PARALLEL I/O AND PORTABLE DATA FORMATS

VSC TRAINING COURSE

06.12.2019 I SEBASTIAN LÜHRS (S.LUEHRS@FZ-JUELICH.DE)

VIENNA SCIENTIFIC CLUSTER

JÜLICH Forschungszentrum

Mitglied der Helmholtz-Gemeinschaft

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

IO-500

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,414,592	148,600.0	200,794.9	10,096
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	Texas Advanced Computing Center/Univ. of Texas United States	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR Dell EMC	448,448	23,516.4	38,745.9	
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray/HPE	387,872	21,230.0	27,154.3	2,384

https://www.top500.org/list/2019/11/

IO500

This is the official list from Supercomputing 2019. The list shows the best result for a given combination of system/institution/filesystem.

Please see also the 10 node challenge ranked list.

#	information								io500		
	list id	institution	system	storage vendor	filesystem type	client nodes	client total procs	data	score	bw GiB/s	md kIOP/s
1	sc19	WekaIO	WekaIO on AWS	WekaIO	WekaIO Matrix	345	8625	zip	938.95	174.74	5045.33
2	sc19	Intel	Wolf	Intel	DAOS	26	728	zip	933.64	183.36	4753.79
3	sc19	National Supercomputing Center in Changsha	Tianhe-2E	National University of Defense Technology	Lustre	480	5280	zip	453.68	209.43	982.78
4	sc19	NVIDIA	DGX-2H SuperPOD	DDN	Lustre	10	400	zip	249.50	86.97	715.76
5	sc19	University of Cambridge	Data Accelerator	Dell EMC	Lustre	128	2048	zip	229.45	131.25	401.13
6	sc19	CEA	Tera-1000	DDN	Lustre	128	4096	zip	210.26	81.01	545.74
7	sc19	WekaIO	WekaIO	WekaIO	WekaIO Matrix	10	2610	zip	156.51	56.22	435.76
8	sc19	CSIRO	bracewell scratch2	Dell/ThinkParQ	beegfs	26	260	zip	94.86	69.10	130.23
9	sc19	State Key Laboratory of High-end Server & Storage Technology（HSS）	TStor3000	INSPUR	BeeGFS	10	300	zip	82.57	41.14	165.71
10	sc19	Google Cloud	EXA5-GCP-PD-SSD	Google Cloud	Lustre	200	1600	zip	67.78	26.93	170.61
11	sc19	DDN	AI400	DDN	Lustre	10	240	zip	63.88	19.65	207.63
12	sc19	Google Cloud	EXA5-GCP-PD-STD	Google Cloud	Lustre	200	1600	zip	52.96	17.31	162.06
13	sc19	Janelia Research Campus, HHMI	Weka	WekaIO	wekaio	18	1368	zip	48.75	26.22	90.62
14	sc19	Oracle Cloud Infrastructure	Oracle Cloud Infrastructure with Block Volume Service running Spectrum Scale	Oracle Cloud Infrastructure Block Volumes Service	Spectrum Scale	30	480	zip	47.55	20.38	110.93

https://www.vi4io.org/io500/start

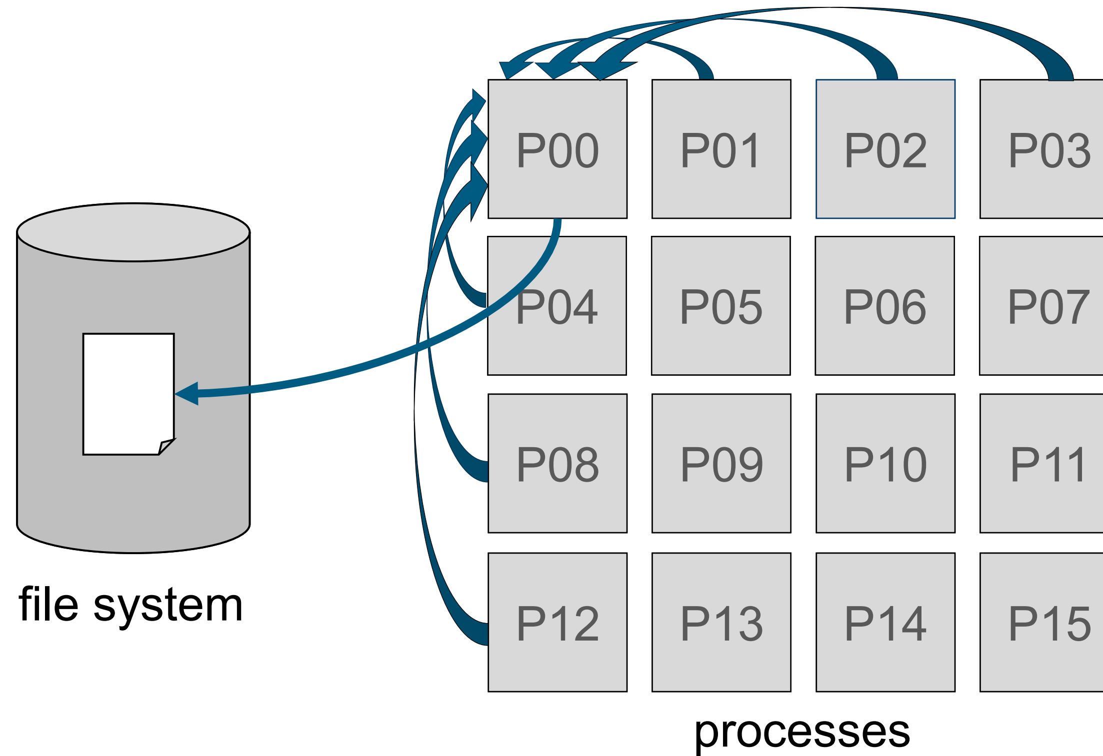
Timetable

- 09:00 - 09:30 Parallel I/O strategies
- 09:30 - 11:00 Parallel NetCDF
- 11:00 - 11:30 Coffee break
- 11:30 - 12:30 Parallel NetCDF
- 12:30 - 13:00 HDF5
- 13:00 - 14:00 Lunch break
- 14:00 - 15:00 HDF5
- 15:00 - 15:30 Coffee break
- 15:30 - 16:00 HDF5
- 16:00 - 17:00 Optimization and Profiling

PARALLEL I/O STRATEGIES

Parallel I/O Strategies

One process performs I/O



Parallel I/O Strategies

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

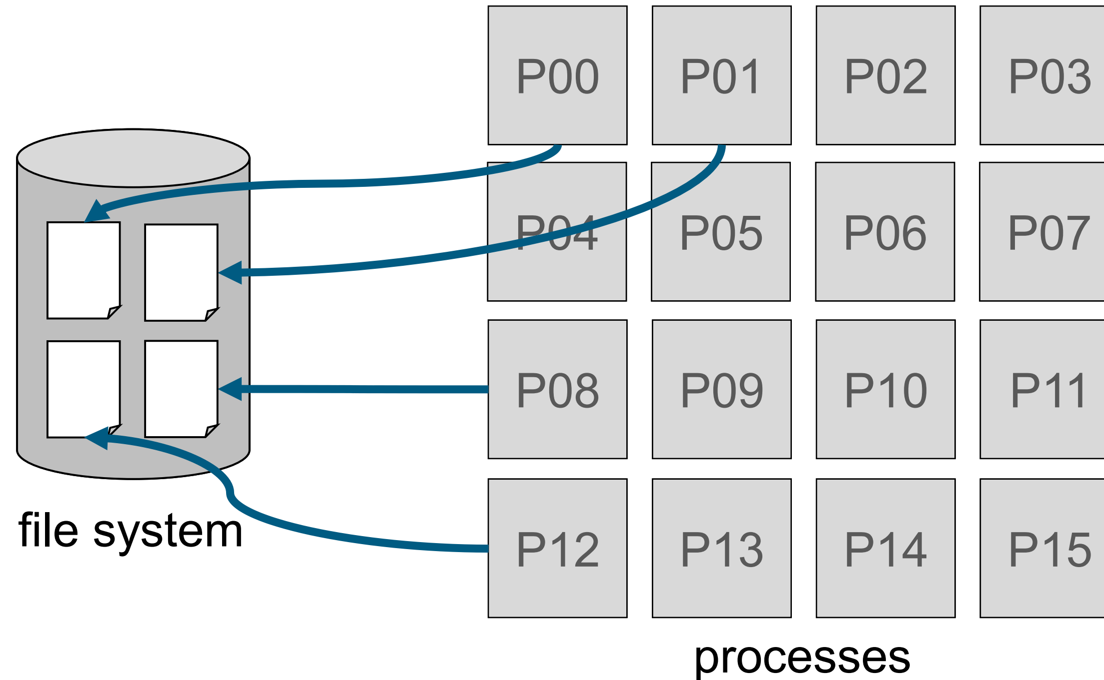
Parallel I/O Pitfalls

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

Parallel I/O Strategies

Task-local files



JÜLICH
Forschungszentrum

Parallel I/O Strategies

Task-local files

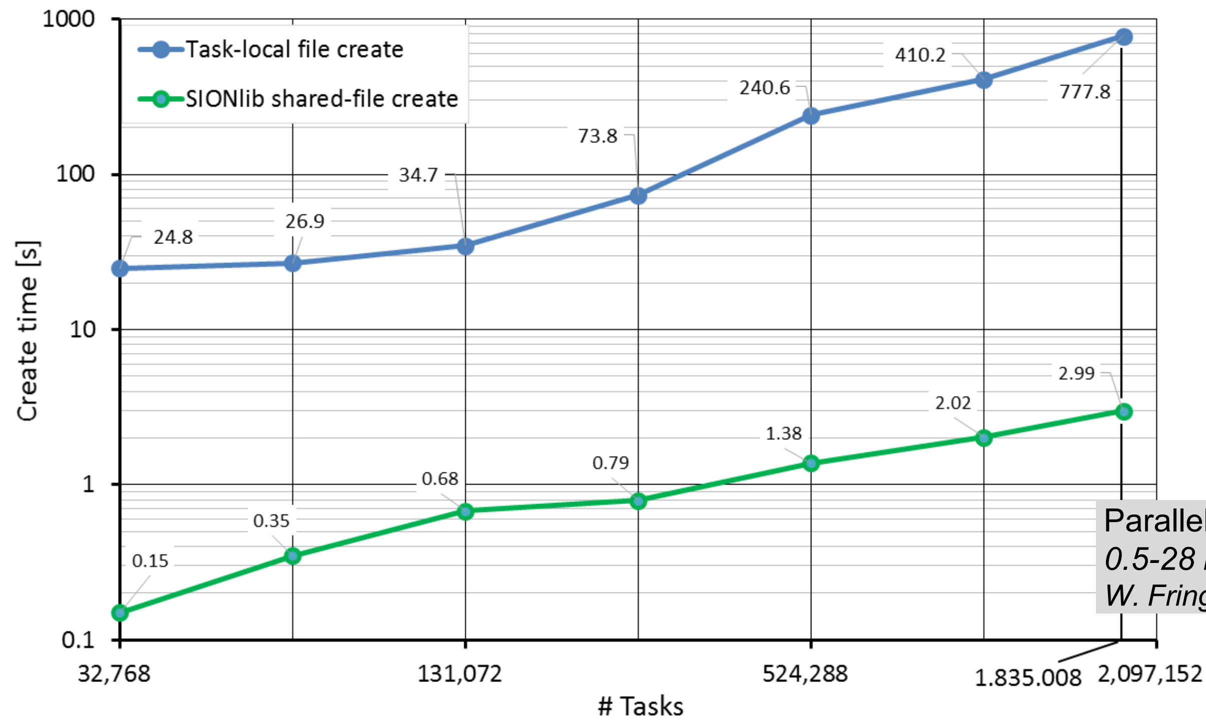
+ Simple to implement
+ No coordination between processes needed
+ No false sharing of file system blocks

