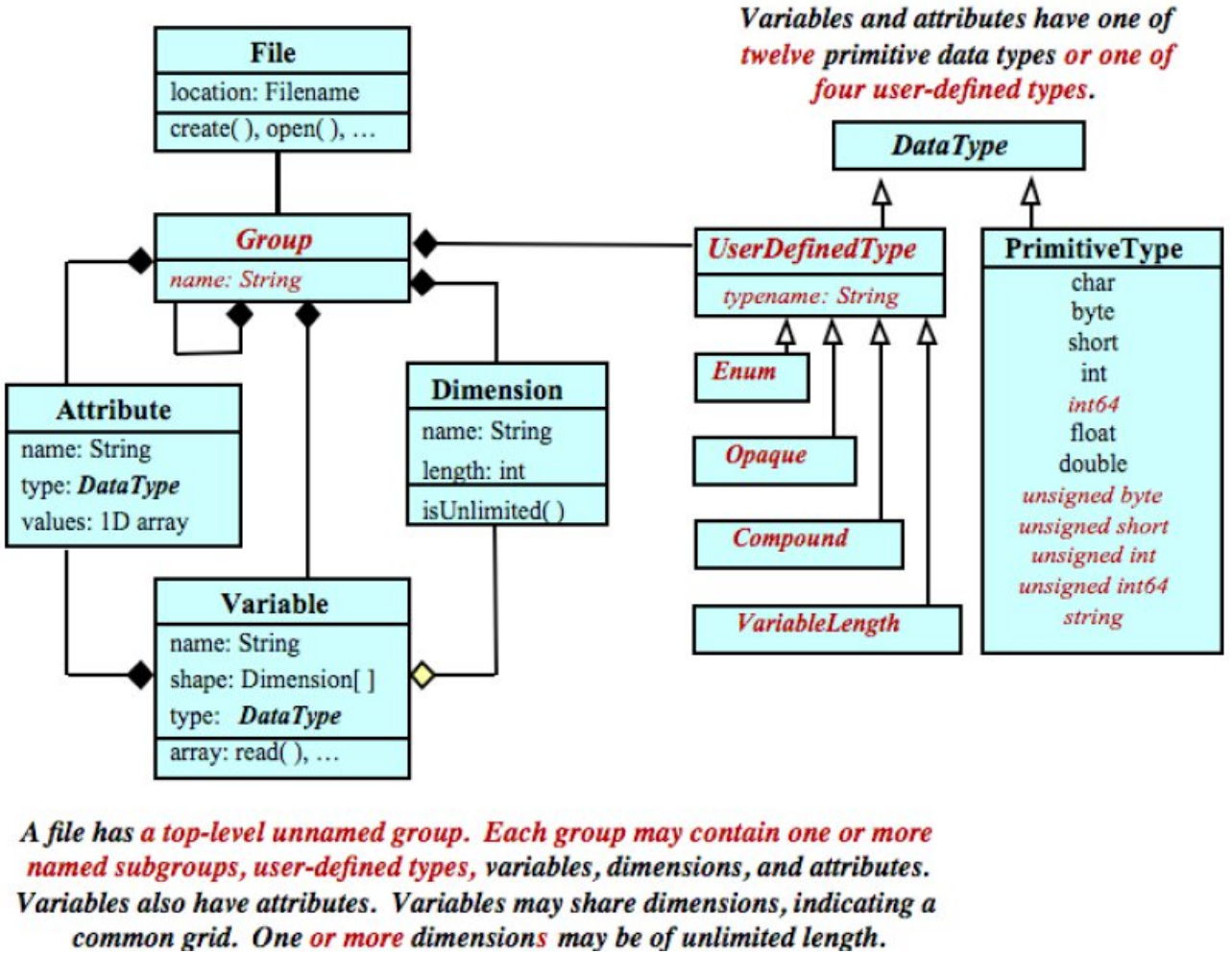
PARALLEL I/O AND PORTABLE DATA FORMATS

NETCDF-4

22.04.2024 I JUNXIAN CHEW (J.CHEW@FZ-JUELICH.DE)

NetCDF-4 model



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

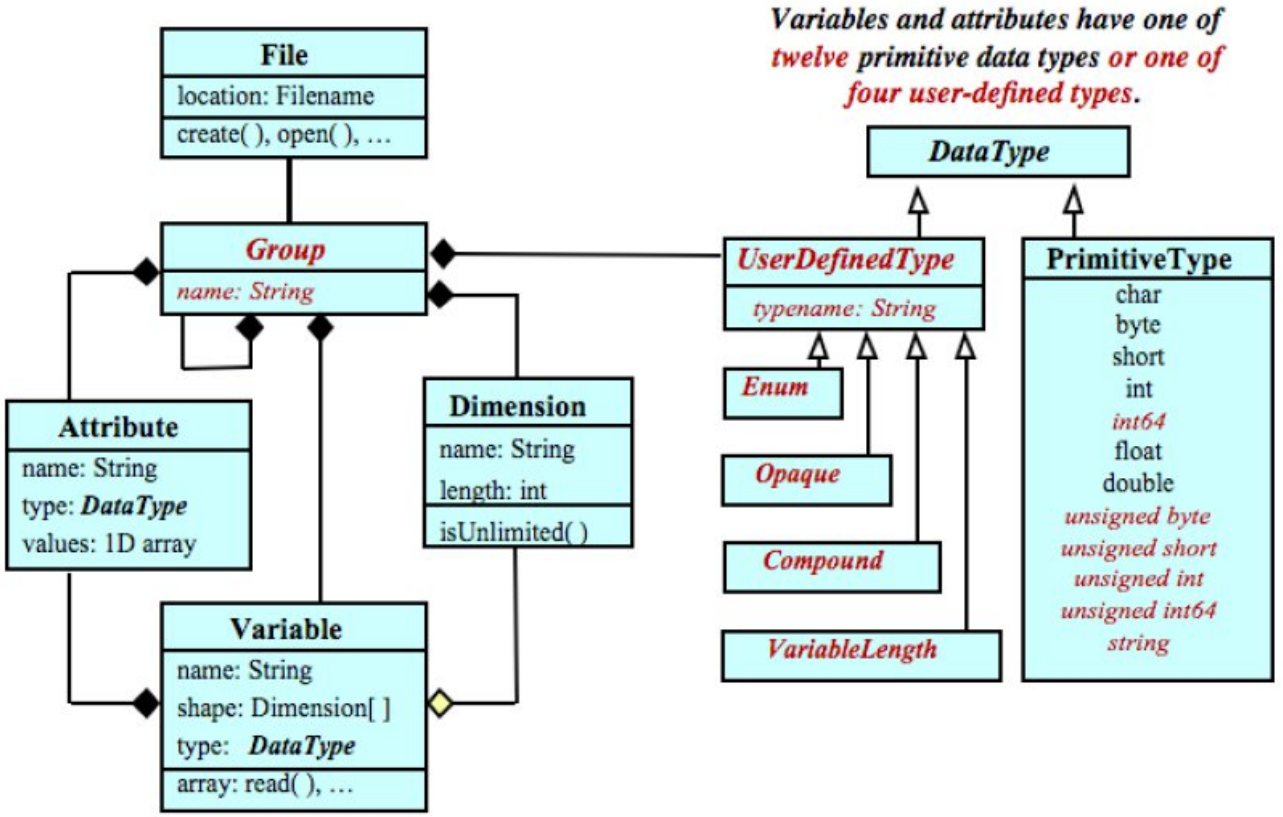
Introduction to NetCDF-4 and Parallel-NetCDF

