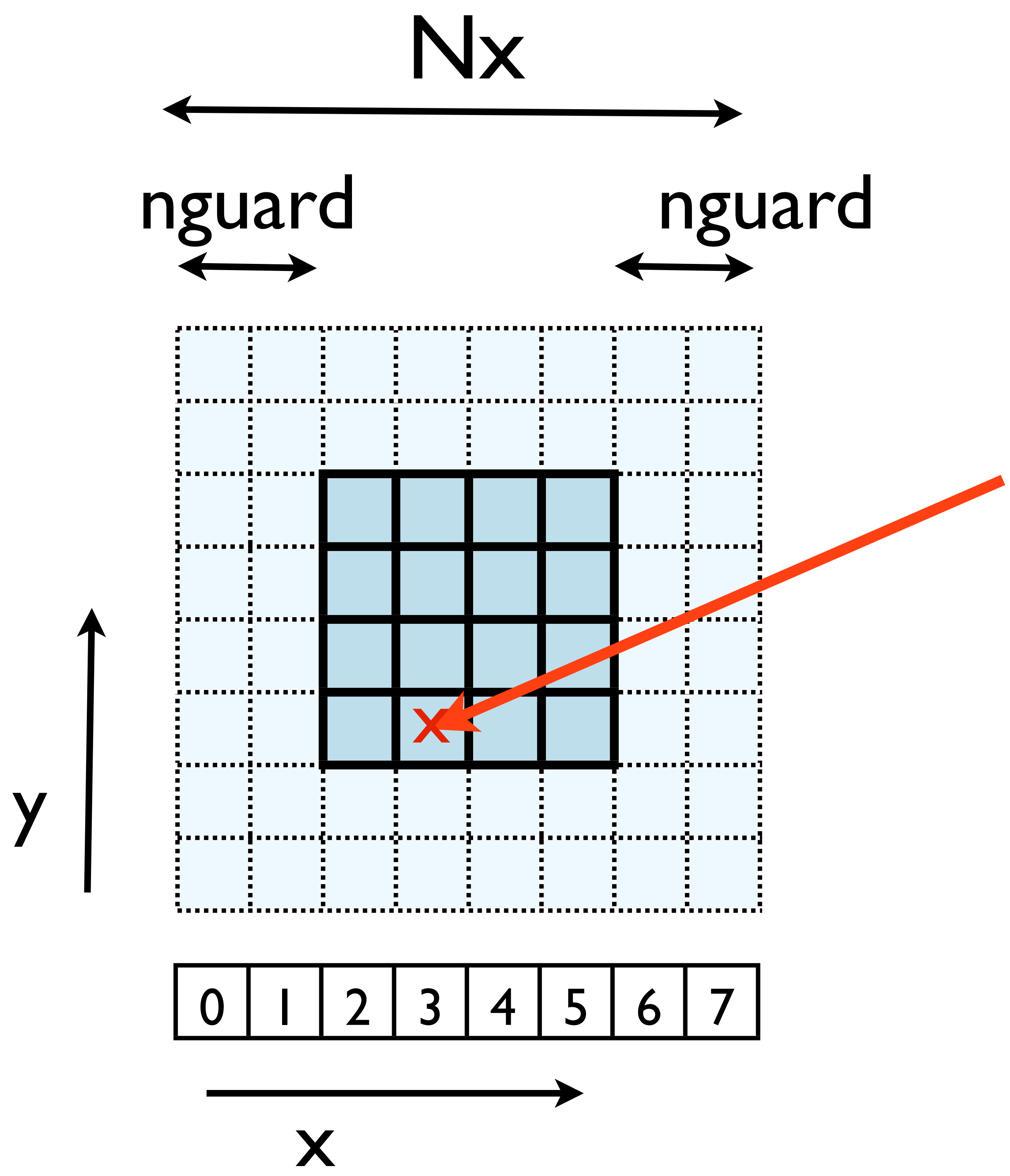
```
if (r < 0.1*sqrt(nx*nx*1.+ny*ny*1.)) {
  u[j][i][IDENS] = projdens;
  u[j][i][IMOMX] = projvel*projdens;
  u[j][i][IMOMY] = 0.;
  u[j][i][IENER] = 0.5*(projdens*projvel*projvel)+
```

solver.c (initialconditions)



```
where (r < 0.1*sqrt(nx*nx*1.+ny*ny))
  u(idens,:,:) =projdens
  u(imomx,:,:) =projdens*projvel
  u(imomy,:,:) =0
  u(iener,:,:) =0.5*(projdens*projvel*projvel)+1./((
elsewhere
  u(idens,:,:) =backgrounddens
  u(imomx,:,:) =0.
  u(imomy,:,:) =0.
  u(iener,:,:) =1./((gamma-1.)*backgrounddens)
endwhere
```

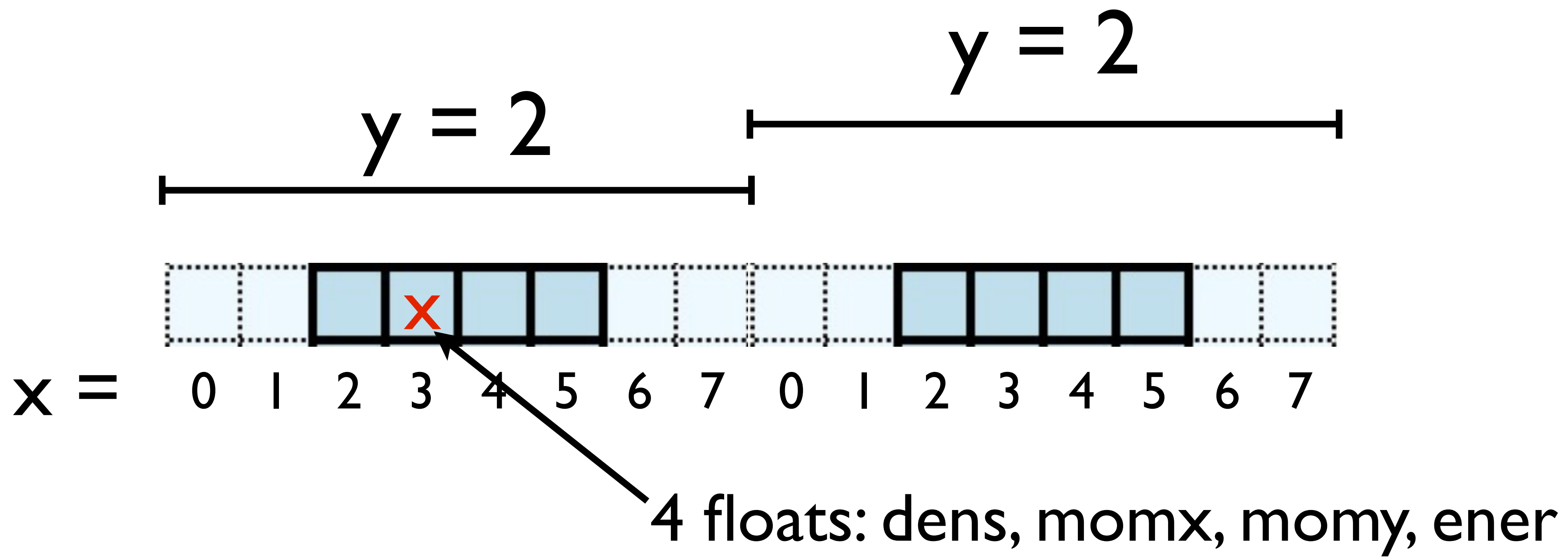
solver.f90 (initialconditions)



```
u[2][3][DENSVAR];
u[2][3][MOMXVAR];
u[2][3][MOMYVAR];
u[2][3][ENERVAR];
```

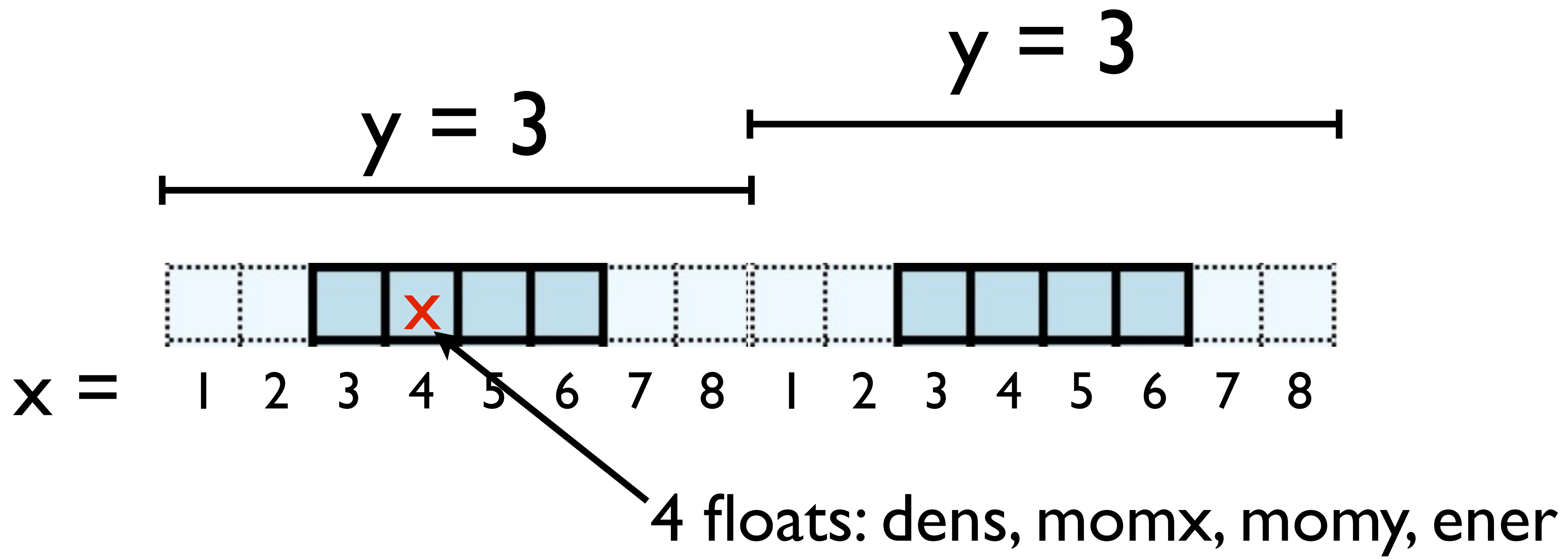
```
u(idens,4,3)
u(imomx,4,3)
u(imomy,4,3)
u(iener,4,3)
```

Laid out in memory (C)


y = 2
y = 2
x
x = 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
4 floats: dens, momx, momy, ener

Same way as in an image file
(one horizontal row at a time)

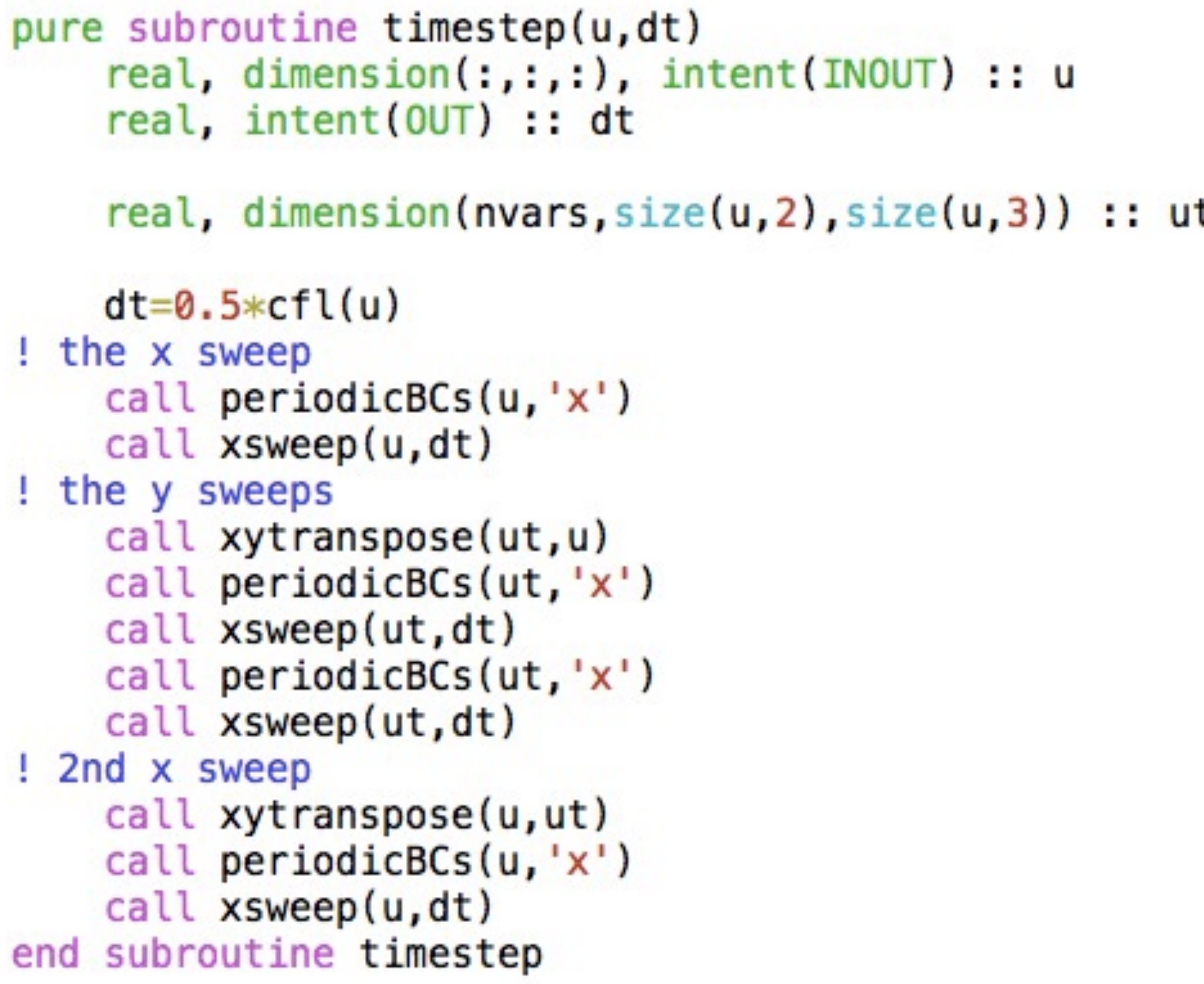
Laid out in memory (FORTRAN)


y = 3
y = 3
x
x = 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8
4 floats: dens, momx, momy, ener

Same way as in an image file
(one horizontal row at a time)

Timestep routine

- Apply boundary conditions
- X sweep, Y sweep
- Transpose entire domain , so Y sweep is just an X sweep
- (unusual approach! But has advantages. Like matrix multiply.)
- Note - dt calculated each step (minimum across domain.)

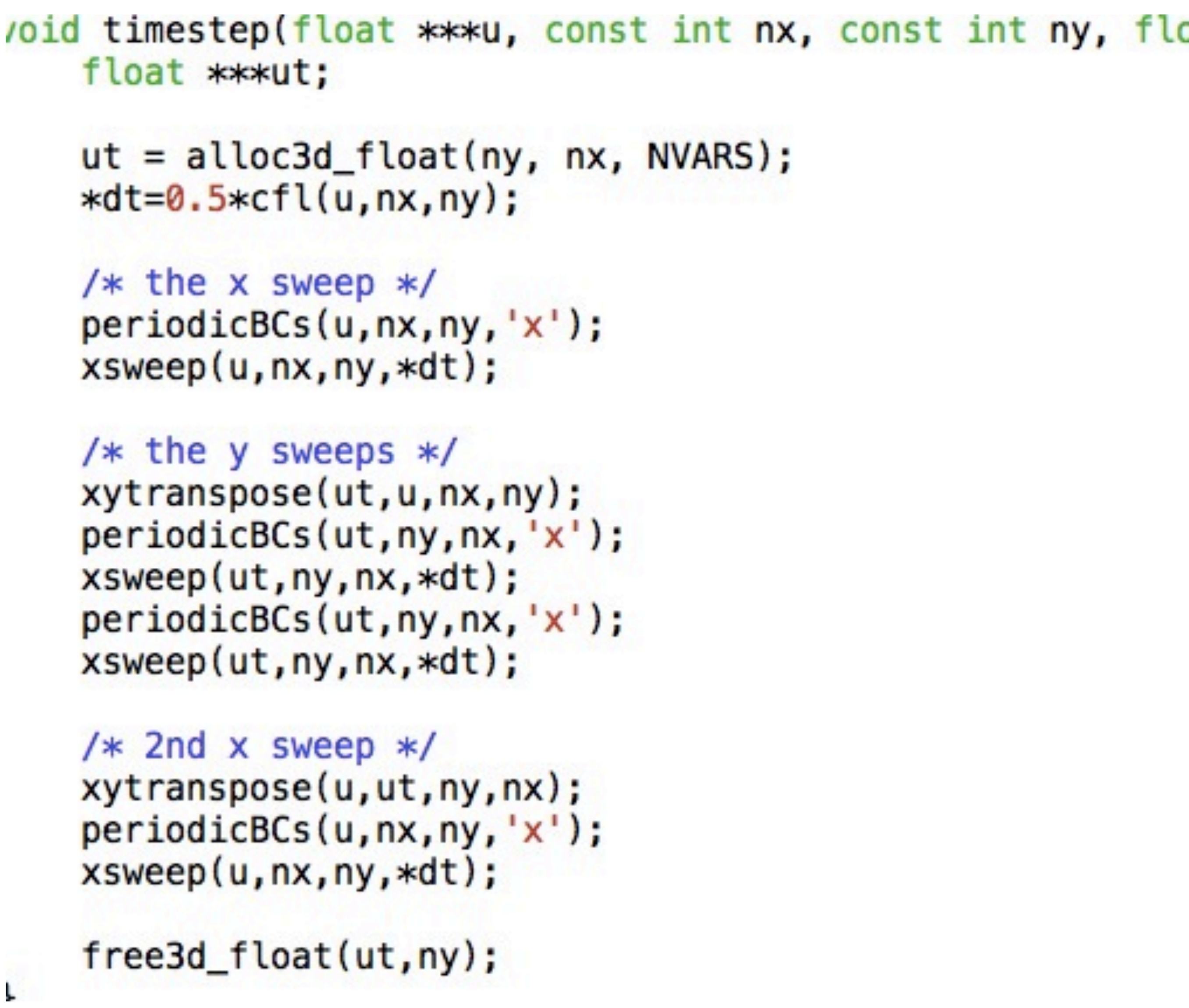


```fortran
pure subroutine timestep(u,dt)
    real, dimension(:,:,:), intent(INOUT) :: u
    real, intent(OUT) :: dt

    real, dimension(nvars,size(u,2),size(u,3)) :: ut

    dt=0.5*cfl(u)
! the x sweep
    call periodicBCs(u,'x')
    call xsweep(u,dt)
! the y sweeps
    call xytranspose(ut,u)
    call periodicBCs(ut,'x')
    call xsweep(ut,dt)
    call periodicBCs(ut,'x')
    call xsweep(ut,dt)
! 2nd x sweep
    call xytranspose(u,ut)
    call periodicBCs(u,'x')
    call xsweep(u,dt)
end subroutine timestep
```

timestep
solver.f90

Timestep routine

- Apply boundary conditions
- X sweep, Y sweep
- Transpose entire domain , so Y sweep is just an X sweep
- (unusual approach! But has advantages. Like matrix multiply.)
- Note - dt calculated each step (minimum across domain.)

