VIENNA SCIENTIFIC CLUSTER

JÜLICH FORSCHUNGSZENTRUM

Parallel I/O and Portable Data Formats

VSC Training Course

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

Vienna, Austria, 6th December 2017

Overview

- **I/O** can be the main **bottleneck of an HPC application**
- In most cases the possible I/O bandwidth **scales with the number of allocated compute cores**
  - Application developer has to take care to use the available bandwidth efficiently
- Designing a good data format and I/O strategy can improve the complete data creation and workflow process

JUST - Juelich Storage Cluster

Timetable

- 09:00 - 09:30 Parallel I/O strategies
- 09:30 - 10:30 Parallel NetCDF (PnetCDF/NetCDF4)
- 10:30 - 11:00 Coffee break
- 11:00 - 12:30 Parallel NetCDF (PnetCDF/NetCDF4)
- 12:30 - 13:30 Lunch break
- 13:30 - 14:30 HDF5
- 14:30 - 15:00 Coffee break
- 15:00 - 16:30 HDF5

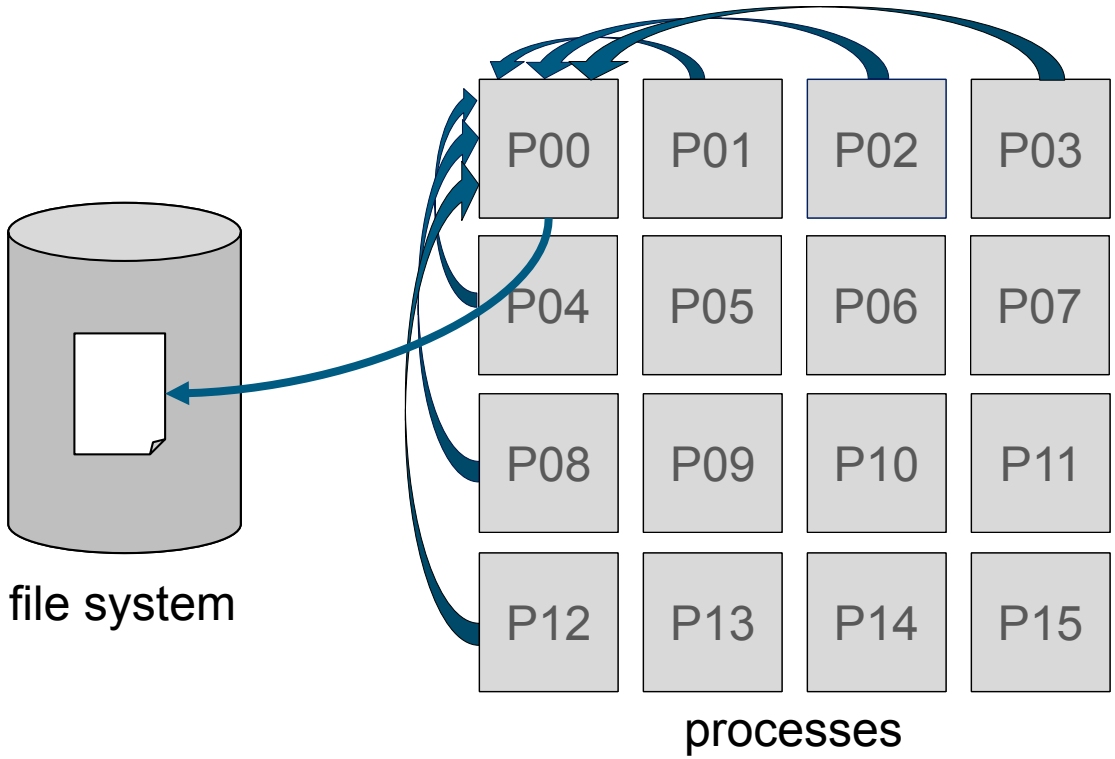
VIENNA SCIENTIFIC CLUSTER

JÜLICH FORSCHUNGSZENTRUM

Parallel I/O Strategies

Vienna, Austria, 6th December 2017

One process performs I/O



One process performs I/O

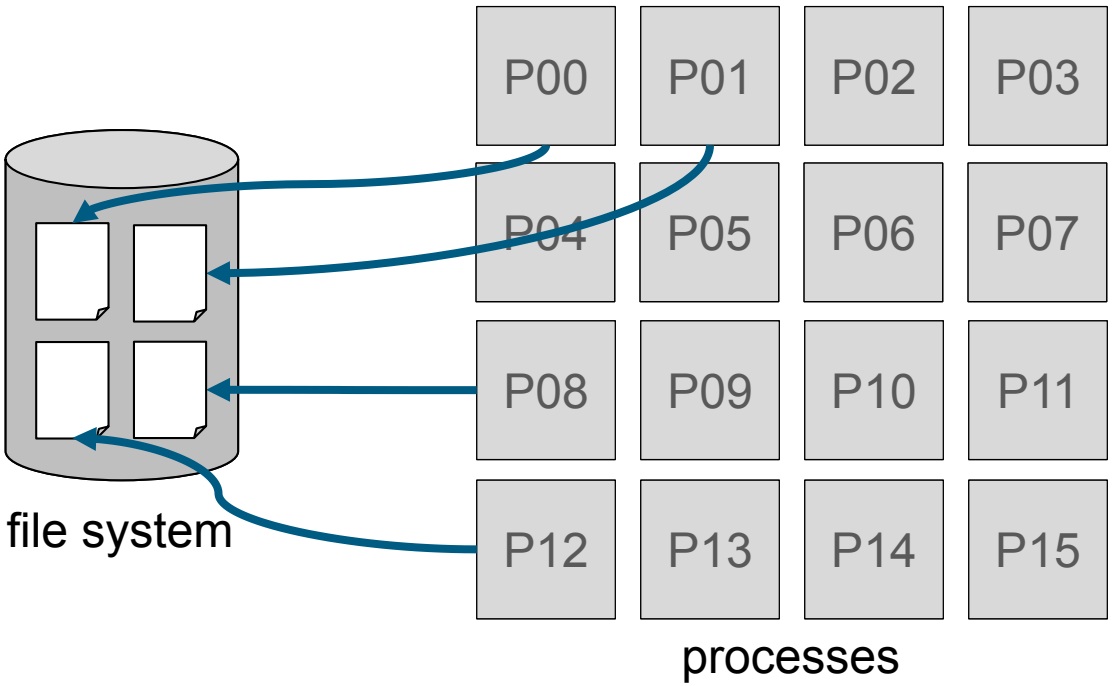
+ Simple to implement

- I/O bandwidth is limited to the rate of this single process
- Additional communication might be necessary
- Other processes may idle and waste computing resources during I/O time

Frequent flushing on small blocks

- Modern file systems in HPC have **large file system blocks** (e.g. 4MB)
- A flush on a file handle forces the file system to perform all pending write operations
- If application writes in small data blocks, the same file system block it has to be **read and written multiple times**
- Performance degradation due to the inability to combine several write calls

Task-local files



Task-local files

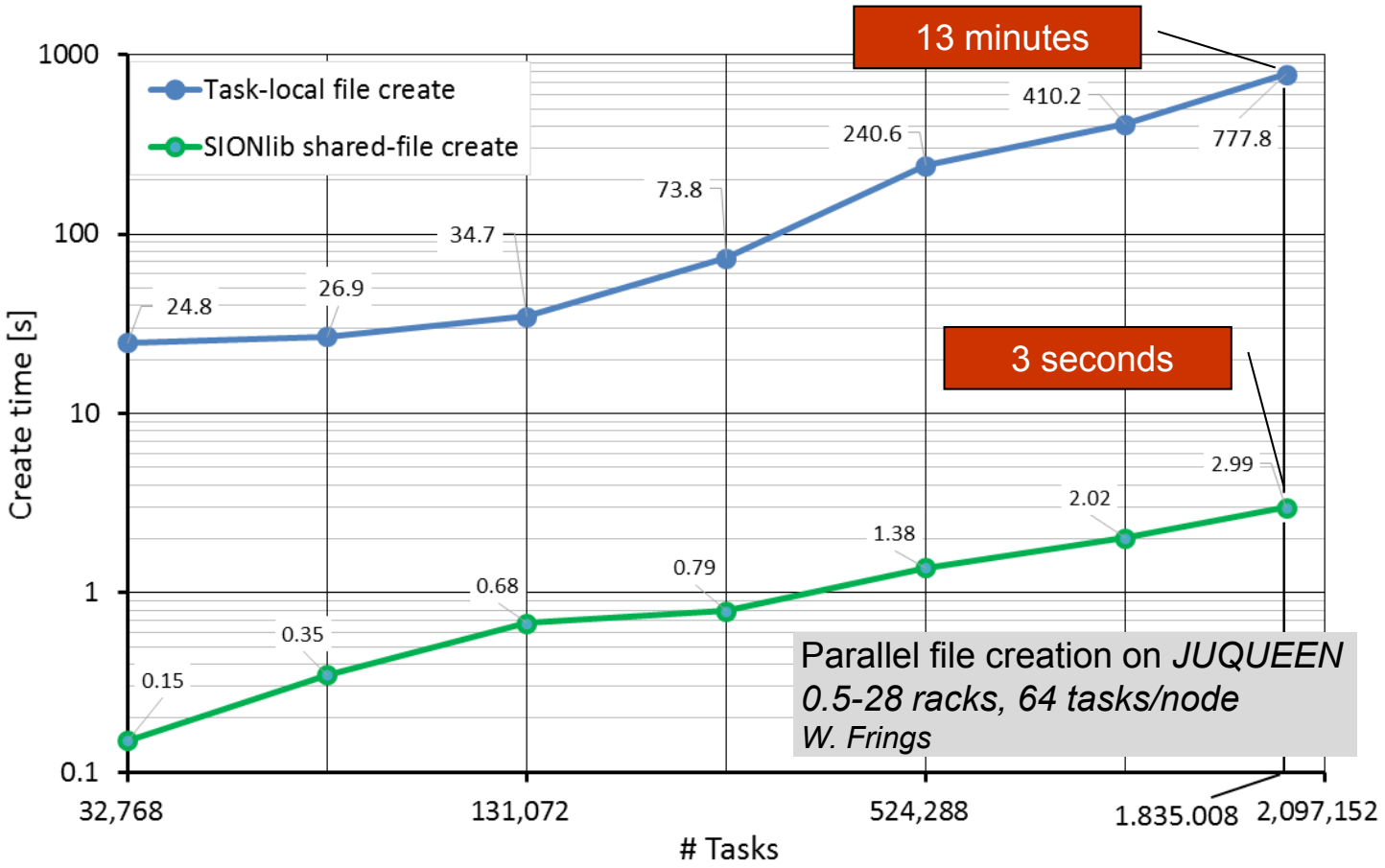
+ Simple to implement
+ No coordination between processes needed
+ No false sharing of file system blocks

- Number of files quickly becomes unmanageable
- Files often need to be merged to create a canonical dataset
- File system might serialize meta data modification