- 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 ≠ NetCDF-4

Focus of this presentation

NetCDF-4 model (revisited)



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

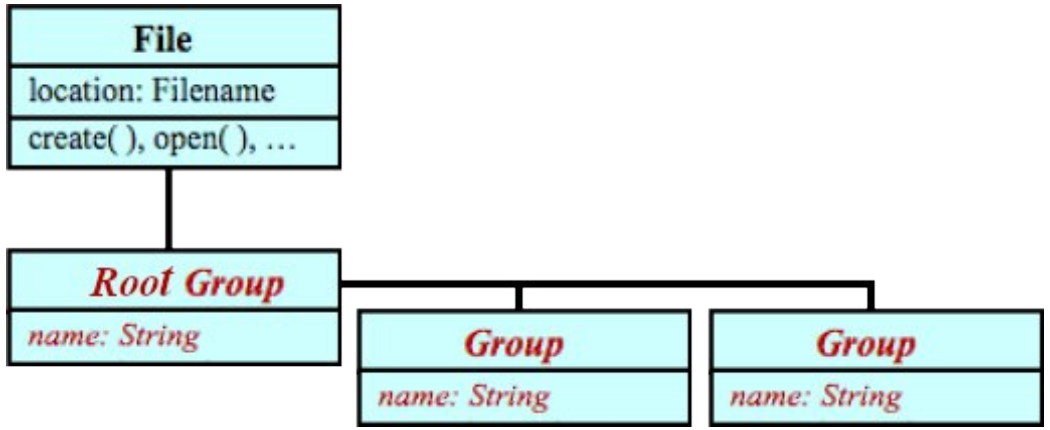
File/Dataset

- Recall the file/dataset format discussed earlier (CDF or HDF5 format)
- File/dataset format is determined with cmode option in `N[C/F90]_create()`

cmode Flag	File format
Default	CDF-1 Format
N[C/F90]_64BIT_OFFSET	CDF-2 Format
N[C/F90]_64BIT_DATA	CDF-5 Format
N[C/F90]_NETCDF4	NetCDF-4/HDF5

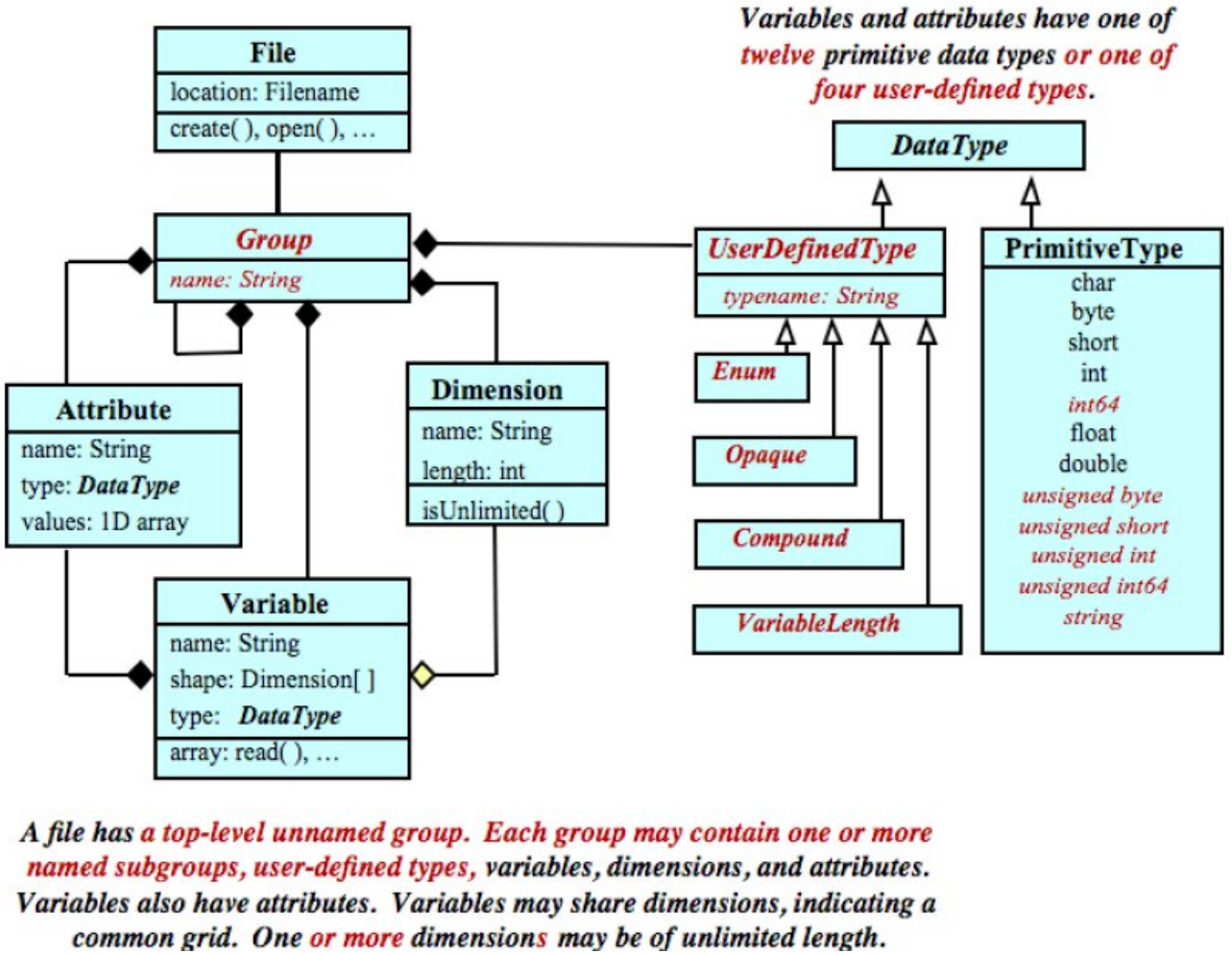
Groups (Not Available for CDF format)

- Every `NC` file will have at least 1 group of data.
- `id` of the root data group returned when `NC` file is opened/created.
- Using the root group as parent, multiple new groups can be created.
- Each group contains dimensions, variables and attributes.