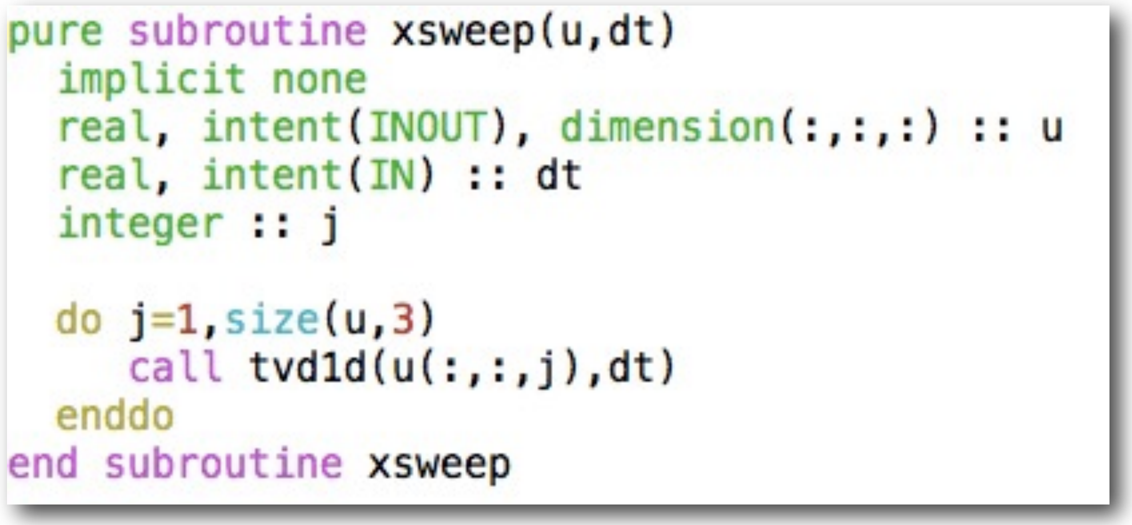


```
/oid timestep(float ***u, const int nx, const int ny, flo
    float ***ut;

    ut = alloc3d_float(ny, nx, NVARS);
    *dt=0.5*cfl(u,nx,ny);

    /* the x sweep */
    periodicBCs(u,nx,ny,'x');
    xsweep(u,nx,ny,*dt);

    /* the y sweeps */
    xytranspose(ut,u,nx,ny);
    periodicBCs(ut,ny,nx,'x');
    xsweep(ut,ny,nx,*dt);
    periodicBCs(ut,ny,nx,'x');
    xsweep(ut,ny,nx,*dt);

    /* 2nd x sweep */
    xytranspose(u,ut,ny,nx);
    periodicBCs(u,nx,ny,'x');
    xsweep(u,nx,ny,*dt);

    free3d_float(ut,ny);
}
```

timestep
solver.c

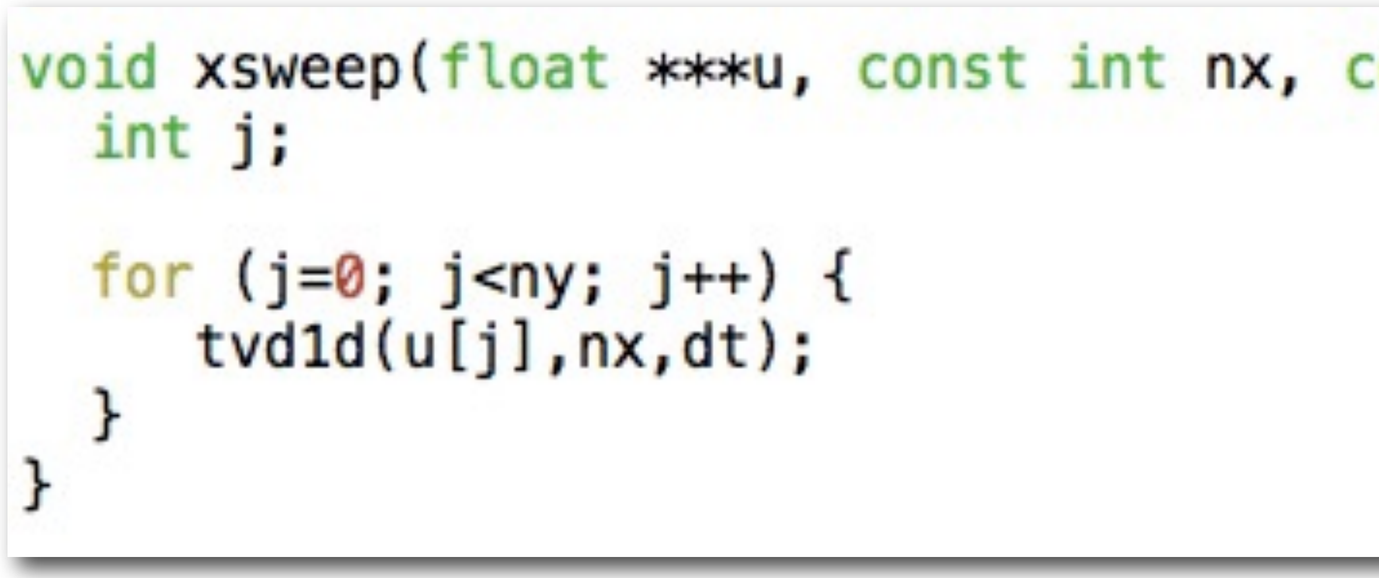
Xsweep routine

- Go through each x "pencil" of cells
- Do 1d hydrodynamics routine on that pencil.



```
pure subroutine xsweep(u,dt)
  implicit none
  real, intent(INOUT), dimension(:,:,:) :: u
  real, intent(IN) :: dt
  integer :: j

  do j=1,size(u,3)
     call tvd1d(u(:,:,j),dt)
  enddo
end subroutine xsweep
```

xsweep
solver.f90



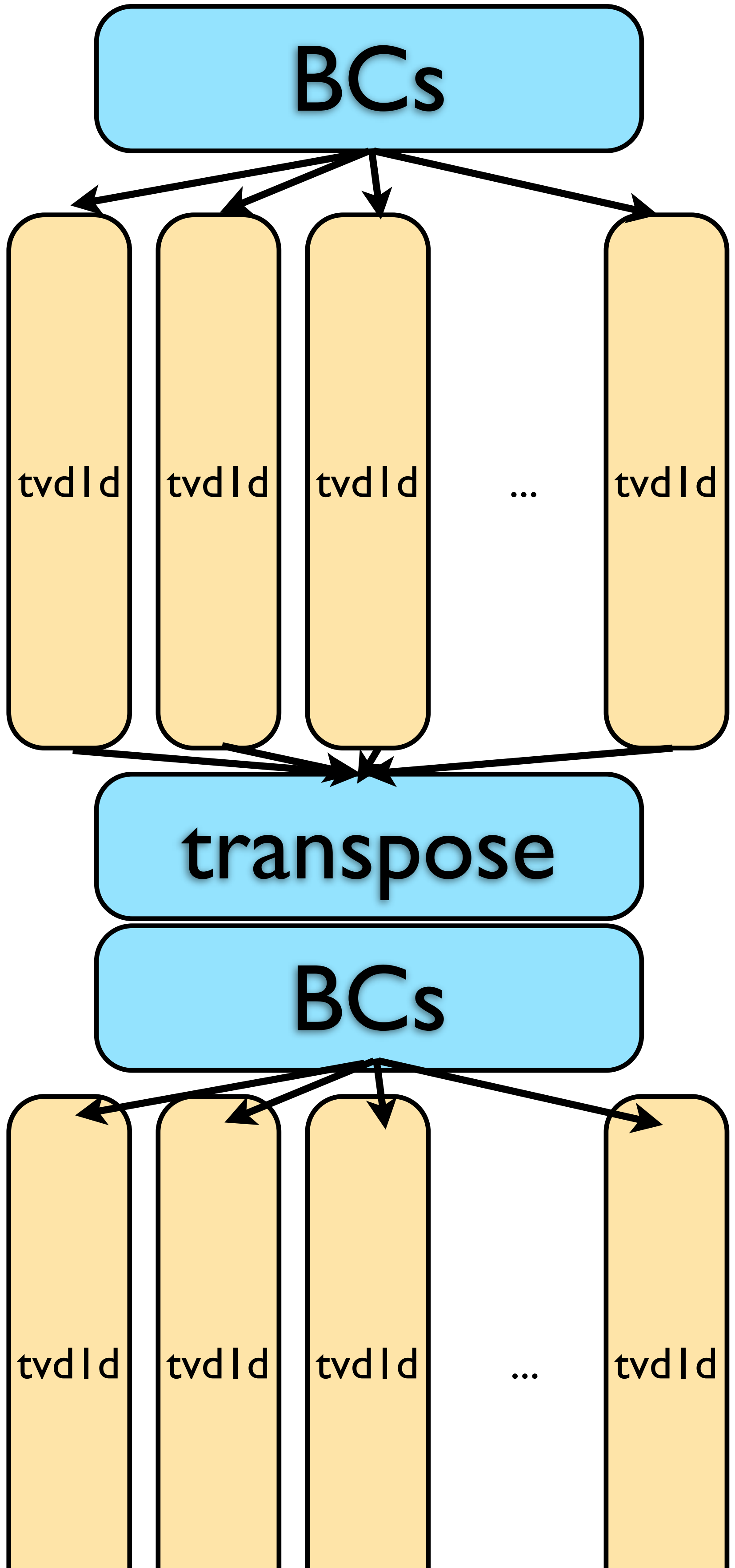
```
void xsweep(float ***u, const int nx, c
  int j;

  for (j=0; j<ny; j++) {
     tvd1d(u[j],nx,dt);
  }
}
```

xsweep
solver.c

What do data dependancies look like for this?

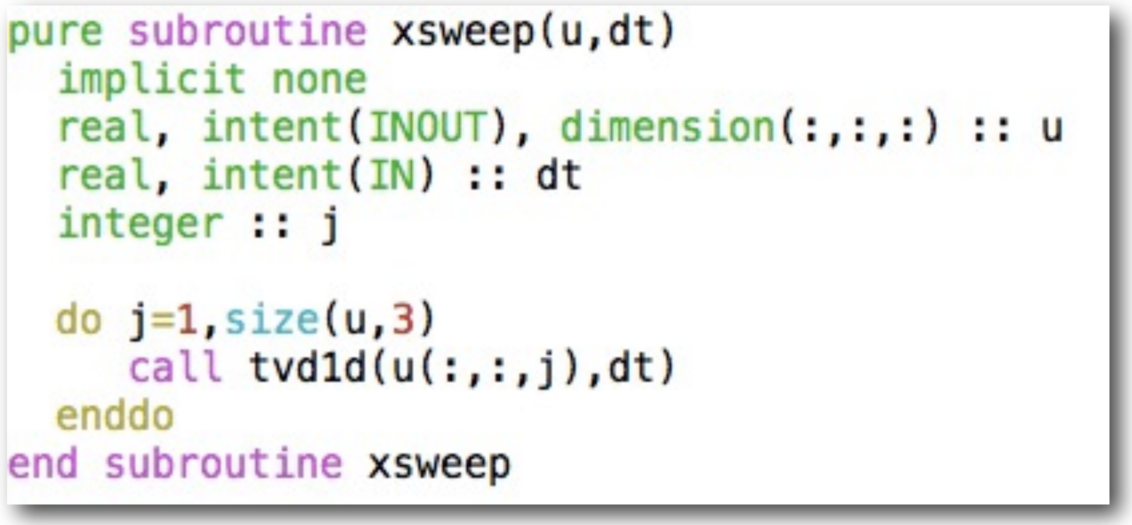
Data dependencies

- Previous timestep must be completed before next one started.
- Within each timestep,
- Each tvd1d "pencil" can be done independently
- All must be done before transpose, BCs



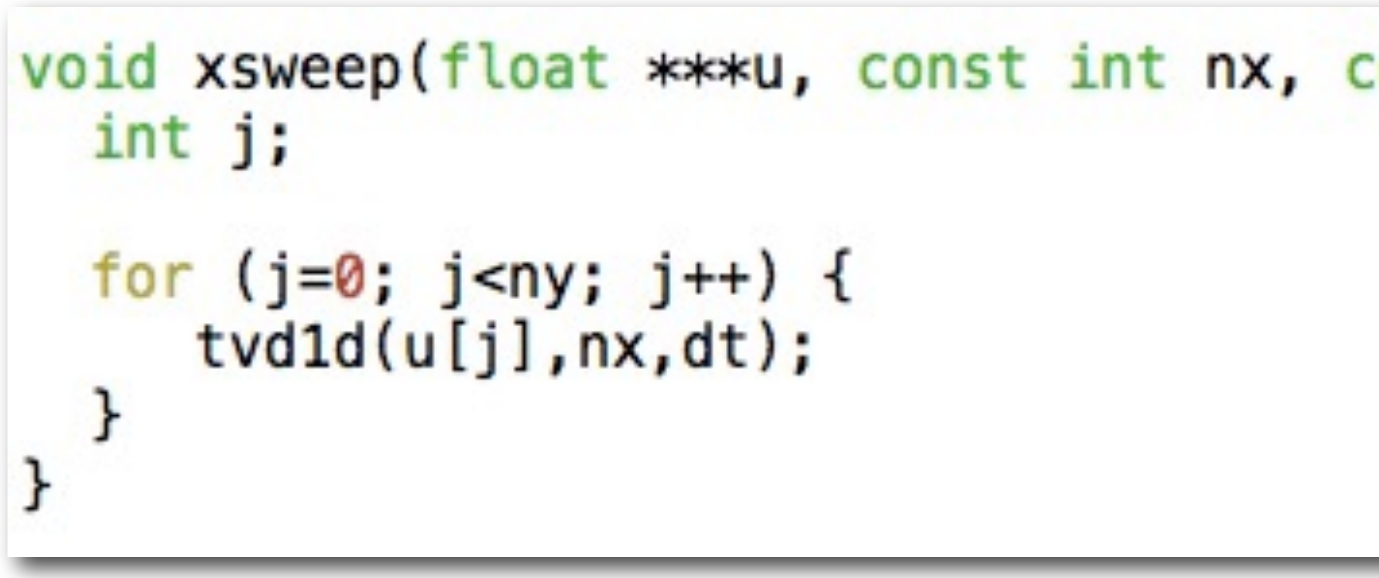
Looks like OpenMP!

- OpenMP of this code is trivial
- Wrap j loop with omp parallel for
- Almost all of the physics is in this tvd1d routine.



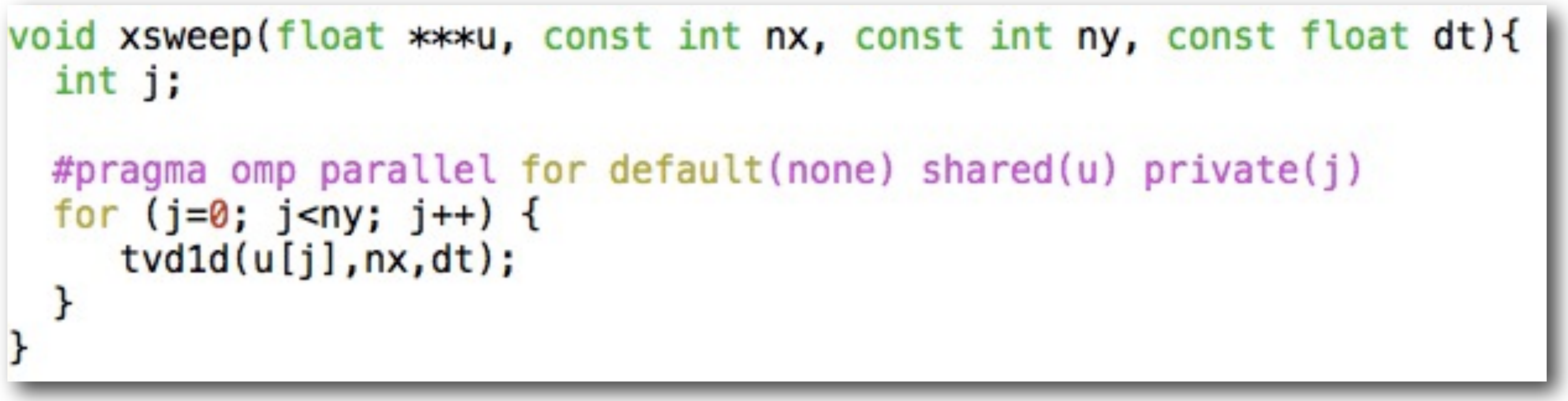
```fortran
pure subroutine xsweep(u,dt)
  implicit none
  real, intent(INOUT), dimension(:,:,:) :: u
  real, intent(IN) :: dt
  integer :: j

  do j=1,size(u,3)
     call tvd1d(u(:,:,j),dt)
  enddo
end subroutine xsweep
```

xsweep
solver.f90



```c
void xsweep(float ***u, const int nx, c
  int j;

  for (j=0; j<ny; j++) {
     tvd1d(u[j],nx,dt);
  }
}
```

xsweep
solver.c

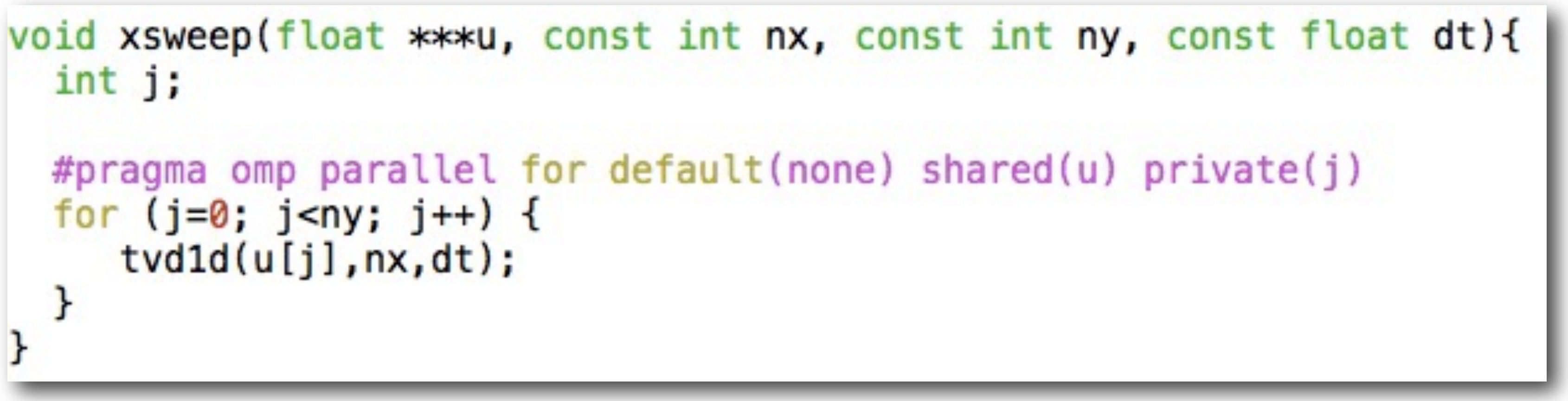


```c
void xsweep(float ***u, const int nx, const int ny, const float dt){
  int j;

  #pragma omp parallel for default(none) shared(u) private(j)
  for (j=0; j<ny; j++) {
     tvd1d(u[j],nx,dt);
  }
}
```

```
$ export OMP_NUM_THREADS=1
$ time ./hydro 100

real	0m7.256s
user	0m7.222s
sys	0m0.003s

$ export OMP_NUM_THREADS=8
$ time ./hydro 100

real	0m1.453s
user	0m11.540s
sys	0m0.044s
```

5x speedup with 1 line of code!