Serialization of meta data modification

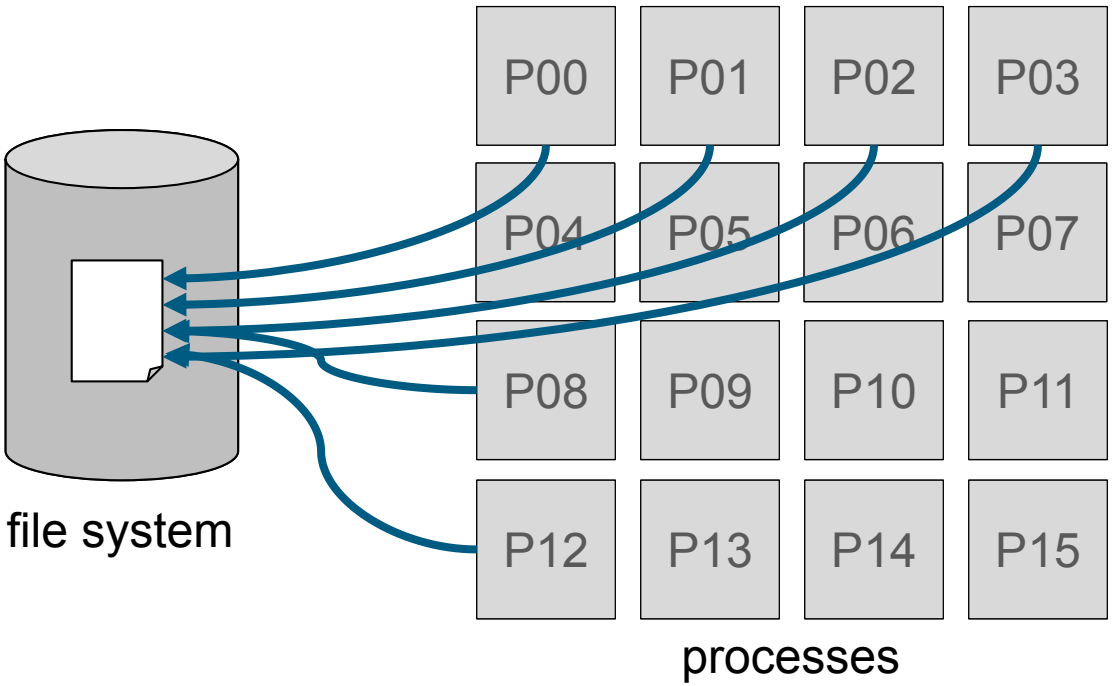
Pitfall 2

Example: Creating files in parallel in the same directory



The creation of 2.097.152 files costs 113.595 core hours on JUQUEEN!

Shared files

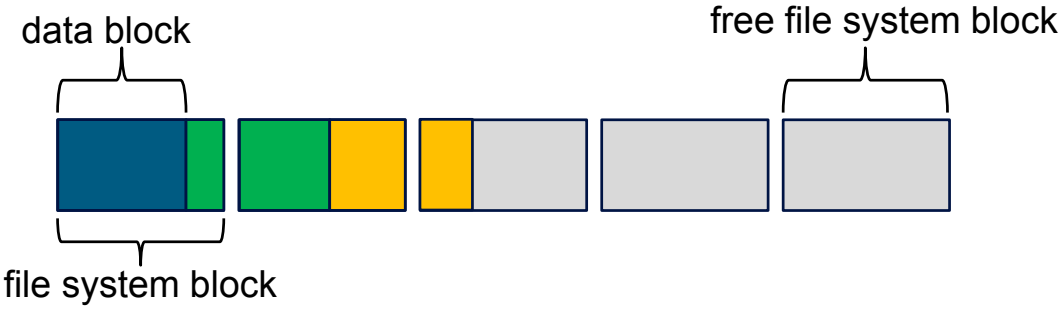


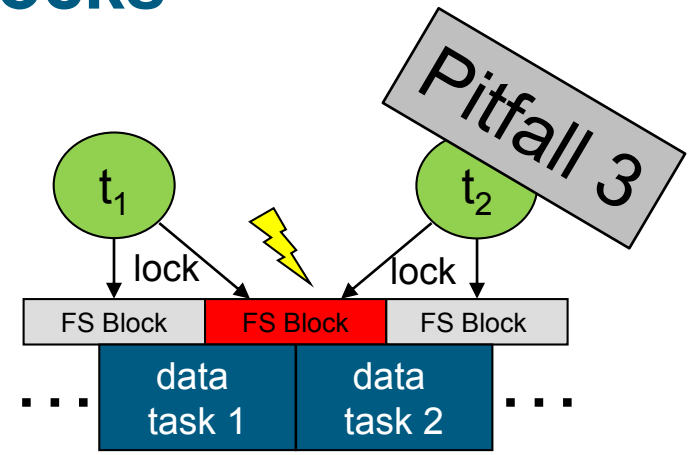
Shared files

+ Number of files is independent of number of processes
+ File can be in canonical representation (no post-processing)

- Uncoordinated client requests might induce time penalties
- File layout may induce false sharing of file system blocks

False sharing of file system blocks



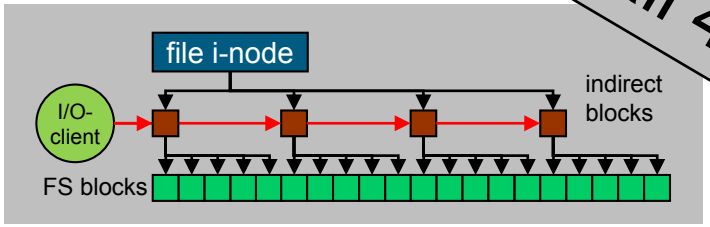


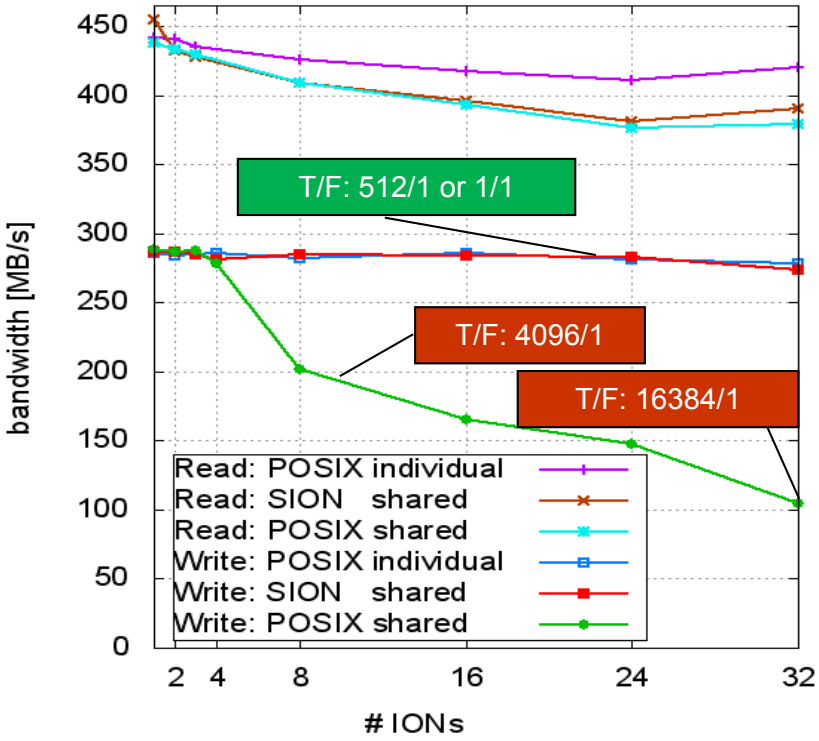
- Data blocks of individual processes **do not fill up a complete file system block**
- Several processes **share a file system block**
- Exclusive access (e.g. write) must be **serialized**
- The more processes have to synchronize the more waiting time will propagate

Number of Tasks per Shared File

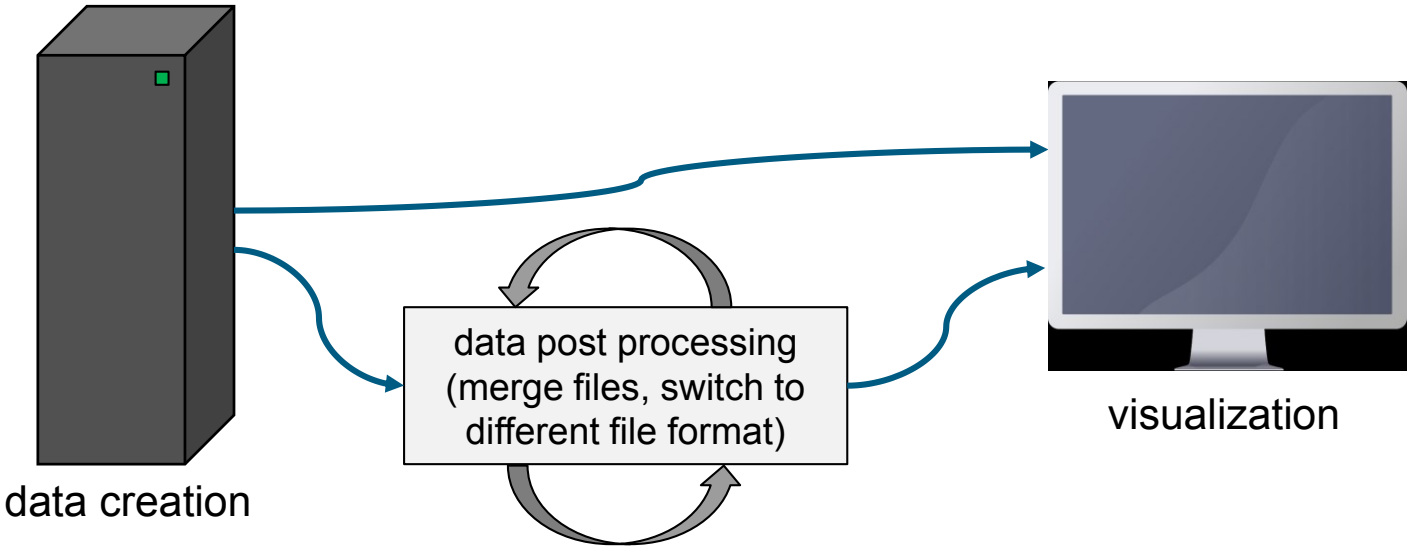
Pitfall 4

- Meta-data wall on file level
  - File meta-data management
  - Locking
- Example Blue Gene/P
  - Jugene (72 racks)
  - I/O forwarding nodes (ION)
  - GPFS client on ION
  - One file per ION





I/O Workflow

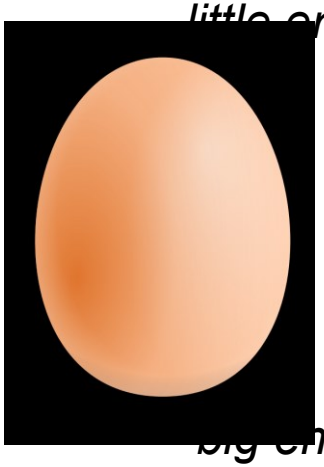


- Post processing can be very time-consuming (> data creation)
  - Widely used portable data formats avoid post processing
- Data transportation time can be long:
  - Use shared file system for file access, avoid raw data transport
  - Avoid renaming/moving of big files (can block backup)

Portability

Pitfall 5

- Endianness (byte order) of binary data



2,712,847,316

=

10100001 10110010 11000011 11010100

Address	Little Endian	Big Endian
1000	11010100	10100001
1001	11000011	10110010
1002	10110010	11000011
1003	10100001	11010100