- https://docs.unidata.ucar.edu/netcdf-c/current/group__groups.html
- https://docs.unidata.ucar.edu/netcdf-fortran/4.6.1/f90_groups.html

NetCDF-4 model (revisited)



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

Dimensions

- Defines the shape of variables
- Have a name and length
- Can have either a fixed length or can be '`N[C/F90]_UNLIMITED`'
- Can be used more than once in a variable declaration
  - Use only more than once, where semantically useful
- Recommended to name dimensions in relation to the variable that uses it
  - Can represent a physical dimension like time, height, latitude, longitude, etc.
  - Can be used to index other quantities, e.g., station number

Station\Time	0800	1000	1200	1400
St. 1	22.51	23.07	23.61	23.95
St. 2	20.49	21.10	21.45	21.73
St. 3	22.03	22.56	23.09	23.34

Soil Surface Temperature

Dimension 1:
- Name = 'Time'
- Length = 4

Dimension 2:
- Name = 'Stn_Num'
- Length = 3

Variables

- Store the bulk data in the file/dataset
- Regarded as *n*-dimensional array
  - Scalar values represented as 0-dimensional arrays
- Have a name, datatype and shape (defined through dimensions)
- Variable shape cannot be deleted or altered after creation
- Default variable types (next slide)
- A position along a dimension can be specified as index
  - Starting at 0 in C and 1 in Fortran
- Example Variable definition:

Name	Datatype	Shape
soil_surf_temp	double	[Stn_Num,Time]

Datatypes

- The CDF conforming file format (`NetCDF classic and 64-bit offset`) only supports basic types

C	Fortran	Storage
NC_BYTE	NF90_BYTE	8-bit signed integer
NC_CHAR	NF90_CHAR	8-bit character
NC_SHORT	NF90_SHORT	16-bit signed integer
NC_INT	NF90_INT	32-bit signed integer
NC_FLOAT	NF90_FLOAT	32-bit floating point
NC_DOUBLE	NF90_DOUBLE	64-bit floating point
NC_STRING	NF90_STRING	Character string

- NetCDF-4 format supports user-defined datatypes.

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
- There is a recommended naming convention:

`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 A: Attribute Conventions of the NetCDF Documentation
  `https://docs.unidata.ucar.edu/netcdf-c/current/attribute_conventions.html`

Recap: Terms and definitions

Dimension
An entity used to define the shape of variables.

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 that stores the metadata of variable or file/dataset. The latter are called *global attributes*.

Recap: PnetCDF or NetCDF4

- NetCDF4:
  - Function Prefix: `nc_... / nf90_...`
  - Uses **HDF5** or **PnetCDF** for parallel file access
  - Supports PnetCDF formats (see below)
  - Also supports `NC_NETCDF4` (HDF5) format
- PnetCDF:
  - Function Prefix: `ncmpi_.../ nf90mpi_...`
  - Uses **mpiio** for parallel file access
  - Created file formats conform to CDF-format
  - Classic, `NC_64BIT_OFFSET` and `NC_64BIT_DATA` format

Think of NetCDF-4 as a top-layer interface. Depending on the format of file/dataset, proper parallel 'engine' will be chosen to work on it. PnetCDF for CDF formats, HDF5 for modern format.

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
  - Variables are prefilled (can be changed by using `nc_set_fill`)
- Store variables in file
- Close file

Workflow: Creating a NetCDF data set

NetCDF functions sneak peek:

```
n[c/f90]_create          # creates the dataset/file
   n[c/f90]_def_dim      # defines the dimensions to describe variables
   n[c/f90]_def_var      # defines the variables using the dimensions
   n[c/f90]_put_att      # attach metadata/attributes to variables/group
n[c/f90]_enddef          # exits define mode, enters the data mode
   n[c/f90]_put_var      # copy data from memory into dataset/file
n[c/f90]_close           # closes the dataset/file
```

Your First NetCDF data set

Header files

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

Fortran
```
use netcdf
```

Contain definition of

- constants
- functions

Python
```
import netCDF4
```

Creating a file – File size limitations

cmode Flag	Max File Size
Default (CDF-1 Format)	2 GiB (2^30 bytes)
`N[C/F90]_64BIT_OFFSET` (CDF-2 Format)	8 EiB (2^60 bytes)
`N[C/F90]_64BIT_DATA` (CDF-5 Format)	Unlimited
`N[C/F90]_NETCDF4` (NetCDF4/HDF5 format)	Unlimited

- CDF formats allows the file to be read by PnetCDF library.
- `N[C/F90]_64BIT_DATA` conforms to CDF format, which DOES NOT support user-defined datatypes.
- File created via `N[C/F90]_NETCDF4` can be read via HDF5 library.
- `N[C/F90]_NETCDF4` flags should be paired with `NC_MPIIO` for NetCDF v4.6.1 and older.

Creating a file

C

```
int nc_create(const char* path, int cmode,int* ncid)
```

Fortran

```
INTEGER nf90_create(PATH, CMODE, NCID)
  character (len = *), intent(in):: PATH
  integer, intent(in)            :: CMODE
  integer, intent(out)           :: NCID
```

- `ncid` is the id of the internal file handle
- `cmode` must specify at least one of the following
  - `N[C/F90]_CLOBBER` – Create new file, overwrite if it existed before. [default]
  - `N[C/F90]_NOCLOBBER` – Only create new file if it did not exist before
- Choose file format on file creation
  - default - classic format (CDF-1 format)
  - `N[C/F90]_64BIT_OFFSET` - 64-bit offset format (classic CDF-2 format)
  - `N[C/F90]_NETCDF4` - NetCDF4 format (HDF5)
- `cmode` options can be combined with bitwise inclusive OR operator.
  - `NC_NOCLOBBER|NC_NETCDF4` (C version)
  - `IOR(NF90_NOCLOBBER,NF90_NETCDF4)` (FORTRAN version)

A newly created file is immediately in `define mode`, doesn't apply to python version!