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

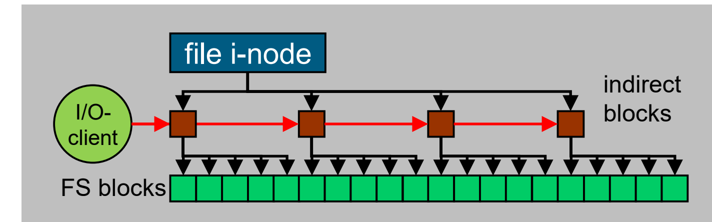
Parallel I/O Pitfalls

Serialization of meta data modification

Example: Creating files in parallel in the same directory



Parallel file creation on *JUQUEEN*
0.5-28 racks, 64 tasks/node
W. Frings

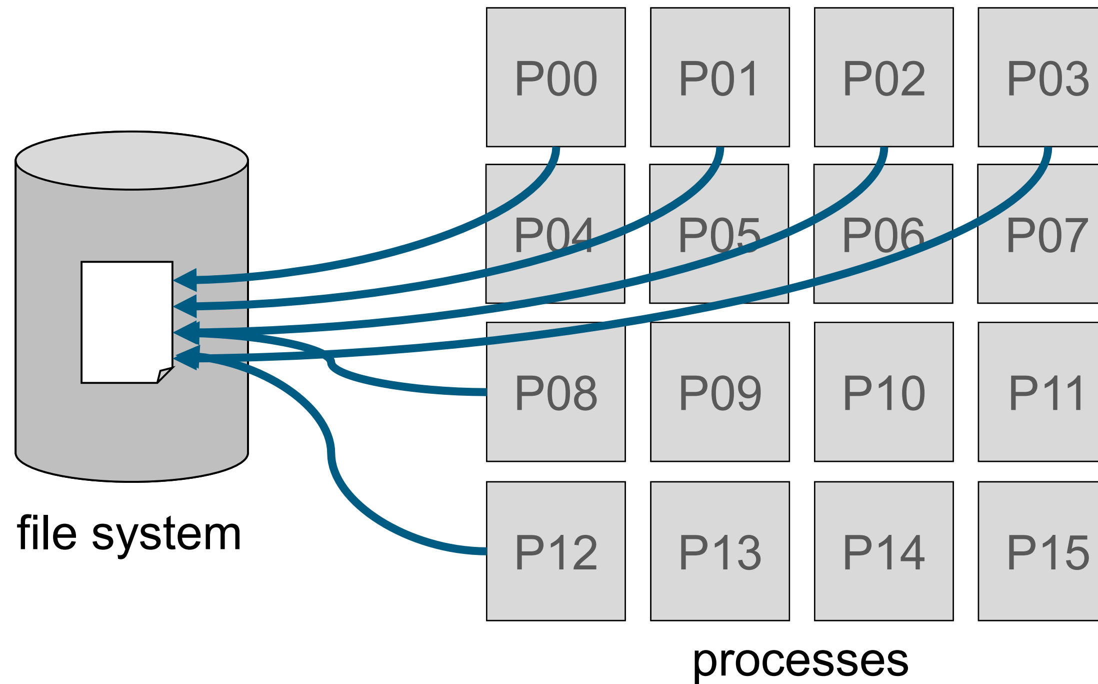


- Meta-data wall on file level
  - File changes by multiple processes can cause serialization
  - File meta-data management
  - Locking

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

Parallel I/O Strategies

Shared files



Parallel I/O Strategies

Shared files

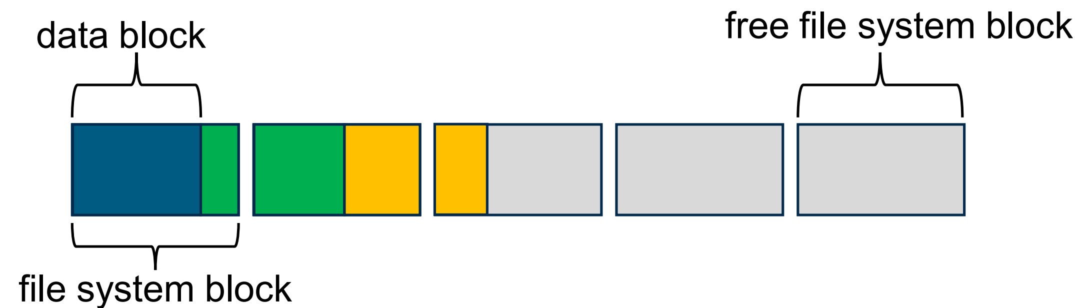
+ Number of files is independent of number of processes
+ File can be in canonical representation (no post-processing)

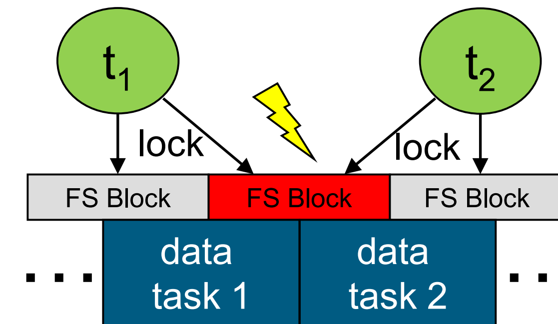
- Uncoordinated client requests might induce time penalties
- File layout may induce false sharing of file system blocks

Parallel I/O Pitfalls

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

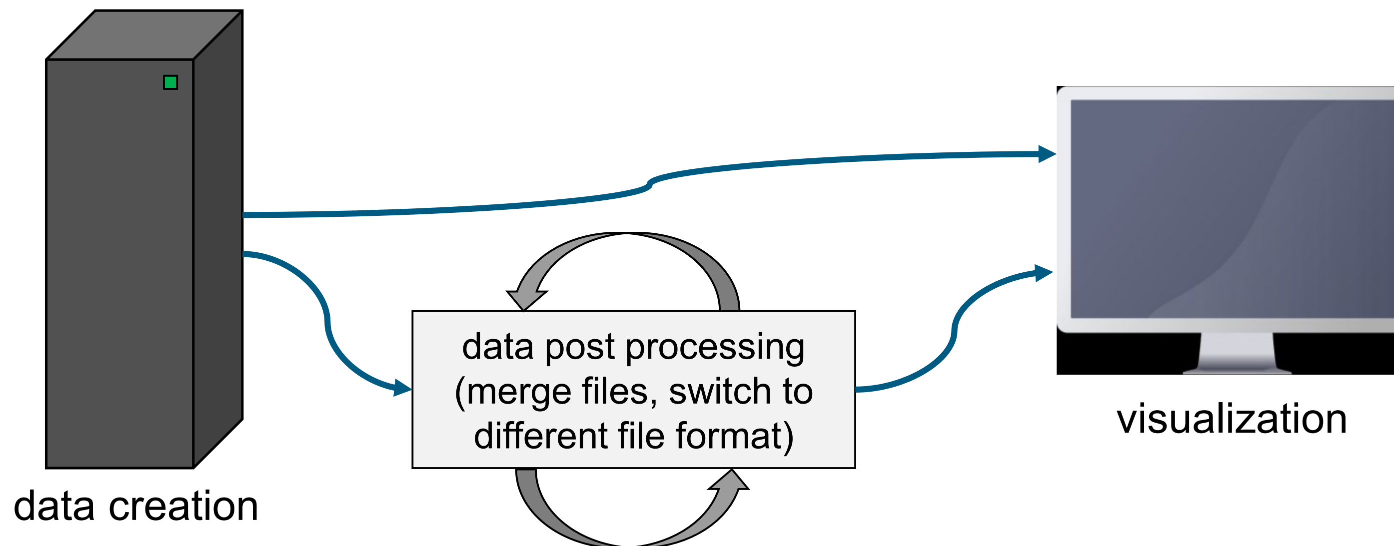




JÜLICH Forschungszentrum

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)



Parallel I/O Pitfalls

Portability

- Endianness (byte order) of binary data
- Conversion of files might be necessary and expensive

