PARALLEL I/O AND PORTABLE DATA FORMATS

PARALLEL NETCDF

22.02.2022 I SEBASTIAN LÜHRS (S.LUEHRS@FZ-JUELICH.DE)

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

PnetCDF or NetCDF4

- NetCDF4:
  - Function Prefix: `nc_... / nf[90]_...`
  - Uses **HDF5** or **PnetCDF** for parallel file access
  - `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
  - 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.

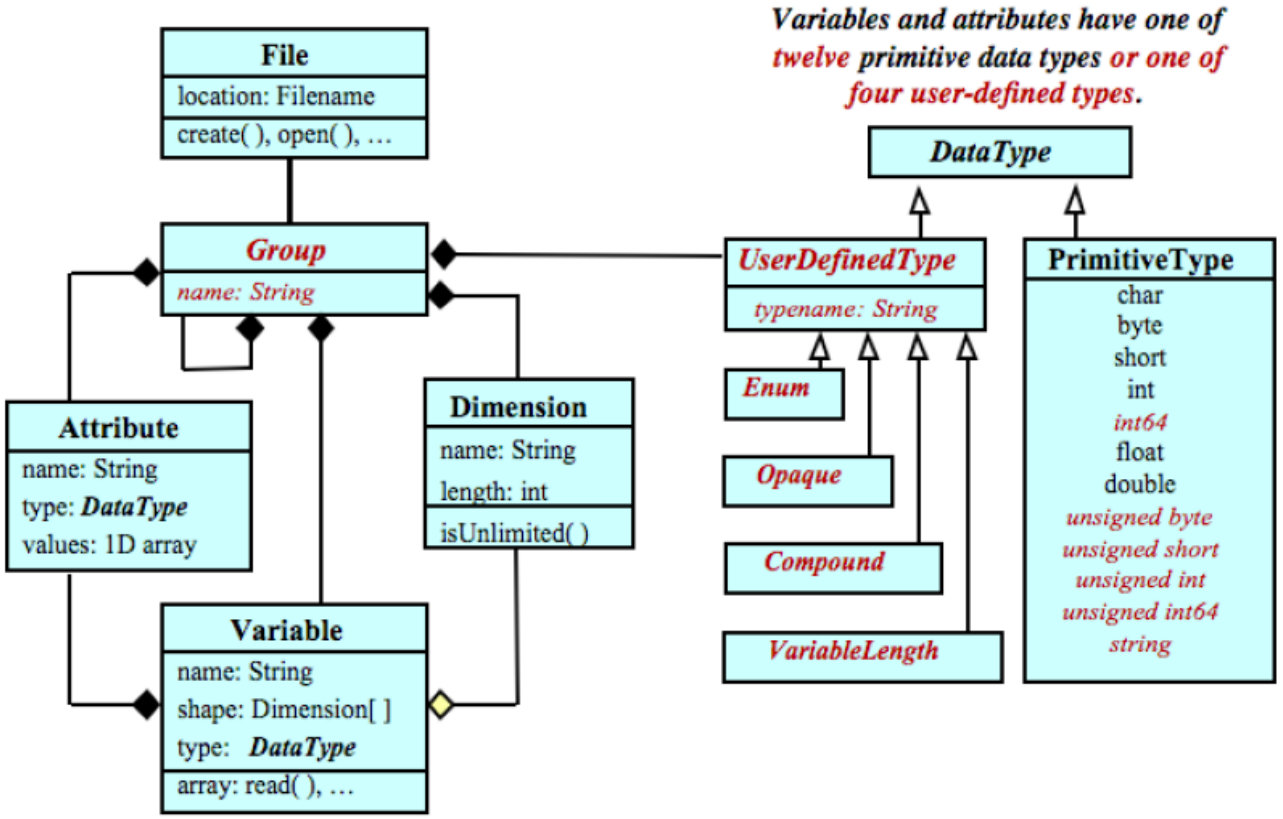
Variable

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*.

