Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Portable Parallel I/O

Parallel netCDF

May 26, 2014 | Wolfgang Frings, Florian Janetzko, Michael Stephan

JÜLICH
FORSCHUNGSZENTRUM

Outline

Introduction

Basic file handling

Advanced file operations

Exercises

JÜLICH
FORSCHUNGSZENTRUM

Introduction to Parallel netCDF

- netCDF is a portable, self-describing file format developed by Unidata at UCAR (University Cooperation for Atmospheric Research)
- netCDF does not provide a parallel API prior to 4.0
  - Classic and 64-bit offset file format
- pnetCDF is maintained by Argonne National Laboratory
  - `http://trac.mcs.anl.gov/projects/parallel-netcdf/`

Header files

C/C++

```
#include <pnetcdf.h>
```

- Contain definition of
  - constants
  - functions

Header files

Fortran

```
! include 'pnetcdf.inc'
#include "pnetcdf.inc"
```

- Contain definition of
  - constants
  - functions

Terms and definitions

Dimension

An entity that can either describe a physical dimension of a dataset, such as time, latitude, etc., as well as index to sets of stations.

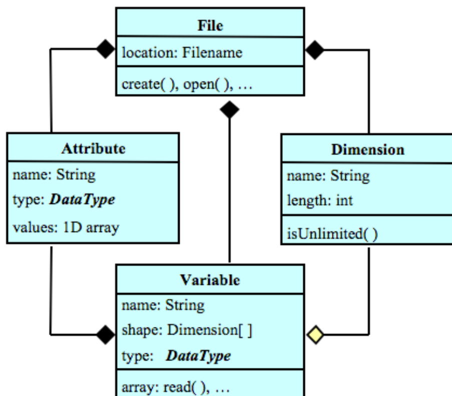
Variable

An entity that stores the bulk of the data. It represents an *n*-dimensional array of values of the same type.

Attribute

An entity to store data on the datasets contained in the file or the file itself. The latter are called *global attributes*.

NetCDF Classic model



Variables and attributes have one of six primitive data types.

DataType
char
byte
short
int
float
double

A file has named variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One dimension may be of unlimited length.

source: Hartnett, E., 2010-09: NetCDF and HDF5 - HDF5 Workshop 2010.

Naming conventions

Dimensions, variables, attributes

- Sequence of alphanumeric characters, underscore '_', period '.', plus '+', hyphen '-', or at sign '@'
- Must begin with a letter or underscore
  - Name with underscores are reserved for system use
- Names are case sensitive
- Other conventions may restrict names even more

JÜLICH FORSCHUNGSZENTRUM

Dimensions

- Can represent a physical dimension like time, height, latitude, longitude, etc.
- Can be used to index other quantities, e.g., station number
- Have a name and length
- Can have either a fixed length or 'UNLIMITED'
  - In *classic* and *64bit offset* files at most one
- Used to define the shape of variables
- Can be used more than once in a variable declaration
  - Use only more than once, where semantically useful

Variables

- Store the bulk data in the dataset
- Regarded as *n*-dimensional array
  - Scalar values represented as 0-dimensional arrays
- Have a name, type and shape
  - Shape is defined through dimensions
- Once created, cannot be deleted or altered in shape
- Variable type must be one of the basic types
  - byte, character, short, int, float, double
- Variables with one unlimited dimension are called *record variables*, otherwise *fixed variables*
- A position along a dimension can be specified as index
  - Starting at 0 in C and 1 in Fortran

Coordinate variables

- Variables can have the same name as dimensions
- Have no special semantic in netCDF itself
- By convention, applications using netCDF should treat them in a special way

- Usually describes a coordinate corresponding to that dimension
- Each coordinate variable is a vector that's shape is defined by the dimension of the same name
- Might provide a more convenient way to access the data
  - By convention, current applications assume coordinate variables to be numeric and strictly monotonic