(all output removed)



```c
void xsweep(float ***u, const int nx, const int ny, const float dt){
  int j;

  #pragma omp parallel for default(none) shared(u) private(j)
  for (j=0; j<ny; j++) {
     tvd1d(u[j],nx,dt);
  }
}
```

```
$ export OMP_NUM_THREADS=1
$ time ./hydro 500

real    3m36.728s
user    3m36.680s
sys     0m0.013s

$ export OMP_NUM_THREADS=8
$ time ./hydro 500

real    0m47.459s
user    6m18.849s
sys     0m0.598s
```

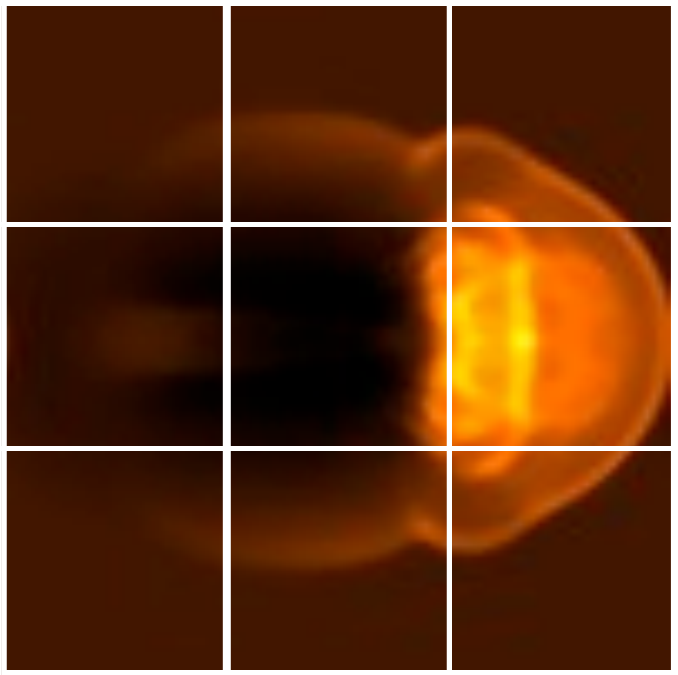
5x speedup with 1 line of code!

(all output removed)

cfl(), xytranspose() could usefully be parallelized.

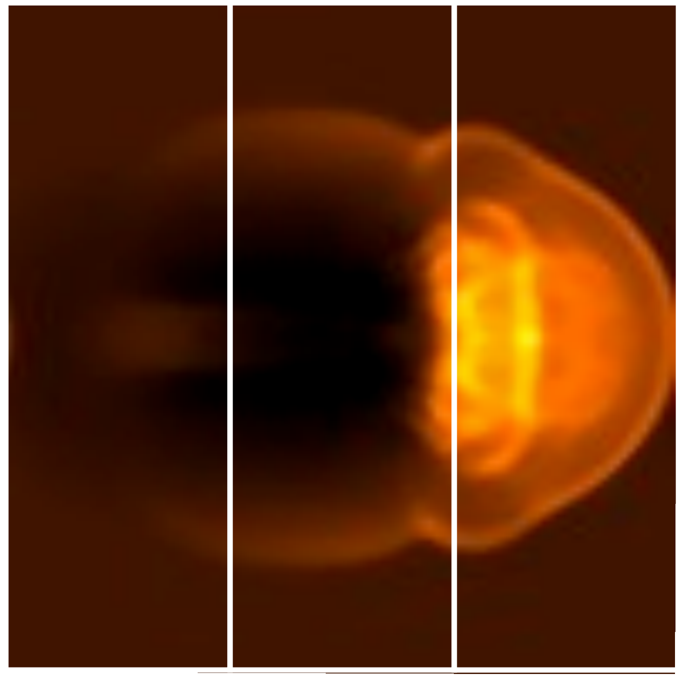
MPIing the code

- Domain decomposition



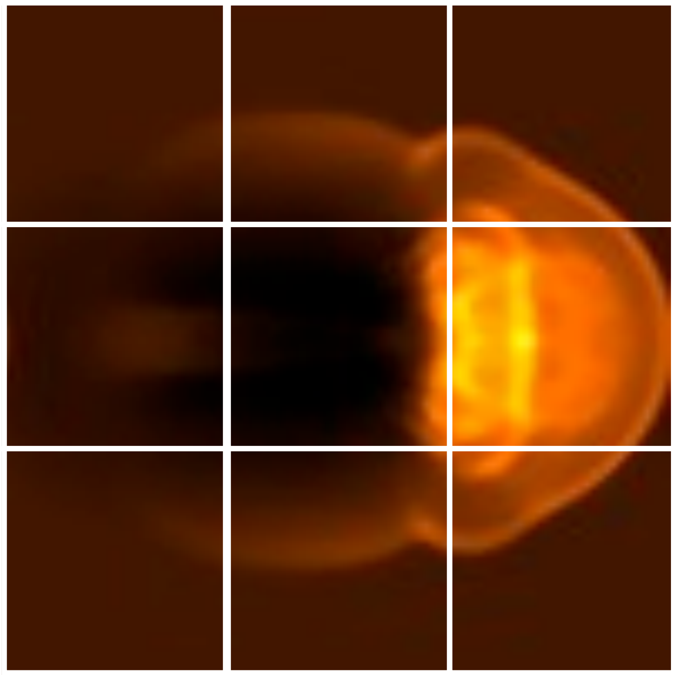
MPIing the code

- Domain decomposition
- For simplicity, for now we'll just implement decomposition in one direction, but we will design for full 2d decomposition



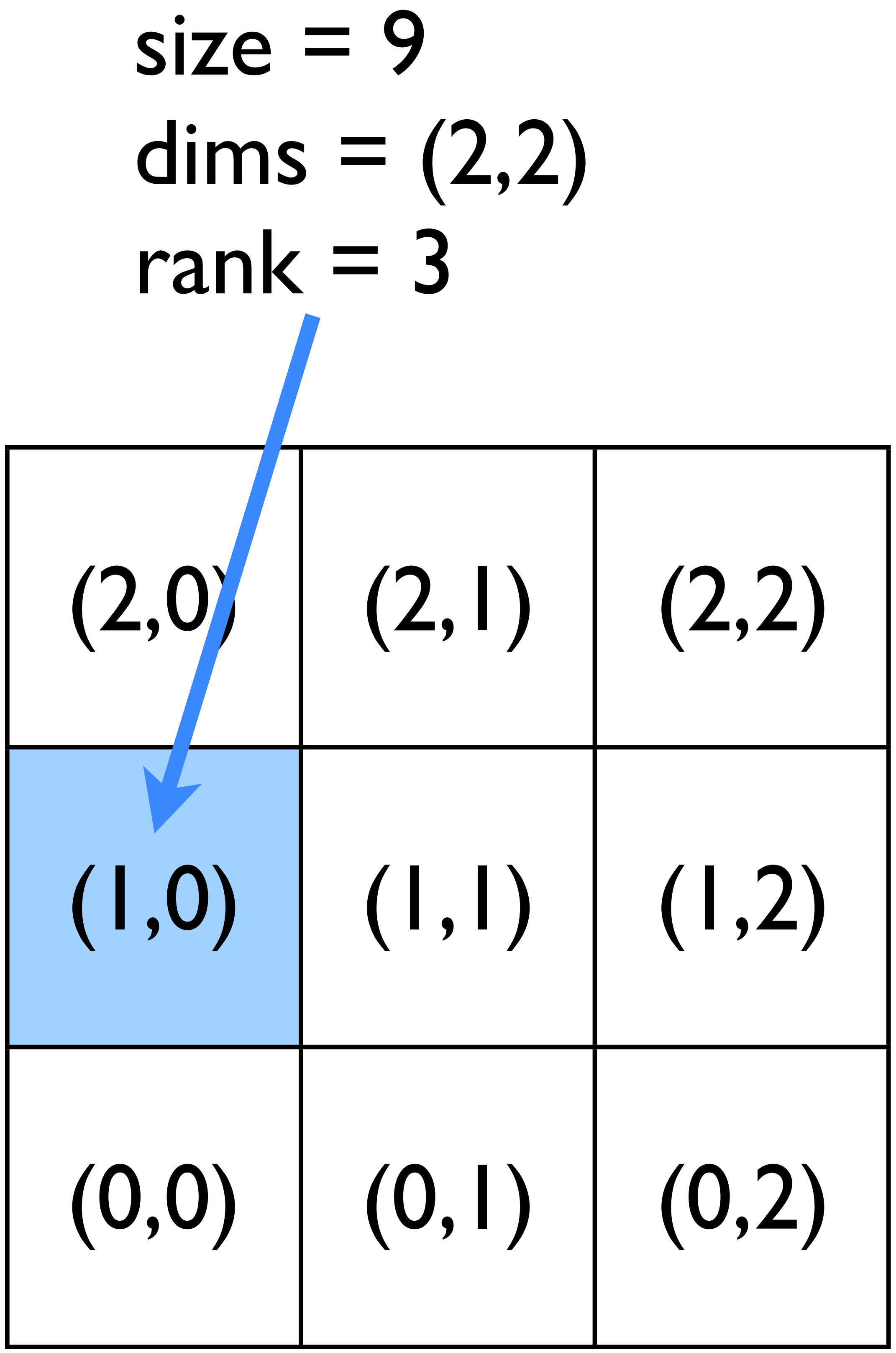
MPIing the code

- Domain decomposition
- We can do as with diffusion and figure out out neighbours by hand, but MPI has a better way...



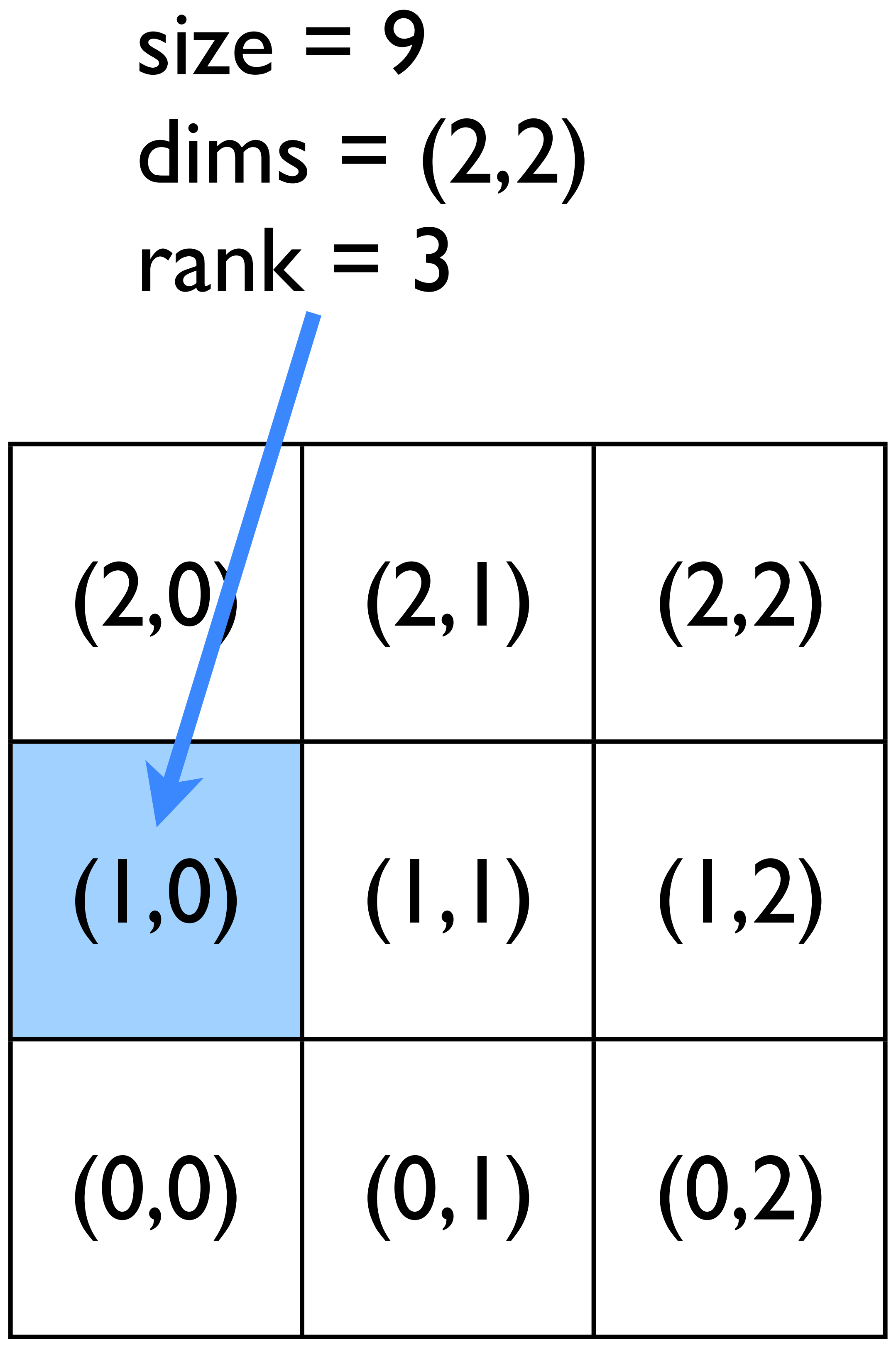
Create new communicator with new topology

- MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

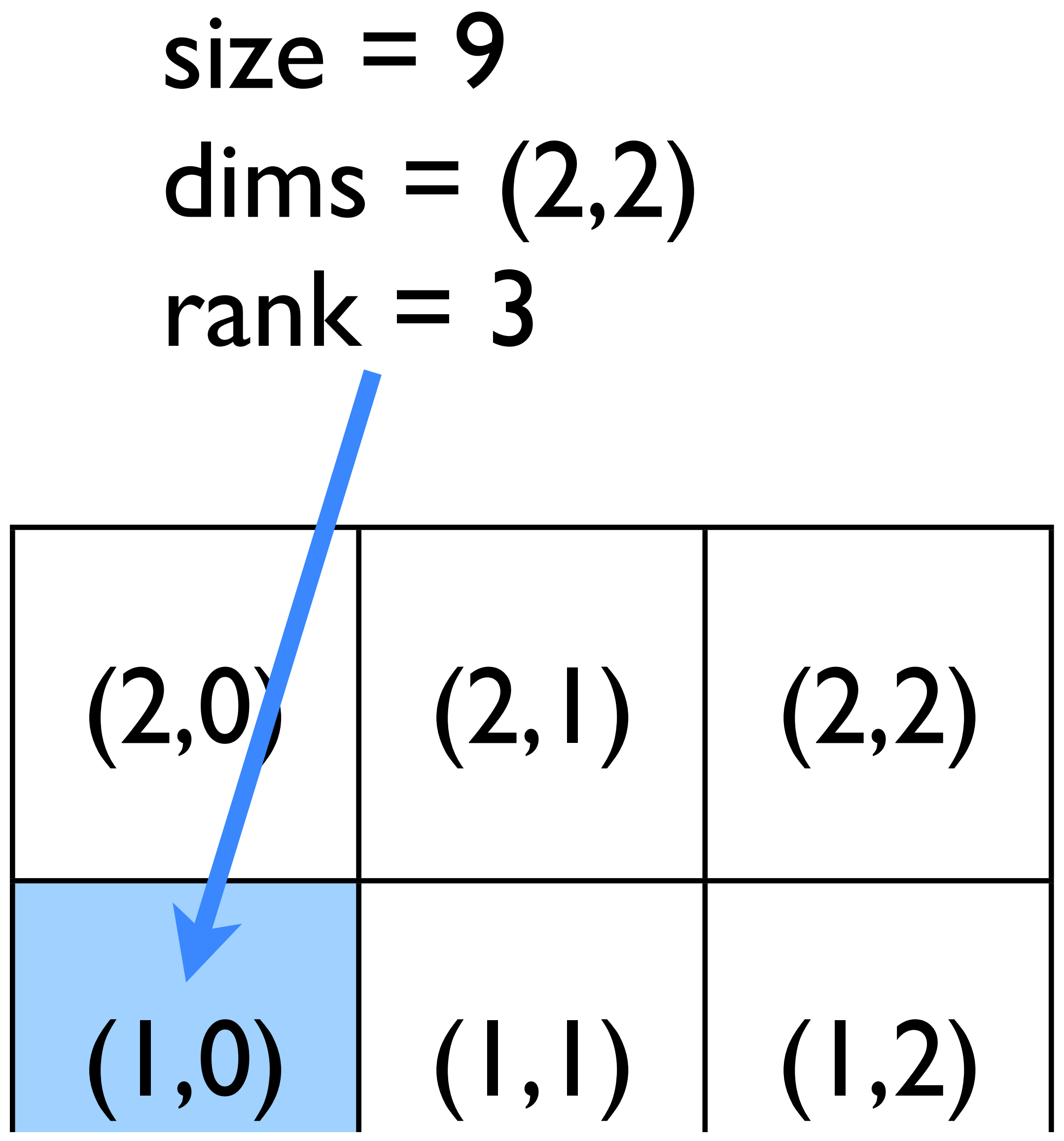


Create new communicator with new topology

- MPI_Cart_create (
  integer comm_old,
  integer ndims,
  integer [dims],
  logical [periods],
  integer reorder,
  integer comm_cart,
  integer ierr)



