JÜLICH
FORSCHUNGSZENTRUM

Parallel I/O and Portable Data Formats

PnetCDF and NetCDF 4

Sebastian Lührs
s.luehrs@fz-juelich.de
Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH

Jülich, March 13th, 2017

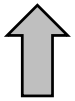
Outline

- Introduction
  - Difference PnetCDF and NetCDF4
- Basic file handling
  - File creation
  - Definitions
  - Writing data
  - Reading data
- Advanced file operations
  - Data set inquiry
  - Flexible data mode interface and access types
  - Performance hints

Introduction to (Parallel) NetCDF

- NetCDF is a portable, self-describing file format developed by Unidata at UCAR (University Cooperation for Atmospheric Research)
  - `http://www.unidata.ucar.edu/software/netcdf/`
- NetCDF does not provide a parallel API prior to 4.0 (NetCDF4 uses HDF5 parallel capabilities)
- PnetCDF is maintained by Argonne National Laboratory (API very similar to standard NetCDF)
  - `http://trac.mcs.anl.gov/projects/parallel-netcdf/`

PnetCDF ≠ NetCDF4



Focus of this presentation, major differences towards NetCDF4 will be highlighted by

PnetCDF or NetCDF4

- NetCDF4:
  - Function Prefix: `nc_... / nf[90]_...`
  - Uses **HDF5** or **PnetCDF** for parallel file access
  - State machine based access mode (default: independent)
  - `size_t` size definition
  - `NC_NETCDF4` format (Classic and 64-Bit-offset format only by using PnetCDF access)
- PnetCDF:
  - Function Prefix: `ncmpi_.../ nf[90]mpi_...`
  - Uses **mpiio** for parallel file access
  - State machine based access mode (default: collective) and function based postfix needed: …`_all`
  - `MPI_Offset` size definition
  - Classic, `NC_64BIT_OFFSET` and `NC_64BIT_DATA` format

