

# Parallel I/O

Jonathan Dursi, SciNet

[ljdursi@scinet.utoronto.ca](mailto:ljdursi@scinet.utoronto.ca)



HPCS2011



compute • calcul  
CANADA

# Agenda

- Intro to I/O
- MPI-IO
- HDF5, NetCDF4
- Parallel HDF5/NetCDF4



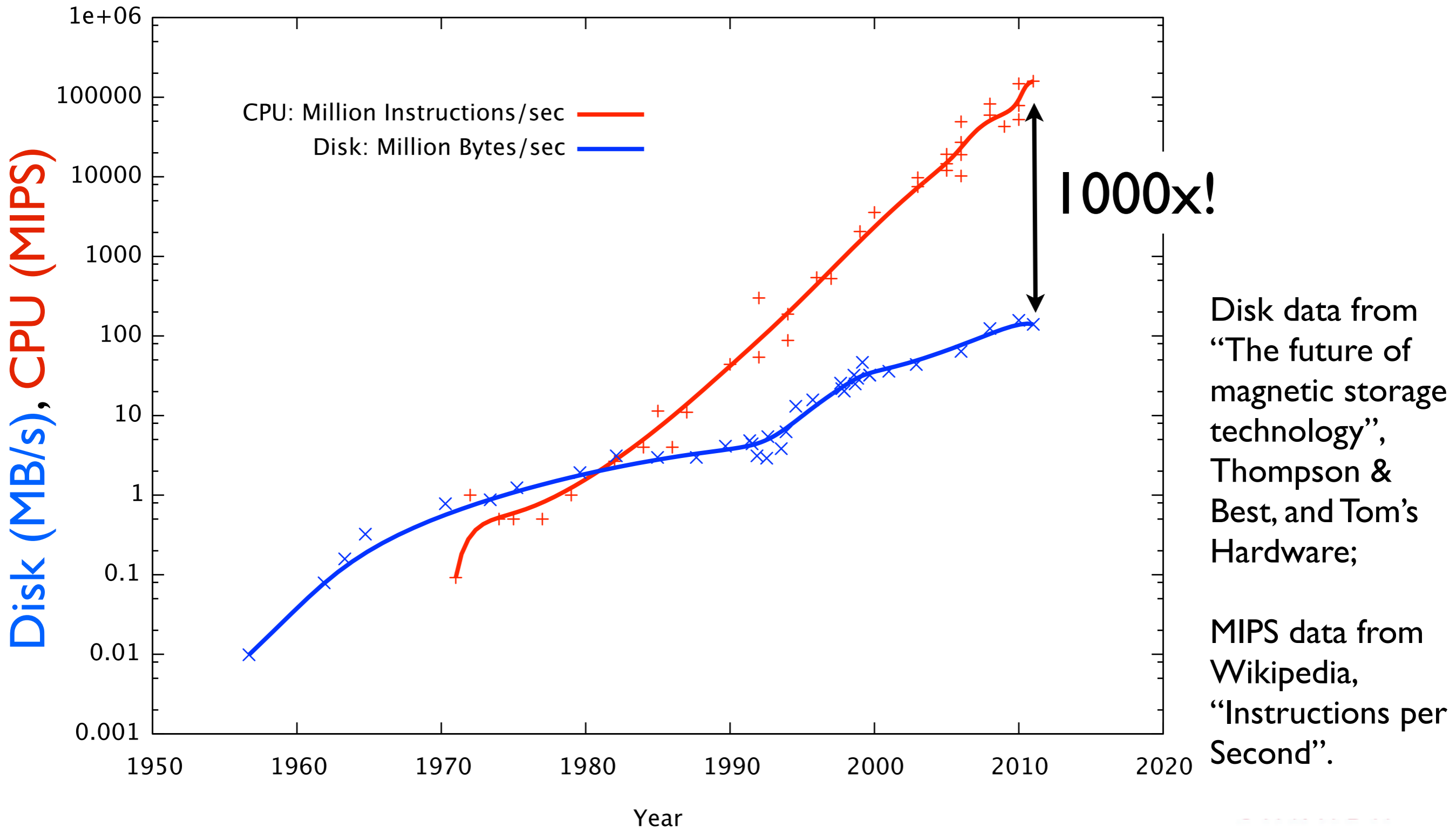
# Data is getting bigger

- Increase in computing power makes simulations larger/more frequent
- Increase in sensor technology makes experiments/observations larger
- Data sizes that used to be measured in MB/GB now measured in TB/PB.
- Easier to generate the data than to store it:



Economist, 27 Feb 2010

# Disks are slower than CPUs (and getting slower)



# Understanding storage performance

- Data rate: MB/s
  - Peak or sustained
  - Write faster than read
  - Network analogy: bandwidth
- IOPS: I/O Operations Per Second
  - open, close, seek, read, write
  - Network analogy: latency



# Understanding storage performance

Device	Bandwidth (MB/s)	IOPS
SATA HDD	100	100
SSD	250	10000

HD:

Open, Write, Close 1000x1kB files: 30.01s (eff: 0.033 MB/s)

Open, Write, Close 1x1MB file: 40ms (eff: 25 MB/s)

# Understanding storage performance

Device	Bandwidth (MB/s)	IOPS
SATA HDD	100	100
SSD	250	10000

SSD:

Open, Write, Close 1000x1kB files: 300ms (eff: 3.3 MB/s)

Open, Write, Close 1x1MB file: 4ms (eff: 232 MB/s)



# Understanding storage performance

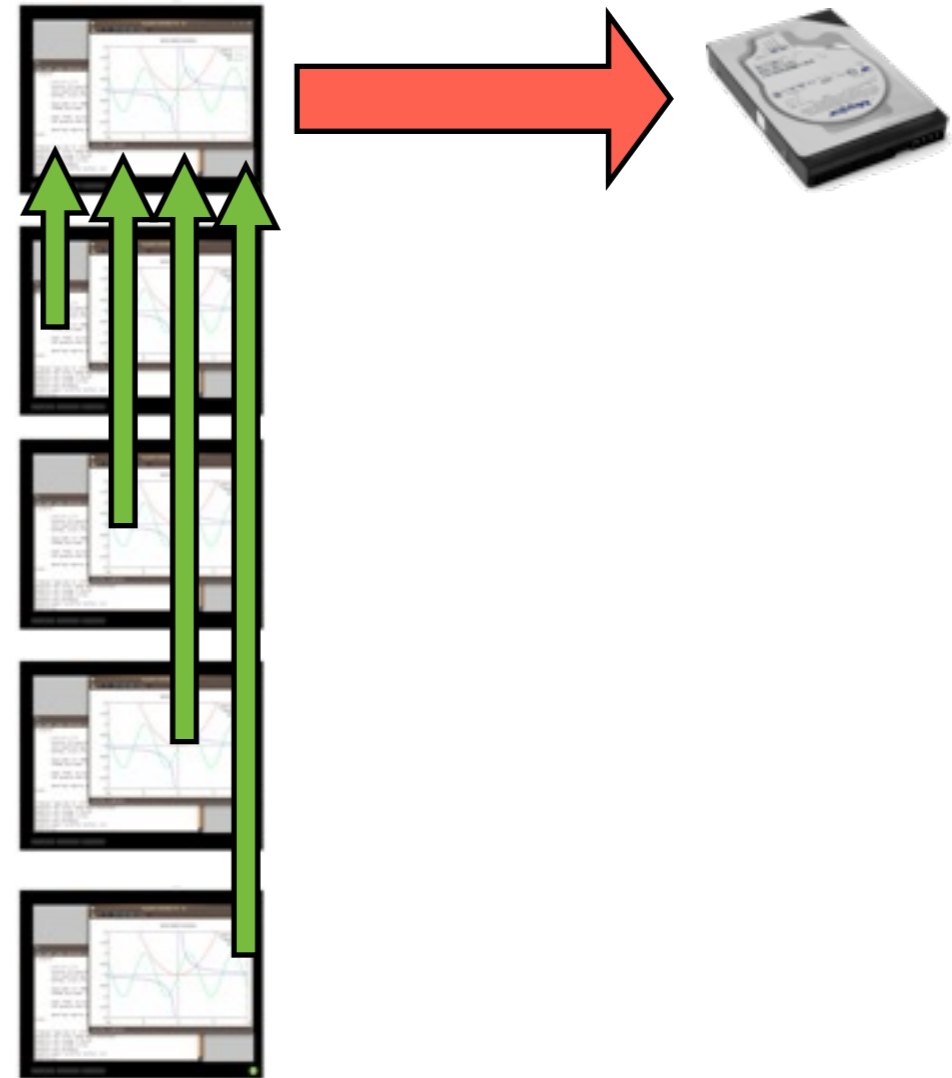
- SSDs are much faster at IOPS:
  - No physical mechanisms that must move to file position
  - Read ~ as fast as write
- But still latency at controller, system calls, etc.
- SSDs will always be much more expensive than disk per unit storage - disk isn't going away.





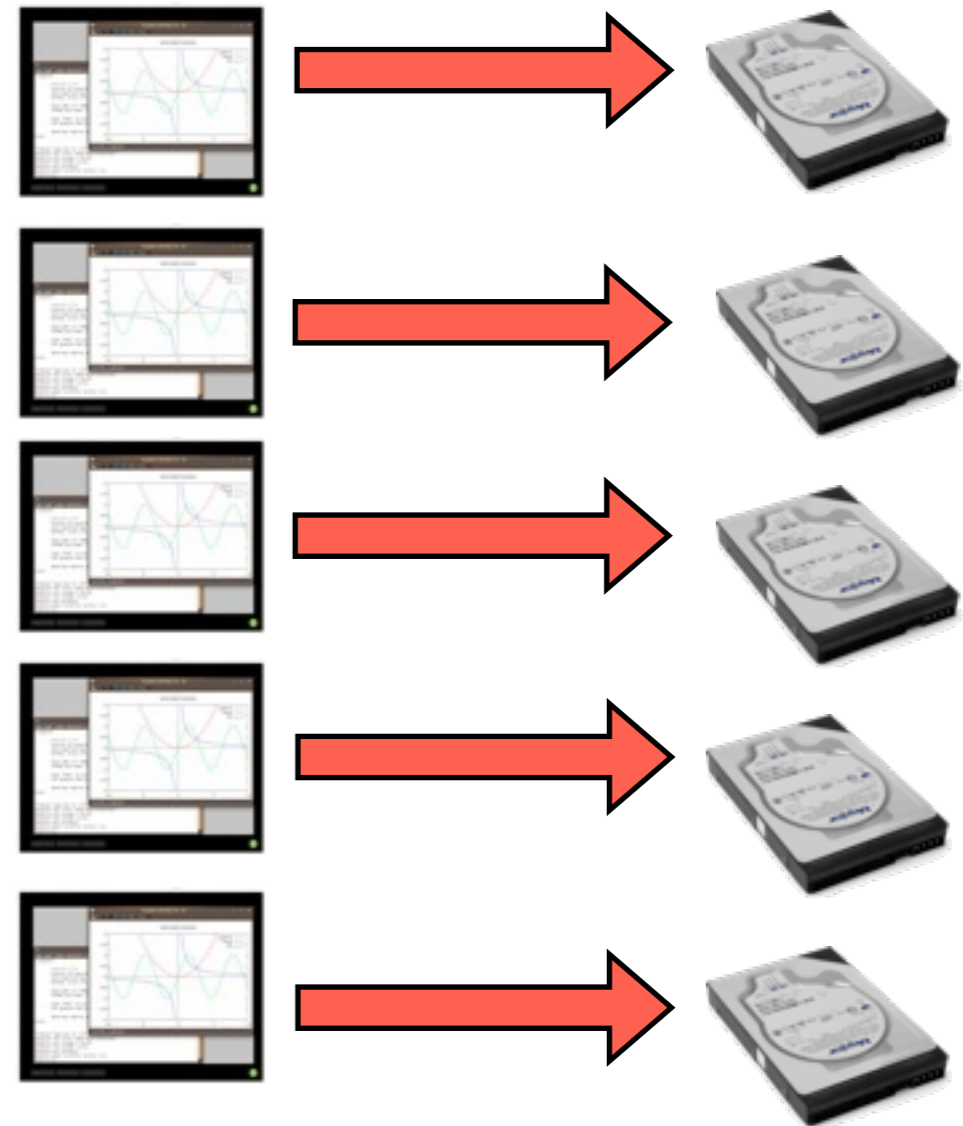
# Planning your I/O

- Parallel computation, several options.
- Everyone sends data to process 0
- Process 0 writes.
- Serialize I/O - huge bottleneck.



# Planning your I/O

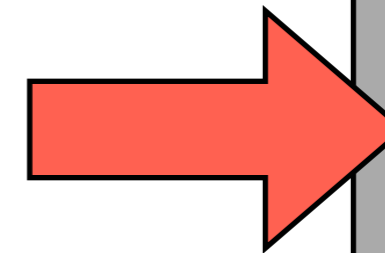
- Parallel computation, several options.
- Each process writes a file, possibly to local disk.
- Postpones the problem - how do you analyze, or restart with different # of procs?



# Planning your I/O

Parallel FS

- Parallel computation, several options.
- We're going to learn to avoid doing this by using Parallel I/O
- **Coordinated** single output of multiple processes.



HPCSC2011



compute + calcul  
CANADA

# Where, how you do I/O matters.

- Binary - smaller files, *much* faster to read/write.
- You're not going to read GB/TB of data yourself; don't bother trying.
- Write in 1 chunk, rather than a few #s at a time.

<b>Large Parallel File System</b>	
<b>ASCII</b> 173s	<b>binary</b> 6s
<b>Ramdisk</b>	
<b>ASCII</b> 174s	<b>binary</b> 1s
<b>Typical work station disk</b>	
<b>ASCII</b> 260s	<b>binary</b> 20s

Timing data: writing 128M  
double-precision numbers



# Where, how you do I/O matters.

- All disk systems do best when reading/writing large, contiguous chunks
- I/O operations (IOPS) are themselves expensive
- moving around within a file
- opening/closing
- Seeks - 3-15ms - enough time to read 0.75 MB!

**Typical work station disk**

**binary - one large read**

14s

**binary - 8k at a time**

20s

**binary - 8k chunks, lots of seeks**

150s

**binary - seeky + open and closes**

205s

Timing data: reading 128M  
double-precision numbers



# Where, how you do I/O matters.

- RAM is much better for random accesses
- Use right storage medium for the job!
- Where possible, read in contiguous large chunks, do random access in memory
- Much better *if* you use most of data read in

## **Large Parallel File System**

<b>ASCII</b>	<b>binary</b>
173s	6s

## **Ramdisk**

<b>ASCII</b>	<b>binary</b>
174s	1s

## **Typical work station disk**

<b>ASCII</b>	<b>binary</b>
260s	20s

## **Ramdisk**

**binary - one large read**  
1s

**binary - 8k at a time**  
1s

**binary - 8k chunks, lots of seeks**  
1s

**binary - seeky + open and closes**  
1.5s



# Parallel I/O and large file systems

- Large disk systems featuring many servers, disks
- Can serve files to many clients concurrently
- Parallel File Systems -
- Lustre, Panasas, GlusterFS, Ceph, GPFS...



SciNet ~2k drives



HPCS2011



compute • calcul  
CANADA

# SciNet's File Systems

- 2x DCS9900 couplets
- 1,790 1TB SATA disk drives
- 1.4 PB of storage
- Single GPFS domain, accessed by all machines (TCS and GPC).
- TCS: through IB. GPC: through GigE (even IB nodes).





# SciNet's File Systems

- Designed for HPC workloads
- High bandwidth to large files - big data.
- Does not do well with millions of little files:
  - wastes huge amount of disk space (4MB block size)
  - lots of small scattered access is *terrible* for performance, even on desktop; multiply by hundreds of processors, can be disastrous



# Understanding storage performance

Device	Bandwidth (MB/s)	IOPS
SATA HDD	100	100
SSD	250	10000
SciNet GPFS	5000	30000

(SciNet GPFS used by ~3000 nodes.)



# Where, how you do I/O matters.

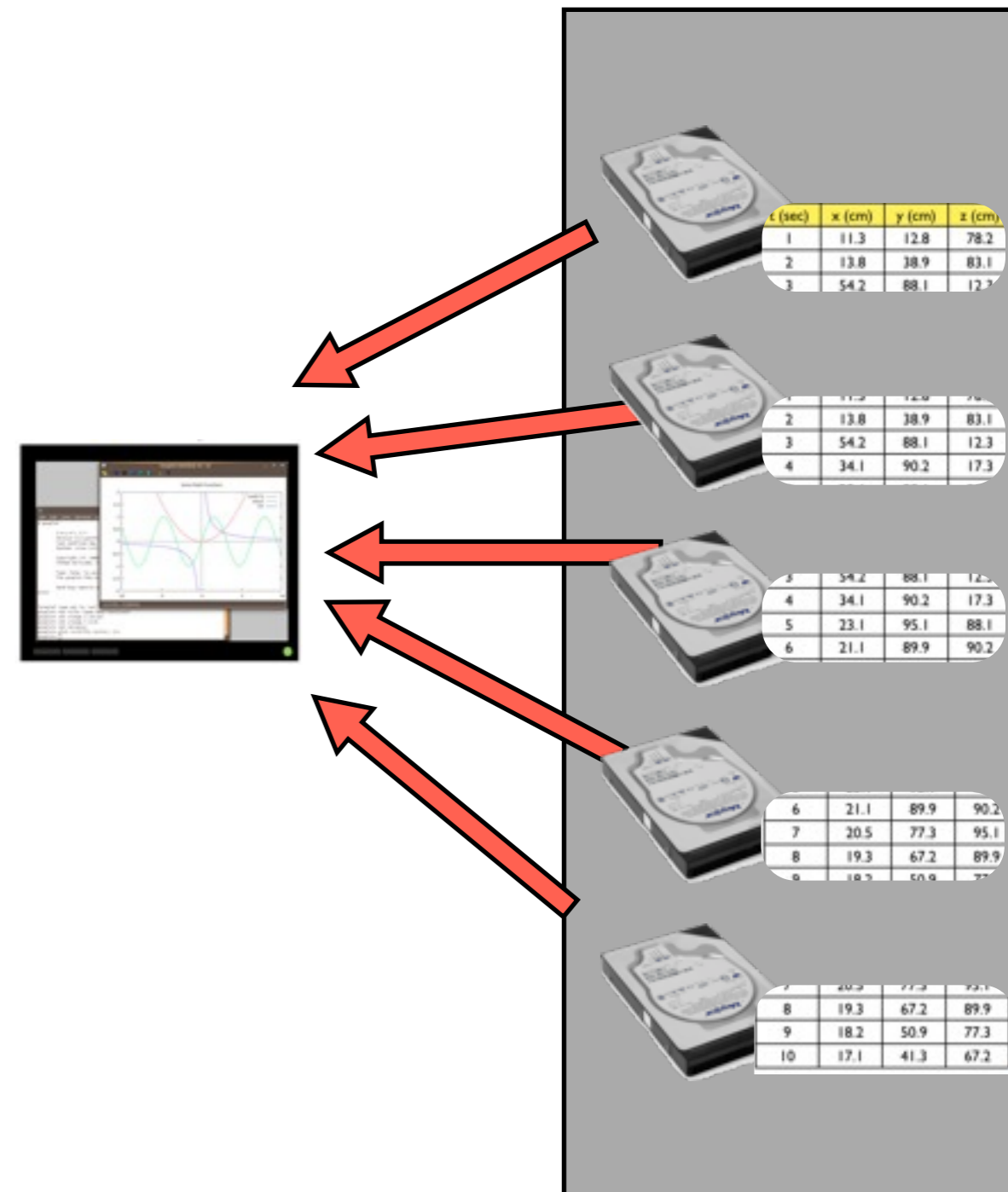
- Well built parallel file systems can greatly increase bandwidth
- But typically even **worse** penalties for seeky/IOPSy operations (coordinating all those disks.)
- Parallel FS can help with big data in two ways

<b>Large Parallel File System</b>	
<b>ASCII</b>	<b>binary</b>
173s	6s
<b>Ramdisk</b>	
<b>ASCII</b>	<b>binary</b>
174s	1s
<b>Typical work station disk</b>	
<b>ASCII</b>	<b>binary</b>
260s	20s

<b>Large Parallel File System</b>	
<b>binary - one large read</b>	
7.5s	
<b>binary - 8k at a time</b>	
62 s	
<b>binary - 8k chunks, lots of seeks</b>	
428 s	
<b>binary - seeky + open and closes</b>	
2137 s	

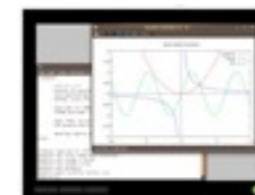
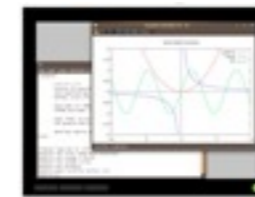
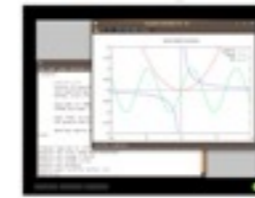
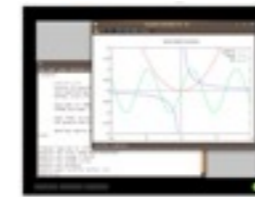
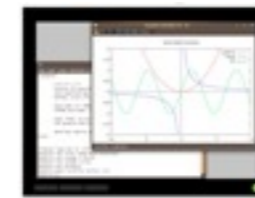
# Striping data across disks

- Single client can make use of multiple disk systems simultaneously
- “Stripe” file across many drives
- One drive can be finding next block while another is sending current block



# Parallel operations on separate data

- Or can do truly parallel operations
- multiple clients doing independent work
- Easy parallelism (good for lots of small data) - process many small files separately

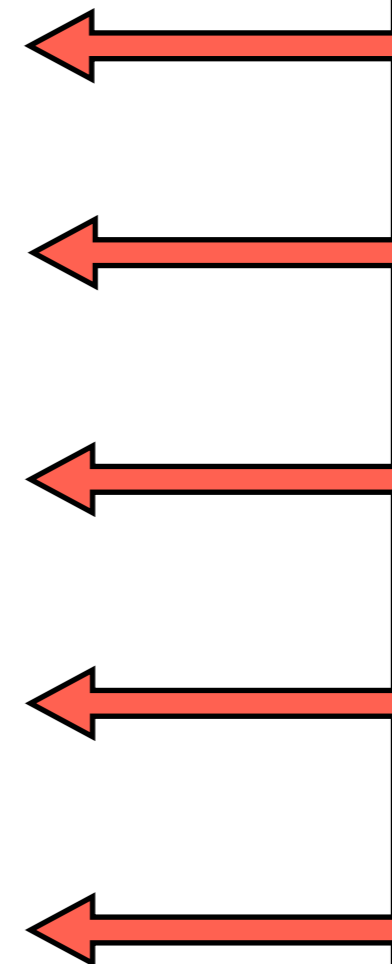


Parallel FS



# Parallel operations on separate data

- Or can do truly parallel operations
- multiple clients doing independent work
- Easy parallelism (good for lots of small data) - process many small files separately
- Harder parallelism - each does part of a larger analysis job on a big file.

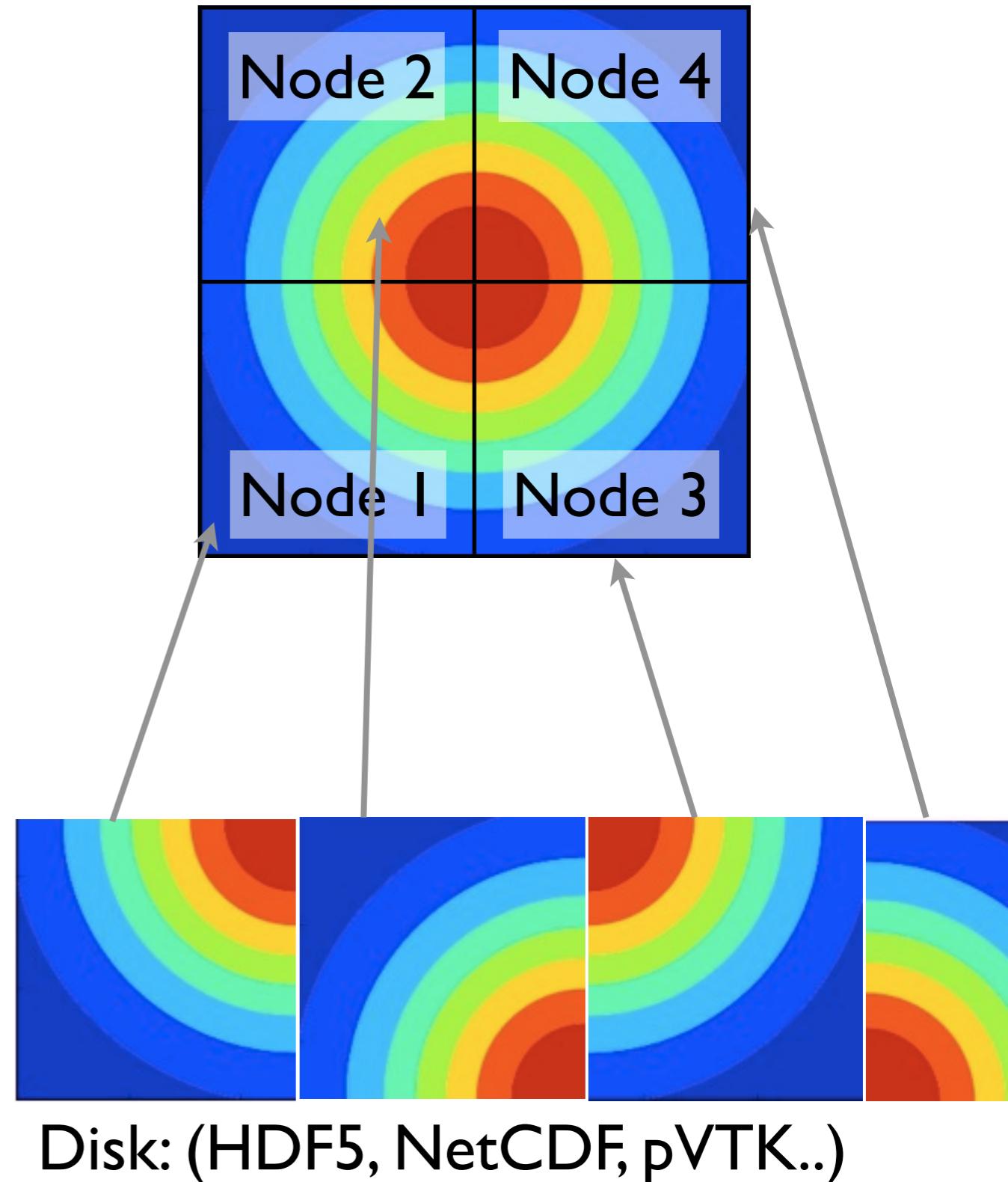


Parallel FS



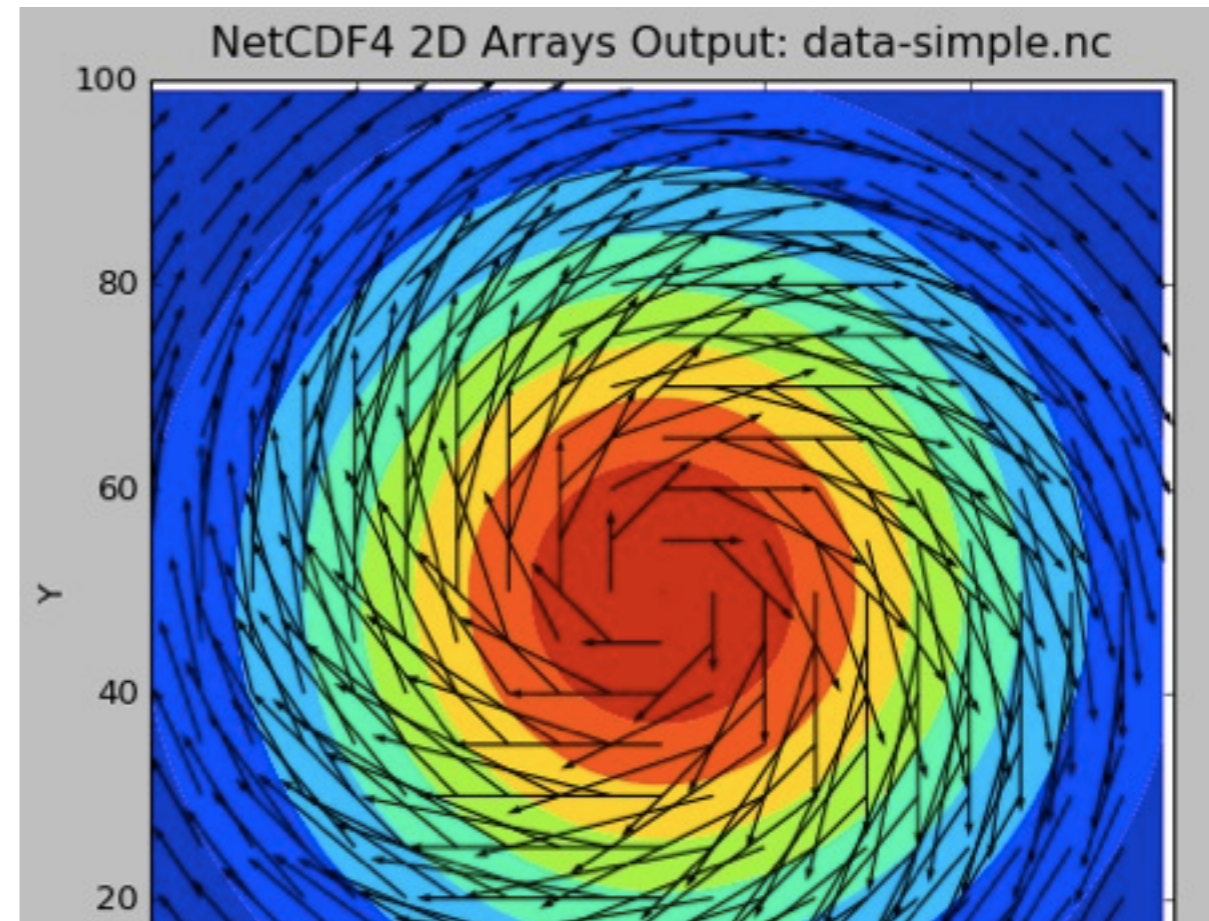
# Data files must take advantage of parallel I/O

