Parallel I/O with netCDF-4

Olga Abramkina[1, 2]

[1] Université Paris-Saclay, UVSQ, CNRS, CEA, **Maison de la Simulation**, Gif-sur-Yvette, France

[2] CNRS, **IDRIS**, Orsay, France

High Performance Parallel IO and post-processing

Maison de la Simulation, March 1-3, 2021

Goals

1. To understand the netCDF "ecosystem"
2. Usage of the netCDF-4 libraries
   - To learn about and to practice basic usage of the netCDF-4 libraries
   - To learn about advanced features of the netCDF-4
3. To get familiarized with available software for displaying and manipulating netCDF files

Table of Contents I

Table of Contents II

NetCDF

NetCDF (Network Common Data Form)

- NetCDF developed by Unidata at UCAR since 1988
- `https://www.unidata.ucar.edu/software/netcdf`
- It includes:
  1. Data and storage model: classic and **enhanced netCDF4/HDF5** formats
  2. Libraries: the **C core library** and **APIs** for various languages
- Supported languages:
  - By Unidata: C, Fortran, Java
  - Third party: C++, Python, R, Ruby, Matlab
- Parallel I/O since release 4.0 (2008), netCDF4/HDF5 format

NetCDF4 advantages

- NetCDF dataset format:
  - Self-describing*: a netCDF file includes information about the data it contains.
  - Portable*: a netCDF file can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
  - Archivable*: access to all earlier forms of netCDF data will be supported by current and future versions of the software.
  - Binary
- It provides efficient parallel I/O
- Wildly used by certain communities, particularly in environmental sciences (oceanography, atmospheric science, geoscience)
- It is supported by set of freely available tools for processing and visualization

*from https://www.unidata.ucar.edu/software/netcdf/

NetCDF classic file format

NetCDF file metadata

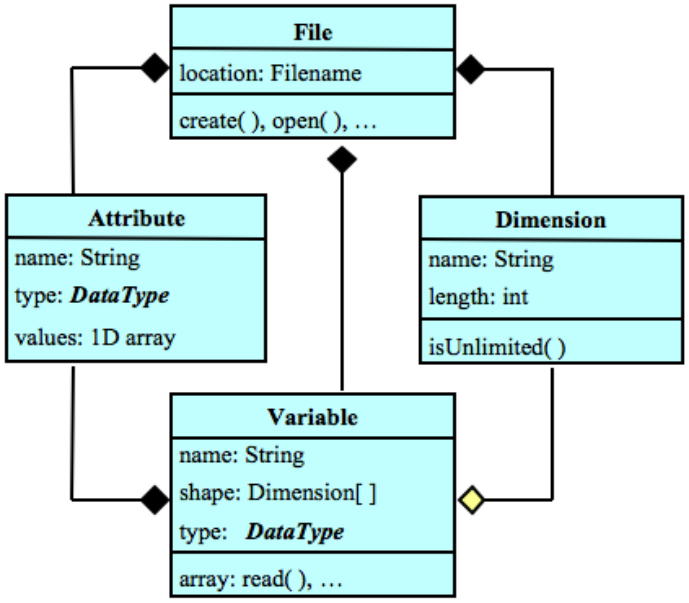
```
$ ncdump -h output.nc
netcdf output {
dimensions:
  x = 10 ;
  y = 10 ;
  time_counter = UNLIMITED ; // (10
     currently)
variables:
  double time_instant(time_counter) ;
    time_instant:standard_name = "time" ;
    time_instant:calendar = "gregorian" ;
    time_instant:units = "seconds" ;
  float field_2d(time_counter, y, x) ;
    field_2d:standard_name = "Temperature
     " ;
    field_2d:units = "K" ;

// global attributes:
    :name = "output" ;
    :Conventions = "CF-1.6" ;
}
```

Classic netCDF entities

- **Dimensions**
  One dimension can be of UNLIMITED type.
- **Variables:**
  multidimensional data arrays
- **Attributes:**
  data about data and the file (global attributes)

Attributes can be of any (allowed) type and size, but usually they don't contain large arrays.

NetCDF classic file format (continued)



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.

Variables and attributes have one of six primitive data types.

DataType
char
byte
short
int
float
double

Image from: https://www.unidata.ucar.edu/software/netcdf/

Classic netCDF storage model

- First, metadata is stored.
- Then fixed-size variables in the order they are added by user.
- Lastly, variables with an unlimited dimension interleaved along the unlimited dimension.