Terms and definitions

Dimension

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

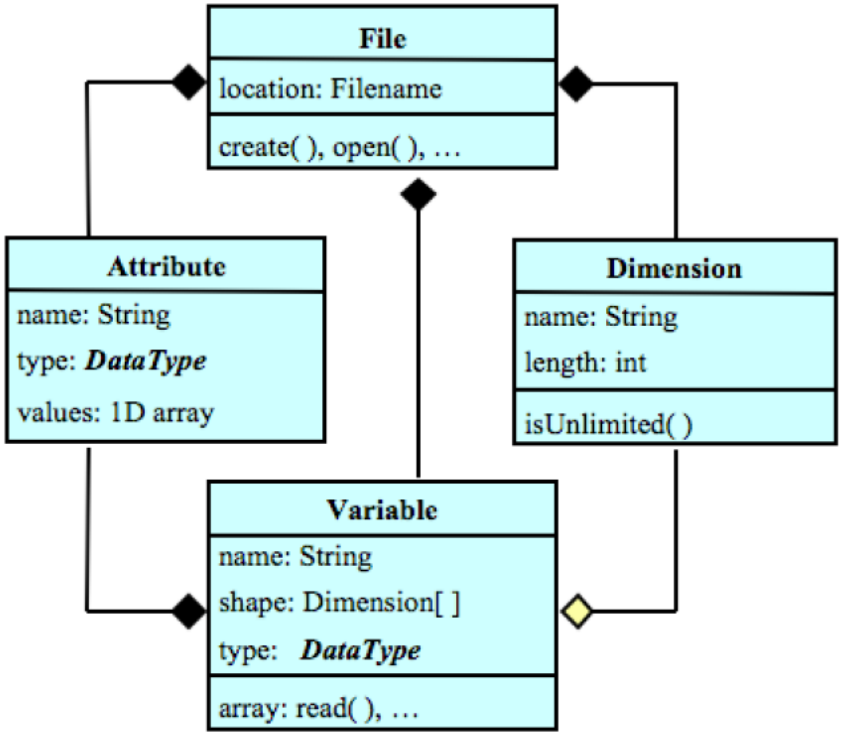
Variable

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



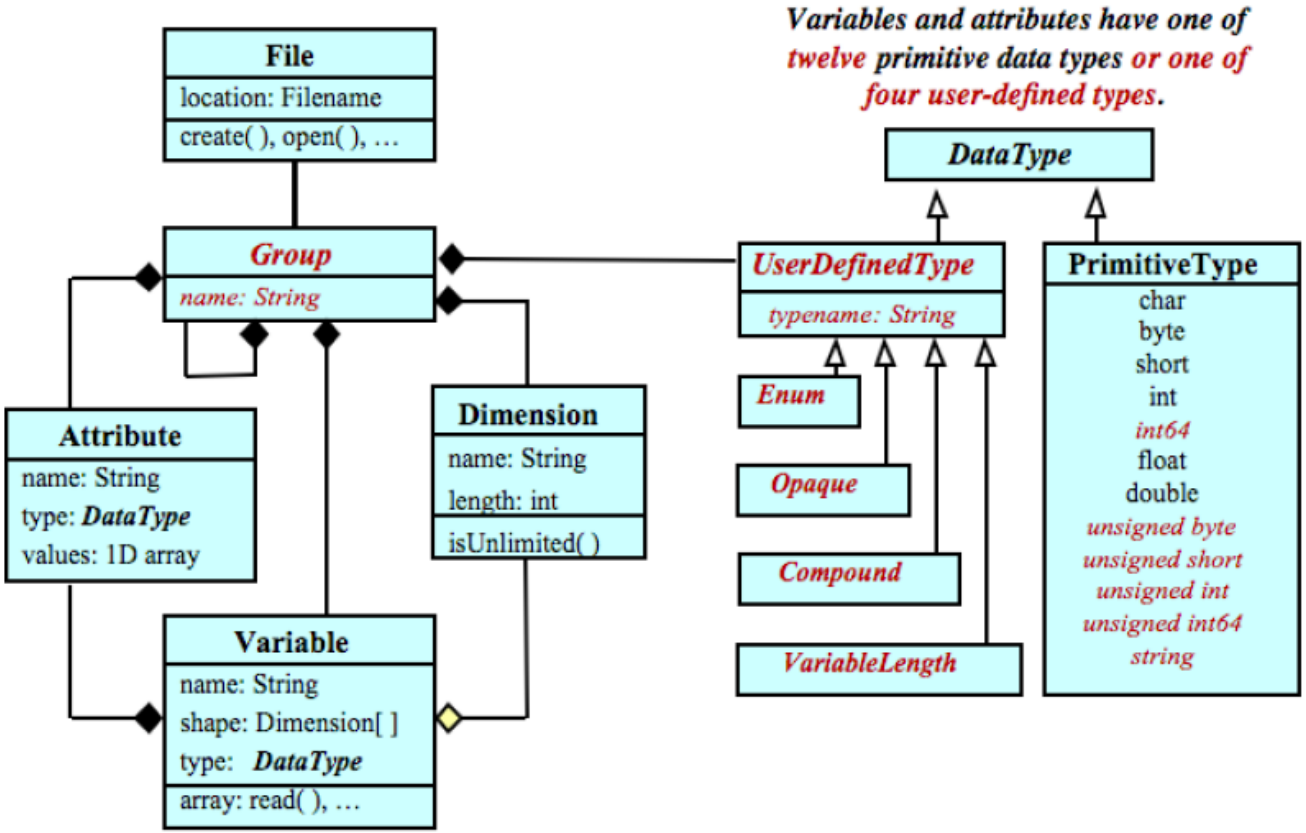
Variables and attributes have one of six primitive data types.

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.

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

NetCDF4 model



A file has a top-level unnamed group. Each group may contain one or more named subgroups, user-defined types, variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One or more dimensions may be of unlimited length.

