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, 23rd November 2018
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
**Member of the Helmholtz-Association**

23.11.2018

**VSC Training Course, Parallel I/O and Portable Data Formats**

## IO-500

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband</td>
<td>2,397,824</td>
<td>143,500.0</td>
<td>200,794.9</td>
<td>9,783</td>
</tr>
<tr>
<td>2</td>
<td>Sierra - IBM Power System S922LC, IBM POWER9 22C 3.16GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband</td>
<td>1,572,480</td>
<td>84</td>
<td>205,064</td>
<td>9,783</td>
</tr>
<tr>
<td>3</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C</td>
<td>10,649,600</td>
<td>93</td>
<td>205,064</td>
<td>9,783</td>
</tr>
<tr>
<td>4</td>
<td>Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUPT</td>
<td>4,981,760</td>
<td>61</td>
<td>205,064</td>
<td>9,783</td>
</tr>
<tr>
<td>5</td>
<td>Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100, Cray Inc.</td>
<td>387,872</td>
<td>21</td>
<td>205,064</td>
<td>9,783</td>
</tr>
<tr>
<td>6</td>
<td>Trinity - Cray XC40, Xeon E5-2690v3 16C 2.3GHz, Intel Xeon Phi 7250 8C 1.4GHz, Aries interconnect, Cray Inc.</td>
<td>979,072</td>
<td>20</td>
<td>205,064</td>
<td>9,783</td>
</tr>
<tr>
<td>7</td>
<td>AI Bridging Cloud Infrastructure (ABCII) - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR, Fujitsu</td>
<td>391,680</td>
<td>19</td>
<td>205,064</td>
<td>9,783</td>
</tr>
<tr>
<td>8</td>
<td>SuperMUC-N0 - ThinkSystem SD530, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path, Lenovo</td>
<td>305,856</td>
<td>19,476.6</td>
<td>26,873.9</td>
<td>6,783</td>
</tr>
</tbody>
</table>
Timetable

- 09:00 - 09:30 Parallel I/O strategies
- 09:30 - 11:00 Parallel NetCDF (PnetCDF/NetCDF4)
- 11:00 - 11:30 Coffee break
- 11:30 - 12:30 Parallel NetCDF (PnetCDF/NetCDF4)
- 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

Vienna, Austria, 23rd November 2018
One process performs I/O

file system

processes

P00
P01
P02
P03
P04
P05
P06
P07
P08
P09
P10
P11
P12
P13
P14
P15
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

Example: Creating files in parallel in the same directory

The creation of 2,097,152 files costs 113,595 core hours on JUQUEEN!

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

file system

processes

P00  P01  P02  P03
P04  P05  P06  P07
P08  P09  P10  P11
P12  P13  P14  P15
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

- 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

Pitfall 4

![Diagram showing file i-node, indirect blocks, and FS blocks.](image)
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

- Endianness (byte order) of binary data

<table>
<thead>
<tr>
<th>Address</th>
<th>Little Endian</th>
<th>Big Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>11010100</td>
<td>10100001</td>
</tr>
<tr>
<td>1001</td>
<td>11000011</td>
<td>10110010</td>
</tr>
<tr>
<td>1002</td>
<td>10110010</td>
<td>11000011</td>
</tr>
<tr>
<td>1003</td>
<td>10100001</td>
<td>11010100</td>
</tr>
</tbody>
</table>

- Conversion of files might be necessary and expensive

2,712,847,316 = 10100001 10110010 11000011 11010100
Portability

- Memory order depends on programming language

<table>
<thead>
<tr>
<th>Address</th>
<th>row-major order (e.g. C/C++)</th>
<th>column-major order (e.g. Fortran)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1002</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1003</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1004</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- 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

- P-HDF5
- NetCDF-4
- PNetCDF
- MPI-I/O
- POSIX I/O
- Parallel file system
- Shared file
- SIONlib
- Task-local files

Parallel application

Data stored in global view in local view
Parallel NetCDF
PnetCDF and NetCDF4

Vienna, Austria, 23rd November 2018
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/](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)

PnetCDF ≠ NetCDF4

Focus of this presentation, major differences towards NetCDF4 will be highlighted by [INSERT GRAPHIC]
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

<table>
<thead>
<tr>
<th><strong>Dimension</strong></th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable</strong></td>
<td>An entity that stores the bulk of the data. It represents an n-dimensional array of values of the same type.</td>
</tr>
<tr>
<td><strong>Attribute</strong></td>
<td>An entity to store data on the datasets contained in the file or the file itself. The latter are called <em>global attributes</em>.</td>
</tr>
</tbody>
</table>
NetCDF Classic model

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.