NetCDF4/HDF5 enhanced file format

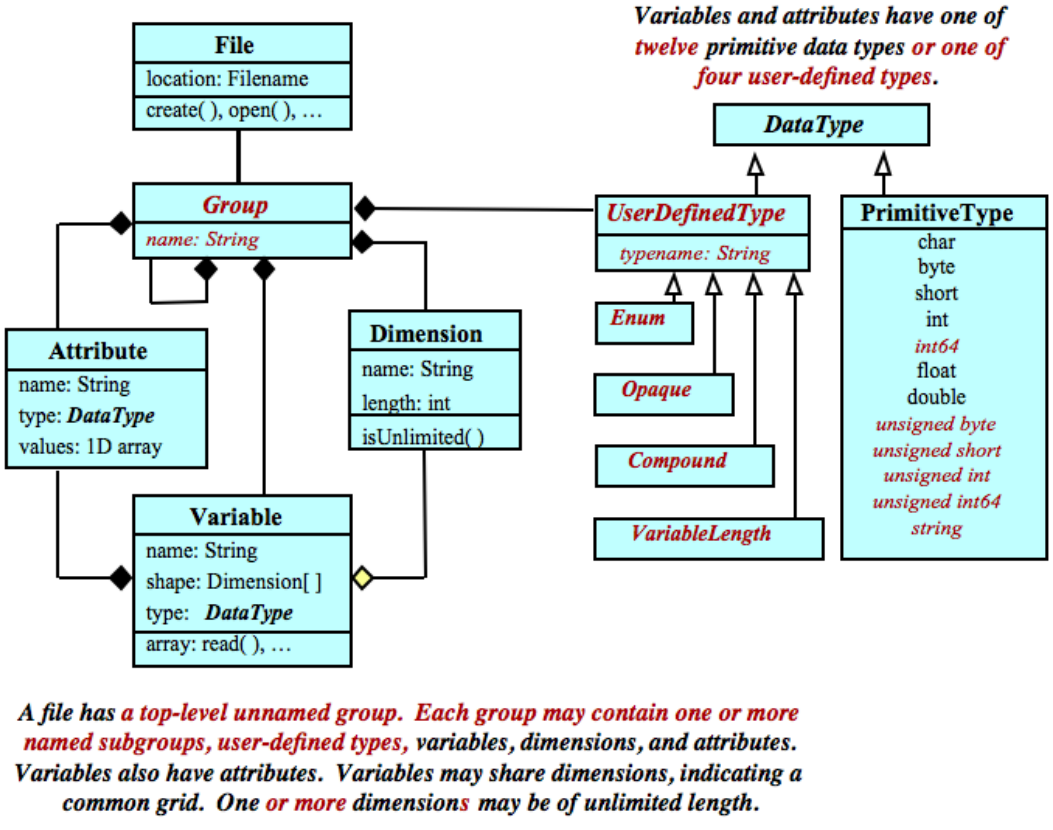


Image from: https://www.unidata.ucar.edu/software/netcdf/

NetCDF4 features:

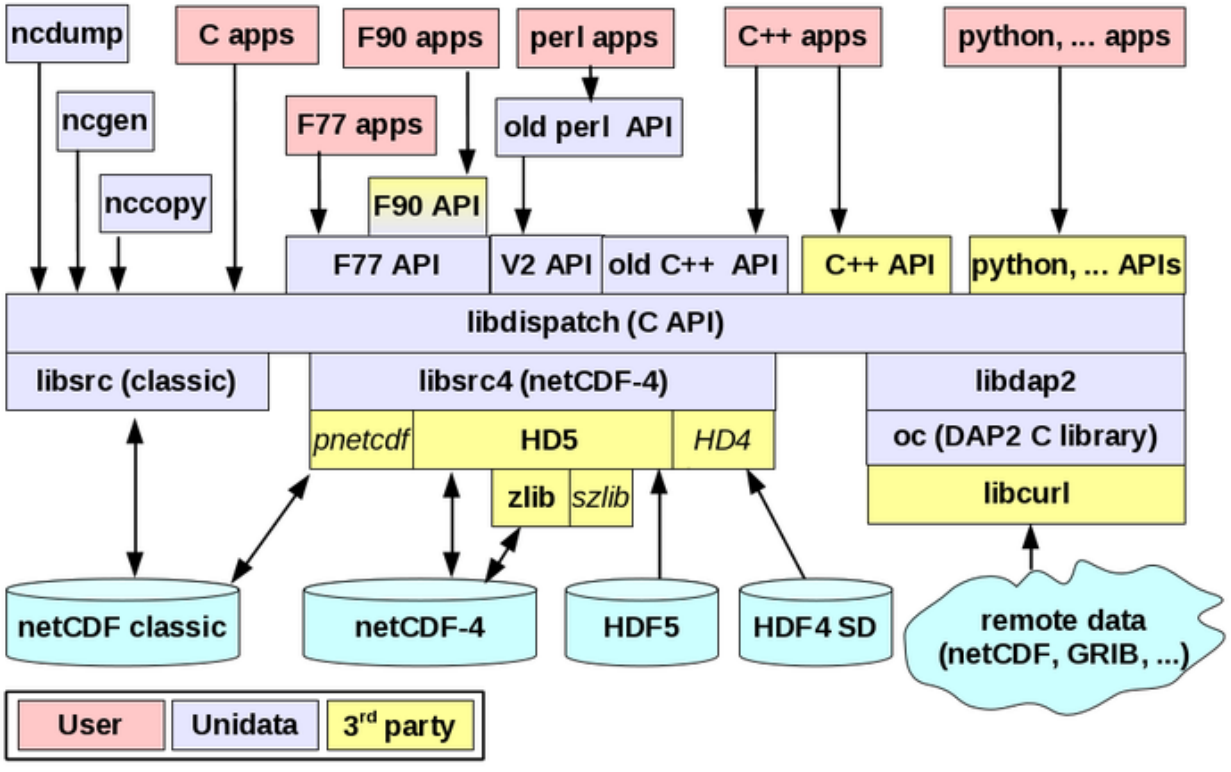
- Groups
- Data types: user-defined and more primitive types
- Multiple unlimited dimensions

NetCDF4 files are HDF5 compliant (the inverse is not always the case).

NetCDF metadata conventions

- Conventions: set of rules imposed on metadata (nomenclature) and data (e.g. missing values)
- Aim: to ensure the self-describing aspect of a dataset and thus
  - to make data clear and readable for a user
  - to remove redundancies
  - to facilitate sharing of data
- Examples:
  - COARDS (Cooperative Ocean-Atmosphere Research Data Service)
  - CF (Climate and Forecast) conventions
  - UGRID conventions for unstructured grids
  - ... and many more.

