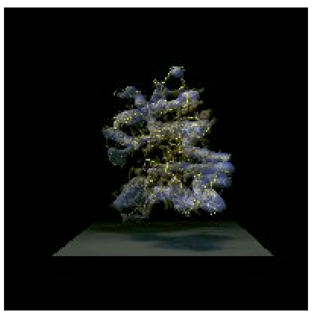
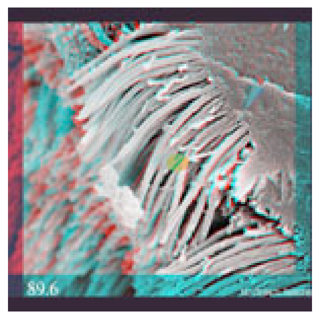
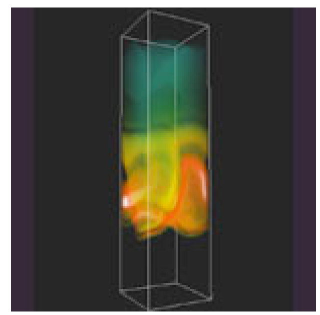
Using HDF5 for Scientific Data Analysis





NERSC Visualization Group

Before We Get Started
Glossary of Terms

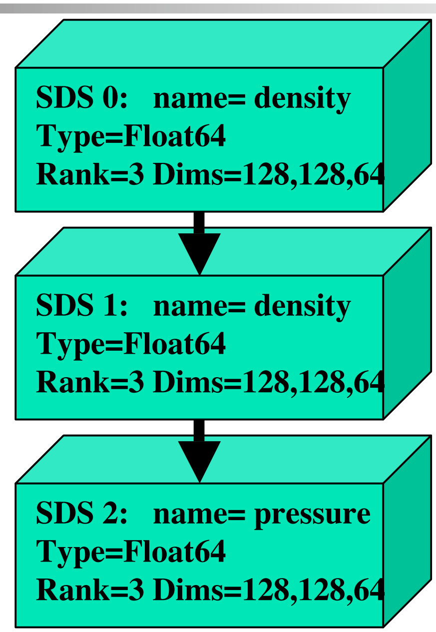
- Data
  - The raw information expressed in numerical form
- Metadata
  - Ancillary information about your data
- Attribute
  - Key/Value pair for accessing metadata
- Data Model / Schema
  - The choice of metadata and hierarchical organization of data to express higher-level relationships and features of a dataset.
- Self-Describing File
  - A file format that embeds explicit/queryable information about the data model
- Parallel I/O
  - Access to a single logical file image across all tasks of a parallel application that scales with the number of tasks.

History (HDF 1-4)

- Architecture Independent self-describing binary file format
  - Well-defined APIs for specific data schemas (RIG, RIS, SDS)
  - Support for wide variety of binary FP formats (Cray Float, Convex Float, DEC Float, IEEE Float, EDBIC)
  - C and Fortran bindings
  - Very limited support for parallelism (CM-HDF, EOS-HDF/PANDA)
  - Not thread-safe
- Relationship with Unidata NetCDF
  - Similar data model for HDF Scientific Data Sets (SDS)
  - Interoperable with NetCDF

HDF4 SDS data schema

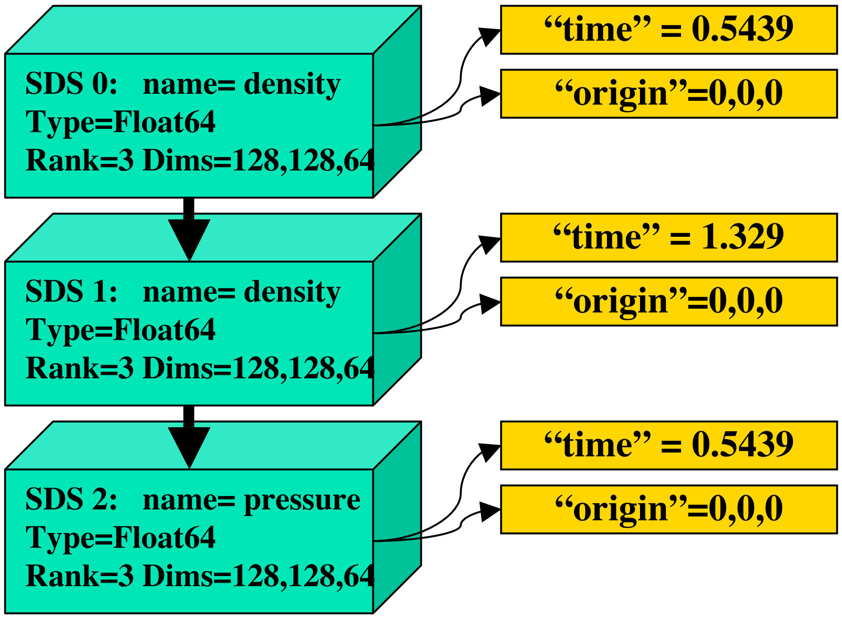
- Datasets
  - Name
  - Datatype
  - Rank,Dims



Datasets are inserted sequentially to the file

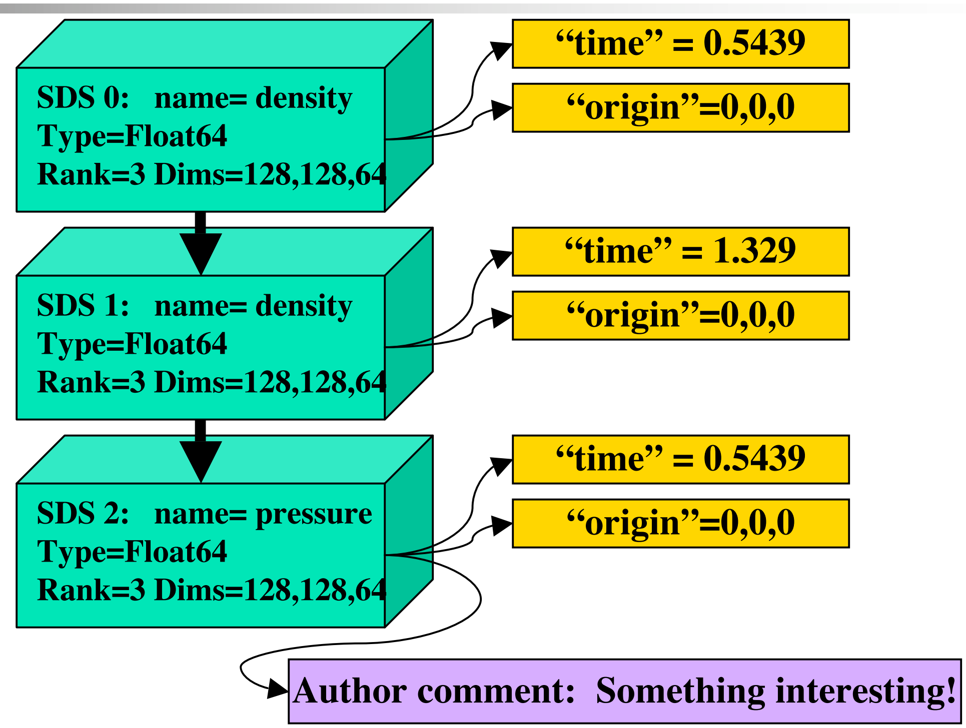
HDF4 SDS data schema

- Datasets
  - Name
  - Datatype
  - Rank,Dims
- Attributes
  - Key/value pair
  - DataType and length



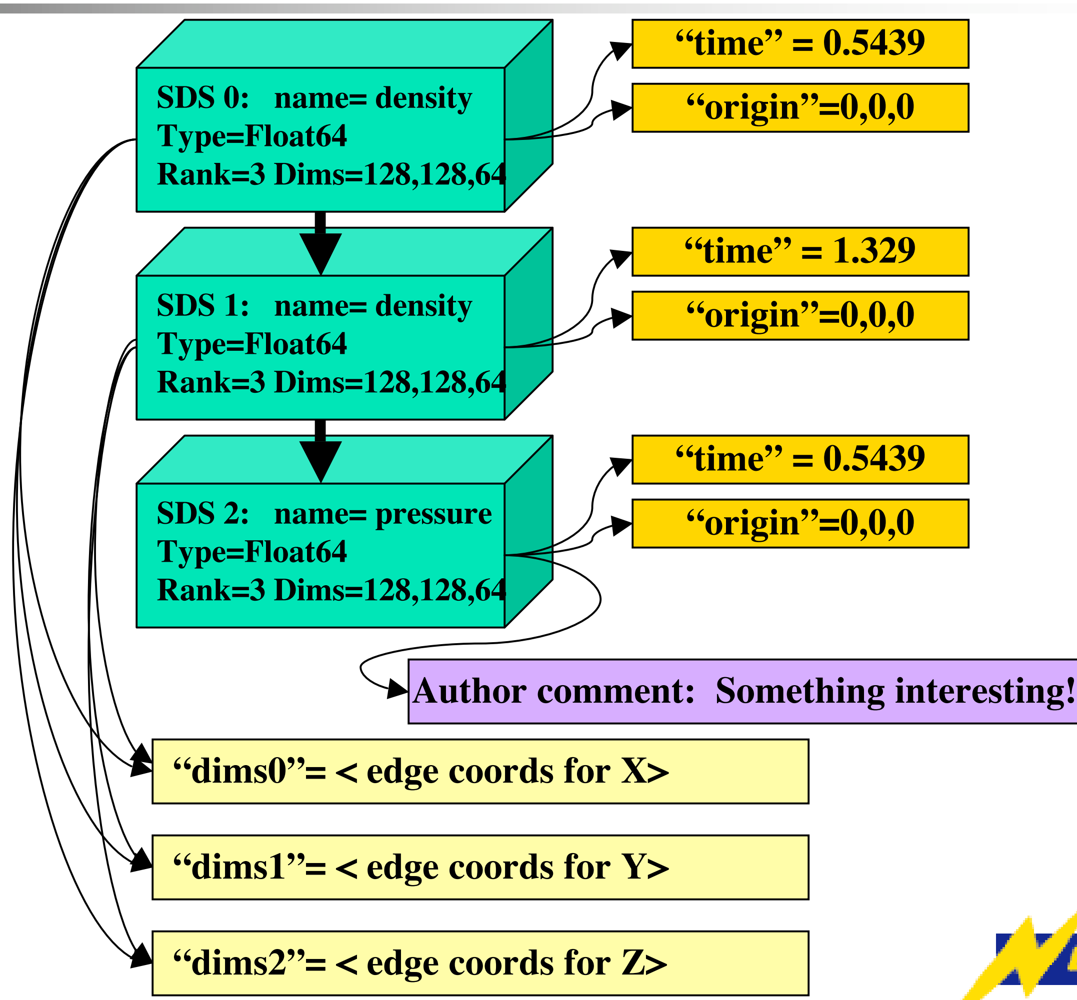
HDF4 SDS data schema

- Datasets
  - Name
  - Datatype
  - Rank,Dims
- Attributes
  - Key/value pair
  - DataType and length
- Annotations
  - Freeform text
  - String Termination