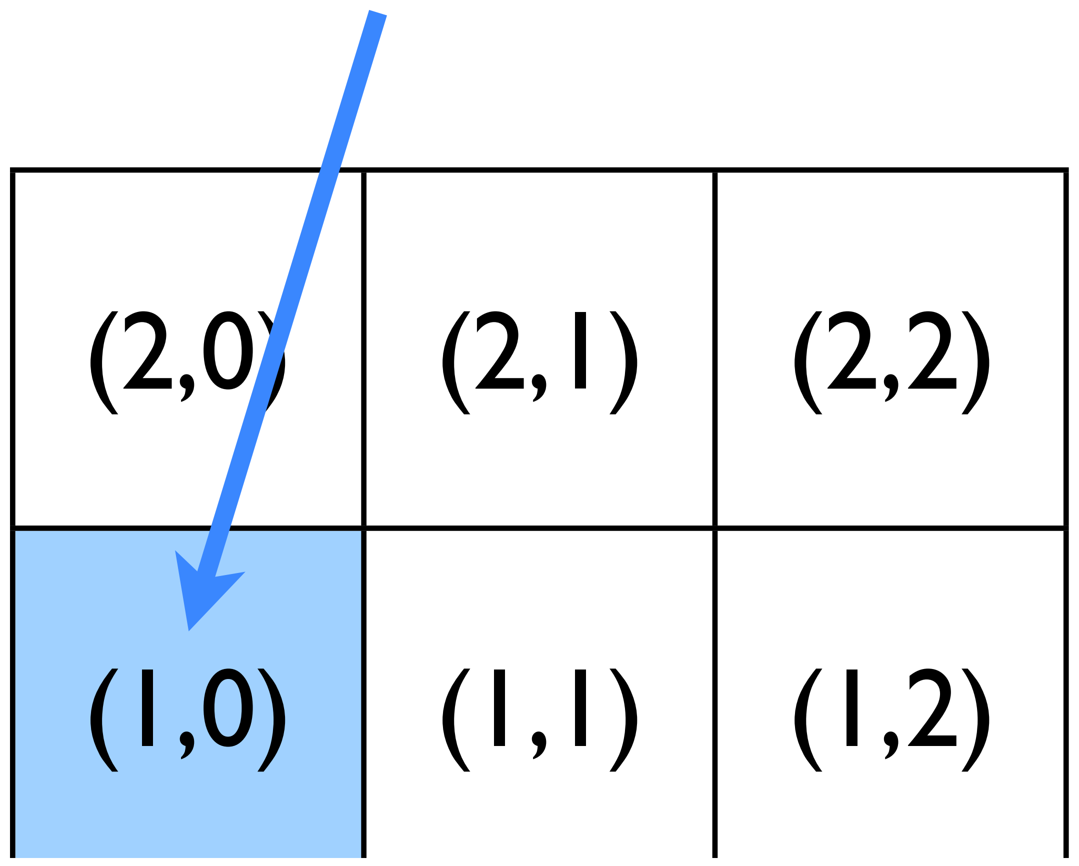
Create new communicator with new topology



```
C
ierr = MPI_Cart_shift(MPI_COMM new_comm, int dim,
       int shift, int *left, int *right)
ierr = MPI_Cart_coords(MPI_COMM new_comm, int rank,
       int ndims, int *gridcoords)
```

size = 9
dims = (2,2)
rank = 3

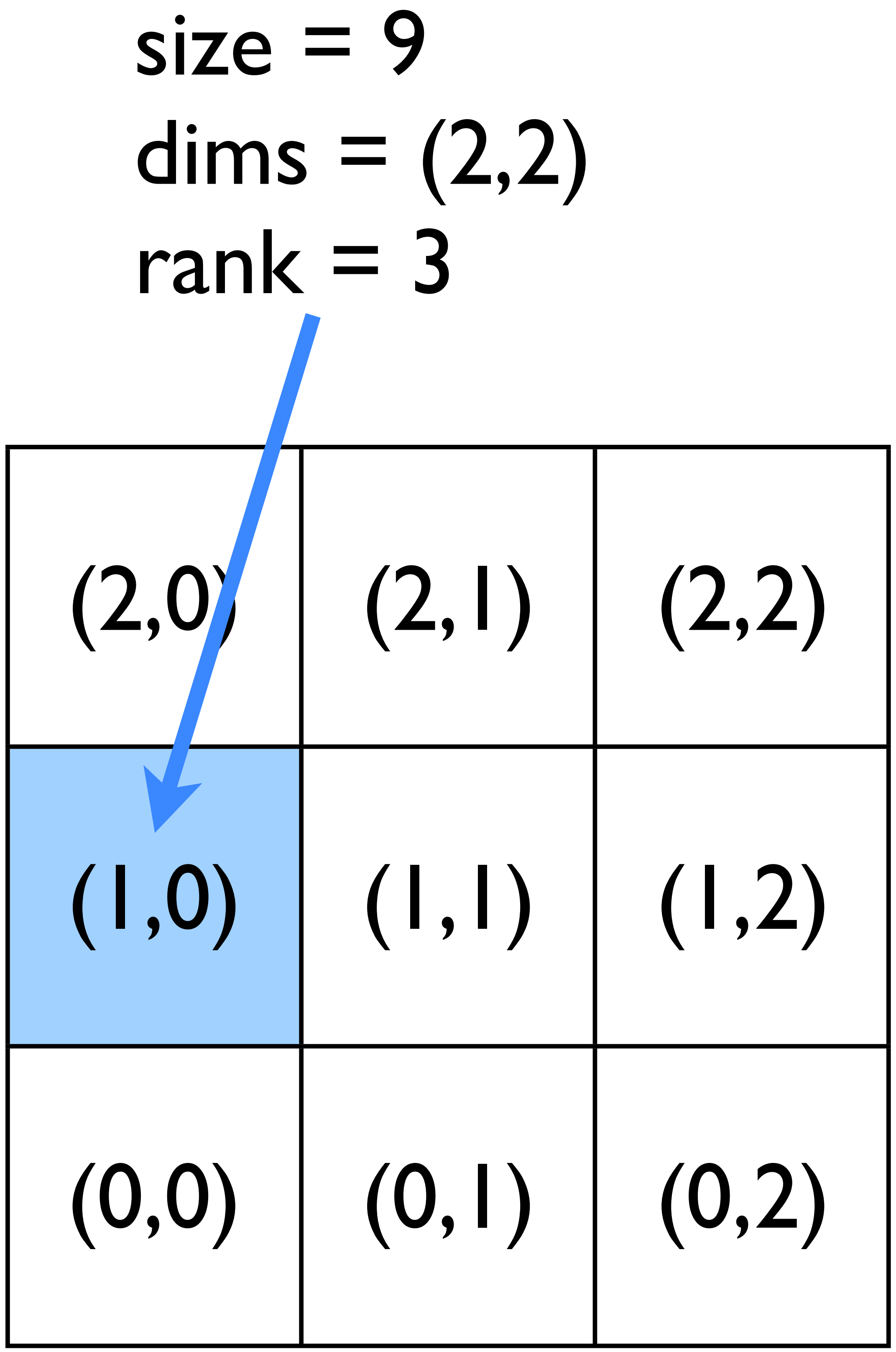
Create new communicator with new topology



```
FORTRAN
call MPI_Cart_shift(integer new_comm, dim, shift,
       left, right, ierr)
call MPI_Cart_coords(integer new_comm, rank,
       ndims, [gridcoords], ierr)
```

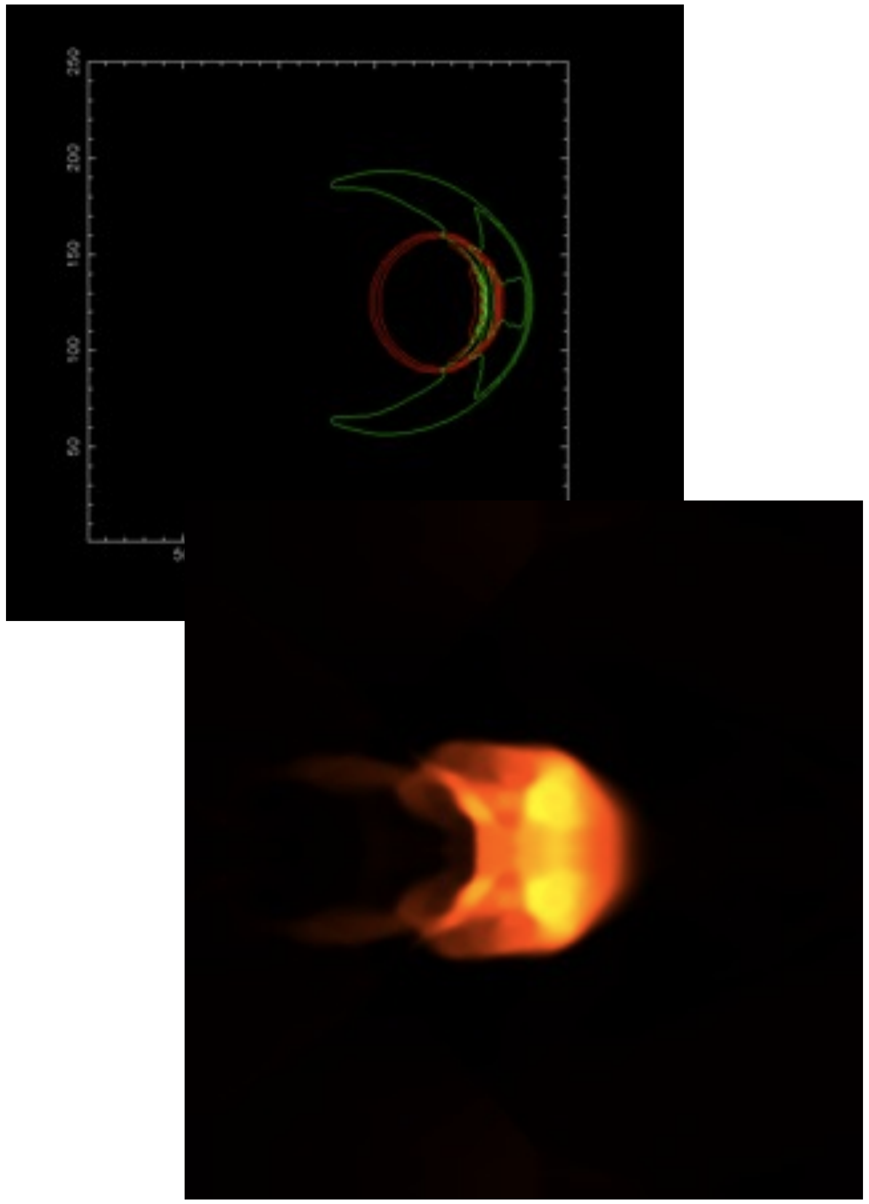
Let's try starting to do this together

- In a new directory:
- add mpi_init, _finalize, comm_size.
- mpi_cart_create
- rank on *new* communicator.
- neighbours
- Only do part of domain



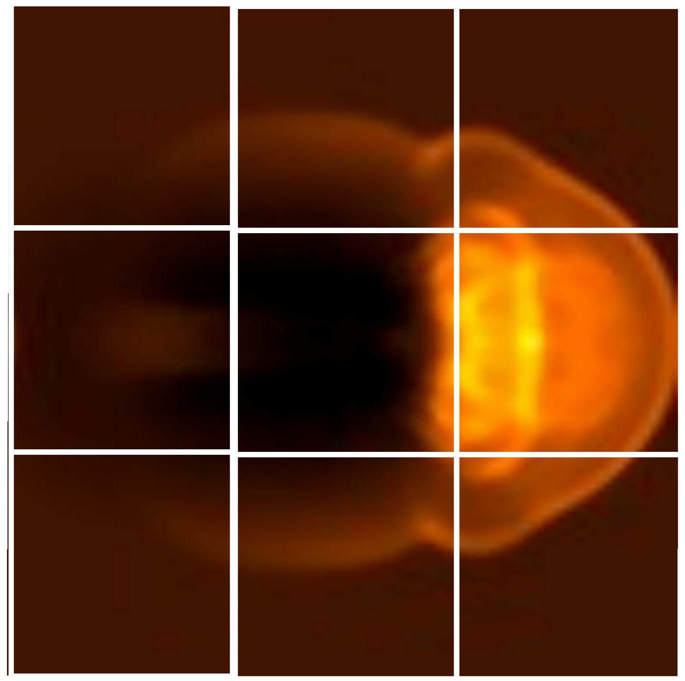
Next

- File IO - have each process write its own file so don't overwrite
- Coordinate min, max across processes for contours, images.
- Coordinate min in cfl routine.



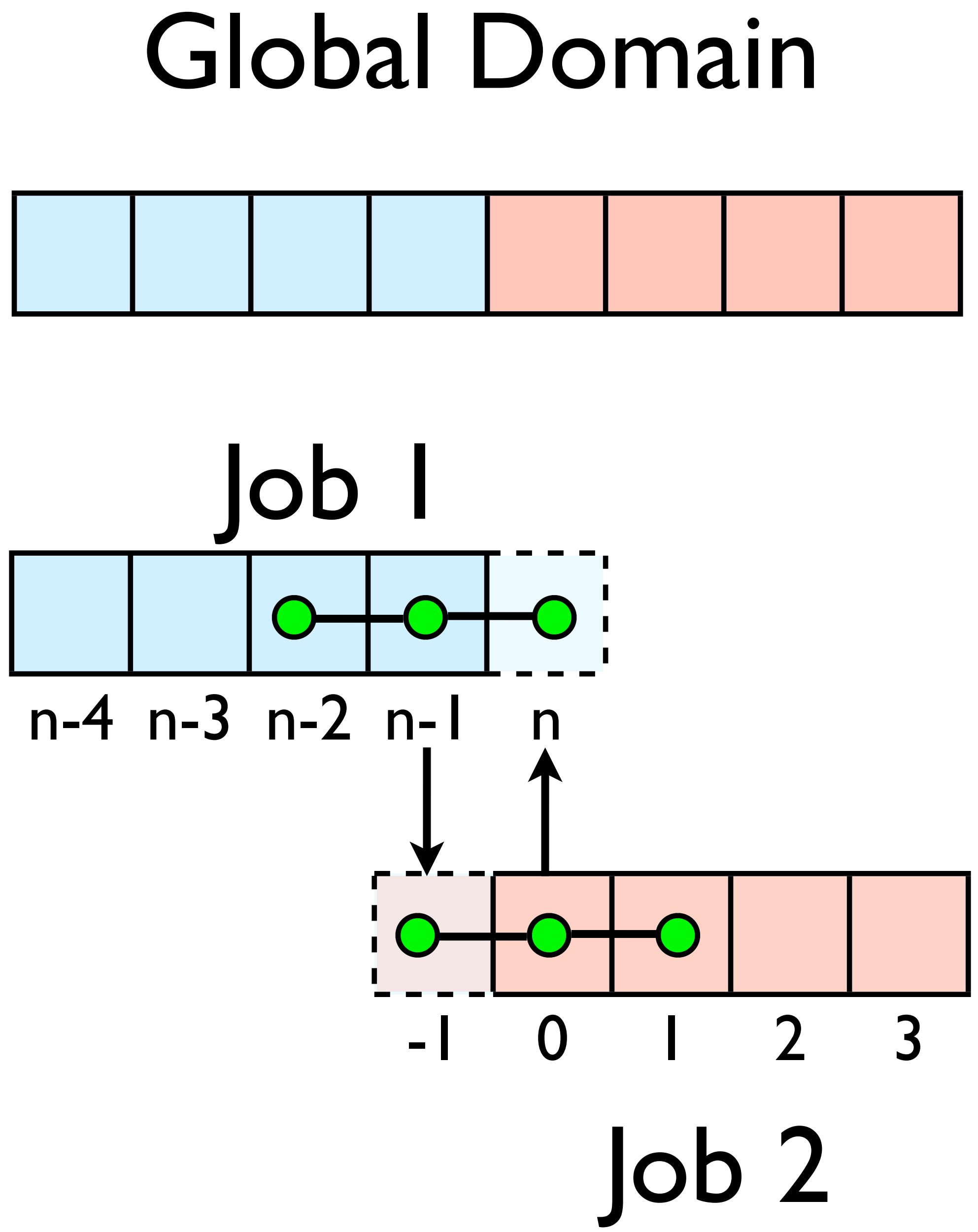
MPIing the code

- Domain decomposition
- Lots of data - ensures locality
- How are we going to handle getting non-local information across processors?



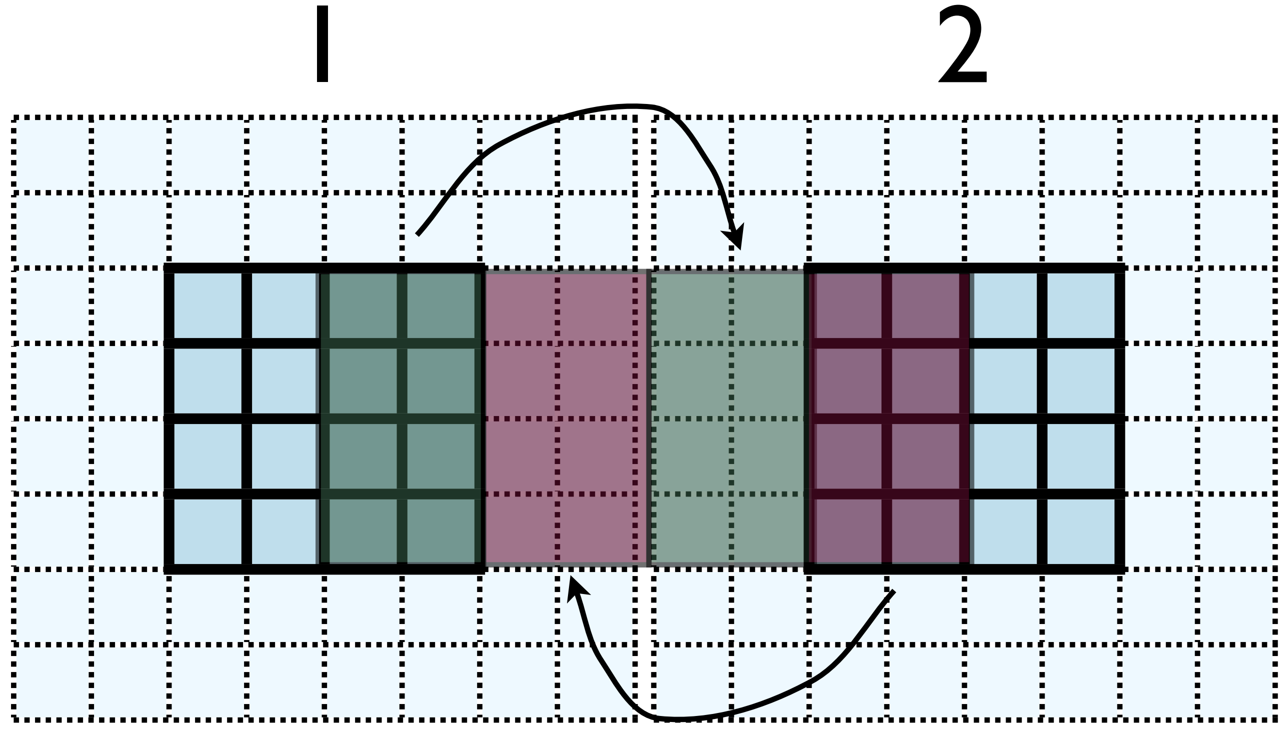
Guardcells

- Works for parallel decomposition!
- Job 1 needs info on Job 2s 0th zone, Job 2 needs info on Job 1s last zone
- Pad array with 'guardcells' and fill them with the info from the appropriate node by message passing or shared memory
- Hydro code: need guardcells 2 deep



Guard cell fill

- When we're doing boundary conditions.
- Swap guardcells with neighbour.