NetCDF library architecture



For parallel I/O to netCDF files use either:

- **netCDF4** library (netCDF4 format) or
- **PnetCDF** library (classic netCDF format) developed by Northwestern University and Argonne National Laboratory starting 2001

Image from: https://www.unidata.ucar.edu/software/netcdf/

Table of Contents I

Table of Contents II

Parallel write to netCDF

1. Create the file
   `nc_create_par / nf90_create`
2. Define metadata
   - Dimensions
     `nc_def_dim / nf90_def_dim`
   - Variables
     `nc_def_var / nf90_def_var`
   - Attributes
     `nc_put_att / nf90_put_att`
3. Close metadata definition
   `nc_enddef / nf90_enddef`
   Once this command is executed file's header is written to disk. It is possible to change the header after writing data, but it will cause the data to be copied.
4. Write (put) data
   `nc_put_vara / nf90_put_var`
5. Close file
   `nc_close / nf90_close`

Collective and independent accesses

- NetCDF operations can be performed **collectively** or **independently**.
- **Independent accesses:** no data transfers between processes, but synchronization is required
- **Collective accesses:** communications between processes allowing to benefit from MPI I/O optimizations (merging smaller I/O requests into larger data accesses, buffering)
- All calls evolving writing metadata are collective.
- Write and read by default are done **independently**.
- In order to change the access mode, use function `nc_var_par_access / nf90_var_par_access`
- Access on variables having an **unlimited** (time) **dimension** should be done **collectively**.

Parallel write to NetCDF: Fortran example

```
program example_par_write
  use netcdf
  implicit none
  include 'mpif.h'

    integer :: ncid,idimid,jdimid,varid
  ...
  ! Create a file
  ierr = nf90_create(FILE_NAME, NF90_NETCDF4, ncid,&
                     comm=MPI_COMM_WORLD, info=MPI_INFO_NULL)

  ! Define two spatial dimensions
  ierr = nf90_def_dim(ncid, "ni", NI, idimid)
  ierr = nf90_def_dim(ncid, "nj", NJ, jdimid)

  ! Define a 2D variable and add an attribute
  ierr = nf90_def_var(ncid, "toto", NF90_INT, (/idimid, jdimid/),
       varid)
  ierr = nf90_put_att(ncid, varid, "standart_name", "Dummy variable")

  ! Close metadata definition
  ierr = nf90_enddef(ncid)
  ...
  ! Write data
  ierr = nf90_put_var(ncid, varid, data, start=starts, count=counts)

  ! Close the file
  ierr = nf90_close(ncid)
end program example_par_write
```

```
$ ncdump -h par_write.nc
netcdf par_write {
dimensions:
  ni = 10 ;
  nj = 10 ;
variables:
  int toto(nj, ni) ;
    toto:standart_name = "Dummy variable" ;
}
```

Parallel write to NetCDF: C example

```c
#include "mpi.h"
#include <netcdf.h>
#include <netcdf_par.h>
...
int main (void)
{
...
  // Create a netcdf file
  nc_create_par(filename, NC_NETCDF4, MPI_COMM_WORLD, MPI_INFO_NULL,
       &ncid);

  // Define two spatial dimensions
  nc_def_dim(ncid, "ni", NI, &idimid);
  nc_def_dim(ncid, "nj", NJ, &jdimid);

  // Define a 2D variable and add an attribute
  nc_def_var(ncid, "toto", NC_INT, NDIMS, dimids, &varid);
  nc_put_att(ncid, varid, "standart_name", NC_CHAR, strlen("Dummy
       variable"), "Dummy variable");

  // Close metadata definition
  nc_enddef(ncid);

  // Write data
  nc_put_vara(ncid, varid, starts, counts, data);

  // Close the file
  nc_close(ncid);
...
}
```

```
$ ncdump -h par_write.nc
netcdf par_write {
dimensions:
  ni = 10 ;
  nj = 10 ;
variables:
  int toto(ni, nj) ;
    toto:standart_name = "Dummy variable" ;
}
```

NetCDF error handling

- Fortran

  Encapsulate NetCDF function calls into the following routine*:

```
subroutine check(status)
  integer,intent(in) :: status
  if(status /= nf90_noerr) then
    print *, trim(nf90_strerror(status))
    stop "Stopped"
  end if
end subroutine check
...
! ierr = nf90_put_var(ncid, varid, data, start=starts, count=counts)
call check(nf90_put_var(ncid, varid, data, start=starts, count=counts))
```