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

Naming conventions

- Sequence of alphanumeric characters, `_`, `.`, `+`, `-`, `@`
- 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

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

NETCDF4 allows multiple unlimited dimensions

- 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 types (classic model)
  - 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

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 as single-element vectors

Attribute Conventions Examples

`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
`https://www.unidata.ucar.edu/software/netcdf/docs/netcdf/Attribute-Conventions.html`

Datatypes

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

C	Fortran	Storage
NC_BYTE	NF[90]_BYTE	8-bit signed integer
NC_CHAR	NF[90]_CHAR	8-bit unsigned integer
NC_SHORT	NF[90]_SHORT	16-bit signed integer
NC_INT	NF[90]_INT	32-bit signed integer
NC_FLOAT	NF[90]_FLOAT	32-bit floating point
NC_DOUBLE	NF[90]_DOUBLE	64-bit floating point

NETCDF4 format also allows NC_STRING, NC_INT64, unsigned datatypes and user defined datatypes

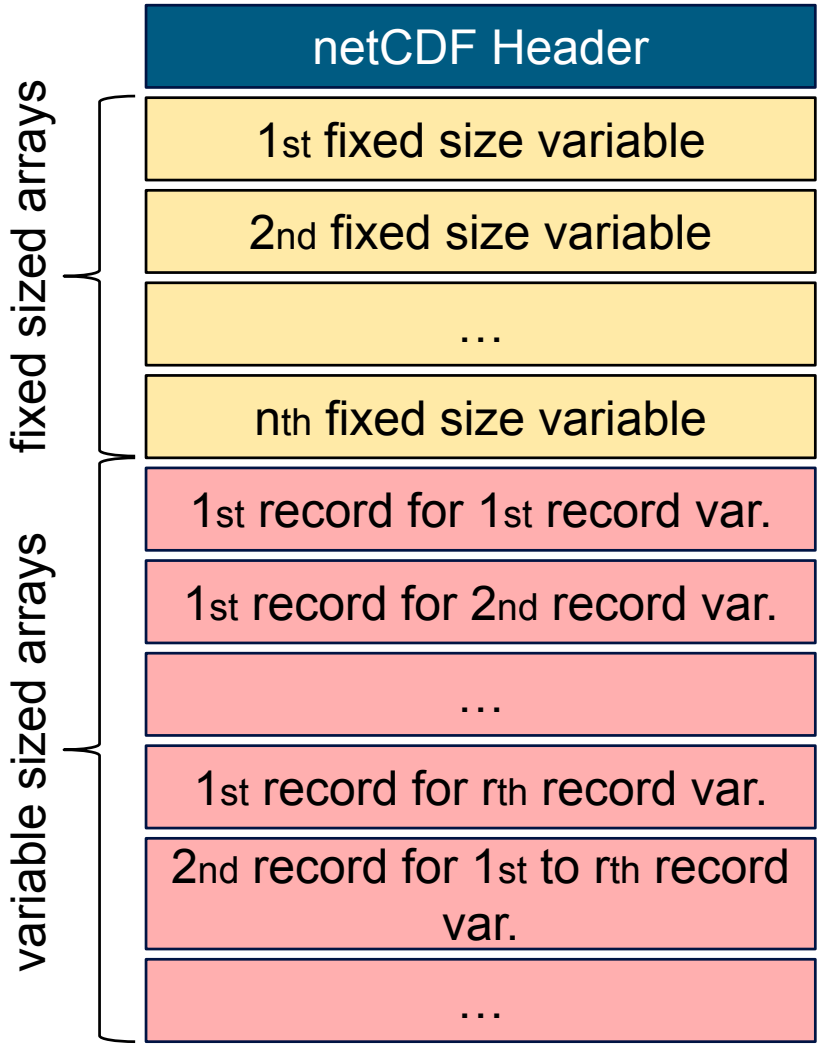
The classic NetCDF file format

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

Limitations	classic	64bit off.	64bit data
max. dim. size	2^31	2^32	> 2^32
max. num. elem.	2^32	2^32	> 2^32
max. var. size	2GiB	4GiB	> 4GiB