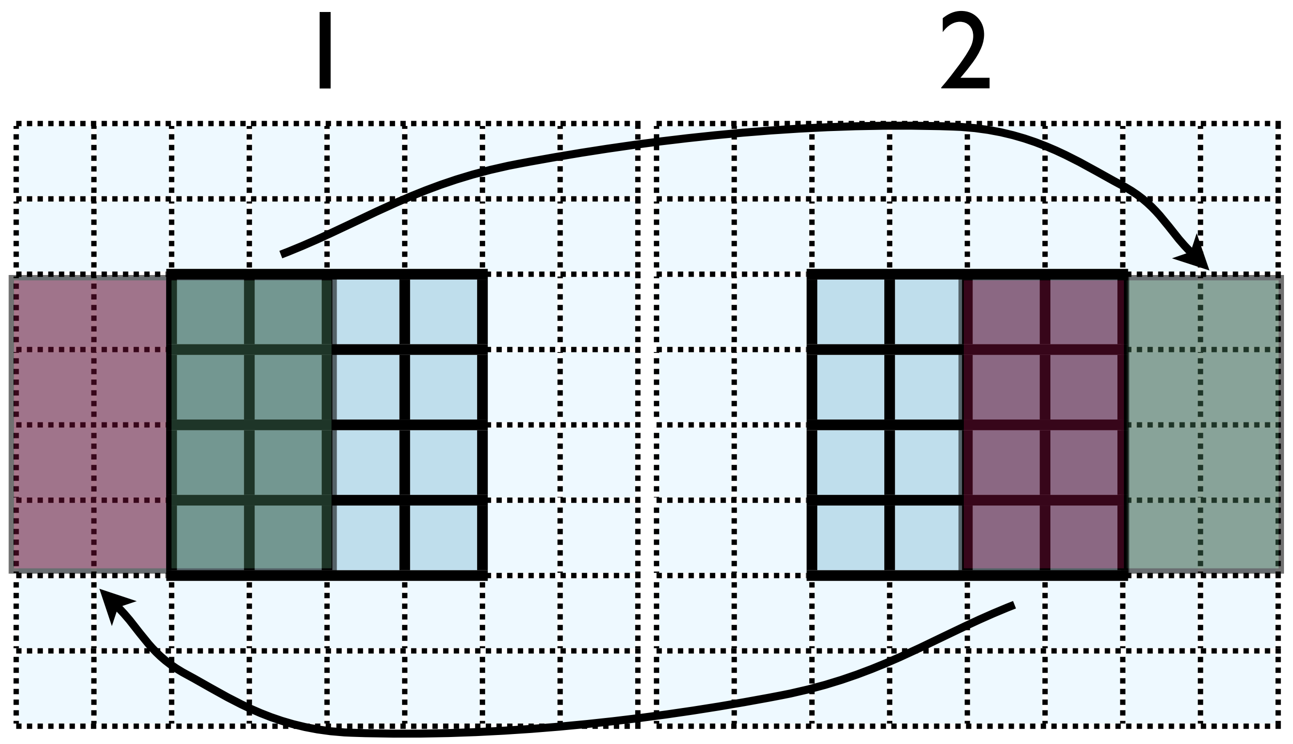
1: u(:, nx:nx+ng, ng:ny-ng)
→ 2: u(:,1:ng, ng:ny-ng)

2: u(:, ng+1:2*ng, ng:ny-ng)
→ 1: u(:, nx+ng+1:nx+2*ng, ng:ny-ng)

(ny-2*ng)*ng values to swap

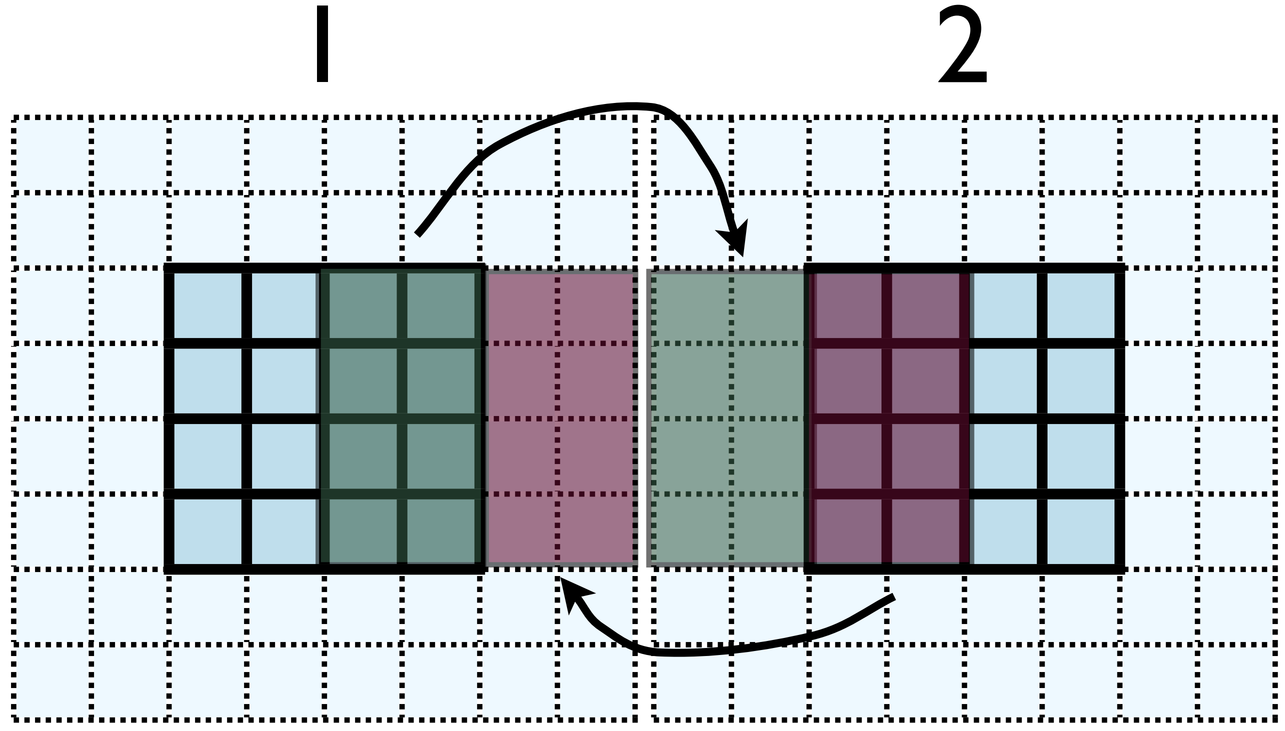
Cute way for Periodic BCs

- Actually make the decomposed mesh periodic;
- Make the far ends of the mesh neighbors
- Don't know the difference between that and any other neighboring grid
- Cart_create sets this up for us automatically upon request.



Implementing in MPI

- No different in principle than diffusion
- Just more values
- And more variables: dens, ener, imomx....
- Simplest way: copy all the variables into an NVARS*(ny-2*ng)*ng sized buffer



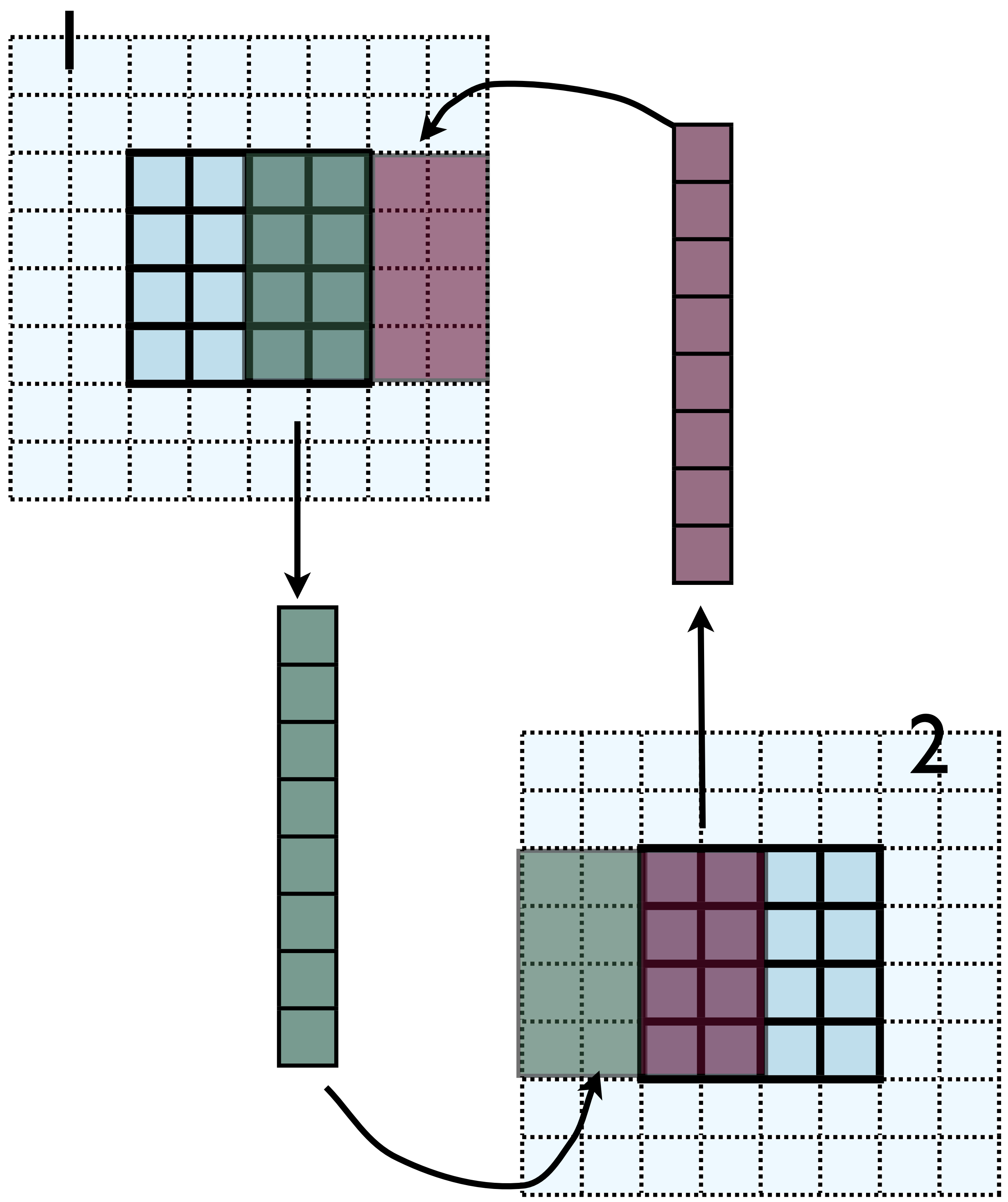
1: u(:, nx:nx+ng, ng:ny-ng)
→ 2: u(:,1:ng, ng:ny-ng)

2: u(:, ng+1:2*ng, ng:ny-ng)
→ 1: u(:, nx+ng+1:nx+2*ng, ng:ny-ng)

nvars*(ny-2*ng)*ng values to swap

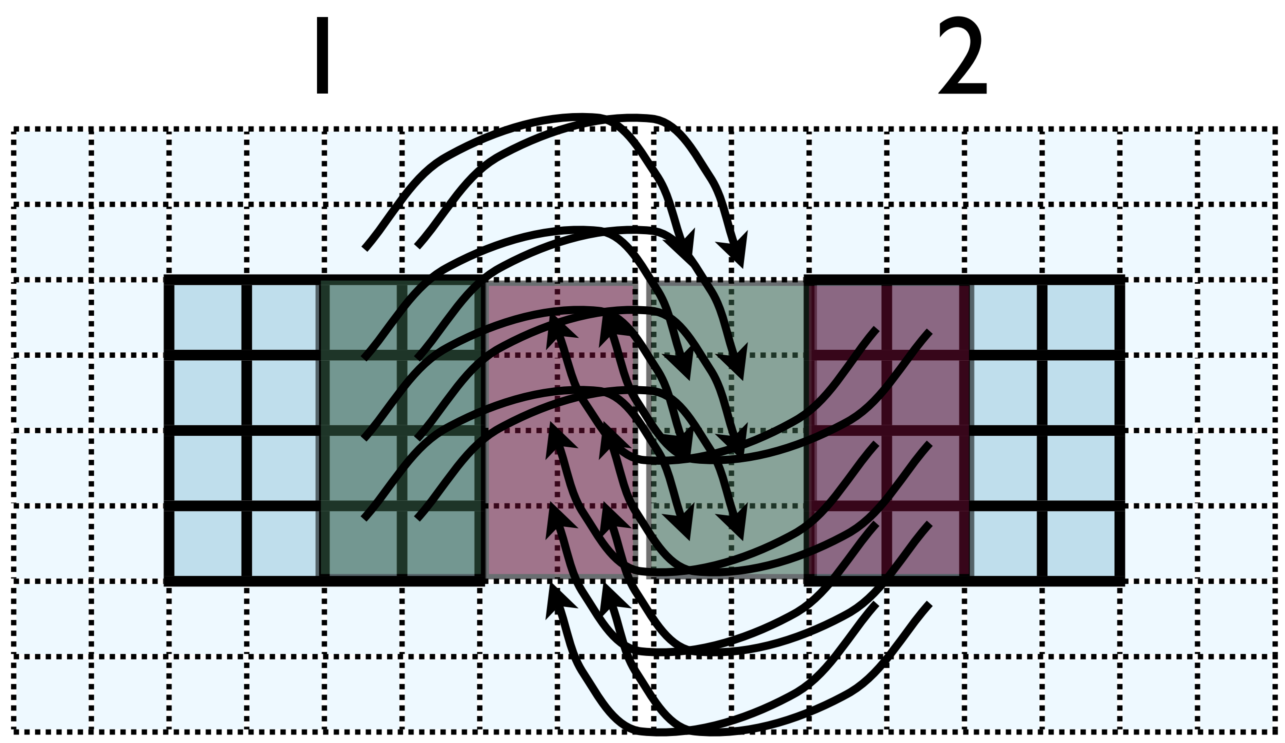
Implementing in MPI

- No different in principle than diffusion
- Just more values
- And more variables: dens, ener, temp....
- Simplest way: copy all the variables into an NVARS*(ny-2*ng)*ng sized buffer



Implementing in MPI

- Even simpler way:
- Loop over values, sending each one, rather than copying into buffer.
- NVARS*nguard*(ny-2*nguard) latency hit.
- Would completely dominate communications cost.

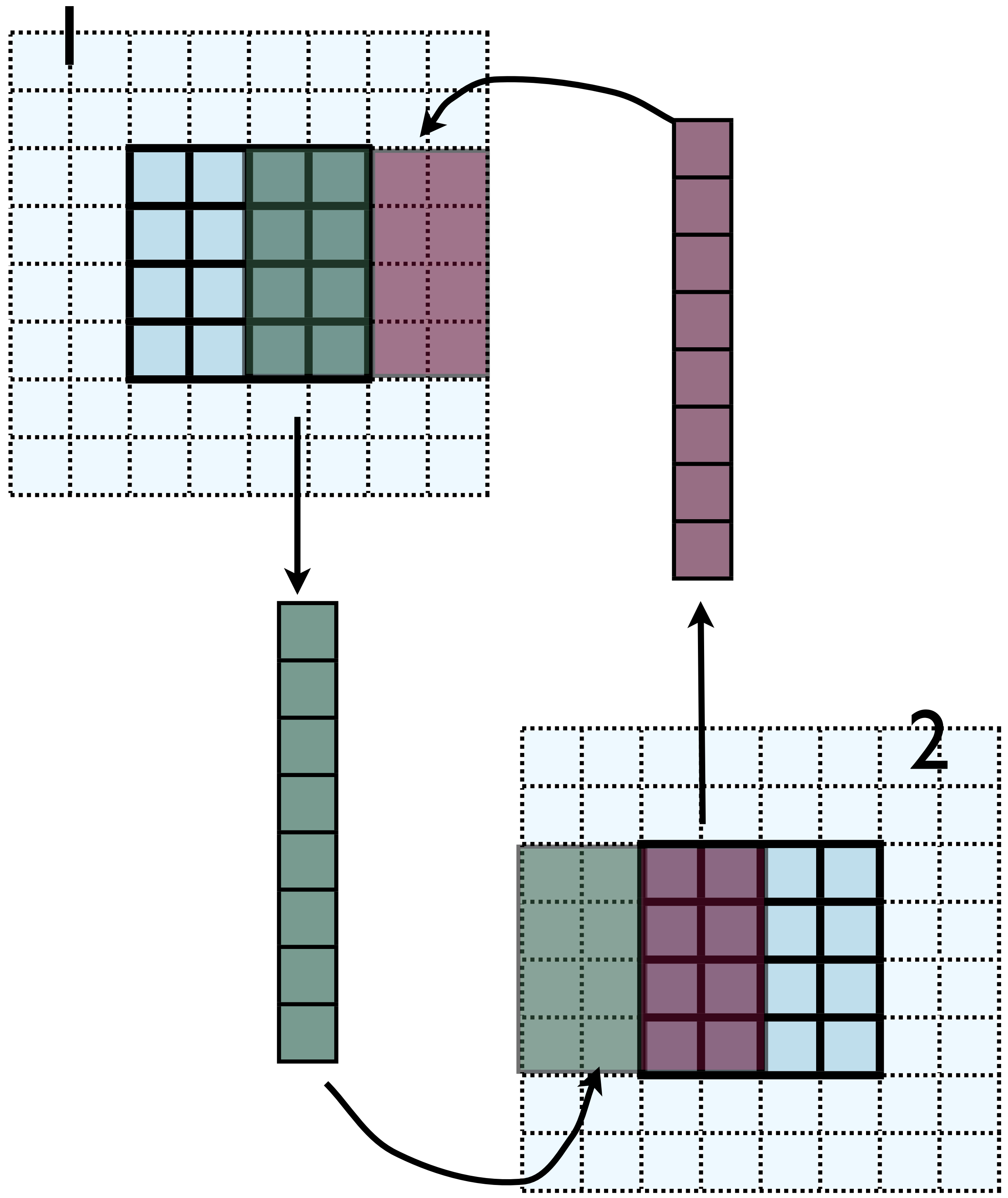


Implementing in MPI

- Let's do this together
- solver.f90; copy periodicBCs to gcBufferBCs
- When do we call this in timestep?