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

Parallel I/O Pitfalls

Portability

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

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
…	…	…

Storage Tiers

Different storage tiers with different optimization targets

Data staging at JSC

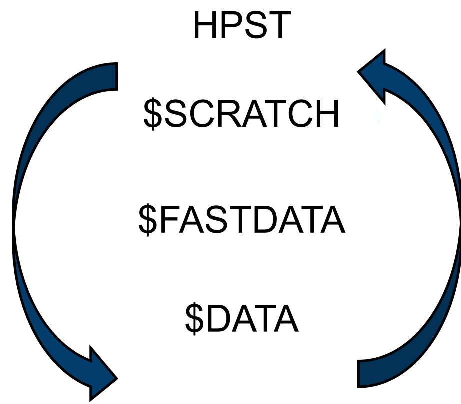
HPST

$SCRATCH

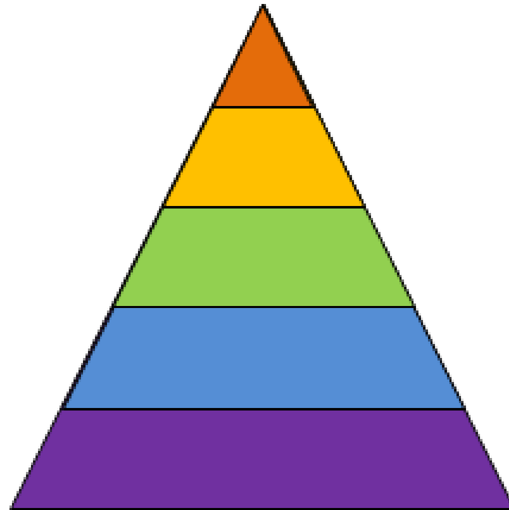
$FASTDATA

$DATA

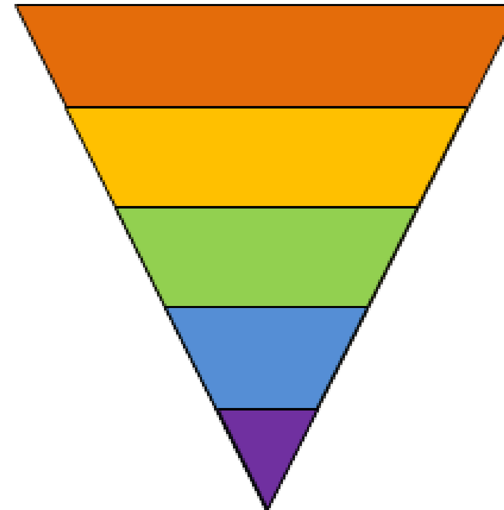
$ARCHIVE



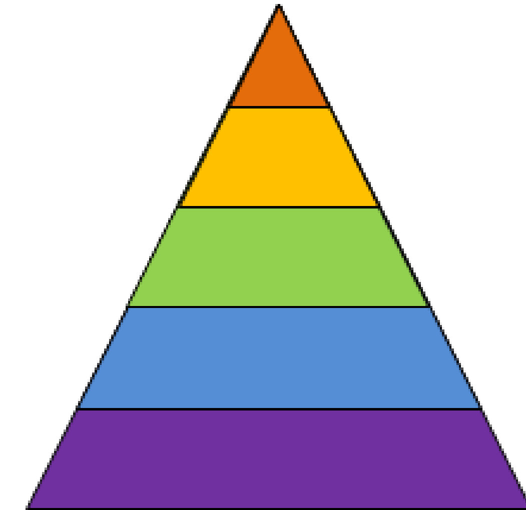
Capacity

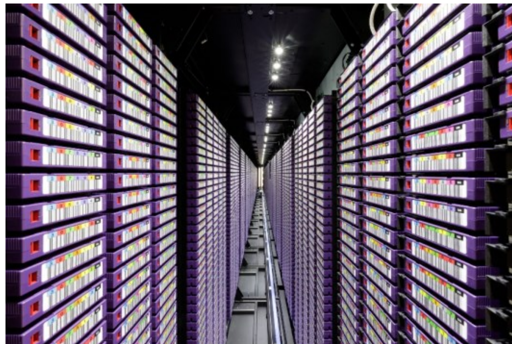


Bandwidth



Retention time





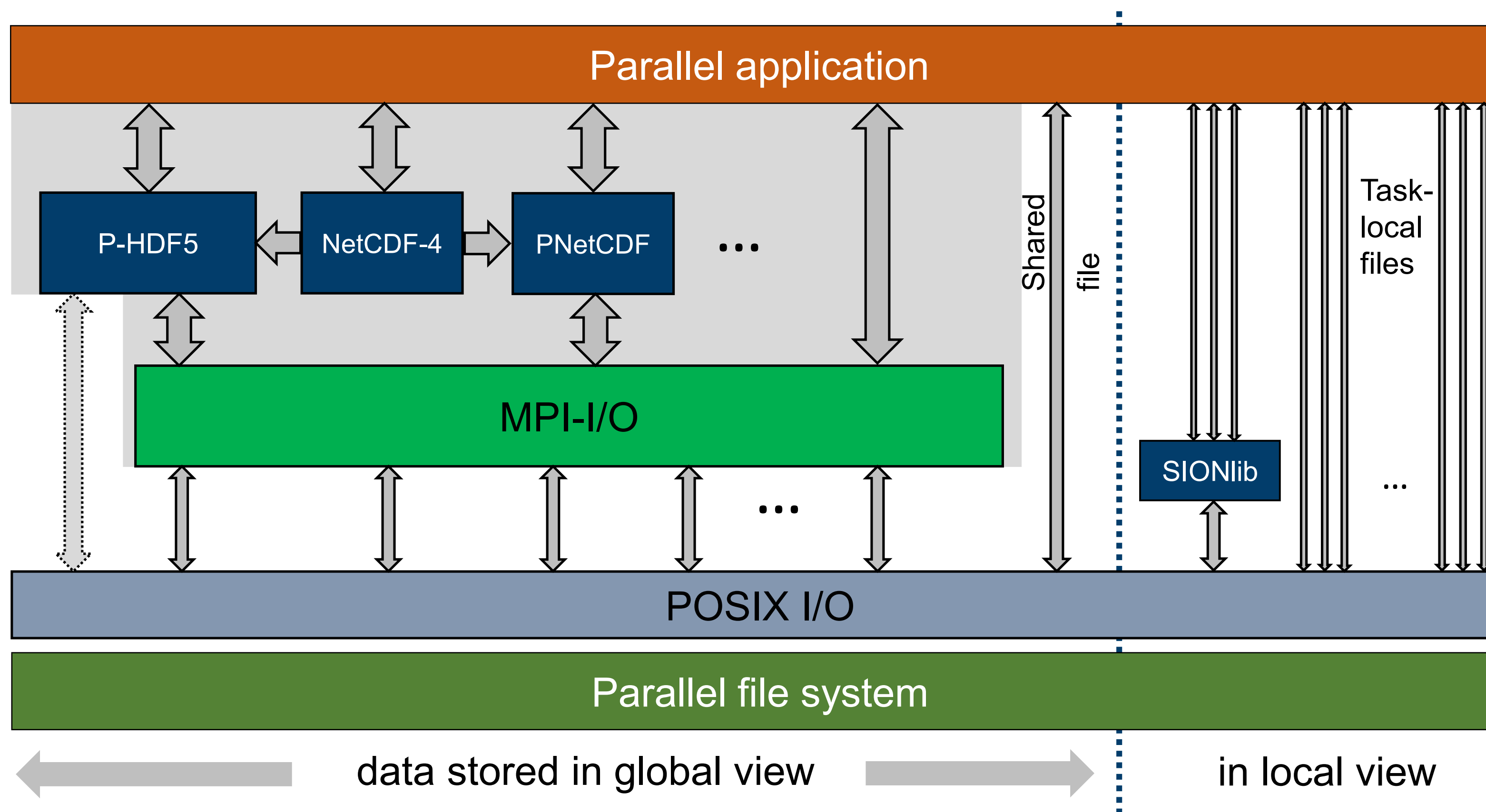
Tape Library

JUST 5

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”
  - External: Write/read data in non-decomposed format (portable, system-independent, self-describing)

Parallel I/O Software Stack



PARALLEL NETCDF

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 ≠ <u>NetCDF4</u>

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.

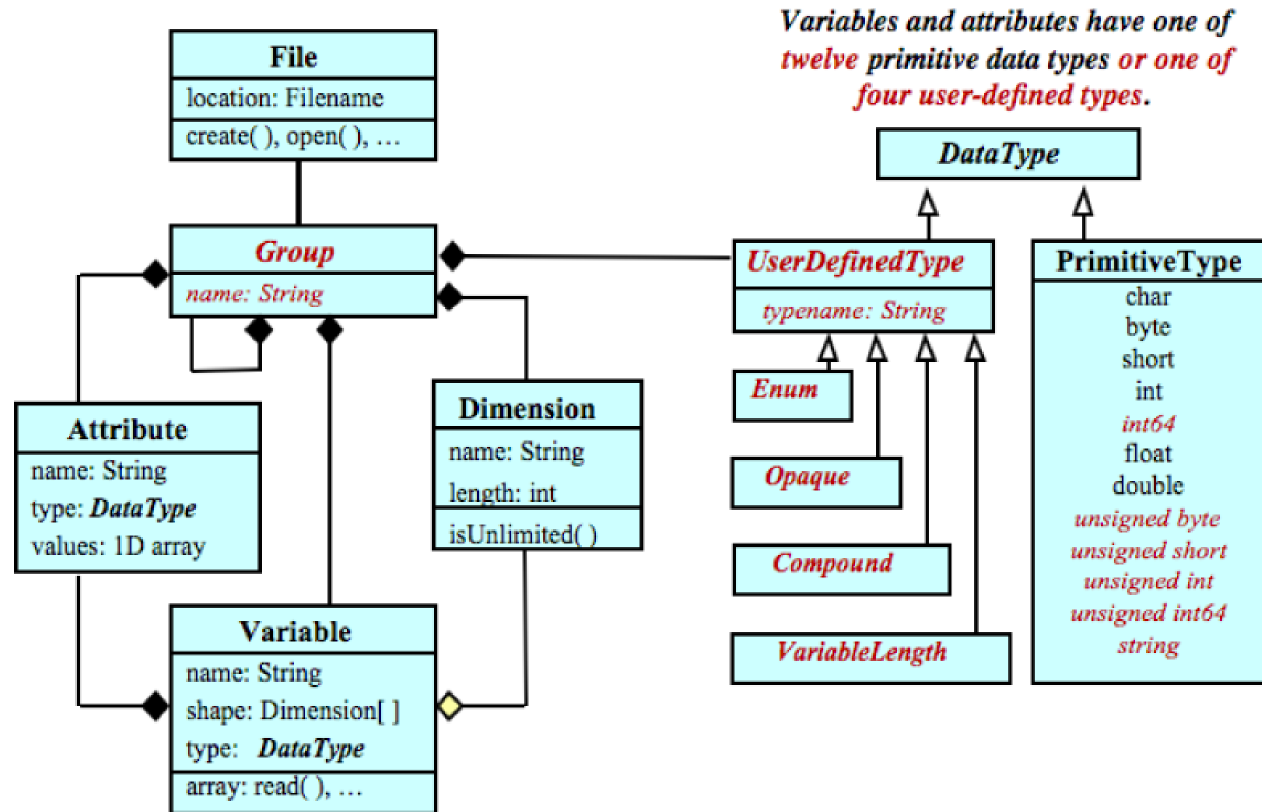
Variable

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.

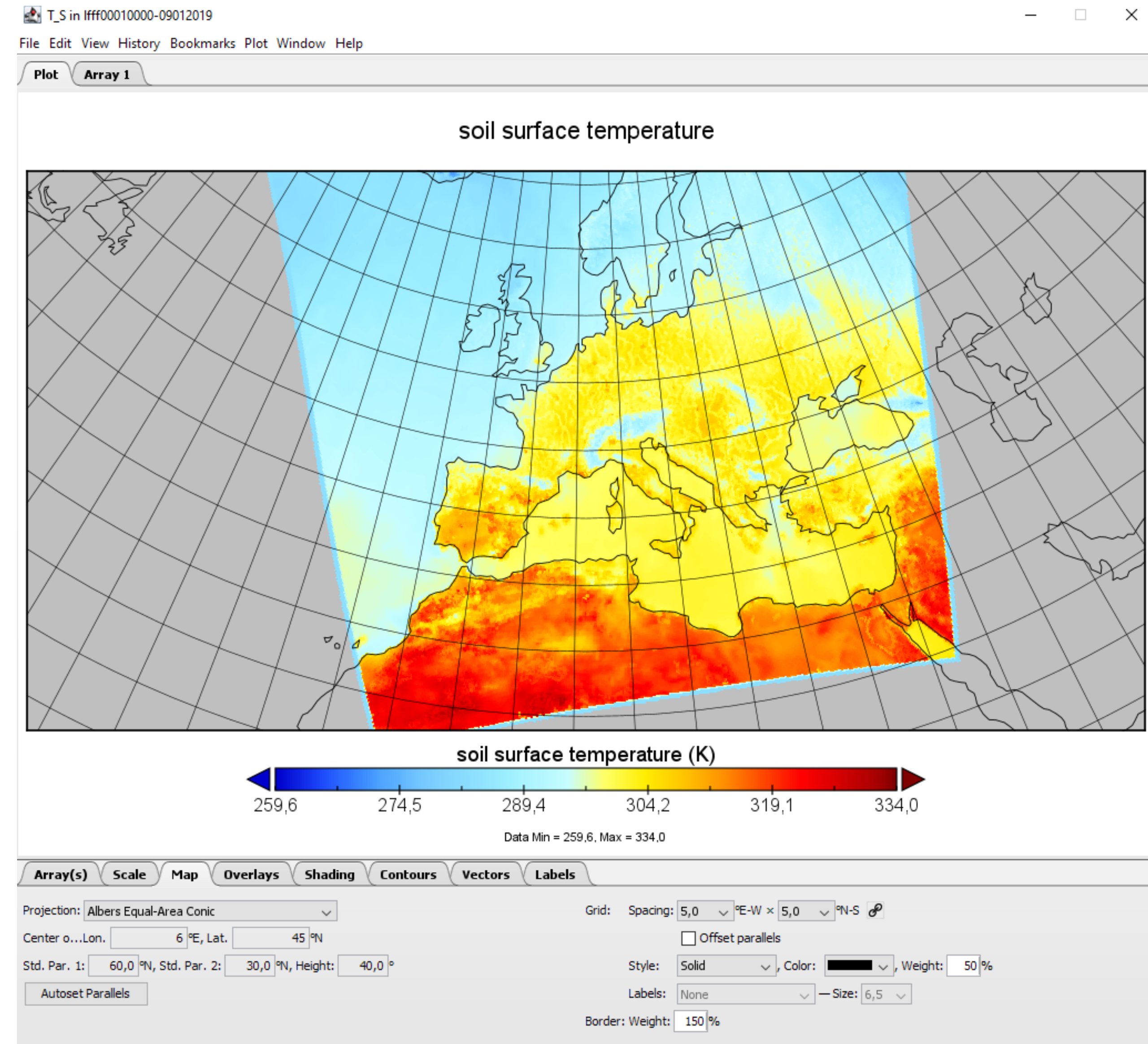
Viewer for NetCDF data

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

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

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

Fortran

```fortran
use netcdf
```

- Contain definition of
  - constants
  - functions

Python

```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|(NC/NF90)_MPIIO` - NetCDF4 format (HDF5)

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

- Create a parallel application (C or Fortran) which creates an empty NetCDF file
- You can use the provided template:
  `helloworld_netcdf_template.c` or
  `helloworld_netcdf_template.f90`
- Compile, link and execute the application