- 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*

- 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



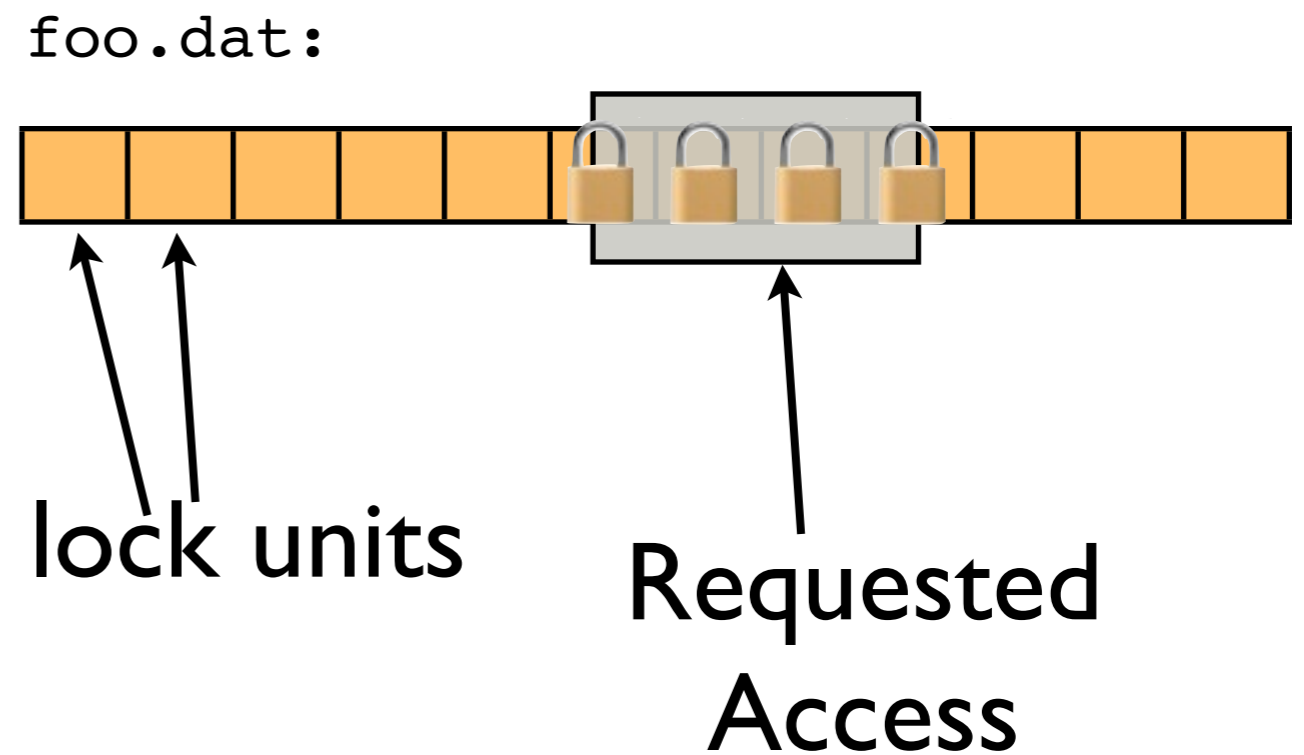
```
$ ncdump -h data-simple-fort.nc
netcdf data-simple-fort {
  dimensions:
    X = 100 ;
    Y = 100 ;
    velocity components = 2 ;
  variables:
    double Density(Y, X) ;
    double Velocity(Y, X, velocity components) ;
}
```

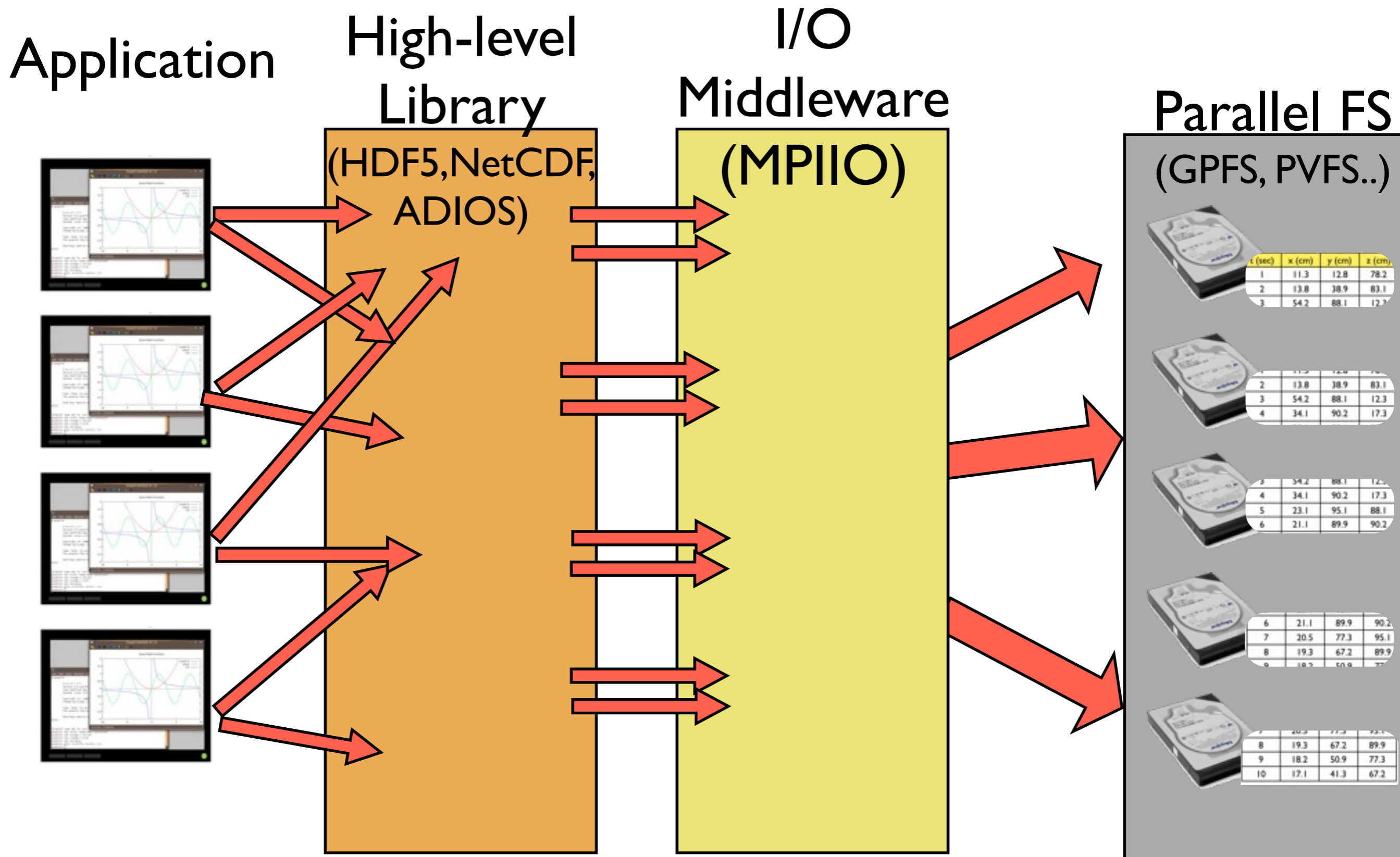




# Coordinating I/O

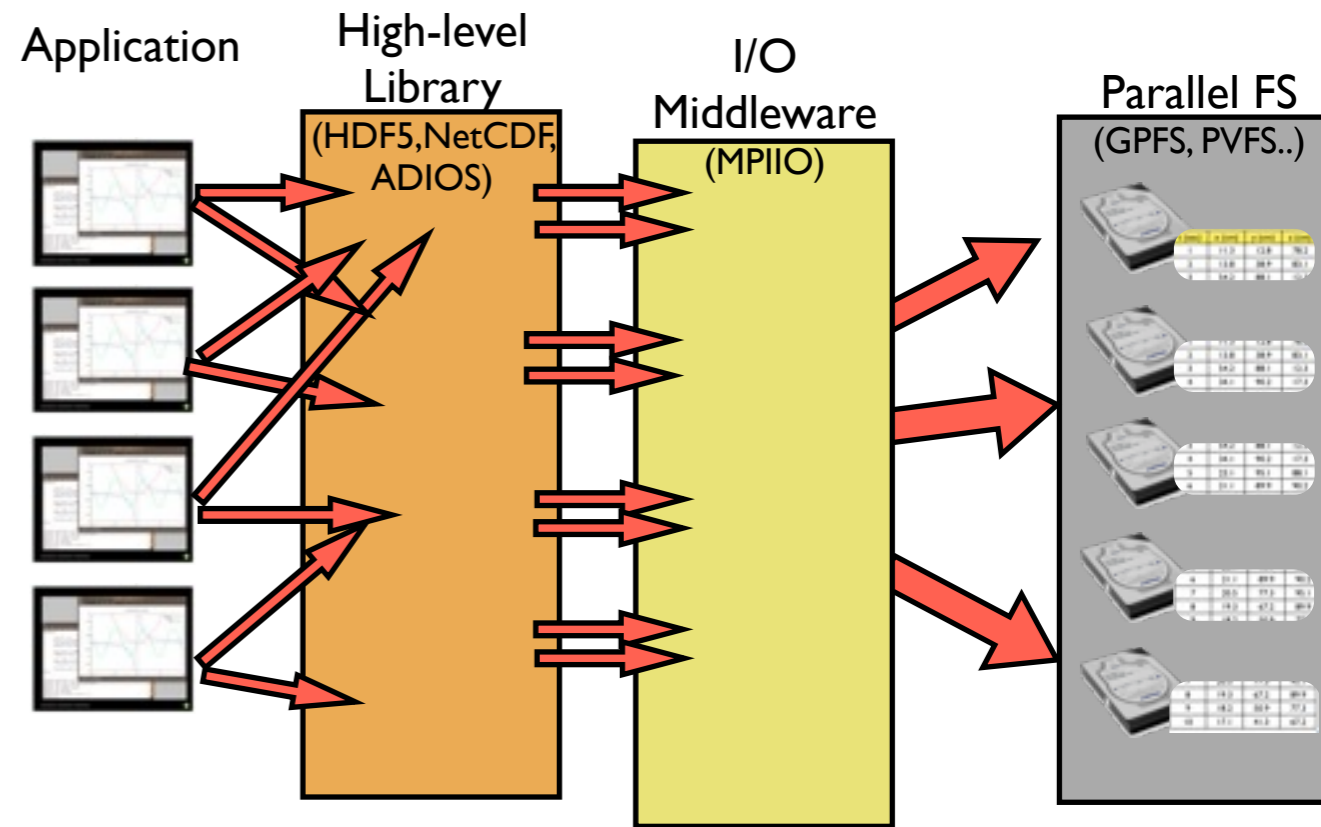
- Multiple nodes all accessing same filesystem.
- To prevent anarchy, locks for some requested accesses.
- File broken up into lock units, locks handed out upon request.
- “False sharing”, etc, possible.
- Files **and** directories.
- Makes (some) IOPS even more expensive





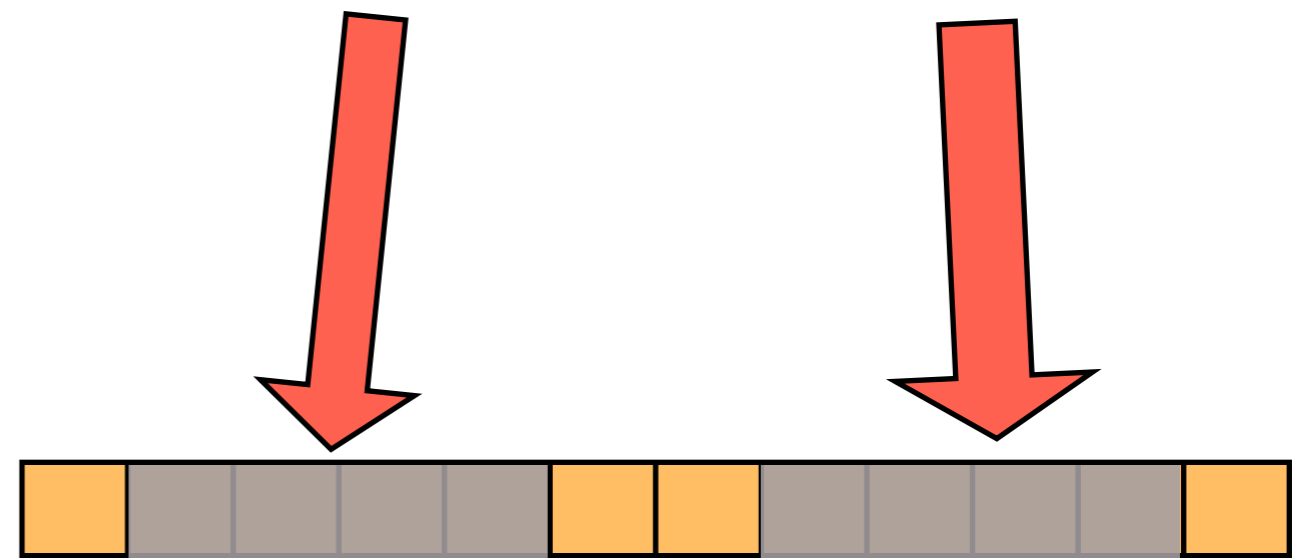
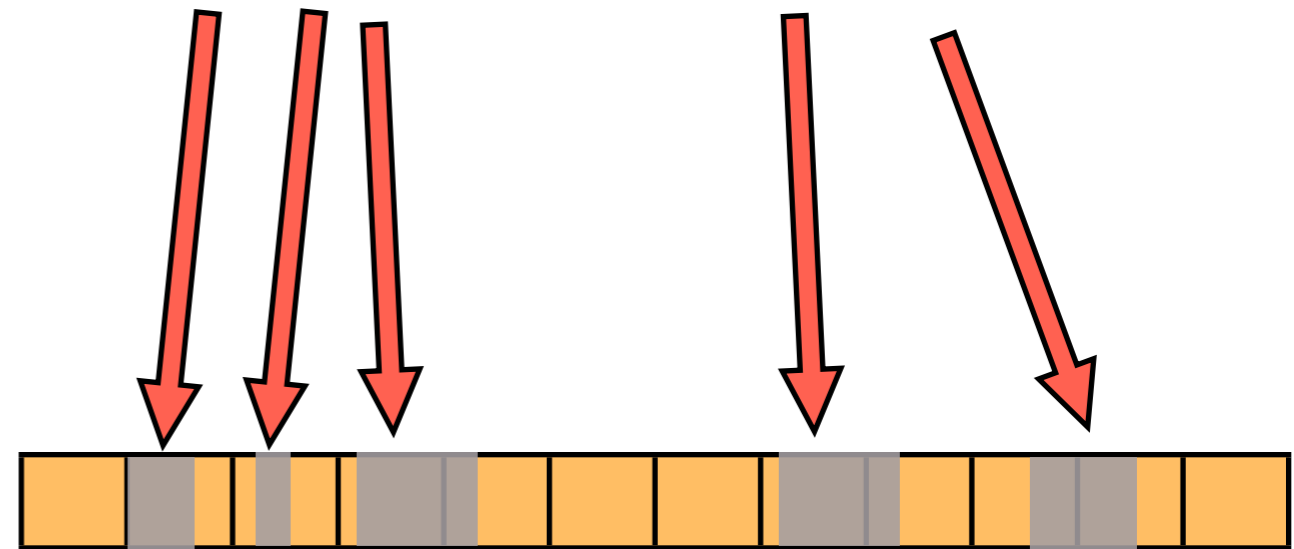
# Abstraction Layers

- 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
  - Data Sieving
  - 2-phase I/O



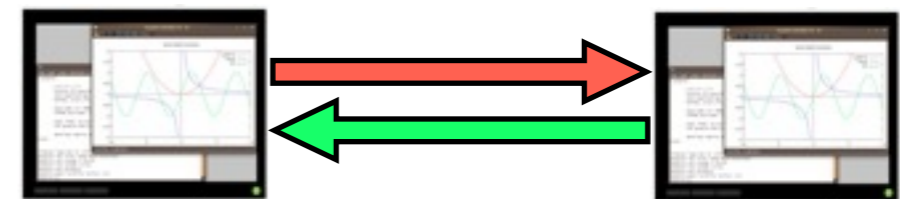
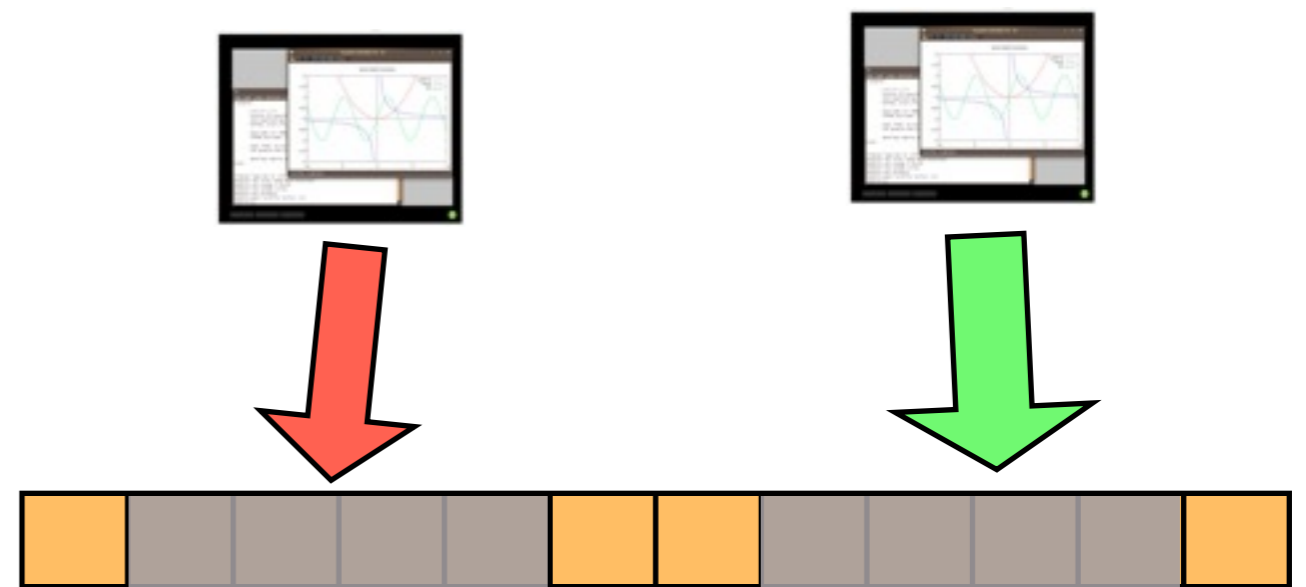
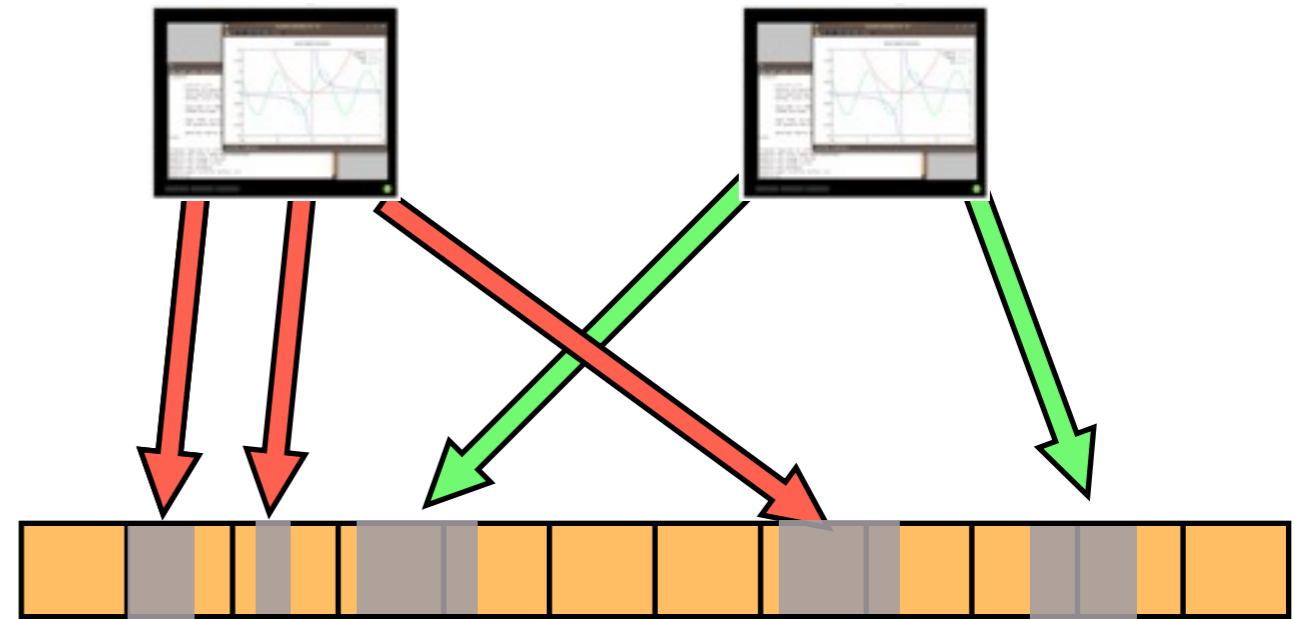
# Data Sieving

- Combine many non-contiguous IO requests into fewer, bigger IO requests
- “Sieve” unwanted data out
- Reduces IOPS, makes use of high bandwidth for sequential IO



# Two-Phase IO

- Collect requests into larger chunks
- Have individual nodes read big blocks
- Then use network communications to exchange pieces
- Fewer IOPS, faster IO
- Network communication usually faster



# MPI-IO

- Part of MPI-2 standard
- Started at IBM Watson
- Maps I/O reads and writes to message passing
- ROMIO is implementation found in MPICH2, OpenMPI
- Really only widely-available scientific computing parallel I/O middleware



```
$ source /groups/h/hpc2011/bin/Parallel_IO.csh
$ cd parIO/samples/mpio
$ make
....

$ mpiexec -n 4 ./helloworldc
Rank 0 has message <Hello >
Rank 1 has message <World!>
Rank 2 has message <Hello >
Rank 3 has message <World!>
Rank 4 has message <Hello >

$ cat helloworld.txt
Hello World!Hello World! $
```

# helloworldc.c

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char **argv) {
    int ierr, rank, size;
    MPI_Offset offset;
    MPI_File file;
    MPI_Status status;
    const int msgsize=6;
    char message[msgsize+1];

    ierr = MPI_Init(&argc, &argv);
    ierr|= MPI_Comm_size(MPI_COMM_WORLD, &size);
    ierr|= MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if ((rank % 2) == 0) strcpy (message, "Hello "); else strcpy (message, "World!");

    offset = (msgsize*rank);

    MPI_File_open(MPI_COMM_WORLD, "helloworld.txt", MPI_MODE_CREATE|MPI_MODE_WRONLY,
                 MPI_INFO_NULL, &file);
    MPI_File_seek(file, offset, MPI_SEEK_SET);
    MPI_File_write(file, message, msgsize, MPI_CHAR, &status);
    MPI_File_close(&file);

    MPI_Finalize();
    return 0;
}
```





# helloworldf.f90

```
program MPIIO_helloworld
  use mpi
  implicit none

  integer(mpi_offset_kind) :: offset
  integer, dimension(mpi_status_size) :: wstatus
  integer, parameter :: msgsize=6
  character(msgsize) :: message
  integer :: ierr, rank, comsize, fileno

  call MPI_Init(ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

  if (mod(rank,2) == 0) then
    message = "Hello "
  else
    message = "World!"
  endif

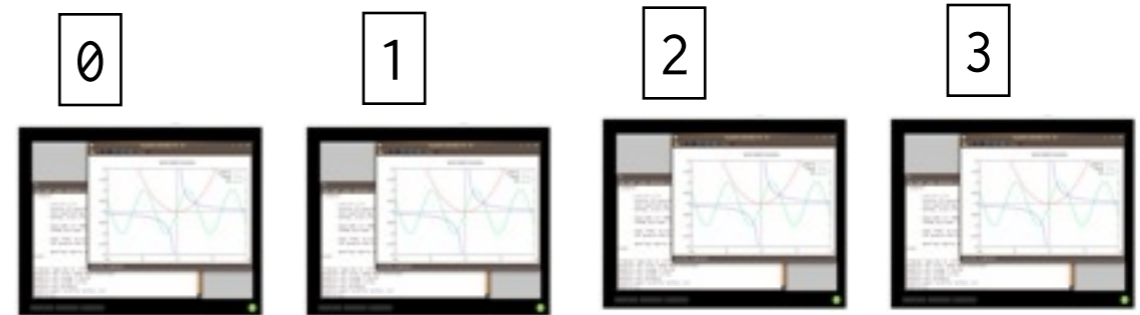
  offset = rank*msgsize

  call MPI_File_open(MPI_COMM_WORLD, "helloworld.txt", ior(MPI_MODE_CREATE, MPI_MODE_WRONLY), &
    MPI_INFO_NULL, fileno, ierr)
  call MPI_File_seek (fileno, offset, MPI_SEEK_SET, ierr)
  call MPI_File_write(fileno, message, msgsize, MPI_CHARACTER, wstatus, ierr)
  call MPI_File_close(fileno, ierr)

  call MPI_Finalize(ierr)

end program MPIIO_helloworld
```

# MPI-IO Hello World



```
mpiexec -np 4 ./helloworldc
```

# MPI-IO

## Hello World

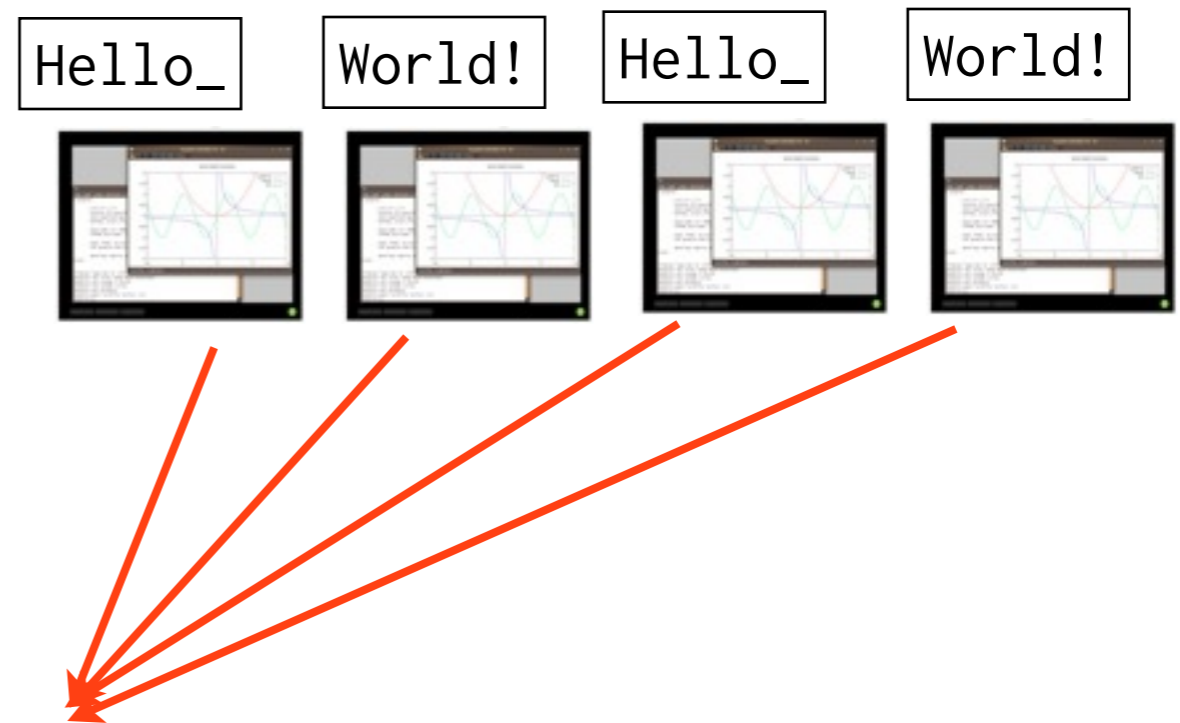