HDF4 SDS data schema

- Datasets
  - Name
  - Datatype
  - Rank,Dims
- Attributes
  - Key/value pair
  - DataType and length
- Annotations
  - Freeform text
  - String Termination
- Dimensions
  - Edge coordinates
  - Shared attribute



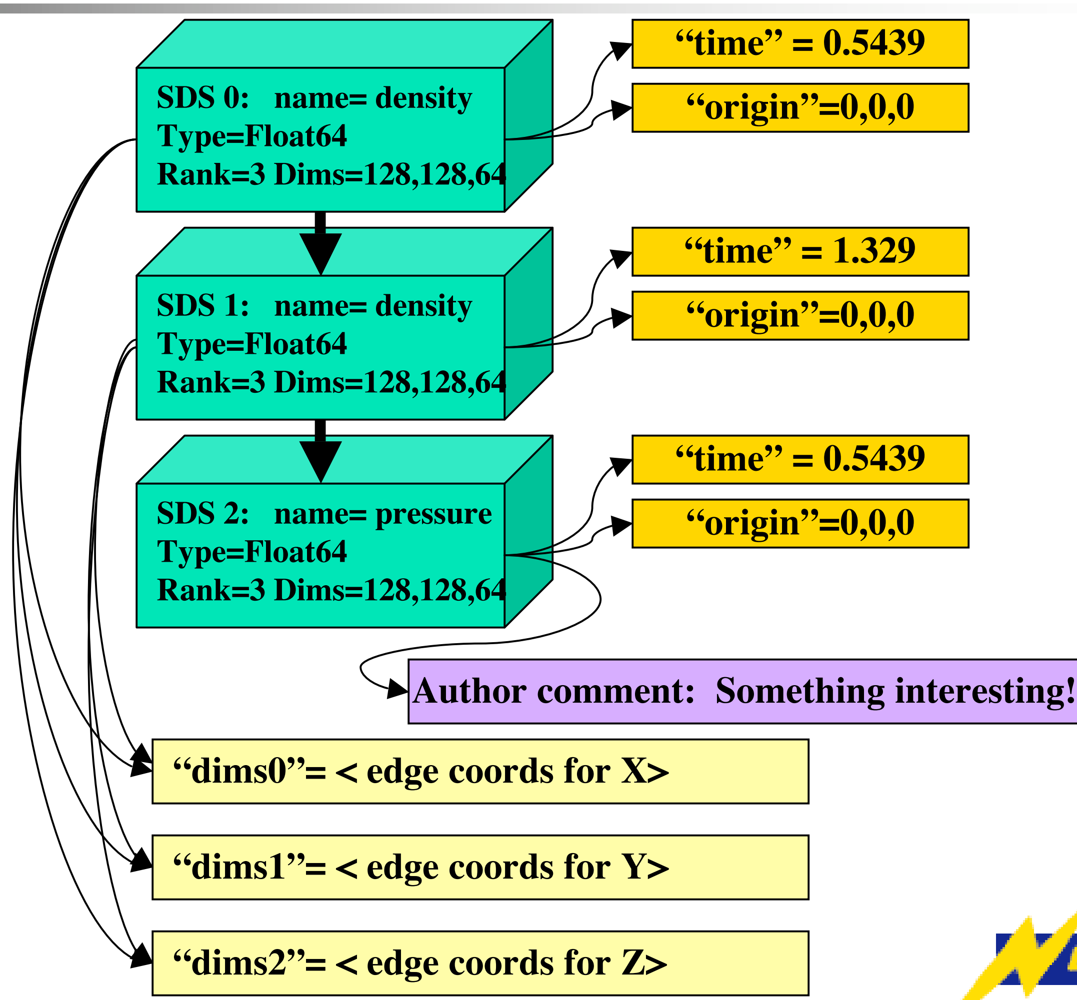
HDF5 Features

- Complete rewrite and API change
- Thread-safety support
- Parallel I/O support (via MPI-IO)
- Fundamental hierarchical grouping architecture
- No bundled data-schema centric APIs
- C and 100% Java implementations
- F90 and C++ API Bindings
- Pluggable compression methods
- Virtual File Driver Layer (RemoteHDF5)

Why Use HDF5?

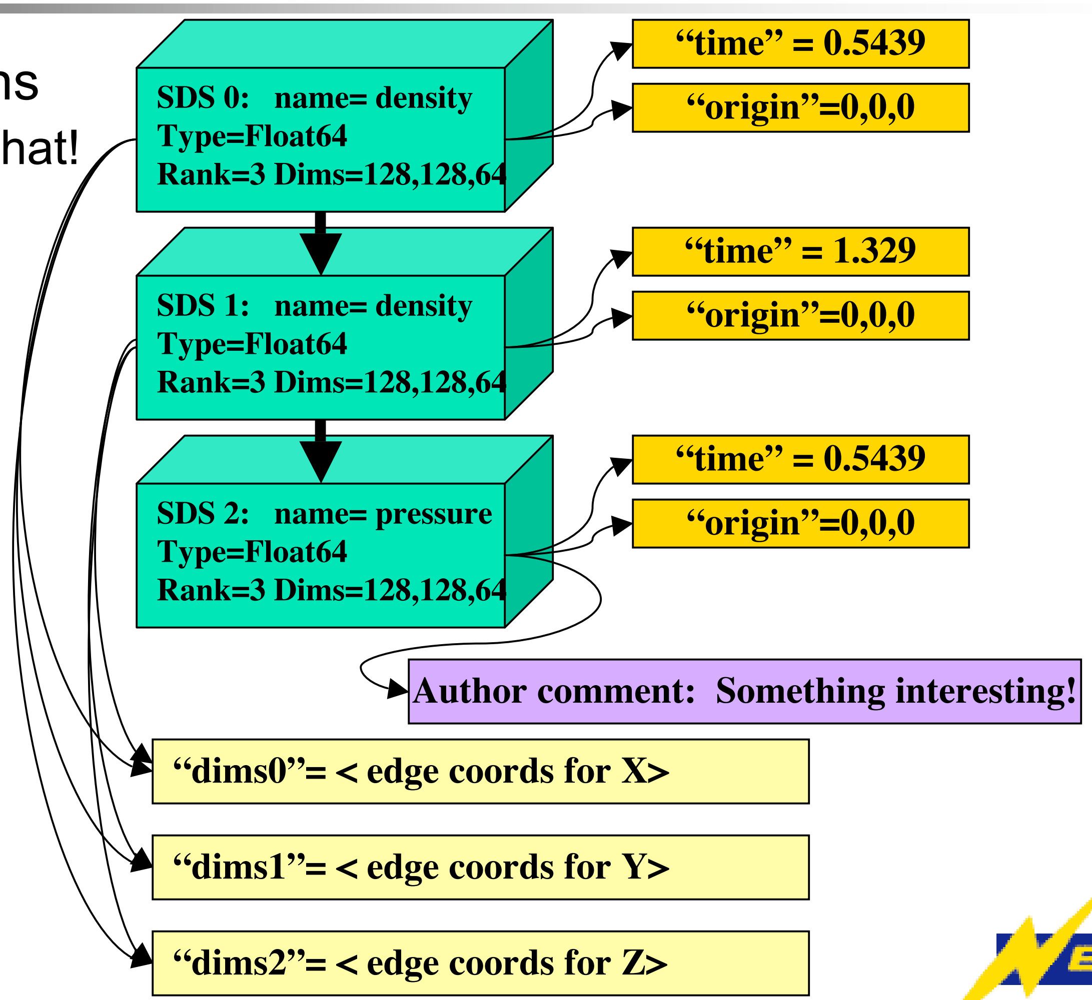
- Reasonably fast
  - faster than F77 binary unformatted I/O!
- Clean and Flexible Data Model
- Cross platform
  - Constant work maintaining ports to many architectures and OS revisions.
- Well documented
  - Members of the group dedicated to web docs
- Well maintained
  - Good track record for previous revs
- In general, less work for you in the long run!

Transition from HDF4-HDF5



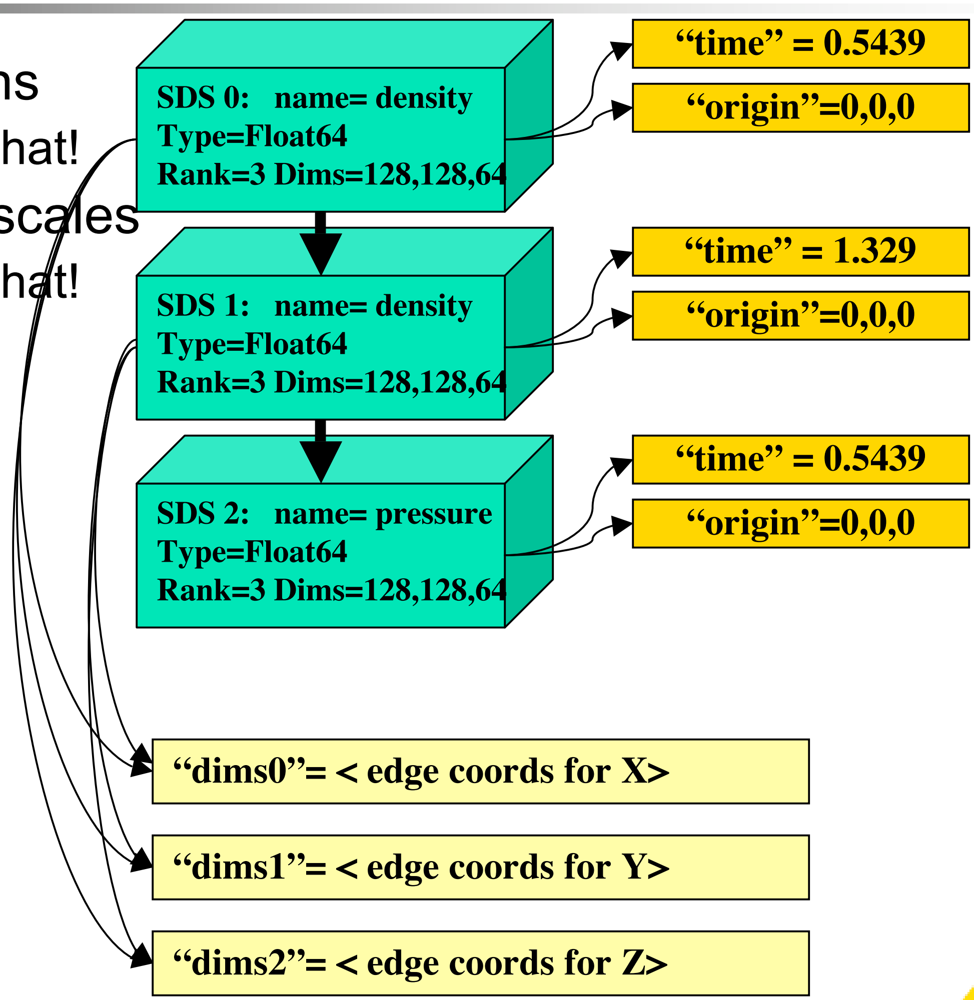
Transition from HDF4-HDF5

- Eliminate Annotations
  - Use attributes for that!