```
module load intel/18 intel-mpi/2018 hdf5/1.8.12-MPI
module load pnetcdf/1.10.0 netcdf_C/4.4.1.1
module load netcdf_Fortran/4.4.4

mpiicc helloworld_netcdf.c -lnetcdf
mpiifort helloworld_netcdf.f90 –lnetcdff
```

To start a job on the compute node:

```
srun -N 1 --ntasks-per-node=16 --reservation=training
--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

```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

```c
int nc_enddef(int ncid)
```

Fortran

```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`

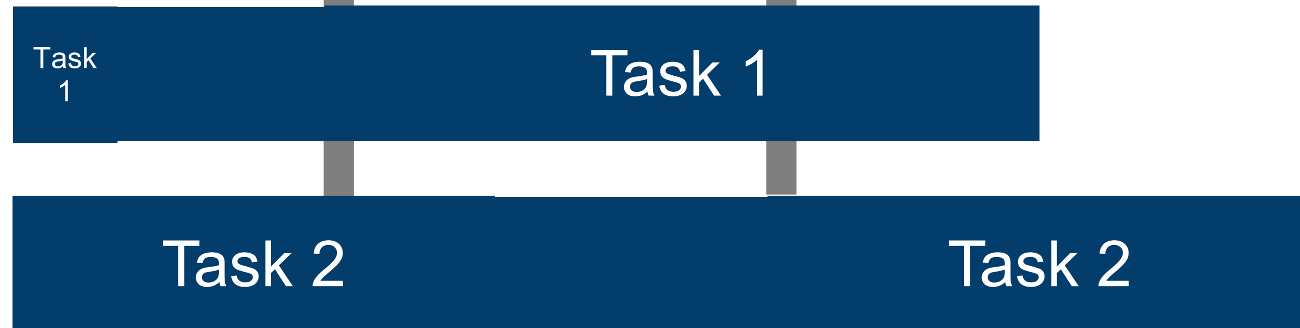
Python

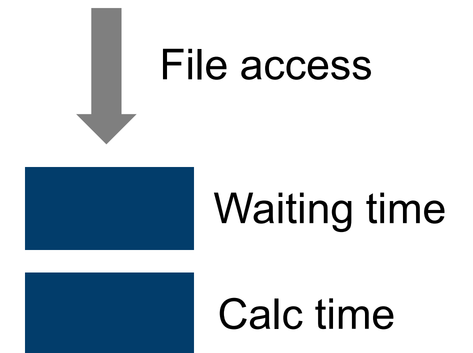
```
var.set_collective(True)
```


JÜLICH Forschungszentrum

Independet vs. collective reading/writing

Independent file access





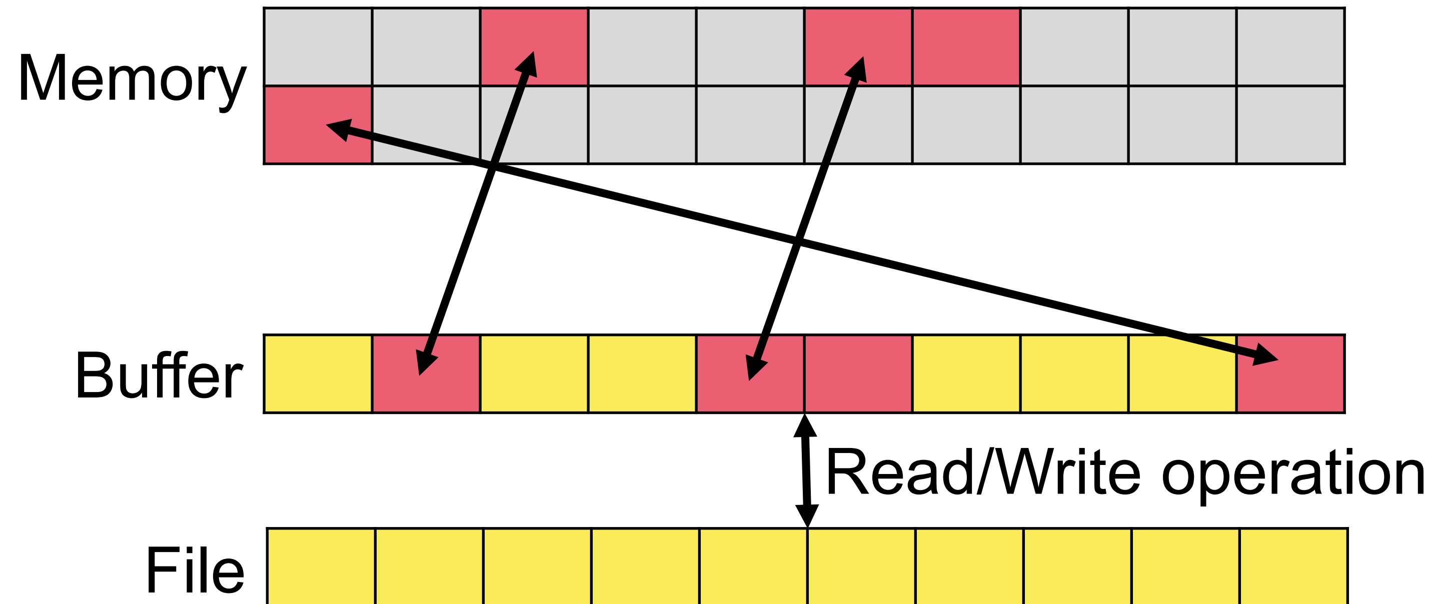
Collective file access

JÜLICH
Forschungszentrum

Benefits of collective file access

Data sieving

- Usage of buffers to group non-contiguous data requests

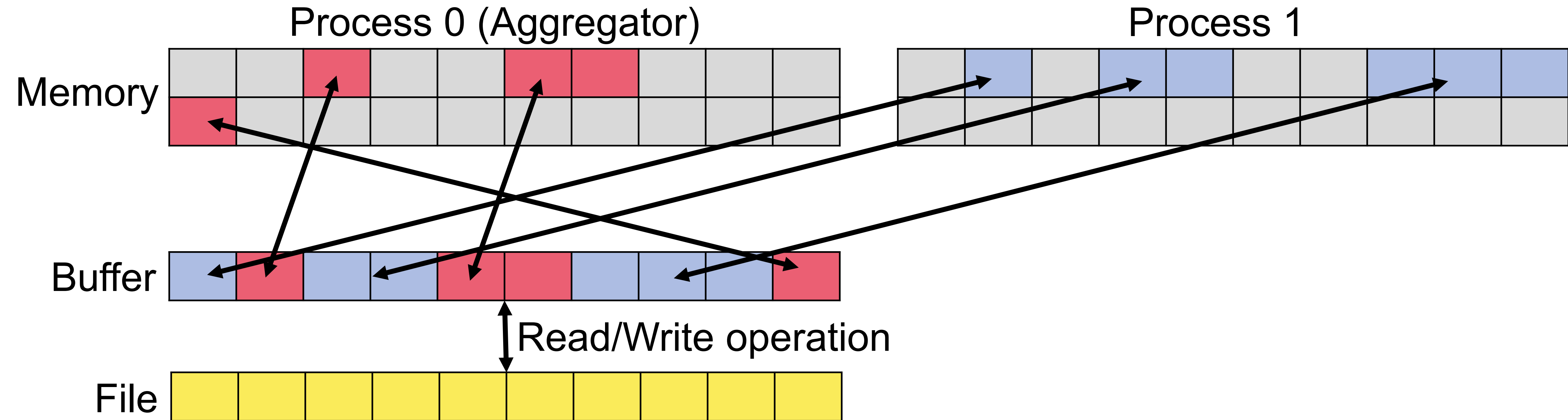


JÜLICH
Forschungszentrum

Collective buffering

Benefits of collective file access

- Aggregate data before writing/reading



JÜLICH
Forschungszentrum

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

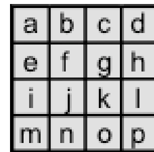
```
var[start,start+count] = data
```


JÜLICH Forschungszentrum

Writing variables

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

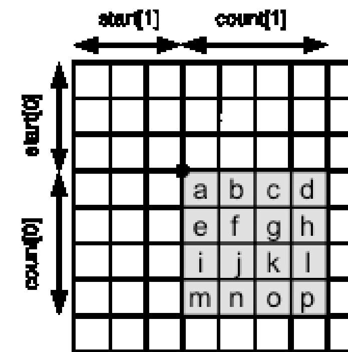
buf in memory can be in any shape,
but must be of size count[0]*count[1]



or

or . . .

the array variable defined
in the file is a 2D array



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)
```

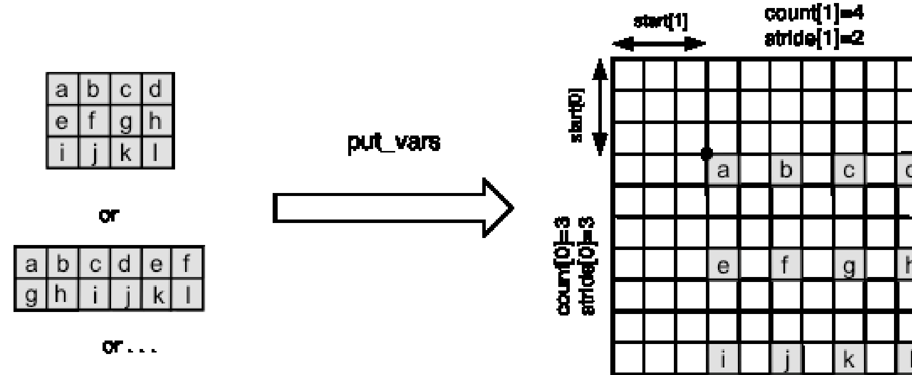
Fortran

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

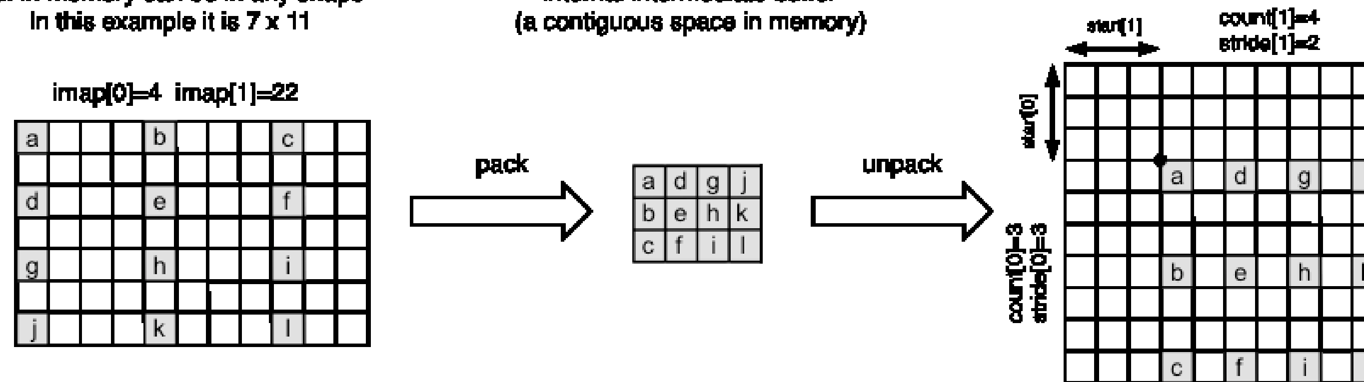
- Reads a *slab* of data of the file referenced by `ncid`
- Slab is defined by *n*-dimensional arrays `start` and `count`

Additional access types

- Subsampled array of values (`vars`)



- Mapped array (



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

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

- Query number of dimensions

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

- Query number of variables

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

- Query number of attributes

Fortran
```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

HDF5

What is HDF5?

