Intro to HDF5

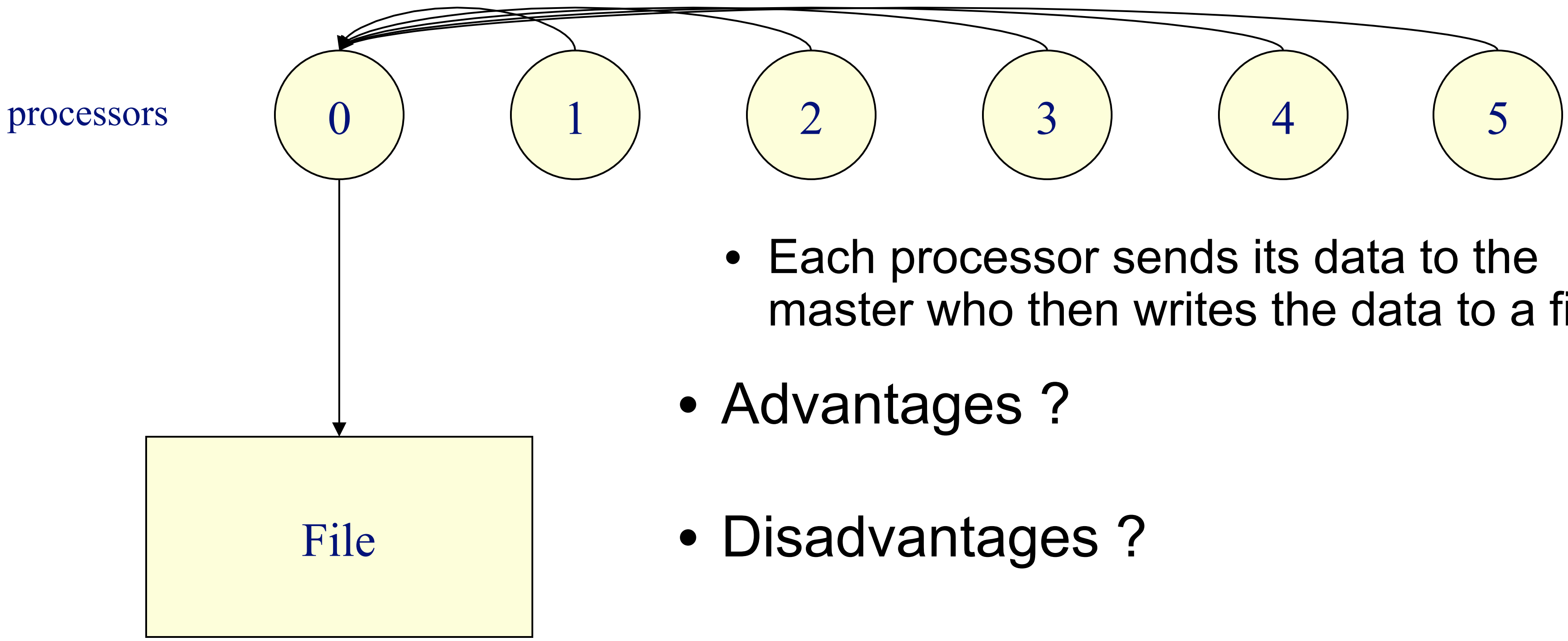
Katie Antypas
NERSC

Tutorial in part from HDF Group

NUG 2010
Berkeley, CA

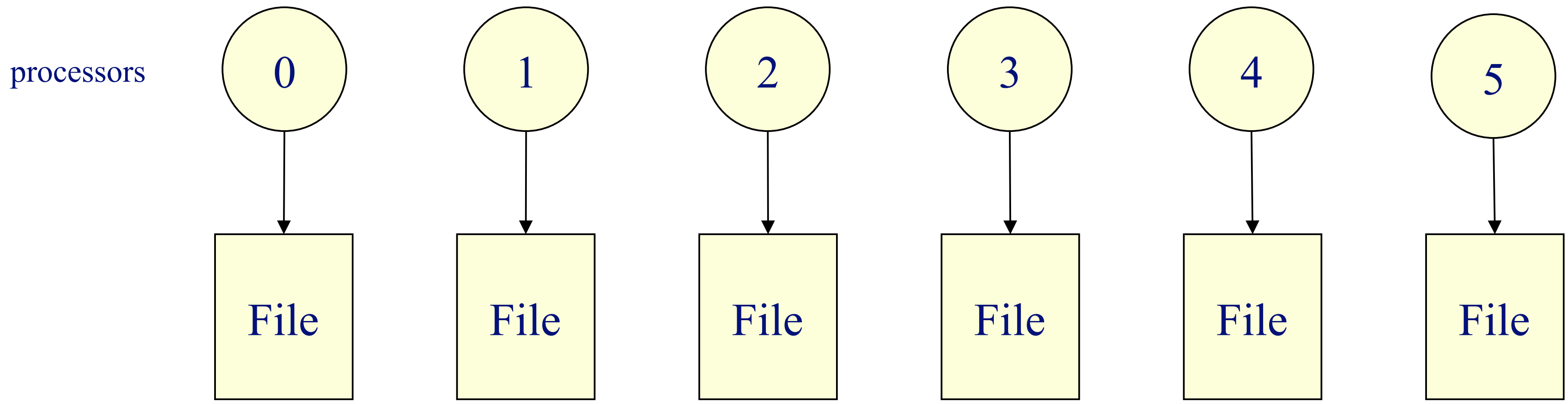
Oct 18th, 2010

Serial I/O



- Each processor sends its data to the master who then writes the data to a file
- Advantages ?
- Disadvantages ?

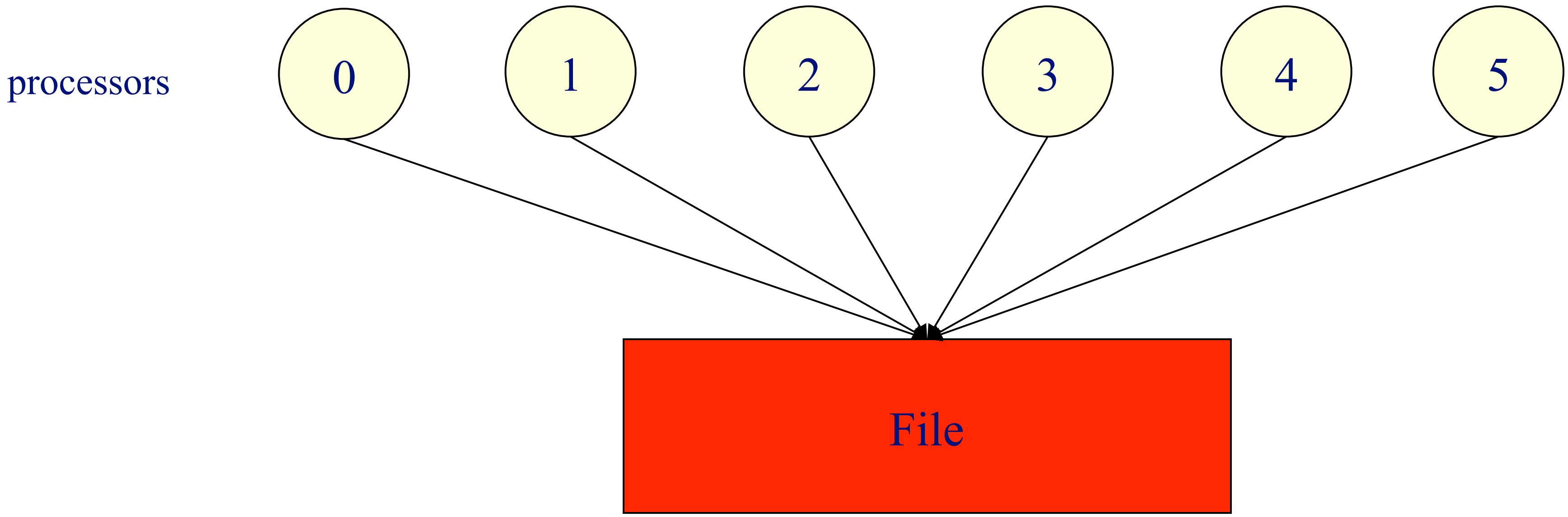
Parallel I/O Multi-file



- Each processor writes its own data to a separate file
  - Advantages ?

  - Disadvantages ?

Parallel I/O Single-file



- Each processor writes its own data to the same file using MPI-IO mapping
- Advantages ?
- Disadvantages ?

What is a High Level Parallel I/O Library?

- An API which helps to express scientific simulation data in a more natural way
  - Multi-dimensional data, labels and tags, non-contiguous data, typed data
- Typically sits on top of MPI-IO layer and can use MPI-IO optimizations
- Offer
  - Simplicity for visualization and analysis
  - Portable formats - can run on one machine and take output to another
  - Longevity - output will last and be accessible with library tools and no need to remember version number of code

Common Storage Formats

- ASCII:
  - Slow
  - Takes more space!
  - Inaccurate
- Binary
  - Non-portable (eg. byte ordering and types sizes)
  - Not future proof
  - Parallel I/O using MPI-IO
- Self-Describing formats
  - NetCDF/HDF4, HDF5, Parallel NetCDF
  - Example in HDF5: API implements Object DB model in portable file
  - Parallel I/O using: pHDF5/pNetCDF (hides MPI-IO)
- Community File Formats
  - FITS, HDF-EOS, SAF, PDB, Plot3D
  - Modern Implementations built on top of HDF, NetCDF, or other self-describing object-model API

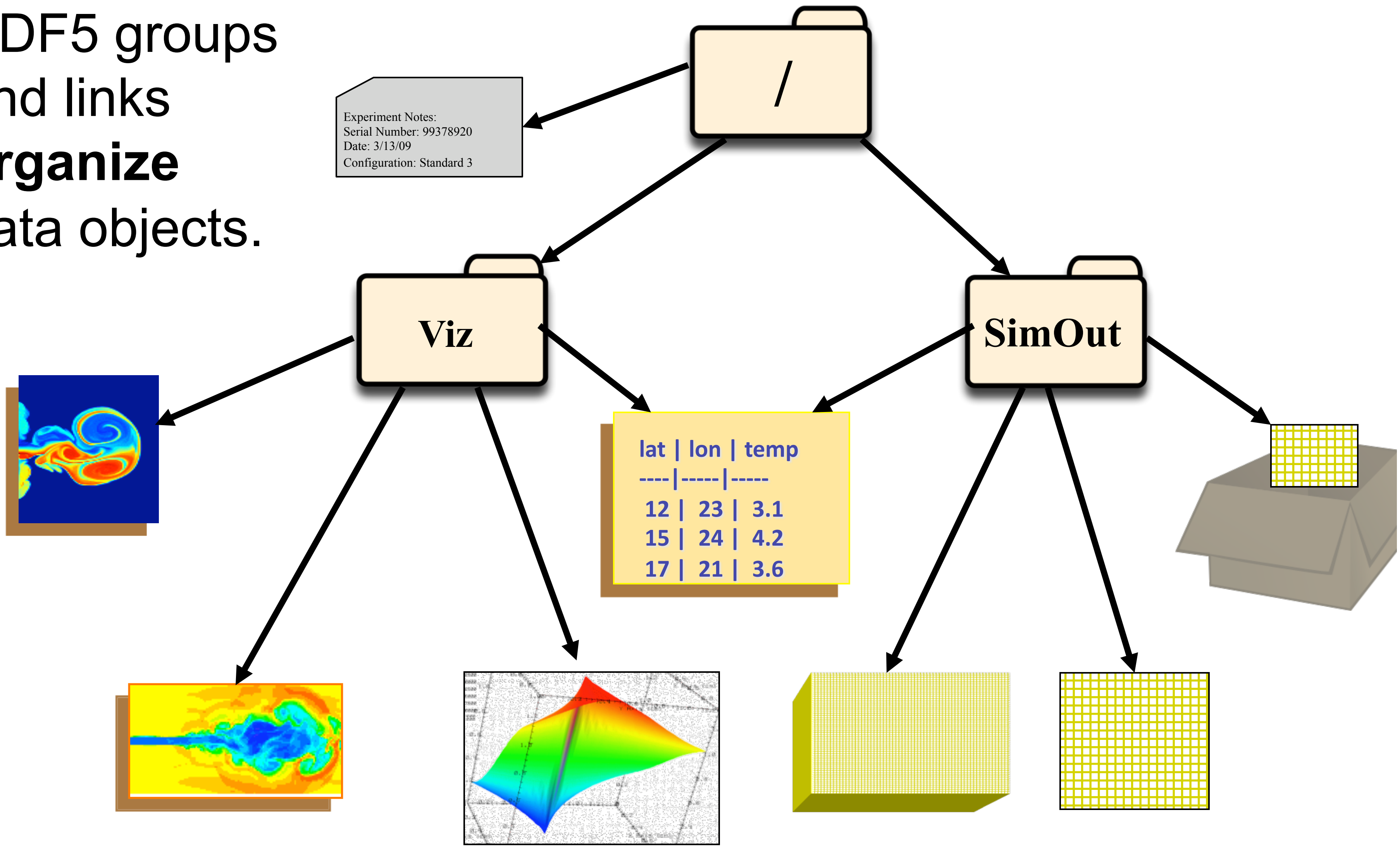
Many NERSC users at this level. We would like to encourage users to transition to a higher IO library

But what about performance?

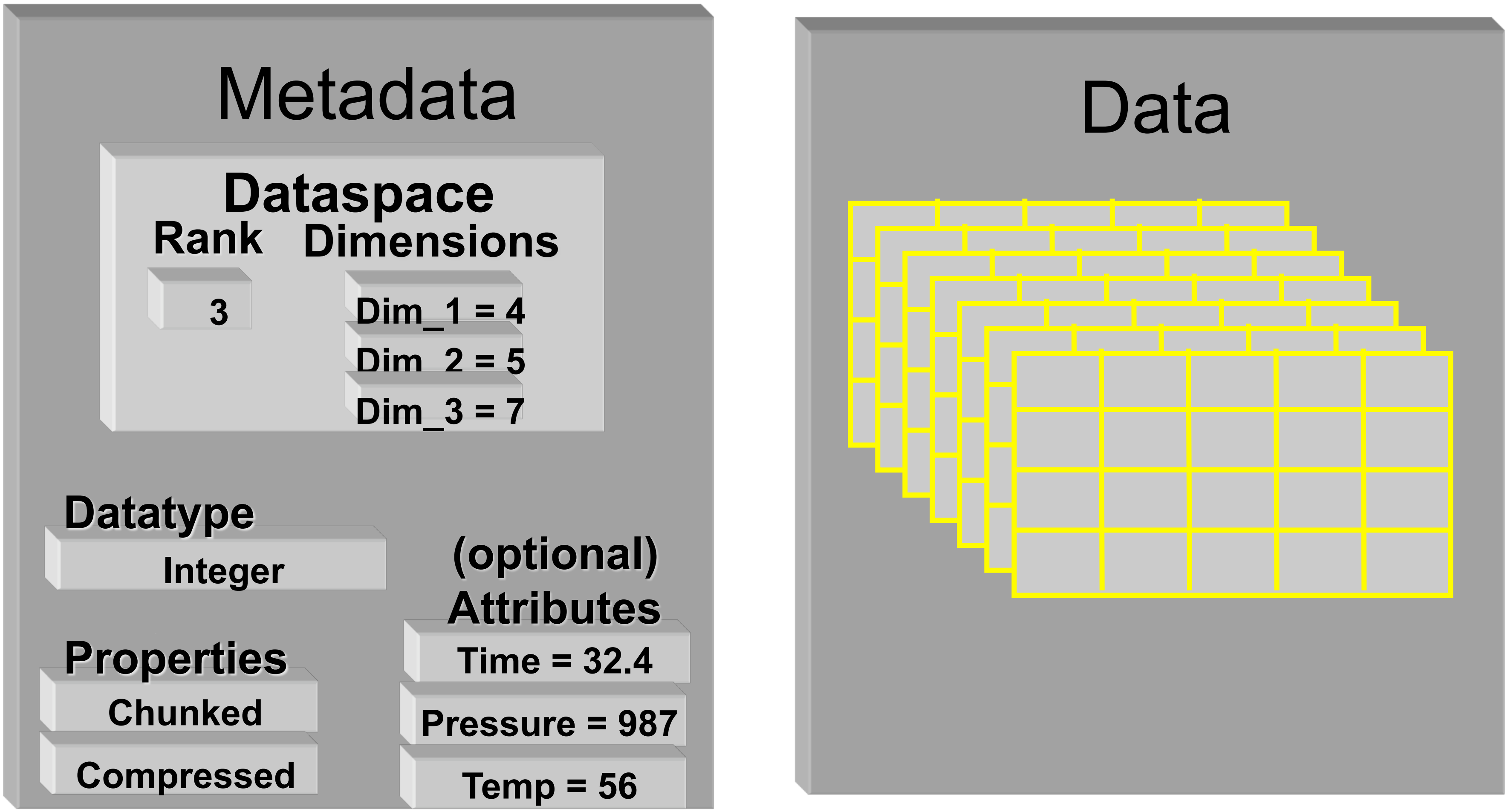
- Hand tuned I/O for a particular application and architecture will likely perform better, but …
- Purpose of I/O libraries is not only portability, longevity, simplicity, but productivity
- Using own binary file format forces user to understand layers below the application to get optimal IO performance
- Every time code is ported to a new machine or underlying file system is changed or upgraded, user is required to make changes to improve IO performance
- Let other people do the work
  - HDF5 can be optimized for given platforms and file systems by library developers
- Goal is for shared file performance to be 'close enough'