- Conversion of files might be necessary and expensive

Portability

Pitfall 6

- Memory order depends on programming language

1	2	3
4	5	6
7	8	9

→

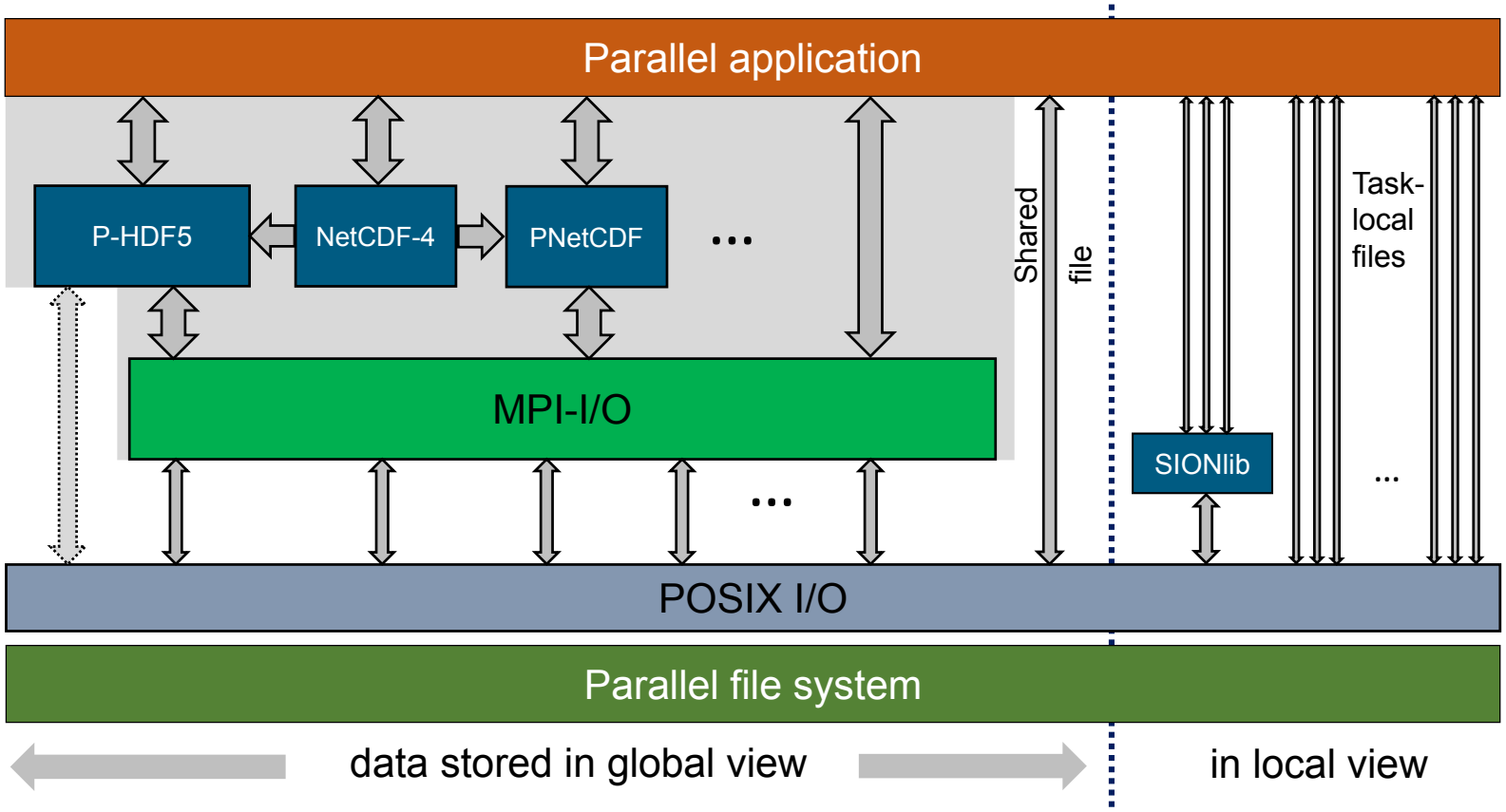
Address	row-major order (e.g. C/C++)	column-major order (e.g. Fortran)
1000	1	1
1001	2	4
1002	3	7
1003	4	2
1004	5	5
…	…	…

- Transpose of array might be necessary when using different programming languages in the same workflow
- Solution: Choosing a portable data format (HDF5, NetCDF)

How to choose the I/O strategy?

- Performance considerations
  - Amount of data
  - Frequency of reading/writing
  - Scalability
- Portability
  - Different HPC architectures
  - Data exchange with others
  - Long-term storage
- E.g. use two formats and converters:
  - Internal: Write/read data “as-is”
    → Restart/checkpoint files
  - External: Write/read data in non-decomposed format
    (portable, system-independent, self-describing)
    → Workflows, Pre-, Post-processing, Data exchange

Parallel I/O Software Stack



VIENNA SCIENTIFIC CLUSTER

JÜLICH FORSCHUNGSZENTRUM

Parallel NetCDF

PnetCDF and NetCDF4

Vienna, Austria, 6th December 2017

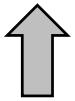
Outline

- 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.

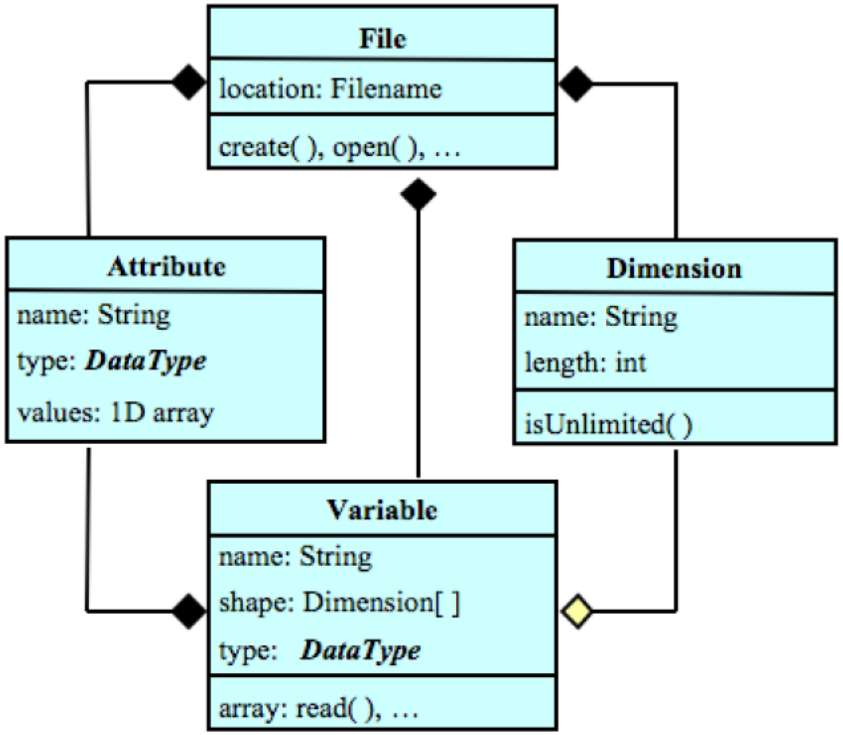
Variable

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



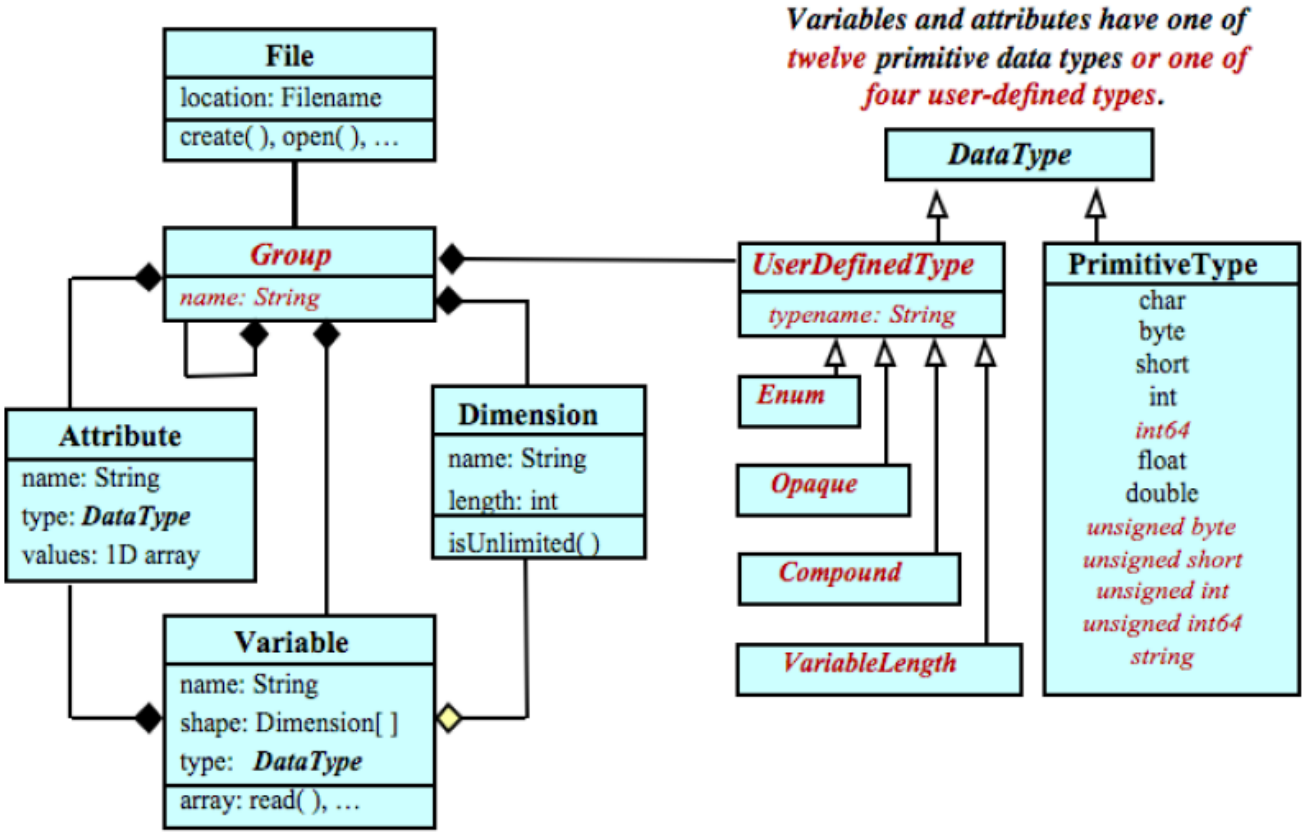
Variables and attributes have one of six primitive data types.

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.

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

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

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

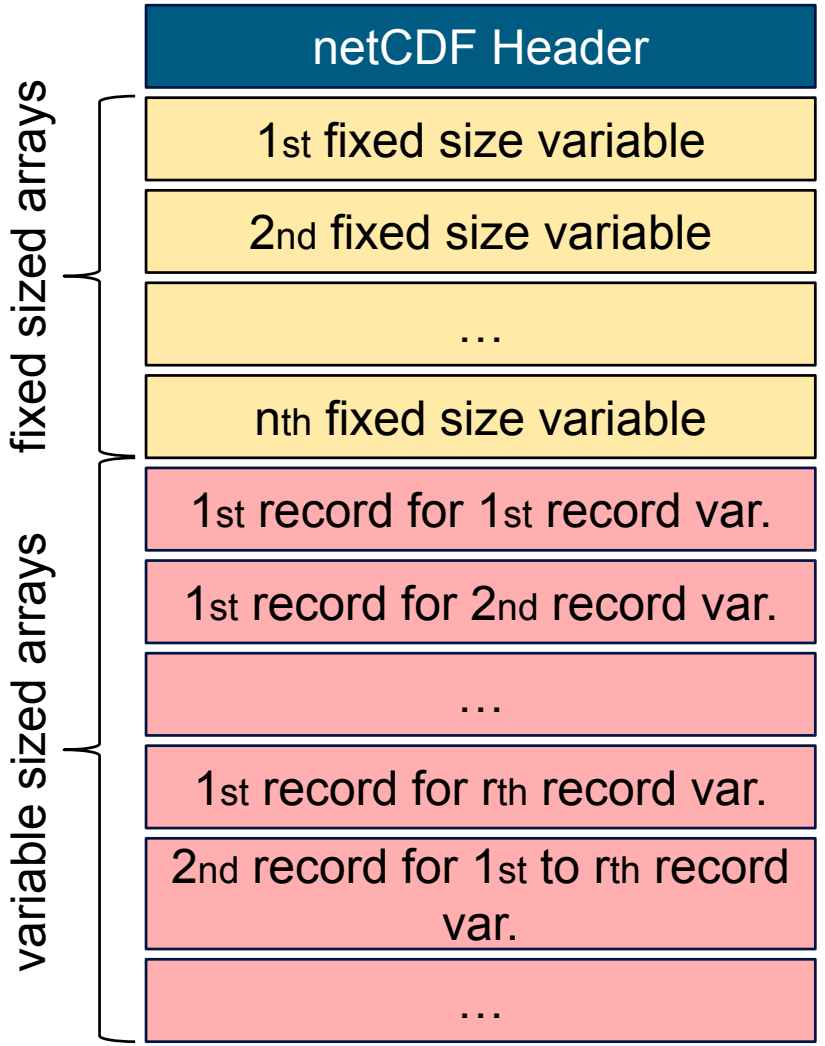
The classic NetCDF file format

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

`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/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)
- You can use the provided template:
  `helloworld_[p]netcdf_template.c` or
  `helloworld_[p]netcdf_template.f90`
- Compile, link and execute the application