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 explicitly
- Close file

Default fill behaviour:
NetCDF: `NC_FILL`, PnetCDF: `NC_NOFILL`,
`nc_set_fill` can change behaviour

NETCDF4: initially independent mode

Header files

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

Fortran
```
use pnetcdf
or
include 'pnetcdf.inc'
```

- Contain definition of
  - constants
  - functions

NETCDF4:
```
#include <netcdf_par.h>
#include <netcdf.h>

use netcdf
or
include 'netcdf.inc'
```

Creating a file

C

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

Fortran

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

`NF90MPI_...` can be used when using F90

- Call is collective over `comm`
- `ncid` is the id of the internal file handle
- `cmode` must specify at least one of the following
  - `(NC/NF)_CLOBBER` – Create new file and overwrite, if it existed before
  - `(NC/NF)_NOCLOBBER` – Create new file only, if it did not exist before
- Choose file format on file creation
  - default - classic format
  - `(NC/NF)_64BIT_OFFSET` - 64-bit offset format
  - `(NC/NF)_64BIT_DATA` - 64-bit data format

Creating a file NETCDF4

C

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

Fortran

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

`NF90_...` can be used when using F90

- Call is collective over `comm`
- `ncid` is the id of the internal file handle
- `cmode` must specify at least one of the following
  - `(NC/NF)_CLOBBER` – Create new file and overwrite, if it existed before
  - `(NC/NF)_NOCLOBBER` – Create new file only, if it did not exist before
- Choose file format on file creation
  - default - classic format (PnetCDF)
  - `(NC/NF)_64BIT_OFFSET` - 64-bit offset format (PnetCDF)
  - `(NC/NF)_NETCDF4|(NC/NF)_MPIIO` - NetCDF4 format (HDF5)

Open an existing NetCDF data set

C

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

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
  - `(NC/NF)_WRITE` – Open file for any kind of change to the file
  - `(NC/NF)_NOWRITE` – Open the file read-only

NETCDF4:
`[nc,nf]_open_par(filename, omode, comm, info, ncid)`

Closing a file

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

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

- Close file associated with `ncid`

NETCDF4:
`[nc,nf]_close(ncid)`

Error handling

C
```
const char * ncmpi_strerror(int status)
```

Fortran
```
CHARACTER*80 NFMPI_STRERROR(STATUS)
  INTEGER STATUS
```

- Return status string representation
- `(NC/NF)_NOERR` can be used to check status

Exercise

Exercise 1 – NetCDF hello world

- Create a parallel application (C or Fortran) which creates an empty NetCDF file (you can use the PnetCDF-API or the NetCDF4-API)
- Compile, link and execute the application

PnetCDF, C:

```
module load intel-para
module load parallel-netcdf/1.7.0
mpicc helloworld_pnetcdf.c -lpnetcdf
```

PnetCDF, Fortran:

```
module load intel-para
module load parallel-netcdf/1.7.0
mpif90 helloworld_pnetcdf.f90 -lpnetcdf
```

NetCDF4, C:

```
module load intel-para
module load netCDF/4.4.0
mpicc helloworld_netcdf.c -lnetcdf
```

NetCDF4, Fortran:

```
module load intel-para
module load netCDF-Fortran/4.4.3
mpif90 helloworld_netcdf.f90 -lnetcdff
```

- Templates and Solutions: `/work/hpclab/train001`

Defining dimensions

C

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

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
  - `(NC/NF)_UNLIMITED` will create an unlimited dimension
- Can only be called in *definition mode*

Defining variables

C

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

Fortran

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

F90: `NDIMS` not needed

- `xtype` specifies the external type of this variable
- `dimids` is an array of size `ndims`
- Can only be called in *definition mode*