- C

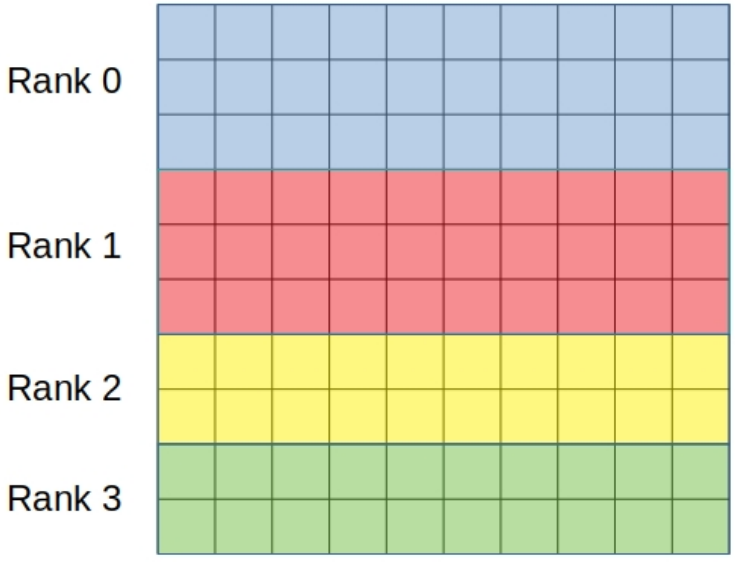
  Use the following macros*:

```
#define ERR(e) {printf("Error: %s\n", nc_strerror(e)); return 2;}
...
// nc_put_vara(ncid, varid, starts, counts, data);
if ((err =  nc_put_vara(ncid, varid, starts, counts, data))) ERR(err);
```

*Copyright: UCAR/Unidata

MPI domain decomposition

Example: 2D global domain 10x10, 4 processes MPI, horizontal domain decomposition



- Fortran

```
integer :: starts(NDIMS), counts(NDIMS)
! Values for rank 2
starts=(/1,7/)
counts=(/10,2/)
ierr = nf90_put_var(ncid, varid, data, start=
       starts, count=counts)
```

- C

```
size_t starts[NDIMS], counts[NDIMS];
// Values for rank 2
starts[0] = 0;       starts[1] = 6;
counts[0] = 10;      counts[1] = 2;
nc_put_vara(ncid, varid, starts, counts, data);
```

Hands-on exercise 1: parallel write

- Make the copy:
  `cp -r $PROJECT_training2022/netCDF $HOME`
- Environment: Intel compilers, IntelMPI
  `$ source ./netcdf.env`
- Examples: `examples/`
- Hands-on templates and solutions: `hands_on/`
- Compilation (either C or Fortran):
  `make hands_on_par_write`
- Execute:
  `srun -n 2 -reservation=parallel-io-day2`
  `./hands_on_par_write`
- Check the resulting file
  `ncdump ./par_write.nc`
- Useful links:
  - Fortran 90 Interface Guide
  - NetCDF Functions
  - Examples in C

Hands-on exercise 1: parallel write (cont'd)

Exercise:

- Create a file containing variables `lat`, `lon` and a temporal 2D variable `var_2d_temp`
- Add attributes, so that the file header looks as following (created by C library):

```
$ ncdump -h par_write.nc
netcdf par_write {
dimensions:
  lat = 5 ;
  lon = 10 ;
  time = UNLIMITED ; // (2 currently)
variables:
  float lat(lat) ;
    lat:standart_name = "latitude" ;
    lat:units = "degrees_north" ;
  float lon(lon) ;
    lon:standart_name = "longitude" ;
    lon:units = "degrees_east" ;
  int var_2d_temp(time, lat, lon) ;
    var_2d_temp:standart_name = "dummy_temporal_variable" ;
```

- Write all three variables in parallel, i.e. each process writes its data, thus the need for `count` and `start` parameters

Remember to set collective access on the temporal variable.

Parallel write: solution in Fortran

```
program hands_on_par_write
  use netcdf
  implicit none
  include 'mpif.h'

  integer :: rank
  integer :: mpisize
  integer :: ierr

  character (len = *), parameter :: FILE_NAME = "par_write.nc"
  integer,parameter :: NDIMS=2
  integer,parameter :: NI_GLO=5
  integer,parameter :: NJ_GLO=10
  integer,parameter :: NTIME=2

  real:: lon_glo(NJ_GLO),lat_glo(NI_GLO)
  integer :: field_glo(NI_GLO,NJ_GLO)
  real,allocatable :: lon(:),lat(:)
  integer,allocatable :: field(:,:)
  integer :: ni,ibegin,nj,jbegin
  integer :: t
  integer :: ncid,lat_dimid,lon_dimid,timedimid,varid,lonid,latid

  integer :: dimids(NDIMS), starts(NDIMS), counts(NDIMS)

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,mpisize,ierr)
```

Parallel write: solution in Fortran (cont'd)

```
! Create the netcdf file.
call check(nf90_create(FILE_NAME,IOR(NF90_NETCDF4,NF90_MPIIO),ncid,&
                       comm=MPI_COMM_WORLD,info=MPI_INFO_NULL))

! Define the dimensions. NetCDF will hand back an ID for each.
call check(nf90_def_dim(ncid, "lat", NI_GLO, lat_dimid))
call check(nf90_def_dim(ncid, "lon", NJ_GLO, lon_dimid))
call check(nf90_def_dim(ncid, "time", NF90_UNLIMITED, timedimid))

! Initialize global arrays.
call init_global(lon_glo,lat_glo,field_glo)

! Decompose global domain. Returned values: ni,nj,ibegin,jbegin.
call decompose_domain(mpisize,rank,ni,ibegin,nj,jbegin)

! Allocate and fill in local arrays.
allocate(lat(ni),lon(nj),field(ni,nj))
lat(:)=lat_glo(ibegin:ibegin-1+ni)
lon(:)=lon_glo(jbegin:jbegin-1+nj)
field(:,:) = field_glo(ibegin:ibegin-1+ni,jbegin:jbegin-1+nj)

! Define variable containing latitude and add its attributes.
call check(nf90_def_var(ncid, "lat", nf90_float, (/lat_dimid/), latid))
call check(nf90_put_att(ncid, latid, "standart_name", "latitude"))
call check(nf90_put_att(ncid, latid, "units", "degrees_north"))

! Define variable containing longitude and add its attributes.
call check(nf90_def_var(ncid, "lon", nf90_float, (/lon_dimid/), lonid))
```