Creating a file – Python version

Python
```
nc = netCDF4.Dataset('filename', 'w',format='NETCDF4')
```

- Instead of the `cmode`, python version calls it access mode
  - Read-only mode: `'r'`
  - File creation mode :
    - Write : `'w'`
    - Append : `'a' or 'r+'`
- Formatting of file, only relevant when access mode is `'w'`
  - `NETCDF3_CLASSIC`
  - `NETCDF3_64BIT_OFFSET`
  - `NETCDF3_64BIT_DATA`
  - `NETCDF4_CLASSIC` – HDF5 file that conforms to CDF-format
  - `NETCDF4`

Defining dimensions

C

```
int nc_def_dim(int ncid, const char* name,
               size_t len, int* dimid)
```

Fortran

```
integer nf90_def_dim(NCID, NAME, LEN, DIMID)
  integer, intent(in)              :: NCID
  character (len = *), intent(in)  :: NAME
  integer, intent(in)              :: LEN
  integer, intent(out)             :: DIMID
```

- `name` represents the name of the dimension
- `len` represents the length of this dimension
  - `N[C/F90]_UNLIMITED` will create an unlimited dimension
- Can only be called in *define mode*

Python

```
dim1 = nc.createDimension('name',size)
```

- Setting size to 0 in python is identical to setting `N[C/F90]_UNLIMITED`

Defining variables

C

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

Fortran

```
integer nf90_def_var(NCID, NAME, XTYPE,
                     DIMIDS, VARID)
  integer, intent(in)             :: NCID
  character(len = *), intent(in):: NAME
  integer, intent(in)             :: XTYPE
  integer, intent(in)             :: DIMIDS(scalar or dimension(:))
  integer, intent(out)            :: VARID
```

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

Python

```
var = nc.createVariable('name', 'numpy_type',('dimension',))
```

- Defining multi-dimensional variable in python via: `('dim1', 'dim2',)`

Datatypes

- Short list of available datatypes

numpy	C	Fortran	Storage
i1	NC_BYTE	NF90_BYTE	8-bit signed integer
u1	NC_CHAR	NF90_CHAR	8-bit character
i2	NC_SHORT	NF90_SHORT	16-bit signed integer
i4	NC_INT	NF90_INT	32-bit signed integer
f4	NC_FLOAT	NF90_FLOAT	32-bit floating point
f8	NC_DOUBLE	NF90_DOUBLE	64-bit floating point
S1	NC_STRING	NF90_STRING	Character string

- Refer to: `https://docs.unidata.ucar.edu/nug/current/md_types.html#data_type`
- NetCDF-4/HDF5 format also supports user-defined datatypes.

Defining attributes

C

```
int nc_put_att(int ncid, int varid,
               const char* name, nc_type xtype,
               size_t len, const void* attr)
```

Fortran

```
integer nf90_put_att(NCID, VARID, NAME, ATTR)
  integer, intent(in)                          :: NCID, VARID
  character (len=*), intent(in)                :: NAME
  Any scalar or rank-1 array type, intent(in)  :: ATTR
```

- Puts the attribute `attr` into the data set
- `varid` is the id annotated variable, or `N[C/F90]_GLOBAL`, if it is a global attribute
- `xtype` specifies the external type of this attribute
- Can only be called in *define mode*
- Recommended to follow attribute convention for `name` (refer to slide 10)

Python

```
element.attribute_name = value
```

- `element` can be Dataset, Group, or Variable constructs.
  - `var.units = "K"`

Defining attributes – string att in C

C

```
int nc_put_att_string(int ncid, int varid,
                      const char* name, size_t len,
                      const char **attr)
int nc_put_att_text(int ncid, int varid,
                      const char* name, size_t len,
                      const char *attr)
```

- `string` or `text` type in attributes requires care in C programming.
- In `string`, `len` refers to the number of strings (NOT characters).
  - Note: Input `attr` is a double pointer variable!
- In `text`, `len` refers to the number of characters.
  - Note: Input `attr` is a pointer variable!

Closing define mode

C	`int nc_enddef(int ncid)`
Fortran	`integer nf90_enddef(NCID)` `  integer, intent(in) :: NCID`

- Ends the definition phase, and switches to independent data mode
- Python version doesn’t have the distinction between *data* and *define mode*

Closing a file

C

```
int nc_close(int ncid)
```

Fortran

```
integer nf90_close(NCID)
  integer, intent(in) :: NCID
```

- Close file associated with `ncid`

Python

```
nc.close()
```

Exercise

Exercise 1 – NetCDF temp data

- Copy the course folder to your workspace and navigate to NetCDF exercises:

```
cd /p/project1/training2403/
cp -r ParIO_course_material $USER
cd $USER/exercises/NetCDF
```

- Create an application to write an empty '`temp_data.nc`' file
- The file should have the following definitions:
- Dimension
  - name : "hours"
  - length : 24
- Variable
  - name : "temp-at-hour"
  - xtype : 32-bit floating point
  - ndims : 1
- Attribute
  - name : "valid_max"
  - xtype : 32-bit floating point
  - value : 70.0
  - Attach to "temp-at-hour" variable
- You can use the provided template: `exercise_1.[c|f90|py]`
- Compile, link and execute the application (see next slide)

Exercise

Exercise 1 – NetCDF temp data

- Modules to load and instructions to compile are provided in file.
- Example of loading modules, compile and run:

```
module load GCC OpenMPI netCDF

gcc exercise_1.c -lnetcdf # Compile C style
```

- Run the application on login node:

```
[<user>@jlogin ~]$ ./a.out
```

- Advanced exercise :
- Create the file with the same definitions as previous,
- with two additional attributes below:

Attribute
- name : “units”
- value : “Celsius”
- Attach to “temp-at-hour” variable

Attribute
- name : “long_name”
- value : “Placeholder for temperature data.”
- Attach to “temp-at-hour” variable

ncdump

- Dumps the content of the written NetCDF file on shell.
- Useful options:
  - -h, header information only without data
  - -v <var_name>, data information only
- Example:

```
[<user>@jlogin ~]$ ncdump -h temp_data.nc
netcdf temp_data {
dimensions:
        hours = 24 ;
variables:
        float temp-at-hour(hours) ;
                temp-at-hour:units = "Celsius" ;
                string temp-at-hour:long_name = "Placeholder for temperature data." ;
                temp-at-hour:valid_max = 70.f ;
}
```

CRUCIAL:
`ncdump` is an executable that is compiled with NetCDF-C API.
The ordering of variable dimension reflects row-major. Fortran users be aware!