Defining attributes

C

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

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
```

F90: `<type>`, `XTYPE` and `LEN` not needed:

```
NF90MPI_PUT_ATT(
  NCID, VARID, NAME,
  ATTR)
```

- Puts the attribute `attr` into the data set
- `varid` is the id annotated variable, or `(NC/NF)_GLOBAL`, if it is a global attribute
- `xtype` specifies the external type of this attribute
- Can only be called in *definition mode*

NETCDF4 uses `size_t` (C) / `INTEGER` (FORTRAN) for len

Reading attributes

C

```
int ncmpi_get_att_<type>(int ncid, int varid,
                         const char* name, <type>*attr)
```

Fortran

```
INTEGER NFMPI_GET_ATT_<type>(NCID, VARID, NAME,
                             ATTR)
  <type> ATTR
  CHARACTER*(*) NAME
  INTEGER NCID, VARID
```

F90:
`NF90MPI_GET_ATT(…)`

- Reads the attribute `attr` from the data set
- `varid` is the id annotated variable, or `(NC/NF)_GLOBAL`, if it is a global attribute

Closing define mode

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

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

- Ends the definition phase, and switches to collective data mode NETCDF4: independent data mode
- Once variables have been put into the data set, definitions must not be altered

Not explicitly needed for NETCDF4 file format

Exercise

Exercise 2 – NetCDF attributes and dimensions

- Extend your existing parallel application (C or Fortran)
- Create a dimension of size `100 x NUMBER_OF_PROCS`
- Add a simple global integer attribute value
- Add an integer variable using your created dimension
- Compile, link and execute the application

Check the resulting file using:
`ncdump`, `ncmpidump` and `h5dump`

Switching data modes

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

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

- Switches from collective data mode to individual data mode

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

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

- Switches from individual data mode to collective data mode

NETCDF4:
```
nc_var_par_access(ncid, varid,
        [NC_INDEPENDENT, NC_COLLECTIVE])
```

Writing variables (collectively)

C

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

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`
- `_all` can only be used in collective mode

NETCDF4: `_all` not needed

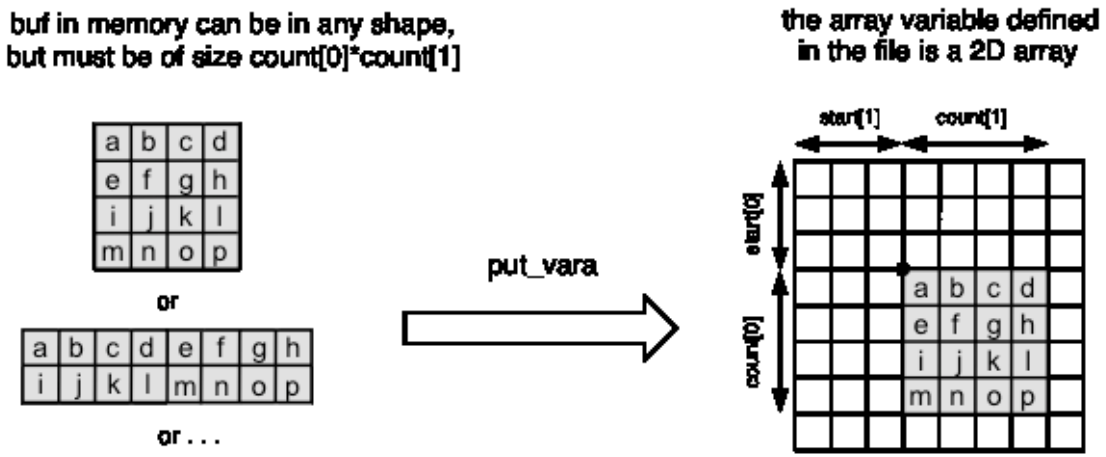
F90:

```
nf90mpi_put_var[_all](
  ncid,varid,var,start?,
  count?,stride?,imap?)
```

Writing variables (collectively)

C

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



source: PnetCDF C Interface Guide, http://cucis.ece.northwestern.edu/projects/PnetCDF/doc/pnetcdf-c/

Reading variables (collectively)

C

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

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 of the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- `_all` can only be used in collective mode

NETCDF4: `_all` not needed

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 explicitly

NETCDF4: initially independent mode

- Close file

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

Inquiry of number of data set entities

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

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

Fortran90: `nf90mpi_inquire_variable`

- Query number of dimensions

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

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

Fortran90: `nf90mpi_inquire_variable`

- Query number of variables

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

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

Fortran90: `nf90mpi_inquire_variable`

- Query number of attributes

Inquiry of ids of data set entities

C
```
int ncmpi_inq_varid(int ncid, const char* name,
                    int* varid)
```

Fortran
```
INTEGER NFMPI_INQ_VARID(NCID, NAME, VARID)
  INTEGER NCID, VARID
  CHARACTER*(*) NAME
```

- Query id of variable given by `name`

C
```
int ncmpi_inq_vardimid(int ncid, int varid,
                       int* dimids)
```

Fortran
```
INTEGER NFMPI_INQ_VARDIMID(NCID, VARID, DIMIDS)
  INTEGER NCID, VARID,
  INTEGER* DIMIDS
```

Fortran90: `nf90mpi_inquire_variable`

- Query `dimids` for given `varid`

Inquiry of dimension lens

C

```
int ncmpi_inq_dimlen(int ncid, int dimid,
                     MPI_Offset* len)
```

Fortran

```
INTEGER NFMPI_INQ_DIMLEN(NCID, DIMID, LEN)
  INTEGER NCID, DIMID
  INTEGER(KIND=MPI_OFFSET_KIND) LEN
```

Fortran90: `nf90mpi_inquire_dimension`

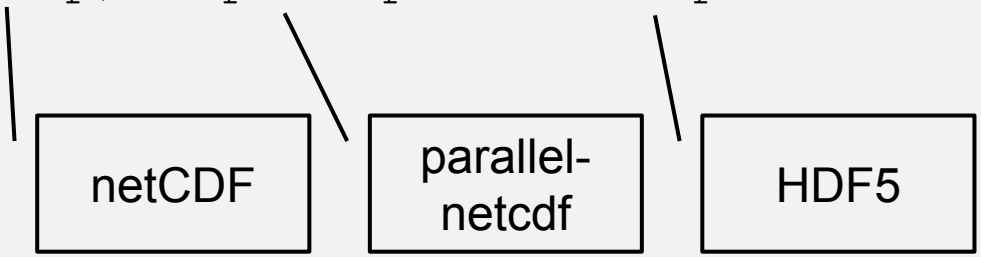
- Reads `len` from a given dimension

Exercise

Exercise 3 – NetCDF write data collectively

- Extend your existing parallel application (C or Fortran)
- Each process should allocate a vector of 100 integers initialized with its task number
- Each task should write its vector to the NetCDF data set as a part of the global vector created in exercise 2:
  e.g.: `0000…1111…2222…`
- Write should be collective

Check the resulting file using:
`ncdump`, `ncmpidump` and `h5dump`



Flexible API

PnetCDF only

C

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

Fortran

```
INTEGER NFMPI_PUT_VARA[_ALL](NCID, VARID, START,
                             COUNT, BUF,
                             ELEMENTS, DATATYPE)
  <type>(*) BUF
  INTEGER NCID, VARID, DATATYPE
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT,
          ELEMENTS
```

F90: not needed

- 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` refer to the data in the file
- `buf`, `elements`, and `datatype` refer to the data in memory

Flexible API

PnetCDF only

C

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

Fortran