```
if ((rank % 2) == 0)
    strcpy (message, "Hello ");
else
    strcpy (message, "World!");
```



# MPI-IO

# Hello World

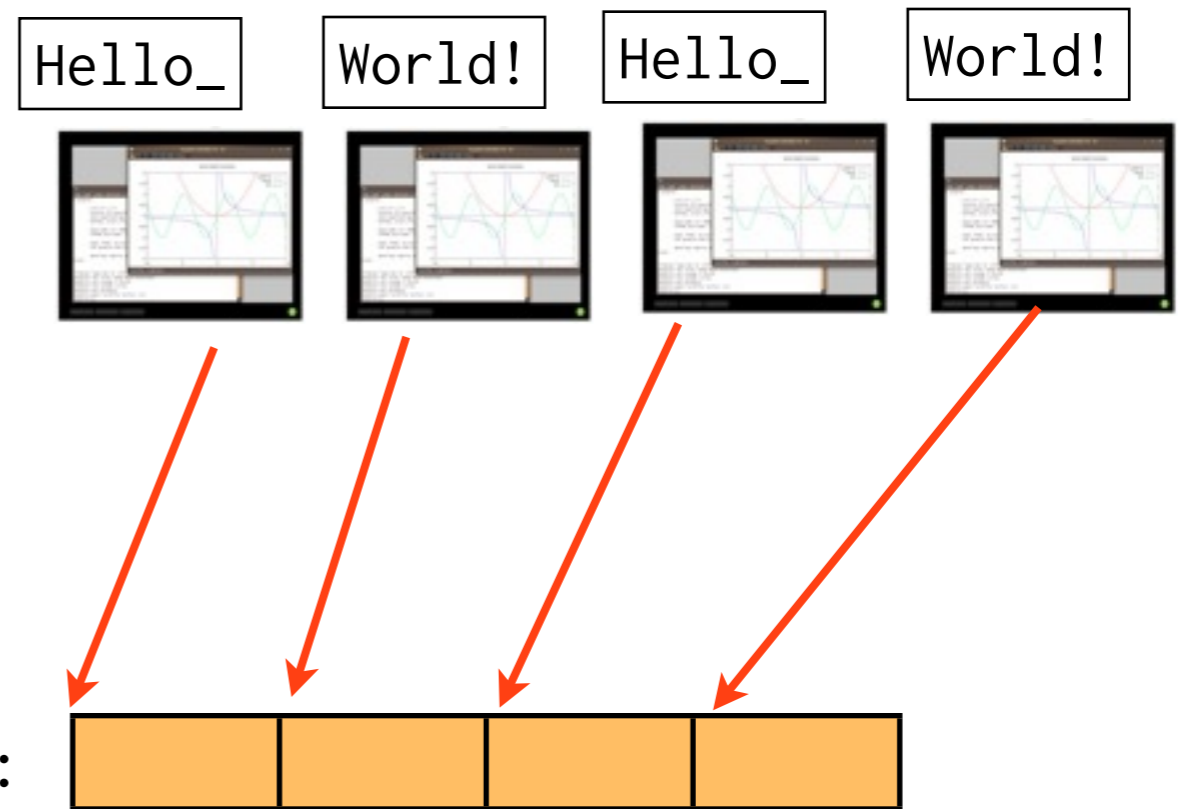


helloworld.txt:

```
MPI_File_open(MPI_COMM_WORLD, "helloworld.txt", MPI_MODE_CREATE|MPI_MODE_WRONLY,  
MPI_INFO_NULL, &file);
```

# MPI-IO

## Hello World

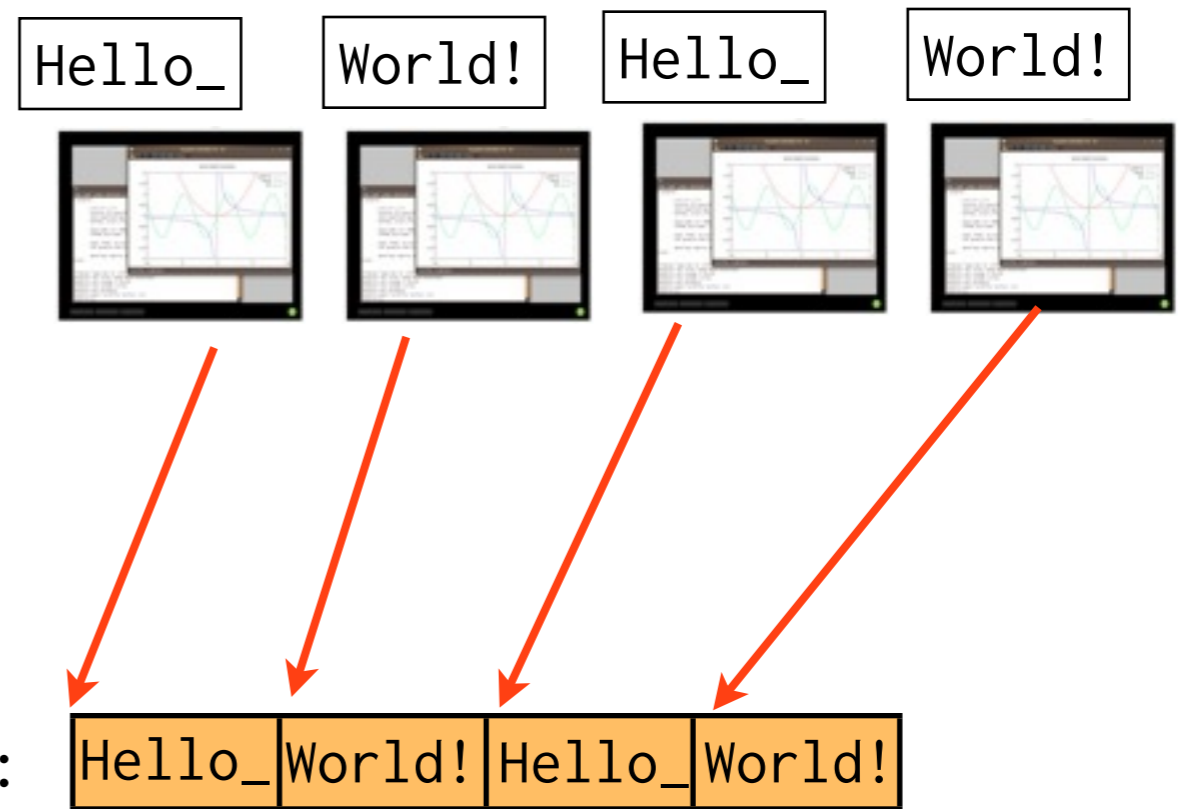


```
offset = (msgsize*rank);
```

```
MPI_File_seek(file, offset, MPI_SEEK_SET);
```



# MPI-IO Hello World



```
MPI_File_write(file, message, msgsize, MPI_CHAR, &status);
```

# MPI-IO

## Hello World



helloworld.txt: `Hello_ World! Hello_ World!`

```
MPI_File_close(&file);
```

# MPI-IO Hello World

Usual MPI  
startup/  
teardown  
boilerplate

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char **argv) {
    int ierr, rank, size;
    MPI_Offset offset;
    MPI_File file;
    MPI_Status status;
    const int msgsize=6;
    char message[msgsize+1];

    ierr = MPI_Init(&argc, &argv);
    ierr|= MPI_Comm_size(MPI_COMM_WORLD, &size);
    ierr|= MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if ((rank % 2) == 0) strcpy (message, "Hello "); else strcpy (message, "World ");

    offset = (msgsize*rank);

    MPI_File_open(MPI_COMM_WORLD, "helloworld.txt", MPI_MODE_CREATE|MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &file);
    MPI_File_seek(file, offset, MPI_SEEK_SET);
    MPI_File_write(file, message, msgsize, MPI_CHAR, &status);
    MPI_File_close(&file);

    MPI_Finalize();
    return 0;
}
```



# MPI\_File\_Open

Communicator;  
**collective**  
operation.

```
call MPI_File_Open( integer      communicator,  
                   character(*) *filename,  
                   integer      mode,  
                   integer      info,  
                   integer      handle,  
                   integer      ierr);
```

```
int MPI_File_Open( MPI_Comm  communicator,  
                  char      *filename,  
                  int       mode,  
                  MPI_Info  info,  
                  MPI_File  *handle);
```



# MPI\_File\_Open

Info allows us to send extra hints to MPI-IO layer about file (performance tuning, special case handling)

MPI\_INFO\_NULL: no extra info.

```
call MPI_File_Open( integer      communicator,  
                   character(*) *filename,  
                   integer      mode,  
                   integer      info,  
                   integer      handle,  
                   integer      ierr);
```

```
int MPI_File_Open( MPI_Comm  communicator,  
                  char      *filename,  
                  int       mode,  
                  MPI_Info  info,  
                  MPI_File  *handle);
```

# Modes for MPI\_File\_Open():

MPI_MODE_RDONLY	read-only
MPI_MODE_RDWR	read-write
MPI_MODE_WRONLY	write-only
MPI_MODE_CREATE	Create if doesn't exist.
MPI_MODE_APPEND	On open, file pointers at end of file.
MPI_MODE_EXCL	Fail if try to create, does exist.
MPI_MODE_UNIQUE_OPEN	No one else is using this.
MPI_MODE_SEQUENTIAL	Will be sequential access only.
MPI_MODE_DELETE_ON_CLOSE	Delete when done. (OOC/scratch).



# MPI\_File\_Seek:

```
int MPI_File_seek( MPI_File mpi_fh,  
                  MPI_Offset offset,  
                  int mode);
```

```
call MPI_File_seek( integer :: mpi_fh,  
                   integer(mpi_offset_kind) :: offset,  
                   integer :: mode  
                   integer :: ierr)
```

MPI_SEEK_SET	Set file pointer to position <i>offset</i>
MPI_SEEK_CUR	pointer ← current position + <i>offset</i>
MPI_SEEK_END	pointer ← end of file - <i>offset</i>

**Not** collective; each adjusts its own local file pointer

# MPI\_File\_write:

```
int MPI_File_write(MPI_File fh,  
                  void *buf,  
                  int count,  
                  MPI_Datatype datatype,  
                  MPI_Status *status)
```

```
call MPI_File_write( integer :: mpi_fh,  
                    buffer,  
                    integer :: count  
                    integer :: datatype  
                    integer :: status(MPI_STATUS_SIZE)  
                    integer :: ierr)
```

**Not** collective; each writes.



# MPI\_File\_write:

- MPI File write is very much like a MPI\_Send.
- “Sending” count of datatype from buf “to” the file.
- Here, writing 6 MPI\_CHARs.
- Contiguous in memory starting in buffer.
- Status like a receive -- contains info about amount of data actually written, etc.

```
int MPI_File_write(MPI_File fh,  
                  void *buf,  
                  int count,  
                  MPI_Datatype datatype,  
                  MPI_Status *status)
```



# MPI\_File\_write:

- To write out data that is non-contiguous in memory, same as MPI\_Send non-contig data:
- Create type that describes data layout in memory
- Send in units of that type.

stride = 2



blocklen = 1

count = 6

```
call MPI_Type_vector(count, blocklen, stride,  
                    MPI_CHARACTER,  
                    everyother, ierr)
```



```
integer, parameter :: msgsize=6, strsize=12
character(strsize) :: message
integer :: everyother
```

## helloworld-noncontigf.f90

```
!....
```

```
if (mod(rank,2) == 0) then
    message = "H@e#l*l^o* A"
else
    message = "WfoQr#l>d@!_"
endif
```

```
!...
```

```
call MPI_Type_vector(msgsize, 1, 2, MPI_CHARACTER, everyother, ierr)
call MPI_Type_commit(everyother, ierr)
```

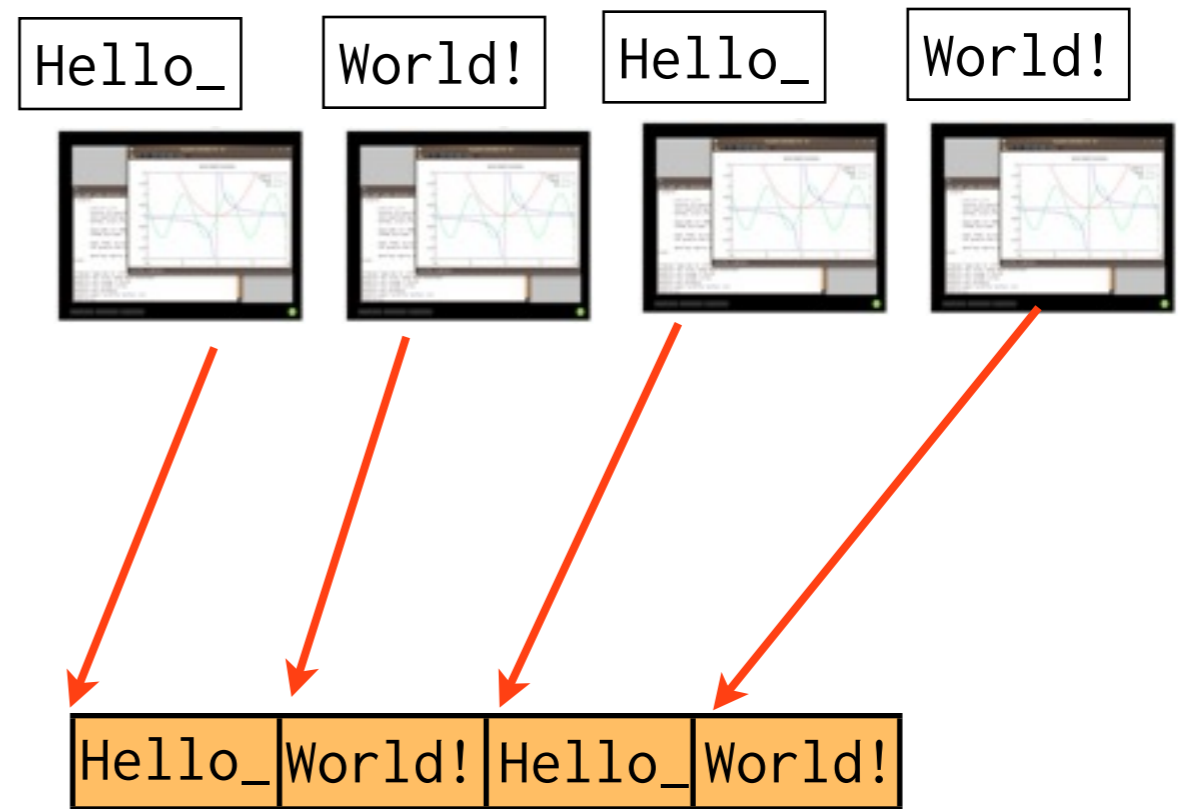
```
call MPI_File_open(MPI_COMM_WORLD, "helloworld-nc.txt", ior(MPI_MODE_CREATE, MPI_MODE_WRONLY), &
    MPI_INFO_NULL, fileno, ierr)
```

```
call MPI_File_seek (fileno, offset, MPI_SEEK_SET, ierr)
call MPI_File_write(fileno, message, 1, everyother, wstatus, ierr)
call MPI_File_close(fileno, ierr)
```



# Seek + Write

- Works, but:
  - Very low level (gets complicated for less trivial data layouts)
  - Completely independent operations (seek, write)
  - Hard for any middleware to coordinate, improve anything.
  - Could do this with POSIX I/O.



# MPI\_File\_write\_at:

```
offset = rank*msgsize
```

```
call MPI_File_open(MPI_COMM_WORLD, "helloworld-at.txt", &  
                  ior(MPI_MODE_CREATE, MPI_MODE_WRONLY), &  
                  MPI_INFO_NULL, fileno, ierr)
```

```
call MPI_File_write_at(fileno, offset, message, msgsize, &  
                      MPI_CHARACTER, wstatus, ierr)
```

```
call MPI_File_close(fileno, ierr)
```

**writeatf.f90**

```
offset = (msgsize*rank);
```

```
MPI_File_open(MPI_COMM_WORLD, "helloworld-at.txt",  
             MPI_MODE_CREATE | MPI_MODE_WRONLY,  
             MPI_INFO_NULL, &file);
```

```
MPI_File_write_at(file, offset, message, msgsize, MPI_CHAR, &status);  
MPI_File_close(&file);
```

**writeatc.c**



# MPI\_File\_write\_at:

```
int MPI_File_write_at(MPI_File fh,  
                     MPI_Offset offset  
                     void *buf,  
                     int count,  
                     MPI_Datatype datatype,  
                     MPI_Status *status)
```

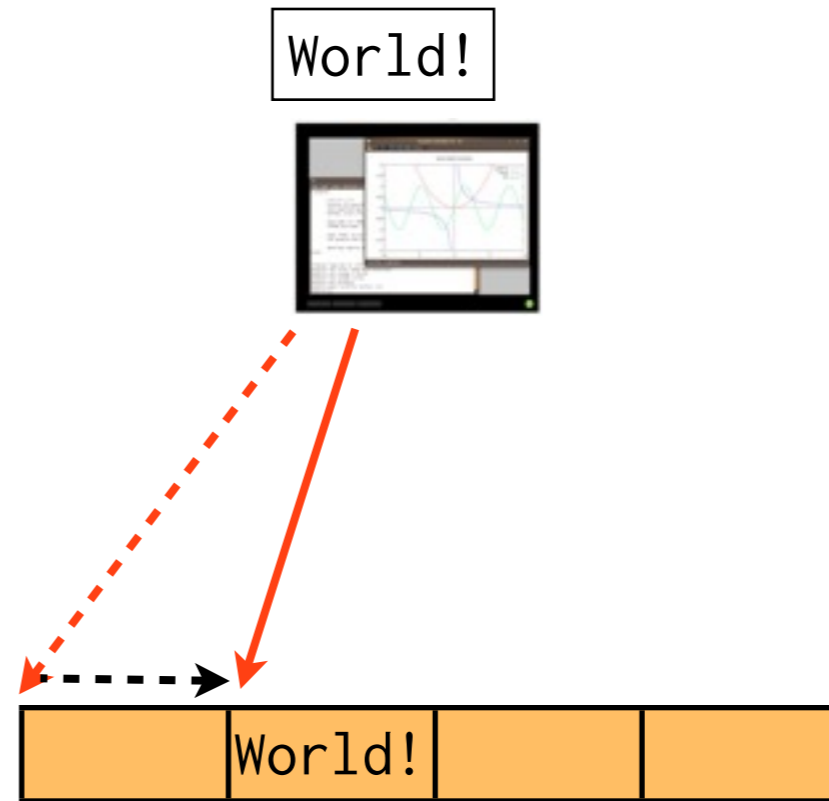
```
call MPI_File_write_at( integer :: mpi_fh,  
                       integer(MPI_OFFSET_KIND) :: offset,  
                       buffer,  
                       integer :: count  
                       integer :: datatype  
                       integer :: status(MPI_STATUS_SIZE)  
                       integer :: ierr)
```

**Writes at a given offset**



# Write\_at

- Seek (relative to current, local, file pointer) + write
- More explicit about what is going to happen (actions aren't independent) - some opt. possible.
- But actions of individual processors still independent - no collective optimization possible.



# MPI\_File\_write\_at\_all:

```
offset = rank*msgsize
```

```
call MPI_File_open(MPI_COMM_WORLD, "helloworld-at.txt",  
                  ior(MPI_MODE_CREATE, MPI_MODE_WRONLY), &  
                  MPI_INFO_NULL, fileno, ierr)
```

```
call MPI_File_write_at_all(fileno, offset, message, msgsize, &  
                           MPI_CHARACTER, wstatus, ierr)
```

```
call MPI_File_close(fileno, ierr)
```

**writeatallf.f90**

```
offset = (msgsize*rank);
```

```
MPI_File_open(MPI_COMM_WORLD, "helloworld-at.txt",  
              MPI_MODE_CREATE | MPI_MODE_WRONLY,  
              MPI_INFO_NULL, &file);
```

```
MPI_File_write_at_all(file, offset, message, msgsize,  
                      MPI_CHAR, &status);
```

```
MPI_File_close(&file);
```

**writeatallc.c**



# MPI\_File\_write\_at\_all:

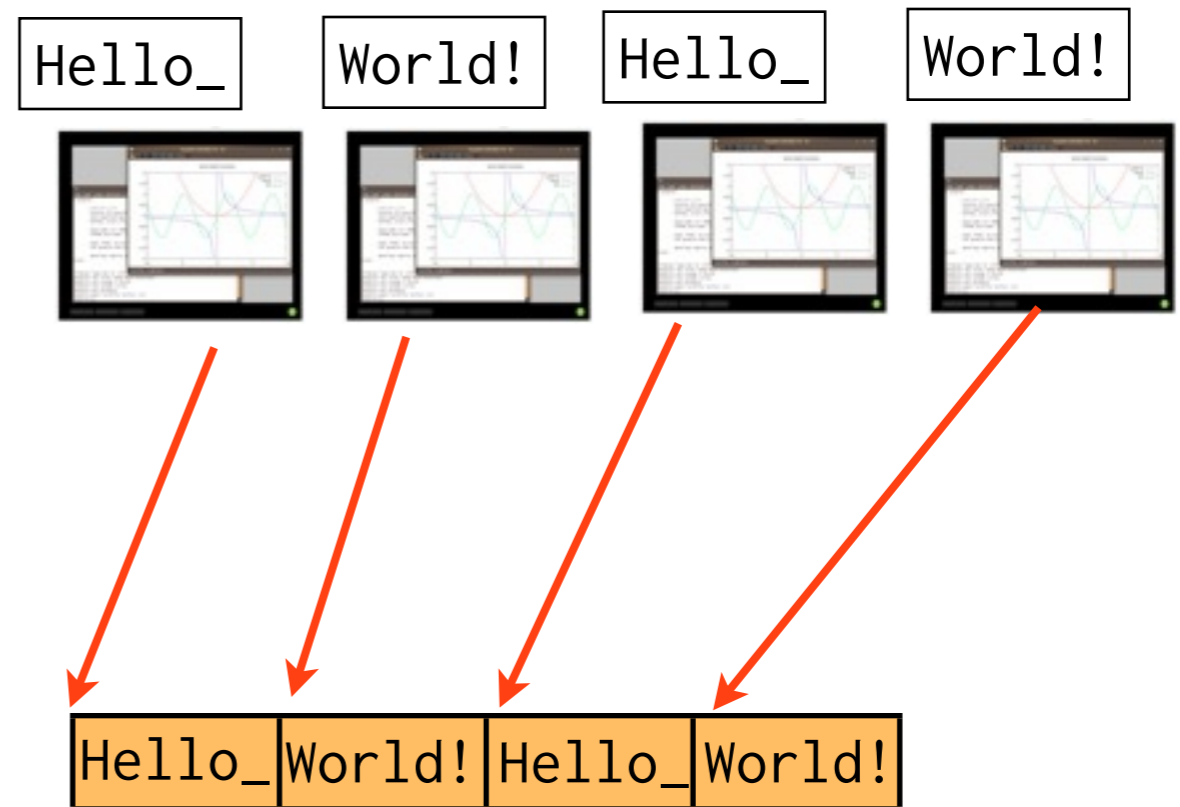
```
int MPI_File_write_at_all(MPI_File fh,  
                          MPI_Offset offset  
                          void *buf,  
                          int count,  
                          MPI_Datatype datatype,  
                          MPI_Status *status)
```

```
call MPI_File_write_at_all( integer :: mpi_fh,  
                            integer(MPI_OFFSET_KIND) :: offset,  
                            buffer,  
                            integer :: count  
                            integer :: datatype  
                            integer :: status(MPI_STATUS_SIZE)  
                            integer :: ierr)
```

**Writes at a given offset - Collective!**



# Write\_at\_all



- Much more explicit about what is going to happen globally.
- Collective operation.
- All processors participate.
- Higher order optimizations possible.



# Non-contiguous in file

- Imagine having to write out a 2d file as to the right, with rank 0 “owning” the yellow data, etc.
- (eg, an image, or a complete checkpoint of a 2d domain).
- Would have to do repeated seeks, writing out one row at a time...

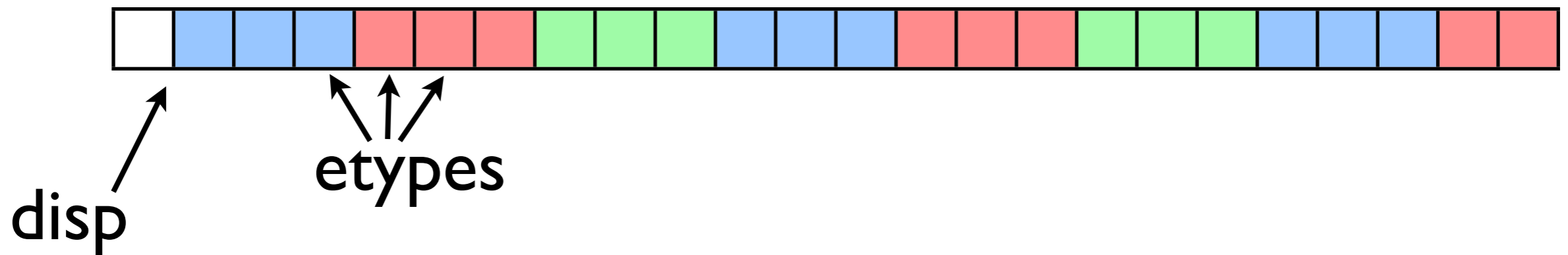
0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
2	2	2	2	2	3	3	3	3	3
2	2	2	2	2	3	3	3	3	3
2	2	2	2	2	3	3	3	3	3
2	2	2	2	2	3	3	3	3	3





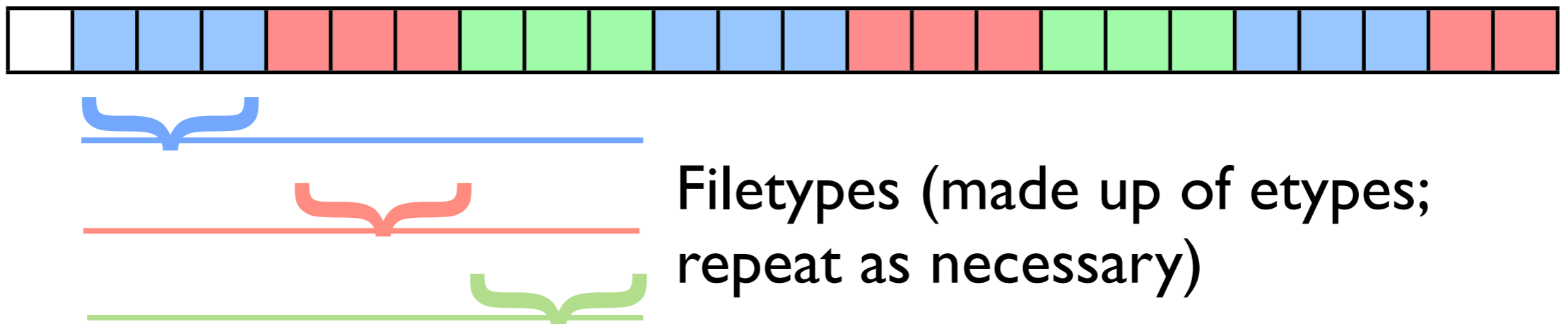
# MPI-IO File View

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



# MPI-IO File View

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



# MPI\_Type\_create\_subarray

- MPI\_Type\_create\_subarray ; piece of a multi-dimensional array.
- *Much* more convenient for higher-dimensional arrays
- (Otherwise, need vectors of vectors of vectors...)
- Here - starts = [0,0], subsizes=[5,5], sizes=[10,10].