HDF5 File is a Container of Objects

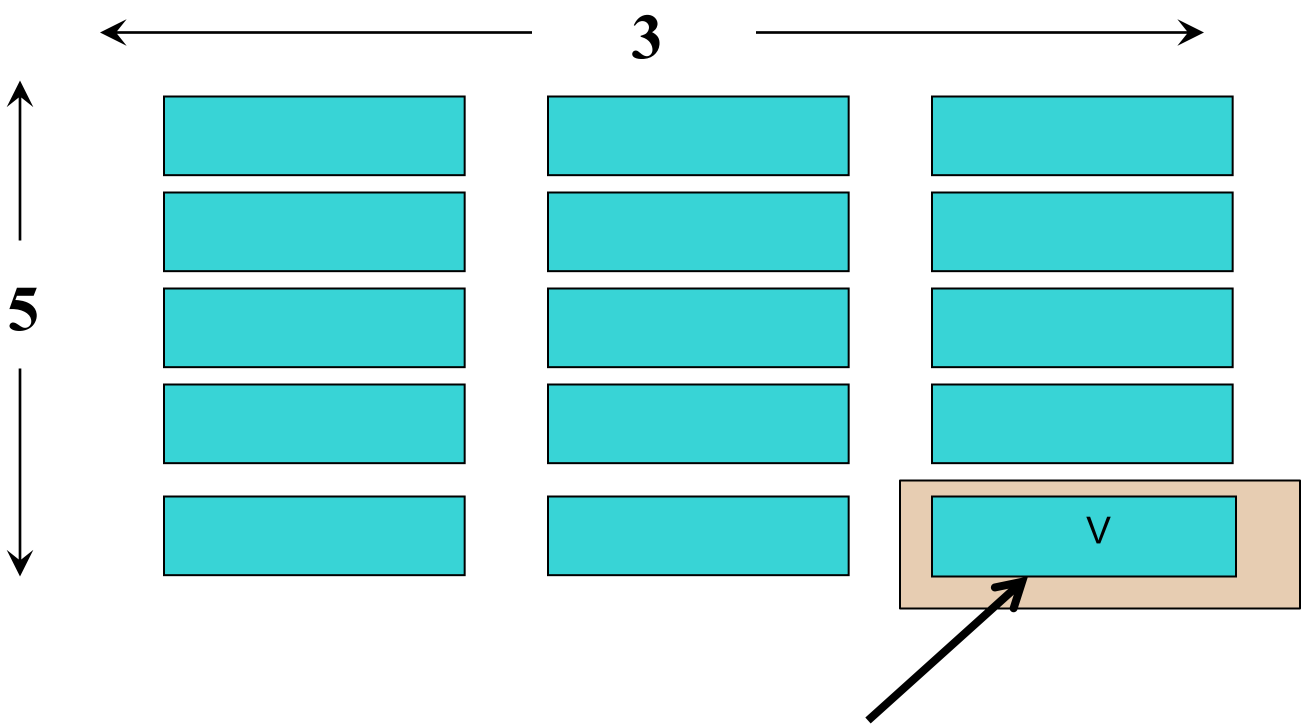
HDF5 groups and links **organize** data objects.



HDF5 Dataset



HDF5 Dataset



Datatype: 16-byte integer

Dataspace: Rank = 2
Dimensions = 5 x 3

HDF5 Datatypes

The HDF5 datatype describes how to interpret individual data elements.

HDF5 datatypes include:

- integer, float, unsigned, bitfield, …
- user-definable (e.g., 13-bit integer)
- variable length types (e.g., strings)
- references to objects/dataset regions
- enumerations - names mapped to integers
- opaque
- compound (similar to C structs)

HDF5 Pre-defined Datatype Identifiers

HDF5 defines set of Datatype Identifiers per HDF5 session.

For example:

C Type	HDF5 File Type	HDF5 Memory Type
int	H5T_STD_I32BE H5T_STD_I32LE	H5T_NATIVE_INT
float	H5T_IEEE_F32BE H5T_IEEE_F32LE	H5T_NATIVE_FLOAT
double	H5T_IEEE_F64BE H5T_IEEE_F64LE	H5T_NATIVE_DOUBLE

HDF5 Defined Types

For portability, the HDF5 library has its own defined types:

hid_t: object identifiers (native *integer*)

hsize_t: size used for dimensions (*unsigned long* or *unsigned long long*)

herr_t: function return value

For **C**, include hdf5.h in your HDF5 application.

Basic Functions

H5Fcreate (H5Fopen)	*create (open) File*
H5Screate_simple/H5Screate	*create fileSpace*
H5Dcreate (H5Dopen)	*create (open) Dataset*
H5Sselect_hyperslab	*select subsections of data*
H5Dread, H5Dwrite	*access Dataset*
H5Dclose	*close Dataset*
H5Sclose	*close fileSpace*
H5Fclose	*close File*

NOTE: Order not strictly specified.

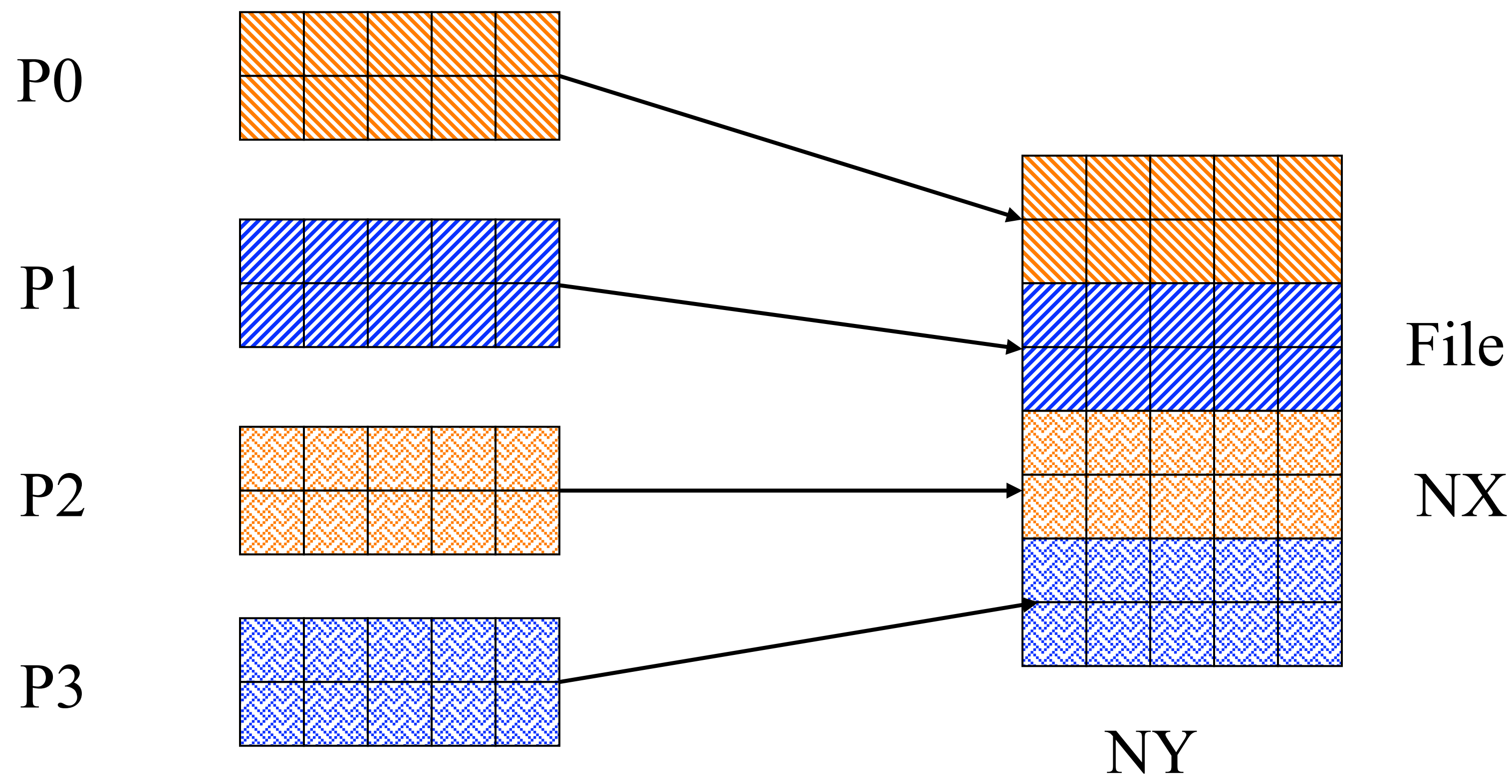
Logistics

- Log into franklin or carver
- “ssh franklin.nersc.gov” or “ssh carver.nersc.gov”
- “cp /project/projectdirs/training/pHDF5_examples.tar $SCRATCH”
- “cd $SCRATCH”
- “tar xvf pHDF5_examples.tar”
- Here you will find the code examples, submission scripts and detailed instructions in “instructions_carver.txt” or “instructions_franklin.txt”

Example : write_grid_rows.c

(or fortran90 version if you prefer)

Example 1: *Writing dataset by rows*



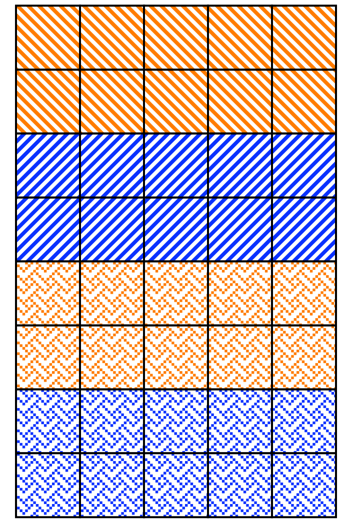
Writing by rows: *Output of h5dump*

```
HDF5 "grid_rows.h5" {
GROUP "/" {
 DATASET "dataset1" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 8, 5 ) / ( 8, 5 ) }
      DATA {
         18, 18, 18, 18, 18,
         18, 18, 18, 18, 18,
         19, 19, 19, 19, 19,
         19, 19, 19, 19, 19,
         20, 20, 20, 20, 20,
         20, 20, 20, 20, 20,
         21, 21, 21, 21, 21,
         21, 21, 21, 21, 21
      }
   }
}
```

Initialize the file for parallel access

```
/* first initialize MPI */

/* create access property list */
plist_id = H5Pcreate(H5P_FILE_ACCESS);

/* necessary for parallel access */
status = H5Pset_fapl_mpio(plist_id,
MPI_COMM_WORLD, MPI_INFO_NULL);

/* Create an hdf5 file */
file_id = H5Fcreate(FILENAME, H5F_ACC_TRUNC,
H5P_DEFAULT, plist_id);


status = H5Pclose(plist_id);
```

Create file filespace and dataset

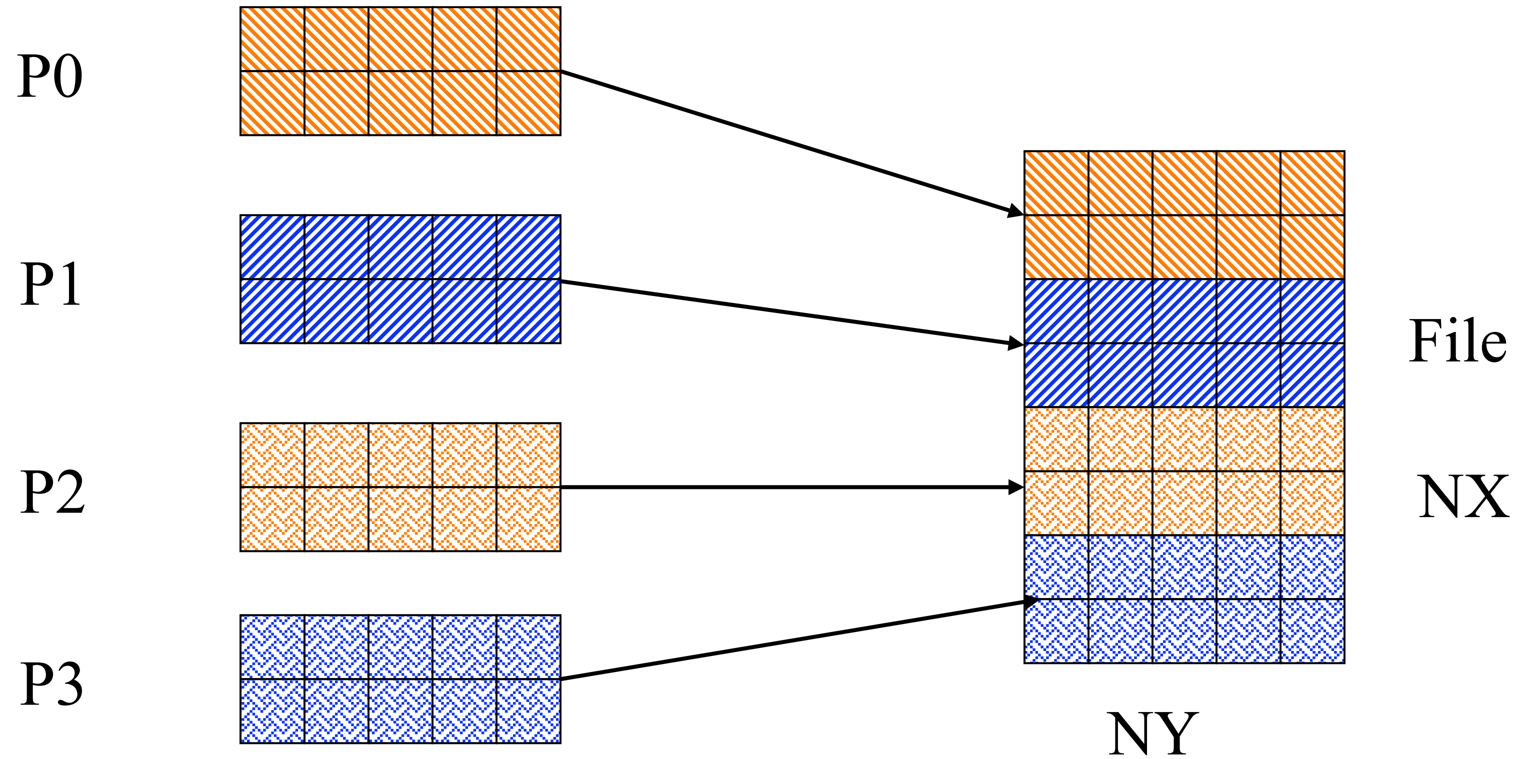


```
/* initialize local grid data */

/* Create the filespace */


dimsf[0] = NX;
dimsf[1] = NY;


filespace = H5Screate_simple(RANK, dimsf,NULL);

/* create a dataset */
dset_id = H5Dcreate(file_id, "dataset1”,
H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT,
H5P_DEFAULT, H5P_DEFAULT);
```

Create Property List

```
/* Create property list for collective dataset
write. */

plist_id = H5Pcreate(H5P_DATASET_XFER);

/* The other option is HDFD_MPIO_INDEPENDENT */
H5Pset_dxpl_mpio(plist_id,H5FD_MPIO_COLLECTIVE);
```

Calculate Offsets



Every processor has a 2d array, which holds the number of blocks to write and the starting offset