```
INTEGER NFMPI_GET_VARA[_ALL](NCID, VARID, START,
                             COUNT, BUF,
                             ELEMENTS, DATATYPE)
  <type>(*) BUF
  INTEGER NCID, VARID, DATATYPE
  INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT,
          ELEMENTS
```

F90: not needed

- Reads a *slab* of data of 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` refer to the data in the file
- `buf`, `elements`, and `datatype` refer to the data in memory

Non-blocking interface

PnetCDF only

- Can aggregate multiple smaller requests into larger ones for better I/O performance

C

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

Fortran

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

- `req_id` must be stored for later access

Non-blocking interface

PnetCDF only

C

```
int ncmpi_wait(int ncid, int num_reqs,
               int req_ids[], int statuses[])
int ncmpi_wait_all(int ncid, int num_reqs,
                   int req_ids[], int statuses[])
```

Fortran

```
INTEGER NFMPI_WAIT(NCID, NUM_REQS, REQ_IDS,
                   STATUSES)
  INTEGER NCID, NUM_REQS,
  INTEGER(*) REQ_IDS, STATUSES
INTEGER NFMPI_WAIT_ALL(NCID, NUM_REQS, REQ_IDS,
                       STATUSES)
```

- Also buffered interface is available (`bget`/`bput`) which allows setting the internal buffer

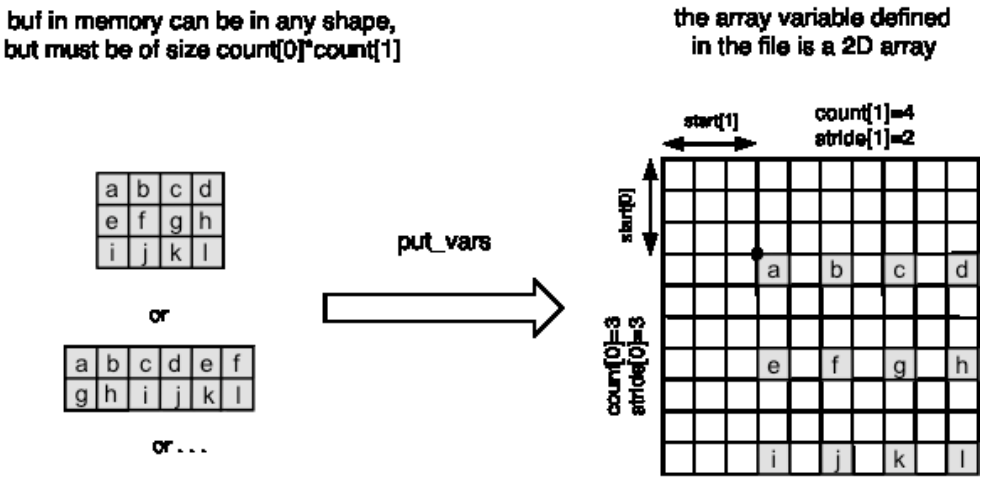
C