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.

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.

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC_BYTE</td>
<td>NF[90]_BYTE</td>
<td>8-bit signed integer</td>
</tr>
<tr>
<td>NC_CHAR</td>
<td>NF[90]_CHAR</td>
<td>8-bit unsigned integer</td>
</tr>
<tr>
<td>NC_SHORT</td>
<td>NF[90]_SHORT</td>
<td>16-bit signed integer</td>
</tr>
<tr>
<td>NC_INT</td>
<td>NF[90]_INT</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>NC_FLOAT</td>
<td>NF[90]_FLOAT</td>
<td>32-bit floating point</td>
</tr>
<tr>
<td>NC_DOUBLE</td>
<td>NF[90]_DOUBLE</td>
<td>64-bit floating point</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Limitations</th>
<th>classic</th>
<th>64bit off.</th>
<th>64bit data</th>
</tr>
</thead>
<tbody>
<tr>
<td>max. dim. size</td>
<td>( 2^{31} )</td>
<td>( 2^{32} )</td>
<td>( &gt; 2^{32} )</td>
</tr>
<tr>
<td>max. num. elem.</td>
<td>( 2^{32} )</td>
<td>( 2^{32} )</td>
<td>( &gt; 2^{32} )</td>
</tr>
<tr>
<td>max. var. size</td>
<td>2GiB</td>
<td>4GiB</td>
<td>( &gt; 4)GiB</td>
</tr>
</tbody>
</table>
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**
```c
#include <pnetcdf.h>
```

**Fortran**
```fortran
use pnetcdf
or
include 'pnetcdf.inc'
```

- Contain definition of
  - constants
  - functions

**NETCDF4:**
```c
#include <netcdf_par.h>
#include <netcdf.h>
```
```fortran
use netcdf
or
include 'netcdf.inc'
```

**Python**
```python
import netCDF4
(version 1.3.1 and mpi4py needed!)
```
Creating a file

<table>
<thead>
<tr>
<th>Language</th>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td><code>ncmpi_create</code></td>
<td><code>MPI_Comm comm, const char* filename, int cmode, MPI_Info info, int* ncid</code></td>
<td>Call is collective over <code>comm</code>&lt;br&gt; <code>ncid</code> is the id of the internal file handle&lt;br&gt; <code>cmode</code> must specify at least one of the following&lt;br&gt; - <code>(NC/NF)_CLOBBER</code> – Create new file and overwrite, if it existed before&lt;br&gt; - <code>(NC/NF)_NOCLOBBER</code> – Create new file only, if it did not exist before&lt;br&gt; Choose file format on file creation&lt;br&gt; - default - classic format&lt;br&gt; - <code>(NC/NF)_64BIT_OFFSET</code> - 64-bit offset format&lt;br&gt; - <code>(NC/NF)_64BIT_DATA</code> - 64-bit data format</td>
</tr>
<tr>
<td>Fortran</td>
<td><code>NFMPI_CREATE</code></td>
<td><code>COMM, FILENAME, CMODE, INFO, NCID</code></td>
<td>CHARACTER*(*) <code>FILENAME</code>&lt;br&gt; INTEGER <code>COMM, MODE, INFO, NCID</code>&lt;br&gt; <code>NF90MPI_...</code> can be used when using F90</td>
</tr>
</tbody>
</table>
### Creating a file NETCDF4

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

- **Fortran**
  ```fortran
  INTEGER NF_CREATE_PAR(FILENAME, CMODE, COMM, INFO, NCID)
  CHARACTER(*) FILENAME
  INTEGER COMM, MODE, INFO, NCID
  ```

  - 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)

- `NF90_...` can be used when using F90
- `comm%MPI_VAL` can be used for `mpi_f08`
Creating a file **NETCDF4**

```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

- 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

