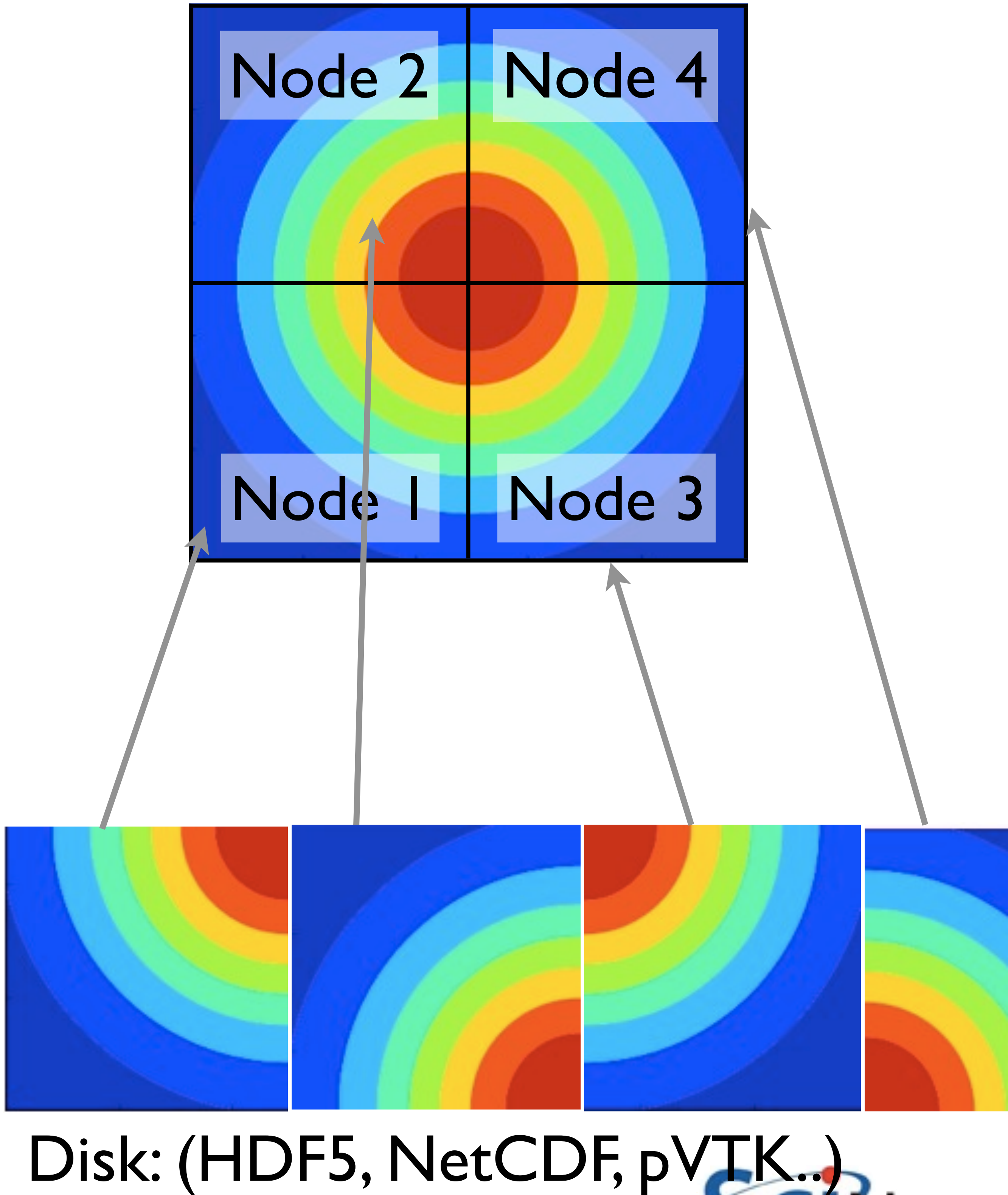
NetCDF/HDF5

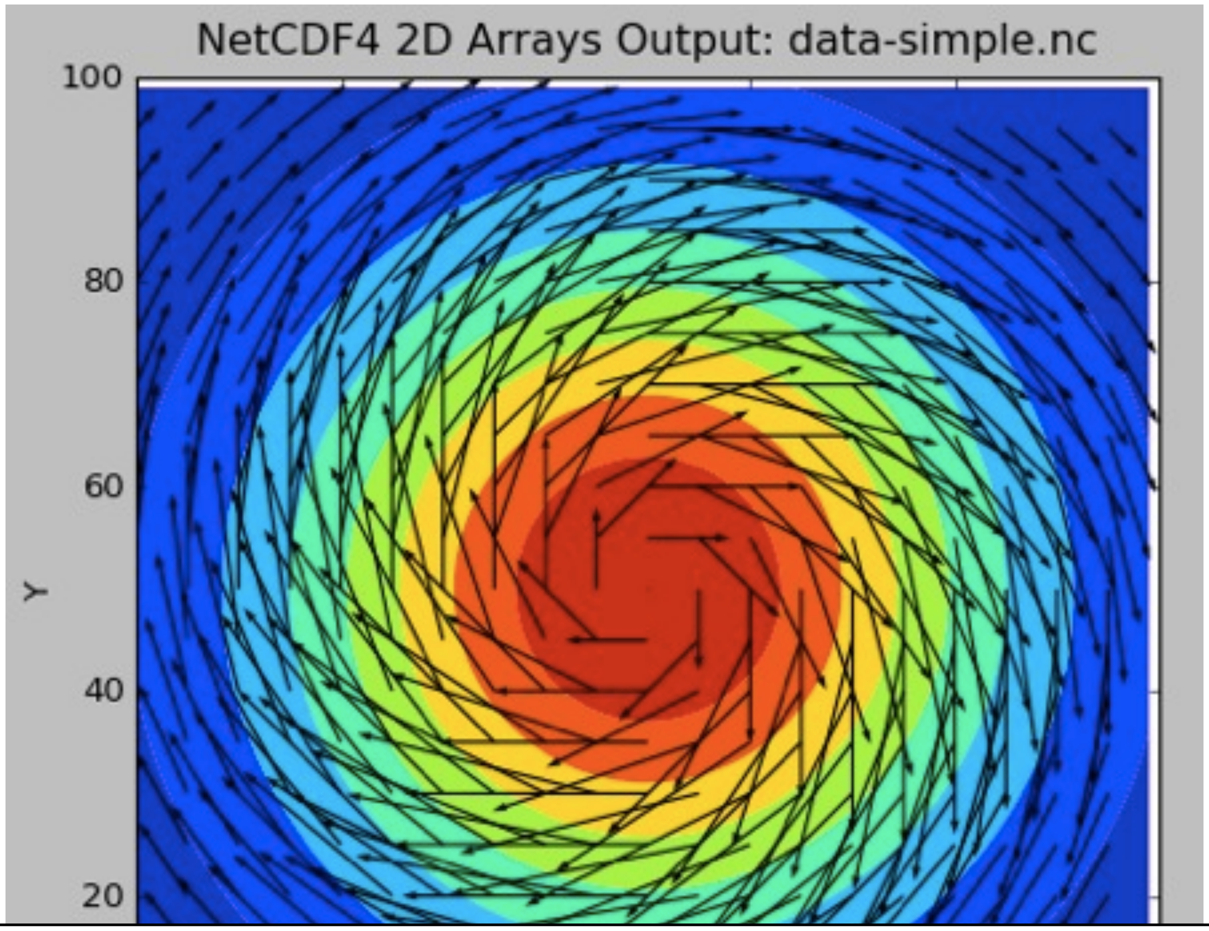
Parallel IO short course
Feb 2013

Data files must take advantage of parallel I/O

- For parallel operations on single big files, parallel filesystem isn't enough
- Data must be written in such a way that nodes can efficiently access relevant subregions
- HDF5, NetCDF formats typical examples for scientific data

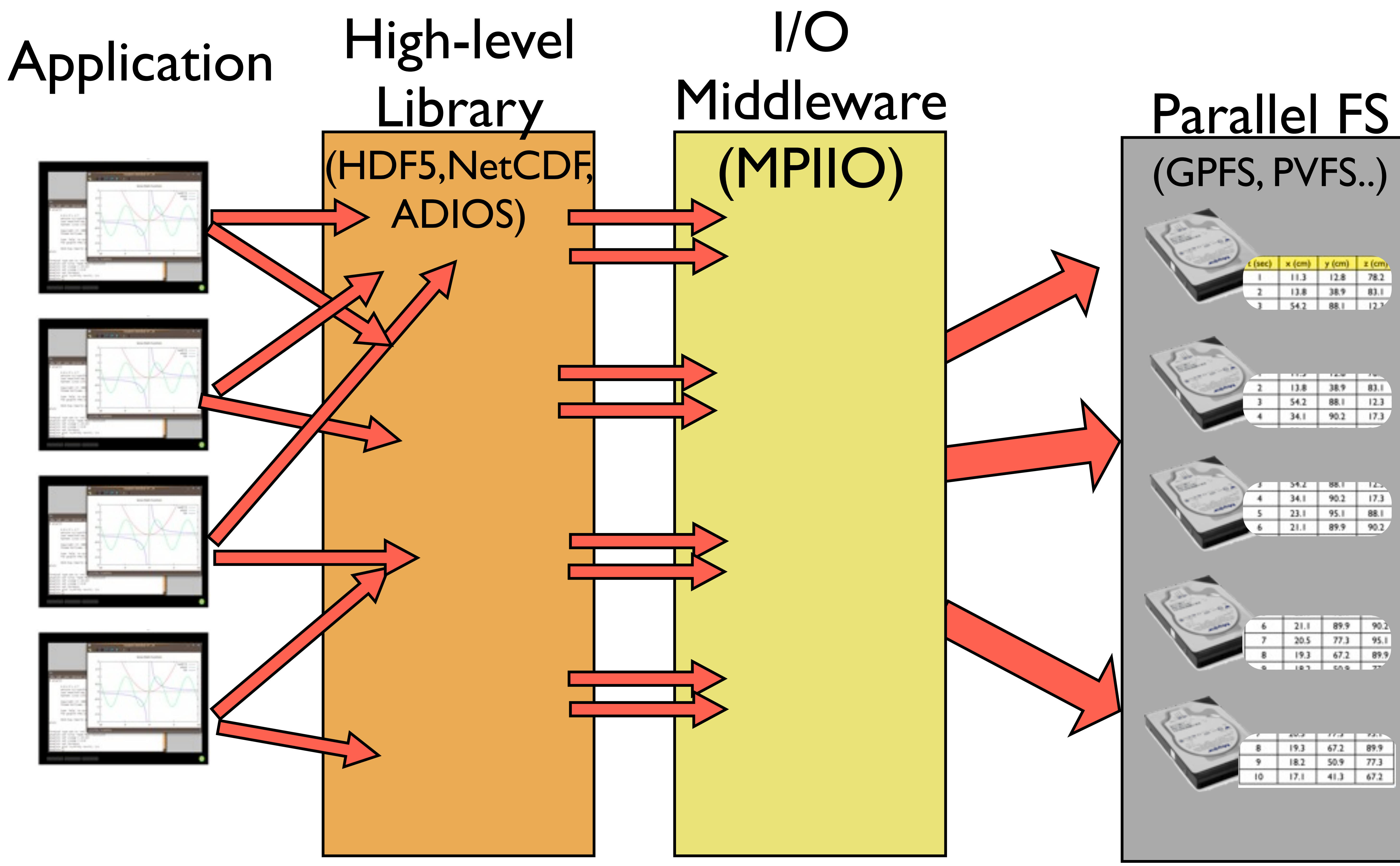


These formats are *self-describing*



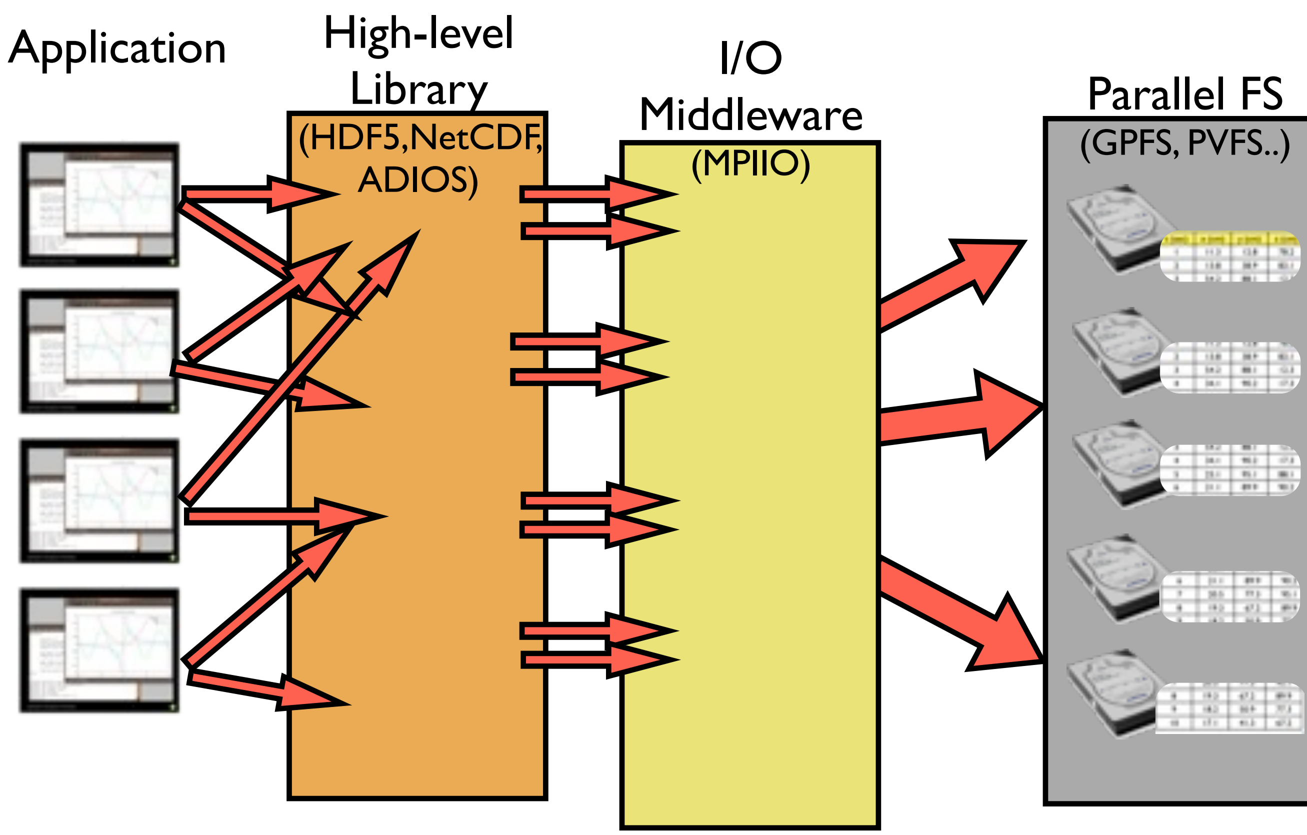
```
$ ncdump -h data-simple-fort.nc
netcdf data-simple-fort {
dimensions:
   X = 100 ;
   Y = 100 ;
   velocity components = 2 ;
variables:
   double Density(Y, X) ;
   double Velocity(Y, X, velocity components) ;
}
```

- HDF5, NetCDF have other advantages anyway
- Binary
- Self describing - contains not only data but names, descriptions of arrays, etc
- Many tools can read these formats
- Big data - formats matter


Application
High-level Library
(HDF5,NetCDF, ADIOS)
I/O Middleware
(MPIIO)
Parallel FS
(GPFS, PVFS..)

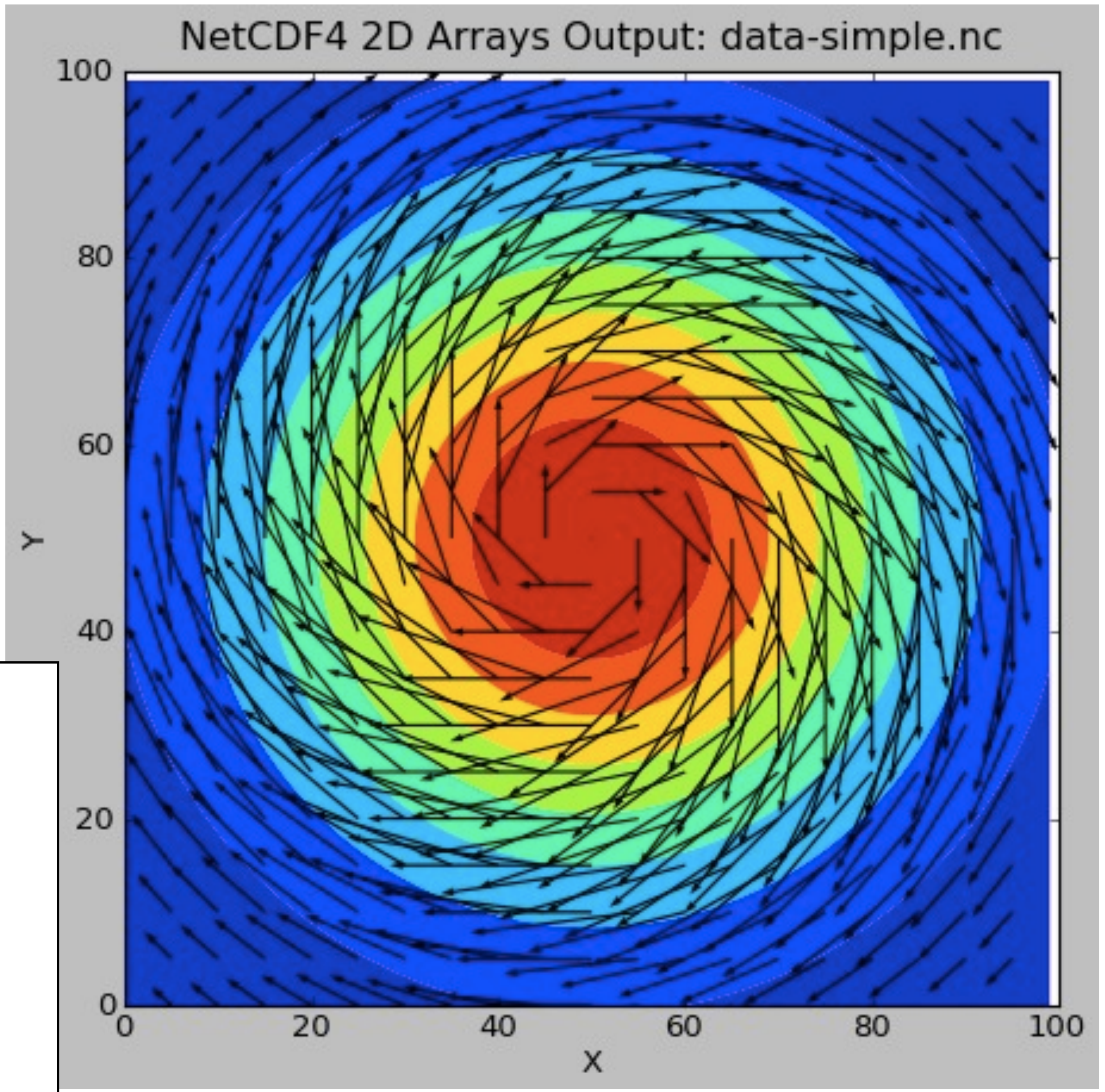
Abstraction Layers

- High Level libraries can simplify programmers tasks
  - Express IO in terms of the data structures of the code, not bytes and blocks
- I/O middleware can coordinate, improve performance



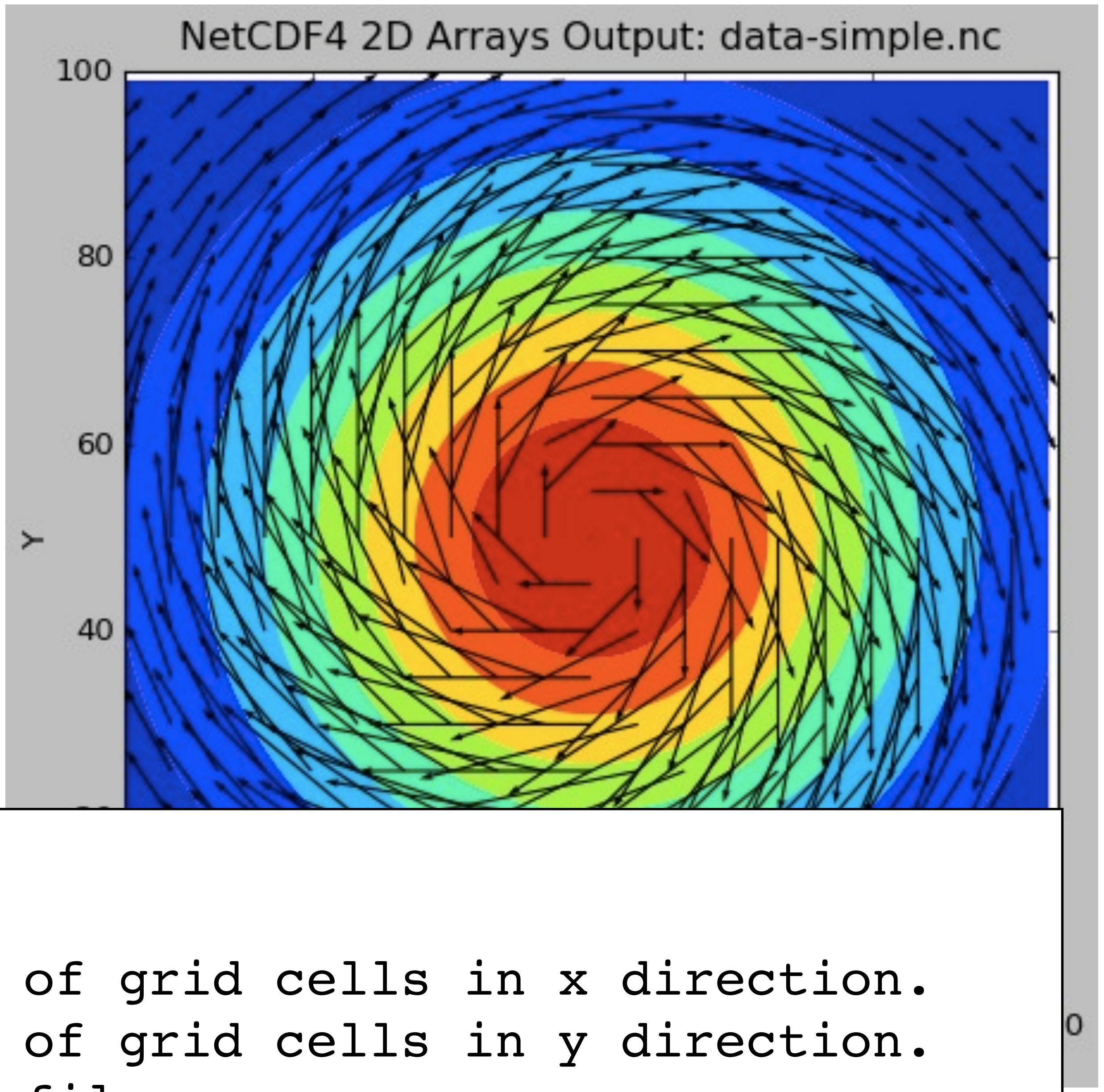
Sample Code

```
$ cd parIO/netcdf

$ make 2darray-simple (C), or
$ make f2darray-simple (F90)

$ ./{f,}2darray-simple

$ ls *.nc
$ ../plots.py *.nc
```



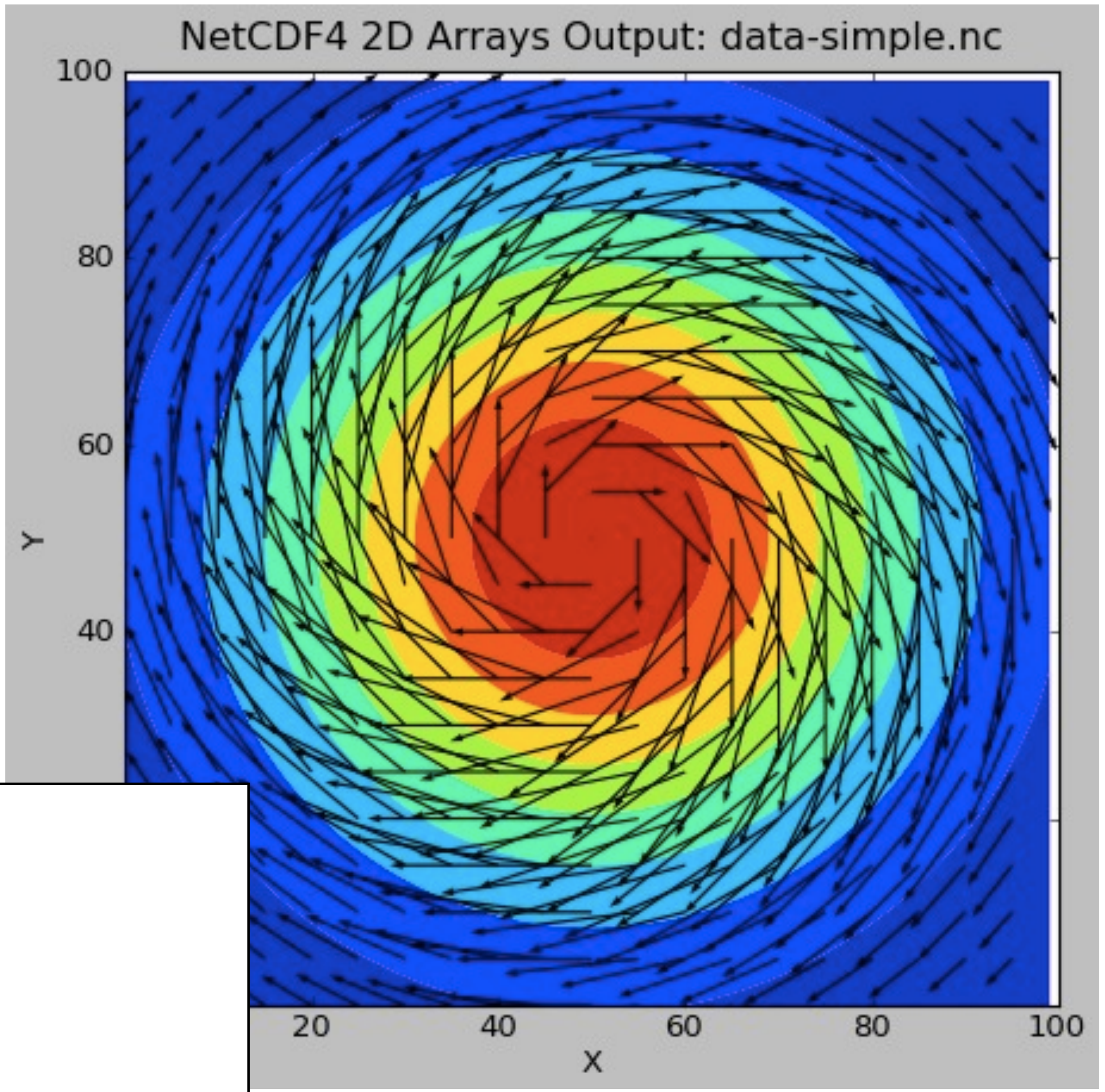
SciNet
compute • calcul
CANADA

Sample Code



```
$ ./2darray-simple --help
Options:
    --nx=N          (-x N): Set the number of grid cells in x direction.
    --ny=N          (-y N): Set the number of grid cells in y direction.
    --filename=S    (-f S): Set the output filename.

$ ./f2darray-simple --help
 Usage: f2darray-simple [--help] [filename [nx [ny]]]
        where filename is output filename, and
        nx, ny are number of points in x and y directions.
```

What is this .nc file?

