NetCDF for Developers and Data Providers

Russ Rew, UCAR Unidata
ICTP Advanced School on High Performance and Grid Computing
14 April 2011
Overview

• Background and motivation
• What is netCDF?
• Data models
• Utilities: ncdump, ngen, nccopy
• Exercises
  • Application Programming Interfaces (API’s)
  • Remote access and OPeNDAP
  • Chunking and compression
  • Parallel I/O
Application Programming Interfaces (API’s)
Programming interfaces to netCDF data

All netCDF APIs are currently implemented over either the C or Java library.

- NetCDF C interface was first API, developed in 1988
- Fortran-77 interface added as a thin layer over C library
- Interfaces for Java, Perl, and first C++ library developed at Unidata
- Collaborated on a Fortran-90 interface
- Other contributed C-based interfaces include Python, Perl, Ruby, NCL, Matlab, IDL, R, Objective C, Ada, and new C++ API for netCDF-4
- Java is most advanced netCDF API, best for use on servers
The C API

- Core library on which all non-Java APIs are built
- Strengths:
  - Well-documented: C Users Guide, man pages for reference
  - Comprehensively tested when library built from source
  - Good support: answers for many questions available
  - Many users: one of the most widely used netCDF interfaces.
  - Type-safe interfaces avoid “void *” arguments and catch compile-time errors
  - The `ncgen` utility can generate C code from CDL
#include <netcdf.h>
...
/* Handle errors by printing an error message and exiting */
#define ERR(e) {printf("Error: \%s\n", nc_strerror(e)); exit(ERRCODE);}
...
/* netCDF file ID and variable ID */
int ncid, varid;
/* array into which we will read values of 2D netCDF variable */
double rh_array[NLAT][NLON];
...
/* Open file with read-only access, indicated by NC_NOWRITE flag */
if ((retval = nc_open("foo.nc", NC_NOWRITE, &ncid)))
    ERR(retval);
/* Get the id of the variable named "rh" */
if ((retval = nc_inq_varid(ncid, "rh", &varid)))
    ERR(retval);
/* Read variable "rh" as doubles, rh_array must be big enough! */
if ((retval = nc_get_var_double(ncid, varid, &rh_array[0][0])))
    ERR(retval);
/* Close the file, freeing all resources. */
if ((retval = nc_close(ncid)))
    ERR(retval);
The Fortran-90 API

Provides current Fortran support for modelers and scientists

• **Strengths:**
  - Overloads var_put and var_get functions for all types and shapes
  - Optional arguments simplify API
  - Many users: one of the most widely used netCDF interfaces

• **Other characteristics**
  - Currently implemented in Fortran-90 as thin layer on Fortran-77 library
  - No **ncgen** utility support (yet) for generating F90 code from CDL
use netcdf

...  
! check(status) function prints error message and exits
...

! netCDF ID for the file and data variable
integer :: ncid, varid

! array into which we will read values of 2D netCDF variable
double rh_array[NLON][NLAT]  ! reversed index order from CDL

! Open file with read-only access, indicated by NF90_NOWRITE flag
call check( nf90_open("foo.nc", NF90_NOWRITE, ncid) )

! Get the id of the variable named "rh"
call check( nf90_inq_varid(ncid, "rh", varid) )

! Read whole variable "rh" as double, rh_array must be big enough!
call check( nf90_get_var(ncid, varid, rh_array) )
...