```c
int ncmapi_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
```

**NETCDF4:**
```
[nc, nf]_open_par(filename, omode, comm, info, ncid)
```
Closing a file

- Close file associated with `ncid`

**C**
```c
int ncmpi_close(int ncid)
```

**Fortran**
```fortran
INTEGER NFMPI_CLOSE(NCID)
INTEGER NCID
```

**Python**
```python
nc.close()
```

**NETCDF4:**
```python
[nc, nf]_close(ncid)
```
Error handling

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

C

```c
const char *ncmpi_strerror(int status)
```

Fortran

```fortran
CHARACTER*80 NFMPI_STRERROR(STATUS)
INTEGER STATUS
```
Exercise

- 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 -lnetcdf
Defining dimensions

- **C**
  
  ```c
  int ncmpi_def_dim(int ncid, const char* name, MPI_Offset len, int* dimid)
  ```

- **Fortran**
  
  ```fortran
  INTEGER NFMPI_DEF_DIM(NCID, NAME, LEN, DIMID)
  CHARACTER(*) NAME
  INTEGER NCID, DIMID
  INTEGER(KIND=MPI_OFFSET_KIND) LEN
  ```

- **Python**
  
  ```python
  dim = nc.createDimension('name',size)
  ```

- **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

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

```
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

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

- 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`
Defining attributes

Python  
`element.attribute_name = value`
Reading attributes

- **C**
  ```c
  int ncmpi_get_att_<type>\n  (int ncid, int varid,\n   const char* name, <type>*attr)
  ```

- **Fortran**
  ```fortran
  INTEGER NFMPI_GET_ATT_<type>(NCID, VARID, NAME,\n                             ATTR)
  ```

- C:
  ```
  <type> ATTR
  CHARACTER*(*) NAME
  INTEGER NCID, VARID
  ```

F90:
  ```
  NF90MPI_GET_ATT(...)\n  ```

- **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

- Ends the definition phase, and switches to collective data mode
- Once variables have been put into the data set, definitions must not be altered

C

```c
int nc mpi_enddef (int ncid)
```

Fortran

```
INTEGER NFMP Ê Ê NDDEF (NCID)
INTEGER NCID
```

Not explicitly needed for NETCDF4 file format
Exercise

- Extend your existing parallel application (C or Fortran)
- Create a dimension of size $100 \times \text{NUMBER\_OF\_PROCS}$
- 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`

netCDF parallel-netcdf
Switching data modes

- Switches from collective data mode to individual data mode

```
C int ncmpid_begin_indep_data(int ncid)

Fortran INTEGER NFMPI_BEGIN_INDEP_DATA(NCID)
INTEGER NCID
```

- Switches from individual data mode to collective data mode

```
C int ncmpid_end_indep_data(int ncid)

Fortran INTEGER NFMPI_END_INDEP_DATA(NCID)
INTEGER NCID
```

NETCDF4: `nc_var_par_access(ncid, varid, [NC_INDEPENDENT, NC_COLLECTIVE])`

Python `var.set_collective(True)`
# Writing variables (collectively)

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int ncmpi_put_vara_&lt;type&gt;[_all]</strong>(int ncid, int varid,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>const MPI_Offset start[],</td>
<td></td>
</tr>
<tr>
<td>const MPI_Offset count[],</td>
<td></td>
</tr>
<tr>
<td>const &lt;type&gt;* var)</td>
<td></td>
</tr>
<tr>
<td><strong>INTEGER</strong></td>
<td><strong>INTEGER</strong></td>
</tr>
<tr>
<td><strong>NFMPI_PUT_VARA_&lt;type&gt;[_ALL]</strong>(NCID, VARID,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>START, COUNT, VAR)</td>
<td></td>
</tr>
<tr>
<td>&lt;type&gt;(*) VAR</td>
<td></td>
</tr>
<tr>
<td>INTEGER NCID, VARID</td>
<td>INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT</td>
</tr>
<tr>
<td>NETCDF4: _all not needed</td>
<td></td>
</tr>
</tbody>
</table>

- 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

F90:  
```fortran
nf90mpi_put_var[_all](ncid,varid,var,start?,count?,stride?,imap?)
```
Writing variables (collectively)

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

---

Reading variables (collectively)

- 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

### C

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

### Fortran