Parallel write: solution in Fortran (cont'd)

```
call check(nf90_put_att(ncid, lonid, "standart_name", "longitude"))
call check(nf90_put_att(ncid, lonid, "units", "degrees_east"))

! Define variable var and add an attribute.
call check(nf90_def_var(ncid, "var_2d_temp", nf90_int, &
                  (/ lat_dimid, lon_dimid, timedimid /), varid))
call check(nf90_put_att(ncid, varid, "standart_name", "dummy_temporal_variable"))

! End define mode. This tells netCDF we are done defining metadata.
call check(nf90_enddef(ncid))

! Write latitude and longitude
call check(nf90_put_var(ncid, latid, lat, start=(/ibegin/), count=(/ni/)))
call check(nf90_put_var(ncid, lonid, lon, start=(/jbegin/), count=(/nj/)))

! Set mode to collective for the temporal variable.
call check(nf90_var_par_access(ncid, varid, nf90_collective))

! Write the temporal variable.
do t=1,NTIME
  call check(nf90_put_var(ncid, varid, field, start=(/ibegin,jbegin,t/), count=(/ni,nj,1/)))
enddo

! Close file.
call check(nf90_close(ncid))

deallocate(lon, lat, field)
call MPI_FINALIZE(ierr)
```

Parallel write: solution in Fortran (cont'd)

```
contains
! This subroutine comes with the netCDF package.
! It handles errors by printing an error message  and exiting with a non-zero status.
subroutine check(status)
  integer,intent(in) :: status
  if(status /= nf90_noerr) then
    print *, trim(nf90_strerror(status))
    stop "Stopped"
  end if
end subroutine check

! This subroutine fills in global arrays containing values of the variable, longitude, and
     latitude.
subroutine init_global(lon_glo,lat_glo,field_glo)
real,intent(inout) :: lon_glo(NJ_GLO),lat_glo(NI_GLO)
integer,intent(inout) :: field_glo(NI_GLO,NJ_GLO)
integer i,j

do i=1,NI_GLO
  lat_glo(i) = -90.+180./NI_GLO*(i-0.5)
enddo
do j=1,NJ_GLO
  lon_glo(j) = -180.+360./NJ_GLO*(j-0.5)
  do i=1,NI_GLO
    field_glo(i,j)=(i-1)+(j-1)*NI_GLO
  enddo
enddo
end subroutine init_global
```

Parallel write: solution in Fortran (cont'd)

```
  ! This subroutine decomposes global domain.
  subroutine decompose_domain(mpisize,rank,ni,ibegin,nj,jbegin)
    integer,intent(in) :: mpisize,rank
    integer,intent(out) :: ni,ibegin,nj,jbegin
    integer :: n
    ni=NI_GLO ; ibegin=1
    jbegin=0
    do n=0,mpisize-1
      nj=NJ_GLO/mpisize
      if (n<MOD(NJ_GLO,mpisize)) nj=nj+1
      if (n==rank) exit
      jbegin=jbegin+nj
    enddo
    jbegin=jbegin+1
  end subroutine decompose_domain

end program hands_on_par_write
```

Parallel write: solution in Fortran (cont'd)

```
$ ncdump -h par_write.nc
netcdf par_write {
dimensions:
  lat = 5 ;
  lon = 10 ;
  time = UNLIMITED ; // (2 currently)
variables:
  float lat(lat) ;
    lat:standart_name = "latitude" ;
    lat:units = "degrees_north" ;
  float lon(lon) ;
    lon:standart_name = "longitude" ;
    lon:units = "degrees_east" ;
  int var_2d_temp(time, lon, lat) ;
    var_2d_temp:standart_name = "dummy_temporal_variable" ;
}
```

Parallel write: solution in C

```c
#include "mpi.h"
#include <netcdf.h>
#include <netcdf_par.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NDIMS      2
#define NI_GLO     5
#define NJ_GLO     10
#define NTIME      2

#define ERR(e) {printf("Error: %s\n", nc_strerror(e)); return 2;}

// Function decomposes global domain of size NI_GLO*NJ_GLO in horizontal direction into local
      domains
void decompose_domain(int mpisize, int mpirank, int* ni, int* ibegin, int* nj, int* jbegin)
{
    *nj=NJ_GLO;
    *ibegin=0;
    *jbegin=0;
    for (int n=0; n<mpisize; ++n)
    {
      *ni=NI_GLO/mpisize;
      if (n<(NI_GLO%mpisize)) ++(*ni);
      if (n==mpirank) break;
      (*ibegin)+=(*ni);
    }
}
```

Parallel write: solution in C (cont'd)

```c
// Function fills in local domain data
void fill_local_data(int* data, int ni, int ibegin, int nj, int jbegin)
{
  int i,j;
  for(i=0; i < ni; i++)
    for(j=0; j < nj; j++)
      data[j+i*nj] = (j+jbegin) + (i+ibegin)*nj;
}

// Function fills in local lon and lat
void fill_lon_lat(float* lon, float* lat, int ni, int ibegin, int nj, int jbegin)
{
  for(int i=0; i < ni; i++)
    lat[i] = -90. + 180./NI_GLO*(i+ibegin+0.5);
  for(int j=0; j < nj; j++)
    lon[j] = -180. + 360./NJ_GLO*(j+jbegin+0.5);
}

int main (void)
{
  int mpirank, mpisize;
  /* Variables holding netCDF ids of the file and three variables stored in it (longitude,
       latitude and a dummy 2D variable) */
  int ncid, lonid, latid, varid;
  /* Variables holding three dimensions: longitude, latitude and a temporal dimension  */
  int lon_dimid, lat_dimid, temp_dimid;
  /* Array holding three dimensions of the dummy 2D varialbe */
  int var_dimids[NDIMS+1];
```