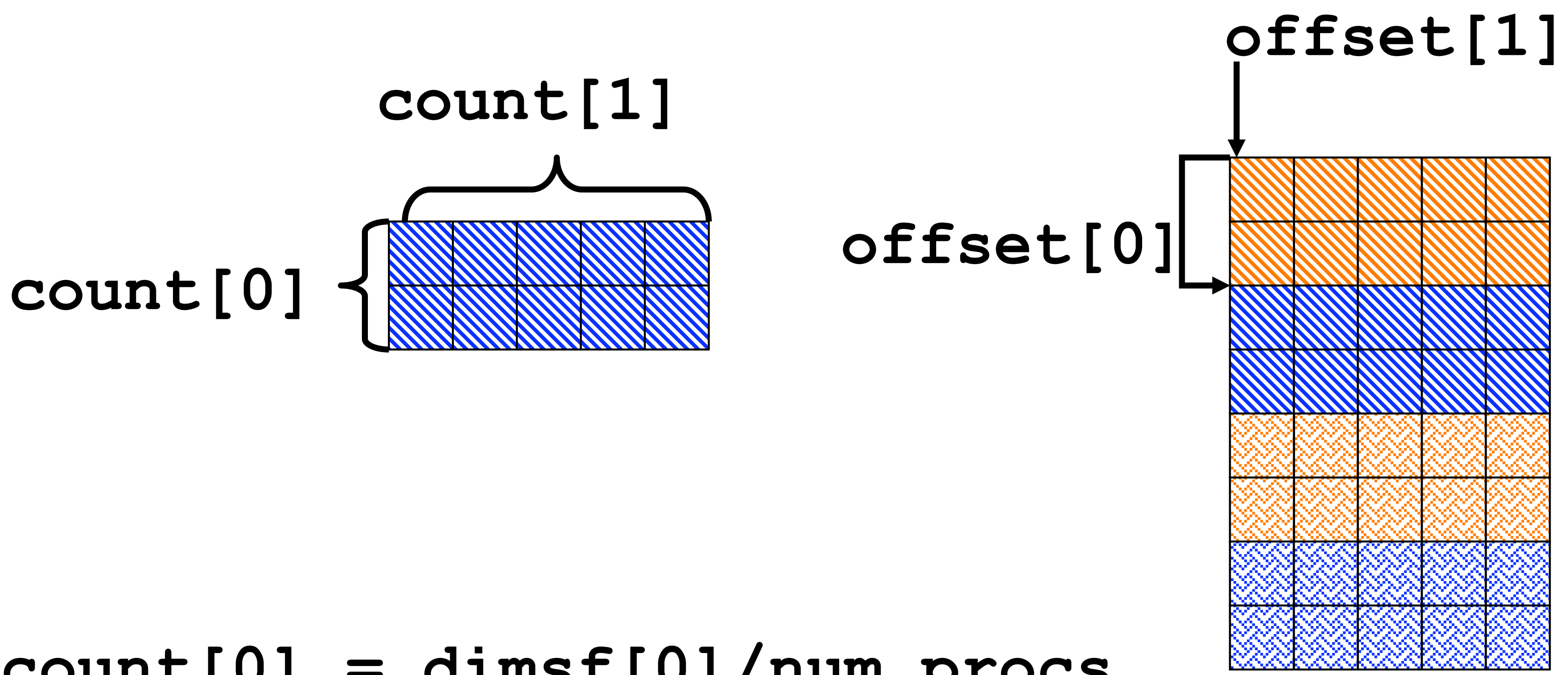
count[0], count[1]
offet[0][offset[1]

Example 1: *Writing dataset by rows*

Process 1

Memory

File

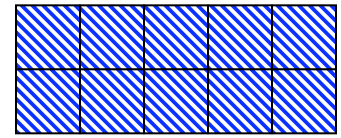



```
count[0] = dimsf[0]/num_procs
count[1] = dimsf[1];
offset[0] = my_proc * count[0];  /* = 2 */
offset[1] = 0;
```

Writing and Reading Hyperslabs

- Distributed memory model: data is split among processes
- PHDF5 uses HDF5 hyperslab model
- Each process defines memory and file hyperslabs
- Each process executes partial write/read call
  - Collective calls
  - Independent calls

Create a Memory Space select hyperslab



```
/* Create the local memory space */
memspace = H5Screate_simple(RANK, count, NULL);


filespace = H5Dget_space (dset_id);

/* Create the hyperslab -- says how you want to
lay out data */

status = H5Sselect_hyperslab(filespace,
H5S_SELECT_SET, offset, NULL, count, NULL);
```

Write Data

Identifier for dataset "dataset1"

Datatype

```
status = H5Dwrite(dset_id, H5T_NATIVE_DOUBLE,
memspace, filespace, plist_id, grid_data);
```

Access Properties:
We choose collective.
This is where other optimizations could be added.

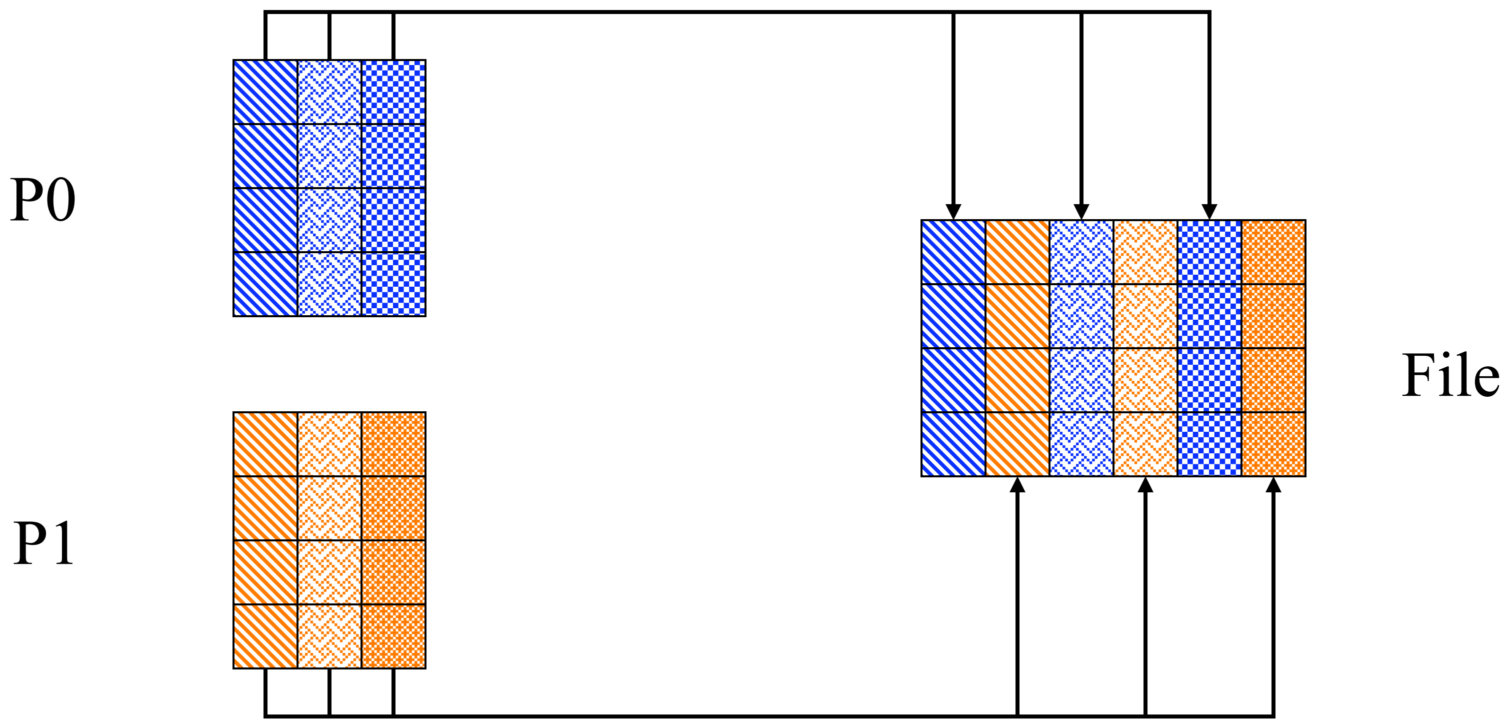
Data buffer

Then close every dataspace and file space that was opened

How to Compile PHDF5 Applications

- h5pcc – HDF5 C compiler command
  - Similar to mpicc
- h5pfc – HDF5 F90 compiler command
  - Similar to mpif90
- To compile:
  - % h5pcc h5prog.c
  - % h5pfc h5prog.f90

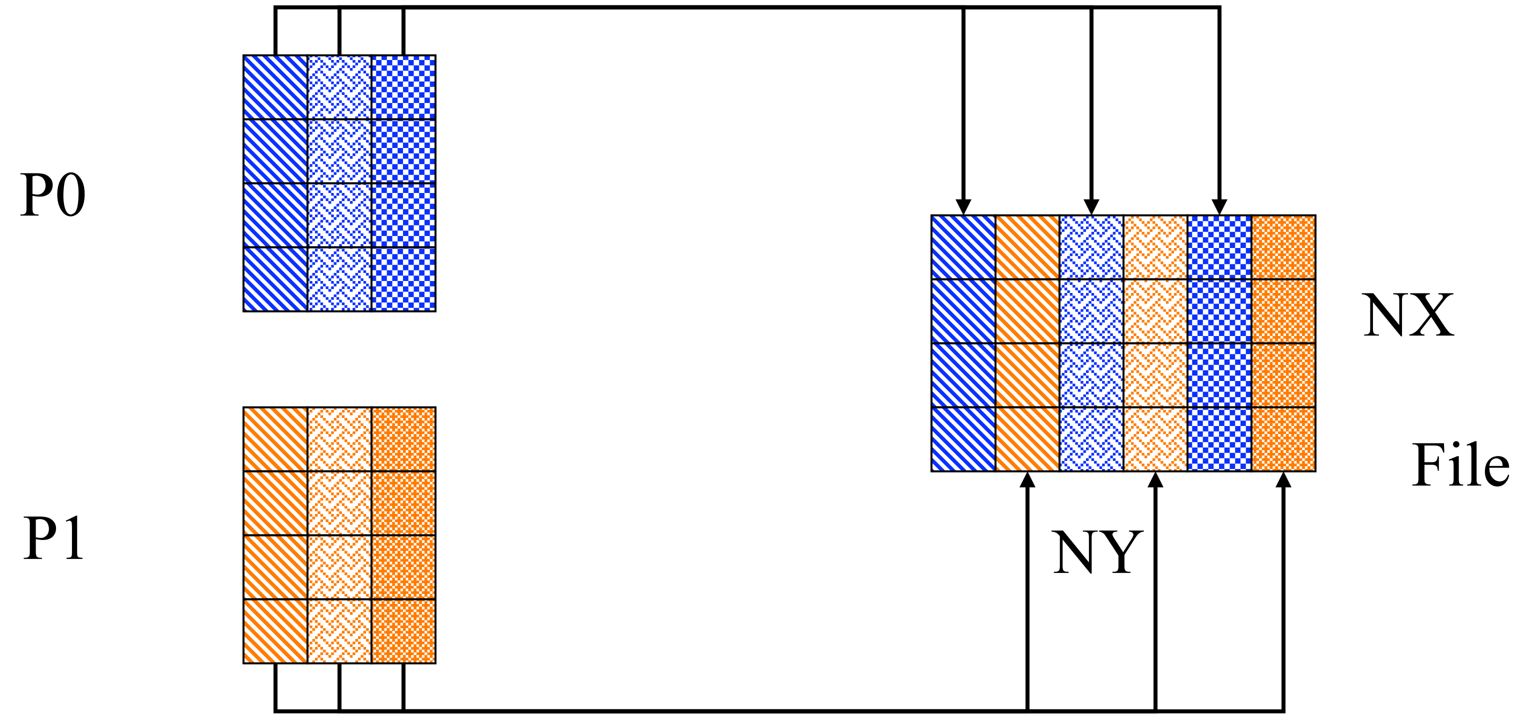
Example 2: *Writing dataset by columns*



Writing by columns: *Output of h5dump,*

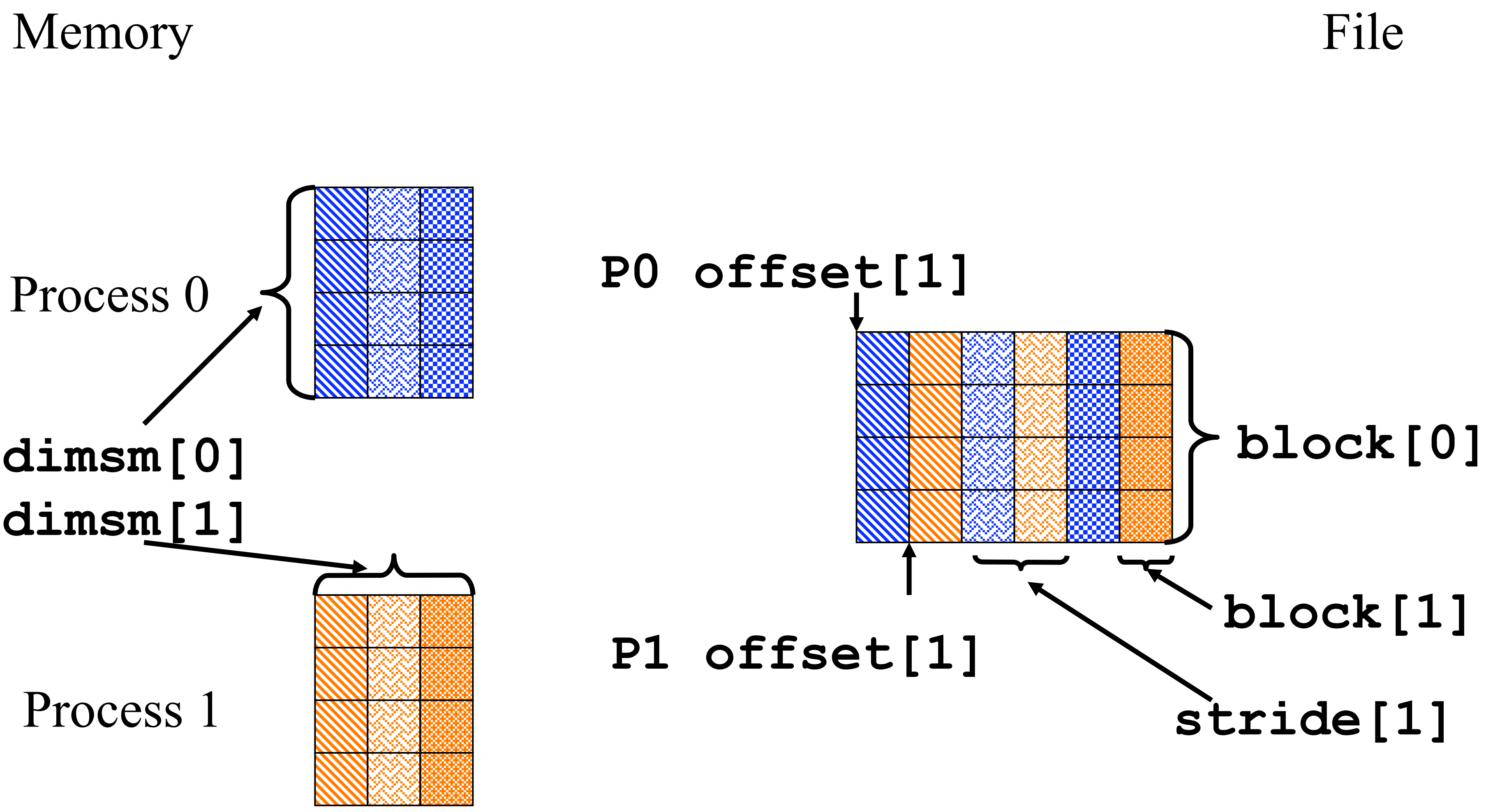
```
HDF5 "grid_cols.h5" {
GROUP "/" {
   DATASET "dataset1" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 8, 6 ) }
      DATA {
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200

      }
   }
 }
}
```

Example 2: *Writing dataset by columns*



- More complicated pattern, describe data layout with 4 arrays
- offset[] - starting position
- stride[] - spacing to the next element
- count[] - how many times to write a contiguous block
- block[] - how many contiguous elements to write

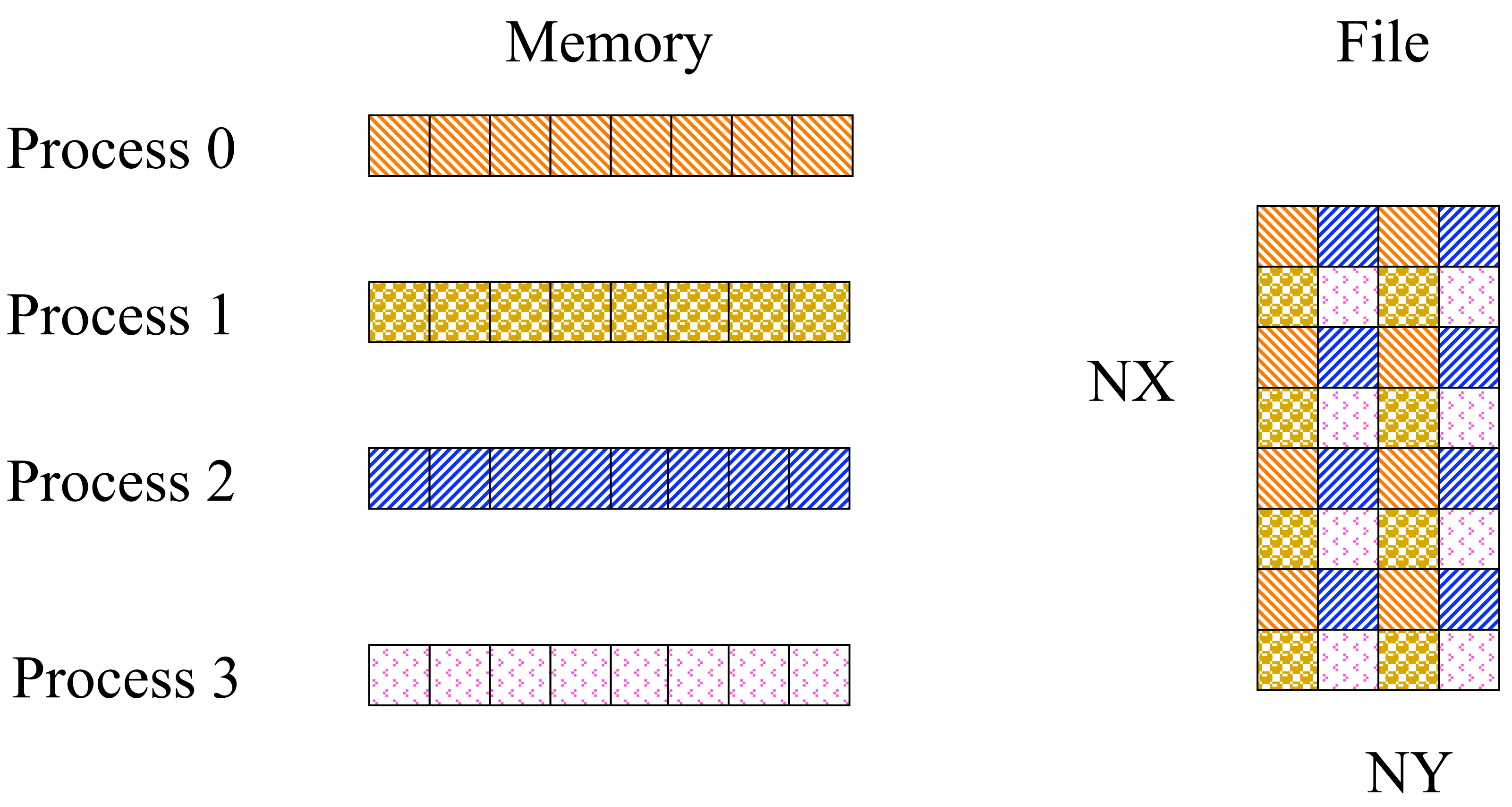
Example 2: *Writing dataset by column*



Example 2: *Writing dataset by column*

```
/* Each process defines hyperslab in
 the file */

count[0] = 1;
count[1] = dimsm[1];
offset[0] = 0;
offset[1] = my_proc;
stride[0] = 1;
 stride[1] = 2;
block[0] = dimsm[0];
block[1] = 1;


/* Each process selects hyperslab.
   filespace = H5Dget_space(dset_id); */

H5Sselect_hyperslab(filespace,
         H5S_SELECT_SET, offset, stride,
         count, block);
```