! Close the file, freeing all resources.
call check( nf90_close(ncid) )
Language independence

- The netCDF data model and format are language-independent.
  - Data written from any language interface can be read from any other language interface
- Fortran API uses Fortran dimension row-major order, 1-based indexing
- Unlike netCDF, CDL is not quite language neutral

variables:
  float rh(time, lat, lon) ;
  real rh(lon, lat) ;

CDL

! time slice
real rh(lon, lat) ;

Fortran
Equivalent examples from various APIs

- Examples of complete sample programs for writing and reading netCDF data from various language interfaces are available from the netCDF program examples page http://www.unidata.ucar.edu/netcdf/examples/programs/

Fortran-77  Fortran-90  C
MATLAB  IDL  C++  Perl  Python  Java
Java netCDF library architecture

Scientific Feature Types

Datatype Adapter

NetcdfDataset

CoordSystem Builder

NetcdfFile

I/O service provider

THREDDS

Catalog.xml

NcML

OPeNDAP

NetCDF-3

NetCDF-4

HDF5

Nexrad

NIDS

GRIB

GINI

DMSP
C netCDF library architecture
Remote access and OPeNDAP
Alternatives for remote data access

• Whole file access
  – ftp, scp, sftp, http for “small” (< 10 GB) files
  – tar for directories
  – gridFTP or Globus Online for large files: fast, parallel, requires certificate

• Subset access
  – OPeNDAP (open network data access protocol)
  – Open Geospatial Consortium services: WCS, WMS, WFS, ...
  – Database queries
When is subset access important?

- For remote accesses to small parts of large files
  - A few variables out of many
  - A small geographic region from a global dataset
  - A small time range from a long time series

- When visualizing or analyzing data subsets
  - One 2D level of atmosphere or ocean
  - One cross section of multidimensional data

- When files are archived at a granularity too large for use or downloading
What are and DAP?

- **DAP** is a widely supported data access protocol for accessing remote science data over http.
- The standard and reference client/server software are maintained by the **OPeNDAP** organization at [http://www.opendap.org/](http://www.opendap.org/).
- DAP was designed for accessing a wide variety of data sources and formats.
- “DAP” and “OPeNDAP” are often used interchangeably.
OPeNDAP and netCDF

- Unidata has merged OPeNDAP client access into both Java- and C-based netCDF libraries.
- This supports transparent remote access to DAP Data Servers through the netCDF API.
- Remote access allows any application linked to the netCDF library to retrieve subsets of data stored on DAP servers across the Internet.
- Only the minimal amount of needed data will be accessed
  - DAP can be much faster than whole file access, such as FTP
• DAP data access is analogous to accessing a web page through a web browser

DAP client-server architecture
Specifying a DAP data source

- Use a URL that refers to the DAP server containing the data
- Used in place of a file name in application or netCDF API call
- Example for whole file: `http://test.opendap.org/opendap/data/nc/3fnoc.nc`
- Example for 3 variables out of file
  `http://test.opendap.org/opendap/data/nc/3fnoc.nc?lat,lon,time`
- Example for subarray of one variable
  `http://test.opendap.org/opendap/data/nc/3fnoc.nc?u[2:5][0:4][0:5]`
- When used in command-line, URL should usually be quoted:
  `ncdump "http://test.opendap.org/opendap/data/nc/3fnoc.nc?u"`
Chunking and compression
Motivation for chunking

- **Problem**: reading a small amount of data along the wrong direction in a multidimensional variable can be *very* slow:
Motivation for chunking

- **Solution**: storing the data in "chunks" along each dimension in a multidimensional variable makes access along any dimension similar.
Example: accessing cross-sections

- “Toy” example: accessing a 6 x 6 x 8 array on a system with small disk blocks
- If array is stored contiguously, then (ignoring caching) number of disk accesses needed to
  - read a single x,y 2D cross-section: 1
  - read a single x,z or y,z 2D cross-section: 8
  - read whole array using x,y slices: 8
  - read whole array using x,z or y,z slices: 48
  - read a single 1D vector along x or y axis: 1
  - read a single 1D vector along z axis: 8
  - read whole array using 1D vectors along x or y axis: 8
  - read whole array using 1D vectors along z axis: 288
- Contiguous same as 6 x 6 x 1 chunks, try 3 x 3 x 4 chunks...
Accessing cross-sections with chunking

