



# I/O Strategies and Libraries

Adrian Jackson  
[adrianj@epcc.ed.ac.uk](mailto:adrianj@epcc.ed.ac.uk)  
@adrianjhpc

# Shared Memory



- Easy to solve in shared memory
  - imagine a shared array called **x**

```
begin serial region
    open the file
    write x to the file
    close the file
end serial region
```

- Simple as every thread can access shared data
  - may not be efficient but it works
- But what about distributed memory?

# I/O Strategies



- Basic one file for a program
  - Works fine for serial
  - Most codes use this initially
  - Works for shared memory parallelism
- Distributed memory
  - Data now not in single memory space
- Master I/O
  - Use communication to get and send all data from one process
  - High overhead
  - Use single file
  - Memory issues, no access to I/O resources at scale

# I/O Strategies cont.

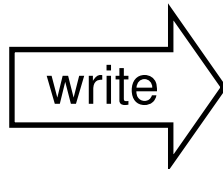


- Individual files
  - Each process writes own file (either on shared filesystem or local scratch space)
  - Use as much of I/O system as possible
  - file contents dependent on number of CPUs and decomposition
  - pre / post-processing steps needed to change number of processes
  - Filesystem breaks down for large numbers of processors
  - File handles or number of files a problem
- Look to better solution
  - I/O libraries

## 2x2 to 1x4 Redistribution



4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13



11	12	15	16
----	----	----	----

data4.dat

9	10	13	14
---	----	----	----

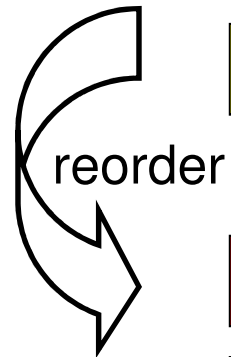
data3.dat

3	4	7	8
---	---	---	---

data2.dat

1	2	5	6
---	---	---	---

data1.dat



4	8	12	16
---	---	----	----

newdata4.dat

3	7	11	15
---	---	----	----

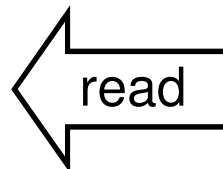
newdata3.dat

2	6	10	14
---	---	----	----

newdata2.dat

1	5	9	13
---	---	---	----

newdata1.dat




# I/O options



- I/O to single file
  - Everyone involved in I/O
    - Processes write their own data
  - I/O Server/I/O Writers
    - Subset of processes do I/O
- Choice depends on scale and operations to be done and filesystem characteristics
- All I/O
  - Good up to reasonable scale for standard parallel filesystems (10,000s processes)
- Sub I/O
  - Good for very large scale applications or where processing of data is required
  - Enables collection of data and in-situ analytics

# Files vs Arrays



- Think of the file as a large array
  - forget that I/O actually goes to disk
  - imagine we are recreating a single large array on a master process
- The I/O system must create this array and save to disk
  - without running out of memory
    - never actually creating the entire array
    - ie without doing naive master I/O
  - and by doing a small number of large I/O operations
    - merge data to write large contiguous sections at a time
  - utilising any parallel features
    - doing multiple simultaneous writes if there are multiple I/O nodes
    - managing any coherency issues re file blocks

# MPI-I/O



- Aim to provide distributed access to single file
  - File shared
  - Control by programmer
  - Look like a serial program has written the data
- Part of MPI-2 standard
  - <http://www.mpi-forum.org/docs/docs.html>
  - Typically available in modern MPI libraries, but if not can use ROMIO (MPI-IO built on MPI-1 calls)
  - Performance dependent on implementation
- Built on MPI collective operations
  - Data structure defined by programmer



# MPI-I/O cont.



- Array based I/O
  - Each process creates description of subset it holds (derived datatype)
  - No checking of correctness
- Library handles read and write to files
  - Don't ever have all in memory
  - Everything done with MPI calls
  - Scale as well as MPI communications
  - Best performance for big reads/writes
- Info object for passing system specific information
  - Lots of optimisations, tweaking, etc...

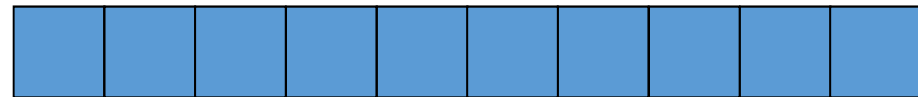
# Basic Datatypes



- MPI has a number of pre-defined datatypes
  - eg **MPI\_INT** / **MPI\_INTEGER**, **MPI\_FLOAT** / **MPI\_REAL**
  - user passes them to send and receive operations
- For example, to send 4 integers from an array x

C: `int[10];`

F: `INTEGER x(10)`



`MPI_Send(x, 4, MPI_INT, ...);`

`MPI_SEND(x, 4, MPI_INTEGER, ...)`



# Simple Example



- Contiguous type

```
MPI Datatype my_new_type;  
MPI_Type_contiguous(count=4, oldtype=MPI_INT, newtype=&my_new_type);  
MPI_Type_commit(&my_new_type);
```

```
INTEGER MY_NEW_TYPE  
CALL MPI_TYPE_CONTIGUOUS(4, MPI_INTEGER, MY_NEW_TYPE, IERROR)  
CALL MPI_TYPE_COMMIT(MY_NEW_TYPE, IERROR)
```

```
MPI_Send(x, 1, my_new_type, ...);
```

```
MPI_SEND(x, 1, MY_NEW_TYPE, ...)
```



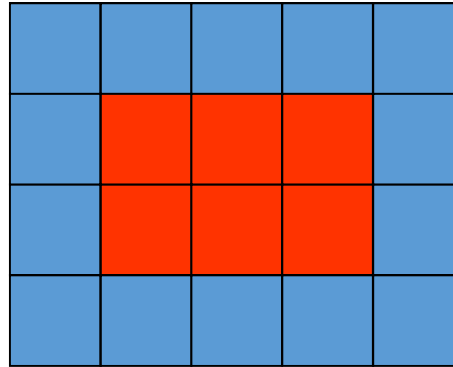
- Vector types correspond to patterns such as