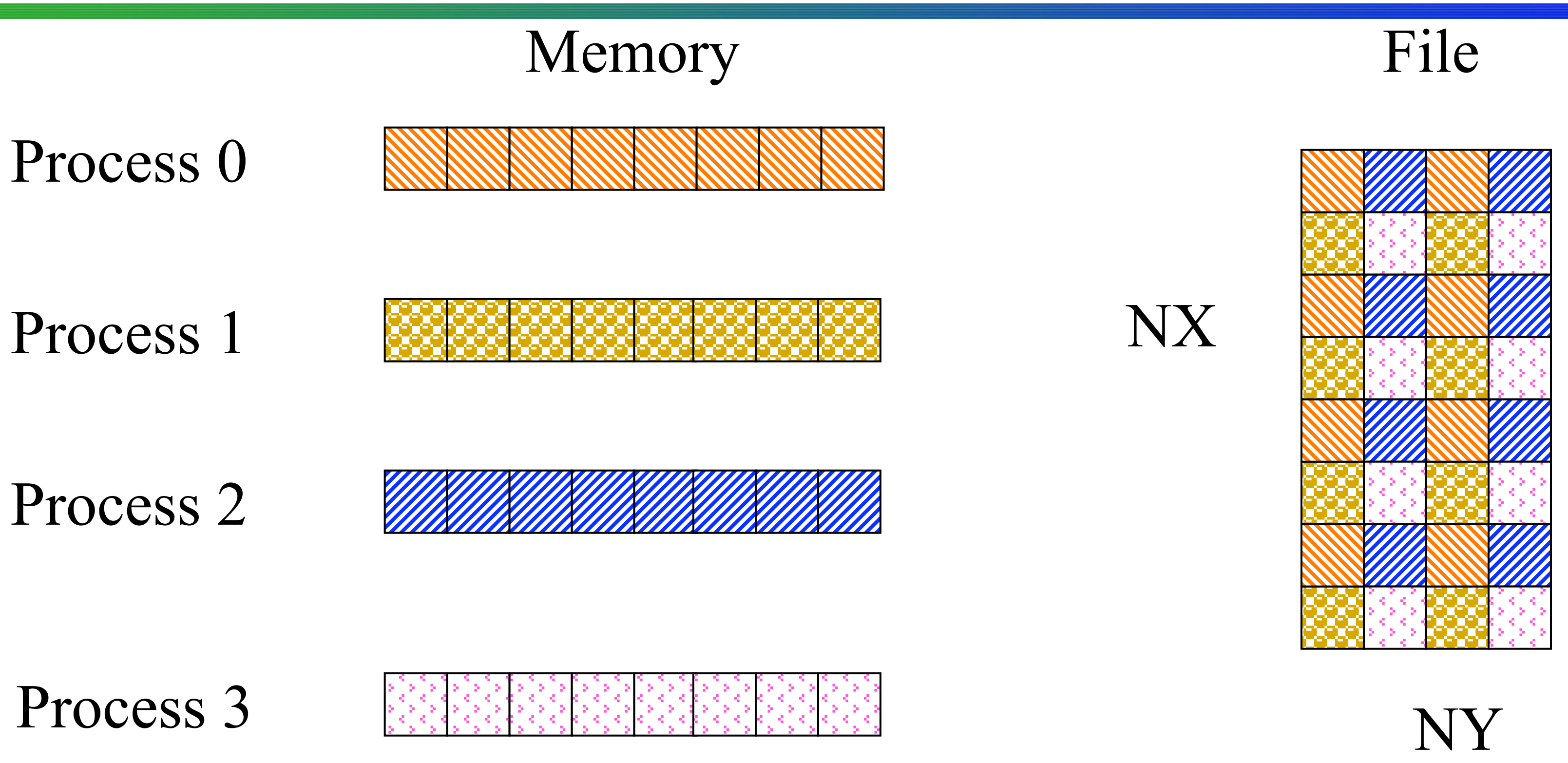
Example 3: *Writing dataset by pattern*



Writing by Pattern: *Output of h5dump*

```
HDF5 "grid_pattern.h5" {
GROUP "/" {
   DATASET "Dataset1" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 8, 4 ) / ( 8, 4 ) }
      DATA {
         1, 3, 1, 3,
         2, 4, 2, 4,
         1, 3, 1, 3,
         2, 4, 2, 4,
         1, 3, 1, 3,
         2, 4, 2, 4,
         1, 3, 1, 3,
         2, 4, 2, 4
      }
   }
}
```

Example 3: *Writing dataset by pattern*



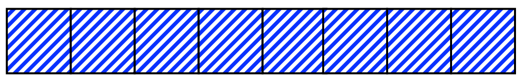
- More complicated pattern, describe data layout with 4 arrays
- offset[] - starting position
- stride[] - spacing to the next element
- count[] - how many times to write a contiguous block
- block[] - how many contiguous elements to write

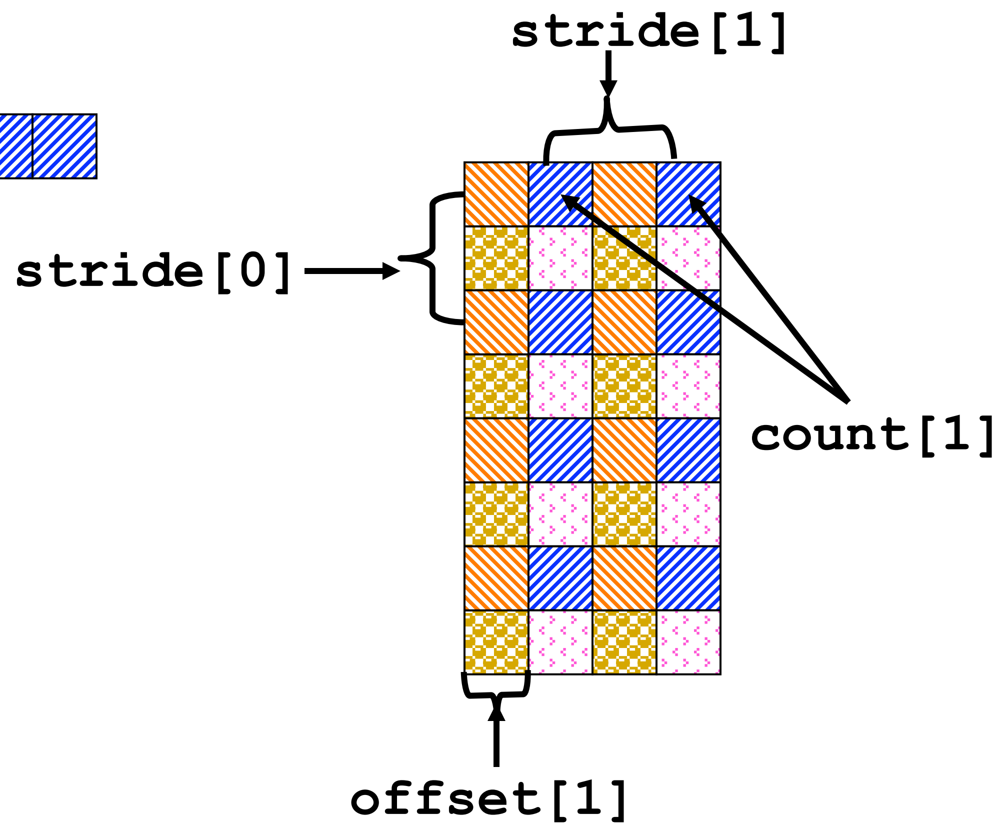
Example 3: Writing dataset by pattern

Memory

File

Process 2



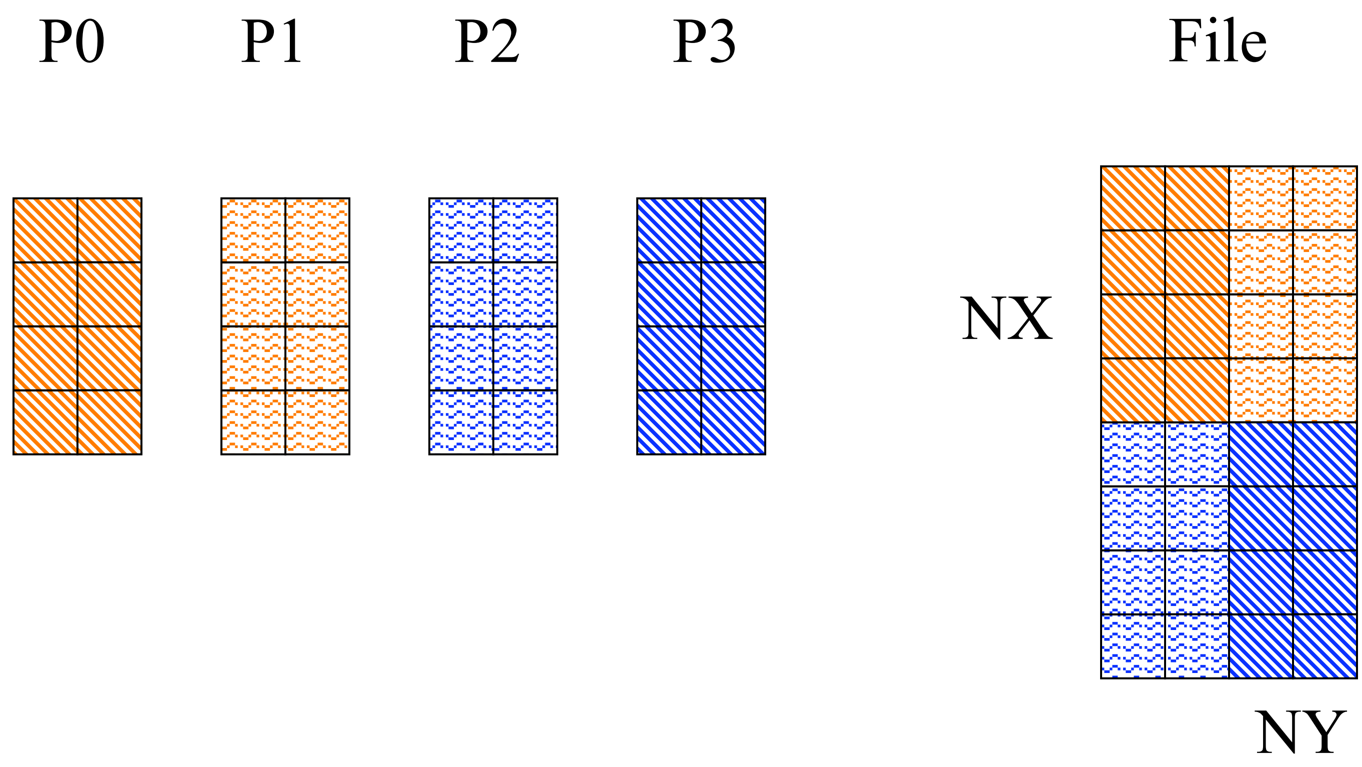



```
offset[0] = 0;
offset[1] = 1;
count[0]  = 4;
count[1]  = 2;
stride[0] = 2;
stride[1] = 2;
```

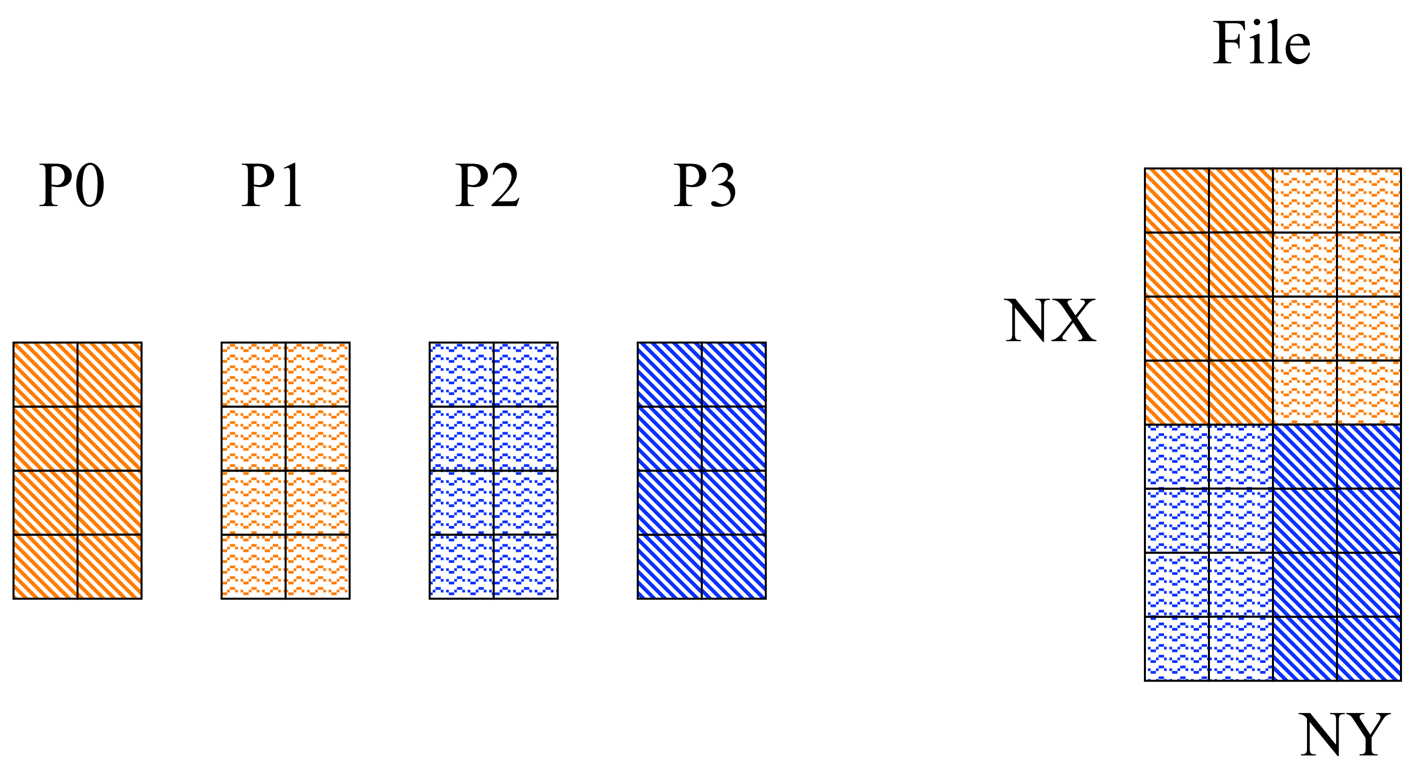
Example 3: Writing by pattern

```
/* Each process defines dataset in memory and
 * writes it to the hyperslab in the file.
 */
count[0] = 4;
count[1] = 2;
stride[0] = 2;
stride[1] = 2;
if(my_proc == 0) {
   offset[0] = 0;
   offset[1] = 0;
}
if(my_proc == 1) {
   offset[0] = 1;
   offset[1] = 0;
}
if(my_proc == 2) {
   offset[0] = 0;
   offset[1] = 1;
}
if(my_proc == 3) {
   offset[0] = 1;
   offset[1] = 1;
}
```