Quiz

We have learned how to create definition of a variable. How would you keep track of the `varid` of tens of different variables?

Parallel I/O – Dataset Creation

Header files

C

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

- `netcdf_par.h` is needed for parallel I/O operations.
- Fortran and Python versions remain the same.

Fortran

```
use netcdf
```

Python

```
import netCDF4
```
(version 1.3.1 and mpi4py needed!)

Creating a file - Parallel

C

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

Fortran

```
INTEGER nf90_create_par(PATH, CMODE, COMM, INFO, NCID)
  character (len = *), intent(in):: PATH
  integer, intent(in)             :: CMODE, COMM, INFO
  integer, intent(out)            :: NCID
```

If `use mpi_f08`, `COMM%MPI_VAL` and `INFO%MPI_VAL` required instead.

- Call is collective over `comm`!
- `cmode` must specify at least one of the following
  - `N[C/F90]_CLOBBER` – Create new file, overwrite if it existed before. [default]
  - `N[C/F90]_NOCLOBBER` – Only create new file if it did not exist before
- Choose file format on file creation
  - default - classic format (CDF-1 format)
  - `N[C/F90]_64BIT_OFFSET` - 64-bit offset format (classic CDF-2 format)
  - `N[C/F90]_NETCDF4` - NetCDF4 format (HDF5)
- Additional `cmode` options can be combined with bitwise inclusive OR operator.
  - `N[C/F90]_MPIIO` is required in older versions, deprecated since 4.6.2
  - Since `N[C/F90]_CLOBBER` is the default behaviour, in order to allow overwrite of files,
  - example `cmode` can be just: `IOR(NF90_NETCDF4,NF90_MPIIO)` (FORTRAN)

Creating a file – Parallel Python version

Python

```
nc = Dataset('filename', 'w',format='NETCDF4',
parallel=True, comm=None, info=None)
```

- Instead of the `cmode`, python version calls it access mode
  - Read-only mode: `'r'`
  - File creation mode :
    - Write : `'w'`
    - Append : `'a' or 'r+'`
- Formatting of file, only relevant when access mode is `'w'`
  - `NETCDF3_CLASSIC`
  - `NETCDF3_64BIT_OFFSET`
  - `NETCDF3_64BIT_DATA`
  - `NETCDF4_CLASSIC` – HDF5 file that conforms to CDF-format
  - `NETCDF4`
- Relevant for parallel access:
  - Set `parallel=True`, requires mpi4py & parallel-enabled netcdf-c and HDF5 libraries.
  - `comm=None` implies `MPI_COMM_WORLD`, `info=None` implies `MPI_INFO_NULL`
  - `comm` and `info` are ignored if `parallel=False`

Error handling

C	`const char * nc_strerror(int status)`
Fortran	`character(len = 80) nf90_sterror(STATUS)` `  integer, intent(in) :: STATUS`

- Return status string representation
- `(NC/NF90)_NOERR` can be used to check status
- Example (C version):

```
err = nc_def_dim(ncid, "hours", dim_len, &dimid)
if (err != NC_NOERR) printf("%s\n", nc_strerror(err))
```

Define mode (Parallel)

- Definition of Dimensions, Variables and Attributes are collective calls in `comm`!

Writing variables

C

```
int nc_put_vara(int ncid, int varid,
                const MPI_Offset start[],
                const MPI_Offset count[],
                const void* var)
```

Fortran

```
integer nf90_put_var(NCID, VARID, VAR, START, COUNT,
                     STRIDE, IMAP)
  integer, intent(in)                    :: NCID, VARID
  any type & any dimension, intent(in):: VAR
  integer, dimension(:), optional,
                         intent(in)    :: START, COUNT,
                                          STRIDE, IMAP
```

- `MPI_Offset` type is to provide compatibility to CDF format.
- `const size_t*` type is the NetCDF-4/HDF5 format default definition.
- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- The `start` and `count` refers to the extent of the variable array in file.

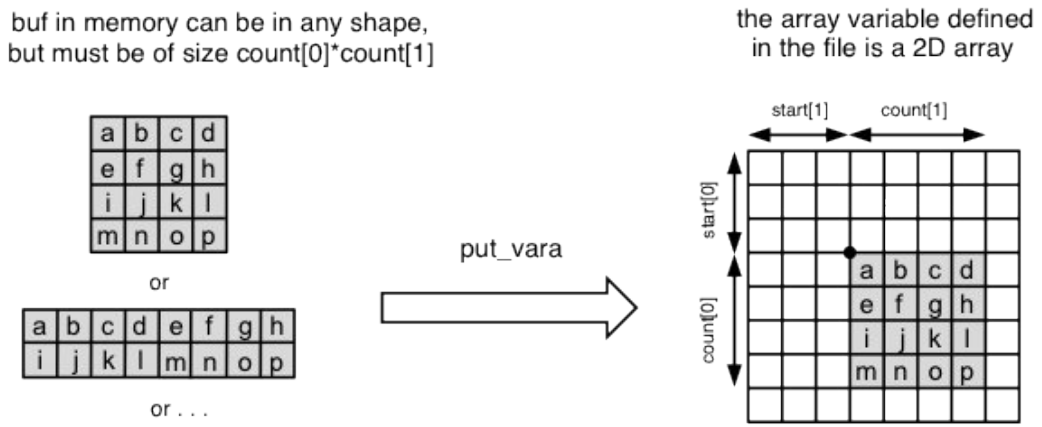
Python

```
var[start:start+count] = data
```