PnetCDF:

```
module load intel/16 intel-mpi/5 hdf5/1.8.12 pnetcdf/1.5.0

mpiicc helloworld_pnetcdf.c -lpnetcdf
mpiifort helloworld_pnetcdf.f90 -lpnetcdf
```

NetCDF4:

```
module load intel/16 intel-mpi/5 hdf5/1.8.18-MPI netcdf_C/4.4.1.1
module load netcdf_Fortran/4.4.4

mpiicc helloworld_pnetcdf.c -lnetcdf
mpiifort helloworld_pnetcdf.f90 -lnetcdff
```

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

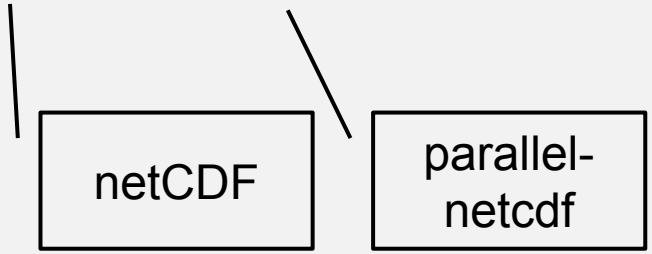
Not explicitly needed for NETCDF4 file format

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` and/or `ncmpidump`



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

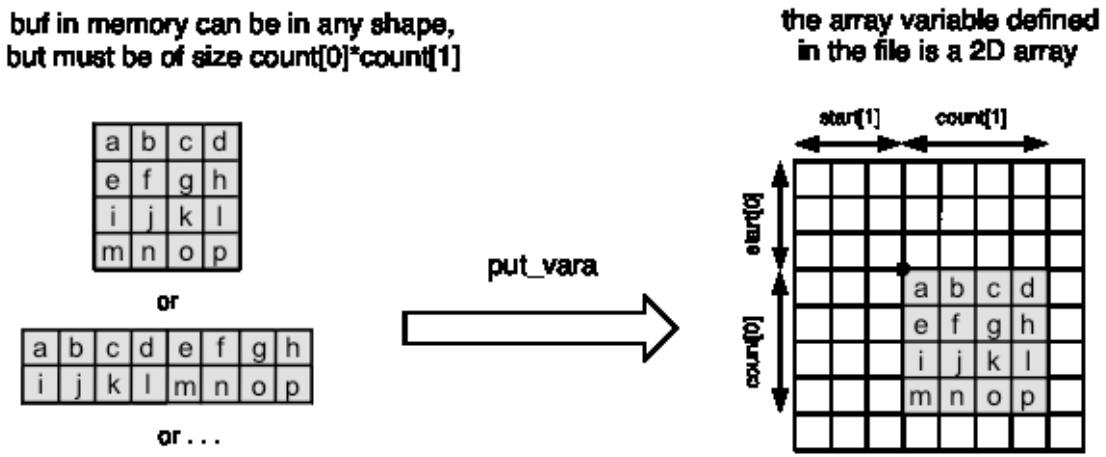
F90:

```
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`

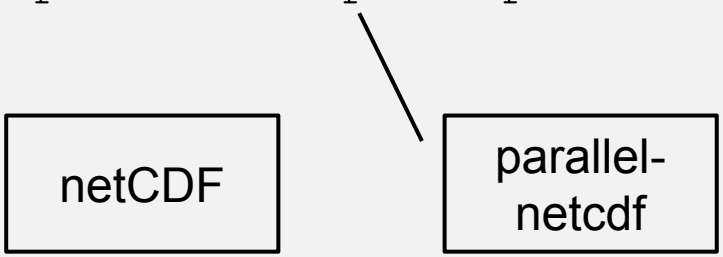
- Reads `len` from a given dimension

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` and or `ncmpidump`



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

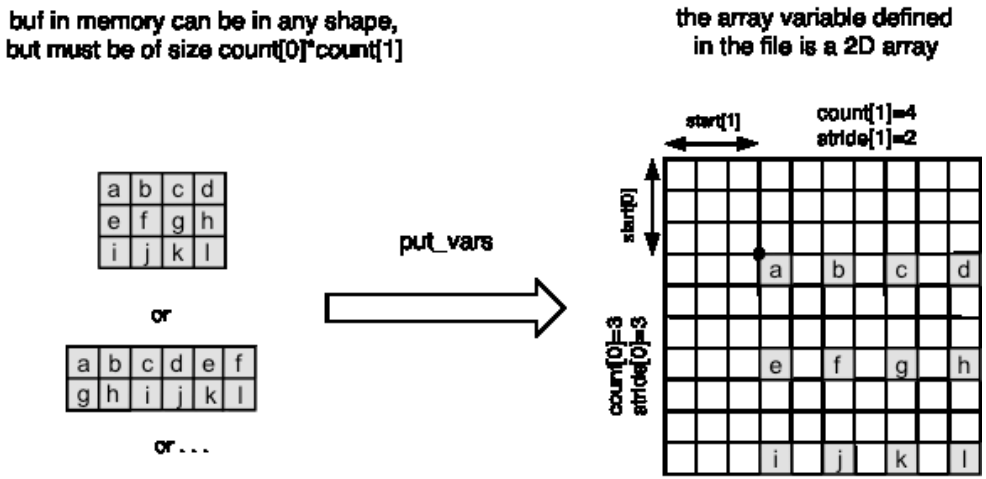
C

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

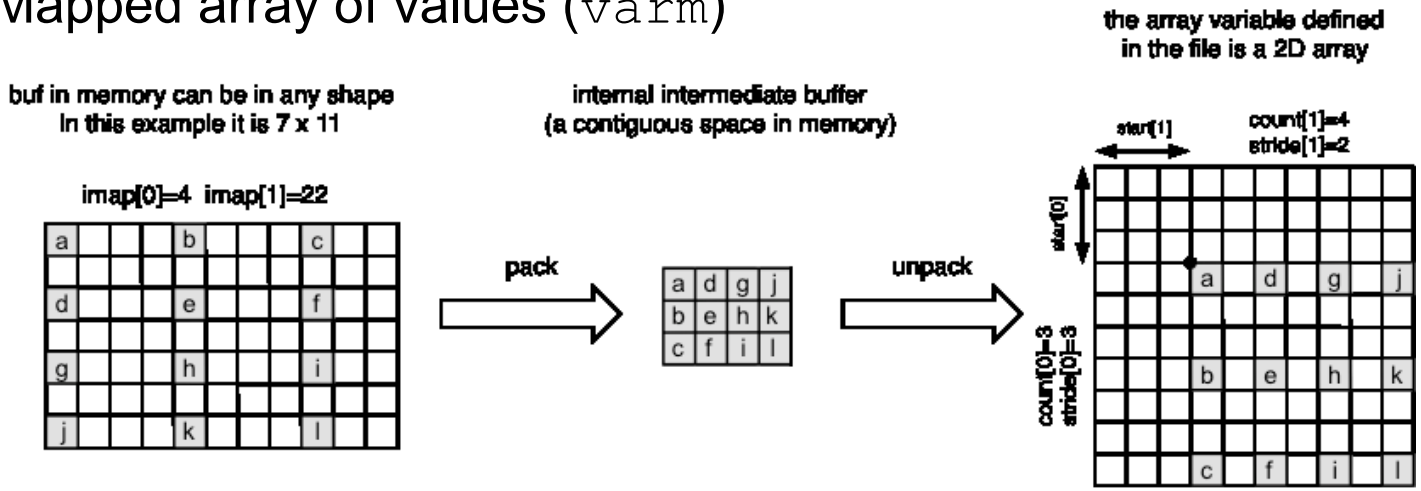
Additional access types

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)
```

VIENNA SCIENTIFIC CLUSTER

JÜLICH FORSCHUNGSZENTRUM

HDF5

Vienna, Austria, 6th December 2017

Outline

- Introduction
  - Structure of the HDF5 library
  - Terms and definitions
- HDF5 - programming model and API
  - Creating/opening HDF5 files
  - Closing HDF5 files and other objects
  - HDF5 predefined datatypes
  - Creating dataspaces
  - Creating datasets
  - Writing/reading data
  - Row major / column major
  - Partial I/O
- Parallel HDF5

What is HDF5?

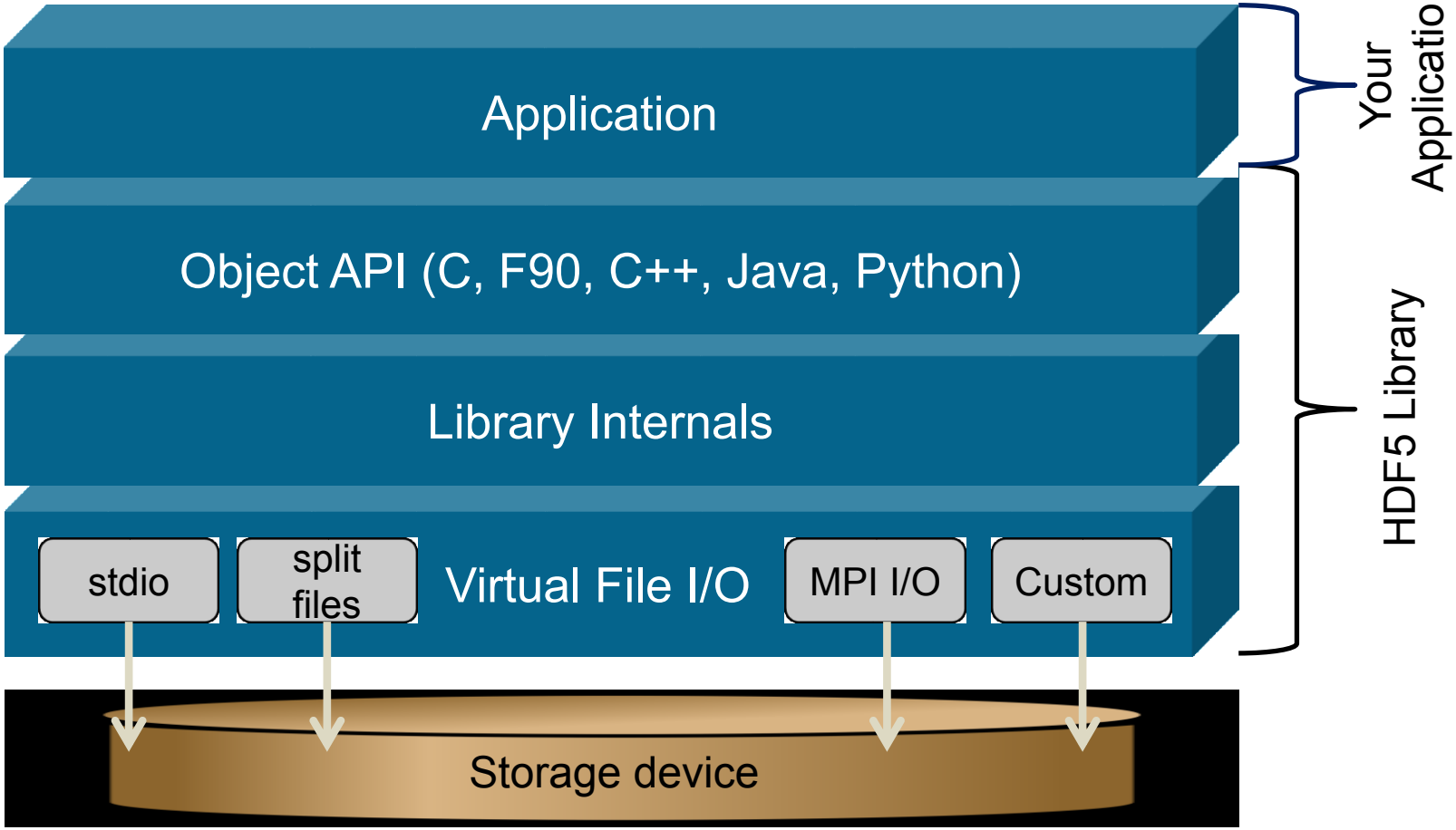
Hierarchical Data Format

- API, data model and file format for I/O management
- Tools suite for accessing data in HDF5 format