NetCDF4 model



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

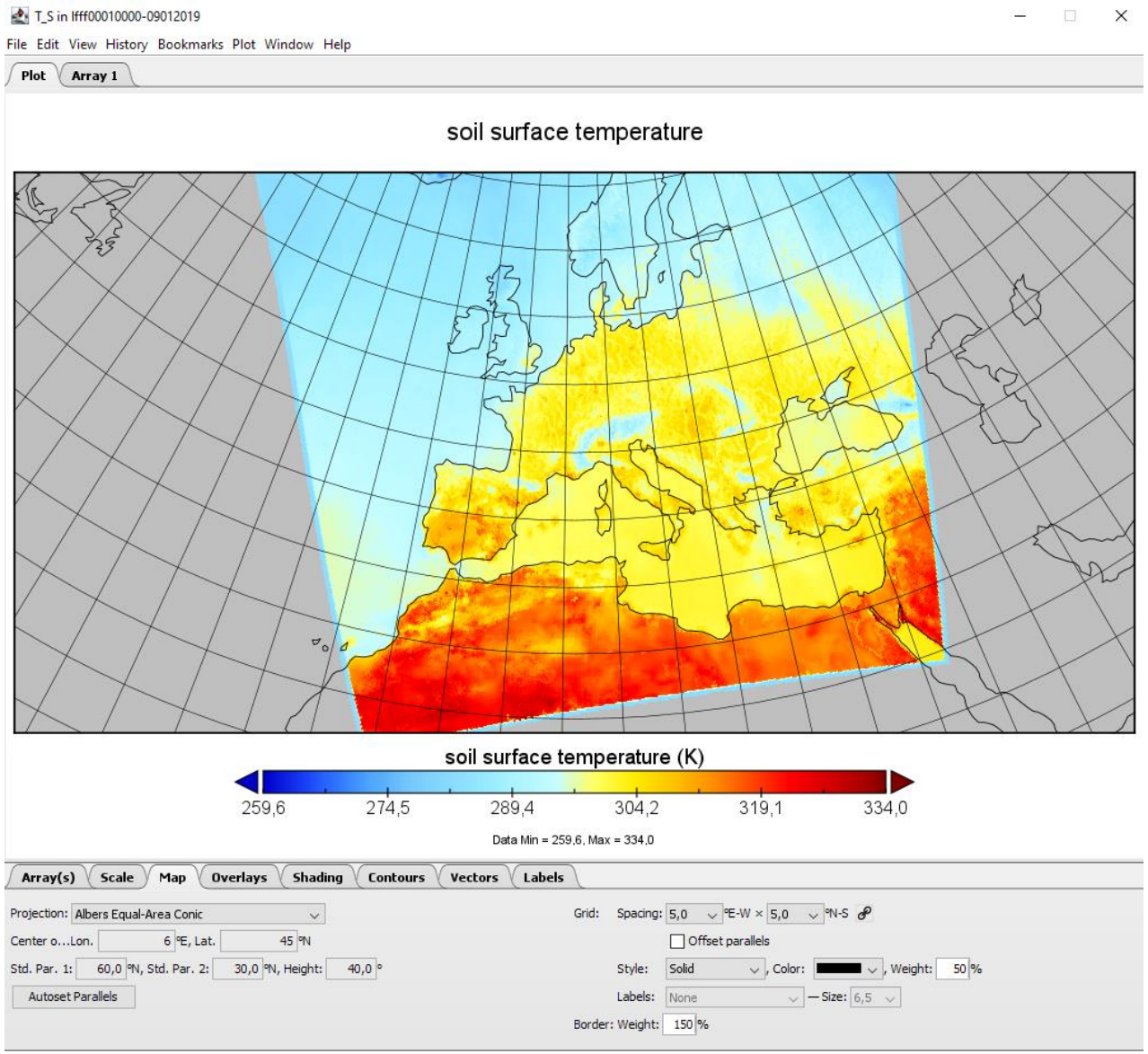
Viewer for NetCDF data

e.g. Panoply (NASA GISS)

https://www.giss.nasa.gov/tools/panoply/download/

Works well if we have some metadata
(we'll get back to that).

Useful alternative: ncview



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 can be ’UNLIMITED’
- 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
  - E.g. byte, character, short, int, float, double
- A position along a dimension can be specified as index
  - Starting at 0 in C and 1 in Fortran

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

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	NF90_BYTE	8-bit signed integer
NC_CHAR	NF90_CHAR	8-bit unsigned integer
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_SRING	NF90_STRING	Character string

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

Header files

C

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

Fortran

```
use netcdf
```

- Contain definition of
  - constants
  - functions

Python

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

Creating a file

C

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

Fortran

```
INTEGER NF90_CREATE_PAR(FILENAME, CMODE, COMM, INFO,
                        NCID)
  CHARACTER*(*) FILENAME
  INTEGER COMM, MODE, INFO, NCID
```

`comm%MPI_VAL` can be used for mpi_f08

- Call is collective over `comm`
- `ncid` is the id of the internal file handle
- `cmode` must specify at least one of the following
  - `(NC/NF90)_CLOBBER` – Create new file and overwrite, if it existed before
  - `(NC/NF90)_NOCLOBBER` – Create new file only, if it did not exist before
- Choose file format on file creation
  - default - classic format (PnetCDF)
  - `(NC/NF90)_64BIT_OFFSET` - 64-bit offset format (PnetCDF)
  - `(NC/NF90)_NETCDF4` - NetCDF4 format (HDF5) (`NC_MPIIO` is set in addition in older setups)

Creating a file

Python

```
nc = Dataset('filename', 'w', parallel=True,
             format='NETCDF4‘,
             comm=MPI.COMM_WORLD, info=MPI.Info())
```

- `MPI` comes from the mpi4py Python package: `from mpi4py import MPI`
- Instead of the creation mode `w` also `a` and `r` are possible
- `MPI_COMM_WORLD` and `MPI_INFO_NULL` are the default values

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=*) 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/NF90)_WRITE` – Open file for any kind of change to the file
  - `(NC/NF90)_NOWRITE` – Open the file read-only

Closing a file

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

Fortran
```
INTEGER NF90_CLOSE(NCID)
  INTEGER NCID
```

Python
```
nc.close()
```

- Close file associated with `ncid`

Error handling

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

Fortran
```
CHARACTER*80 NF90_STRERROR(STATUS)
  INTEGER STATUS
```

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

Exercise

Exercise 1 – NetCDF hello world

- Directory preparation:
  - `/p/project/training2202/netcdf/copy.sh`
  - `cd /p/project/training2202/<username>/netcdf`
- Create a parallel application (C or Fortran) which creates an empty NetCDF file
- You can use the provided template:
  `helloworld_netcdf.c` or `helloworld_netcdf.f90`
- Compile, link and execute the application

```
module load Intel ParaStationMPI # Load compiler and MPI
module load netCDF/4.8.1 # Load netCDF C libs
module load netCDF-Fortran/4.5.3 # Load netCDF Fortran libs

mpicc helloworld_netcdf.c -lnetcdf # Compile C style
mpif90 helloworld_netcdf.f90 -lnetcdff # Compile Fortran style
```