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

```
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 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**.



## fileviewc.c

```
MPI_Type_create_subarray(2, globalsize, subsize, start, MPI_ORDER_C,  
                        MPI_CHAR, &viewtype);  
MPI_Type_commit(&viewtype);  
  
offset = 0;  
  
MPI_File_open(MPI_COMM_WORLD, "viewtype.txt",  
             MPI_MODE_CREATE|MPI_MODE_WRONLY,  
             MPI_INFO_NULL, &file);  
  
MPI_File_set_view(file, offset, MPI_CHAR, viewtype,  
                 "native", MPI_INFO_NULL);  
  
MPI_File_write_all(file, &(mydata[0][0]), locnrows*locncols, MPI_CHAR, &status);  
MPI_File_close(&file);
```

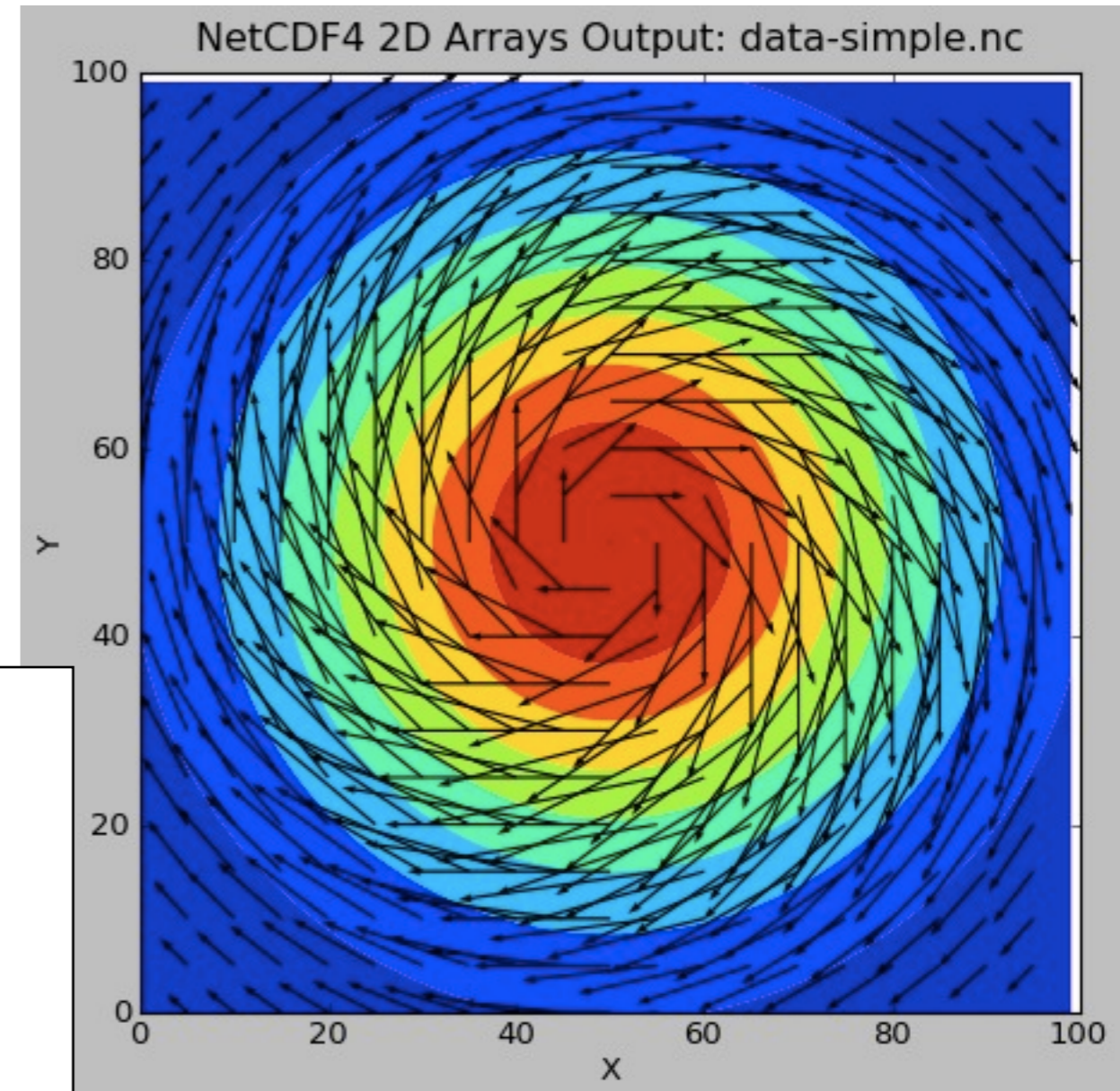
write  $\text{locnrows} * \text{locncols}$  contiguous MPI\_CHARs to  
the (non-contiguous) view in file.

# Formats for Scientific Data Management

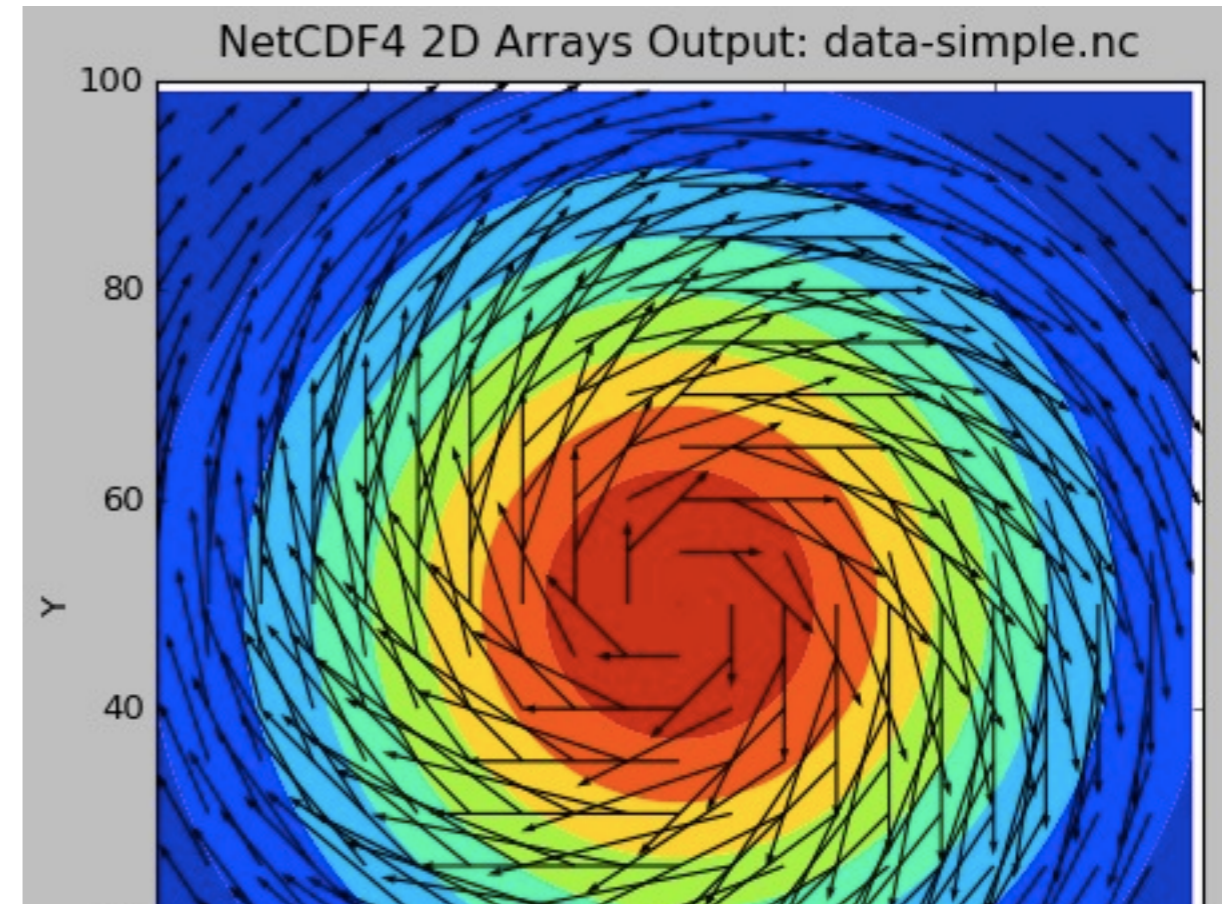
NetCDF4, HDF5, and whatnot

# Sample Code

```
$ cd  
$ cd parIO/netcdf  
  
$ source ../seriallibs  
$ make 2darray-simple (C), or  
$ make f2darray-simple (F90)  
  
$ ./{f,}2darray-simple  
  
$ ls *.nc  
$ ../plots.py *.nc
```



# 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.
```

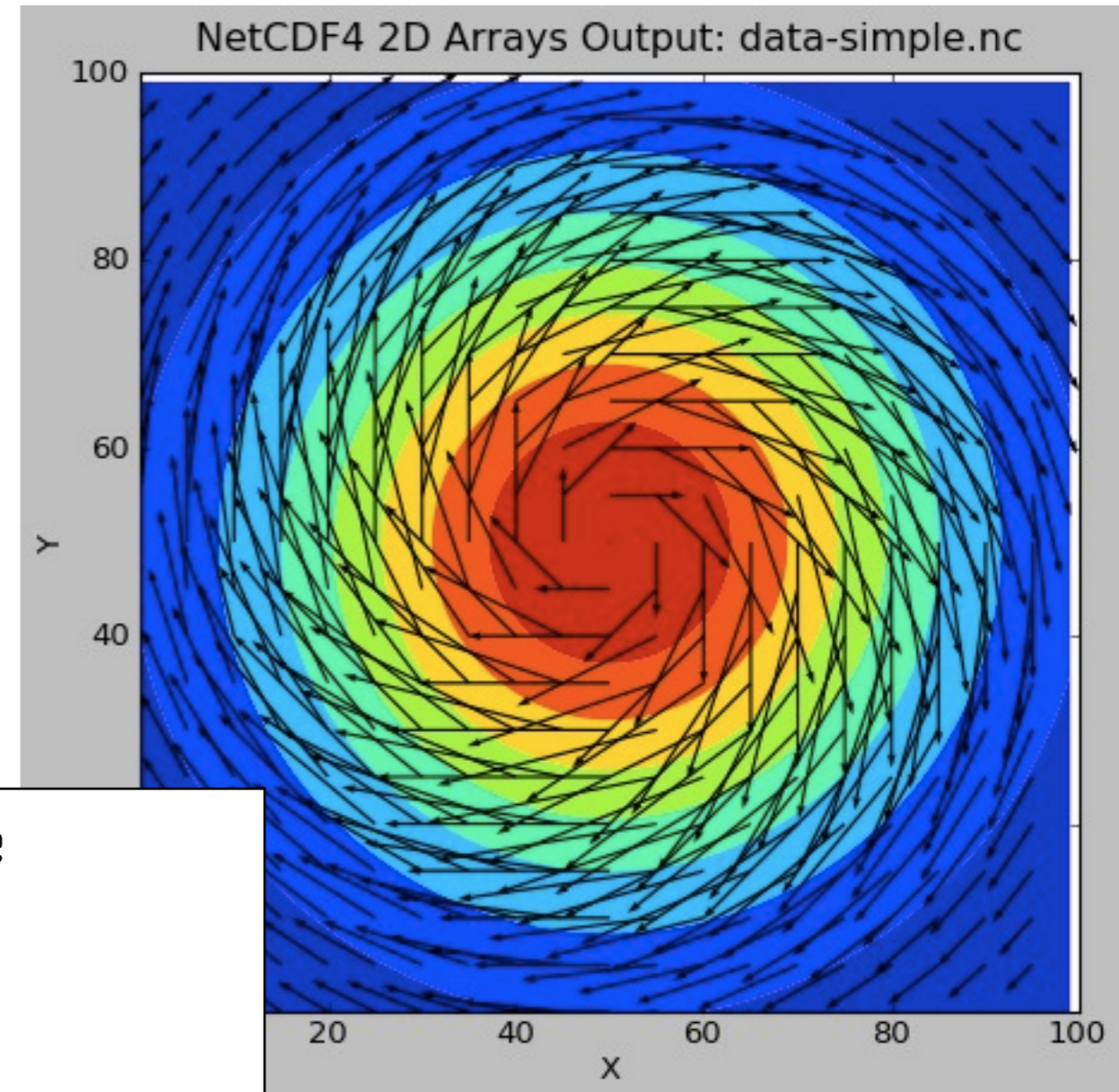
```
$ ./f2darray-simple --help
```

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



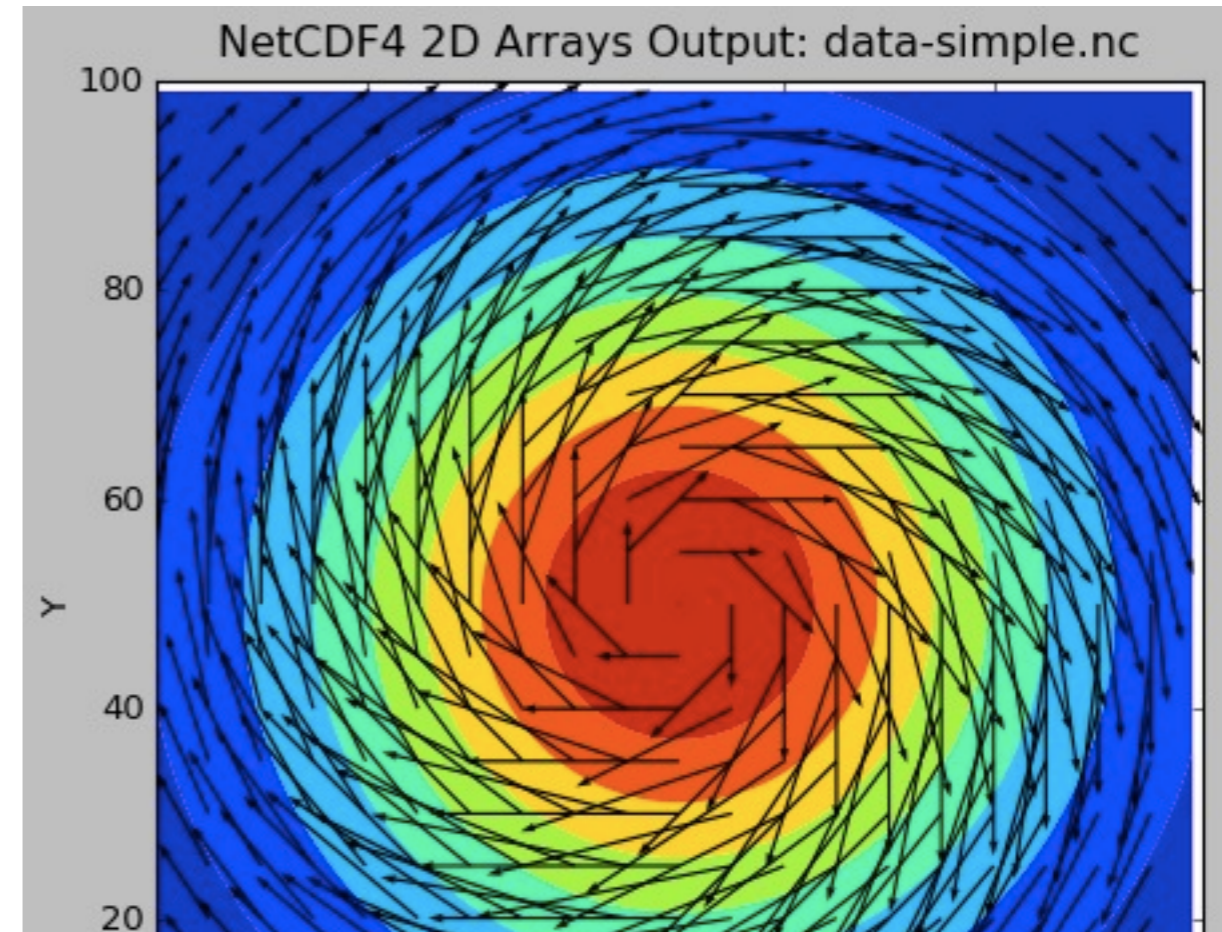
# What is this .nc file?

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



# NetCDF

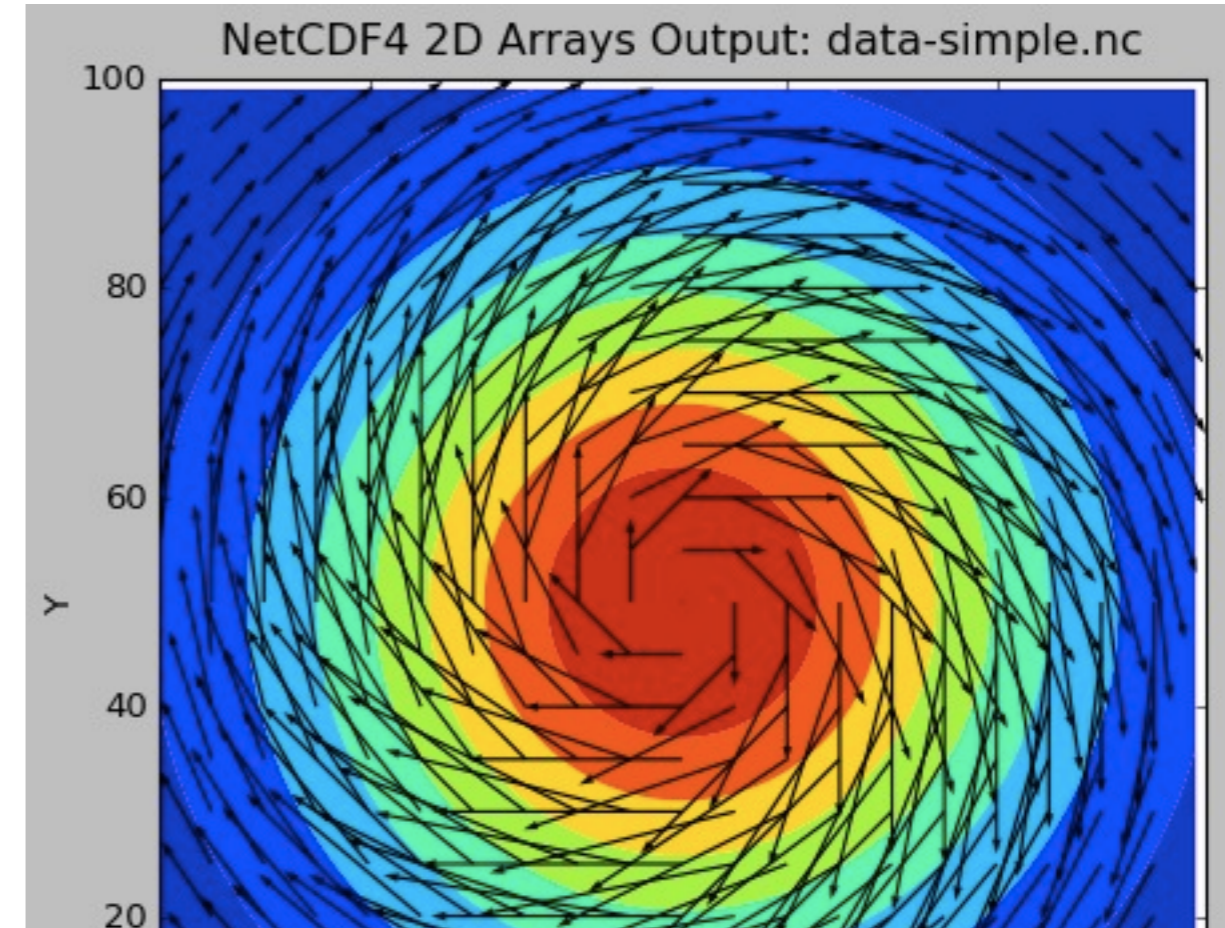
- NetCDF is a set of libraries and formats for:
  - portable,
  - efficient
  - “self-describing”
- way of storing and accessing large arrays (eg, for scientific data)
- Current version is NetCDF4



```
$ 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) ;
}
```

# NetCDF: *Portable*

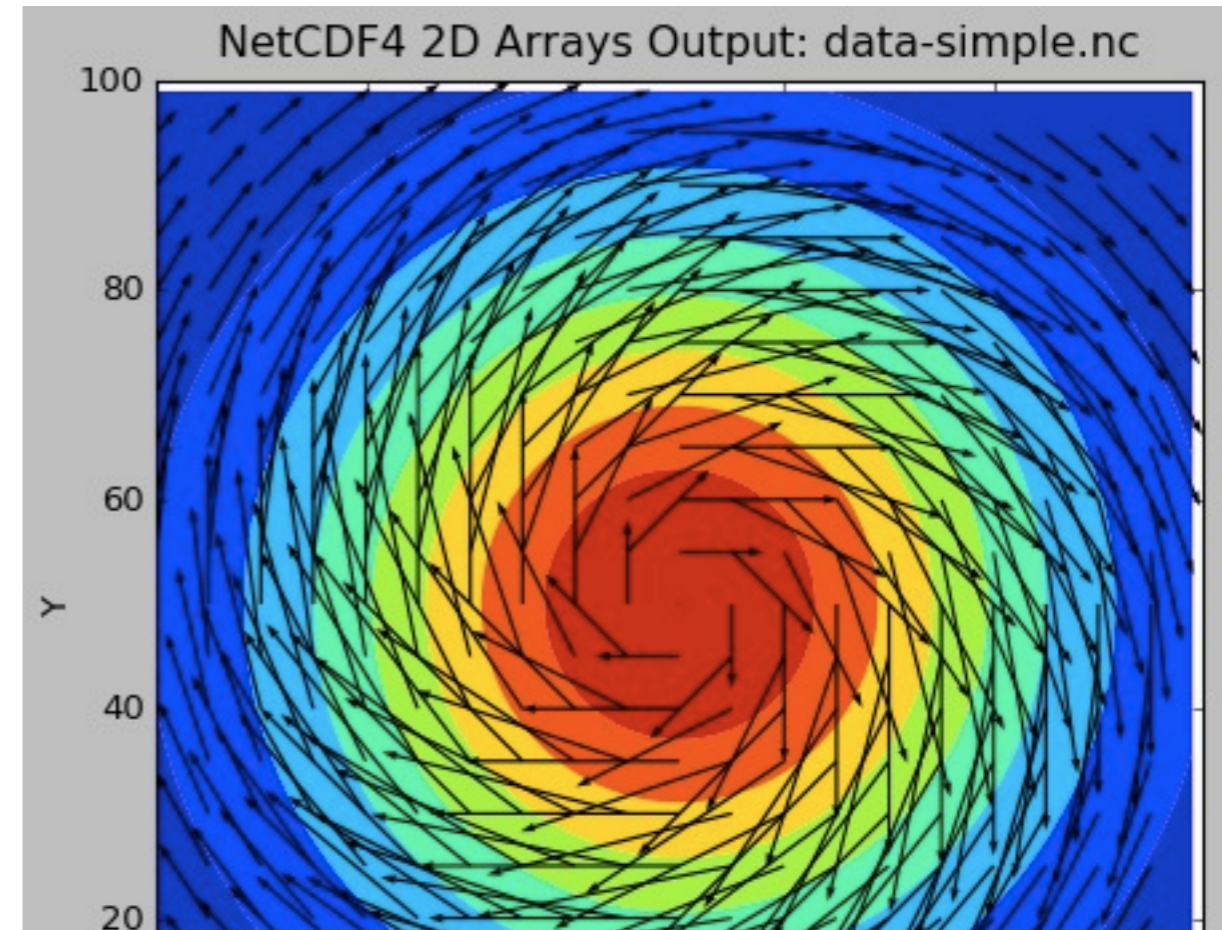
- Binary files, but common output format so that different sorts of machines can share files.
- Libraries accessible from C, C++, Fortran-77, Fortran 90/95/2003, python, etc.



```
$ 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*

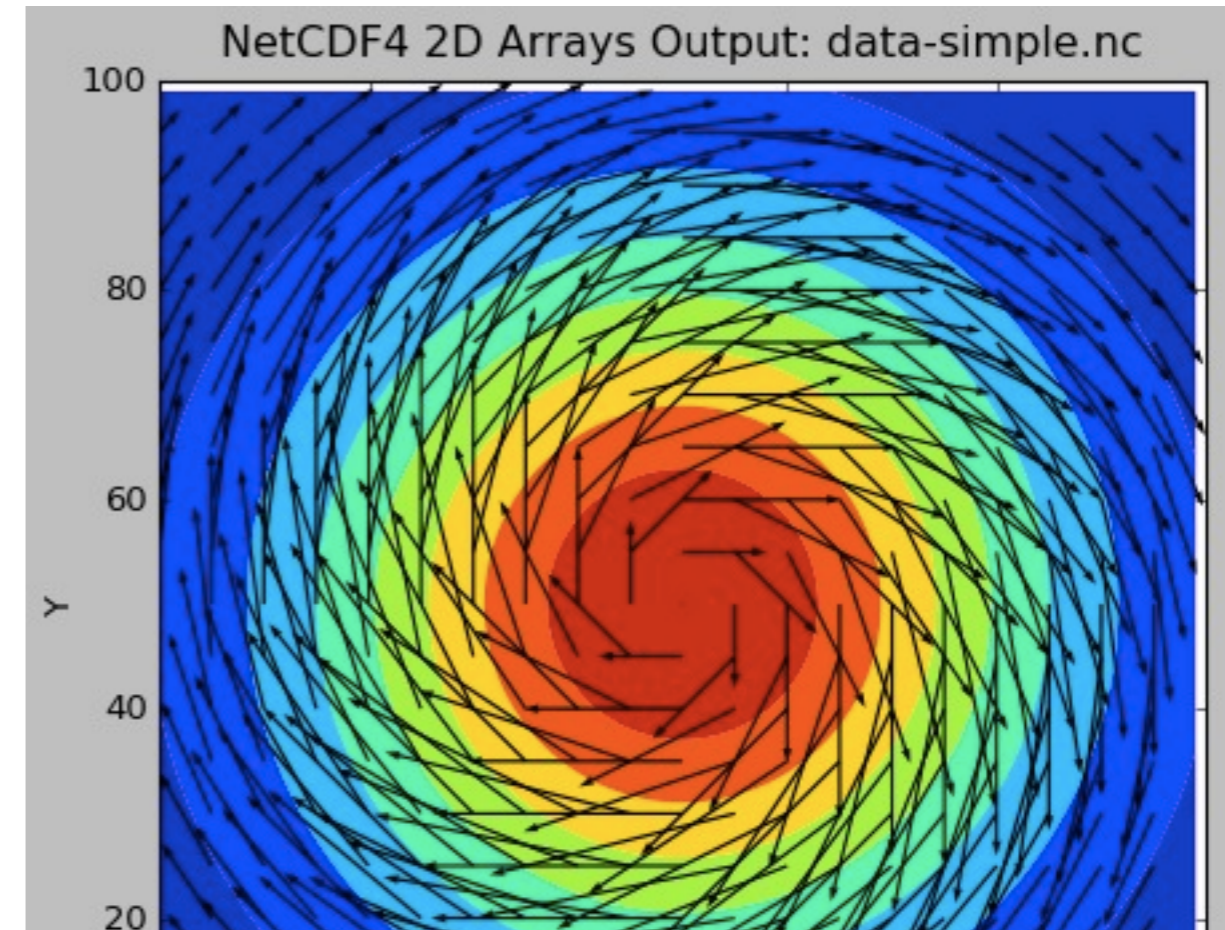
- Header contains the metadata to describe the big data
- Lists:
  - Array names
  - Dimensions
  - *shared* dimensions - information about how the arrays relate
  - Other, related information



```
$ 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 writenetcdf(file(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 writenetcdf(file(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 writenetcdf(file(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 writenetcdf(file(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 writenetcdf(file(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 writenetcdf(file, 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 writenetcdf(file, 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
```

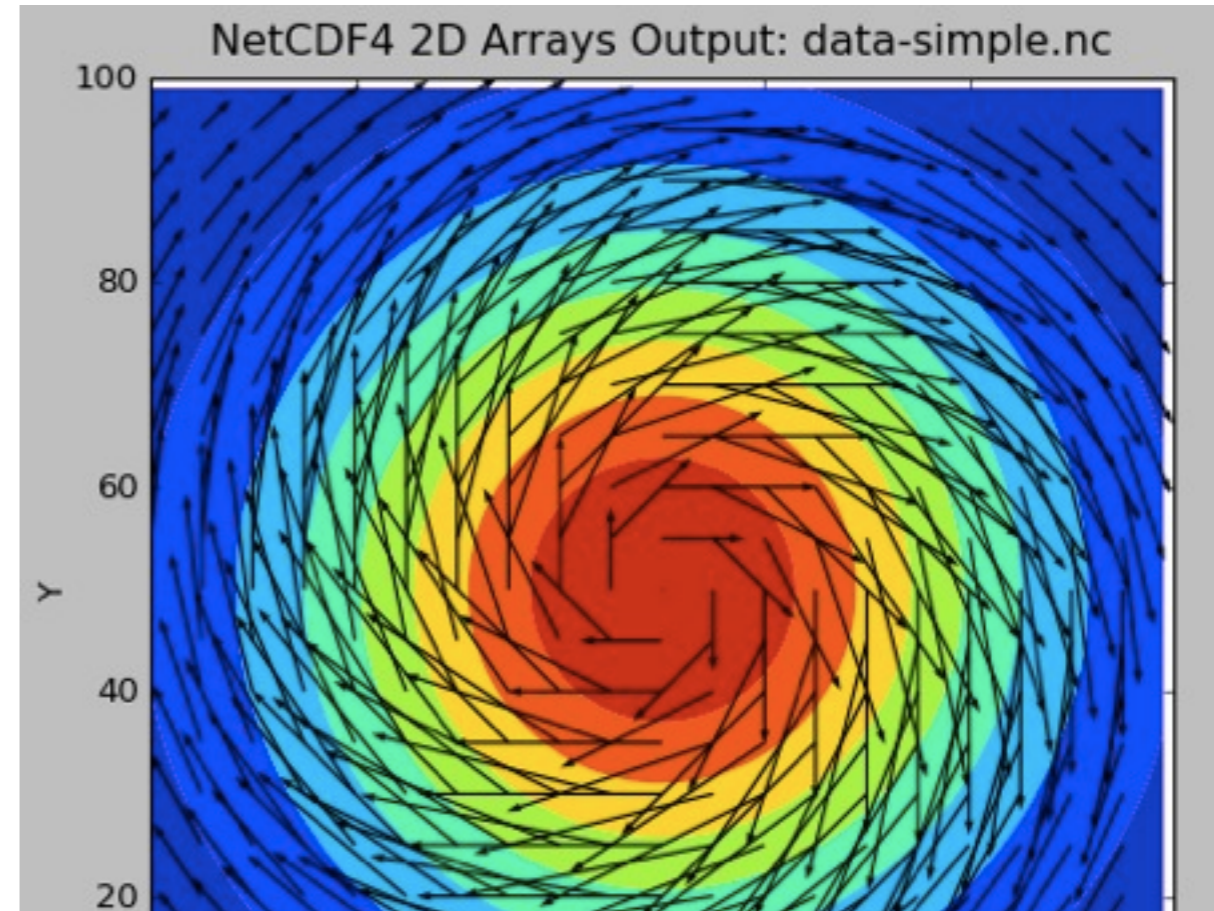
```
    print *, 'Could not open file ', rundata%filename
```

```
  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 /)
```

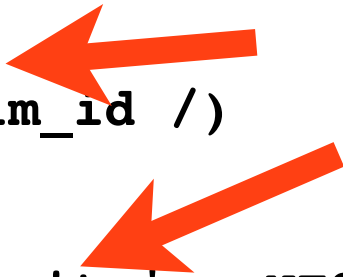
```
status = nf90_def_var(file_id, 'Density', NF90_DOUBLE, densdims, dens_id)
```