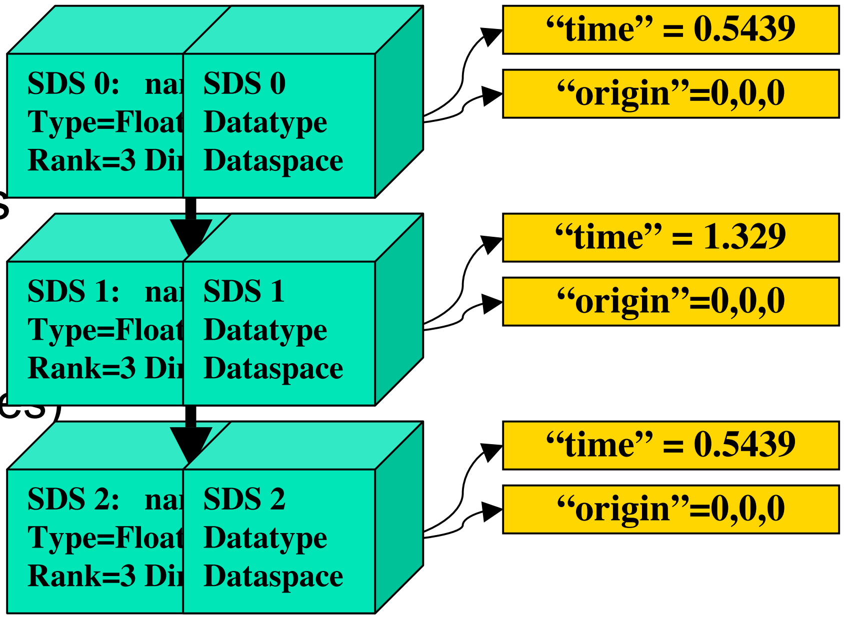
Transition from HDF4-HDF5

- Eliminate Annotations
  - Use attributes for that!
- Eliminate Dims/Dimscales
  - Use attributes for that!



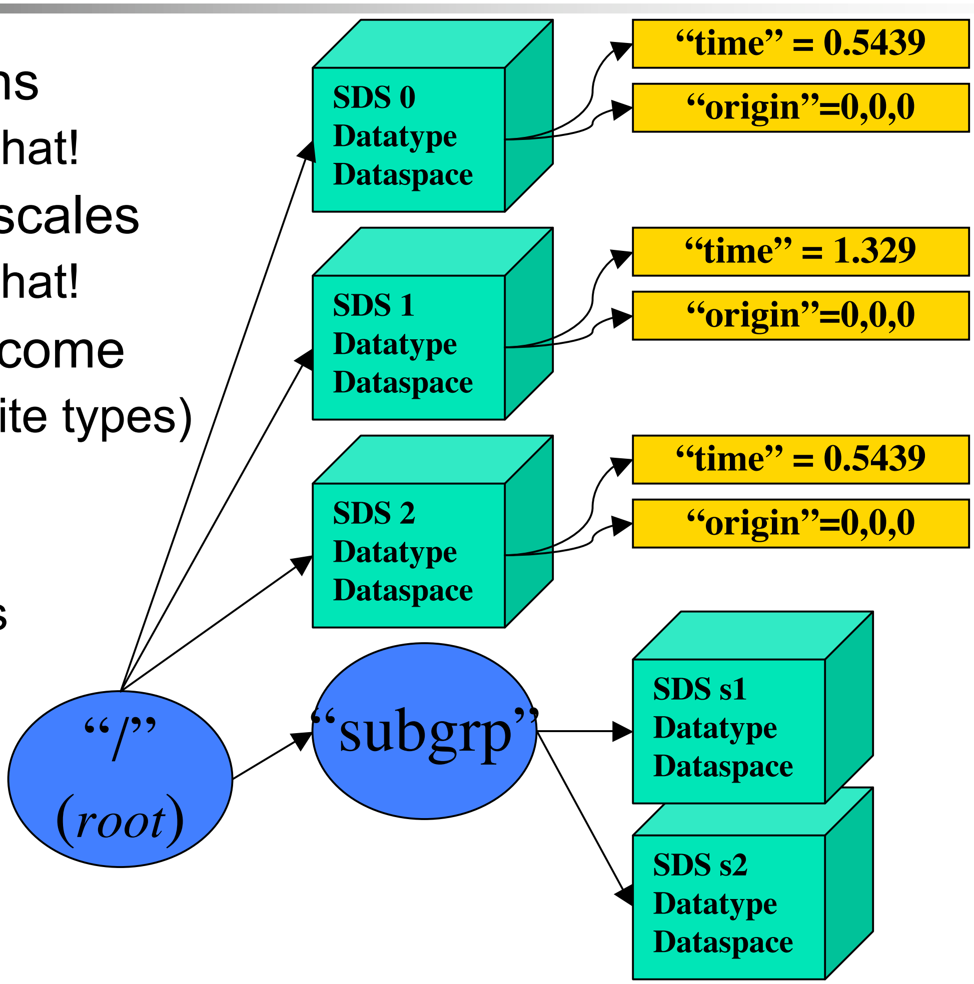
Transition from HDF4-HDF5

- Eliminate Annotations
  - Use attributes for that!
- Eliminate Dims/Dimscales
  - Use attributes for that!
- Rank, dims, type become
  - Datatype (composite types)
  - Dataspace



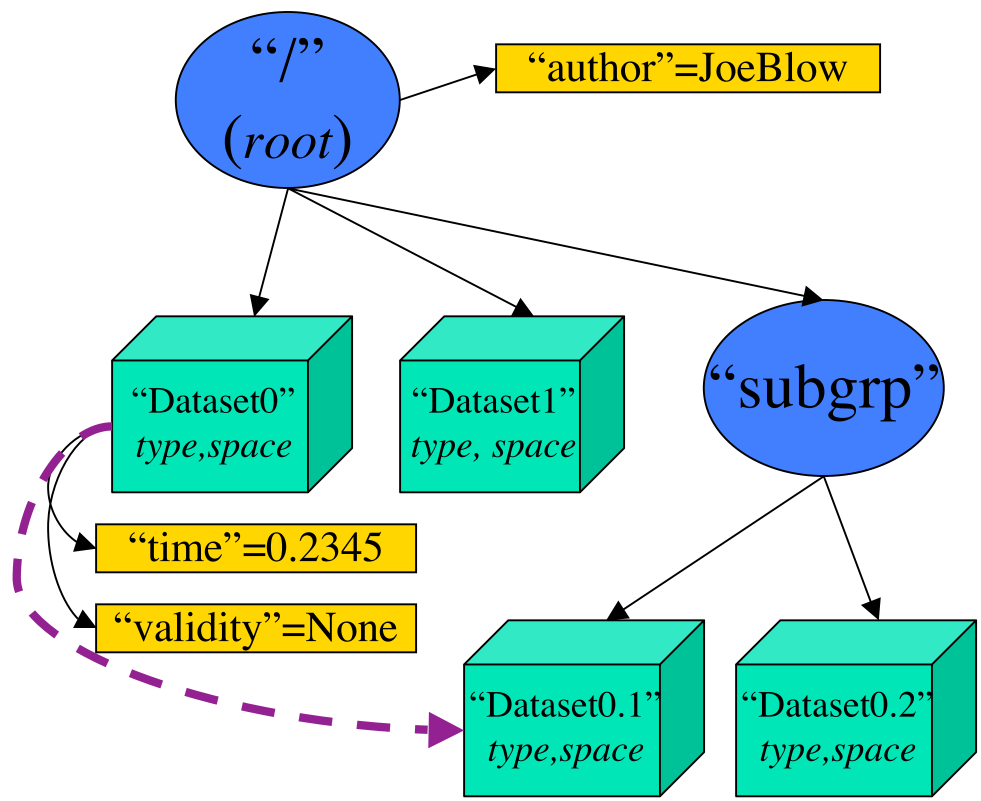
Transition from HDF4-HDF5

- Eliminate Annotations
  - Use attributes for that!
- Eliminate Dims/Dimscales
  - Use attributes for that!
- Rank, dims, type become
  - Datatype (composite types)
  - Dataspace
- Groups
  - Datasets in groups
  - Groups of groups
  - Recursive nesting



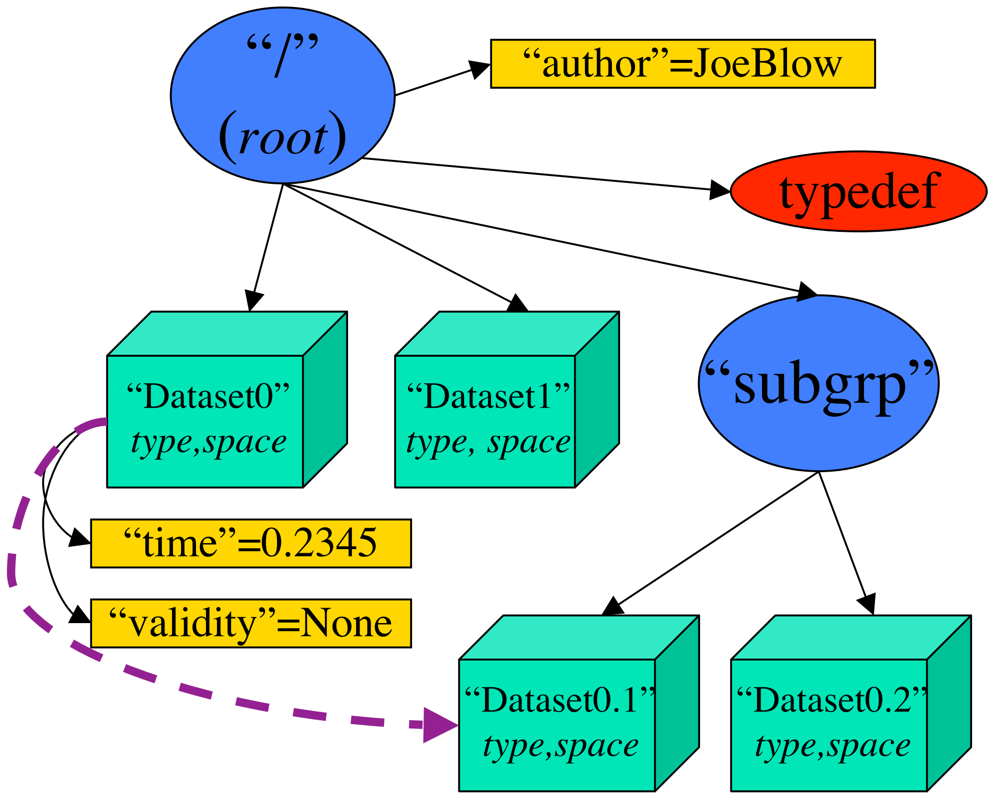
HDF5 Data Model

- Groups
  - Arranged in directory hierarchy
  - root group is always '/'
- Datasets
  - Dataspace
  - Datatype
- Attributes
  - Bind to Group & Dataset
- References
  - Similar to softlinks
  - Can also be subsets of data