HDF5 - Features

- Supports parallel I/O
- Self describing data model which allows the management of complex data sets
- Portable file format
- Available on a variety of platforms
- Supports C, C++, Fortran 90 and Java
  - Pythonic interfaces also available
- Provides tools to operate on HDF5 files and data

Layers of the HDF5 Library



File organization

- HDF5 file structure corresponds in many respects to a Unix/Linux file system (fs)

HDF5		Unix/Linux fs
Group	↔	Directory
Data set	↔	File

/DataSet1

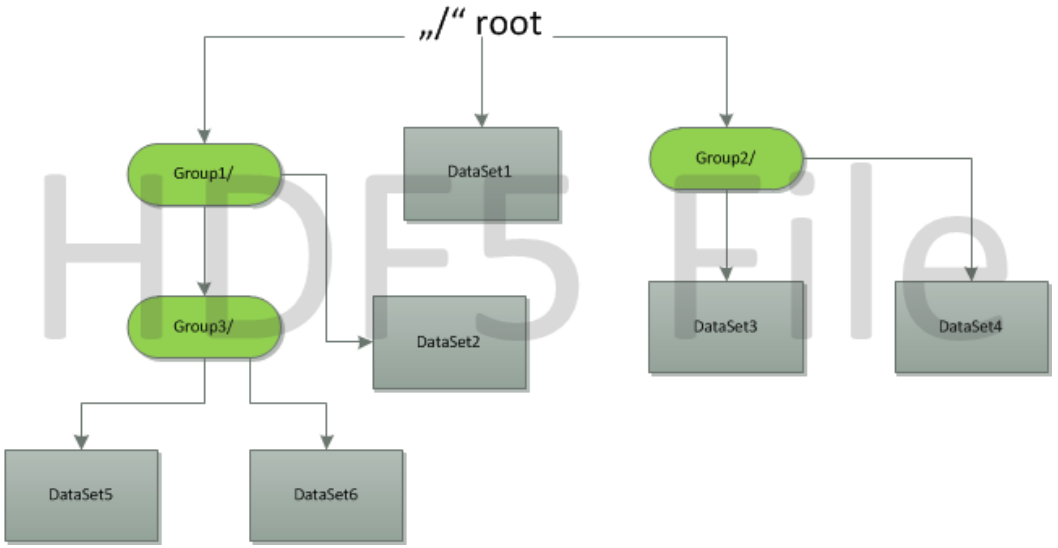
/Group1/DataSet2

/Group1/Group3/DataSet5

/Group1/Group3/DataSet6

/Group2/DataSet3

/Group2/Dataset4



Terminology

File

Container for storing data

Group

Structure which may contain HDF5 objects, e.g. datasets, attributes, datasets

Attribute

Can be used to describe datasets and is attached to them

Dataspace

Describes the dimensionality of the data array and the shape of the data points respectively, i.e. it describes the shape of a dataset

Dataset

Multi-dimensional array of data elements

Library specific types

C

```
#include hdf5.h
hid_t      Object identifier
herr_t     Function return value
hsize_t    Used for dimensions
hssize_t   Used for coordinates and dimensions
hvl_t      Variable length datatype
```

Fortran

```
use hdf5
INTEGER(HID_T)      Object identifier
INTEGER(HSIZE_T)    Used for dimensions
INTEGER(HSSIZE_T)   Used for coordinates and dimensions
```

- Defined types are integers of different size
- Own defined types ensure portability

Fortran HDF5 open

- The HDF5 library interface needs to be initialized (e.g. global variables) by calling `H5OPEN_F` before it can be used in your code and closed (`H5CLOSE_F`) at the end.

Fortran

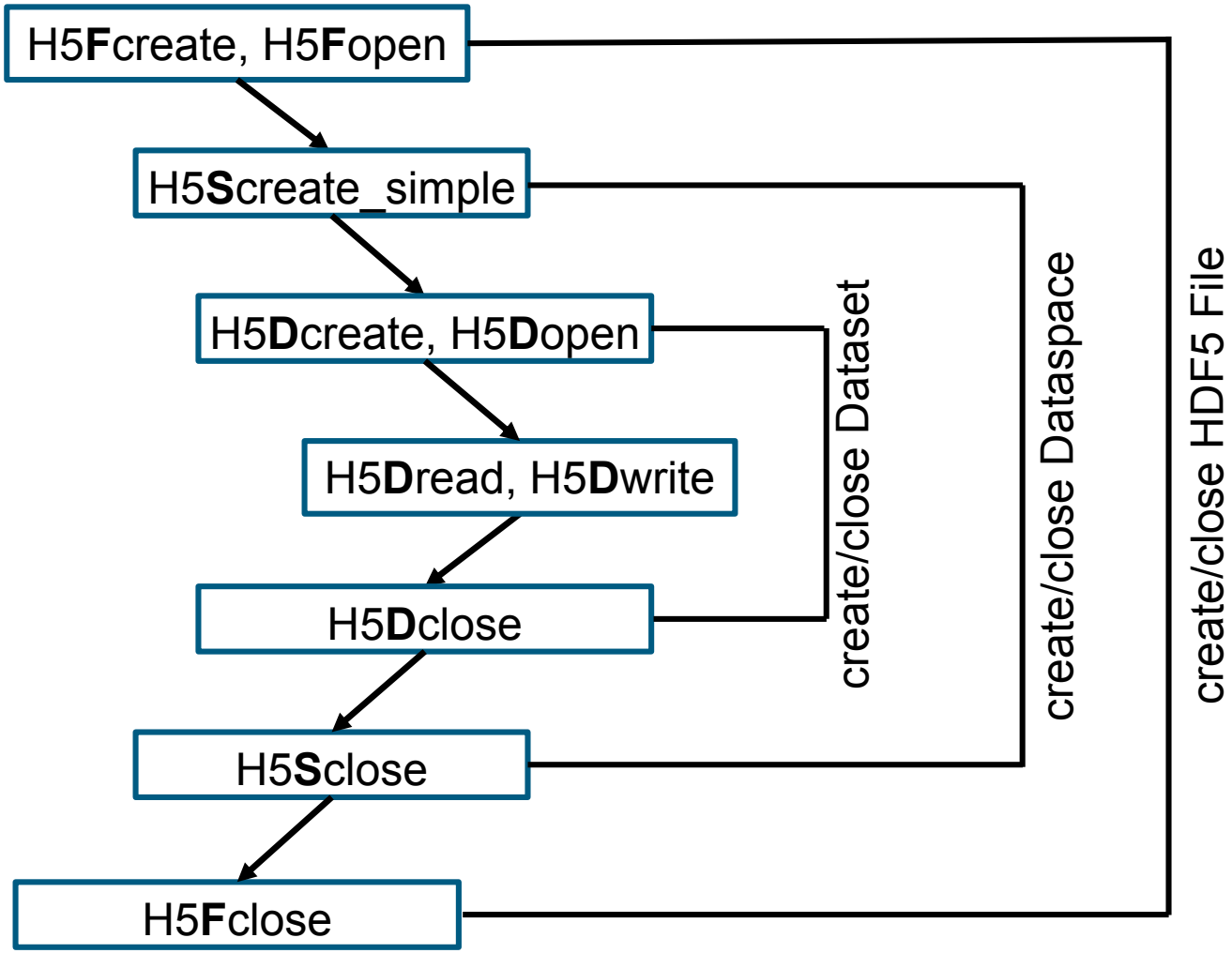
```
H5OPEN_F(STATUS)
  INTEGER, INTENT(OUT) :: STATUS

H5CLOSE_F(STATUS)
  INTEGER, INTENT(OUT) :: STATUS
```

- `status` returns `0` if successful

API naming scheme (excerpt)

- `H5`
  - Library functions: general-purpose functions
- `H5D`
  - Dataset interface: dataset access and manipulation routines
- `H5G`
  - Group interface: group creation and manipulation routines
- `H5F`
  - File interface: file access routines
- `H5P`
  - Property list interface: object property list manipulation routines
- `H5S`
  - Dataspace interface: dataspace definition and access routines

General Procedure



Creating an HDF5 file

C

```
hid_t H5Fcreate(const char *name, unsigned
                access_flag, hid_t creation_prp,
                hid_t access_prp)
```

Fortran

```
H5FCREATE_F(NAME, ACCESS_FLAGS, FILE_ID, HDFERR,
            CREATION_PRP, ACCESS_PRP)
  CHARACTER(*), INTENT(IN) :: NAME
  INTEGER, INTENT(IN) :: ACCESS_FLAGS
  INTEGER(KIND=HID_T), INTENT(OUT) :: FILE_ID
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(KIND=HID_T), OPTIONAL, INTENT(IN) ::
    CREATION_PRP, ACCESS_PRP
```

- `name`: Name of the file
- `access_flags`: File access flags
- `creation_prp` and `access_prp`: File creation and access property list, `H5P_DEFAULT[_F]` if not specified
- Fortran uses `file_id` as return value

Opening an existing HDF5 file

C

```
hid_t H5Fopen(const char *name, unsigned flags,
              hid_t access_prp)
```

Fortran

```
H5FOPEN_F(NAME, FLAGS, FILE_ID, HDFERR,
          ACCESS_PRP)
  CHARACTER(*), INTENT(IN) :: NAME
  INTEGER, INTENT(IN) :: FLAGS
  INTEGER(KIND=HID_T), INTENT(OUT) :: FILE_ID
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(KIND=HID_T), OPTIONAL, INTENT(IN) ::
    ACCESS_PRP
```

- `name`: Name of the file
- `access_prp`: File access property list, `H5P_DEFAULT[_F]` if not specified
- Fortran uses `file_id` as return value
- Avoid multiple opens of the same file

Access modes

- `H5F_ACC_TRUNC[_F]`: Create a new file, overwrite an existing file
- `H5F_ACC_EXCL[_F]`: Create a new file, `H5Fcreate` fails if file already exists
- `H5F_ACC_RDWR[_F]`: Open file in read-write mode, irrelevant for `H5Fcreate[_f]`
- `H5F_ACC_RDONLY[_F]`: Open file in read-only mode, irrelevant for `H5Fcreate[_f]`
- More specific settings are controlled through file creation property list (`creation_prp`) and file access property lists (`access_prp`) which defaults to `H5P_DEFAULT[_F]`
- `creation_prp` controls file metadata
- `access_prp` controls different methods of performing I/O on files