```
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
```

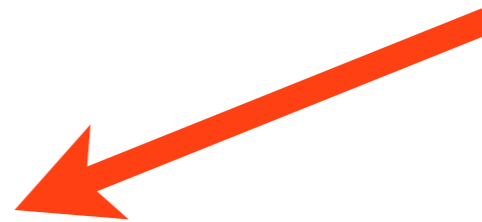
```
status = nf90_put_var(file_id, dens_id, dens)
```

```
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 '
```

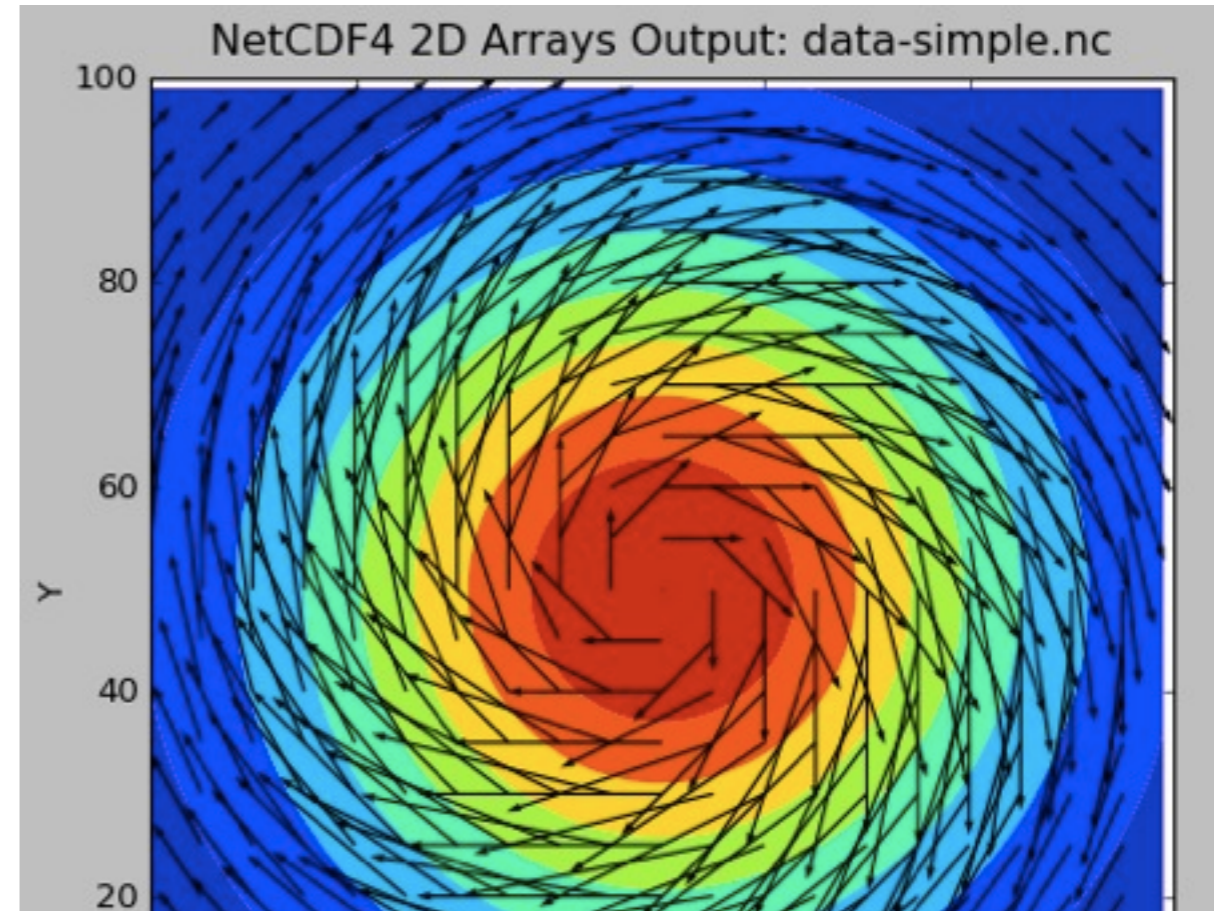
```
status = nf90_close(file_id)
```



**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

```
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)
```

# 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" ;
}
```

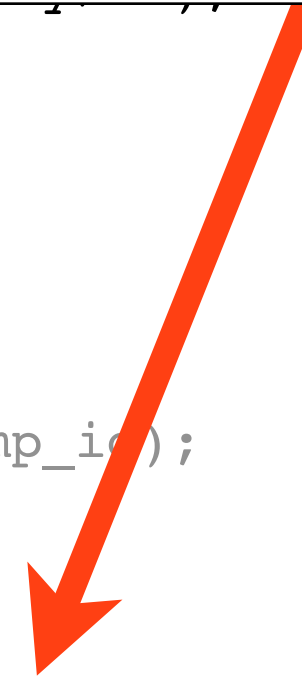
# 2darray.c

```
float *x, *y;
const char *coordunit="cm";
...
for (i=0; i<rundata.nx; i++) x[i] = (1.*i-rundata.nx/2);
for (i=0; i<rundata.ny; i++) y[i] = (1.*i-rundata.ny/2);
...
/* 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);

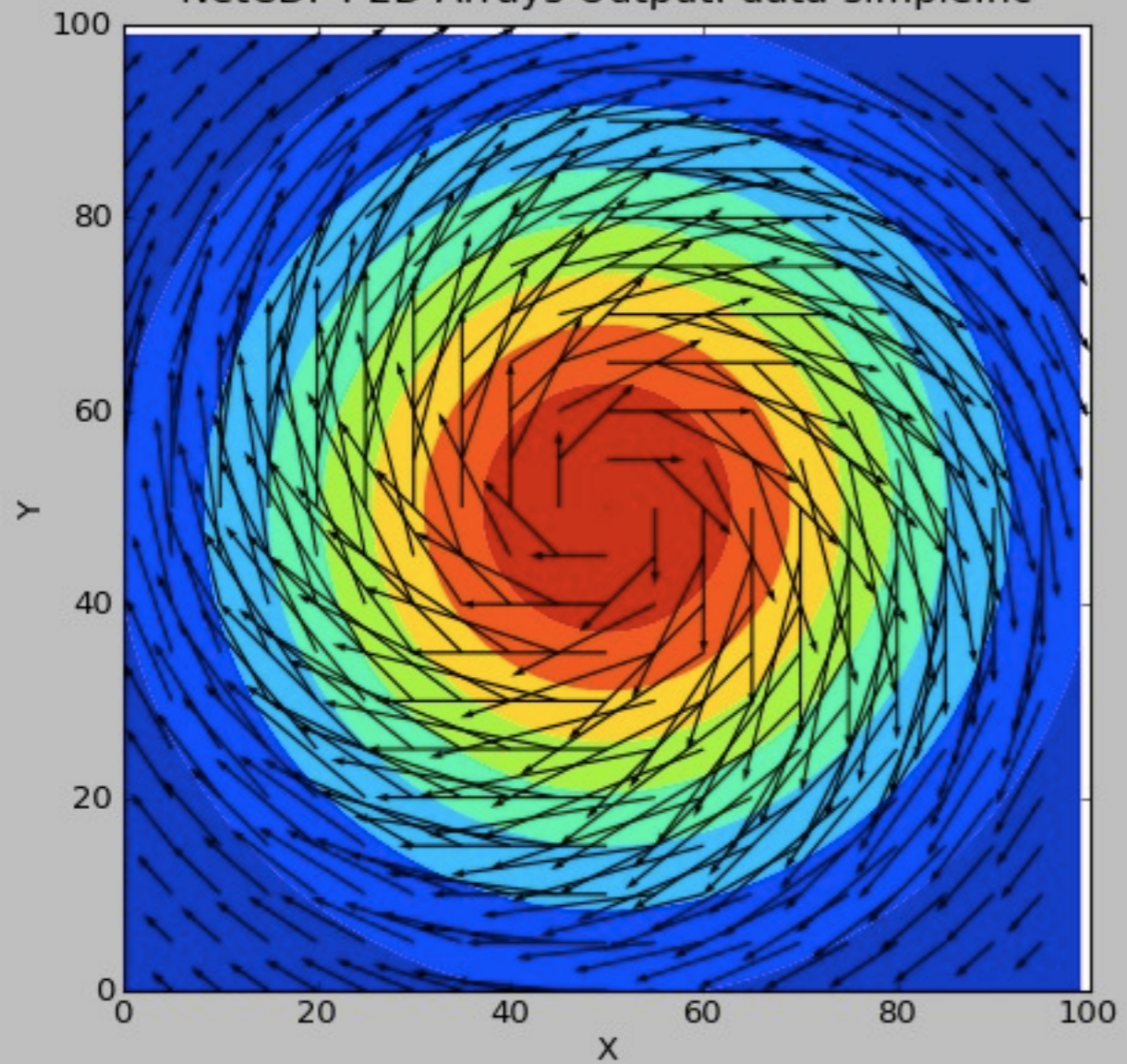
/* define the coordinate variables,... */
nc_def_var(file_id, "X coordinate", NC_FLOAT, 1, &xdim_id, &xcoord_id);
nc_def_var(file_id, "Y coordinate", NC_FLOAT, 1, &ydim_id, &ycoord_id);

/* ...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);
```

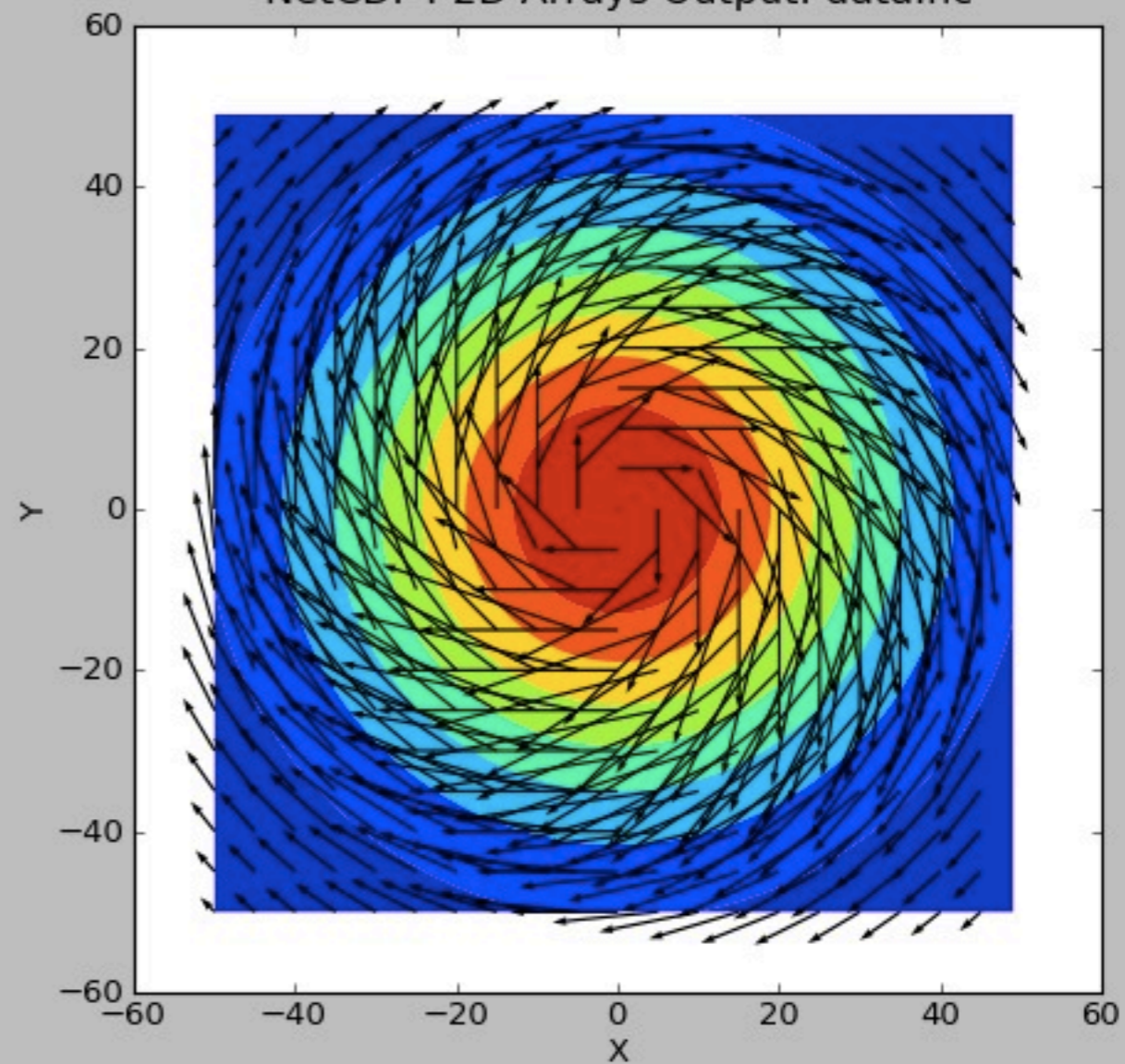
Typically not only  
define dimensions but  
give coordinate values



NetCDF4 2D Arrays Output: data-simple.nc



NetCDF4 2D Arrays Output: data.nc



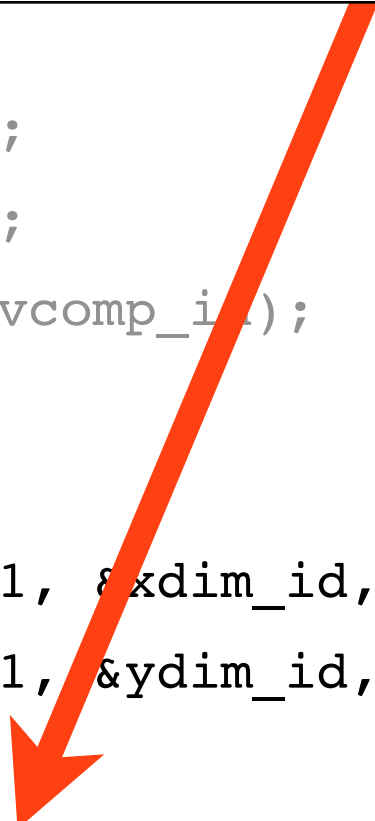
# 2darray.c

```
float *x, *y;
const char *coordunit="cm";
...
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.);
...
/* 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



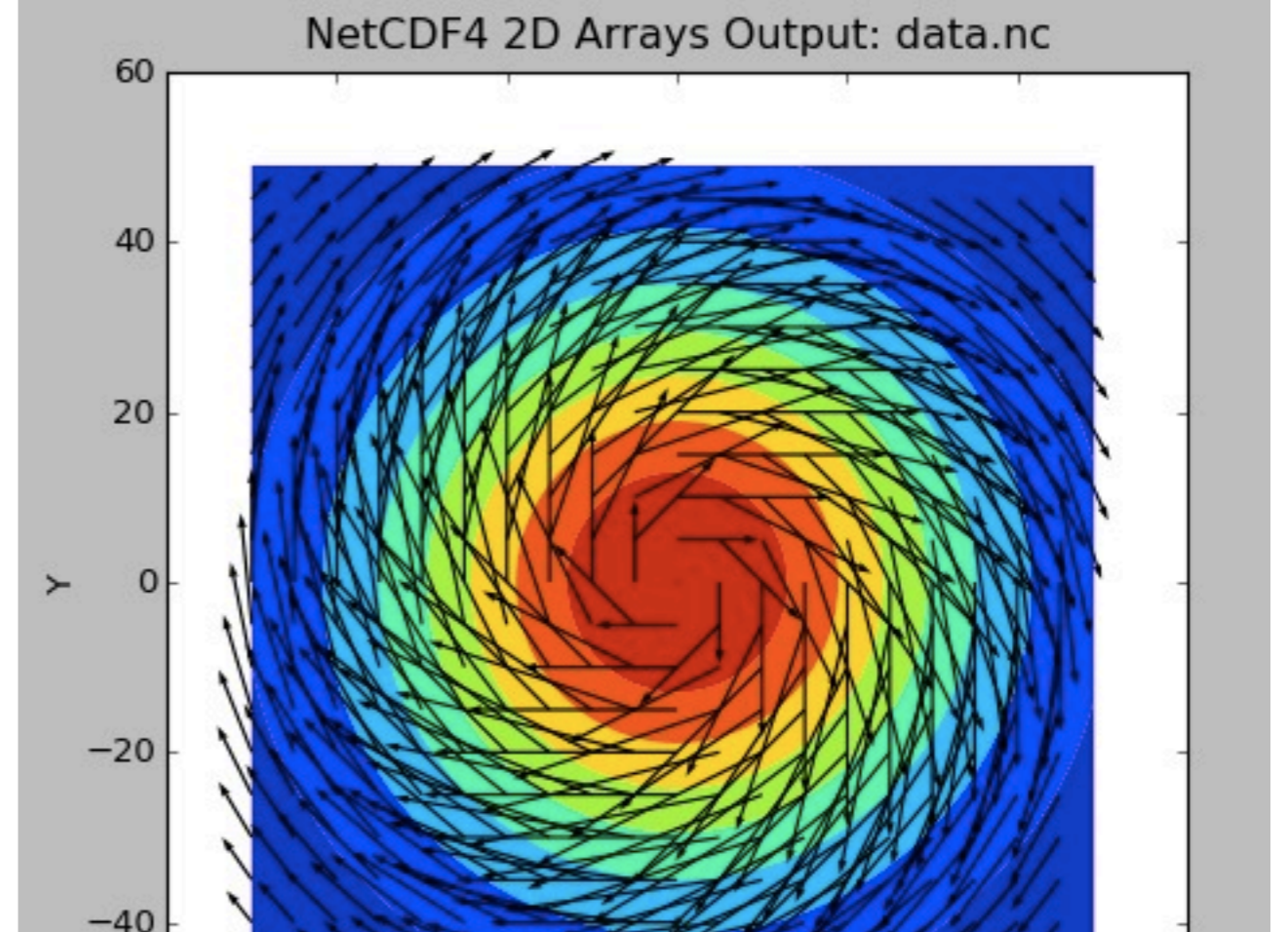
## NetCDF Conventions

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

- [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 processed data*)
- [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*)
- [RMFL ERIC Conventions](#)

# Big advantage of self- describing:

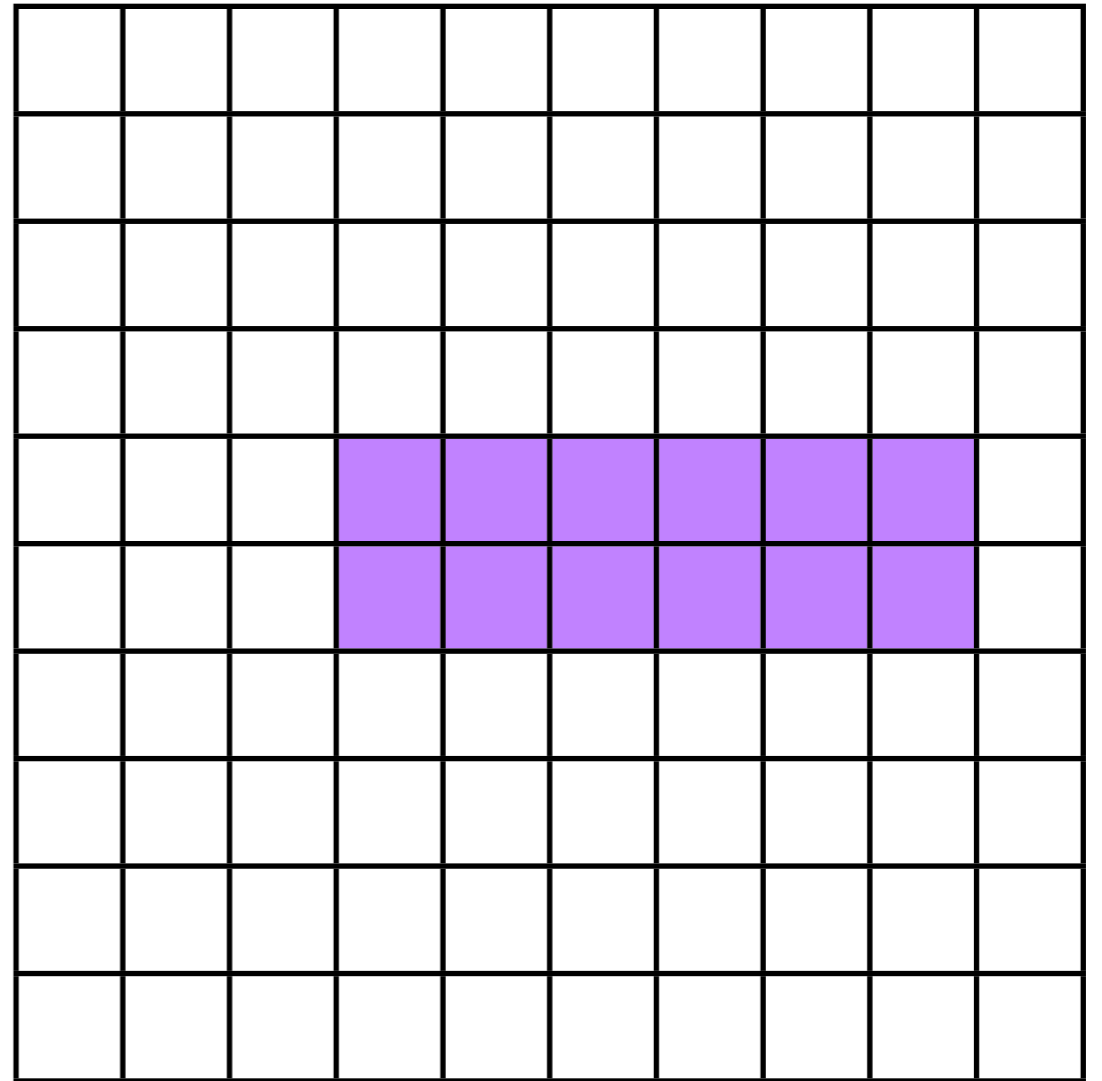
- Old program could easily read new file, even though data layout changed!
- Doesn't even need to know about attributes...
- New variables don't cause any problems - don't have to read them!
- Backwards compatibility
- But can look for them and use if available.



```
$ 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
```

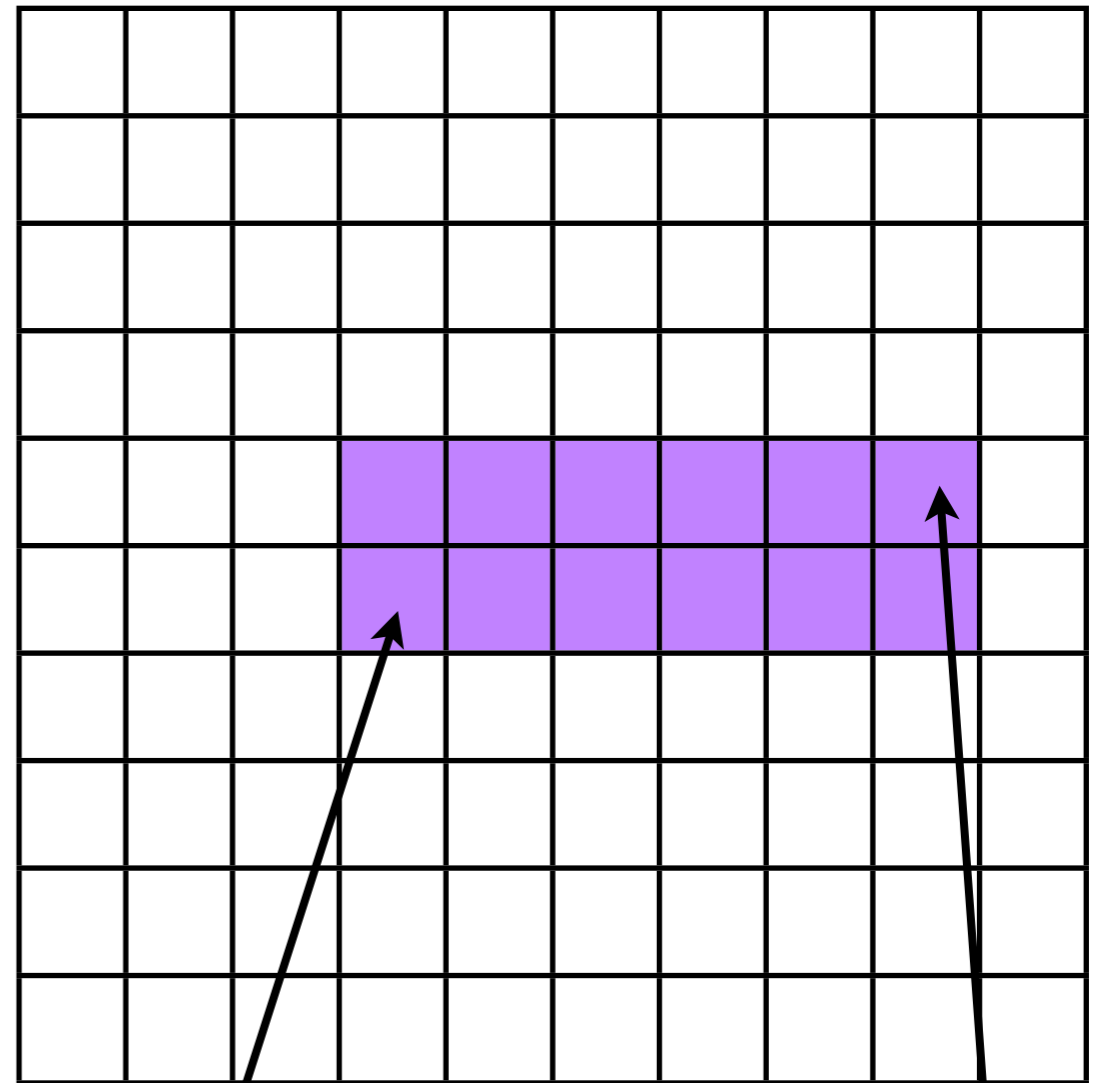
```
start(2) = 5
```