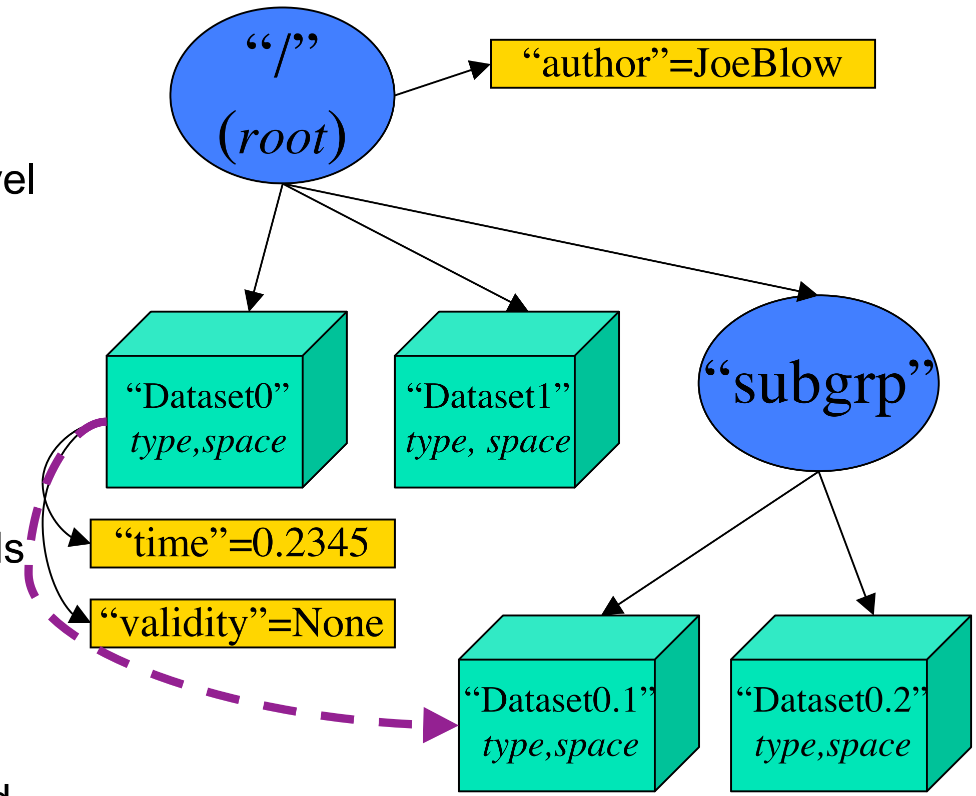
HDF5 Data Model (funky stuff)

- Complex Type Definitions
  - Not commonly used feature of the data model.
  - Potential pitfall if you commit complex datatypes to your file
- Comments
  - Yes, annotations actually do live on.



HDF5 Data Model (caveats)

- Flexible/Simple Data Model
  - You can do anything you want with it!
  - You typically define a higher level data model on top of HDF5 to describe domain-specific data relationships
  - Trivial to represent as XML!
- The perils of flexibility!
  - Must develop community agreement on these data models to *share* data effectively
  - Multi-Community-standard data models required across for reusable visualization tools
  - Preliminary work on Images and tables



Acquiring HDF5

- HDF5 home site
  - Information/Documentation http://hdf.ncsa.uiuc.edu/hdf5
  - Libraries (binary and source) ftp://ftp.ncsa.uiuc.edu/HDF/hdf5
- Module on NERSC and LBL systems
  - *module load hdf5*
  - *module load hdf5_par* (for parallel implementation)
- Typically build using
  - *./configure --prefix=<where you want it>*
  - *make*
  - *make install*

Building With HDF5

- Build Apps using
  - *#include <hdf5.h>*
  - Compile options: *-I$H5HOME/include*
- Link using
  - Normal Usage: *-L$H5HOME/lib -lhdf5*
  - With Compression: *-L$H5HOME/lib -lhdf5 -lz*
- F90/F77
  - *inc "fhdf5.inc"*
  - Use C linker with: *-lftn -lftn90*
  - or Use F90 linker with: *-lC*

The HDF5 API Organization

- API prefix denotes functional category
  - Datasets: H5D
  - Data Types: H5T
  - Data Spaces: H5S
  - Groups: H5G
  - Attributes: H5A
  - File: H5F
  - References: H5R
  - *Others (compression, error management, property lists, etc…)*

Comments about the API

- Every HDF5 object has a unique numerical Identifier
  - hid_t in C, INTEGER in F90
  - These IDs are explicitly allocated and deallocated (handles for objects)
  - Common pitfall: Failure to release a handle
- All objects in the file have a name
  - Objects in the same group *must* have unique names
  - API has many functions to convert between name and integer ID/handle (sometimes confusing as to when you should use ID/handle vs. when to use name string)

Open/Close A File (H5F)

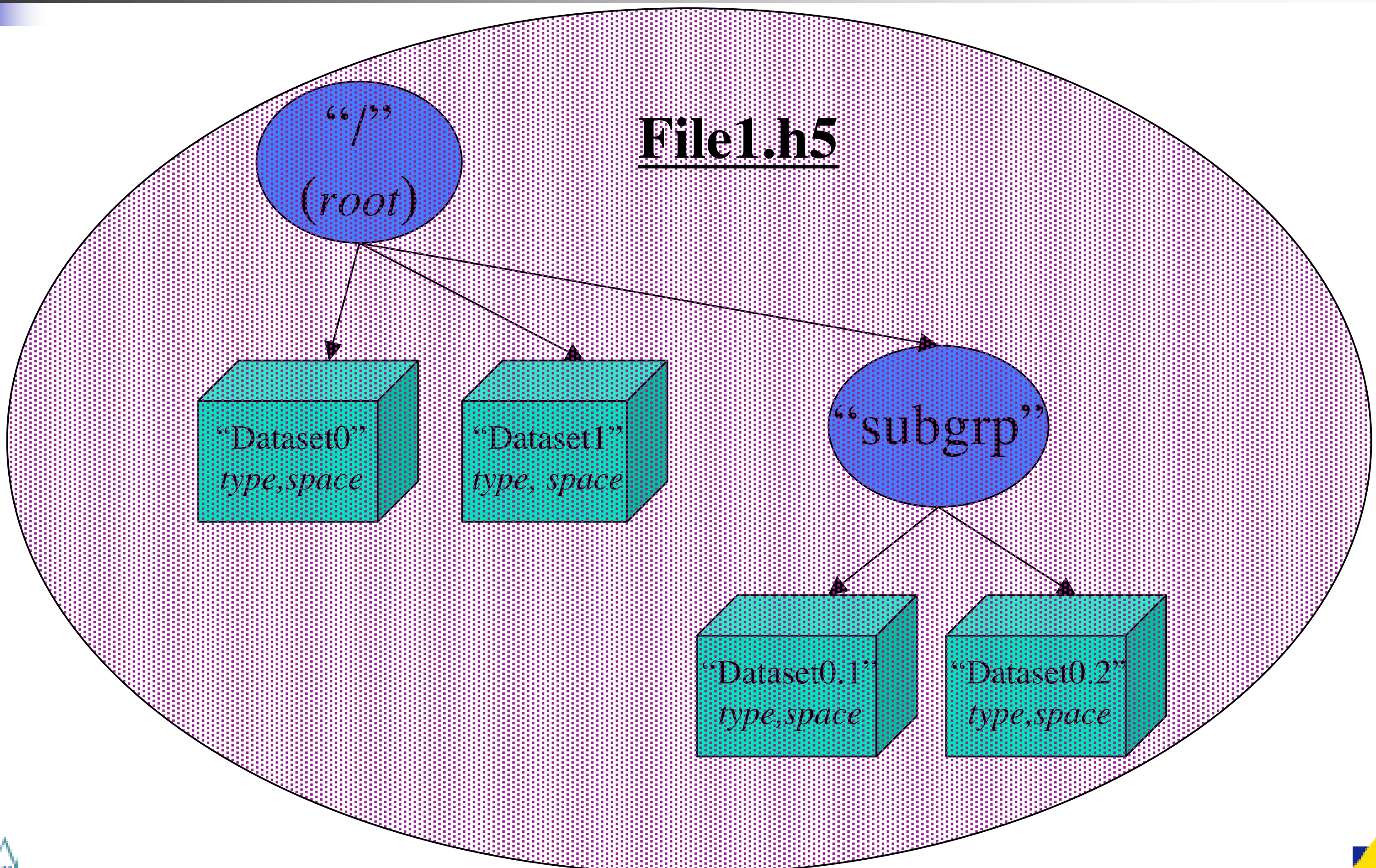
- Similar to standard C file I/O
  - *H5Fishdf5()* : check filetype before opening
  - *H5Fcreate(),H5Fopen()* : create/open files
  - *H5Fclose()* : close file
  - *H5Fflush()* : explicitly synchronize file contents
  - *H5Fmount()/H5Funmount()* : create hierarchy
- Typical use

handle = H5Fcreate("filename",H5F_ACC_TRUNC,
H5P_DEFAULT,H5P_DEFAULT);