Group creation

C

```
hid_t H5Gcreate(hid_t loc_id, const char *name,
                hid_t lcpl_id, hid_t gcpl_id,
                hid_t gapl_id )
```

Fortran

```
H5GCREATE_F(LOC_ID, NAME, GRP_ID, HDFERR,
            SIZE_HINT, LCPL_ID, GCPL_ID, GAPL_ID)
  INTEGER(KIND=HID_T), INTENT(IN) :: LOC_ID
  CHARACTER(LEN=*), INTENT(IN) :: NAME
  INTEGER(KIND=HID_T), INTENT(OUT) :: GRP_ID
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(KIND=SIZE_T), OPTIONAL, INTENT(IN) ::
    SIZE_HINT
  INTEGER(KIND=HID_T), OPTIONAL, INTENT(IN) ::
    LCPL_ID, GCPL_ID, GAPL_ID
```

- `loc_id`: Can be the `file_id` or another `group_id`
- `name` can be an absolute or relative path
- `lcpl_id, gcpl_id, gapl_id`: Property lists for link/group
- use `H5Gclose[_f]` to finalize group access

Closing an HDF5 file

C

```
herr_t H5Fclose(hid_t file_id)
```

Fortran

```
H5FCLOSE_F(FILE_ID, HDFERR)
  INTEGER(KIND=HID_T), INTENT(IN) :: FILE_ID
  INTEGER, INTENT(OUT) :: HDFERR
```

Exercise

Exercise 4 – HDF5 hello world

- Write a serial program in C or Fortran which creates and closes an HDF5 file
- Create a group “data” inside of this file

Check the resulting file using:
`h5dump`

```
module load intel/16 intel-mpi/5 hdf5/1.8.18-MPI

mpiicc helloworld_hdf5.c -lhdf5

mpiifort helloworld_hdf5.f90 -lhdf5_fortran
```

HDF5 pre-defined datatypes (excerpt)

C	C type	HDF5 file type (pre-defined)	HDF5 memory type (native)
	`int`	`H5T_STD_I32[BE,LE]`	`H5T_NATIVE_INT`
	`float`	`H5T_IEEE_F32[BE,LE]`	`H5T_NATIVE_FLOAT`
	`double`	`H5T_IEEE_F64[BE,LE]`	`H5T_NATIVE_DOUBLE`

Fortran	F type	HDF5 file type (pre-defined)	HDF5 memory type (native)
	`integer`	`H5T_STD_I32[BE,LE]`	`H5T_NATIVE_INTEGER`
	`real`	`H5T_IEEE_F32[BE,LE]`	`H5T_NATIVE_REAL`

- Native datatype might differ from platform to platform
- HDF5 file type depends on compiler switches and underlying platform
- Native datatypes are not in an HDF file but the pre-defined ones which are referred to by native datatypes appear in the HDF5 files.

Dataspace

- The dataspace is part of the metadata of the underlying dataset
- Metadata are:
  - Dataspace
  - Datatype
  - Attributes
  - Storage info
- The dataspace describes the size and shape of the dataset

Simple dataspace

```
rank: int
current_size: hsize_t[rank]
maximum_size: hsize_t[rank]
```

rank = `2`, dimensions = `2x5`

Creating a dataspace

C

```
hid_t H5Screate_simple(int rank,
                       const hsize_t *current_dims,
                       const hsize_t *maximum_dims)
```

Fortran

```
H5SCREATE_SIMPLE_F(RANK, DIMS, SPACE_ID, HDFERR,
                   MAXDIMS)
  INTEGER, INTENT(IN) :: RANK
  INTEGER(KIND=HISIZE_T)(*), INTENT(IN) :: DIMS
  INTEGER(KIND=HID_T), INTENT(OUT) :: SPACE_ID
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(KIND=HISIZE_T)(*), OPTIONAL,
    INTENT(OUT) :: MAXDIMS
```

- `rank`: Number of dimensions
- `maximum_dims` may be `NULL`. Then `maximum_dims` and `current_dims` are the same
- `H5S_UNLIMITED[_F]` can be used as `maximum_dims` to set dimensions to “infinite” size
- use `H5Sclose[_f]` to finalize dataspace access

Creating a dataspace

C
```
hid_t H5Screate(H5S_class_t type)
```

Fortran
```
H5SCREATE_F(CLASSTYPE, SPACE_ID, HDFERR)
  INTEGER, INTENT(IN) :: CLASSTYPE
  INTEGER(HID_T), INTENT(OUT) :: SPACE_ID
  INTEGER, INTENT(OUT) :: HDFERR
```

- classtype: H5S_SCALAR[_F] or H5S_SIMPLE[_F]

Creating an Attribute

C

```
hid_t H5Acreate(hid_t loc_id, const char *attr_name,
                hid_t type_id, hid_t space_id,
                hid_t acpl_id, hid_t aapl_id)
```

Fortran

```
H5ACREATE_F(LOC_ID, NAME, TYPE_ID, SPACE_ID,
            ATTR_ID, HDFERR, ACPL_ID, AAPL_ID)
  INTEGER(KIND=HID_T), INTENT(IN) :: LOC_ID
  CHARACTER(LEN=*), INTENT(IN) :: NAME
  INTEGER(KIND=HID_T), INTENT(IN) :: TYPE_ID,
    SPACE_ID
  INTEGER(KIND=HID_T), INTENT(OUT) :: ATTR_ID
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(KIND=HID_T), OPTIONAL, INTENT(IN) ::
    ACPL_ID, AAPL_ID
```

- `loc_id` may be any HDF5 object identifier (group, dataset, or committed datatype) or an HDF5 file identifier
- `ACPL_ID, AAPL_ID`: `H5P_DEFAULT[_F]` if not specified
- use `H5Aclose[_f]` to finalize the attribute access

Writing an Attribute

C

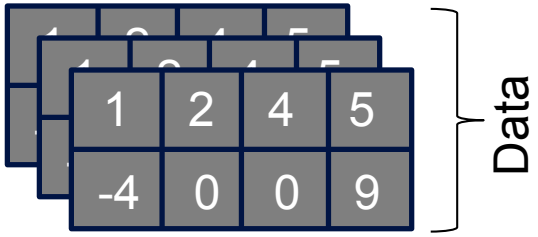
```
herr_t H5Awrite(hid_t attr_id, hid_t mem_type_id,
                const void *buf)
```

Fortran

```
H5AWRITE_F(ATTR_ID, MEMTYPE_ID, BUF, DIMS, HDFERR)
    INTEGER(KIND=HID_T), INTENT(IN) :: ATTR_ID
    INTEGER(KIND=HID_T), INTENT(IN) :: MEMTYPE_ID
    TYPE, INTENT(IN) :: BUF
    INTEGER(KIND=HSIZE_T)(*), INTENT(IN):: DIMS
    INTEGER, INTENT(OUT) :: HDFERR
```

- Fortran: `DIMS` array to hold corresponding dimension sizes of data buffer `buf` (new since 1.4.2)

Dataset (metadata + data)



Metadata

Dataspace
- rank = 3
- dim[0] = 2
- dim[1] = 4
- dim[2] = 3

Datatype
- Integer

Attributes
- Time = 2.1
- Temp = 122

Storage
- Contiguous

Creating a Dataset

C

```
hid_t H5Dcreate(hid_t loc_id, const char *name,
                hid_t dtype_id, hid_t space_id,
                hid_t lcpl_id, hid_t dcpl_id,
                hid_t dapl_id)
```

Fortran

```
H5DCREATE_F(LOC_ID, NAME, TYPE_ID, SPACE_ID,
          DSET_ID, HDFERR, DCPL_ID, LCPL_ID, DAPL_ID)
  INTEGER(KIND=HID_T), INTENT(IN) :: LOC_ID
  CHARACTER(LEN=*), INTENT(IN) :: NAME
  INTEGER(KIND=HID_T), INTENT(IN) :: TYPE_ID,
    SPACE_ID
  INTEGER(KIND=HID_T), INTENT(OUT) :: DSET_ID
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(KIND=HID_T), OPTIONAL, INTENT(IN) ::
    DCPL_ID, LCPL_ID, DAPL_ID
```

- use `H5Dclose[_f]` to finalize the dataset access

Creating a Dataset

- `type_id`: Datatype identifier
- `space_id`: Dataspace identifier
- `dcpl_id`: Dataset creation property list
- `lcpl_id`: Link creation property list
- `dapl_id`: Dataset access property list

Property Lists

- Property lists (`H5P`) can be used to change the internal data handling in HDF5
- Default: `H5P_DEFAULT[_F]`
- Creation properties
  - Whether a dataset is stored in a compact, contiguous, or chunked layout
  - Specify filters to be applied to a dataset (e.g. gzip compression or checksum evaluation)
- Access properties
  - The driver used to open a file (e.g. MPI-I/O or Posix)
  - Optimization settings in specialized environments
- Transfer properties
  - Collective or independent I/O

Recipe: Creating an empty dataset

1. Get identifier for dataset location
2. Specify datatype (integer, composite etc.)
3. Define dataspace
4. Specify property lists (or `H5P_DEFAULT[_F]`)
5. Create dataset
6. Close all opened objects

Exercise

Exercise 5 – HDF5 metadata handling

- Extend your serial program
- Create inside the “data” group an empty dataset which should be a two dimensional array (5x20 elements) of integer values
- Add a string attribute connected to this dataset (the string type definition is already available within the template file)
- Write a string value into this attribute

Check the resulting file using:

```
h5dump
```

Writing to a dataset

C

```
herr_t H5Dwrite(hid_t dataset_id, hid_t mem_type_id,
                hid_t mem_space_id, hid_t
                file_space_id, hid_t xfer_plist_id,
                const void * buf )
```

Fortran

```
H5DWRITE_F(DSET_ID, MEM_TYPE_ID, BUF, DIMS, HDFERR,
           MEM_SPACE_ID, FILE_SPACE_ID, XFER_PRP)
  INTEGER(HID_T), INTENT(IN) :: DSET_ID, MEM_TYPE_ID
  TYPE, INTENT(IN) :: BUF
  DIMENSION(*), INTEGER(HSIZE_T), INTENT(IN) :: DIMS
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(HID_T), OPTIONAL, INTENT(IN) ::
    MEM_SPACE_ID, FILE_SPACE_ID, XFER_PRP
```

- `H5S_ALL[_F]` can be used to specify no special `mem_space` or `file_space` identifier
- `xfer_plist_id/xfer_prp` is a transfer property (e.g. to specify collective or independent parallel I/O)

Writing to a dataset

mem_space_id	file_space_id	Behaviour
dataspace id	dataspace id	use dataspaces as is
H5S_ALL	dataspace id	use given file_space dataspace also for mem_space dataspace (including the selection)
dataspace id	H5S_ALL	use *all* selection for default file_space
H5S_ALL	H5S_ALL	use default file_space also for mem_space, set *all* selection for both

Open a existing dataset

C

```
hid_t H5Dopen(hid_t loc_id, const char *name, hid_t
              dapl_id)
```

Fortran

```
H5DOPEN_F(LOC_ID, NAME, DSET_ID, HDFERR)
  INTEGER(HID_T), INTENT(IN) :: LOC_ID
  CHARACTER(LEN=*), INTENT(IN) :: NAME
  INTEGER(HID_T), INTENT(OUT) :: DSET_ID
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(HID_T), OPTIONAL, INTENT(IN) :: DAPL_ID
```

- `dapl_id`: Dataset access property list

Dataspace inquiry

C
```
hid_t H5Dget_space(hid_t dataset_id)
```

Fortran
```
H5DGET_SPACE_F(DATASET_ID, DATASPACE_ID, HDFERR)
  INTEGER(HID_T), INTENT(IN) :: DATASET_ID
  INTEGER(HID_T), INTENT(OUT) :: DATASPACE_ID
  INTEGER, INTENT(OUT) :: HDFERR
```