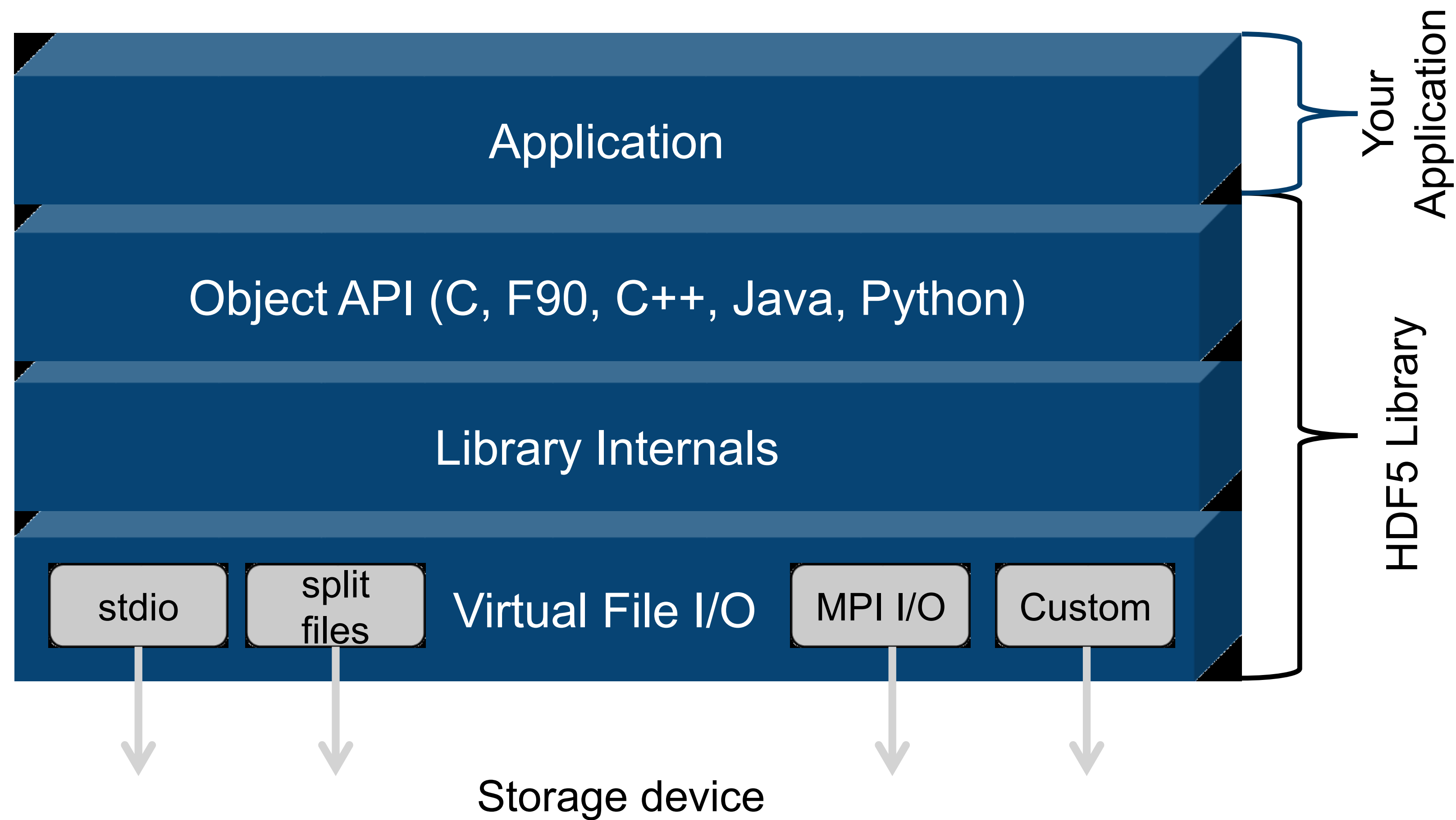
Hierarchical Data Format

- 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, Python and Java
- Provides tools to operate on HDF5 files and data

Layers of the HDF5 Library

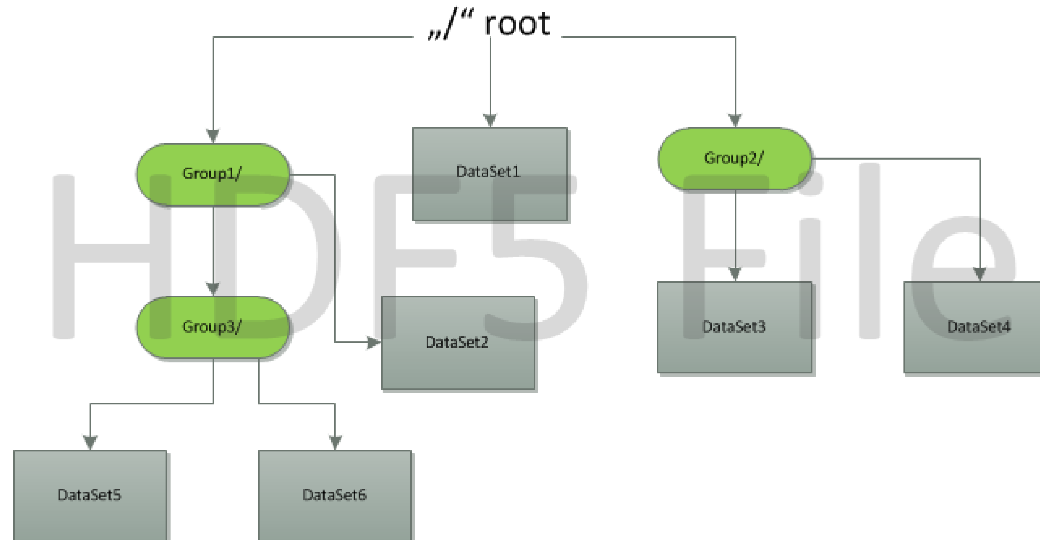


File organization

- 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
```

Fortran

```
use hdf5
INTEGER(HID_T)      Object identifier
INTEGER(HSIZE_T)    Used for 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

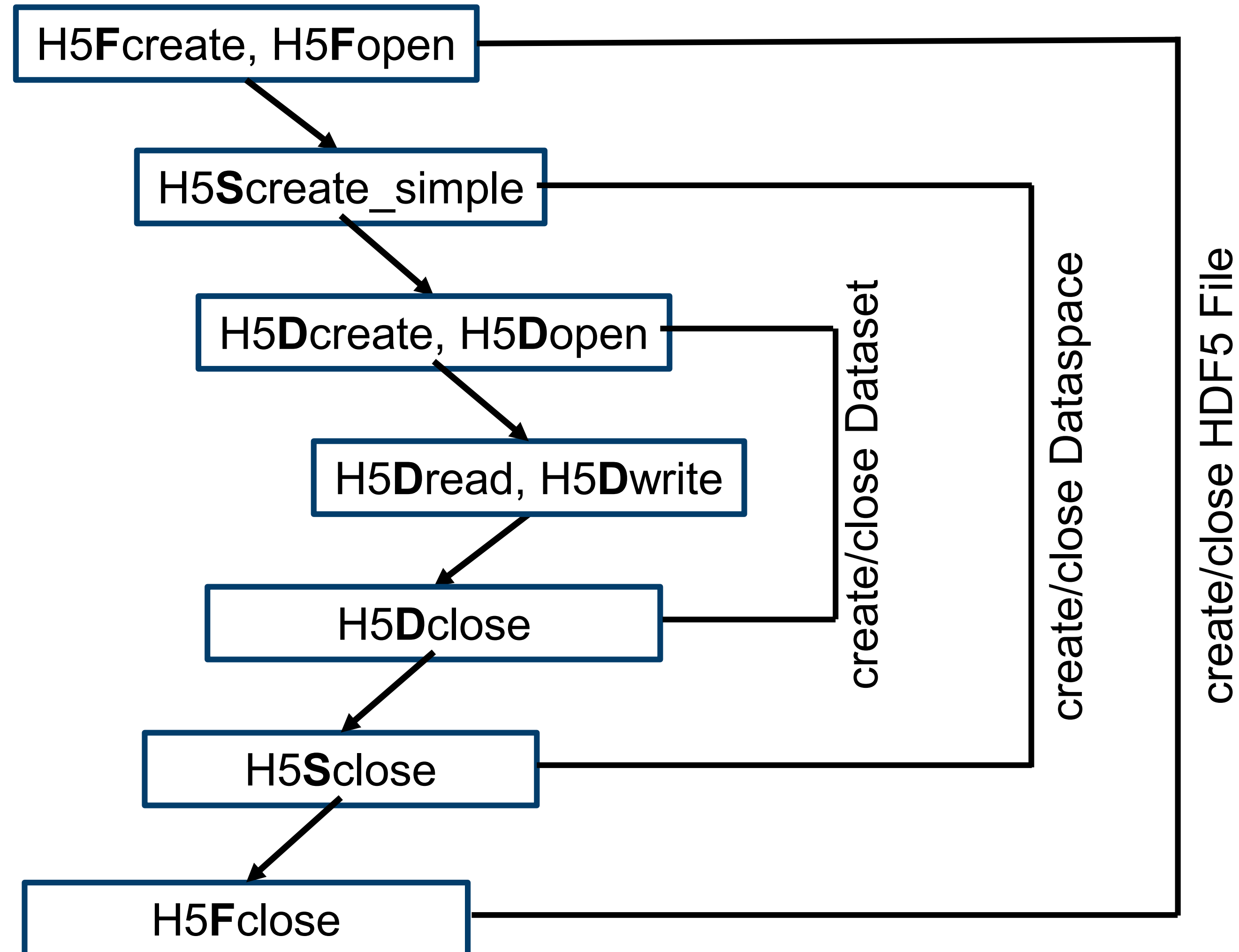
```
H5OPEN_F(STATUS)
   INTEGER, INTENT(OUT) :: STATUS

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

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

Fortran

```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 purge #to clean up the old environment
module load intel/18 intel-mpi/2018 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

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

Fortran

```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

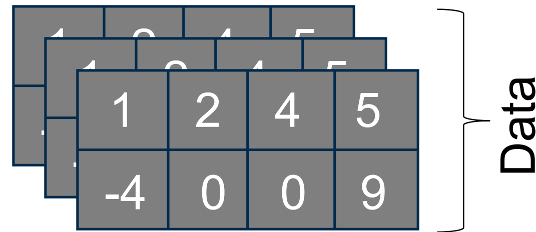
```
herr_t H5Awrite(hid_t attr_id, hid_t mem_type_id,
                const void *buf)
```

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

<u>Dataspace</u>
- rank = 3
- dim[0] = 2
- dim[1] = 4
- dim[2] = 3

<u>Datatype</u>
- Integer

<u>Attributes</u>
- Time = 2.1
- Temp = 122

<u>Storage</u>
- Contiguous

Creating a Dataset

C

```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

```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

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
```

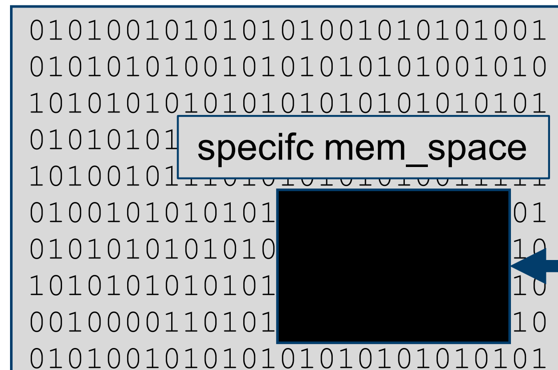
- `H5S_ALL[_F]` can be used to specify no special `mem_space` or `file_space` identifier
- `xfer_plist_id/xfer_prp` is a transfer property (e.g. to specify collective or independent parallel I/O)