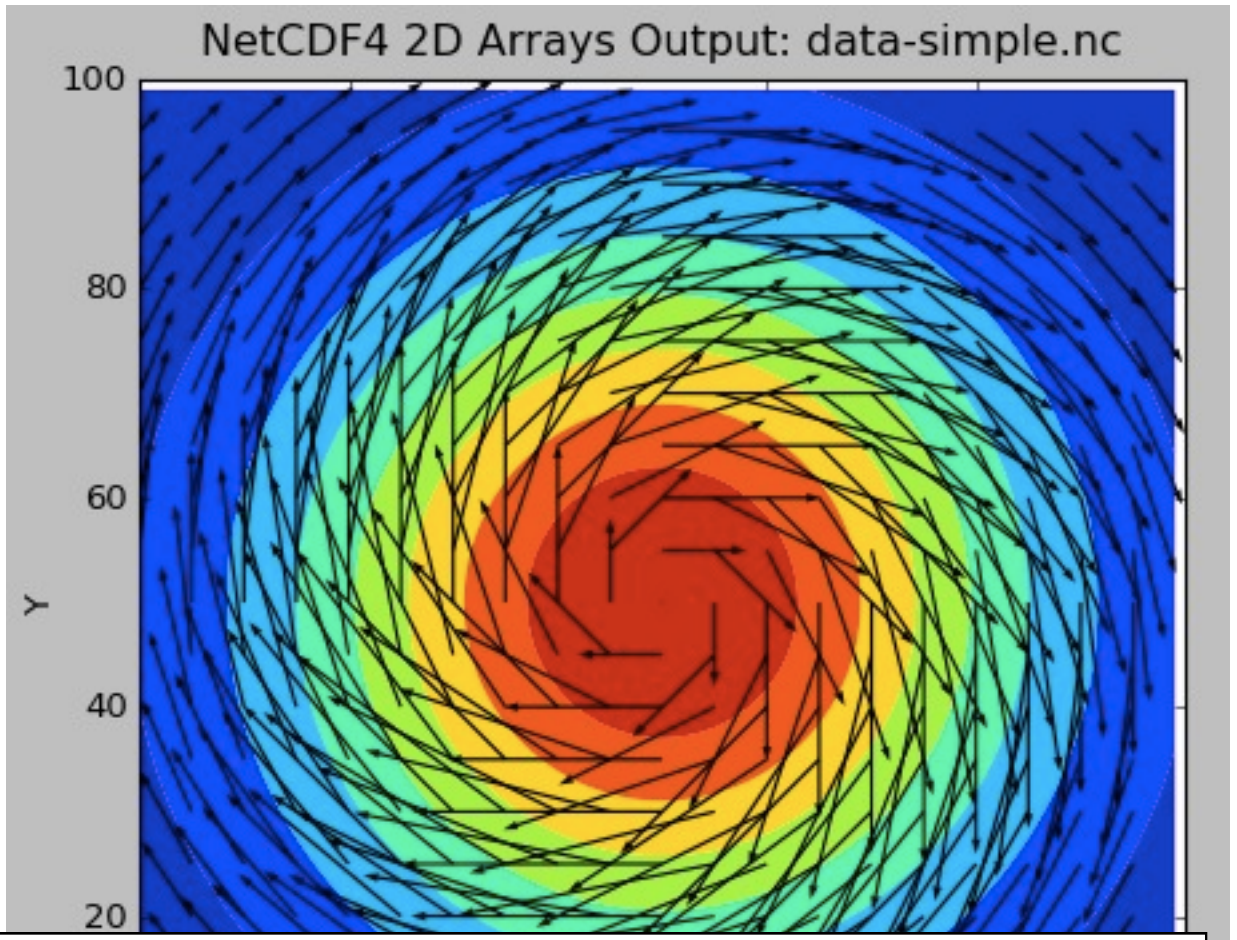


```
$ ncdump -h data-simple-fort.nc
netcdf data-simple-fort {
dimensions:
    X = 100 ;
    Y = 100 ;
    velocity\ components = 2 ;
variables:
    double Density(Y, X) ;
    double Velocity(Y, X, velocity\
components) ;
}
```

NetCDF

- NetCDF is a set of libraries and formats for:
  - portable,
  - efficient
  - "self-describing"
- way of storing and accessing large arrays (eg, for scientific data)

- Current version is NetCDF4

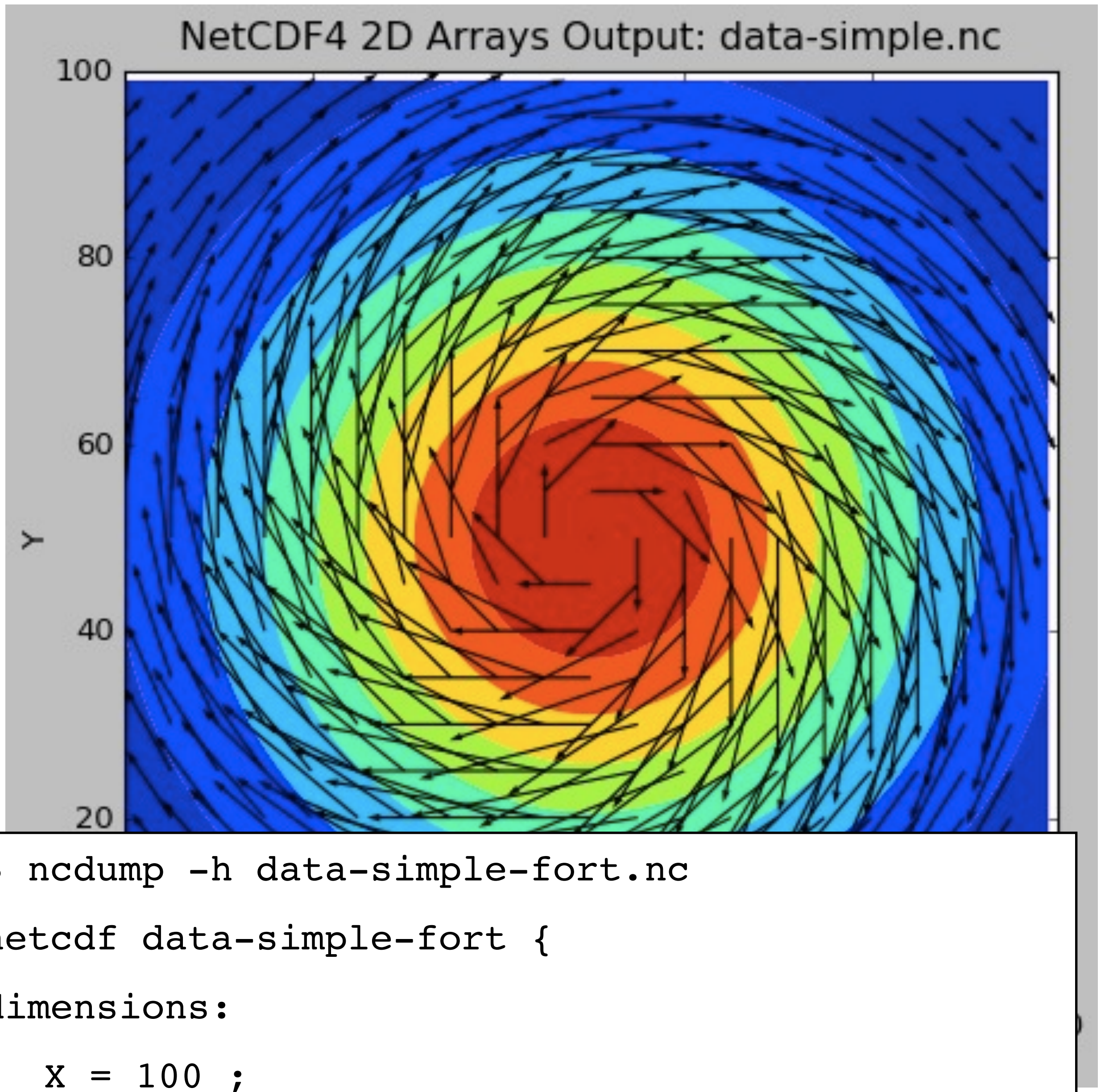


```
$ ncdump -h data-simple-fort.nc
netcdf data-simple-fort {
dimensions:
     X = 100 ;
     Y = 100 ;
     velocity\ components = 2 ;
variables:
     double Density(Y, X) ;
     double Velocity(Y, X, velocity\
components) ;
}
```

compute • calcul
CANADA

NetCDF: *Portable*

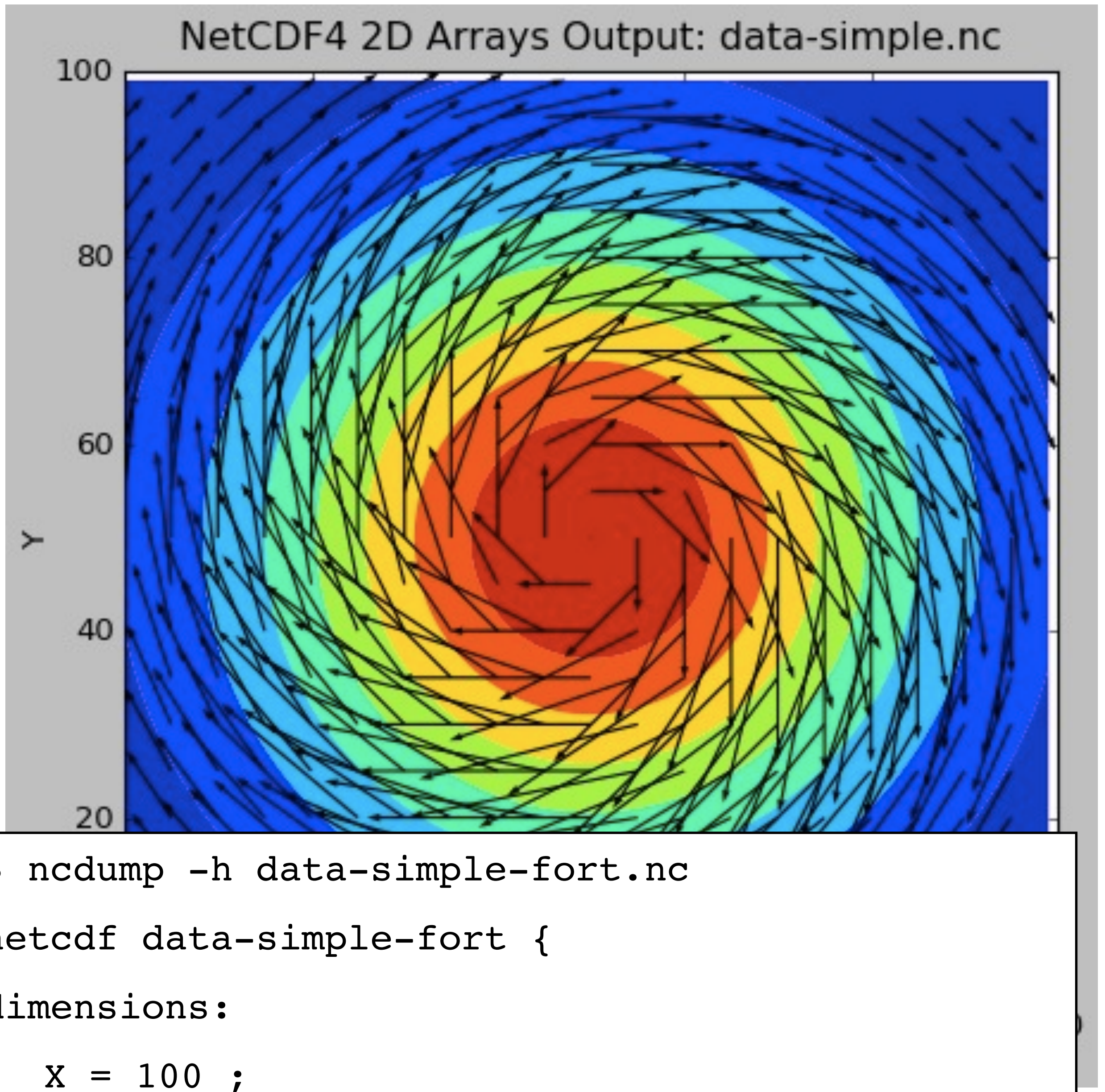
- Binary files, but common output format so that different sorts of machines can share files.
- Libraries accessible from C, C++, Fortran-77, Fortran 90/95/2003, python, etc.



```
$ ncdump -h data-simple-fort.nc
netcdf data-simple-fort {
dimensions:
   X = 100 ;
   Y = 100 ;
   velocity\ components = 2 ;
variables:
   double Density(Y, X) ;
   double Velocity(Y, X, velocity\
components) ;
}
```

NetCDF: *Self-Describing*

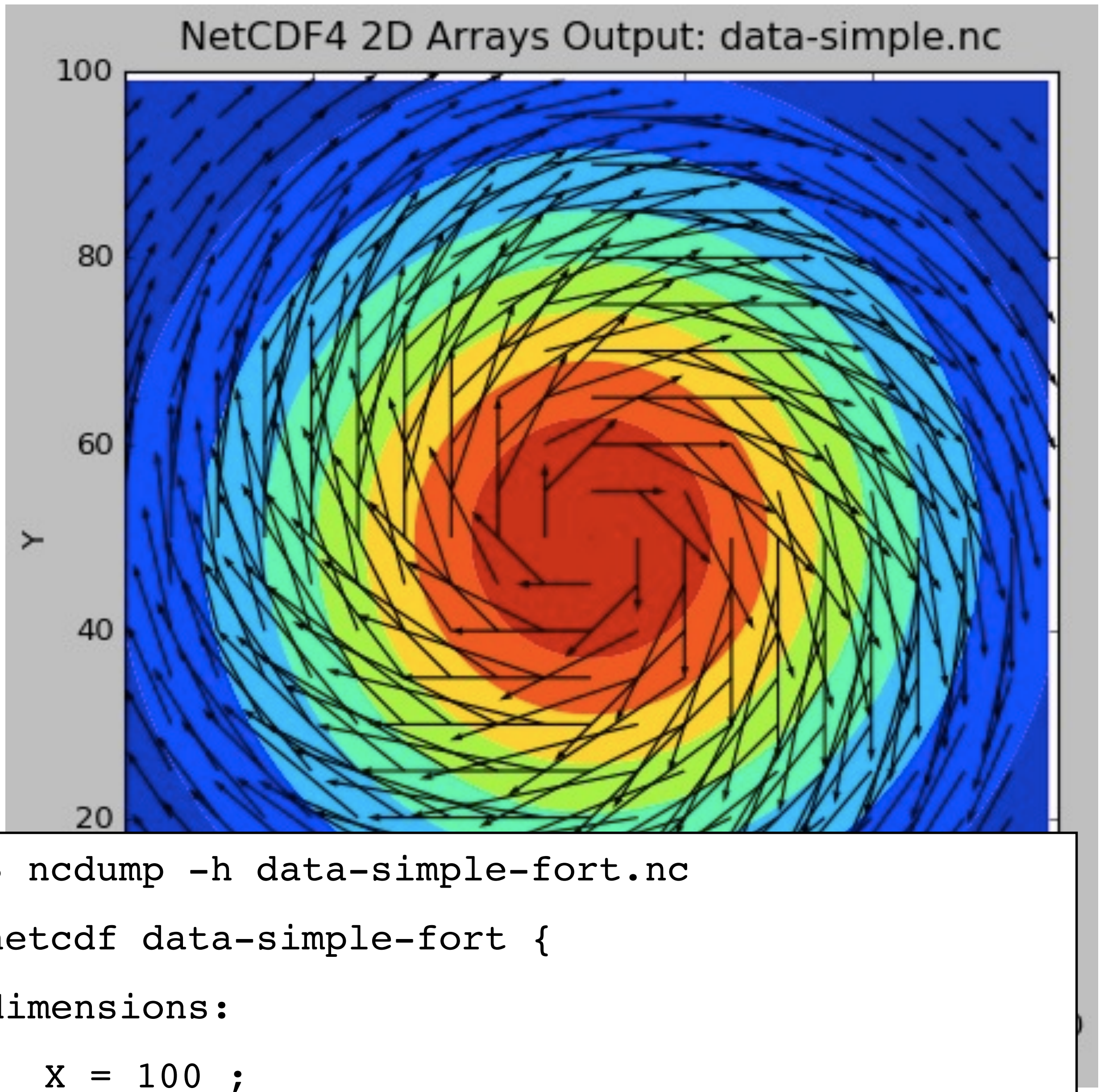
- Header contains the metadata to describe the big data
- Lists:
  - Array names
  - Dimensions
  - *shared* dimensions - information about how the arrays relate
  - Other, related information



```
$ ncdump -h data-simple-fort.nc
netcdf data-simple-fort {
dimensions:
   X = 100 ;
   Y = 100 ;
   velocity\ components = 2 ;
variables:
   double Density(Y, X) ;
   double Velocity(Y, X, velocity\
components) ;
}
```

NetCDF: *Efficient*

- Binary, so less translation (as little is used as possible)
- IO libraries themselves are written for performance
- API, data format makes it easy to efficiently read,write subregions of arrays (slices, or 'hyperslabs')
- Still possible to make things slow - lots of metadata queries, modifications



```
$ ncdump -h data-simple-fort.nc
netcdf data-simple-fort {
dimensions:
   X = 100 ;
   Y = 100 ;
   velocity\ components = 2 ;
variables:
   double Density(Y, X) ;
   double Velocity(Y, X, velocity\
components) ;
}
```

2darray-simple.c

```
#include "netcdf.h"

...

void writenetcdffile(rundata_t rundata, double **dens,
                     double ***vel) {
    /* identifiers */
    int file_id;

...
    /* return status */
    int status;

    /* Create a new file - clobber anything existing */
    status = nc_create(rundata.filename, NC_CLOBBER, &file_id);
    /* netCDF routines return NC_NOERR on success */
    if (status != NC_NOERR) {
        fprintf(stderr,"Could not open file %s\n", rundata.filename);
```

Include function definitions

2darray-simple.c

```
#include "netcdf.h"


...


void writenetcdffile(rundata_t rundata, double **dens,
                     double ***vel) {
    /* identifiers */
    int file_id;


...
    /* return status */
    int status;


    /* Create a new file, clobber anything existing */
    status = nc_create(rundata.filename, NC_CLOBBER, &file_id);
    /* netCDF routines return NC_NOERR on success */
    if (status != NC_NOERR) {
        fprintf(stderr,"Could not open file %s\n", rundata.filename);
```

Create a new file, with name rundata.filename

2darray-simple.c

```
#include "netcdf.h"


...


void writenetcdffile(rundata_t rundata, double **dens,
                     double ***vel) {
    /* identifiers */
    int file_id;

...
    /* return status */
    int status;

    /* Create a new file - clobber anything existing */
    status = nc_create(rundata.filename, NC_CLOBBER, &file_id);
    /* netCDF routines return NC_NOERR on success */
    if (status != NC_NOERR) {
        fprintf(stderr,"Could not open file %s\n", rundata.filename);
```

Clobber anything already in the file

2darray-simple.c

```
#include "netcdf.h"

...

void writenetcdffile(rundata_t rundata, double **dens,
                     double ***vel) {
    /* identifiers */
    int file_id;

...
    /* return status */
    int status;

    /* Create a new file - clobber anything existing */
    status = nc_create(rundata.filename, NC_CLOBBER, &file_id);
    /* netCDF routines return NC_NOERR on success */
    if (status != NC_NOERR) {
        fprintf(stderr,"Could not open file %s\n", rundata.filename);
```

Test the return codes

f2darray-simple.f90

```
subroutine writenetcdffile(rundata, dens, vel)
     use netcdf
     implicit none
     type(rundata_t), intent(IN) :: rundata
     double precision, intent(IN), dimension(:,:) :: dens
     double precision, intent(IN), dimension(:,:,:) :: vel

     integer :: file_id
...
     integer :: status

     ! create the file, check return code

     status = nf90_create(path=rundata%filename, cmode=NF90_CLOBBER,
ncid=file_id)
     if (status /= NF90_NOERR) then
         print *,'Could not open file ', rundata%filename
     return
```

Import definitions

f2darray-simple.f90

```
subroutine writenetcdffile(rundata, dens, vel)
     use netcdf
     implicit none
     type(rundata_t), intent(IN) :: rundata
     double precision, intent(IN), dimension(:,:) :: dens
     double precision, intent(IN), dimension(:,:,:) :: vel

     integer :: file_id
...
     integer :: status

     ! create the file, check return code

     status = nf90_create(path=rundata%filename, cmode=NF90_CLOBBER,
ncid=file_id)
     if (status /= NF90_NOERR) then
         print *,'Could not open file ', rundata%filename
     return
```

Create file

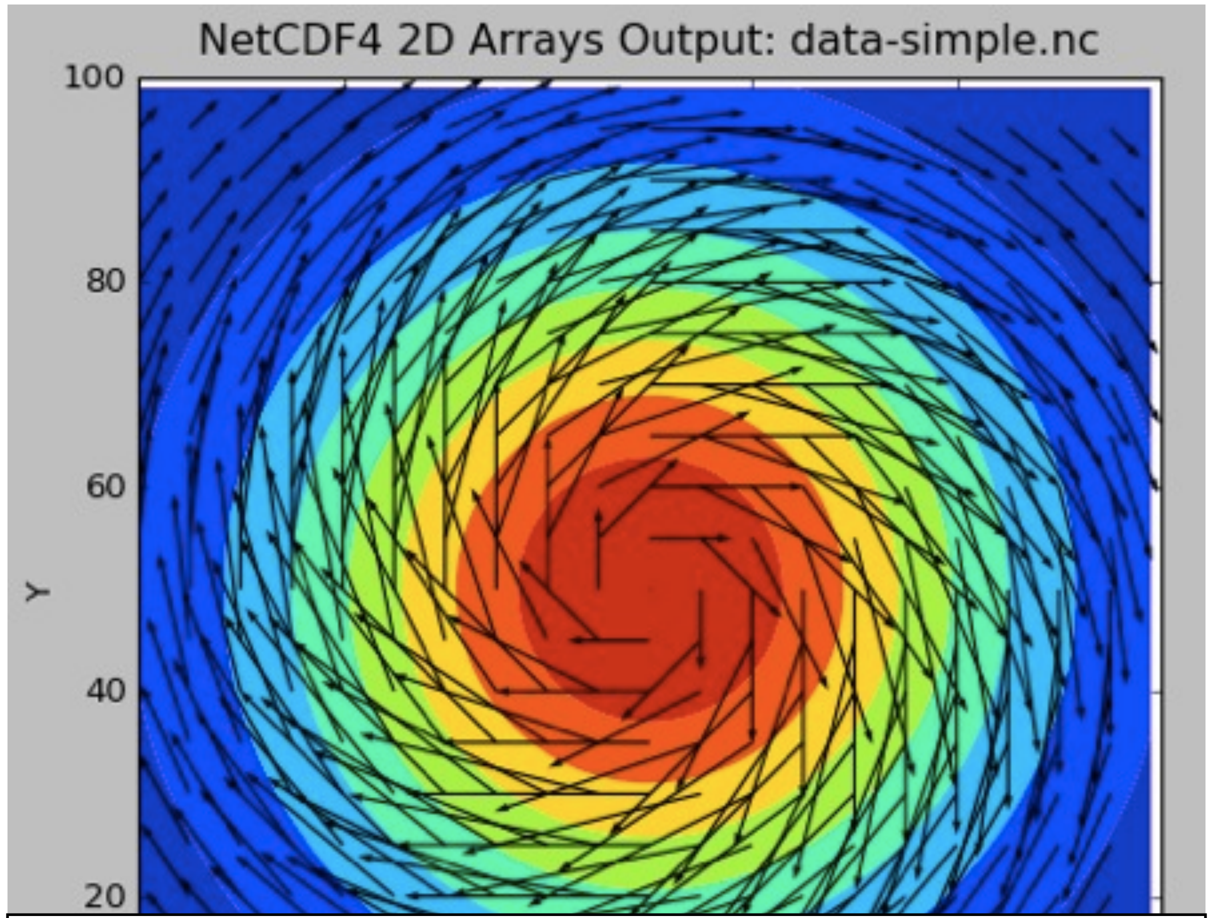
f2darray-simple.f90

```
subroutine writenetcdffile(rundata, dens, vel)
     use netcdf
     implicit none
     type(rundata_t), intent(IN) :: rundata
     double precision, intent(IN), dimension(:,:) :: dens
     double precision, intent(IN), dimension(:,:,:) :: vel

     integer :: file_id
...
     integer :: status

     ! create the file, check return code

     status = nf90_create(path=rundata%filename, cmode=NF90_CLOBBER,
ncid=file_id)
     if (status /= NF90_NOERR) then
         print *,'Could not open file ', rundata%filename
     return
```

C definitions are NC_,
F90 are NF90_

Writing a NetCDF File

- To write a NetCDF file, we go through the following steps:
  - **Create** the file (or open it for appending)
  - **Define dimensions** of the arrays we'll be writing
  - **Define variables** on those dimensions
  - **End definition** phase
  - **Write variables**
  - **Close file**



```
$ ncdump -h data-simple-fort.nc
netcdf data-simple-fort {
dimensions:
    X = 100 ;
    Y = 100 ;
    velocity\ components = 2 ;
variables:
    double Density(Y, X) ;
    double Velocity(Y, X, velocity\ components) ;
}
```

f2darray-simple.f90

```
integer :: file_id, xdim_id, ydim_id, vcomp_id
integer :: dens_id, vel_id
integer, dimension(2) :: densdims
integer, dimension(3) :: veldims

...

status = nf90_def_dim(file_id, 'X', rundata%nx, xdim_id)
status = nf90_def_dim(file_id, 'Y', rundata%ny, ydim_id)
status = nf90_def_dim(file_id, 'velocity components', 2, vcomp_id)

densdims = (/ xdim_id, ydim_id /)
veldims = (/ vcomp_id, xdim_id, ydim_id /)

status = nf90_def_var(file_id, 'Density',  NF90_DOUBLE, densdims, dens_id)
if (status /= NF90_NOERR) print *, trim(nf90_strerror(status)), ' Dens'
status = nf90_def_var(file_id, 'Velocity', NF90_DOUBLE, veldims,  vel_id)

status = nf90_enddef(file_id)
```

Define the dimensions in the file: name, size, id

f2darray-simple.f90

```
integer :: file_id, xdim_id, ydim_id, vcomp_id
integer :: dens_id, vel_id
integer, dimension(2) :: densdims
integer, dimension(3) :: veldims

...

status = nf90_def_dim(file_id, 'X', rundata%nx, xdim_id)
status = nf90_def_dim(file_id, 'Y', rundata%ny, ydim_id)
status = nf90_def_dim(file_id, 'velocity components', 2, vcomp_id)

densdims = (/ xdim_id, ydim_id /)
veldims = (/ vcomp_id, xdim_id, ydim_id /)

status = nf90_def_var(file_id, 'Density',  NF90_DOUBLE, densdims, dens_id)
if (status /= NF90_NOERR) print *, trim(nf90_strerror(status)), ' Dens'
status = nf90_def_var(file_id, 'Velocity', NF90_DOUBLE, veldims,  vel_id)

status = nf90_enddef(file_id)
```

Variables are defined in terms of these dims

f2darray-simple.f90

Once you're done defining things,

```
status = nf90_enddef(file_id)

! Write out the values
status = nf90_put_var(file_id, dens_id, dens)
if (status /= NF90_NOERR) print *, trim(nf90_strerror(status)), ' Dens'
status = nf90_put_var(file_id, vel_id,  vel)
if (status /= NF90_NOERR) print *, trim(nf90_strerror(status)), ' Vel'

status = nf90_close(file_id)
```

f2darray-simple.f90

Writing data is easy.

```
status = nf90_enddef(file_id)

! Write out the values
status = nf90_put_var(file_id, dens_id, dens)
if (status /= NF90_NOERR) print *, trim(nf90_strerror(status)), ' Dens'
status = nf90_put_var(file_id, vel_id,  vel)
if (status /= NF90_NOERR) print *, trim(nf90_strerror(status)), ' Vel'

status = nf90_close(file_id)
```

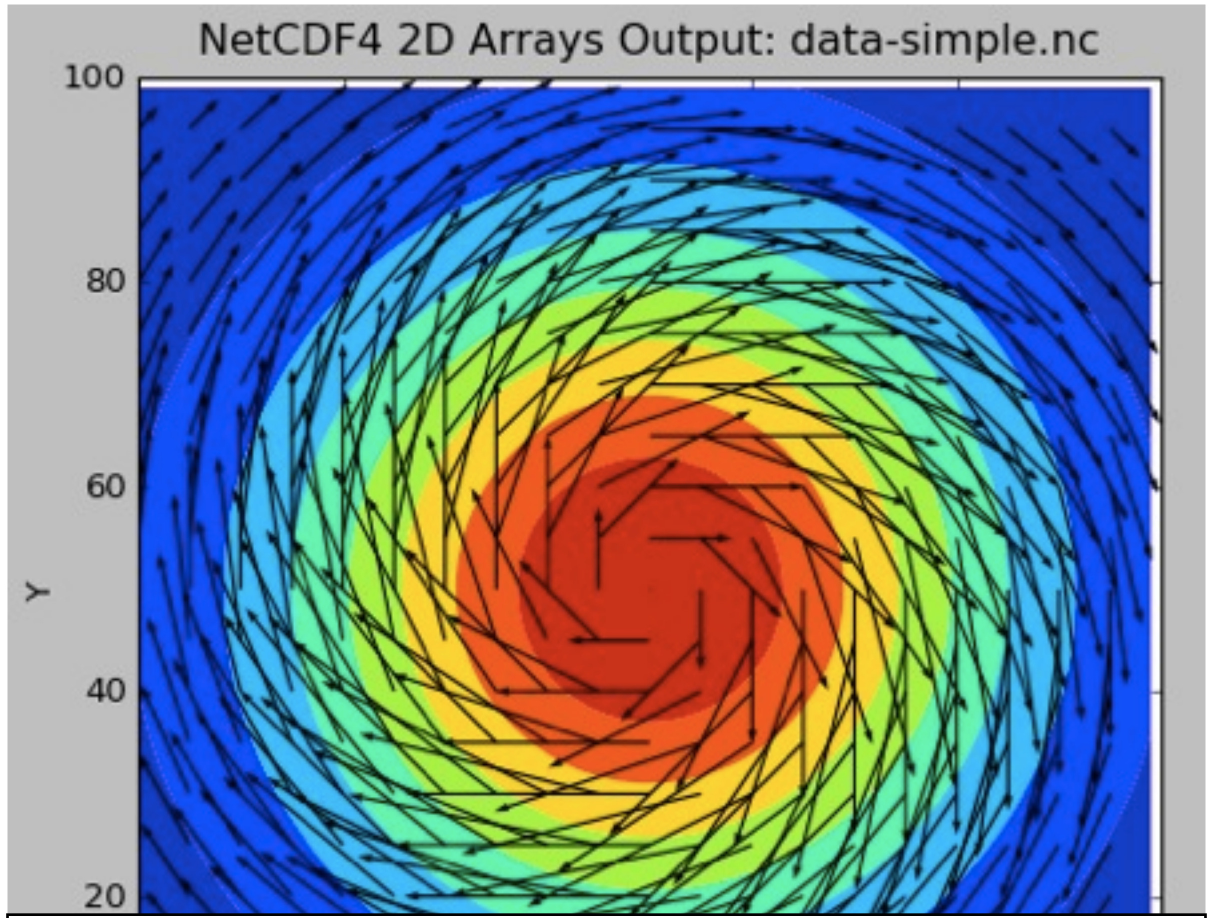
f2darray-simple.f90

```
status = nf90_enddef(file_id)

! Write out the values
status = nf90_put_var(file_id, dens_id, dens)
if (status /= NF90_NOERR) print *, trim(nf90_strerror(status)), ' Dens'
status = nf90_put_var(file_id, vel_id,  vel)
if (status /= NF90_NOERR) print *, trim(nf90_strerror(status)), ' Vel'

status = nf90_close(file_id)
```

Closing the file is important!!