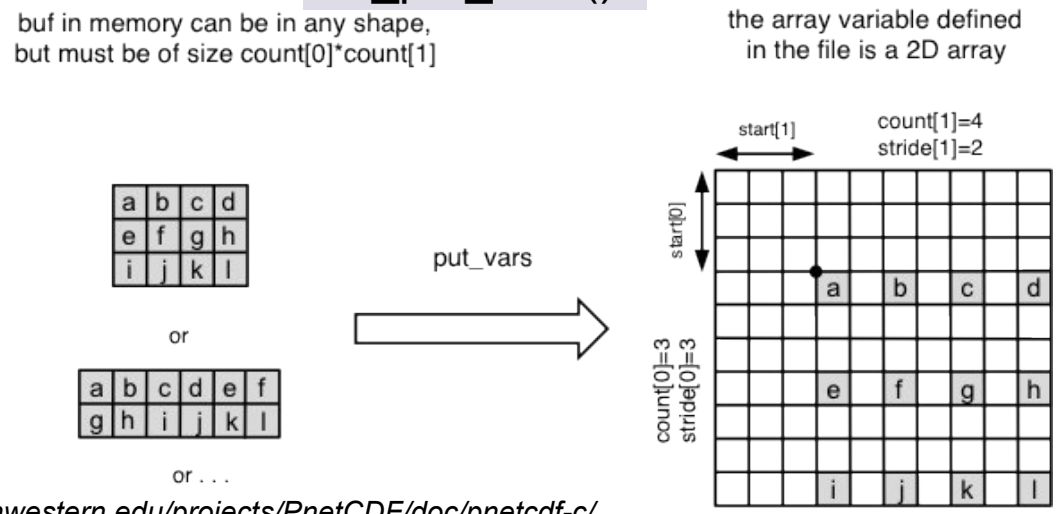
Writing variables

- Function name decides writing pattern in C version.
- Fortran version decides pattern via given argument.
- Mapped read/write version `_varm()` is deprecated (NetCDF v4.9.2)!

nc_put_vara()



nc_put_vars()



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

C

```
int nc_put_vars(int ncid, int varid,
                const size_t* start,
                const size_t* count,
                const ptrdiff_t* stride,
                const void* var)
```

Exercise

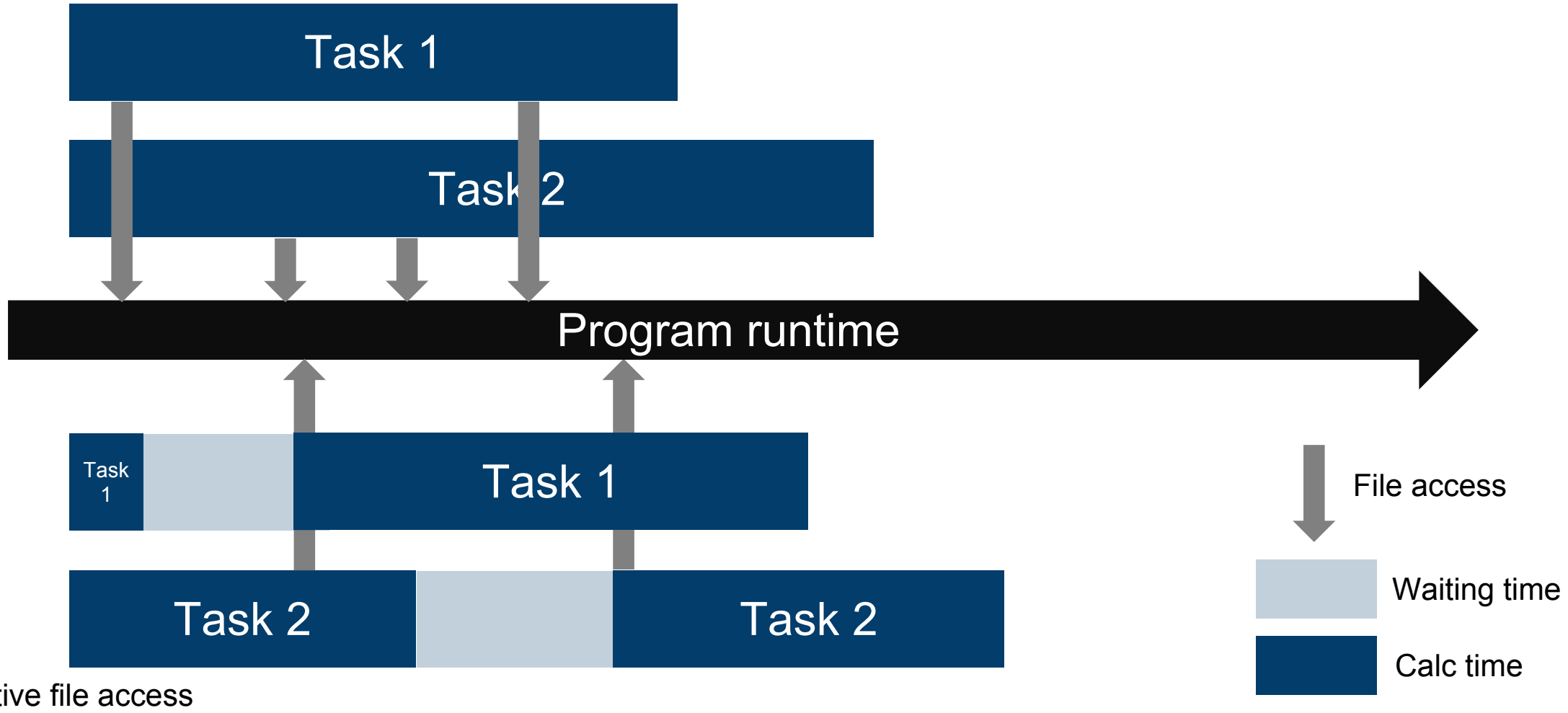
Exercise 2 – NetCDF Parallel write

- Description of the target application:
- Application should create a dataset in parallel.
- The created the dataset should be named `temp_data_par.nc`
- The dataset will have identical definitions as in Ex. 1 (sl. 26)
- The compiled executable should run on 4 MPI processes
- Each process fills in predefined number of temperature entries of a day
  - Rank 0 fills in 3 entries of 32°C
  - Rank 1 fills in 5 entries of 34°C
  - Rank 3 should fill in 38°C etc.
  - All ranks must know the total temperature entries before defining dimension.
  - All ranks should write into file without overwriting each other.
  - Template for the exercise is `exercise_2.[c|f90|py]`
  - Note: the programmer is not required to know the number of entries per rank :)

- Feel free to use the provided jobscript. It must be used to submit an MPI parallel job on the compute nodes.
  - `/p/project1/training2403/ParIO_course_material/submit.job`

Independent vs. collective reading/writing

Independent file access



Collective file access

Switching data modes

C
```
int nc_var_par_access(int ncid, int varid, int mode)
```

Fortran
```
integer nf90_var_par_access(NCID, VARID, MODE)
  integer, intent(in) :: NCID, VARID, MODE
```