To start a job on the compute node:

```
srun -N 1 --ntasks-per-node=48 --reservation=pario-2022-02-22
--account=training2202 --time=00:02:00 ./a.out
```

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)
  CHARACTER(len=*) NAME
  INTEGER NCID, DIMID
  INTEGER LEN
```

- `name` represents the name of the dimension
- `len` represents the value
  - `(NC/NF90)_UNLIMITED` will create an unlimited dimension
- Can only be called in *definition mode*

Python

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

Defining variables

C

```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)
  CHARACTER(len=*) :: NAME
  INTEGER NCID, XTYPE, VARID
  INTEGER, dimension(:) :: DIMIDS
```

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

Python

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

Defining attributes

C

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

Fortran

```
INTEGER NF90_PUT_ATT(NCID, VARID, NAME, ATTR)
  <type> ATTR
  CHARACTER(len=*) :: NAME
  INTEGER NCID, VARID
```

- Puts the attribute `attr` into the data set
- `varid` is the id annotated variable, or `(NC/NF90)_GLOBAL`, if it is a global attribute
- `xtype` specifies the external type of this attribute
- Can only be called in *definition mode*

Python

```
element.attribute_name = value
```

Reading attributes

C

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

Fortran

```
INTEGER NF90_GET_ATT(NCID, VARID, NAME, ATTR)
  <type> ATTR
  CHARACTER(len=*) :: NAME
  INTEGER NCID, VARID
```

- Reads the attribute `attr` from the data set
- `varid` is the id annotated variable, or `(NC/NF90)_GLOBAL`, if it is a global attribute

Python

```
value = element.attribute_name
```

Closing define mode

C
```
int nc_enddef(int ncid)
```

Fortran
```
INTEGER NF90_ENDDEF(NCID)
  INTEGER NCID
```

- Ends the definition phase, and switches to independent data mode
- 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
```

Switching data modes

C

```
int nc_var_par_access(int ncid, int varid, int mode)
        [NC_INDEPENDENT, NC_COLLECTIVE])
```

Fortran

```
INTEGER NF90_VAR_PAR_ACCESS(NCID, VARID, MODE)
  INTEGER NCID, VARID, MODE
```

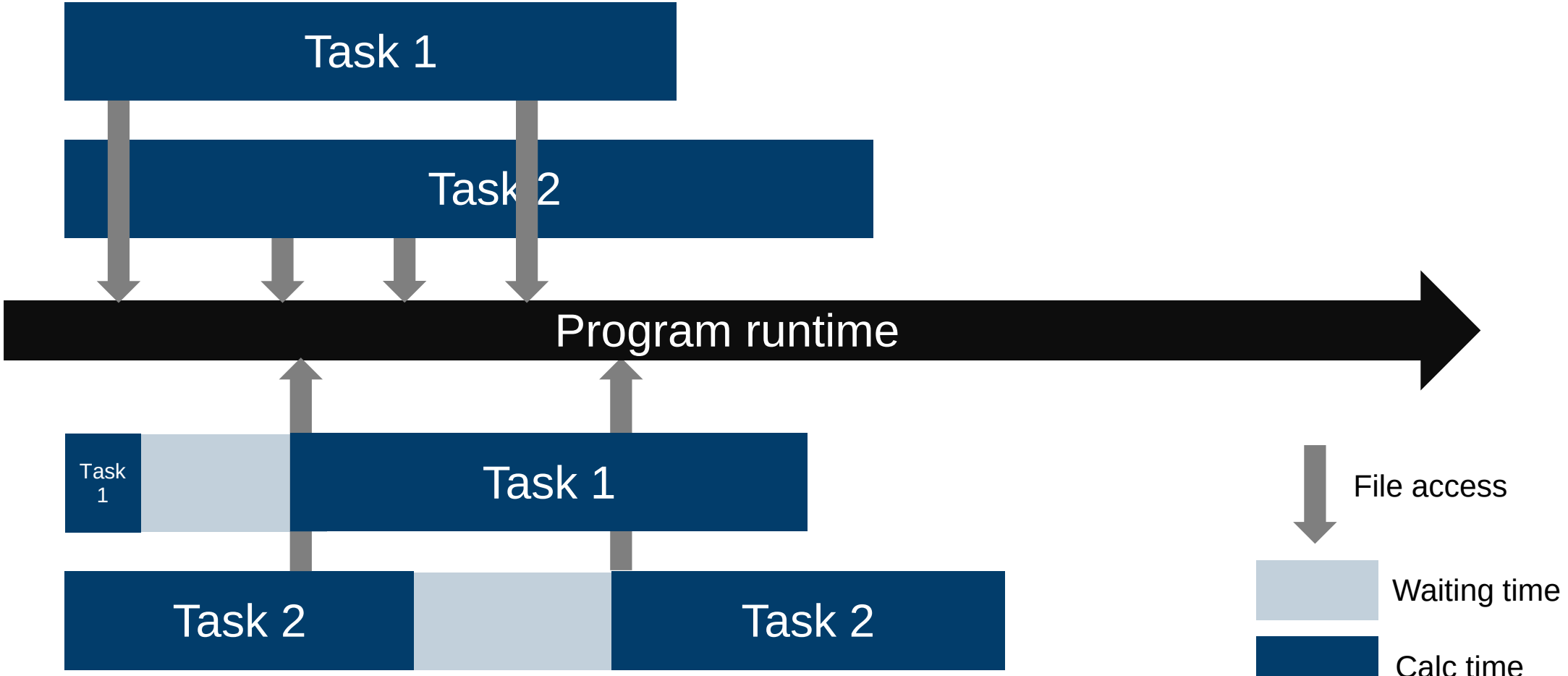
- Switches variable to collective or individual data mode
- `mode` can be `(NC/NF90)_INDEPENDENT` or `(NC/NF90)_COLLECTIVE`