Example 4: Writing dataset by chunks



Example 4: Writing dataset by chunks



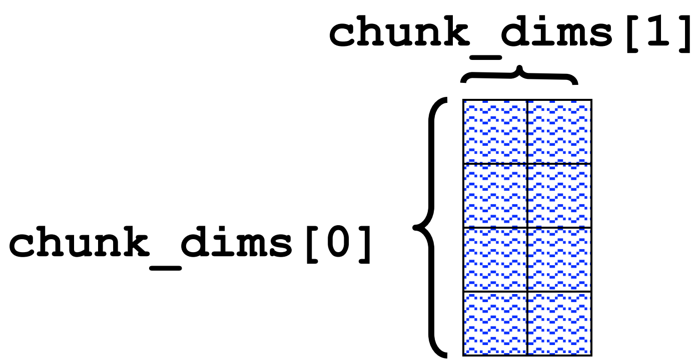
- More complicated pattern, describe data layout with 4 arrays
- offset[] - starting position
- stride[] - spacing to the next element
- count[] - how many times to write a contiguous block
- block[] - how many contiguous elements to write

Writing by Chunks: Output of h5dump

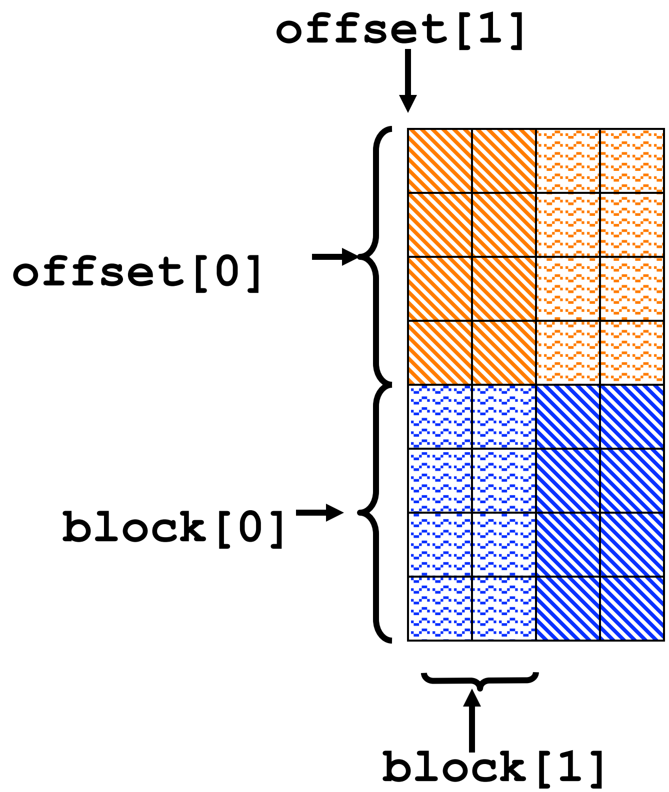
```
HDF5 "write_chunks.h5" {
GROUP "/" {
   DATASET "Dataset1" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 8, 4 ) / ( 8, 4 ) }
      DATA {
         1, 1, 2, 2,
         1, 1, 2, 2,
         1, 1, 2, 2,
         1, 1, 2, 2,
         3, 3, 4, 4,
         3, 3, 4, 4,
         3, 3, 4, 4,
         3, 3, 4, 4
      }
   }
}
```

Example 4: Writing dataset by chunks

Process 2: Memory



File




```
block[0]  = chunk_dims[0];
block[1]  = chunk_dims[1];
offset[0] = chunk_dims[0];
offset[1] = 0;
```

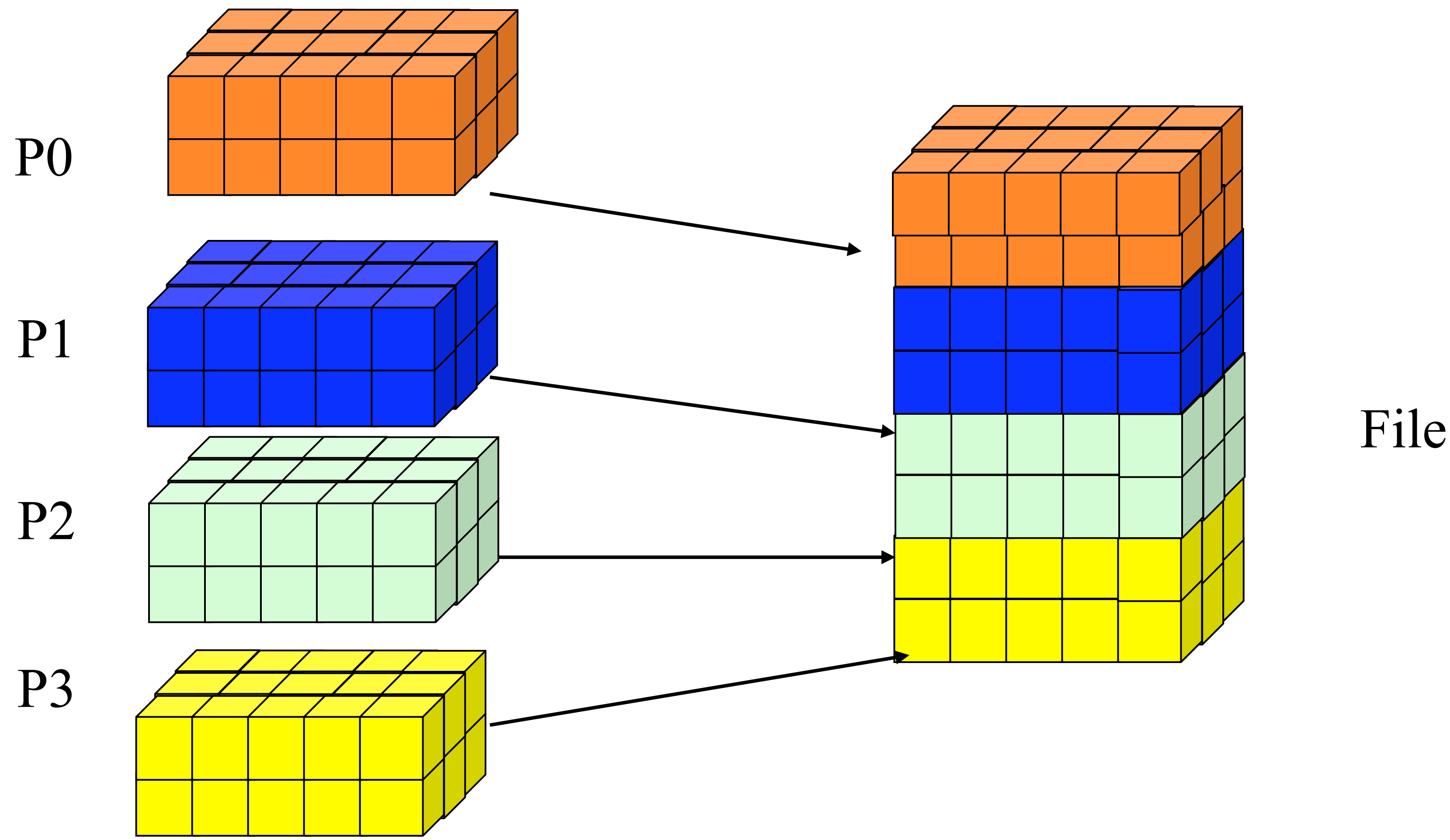
Example 4: Writing by chunks

```
count[0] = 1;
count[1] = 1 ;
stride[0] = 1;
stride[1] = 1;
block[0] = chunk_dims[0];
block[1] = chunk_dims[1];
if(mpi_rank == 0) {
   offset[0] = 0;
   offset[1] = 0;
}
if(mpi_rank == 1) {
   offset[0] = 0;
   offset[1] = chunk_dims[1];
}
if(mpi_rank == 2) {
   offset[0] = chunk_dims[0];
   offset[1] = 0;
}
if(mpi_rank == 3) {
   offset[0] = chunk_dims[0];
   offset[1] = chunk_dims[1];
}
```

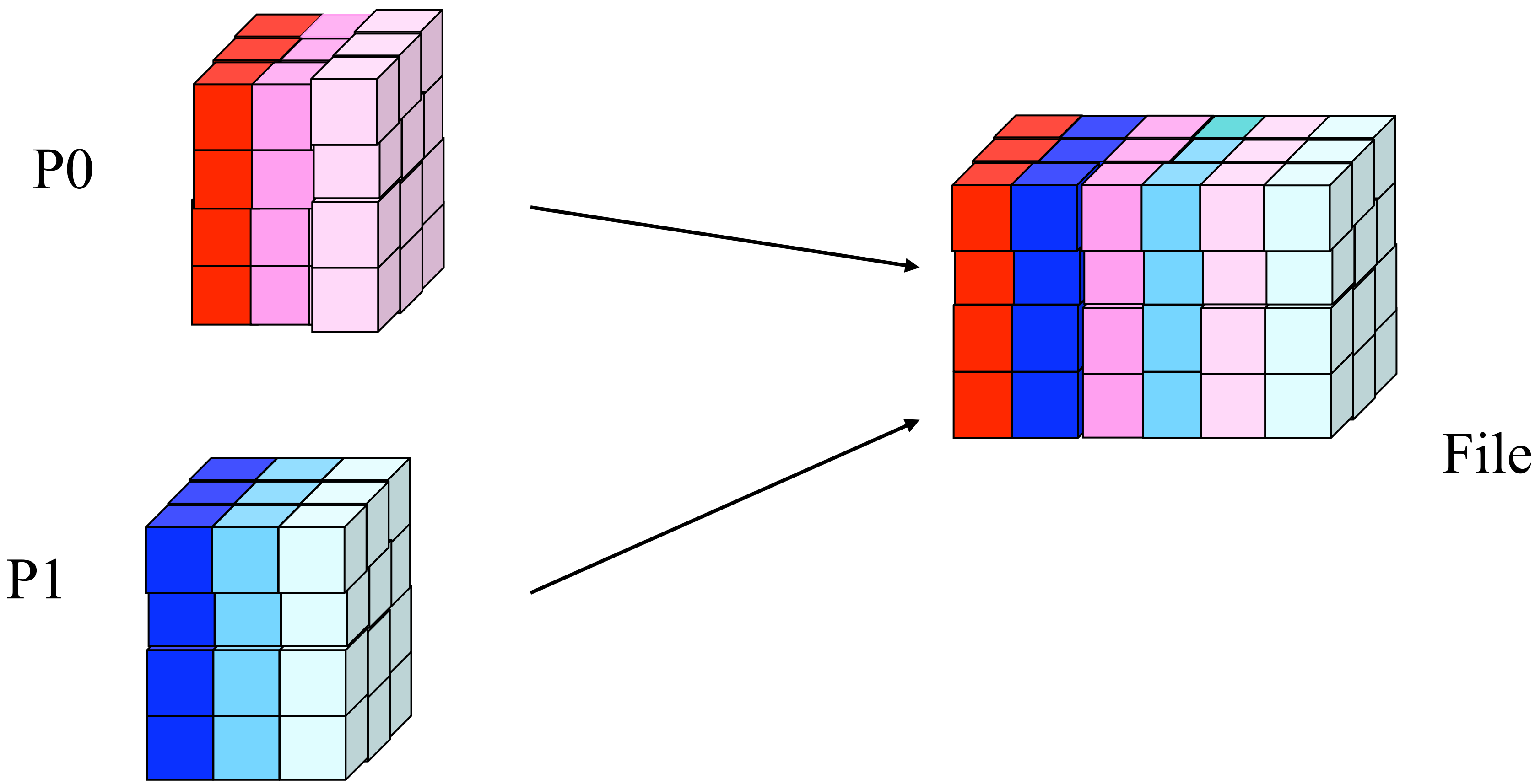
Fortran Tips and Tricks

- Fortran interfaces require an extra initialization and finalize call:
  - CALL h5open_f(error)
  - CALL h5close_f(error)
- Some differences in argument order to API from C version
- Remember Fortran arrays start at 1 not 0.
- Remember row and column order switched from C programs. See write_grid_rows.f90 for example

Problem 1: *Writing dataset by rows 3d*



Problem 2: *Writing dataset by cols 3d*



HDF5 Compile Scripts

- h5pcc – HDF5 C compiler command
- h5pfc – HDF5 F90 compiler command

To compile:

```
% h5pcc h5prog.c
% h5pfc h5prog.f90
```

Parallel HDF5 in a little more detail

MPI-IO vs. HDF5

- MPI-IO is an Input/Output API.
- It treats the data file as a “linear byte stream” and each MPI application needs to provide its own file view and data representations to interpret those bytes.
- All data stored are machine dependent except the “external32” representation.
- External32 is defined in Big Endianness
  - Little endian machines have to do the data conversion in both read or write operations.
  - 64bit sized data types may lose information.

MPI-IO vs. HDF5 Cont.

- HDF5 is a data management software.
- It stores the data and metadata according to the HDF5 data format definition.
  - HDF5 file is self-described.
  - Each machine can store the data in its own native representation for efficient I/O without loss of data precision.
  - Any necessary data representation conversion is done by the HDF5 library automatically.

Examples of PHDF5 API

- Examples of PHDF5 collective API
  - File operations: H5Fcreate, H5Fopen, H5Fclose
  - Objects creation: H5Dcreate, H5Dopen, H5Dclose
  - Objects structure: H5Dextend (increase dimension sizes)
- Array data transfer can be collective or independent
  - Dataset operations: H5Dwrite, H5Dread
  - Collectiveness is indicated by function parameters, not by function names as in MPI API

What Does PHDF5 Support ?

- After a file is opened by the processes of a communicator
  - All parts of file are accessible by all processes
  - All objects in the file are accessible by all processes
  - Multiple processes may write to the same data array
  - Each process may write to individual data array

Collective vs. Independent Calls

- MPI definition of collective call
  - All processes of the communicator must participate in the right order. E.g.,
    - Process1 Process2
    - call A(); call B(); call A(); call B(); **right**
    - call A(); call B(); call B(); call A(); **wrong**
- Independent means not collective
- Collective is not necessarily synchronous

Programming Restrictions

- Most PHDF5 APIs are collective
- PHDF5 opens a parallel file with a communicator
  - Returns a file-handle
  - Future access to the file via the file-handle
  - All processes must participate in collective PHDF5 APIs
  - Different files can be opened via different communicators

Programming model for creating and accessing a file

- HDF5 uses access template object (property list) to control the file access mechanism
- General model to access HDF5 file in parallel:
  - Setup MPI-IO access template (access property list)
  - Open File
  - Access Data
  - Close File

Setup MPI-IO access template

Each process of the MPI communicator creates an access template and sets it up with MPI parallel access information

C:

```
herr_t H5Pset_fapl_mpio(hid_t plist_id,
          MPI_Comm comm,  MPI_Info info);
```

F90:

```
h5pset_fapl_mpio_f(plist_id, comm, info)
integer(hid_t) :: plist_id
integer        :: comm, info
```

plist_id is a file access property list identifier

C Example Parallel File Create

```
comm = MPI_COMM_WORLD;
info = MPI_INFO_NULL;
/*
 * Initialize MPI
 */
MPI_Init(&argc, &argv);
/*
 * Set up file access property list for MPI-IO access
 */
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC,
    H5P_DEFAULT, plist_id);
/*
 * Close the file.
 */
H5Fclose(file_id);
MPI_Finalize();
```

F90 Example Parallel File Create

```
  comm = MPI_COMM_WORLD
  info = MPI_INFO_NULL
  CALL MPI_INIT(mpierror)
  !
  ! Initialize FORTRAN predefined datatypes
  CALL h5open_f(error)
  !
  ! Setup file access property list for MPI-IO access.
->CALL h5pcreate_f(H5P_FILE_ACCESS_F, plist_id, error)
->CALL h5pset_fapl_mpio_f(plist_id, comm, info, error)
  !
  ! Create the file collectively.
->CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id,
          error, access_prp = plist_id)
  !
  ! Close the file.
  CALL h5fclose_f(file_id, error)
  !
  ! Close FORTRAN interface
  CALL h5close_f(error)
  CALL MPI_FINALIZE(mpierror)
```

Creating and Opening Dataset

- All processes of the communicator open/close a dataset by a collective call
  - ✓C: H5Dcreate or H5Dopen; H5Dclose
  - ✓F90: h5dcreate_f or h5dopen_f; h5dclose_f
- All processes of the communicator must extend an unlimited dimension dataset before writing to it
  - ✓C: H5Dextend
  - ✓F90: h5dextend_f

C Example: Create Dataset

```
file_id = H5Fcreate(…);
/*
 * Create the dataspace for the dataset.
 */
dimsf[0] = NX;
dimsf[1] = NY;
filespace = H5Screate_simple(RANK, dimsf, NULL);

/*
 * Create the dataset with default properties collective.
 */
dset_id = H5Dcreate(file_id, "dataset1", H5T_NATIVE_INT,
                    filespace, H5P_DEFAULT);

H5Dclose(dset_id);
/*
 * Close the file.
 */
H5Fclose(file_id);
```

F90 Example: Create Dataset

```
CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id,
     error, access_prp = plist_id)
CALL h5screate_simple_f(rank, dimsf, filespace, error)
!
! Create the dataset with default properties.
!
CALL h5dcreate_f(file_id, "dataset1", H5T_NATIVE_INTEGER,
                 filespace, dset_id, error)
!
! Close the dataset.
CALL h5dclose_f(dset_id, error)
!
! Close the file.
CALL h5fclose_f(file_id, error)
```