Reading a NetCDF File

- Flow is slightly different
  - **Open** the file for reading
  - **Get dimension ids** of the the dimensions in the files
  - **Get dimension lengths** so you can allocate the files
  - **Get variable ids** so you can access the data
  - **Read variables**
  - **Close file**



```
$ ncdump -h data-simple-fort.nc
netcdf data-simple-fort {
dimensions:
    X = 100 ;
    Y = 100 ;
    velocity\ components = 2 ;
variables:
    double Density(Y, X) ;
    double Velocity(Y, X, velocity\ components) ;
}
```

fread2darray-simple.f90

```
status = nf90_open(path=rundata%filename, mode=NF90_NOWRITE, ncid=file_id)
...
! find the dimensions
status = nf90_inq_dimid(file_id, 'X', xdim_id)
status = nf90_inq_dimid(file_id, 'Y', ydim_id)
status = nf90_inq_dimid(file_id, 'velocity components', vcomp_id)
! find the dimension lengths
status = nf90_inquire_dimension(file_id, xdim_id, len = rundata % nx)
status = nf90_inquire_dimension(file_id, ydim_id, len = rundata % ny )
status = nf90_inquire_dimension(file_id, vcomp_id,len = rundata % nvelcomp)
! now we can allocate variable sizes
allocate(dens(rundata%nx, rundata%ny)) !...etc...

status = nf90_inq_varid(file_id, 'Density',  dens_id)
status = nf90_inq_varid(file_id, 'Velocity', vel_id)

status = nf90_get_var(file_id, dens_id, dens)
status = nf90_get_var(file_id, vel_id,  vel)

status = nf90_close(file_id)
```

fread2darray-simple.f90

```
status = nf90_open(path=rundata%filename, mode=NF90_NOWRITE, ncid=file_id)
...
! find the dimensions
status = nf90_inq_dimid(file_id, 'X', xdim_id)
status = nf90_inq_dimid(file_id, 'Y', ydim_id)
status = nf90_inq_dimid(file_id, 'velocity components', vcomp_id)
! find the dimension lengths
status = nf90_inquire_dimension(file_id, xdim_id, len = rundata % nx)
status = nf90_inquire_dimension(file_id, ydim_id, len = rundata % ny )
status = nf90_inquire_dimension(file_id, vcomp_id,len = rundata % nvelcomp)
! now we can allocate variable sizes
allocate(dens(rundata%nx, rundata%ny)) !...etc...

status = nf90_inq_varid(file_id, 'Density',  dens_id)
status = nf90_inq_varid(file_id, 'Velocity', vel_id)

status = nf90_get_var(file_id, dens_id, dens)
status = nf90_get_var(file_id, vel_id,  vel)

status = nf90_close(file_id)
```

fread2darray-simple.f90

```
status = nf90_open(path=rundata%filename, mode=NF90_NOWRITE, ncid=file_id)
...
! find the dimensions
status = nf90_inq_dimid(file_id, 'X', xdim_id)
status = nf90_inq_dimid(file_id, 'Y', ydim_id)
status = nf90_inq_dimid(file_id, 'velocity components', vcomp_id)
! find the dimension lengths
status = nf90_inquire_dimension(file_id, xdim_id, len = rundata % nx)
status = nf90_inquire_dimension(file_id, ydim_id, len = rundata % ny )
status = nf90_inquire_dimension(file_id, vcomp_id,len = rundata % nvelcomp)
! now we can allocate variable sizes
allocate(dens(rundata%nx, rundata%ny)) !...etc...

status = nf90_inq_varid(file_id, 'Density',  dens_id)
status = nf90_inq_varid(file_id, 'Velocity', vel_id)

status = nf90_get_var(file_id, dens_id, dens)
status = nf90_get_var(file_id, vel_id,  vel)

status = nf90_close(file_id)
```

fread2darray-simple.f90

```
status = nf90_open(path=rundata%filename, mode=NF90_NOWRITE, ncid=file_id)
...
! find the dimensions
status = nf90_inq_dimid(file_id, 'X', xdim_id)
status = nf90_inq_dimid(file_id, 'Y', ydim_id)
status = nf90_inq_dimid(file_id, 'velocity components', vcomp_id)
! find the dimension lengths
status = nf90_inquire_dimension(file_id, xdim_id, len = rundata % nx)
status = nf90_inquire_dimension(file_id, ydim_id, len = rundata % ny )
status = nf90_inquire_dimension(file_id, vcomp_id,len = rundata % nvelcomp)
! now we can allocate variable sizes
allocate(dens(rundata%nx, rundata%ny)) !...etc...

status = nf90_inq_varid(file_id, 'Density',  dens_id)
status = nf90_inq_varid(file_id, 'Velocity', vel_id)

status = nf90_get_var(file_id, dens_id, dens)
status = nf90_get_var(file_id, vel_id,  vel)

status = nf90_close(file_id)
```

fread2darray-simple.f90

```fortran
status = nf90_open(path=rundata%filename, mode=NF90_NOWRITE, ncid=file_id)
...
! find the dimensions
status = nf90_inq_dimid(file_id, 'X', xdim_id)
status = nf90_inq_dimid(file_id, 'Y', ydim_id)
status = nf90_inq_dimid(file_id, 'velocity components', vcomp_id)
! find the dimension lengths
status = nf90_inquire_dimension(file_id, xdim_id, len = rundata % nx)
status = nf90_inquire_dimension(file_id, ydim_id, len = rundata % ny )
status = nf90_inquire_dimension(file_id, vcomp_id,len = rundata % nvelcomp)
! now we can allocate variable sizes
allocate(dens(rundata%nx, rundata%ny)) !...etc...

status = nf90_inq_varid(file_id, 'Density',  dens_id)
status = nf90_inq_varid(file_id, 'Velocity', vel_id)

status = nf90_get_var(file_id, dens_id, dens)
status = nf90_get_var(file_id, vel_id,  vel)

status = nf90_close(file_id)
```

fread2darray-simple.f90

```fortran
status = nf90_open(path=rundata%filename, mode=NF90_NOWRITE, ncid=file_id)
...
! find the dimensions
status = nf90_inq_dimid(file_id, 'X', xdim_id)
status = nf90_inq_dimid(file_id, 'Y', ydim_id)
status = nf90_inq_dimid(file_id, 'velocity components', vcomp_id)
! find the dimension lengths
status = nf90_inquire_dimension(file_id, xdim_id, len = rundata % nx)
status = nf90_inquire_dimension(file_id, ydim_id, len = rundata % ny )
status = nf90_inquire_dimension(file_id, vcomp_id,len = rundata % nvelcomp)
! now we can allocate variable sizes
allocate(dens(rundata%nx, rundata%ny)) !...etc...

status = nf90_inq_varid(file_id, 'Density',  dens_id)
status = nf90_inq_varid(file_id, 'Velocity', vel_id)

status = nf90_get_var(file_id, dens_id, dens)
status = nf90_get_var(file_id, vel_id,  vel)

status = nf90_close(file_id)
```

read2darray-simple.c

```
status = nc_open(rundata->filename, NC_NOWRITE, &file_id);

/* Get the dimensions */
status = nc_inq_dimid(file_id, "X", &xdim_id);
if (status != NC_NOERR) fprintf(stderr, "Could not get X\n");
status = nc_inq_dimid(file_id, "Y", &ydim_id);
status = nc_inq_dimid(file_id, "velocity component", &vcomp_id);

status = nc_inq_dimlen(file_id, xdim_id, &(rundata->nx));
status = nc_inq_dimlen(file_id, ydim_id, &(rundata->ny));
status = nc_inq_dimlen(file_id, vcomp_id, &(rundata->nveldims));

...
nc_inq_varid(file_id, "Density", &dens_id);
nc_inq_varid(file_id, "Velocity", &vel_id);

nc_get_var_double(file_id, dens_id, &((*dens)[0][0]));
nc_get_var_double(file_id, vel_id, &((*vel)[0][0][0]));

nc_close(file_id);
```

A Better example

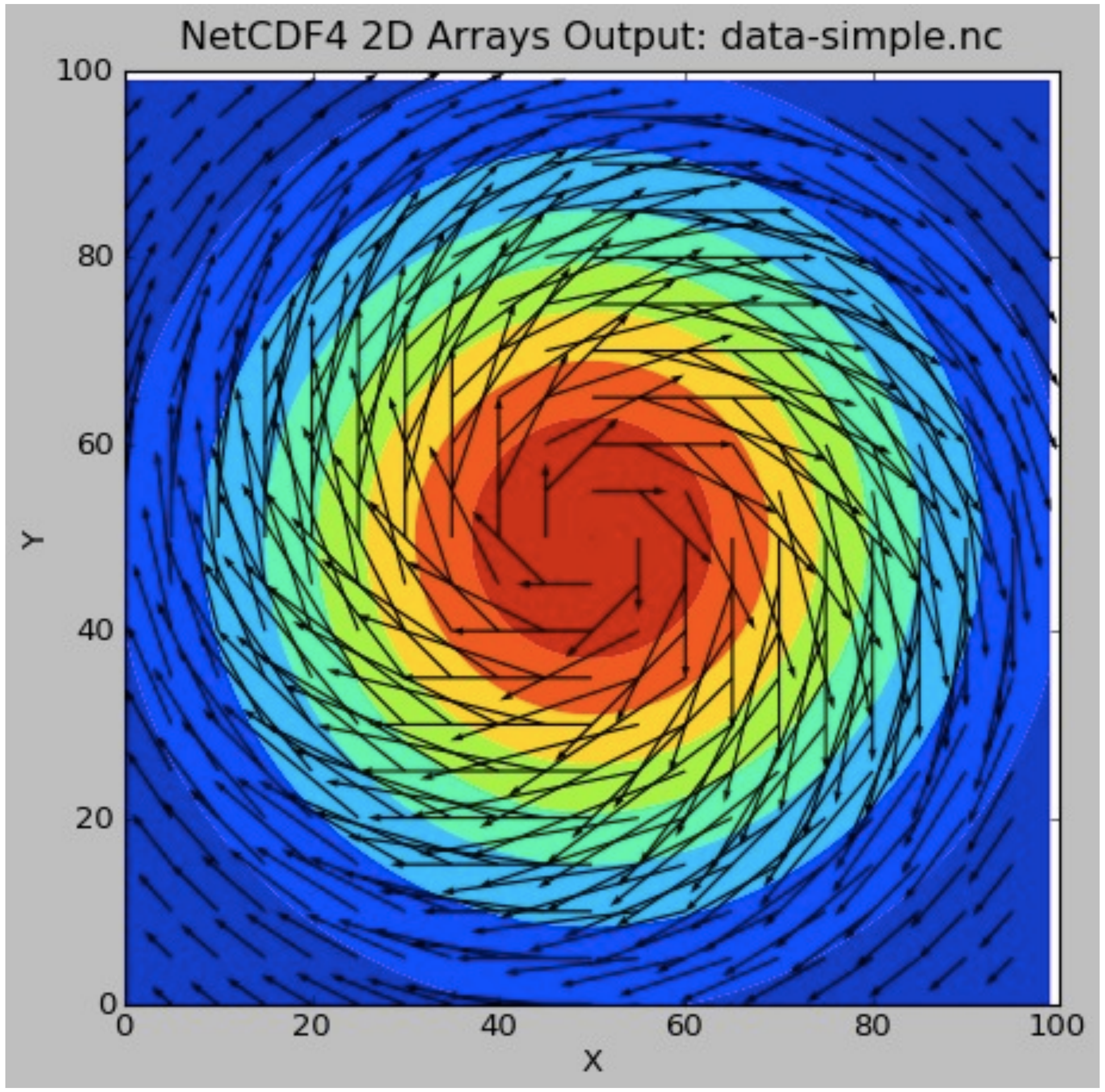
- The above example is much more austere than a typical NetCDF file
- A more typical example is given in 2darray (or f2darray)
- make this, then run it
- ../plots.py data.nc
- (Same options as previous example)

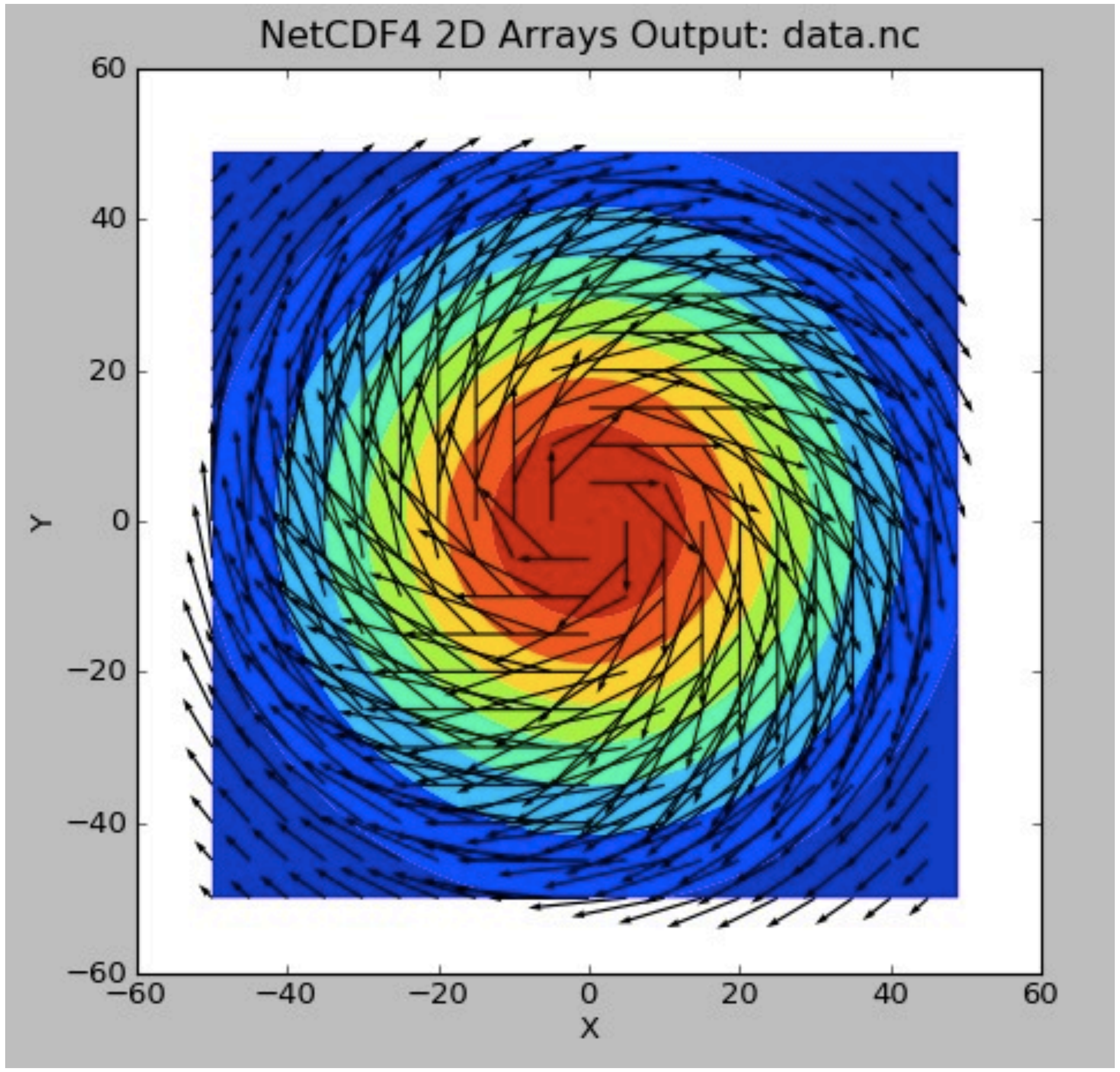
```
$ ncdump -h data.nc
netcdf data {
dimensions:
   X = 100 ;
   Y = 100 ;
   velocity\ component = 2 ;
variables:
   float X\ coordinate(X) ;
      X\ coordinate:units = "cm" ;
   float Y\ coordinate(Y) ;
      Y\ coordinate:units = "cm" ;
   double Density(X, Y) ;
      Density:units = "g/cm^3" ;
   double Velocity(velocity\
component, X, Y) ;
      Velocity:units = "cm/s" ;
}
```

2darray.c

```
float *x, *y;
const char *coordunit="cm";
...
for (i=0; i<rundata.nx; i++) x[i] = (1.*i-r
for (i=0; i<rundata.ny; i++) y[i] = (1.*i-r
...
/* define the dimensions */
nc_def_dim(file_id, "X", rundata.nx, &xdim_id);
nc_def_dim(file_id, "Y", rundata.ny, &ydim_id);
nc_def_dim(file_id, "velocity component", 2, &vcomp_id);

/* define the coordinate variables,... */
nc_def_var(file_id, "X coordinate", NC_FLOAT, 1, &xdim_id, &xcoord_id);
nc_def_var(file_id, "Y coordinate", NC_FLOAT, 1, &ydim_id, &ycoord_id);

/* ...and assign units to them as an attribute */
nc_put_att_text(file_id, xcoord_id, "units", strlen(coordunit), coordunit);
nc_put_att_text(file_id, ycoord_id, "units", strlen(coordunit), coordunit);
```

Typically not only define dimensions but give coordinate values


NetCDF4 2D Arrays Output: data-simple.nc
Y
X
0
20
40
60
80
100


NetCDF4 2D Arrays Output: data.nc
Y
X
−60
−40
−20
0
20
40
60

2darray.c

```
float *x, *y;
const char *coordunit="cm";
...
for (i=0; i<rundata.nx; i++) x[i] = (1.*i-r
for (i=0; i<rundata.ny; i++) y[i] = (1.*i-r
...
/* define the dimensions */
nc_def_dim(file_id, "X", rundata.nx, &xdim_id);
nc_def_dim(file_id, "Y", rundata.ny, &ydim_id);
nc_def_dim(file_id, "velocity component", 2, &vcomp_id);

/* define the coordinate variables,... */
nc_def_var(file_id, "X coordinate", NC_FLOAT, 1, &xdim_id, &xcoord_id);
nc_def_var(file_id, "Y coordinate", NC_FLOAT, 1, &ydim_id, &ycoord_id);

/* ...and assign units to them as an attribute */
nc_put_att_text(file_id, xcoord_id, "units", strlen(coordunit), coordunit);
nc_put_att_text(file_id, ycoord_id, "units", strlen(coordunit), coordunit);
```

Variables (or anything else) can have **attributes:** Name, and arbitrary data

NetCDF Attributes

- Any NetCDF object (data set, dimension) can have an arbitrary number of attributes associated with it
- Name, and any type or size...
- Like a variable! (But can't access only part of it).

```
$ ncdump -h data.nc
netcdf data {
dimensions:
	X = 100 ;
	Y = 100 ;
	velocity\ component = 2 ;
variables:
	float X\ coordinate(X) ;
		X\ coordinate:units = "cm" ;
	float Y\ coordinate(Y) ;
		Y\ coordinate:units = "cm" ;
	double Density(X, Y) ;
		Density:units = "g/cm^3" ;
	double Velocity(velocity\
component, X, Y) ;
		Velocity:units = "cm/s" ;
}
```

NetCDF Attributes

- Attributes are assumed to be "small", though.
- Stored in header information (not with big data)
- Don't put large arrays in there

```
$ ncdump -h data.nc
netcdf data {
dimensions:
    X = 100 ;
    Y = 100 ;
    velocity\ component = 2 ;
variables:
    float X\ coordinate(X) ;
        X\ coordinate:units = "cm" ;
    float Y\ coordinate(Y) ;
        Y\ coordinate:units = "cm" ;
    double Density(X, Y) ;
        Density:units = "g/cm^3" ;
    double Velocity(velocity\
component, X, Y) ;
        Velocity:units = "cm/s" ;
}
```

NetCDF Attributes

- Units are particularly useful attributes, as if a code needs data in some other units (MKS), can convert.

```
$ ncdump -h data.nc
netcdf data {
dimensions:
   X = 100 ;
   Y = 100 ;
   velocity\ component = 2 ;
variables:
   float X\ coordinate(X) ;
      X\ coordinate:units = "cm" ;
   float Y\ coordinate(Y) ;
      Y\ coordinate:units = "cm" ;
   double Density(X, Y) ;
      Density:units = "g/cm^3" ;
   double Velocity(velocity\
component, X, Y) ;
      Velocity:units = "cm/s" ;
}
```

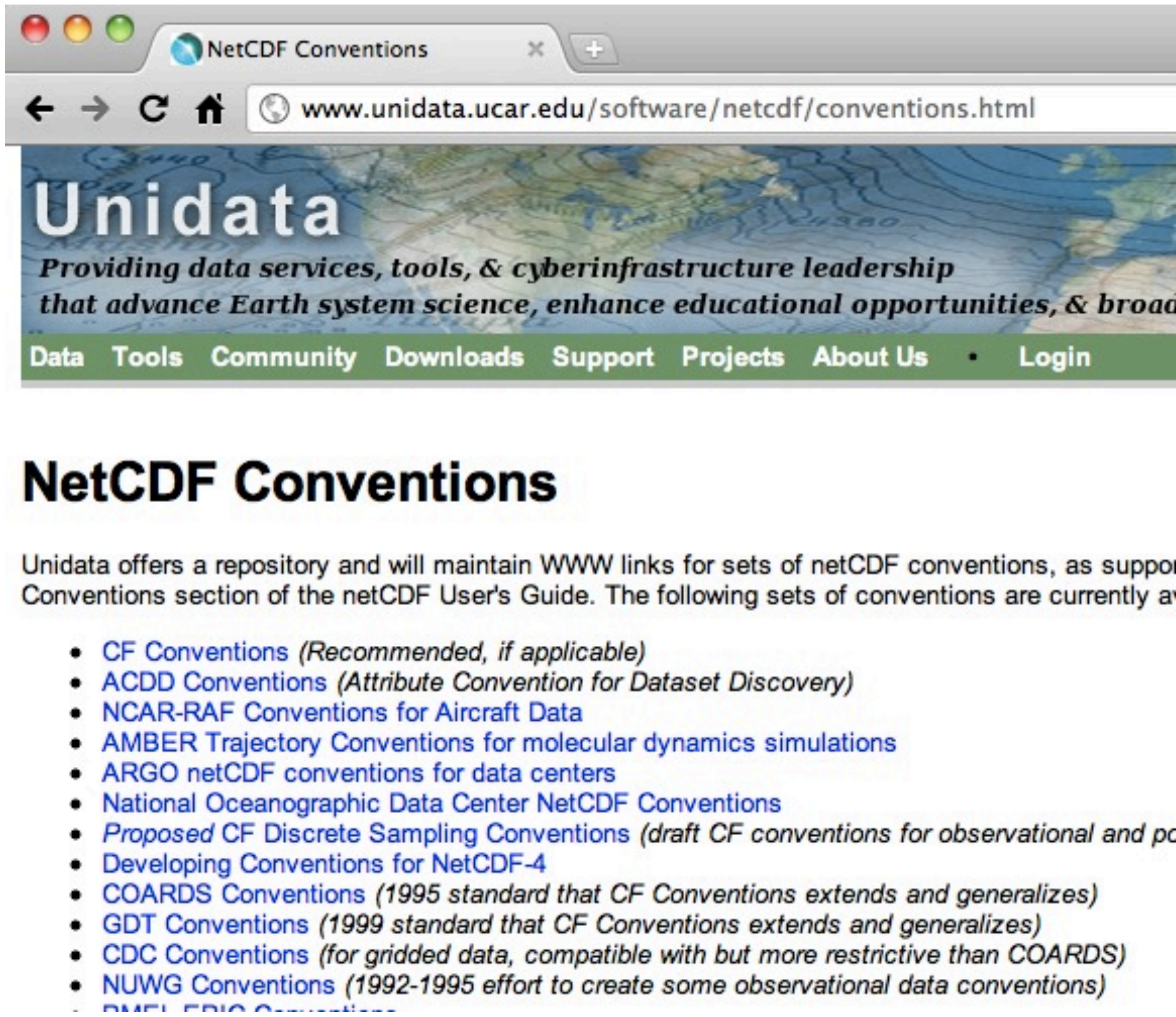
Limits to Self-Description

- But what if some codes expect "centimetre" and you use cm?
- Or their code uses "Dens" or "Rho" and yours uses "Density?" Or uses momentum rather than velocity?

```
$ ncdump -h data.nc
netcdf data {
dimensions:
   X = 100 ;
   Y = 100 ;
   velocity\ component = 2 ;
variables:
   float X\ coordinate(X) ;
      X\ coordinate:units = "cm" ;
   float Y\ coordinate(Y) ;
      Y\ coordinate:units = "cm" ;
   double Density(X, Y) ;
      Density:units = "g/cm^3" ;
   double Velocity(velocity\
component, X, Y) ;
      Velocity:units = "cm/s" ;
}
```

Conventions

- There are lists of conventions that you can follow for variable names, unit names, etc.
- If you are planning for interoperability with other codes, this is the way to go
- Codes expecting data following (say) CF conventions for geophys should recognize data in that convention


NetCDF Conventions

www.unidata.ucar.edu/software/netcdf/conventions.html

Unidata

Providing data services, tools, & cyberinfrastructure leadership
that advance Earth system science, enhance educational opportunities, & broad

Data Tools Community Downloads Support Projects About Us • Login

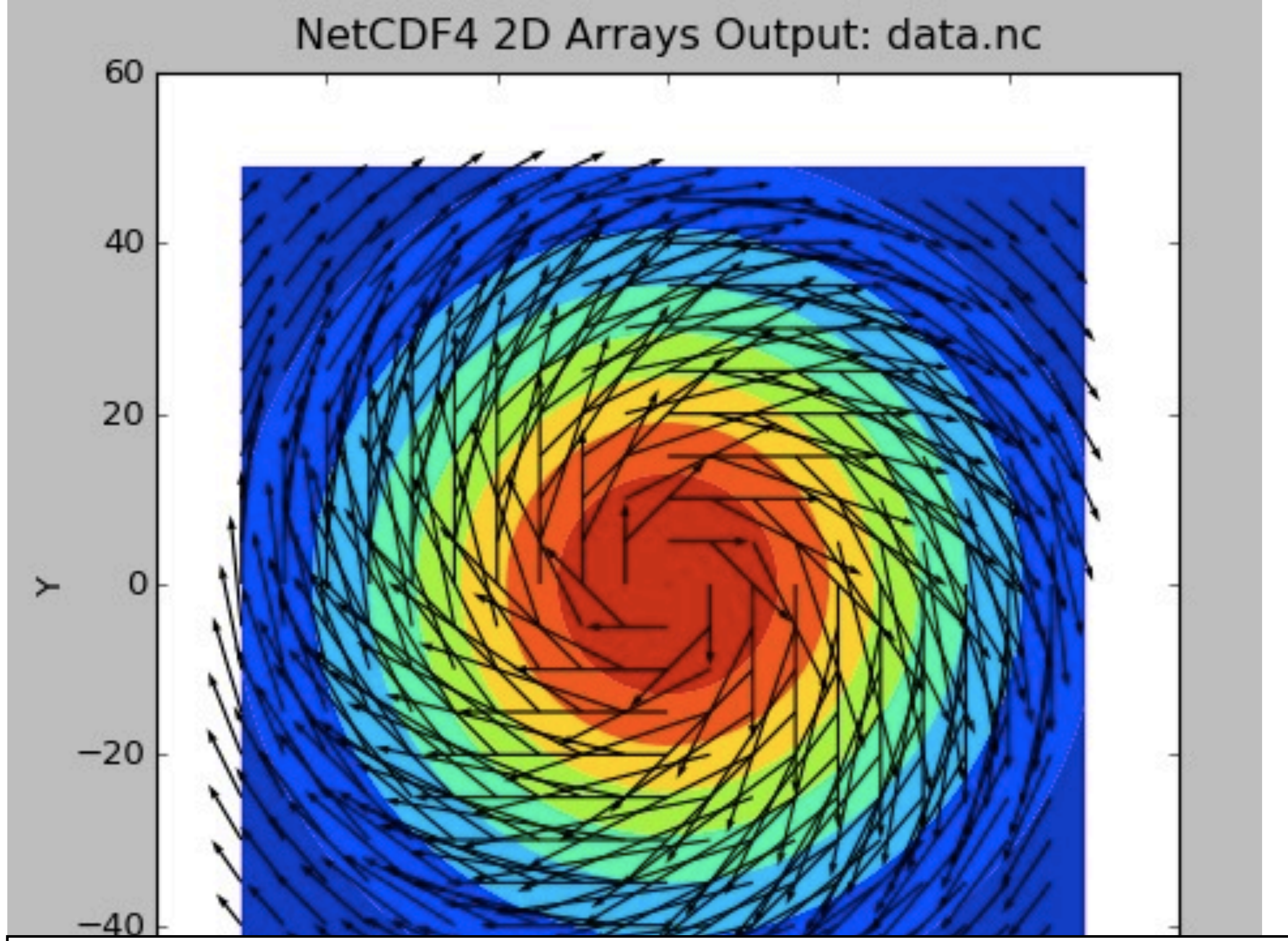
NetCDF Conventions

Unidata offers a repository and will maintain WWW links for sets of netCDF conventions, as suppo
Conventions section of the netCDF User's Guide. The following sets of conventions are currently a

- CF Conventions *(Recommended, if applicable)*
- ACDD Conventions *(Attribute Convention for Dataset Discovery)*
- NCAR-RAF Conventions for Aircraft Data
- AMBER Trajectory Conventions for molecular dynamics simulations
- ARGO netCDF conventions for data centers
- National Oceanographic Data Center NetCDF Conventions
- *Proposed* CF Discrete Sampling Conventions *(draft CF conventions for observational and po*
- Developing Conventions for NetCDF-4
- COARDS Conventions *(1995 standard that CF Conventions extends and generalizes)*
- GDT Conventions *(1999 standard that CF Conventions extends and generalizes)*
- CDC Conventions *(for gridded data, compatible with but more restrictive than COARDS)*
- NUWG Conventions *(1992-1995 effort to create some observational data conventions)*

Big advantage of self-describing:

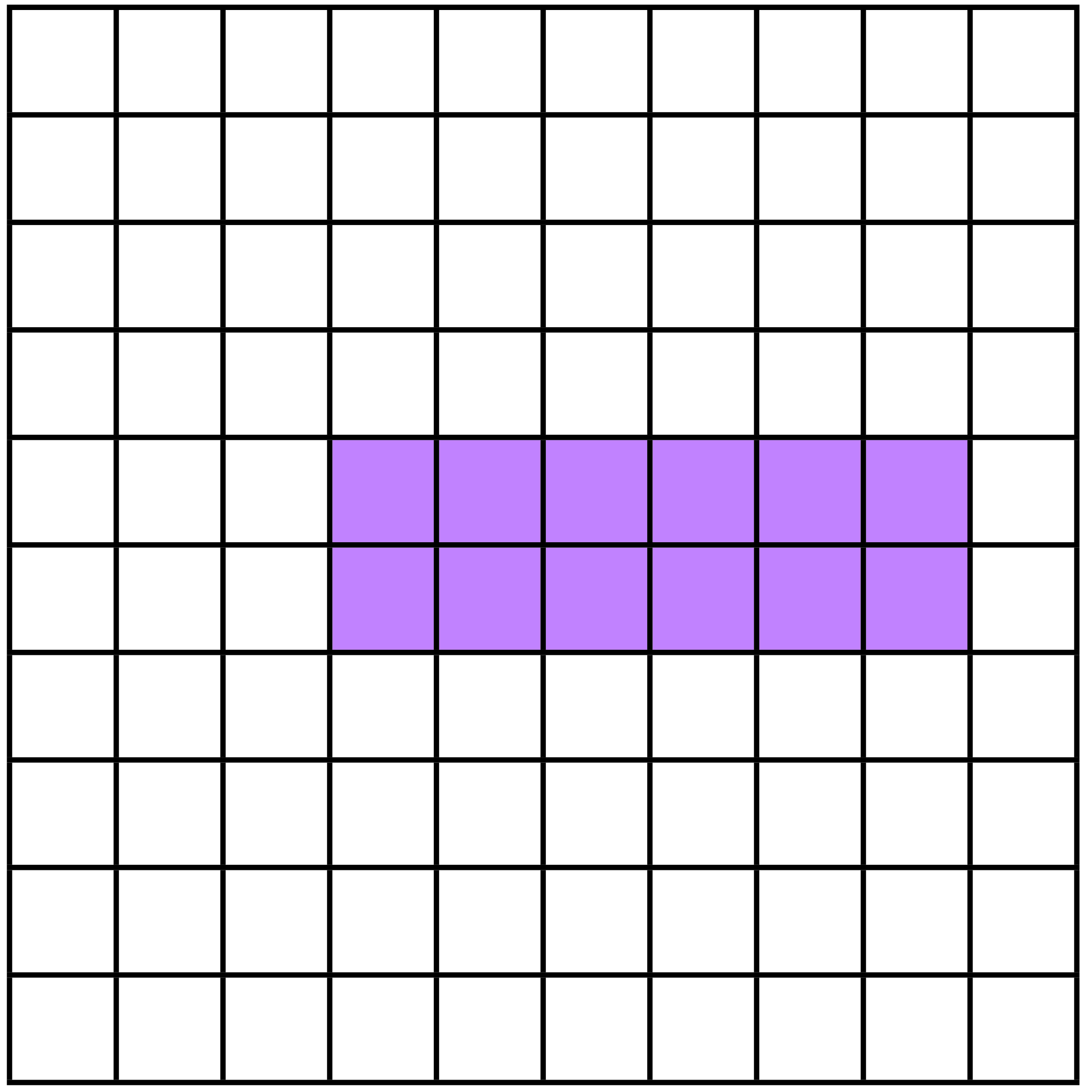
- Old program could easily read new file, even though data layout changed!
- Doesn't even need to know about attributes...
- New variables don't cause any problems - don't have to read them!
- Backwards compatibility
- But can look for them and use if available.