Attributes

- Used to store meta data of variables or the complete data set (global attributes)
- Have a name, a type, a length, and a value
- Treated as vector
  - Scalar values a single-element vectors
- Can be deleted and changed in shape at any time
- Please adhere existing conventions for attributes

Attribute Conventions

`units` – character string that specifies the units used for a variable

`long_name` – long descriptive name for a variable

`valid_min` – value specifying the minimum valid value for a variable

`valid_max` – value specifying the maximum valid value for a variable

`valid_range` – vector of two numbers specifying the minimum and maximum valid value for a variable

`...`

For more, please read the *Appendix B: Attribute Conventions* of the netCDF User Guide

Datatypes

The netCDF classic and 64-bit offset file format only support basic types

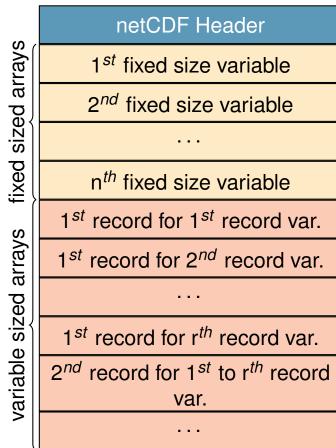
C	Fortran	Storage
NC_BYTE	nf_byte	8-bit signed integer
NC_CHAR	nf_char	8-bit unsigned integer
NC_SHORT	nf_short	16-bit signed integer
NC_INT	nf_int	32-bit signed integer
NC_FLOAT	nf_float	32-bit floating point
NC_DOUBLE	nf_double	64-bit floating point

The netCDF file format

netCDF dataset definition

n arrays of fixed dimensions

r arrays with its most significant dimension set to UNLIMITED records are defined by the remaining dimensions



JÜLICH FORSCHUNGSZENTRUM

netCDF file format characteristics

- A netCDF (classic and 64-bit offset format) file consists of three regions
  - Header
  - Non-record variables, multi-dimensional data with fixed size in each dimension
  - Record variables, multi-dimensional data with a single dimension of UNLIMITED size, and the remaining dimensions fixed
- All data is written in big-endian format in an internal format similar to XDR

JÜLICH FORSCHUNGSZENTRUM

Performance Implications

- The header is dense, i.e., changing the header after variables have been added, will result in the copy of all subsequent data
  - Avoid later additions and renaming of netCDF components
  - Use `nc__enddef` to reserve header space
- Record variables are interleaved
  - Using more than one per file will result in non-contiguous buffers, and performance degradation is likely

JÜLICH
FORSCHUNGSZENTRUM

netCDF classic format limitations

- If no unlimited dimension is used, only *one* variable can exceed 2 GiB (but it can be as large as the FS permits)
  - It must be the last variable in the data set
  - The start offset must be less than $2^{31} - 4$ bytes (approx. 2 GiB)
- If the unlimited dimension is used, record variables may exceed 2 GiB in size
  - The start offset of each record variable must be less than $2^{31} - 4$ bytes (approx. 2 GiB)

netCDF 64-bit offset format limitations

- If no unlimited dimension is used, only *one* variable can exceed 2 GiB (but it can be as large as the FS permits)
  - It must be the last variable in the data set
- A data set can contain $2^{32} - 1$ fixed sized variables, each less 4 GiB in size
- A record variable cannot use more than 4 GiB
  - Last record variable can be any size

Outline

Introduction

Basic file handling

Advanced file operations

Exercises

JÜLICH FORSCHUNGSZENTRUM

Workflow: Creating a netCDF data set

- Create a new dataset
  - A new file is created and netCDF is left in define mode
- Describe contents of the file
  - Define dimensions for the variables
  - Define variables using the dimensions
  - Store attributes if needed
- Switch to data mode
  - Header is written and definition of the file content is completed
- Store variables in file
  - Parallel netCDF distinguishes between collective and individual data mode
  - Initially in collective mode, user has to switch to individual data mode explicitely
- Close file