Parallel write: solution in C (cont'd)

```c
/* Start and count for parallel write of longitude, latitude and the variable */
size_t start, count;
size_t var_starts[NDIMS+1], var_counts[NDIMS+1];
int err;
int ni, nj, ibegin, jbegin;
char filename[128] = "par_write.nc";

//-----------------------------------------
// Initialization MPI
//-----------------------------------------

MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD,&mpirank);
MPI_Comm_size(MPI_COMM_WORLD,&mpisize);

//-----------------------------------------
//  NetCDF metadata definitions
//-----------------------------------------

/* Create a netcdf file */
if (err=nc_create_par(filename, NC_NETCDF4, MPI_COMM_WORLD, MPI_INFO_NULL, &ncid))
  ERR(err);

/* Define two spatial dimensions */
if (err=nc_def_dim(ncid, "lat", NI_GLO, &lat_dimid))
  ERR(err);
if (err=nc_def_dim(ncid, "lon", NJ_GLO, &lon_dimid))
  ERR(err);
```

Parallel write: solution in C (cont'd)

```
/* Define a temporal dimension */
if (err=nc_def_dim(ncid, "time", NC_UNLIMITED, &temp_dimid))
  ERR(err);

/* Define 1D variables that will hold latitude and longitude */
if (err=nc_def_var(ncid, "lat", NC_FLOAT, 1, &lat_dimid, &latid))
  ERR(err);
if (err=nc_put_att(ncid, latid, "standart_name", NC_CHAR, strlen("latitude"), "latitude"))
  ERR(err);
if (err=nc_put_att(ncid, latid, "units", NC_CHAR, strlen("degrees_north"), "degrees_north"))
  ERR(err);
if (err=nc_def_var(ncid, "lon", NC_FLOAT, 1, &lon_dimid, &lonid))
  ERR(err);
if (err=nc_put_att(ncid, lonid, "standart_name", NC_CHAR, strlen("longitude"), "longitude"))
  ERR(err);
if (err=nc_put_att(ncid, lonid, "units", NC_CHAR, strlen("degrees_east"), "degrees_east"))
  ERR(err);

/* Define a 2D temporal variable and add an attribute */
var_dimids[0] = temp_dimid;
var_dimids[1] = lat_dimid;
var_dimids[2] = lon_dimid;
if (err=nc_def_var(ncid, "var_2d_temp", NC_INT, NDIMS+1, var_dimids, &varid))
  ERR(err);
if (err=nc_put_att(ncid, varid, "standart_name", NC_CHAR, strlen("dummy_temporal_variable"), "
    dummy_temporal_variable"))
  ERR(err);
```

Parallel write: solution in C (cont'd)

```
/* Close metadata definition */
if (err=nc_enddef(ncid))
  ERR(err);

//-----------------------------------------
// Local data initialization
//-----------------------------------------

/* Decompose global domain. Returned values: ni,nj,ibegin,jbegin */
decompose_domain(mpisize,mpirank,&ni,&ibegin,&nj,&jbegin);

/* Fill in local data: lon, lat and temporal varialbe */
int *data = (int *)malloc(sizeof(int)*ni*nj);
fill_local_data(data, ni, ibegin, nj, jbegin);

float *lon = (float *)malloc(sizeof(float)*nj);
float *lat = (float *)malloc(sizeof(float)*ni);
fill_lon_lat(lon, lat, ni, ibegin, nj, jbegin);

//-----------------------------------------
// Write data and close the file
//-----------------------------------------

/* Write lon, lat.
 * Here, the third and the fourth arguments, start and count, should be of type 'size_t *'.*/
start = ibegin; count = ni;
if (err=nc_put_vara(ncid, latid, &start, &count, lat)) ERR(err);
start = jbegin; count = nj;
```

Parallel read from netCDF

1. Open a file for reading in parallel
   `nc_open_par / nf90_open`
2. Get (inquire about) dimension IDs given their names
   `nc_inq_dimid / nf90_inq_dimid`
3. Get (inquire about) dimension sizes given their IDs
   `nc_inq_dimlen / nf90_inquire_dimension`
4. Get (inquire about) a variable given its name
   `nc_inq_varid / nf90_inq_varid`
5. Read (get) a variable given its ID
   `nc_get_vara / nf90_get_var`
6. Close file
   `nc_close / nf90_close`