Accessing a Dataset

- All processes that have opened dataset may do collective I/O
- Each process may do independent and arbitrary number of data I/O access calls
  - C: H5Dwrite and H5Dread
  - F90: h5dwrite_f and h5dread_f

Programming model for dataset access

- Create and set dataset transfer property
  - C: H5Pset_dxpl_mpio
    - H5FD_MPIO_COLLECTIVE
    - H5FD_MPIO_INDEPENDENT (default)
  - F90: h5pset_dxpl_mpio_f
    - H5FD_MPIO_COLLECTIVE_F
    - H5FD_MPIO_INDEPENDENT_F (default)
- Access dataset with the defined transfer property

C Example: Collective write

```
/*
 * Create property list for collective dataset write.
 */
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
```

F90 Example: Collective write

```
! Create property list for collective dataset write
!
CALL h5pcreate_f(H5P_DATASET_XFER_F, plist_id, error)
CALL h5pset_dxpl_mpio_f(plist_id, &
                        H5FD_MPIO_COLLECTIVE_F, error)

!
! Write the dataset collectively.
!
CALL h5dwrite_f(dset_id, H5T_NATIVE_INTEGER, data, &
                error, &
                file_space_id = filespace, &
                mem_space_id = memspace, &
                xfer_prp = plist_id)
```

Writing and Reading Hyperslabs

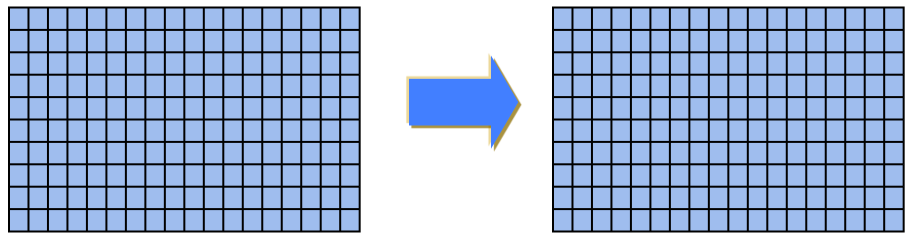
- Distributed memory model: data is split among processes
- PHDF5 uses HDF5 hyperslab model
- Each process defines memory and file hyperslabs
- Each process executes partial write/read call
  - Collective calls
  - Independent calls

HDF5 Properties

- Properties (also known as Property Lists) are characteristics of HDF5 objects that can be modified
- Default properties handle most needs
- By changing properties one can take advantage of the more powerful features in HDF5

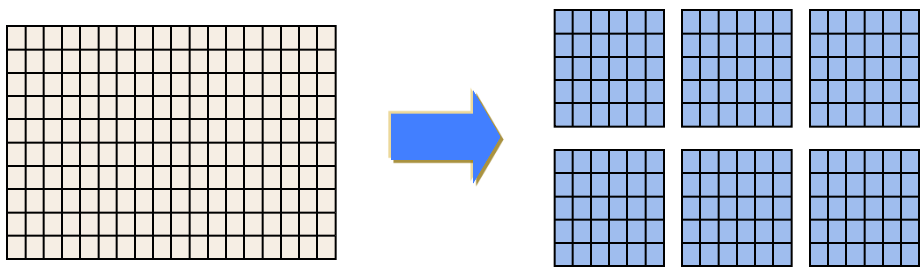
Storage Properties

Contiguous (default)



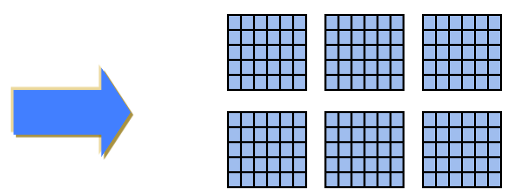
Data elements stored physically adjacent to each other

Chunked



Better access time for subsets; extensible

Chunked & Compressed



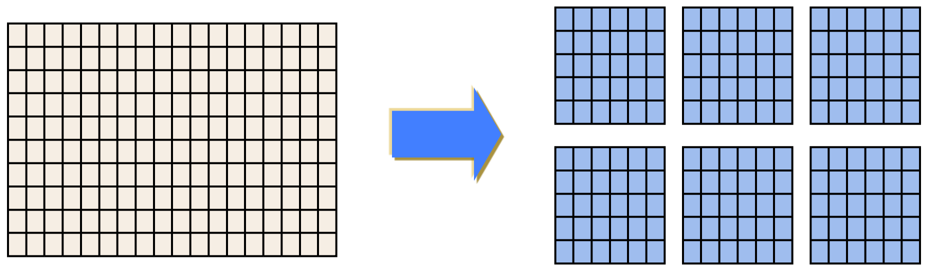
Improves storage efficiency, transmission speed

HDF5 Attributes (optional)

- An HDF5 attribute has a name and a value
- Attributes typically contain user metadata
- Attributes may be associated with
  - HDF5 groups
  - HDF5 datasets
  - HDF5 named datatypes
- An attribute's value is described by a datatype and a dataspace
- Attributes are analogous to datasets except…
  - they are NOT extensible
  - they do NOT support compression or partial I/O

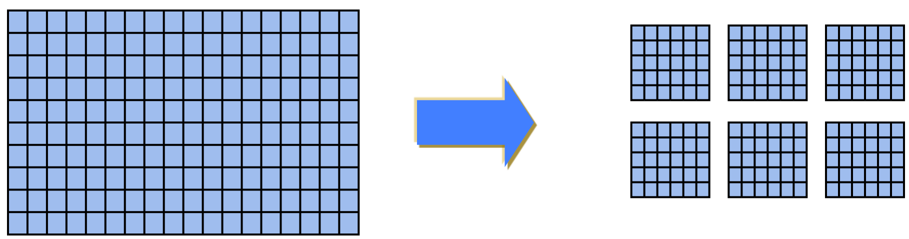
Dataset Creation Property List

Chunked



Better access time for subsets; extensible

Chunked & Compressed



Improves storage efficiency, transmission speed

H5P_DEFAULT: contiguous

Dataset creation property list: **information on how to organize data in storage.**

Steps to Create a Group

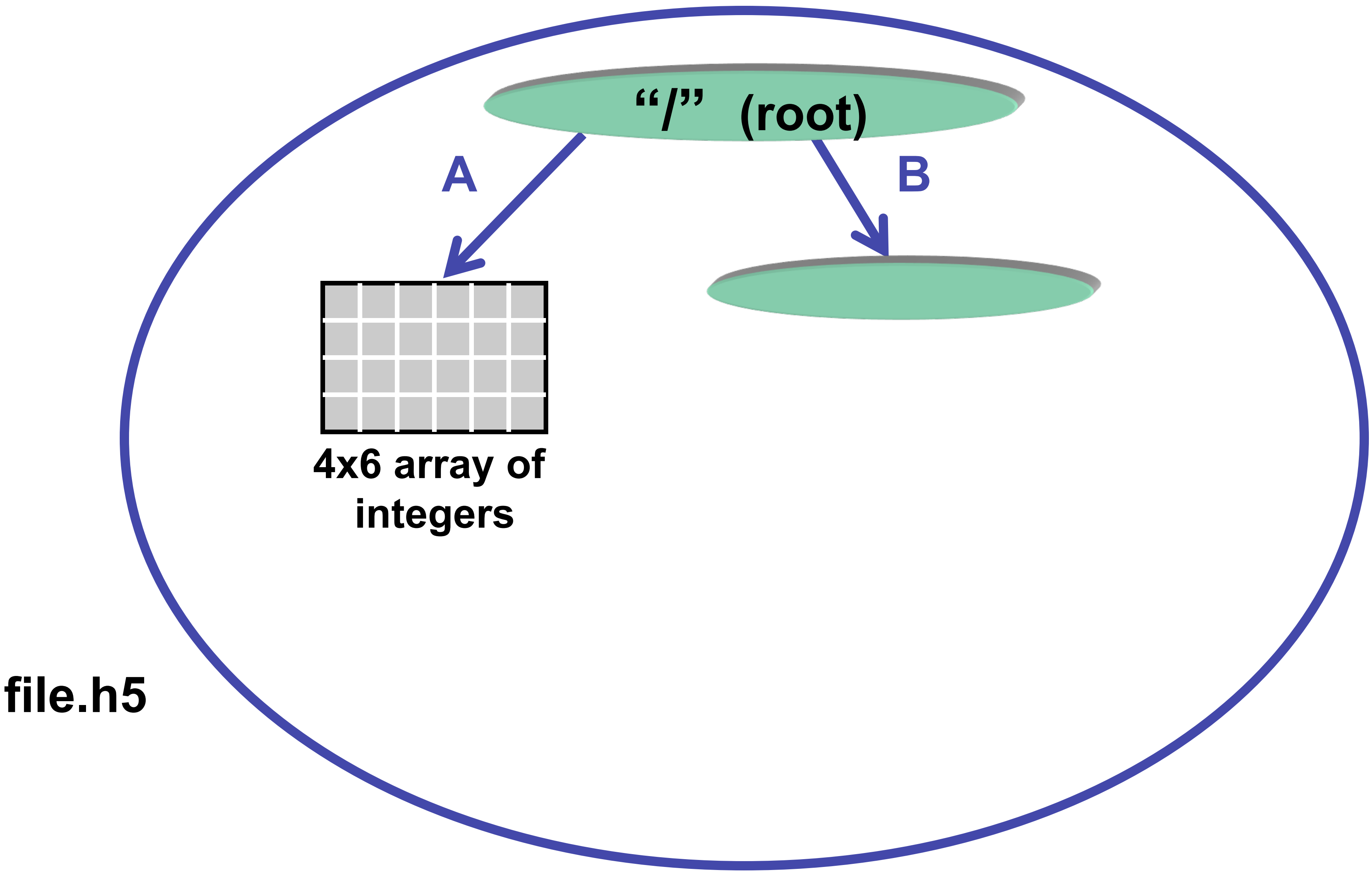
1. Decide where to put it – “root group”
   - Obtain location ID

2. Define properties or use H5P_DEFAULT

5. Create group in file.

4. Close the group.

Example: Create a Group



Code: Create a Group

```
hid_t file_id, group_id;
...
/* Open “file.h5” */
file_id = H5Fopen (“file.h5”, H5F_ACC_RDWR,
                   H5P_DEFAULT);

/* Create group "/B" in file. */
group_id = H5Gcreate (file_id,"B", H5P_DEFAULT,
                      H5P_DEFAULT, H5P_DEFAULT);

/* Close group and file. */
status = H5Gclose (group_id);
status = H5Fclose (file_id);
```

Intermediate Parallel HDF5

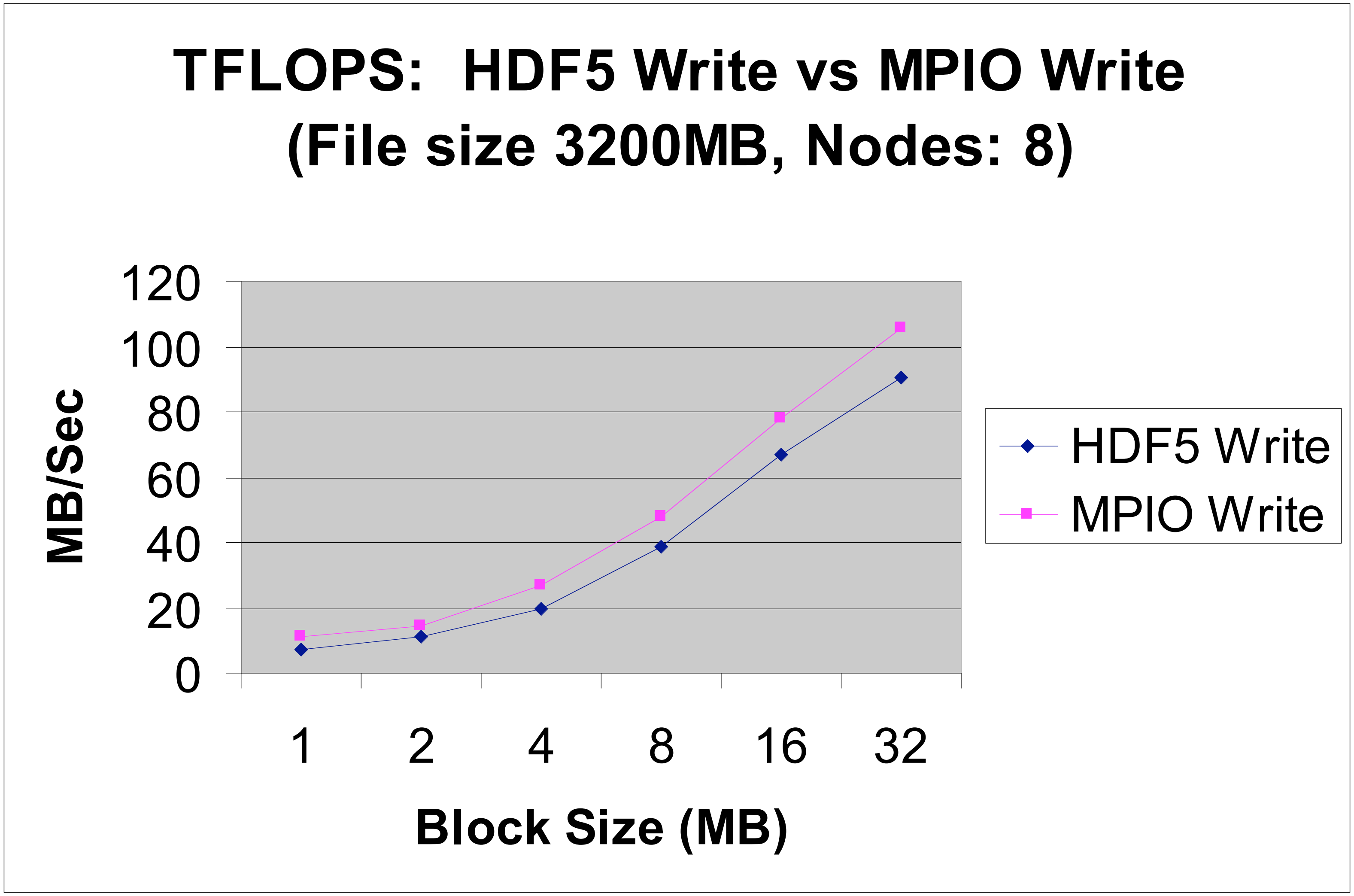
Outline

- Performance
- Parallel tools

My PHDF5 Application I/O is slow

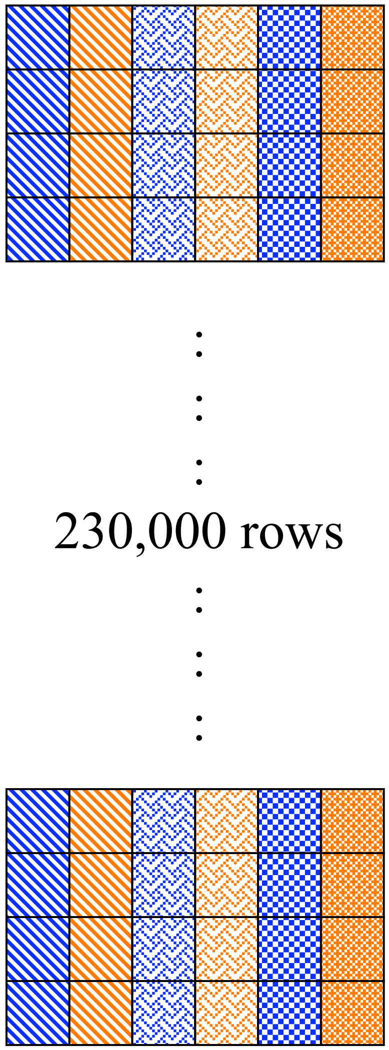
- If my application I/O performance is slow, what can I do?
  - Use larger I/O data sizes
  - Independent vs. Collective I/O
  - Specific I/O system hints
  - Increase Parallel File System capacity

Write Speed vs. Block Size



Independent Vs Collective Access

- User reported Independent data transfer mode was much slower than the Collective data transfer mode
- Data array was tall and thin: 230,000 rows by 6 columns