```fortran
INTEGER NFMI_GET_VARA_<type>[_ALL](NCID, VARID,
START, COUNT, VAR)

<type>(*) VAR
INTEGER NCID, VARID
INTEGER(KIND=MPI_OFFSET_KIND) START, COUNT
```

**NETCDF4:** `_all` not needed
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

- Query number of dimensions

```
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 variables

```
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 attributes

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

```
Fortran
INTEGER NFMPI_INQ_NATTS(NCID, NATTS)
    INTEGER NCID, NATTS
```

Fortran90: `nf90mpi_inquire_variable`
## Inquiry of ids of data set entities

**C**

```c
int ncmci_inq_varid(int ncid, const char* name, int* varid)
```

**Fortran**

```fortran
INTEGER NFMPI_INQ_VARID(NCID, NAME, VARID)
 INTEGER NCID, VARID
 CHARACTER(*) NAME
```

- Query id of variable given by `name`

**C**

```c
int ncmci_inq_vardimid(int ncid, int varid, int* dimids)
```

**Fortran**

```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

- Reads `len` from a given dimension

**C**

```c
int ncmpi_inq_dimlen(int ncid, int dimid,
                       MPI_Offset* len)
```

**Fortran**

```fortran
INTEGER NFMPI_INQ_DIMLEN(NCID, DIMID, LEN)
INTEGER NCID, DIMID
INTEGER(KIND=MPI_OFFSET_KIND) LEN
```

**Fortran90**

```fortran
nf90mpi_inquire_dimension
```
Exercise

- 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
```

netCDF  parallel-netcdf
Non-blocking interface

- Can aggregate multiple smaller requests into larger ones for better I/O performance

C

```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

```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

```c
int ncmpl_wait(int ncid, int num_reqs,
    int req_ids[], int statuses[])

int ncmpl_wait_all(int ncid, int num_reqs,
    int req_ids[], int statuses[])
```

Fortran

```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

```c
int ncmpl_buffer_attach(int ncid, int bufsize)

int ncmpl_buffer_detach(int ncid)
```
Additional access types

- **Subsampled array of values** *(vars)*
  - buf in memory can be in any shape, but must be of size \(\text{count}[0]*\text{count}[1]\)
  - the array variable defined in the file is a 2D array

  ![Subsampled array diagram]

- **Mapped array of values** *(varm)*
  - buf in memory can be in any shape
    - In this example it is 7 x 11
  - internal intermediate buffer
    - (a contiguous space in memory)

  ![Mapped array diagram]


F90: not needed
## API matrix

<table>
<thead>
<tr>
<th></th>
<th>blocking</th>
<th>non-blocking</th>
<th>buffered</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>single data value</strong></td>
<td>put_var1</td>
<td>iput_var1</td>
<td>bput_var1 (since 1.3.0)</td>
</tr>
<tr>
<td></td>
<td>get_var1</td>
<td>iget_var1</td>
<td>bget_var1 (since 1.3.0)</td>
</tr>
<tr>
<td><strong>array of values</strong></td>
<td>put_vara</td>
<td>iput_vara</td>
<td>bput_vara (since 1.3.0)</td>
</tr>
<tr>
<td></td>
<td>get_vara</td>
<td>iget_vara</td>
<td>bget_vara (since 1.3.0)</td>
</tr>
<tr>
<td><strong>subsampled array of values</strong></td>
<td>put_vars</td>
<td>iput_vars</td>
<td>bput_vars (since 1.3.0)</td>
</tr>
<tr>
<td></td>
<td>get_vars</td>
<td>iget_vars</td>
<td>bget_vars (since 1.3.0)</td>
</tr>
<tr>
<td><strong>mapped array of values</strong></td>
<td>put_varm</td>
<td>iput_varm</td>
<td>bput_varm (since 1.3.0)</td>
</tr>
<tr>
<td></td>
<td>get_varm</td>
<td>iget_varm</td>
<td>bget_varm (since 1.3.0)</td>
</tr>
<tr>
<td><strong>list of subarrays of values</strong></td>
<td>put_varn</td>
<td>iput_varn</td>
<td>bput_varn (since 1.6.0)</td>
</tr>
<tr>
<td></td>
<td>get_varn</td>
<td>iget_varn</td>
<td>bget_varn (since 1.6.0)</td>
</tr>
</tbody>
</table>
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)
```
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?

- 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

- Application
- Object API (C, F90, C++, Java, Python)
- Library Internals
- Virtual File I/O
- MPI I/O
- Custom
- stdio
- split files

Storage device
File organization

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

<table>
<thead>
<tr>
<th>HDF5</th>
<th>Unix/Linux fs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group ↔ Directory</td>
<td></td>
</tr>
<tr>
<td>Data set ↔ File</td>
<td></td>
</tr>
</tbody>
</table>

/DataSet1
/Group1/DataSet2
/Group1/Group3/DataSet5
/Group1/Group3/DataSet6
/Group2/DataSet3
/Group2/Dataset4
## Terminology

<table>
<thead>
<tr>
<th><strong>File</strong></th>
<th>Container for storing data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Group</strong></td>
<td>Structure which may contain HDF5 objects, e.g. datasets, attributes, datasets</td>
</tr>
<tr>
<td><strong>Attribute</strong></td>
<td>Can be used to describe datasets and is attached to them</td>
</tr>
<tr>
<td><strong>Dataspace</strong></td>
<td>Describes the dimensionality of the data array and the shape of the data points respectively, i.e. it describes the shape of a dataset</td>
</tr>
<tr>
<td><strong>Dataset</strong></td>
<td>Multi-dimensional array of data elements</td>
</tr>
</tbody>
</table>
Library specific types

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

**C**