Python

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

Independet vs. collective reading/writing

Independent file access

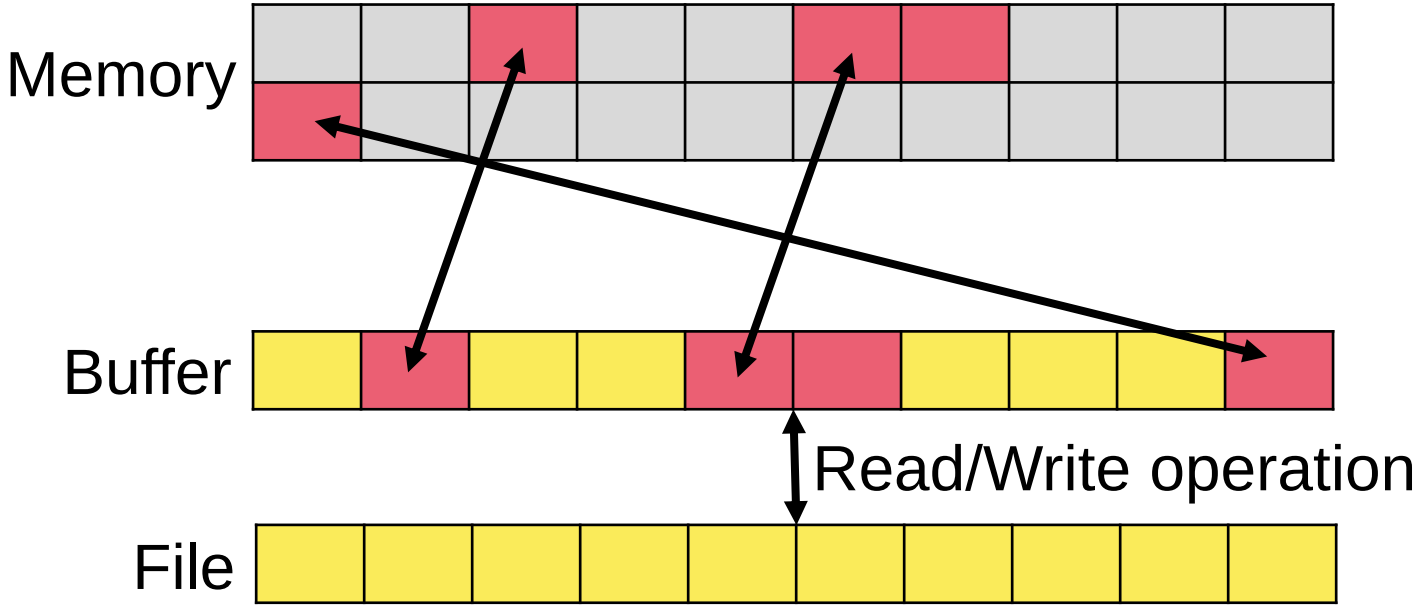


Collective file access

Benefits of collective file access

Data sieving

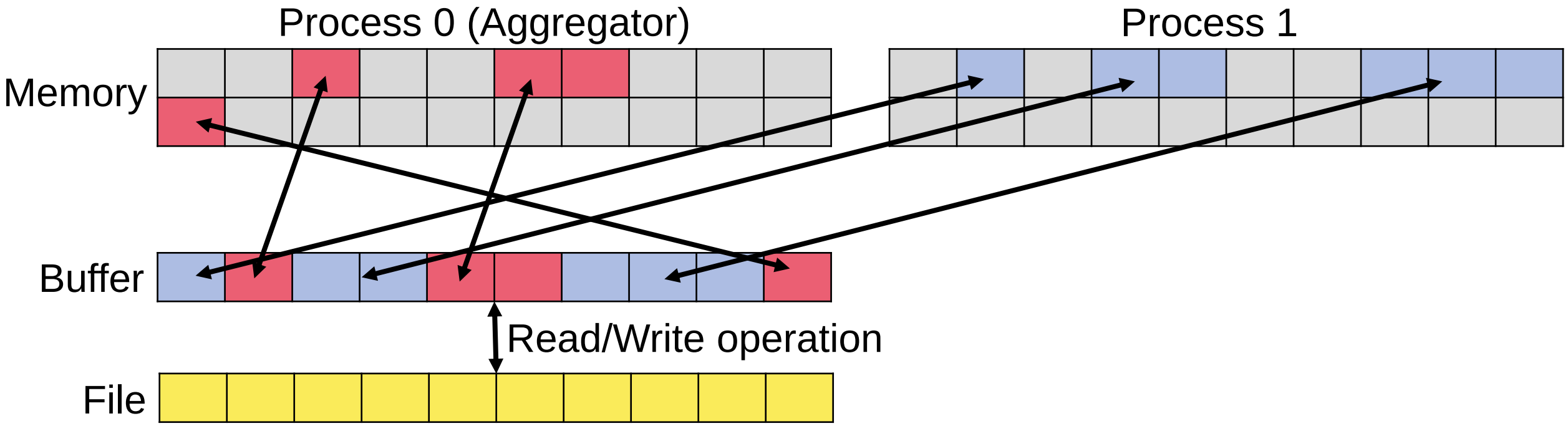
- Usage of buffers to group non-contiguous data requests