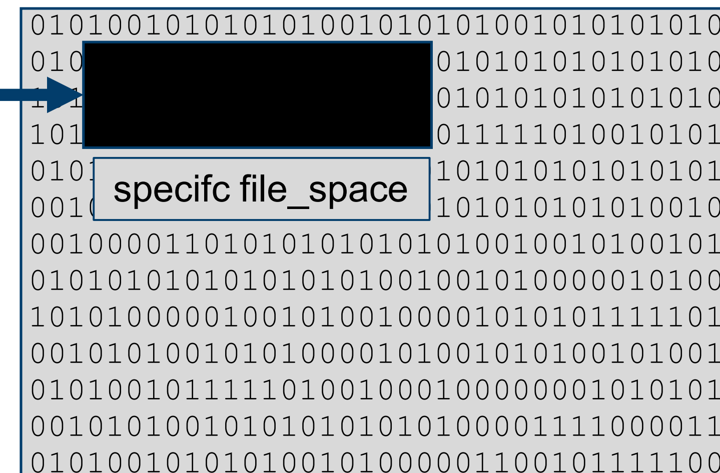
Writing to a dataset

`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

Global default filespace (given during dataset creation, ALL selection)

Global memory space (ALL selection)





Open a existing dataset

C

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

Fortran

```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`

JÜLICH Forschungszentrum

Excursion: row-major / column-major order

- “Logical” data view:
  - M[i,j] = $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$

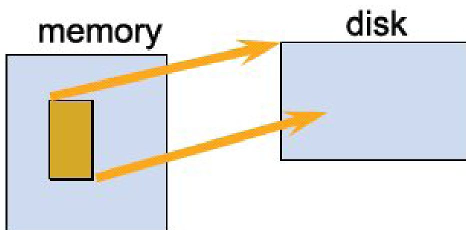
Adress	1	2	3	4	5	6
Value C	1	2	3	4	5	6
Value Fortran	1	3	5	2	4	6

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

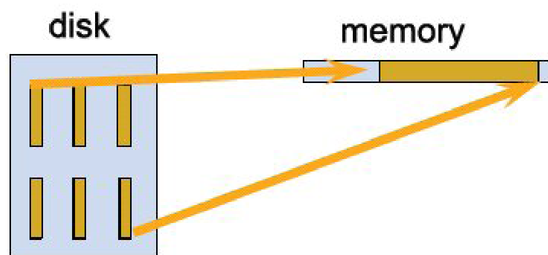
  - C: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$ Fortran: $\begin{bmatrix} 1 & 3 \\ 5 & 2 \\ 4 & 6 \end{bmatrix}$

- Storing data in a `2x3` dimensional dataset:
  - Fortran: $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$

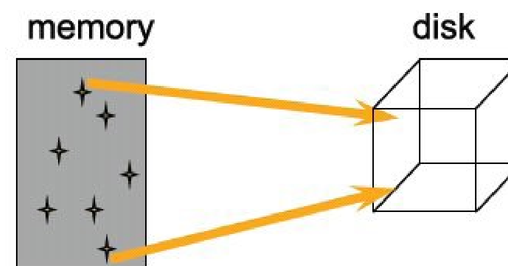
Partial I/O - Hyperslabs



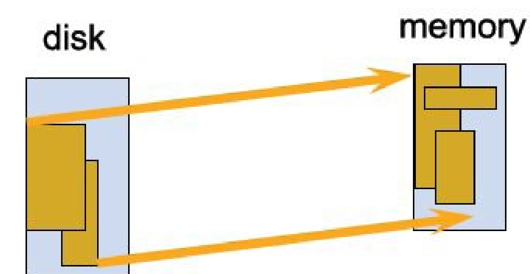
(a) Hyperslab from a 2D array to the corner of a smaller 2D array



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



(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.

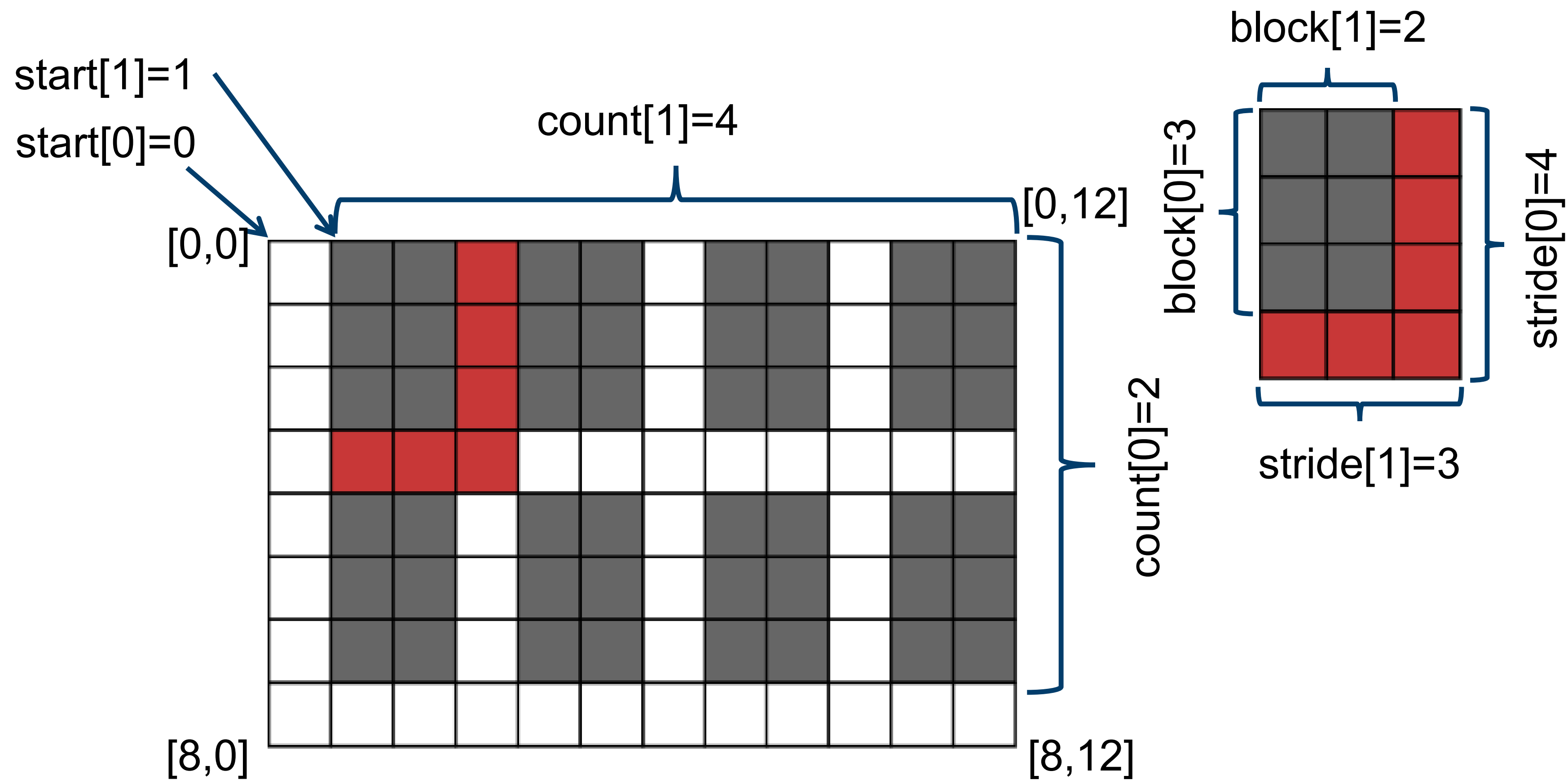
JÜLICH Forschungszentrum

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



JÜLICH
Forschungszentrum

Creating hyperslabs

C

```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

```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

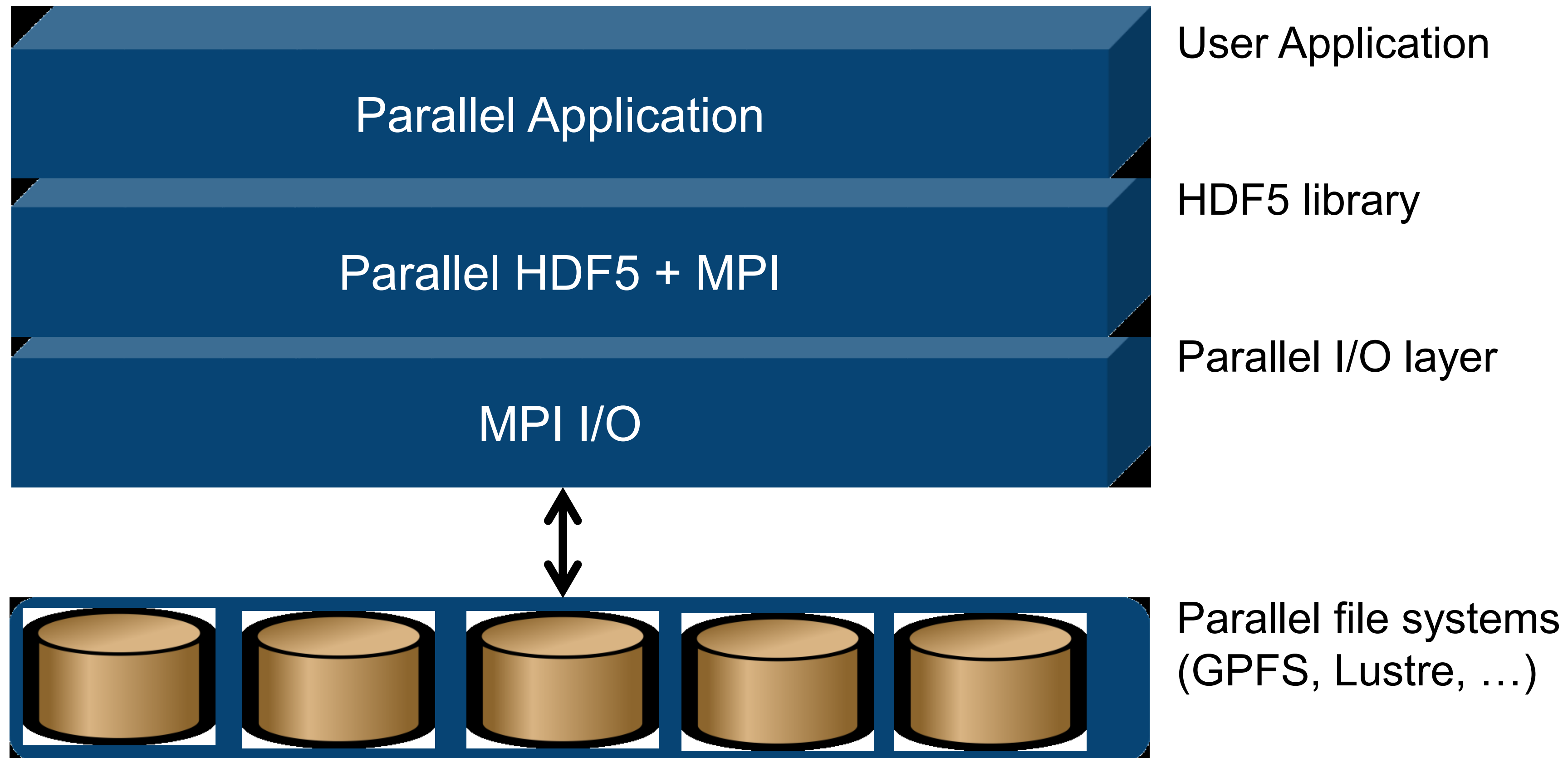
- The following operators (`op`) are supported to combine old and new selections:
- `H5S_SELECT_SET[_F]`: Replaces the existing selection with the parameters from this call. Overlapping blocks are not supported with this operator.
- `H5S_SELECT_OR[_F]`: Adds the new selection to the existing selection.
- `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.

PARALLEL HDF5

VIENNA SCIENTIFIC CLUSTER

JÜLICH Forschungszentrum

Implementation layers



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

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

Fortran