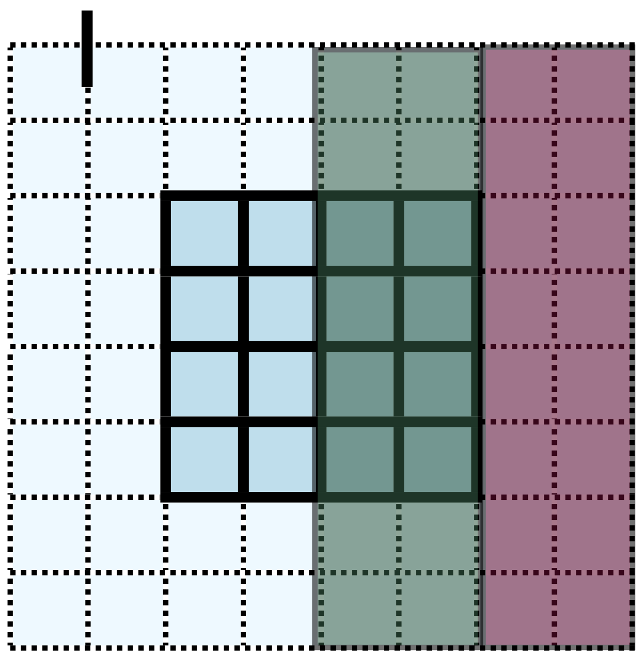
Implementing in MPI

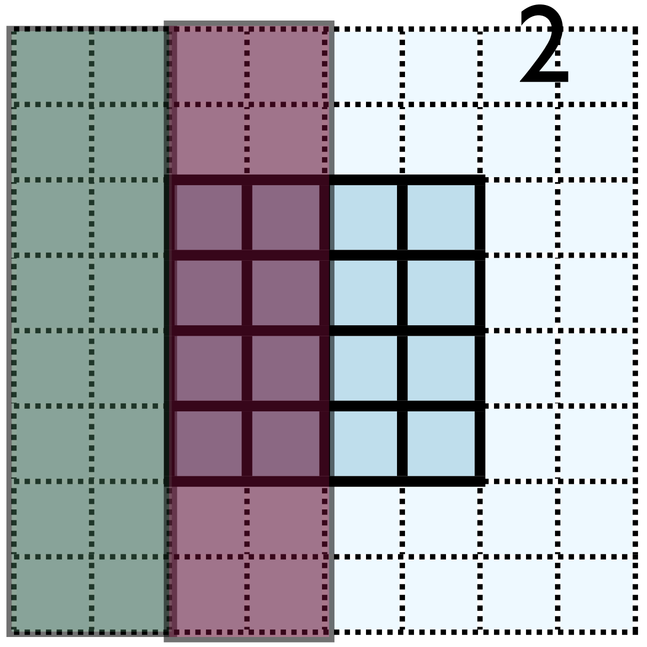
- This approach is simple, but introduces extraneous copies
- Memory bandwidth is already a bottleneck for these codes
- It would be nice to just point at the start of the guardcell data and have MPI read it from there.



Implementing in MPI

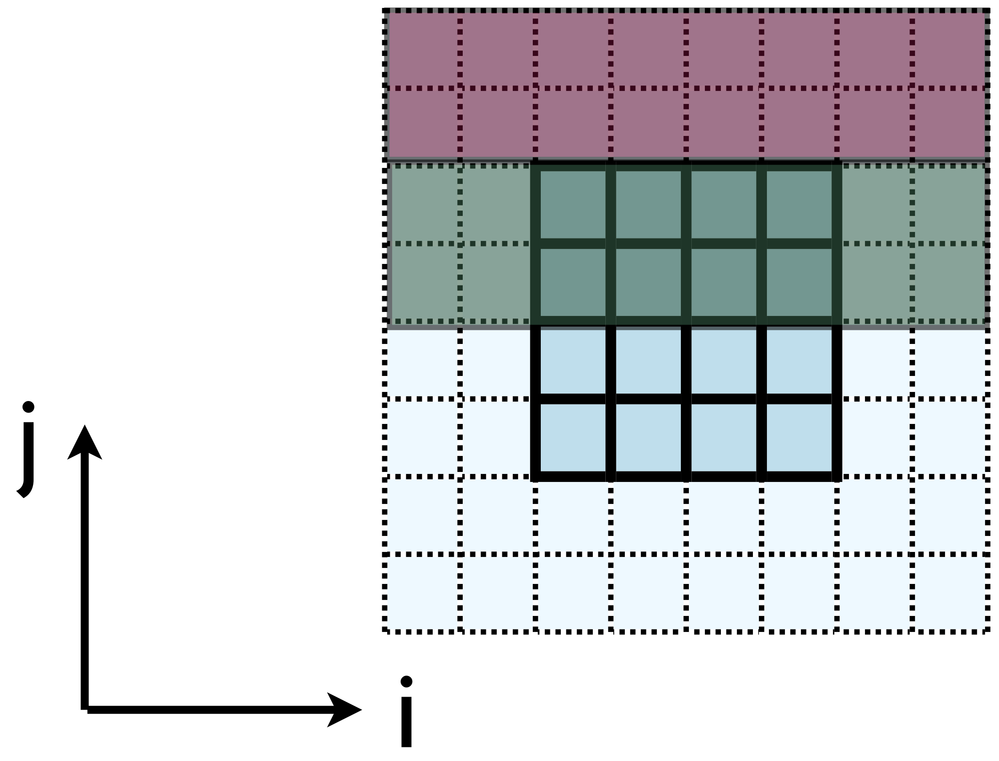
- Let me make one simplification for now; copy whole stripes
- This isn't necessary, but will make stuff simpler at first
- Only a cost of $2xNg^2 = 8$ extra cells (small fraction of ~200-2000 that would normally be copied)

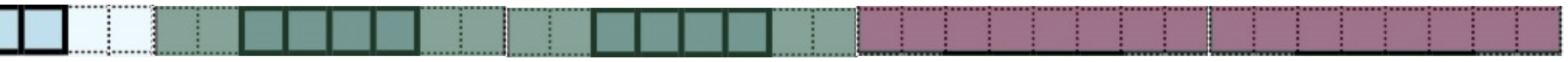




Implementing in MPI

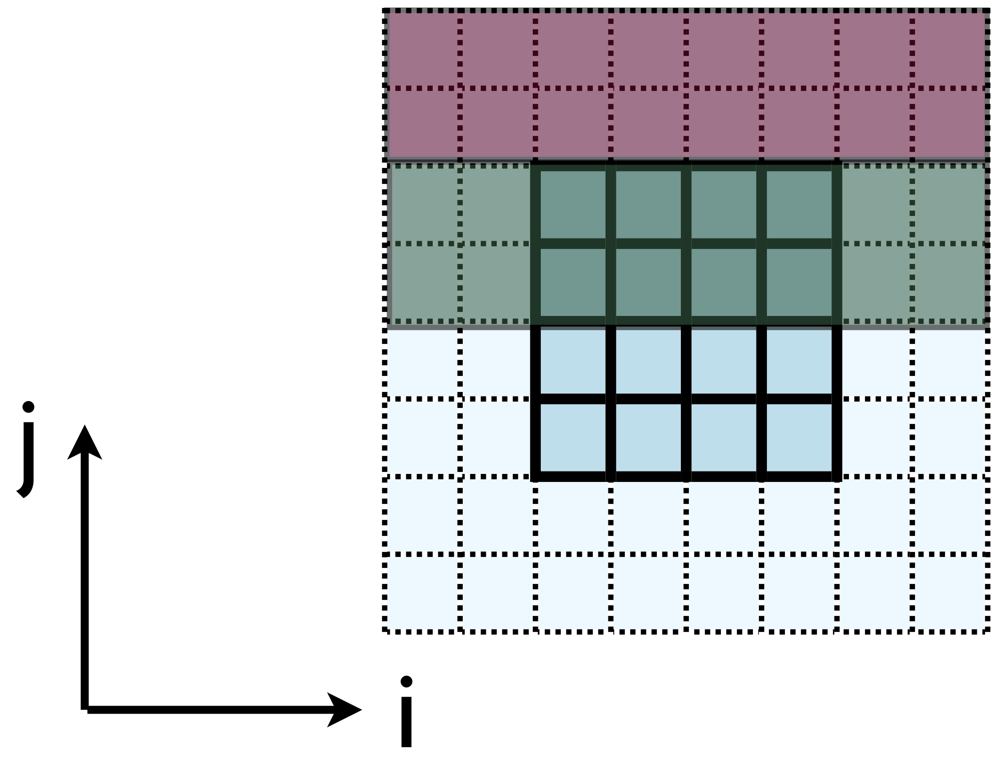
- Recall how 2d memory is laid out
- y-direction guardcells contiguous



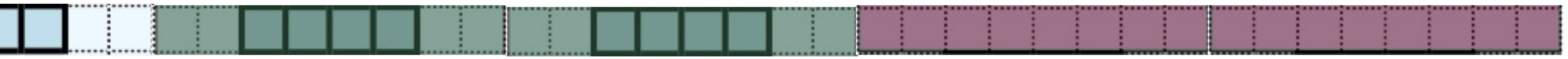


Implementing in MPI

- Can send in one go:

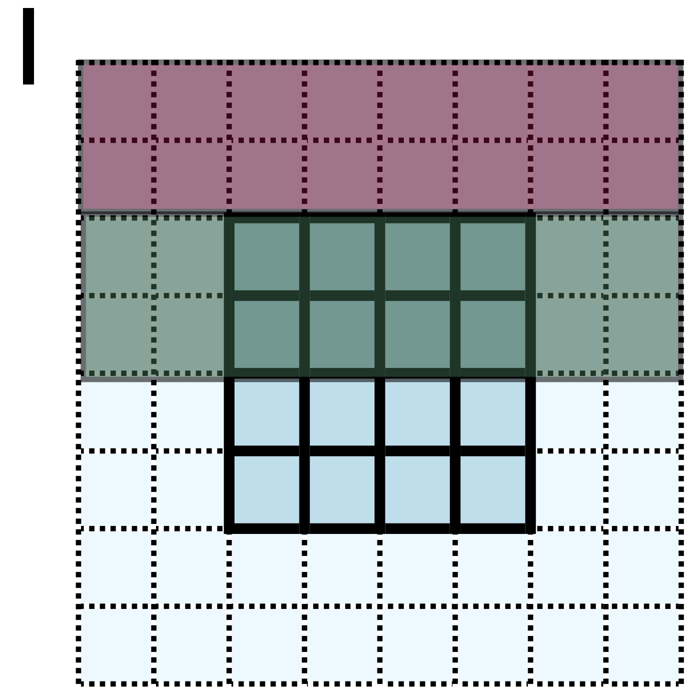


```
call MPI_Send(u(1,1,ny), nvars*nguard*ny, MPI_REAL, ....)
ierr = MPI_Send(&(u[ny][0][0]), nvars*nguard*ny, MPI_FLOAT, ....)
```



Implementing in MPI

- Creating MPI Data types.
- MPI_Type_contiguous: simplest case. Lets you build a string of some other type.

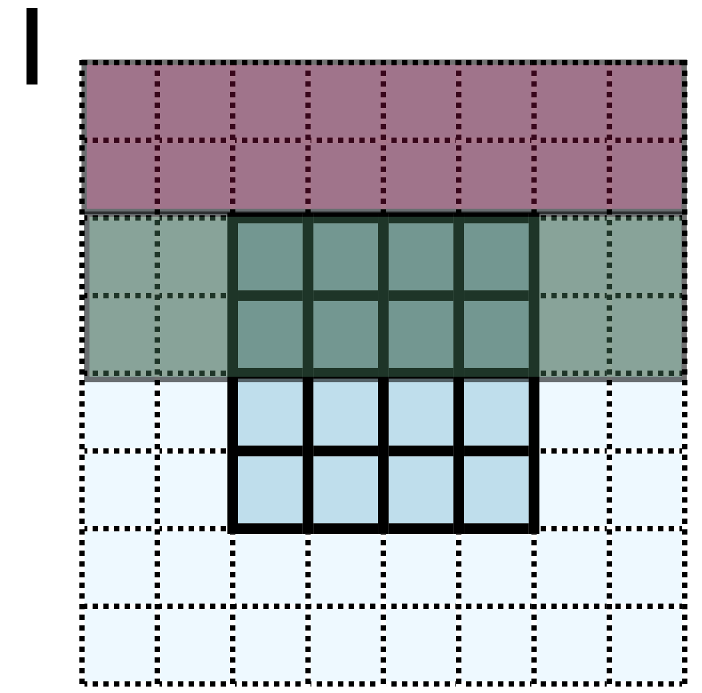


Count OldType &NewType

```
MPI_Datatype ybctype;

ierr = MPI_Type_contiguous(nvals*nguard*(ny), MPI_REAL, &ybctype);
ierr = MPI_Type_commit(&ybctype);

MPI_Send(&(u[ny][0][0]), 1, ybctype, ....)

ierr = MPI_Type_free(&ybctype);
```

Implementing in MPI

- Creating MPI Data types.
- MPI_Type_contiguous: simplest case. Lets you build a string of some other type.

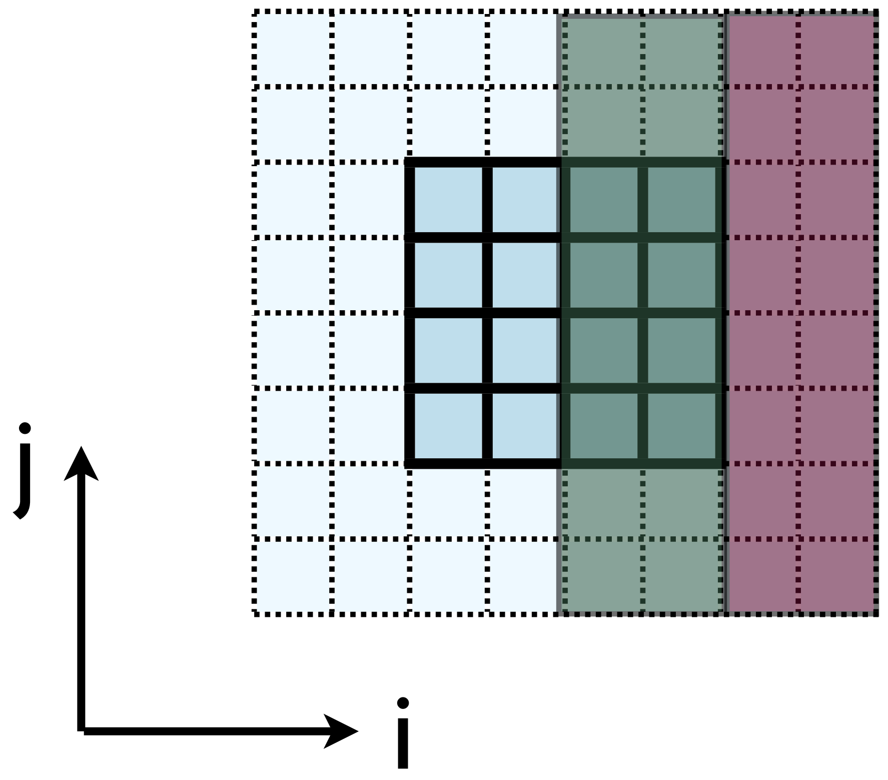


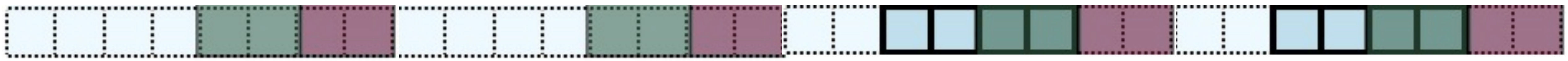
Count OldType NewType

```
    integer :: ybctype

    call MPI_Type_contiguous(nvals*nguard*(ny), MPI_REAL, ybctype, ierr)
    call MPI_Type_commit(ybctype, ierr)

    MPI_Send(u(1,1,ny), 1, ybctype, ....)

    call MPI_Type_free(ybctype, ierr)
```

Implementing in MPI

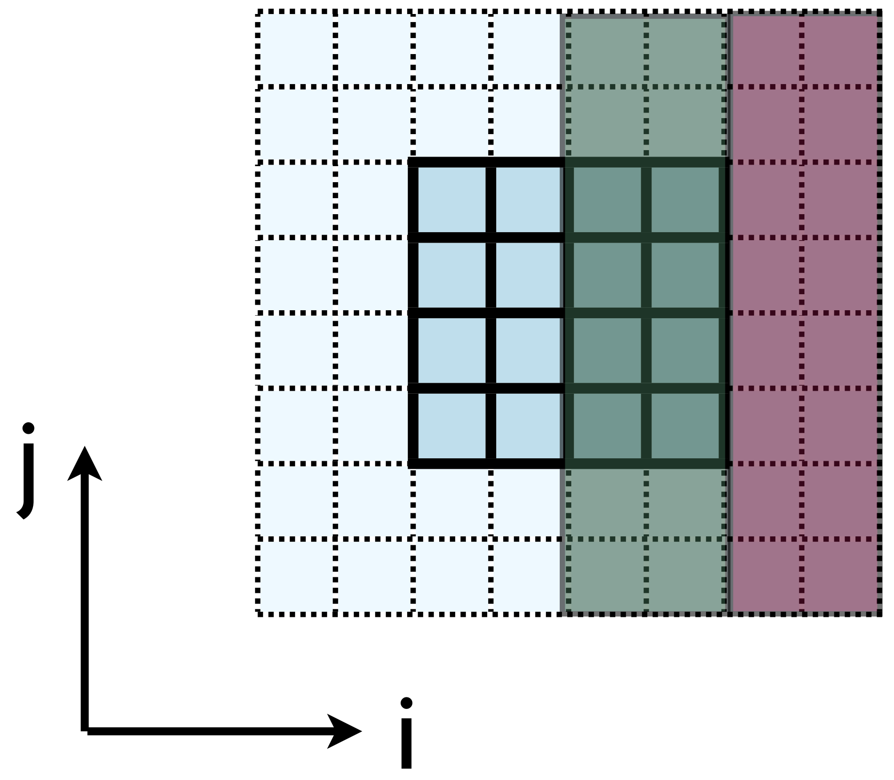
- Recall how 2d memory is laid out
- x gcs or boundary values *not* contiguous
- How do we do something like this for the x-direction?