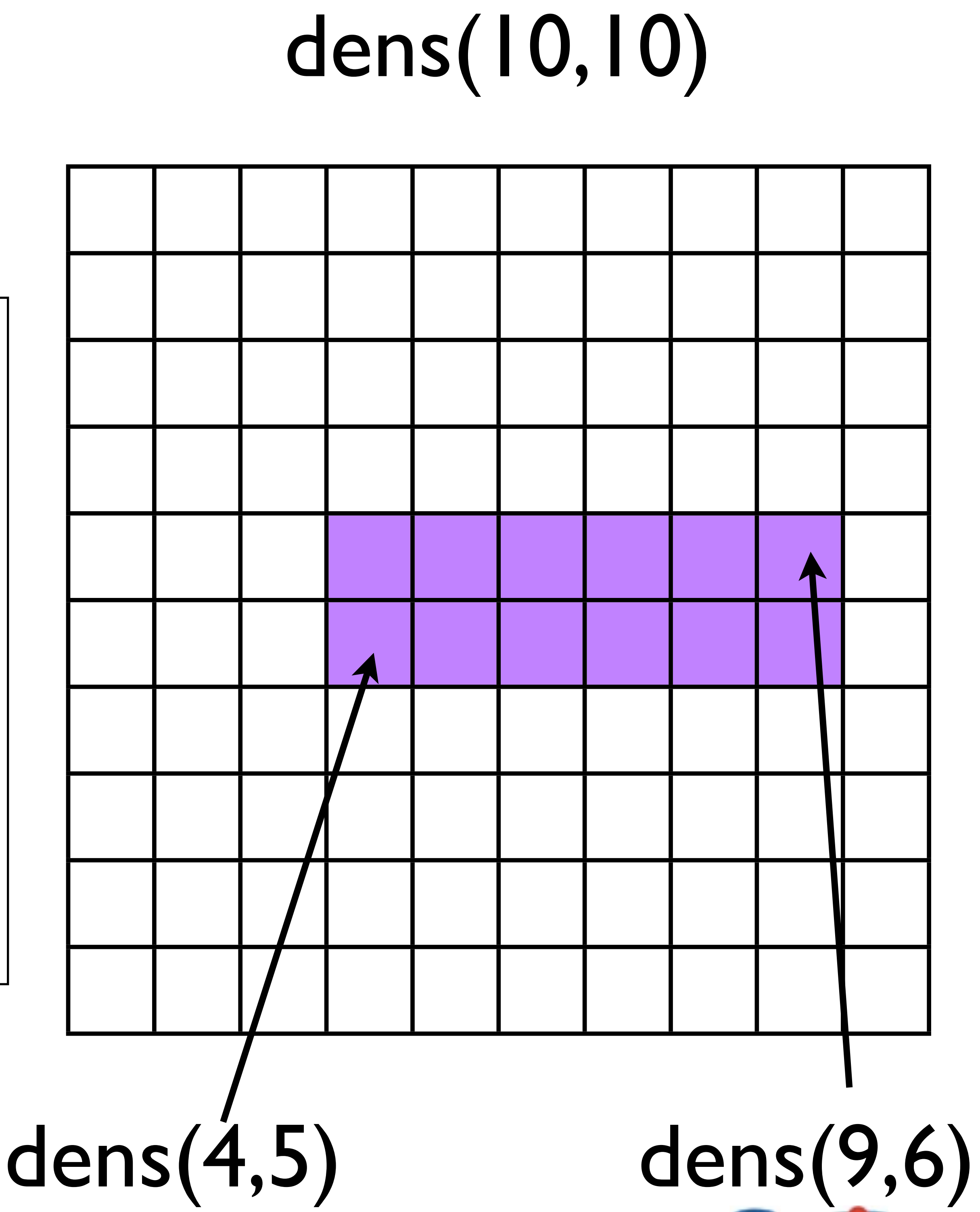
```
$ ncdump -h data.nc
netcdf data {
dimensions:
    X = 100 ;
    Y = 100 ;
    velocity\ component = 2 ;
variables:
    float X\ coordinate(X) ;
        X\ coordinate:units = "cm" ;
    float Y\ coordinate(Y) ;
        Y\ coordinate:units = "cm" ;
    double Density(X, Y) ;
        Density:units = "g/cm^3" ;
    double Velocity(velocity\ component, X, Y
        Velocity:units = "cm/s" ;
}
```

Accessing subregions in file

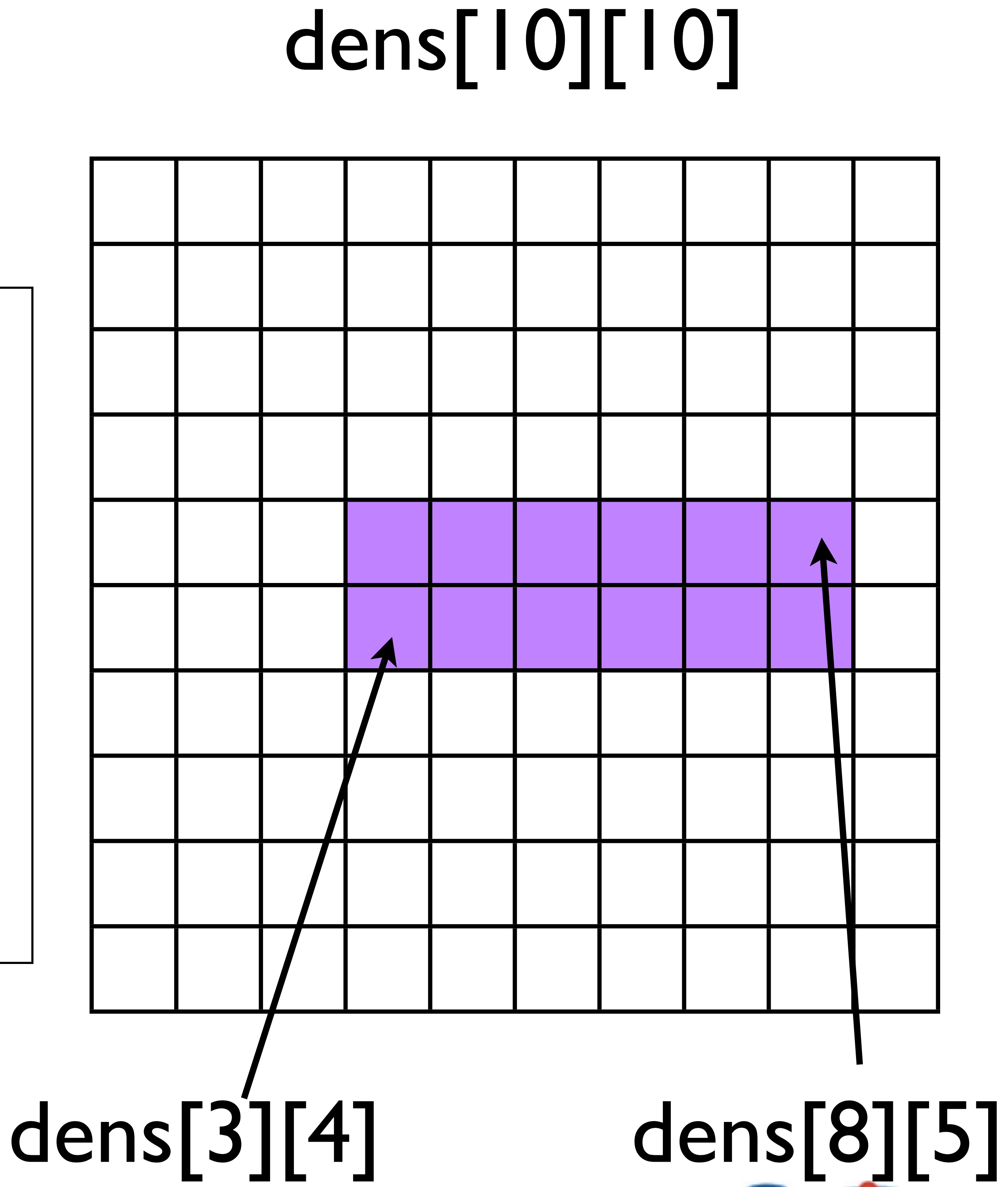
nc_put_var_type or nf90_put_var puts whole array(by default)
Subarrays can be specified with starts and counts



```
start(1) = 4
start(2) = 5

count(1) = 6
count(2) = 2

nf90_put_var(file_id, dens_id,
data, START=start, COUNT=count)
```

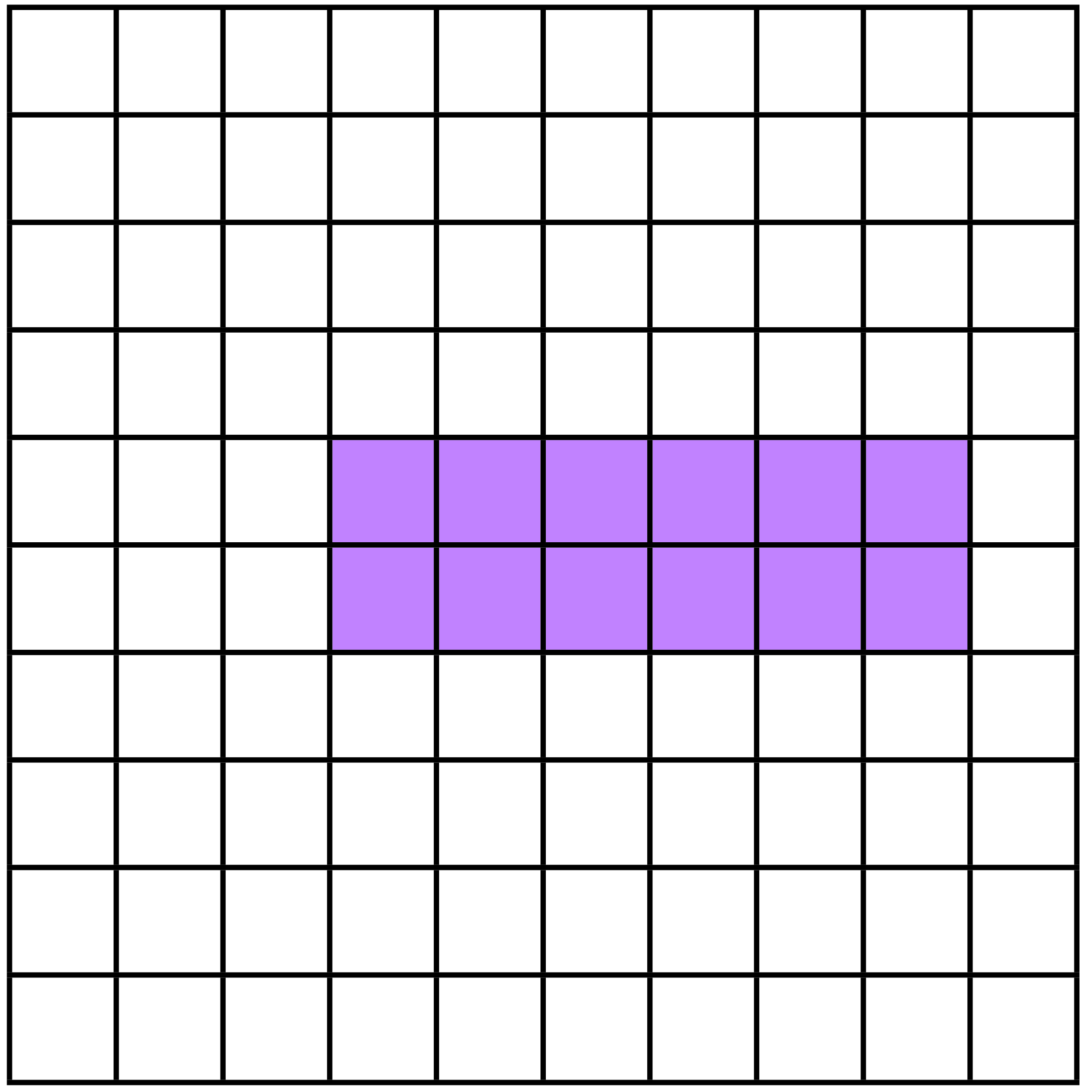


```
start[0] = 3;
start[1] = 4;

count[0] = 6;
count[1] = 2;

nc_put_vara_double(file_id,
dens_id, start, count, data);
```



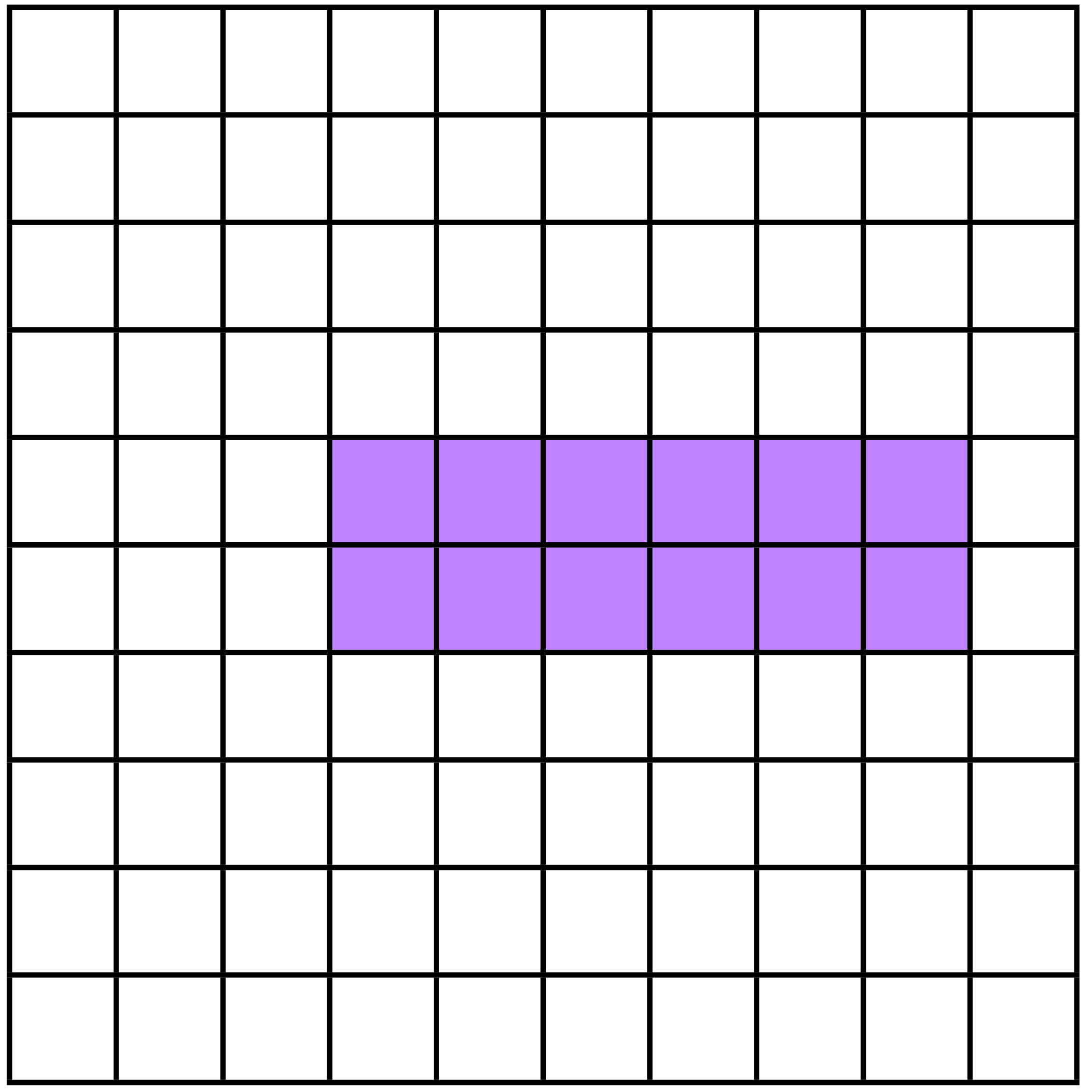
Accessing subregions in file

Note that NetCDF libraries accepts starting conventions of C, Fortran as appropriate.



Accessing subregions in file

Another thing this is good for; arrays in NetCDF can have a dimension of unlimited size (eg, can grow) - NetCDF3, only one dimension, NetCDF4, any
Can use for timesteps, for instance.
Any access to such a dataset is necessarily via subregions.



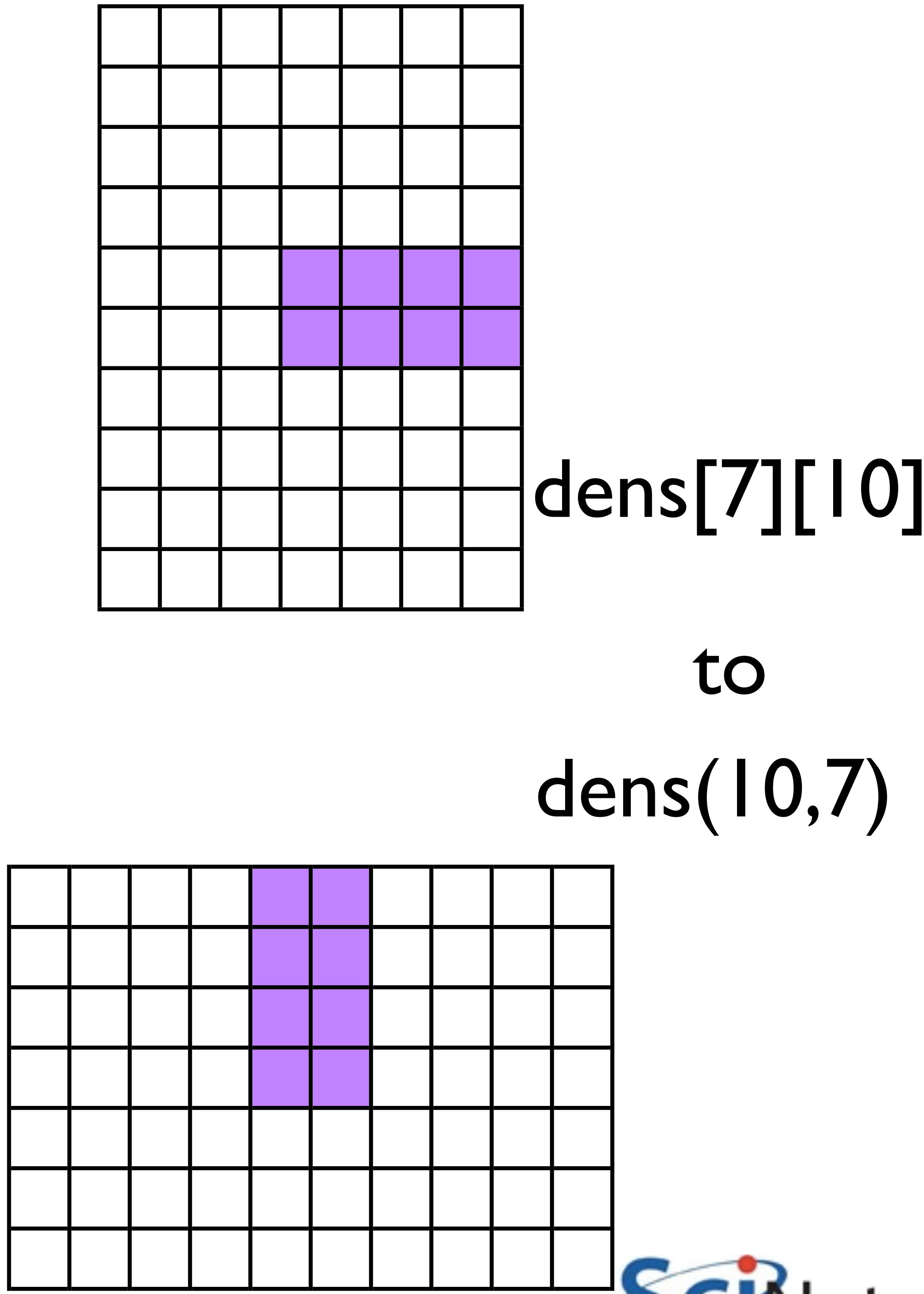
Fortran vs C array conventions

```
$ ncdump -h data.nc
netcdf data {
dimensions:
  X = 100 ;
  Y = 100 ;
  velocity\ component = 2 ;
variables:
  float X\ coordinate(X) ;
      X\ coordinate:units = "cm" ;
  float Y\ coordinate(Y) ;
      Y\ coordinate:units = "cm" ;
  double Density(X, Y) ;
      Density:units = "g/cm^3" ;
  double Velocity(velocity\
component, X, Y) ;
      Velocity:units = "cm/s" ;
}
```

```
$ ncdump -h data-fort.nc
netcdf data-fort {
dimensions:
  X = 100 ;
  Y = 100 ;
  velocity\ components = 2 ;
variables:
  float X\ coordinate(X) ;
      X\ coordinate:units = "cm" ;
  float Y\ coordinate(Y) ;
      Y\ coordinate:units = "cm" ;
  double Density(Y, X) ;
      Density:units = "g/cm^3" ;
  double Velocity(Y, X, velocity\
components) ;
      Velocity:units = "cm/s" ;
}
```

Mapping memory space to file

Say in C you wanted to output in FORTRAN convention (i,j) in your array corresponds to (j,i) in data space in file nc_put_var**m** allows you to do this by mapping how indicies vary in memory compared to in file.

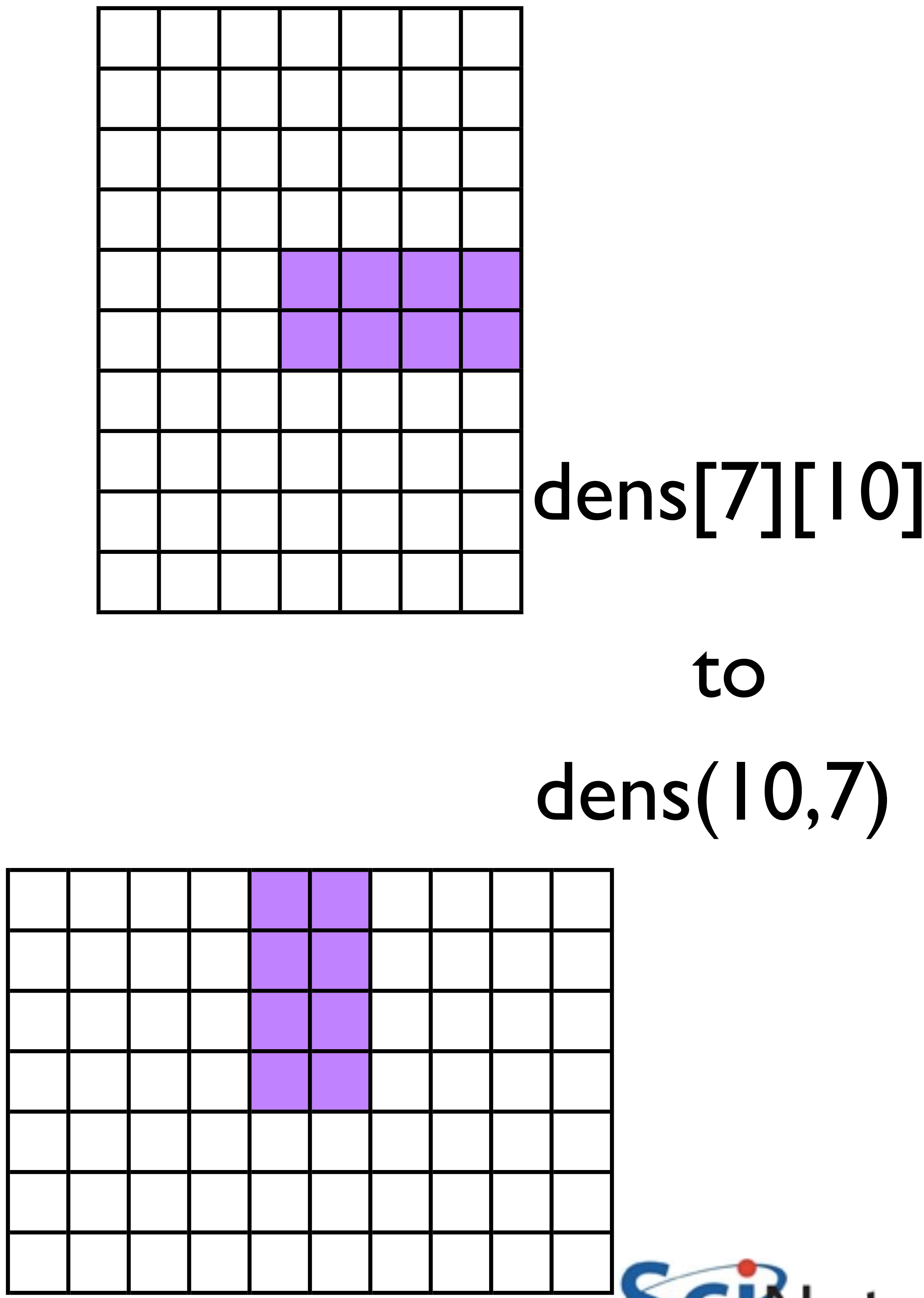


Mapping memory space to file

Note - this requires understanding how memory is laid out in your data structures, as with MPI & MPI-IO

This is *crucial* for I/O, and for HPC in general

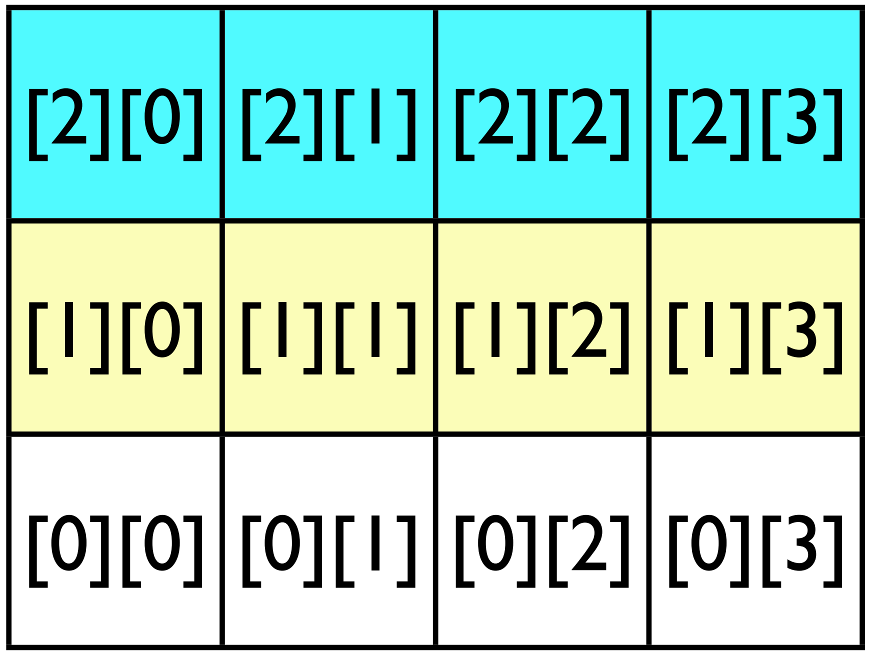
C has more flexibility (== potential problems) in this regard.



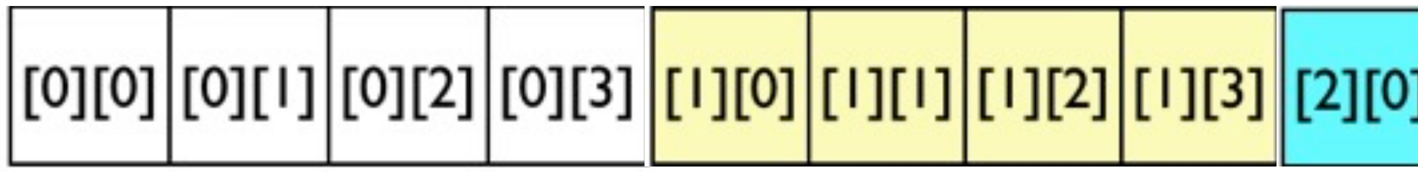
Mapping memory space to file

C: first array index most slowly varying.
Eg, for a 3x4 array, each step in the 2nd index jumps you one position in memory,
and in the first index, jumps yo
by 4.
You could write this as (4,1)

Your picture of the array



In memory

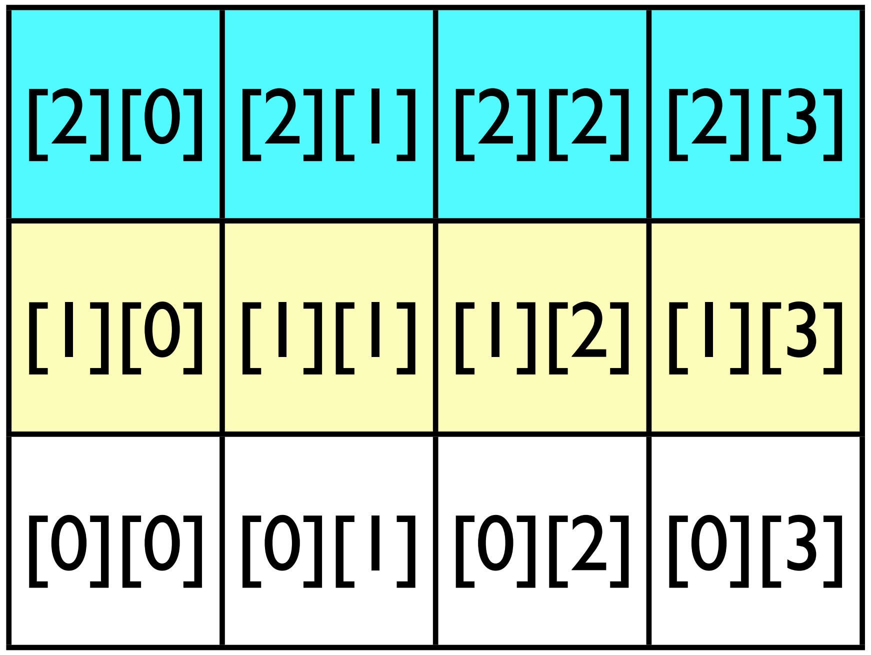


Mapping memory space to file

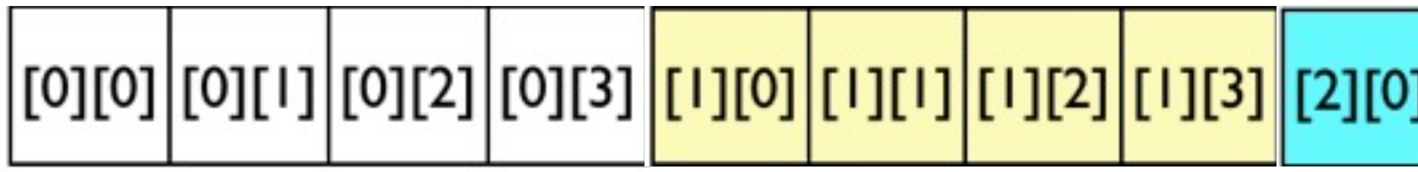
But if you're writing to a fortran-convention file, you want this to go the other way

In the file, one step in the **1st** index should jump you by 1, and the second by...

Your picture of the array



In memory



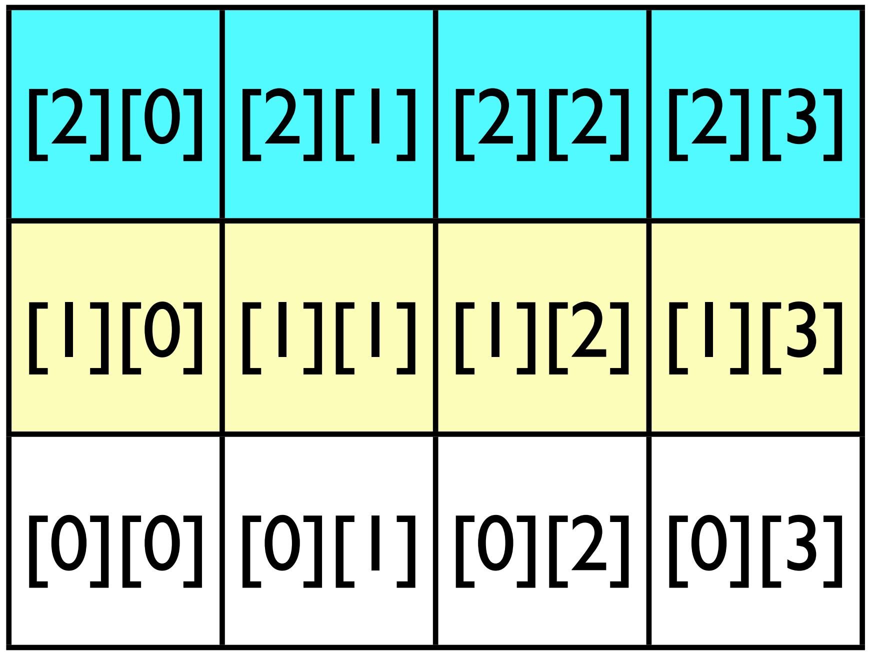
Mapping memory space to file

But if you're writing to a fortran-convention file, you want this to go the other way

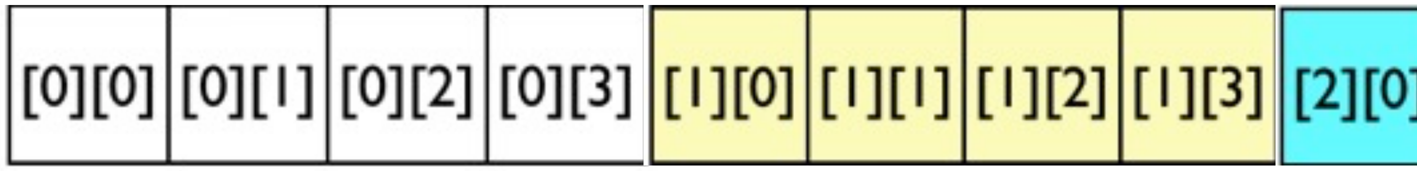
In the file, one step in the **1st** index should jump you by 1, and the second by **3**.