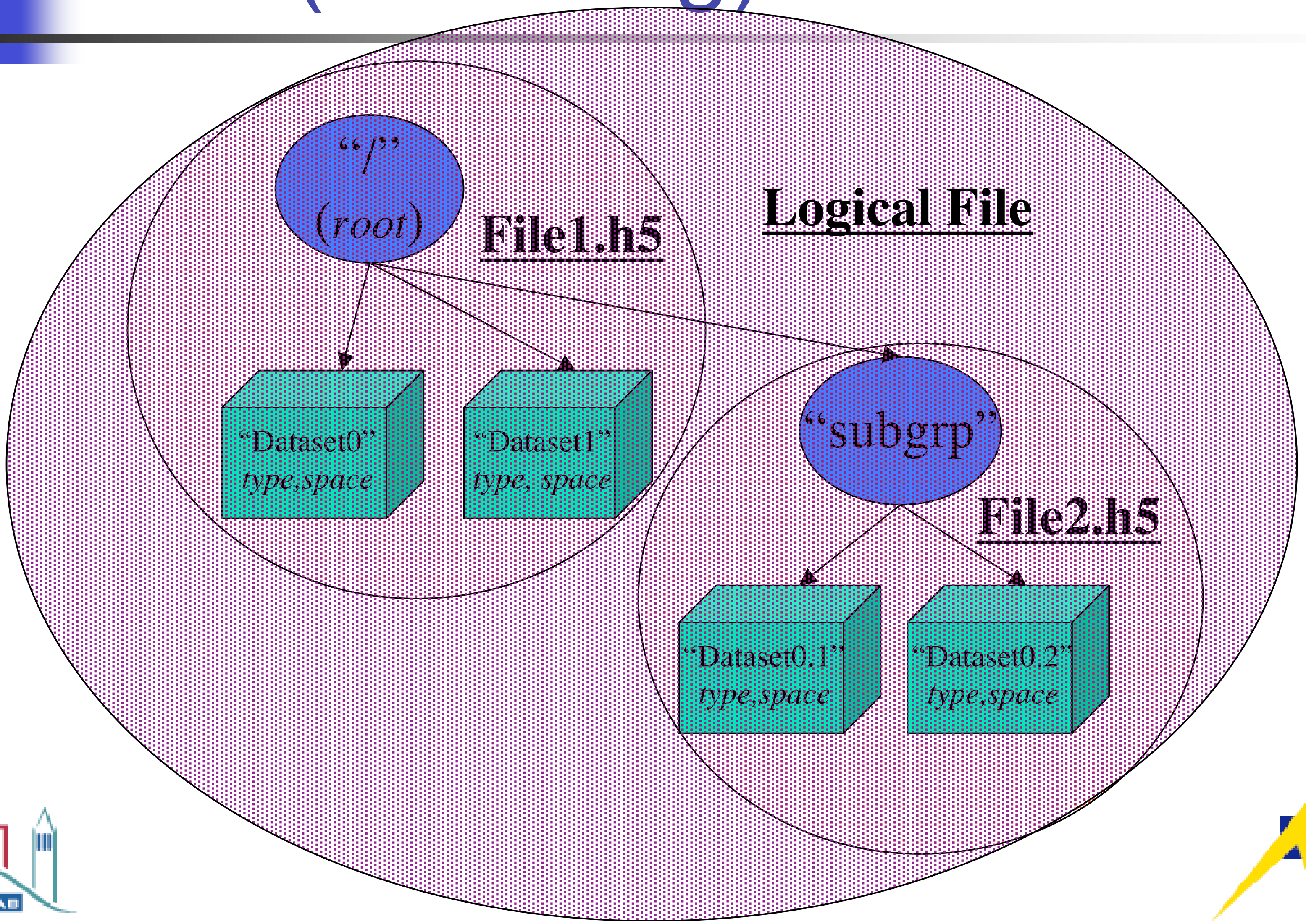
…. Do stuff with file*….*

H5Fclose(handle);

H5F (opening)


File1.h5
"/"
(root)
"Dataset0"
type,space
"Dataset1"
type, space
"subgrp"
"Dataset0.1"
type,space
"Dataset0.2"
type,space

H5F (mounting)

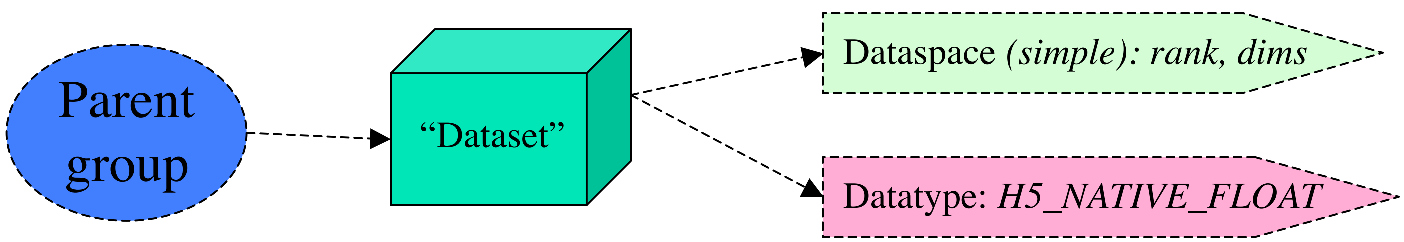


Datasets (H5D)

- A serialization of a multidimensional logical structure composed of elements of same datatype
  - Each element can be complex/composite datatype
  - Logical layout (the topological meaning *you* assign to the dataset)
  - Physical layout (the actual serialization to disk or to memory. Can be a contiguous array or chunks that are treated as logically contiguous)
- Datasets require a “Dataspace” that defines relationship between logical and physical layout
- Dimensions are in row-major order!!!
  - ie. dims[0] is least rapidly changing and dim[n] is most rapidly changing

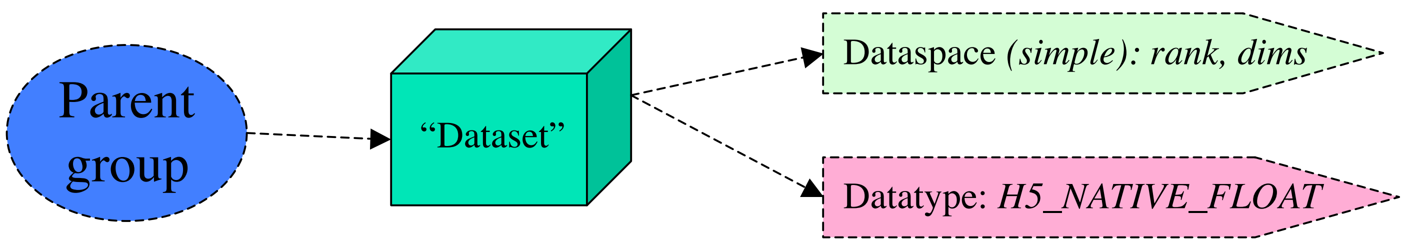
Write a Dataset (H5D)

- Required objects
  - ID for Parent Group
  - ID for DataType:
    H5T_NATIVE_FLOAT, H5T_IEEE_F32BE, H5T_IEEE_F32LE
    H5T_NATIVE_DOUBLE, H5T_NATIVE_INT, H5T_NATIVE_CHAR
  - ID for DataSpace: (logical shape of data array)
    - Simple case: rank and dimensions for logical Ndim array



Write a Dataset (H5D)

- Operations
  1. H5Fcreate() : Open a new file for writing
  2. H5Screate_simple(): Create data space (rank,dims)
  3. H5Dcreate() : Create a dataset
  4. H5Dwrite() : Commit the dataset to disk (mem-to-disk transfer)
  5. H5Sclose() : Release data space handle
  6. H5Fclose() : Close the file



H5D (example writing dataset)

```
hsize_t dims[3]={64,64,64};
float data[64*64*64];
hid_t file,dataspace,datatype,dataset;
file = H5Fcreate("myfile.h5",H5F_ACC_TRUNC,
    H5P_DEFAULT,H5P_DEFAULT);   /* open file (truncate if it exists) */
dataspace = H5Screate_simple(3,dims,NULL); /* define 3D logical array */
datatype = H5T_NATIVE_FLOAT;   /* use simple built-in native datatype */
    /* create simple dataset */
dataset = H5Dcreate(file,"Dataset1",datatype,dataspace,H5P_DEFAULT);
    /* and write it to disk */
H5Dwrite(dataset,datatype,H5S_ALL,H5S_ALL,H5P_DEFAULT,data);
H5Dclose(dataset);          /* release dataset handle */
H5Sclose(dataspace);        /* release dataspace handle */
H5Fclose(file);             /* release file handle */
```

Read a Dataset (H5D)

1. H5Fopen() : Open an existing file for reading
2. H5Dopen() : Open a dataset in the file
3. H5Dget_type() : The data type stored in the dataset
   - H5Tget_class(): is it integer or float (H5T_INTEGER/FLOAT)
   - H5Tget_order(): little-endian or big-endian (H5T_LE/BE)
   - H5Tget_size(): nbytes in the data type
4. H5Dget_space() : Get the data layout (the dataspace)
   - H5Dget_simple_extent_ndims() : Get number of dimensions
   - H5Dget_simple_extent_dims() : Get the dimensions
   - … allocate memory for the dataset ….
5. H5Dread() : Read the dataset from disk
6. H5Tclose() : Release dataset handle
7. H5Sclose() : Release data space handle
8. H5Fclose() : Close the file

Reading a Dataset (H5D)

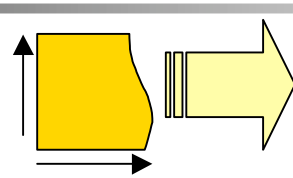
```
hsize_t dims[3], ndims;
float *data;
hid_t file,dataspace,file_dataspace,mem_dataspace,datatype,dataset;
file = H5Fopen("myfile.h5",H5F_ACC_RDONLY,H5P_DEFAULT); /* open file read_only */
datatype = H5Dget_type(dataset);          /* get type of elements in dataset */
dataspace = H5Dget_space(dataset);     /* get data layout on disk (dataspace) */
mem_dataspace = file_dataspace=dataspace;          /* make them equivalent */
ndims = H5Sget_simple_extents_ndims(dataspace);  /* get rank */
H5Sget_simple_extent_dims(dataspace,dims,NULL); /* get dimensions */
data = malloc(sizeof(float)*dims[0]*dims[1]*dims[2]);   /* alloc memory for storage*/
H5Dread(dataset,H5T_NATIVE_FLOAT,mem_dataspace,file_dataspace, /* H5S_ALL */
     H5P_DEFAULT,data);      /* read the data into memory buffer (H5S_ALL) */
H5Dclose(dataset);                /* release dataset handle */
H5Tclose(datatype);               /* release datatype handle */
H5Sclose(dataspace);              /* release dataspace handle */
H5Fclose(file);                   /* release file handle */
```

DataSpaces (H5S)

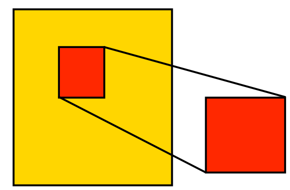
- Data Space Describes the mapping between logical and physical data layout
  - Physical is serialized array representation
  - Logical is the desired topological relationship between data elements
- Use a DataSpace to describe data layout both in memory and on disk
  - Each medium has its own DataSpace object!
  - Transfers between media involve physical-to-physical remapping but is defined as a logical-to-logical remapping operation