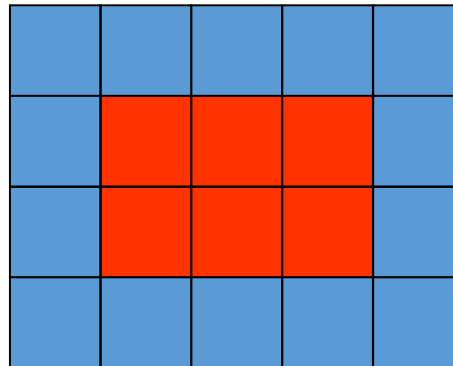
# Array Subsections in Memory



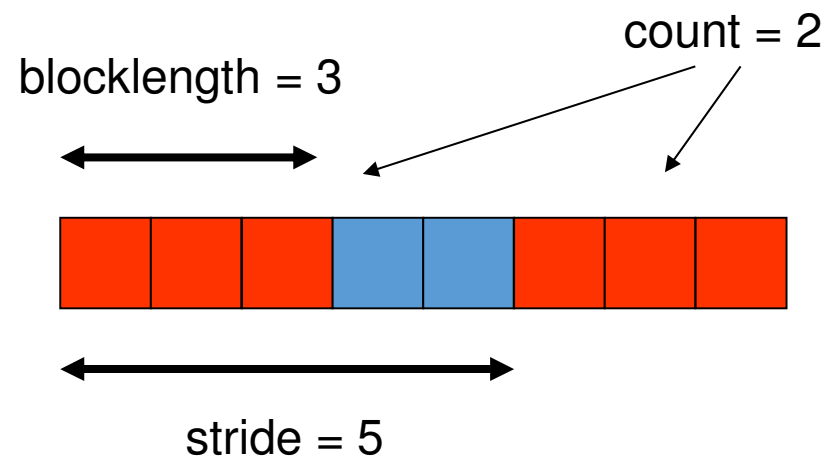
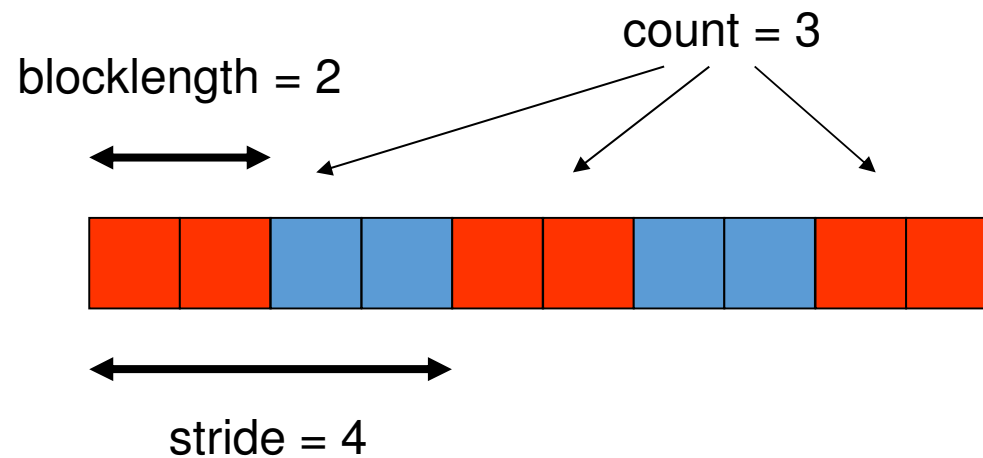
C: **x**[5][4]



F: **x**(5, 4)



# Equivalent Vector Datatypes



# Definition in MPI



```
MPI_Type_vector(int count, int blocklength, int stride,  
                MPI_Datatype oldtype, MPI_Datatype *newtype);
```

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE,  
                OLDTYPE, NEWTYPE, IERR)
```

```
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE
```

```
INTEGER NEWTYPE, IERR
```

```
MPI_Datatype vector3x2;
```

```
MPI_Type_vector(3, 2, 4, MPI_FLOAT, &vector3x2)
```

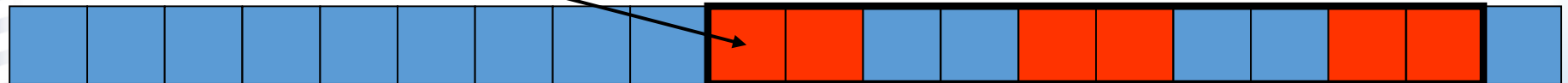
```
MPI_Type_commit(&vector3x2)
```

```
integer vector3x2
```

```
call MPI_TYPE_VECTOR(2, 3, 5, MPI_REAL, vector3x2, ierr)
```

```
call MPI_TYPE_COMMIT(vector3x2, ierr)
```

# Datatypes as Floating Templates



# MPI-IO vs Master IO



- Can use MPI-I/O derived types to do master I/O
  - Used them to do multiple sends from a master
- This requires a buffer to hold entire file on master
  - not scalable to many processes due to memory limits
- MPI-I/O model
  - each process defines the datatype for its section of the file
  - these are passed into the MPI-I/O routines
  - data is automatically read and transferred directly to local memory
  - there is no single large buffer and no explicit master process



# MPI-I/O Approach



- Four stages
  - open file
  - set file view
  - read or write data
  - close file
- All the complexity is hidden in setting the file view
  - this is where the derived datatypes appear
- Write is probably more important in practice than read
  - but exercises concentrate on read
  - makes for an easier progression from serial to parallel I/O examples

# Opening a File



```
MPI_File_open(MPI_Comm comm, char *filename, int amode,  
              MPI_Info info, MPI_File *fh)
```

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERR)
```

```
CHARACTER*(*) FILENAME
```

```
INTEGER COMM, AMODE, INFO, FH, IERR
```

- Attaches a file to the File Handle
  - use this handle in all future IO calls
  - analogous to C file pointer or Fortran unit number
- Routine is collective across the communicator
  - must be called by all processes in that communicator
- Access mode specified by amode
  - common values are: `MPI_MODE_CREATE`, `MPI_MODE_RDONLY`,  
`MPI_MODE_WRONLY`, `MPI_MODE_RDWR`

# Examples



```
MPI_File fh;  
int amode = MPI_MODE_RDONLY;  
MPI_File_open(MPI_COMM_WORLD, "data.in", amode,  
              MPI_INFO_NULL, &fh);
```

```
integer fh  
integer amode = MPI_MODE_RDONLY  
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'data.in', amode,  
                   MPI_INFO_NULL, fh, ierr)
```

- Must specify create as well as write for new files

```
int      amode = MPI_MODE_CREATE | MPI_MODE_WRONLY;  
integer amode = MPI_MODE_CREATE + MPI_MODE_WRONLY
```

- will return to the `info` argument later

# Closing a File



```
MPI_File_close(MPI_File *fh)
```

```
MPI_FILE_CLOSE(FH, IERR)
```

```
INTEGER FH, IERR
```

- Routine is collective across the communicator
  - must be called by all processes in that communicator

# Reading Data



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

```
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERR)  
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERR
```

- Reads **count** objects of type **datatype** from the file on each process
  - this is collective across the communicator associated with **fh**
  - similar in operation to C **fread** or Fortran **read**
- No offsets into the file are specified in the read
  - but processes do not all read the same data!
  - actual positions of read depends on the process's own file view
- Similar syntax for write

# Setting the File View



```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,  
                      MPI_Datatype etype, MPI_Datatype filetype,  
                      char *datarep, MPI_Info info);
```

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO,  
                  IERROR)
```

```
INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
```

```
CHARACTER*(*) DATAREP
```

```
INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

- **disp** specifies the starting point in the file *in bytes*
- **etype** specifies the elementary datatype which is the building block of the file
- **filetype** specifies which subsections of the global file each process accesses
- **datarep** specifies the format of the data in the file
- **info** contains hints and system-specific information – see later

# File Views



- Once set, the process only sees the data in the view
  - data starts at different positions in the file depending on the displacement and/or leading gaps in fixed datatype
  - can then do linear reads – holes in datatype are skipped over