The map you want is (1,3)

Your picture of the array



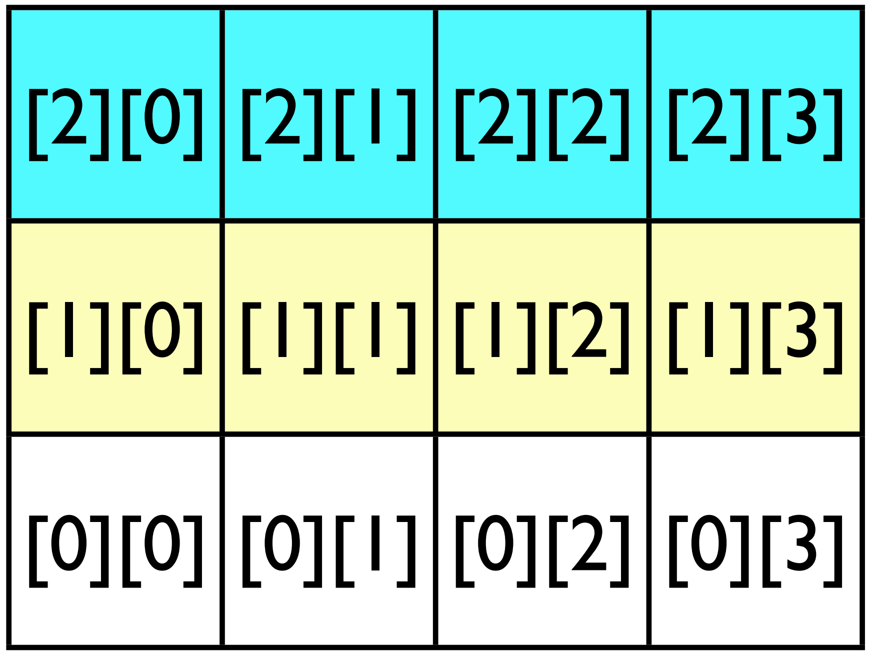
In memory



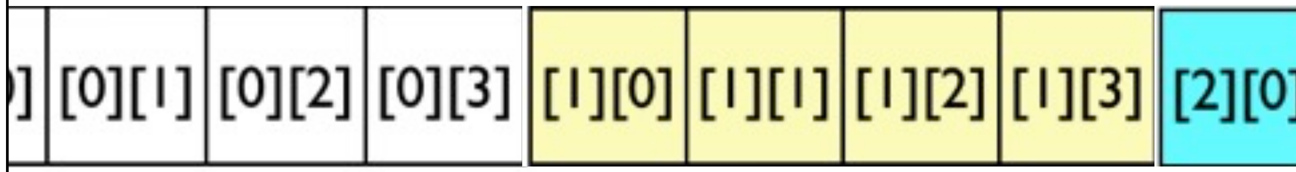
Mapping memory space to file

```
start = count = stride = NULL;
int imap[2] = {1,3};

nc_put_varm_double(file_id,
dens_id, start, count, stride,
imap, data);


nf90_put_var(file_id, dens_id,
data, MAP=(/4,1/))
```

Your picture of the array



In memory



More on NetCDF

http://www.unidata.ucar.edu/software/netcdf/

Docs, mailing lists, tutorials, sample code, API, etc.

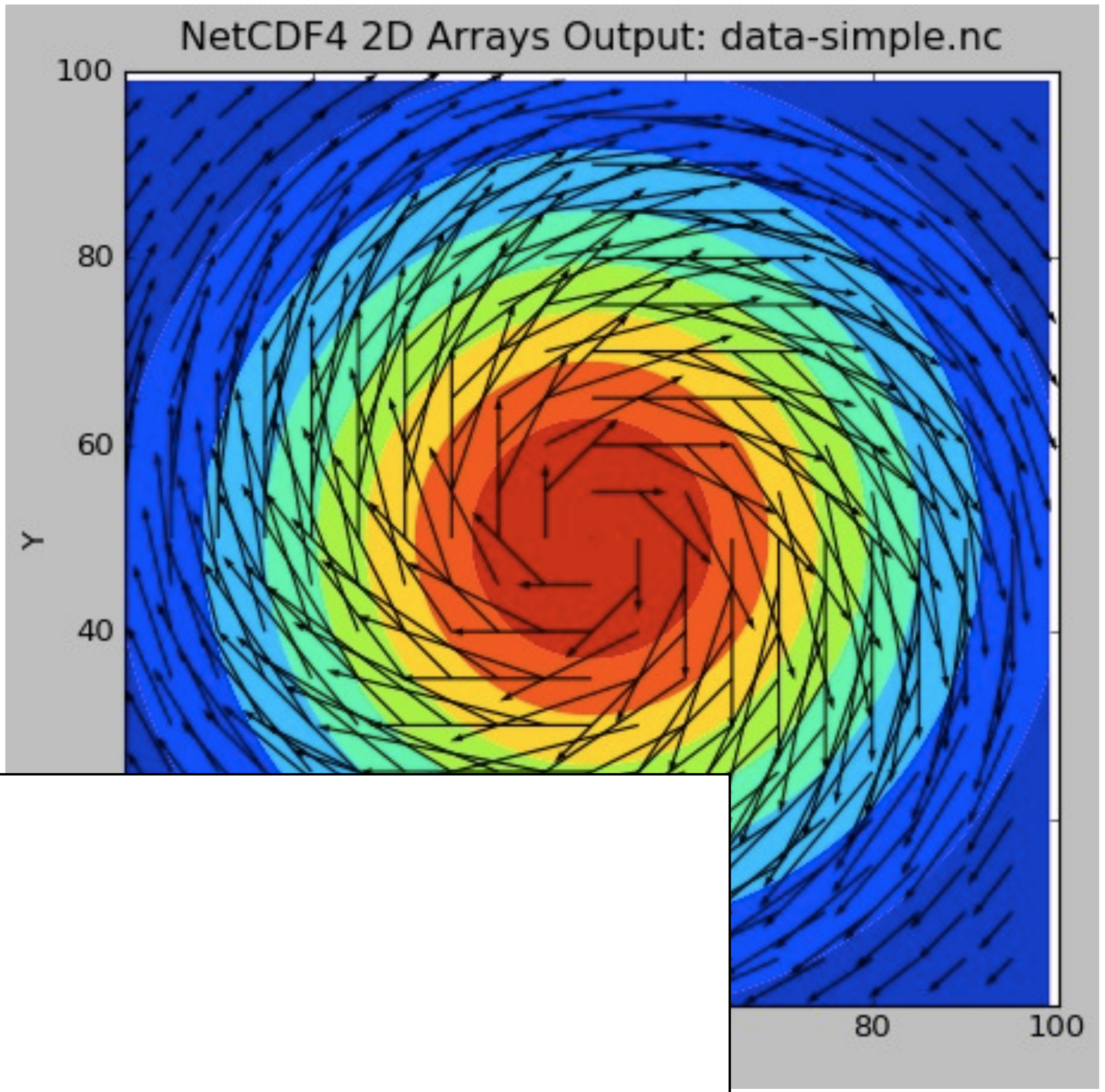
Sample Code

The HDF Group

http://www.hdfgroup.org/HDF5/

```
$ cd parIO/hdf5

$ source ../parallellibs
$ make serial or
$ make 2darray (C), or
$ make f2darray (F90)

$ ./{f,}2darray
$ ls *.h5

$ ../plots.py *.h5
```

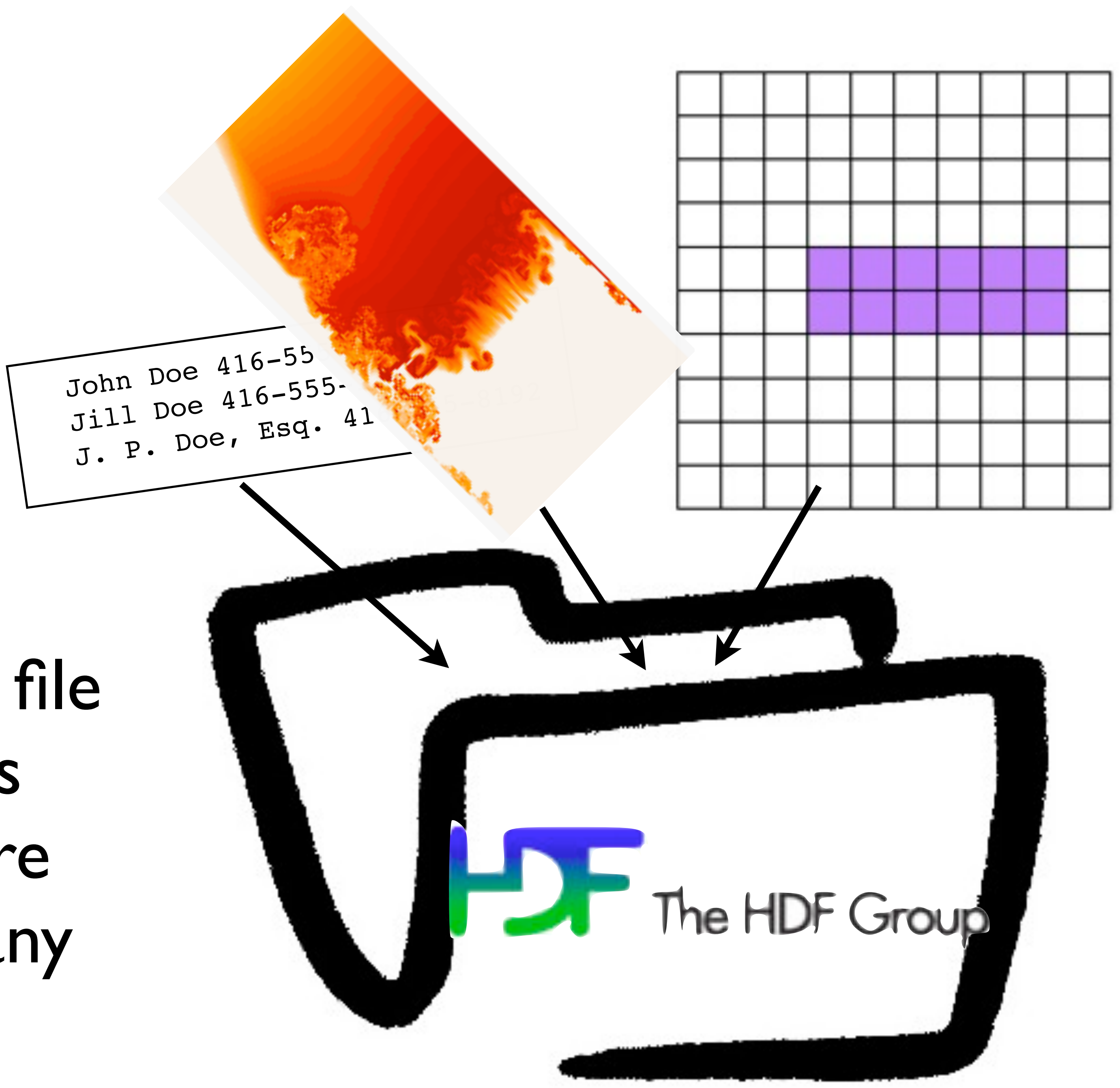
What is this .h5 file?



```
$ h5ls data-fort.h5
ArrayData                Group
OtherStuff               Group

$ h5ls data-fort.h5/ArrayData
dens                     Dataset {100, 100}
vel                      Dataset {100, 100, 2}
```

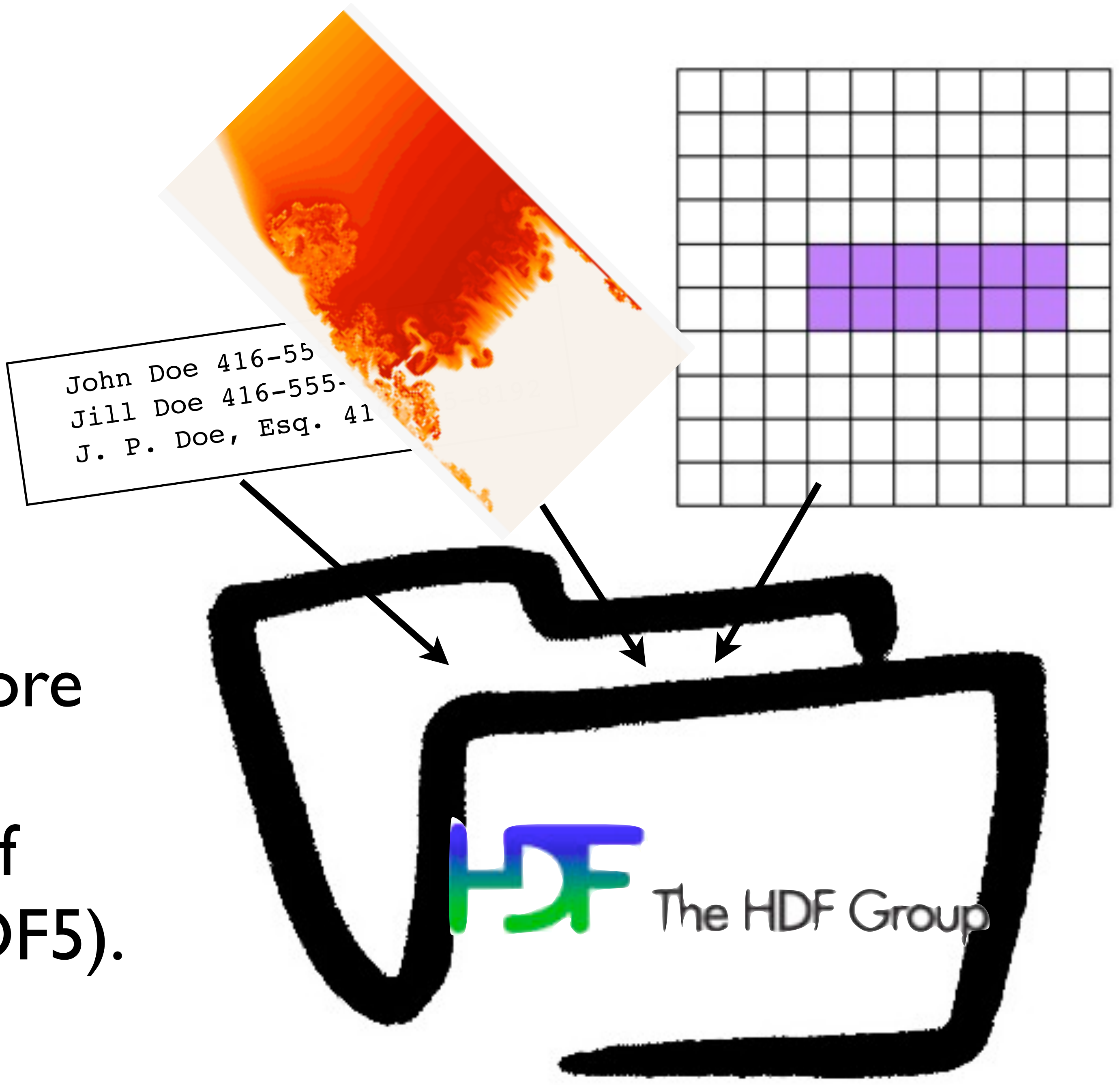
HDF5

HDF5 is also self-describing file format and set of libraries
Unlike NetCDF, much more general; can shove almost any type of data in there
(We'll just be looking at large arrays, since that's our usual use case)



HDF5

Much more general, and more low-level than NetCDF.
(In fact, newest version of NetCDF implemented in HDF5).
Pro: *can* do more!
Con: **have** to do more.



2darray-simple.c

```
/* identifiers */
hid_t file_id, dens_dataset_id, vel_dataset_id;
hid_t dens_dataspace_id, vel_dataspace_id;

/* sizes */
hsize_t densdims[2], veldims[3];

/* status */
herr_t status;

/* Create a new file - truncate anything existing, use default properties
*/
file_id = H5Fcreate(rundata.filename, H5F_ACC_TRUNC, H5P_DEFAULT,
H5P_DEFAULT);

/* HDF5 routines generally return a negative number on failure.
 * Should check return values! */
if (file_id < 0) {
    fprintf(stderr,"Could not open file %s\n", rundata.filename);
    return;}
```

2darray-simple.c

```
/* identifiers */
hid_t file_id, dens_dataset_id, vel_dataset_id;
hid_t dens_dataspace_id, vel_dataspace_id;

/* sizes */
hsize_t densdims[2], veldims[3];

/* status */
herr_t status;

/* Create a new file - truncate anything
*/
file_id = H5Fcreate(rundata.filename, H5F_ACC_TRUNC, H5P_DEFAULT,
H5P_DEFAULT);

/* HDF5 routines generally return a negative number on failure.
 * Should check return values! */
if (file_id < 0) {
    fprintf(stderr,"Could not open file %s\n", rundata.filename);
    return;}
```

NetCDF used ints for everything - HDF5 distinguishes between ids, sizes, errors, uses its own types.

2darray-simple.c

```
/* identifiers */
hid_t file_id, dens_dataset_id, vel_dataset_id;
hid_t dens_dataspace_id, vel_dataspace_id;


/* sizes */
hsize_t densdims[2], veldims[3];


/* status */
herr_t status;


/* Create a new file - truncate anything existing, use default properties
*/
file_id = H5Fcreate(rundata.filename, H5F_ACC_TRUNC, H5P_DEFAULT,
H5P_DEFAULT);


/* HDF5 routines generally return a negative number on failure.
 * Should check return values! */
if (file_id < 0) {
    fprintf(stderr,"Could not open file %s\n", rundata.filename);
    return;}
```

H5F, H5P... ?

Decomposing the HDF5 API

HDF5 API is large

Constants, function calls start with H5x; x tells you what part of the library

Table tells you (some) of those parts...

Fortran the same, but usually end with _F

H5A	**A**ttributes
H5D	**D**atasets
H5E	**E**rrors
H5F	**F**iles
H5G	**G**roups
H5P	**P**roperties
H5S	Data **S**paces
H5T	Data **T**ypes

2darray-simple.c

```
/* Create the data space for the two datasets. */
densdims[0] = rundata.nx; densdims[1] = rundata.ny;
veldims[0] = 2; veldims[1] = rundata.nx; veldims[2] = rundata.ny;

dens_dataspace_id = H5Screate_simple(2, densdims, NULL);
vel_dataspace_id  = H5Screate_simple(3, veldims,  NULL);

/* Create the datasets within the file.
 * H5T_IEEE_F64LE is a standard (IEEE) double precision (64 bit)
 * floating (F) data type and will work on any machine.
 * H5T_NATIVE_DOUBLE would work too */

dens_dataset_id = H5Dcreate(file_id, "dens", H5T_IEEE_F64LE,
                                dens_dataspace_id, H5P_DEFAULT,
                                H5P_DEFAULT, H5P_DEFAULT);

vel_dataset_id  = H5Dcreate(file_id, "vel",  H5T_IEEE_F64LE,
                                vel_dataspace_id,  H5P_DEFAULT,
                                H5P_DEFAULT, H5P_DEFAULT);
```

2darray-simple.c

```
/* Create the data space for the two datasets. */
densdims[0] = rundata.nx; densdims[1] = rundata.ny;
veldims[0] = 2; veldims[1] = rundata.nx; veldims[2] = rundata.ny;

dens_dataspace_id = H5Screate_simple(2, densdims, NULL);
vel_dataspace_id  = H5Screate_simple(3, veldims,  NULL);

/* Create the datasets within the file.
 * H5T_IEEE_F64LE is a standard (IEEE)
 * floating (F) data type and will work
 * H5T_NATIVE_DOUBLE would work too */

dens_dataset_id = H5Dcreate(file_id, "d
                                dens_da
                                H5P_DEF

vel_dataset_id  = H5Dcreate(file_id, "v
                                vel_dat
                                H5P_DEF
```

All data (in file or in mem) in HDF5 has a dataspace it lives in.

In NetCDF, just cartesian product of dimensions; here more general

2darray-simple.c

```
/* Create the data space for the two datasets. */
densdims[0] = rundata.nx; densdims[1] =
veldims[0] = 2; veldims[1] = rundata.nx

dens_dataspace_id = H5Screate_simple(2,
vel_dataspace_id  = H5Screate_simple(3,

/* Create the datasets within the file.
 * H5T_IEEE_F64LE is a standard (IEEE) 
 * floating (F) data type and will work
 * H5T_NATIVE_DOUBLE would work too */

dens_dataset_id = H5Dcreate(file_id, "dens", H5T_IEEE_F64LE,
                                dens_dataspace_id, H5P_DEFAULT,
                                H5P_DEFAULT, H5P_DEFAULT);

vel_dataset_id  = H5Dcreate(file_id, "vel",  H5T_IEEE_F64LE,
                                vel_dataspace_id,  H5P_DEFAULT,
                                H5P_DEFAULT, H5P_DEFAULT);
```

Creating a data set like defining a variable in NetCDF.
Also declare the type you want it to be on disk.

2darray-simple.c

```
/* Write the data.  We're writing it from memory, where it is saved
 * in NATIVE_DOUBLE format */
status = H5Dwrite(dens_dataset_id, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL,
H5P_DEFAULT, &(dens[0][0]));
status = H5Dwrite(vel_dataset_id,  H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL,
H5P_DEFAULT, &(vel[0][0][0]));


/* End access to groups & data sets and release resources used by them */
status = H5Sclose(dens_dataspace_id);
status = H5Dclose(dens_dataset_id);
status = H5Sclose(vel_dataspace_id);
status = H5Dclose(vel_dataset_id);


/* Close the file */
status = H5Fclose(file_id);
```

Write memory from all of memory to all of the dataset on the file. Values in mem are in the native double precision format.

2darray-simple.c

```
/* Write the data.  We're writing it from memory, where it is saved
 * in NATIVE_DOUBLE format */
status = H5Dwrite(dens_dataset_id, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL,
H5P_DEFAULT, &(dens[0][0]));
status = H5Dwrite(vel_dataset_id,  H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL,
H5P_DEFAULT, &(vel[0][0][0]));


/* End access to groups & data sets and release resources used by them */
status = H5Sclose(dens_dataspace_id);
status = H5Dclose(dens_dataset_id);
status = H5Sclose(vel_dataspace_id);
status = H5Dclose(vel_dataset_id);

/* Close the file */
status = H5Fclose(file_id);
```

Close everything

f2darray-simple.f90

```
integer(hid_t) :: file_id
integer(hid_t) :: dens_space_id, vel_space_id
integer(hid_t) :: dens_id, vel_id
integer(hsize_t), dimension(2) :: densdims
integer(hsize_t), dimension(3) :: veldims

integer :: status

! first we have to open the FORTRAN inter
call h5open_f(status)

! create the file, check return code
call h5fcreate_f(rundata%filename, H5F_ACC_TRUNC_F, file_id, status)
if (status /= 0) then
    print *,'Could not open file ', rundata%filename
    return
endif
```

Fortran: values are integer(hid_t) or integer(hsize_t)

f2darray-simple.f90

```
integer(hid_t) :: file_id
integer(hid_t) :: dens_space_id, vel_space_id
integer(hid_t) :: dens_id, vel_id
integer(hsize_t), dimension(2) :: densdims
integer(hsize_t), dimension(3) :: veldims

integer :: status

! first we have to open the FORTRAN interface.
call h5open_f(status)

! create the file, check return code
call h5fcreate_f(rundata%filename, H5F_ACC_TRUNC_F, file_id, status)
if (status /= 0) then
    print *,'Could not open file ', rundata%filename
    return
endif
```

Have to start the FORTRAN interface

f2darray-simple.f90

```
integer(hid_t) :: file_id
integer(hid_t) :: dens_space_id, vel_space_id
integer(hid_t) :: dens_id, vel_id
integer(hsize_t), dimension(2) :: densdims
integer(hsize_t), dimension(3) :: veldims

integer :: status

! first we have to open the FORTRAN interface.
call h5open_f(status)

! create the file, check return code
call h5fcreate_f(rundata%filename, H5F_ACC_TRUNC_F, file_id, status)
if (status /= 0) then
    print *,'Could not open file ', rundata%filename
    return
endif
```

See what I mean about _F?

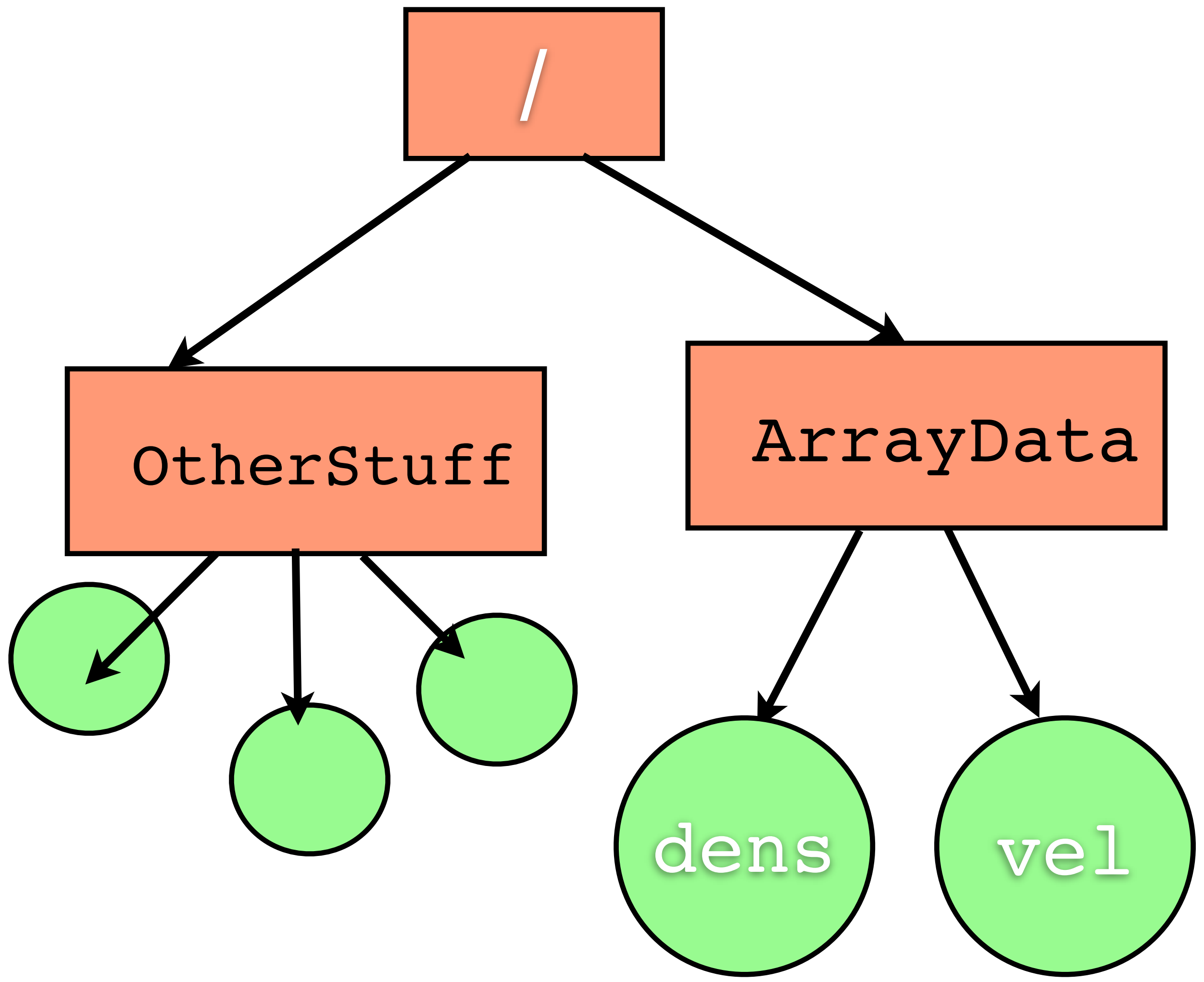
f2darray-simple.f90

```
! create the dataspaces corresponding to our variables
densdims = (/ rundata % nx, rundata % ny /)
call h5screate_simple_f(2, densdims, dens_space_id, status)

veldims = (/ 2, rundata % nx, rundata % ny /)
call h5screate_simple_f(3, veldims, vel_space_id, status)

! now that the dataspaces are defined, we can define variables on them

call h5dcreate_f(file_id, "dens", H5T_IEEE_F64LE, dens_space_id, dens_id,
status)
call h5dcreate_f(file_id, "vel" , H5T_IEEE_F64LE, vel_space_id,  vel_id,
status)
```

In F90 interface, a lot of less-common arguments are optional; fewer H5P_DEFAULTs kicking around

HDF5 Groups

HDF5 has a structure a bit like a unix filesystem:
"Groups" - directories
"Datasets" - files
NetCDF4 now has these, but breaks compatibility with NetCDF3 files



2darray.c

```
/* Create a new group within the new file */
arr_group_id = H5Gcreate(file_id,"/ArrayData", H5P_DEFAULT, H5P_DEFAULT,
H5P_DEFAULT);
...


dens_dataset_id = H5Dcreate(file_id, "/ArrayData/dens", H5T_IEEE_F64LE,
                                dens_dataspace_id, H5P_DEFAULT,
                                H5P_DEFAULT, H5P_DEFAULT);
vel_dataset_id  = H5Dcreate(file_id, "/ArrayData/vel",  H5T_IEEE_F64LE,
                                vel_dataspace_id,  H5P_DEFAULT,
                                H5P_DEFAULT, H5P_DEFAULT);
```

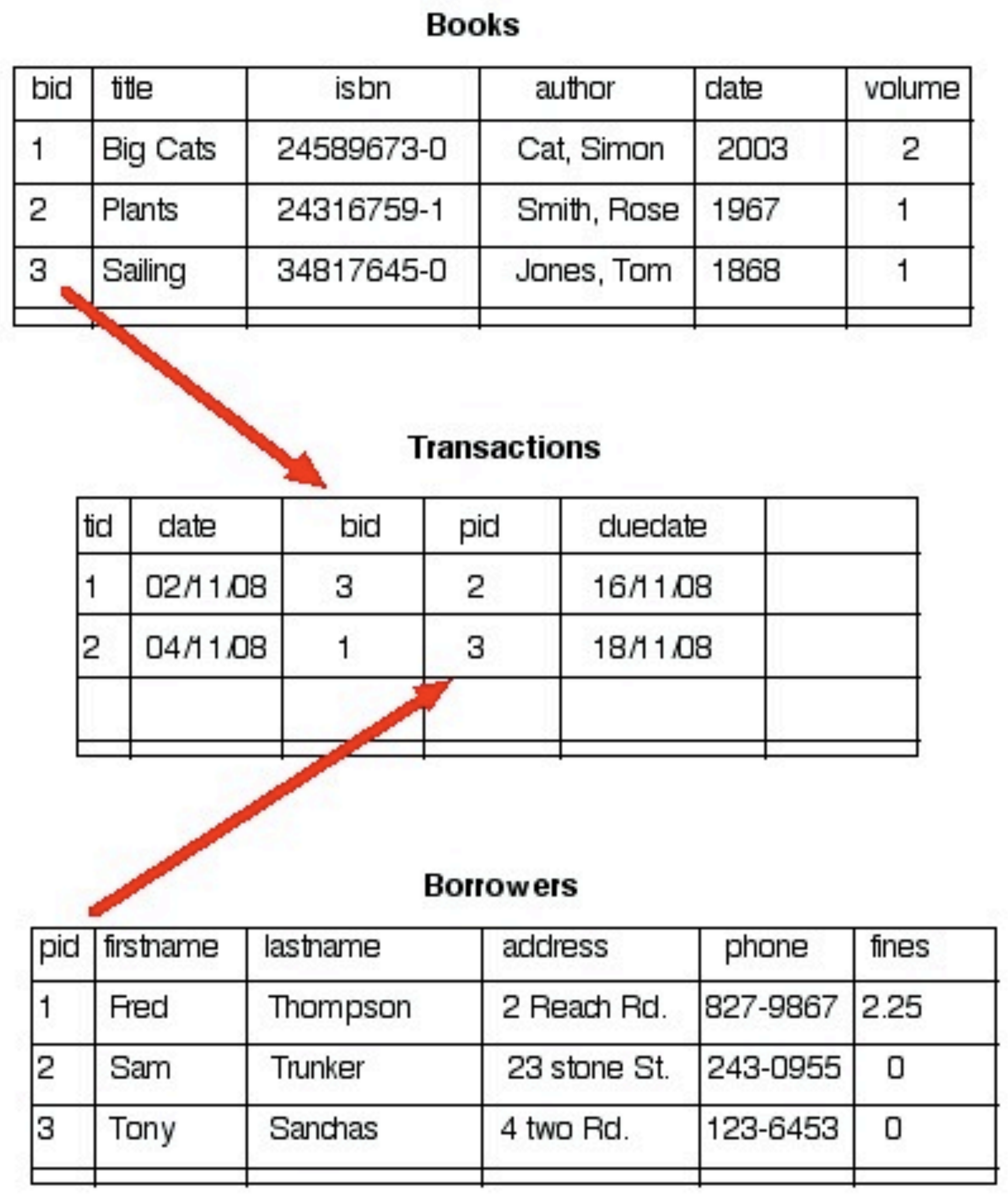
Can specify that a dataset goes in a group by giving it an "absolute path"...

```
/* Create a new group within the new file */
arr_group_id = H5Gcreate(file_id,"/ArrayData", H5P_DEFAULT, H5P_DEFAULT,
H5P_DEFAULT);
...

dens_dataset_id = H5Dcreate(arr_group_id, "dens", H5T_IEEE_F64LE,
                                dens_dataspace_id, H5P_DEFAULT,
                                H5P_DEFAULT, H5P_DEFAULT);
vel_dataset_id  = H5Dcreate(arr_group_id, "vel",  H5T_IEEE_F64LE,
                                vel_dataspace_id,  H5P_DEFAULT,
                                H5P_DEFAULT, H5P_DEFAULT);
```

...or just by creating it *in* the group, rather than the file.