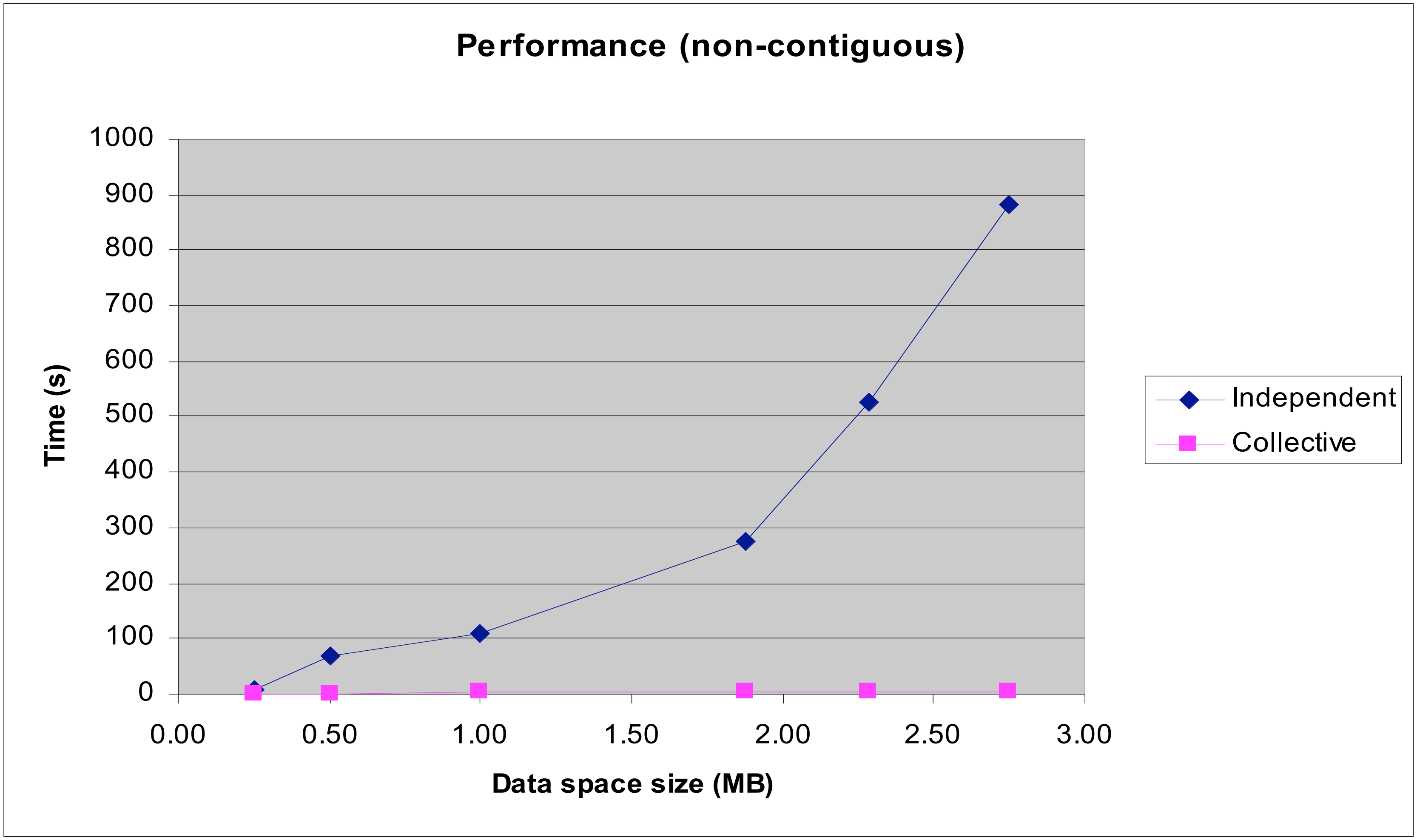


Independent vs. Collective write

6 processes, IBM p-690, AIX, GPFS

# of Rows	Data Size (MB)	Independent (Sec.)	Collective (Sec.)
16384	0.25	8.26	1.72
32768	0.50	65.12	1.80
65536	1.00	108.20	2.68
122918	1.88	276.57	3.11
150000	2.29	528.15	3.63
180300	2.75	881.39	4.12

Independent vs. Collective write (cont.)



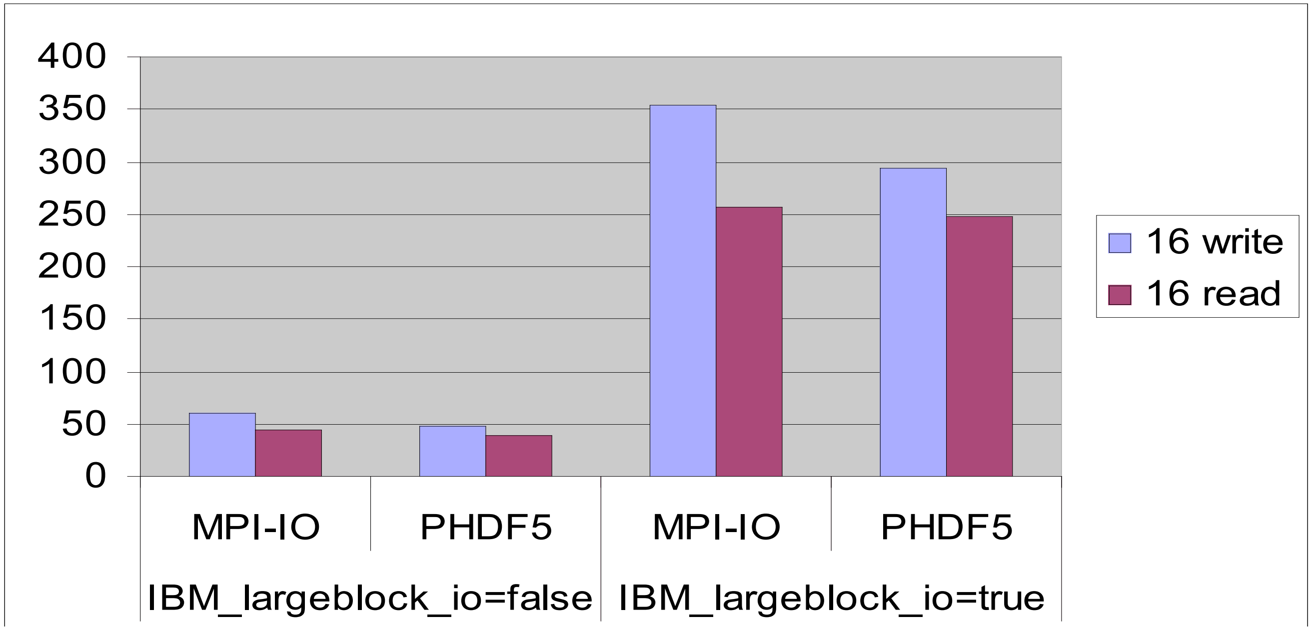
Effects of I/O Hints: IBM_largeblock_io

- GPFS at LLNL Blue
  - 4 nodes, 16 tasks
  - Total data size 1024MB
  - I/O buffer size 1MB

		IBM_largeblock_io=false		IBM_largeblock_io=true	
Tasks		MPI-IO	PHDF5	MPI-IO	PHDF5
16	write (MB/S)	60	48	354	294
16	read (MB/S)	44	39	256	248

Effects of I/O Hints: IBM_largeblock_io

- GPFS at LLNL ASCI Blue machine
  - 4 nodes, 16 tasks
  - Total data size 1024MB
  - I/O buffer size 1MB



Parallel Tools

- ph5diff
  - Parallel version of the h5diff tool
- h5perf
  - Performance measuring tools showing I/O performance for different I/O API

ph5diff

- An parallel version of the h5diff tool
  - Supports all features of h5diff
  - An MPI parallel tool
  - Manager process (proc 0)
    - coordinates each the remaining processes (workers) to “diff” one dataset at a time;
    - collects any output from each worker and prints them out.
  - Works best if there are many datasets in the files with few differences.
  - Available in v1.8.

h5perf

- An I/O performance measurement tool
- Test 3 File I/O API
  - POSIX I/O (open/write/read/close…)
  - MPIO (MPI_File_{open,write,read,close})
  - PHDF5
    - H5Pset_fapl_mpio (using MPI-IO)
    - H5Pset_fapl_mpiposix (using POSIX I/O)

h5perf: Some features

- Check (-c) verify data correctness
- Added 2-D chunk patterns in v1.8
- -h shows the help page.

h5perf: example output 1/3

```
% mpirun -np 4 h5perf
Number of processors = 4
    Transfer Buffer Size: 131072 bytes, File size: 1.00 MBs
      # of files: 1, # of datasets: 1, dataset size: 1.00 MBs
        IO API = POSIX
            Write (1 iteration(s)):
                Maximum Throughput:  18.75 MB/s
                Average Throughput:  18.75 MB/s
                Minimum Throughput:  18.75 MB/s
            Write Open-Close (1 iteration(s)):
                Maximum Throughput:  10.79 MB/s
                Average Throughput:  10.79 MB/s
                Minimum Throughput:  10.79 MB/s
            Read (1 iteration(s)):
                Maximum Throughput: 2241.74 MB/s
                Average Throughput: 2241.74 MB/s
                Minimum Throughput: 2241.74 MB/s
            Read Open-Close (1 iteration(s)):
                Maximum Throughput: 756.41 MB/s
                Average Throughput: 756.41 MB/s
                Minimum Throughput: 756.41 MB/s
```

h5perf: example output 2/3

```
% mpirun -np 4 h5perf
…
        IO API = MPIO
            Write (1 iteration(s)):
                Maximum Throughput: 611.95 MB/s
                Average Throughput: 611.95 MB/s
                Minimum Throughput: 611.95 MB/s
            Write Open-Close (1 iteration(s)):
                Maximum Throughput:  16.89 MB/s
                Average Throughput:  16.89 MB/s
                Minimum Throughput:  16.89 MB/s
            Read (1 iteration(s)):
                Maximum Throughput: 421.75 MB/s
                Average Throughput: 421.75 MB/s
                Minimum Throughput: 421.75 MB/s
            Read Open-Close (1 iteration(s)):
                Maximum Throughput: 109.22 MB/s
                Average Throughput: 109.22 MB/s
                Minimum Throughput: 109.22 MB/s
```

h5perf: example output 3/3

```
% mpirun -np 4 h5perf
…
        IO API = PHDF5 (w/MPI-I/O driver)
            Write (1 iteration(s)):
                Maximum Throughput: 304.40 MB/s
                Average Throughput: 304.40 MB/s
                Minimum Throughput: 304.40 MB/s
            Write Open-Close (1 iteration(s)):
                Maximum Throughput:  15.14 MB/s
                Average Throughput:  15.14 MB/s
                Minimum Throughput:  15.14 MB/s
            Read (1 iteration(s)):
                Maximum Throughput: 1718.27 MB/s
                Average Throughput: 1718.27 MB/s
                Minimum Throughput: 1718.27 MB/s
            Read Open-Close (1 iteration(s)):
                Maximum Throughput:  78.06 MB/s
                Average Throughput:  78.06 MB/s
                Minimum Throughput:  78.06 MB/s
    Transfer Buffer Size: 262144 bytes, File size: 1.00 MBs
      # of files: 1, # of datasets: 1, dataset size: 1.00 MBs
```

Useful Parallel HDF Links

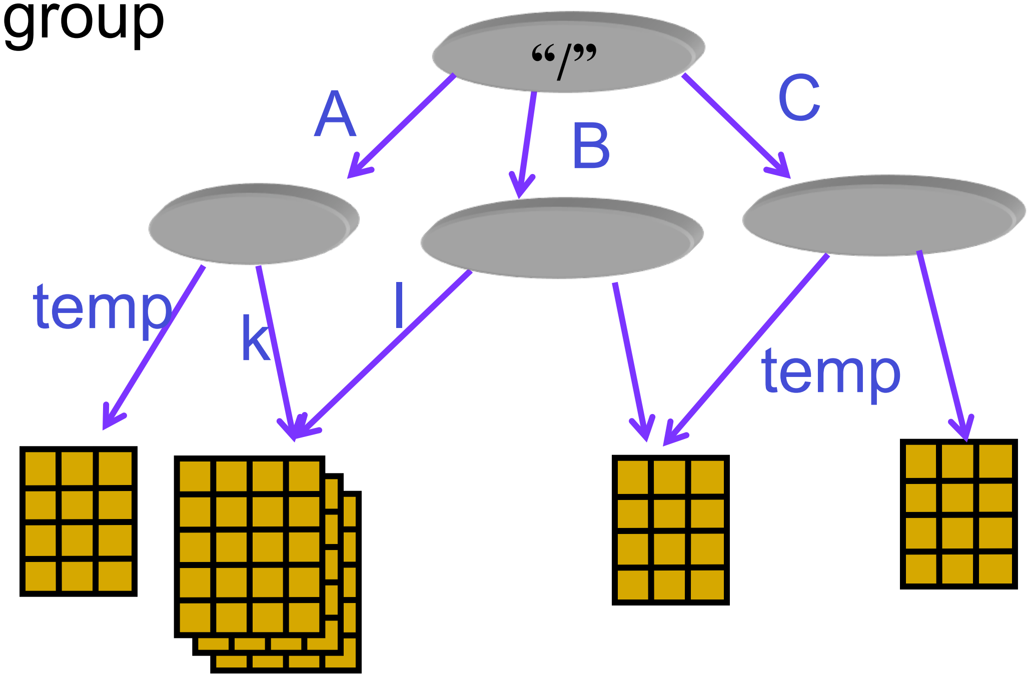
- Parallel HDF information site
  http://www.hdfgroup.org/HDF5/PHDF5/
- Parallel HDF5 tutorial available at
  http://www.hdfgroup.org/HDF5/Tutor/
- HDF Help email address
  help@hdfgroup.org

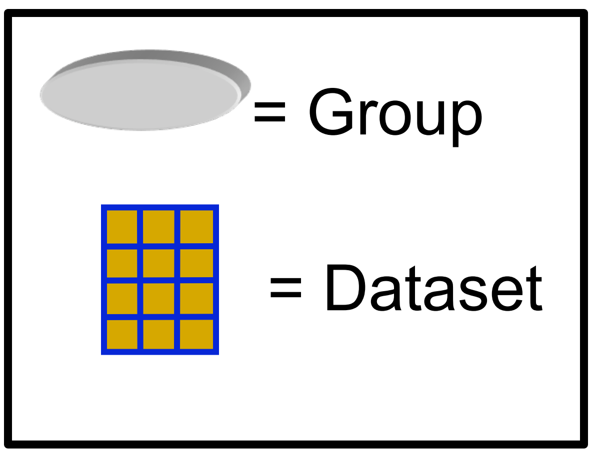
Questions?

End of Part IV

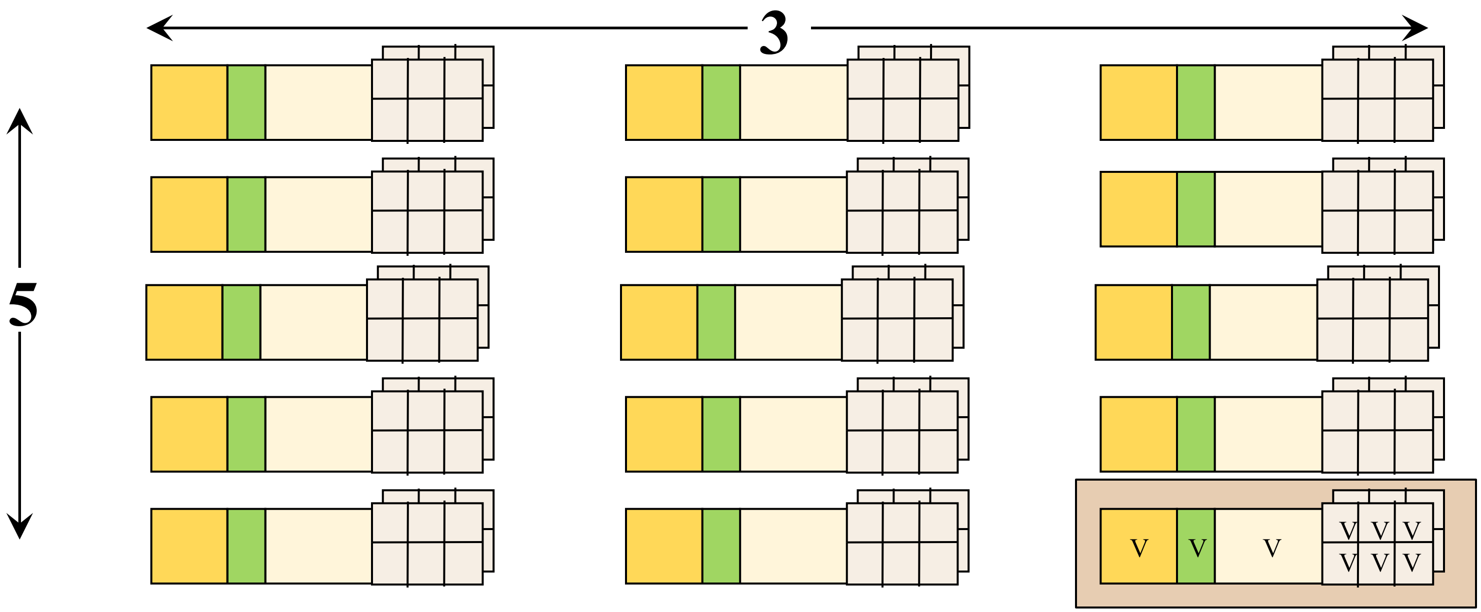
HDF5 Groups

- Used to organize collections
- Every file starts with a root group
- Similar to UNIX directories
- Path to object defines it
- Objects can be shared:
  /A/k and /B/l are the same

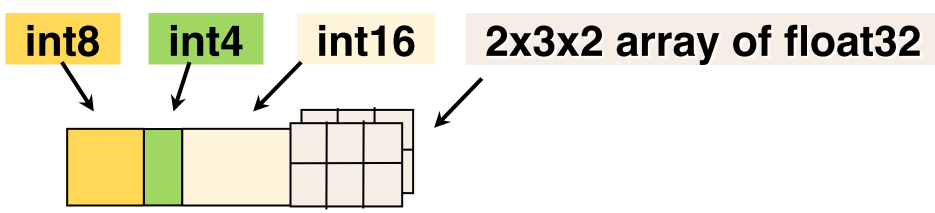




HDF5 Dataset with Compound Datatype



Compound Datatype:



Dataspace: **Rank = 2**

Dimensions = 5 x 3

Link Creation/Dataset Access Properties

- Link Creation:
  - Creating intermediate groups

- Dataset Access:
  - Retrieve the raw data chunk cache parameters

Group Properties

- Link Creation
  - Creating intermediate groups
- Group Creation
  - Creation order tracking and indexing for links in a group.
  - Set Number of links and length of link names in a group.
- Group Access (not used)

Compile option: -show

-show: displays the compiler commands and options without executing them

```
% h5cc -show Sample_c.c
```

Will show the correct paths and libraries used by the installed HDF5 library.