```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

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  …
…
```

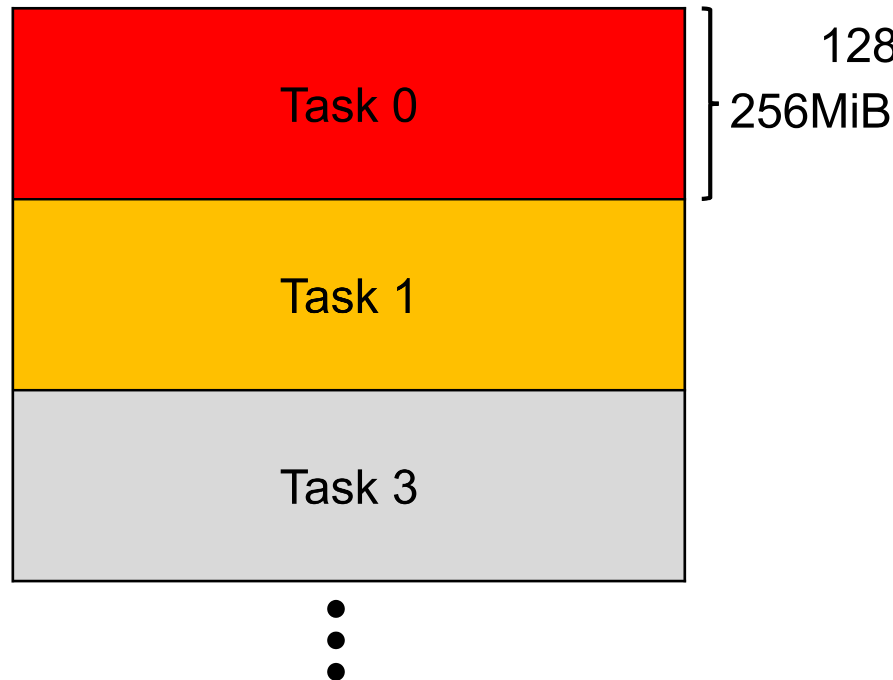
#cores x 5

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

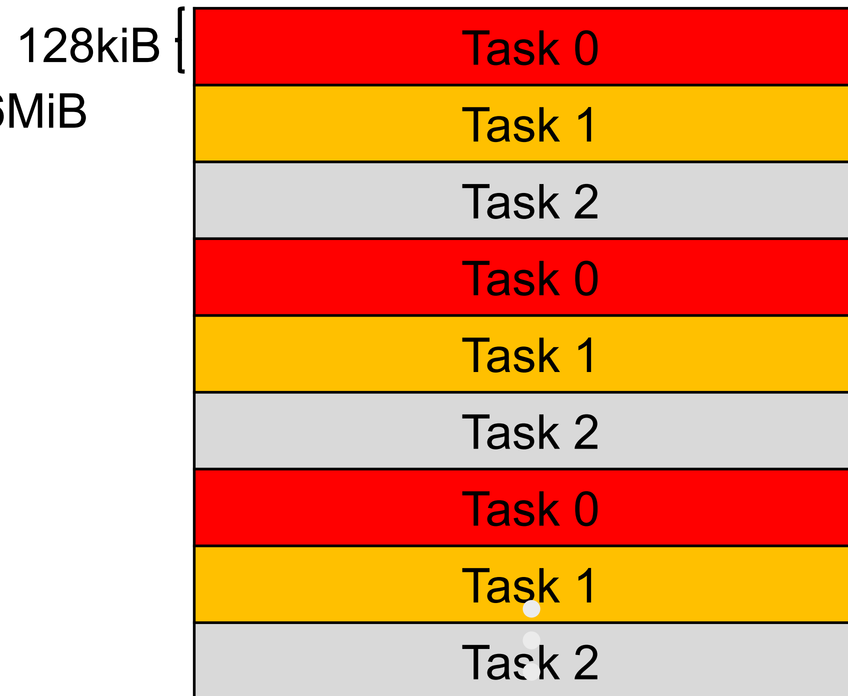
OPTIMIZATION AND PROFILING

I/O patterns

continuous
- Large continuous data blocks for each individual process



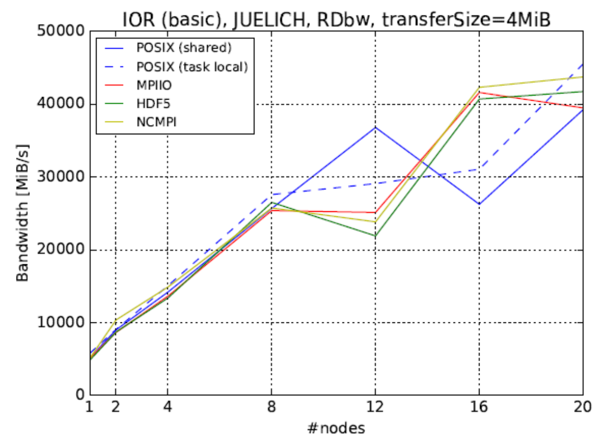
striped
- Pattern often found while handling multi dimensional arrays

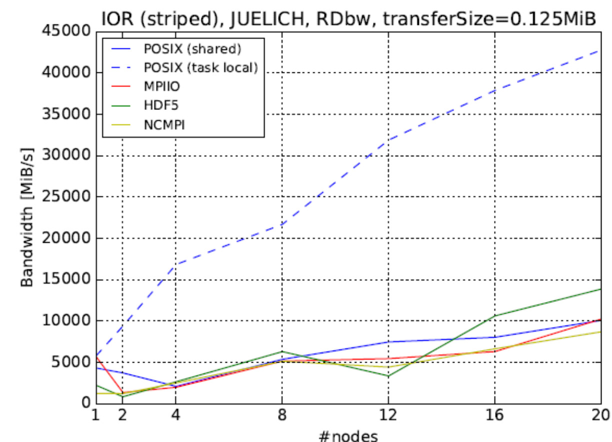


I/O pattern bandwidth

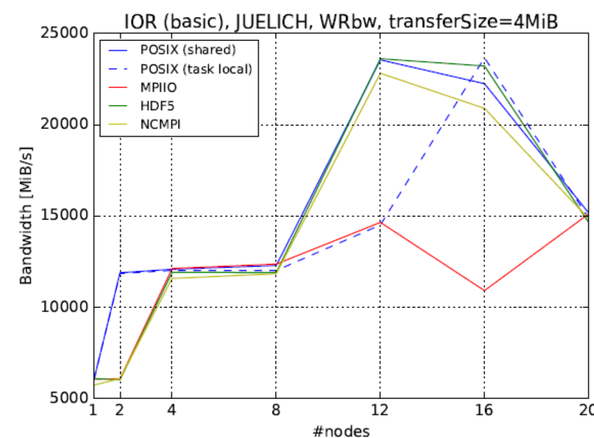
continuous | **striped**

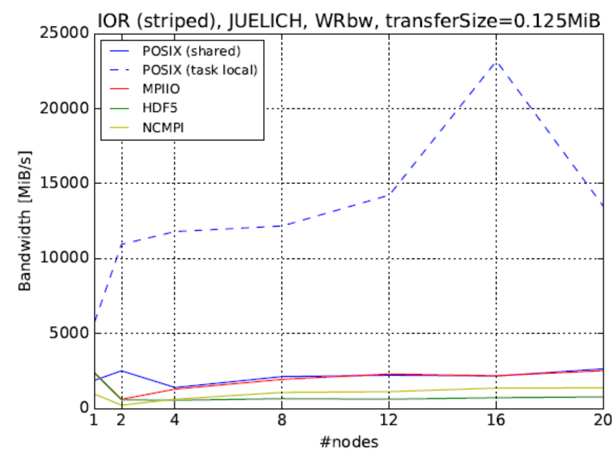
read bandwidth





write bandwidth





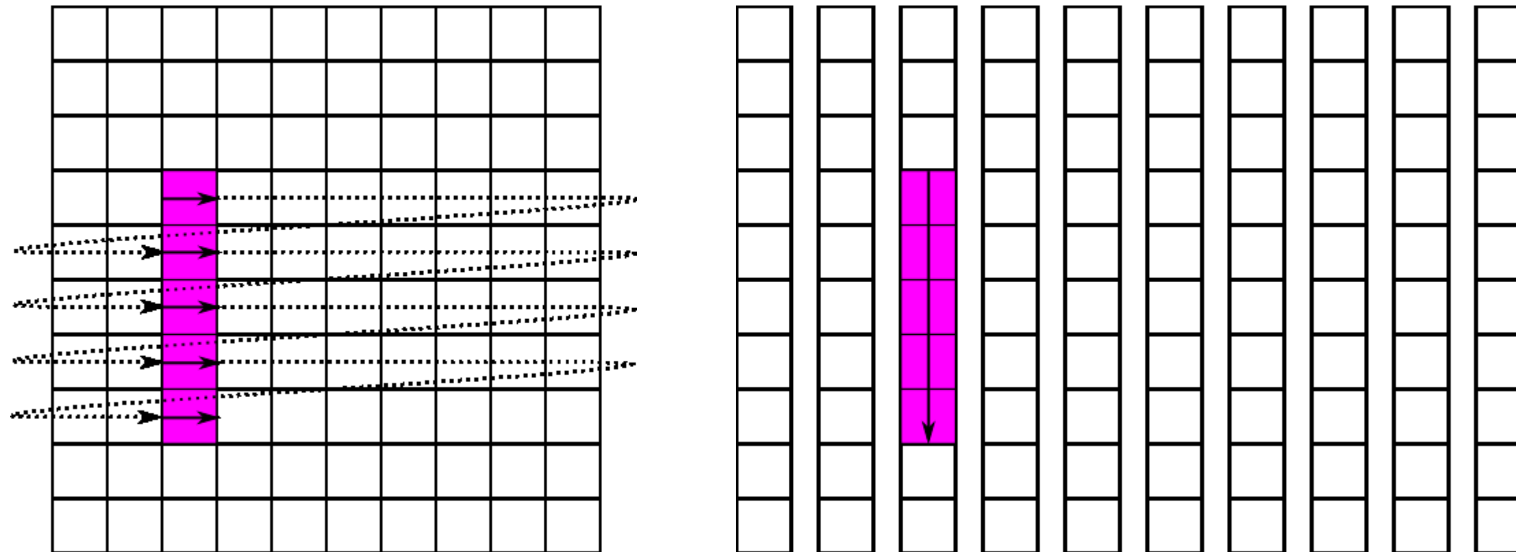
Measurements on JURECA at JSC

This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union.

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

Compression

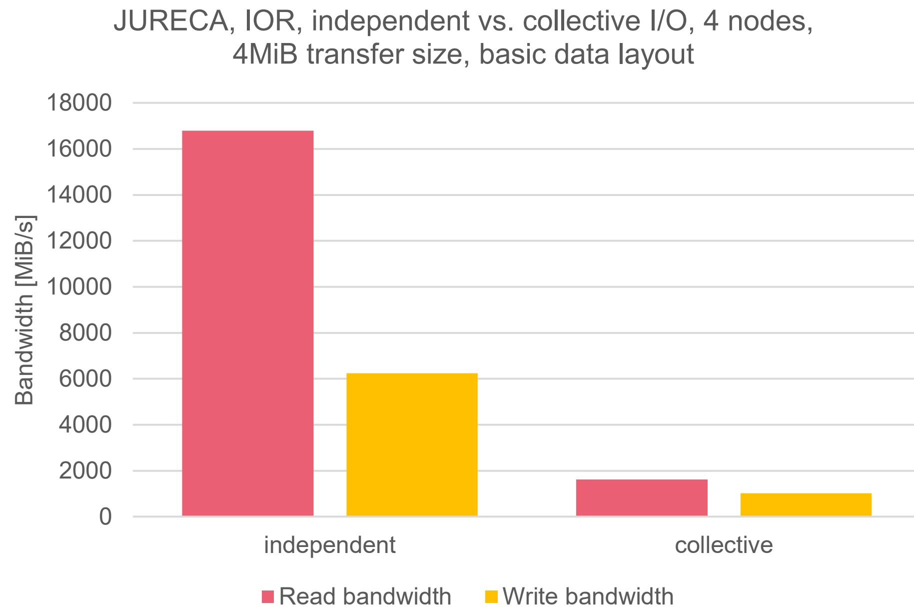
- In-transit compression can help to lower the overall datasize:
- HDF5 allows compression within a parallel, collective write commands for chunked datasets
- NetCDF4 only allows compression within serial programs (so far)
- Gzip (`deflate`) compression available by default (szip can be added on demand)
- Other compression techniques are available by using filters and external plugins: https://support.hdfgroup.org/services/filters.html
- ZFP compression example:

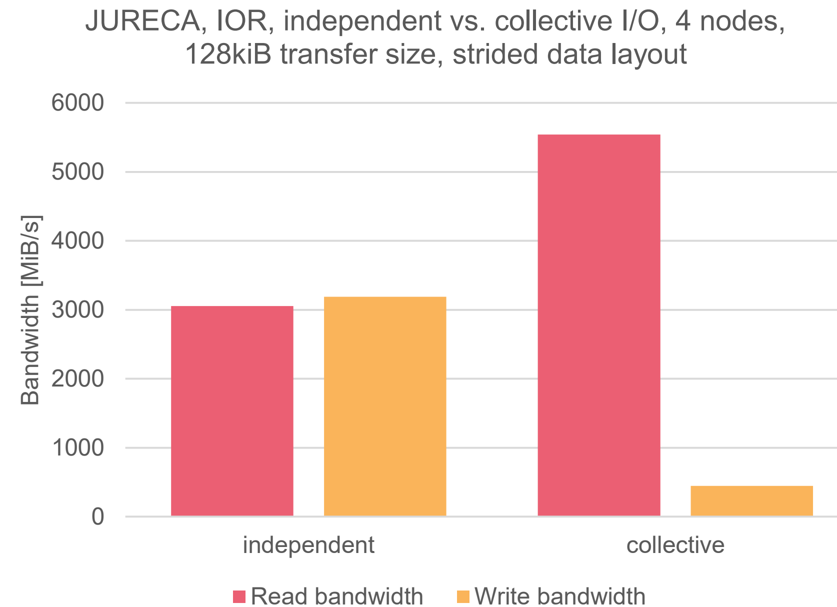
C