```
count(1) = 6
```

```
count(2) = 2
```

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

# dens(10,10)



dens(4,5)

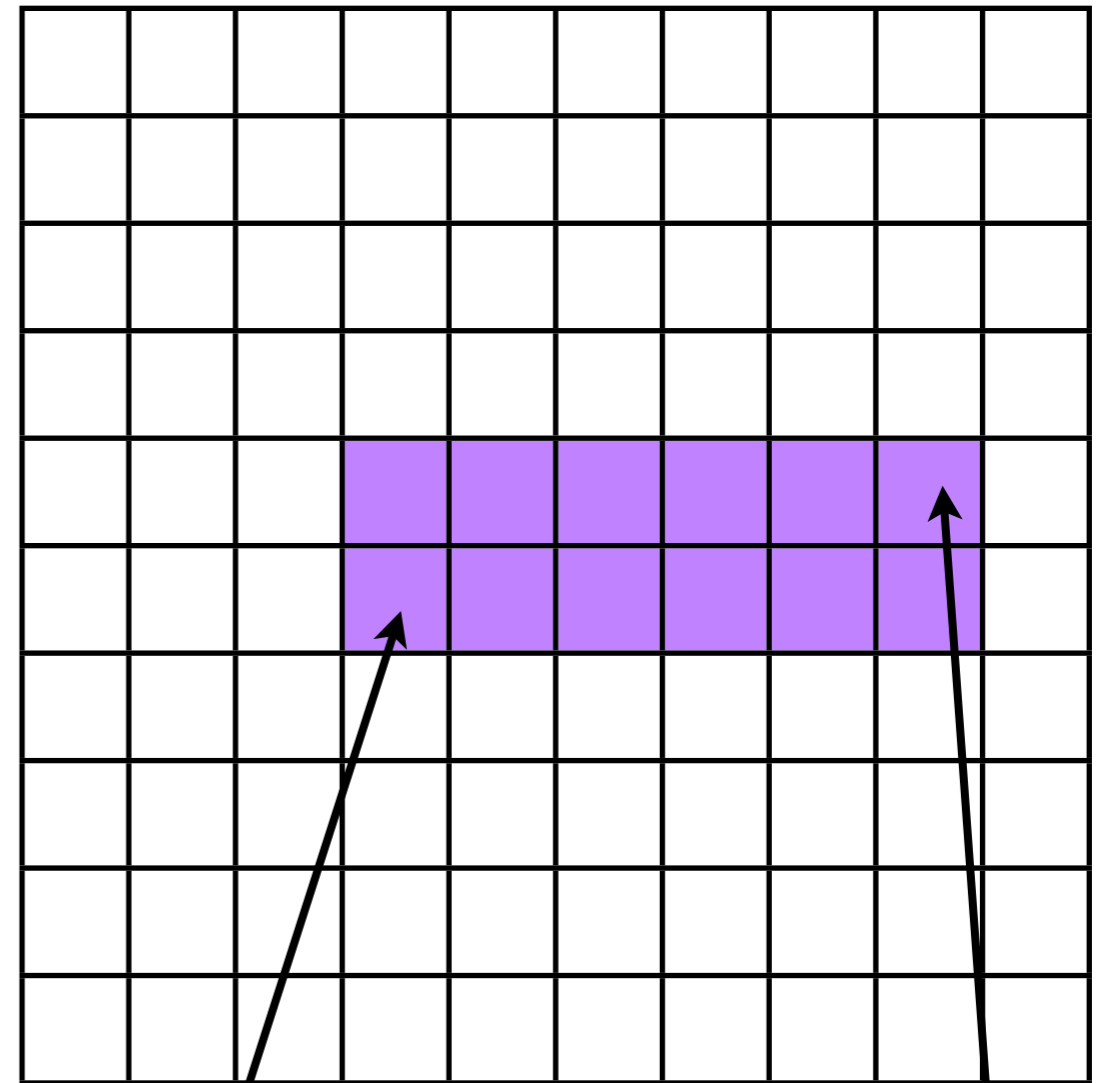
dens(9,6)

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

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

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

**dens[10][10]**

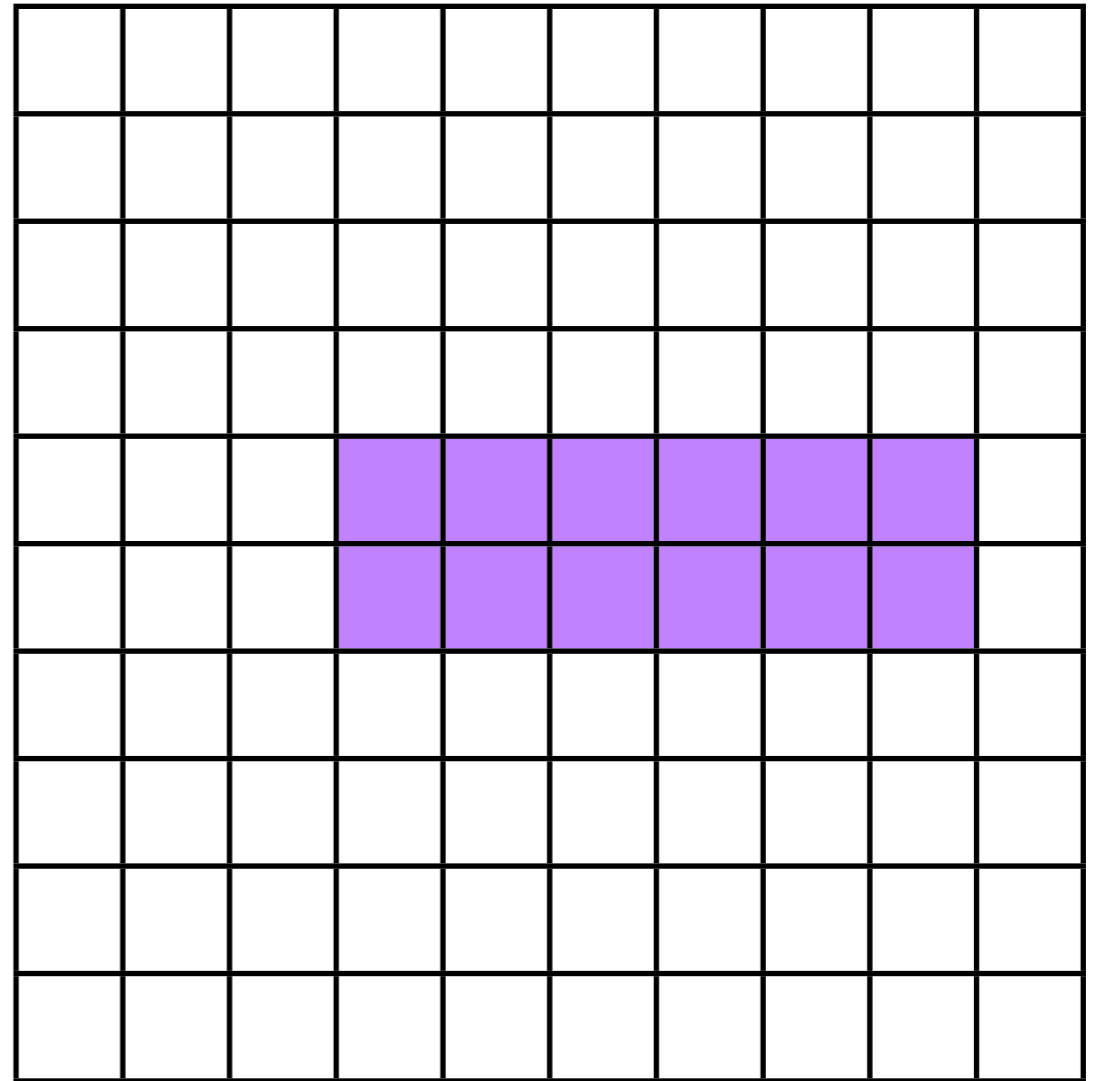


**dens[3][4]**

**dens[8][5]**

# Accessing subregions in file

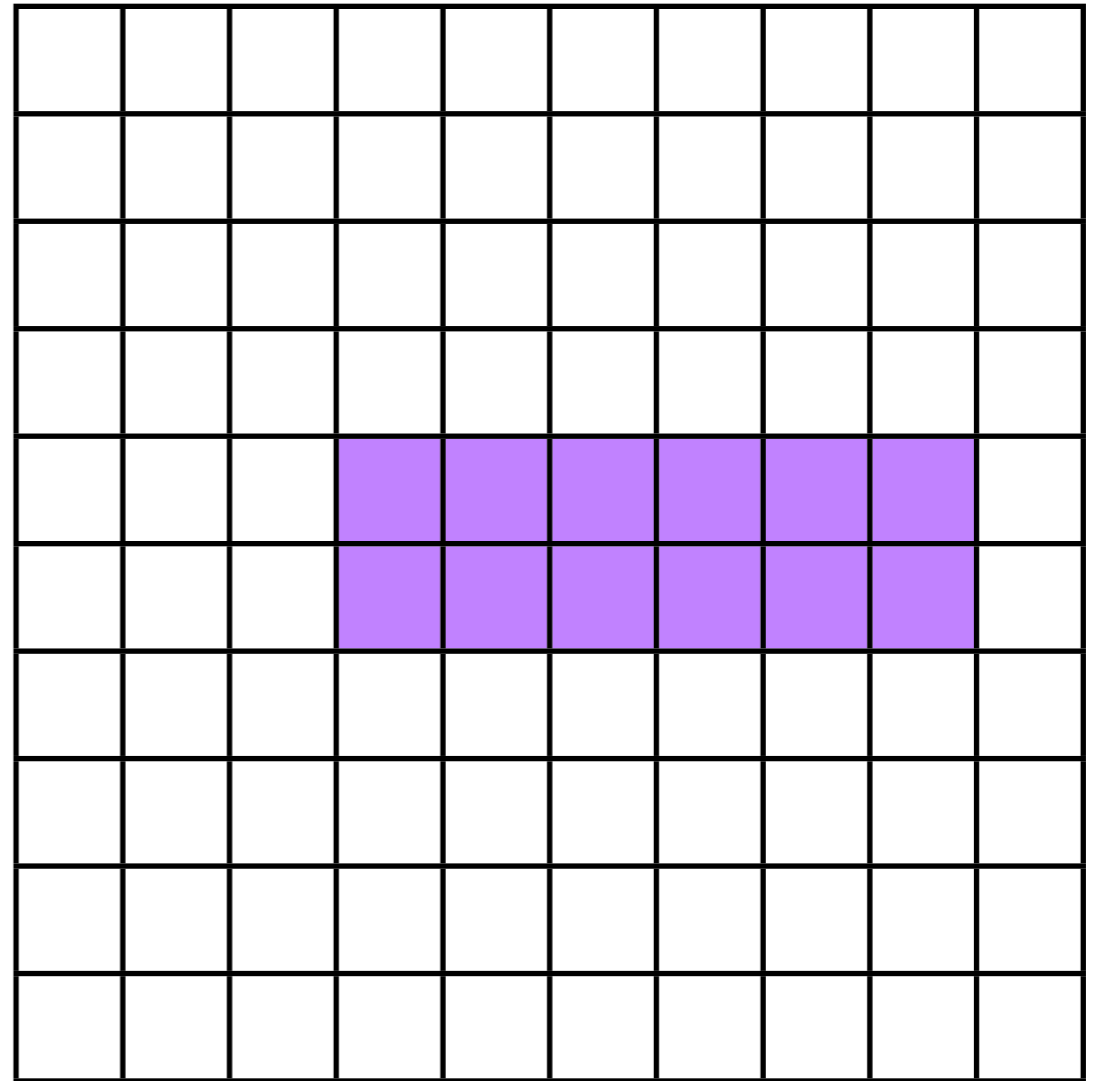
- Note that NetCDF libraries accepts starting conventions of C, Fortran as appropriate.





# 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.



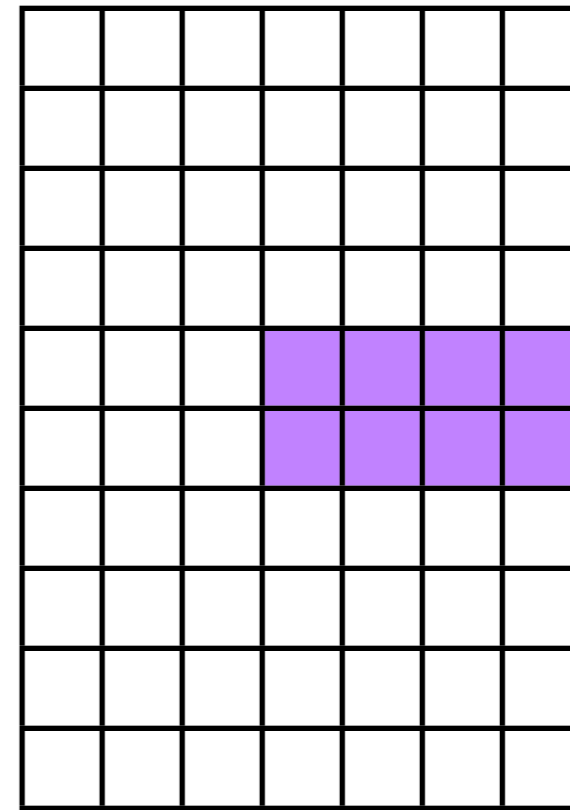
# Fortran vs C array conventions

```
$ 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

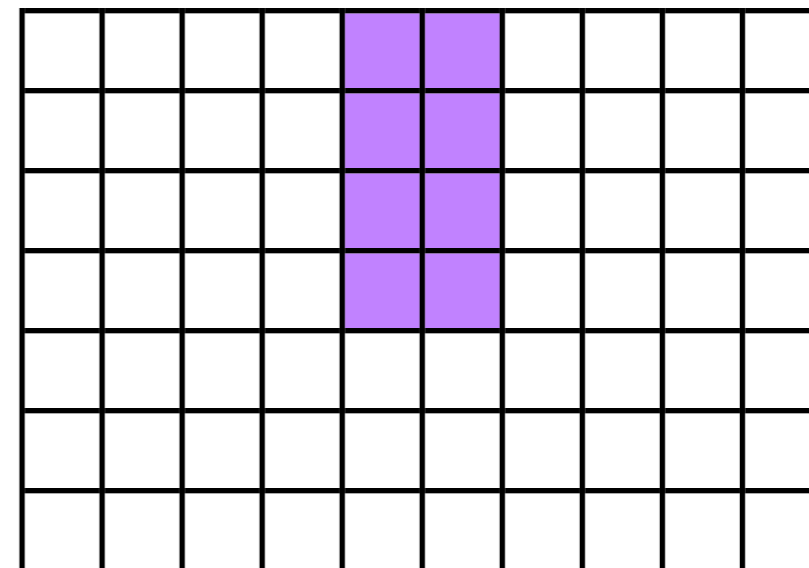
- Say in C you wanted to output in FORTRAN convention
- $(i,j)$  in your array corresponds to  $(j,i)$  in data space in file
- `nc_put_var` allows you to do this by mapping how indicies vary in memory compared to in file.



`dens[7][10]`

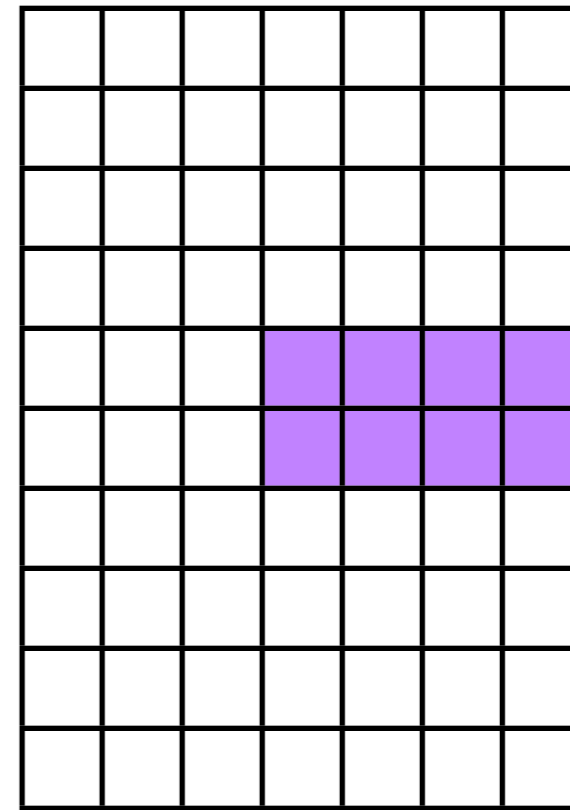
to

`dens(10,7)`



# Mapping memory space to file

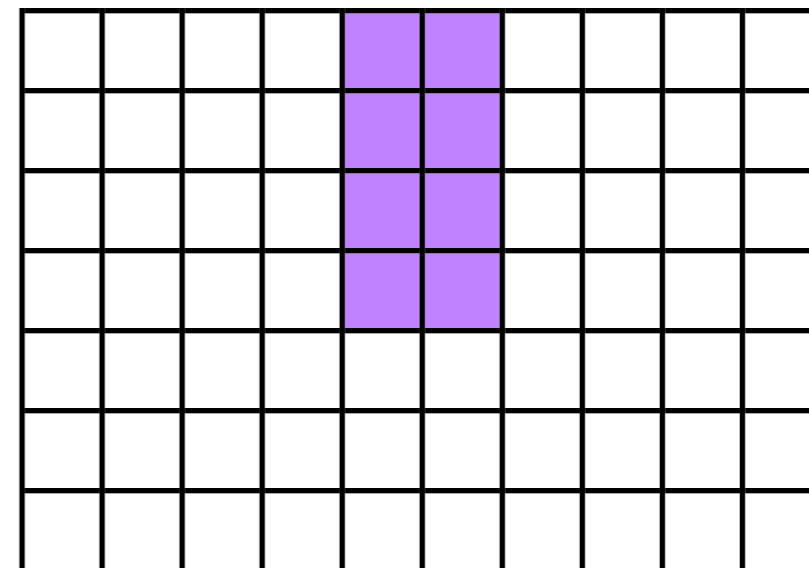
- Note - this requires understanding how memory is laid out in your data structures, as with MPI & MPI-IO
- This is *crucial* for I/O, and for HPC in general
- C has more flexibility (== potential problems) in this regard.



`dens[7][10]`

to

`dens(10,7)`



# 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 you by 4.
- You could write this as (4, 1)

Your picture of the array

[2][0]	[2][1]	[2][2]	[2][3]
[1][0]	[1][1]	[1][2]	[1][3]
[0][0]	[0][1]	[0][2]	[0][3]

In memory

[0][0]	[0][1]	[0][2]	[0][3]	[1][0]	[1][1]	[1][2]	[1][3]	[2][0]
--------	--------	--------	--------	--------	--------	--------	--------	--------

# 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 **lst** index should jump you by 1, and the second by...

Your picture of the array

[2][0]	[2][1]	[2][2]	[2][3]
[1][0]	[1][1]	[1][2]	[1][3]
[0][0]	[0][1]	[0][2]	[0][3]

In memory

[0][0]	[0][1]	[0][2]	[0][3]	[1][0]	[1][1]	[1][2]	[1][3]	[2][0]
--------	--------	--------	--------	--------	--------	--------	--------	--------

# 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

[2][0]	[2][1]	[2][2]	[2][3]
[1][0]	[1][1]	[1][2]	[1][3]
[0][0]	[0][1]	[0][2]	[0][3]

In memory

[0][0]	[0][1]	[0][2]	[0][3]	[1][0]	[1][1]	[1][2]	[1][3]	[2][0]
--------	--------	--------	--------	--------	--------	--------	--------	--------

# 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

[2][0]	[2][1]	[2][2]	[2][3]
[1][0]	[1][1]	[1][2]	[1][3]
[0][0]	[0][1]	[0][2]	[0][3]

In memory

[0][0]	[0][1]	[0][2]	[0][3]	[1][0]	[1][1]	[1][2]	[1][3]	[2][0]
--------	--------	--------	--------	--------	--------	--------	--------	--------



# More on NetCDF

- <http://www.unidata.ucar.edu/software/netcdf/>
- Docs, mailing lists, tutorials, sample code, API, etc.



# Sample Code



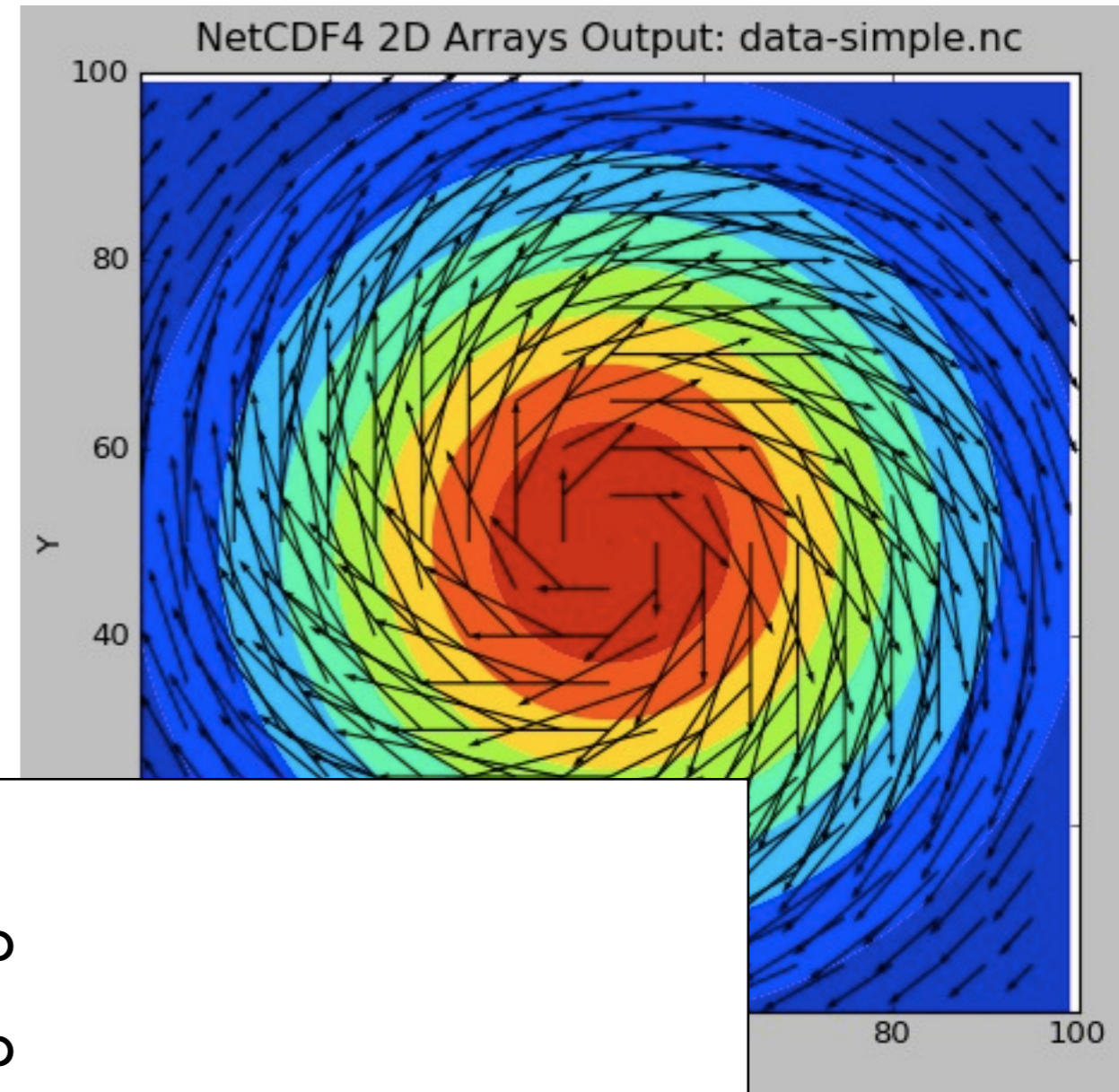
```
$ cd
$ cd parIO/hdf5

$ source ../seriallibs
$ 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
```

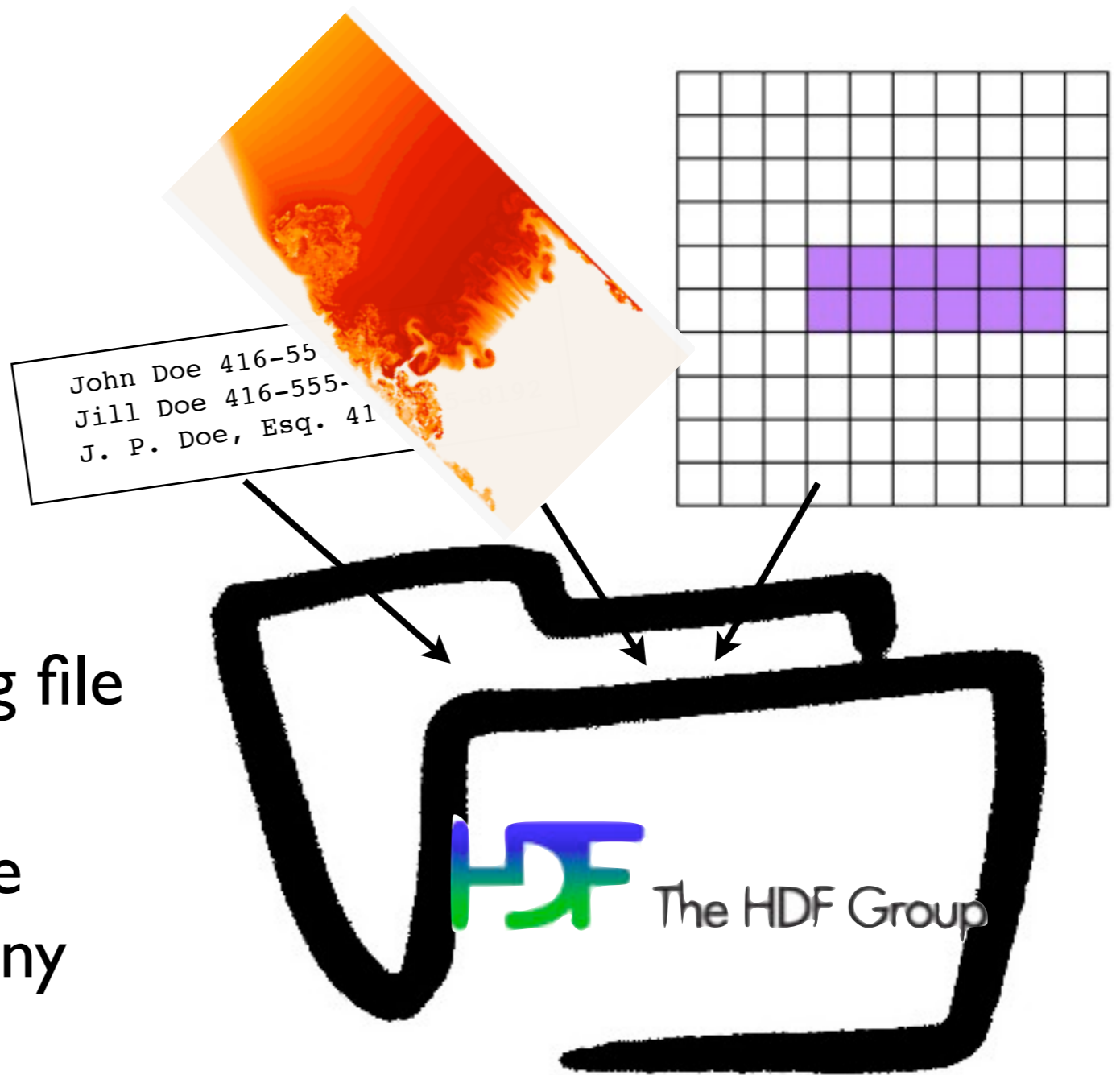
```
$ h5ls data-fort.h5/ArrayData
```

```
dens              Dataset {100, 100}
```

```
vel              Dataset {100, 100, 2}
```

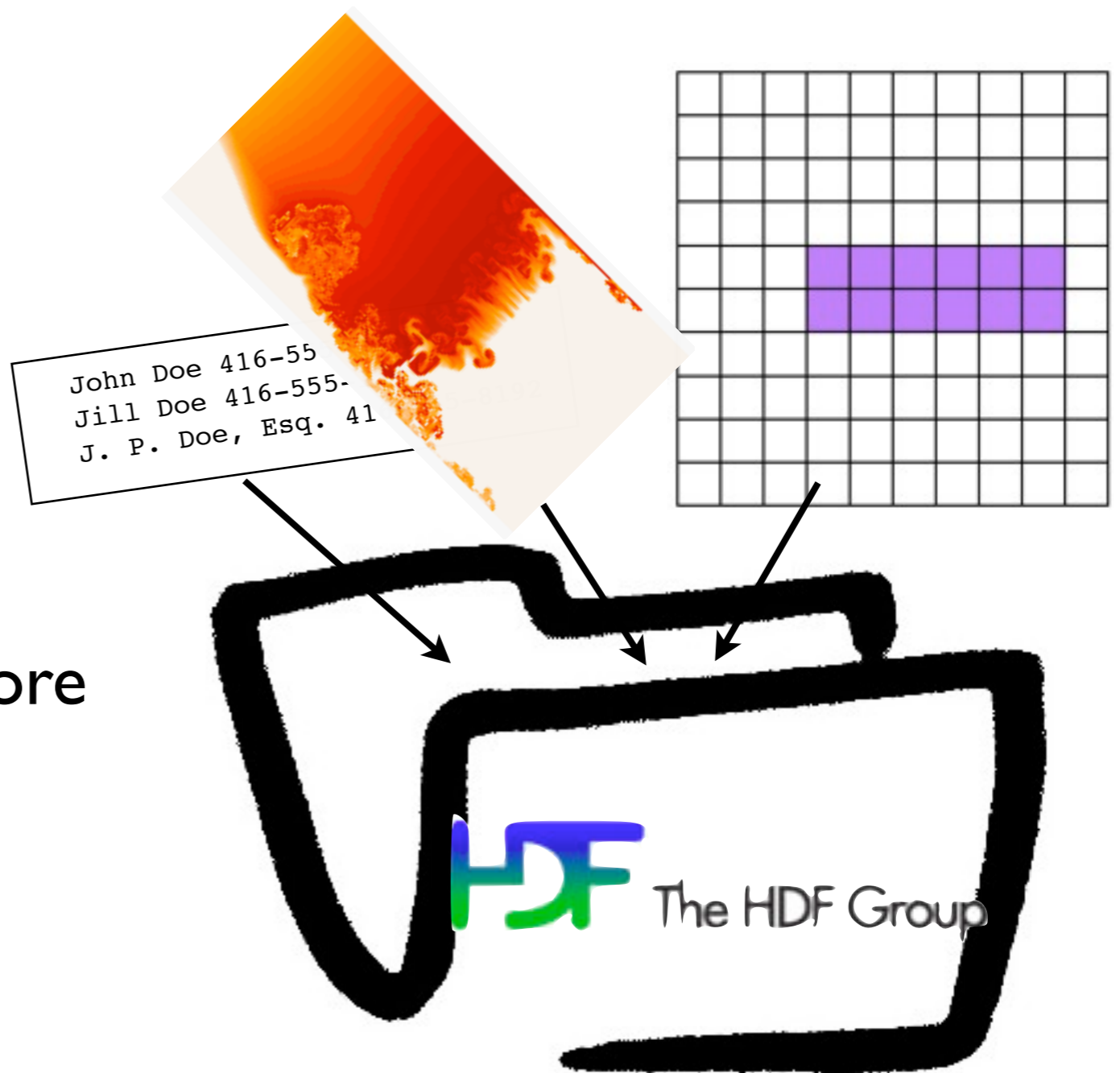
# HDF5

- 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;

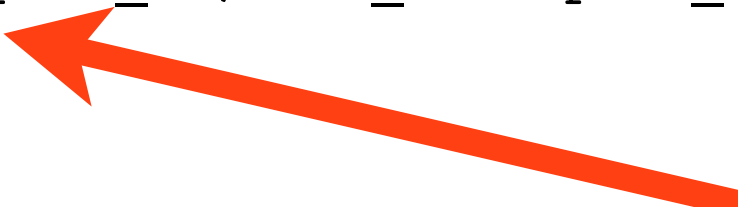
/* sizes */
hsize_t densdims[2], veldims[3];

/* 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];