- Returns an identifier for a copy of the dataspace for a dataset.
- `H5Sget_simple_extent_ndims` and `H5Sget_simple_extent_dims` can be used to extract dimension information

Reading a dataset

C

```
herr_t H5Dread(hid_t dataset_id, hid_t mem_type_id,
               hid_t mem_space_id, hid_t
               file_space_id, hid_t xfer_plist_id,
               void * buf)
```

Fortran

```
H5DREAD_F(DSET_ID, MEM_TYPE_ID, BUF, DIMS, HDFERR,
          MEM_SPACE_ID, FILE_SPACE_ID, XFER_PRP)
  INTEGER(HID_T), INTENT(IN) :: DSET_ID, MEM_TYPE_ID
  TYPE, INTENT(IN) :: BUF
  DIMENSION(*), INTEGER(HSIZE_T), INTENT(IN) :: DIMS
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(HID_T), OPTIONAL, INTENT(IN) ::
    MEM_SPACE_ID, FILE_SPACE_ID, XFER_PRP
```

- `H5S_ALL[_F]` can be used to specify no special `mem_space` or `file_space` identifier
- `xfer_plist_id/xfer_prp` is a transfer property (e.g. to specify collective or independent parallel I/O)

Exercise

Exercise 6 – HDF5 write data

- Extend your serial program
- Create a two dimensional array with values 1 up to 100

```
 1  2  3  4  5  6  7 …
21 22 23 24 25 26 27 …
41 42 43 44 45 46 47 …
…
```

- Write this array into the existing empty HD5 dataset

Check the resulting file using:

```
h5dump
```

Excursion: row-major / column-major order

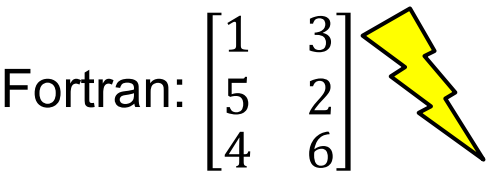
"Logical" data view:

$$M[i,j] = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Adress	1	2	3	4	5	6
Value C	1	2	3	4	5	6
Value Fortran	1	3	5	2	4	6

Storing data in a `3x2` dimensional HDF5 dataset:

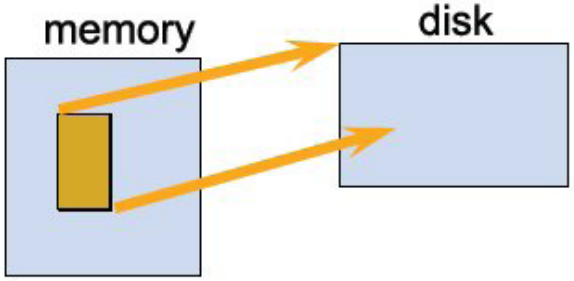
$$\text{C: } \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad \text{Fortran: } \begin{bmatrix} 1 & 3 \\ 5 & 2 \\ 4 & 6 \end{bmatrix}$$



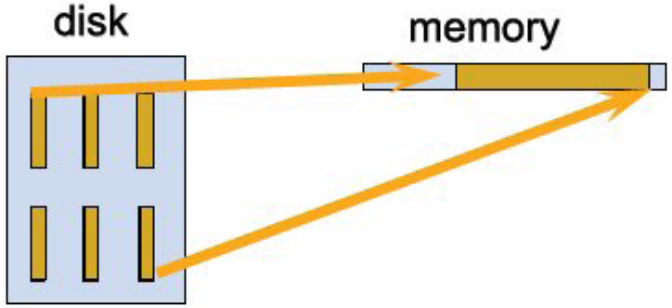
Storing data in a `2x3` dimensional dataset:

$$\text{Fortran: } \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Partial I/O - Hyperslabs

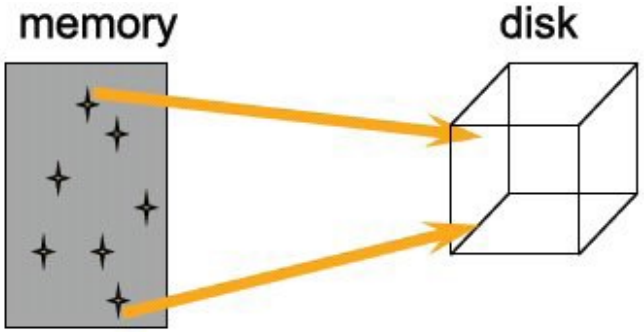


(a) Hyperslab from a 2D array to the corner of a smaller 2D array

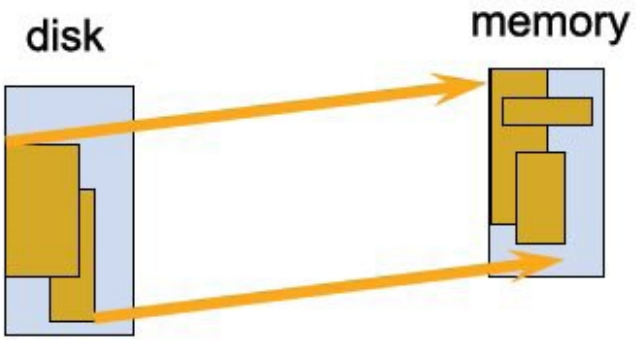


(b) Regular series of blocks from a 2D array to a contiguous sequence at a certain offset in a 1D array

Partial I/O - Hyperslabs



(c) A sequence of points from a 2D array to a sequence of points in a 3D array.



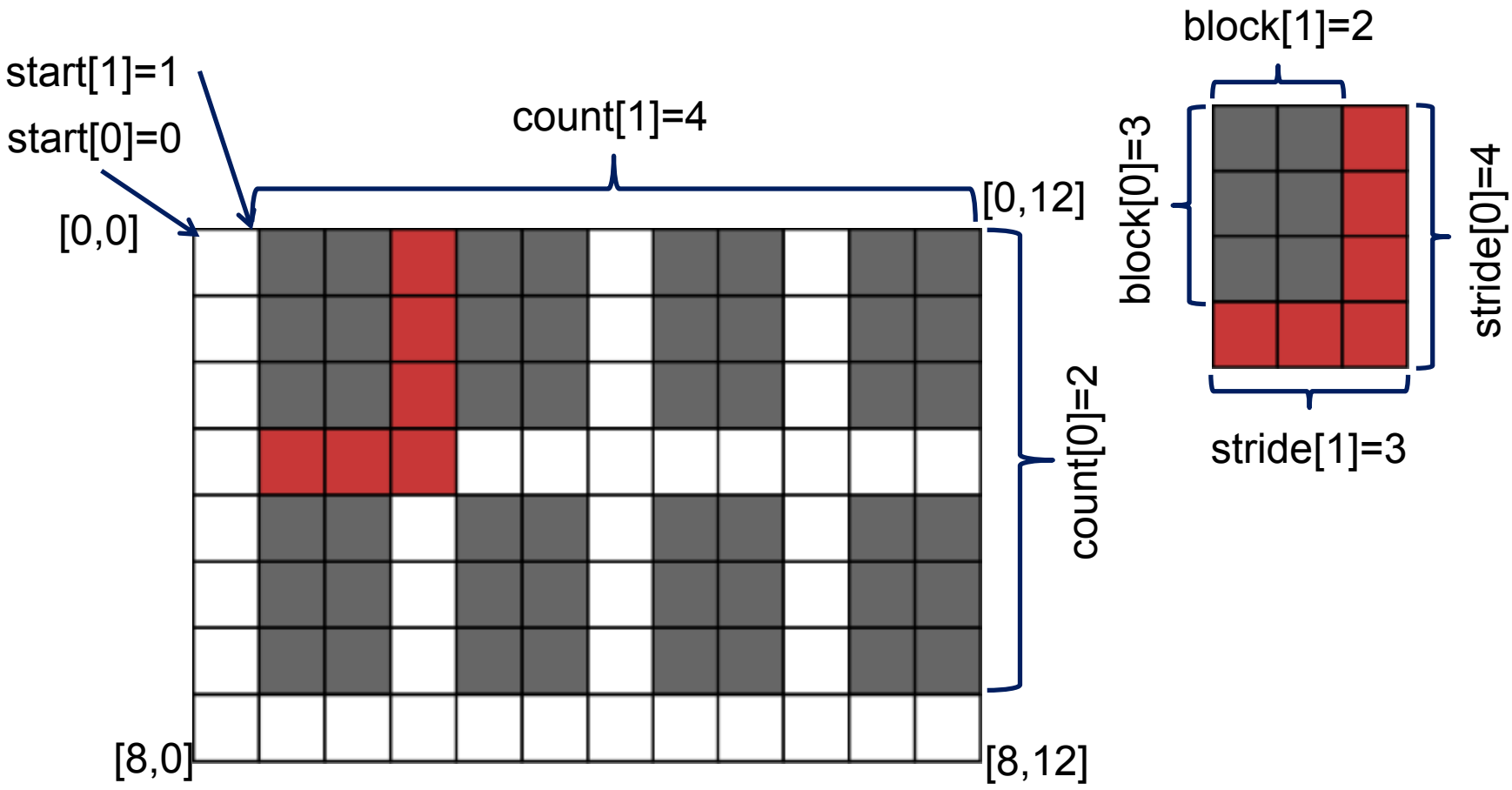
(d) Union of hyperslabs in file to union of hyperslabs in memory.

Partial I/O - Hyperslabs

- Hyperslabs are portions of datasets
  - Contiguous collection of points in a dataspace
  - Regular pattern of points in a dataspace
  - Blocks in a dataspace

- Hyperslabs are described by four parameters:
  - start: (or offset): starting location
  - stride: separation blocks to be selected
  - count: number of blocks to be selected
  - block: size of block to be selected from dataspace
  - **Dimension of these four parameters corresponds to dimension of the underlying dataspace**

Hyperslab example



Creating hyperslabs

C

```
herr_t H5Sselect_hyperslab(hid_t space_id,
            H5S_seloper_t op, const hsize_t *start,
            const hsize_t *stride, const hsize_t
            *count, const hsize_t *block )
```

Fortran

```
H5SSELECT_HYPERSLAB_F(SPACE_ID, OPERATOR, START,
          COUNT, HDFERR, STRIDE, BLOCK)
  INTEGER(HID_T), INTENT(IN) :: SPACE_ID
  INTEGER, INTENT(IN) :: OP
  INTEGER(HSIZE_T), DIMENSION(*), INTENT(IN) ::
    START, COUNT
  INTEGER, INTENT(OUT) :: HDFERR
  INTEGER(HSIZE_T), DIMENSION(*), OPTIONAL,
    INTENT(IN) :: STRIDE, BLOCK
```

Creating hyperslabs

The following operators (`op`) are supported to combine old and new selections:

- `H5S_SELECT_SET[_F]`: Replaces the existing selection with the parameters from this call. Overlapping blocks are not supported with this operator.
- `H5S_SELECT_OR[_F]`: Adds the new selection to the existing selection.
- `H5S_SELECT_AND[_F]`: Retains only the overlapping portions of the new selection and the existing selection.
- `H5S_SELECT_XOR[_F]`: Retains only the elements that are members of the new selection or the existing selection, excluding elements that are members of both selections.
- `H5S_SELECT_NOTB[_F]`: Retains only elements of the existing selection that are not in the new selection.
- `H5S_SELECT_NOTA[_F]`: Retains only elements of the new selection that are not in the existing selection.