FORSCHUNGSZENTRUM JÜLICH

Creating a file

C/C++

```
int ncmpi_create(MPI_Comm comm, const char* filename, int cmode,
        MPI_Info info, int* ncid)
```

- Call is collective over `comm`
- `ncid` is the id of the internal file handle
- `cmode` must specify at least one of the following
  - `NC_CLOBBER` – Create new file and overwrite, if it existed before
  - `NC_NOCLOBBER` – Create new file only, if it did not exist before
- Choose file format on file creation
  - `NC_FORMAT_CLASSIC` – 32-bit offsets
  - `NC_FORMAT_64BIT` – 64-bit offsets

JÜLICH FORSCHUNGSZENTRUM

Creating a file

Fortran

```
INTEGER NFMPI_CREATE(COMM, FILENAME, CMODE, INFO, NCID)
  CHARACTER*(*) FILENAME
  INTEGER       COMM, MODE, INFO, NCID
```

- Call is collective over `comm`
- `ncid` is the id of the internal file handle
- `cmode` must specify at least one of the following
  - `NF_CLOBBER` – Create new file and overwrite, if it existed before
  - `NF_NOCLOBBER` – Create new file only, if it did not exist before
- Choose file format on file creation
  - `NF_FORMAT_CLASSIC` – 32-bit offsets
  - `NF_FORMAT_64BIT` – 64-bit offsets

Open an existing netCDF data set

C/C++

```
int ncmpi_open(MPI_Comm comm, const char* filename, int omode,
        MPI_Info info, int* ncid)
```

- Call is collective over `comm`
- `ncid` is the id of the internal file handle
- `omode` must specify at least one of the following
  - `NC_WRITE` – Open file for any kind of change to the file
  - `NC_NOWRITE` – Open the file read-only

JÜLICH FORSCHUNGSZENTRUM

Open an existing netCDF data set

Fortran

```
INTEGER NFMPI_OPEN(COMM, FILENAME, OMODE, INFO, NCID)
  CHARACTER*(*) FILENAME
  INTEGER       COMM, OMODE, INFO, NCID
```

- Call is collective over `comm`
- `ncid` is the id of the internal file handle
- `omode` must specify at least one of the following
  - `NF_WRITE` – Open file for any kind of change to the file
  - `NF_NOWRITE` – Open the file read-only

Closing a file

C/C++

```
int ncmpi_close(int ncid)
```

- Close file associated with `ncid`

JÜLICH
FORSCHUNGSZENTRUM

Closing a file

Fortran

```
INTEGER NFMPI_CLOSE(NCID)
  INTEGER NCID
```

- Close file associated with `ncid`

Defining dimensions

C/C++

```
int ncmpi_def_dim(int ncid, const char* name, MPI_Offset len,
        int* dimid)
```

- `name` represents the name of the dimension
- `len` represents the value
  - NC_UNLIMITED will create an unlimited dimension
- Can only be called in *definition mode*

JÜLICH FORSCHUNGSZENTRUM

Defining dimensions

Fortran

```
INTEGER NFMPI_DEF_DIM(NCID, NAME, LEN, DIMID)
  CHARACTER*(*) NAME
  INTEGER       NCID, DIMID
  INTEGER(KIND=MPI_OFFSET_KIND) LEN
```

- `name` represents the name of the dimension
- `len` represents the value
  - NF_UNLIMITED will create an unlimited dimension
- Can only be called in *definition mode*

JÜLICH FORSCHUNGSZENTRUM

Defining variables

C/C++

```
int ncmpi_def_var(int ncid, const char* name, nc_type xtype,
        int ndims, const int* dimids, int* varid)
```

- `xtype` specifies the external type of this variable
- `dimids` is an array of size `ndims`