4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13

rank 1 (0,1)	rank 3 (1,1)
rank 0 (0,0)	rank 2 (1,0)

global file

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----

rank 1 filetype

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

(fixed type, disp = 0)

rank 1 view of file

3	4	7	8
---	---	---	---

# Data Representation



- **datarep** is a string that can be
  - "native"
  - "internal"
  - "external32"
- Fastest is "native"
  - raw bytes are written to file exactly as in memory
- Most portable is "external32"
  - should be readable by MPI-IO on any platform
- Middle ground is "internal"
  - portability depends on the implementation
- Recommend "native"
  - convert file format by hand as and when necessary



# Choice of Parameters (1)



- Many different combinations are possible
  - choices of displacements, filetypes, etypes, datatypes, ...
- Simplest approach is to set **disp** = 0 everywhere
  - then specify offsets into files using fixed datatypes when setting view
    - non-zero **disp** could be useful for skipping global header (eg metadata)
  - **disp** must be of the correct type in Fortran (NOT a default integer)
  - **CANNOT** specify '0' for the displacement: need to use a variable

```
INTEGER(KIND=MPI_OFFSET_KIND) DISP = 0
```

```
CALL MPI_FILE_SET_VIEW(FH, DISP, ...)
```

- Recommend setting the view with fixed datatypes
  - and zero displacements

## Choice of Parameters (2)

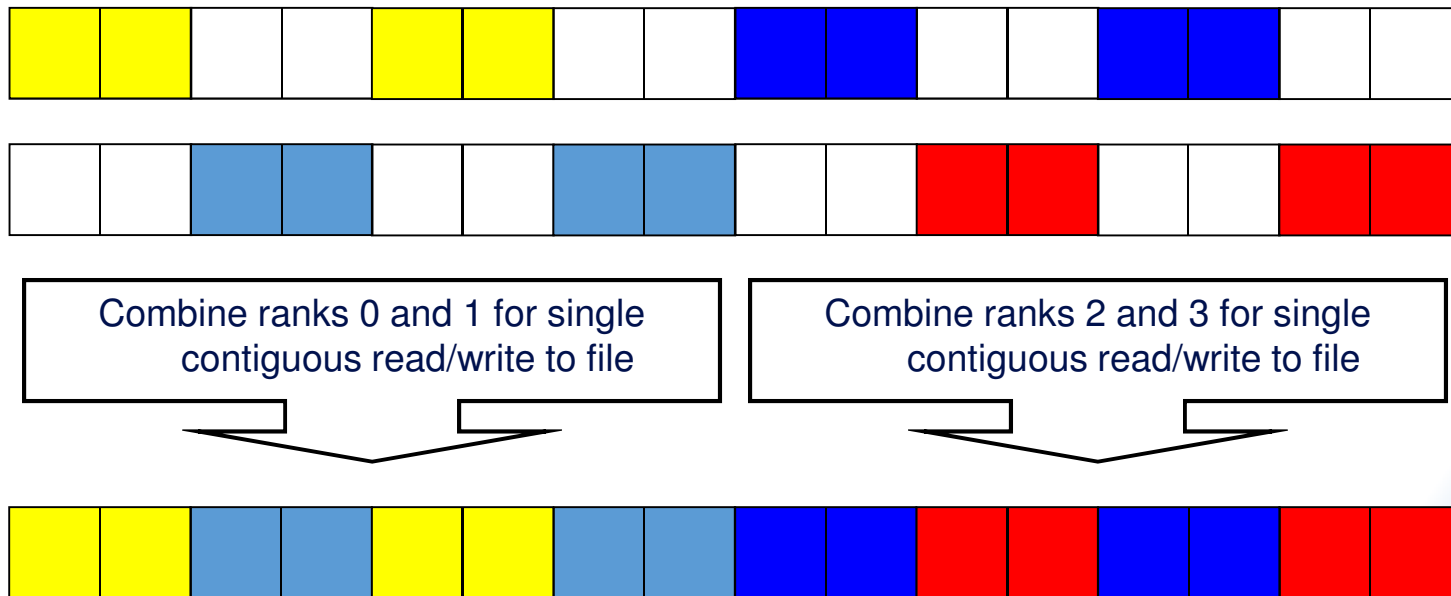


- Can also use floating datatypes in the view
  - each process then specifies a different, non-zero value of `disp`
- Problems
  - `disp` is specified in bytes so need to know the size of the `etype`
  - files are linear 1D arrays
    - need to do a calculation for displacement of element of 2D array
    - something like  $i * NY + j$  (in C) or  $j * NX + i$  (in Fortran)
    - then multiply by the number of bytes in a float or REAL
- `etype` is normally something like `MPI_REAL` or `MPI_FLOAT`
  - `datatype` in read/write calls is usually the same as the `etype`

# Collective I/O



- For read and write, “\_a11” means operation is collective
  - all processes attached to the file are taking part
- Other I/O routines exist which are individual (delete “\_all”)
  - functionality is the same but performance will be slower
  - collective routines can aggregate reads/writes for better performance



## Other individual operations



- Alternative approach
  - let everyone see the whole file (i.e. do not set a view)
  - manually seek to correct location using, e.g., **`MPI_File_write_at()`**
  - displacement is in units of the extent of the **`datatype`**
- Disadvantages
  - a very low-level, manual approach less amenable to I/O optimisation
  - danger that each request is handled individually with no aggregation
  - can use **`MPI_File_write_at_all()`** but might still be slow

# INFO Objects and Performance



- Used to pass optimisation hints to MPI-I/O
  - implementations can define any number of allowed values
  - these are portable in as much as they can be ignored!
  - can use the default value `info = MPI_INFO_NULL`
- Info objects can be created, set and freed
  - `MPI_Info_create`
  - `MPI_Info_set`
  - `MPI_Info_free`
  - see man pages for details
- Using appropriate values may be key to performance
  - e.g. setting buffer sizes, blocking factors, number of IO nodes, ...
  - but is dependent on the system and the MPI implementation
  - need to consult the MPI manual for your machine
  - on ARCHER, easier to tune Lustre file system than use MPI-I/O hints

# Non-blocking I/O in MPI-I/O



- Two forms
  - General non-blocking
    - `MPI_File_iread(fh, buf, count, datatype, request)`
    - finish by waiting on `request`
    - but no collective version
  - Split collective
    - `MPI_File_write_all_begin(fh, buf, count, datatype)`
    - `MPI_File_write_all_end(fh, buf, status)`
    - only a single outstanding I/O operation at any one time
    - allows for collective version

# MPI-I/O



- MPI-I/O calls deceptively simple
- User must define appropriate filetypes so file view is correct on each process
  - this is the difficult part!
- Use collective calls whenever you can
  - enables I/O library to merge reads and writes
  - enables a smaller number of larger I/O operations from/to disk