What NetCDF, HDF *aren't*

Databases
Seem like - lots of information, in key value pairs.
Relational databases - interrelated tables of **small** pieces of data
Very easy/fast to query
But can't do subarrays, etc..



Books

bid	title	isbn	author	date	volume
1	Big Cats	24589673-0	Cat, Simon	2003	2
2	Plants	24316759-1	Smith, Rose	1967	1
3	Sailing	34817645-0	Jones, Tom	1868	1

Transactions

tid	date	bid	pid	duedate	
1	02/11/08	3	2	16/11/08	
2	04/11/08	1	3	18/11/08	

Borrowers

pid	firstname	lastname	address	phone	fines
1	Fred	Thompson	2 Reach Rd.	827-9867	2.25
2	Sam	Trunker	23 stone St.	243-0955	0
3	Tony	Sanchas	4 two Rd.	123-6453	0

Databases for science

```
INSERT INTO benchmarkruns
values (newrunnum, datestr,
timestr, juliannum)

...
SELECT nprocs, test, size,
transport, mpitype, runtime,
mopsperproc, run FROM
mpirundata WHERE (success=1)
```

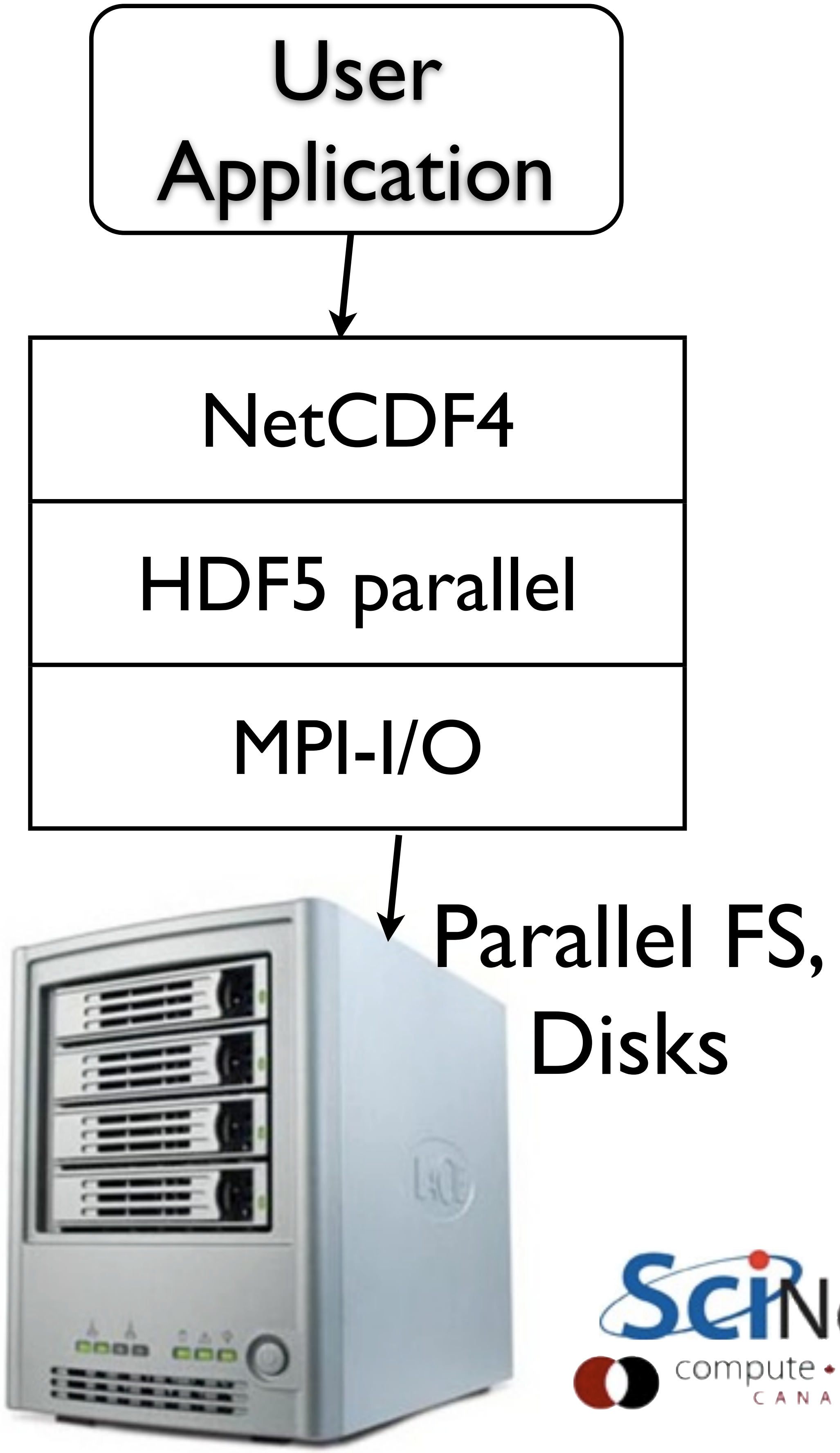
run#	success	size	transport	...
93	no	12k	eth	
1	yes	512	eth	
87	yes	64	ib	
13	no	32	eth	

...

Parallel I/O using NetCDF4, HDF5

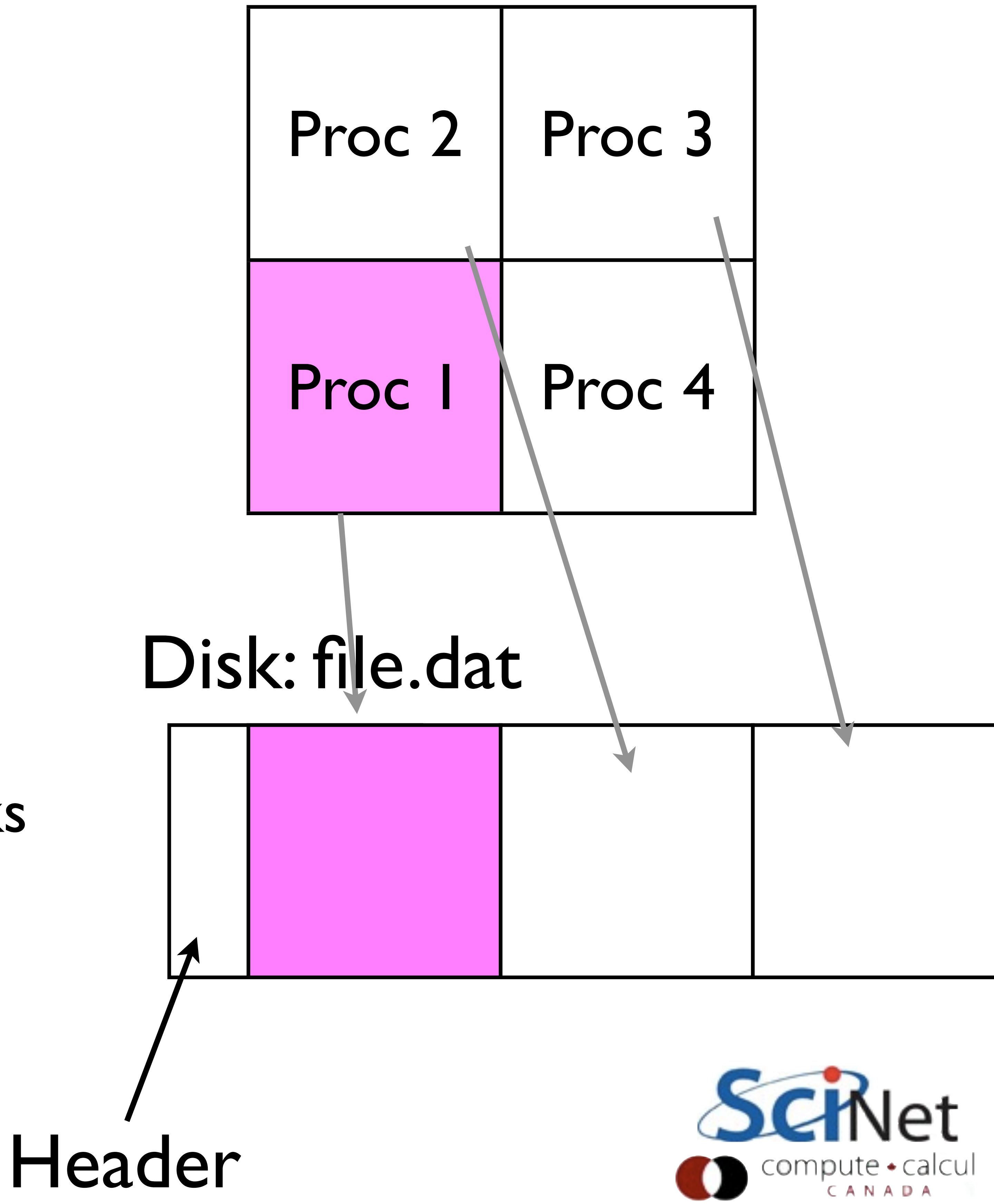
Parallel I/O libraries

Can use the same NetCDF(4), HDF5 libraries to do Parallel IO on top of the MPI-I/O library
Reading file afterwards, can't tell the difference.
Fairly minor differences in function calls to do parallel I/O
Hard part is figuring out what/where to write



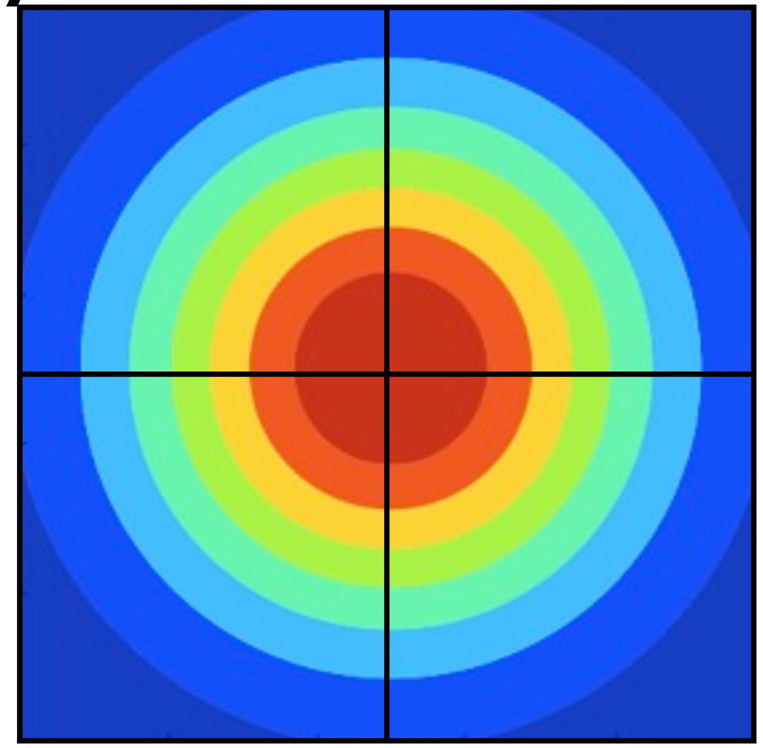
Parallel IO to One file

Can be made to work efficiently, but must write to *disjoint* chunks of file
Should write *big* disjoint chunks of file.

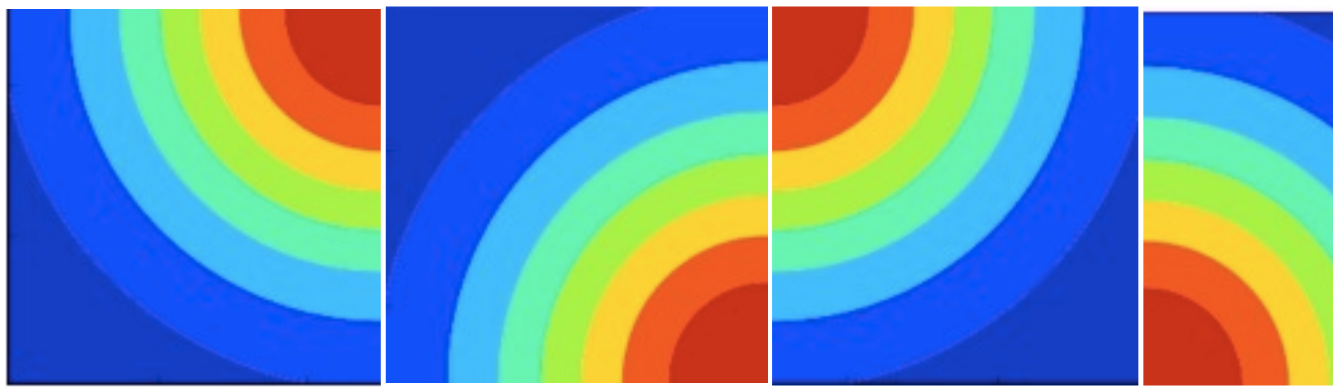


How do you decide where to write?

Memory:



Disk:



One possibility: each processor writes out its part of problem, in order.

Pros - can be super fast.

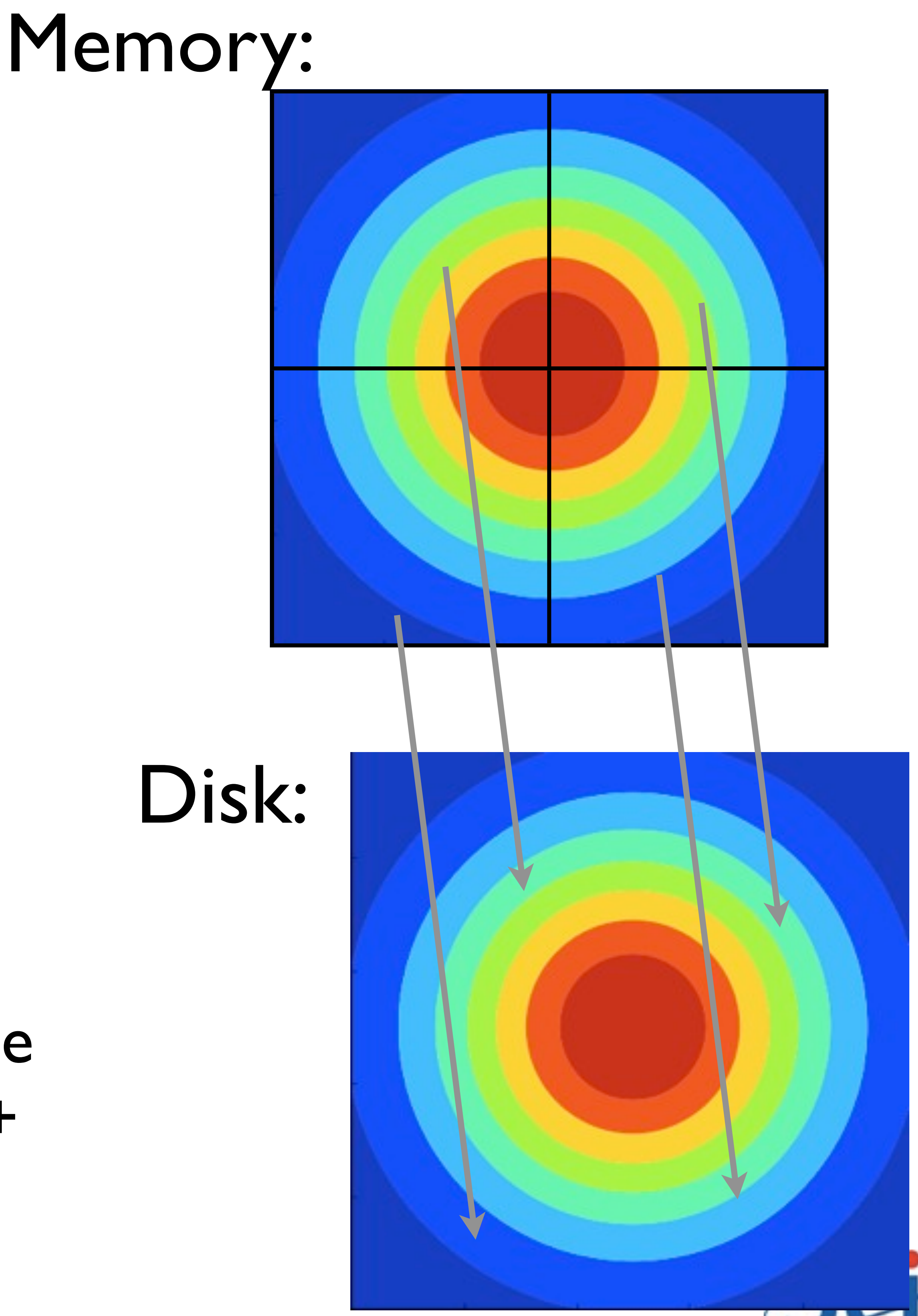
Cons - Output depends on number of processors run on. Analysis routines, restarts...

How do you decide where to write?

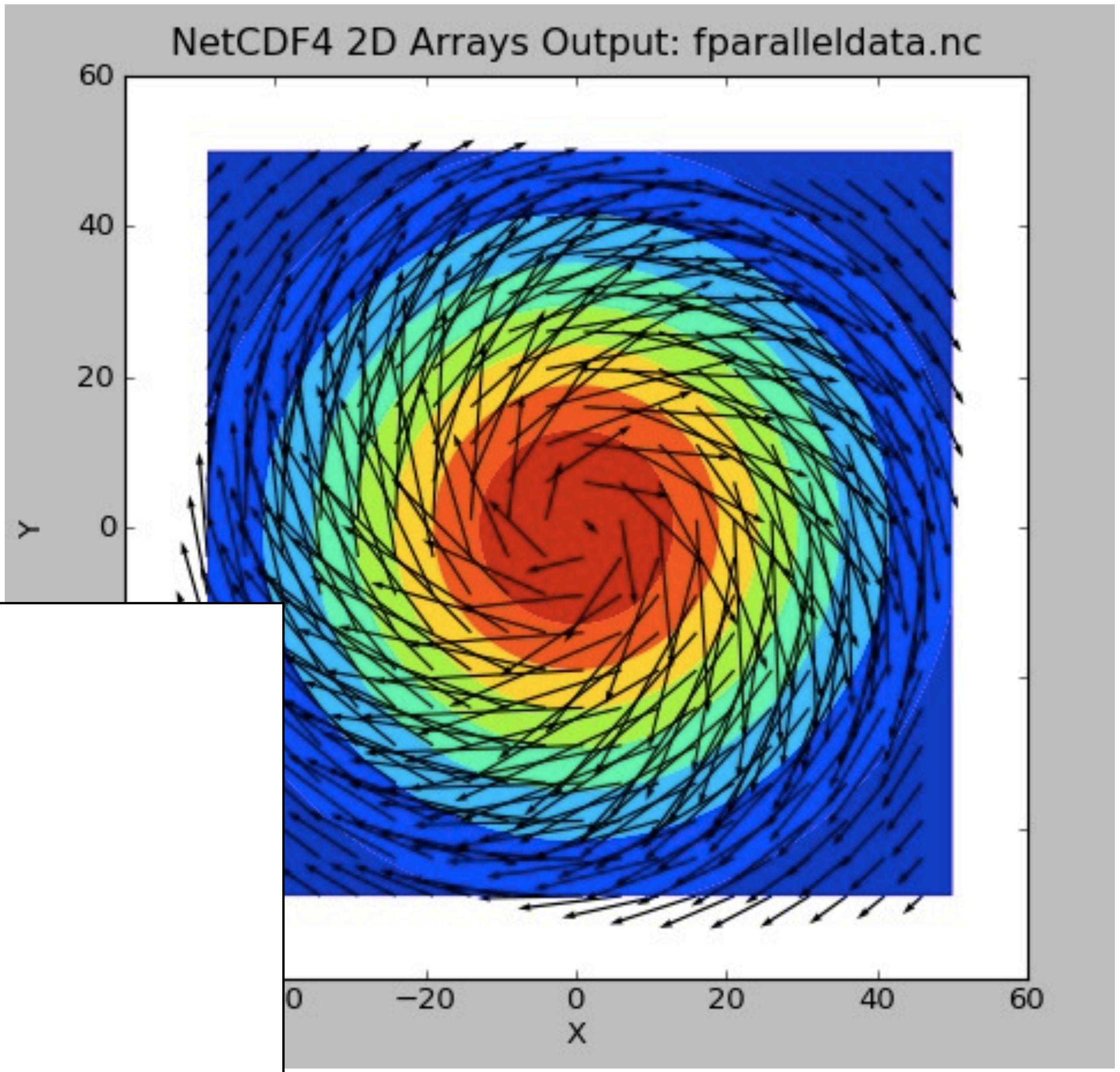
Other possibility: Write out chunks as they would be in memory on serial machine

Pros: File looks the same no matter how many processes were used to write.

Cons: Noncontig access; may be slower, but MPI-IO collective + good parallel FS should make competitive.



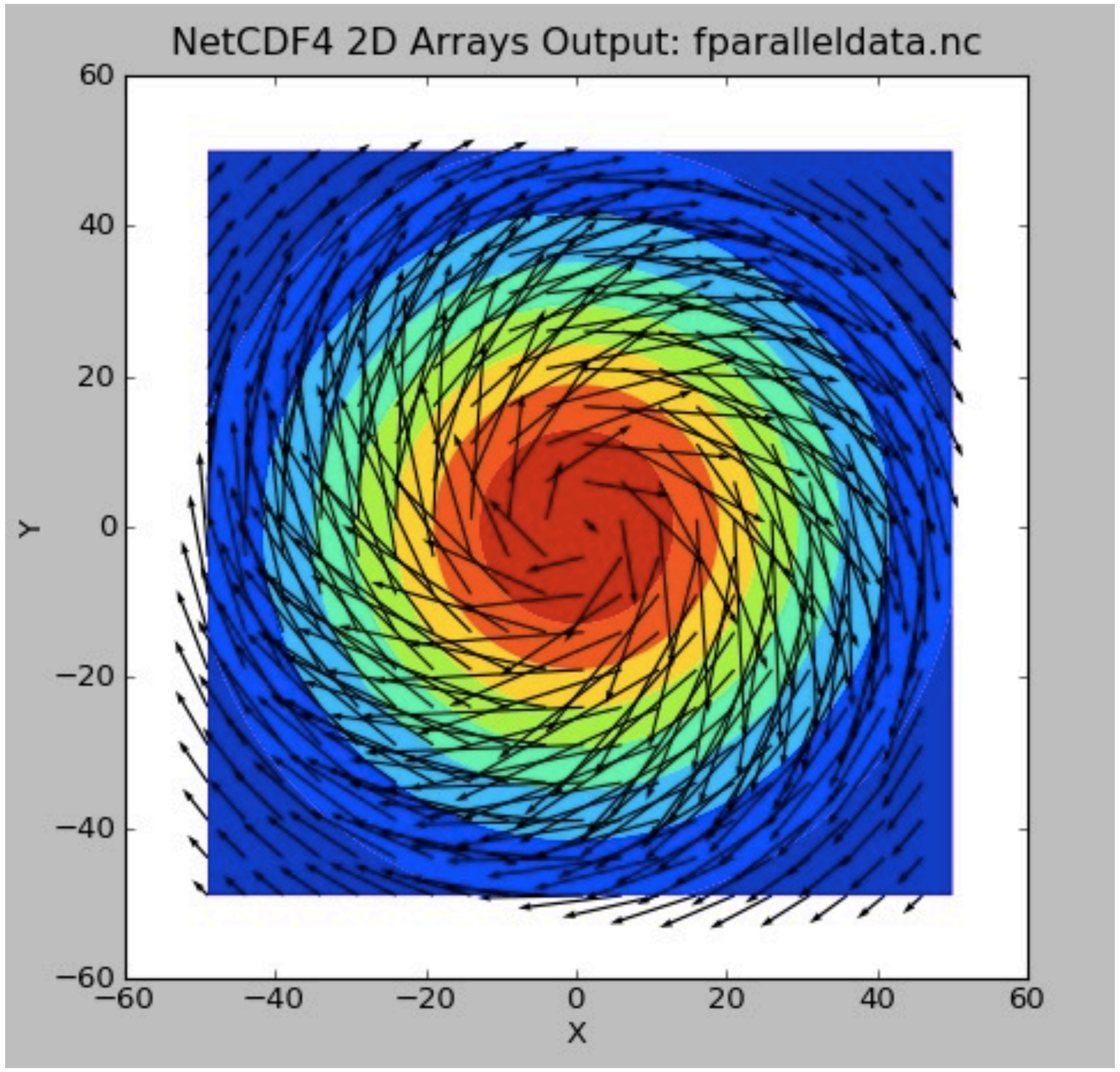
Sample Code



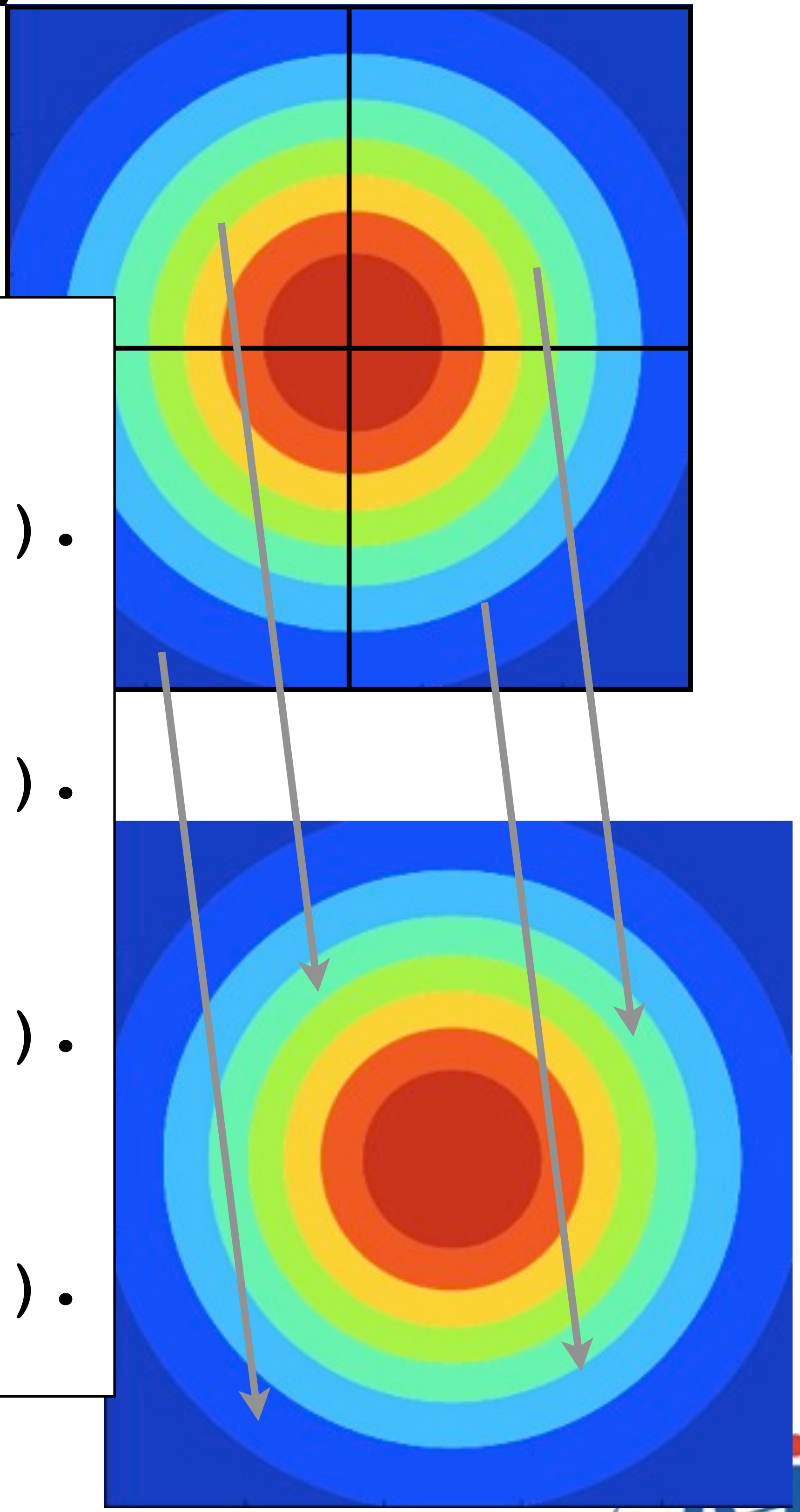
```
$ cd
$ cd parIO/netcdf

$ make parallel2darray (C), or
$ make fparallel2darray (F90)

$ mpirun -np 4 parallel2darray

$ ls *.nc
$ source ../seriallibs
$ ../plots.py paralleldata.nc
```

Sample Code

- Can do an ncdump -h...
- No trace of being written by different files
- Looks the same; code to read in is identical
- And not that much harder to code!
- By far the trickiest part is figuring out where in the file to write.