Collective buffering

Benefits of collective file access

- Aggregate data before writing/reading



Writing variables

C

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

Fortran

```
INTEGER NF90_PUT_VAR(NCID, VARID, VAR, START, COUNT,
                     STRIDE, IMAP)
  <type>(*) VAR
  INTEGER NCID, VARID
  INTEGER, DIMENSION(:), OPTIONAL :: START, COUNT,
                                     STRIDE, IMAP
```

- Writes a *slab* of data to the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`

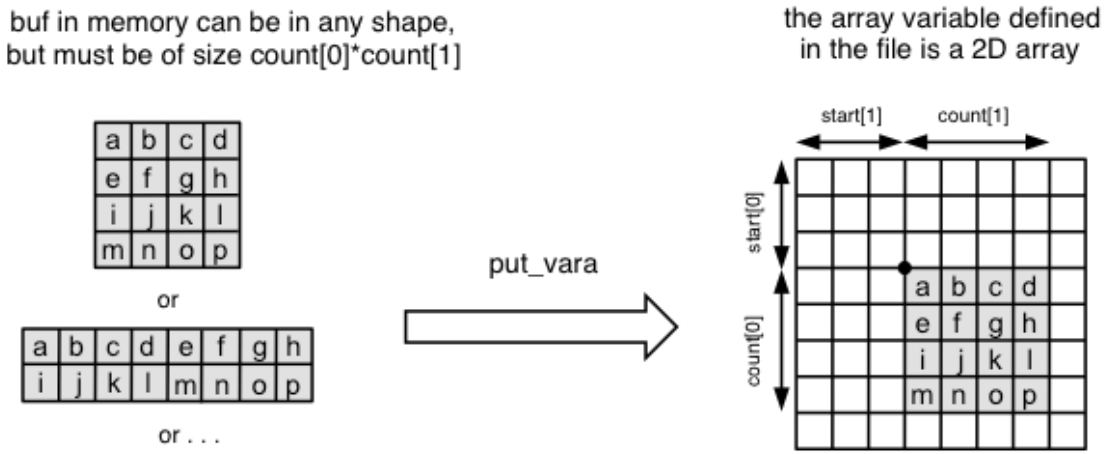
Python

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

Writing variables

C

```
int nc_put_vara_<type>(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

C

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

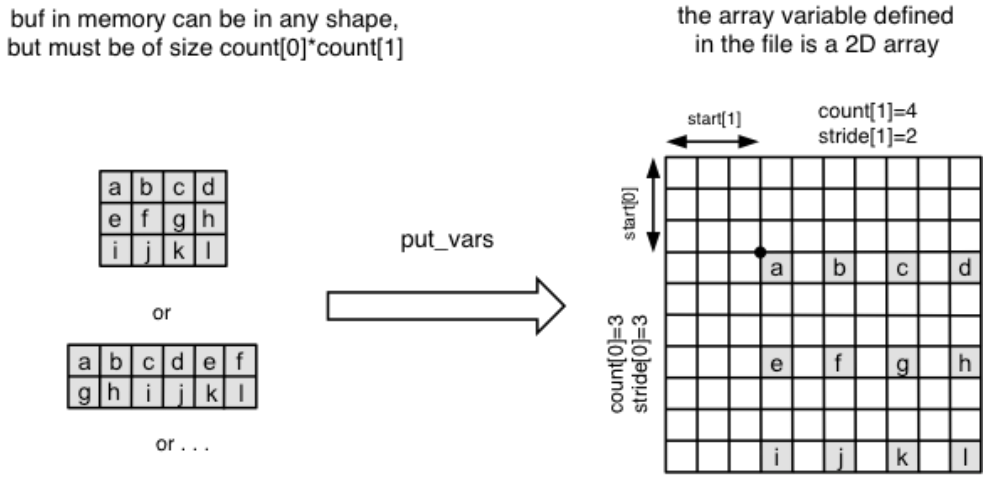
Fortran