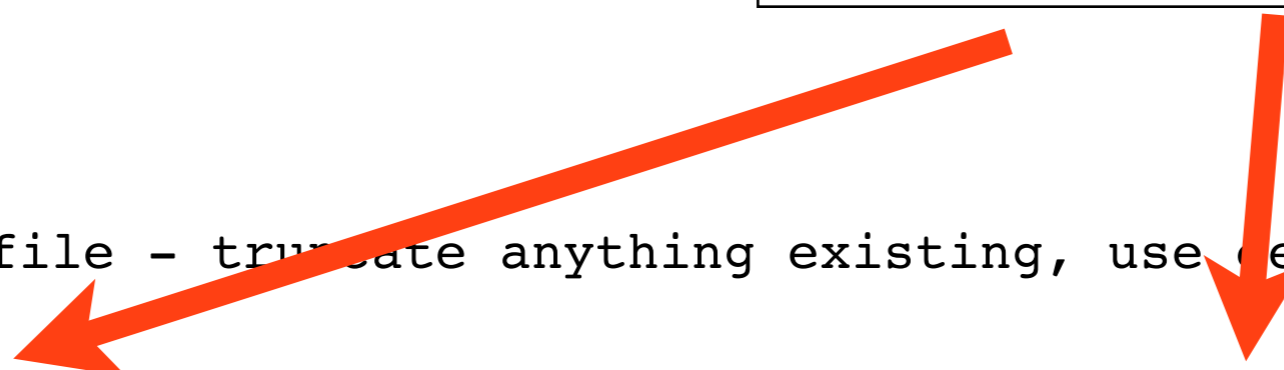
/* status */
herr_t status;

/* 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	<b>A</b> ttributes
H5D	<b>D</b> atasets
H5E	<b>E</b> rrors
H5F	<b>F</b> iles
H5G	<b>G</b> roups
H5P	<b>P</b> roperties
H5S	Data <b>S</b> paces
H5T	Data <b>T</b> 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, "dens_data",
                           H5T_IEEE_F64LE, dens_dataspace_id,
                           H5P_DEFAULT);

vel_dataset_id  = H5Dcreate(file_id, "vel_data",
                           H5T_NATIVE_DOUBLE, vel_dataspace_id,
                           H5P_DEFAULT);
```

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);
```

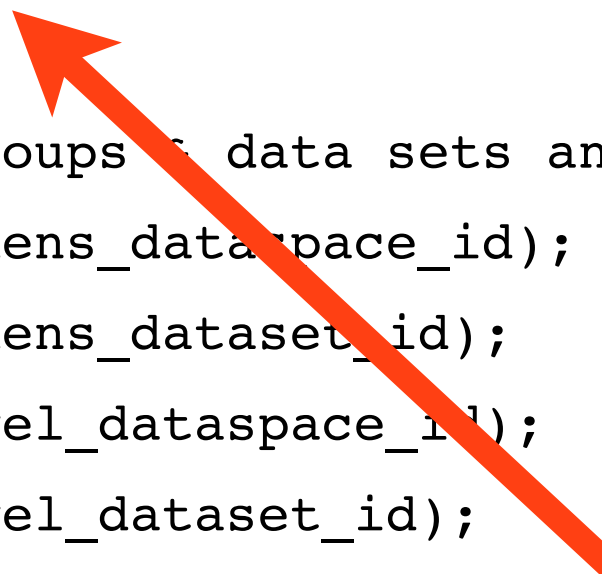
**Creating a data set like  
defining a variable in  
NetCDF.  
Also declare the type  
you want it to be on  
disk.**

# 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 of 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);
```



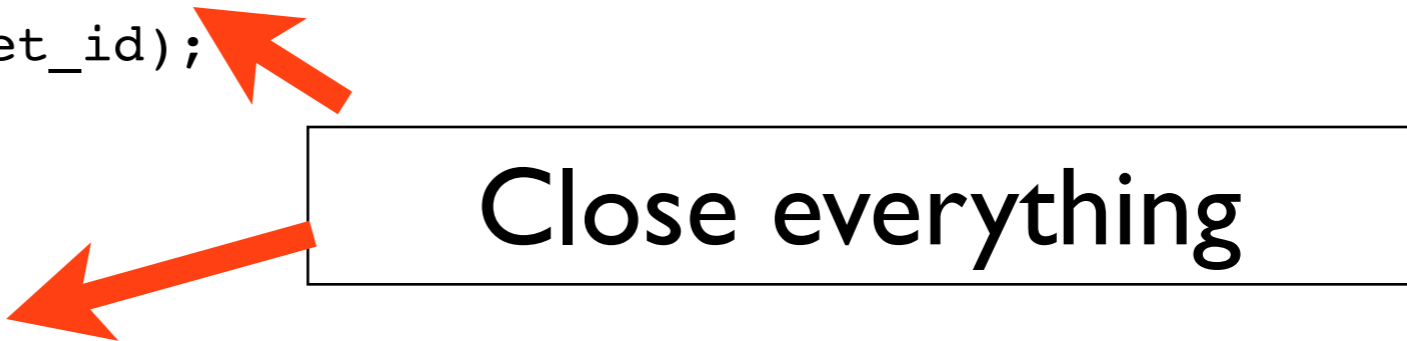
**Write memory from all of memory to all of the dataset on the file. Values in mem are in the native double precision format.**

# 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);
```



# 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()

! 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

endif
```

See what I mean about  
\_F?

# f2darray-simple.f90

```
! create the dataspace 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)

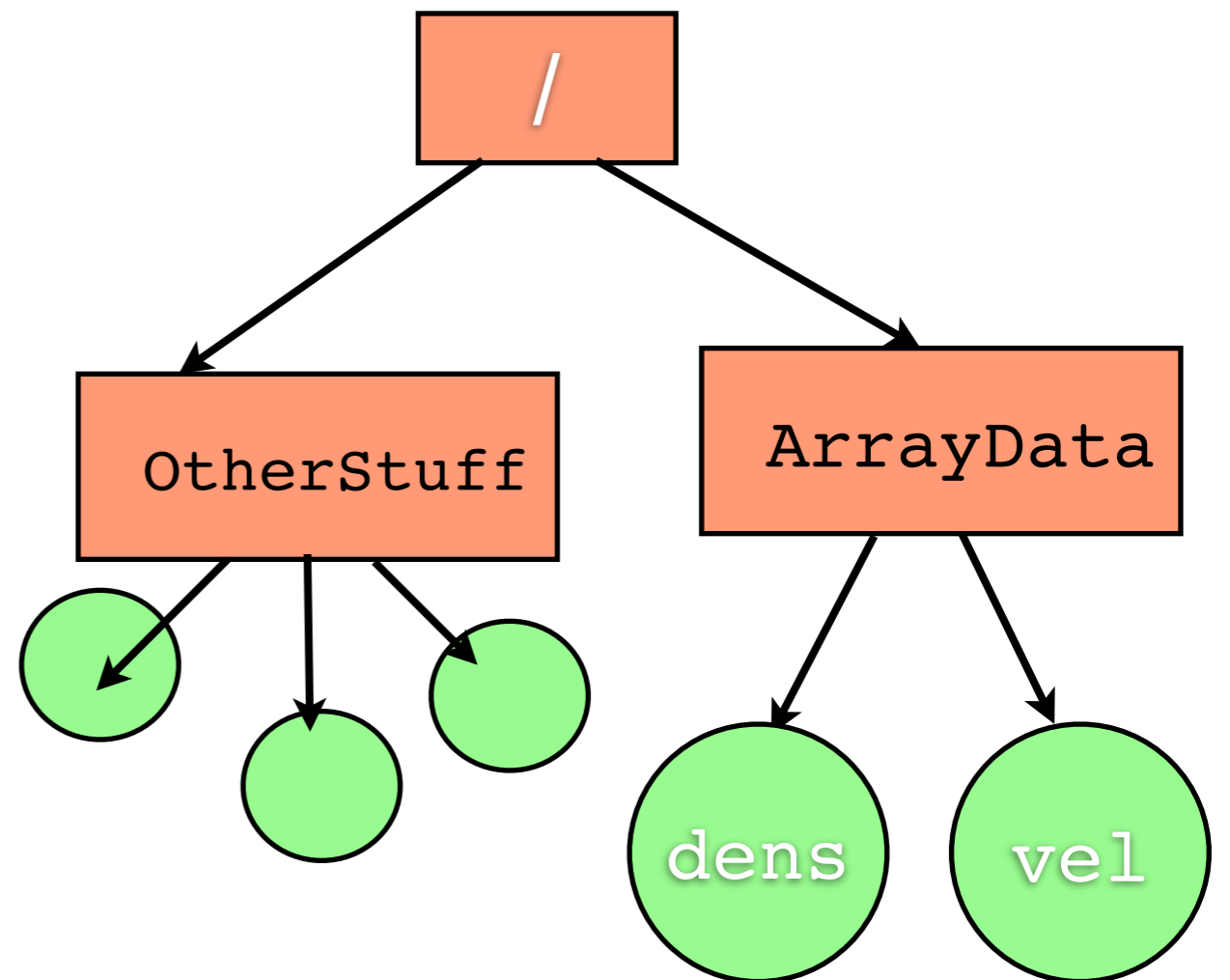
! now that the dataspace are defined, we can define variables on them

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);

...

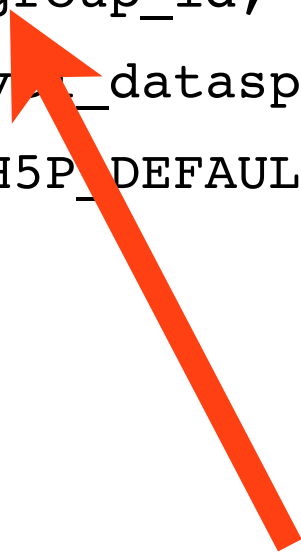
dens_dataset_id = H5Dcreate(file_id, "/ArrayData/dens", H5T_IEEE_F64LE,
dens_dataspace_id, H5P_DEFAULT,
H5P_DEFAULT, H5P_DEFAULT);
vel_dataset_id = H5Dcreate(file_id, "/ArrayData/vel", H5T_IEEE_F64LE,
vel_dataspace_id, H5P_DEFAULT,
H5P_DEFAULT, H5P_DEFAULT);
```



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	finer
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	

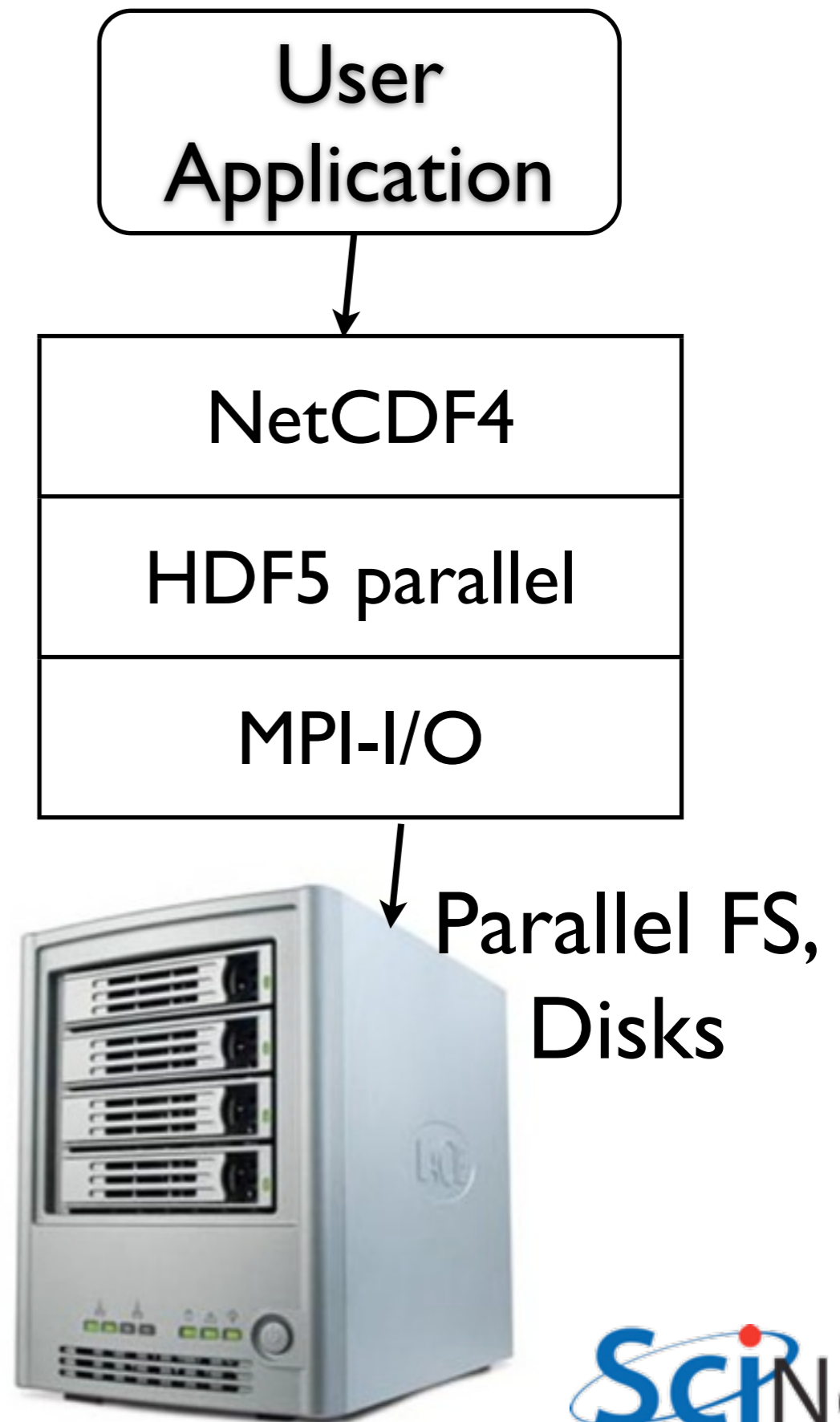
...

# Parallel I/O using NetCDF4, HDF5



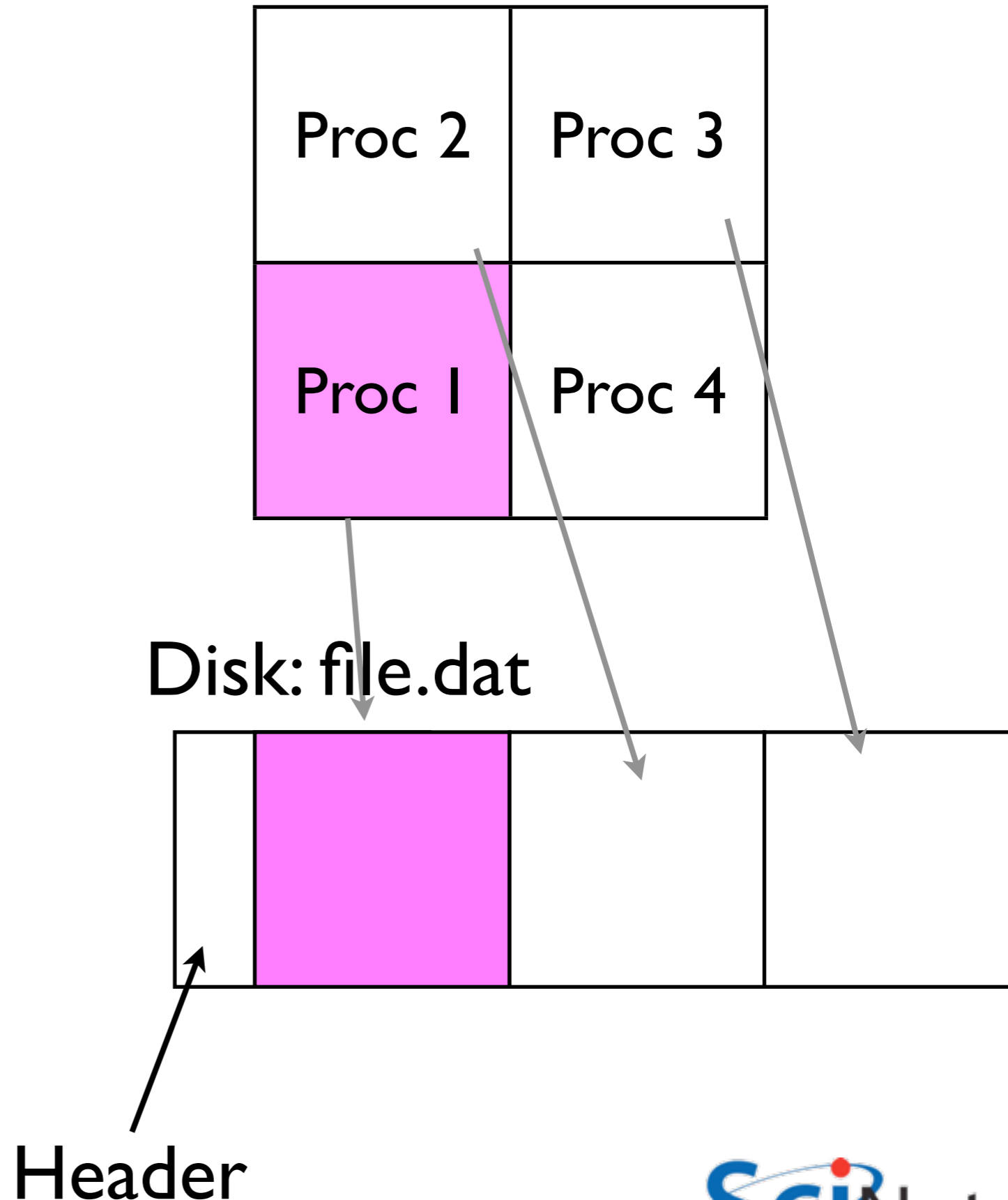
# Parallel I/O libraries

- 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



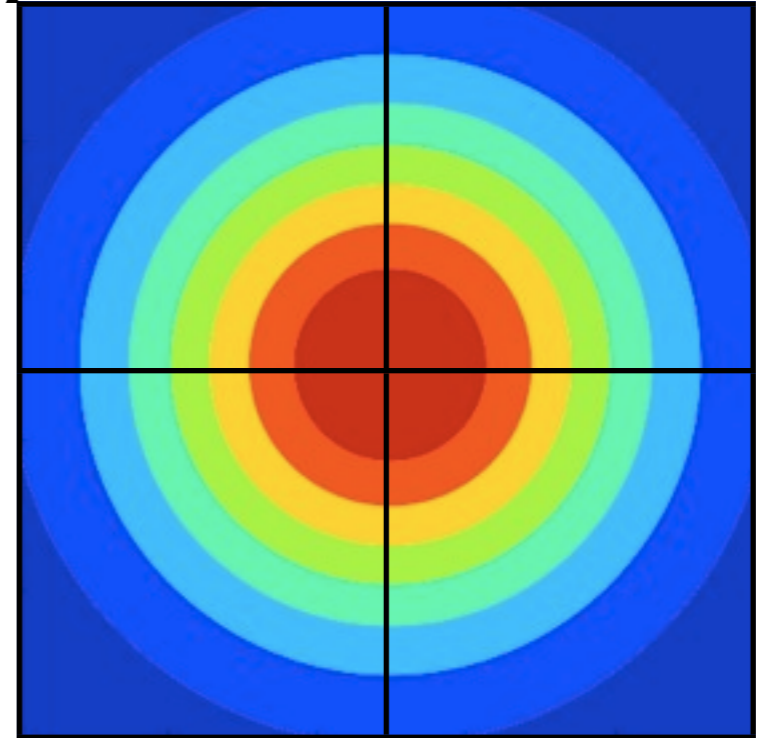
# Parallel IO to One file

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



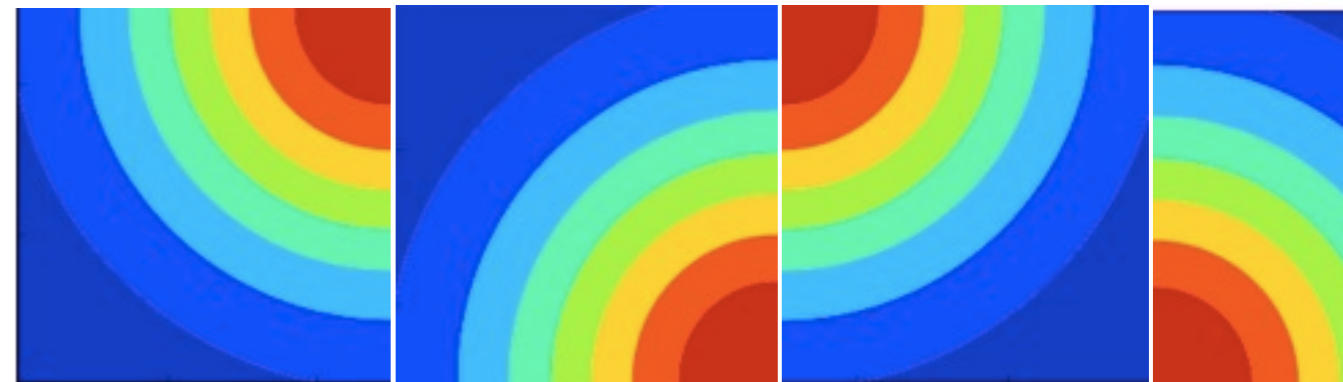
# How do you decide where to write?

Memory:



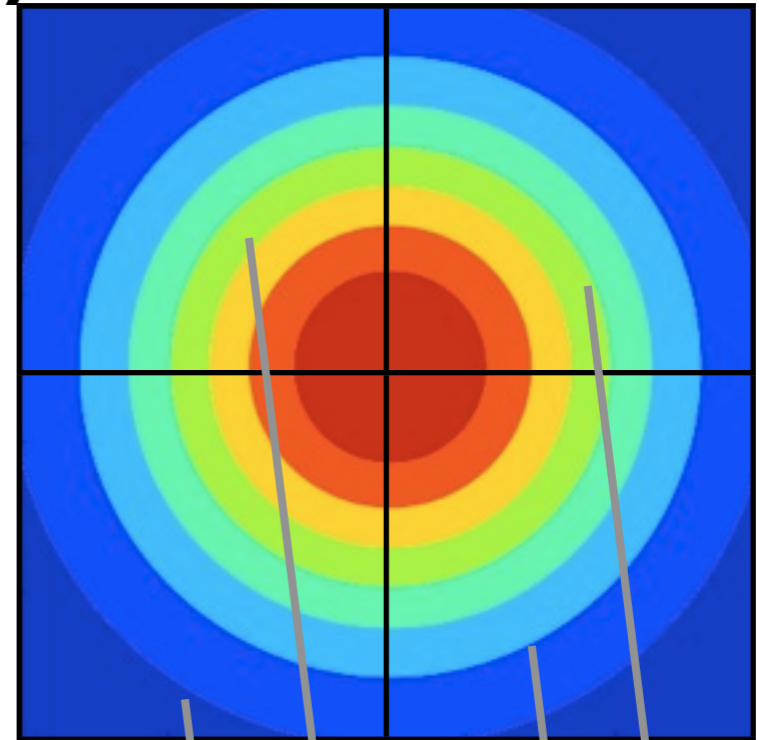
- One possibility: each processor writes out its part of problem, in order.
- Pros - can be super fast.
- Cons - Output depends on number of processors run on. Analysis routines, restarts...

Disk:



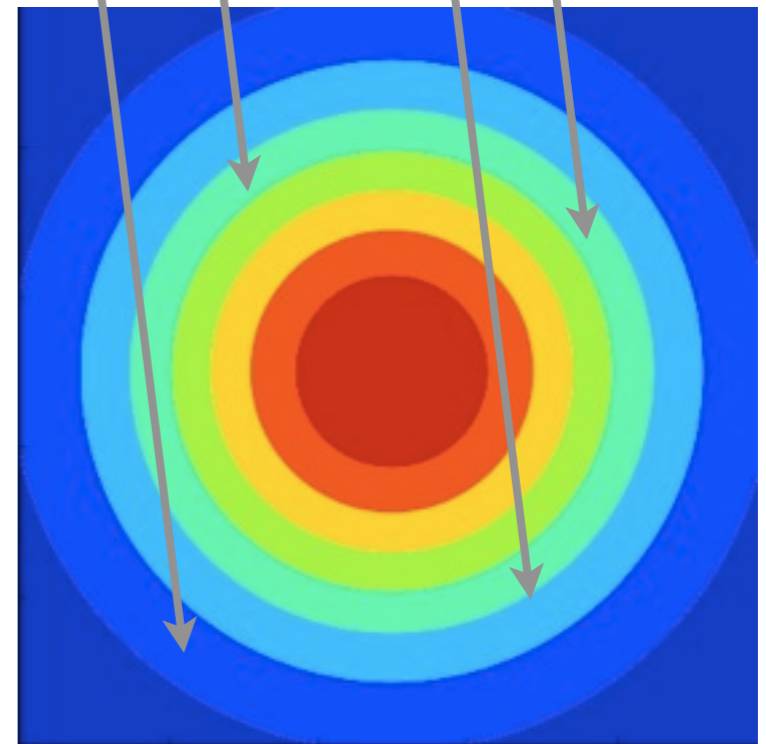
# How do you decide where to write?

Memory:



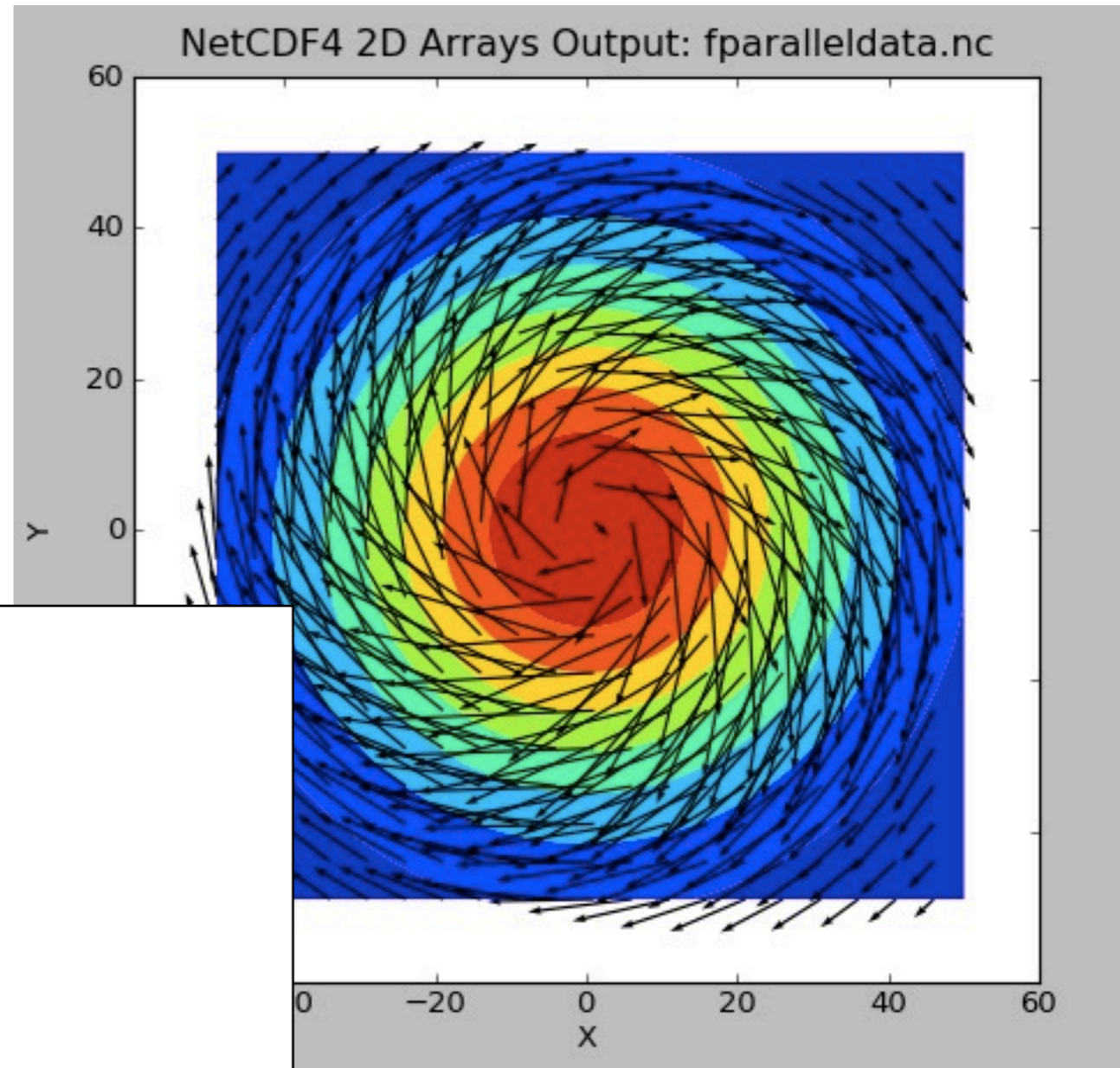
- Other possibility: Write out chunks as they would be in memory on serial machine
- Pros: File looks the same no matter how many processes were used to write.
- Cons: Noncontig access; *may* be slower, but MPI-IO collective + good parallel FS should make competitive.

Disk:



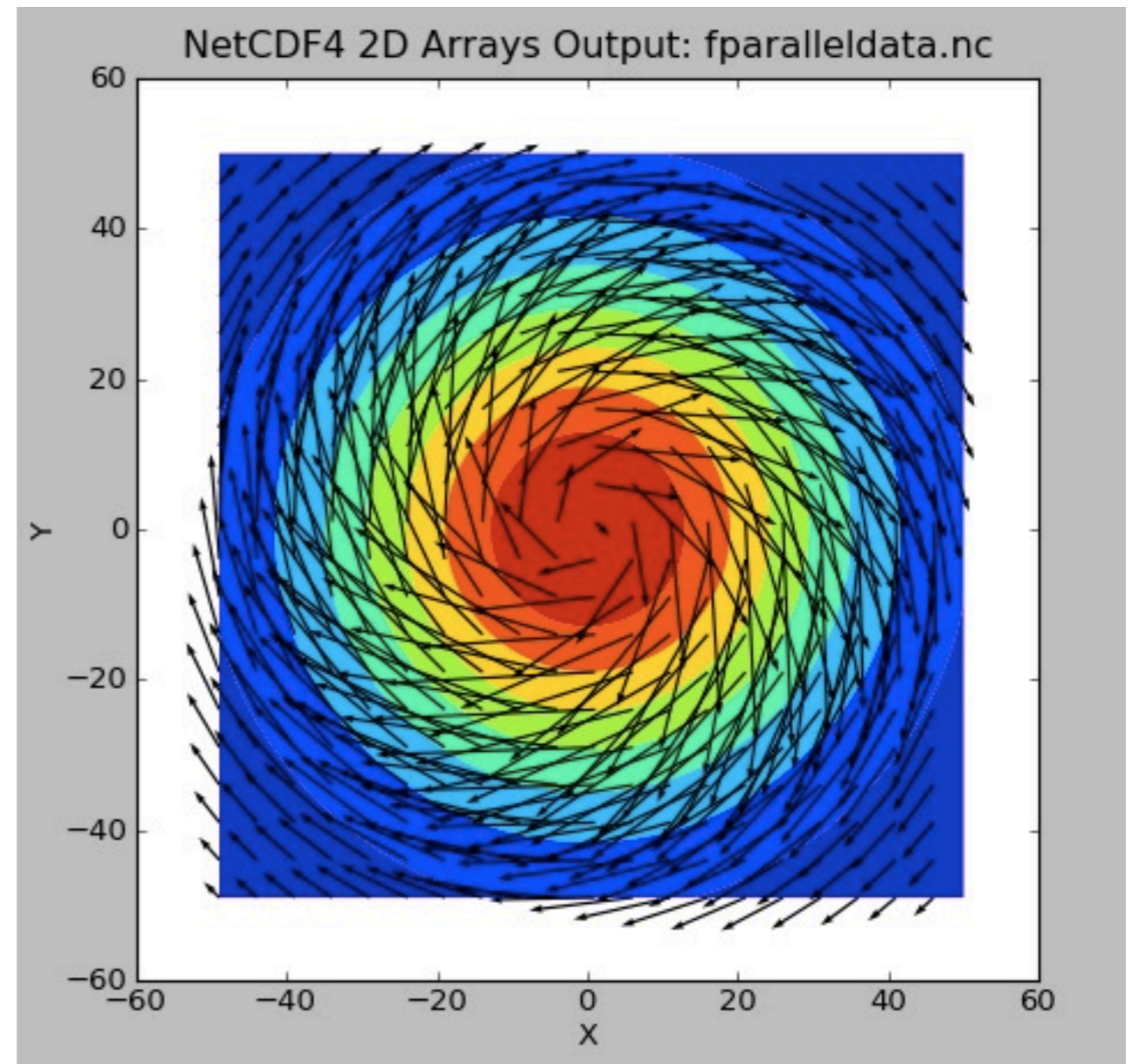
# Sample Code

```
$ cd  
$ cd parIO/netcdf  
  
$ source ../parallellibs  
$ 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.