Will show the correct flags to specify when building an application with that HDF5 library.

Other General HDF5 Slides

Help

The HDF Group Page: http://hdfgroup.org/

HDF5 Home Page: http://hdfgroup.org/HDF5/

HDF Helpdesk: help@hdfgroup.org

HDF Mailing Lists: http://hdfgroup.org/services/support.html

HDF5 is designed …

- for high volume and/or complex data
- for every size and type of system (portable)
- for flexible, efficient storage and I/O
- to enable applications to evolve in their use of HDF5 and to accommodate new models
- to support long-term data preservation

HDF5 Home Page

HDF5 home page: http://hdfgroup.org/HDF5/

- Two releases: HDF5 1.8 and HDF5 1.6

HDF5 source code:

- Written in C, and includes optional C++, Fortran 90 APIs, and High Level APIs
- Contains command-line utilities (h5dump, h5repack, h5diff, ..) and compile scripts

HDF pre-built binaries:

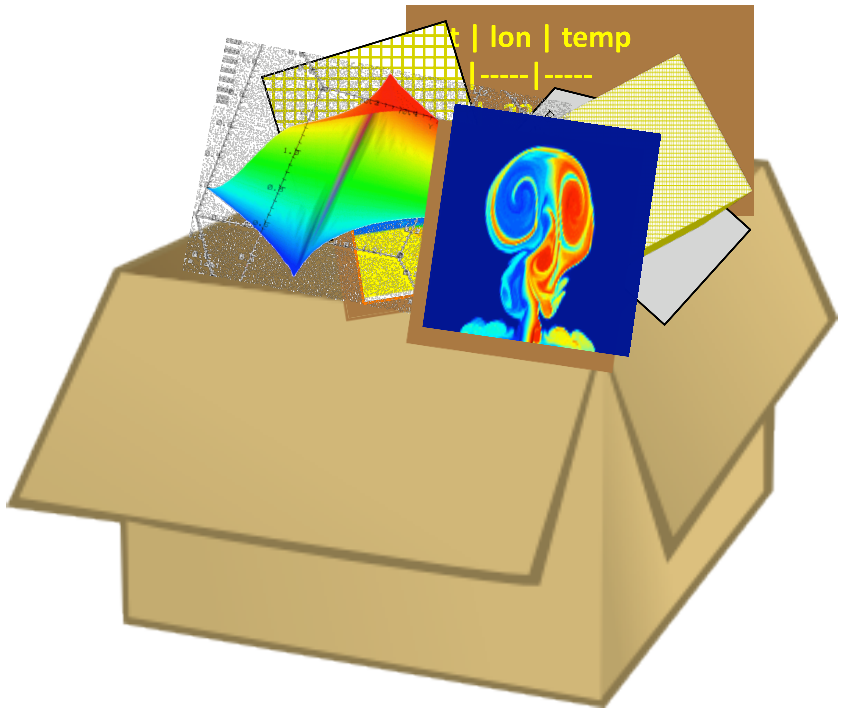
- When possible, include C, C++, F90, and High Level libraries. Check ./lib/libhdf5.settings file.
- Built with and require the SZIP and ZLIB external libraries

HDF5 Technology

- HDF5 (Abstract) Data Model
  - Defines the “building blocks” for data organization and specification
  - Files, Groups, Datasets, Attributes, Datatypes, Dataspaces, …
- HDF5 Library (C, Fortran 90, C++ APIs)
  - Also Java Language Interface and High Level Libraries
- HDF5 Binary File Format
  - Bit-level organization of HDF5 file
  - Defined by HDF5 File Format Specification
- Tools For Accessing Data in HDF5 Format
  - h5dump, h5repack, HDFView, …

HDF5 File

An HDF5 file is a **container** that holds data objects.



HDF5 Datasets

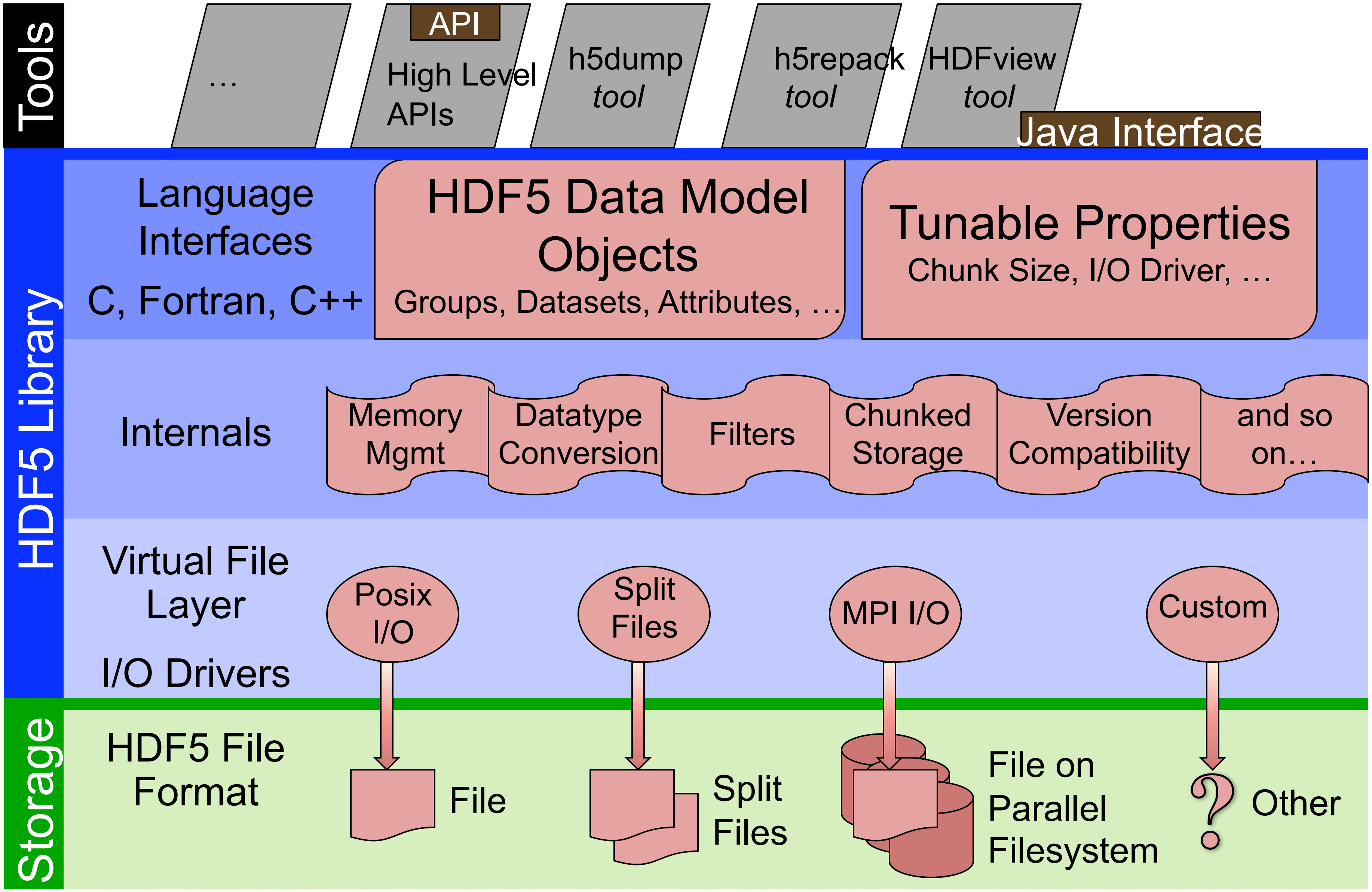
HDF5 Datasets organize and contain your "raw data values". They consist of:

- Your raw data
- Metadata describing the data:
  - The information to interpret the data (Datatype)
  - The information to describe the logical layout of the data elements (Dataspace)
  - Characteristics of the data (Properties)
  - Additional optional information that describes the data (Attributes)

HDF5 Abstract Data Model Summary

- The Objects in the Data Model are the “building blocks” for data organization and specification
- Files, **Groups**, Links, **Datasets**, Datatypes, Dataspaces, Attributes, …
- Projects using HDF5 “map” their data concepts to these HDF5 Objects

HDF5 Software Layers & Storage



Useful Tools For New Users

h5dump:

Tool to “dump” or display contents of HDF5 files

h5pcc,, h5pfc:

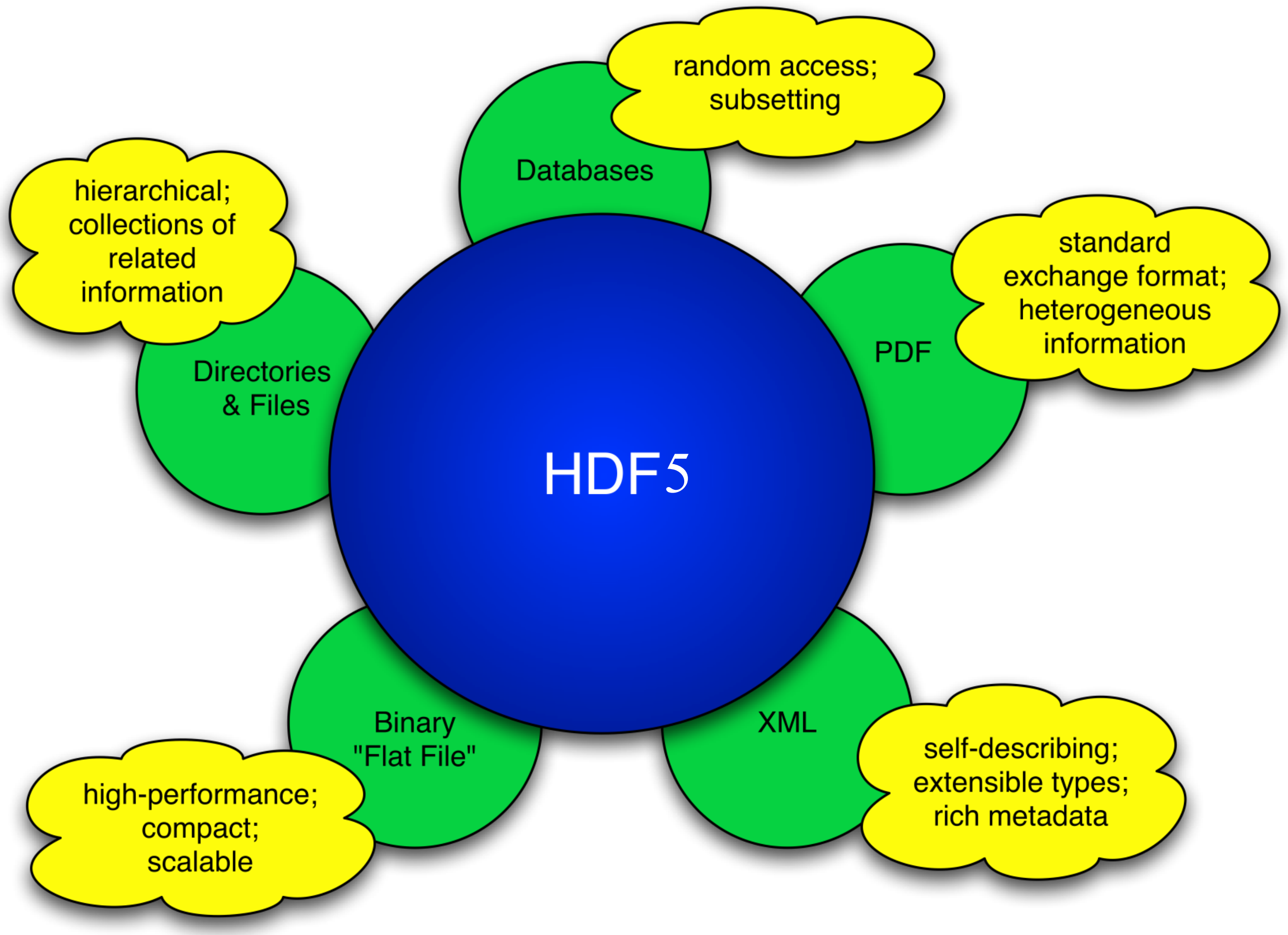
Scripts to compile applications

HDFView:

Java browser to view HDF4 and HDF5 files

http://www.hdfgroup.org/hdf-java-html/hdfview/

HDF5 is like…


HDF5
Databases
random access; subsetting
Directories & Files
hierarchical; collections of related information
PDF
standard exchange format; heterogeneous information
Binary "Flat File"
high-performance; compact; scalable
XML
self-describing; extensible types; rich metadata

h5dump Utility

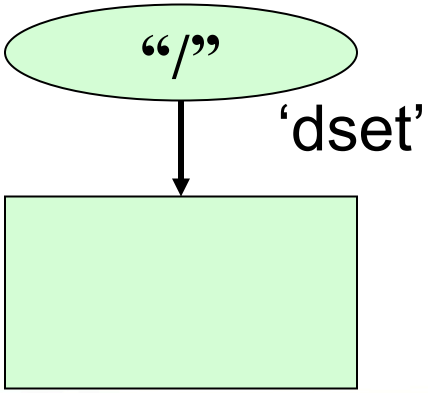
h5dump [options] [file]

-H, --header	Display header only – no data
-d <names>	Display the specified dataset(s).
-g <names>	Display the specified group(s) and all members.
-p	Display properties.

<*names*> is one or more appropriate object names.

Example of h5dump Output

```
HDF5 "dset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE { H5T_STD_I32BE }
      DATASPACE { SIMPLE ( 4, 6 ) / ( 4, 6 ) }
      DATA {
         1, 2, 3, 4, 5, 6,
         7, 8, 9, 10, 11, 12,
         13, 14, 15, 16, 17, 18,
         19, 20, 21, 22, 23, 24
      }
   }
}
}
```



Pre-defined Native Datatypes

Examples of predefined native types in C:

H5T_NATIVE_INT	(int)
H5T_NATIVE_FLOAT	(float)
H5T_NATIVE_UINT	(unsigned int)
H5T_NATIVE_LONG	(long)
H5T_NATIVE_CHAR	(char)

NOTE: Memory types.
Different for each machine.
Used for reading/writing.

Other Common Functions

DataSpaces:	H5Sselect_hyperslab H5Sselect_elements H5Dget_space
Groups:	H5Gcreate, H5Gopen, H5Gclose
Attributes:	H5Acreate, H5Aopen_name, H5Aclose, H5Aread, H5Awrite
Property lists:	H5Pcreate, H5Pclose H5Pset_chunk, H5Pset_deflate

HDF = Hierarchical Data Format

HDF5 is the *second* HDF format

- Development started in 1996
- First release was in 1998

HDF4 is the *first* HDF format

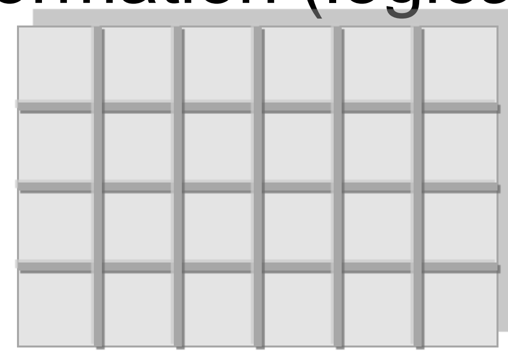
- Originally called HDF
- Development started in 1987
- Still supported by The HDF Group

HDF5 Dataspaces

Two roles:

Dataspace contains spatial information (logical layout) about a dataset stored in a file

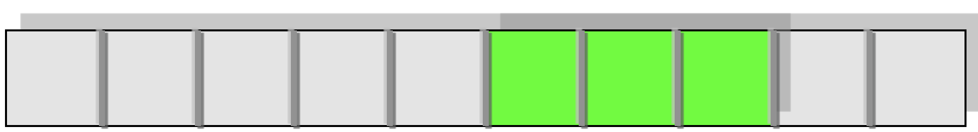
- Rank and dimensions
- Permanent part of dataset definition



Rank = 2

Dimensions = 4x6

Subsets: Dataspace describes application's data buffer and data elements participating in I/O



Rank = 1

Dimension = 10