Parallel read from NetCDF: Fortran example

```
program example_par_read
  use netcdf
  implicit none
  include 'mpif.h'

  integer :: ncid,idimid,jdimid,varid
  ...
  ! Open a file for read
  call check( nf90_open(FILE_NAME, NF90_NOWRITE, ncid,&
              comm=MPI_COMM_WORLD, info=MPI_INFO_NULL) )

  ! Get IDs of the two spatial dimensions given their names
  call check( nf90_inq_dimid(ncid, "ni", nidimid) )
  call check( nf90_inq_dimid(ncid, "nj", njdimid) )

  ! Get the two spatial dimensions given their IDs
  call check( nf90_inquire_dimension(ncid, nidimid, len = ni_glo) )
  call check( nf90_inquire_dimension(ncid, njdimid, len = nj_glo) )

  ! Get ID of the spacial variable given its name
  call check( nf90_inq_varid(ncid, "toto", varid) )

  ! MPI domain decomposition of the global domain (ni_glo x nj_glo)
       should be provided by a user
  ...
  ! Read data
  call check( nf90_get_var(ncid, varid, field, &
                    start=(/ibegin,jbegin/), count=(/ni,nj/)) )

  ! Close the file
  ierr = nf90_close(ncid)
end program example_par_read
```

```
$ ncdump -h par_write.nc
netcdf par_write {
dimensions:
  ni = 5 ;
  nj = 10 ;
variables:
  int toto(nj, ni) ;
    toto:standart_name = "Dummy variable" ;
}
```

Parallel read from NetCDF: C example

```c
#include "mpi.h"
#include <netcdf.h>
#include <netcdf_par.h>
...
#define ERR(e) {printf("Error: %s\n", nc_strerror(e)); return 2;}
int main (void) {
...
  // Open a netcdf file
  if (err==nc_open_par(filename, NC_NOWRITE, MPI_COMM_WORLD,
       MPI_INFO_NULL, &ncid)) ERR(err);

  // Get IDs of two spatial dimensions given their names
  if (err==nc_inq_dimid(ncid, "ni",  &nidimid)) ERR(err);
  if (err==nc_inq_dimid(ncid, "nj",  &njdimid)) ERR(err);

  // Get two spatial dimensions given their IDs
  if (err==nc_inq_dimlen(ncid, nidimid, &ni_glo)) ERR(err);
  if (err==nc_inq_dimlen(ncid, njdimid, &nj_glo)) ERR(err);

  // Get ID of the variable given its name
  if (err==nc_inq_varid(ncid, "toto",  &varid)) ERR(err);
...
  // MPI domain decomposition of the global domain (ni_glo x nj_glo)
       should be provided by a user
  starts[0] = ibegin; starts[1] = jbegin;
  counts[0] = ni;     counts[1] = nj;
  if (err==nc_get_vara(ncid, varid, starts, counts, data)) ERR(err);

  // Close the file
  if (err==nc_close(ncid)) ERR(err);
...
}
```

```
$ ncdump -h par_write.nc
netcdf par_write {
dimensions:
  ni = 5 ;
  nj = 10 ;
variables:
  int toto(ni, nj) ;
    toto:standart_name = "Dummy variable" ;
}
```

Hands-on exercise 2: parallel read

Exercise:

- Read the three variables stored in the file `par_write.nc` created in the previous exercise
  (File is available in folder solutions.)
- Read `lat` and `lon` entirely by each process, but only a part of the temporal variable `var_2d_temp`.
  Functions for MPI domain decomposition are included into templates.

Table of Contents I

Table of Contents II

User defined types

- Enum: integer numbers are assigned to text values; the numbers are stored

  `examples/example_write_enum.c`
- Compound : as a C struct, a compound type is a collection of types

  `examples/example_write_struct.c`
- Opaque: arrays of unknown contents but of known size

  `examples/example_write_opaque.c`
- Variable length array: an array of arrays of which the member arrays can be of different lengths.
  Array elements can be of user defined type (enum, compound or opaque).

  `examples/example_write_vlen.c`

User defined types (cont'd)

- **Enum** (The example of enum is taken from Unidata)

```
netcdf write_enum {
types:
  uint enum cloud_types {Clear = 0, Cumulonimbus = 1, Stratus = 2,
      Stratocumulus = 3, Cumulus = 4, Altostratus = 5, Nimbostratus = 6,
      Altocumulus = 7} ;
dimensions:
  n = 10 ;
variables:
  cloud_types clouds(n) ;
data:

 clouds = Cumulonimbus, Stratus, Cumulus, Altocumulus, Clear, Clear, Clear,
    Clear, Clear, Clear ;
}
```

- **Compound**

```
netcdf write_struct {
types:
  compound point_3d {
    float x ;
    float y ;
    float z ;
  }; // point_3d
dimensions:
  n = 10 ;
variables:
  point_3d toto(n) ;
}
```

User defined types (cont'd)

- **Opaque** (Example 11-byte raw sensor data in hexadecimal notation from Unidata)

```
netcdf write_opaque {
types:
  opaque(11) opaque_t ;
dimensions:
  n = 5 ;
variables:
  opaque_t sensor_data(n) ;
data:

 sensor_data = 0X01020304050607080090A0B, 0XAABBCCDDEEFFEEDDCCBBAA,
    0XFFFFFFFFFFFFFFFFFFFFFF, 0XCAFEBABECAFEBABECAFEBA,
    0XCF0DEFACED0CAFE0FACADE ;
}
```

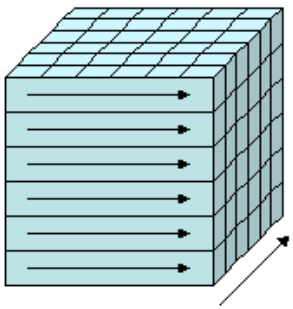
- **Variable length array**

```
netcdf write_vlen {
types:
  float(*) vlen ;
dimensions:
  n = 5 ;
variables:
  vlen ragged_array(n) ;
data:

 ragged_array = {0}, {1, 2}, {3, 4, 5}, {6, 7}, {8} ;
}
```

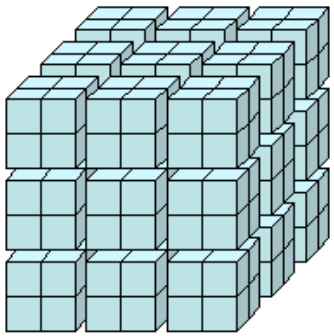
In many cases where VLENs seem useful, one should consider instead fixed array sizes and on-the-fly compression.