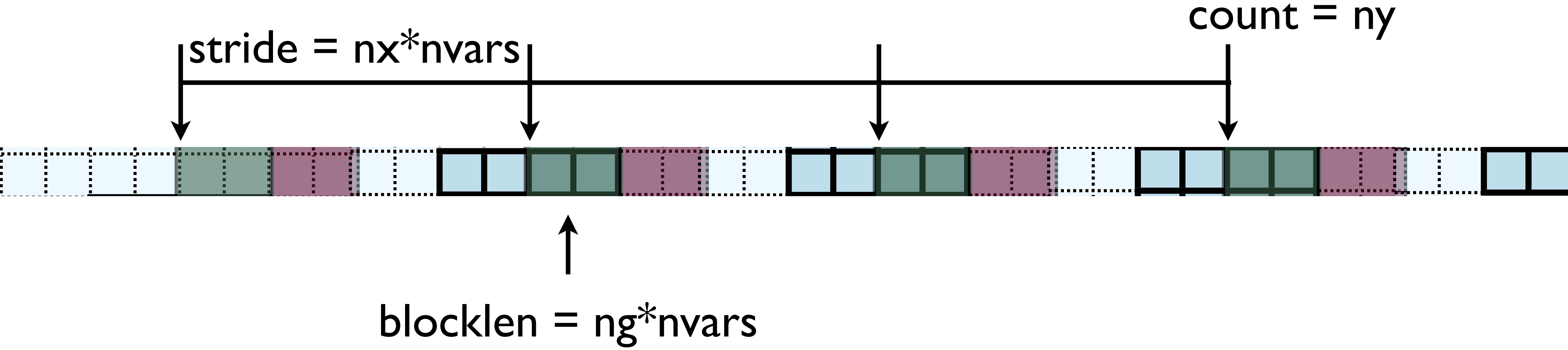




Implementing in MPI

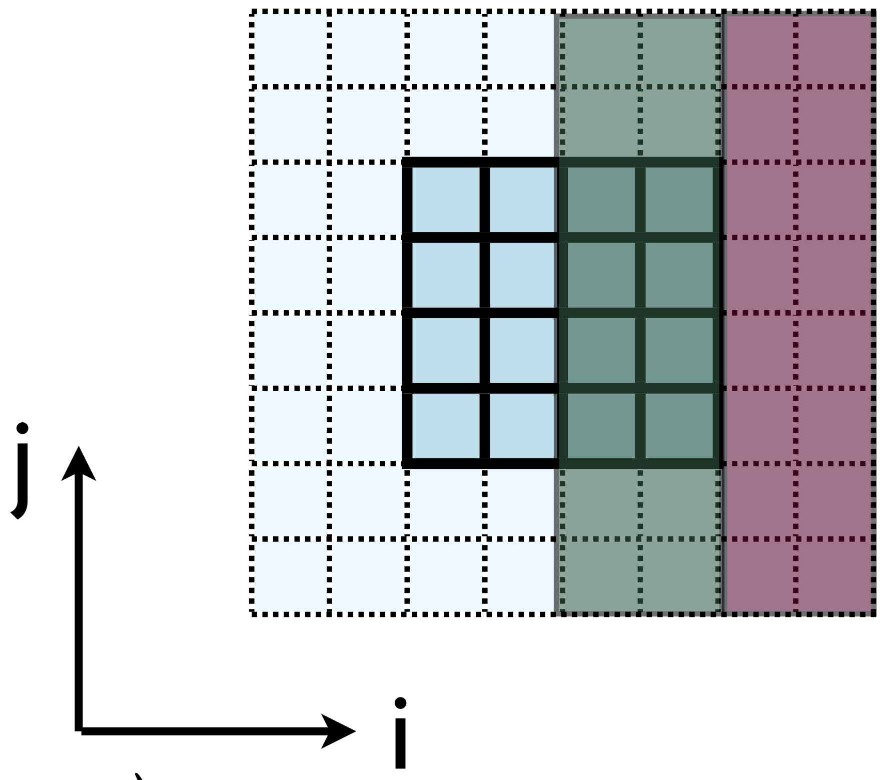
```
int MPI_Type_vector(
        int count,
        int blocklen,
        int stride,
        MPI_Datatype old_type,
        MPI_Datatype *newtype );
```

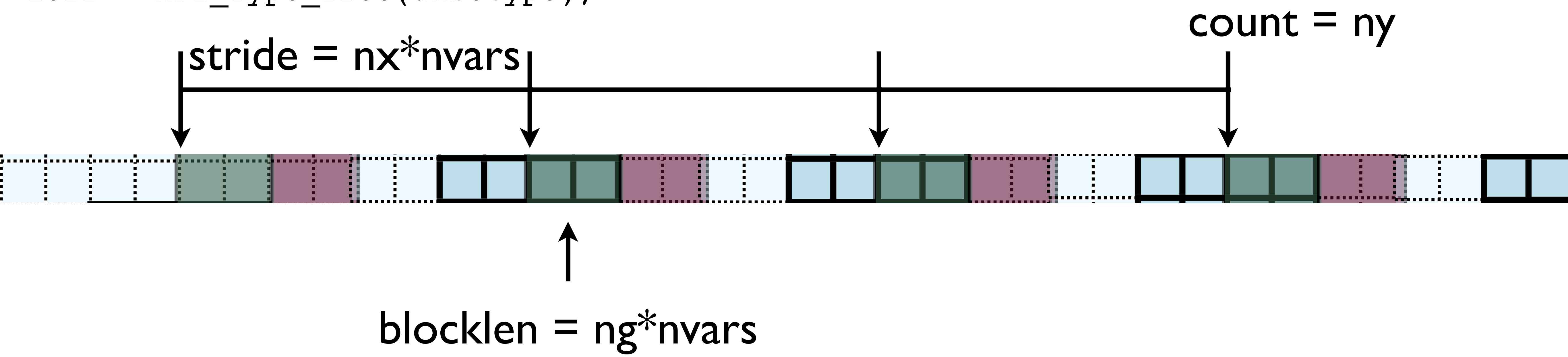




Implementing in MPI

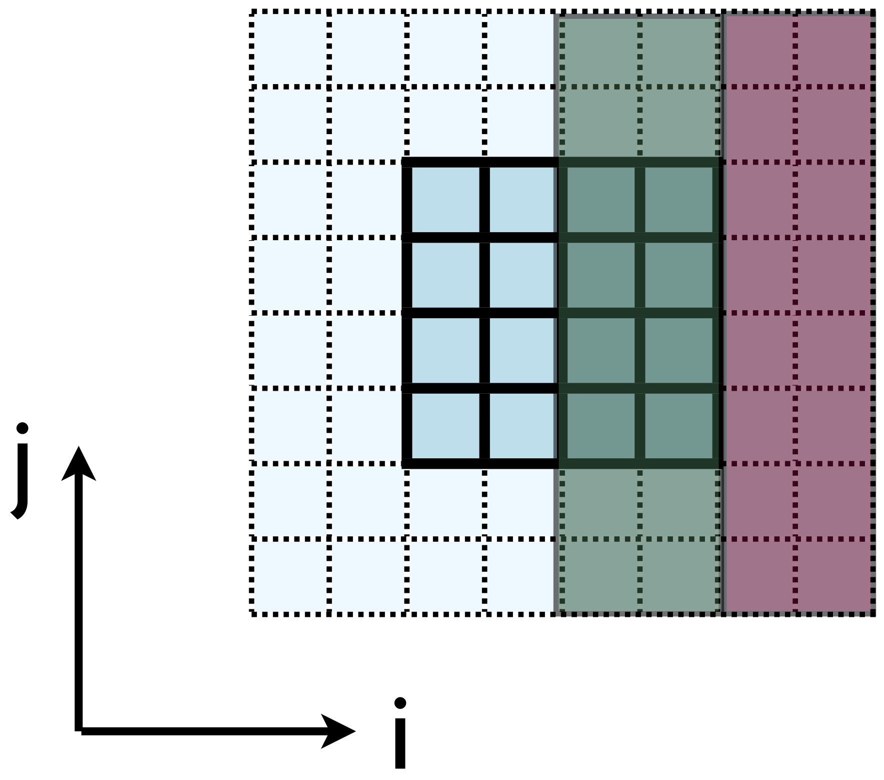
```
ierr = MPI_Type_vector(ny, nguard*nvars,
         nx*nvars, MPI_FLOAT, &xbctype);

ierr = MPI_Type_commit(&xbctype);

ierr = MPI_Send(&(u[0][nx][0]), 1, xbctype, ....)

ierr = MPI_Type_free(&xbctype);
```

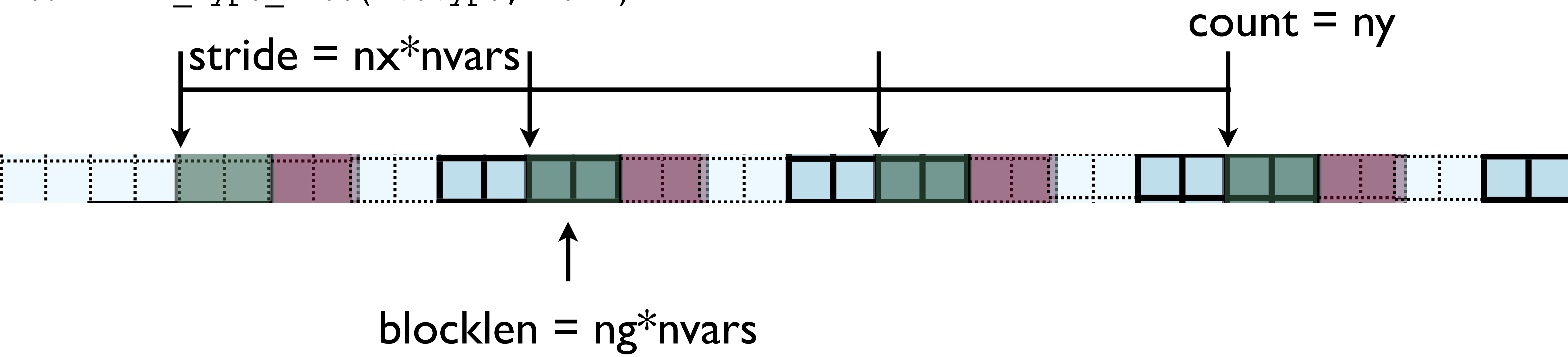




Implementing in MPI

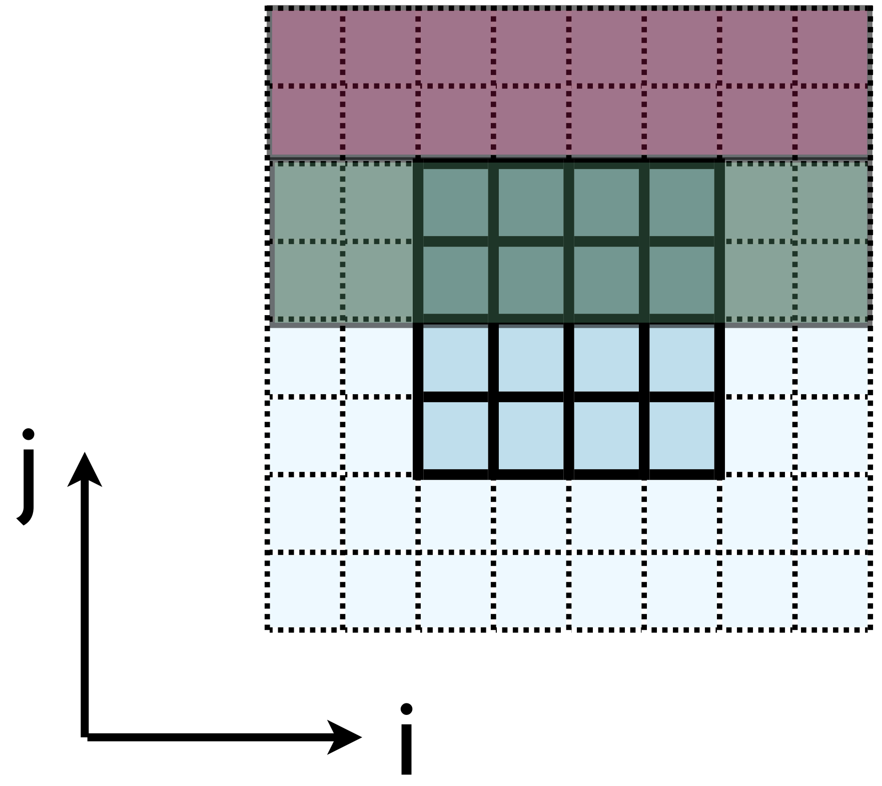
```
call MPI_Type_vector(ny, nguard*nvars,
        nx*nvars, MPI_REAL, xbctype, ierr)

call MPI_Type_commit(xbctype, ierr)

call MPI_Send(u(1,nx,1), 1, ybctype, ....)

call MPI_Type_free(xbctype, ierr)
```

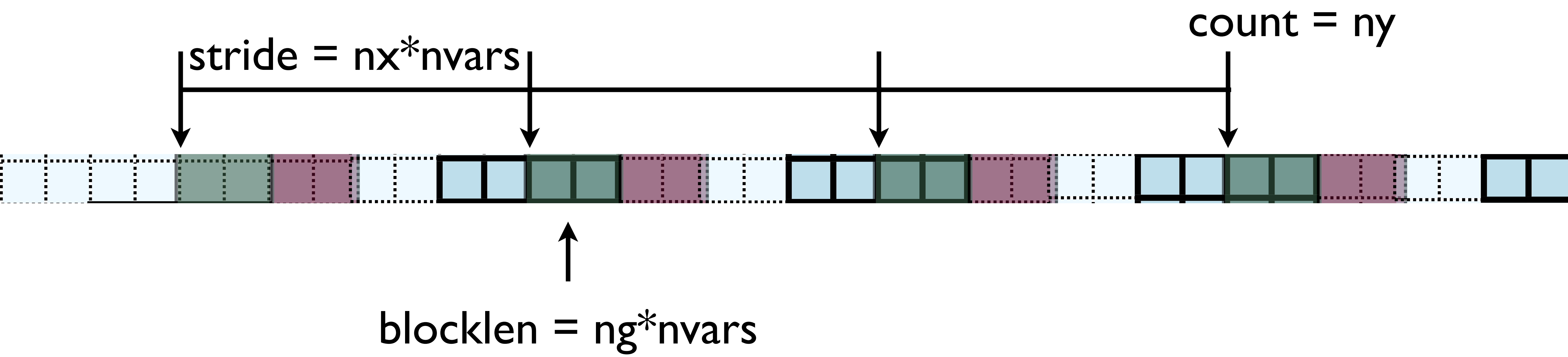




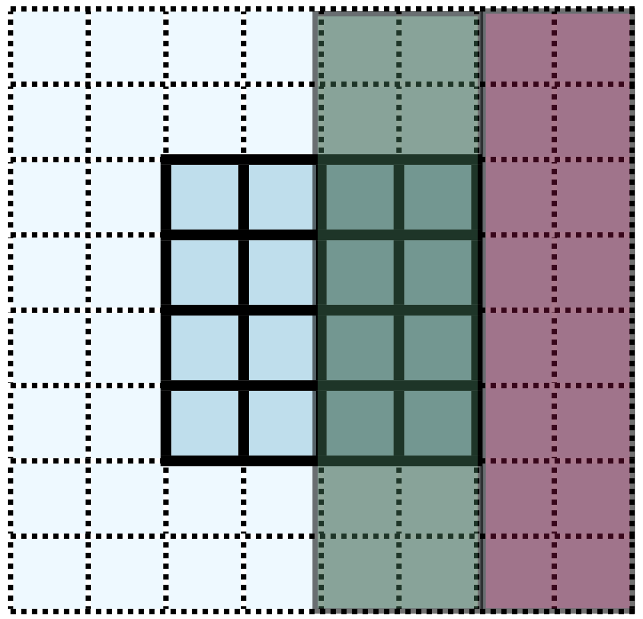
Implementing in MPI

- Check: total amount of data = blocklen*count = ny*ng*nvars
- Skipped over stride*count = nx*ny*nvars



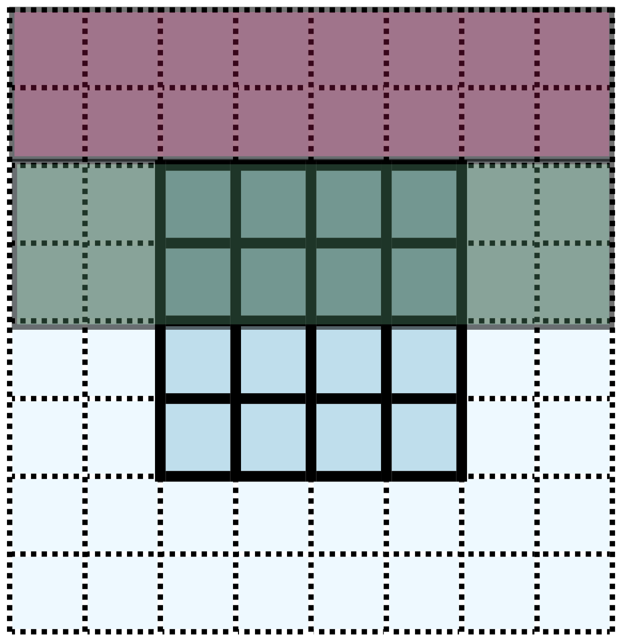


Implementing in MPI



- Hands-On: Implement X guardcell filling with types.
- Copy gcBufferBC to gcTypeBC, implement.
- For now, create/free type each cycle through; ideally, we'd create/free these once.

In MPI, there's always more than one way..