```
INTEGER NF90_GET_VAR(NCID, VARID, VAR,
                     START, COUNT, STRIDE, MAP)
  <type>(*) VAR
  INTEGER NCID, VARID
  INTEGER, DIMENSION(:), OPTIONAL :: START, COUNT,
                                     STRIDE, MAP
```

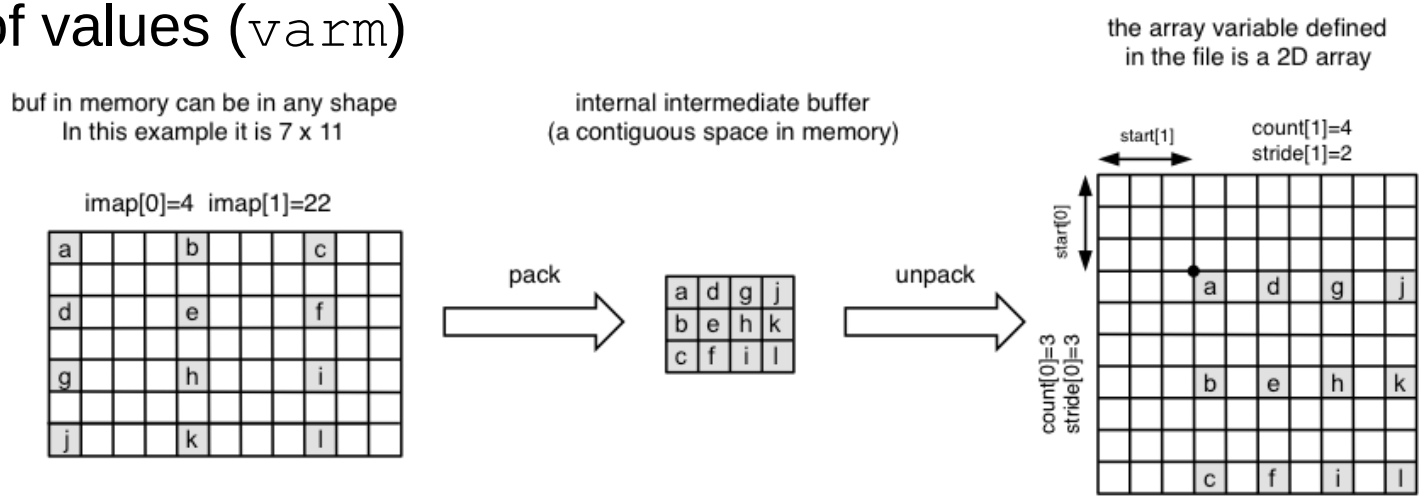
- Reads a *slab* of data of the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`

Additional access types

- 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/

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
- 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 nc_inq_ndims(int ncid,  int* ndims)
```

- Query number of dimensions

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

- Query number of variables

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

- Query number of attributes

Fortran
```
INTEGER NF90_INQUIRE (NCID, NDIMS, NVARS, NATTS,
                      UNLIMITED_DIM_ID)
  INTEGER NCID
  INTEGER, OPTIONAL :: NDIMS, NVARS, NATTS,
                       UNLIMITED_DIM_ID
```

Inquiry of ids of data set entities

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

Fortran
```
INTEGER NF90_INQ_VARID(NCID, NAME, VARID)
  INTEGER NCID, VARID
  CHARACTER(len=*) NAME
```

- Query id of variable given by `name`

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

Fortran
```
INTEGER NF90_INQUIRE_VARIABLE(NCID, VARID, NAME,
                              XTYPE, NDIMS, DIMIDS,
                              NATTS)
  INTEGER NCID, VARID
  INTEGER(len=*), OPTIONAL :: NAME
  INTEGER, OPTIONAL :: XTYPE, NDIMS, NATTS
  INTEGER, DIMENSION(:), OPTIONAL :: DIMIDS
```

- Query `dimids` for given `varid`

Inquiry of dimension lens

C

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

Fortran

```
INTEGER NF90_INQUIRE_DIMENSON(NCID, DIMID, NAME,
                              LEN)
  INTEGER NCID, DIMID
  CHARACTER(len=*), OPTIONAL :: NAME
  INTEGER, OPTIONAL :: LEN
```

- 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:
  The resulting global vector in the file will look like:
  `0000…1111…2222…`
- Write should be collective

Check the resulting file using:
`ncdump`

Performance hints

Chunking

C

```
int nc_def_var_chunking(ncid, varid,mode
   [NC_CONTIGUOUS, NC_CHUNKED], size_t *chunksize)
```

Fortran

```
INTEGER NF90_DEF_VAR_CHUNKING(NCID, VARID, MODE,
                              CHUNKSIZE)
  INTEGER NCID, VARID, MODE
  INTEGER, dimension(:) :: CHUNKSIZE
```