Defining variables

Fortran

```
INTEGER NFMPI_DEF_VAR(NCID, NAME, XTYPE, NDIMS, DIMIDS, VARID)
  CHARACTER*(*) NAME
  INTEGER,      NCID, XTYPE, NDIMS, VARID
  INTEGER(*)    DIMIDS
```

- `xtype` specifies the external type of this variable
- `dimids` is an array of size `ndims`

Defining attributes

C/C++

```
int ncmpi_put_att_<type>(int ncid, int varid, const char* name,
        nc_type xtype, MPI_Offset len, const <type>*attr)
```

- Puts the attribute `attr` into the data set
- `varid` is the id annotated variable, or 0, if it is a global attribute
- `xtype` specifies the external type of this attribute

JÜLICH FORSCHUNGSZENTRUM

Defining attributes

Fortran

```
INTEGER NFMPI_PUT_ATT_<type>(NCID, VARID, NAME, XTYPE, LEN,
            ATTR)
  <type>        ATTR
  CHARACTER*(*) NAME
  INTEGER       NCID, VARID, XTYPE
  INTEGER(KIND=MPI_OFFSET_KIND) LEN
```

- Puts the attribute `attr` into the data set
- `varid` is the id annotated variable, or 0, if it is a global attribute
- `xtype` specifies the external type of this attribute

JÜLICH FORSCHUNGSZENTRUM

Closing define mode

C/C++

```
int ncmpi_enddef(int ncid)
```

- Ends the definition phase, and switches to collective data mode
- Once variables have been put into the data set, definitions should not be altered

Closing define mode

Fortran

```
INTEGER NFMPI_ENDDEF(NCID)
  INTEGER NCID
```

- Ends the definition phase, and switches to collective data mode
- Once variables have been put into the data set, definitions should not be altered

Writing variables collectively to the file

C/C++

```
int ncmpi_put_vara_<type>_all(int ncid, int varid,
        const MPI_Offset start[], const MPI_Offset count[],
        const <type>* var)
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode

Writing variables collectively to the file

Fortran

```
INTEGER NFMPI_PUT_VARA_<type>_ALL(NCID, VARID, START, COUNT,
            VAR)
  <type>(*) VAR
  INTEGER   NCID, VARID
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode

Writing variables individually to the file

C/C++

```
int ncmpi_put_vara_<type>(int ncid, int varid,
        const MPI_Offset start[], const MPI_Offset count[],
        const <type>* var)
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in individual data mode

Writing variables individually to the file

Fortran

```
INTEGER NFMPI_PUT_VARA_<type>(NCID, VARID, START, COUNT, VAR)
  <type>(*) VAR
  INTEGER   NCID, VARID
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in individual data mode

Reading variables collectively from the file

C/C++

```
int ncmpi_get_vara_<type>_all(int ncid, int varid,
        const MPI_Offset start[], const MPI_Offset count[],
        const <type>* var)
```

- Reads a *slab* of data from the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode

Reading variables collectively from the file

Fortran

```
INTEGER NFMPI_GET_VARA_<type>_all(NCID, VARID, START, COUNT,
            VAR)
  <type>(*) VAR
  INTEGER   NCID, VARID
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

- Reads a *slab* of data from the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode

Reading variables individually from the file

C/C++

```
int ncmpi_get_vara_<type>(int ncid, int varid,
        const MPI_Offset start[], const MPI_Offset count[],
        const <type>* var)
```

- Reads a *slab* of data from the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in individual data mode

Reading variables individually from the file

Fortran

```
INTEGER NFMPI_GET_VARA_<type>(NCID, VARID, START, COUNT, VAR)
  <type>(*) VAR
  INTEGER   NCID, VARID
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

- Reads a *slab* of data from the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in individual data mode

Switching data modes

C/C++

```
int ncmpid_begin_indep_data(int ncid)
```

- Switches from collective data mode to individual data mode

```
int ncmpid_end_indep_data(int ncid)
```