- Same data: 6 x 6 x 8 array
- If array is stored using 3 x 3 x 4 chunks, then number of disk accesses needed to:
  - read a single x,y 2D cross-section: 4
  - read a single x,z or y,z 2D cross-section: 4
  - read whole array using x,y slices: 32
  - read whole array using x,z or y,z slices: 32
  - read a single 1D vector along x or y axis: 2
  - read a single 1D vector along z axis: 2
  - read whole array using 1D vectors on x or y axis: 96
  - read whole array using 1D vectors along z axis: 72
Accessing cross-sections with chunking

- Same data: 6 x 6 x 8 array

<table>
<thead>
<tr>
<th>Access</th>
<th>Contiguous (disk accesses)</th>
<th>Chunking (disk accesses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D x,y cross-section</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2D x,z or y,z cross-section</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>3D array using x,y slices</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>3D array using x,z or y,z slices</td>
<td>48</td>
<td>32</td>
</tr>
<tr>
<td>1D vector along x or y axis</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1D vector along z axis</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>3D array using x or y vectors</td>
<td>8</td>
<td>96</td>
</tr>
<tr>
<td>3D array using z vectors</td>
<td>288</td>
<td>72</td>
</tr>
</tbody>
</table>
Actual timings accessing cross-sections with chunking

- 432 x 432 x 432 array of floats with chunk sizes of 36 x 36 x 36

<table>
<thead>
<tr>
<th>Access</th>
<th>Contiguous (seconds)</th>
<th>Chunking (seconds)</th>
<th>Slowdown or speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D x,y cross-section write</td>
<td>0.559</td>
<td>1.97</td>
<td>3.5 x slower</td>
</tr>
<tr>
<td>2D x,z cross-section write</td>
<td>18.1</td>
<td>1.5</td>
<td>12 x faster</td>
</tr>
<tr>
<td>2D y,z cross-section write</td>
<td>223</td>
<td>9.55</td>
<td>23 x faster</td>
</tr>
<tr>
<td>2D x,y cross-section read</td>
<td>0.353</td>
<td>1.06</td>
<td>3 x slower</td>
</tr>
<tr>
<td>2D x,z cross-section read</td>
<td>6.22</td>
<td>1.45</td>
<td>4.3 x faster</td>
</tr>
<tr>
<td>2D y,z cross-section read</td>
<td>77.1</td>
<td>7.68</td>
<td>10 x faster</td>
</tr>
</tbody>
</table>

- Fast accesses slow down a little, slow accesses speed up a lot
Benefits of chunking

• As a general principle, organize data for readers, not writer
  – Chunking should match most common access patterns
  – Chunking may also improve compression
• Chunked storage can provide significant performance benefits
  – Allows efficient access to multidimensional data along multiple axes
  – Default chunking parameters make access performance similar along different dimensions
  – In netCDF-4 (with HDF5 storage) variables may be chunked independently with custom chunk sizes
  – Can improve I/O performance for large arrays and compressed variables
Compression: why not just use zip?

- Unix utilities are available for compressing whole files, e.g. bzip2, gzip, zip, compress. Why not just use one of those?
  - Accessing data from a compressed file requires uncompresing whole file first
  - So accessing a small amount of data from a large compressed file can be very slow
  - Changing one value in a compressed file requires uncompressed it, writing the new value, and recompressing it

- **Solution:** chunking and per-variable compression
Compression in netCDF-4

- Readers access data from compressed variables transparently, without needing to know they are compressed.
- Compressed variables are stored with chunked storage.
- Each chunk is compressed or uncompressed independently.
- Permits efficient access to small subsets of a large compressed variable without uncompressing entire variable.
- Per-variable chunk caches keep recently accessed chunks uncompressed.
- Better compression can be achieved with custom chunking.
  - example: horizontal layers of the atmosphere for a variable that is fairly uniform within a layer, such as temperature.
  - Per-variable compression means variables may be compressed independently.
Benefits of netCDF-4 classic model format

- **What is the netCDF-4 classic model format?**
  - Uses classic data model for simplicity, compatibility
  - Uses netCDF-4 (HDF5-based) storage for performance features

- **This format has become popular for several reasons:**
  - Easy to use: specify format only in netCDF create call
  - Features like chunking, compression available to writers
  - Data written in this format can be read transparently by old programs, after relinking to new library

- **Supports easier transition from netCDF-3**
Parallel I/O
Why parallel I/O?