- 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

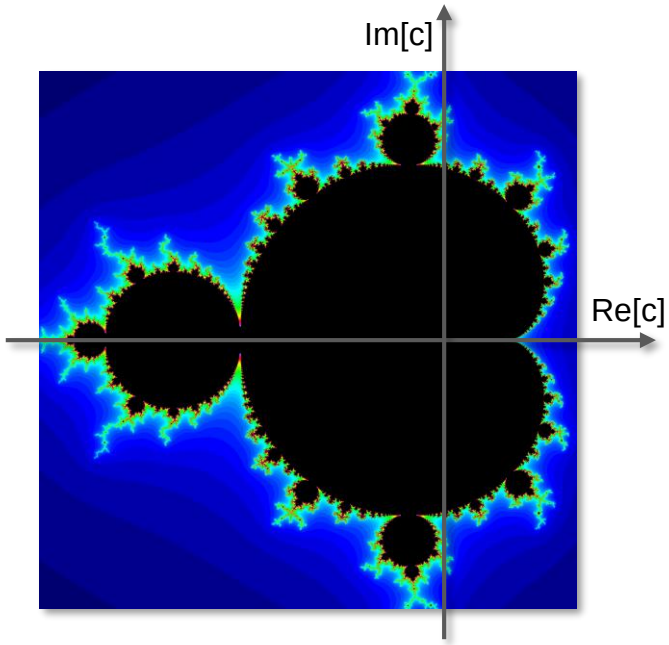
Exercise

Mandelbrot set

Set of all complex numbers c in the complex plane for which

$$z_{n+1} = {z_n}^2 + c$$
$$z_0 = 0$$

does not approach infinity



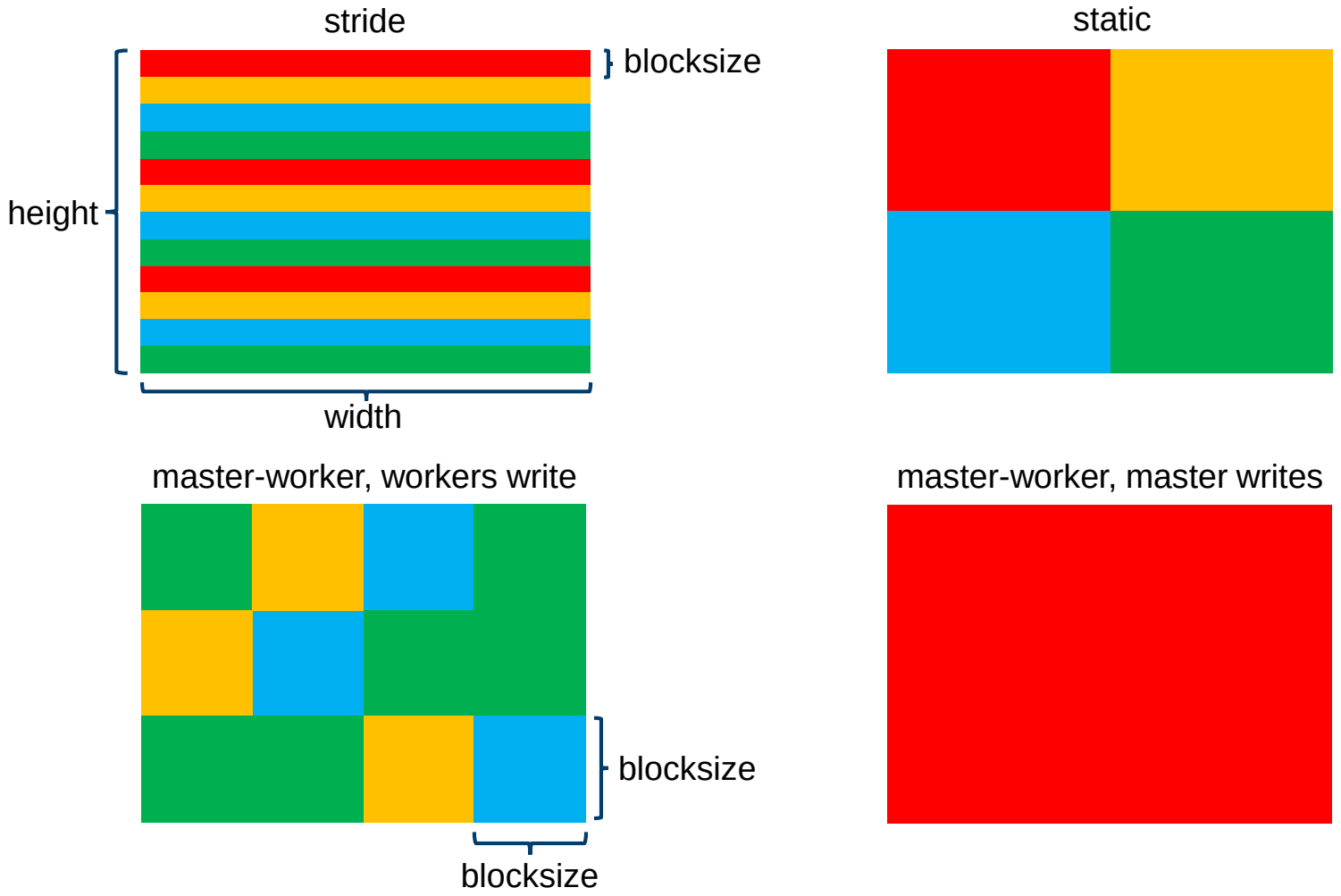
Exercise

Mandelbrot set

- I/O comparison example
- Four different decomposition types
  - stride
  - static
  - master-worker (workers write)
  - master-worker (master writes)
- Five different output formats
  - SIONlib
  - HDF5
  - MPI-IO
  - parallel-netcdf
  - netcdf4

Exercise

Decomposition types



mandelmpi

Command line options

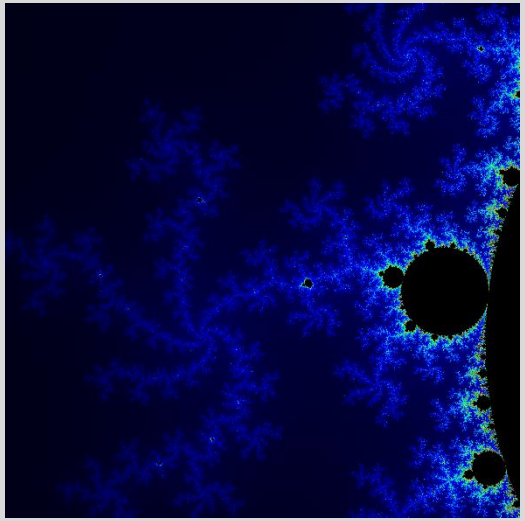
`-v`	use verbose mode
`-t`	decomposition type (`0`: stride, `1`: static, `2`: master-worker worker write, `3`: master-worker master write), default: `0`
`-w`	width, default: `256`
`-h`	height, default: `256`
`-b`	blocksize (not used for type = `1`), default: `64`
`-p`	number of procs in x-direction (only used for type = `1`)
`-q`	number of procs in y-direction (only used for type = `1`)
`-x`	coordinates of initial area: `x1 x2 y1 y2`, default: `-1.5 0.5 -1.0 1.0`
`-i`	max. iterations, default `256`
`-f`	output type (`0`: SIONlib, `1`: HDF5, `2`: MPI-IO, `3`: pnetcdf, `4`: netcdf4), default: `0`