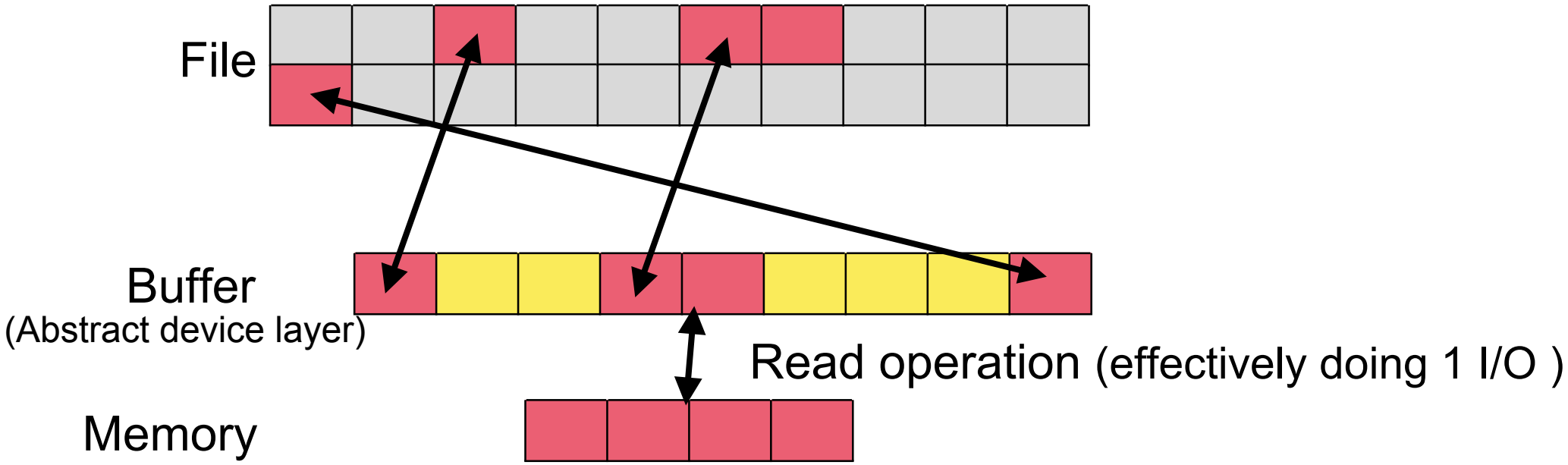
- Default mode of access is `independent`
- Switches variable to collective or independent data mode
- `mode` can be `N[C/F90]_INDEPENDENT` or `N[C/F90]_COLLECTIVE`
- CDF-format does not support per-variable access mode.
  - `N[C/F90]_GLOBAL` should be used in `varid`.

Python
```
var.set_collective(True)
```

Benefits of using middleware I/O libraries

Data sieving (implemented in ROMIO, active in MPICH based MPI lib + ported to OpenMPI)

- Usage of buffers to group non-contiguous data requests

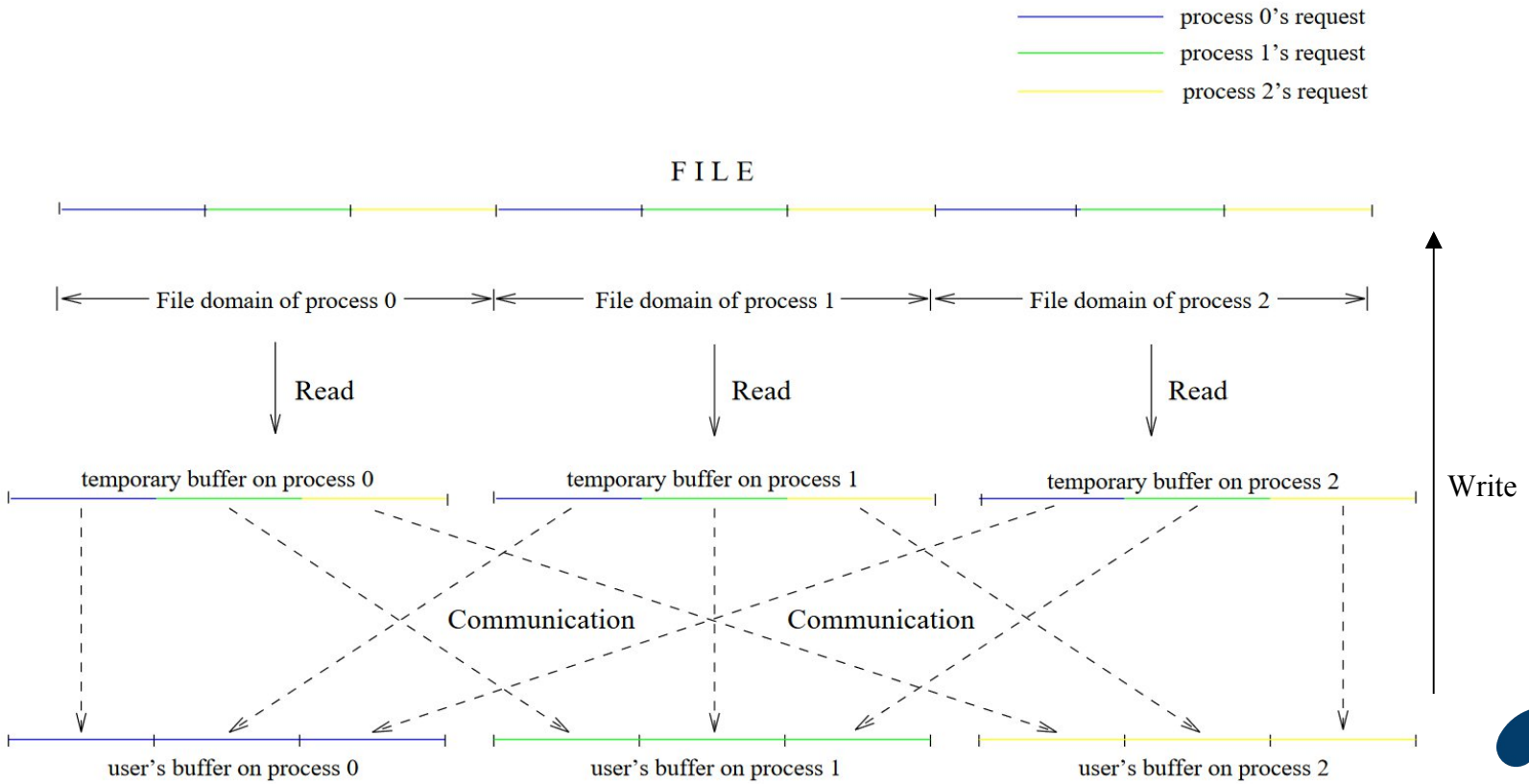


- MPI_File_set_view will affect the extent of the read buffer from file, even for write operations!