```
int ncmpi_buffer_attach(int ncid, int bufsize)
int ncmpi_buffer_detach(int ncid)
```

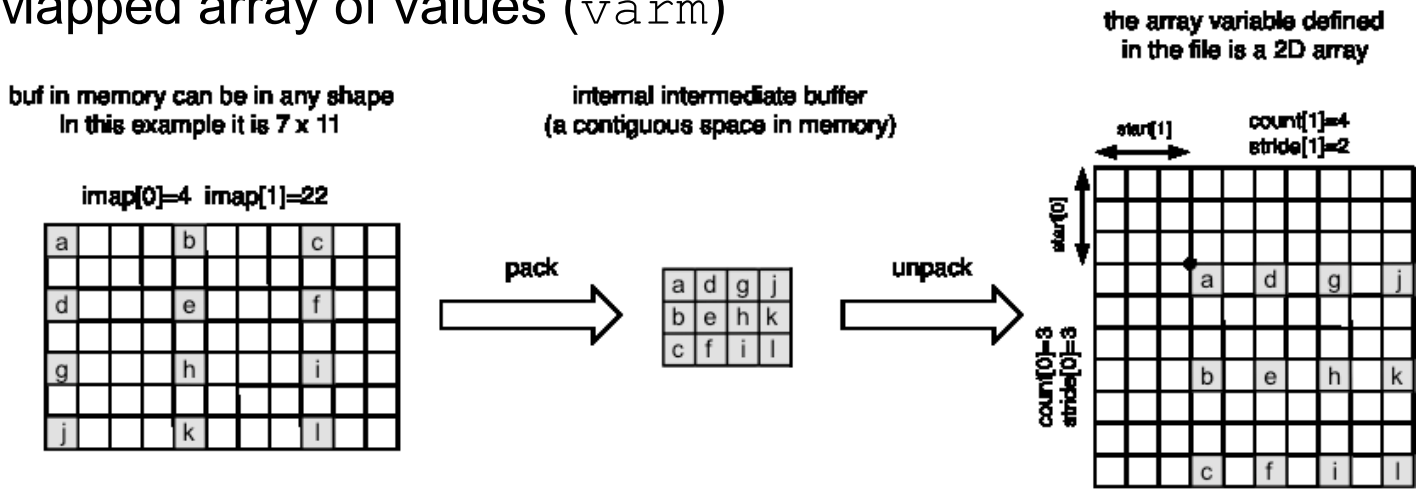
Additional access types

F90: not needed

- Subsampled array of values (`vars`)



- Mapped array of values (`varm`)



source: PnetCDF C Interface Guide, http://cucis.ece.northwestern.edu/projects/PnetCDF/doc/pnetcdf-c/

API matrix

PnetCDF only

ncmpi_... nfmpi_...	blocking	non-blocking	buffered
single data value	put_var1 get_var1	iput_var1 iget_var1	bput_var1 bget_var1 (since 1.3.0)
array of values	put_vara get_vara	iput_vara iget_vara	bput_vara bget_vara (since 1.3.0)
subsampled array of values	put_vars get_vars	iput_vars iget_vars	bput_vars bget_vars (since 1.3.0)
mapped array of values	put_varm get_varm	iput_varm iget_varm	bput_varm bget_varm (since 1.3.0)
list of subarrays of values PnetCDF only	put_varn get_varn (since 1.4.0)	iput_varn iget_varn (since 1.6.0)	bput_varn bget_varn (since 1.6.0)

Performance hints

- **Subfiling** PnetCDF only
  - Divides a file transparently into several smaller subfiles
  - Master file contains all metadata about array partitioning information among the subfiles
  - Transparent for user

C
```
MPI_Info_create(&info)
MPI_Info_set(info, "nc_num_subfiles", "4")
ncmpi_create(…, info, fh)
```

- **Chunking** NetCDF4 only
  - When a dataset is chunked, each chunk is read or written as a single I/O operation, and individually passed from stage to stage of the pipeline and filters (based on HDF5 chunking)

C
```
int nc_def_var_chunking(ncid, varid,
   [NC_CONTIGUOUS, NC_CHUNKED], size_t *chunksizesp)
```

Exercise

Exercise 4 – Mandelbrot set

- Implement a solution for the stride decomposition (`type = 0`) of the Mandelbrot example (`mandelpnetcdf.c` or `mandelpnetcdf.f90`)
- Use a collective blocking write call
- You will need the following pnetcdf (or equivalent NetCDF4) routines
  - `ncmpi_create / nf90mpi_create`
  - `ncmpi_def_dim / nf90mpi_def_dim`
  - `ncmpi_put_att_double / nf90mpi_put_att`
  - `ncmpi_def_var / nf90mpi_def_var`
  - `ncmpi_put_vara_int_all / nf90mpi_put_vara_all`
  - `ncmpi_close / nf90mpi_close`

Hints

Options for running the program:

`-t 0` (stride decomposition)

`-f 3` (use pnetcdf)

```
/work/hpclab/train001/par_mandel[_fortran]/
```