# NetCDF



- **Network Common Data Format**
  - Data model
  - File format
  - Application programming interface (API)
  - Library implementing the API
- NetCDF
  - Created in the US by unidata for earth science and geoscience data, supported by the NSF
- NetCDF
  - Software library and self-describing data format
  - Portable, machine independent data
  - Can use HDF5 or NetCDF format (HDF5 gives larger files and unlimited array dimensions) in NetCDF 4.0 (latest version)



# NetCDF

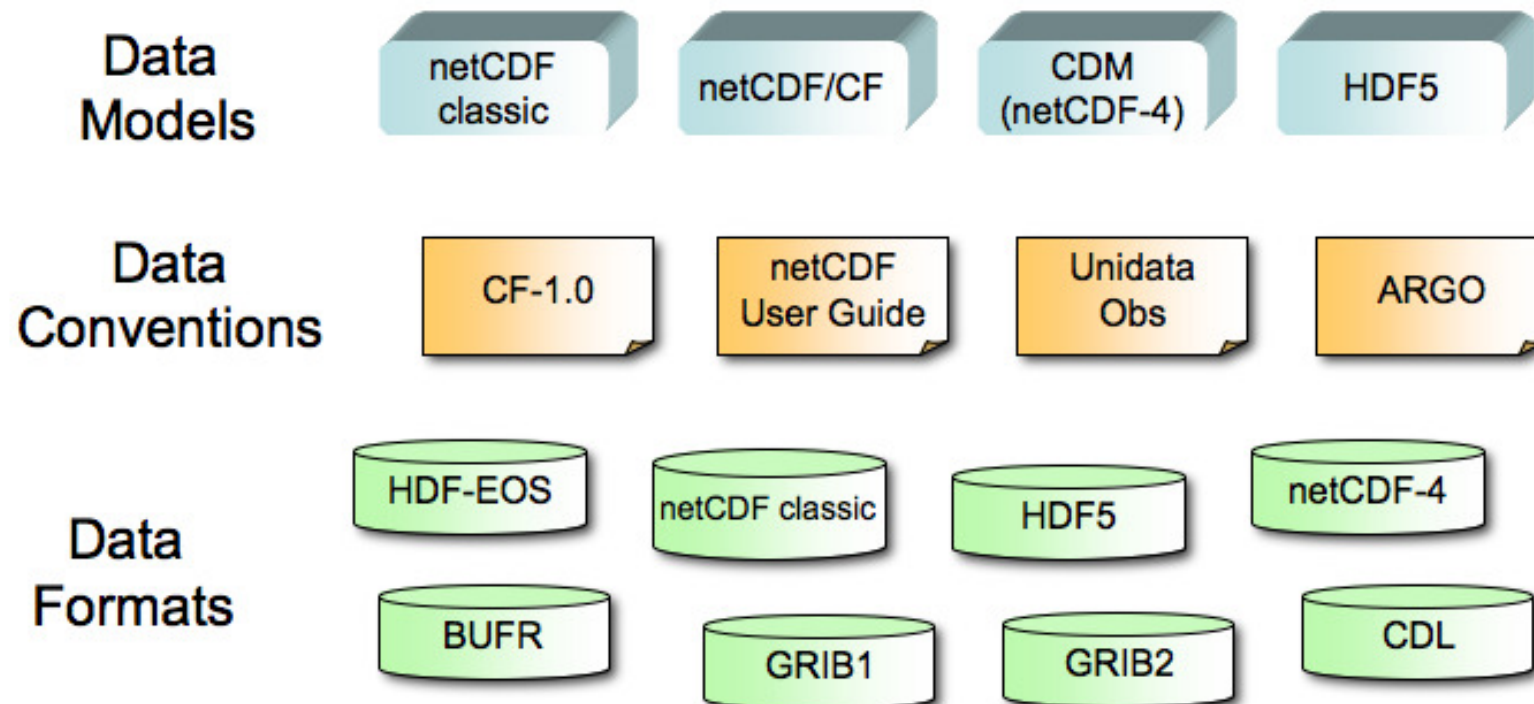


- The netCDF niche is array-oriented scientific data.
  - Uses portable files as unit of self-describing data (unlike databases)
  - Emphasizes efficient direct access to data within files (unlike XML)
  - Provides a multidimensional array abstraction for scientific applications (unlike databases and XML)
  - Avoids dependencies on external tables and registries (unlike GRIB and BUFR)
  - Emphasizes simplicity over power (unlike HDF5)
  - Has built-in client support for network access to structured data from servers
  - Has a large enough community of users to foster development of:
    - support in many third-party applications
    - third-party APIs for other programming and scripting languages
    - community conventions, such as Climate and Forecast (CF) metadata conventions

# NetCDF



- NetCDF has changed over time, so it includes the following:
  - Two data models
    - classic model (netCDF-1, netCDF-2, netCDF-3)
    - enhanced model (netCDF-4)
  - Two formats with variants
    - classic format and 64-bit offset variant for large files
    - netCDF-4 (HDF5-based) format and classic model variant
  - Two independent flavors of APIs
    - C-based interfaces (C, C++, Fortran-77, Fortran-90, Perl, Python, Ruby, Matlab, ...)
    - Java interface
- However, newer versions support:
  - all previous netCDF data models
  - all previous netCDF formats and their variants
  - all previous APIs
  - Files written through one language API are readable through other language APIs.





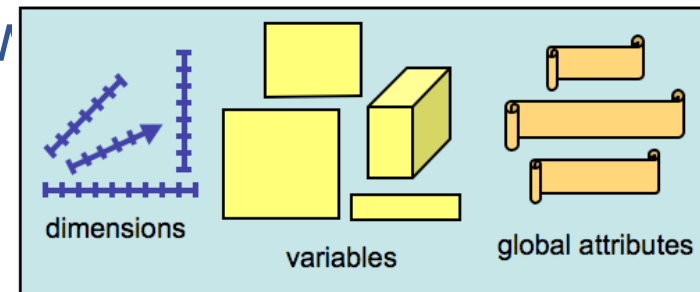
# NetCDF

- Common data model
  - Variables: N-dimensional arrays of char, byte, short, int, float, double
  - Dimensions: Name and length
  - Attributes: Annotations and other metadata
  - Groups: Hierarchical, similar to directories
  - User-defined types
- Parallel functionality
  - Parallel HDF5
    - NetCDF4 version
  - Parallel NetCDF (PNetCDF)
    - Separate library, can write NetCDF 3 (and below) files in parallel
    - Later versions of NetCDF 4 also include some PNetCDF functions

# NetCDF file



- NetCDF files are containers for Dimensions, Variables, and Global Attributes
- File (dataset) contains the following
  - path name
  - dimensions\*
  - variables\*
  - global (file-level) attribute\*
  - data values associated with the variables.\*
  - (\*optional)
- enhanced data model can contain multiple groups
  - group -> dataset
  - groups can be nested



# NetCDF file



```
netcdf pres_temp_4D {  
    dimensions:  
        level = 2 ;  
        latitude = 6 ;  
        longitude = 12 ;  
        time = UNLIMITED ;  
  
    variables:  
        float latitude(latitude);  
            latitude:units = "degrees_north" ;  
        float longitude(longitude) ;  
            longitude:units = "degrees_east" ;  
        float pressure(time, level, latitude, longitude) ;  
            pressure:units = "hPa" ;  
        float temperature(time, level, latitude, longitude) ;  
            temperature:units = "celsius" ;  
  
    data:  
        latitude = 25, 30, 35, 40, 45, 50 ;  
        longitude = -125, -120, ... ;  
        pressure = 900, 901, 902, ... ;  
        temperature = 9, 10, 11, ...;  
}
```

# NetCDF dimensions



- Specify variable shapes, common grids, and co-ordinate systems
  - Has a name and length
  - can be used by multiple variables
  - can associated with *coordinate variables* to identify coordinate axes.
- classic netCDF
  - at most one dimension can have the *unlimited* length (record dimension)
- enhanced netCDF
  - multiple dimensions can have the unlimited length.