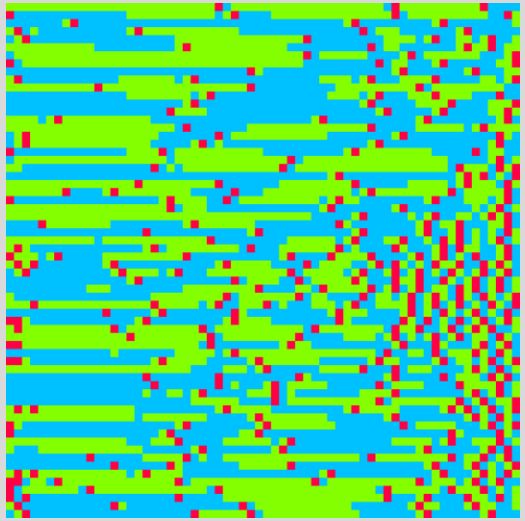
mandelseq

Command line options

`-f` output type (`0`: SIONlib, `1`: HDF5, `2`: MPI-IO, `3`: pnetcdf, `4`: netcdf4), default: `0`

Output





process distribution image
only available for SIONlib

Mandelbrot exercise API

C

```c
typedef struct _infostruct
{
  int type; int width; int height;
  int numprocs;
  double xmin; double xmax; double ymin; double ymax;
  int maxiter;
} _infostruct;
```

Fortran

```fortran
type :: t_infostruct
  integer :: type, width, height
  integer :: numprocs
  real :: xmin, xmax, ymin, ymax
  integer :: maxiter
end type t_infostruct
```

Mandelbrot exercise API

C

```
void open_<lib>(<type> *fid, _infostruct *infostruct,
                int *blocksize, int *start, int rank)
```

Fortran

```
open_<lib>(fid, info, blocksize, start, rank)
  <type>, intent(out) :: fid
  type(t_infostruct), intent(in) :: info
  integer, dimension(2), intent(in) :: blocksize
  integer, dimension(2), intent(in) :: start
  integer, intent(in) :: rank
```

`fid`	lib specific file_id (can occurs twice if multiple ids needed)
`infostruct`	global information structure
`blocksize`	chosen (or calculated) blocksizes (C: [y,x], Fortran: [x,y])
`start`	calculated start point (C: [y,x], Fortran: [x,y], starting at `0`)
`rank`	process MPI rank

Mandelbrot exercise API

C

```
void close_<lib>(<type> *fid, _infostruct *infostruct,
                 int rank)
```

Fortran

```
close_<lib>(fid, info, rank)
  <type>, intent(inout) :: fid
  type(t_infostruct), intent(in) :: info
  integer, intent(in) :: rank
```

`fid`	lib specific file_id (can occurs twice if multiple ids needed)
`infostruct`	global information structure
`rank`	process MPI rank

Mandelbrot exercise API

C

```
void write_to_<lib>_file(
  <type> *fid, _infostruct *infostruct, int *iterations,
  int width, int height, int xpos, int ypos)
```

Fortran

```
write_to_<lib>_file(fid, info, iterations, width, height,
                    xpos, ypos)
  <type>, intent(in) :: fid
  type(t_infostruct), intent(in) :: info
  integer, dimension(:), intent(in) :: iterations
  integer, intent(in) :: width
  integer, intent(in) :: height
  integer, intent(in) :: xpos
  integer, intent(in) :: ypos
```

`iterations`	data array
`width, height`	size of current data block (pixel coordinates)
`xpos, ypos`	position of current data block (pixel coordinates starting at `0`)

Exercise

Exercise 4 – Mandelbrot set

- Directory preparation:
  - `/p/project/training2202/mandel_c/copy.sh`
  - `/p/project/training2202/mandel_fortran/copy.sh`
  - `cd /p/scratch/training2202/<username>/mandel_c`
  - `cd /p/scratch/training2202/<username>/mandel_fortran`
- Run the example or implement your own solution for the stride decomposition (`type = 0`) of the Mandelbrot example (`mandelnetcdf4.c` or `mandelnetcdf4.f90`)
- To implement your own solution copy the `workshop/mandelnetcdf4.c` or `workshop/mandelnetcdf4.f90` template file into the main directory
- Try to use a collective write call
- You will need the following NetCDF4 routines
  - `nc_create_par / nf90_create_par`
  - `nc_def_dim / nf90_def_dim`
  - `nc_put_att_double / nf90_put_att`
  - `nc_def_var / nf90_def_var`
  - `nc_put_vara_int / nf90_put_var`
  - `nc_close / nf90_close`

Hints

Options for running the program:
`-t 0` (stride decomposition)
`-f 4` (use netCDF4)