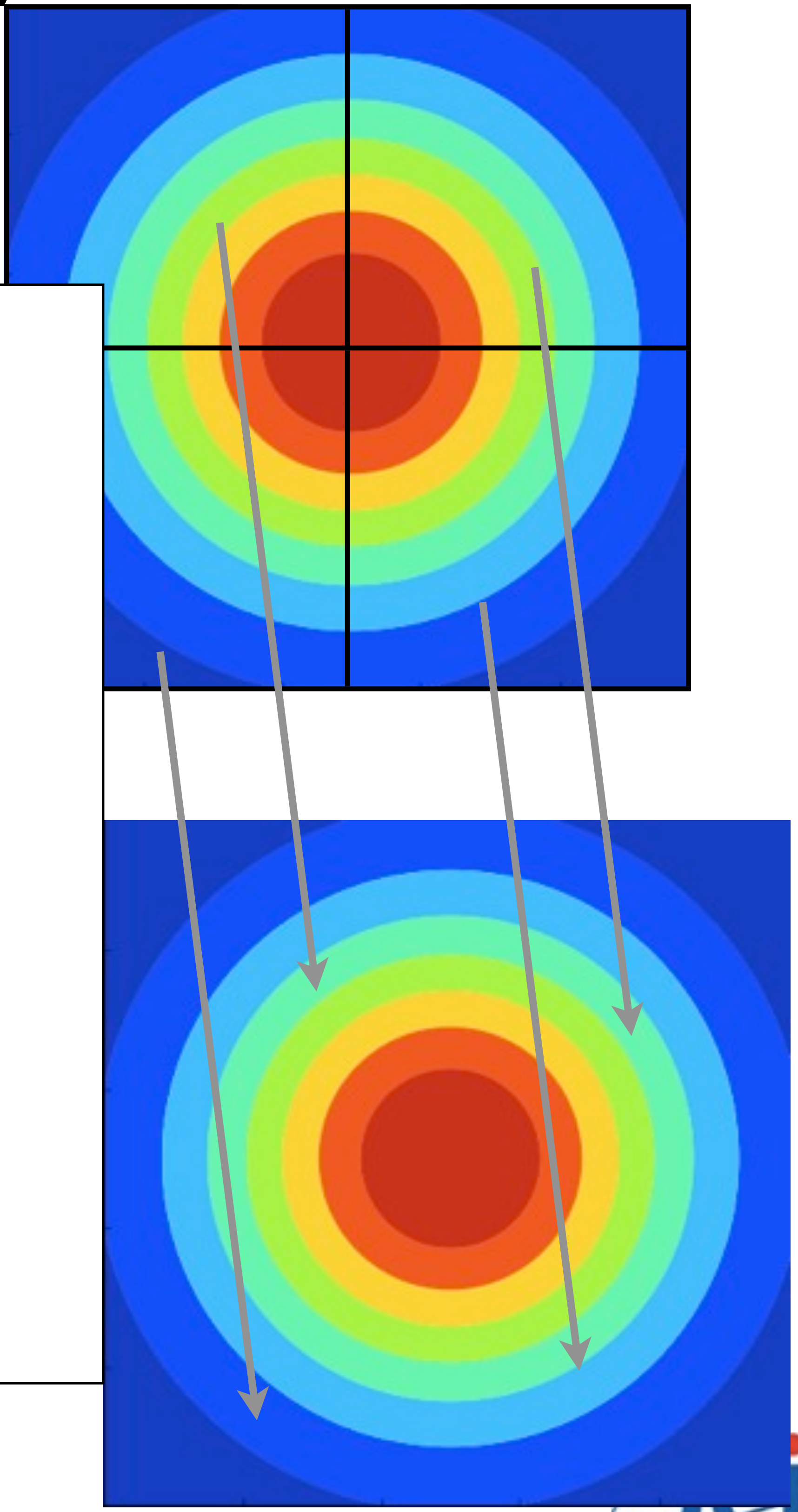


Memory:



```
$ mpirun -np 4 ./fparallel2darray
[   0] gets ( 0,   0): local points =
( 50, 50); global points = (100,100).

[   1] gets ( 1,   0): local points =
( 50, 50); global points = (100,100).

[   2] gets ( 0,   1): local points =
( 50, 50); global points = (100,100).

[   3] gets ( 1,   1): local points =
( 50, 50); global points = (100,100).
```

Memory:



```
[   0]: denstarts, denscounts
        =    1    1   50   50
[   1]: denstarts, denscounts
        =   51    1   50   50
[   2]: denstarts, denscounts
        =    1   51   50   50
[   3]: denstarts, denscounts
        =   51   51   50   50
```

fparallel2darray.f90

```
call MPI_Info_create(info, status)
call MPI_Info_set(info,"IBM_largeblock_io","true", status)

mode_flag = IOR(NF90_MPIIO, NF90_CLOBBER)
mode_flag = IOR(mode_flag, NF90_NETCDF4)
status = nf90_create_par(rundata%filename, mode_flag,
MPI_COMM_WORLD, info, file_id)
if (status /= NF90_NOERR) then
      print *,'Could not open file ', rundata%filename
      return
endif
```

fparallel2darray.f90

```
call MPI_Info_create(info, status)
call MPI_Info_set(info,"IBM_largeblock_io","true", status)

mode_flag = IOR(NF90_MPIIO, NF90_CLOBBER)
mode_flag = IOR(mode_flag, NF90_NETCDF4)
status = nf90_create_par(rundata%filename, mode_flag,
MPI_COMM_WORLD, info, file_id)
if (status /= NF90_NOERR) then
      print *,'Could not open file ', rundata%filename
      return
endif
```

create_par rather than create

fparallel2darray.f90

```
call MPI_Info_create(info, status)
call MPI_Info_set(info,"IBM_largeblock_io","true", status)

mode_flag = IOR(NF90_MPIIO, NF90_CLOBBER)
mode_flag = IOR(mode_flag, NF90_NETCDF4)
status = nf90_create_par(rundata%filename, mode_flag,
MPI_COMM_WORLD, info, file_id)
if (status /= NF90_NOERR) then
      print *,'Could not open file ', rundata%filename
      return
endif
```

mode_flag = CLOBBER | MPIIO | NETCDF4

fparallel2darray.f90

```
call MPI_Info_create(info, status)
call MPI_Info_set(info,"IBM_largeblock_io","true", status)

mode_flag = IOR(NF90_MPIIO, NF90_CLOBBER)
mode_flag = IOR(mode_flag, NF90_NETCDF4)
status = nf90_create_par(rundata%filename, mode_flag,
MPI_COMM_WORLD, info, file_id)
if (status /= NF90_NOERR) then
      print *,'Could not open file ', rundata%filename
      return
endif
```

Extra arguments: communicator that will do the I/O

fparallel2darray.f90

```
call MPI_Info_create(info, status)
call MPI_Info_set(info,"IBM_largeblock_io","true", status)

mode_flag = IOR(NF90_MPIIO, NF90_CLOBBER)
mode_flag = IOR(mode_flag, NF90_NETCDF4)
status = nf90_create_par(rundata%filename, mode_flag,
MPI_COMM_WORLD, info, file_id)
if (status /= NF90_NOERR) then
      print *,'Could not open file ', rundata%filename
      return
endif
```

Extra arguments: MPI Info; can pass MPI-I/O "hints"

fparallel2darray.f90

```
status = nf90_def_dim(file_id, 'X', rundata%globalnx, xdim_id)
status = nf90_def_dim(file_id, 'Y', rundata%globalny, ydim_id)
status = nf90_def_dim(file_id, 'velocity components', 2,
vcomp_id)

! now that the dimensions are defined, define variables

densdims = (/ xdim_id, ydim_id /)
veldims = (/ vcomp_id, xdim_id, ydim_id /)

status = nf90_def_var(file_id, 'Density',  NF90_DOUBLE, densdims,
dens_id)
status = nf90_def_var(file_id, 'Velocity', NF90_DOUBLE, veldims,
vel_id)
```

Defining variables identical (but global v local)

fparallel2darray.f90

```
status = nf90_var_par_access(file_id, dens_id, NF90_COLLECTIVE)
status = nf90_var_par_access(file_id, vel_id,  NF90_COLLECTIVE)


status = nf90_put_var(file_id, dens_id, dens, start=densstarts,
count=denscounts)
status = nf90_put_var(file_id, vel_id,  vel, start=velstarts,
count=velcounts)


status = nf90_close(file_id)
```

Define how we'll be accessing *variables* - COLLECTIVE vs INDEPENDANT. (eg, Write_all vs. Write).

fparallel2darray.f90

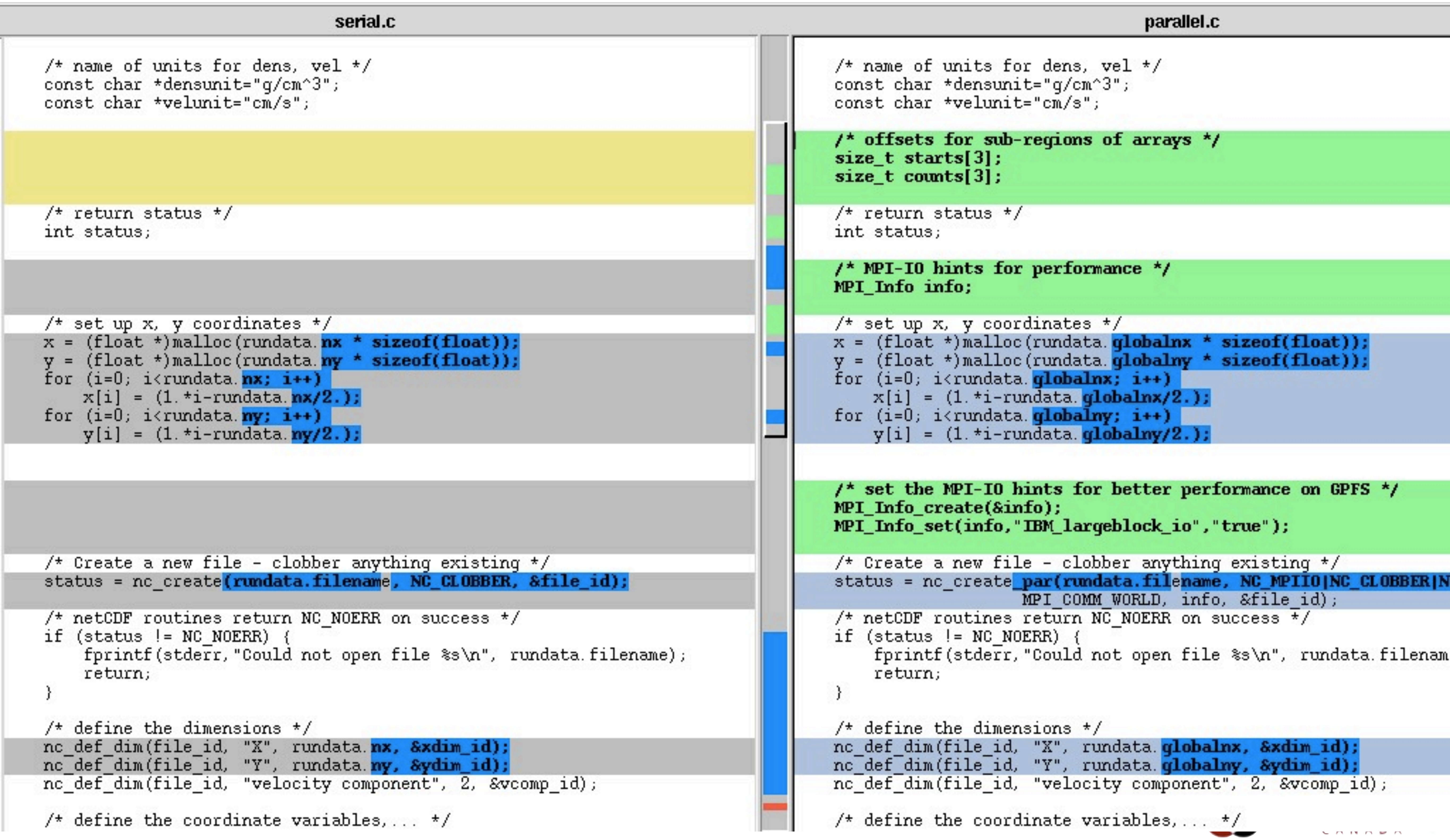
```
status = nf90_var_par_access(file_id, dens_id, NF90_COLLECTIVE)
status = nf90_var_par_access(file_id, vel_id,  NF90_COLLECTIVE)


status = nf90_put_var(file_id, dens_id, dens, start=densstarts,
count=denscounts)
status = nf90_put_var(file_id, vel_id,  vel, start=velstarts,
count=velcounts)


status = nf90_close(file_id)
```

put_var is exactly like serial with subsections - starts, counts

fparallel2darray.f90

```
status = nf90_var_par_access(file_id, dens_id, NF90_COLLECTIVE)
status = nf90_var_par_access(file_id, vel_id,  NF90_COLLECTIVE)


status = nf90_put_var(file_id, dens_id, dens, start=densstarts,
count=denscounts)
status = nf90_put_var(file_id, vel_id,  vel, start=velstarts,
count=velcounts)


status = nf90_close(file_id)
```

close is the same as ever.

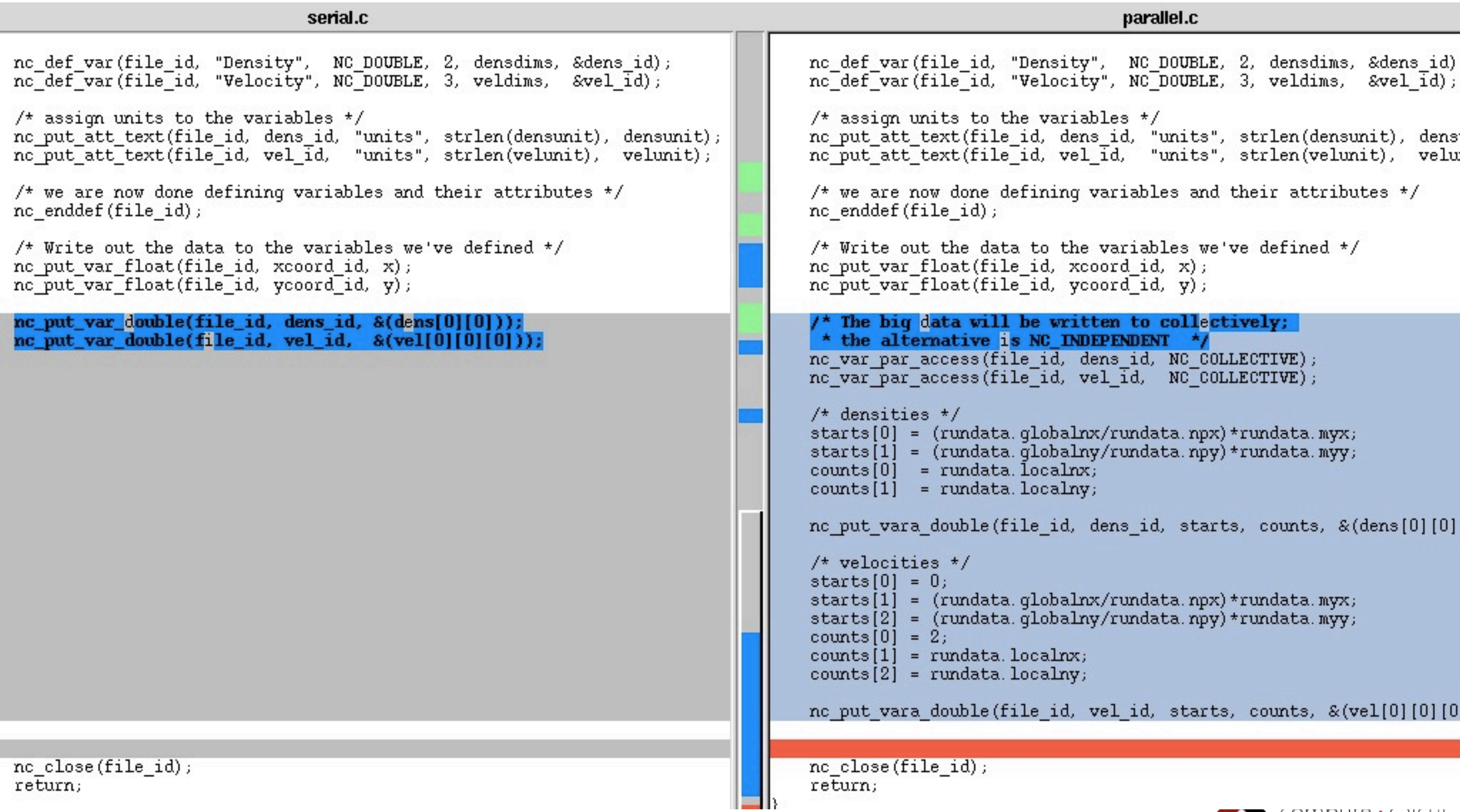


serial.c

```
/* name of units for dens, vel */
const char *densunit="g/cm^3";
const char *velunit="cm/s";



/* return status */
int status;


/* set up x, y coordinates */
x = (float *)malloc(rundata.nx * sizeof(float));
y = (float *)malloc(rundata.ny * sizeof(float));
for (i=0; i<rundata.nx; i++)
    x[i] = (1.*i-rundata.nx/2.);
for (i=0; i<rundata.ny; i++)
    y[i] = (1.*i-rundata.ny/2.);




/* Create a new file - clobber anything existing */
status = nc_create(rundata.filename, NC_CLOBBER, &file_id);

/* netCDF routines return NC_NOERR on success */
if (status != NC_NOERR) {
    fprintf(stderr,"Could not open file %s\n", rundata.filename);
    return;
}

/* define the dimensions */
nc_def_dim(file_id, "X", rundata.nx, &xdim_id);
nc_def_dim(file_id, "Y", rundata.ny, &ydim_id);
nc_def_dim(file_id, "velocity component", 2, &vcomp_id);

/* define the coordinate variables,... */
```

parallel.c

```
/* name of units for dens, vel */
const char *densunit="g/cm^3";
const char *velunit="cm/s";

/* offsets for sub-regions of arrays */
size_t starts[3];
size_t counts[3];

/* return status */
int status;

/* MPI-IO hints for performance */
MPI_Info info;

/* set up x, y coordinates */
x = (float *)malloc(rundata.globalnx * sizeof(float));
y = (float *)malloc(rundata.globalny * sizeof(float));
for (i=0; i<rundata.globalnx; i++)
    x[i] = (1.*i-rundata.globalnx/2.);
for (i=0; i<rundata.globalny; i++)
    y[i] = (1.*i-rundata.globalny/2.);


/* set the MPI-IO hints for better performance on GPFS */
MPI_Info_create(&info);
MPI_Info_set(info,"IBM_largeblock_io","true");

/* Create a new file - clobber anything existing */
status = nc_create_par(rundata.filename, NC_MPIIO|NC_CLOBBER|N
                    MPI_COMM_WORLD, info, &file_id);
/* netCDF routines return NC_NOERR on success */
if (status != NC_NOERR) {
    fprintf(stderr,"Could not open file %s\n", rundata.filenam
    return;
}

/* define the dimensions */
nc_def_dim(file_id, "X", rundata.globalnx, &xdim_id);
nc_def_dim(file_id, "Y", rundata.globalny, &ydim_id);
nc_def_dim(file_id, "velocity component", 2, &vcomp_id);

/* define the coordinate variables,... */
```



serial.c

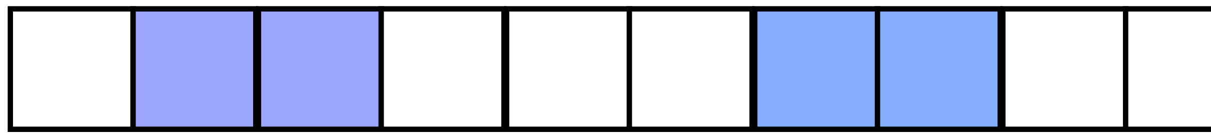
```
nc_def_var(file_id, "Density",  NC_DOUBLE, 2, densdims, &dens_id);
nc_def_var(file_id, "Velocity", NC_DOUBLE, 3, veldims,  &vel_id);

/* assign units to the variables */
nc_put_att_text(file_id, dens_id, "units", strlen(densunit), densunit);
nc_put_att_text(file_id, vel_id,  "units", strlen(velunit),  velunit);

/* we are now done defining variables and their attributes */
nc_enddef(file_id);

/* Write out the data to the variables we've defined */
nc_put_var_float(file_id, xcoord_id, x);
nc_put_var_float(file_id, ycoord_id, y);

nc_put_var_double(file_id, dens_id, &(dens[0][0]));
nc_put_var_double(file_id, vel_id,  &(vel[0][0][0]));

nc_close(file_id);
return;
```

parallel.c

```
nc_def_var(file_id, "Density",  NC_DOUBLE, 2, densdims, &dens_id)
nc_def_var(file_id, "Velocity", NC_DOUBLE, 3, veldims,  &vel_id);

/* assign units to the variables */
nc_put_att_text(file_id, dens_id, "units", strlen(densunit), dens
nc_put_att_text(file_id, vel_id,  "units", strlen(velunit),  velu

/* we are now done defining variables and their attributes */
nc_enddef(file_id);

/* Write out the data to the variables we've defined */
nc_put_var_float(file_id, xcoord_id, x);
nc_put_var_float(file_id, ycoord_id, y);

/* The big data will be written to collectively;
 * the alternative is NC_INDEPENDENT  */
nc_var_par_access(file_id, dens_id, NC_COLLECTIVE);
nc_var_par_access(file_id, vel_id,  NC_COLLECTIVE);

/* densities */
starts[0]  = (rundata.globalnx/rundata.npx)*rundata.myx;
starts[1]  = (rundata.globalny/rundata.npy)*rundata.myy;
counts[0]   = rundata.localnx;
counts[1]   = rundata.localny;

nc_put_vara_double(file_id, dens_id, starts, counts, &(dens[0][0]

/* velocities */
starts[0] = 0;
starts[1] = (rundata.globalnx/rundata.npx)*rundata.myx;
starts[2] = (rundata.globalny/rundata.npy)*rundata.myy;
counts[0] = 2;
counts[1] = rundata.localnx;
counts[2] = rundata.localny;

nc_put_vara_double(file_id, vel_id, starts, counts, &(vel[0][0][0

nc_close(file_id);
return;
}
```

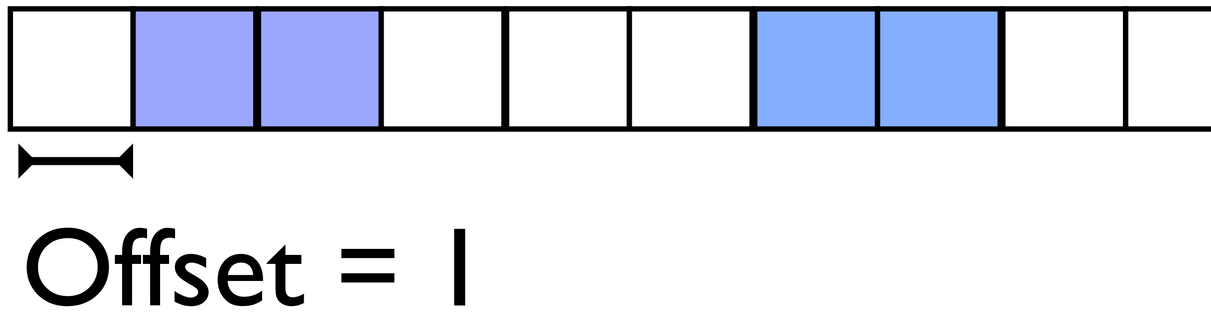
HDF5 Hyperslabs

- Parallel HDF5 similar to parallel NetCDF - fairly modest changes to structure of code
- Different (more low-level, natch) way of dealing with sub-regions
- Offset, block, count, stride



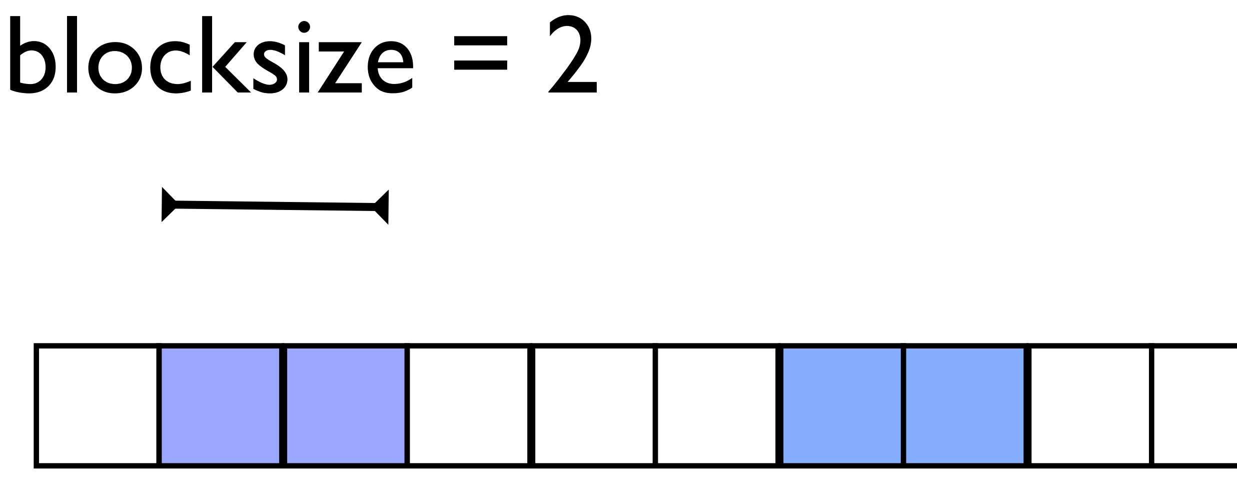
HDF5 Hyperslabs

- Parallel HDF5 similar to parallel NetCDF - fairly modest changes to structure of code
- Different (more low-level, natch) way of dealing with sub-regions
- Offset, block, count, stride



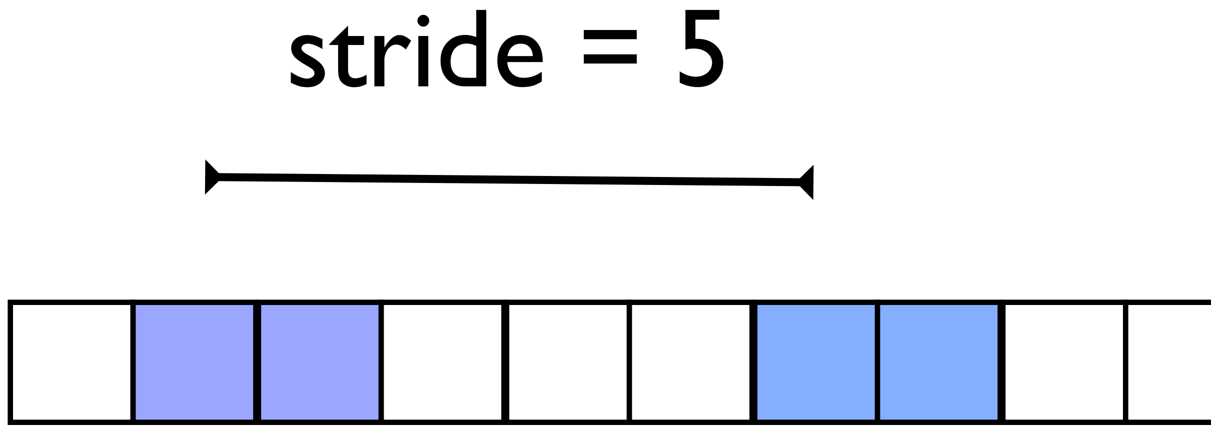
HDF5 Hyperslabs

- Parallel HDF5 similar to parallel NetCDF - fairly modest changes to structure of code
- Different (more low-level, natch) way of dealing with sub-regions
- Offset, block, count, stride



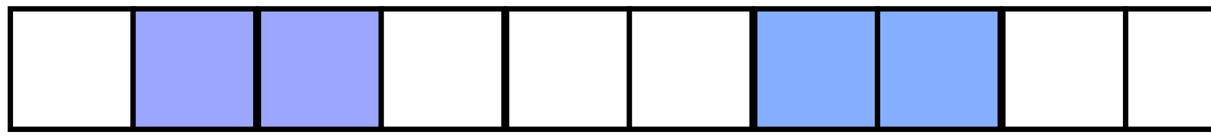
HDF5 Hyperslabs

- Parallel HDF5 similar to parallel NetCDF - fairly modest changes to structure of code
- Different (more low-level, natch) way of dealing with sub-regions
- Offset, block, count, stride



HDF5 Hyperslabs

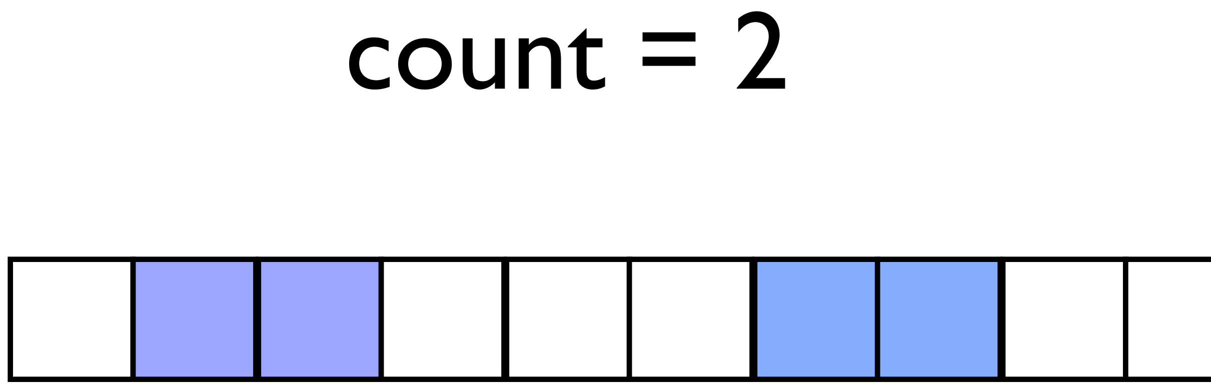
count = 2



- Parallel HDF5 similar to parallel NetCDF - fairly modest changes to structure of code
- Different (more low-level, natch) way of dealing with sub-regions
- Offset, block, count, stride
- (MPI_Type_vector)

HDF5 Hyperslabs

- Parallel HDF5 similar to parallel NetCDF - fairly modest changes to structure of code
- Different (more low-level, natch) way of dealing with sub-regions
- Offset, block, count, stride
- Hyperslab - one of these per dimensions.
- (offset,block) just like (start, counts) in netcdf.