```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**

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

- `H5OPEN_F` function:
  ```fortran
  INTEGER, INTENT(OUT) :: STATUS
  ```

- `H5CLOSE_F` function:
  ```fortran
  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

H5Fcreate, H5Fopen

H5Screate_simple

H5Dcreate, H5Dopen

H5Dread, H5Dwrite

H5Dclose

H5Sclose

H5Fclose

create/close HDF5 File

create/close Dataspace
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)
```

- **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**

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

**Fortran**

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

- **Fortran**
  ```fortran
  H5GCCREATE_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

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

Fortran

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

- 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)

<table>
<thead>
<tr>
<th>C type</th>
<th>HDF5 file type (pre-defined)</th>
<th>HDF5 memory type (native)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>H5T_STD_I32[BE,LE]</td>
<td>H5T_NATIVE_INT</td>
</tr>
<tr>
<td>float</td>
<td>H5T_IEEE_F32[BE,LE]</td>
<td>H5T_NATIVE_FLOAT</td>
</tr>
<tr>
<td>double</td>
<td>H5T_IEEE_F64[BE,LE]</td>
<td>H5T_NATIVE_DOUBLE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fortran</th>
<th>HDF5 file type (pre-defined)</th>
<th>HDF5 memory type (native)</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>H5T_STD_I32[BE,LE]</td>
<td>H5T_NATIVE_INTEGER</td>
</tr>
<tr>
<td>real</td>
<td>H5T_IEEE_F32[BE,LE]</td>
<td>H5T_NATIVE_REAL</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Simple dataspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank: int</td>
</tr>
<tr>
<td>current_size: hsize_t[rank]</td>
</tr>
<tr>
<td>maximum_size: hsize_t[rank]</td>
</tr>
</tbody>
</table>

rank = 2, dimensions = 2x5
Creating a dataspace

\[ \text{hid_t H5Screate_simple}(\text{int rank}, \]
\[ \quad \text{const hsize_t *current_dims}, \]
\[ \quad \text{const hsize_t *maximum_dims}) \]

**Fortran**

\[ \text{H5SCREATE_SIMPLE_F}(\text{RANK, DIMS, SPACE_ID, HDFERR, MAXDIMS}) \]

- **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

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>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)</td>
<td>H5ACREATE_F(LOC_ID, NAME, TYPE_ID, SPACE_ID, ATTR_ID, HDFERR, ACPL_ID, AAPL_ID)</td>
</tr>
<tr>
<td></td>
<td>INTEGER(KIND=HID_T), INTENT(IN) :: LOC_ID</td>
</tr>
<tr>
<td></td>
<td>CHARACTER(LEN=*), INTENT(IN) :: NAME</td>
</tr>
<tr>
<td></td>
<td>INTEGER(KIND=HID_T), INTENT(IN) :: TYPE_ID, SPACE_ID</td>
</tr>
<tr>
<td></td>
<td>INTEGER(KIND=HID_T), INTENT(OUT) :: ATTR_ID</td>
</tr>
<tr>
<td></td>
<td>INTEGER, INTENT(OUT) :: HDFERR</td>
</tr>
<tr>
<td></td>
<td>INTEGER(KIND=HID_T), OPTIONAL, INTENT(IN) :: ACPL_ID, AAPL_ID</td>
</tr>
</tbody>
</table>

- `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

<table>
<thead>
<tr>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>herr_t <strong>H5Awrite</strong>(hid_t attr_id, hid_t mem_type_id, const void *buf)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fortran</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H5AWRITE_F</strong>(ATTR_ID, MEMTYPE_ID, BUF, DIMS, HDFERR)</td>
<td></td>
</tr>
<tr>
<td>INTEGER(KIND=HID_T), INTENT(IN) :: ATTR_ID</td>
<td></td>
</tr>
<tr>
<td>INTEGER(KIND=HID_T), INTENT(IN) :: MEMTYPE_ID</td>
<td></td>
</tr>
<tr>
<td>TYPE, INTENT(IN) :: BUF</td>
<td></td>
</tr>
<tr>
<td>INTEGER(KIND=HSIZE_T)(*), INTENT(IN) :: DIMS</td>
<td></td>
</tr>
<tr>
<td>INTEGER, INTENT(OUT) :: HDFERR</td>
<td></td>
</tr>
</tbody>
</table>

- **Fortran**: **DIMS** array to hold corresponding dimension sizes of data buffer **buf** (new since 1.4.2)
Dataset (metadata + data)

Data

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>

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

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