# Variables



- Variables define the things that hold data:
  - Has a name, type, shape, can have attributes, and values.
  - Type:
    - Classic NetCDF type is the *external type* of its data as represented on disk, i.e.
      - char
      - byte (8 bits)
      - short (16 bits)
      - int (32 bits)
      - float (32 bits)
      - double (64 bits)
    - Enhanced NetCDF
      - Adds unsigned type; ubyte, ushort, uint, uint64
      - Adds int64 (64 bits), string (variable-length string of characters)
      - User defined types
  - Shape:
    - list of dimensions.
    - no dimensions: a *scalar variable* with only one value
    - 1 dimension: a 1-D (vector) variable
    - 2 dimensions: a 2-D (matrix or grid) variable
  - Attribute:
    - specify properties, i.e. units



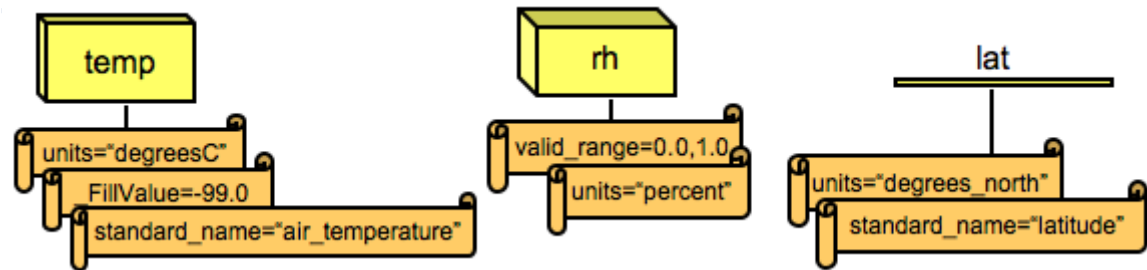
# Attributes



- Metadata about variables or datasets
- Attribute has:
  - Name
  - Type (same as variable types)
  - Values
- Can have scalar or 1-D values
- Cannot be nes

## When to use attributes

- intended for metadata for single values, strings, or small 1-D arrays
- atomic access, must be written or read all at once
- values typically don't change after creation
- length specified when created
- attributes are read when file is opened



# Co-ordinate variables



- Variable with same name as a dimension
  - By convention these specify physical co-ordinate (i.e. lat, lon, level, time, etc...) associated with that dimension
  - Not special in NetCDF, but often interpreted by programs that use NetCDF as special.
  - Allows indexing through position on dimension and matching to co-ordinates

# CDL (Common Data Language)



- Human readable notation for NetCDF datasets and data
  - Obtain from NetCDF file using the `ncdump` program

```
netcdf example {    // example of CDL notation
    dimensions:
        x = 3 ;
        y = 8 ;
    variables:
        float rh(x, y) ;
            rh:units = "percent" ;
            rh:long_name = "relative humidity" ;
    // global attributes
        :title = "simple example, lacks some conventions" ;
    data:
        rh =
            2, 3, 5, 7, 11, 13, 17, 19,
            23, 29, 31, 37, 41, 43, 47, 53,
            59, 61, 67, 71, 73, 79, 83, 89 ;
}
```

# NetCDF utilities



- `ncdump`
  - Produce CDL version of NetCDF file
  - Dump everything, or subset, or just metadata, show indices in C or FORTRAN order, etc...
- `ncgen`
  - Generate NetCDF file from CDL version
  - Generate C, FORTRAN, or Java program which would produce the NetCDF file
- `ncdump` and `ncgen` let you edit NetCDF files manually, or create the program structure that will read/write a NetCDF file in the format you desire automatically
- `nccopy`
  - Copy NetCDF file to new file
  - Can compress and change file format (i.e. classic to enhanced)
- `nc-config`
  - Generate flags necessary to link a program with NetCDF, i.e.:  

```
cc `nc-config --cflags` myapp.c -o myapp `nc-config --libs`  
f95 `nc-config --fflags` yrapp.f -o yrapp `nc-config --flibs`
```

# NetCDF programming interfaces



- NetCDF APIs
  - C, FORTRAN 77, FORTRAN 90, C++, Perl, Java, Python, Ruby, NCL, Matlab, Objective C, Ada, R
  - Some of these are third party
- C interface is used as the core of all but the Java interface

# C API example



```
#include <netcdf.h>

...

int ncid, x_dimid, y_dimid, varid;
int dimids[NDIMS];
int data_out[NX][NY];

...

if ((retval = nc_create(FILE_NAME, NC_CLOBBER, &ncid))) {
    printf("Error: %s\n", nc_strerror(retval));
    exit(1);
}

nc_def_dim(ncid, "x", NX, &x_dimid);
nc_def_dim(ncid, "y", NY, &y_dimid);
dimids[0] = x_dimid;
dimids[1] = y_dimid;
nc_def_var(ncid, "data", NC_INT, NDIMS, dimids, &varid);
nc_enddef(ncid);
nc_put_var_int(ncid, varid, &data_out[0][0]);
nc_close(ncid)
```

# F90 API example



```
use netcdf

integer :: ncid, varid, dimids(NDIMS)
integer :: x_dimid, y_dimid

call check( nf90_create(FILE_NAME, NF90_Clobber, ncid) )
call check( nf90_def_dim(ncid, "x", NX, x_dimid) )
call check( nf90_def_dim(ncid, "y", NY, y_dimid) )
dimids = (/ y_dimid, x_dimid /)
call check( nf90_def_var(ncid, "data", NF90_INT, dimids, varid) )
call check( nf90_enddef(ncid) )
call check( nf90_put_var(ncid, varid, data_out) )
call check( nf90_close(ncid) )

contains

subroutine check(status)
  integer, intent ( in) :: status

  if(status /= nf90_noerr) then
    print *, trim(nf90_strerror(status))
    stop "Stopped"
  end if
end subroutine check
```