Data Storage Layout (H5S)

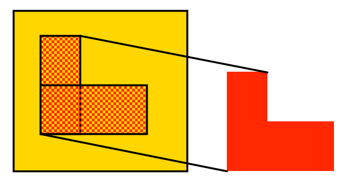
- Elastic Arrays



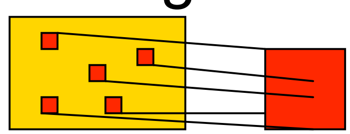
- Hyperslabs
  - Logically contiguous chunks of data
  - Multidimensional Subvolumes
  - Subsampling (striding, blocking)



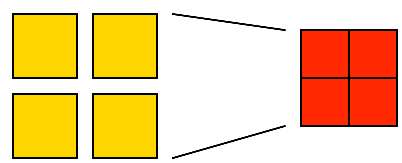
- Union of Hyperslabs
  - Reading a non-rectangular sections



- Gather/Scatter



- Chunking
  - Usually for efficient Parallel I/O



Dataset Selections (H5S)

- Array: *H5Sselect_hyperslab(H5S_SELECT_SET)*
  - Offset: start location of the hyperslab *(default={0,0…})*
  - Count: number of elements or blocks to select in each dimension *(no default)*
  - Stride: number of elements to separate each element or block to be selected *(default={1,1…})*
  - Block: size of block selected from dataspace *(default={1,1…})*
- Unions of Sets: *H5Sselect_hyperslab(H5S_SELECT_OR)*
- Gather Scatter *H5Sselect_elements(H5S_SELECT_SET)*
  - Point lists

Dataspace Selections (H5S)

Transfer a subset of data from disk to fill a memory buffer

Disk Dataspace

H5Sselect_hyperslab(disk_space, H5S_SELECT_SET,
offset[3]={1,2},NULL,count[2]={4,6},NULL)

Memory Dataspace

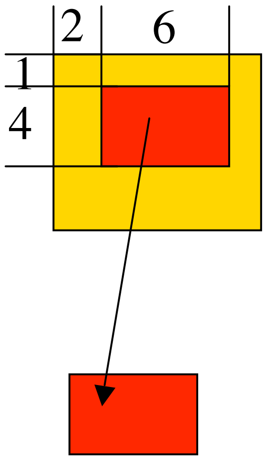
mem_space = H5S_ALL

Or

mem_space =H5Dcreate(rank=2,dims[2]={4,6});

Transfer/Read operation

H5Dread(dataset,mem_datatype, mem_space, disk_space,
H5P_DEFAULT, mem_buffer);



Dataspace Selections (H5S)

Transfer a subset of data from disk to subset in memory

Disk Dataspace

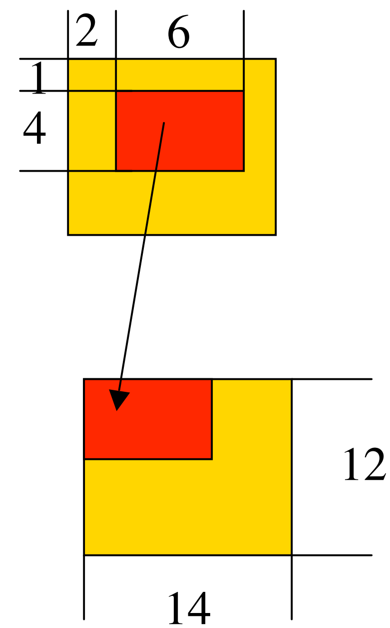
H5Sselect_hyperslab(disk_space, H5S_SELECT_SET,
offset[3]={1,2},NULL,count[2]={4,6},NULL)

Memory Dataspace

mem_space =H5Dcreate_simple(rank=2,dims[2]={12,14});
H5Sselect_hyperslab(mem_space, H5S_SELECT_SET,
offset[3]={0,0},NULL,count[2]={4,6},NULL)

Transfer/Read operation

H5Dread(dataset,mem_datatype, mem_space, disk_space,
H5P_DEFAULT, mem_buffer);



Dataspace Selections (H5S)

Row/Col Transpose

Disk Dataspace

disk_space = H5Dget_space(dataset)

rank=2 dims={12,14} canonically row-major order

Memory Dataspace

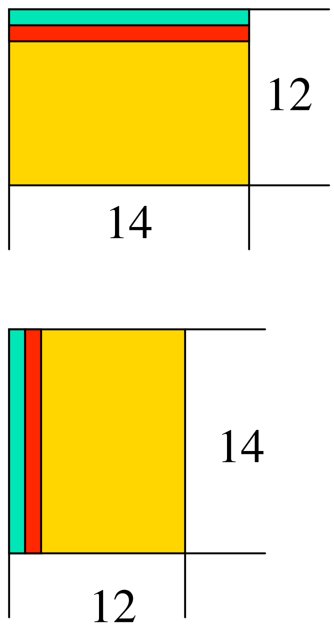
mem_space =H5Dcreate_simple(rank=2,dims[2]={14,12});

H5Sselect_hyperslab(mem_space, H5S_SELECT_SET,

offset[3]={0,0},stride={14,1},count[2]={1,12},block={14,1})

Transfer/Read operation

H5Dread(dataset,mem_datatype, mem_space, disk_space, H5P_DEFAULT, mem_buffer);



DataTypes Native(H5T)

- Native Types
  - H5T_NATIVE_INT, H5T_NATIVE_FLOAT, H5T_NATIVE_DOUBLE, H5T_NATIVE_CHAR, … H5T_NATIVE_<foo>
- Arch Dependent Types
  - Class: H5T_FLOAT, H5T_INTEGER
  - Byte Order: H5T_LE, H5T_BE
  - Size: 1,2,4,8,16 byte datatypes
  - Composite:
    - Integer: H5T_STD_I32LE, H5T_STD_I64BE
    - Float: H5T_IEEE_F32BE, H5T_IEEE_F64LE
    - String: H5T_C_S1, H5T_FORTRAN_S1
  - Arch: H5T_INTEL_I32, H5T_INTEL_F32

DataTypes (H5T)

- Type Translation for writing
  - Define Type in Memory
  - Define Type in File (native or for target machine)
  - Translates type automatically on retrieval
- Type Translation for reading
  - Query Type in file (class, size)
  - Define Type for memory buffer
  - Translates automatically on retrieval

DataTypes (H5T)

- Writing

```
dataset=H5Dcreate(file,name,mem_datatype,dataspace,…);

H5Dwrite(dataset,file_datatype, memspace, filespace,…);
```

- Reading

```
dataset=H5Dopen(file,name,mem_datatype,dataspace,…);
```

Complex DataTypes (H5T)

- Array Datatypes:
  - Vectors
  - Tensors
- Compound Objects
  - C Structures
- Variable Length Datatypes
  - Strings
  - Elastic/Ragged arrays
  - Fractal arrays
  - Polygon lists
  - Object tracking

Caveats (H5T)

- Elements of datasets that are compound/complex must be accessed in their entirety
  - It may be notationally convenient to store a tensor in a file as a dataset. However, you can select subsets of the dataset, but not sub-elements of each tensor!
  - Even if they could offer this capability, there are fundamental reasons why you would not want to use it!

Array Data Types (H5T)

- Create the Array Type
  - atype=H5Tarray_create(basetype,rank,dims,NULL)

```
hsize_t vlen=3;
flowfield = H5Tarray_create(H5T_NATIVE_DOUBLE,1,&vlen,NULL);
```

- Query the Array Type
  - H5Tget_array_ndims(atype)
  - H5Tget_array_dims(atype,dims[],NULL)

Attributes (H5A)

- Attributes are simple arrays bound to objects (datasets, groups, files) as key/value pairs
  - Name
  - Datatype
  - DataSpace: Usually a scalar H5Screate(H5S_SCALAR)
  - Data
- Retrieval
  - By name: H5Aopen_name()
  - By index: H5Aopen_idx()
  - By iterator: H5Aiterate()

Writing/Reading Attribs (H5A)

- Write an Attribute
  1. Create Scalar Dataspace: H5Screate(H5S_SCALAR)
  2. Create Attribute: H5Acreate()
  3. Write Attribute (bind to obj): H5Awrite()
  4. Release dataspace, attribute H5Sclose(), H5Aclose()

```
space=H5Screate(H5S_SCALAR);
attrib=H5Acreate(object_to_bind_to,
          "attribname",
          mem_datatype,space,
          NULL);
H5Awrite(attrib,file_datatype,data)
H5Sclose(space);
H5Aclose(attrib);
```

- Read an Attribute
  1. Open Attrib: H5Aopen_name()
  2. Read Attribute: H5Aread()
  3. Release H5Aclose()

```
Attrib=H5Aopen_name(object,"attribname");
H5Aread(attrib,mem_datatype,data);
H5Aclose(attrib);
```

Caveats about Attribs (H5A)

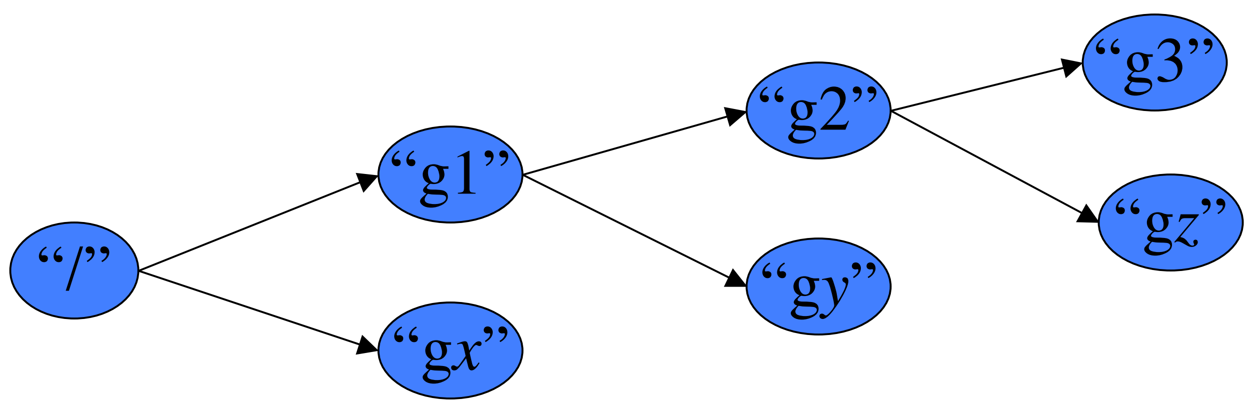
- Do not store large or complex objects in attributes
- Do not do collective operations (parallel I/O) on attributes
- Make your life simple and make datatypes implicitly related to the attribute name
  - ie. “iorigin” vs. “origin”
  - Avoid type class discovery when unnecessary