- Gets around some input/output bottlenecks in multi-processor systems
- Lets each processor read and write data independently
What is parallel I/O?

- A parallel I/O file system is required for much improvement in I/O throughput
- NetCDF-4 works with the Message Passing Interface, version 2 (MPI2)
- Any supercomputer will have an MPI2 library
- For netCDF testing we use the MPICH2 library
The Argonne parallel-netCDF package

- **parallel-netcdf** (formerly "pnetcdf") from Argonne Labs and Northwestern University can be used for parallel I/O with classic netCDF data.

- Not Unidata software, but well-tested and maintained

- Uses MPI I/O to perform parallel I/O, a complete rewrite of the core C library using MPI I/O

- Implements different API from netCDF, making portability with other netCDF code a problem

- However, netCDF-4 can now use the parallel netCDF library for classic and 64-bit offset files using parallel I/O

- Use the NC_PNETCDF flag (or NF90_PNETCDF for Fortran):

```c
if (nc_create_par(file_name, NC_PNETCDF, mpicomm, info, &ncid)) ERR;
```
Parallel I/O in netCDF-4

- Provides the parallel I/O features of HDF5 with a netCDF API
- Allows $n$ processes on $m$ processors to read and write netCDF data, where $n$ and $m$ are integers usually < 10K
- Requires a library implementing MPI2, for example MPICH2
- HDF5 must be built with --enable-parallel
- Typically CC environment variable is set to mpicc, and FC to mpifc. You must build HDF5 and netCDF-4 with same compiler and compiler options.
- The netCDF configure script will detect the parallel capability of HDF5 and build the netCDF-4 parallel I/O features automatically.
- For parallel applications you must include "netcdf_par.h" before netcdf.h.
- Parallel tests output can tell you a lot about your parallel platform.
Using parallel I/O in netCDF-4

- Special `nc_create_par` and `nc_open_par` functions are used to create/open a netCDF file.
- The files they open are normal NetCDF-4/HDF5 files, but these functions also take MPI parameters.
- Parallel access is not a characteristic of data file, but the way it was opened.

```c
external int
nc_create_par(const char *path, int cmode,
               MPI_Comm comm, MPI_Info info, int *ncidp);

external int
nc_open_par(const char *path, int mode,
             MPI_Comm comm, MPI_Info info, int *ncidp);
```
Collective and independent operations

- Some netCDF operations are **collective** (must be done by all processes at the same time)
- Others are **independent** (can be done by any process at any time)
- All netCDF metadata writing operations are collective. That is, all creation of groups, types, variables, dimensions, or attributes.
- Data reads and writes may be independent (the default) or collective.
- To make writes to a variable collective, call the

```c
if( nc_var_par_access(ncid, varid, NC_COLLECTIVE) )
   ERR;
```
Conclusion

- Data providers may begin to use compression/chunking with confidence that most users and software can read it transparently, after relinking with netCDF-4.
- Developers may adapt software to netCDF-4 format by relinking.
- Developers may adapt software to enhanced data model incrementally, with examples that such adaptation is practical.
- Upgrading software to make use of higher-level abstractions of netCDF-4 enhanced data model has significant benefits:
  - Data providers can use more natural representation of complex data semantics.
  - More natural conventions become possible.
  - End users can access other types of data through netCDF APIs.
- As we keep pushing common tasks into libraries, scientists can focus on doing science instead of data management.
Thank you!