- Switches from individual data mode to collective data mode

JÜLICH
FORSCHUNGSZENTRUM

Switching data modes

Fortran

```
INTEGER NFMPI_BEGIN_INDEP_DATA(NCID)
  INTEGER NCID
```

- Switches from collective data mode to individual data mode

```
INTEGER NFMPI_END_INDEP_DATA(NCID)
  INTEGER NCID
```

- Switches from individual data mode to collective data mode

Example Write (C), Part I

```
/* from pnetcdf tutorial: simple demonstration of pnetcdf text attribute
   on dataset write out rank into 1-d array collectively.  The most basic
   way to do parallel i/o with pnetcdf */
#include <stdlib.h>
#include <mpi.h>
#include <pnetcdf.h>
#include <stdio.h>
static void handle_error(int status){
        fprintf(stderr, "%s", ncmpi_strerror(status));exit(-1);}
int main(int argc, char **argv) {
        int ret, ncfile, nprocs, rank, dimid, varid1, varid2, ndims=1;
        MPI_Offset start, count=1;      int data;
        char buf[13] = "Hello World";

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

        ret = ncmpi_create(MPI_COMM_WORLD, argv[1],
                           NC_WRITE|NC_64BIT_OFFSET,
                           MPI_INFO_NULL, &ncfile);
        if (ret != NC_NOERR) handle_error(ret);
        ret = ncmpi_def_dim(ncfile, "d1", nprocs, &dimid);
        if (ret != NC_NOERR) handle_error(ret);
```

JÜLICH FORSCHUNGSZENTRUM

Example Write (C), Part II

```
    ret = ncmpi_def_var(ncfile, "v1", NC_INT, ndims, &dimid, &varid1);
    if (ret != NC_NOERR) handle_error(ret);
    ret = ncmpi_def_var(ncfile, "v2", NC_INT, ndims, &dimid, &varid2);
    if (ret != NC_NOERR) handle_error(ret);

    ret = ncmpi_put_att_text(ncfile, NC_GLOBAL, "string", 13, buf);
    if (ret != NC_NOERR) handle_error(ret);

    ret = ncmpi_enddef(ncfile); if (ret != NC_NOERR) handle_error(ret);
    start=rank, count=1, data=rank;

    /* in this simple example every process writes its rank to
       two 1d variables */
    ret = ncmpi_put_vara_int_all(ncfile, varid1, &start, &count, &data);
    if (ret != NC_NOERR) handle_error(ret);

    ret = ncmpi_put_vara_int_all(ncfile, varid2, &start, &count, &data);
    if (ret != NC_NOERR) handle_error(ret);
    ret = ncmpi_close(ncfile);
    if (ret != NC_NOERR) handle_error(ret);
    MPI_Finalize();
    return 0;
}
```

Outline

Introduction

Basic file handling

Advanced file operations
- Flexible data mode interface
- Data set inquiry
- Release Info

Exercises

Motivation for flexible API

- Original interface brings a lot of function name cruft with individual function calls for each data type
- The flexible data mode API is used in the background
- The user can also specify the in-memory storage using MPI data types

Writing variables collectively to the file

C/C++

```
int ncmpi_put_vars_all(int ncid, int varid,
        const MPI_Offset start[], const MPI_Offset count[],
        const MPI_Offset stride[], const void* buf,
        int elements, MPI_Datatype datatype)
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode
- `start`, `count`, and `stride` refer to the data in the file
- `buf`, `elements`, and `datatype` refer to the data in memory

JÜLICH FORSCHUNGSZENTRUM

Writing variables collectively to the file

Fortran

```
INTEGER NFMPI_PUT_VARS_ALL(NCID, VARID, START, COUNT,
            STRIDE, BUF, ELEMENTS, DATATYPE)
  <type>(*) BUF
  INTEGER   NCID, VARID, ELEMENTS, DATATYPE
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT, STRIDE
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode
- `start`, `count`, and `stride` refer to the data in the file
- `buf`, `elements`, and `datatype` refer to the data in memory