```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`
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

- `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

<table>
<thead>
<tr>
<th>mem_space_id</th>
<th>file_space_id</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>datasource id</td>
<td>datasource id</td>
<td>use dataspace as is</td>
</tr>
<tr>
<td>H5S_ALL</td>
<td>datasource id</td>
<td>use given file_space datasource also for mem_space datasource (including the selection)</td>
</tr>
<tr>
<td>datasource id</td>
<td>H5S_ALL</td>
<td>use all selection for default file_space</td>
</tr>
<tr>
<td>H5S_ALL</td>
<td>H5S_ALL</td>
<td>use default file_space also for mem_space, set all selection for both</td>
</tr>
</tbody>
</table>
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**

```c
hid_t H5Dget_space(hid_t dataset_id)
```

**Fortran**

```fortran
H5DGET_SPACE_F(DATASET_ID, DATASPACE_ID, 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

Python

```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

```fortran
H5DREAD_F(DSET_ID, MEM_TYPE_ID, BUF, DIMS, HDFERR, 
           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

- 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:

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

<table>
<thead>
<tr>
<th>Address</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value C</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Value Fortran</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Storing data in a $3 \times 2$ 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 $2 \times 3$ 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
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

start[0] = 0
start[1] = 1

[0,0] – [8,0]
[0,12] – [8,12]

count[0] = 2
count[1] = 4

block[0] = 3
block[1] = 2

stride[0] = 4
stride[1] = 3
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 (\texttt{op}) are supported to combine old and new selections:

- \texttt{H5S\_SELECT\_SET[\_F]}: Replaces the existing selection with the parameters from this call. Overlapping blocks are not supported with this operator.

- \texttt{H5S\_SELECT\_OR[\_F]}: Adds the new selection to the existing selection.

- \texttt{H5S\_SELECT\_AND[\_F]}: Retains only the overlapping portions of the new selection and the existing selection.

- \texttt{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.

- \texttt{H5S\_SELECT\_NOTB[\_F]}: Retains only elements of the existing selection that are not in the new selection.

- \texttt{H5S\_SELECT\_NOTA[\_F]}: Retains only elements of the new selection that are not in the existing selection.
Parallel HDF5

Vienna, Austria, 23rd November 2018
Implementation layers

- User Application
- HDF5 library
- Parallel I/O layer
- Parallel file systems (GPFS, Lustre, …)

- Parallel Application
- Parallel HDF5 + MPI
- MPI I/O
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

- \texttt{cls\_id/classtype} \textbf{must be} \texttt{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

\begin{Verbatim}
hid_t H5Pcreate(hid_t cls_id);
herr_t H5Pset_fapl_mpio(hid_t fapl_id, MPI_Comm comm, MPI_Info info)
\end{Verbatim}

\begin{Verbatim}
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
\end{Verbatim}
Dataset transfer property

- 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);
```