# Java API example

```
import ucar.ma2.*;
import ucar.nc2.*;

NetcdfFileWriter dataFile = null;
try {
    dataFile = NetcdfFileWriter.createNew(NetcdfFileWriter.Version.netcdf3, filename);
    Dimension xDim = dataFile.addDimension(null, "x", NX);
    Dimension yDim = dataFile.addDimension(null, "y", NY);
    List<Dimension> dims = new ArrayList<>();
    dims.add(xDim);
    dims.add(yDim);
    Variable dataVariable = dataFile.addVariable(null, "data", DataType.INT, dims);
    dataFile.create();
    dataFile.write(dataVariable, dataOut);
} catch (IOException e) {
    e.printStackTrace();
} catch (InvalidRangeException e) {
    e.printStackTrace();
} finally {
    if (null != dataFile)
        try {
            dataFile.close();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
}
```





# Python API example



```
#from netCDF4_classic import Dataset
#from numpy import arange, dtype
nx = 6; ny = 12
ncfile = Dataset('simple_xy.nc','w')
data_out = arange(nx*ny) # 1d array
data_out.shape = (nx,ny) # reshape to 2d array.
ncfile.createDimension('x',nx)
ncfile.createDimension('y',ny)
data =
ncfile.createVariable('data',dtype('int32').char,('x','y'
'))
data[:] = data_out
ncfile.close()
print '*** SUCCESS writing example file simple_xy.nc!'
```

# C++ API example



```
#include <netcdf>

using namespace netCDF;
using namespace netCDF::exceptions;

try
{
    NcFile dataFile("simple_xy.nc", NcFile::replace);

    NcDim xDim = dataFile.addDim("x", NX);
    NcDim yDim = dataFile.addDim("y", NY);

    vector<NcDim> dims;
    dims.push_back(xDim);
    dims.push_back(yDim);

    NcVar data = dataFile.addVar("data", ncInt, dims);
    data.putVar(dataOut);

    return 0;
}
catch(NcException& e)
{
    {e.what();
    return NC_ERR;
}
}
```

# High-performance NetCDF



- Enhanced NetCDF (version 4 and beyond)
  - Built on HDF5 for parallel/high performance I/O
  - Files need to be stored in HDF5 format

```
#include "netcdf.h"
#include "hdf5.h"

MPI_Comm comm = MPI_COMM_WORLD;
MPI_Info info = MPI_INFO_NULL;
int ncid, vlid, dimids[NDIMS];
size_t start[NDIMS], count[NDIMS];
res = nc_create_par(FILE, NC_NETCDF4|NC_MPIIO, comm, info, &ncid);
res = nc_def_dim(ncid, "d1", DIMSIZE, dimids);
res = nc_def_dim(ncid, "d2", DIMSIZE, &dimids[1]);
res = nc_def_var(ncid, "v1", NC_INT, NDIMS, dimids, &vlid);
res = nc_enddef(ncid);
start[0] = mpi_rank * DIMSIZE/mpi_size;
start[1] = 0;
count[0] = DIMSIZE/mpi_size;
count[1] = DIMSIZE;
res = nc_var_par_access(ncid, vlid, NC_INDEPENDENT);
res = nc_put_vara_int(ncid, vlid, start, count, &data[mpi_rank*QTR_DATA]);
res = nc_close(ncid);
MPI_Finalize();
```

# Parallel NetCDF



- PNetCDF
  - Separate library, add parallel I/O library to support parallel I/O in NetCDF (CDF-1 and CDF-2)
  - Also supports extended CDF-2 (CDF-5)
  - Only functionality that will support parallel I/O for NetCDF format files
  - Some PnetCDF functionality also integrated into later versions of NetCDF 4

```
ret = ncmpi_create(MPI_COMM_WORLD, argv[1],
                  NC_CLOBBER|NC_64BIT_OFFSET,MPI_INFO_NULL,&ncfile);

ret = ncmpi_def_dim(ncfile, "d1", nprocs, &dimid);
ret = ncmpi_def_var(ncfile, "v1", NC_INT, ndims, &dimid, &varid1);
ret = ncmpi_def_var(ncfile, "v2", NC_INT, ndims, &dimid, &varid2);
ret = ncmpi_put_att_text(ncfile, NC_GLOBAL, "string", 13, buf);
ret = ncmpi_enddef(ncfile);
ret = ncmpi_put_vara_int_all(ncfile, varid1, &start, &count, &data);
ret = ncmpi_put_vara_int_all(ncfile, varid2, &start, &count, &data);
ret = ncmpi_close(ncfile);
MPI_Finalize();
```

# NetCDF on ARCHER



- We have three versions of NetCDF on ARCHER all available through modules:
  - NetCDF version 4 (Serial NetCDF)
    - module: cray-netcdf: versions: **4.3.2** 4.3.0, 4.3.1, 4.3.2
  - NetCDF version 4 built with HDF5 parallel functionality
    - module:cray-netcdf-hdf5parallel: versions: **4.3.2** 4.3.0, 4.3.1, 4.3.2
  - Parallel NetCDF (Separate parallel library that can write NetCDF files in parallel)
    - module: cray-parallel-netcdf: versions: **1.5.0** 1.3.1.1, 1.4.0, 1.4.1, 1.5.0



What is HDF5?



## Hierarchical Data Format (version 5)

From [www.hdfgroup.org](http://www.hdfgroup.org):

*HDF5 is a unique technology suite that makes possible the management of extremely large and complex data collections.*

# What is HDF5?



- A versatile **data model** that can represent very complex data objects and a wide variety of metadata.
- A completely **portable file format** with no limit on the number or size of data objects in the collection.
- A **software library that** runs on a range of computational platforms, from laptops to massively parallel systems, and implements a high-level API with C, C++, Fortran 90, and Java interfaces.
- A rich set of integrated **performance** features that allow for access time and storage space optimizations.
- Tools and applications for managing, manipulating, viewing, and analyzing the data in the collection.

# Data Model



- In very basic terms, HDF is like a directory and file *hierarchy* in a file
- The data model is based on *groups* and *datasets*
  - can think of groups like directories/folders, datasets like files
  - both can have (user-defined) *attributes*



# Portable File Format



- HDF5 files are binary but portable
  - HDF5 model take care of types, endianness etc.

# Why HDF5?

- Structure
- Portability
- Performance
- Free & Open Source!