JÜLICH
FORSCHUNGSZENTRUM

Reading variables collectively from the file

C/C++

```
int ncmpi_get_vars_all(int ncid, int varid,
        const MPI_Offset start[], const MPI_Offset count[],
        const MPI_Offset stride[], const void* buf,
        int elements, MPI_Datatype datatype)
```

- Reads a *slab* of data from the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode
- `start`, `count`, and `stride` refer to the data in the file
- `buf`, `elements`, and `datatype` refer to the data in memory

JÜLICH FORSCHUNGSZENTRUM

Reading variables collectively from the file

Fortran

```
INTEGER NFMPI_GET_VARS_ALL(NCID, VARID, START, COUNT, STRIDE,
            BUF, ELEMENTS, DATATYPE)
  <type>(*) BUF
  INTEGER   NCID, VARID, ELEMENTS, DATATYPE
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT, STRIDE
```

- Reads a *slab* of data from the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- Can only be used in collective data mode
- `start`, `count`, and `stride` refer to the data in the file
- `buf`, `elements`, and `datatype` refer to the data in memory

Motivation for data set inquiry

- A generic application should be able to handle the data set correctly
- Semantic information must be encoded in names and attributes
  - Conventions need to be set up and used for a given data set class
- Data set structure can be reconstructed from the file

Workflow: Reading a netCDF data set

- Open a data set
- Inquire contents of data set
  - Inquire dimensions for allocation dimensions
  - Inquire variables for id of the desired variable
  - Inquire attributes for additional information
- Allocate memory according to shape of variables
- Read variables from file
  - Parallel netCDF distinguishes between collective and individual data mode
  - Initially in collective mode, user has to switch to individual data mode explicitely
- Close file

Inquiry of number of data set entities

C/C++

```
int ncmpi_inq_ndims(int ncid, int* ndims)
```

- Query number of dimensions

```
int ncmpi_inq_nvars(int ncid, int* nvars)
```

- Query number of variables

```
int ncmpi_inq_natts(int ncid, int* natts)
```

- Query number of attributes

JÜLICH FORSCHUNGSZENTRUM

Inquiry of number of data set entities

Fortran

```
INTEGER NFMPI_INQ_NDIMS(NCID, NDIMS)
  INTEGER NCID, NDIMS
```

- Query number of dimensions

```
INTEGER NFMPI_INQ_NVARS(NCID, NVARS)
  INTEGER NCID, NVARS
```

- Query number of variables

```
INTEGER NFMPI_INQ_NATTS(NCID, NATTS)
  INTEGER NCID, NATTS
```

- Query number of attributes

Version 1.3.1 of parallel-netcdf

- QuickTutorial `http://trac.mcs.anl.gov/projects/parallel-netcdf/wiki/QuickTutorial`
  Sources in `/(bgsys,usr)/local/parallel-netcdf/v1.3.1/examples/tutorial_sources/`
- PnetCDF now duplicates the MPI communicator internally
- New datatypes `NC_UBYTE`, `NC_USHORT`, `NC_UINT`, `NC_INT64`, `NC_UINT64`, and `NF_INT64` (CDF-5)
- New C APIs: `ncmpi_put_vara_ushort`, `..._uint`, `..._longlong`, and `..._ulonglong`. Similarly for var1, var, vars and varm APIs. Also for get and nonblocking APIs
- New Fortran APIs: `nfmpi_put_vars_int8` and similarly for var1, var, vars, varm, get, and nonblocking APIs
- New set of buffered put APIs (e.g. `ncmpi_bput_vara_float`) (see BufferedInterface)