Chunking

Storage layout: contiguous or chunked



index order



chunked

By default, **contiguous** layout for **non-record** (fixed-size dimensions) variables and **chunked** layout for **record** (one or more unlimited dimensions) or **compressed** variables.

```
$ ncdump -s -h par_write.nc
netcdf par_write {
dimensions:
  lat = 5 ;
  lon = 10 ;
  time = UNLIMITED ; // (2 currently)
variables:
  float lat(lat) ;
    lat:standart_name = "latitude" ;
    lat:units = "degrees_north" ;
    lat:_Storage = "contiguous" ;
    lat:_Endianness = "little" ;
  float lon(lon) ;
    lon:standart_name = "longitude" ;
    lon:units = "degrees_east" ;
    lon:_Storage = "contiguous" ;
    lon:_Endianness = "little" ;
  int var_2d_temp(time, lat, lon) ;
    var_2d_temp:standart_name = "
        dummy_temporal_variable" ;
    var_2d_temp:_Storage = "chunked" ;
    var_2d_temp:_ChunkSizes = 1, 5, 10 ;
    var_2d_temp:_Endianness = "little" ;
...
```

Image from: https://www.unidata.ucar.edu/software/netcdf/workshops/most-recent/nc4chunking/WhatIsChunking.html

Read/write operations on chunked variables are done in chucks.

- How to get/set the chunk size?
  `nc_inq_var_chunking / nf90_inq_var_chunking`
  `nc_def_var_chunking / nf90_def_var_chunking`
- How to choose the chunk size?
  No rule of thumb, try it.
  Here are some discussions and documentation on chunking:
  - `https://www.unidata.ucar.edu/blogs/developer/entry/chunking_data_why_it_matters`
  - `https://www.unidata.ucar.edu/blogs/developer/en/entry/chunking_data_choosing_shapes`
  - `https://www.unidata.ucar.edu/software/netcdf/docs/netcdf_perf_chunking.html`

Data can be rechunked using, for example, the netCDF utility `nccopy`.

Compression

When using compression, it is applied on the per-chunk basis.

- On-the-fly HDF5/zlib compression

  `nc_def_var_deflate / nf90_def_var_deflate`

  Hands-on exercise 3: compression

  - Compilation (either C or Fortran): `make hands_on_compression`
  - Execution: `./hands_on_compression`
  - Play with the compression parameters, check the resulting file size
  - Useful links:
    - C documentation
    - Fortran documentation

- Post-treatement compression

  `nccopy -d1 original_file.nc compressed_file.nc`

Table of Contents I

Table of Contents II

NetCDF tool landscape

- NetCDF
  - ncdump: a netCDF file to a CDL (network Common Data form Language) text file
  - ncgen: a CDL file to a netCDF file
  - nccopy converts between netCDF file formats
- NCO (netCDF Operators) suit
  - ncks: kitchen sink
  - ncatted: attribute editor
  - ...
- Viewers
  - Ncview `http://meteora.ucsd.edu/~pierce/ncview_home_page.html`
  - Panoply `https://www.giss.nasa.gov/tools/panoply/`
  - VisIt `https://wci.llnl.gov/simulation/computer-codes/visit`
  - Ferret `https://ferret.pmel.noaa.gov/Ferret/`
  - VTK `https://vtk.org/`

NetCDF and NCO command line utilities

- Commands operating on netCDF files: basic usage

Print out the entire file to screen:

```
ncdump foo.nc
```

```
ncks foo.nc
```

Print out the metadata to screen:

```
ncdump -h foo.nc
```

```
ncks -m output.nc
```

Print out a variable to screen:

```
ncdump -v var foo.nc
```

```
ncks -v var foo.nc
```

ncatted: attribute editor

```
ncatted
```

NetCDF and NCO command line utilities (cont'd)

- Commands operating on netCDF files: advanced usage

Extract a data subset (5x5 2D domain, 3rd time step) into file toto.nc:

```
ncks -d y,0,5 -d x,0,5 -d time_counter,3 -v field_A output.nc toto.nc
```

(Here x, y, and time_counter are netCDF dimensions defined in output.nc)

Arithmetic operations with ncap2

```
ncap2
```

- Commands operating on CDL text files

Convert a CDL text file to netCDF:

```
ncgen -o foo.nc foo.cdl
```

Generate a C program to create a netCDF file based on the CDL file:

```
ncgen -lc foo.cdl
```

Hands-on exercises on tools

- Compare the ouput with ncdump, h5dump
- Starting from file examples/C/write_enum.nc generate a code in C, compare it with the original code that generated the netCDF file

Table of Contents I

Table of Contents II

Conclusion

- Advantages of using netCDF-4
  - Efficient parallel I/O library
  - Relatively simple interface
  - Well-established format
- How to choose an I/O library?
  - Use high-level libraries (HDF5, netCDF, PnetCDF, SIONlib, PDI)
  - Select according to format accepted in your scientific community
  - Prefer a library that offers the right amount of functionalities (more is not necessarily better)
  - Use libraries providing processes dedicated to asynchronous I/O and in-situ data analysis (DAMARIS (HDF5), XIOS (netCDF4))