Groups (H5G)

- Similar to a “directory” on a filesystem
  - Has a name
  - Parent can only be another group (except root)
- There is always a root group in a file called “/”
- Operations on Groups
  - H5Gcreate: Create a new group (ie. Unix ‘mkdir’)
  - H5Gopen/H5Gclose: Get/release a handle to a group
  - H5Gmove: Move a directory (ie. Unix ‘mv’ command)
  - H5Glink/H5Gunlink: Hardlinks or softlinks (Unix ‘ln’) Unlink is like ‘rm’)
  - H5Giterate: Walk directory structures recursively

Groups (H5G)

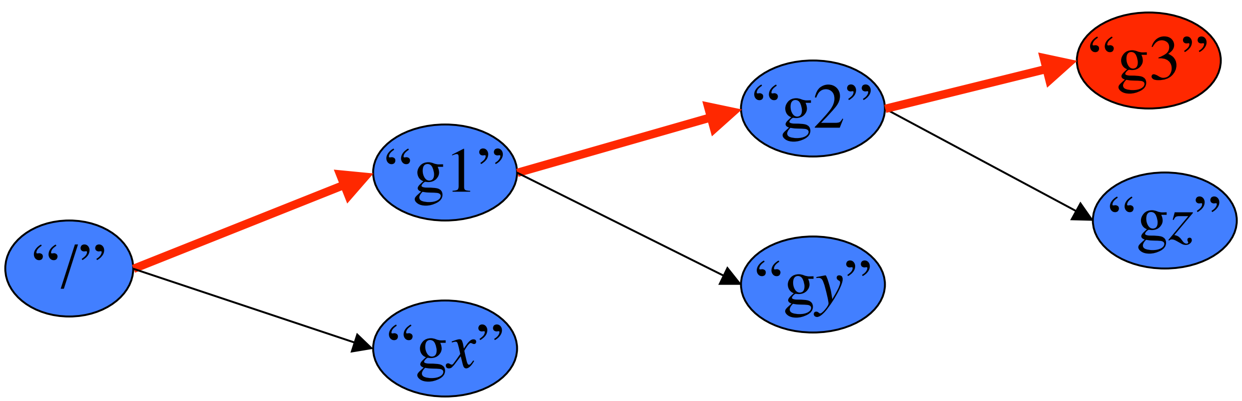
- Navigation (walking the tree)
  - Navigate groups using directory-like notation



Groups (H5G)

- Navigation (walking the tree)
  - Navigate groups using directory-like notation

(select group "/g1/g2/g3")



Groups (H5G)

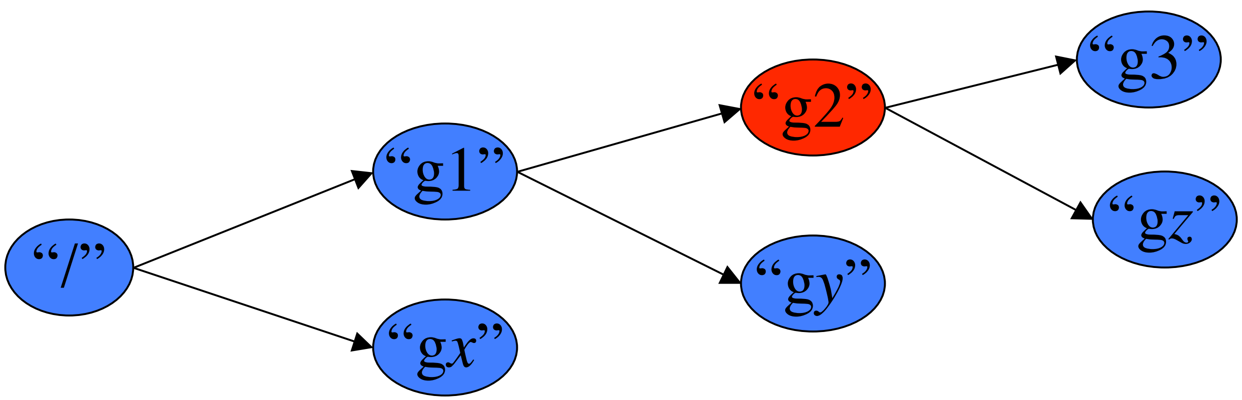
- Navigation (walking the tree)
  - Navigate groups using directory-like notation

Simple Example: Opening a particular Group "g2"

hid_t gid=H5Gopen(file_ID,"/g1/g2");

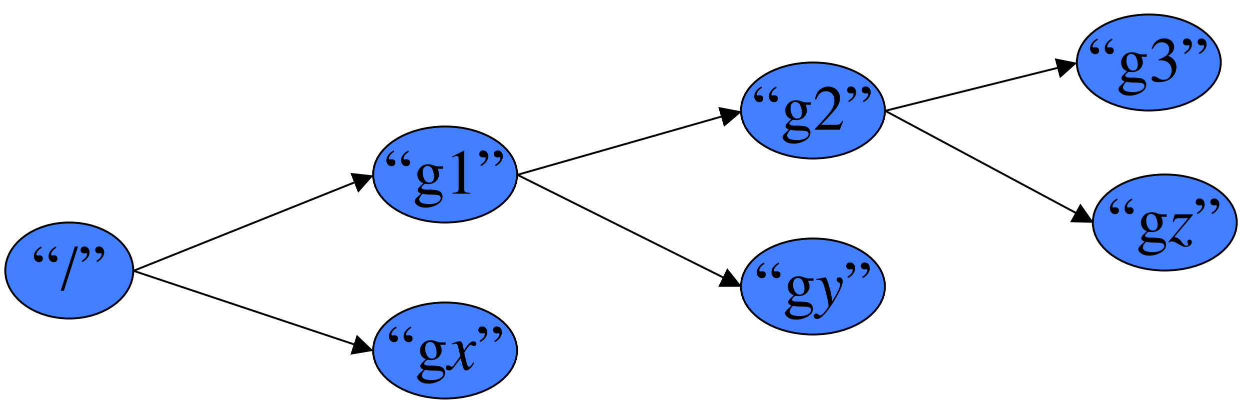
Is equivalent to

hid_t gid=H5Gopen(gid_g1,"g2");



Groups (H5G)

- Navigation
  - Navigate groups using directory-like notation
  - Navigate using iterators (a recursive walk through the tree)



H5G (Group Iterators)

- Why use iterators
  - Allows discovery of directory structure without knowledge of the group names
  - Simplifies implementation recursive tree walks
- Iterator Method (user-defined callbacks)
  - Iterator applies user callback to each item in group
  - Callback returns 1: Iterator stops immediately and returns current item ID to caller
  - Callback returns 0: Iterator continues to next item
- Pitfalls
  - No guaranteed ordering of objects in group (not by insertion order, name, or ID)
  - Infinite recursion (accidental of course)

H5G (using Group Iterators)

- Print names of all items in a group (pseudocode)

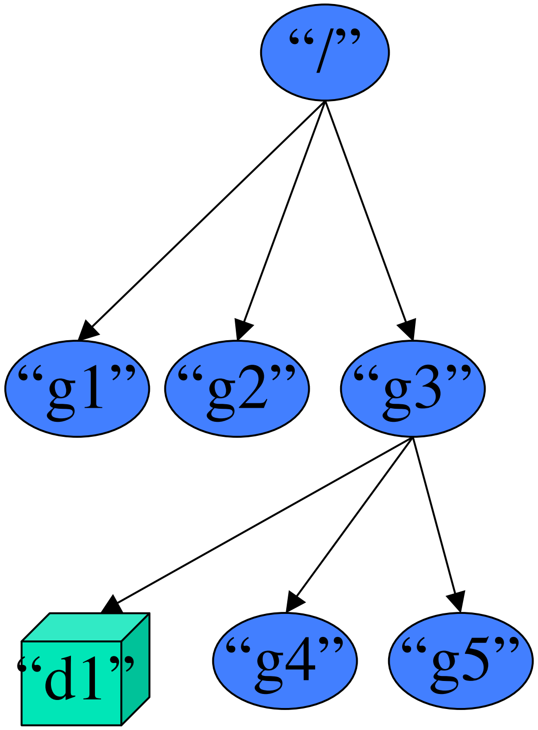
herr_t PrintName(objectID,objectName){
print(objectName)
return 0
}

H5Giterate(fileID,"/",PrintName,NULL(userData))

Outputs: "g1 g2 g3"

H5Giterate(fileID,"/g3",PrintName,NULL(userData))

Outputs: "d1 g4 g5"



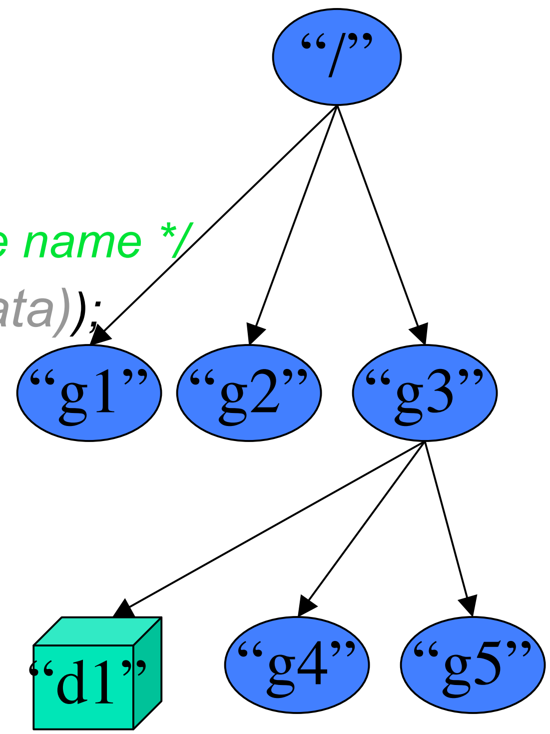
H5G (using Group Iterators)

- Depth-first walk(pseudocode)

```
herr_t DepthWalk(objectID,objectName){
    print(objectName)
    /* note: uses objectID instead of file and NULL for the name */
    H5Giterate(objectID,NULL,DepthWalk,NULL(userData));
    return 0
}

H5Giterate(fileID,"/",DepthWalk,NULL(userData))
```

Outputs: "g1 g2 g3 d1 g4 g5"