Exercise

- 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:
  
  \[
  \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  \vdots \\
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  \vdots 
  \end{array}
  \]

  \#cores x 5

- Write the data collectively into the file
- Check the resulting file using: \texttt{h5dump}
Optimization and Profiling

Vienna, Austria, 23rd November 2018
I/O patterns

continuous
• Large continuous data blocks for each individual process

striped
• Pattern often found while handling multi dimensional arrays

![Diagram showing continuous and striped I/O patterns](image-url)
I/O pattern bandwidth

**continuous**

**striped**

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.
Collective buffering

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

<table>
<thead>
<tr>
<th>Access Size</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI-IO</td>
<td>4,194,304</td>
</tr>
<tr>
<td>POSIX</td>
<td>16,777,216</td>
</tr>
</tbody>
</table>

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.
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
- Integration by using LD_PRELOAD:
  - LD_PRELOAD=.../lib/libdarshan.so
Profiling with Darshan

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

<table>
<thead>
<tr>
<th>type</th>
<th>number of files</th>
<th>avg. size</th>
<th>max size</th>
</tr>
</thead>
<tbody>
<tr>
<td>total opened</td>
<td>4</td>
<td>7.6G</td>
<td>30G</td>
</tr>
<tr>
<td>read-only files</td>
<td>1</td>
<td>711</td>
<td>711</td>
</tr>
<tr>
<td>write-only files</td>
<td>2</td>
<td>1.7K</td>
<td>3.2K</td>
</tr>
<tr>
<td>read/write files</td>
<td>1</td>
<td>30G</td>
<td>30G</td>
</tr>
<tr>
<td>created files</td>
<td>3</td>
<td>11G</td>
<td>30G</td>
</tr>
</tbody>
</table>

Most Common Access Sizes
(POSIX or MPI-IO)

<table>
<thead>
<tr>
<th>access size</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX</td>
<td>131072</td>
</tr>
</tbody>
</table>
Profiling with Darshan

**POSIX I/O Pattern**

*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)**

<table>
<thead>
<tr>
<th>File Suffix</th>
<th>Processes</th>
<th>Fastest</th>
<th>Slowest</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Rank</td>
<td>Time</td>
<td>Bytes</td>
</tr>
<tr>
<td>...ehrs/IOR/2_1</td>
<td>48</td>
<td>35</td>
<td>7.507493</td>
<td>1.3G</td>
</tr>
<tr>
<td>...or_input.cfg</td>
<td>48</td>
<td>32</td>
<td>0.003404</td>
<td>711</td>
</tr>
<tr>
<td>...&lt;STDOUT&gt;</td>
<td>48</td>
<td>1</td>
<td>0.000000</td>
<td>0</td>
</tr>
<tr>
<td>...&lt;STDERR&gt;</td>
<td>48</td>
<td>1</td>
<td>0.000000</td>
<td>0</td>
</tr>
</tbody>
</table>
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

0 0 0 0 0 0 0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
1 write operation per transfer

- interleaved

0 0 1 1 2 2 2 0 0 1 1 2 2 0 0 1 1 2 2
2 write operations per transfer

## Benchmarking

### Example (2D):
- num-processes = 2
- bytes-per-process = 4
- block-size = 2
- transfer-buffer-size = 8

<table>
<thead>
<tr>
<th>contiguously</th>
<th>interleaved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

1 write operation per transfer

8 write operations per transfer

Exercise

- Use \texttt{h5perf} to measure I/O performance

\begin{verbatim}
HDF5_PARAPREFIX=$GLOBAL srun -N 2 \ 
--ntasks-per-node=16 --qos=devel_0128 \ 
--partition=mem_0128 --time=00:05:00 \ 
h5perf --min-num-processes=32 -x 4M -X 4M \ 
-e 128M -B 1M -i 3
\end{verbatim}

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