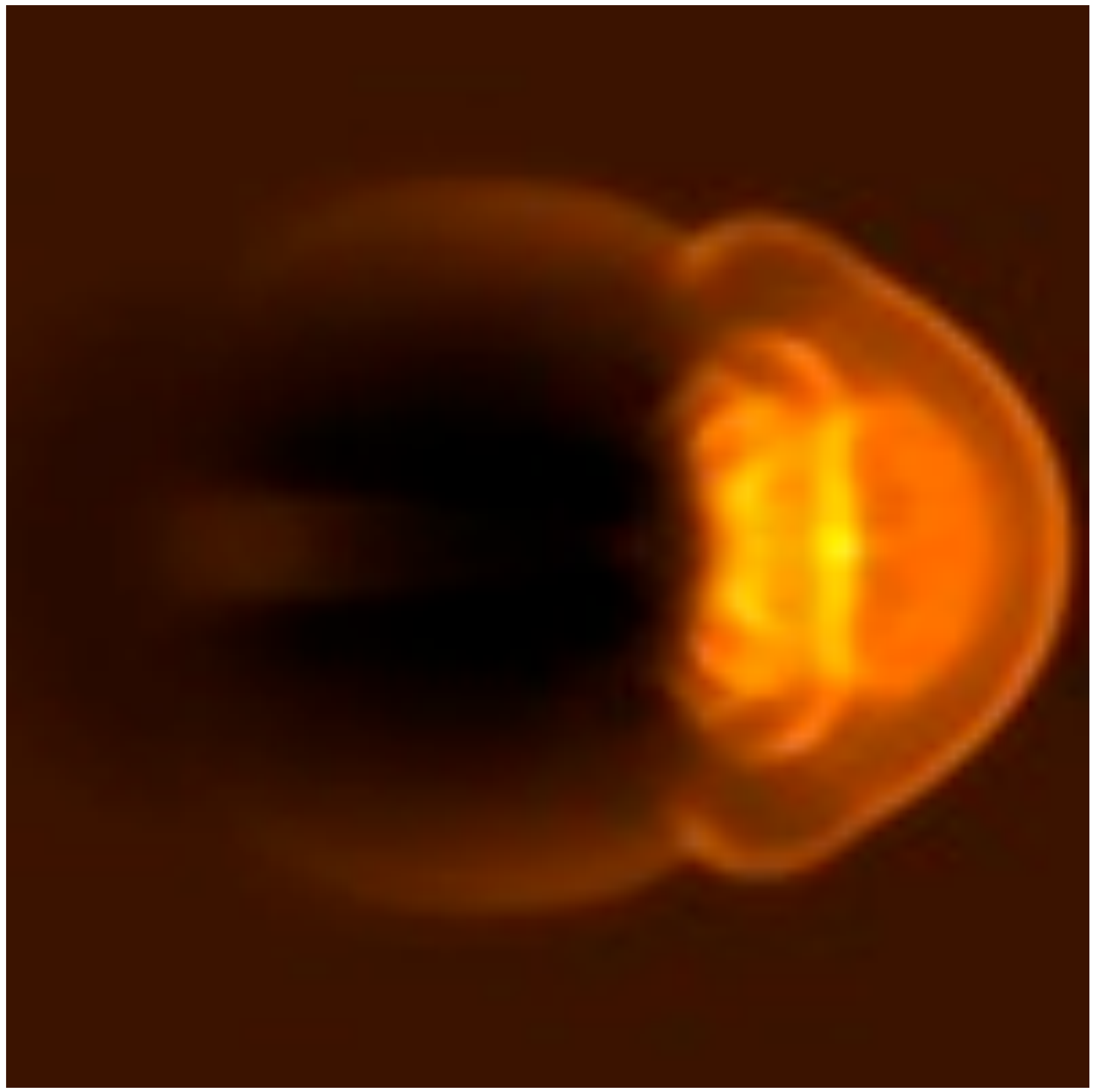


- MPI_Type_create_subarray ; piece of a multi-dimensional array.
- *Much* more convenient for higher-dimensional arrays
- (Otherwise, need vectors of vectors of vectors...)

```
int MPI_Type_create_subarray(
      int ndims, int *array_of_sizes,
      int *array_of_subsizes,
      int *array_of_starts,
      int order,
      MPI_Datatype oldtype,
      MPI_Datatype &newtype);

call MPI_Type_create_subarray(
      integer ndims, [array_of_sizes],
      [array_of_subsizes],
      [array_of_starts],
      order, oldtype,
      newtype, ierr)
```

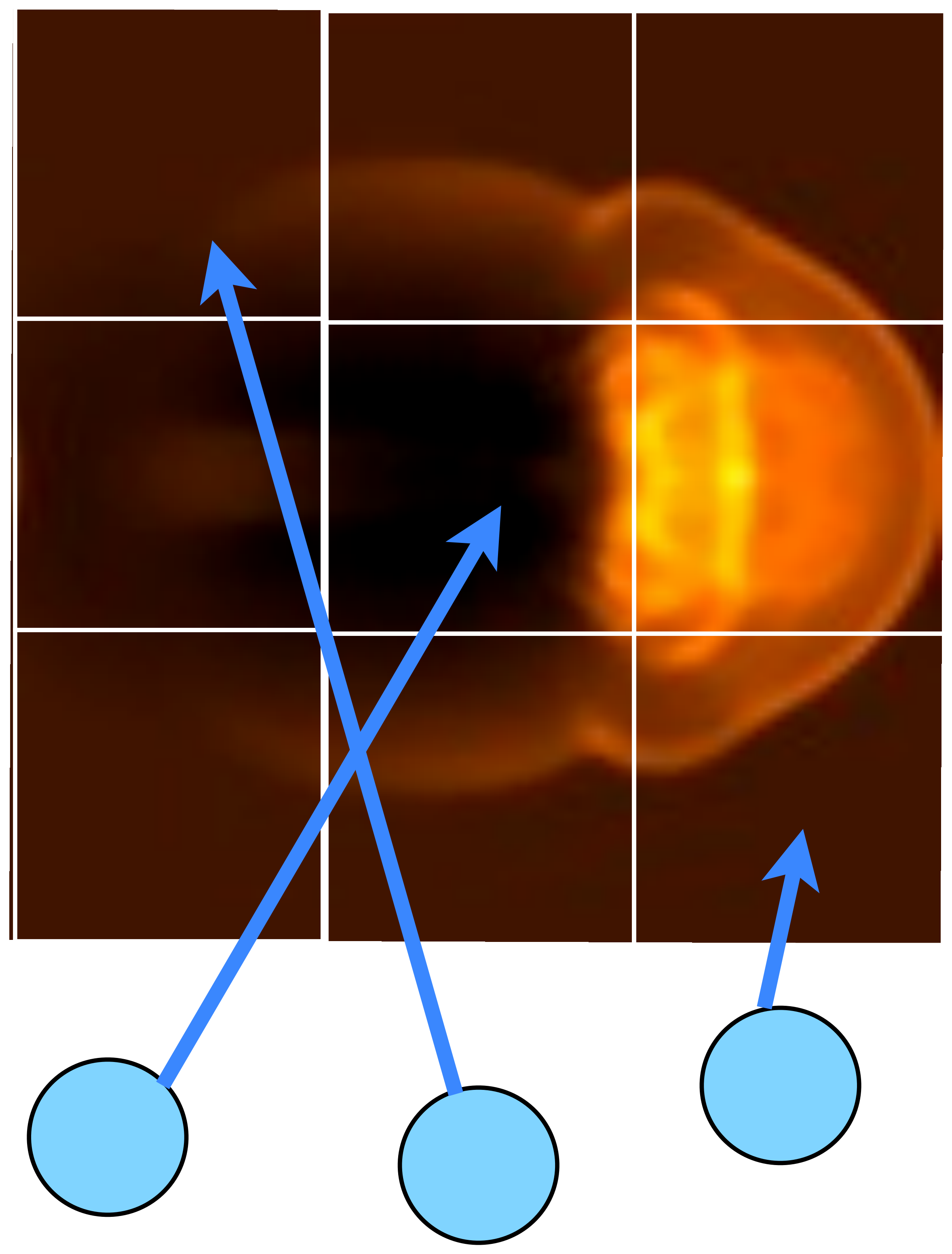
MPI-IO

- Would like the new, parallel version to still be able to write out single output files.
- But at no point does a single processor have entire domain...



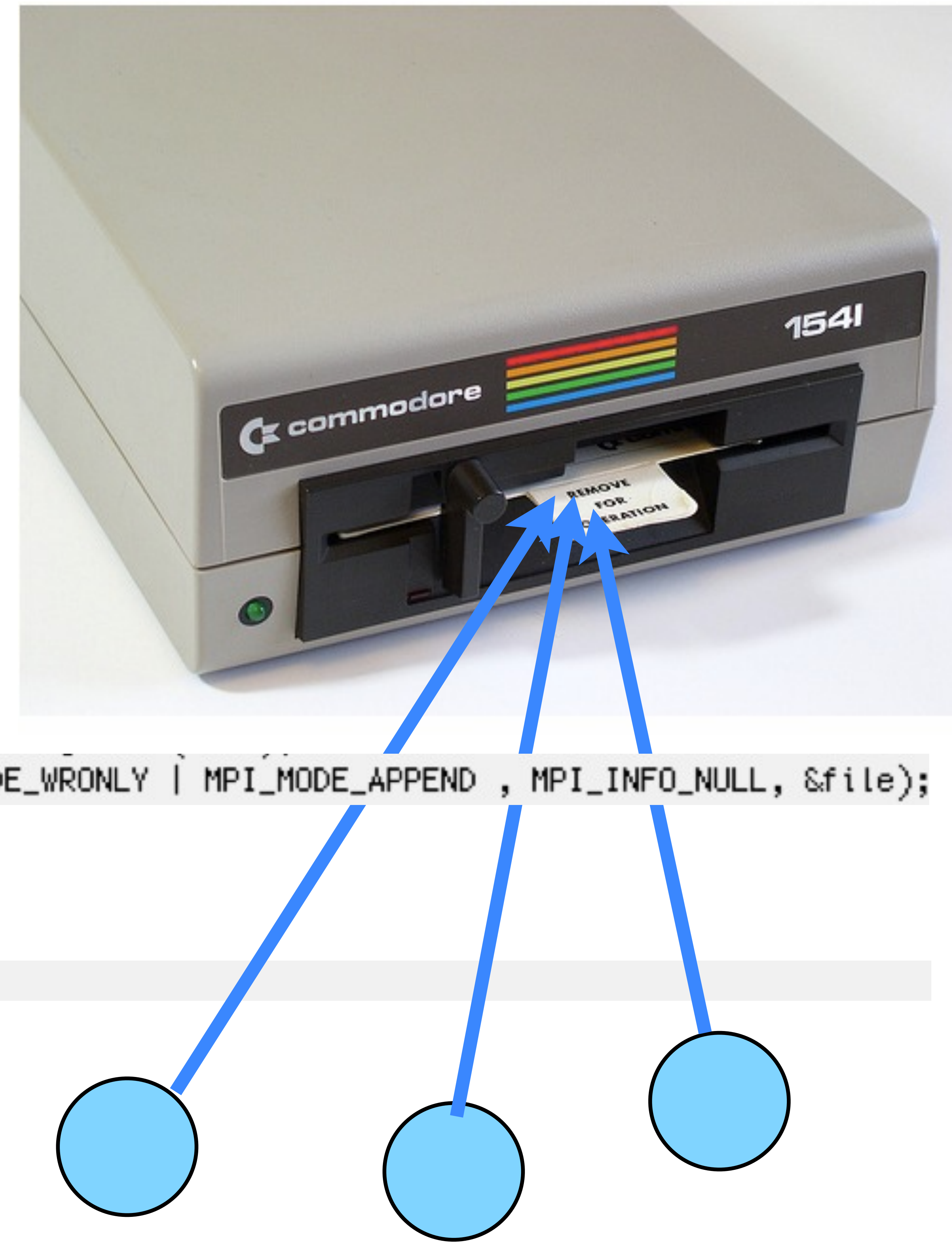
Parallel I/O

- Each processor has to write its own piece of the domain..
- without overwriting the other.
- Easier if there is global coordination



MPI-IO

- Uses MPI to coordinate reading/writing to single file



```
ierr = MPI_File_open(MPI_COMM_WORLD,filename, MPI_MODE_WRONLY | MPI_MODE_APPEND , MPI_INFO_NULL, &file);
```

...stuff...

```
ierr = MPI_File_close(&file);
```

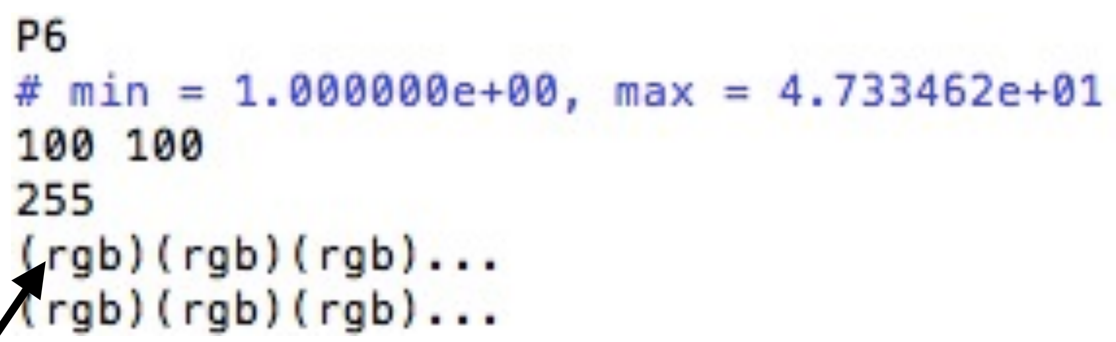
- Coordination -- *collective* operations.

PPM file format

- Simple file format
- Someone has to write a header, then each PE has to output only its 3-bytes pixels skipping everyone elses.

header -- ASCII characters

'P6', comments, height/width, max val

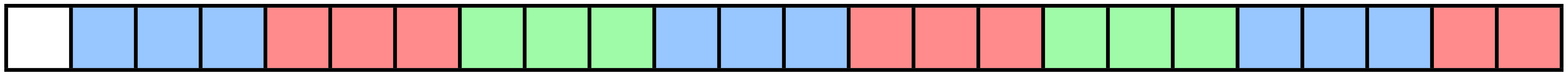


```
P6
# min = 1.000000e+00, max = 4.733462e+01
100 100
255
(rgb)(rgb)(rgb)...
(rgb)(rgb)(rgb)...
```

row by row triples of bytes: each pixel = 3 bytes

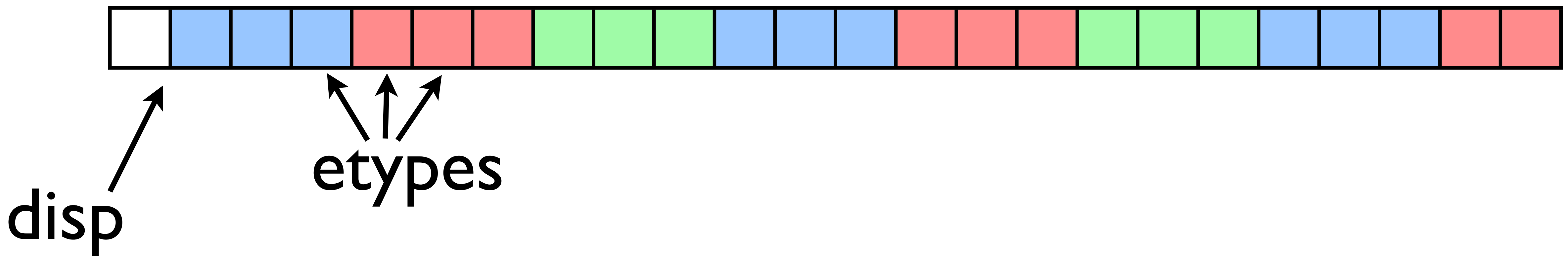
MPI-IO File View

- Each process has a view of the file that consists of only of the parts accessible to it.
- For writing, hopefully non-overlapping!
- Describing this - how data is laid out in a file - is very similar to describing how data is laid out in memory...



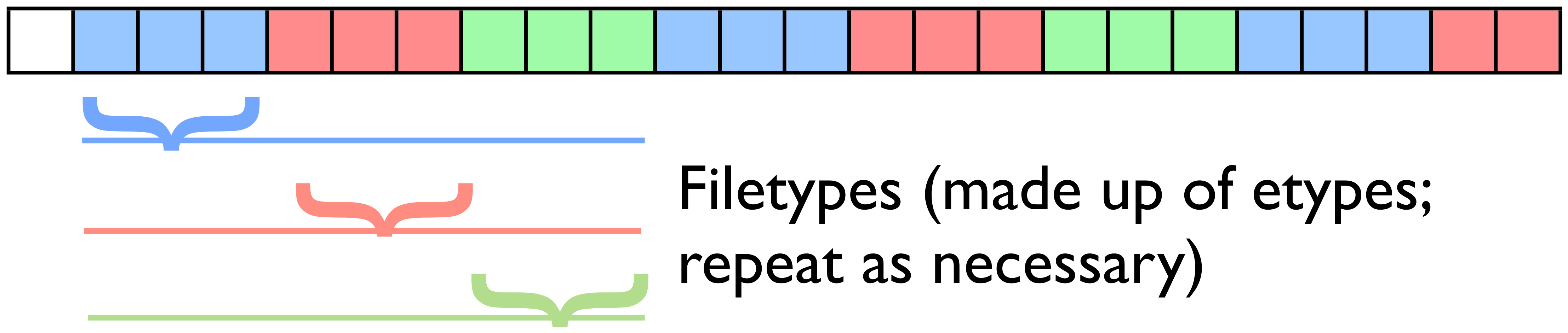
MPI-IO File View

- int MPI_File_set_view(
```
    MPI_File fh,
    MPI_Offset disp,          /* displacement in bytes from start */
    MPI_Datatype etype,       /* elementary type */
    MPI_Datatype filetype,    /* file type; prob different for each proc */
    char *datarep,            /* ‘native’ or ‘internal’ */
    MPI_Info info)            /* MPI_INFO_NULL for today */
```



MPI-IO File View

- int MPI_File_set_view(
    MPI_File fh,
    MPI_Offset disp, /* displacement in bytes from start */
    MPI_Datatype etype, /* elementary type */
    MPI_Datatype filetype, /* file type; prob different for each proc */
    char *datarep, /* 'native' or 'internal' */
    MPI_Info info) /* MPI_INFO_NULL */



Filetypes (made up of etypes; repeat as necessary)

MPI-IO File Write

- int MPI_File_write_all(
  MPI_File fh,
  void *buf,
  int count,
  MPI_Datatype datatype,
  MPI_Status *status)

Writes (_all: collectively) to part of file within view.

Hands On

- Implement the ppm routines collectively using the subarray type.