```
H5Pset_zfp_reversible_cdata(cd_nelmts, cd_values);
nc_def_var_filter(nc_file_id,nc_variable,H5Z_FILTER_ZFP,
                  cd_nelmts,cd_values);
```

Collective buffering

- Collective I/O operations not always speed up the general I/O, as more data might be processed than needed





	access size [Byte]	count
MPI-IO	4,194,304	184,320
POSIX	16,777,216	264,574

This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union.

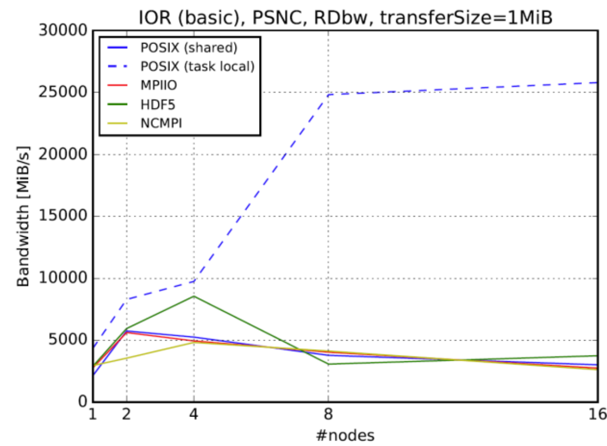
MPI-IO hints

- romio_cb_read: Enable collective buffering (reading)
- romio_cb_write: Enable collective buffering (writing)
- cb_buffer_size: Collective buffering, buffer size
- cb_nodes: Aggregator nodes
- romio_ds_read: Enable data sieving (reading)
- romio_ds_write: Enable collective buffering (writing)

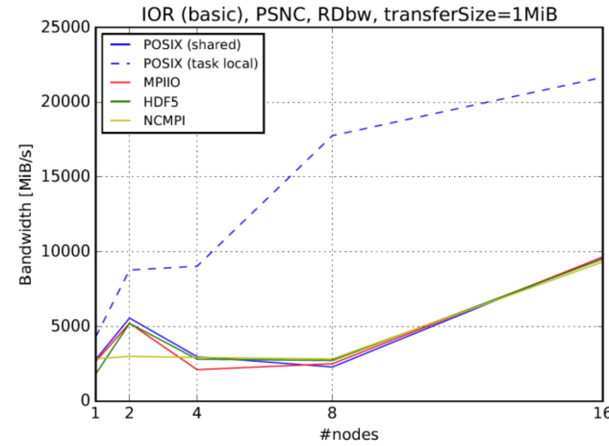
```
export ROMIO_HINTS=romio_hints_file
```

Filesystem specific options

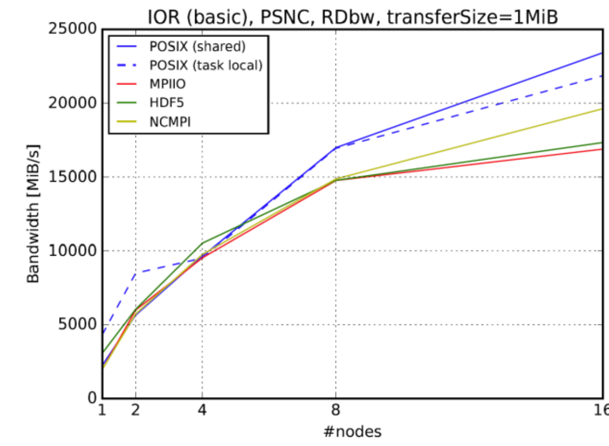
- On Lustre filesystems the user can influence the striping size and the number of involved object storage targets



Default number of OSTs (12) and default strip-size setting (1MiB)



Increased number of OSTs (126)



Increased stripe size to align with the individual amount of data per process (256MiB)

Measurements on Eagle at PSNC

This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union.

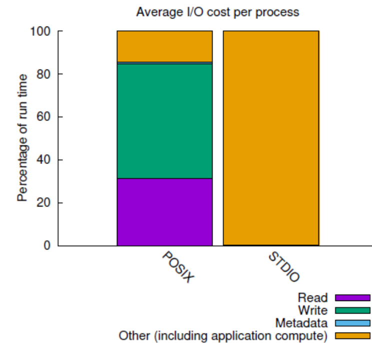
Profiling with Darshan

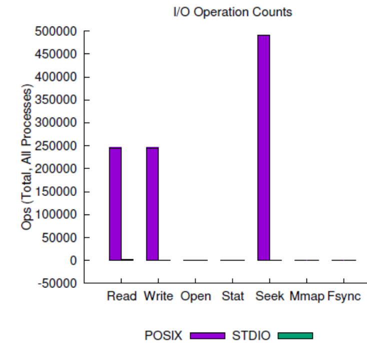
- I/O profiling tool for parallel applications
  - http://www.mcs.anl.gov/research/projects/darshan/
- Integration by using LD_PRELOAD:
  - `LD_PRELOAD=.../lib/libdarshan.so`

ior (3/9/2018) 1 of 3

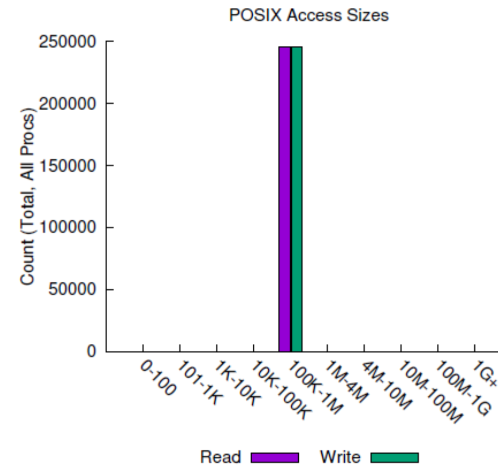
jobid: 4941235	uid: 11901	nprocs: 48	runtime: 10 seconds

I/O performance *estimate* (at the POSIX layer): transferred 37431 MiB at 6692.22 MiB/s
I/O performance *estimate* (at the STDIO layer): transferred 0.0 MiB at 5.27 MiB/s





Profiling with Darshan



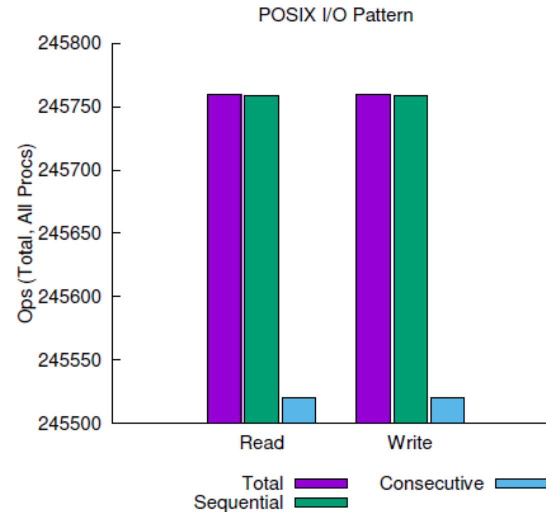
Most Common Access Sizes
(POSIX or MPI-IO)

	access size	count
POSIX	131072	491520

File Count Summary
(estimated by POSIX I/O access offsets)

type	number of files	avg. size	max size
total opened	4	7.6G	30G
read-only files	1	711	711
write-only files	2	1.7K	3.2K
read/write files	1	30G	30G
created files	3	11G	30G

Profiling with Darshan



sequential: An I/O op issued at an offset greater than where the previous I/O op ended.
consecutive: An I/O op issued at the offset immediately following the end of the previous I/O op.

Variance in Shared Files (POSIX and STDIO)

File Suffix	Processes	Fastest			Slowest			σ	
		Rank	Time	Bytes	Rank	Time	Bytes	Time	Bytes
...ehrs/IOR/2_1	48	35	7.507493	1.3G	33	9.180811	1.3G	0.397	0
...or_input.cfg	48	32	0.003404	711	2	0.006366	711	0	0
...<STDOUT>	48	1	0.000000	0	0	0.000392	3.2K	0	455
...<STDERR>	48	1	0.000000	0	0	0.000014	119	0	17

Benchmarking

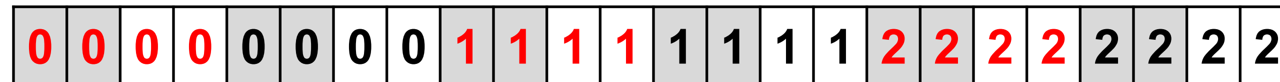
h5perf

- 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`)

Benchmarking

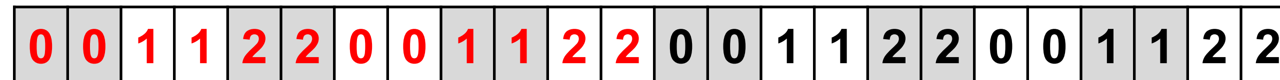
Example (1D):

- `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

Benchmarking

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 8 – Benchmarking

- Use `h5perf` to measure I/O performance

```
HDF5_PARAPREFIX=$GLOBAL srun -N 2 \
--ntasks-per-node=16 --reservation=training \
--time=00:05:00 \
h5perf --min-num-processes=32 -x 4M -X 4M \
-e 128M -B 1M -i 3

You can also try:
-g 2d layout, data sizes must be reduced
-I Interleaved layout
-c Enable HDF5 chunking
-C Enable collective operations
```