parallel2darray.c

```
/* set the MPI-IO hints for better performance on GPFS */
MPI_Info_create(&info);
MPI_Info_set(info,"IBM_largeblock_io","true");

/* Set up the parallel environment for file access*/
fap_id = H5Pcreate(H5P_FILE_ACCESS);
/* Include the file access property with IBM hint */
H5Pset_fapl_mpio(fap_id, MPI_COMM_WORLD, info);

/* Set up the parallel environment */
dist_id = H5Pcreate(H5P_DATASET_XFER);
/* we'll be writing collectively */
H5Pset_dxpl_mpio(dist_id, H5FD_MPIO_COLLECTIVE);
```

parallel2darray.c

```
/* set the MPI-IO hints for better performance on GPFS */
MPI_Info_create(&info);
MPI_Info_set(info,"IBM_largeblock_io","true");

/* Set up the parallel environment for file access*/
fap_id = H5Pcreate(H5P_FILE_ACCESS);
/* Include the file access property with IBM hint */
H5Pset_fapl_mpio(fap_id, MPI_COMM_WORLD, info);

/* Set up the parallel environment */
dist_id = H5Pcreate(H5P_DATASET_XFER);
/* we'll be writing collectively */
H5Pset_dxpl_mpio(dist_id, H5FD_MPIO_COLLECTIVE);
```

Same as NetCDF; this is a property of the *file*

parallel2darray.c

```
/* set the MPI-IO hints for better performance on GPFS */
MPI_Info_create(&info);
MPI_Info_set(info,"IBM_largeblock_io","true");

/* Set up the parallel environment for file access*/
fap_id = H5Pcreate(H5P_FILE_ACCESS);
/* Include the file access property with IBM hint */
H5Pset_fapl_mpio(fap_id, MPI_COMM_WORLD, info);

/* Set up the parallel environment */
dist_id = H5Pcreate(H5P_DATASET_XFER);
/* we'll be writing collectively */
H5Pset_dxpl_mpio(dist_id, H5FD_MPIO_COLLECTIVE);
```

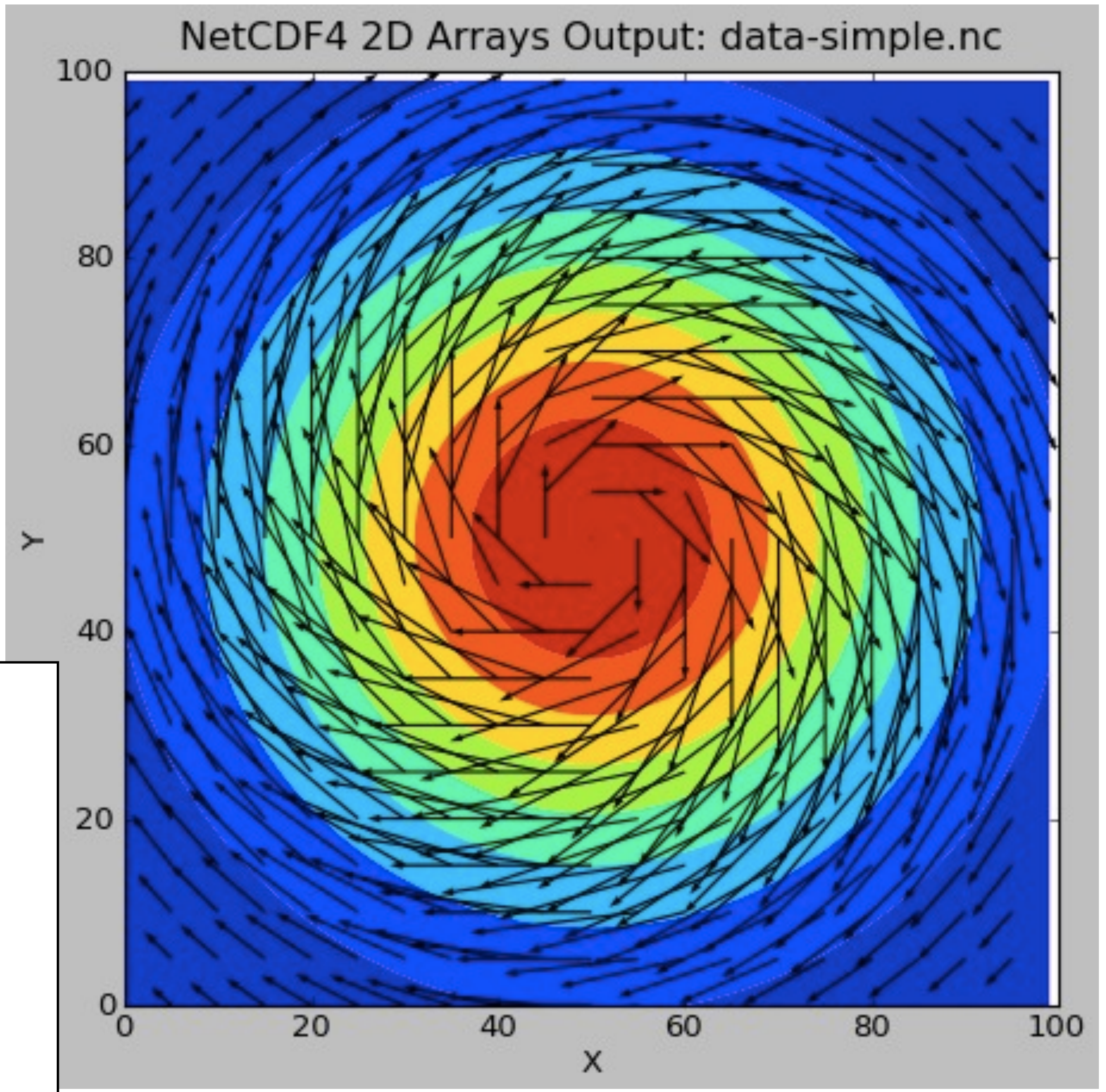
Collective/independant: this is a property of accessing a *variable*

parallel2darray.c

```
offsets[0] = (rundata.globalnx/rundata.npx)*rundata.myx;
offsets[1] = (rundata.globalny/rundata.npy)*rundata.myy;
blocks[0]  = rundata.localnx;
strides[0] = strides[1] = 1;
counts[0] = counts[1] = 1;

globaldensspace = H5Dget_space(dens_dataset_id);
H5Sselect_hyperslab(globaldensspace,H5S_SELECT_SET, offsets,
strides, counts, blocks);

status = H5Dwrite(dens_dataset_id, H5T_NATIVE_DOUBLE,
loc_dens_dataspace_id, globaldensspace, dist_id, &(dens[0]
[0]));
```

Select hyperslab, and write; parallelism is in distribution_id

Projects



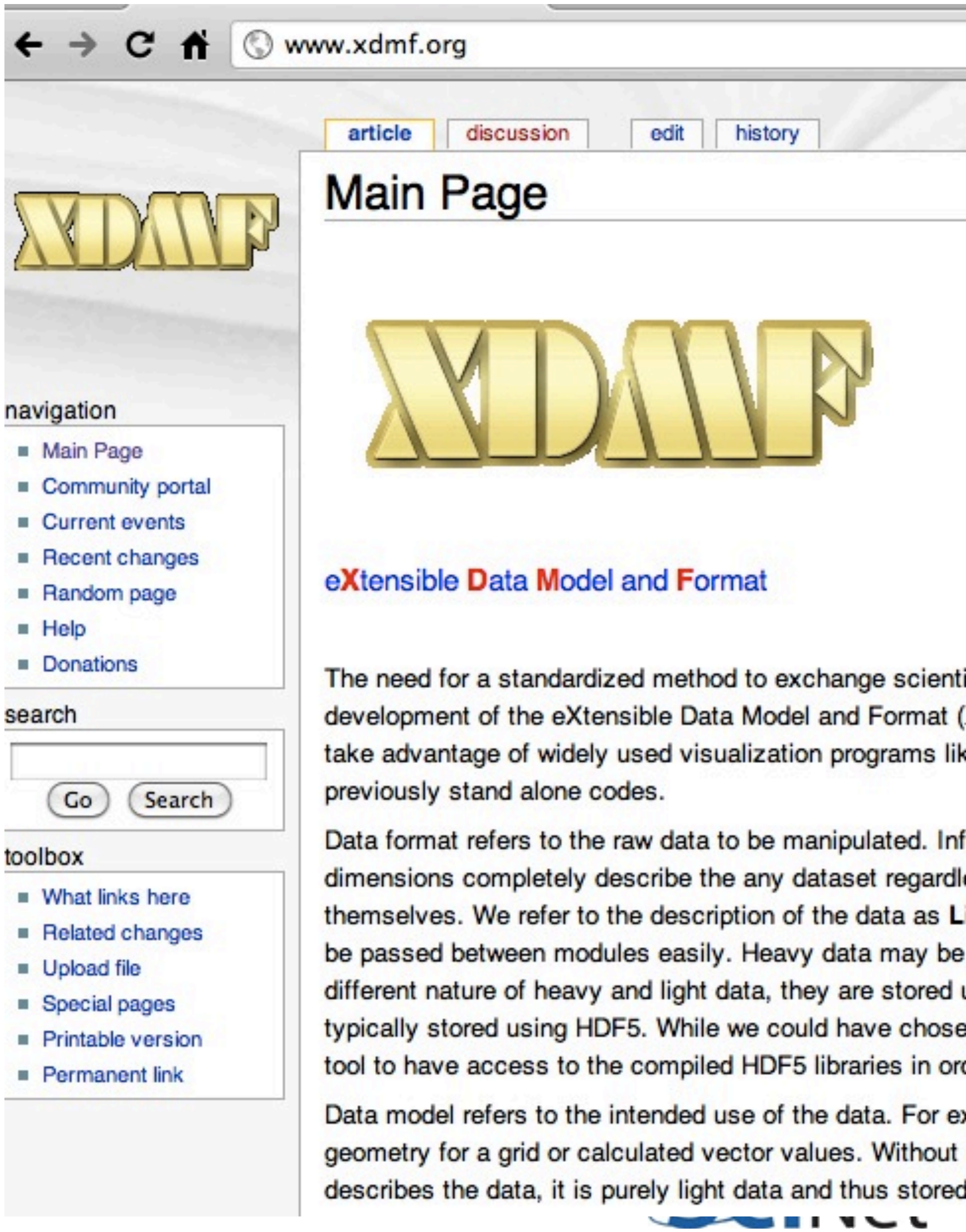
```
$ cd parIO/hydro{c,f}
```
Write hdf5, netcdf outputs

```
$ cd parIO/hydro{c,f}-mpi
```
Write ppm output in MPI-IO, (started) and output in parallel hdf5, netcdf

```
$ cd parIO/nbody
```
Write parallel hdf5, netcdf, MPI-IO outputs for gravitational particles (FORTRAN)

Conventions for HDF5

XDMF
An XML description of your HDF5 files
A way of encoding "conventions" for HDF5
Important for interoperatbility (eg, w/ viz packages)



Main Page

eXtensible Data Model and Format

The need for a standardized method to exchange scienti
development of the eXtensible Data Model and Format (
take advantage of widely used visualization programs lik
previously stand alone codes.

Data format refers to the raw data to be manipulated. Inf
dimensions completely describe the any dataset regardl
themselves. We refer to the description of the data as L
be passed between modules easily. Heavy data may be
different nature of heavy and light data, they are stored
typically stored using HDF5. While we could have chose
tool to have access to the compiled HDF5 libraries in or

Data model refers to the intended use of the data. For ex
geometry for a grid or calculated vector values. Without
describes the data, it is purely light data and thus stored

compute • calcul
CANADA

Adaptable IO System

ADIOS

A library for IO for scientific code

Uses MPIIO, HDF5, etc... under the hood

Allows changing of IO strategy, method; no rewriting code and maybe not even a recompile.

www.olcf.ornl.gov/center-projects/adios/
OLCF
Oak Ridge Leadership Computing Facility
Search OLCF.ORN
HOME
ABOUT OLCF
LEADERSHIP SCIENCE
COMPUTING RESOURCES
CENTER P
Home > Center Projects > Adios
Overview
Download & Installation
Press & Publications
ADIOS
ADIOS 1.2 is Here!
easy-to-use, fast, scalable, and portable I/O

Adios

The Adaptable IO System (ADIOS) provides a simple, flexible way for scientists to desribe the data in their code that may need to be written, read, or processed outside of the running simulation. By providi an external to the code XML file describing the various elements, their types, and how you wish to process them this run, the routines in the host code (either Fortran or C) can transparently change how they process the data.

The in code IO routines were modeled after standard Fortran POSIX IO routines for simplicity and clari The additional complexity including organization into hierarchies, data type specifications, process

parIO/adios/parallel2darray.{c,f90}

```
void writeadiosfile(rundata_t *rundata, double **dens, double ***vel) {
    int         adios_err=0;
    uint64_t    adios_groupsize, adios_totalsize;
    int64_t     adios_handle;
    MPI_Comm    comm = MPI_COMM_WORLD;
    int size;

    MPI_Comm_size(comm, &size);

    adios_init ("adios_global.xml");
    adios_open (&adios_handle, "ArrayData", rundata->filename, "w", &comm);
    #include "gwrite_ArrayData.ch"

    if (adios_err)
        fprintf(stderr,"Error doing adios write.\n");

    adios_close (adios_handle);
}
```

parIO/adios/{,f}array_global.xml

```
<?xml version="1.0"?>
<adios-config host-language="C">

    <adios-group name="ArrayData" coordination-communicator="comm">
        <var name="rundata->localnx"  type="integer" />
        <var name="rundata->localny"  type="integer" />
        <var name="rundata->globalnx" type="integer" />
        <var name="rundata->globalny" type="integer" />
        <var name="rundata->startx"   type="integer" />
        <var name="rundata->starty"   type="integer" />
        <var name="size" type="integer" />
        <global-bounds dimensions="2,rundata->globalnx,rundata->globalny"
                       offsets="0,rundata->startx,rundata->starty">
           <var name="vel" gwrite="vel[0][0]" type="double"
                       dimensions="2,rundata->localnx,rundata->localny" />
        </global-bounds>
        <global-bounds dimensions="rundata->globalnx,rundata->globalny"
                       offsets="rundata->startx,rundata->starty">
           <var name="dens" gwrite="dens[0]" type="double"
                       dimensions="rundata->localnx,rundata->localny" />
        </global-bounds>
    </adios-group>

<method group="ArrayData" method="PHDF5" />

<buffer size-MB="2" allocate-time="now"/>

</adios-config>
```

ADIOS workflow

Write XML file describing data, layout

gpp.py [file].xml - generates C or Fortran code: adios calls, size calculation

Build code

Separates data layout, code.

```
<?xml version="1.0"?>
<adios-config host-language="C">
    <adios-group name="ArrayData" coordination-communicator="
        <var name="rundata->localnx"  type="integer" />
        <var name="rundata->localny"  type="integer" />
        <var name="rundata->globalnx" type="integer" />
        <var name="rundata->globalny" type="integer" />
        <var name="rundata->startx"   type="integer" />
        <var name="rundata->starty"   type="integer" />
        <var name="size" type="integer" />
        <global-bounds dimensions="2,rundata->globalnx,rundat
                       offsets="0,rundata->startx,rundata->st
           <var name="vel" gwrite="vel[0][0]" type="double"
                       dimensions="2,rundata->localnx,rundata
        </global-bounds>
        <global-bounds dimensions="rundata->globalnx,rundata-
                       offsets="rundata->startx,rundata->star
           <var name="dens" gwrite="dens[0]" type="double"
                       dimensions="rundata->localnx,rundata->
        </global-bounds>
    </adios-group>
<method group="ArrayData" method="PHDF5" />
<buffer size-MB="2" allocate-time="now"/>
</adios-config>
```

```
void writeadiosfile(rundata_t *rundata, double **dens, double
    int         adios_err=0;
    uint64_t    adios_groupsize, adios_totalsize;
    int64_t     adios_handle;
    MPI_Comm    comm = MPI_COMM_WORLD;
    int size;

    MPI_Comm_size(comm, &size);

    adios_init ("adios_global.xml");
    adios_open (&adios_handle, "ArrayData", rundata->filename
    #include "gwrite_ArrayData.ch"

    if (adios_err)
        fprintf(stderr,"Error doing adios write.\n");

    adios_close (adios_handle);
}
```

ADIOS workflow

Separation isn't perfect; xml file references code variables, etc. But allows "componentization" of I/O.

Changes that don't result in changes to grwrite_Array.ch don't require recompilation (eg, only changing number, size of variables in group).

```
<?xml version="1.0"?>
<adios-config host-language="C">
    <adios-group name="ArrayData" coordination-communicator="
        <var name="rundata->localnx"  type="integer" />
        <var name="rundata->localny"  type="integer" />
        <var name="rundata->globalnx" type="integer" />
        <var name="rundata->globalny" type="integer" />
        <var name="rundata->startx"   type="integer" />
        <var name="rundata->starty"   type="integer" />
        <var name="size" type="integer" />
        <global-bounds dimensions="2,rundata->globalnx,rundat
                       offsets="0,rundata->startx,rundata->st
           <var name="vel" gwrite="vel[0][0]" type="double"
                       dimensions="2,rundata->localnx,rundata
        </global-bounds>
        <global-bounds dimensions="rundata->globalnx,rundata-
                       offsets="rundata->startx,rundata->star
           <var name="dens" gwrite="dens[0]" type="double"
                       dimensions="rundata->localnx,rundata->
        </global-bounds>
    </adios-group>
<method group="ArrayData" method="PHDF5" />
<buffer size-MB="2" allocate-time="now"/>
</adios-config>
```

```
void writeadiosfile(rundata_t *rundata, double **dens, double
    int         adios_err=0;
    uint64_t    adios_groupsize, adios_totalsize;
    int64_t     adios_handle;
    MPI_Comm    comm = MPI_COMM_WORLD;
    int size;

    MPI_Comm_size(comm, &size);

    adios_init ("adios_global.xml");
    adios_open (&adios_handle, "ArrayData", rundata->filename
    #include "gwrite_ArrayData.ch"

    if (adios_err)
        fprintf(stderr,"Error doing adios write.\n");

    adios_close (adios_handle);
}
```

SciNet compute • calcul CANADA

Variable Groups

Multiple groups of variables possible: (eg) restart files vs. files for analysis

Variables can appear in mutiple groups

Each group can be handled with different methods

```
<?xml version="1.0"?>
<adios-config host-language="C">
    <adios-group name="ArrayData" coordination-communicator="
        <var name="rundata->localnx"  type="integer" />
        <var name="rundata->localny"  type="integer" />
        <var name="rundata->globalnx" type="integer" />
        <var name="rundata->globalny" type="integer" />
        <var name="rundata->startx"   type="integer" />
        <var name="rundata->starty"   type="integer" />
        <var name="size" type="integer" />
        <global-bounds dimensions="2,rundata->globalnx,rundat
                       offsets="0,rundata->startx,rundata->st
           <var name="vel" gwrite="vel[0][0]" type="double"
                       dimensions="2,rundata->localnx,rundata
        </global-bounds>
        <global-bounds dimensions="rundata->globalnx,rundata-
                       offsets="rundata->startx,rundata->star
           <var name="dens" gwrite="dens[0]" type="double"
                       dimensions="rundata->localnx,rundata->
        </global-bounds>
    </adios-group>
<method group="ArrayData" method="PHDF5" />
<buffer size-MB="2" allocate-time="now"/>
</adios-config>
```

```
void writeadiosfile(rundata_t *rundata, double **dens, double
    int         adios_err=0;
    uint64_t    adios_groupsize, adios_totalsize;
    int64_t     adios_handle;
    MPI_Comm    comm = MPI_COMM_WORLD;
    int size;

    MPI_Comm_size(comm, &size);

    adios_init ("adios_global.xml");
    adios_open (&adios_handle, "ArrayData", rundata->filename
    #include "gwrite_ArrayData.ch"

    if (adios_err)
        fprintf(stderr,"Error doing adios write.\n");

    adios_close (adios_handle);
}
```

I/O methods

Possible methods: parallel HDF5 (PHDF5), NetCDF (NC4), one-per-process posix files (POSIX), it's own native format (BP) using MPI-IO (MPI)

Change between methods: edit xml file, that's it.

P-1 (PHDF5, NC4,MPI), P-P (POSIX), or even P-M possible (PHDF5, etc with multiple communicators)

```
<?xml version="1.0"?>
<adios-config host-language="C">
    <adios-group name="ArrayData" coordination-communicator="
        <var name="rundata->localnx"  type="integer" />
        <var name="rundata->localny"  type="integer" />
        <var name="rundata->globalnx" type="integer" />
        <var name="rundata->globalny" type="integer" />
        <var name="rundata->startx"   type="integer" />
        <var name="rundata->starty"   type="integer" />
        <var name="size" type="integer" />
        <global-bounds dimensions="2,rundata->globalnx,rundat
                       offsets="0,rundata->startx,rundata->st
           <var name="vel" gwrite="vel[0][0]" type="double"
                       dimensions="2,rundata->localnx,rundata
        </global-bounds>
        <global-bounds dimensions="rundata->globalnx,rundata-
                       offsets="rundata->startx,rundata->star
           <var name="dens" gwrite="dens[0]" type="double"
                       dimensions="rundata->localnx,rundata->
        </global-bounds>
    </adios-group>
<method group="ArrayData" method="PHDF5" />
<buffer size-MB="2" allocate-time="now"/>
</adios-config>
```

```
void writeadiosfile(rundata_t *rundata, double **dens, double
    int         adios_err=0;
    uint64_t    adios_groupsize, adios_totalsize;
    int64_t     adios_handle;
    MPI_Comm    comm = MPI_COMM_WORLD;
    int size;

    MPI_Comm_size(comm, &size);

    adios_init ("adios_global.xml");
    adios_open (&adios_handle, "ArrayData", rundata->filename
    #include "gwrite_ArrayData.ch"

    if (adios_err)
        fprintf(stderr,"Error doing adios write.\n");

    adios_close (adios_handle);
}
```

SciNet compute • calcul CANADA

Simplifies IO code

Even if you aren't planning to switch between IO strategies, can greatly simplify code

Many mechanical steps (eg, pasting together rectangular multi-dimentional arrays) done for you.

Eliminates tedious, error-prone boilerplate code

```
<?xml version="1.0"?>
<adios-config host-language="C">
    <adios-group name="ArrayData" coordination-communicator="
        <var name="rundata->localnx"  type="integer" />
        <var name="rundata->localny"  type="integer" />
        <var name="rundata->globalnx" type="integer" />
        <var name="rundata->globalny" type="integer" />
        <var name="rundata->startx"   type="integer" />
        <var name="rundata->starty"   type="integer" />
        <var name="size" type="integer" />
        <global-bounds dimensions="2,rundata->globalnx,rundat
                       offsets="0,rundata->startx,rundata->st
           <var name="vel" gwrite="vel[0][0]" type="double"
                       dimensions="2,rundata->localnx,rundata
        </global-bounds>
        <global-bounds dimensions="rundata->globalnx,rundata-
                       offsets="rundata->startx,rundata->star
           <var name="dens" gwrite="dens[0]" type="double"
                       dimensions="rundata->localnx,rundata->
        </global-bounds>
    </adios-group>
<method group="ArrayData" method="PHDF5" />
<buffer size-MB="2" allocate-time="now"/>
</adios-config>
```

```
void writeadiosfile(rundata_t *rundata, double **dens, double
    int         adios_err=0;
    uint64_t    adios_groupsize, adios_totalsize;
    int64_t     adios_handle;
    MPI_Comm    comm = MPI_COMM_WORLD;
    int size;

    MPI_Comm_size(comm, &size);

    adios_init ("adios_global.xml");
    adios_open (&adios_handle, "ArrayData", rundata->filename
    #include "gwrite_ArrayData.ch"

    if (adios_err)
        fprintf(stderr,"Error doing adios write.\n");

    adios_close (adios_handle);
}
```

```
void writeadiosfile(rundata_t *rundata, double **dens, double ***vel) {
    int          adios_err=0;
    uint64_t     adios_groupsize, adios_totalsize;
    int64_t      adios_handle;
    MPI_Comm     comm = MPI_COMM_WORLD;
    int size;

    MPI_Comm_size(comm, &size);

    adios_init ("adios_global.xml");
    adios_open (&adios_handle, "ArrayData", rundata->filename, "w", &comm);
    #include "gwrite_ArrayData.ch"

    if (adios_err)
        fprintf(stderr,"Error doing adios write.\n");

    adios_close (adios_handle);
}
```

```
void writehdf5file(rundata_t rundata, double **dens, double ***vel) {
    /* identifiers */
    hid_t file_id, arr_group_id, dens_dataset_id, vel_dataset_id;
    hid_t dens_dataspace_id, vel_dataspace_id;
    hid_t loc_dens_dataspace_id, loc_vel_dataspace_id;
    hid_t globaldensspace,globalvelspace;
    hid_t dist_id;
    hid_t fap_id;

    /* sizes */
    hsize_t densdims[2], veldims[3];
    hsize_t locdensdims[2], locveldims[3];

    /* status */
    herr_t status;

    /* MPI-IO hints for performance */
    MPI_Info info;

    /* parameters of the hyperslab */
    hsize_t counts[3];
    hsize_t strides[3];
    hsize_t offsets[3];
    hsize_t blocks[3];
```

1/7

```
/* set the MPI-IO hints for better performance on GPFS */
MPI_Info_create(&info);
MPI_Info_set(info,"IBM_largeblock_io","true");

/* Set up the parallel environment for file access*/
fap_id = H5Pcreate(H5P_FILE_ACCESS);
/* Include the file access property with IBM hint */
H5Pset_fapl_mpio(fap_id, MPI_COMM_WORLD, info);

/* Set up the parallel environment */
dist_id = H5Pcreate(H5P_DATASET_XFER);
/* we'll be writing collectively */
H5Pset_dxpl_mpio(dist_id, H5FD_MPIO_COLLECTIVE);

/* Create a new file - truncate anything existing, use default properties */
file_id = H5Fcreate(rundata.filename, H5F_ACC_TRUNC, H5P_DEFAULT, fap_id);

/* HDF5 routines generally return a negative number on failure.
 * Should check return values! */
if (file_id < 0) {
    fprintf(stderr,"Could not open file %s\n", rundata.filename);
    return;
}
```

```
/* Create a new group within the new file */
arr_group_id = H5Gcreate(file_id,"/ArrayData", H5P_DEFAULT, H5P_DEFAULT,
                         H5P_DEFAULT);

/* Give this group an attribute listing the time of calculation */
{
    hid_t attr_id,attr_sp_id;
    struct tm *t;
    time_t now;
    int yyyymm;
    now = time(NULL);
    t = localtime(&now);
    yyyymm = (1900+t->tm_year)*100+t->tm_mon;

    attr_sp_id = H5Screate(H5S_SCALAR);
    attr_id = H5Acreate(arr_group_id, "Calculated on (YYYYMM)", H5T_STD_U32LE,
                        attr_sp_id, H5P_DEFAULT, H5P_DEFAULT);
    printf("yymm = %d\n",yyyymm);
    H5Awrite(attr_id, H5T_NATIVE_INT, &yyyymm);
    H5Aclose(attr_id);
    H5Sclose(attr_sp_id);
}

/* Create the data space for the two global datasets. */
densdims[0] = rundata.globalnx; densdims[1] = rundata.globalny;
veldims[0] = 2; veldims[1] = rundata.globalnx; veldims[2] = rundata.globalny;
```

3/7

```
    dens_dataspace_id = H5Screate_simple(2, densdims, NULL);
    vel_dataspace_id  = H5Screate_simple(3, veldims,  NULL);

    /* Create the datasets within the file.
     * H5T_IEEE_F64LE is a standard (IEEE) double precision (64 bit) floating (F) data
     * and will work on any machine.  H5T_NATIVE_DOUBLE would work too, but would give
     * different results on GPC and TCS */

    dens_dataset_id = H5Dcreate(file_id, "/ArrayData/dens", H5T_IEEE_F64LE,
                                dens_dataspace_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAUI
    vel_dataset_id  = H5Dcreate(file_id, "/ArrayData/vel",  H5T_IEEE_F64LE,
                                vel_dataspace_id,  H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAUI

    /* Now create the data space for our sub-regions.   These are the data spaces
     * of our actual local data in memory. */
    locdensdims[0] = rundata.localnx; locdensdims[1] = rundata.localny;
    locveldims[0] = 2; locveldims[1] = rundata.localnx; locveldims[2] = rundata.localny

    loc_dens_dataspace_id = H5Screate_simple(2, locdensdims, NULL);
    loc_vel_dataspace_id  = H5Screate_simple(3, locveldims,  NULL);
```

```
/*
  *
  * Now we have to figure out the `hyperslab' within the global
  * data that corresponds to our local data.
  *
  * Hyperslabs are described by an array of counts, strides, offsets,
  * and block sizes.
  *
  *       |-offx--|
  *       +-------|----|-------+   -+-
  *       |                    |    |
  *       |                    |   offy
  *       |                    |    |
  *       -       +----+       -   -+-
  *       |       |    |       |    |
  *       |       |    |       |  localny
  *       |       |    |       |    |
  *       -       +----+       -   -+-
  *       |                    |
  *       |                    |
  *       +-------|----|-------+
  *               localnx
  *
  *  In this case the blocksizes are (localnx,localny) and the offsets are
  *  (offx,offy) = ((myx)/nxp*globalnx, (myy/nyp)*globalny)
  */
```

```
offsets[0] = (rundata.globalnx/rundata.npx)*rundata.myx;
offsets[1] = (rundata.globalny/rundata.npy)*rundata.myy;
blocks[0]  = rundata.localnx;
blocks[1]  = rundata.localny;
strides[0] = strides[1] = 1;
counts[0] = counts[1] = 1;

/* select this subset of the density variable's space in the file */
globaldensspace = H5Dget_space(dens_dataset_id);
H5Sselect_hyperslab(globaldensspace,H5S_SELECT_SET, offsets, strides, counts, block

/* For the velocities, it's the same thing but there's a count of two,
 * (one for each velocity component) */

offsets[1] = (rundata.globalnx/rundata.npx)*rundata.myx;
offsets[2] = (rundata.globalny/rundata.npy)*rundata.myy;
blocks[1]  = rundata.localnx;
blocks[2]  = rundata.localny;
strides[0] = strides[1] = strides[2] = 1;
counts[0] = 2; counts[1] = counts[2] = 1;
offsets[0] = 0;
blocks[0] = 1;
```

```
    globalvelspace = H5Dget_space(vel_dataset_id);
    H5Sselect_hyperslab(globalvelspace,H5S_SELECT_SET, offsets, strides, counts, blocks

    /* Write the data.  We're writing it from memory, where it is saved
     * in NATIVE_DOUBLE format */
    status = H5Dwrite(dens_dataset_id, H5T_NATIVE_DOUBLE, loc_dens_dataspace_id, global
dist_id, &(dens[0][0]));
    status = H5Dwrite(vel_dataset_id,  H5T_NATIVE_DOUBLE, loc_vel_dataspace_id, globalv
dist_id, &(vel[0][0][0]));

    /* End access to groups & data sets and release resources used by them */
    status = H5Sclose(dens_dataspace_id);
    status = H5Dclose(dens_dataset_id);
    status = H5Sclose(vel_dataspace_id);
    status = H5Dclose(vel_dataset_id);
    status = H5Gclose(arr_group_id);
    status = H5Pclose(fap_id);
    status = H5Pclose(dist_id);

    /* Close the file */
    status = H5Fclose(file_id);
    return;
```

7/7

ADIOS hands-on:

Modify XML file, try outputting with method of MPI (use bpls or bp2hdf on resulting file), POSIX (Netcdf won't work at this point)

Try a different IO strategy; do contiguous parallel IO by having single file but with each process' data contiguous in file, one after another. With large file size (--nx=10000 --ny=10000) and 8/16 processes, what are the timings between "straight" PHDF5, MPI, POSIX, and this approach? (And how long would it have taken you to do this without ADIOS?)

Advanced: Break up MPI_COMM_WORLD into 2 communicators using MPI_Comm_split, call the new communicator comm, and output 2 files from the 8/16 processes using PHDF5 or MPI.