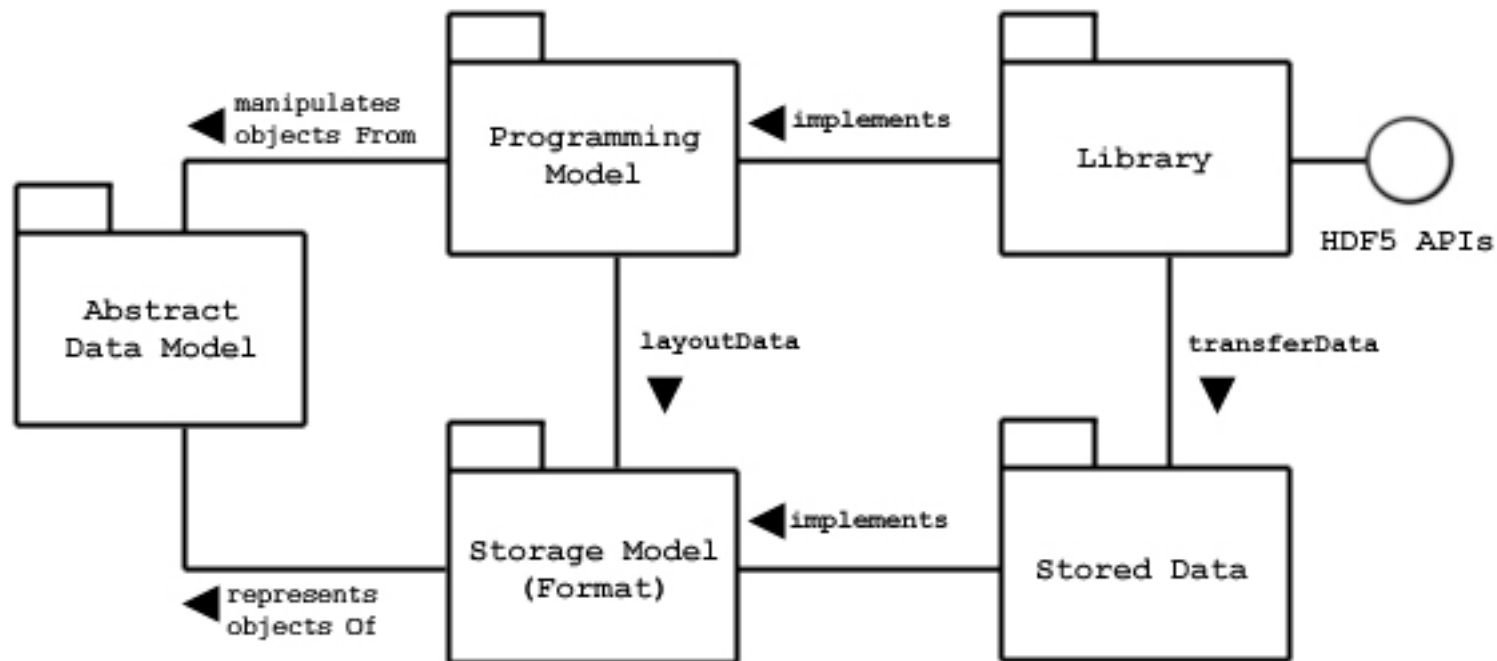
HDF5 files are Self-Describing

Tool Support

Pre-Optimised

Parallel-Ready

# HDF5 Data Model and File Structure



# HDF5 Groups

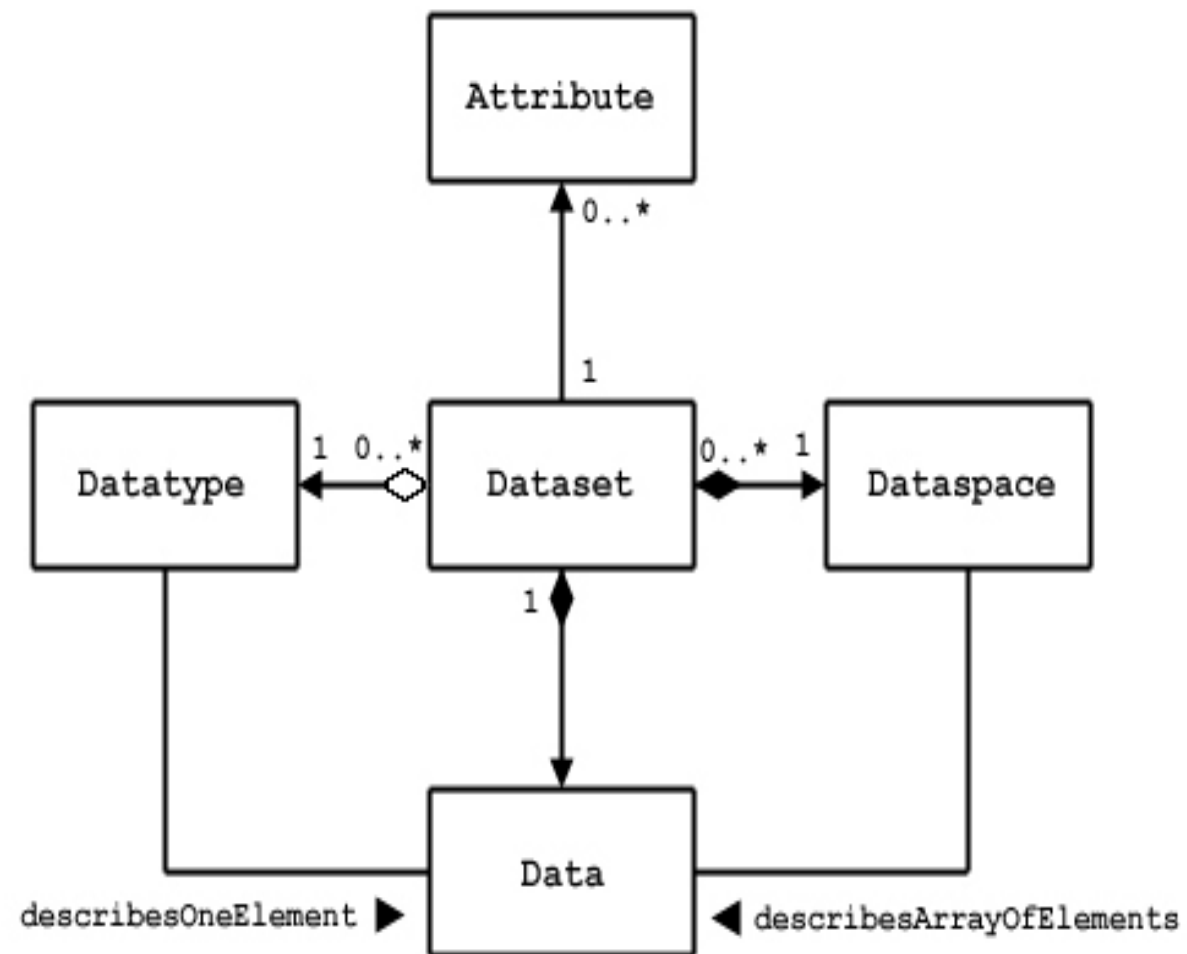


- *HDF5 group*: “a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.”
- A group has two parts:
  - A *group header* containing name & attributes
  - A *group symbol table* listing the group’s contents
- Like UNIX directories, you can identify an object with a path name:
  - / the root group
  - /**foo** a member of the root group, called foo
  - /**foo/zoo** a member of the foo group, called zoo

# HDF5 Datasets



- A dataset has two parts:
  - A *header*
  - A *data array*
- Header provides information on:
  - Name
    - The dataset name. A sequence of alphanumeric characters.
  - Datatypes
    - Atomic, Compound, NATIVE, Named
  - Dataspace
    - Describes the dimensionality (including *unlimited* option)
  - Storage Layout
    - Contiguous, compact, chunked



# HDF5 Attributes



- *Attributes* are small named datasets that are attached to primary datasets, groups, or named datatypes
- Name, value pairs
  - Value can have multiple entries of the same datatype
- There's a separate API for attribute read/write
- Excessively large attribute sets will impact performance

# The HDF5 API



- **H5F:** File-level access routines
  - e.g. H5Fopen
- **H5G:** Group functions, for creating and operating on groups of objects
  - e.g. H5Gset
- **H5T:** Data**T**ype functions, for creating and operating on simple and compound datatypes to be used as the elements in data arrays
- **H5S:** Data**S**pace functions, which create and manipulate the dataspace in which the elements of a data array are stored



- **H5D:** Dataset functions, which manipulate the data within datasets and determine how the data is to be stored in the file.
- **H5P:** Property list functions, for manipulating object creation and access properties.
- **H5A:** Attribute access and manipulating routines.
- **H5Z:** Compression registration routine.
- **H5E:** Error handling routines.
- **H5R:** Reference routines.
- **H5I:** Identifier routine.