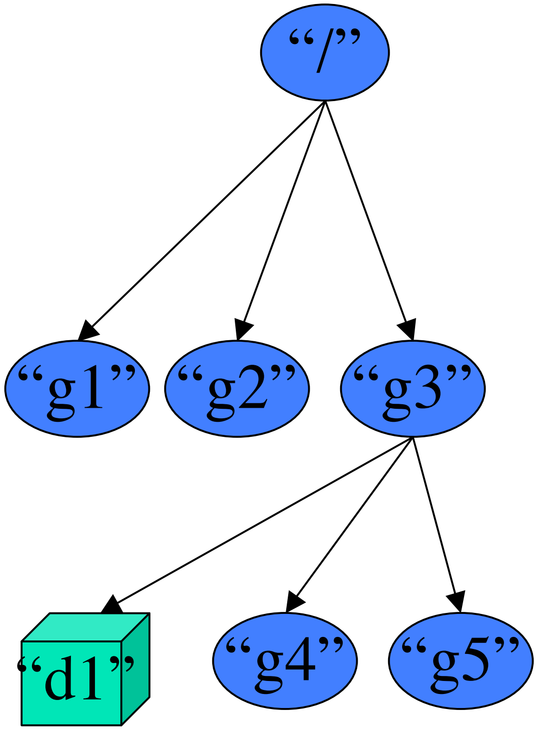


H5G (using Group Iterators)

- Count items (pseudocode)

```
herr_t CountCallback(objectID,objectName,count){
    *userdata++;
    H5Giterate(objectID,NULL,UserCallback,count);
    return 0
}
Int count;
H5Giterate(fileID,"/",CountCallback,&count)
Print(*count)
```

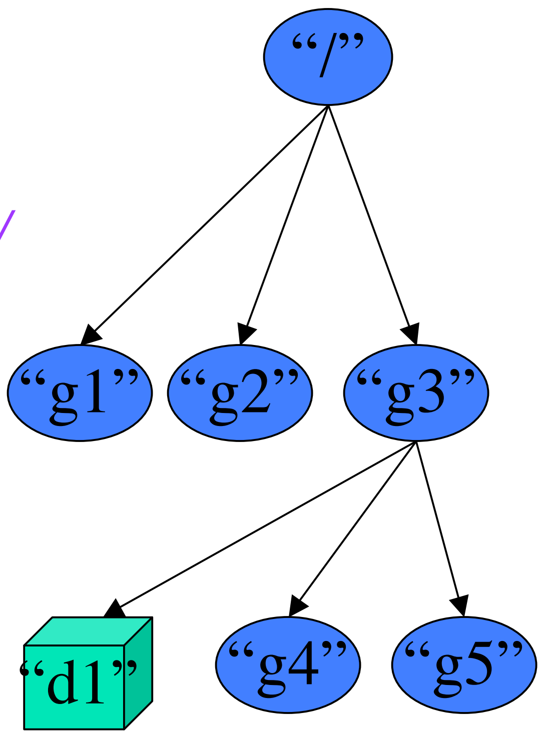
Outputs: "6"



H5G (using Group Iterators)

- Select item with property (pseudocode)

```
herr_t SelectCallback(objectID,objectName,property){
    if(getsomeproperty(objectID)==*property)
        return 1 /* terminate early: we found a match! */
    H5Giterate(objectID,NULL,UserCallback,property)
    return 0
}
match=H5Giterate(fileID,"/",SelectCallback,&property)
```



Returns item that matches property

Parallel I/O (pHDF5)

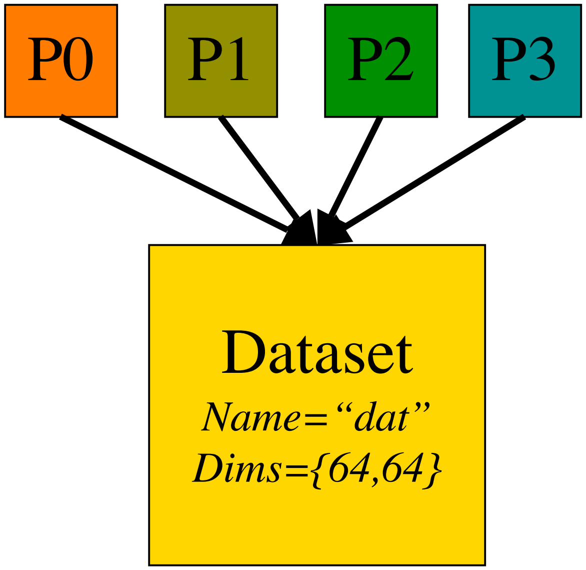
- Restricted to collective operations on datasets
  - Selecting *one* dataset at a time to operate on collectively or independently
  - Uses MPI/IO underneath (and hence is ***not*** for OpenMP threading. Use ThreadSafe HDF5 for that!)
  - Declare communicator ***only*** at file open time
  - Attributes are only actually processed by process with rank=0
- Writing to datafiles
  - Declare overall data shape (dataspace) collectively
  - Each processor then uses H5Sselect_hypeslab() to select each processor's subset of the overall dataset (the domain decomposition)
  - Write collectively or independently to those (preferably) non-overlapping offsets in the file.

pHDF5 (example 1)

- File open requires explicit selection of Parallel I/O layer.
- All PE's collectively open file and declare the overall size of the dataset.

All MPI Procs!

```
props = H5Pcreate(H5P_FILE_ACCESS);
/* Create file property list and set for Parallel I/O */
H5Pset_fapl_mpio(prop, MPI_COMM_WORLD,
MPI_INFO_NULL);
file=H5Fcreate(filename,H5F_ACC_TRUNC,
            H5P_DEFAULT,props); /* create file */
H5Pclose(props); /* release the file properties list */
filespace = H5Screate_simple(rank=2,dims[2]={64,64},
            NULL)
dataset = H5Dcreate(file,"dat",H5T_NATIVE_INT,
            space,H5P_DEFAULT); /* declare dataset */
```

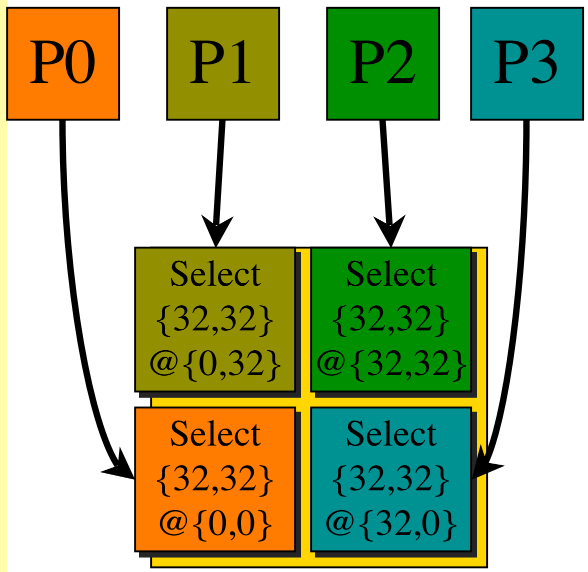


pHDF5 (example 1 cont…)

- Each proc selects a hyperslab of the dataset that represents its portion of the domain-decomposed dataset and read/write collectively or independently.

All MPI Procs!

```
/* select portion of file to write to */
H5Sselect_hyperslab(filespace, H5S_SELECT_SET,
        start= P0{0,0}:P1{0,32}:P2{32,32}:P3{32,0},
        stride= {32,1},count={32,32},NULL);
/* each proc independently creates its memspace */
memspace = H5Screate_simple(rank=2,dims={32,32},
        NULL);
/* setup collective I/O prop list */
xfer_plist = H5Pcreate (H5P_DATASET_XFER);
H5Pset_dxpl_mpio(xfer_plist, H5FD_MPIO_COLLECTIVE);
H5Dwrite(dataset,H5T_NATIVE_INT, memspace, filespace,
        xfer_plist, local_data); /* write collectively */
```



SP2 Caveats

- Must use thread-safe compilers, or it won't even recognize the mpi/io routines.
  - mpcc_r , mpxlf90_r, mpCC_r
- Must link to -lz which isn't in default path
  - Use path to hdf4 libs to get libz
- Location of libs and includes

  -I/usr/common/usg/hdf5/1.4.4/parallel/include

  -L/usr/common/usg/hdf5/1.4.4/parallel/lib -lhdf5

  -L/usr/common/usg/hdf/4.1r5 -lz -lm

Performance Caveats

- If data reorganization proves costly, then put off it off until the data analysis stage
  - The fastest writes are when layout on disk == layout in memory
  - If MPP hours are valuable, then don't waste them on massive in-situ data reorganization
  - Data reorg is usually more efficient for parallel reads than for parallel writes (especially for SMPs)
  - Take maximum advantage of "chunking"
- Parallel I/O performance issues usually are direct manifestations of MPI/IO performance
- Don't store ghost zones!
- Don't store large arrays in attributes

Serial I/O Benchmarks

System	HDF4.1r5 (netCDF)	HDF5 v1.4.4	FlexIO (Custom)	F77 Unf
SGI Origin 3400 *(escher.nersc.gov)*	111M/s	189M/s	180M/s	140M/s
IBM SP2 *(seaborg.nersc.gov)*	65M/s	127M/s	110M/s	110M/s
Linux IA32 *(platinum.ncsa.uiuc.edu)*	34M/s	40M/s	62M/s	47M/s
Linux IA64 Teragrid node *(titan.ncsa.uiuc.edu)*	26M/s	83M/s	77M/s	112M/s
NEC/Cray SX-6 *(rime.cray.com)*				

- Write 5-40 datasets of 128^3 DP float data
- Single CPU (multiple CPU's can improve perf. until interface saturates
- Average of 5 trials

GPFS MPI-I/O Experiences

nTasks	I/O Rate 16 Tasks/node	I/O Rate 8 tasks per node
8	-	131 Mbytes/sec
16	7 Mbytes/sec	139 Mbytes/sec
32	11 Mbytes/sec	217 Mbytes/sec
64	11 Mbytes/sec	318 Mbytes/sec
128	25 Mbytes/sec	471 Mbytes/sec

- Block domain decomp of 512^3 3D 8-byte/element array in memory written to disk as single undecomposed 512^3 logical array.
- Average throughput for 5 minutes of writes x 3 trials

Whats Next for HDF5?

- Standard Data Models with Associated APIs
  - Ad-Hoc (one model for each community)
  - Fiber Bundle (unified data model)
  - Whats in-between?
- Web Integration
  - 100% pure java implementation
  - XML/WSDL/OGSA descriptions
- Grid Integration:
  - http://www.zib.de/Visual/projects/TIKSL/HDF5-DataGrid.html
  - RemoteHDF5
  - StreamingHDF5

More Information

- HDF Website
  - Main: http://hdf.ncsa.uiuc.edu
  - HDF5: http://hdf.ncsa.uiuc.edu/HDF5
  - pHDF5: http://hdf.ncsa.uiuc.edu/Parallel_HDF
- HDF at NASA (HDF-EOS)
  - http://hdfeos.gsfc.nasa.gov/
- HDF5 at DOE/ASCI (Scalable I/O Project)
  - http://www.llnl.gov/icc/lc/siop/