VIENNA SCIENTIFIC CLUSTER

JÜLICH FORSCHUNGSZENTRUM

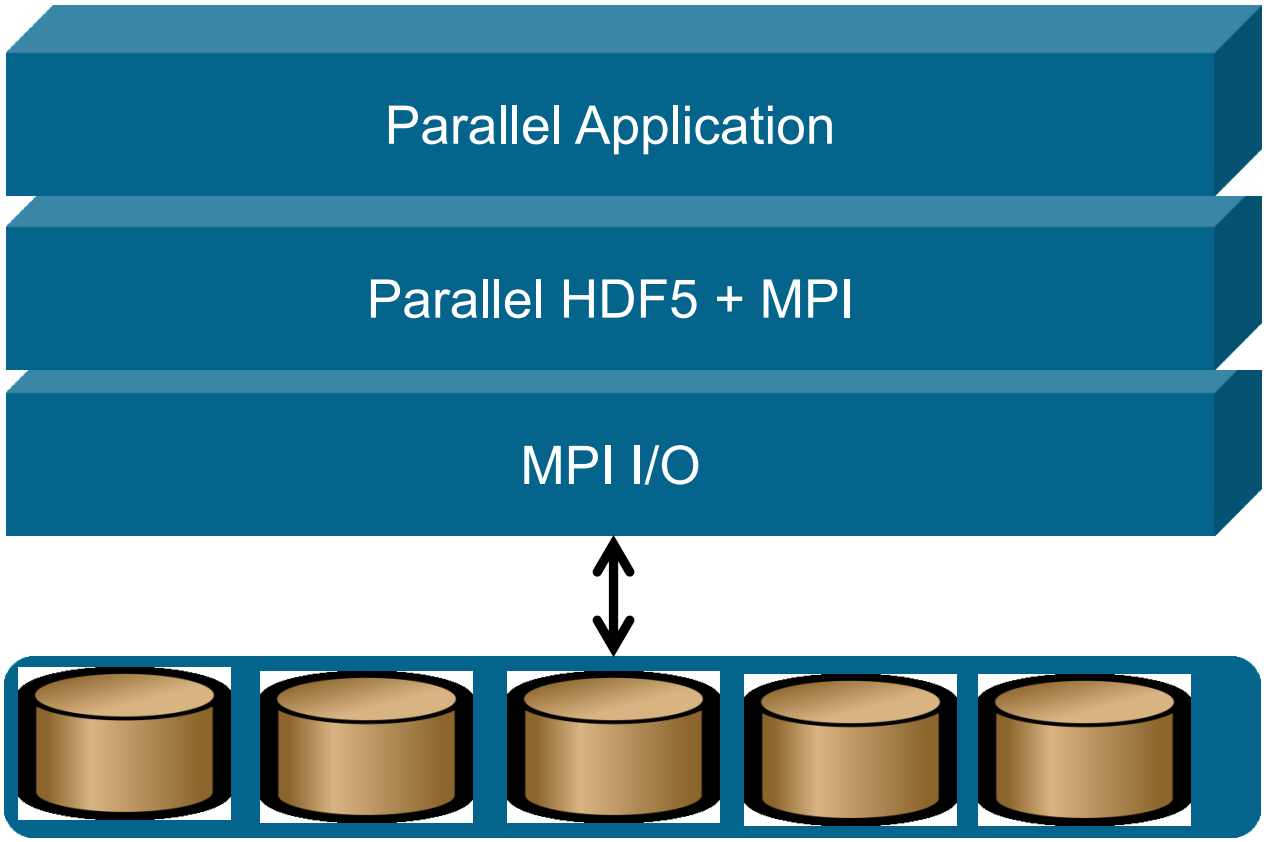
Parallel HDF5

Vienna, Austria, 6th December 2017

Factoids

- Supports MPI programming
- PHDF5 files compatible with serial HDF5 files
  - Shareable between different serial or parallel platforms
- Single file image to all processes
  - One file per process design is undesirable
- A standard parallel I/O interface must be portable to different platforms.

Implementation layers



User Application

HDF5 library

Parallel I/O layer

Parallel file systems
(GPFS, Lustre, …)

Important to know

- Most functions of the PHDF5 API are collectives
  - i.e. all processes of the communicator must participate
- PHDF5 opens a parallel file with a communicator
  - Returns a file-handle
  - Future access to the file via the file-handle
  - Different files can be opened via different communicators
- After a file is opened by the processes of a communicator
  - All parts of file are accessible by all processes
  - All objects in the file are accessible by all processes
  - Multiple processes may write to the same data array
  - Each process may write to an individual data array

MPI-IO access template

C

```
hid_t H5Pcreate(hid_t cls_id);
herr_t H5Pset_fapl_mpio(hid_t fapl_id, MPI_Comm
                        comm, MPI_Info info)
```

Fortran

```
H5PCREATE_F(CLASSTYPE, PRP_ID, HDFERR)
  INTEGER, INTENT(IN) :: CLASSTYPE
  INTEGER(HID_T), INTENT(OUT) :: PRP_ID
  INTEGER, INTENT(OUT) :: HDFERR
H5PSET_FAPL_MPIO_F(PRP_ID, COMM, INFO, HDFERR)
  INTEGER(HID_T), INTENT(IN) :: PRP_ID
  INTEGER, INTENT(IN) :: COMM
  INTEGER, INTENT(IN) :: INFO
  INTEGER, INTENT(OUT) :: HDFERR
```

- `cls_id/classtype` must be `H5P_FILE_ACCESS[_F]`
- Property is used during file creation/access
- Each process of the MPI communicator creates an access template and sets it up with MPI parallel access information

Dataset transfer property

C

```
hid_t H5Pcreate(hid_t cls_id);
herr_t H5Pset_dxpl_mpio(hid_t dxpl_id,
                        H5FD_mpio_xfer_t xfer_mode )
```

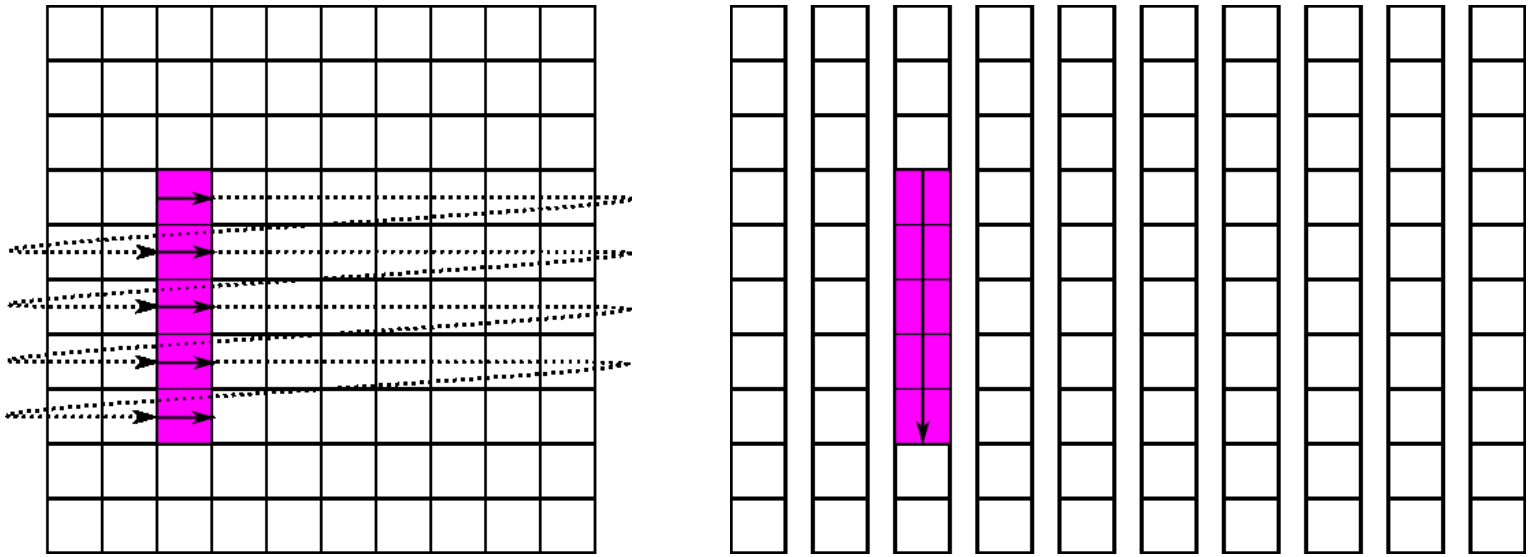
Fortran

```
H5PCREATE_F(CLASSTYPE, PRP_ID, HDFERR)
  INTEGER, INTENT(IN) :: CLASSTYPE
  INTEGER(HID_T), INTENT(OUT) :: PRP_ID
  INTEGER, INTENT(OUT) :: HDFERR
H5PSET_DXPL_MPIO_F(PRP_ID, DATA_XFER_MODE, HDFERR)
  INTEGER(HID_T), INTENT(IN) :: PRP_ID
  INTEGER, INTENT(IN) :: DATA_XFER_MODE
  INTEGER, INTENT(OUT) :: HDFERR
```

- `cls_id/classtype` must be `H5P_DATASET_XFER[_F]`
- `xfer_modes:`
  - `H5FD_MPIO_INDEPENDENT[_F]`: Use independent I/O access (default)
  - `H5FD_MPIO_COLLECTIVE[_F]`: Use collective I/O access

Performance hints

- **Chunking:** Contiguous datasets are stored in a single block in the file, chunked datasets are split into multiple chunks which are all stored separately in the file.
- Additional chunk cache is possible



C

```
dcpl_id = H5Pcreate(H5P_DATASET_CREATE);
H5Pset_chunk(dcpl_id, 2, chunk_dims);
```

https://www.hdfgroup.org/HDF5/doc/Advanced/Chunking/

Performance hints

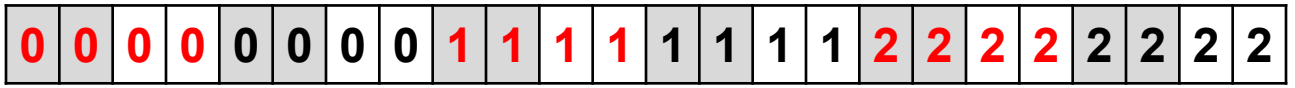
h5perf

- Simple HDF5 I/O-benchmark application
- 1D or 2D dataset
- Part of the standard HDF5 installation
- Contiguous or interleaved access pattern
- Independent and collective I/O
- Chunking
- Example Options (`h5perf -h`):
  - 1D / 2D (`-g`)
  - Bytes per Process (`-e`)
  - Block size (`-B`)
  - Transfer size (`-x / -X`)
  - Number of datasets (`-d`)

Performance hints

Example (1D):

- `num-processes = 3`
- `bytes-per-process = 8`
- `block-size = 2`
- `transfer-buffer-size = 4`
- `contiguous`



1 write operation per transfer

- `interleaved`

2 write operations per transfer

https://www.hdfgroup.org/HDF5/doc/Tools/h5perf_parallel/h5perf_parallel.pdf

Performance hints

Example (2D):

- `num-processes = 2`
- `bytes-per-process = 4`
- `block-size = 2`
- `transfer-buffer-size = 8`

`contiguous`

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

1 write operation per transfer

`interleaved`

0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1

8 write operations per transfer

https://www.hdfgroup.org/HDF5/doc/Tools/h5perf_parallel/h5perf_parallel.pdf

Exercise

Exercise 7 – parallel HDF5

- Extend your serial program to a parallel program
- Fill your two dimensional array with the rank number
- Create a combined dataset of all processes involved
- Logical view:

```
0 0 0 0 0 0 0 0 …
0 0 0 0 0 0 0 0 …
…
1 1 1 1 1 1 1 1 …
1 1 1 1 1 1 1 1 …
…
```

#cores x 5

- Write the data collectively into the file
- Check the resulting file using: `h5dump`

- Time left? Try to run a `h5perf` benchmark (use `h5perf -h` to see all available options)