Collective buffering – interleaved access

Benefits of collective file access

- Auto optimization by MPI-IO.
- Constructs contiguous temporary buffer to store interleaved data before read/write.



Opening/Reading a dataset

Workflow: Reading a NetCDF data set (ncdump)

- `ncdump -h ./dataset`
- Open the data set
- Read definitions from file (using variable/dimension/attribute names from `ncdump`)
- Allocate memory according to shape of variables
- Read data from file into allocated memory
- Close file

Open an existing NetCDF data set

C

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

Fortran

```
integer nf90_open_par(FILENAME, OMODE, COMM, INFO,
                      NCID)
  character (len = *), intent(in) :: FILENAME
  integer, intent(in) :: OMODE, COMM, 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
  - `N[C/F90]_WRITE` – Open file for any kind of change to the file
  - `N[C/F90]_NOWRITE` – Open the file read-only
- Opened file is directly in data mode!
- Python function call is previously described in sl. 18

Reading Variables

C

```
int nc_inq_varid(int ncid, const char* name,int*
varid)
```

Fortran

```
integer nf90_inq_varid(NCID, NAME, VARID)
  integer, intent(in) :: ncid
  character (len = *), intent( in) :: name
  integer, intent(out) :: varid
```

- Uses both `ncid` and the `name` from `ncdump`. Returns `varid`.
- `varid` is the id annotated variable, required for detailed investigation of variable.
- Python version stores variables similar to `numpy` dictionary entries.
- Specific variable is accessed via variable name.

Python

```
nc = netCDF.Dataset("filename", ...)
print(nc.variables)
```

Reading Variables

C

```
int nc_inq_var(int ncid, int varid, char* name,
nc_type* xtype, int* ndims,
                int* dimids, int* n_atts)
```

Fortran

```
integer nf90_inquire_variable(NCID, VARID, NAME,
XTYPE, NDIMS, DIMIDS, N_ATTS)
  integer, intent(in)      :: NCID, VARID
  character (len = *), optional, intent(out) :: NAME
  integer, optional, intent(out) :: XTYPE, NDIMS
  integer, dimension(:), optional, intent(out) :: DIMIDS
  integer, optional, intent(out) :: N_ATTS
```

- Using the previously obtained `varid`, information of the name, dimensions and types can be obtained.
- Obtained `xtype`, `ndims` and `dimids` can be used to allocate array to store opened data.

C

```
int nc_inq_dim(int ncid, int dimid, char* name, size_t* len)
```

Fortran

```
integer nf90_inquire_dimension(ncid, dimid, name, len)
  integer,                    intent( in) :: ncid, dimid
  character (len = *), optional, intent(out) :: name
  integer,             optional, intent(out) :: len
```

Reading Variables - Python

- Using the previously obtained `variable name`, dimensions' lengths and names can be obtained.

Python

```
nc = netCDF.Dataset("filename", ...)
print(nc.variables["var_name"].shape)
print(nc.variables["var_name"].dimensions)
```

- Entries of the dimensions' lengths and names can be accessed like numpy array indices.

Python

```
dim_length_array = nc.variables["var_name"].shape
first_dim_length = dim_length_array[0]
```

Reading Variables

C

```
int nc_get_vara(int ncid, int varid,
                const size_t*  start,
                const size_t*  count,
                void* values)
```

Fortran

```
integer nf90_get_var(ncid, varid, values,
                     start, count, stride, map)
  integer, intent( in) :: ncid, varid
  any valid type, scalar or array, intent(out) :: values
  integer, dimension(:),
       optional, intent( in) :: start, count, stride, map
```

- Reads a *slab* of data referenced by `ncid` and `varid`
- Slab is defined by *n*-dimensional arrays `start` and `count`
- The `start` and `count` refers to the extent of the variable array in file.
- Read data is stored in `values`

Reading Variables - Python

- Copying the complete actual data to python array uses "`[:]`" index notation, regardless of n-dimensional array

Python

```
var = nc.variables["var_name"]
data_array = var[:]
```

- Slices of total dataset can be selected by giving proper ranges in each dimension!

Python

```
var = nc.variables["var_name"]
data_array = var[2:4,:]
```

Exercise

Exercise 3 – Reading NetCDF Dataset

- Your task is to implement an application that reads the file content in parallel.

- Compile and run `data_generate.[c/f90]` on login node
  - This generates `temp_2D_data.nc`
  - Use `ncdump` to see the file structure.

- The reader application should run on 4 MPI ranks, reading `temp_2D_data.nc`.
- The dataset contains hourly temperature measurements across 8 stations.
- Your application should coordinate the reading pattern, such that each process reads hourly data of 2 stations.
- Each MPI process then prints its own two sets of 24 hourly temperature values.
- Compare the printed application output with `ncdump` for correctness!

- Feel free to use the given template for this exercise, or adapt from previous exercises.
  - `$USER/exercises/NetCDF/exercise_3.[c|f90|py]`

Reading dataset- without ncdump???

It is indeed possible!

- Dataset structure can be reconstructed from file itself.
- General knowledge of NetCDF structure is sufficient (sl. 4)
- Refer to the user guide for comprehensive list of `inquire` functions
  - https://docs.unidata.ucar.edu/netcdf-c/current/modules.html
  - https://docs.unidata.ucar.edu/netcdf-fortran/4.6.1/index.html

What is the possible motivation?

- To write an application that can open ANY NetCDF files in general.
- In (edge) cases, where `ncdump` is not available.

Requirement

- Semantic information must be encoded in names and attributes
  - Conventions need to be set up and used for a given data set class

Performance hints

Chunking

C

```
int nc_def_var_chunking(ncid, varid, mode, size_t* chunksize)
```

Fortran

```
integer nf90_def_var(ncid, name, xtype, dimids, varid,
contiguous, chunksizes)
  logical, optional, intent(in) :: contiguous
  integer, optional, dimension(:), intent(in) :: chunksizes
```

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)

- `mode` can be `NC_CONTIGUOUS` or `NC_CHUNKED`
- `chunksize` must be defined for each dimension
- Only for Fortran version, link to detailed subroutine arguments:
  - https://github.com/Unidata/netcdf-fortran/blob/main/docs/netcdf-f90-sec6-variables.md#usage
- Python, like Fortran, defines chunksizes within `createVariable` function (see sl. 20)

Python

```
var = nc.createVariable('name', 'numpy_type',('dimension',),
chunksizes=[ , , ])
```