JÜLICH FORSCHUNGSZENTRUM

New in version 1.4.1 of parallel-netcdf

- Version 1.4.1 (Nov 13) to be installed/tested on JUROPA/JUQUEEN
- Fortran API syntax changes in `nfmpi_put_att` and `nf90mpi_put_att family` (e.g. Intent(IN), ...)
- Introduction of **Subfiling**:
  Divides a file transparently into several smaller subfiles; master file contains all metadata about array partitioning information among the subfiles; transparently for user
- New in version 1.5.0pre of parallel-netcdf: C++ API, (May 14)

Outline

Introduction

Basic file handling

Advanced file operations

Exercises

JÜLICH
FORSCHUNGSZENTRUM

Exercise I: NetCDF hello world

C/C++

- Create a generic parallel application (C, Fortran) which creates an empty netcdf file
- Compile, link and execute the application

```
# Compile, Link Blue Gene/Q, JUQUEEN
  module load parallel-netcdf
  mpixlc -I$PNETCDF_INCLUDE helloworld.c
         -L$PNETCDF_LIB -lpnetcdf -o helloworld_c

# Compile, Link Juropa
  module load parallel-netcdf
  mpicc -I$PNETCDF_INCLUDE helloworld.c
        -L$PNETCDF_LIB -lpnetcdf -o helloworld_c
```

Exercise I: NetCDF hello world

Fortran

- Create a generic parallel application (C, Fortran) which creates an empty netcdf file
- Compile, link and execute the application

```
# Compile, Link Blue Gene/Q, JUQUEEN
  module load parallel-netcdf
  mpixlf90 -I$PNETCDF_INCLUDE helloworld_f.F90
           -L$PNETCDF_LIB -lpnetcdf -o helloworld_f

# Compile, Link Juropa
  module load parallel-netcdf
  mpif90 -I$PNETCDF_INCLUDE helloworld_f.F90
         -L$PNETCDF_LIB -lpnetcdf -o helloworld_f
```

JÜLICH FORSCHUNGSZENTRUM

Exercise II: 1d array

- Create two parallel applications (C, Fortran):

  1. Write:
     - Creating a NetCDF data set, containing one one-dimensional variable
     - A local vector of 10000 integers should be allocated and initialized with the task number
     - Each task should write the vector to the NetCDF data set as a part of the global vector
  2. Read
     - Read the NetCDF data set into memory
     - Each task should first read in the dimension and size of the NetCDF variable,
     - allocate then the memory for the local vector,
     - read in the data
     - and check if the data is consistent (task number)

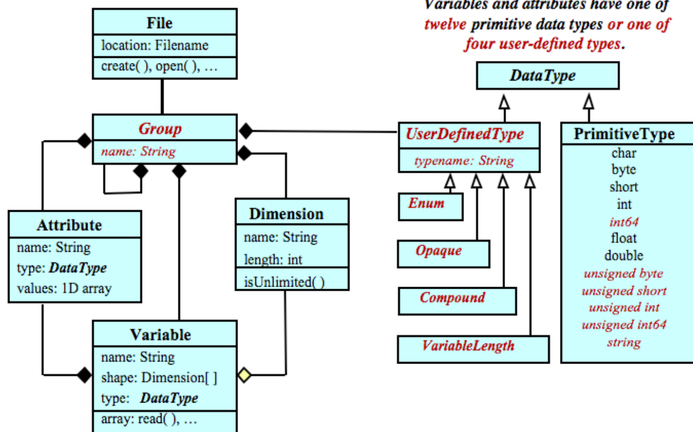
JÜLICH
FORSCHUNGSZENTRUM

Exercise III: 2d array

- Modify the Write/Read-Pogramms of Exercise II as follows:
  - Instead of the vector a two-dimensional integer-array of size 32 x 256 should be written and read by the programs
  - The two-dimensional array should be decomposed in both dimensions
  - The NetCDF dataset should contain a two-dimensional variable

JÜLICH
FORSCHUNGSZENTRUM

NetCDF 4 model



source: Hartnett, E., 2010-09: NetCDF and HDF5 - HDF5 Workshop 2010.