## Example: Create and Close a File



A type defined in HDF5. An HDF ID, used to keep track of objects like files

```
hid_t      file;                                /* identifier */
```

```
/*
```

```
* Create a new file using H5ACC_TRUNC access,
```

```
* default file creation properties, and default file
```

```
* access properties.
```

```
* Then close the file.
```

```
*/
```

```
file = H5Fcreate(FILE, H5ACC_TRUNC, H5P_DEFAULT,  
H5P_DEFAULT);
```

```
status = H5Fclose(file);
```

Allows an existing file (if present) to be overwritten

## Example: Creating a dataset so that data can be written to it




```
hid_t    dataset, datatype, dataspace; /* declare identifiers */
/* Create dataspace: Describe the size of the array and
 * create the data space for fixed size dataset.
 */
dimsf[0] = NX;
dimsf[1] = NY;
dataspace = H5Screate_simple(RANK, dimsf, NULL);
/* Define datatype for the data in the file.
 * We will store little endian integer numbers.
 */
datatype = H5Tcopy(H5T_NATIVE_INT);
status = H5Tset_order(datatype, H5T_ORDER_LE);
/* Create a new dataset within the file using defined
 * dataspace and datatype and default dataset creation
 * properties.
 */
dataset = H5Dcreate(file, DATASETNAME, datatype, dataspace, H5P_DEFAULT);
```

ISG17: Data Storing and Input/Output

## Example: Write data to a file



```
/*  
 * Write the data to the dataset using default transfer  
 * properties.  
 */  
status = H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL,  
                  H5S_ALL, H5P_DEFAULT, data);
```



Here's the data itself, stored as a  
standard array of ints in C.

## Example: Read data from a file



```
/*  
 * Write the data to the dataset using default transfer  
 * properties.  
 */  
status = H5Dread(dataset, H5T_NATIVE_INT, H5S_ALL,  
                 H5S_ALL, H5P_DEFAULT, data);
```

- Exactly analogous to write!

# Hyperslabs



- Hyperslabs are portions of datasets

start = (0,1)

stride = (4,3)

count = (2,4)

block = (3,2)

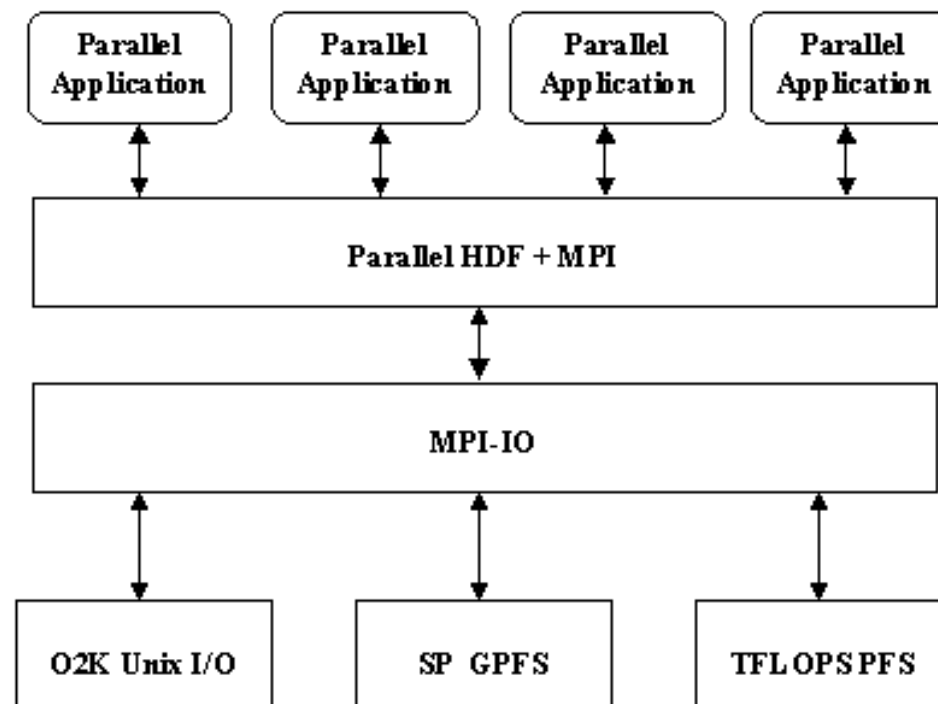
Hyperslab selection

	x	x		x	x		x	x		x	x
	x	x		x	x		x	x		x	x
	x	x		x	x		x	x		x	x
	x	x		x	x		x	x		x	x
	x	x		x	x		x	x		x	x
	x	x		x	x		x	x		x	x

# Parallel HDF5



- Designed to work with MPI and MPI-IO
- Parallel HDF5 files are compatible with serial HDF5 files and sharable between different serial and parallel platforms
- Parallel HDF5 had to be designed to have a single file image to all processes, rather than having one file per process. Having one file per process can cause expensive post processing, and the files are not usable by different processes.
- A standard parallel I/O interface had to be portable to different platforms.



User Applications

HDF library

Parallel I/O layer

Parallel Filesystems



# What software to use?



- This all assumes you are interested in parallel computing
- If raw performance is biggest issue for you
  - MPI-I/O
- If metadata/storage format is biggest issue for you
  - HDF5
- If you want to integrate with earth science tools
  - NetCDF

# What I/O strategy to use



- Small numbers of MPI processes:
  - Single file per process
  - Implement tools to redistribute for different core counts
  - What is “small number”?
- General parallel programs:
  - Single file parallel I/O
  - Limit number of files
  - Limit number of I/O operations
  - Collective important
- Large scale parallel programs or in-situ data analytics
  - Sub I/O
  - Implement data gathering in standard MPI
  - Use MPI I/O for writing data
  - May need to implement different communicators to allow collective I/O on subset of processes

# Practical



- MPI-I/O hands on exercise
  - See handout:
- Two options
  - Coding: Add MPI operations into skeleton code to make it work
  - Run and analyse results: Run working version on different core counts to see what kind of variability you experience and what approach works best at different scales
- C and Fortran code
  - C and Csol, F and Fsol
  - Xsol has the completed solutions
  - The skeleton code is for:
    - master
    - read
    - readall
  - Individual, Bcast, and Ssend are already completed
  - Only timing/parallelising read operations.