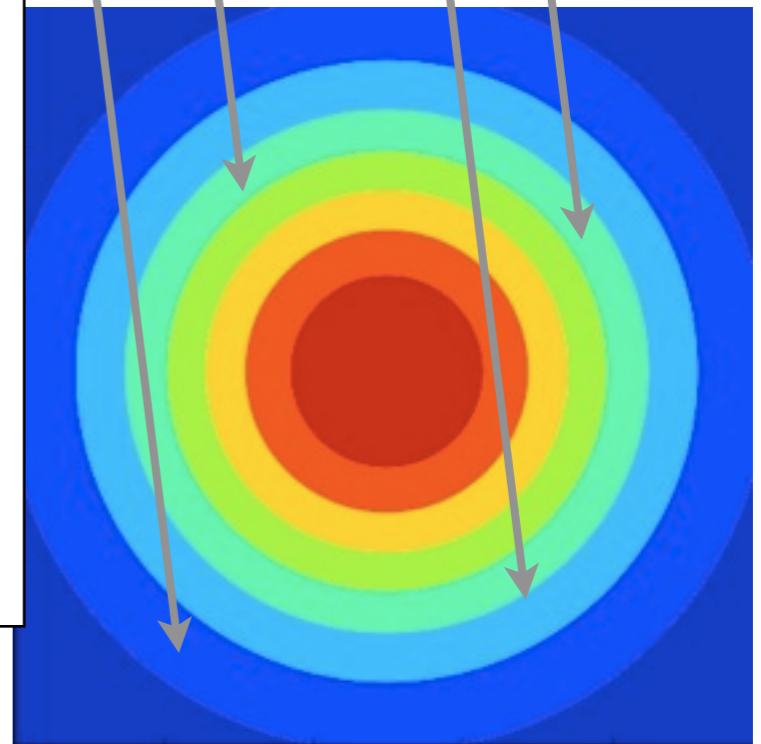
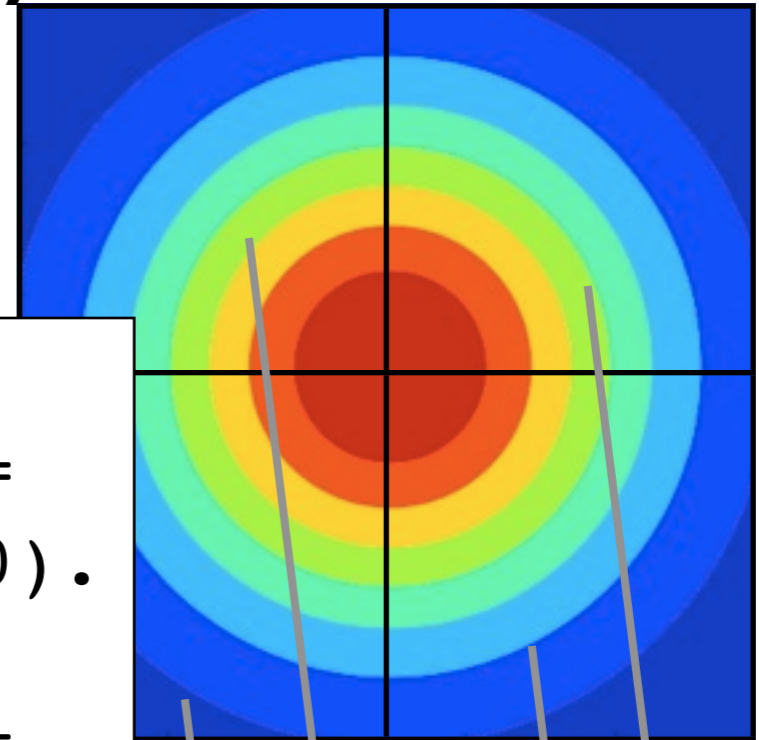
# Memory:

```
$ 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).

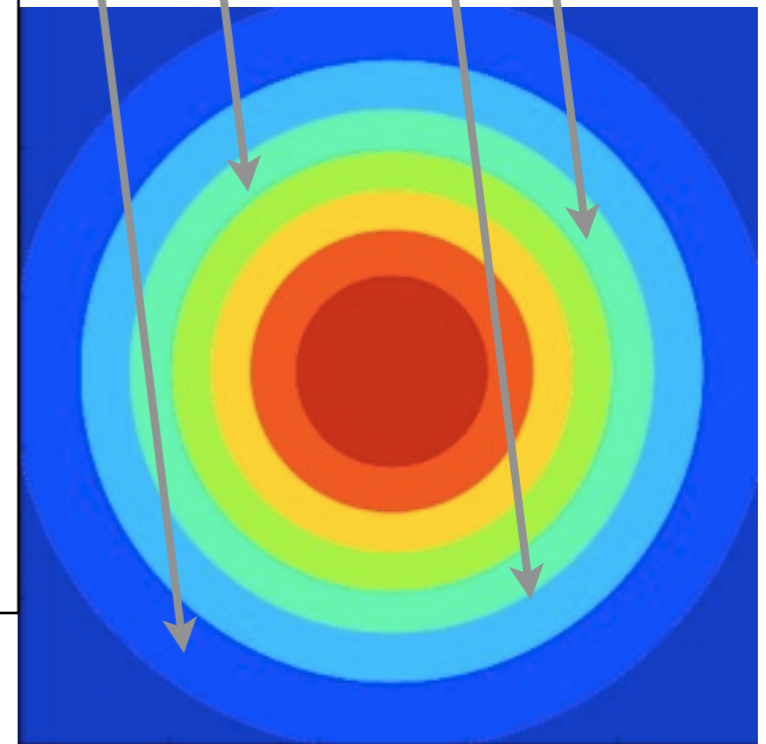
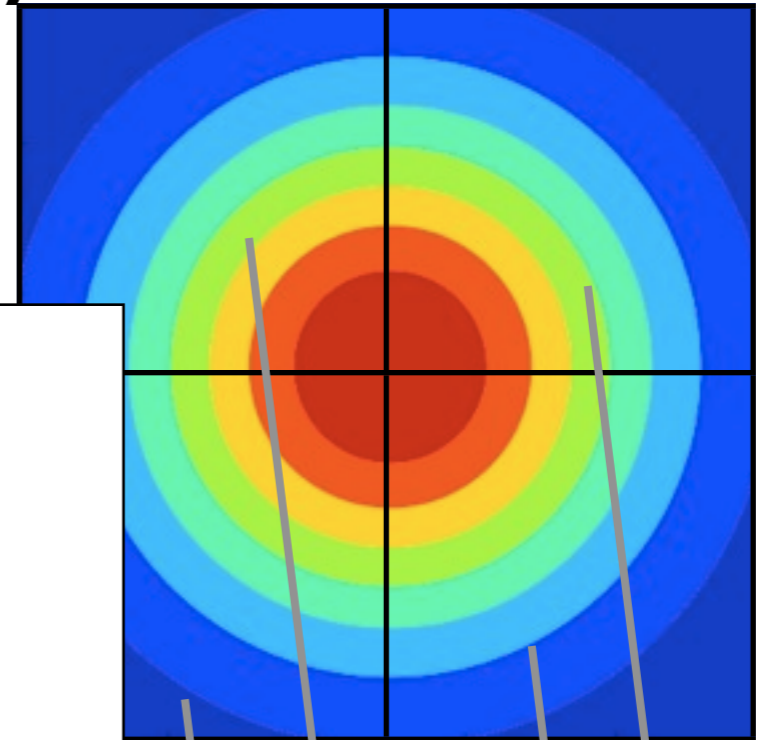
[ 2] gets ( 0, 1): local points =
( 50, 50); global points = (100,100).

[ 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|NC_
    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.filename);
    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), 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);

/* 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][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][0]));

nc_close(file_id);
return;
}

```

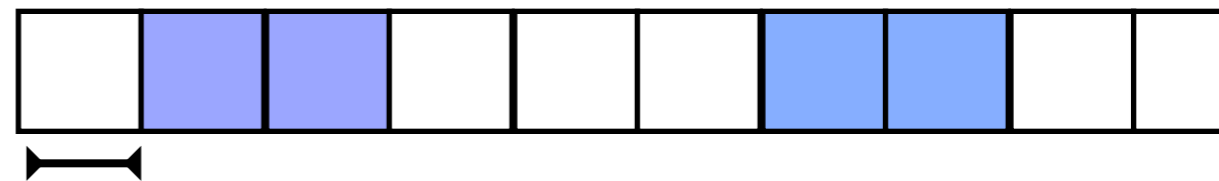
# HDF5 Hyperslabs

- 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

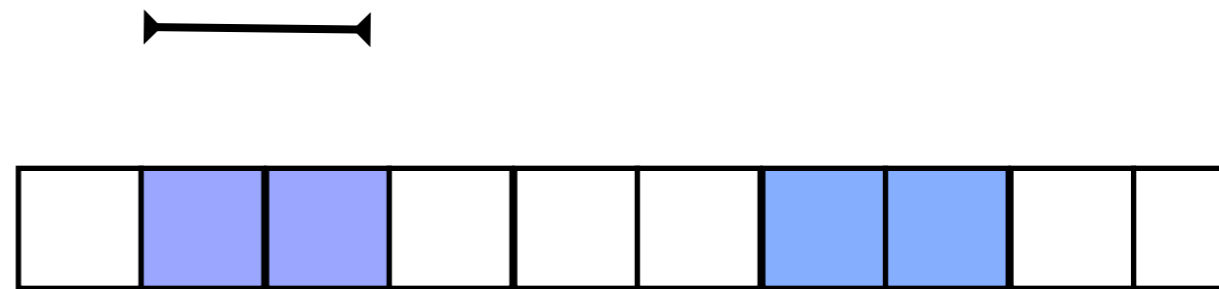


Offset = 1

# HDF5 Hyperslabs

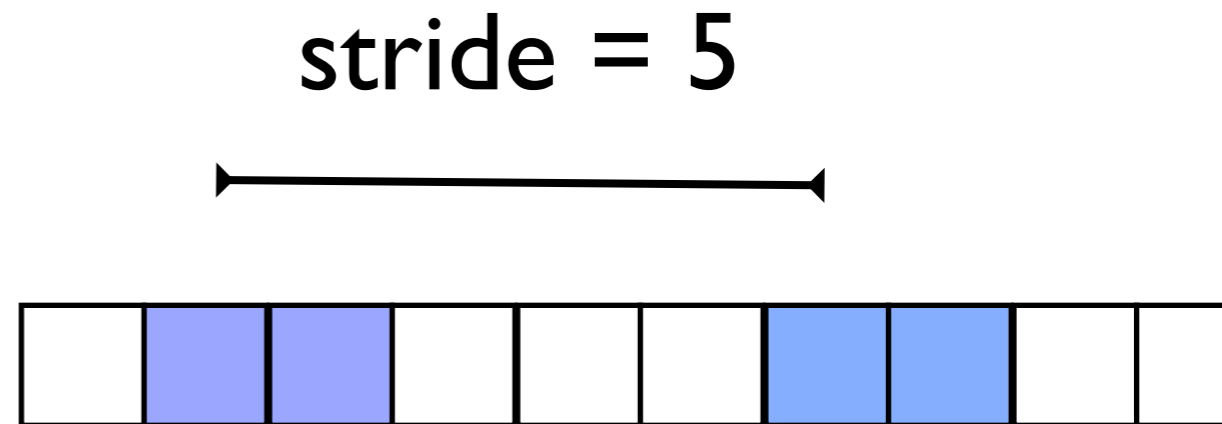
- Parallel HDF5 similar to parallel NetCDF - fairly modest changes to structure of code
- Different (more low-level, natch) way of dealing with sub-regions
- Offset, block, count, stride

blocksize = 2



# HDF5 Hyperlabs

- 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 Hyperlabs

- 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)

count = 2





# HDF5

## Hyperlabs

- 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
- Hyperlab - one of these per dimensions.
- (offset,block) just like (start, counts) in netcdf.

count = 2



# 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]));
```

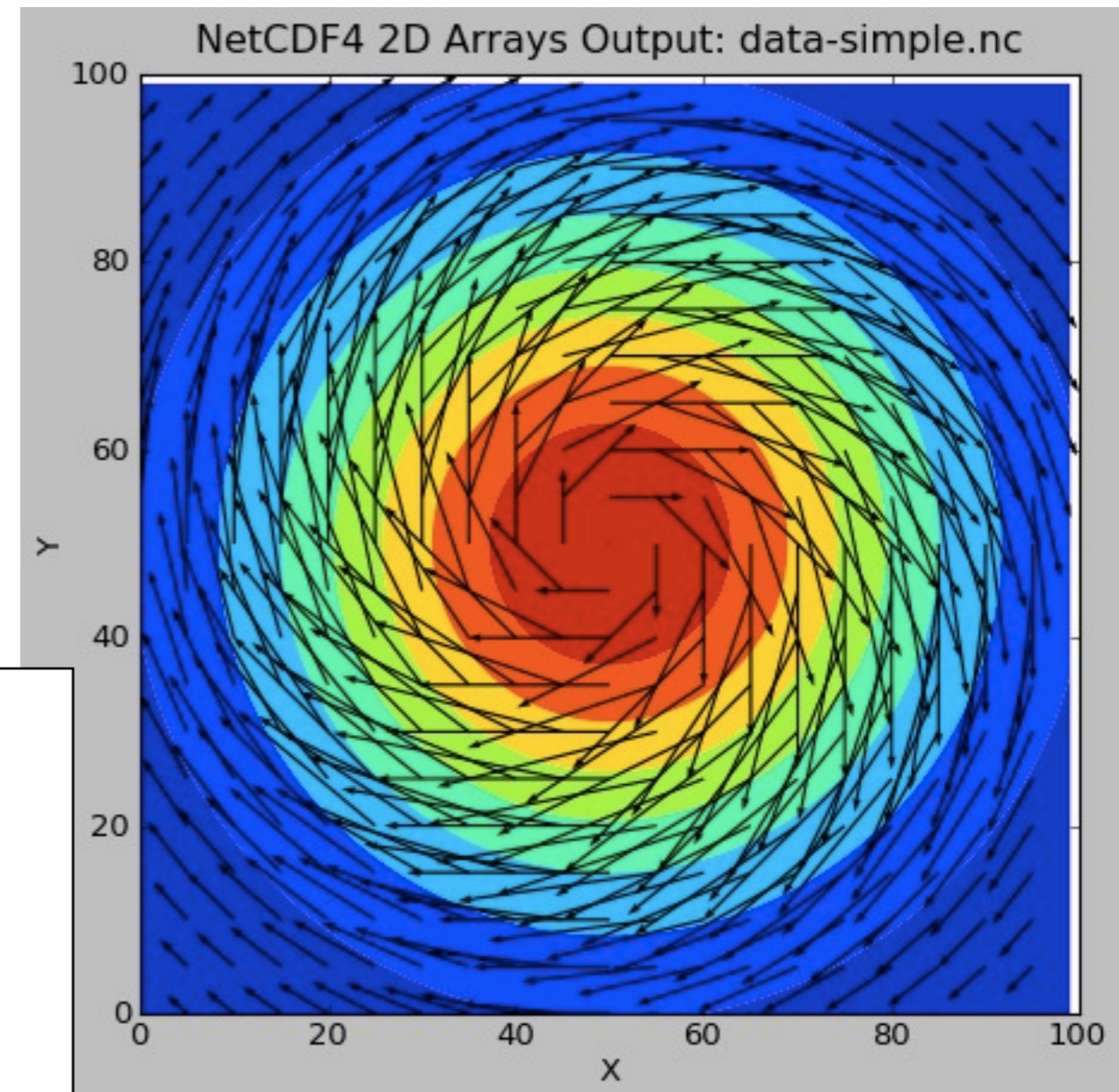
Select hyperslab, and write; parallelism is in  
distribution\_id

# Projects

```
$ cd parIO/hydro  
Write hdf5, netcdf outputs  
(soln in parIO/hydro/answers)
```

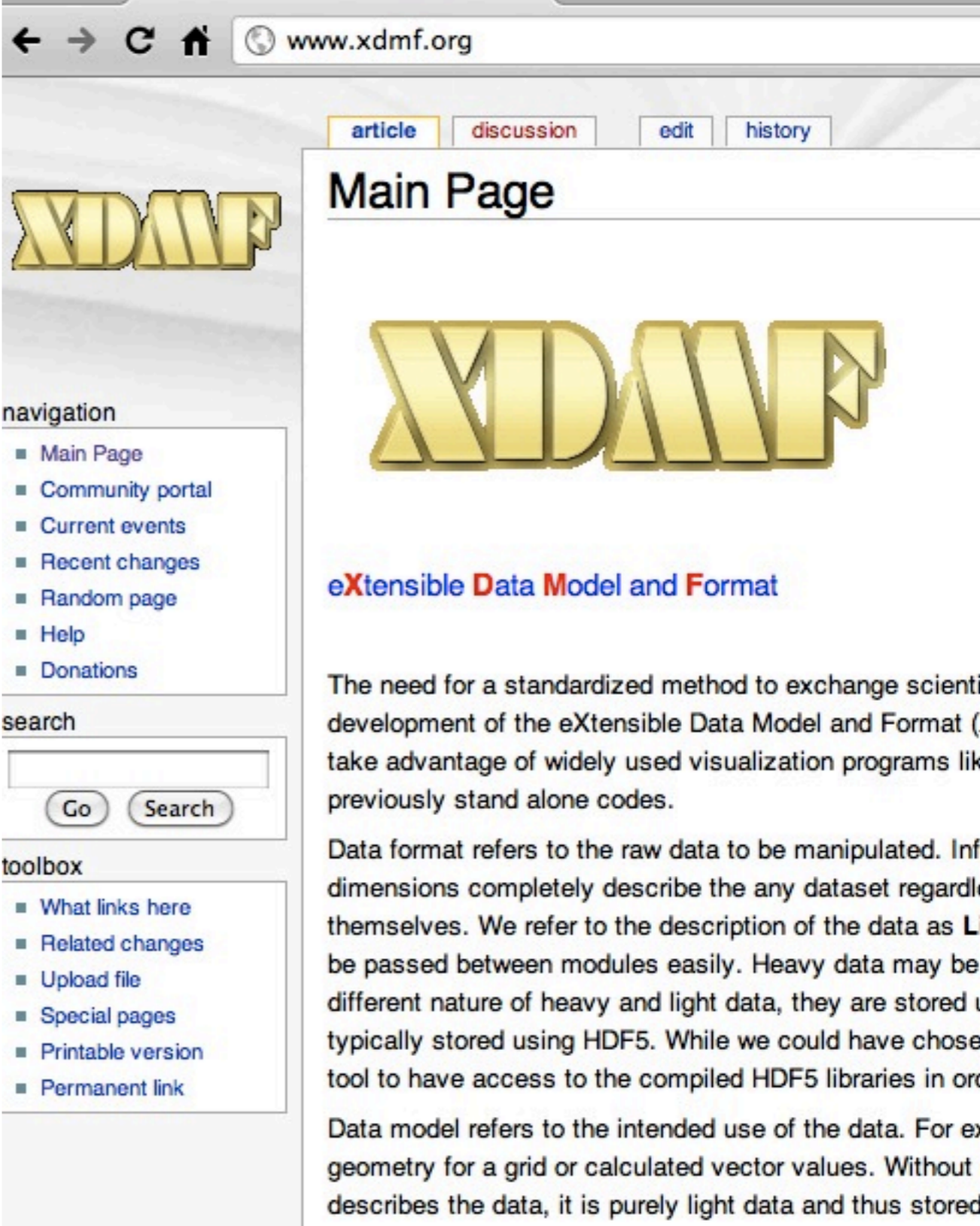
```
$ cd parIO/hydro-mpi  
Write ppm output in MPI-IO, and output in  
parallel hdf5, netcdf  
(partial soln in parIO/hydro/answers)
```

```
$ 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 interoperability (eg, w/ viz packages)




The screenshot shows the main page of the XDMF website (www.xdmf.org). The page features a navigation menu with links to Main Page, Community portal, Current events, Recent changes, Random page, Help, and Donations. A search box is also present. The main content area displays the XDMF logo and the text "eXtensible Data Model and Format". Below this, there is a paragraph explaining the need for a standardized method to exchange scientific data and the development of the eXtensible Data Model and Format (XDMF). The text continues to describe the data format and data model.

← → ↻ 🏠 [www.xdmf.org](http://www.xdmf.org)

[article](#) [discussion](#) [edit](#) [history](#)

## Main Page



**eXtensible Data Model and Format**

The need for a standardized method to exchange scientific data led to the development of the eXtensible Data Model and Format (XDMF). XDMF files take advantage of widely used visualization programs like ParaView and previously stand alone codes.

Data format refers to the raw data to be manipulated. Information dimensions completely describe the any dataset regardless of its size. They themselves. We refer to the description of the data as Light Data Model (LDM) be passed between modules easily. Heavy data may be of a different nature of heavy and light data, they are stored using HDF5. While we could have chosen a different tool to have access to the compiled HDF5 libraries in order to store the data, it is purely light data and thus stored using HDF5.

Data model refers to the intended use of the data. For example, a data model for geometry for a grid or calculated vector values. Without a data model, it describes the data, it is purely light data and thus stored using HDF5.

**navigation**

- [Main Page](#)
- [Community portal](#)
- [Current events](#)
- [Recent changes](#)
- [Random page](#)
- [Help](#)
- [Donations](#)

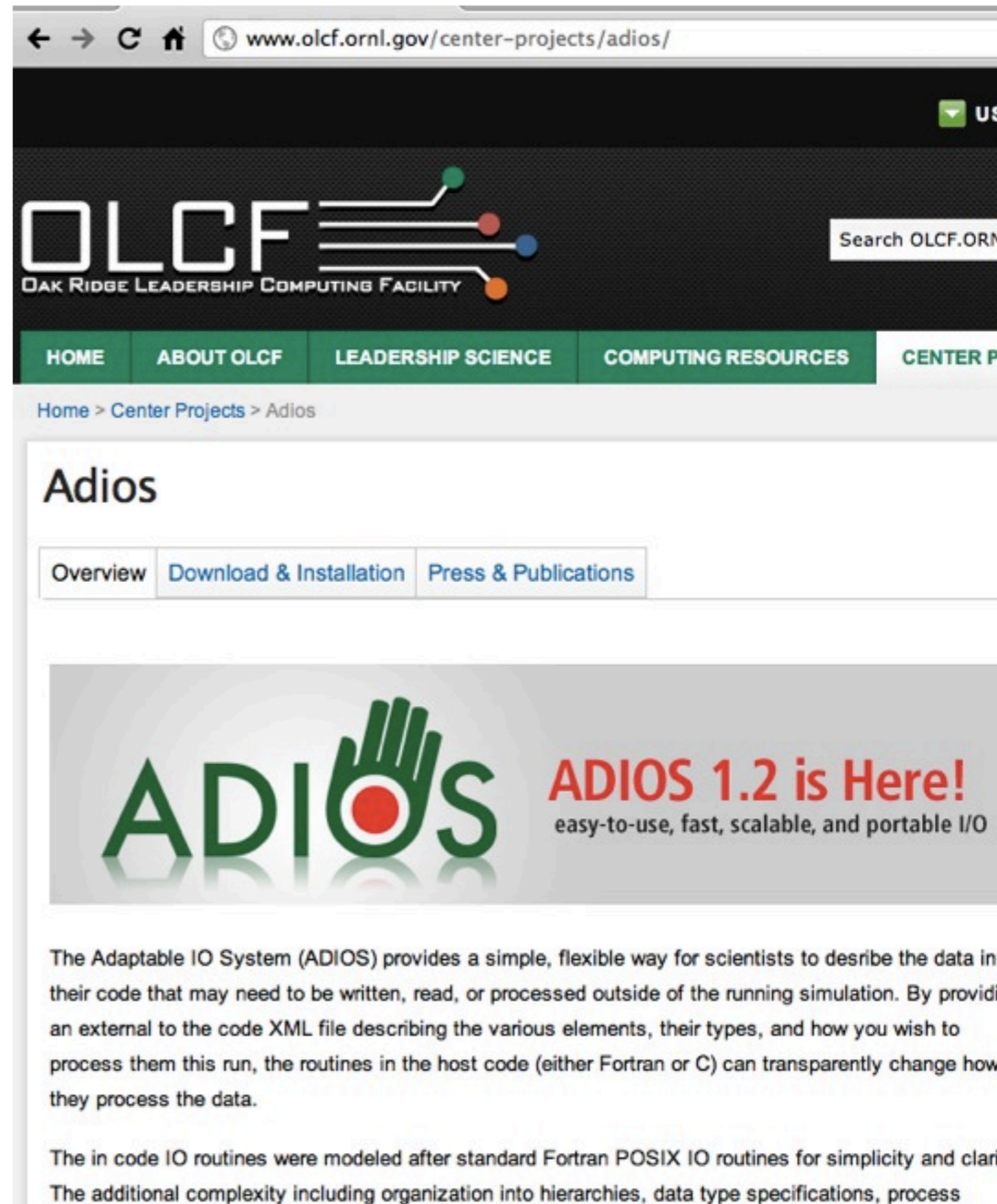
**search**

**toolbox**

- [What links here](#)
- [Related changes](#)
- [Upload file](#)
- [Special pages](#)
- [Printable version](#)
- [Permanent link](#)

# Adaptable IO System

- ADIOS
- A library for IO for scientific code
- Uses MPIIO, HDF5, etc... under the hood
- Allows changing of IO strategy, method at run time, not rewrite code and recompile
- Promising, but new.



The screenshot shows the website for the Adaptable IO System (ADIOS) at the Oak Ridge Leadership Computing Facility (OLCF). The page features the OLCF logo at the top, a navigation menu with links for HOME, ABOUT OLCF, LEADERSHIP SCIENCE, COMPUTING RESOURCES, and CENTER P, and a search bar. The main content area is titled "Adios" and includes tabs for Overview, Download & Installation, and Press & Publications. A prominent banner for "ADIOS 1.2 is Here!" is displayed, with the tagline "easy-to-use, fast, scalable, and portable I/O". Below the banner, the text describes the system as a simple, flexible way for scientists to describe data in their code that may need to be written, read, or processed outside of the running simulation. It mentions that the system provides an external XML file to describe data elements and their types, allowing host code routines to change how they process data. The text also notes that the in-code IO routines were modeled after standard Fortran POSIX IO routines for simplicity and clarity, and that the system includes additional complexity for organization into hierarchies, data type specifications, and process management.