JÜLICH
FORSCHUNGSZENTRUM

Parallel I/O and Portable Data Formats

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

PRACE Winter School 2017, Tel Aviv, February 6th - 9th 2017

Outline

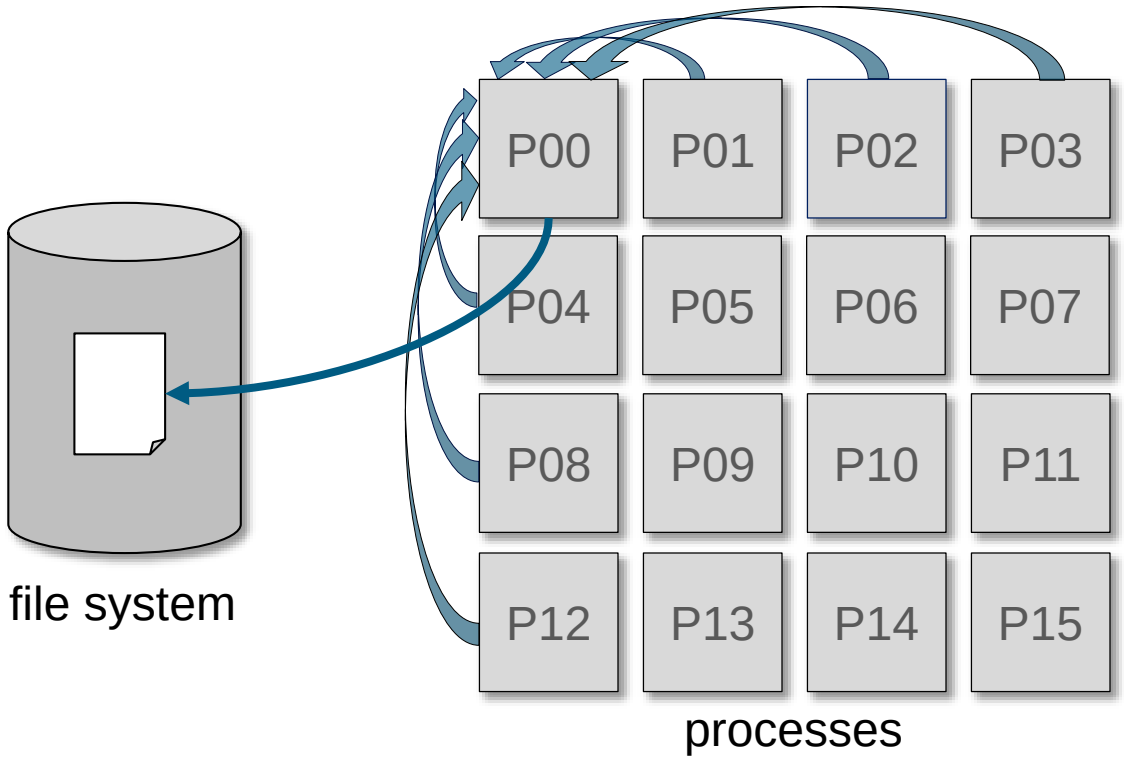
- February 6th
  - I/O strategies
  - I/O workflow
  - Parallel I/O software stack
  - MPI-I/O
    - General Overview
    - APIs
    - Hands-On
- February 7th
  - HDF-5
    - General Overview
    - APIs
    - Hands-On

JÜLICH
FORSCHUNGSZENTRUM

I/O strategies

PRACE Winter School 2017, Tel Aviv, February 6th - 9th 2017

One process performs I/O



One process performs I/O

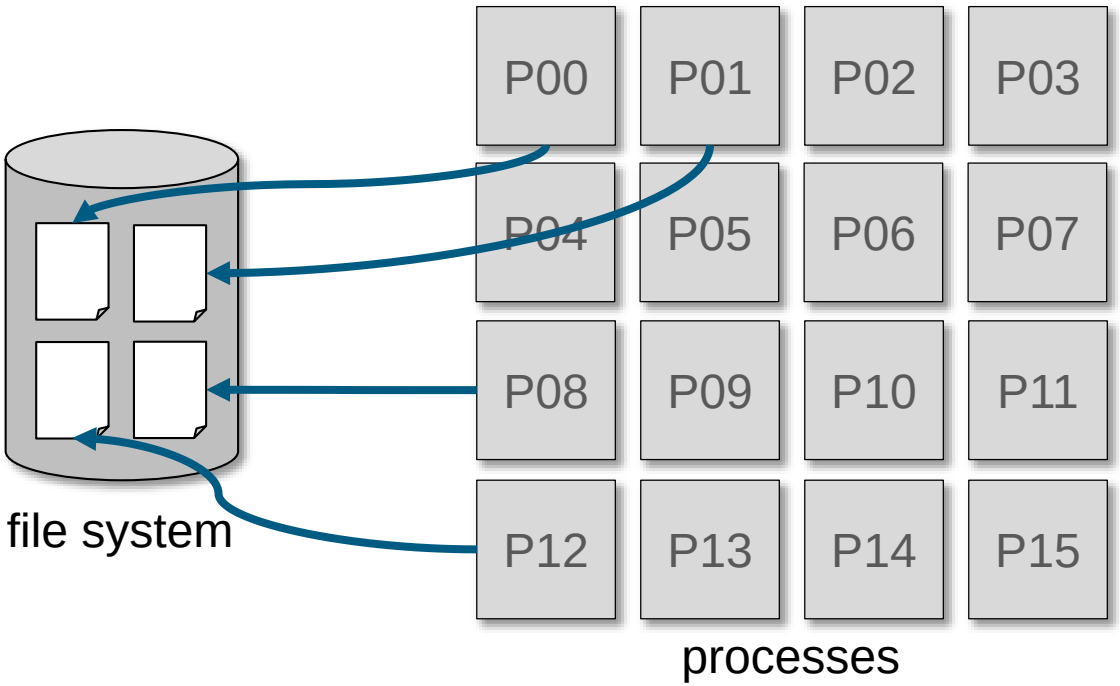
+ Simple to implement

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

Frequent flushing on small blocks

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

Task-local files



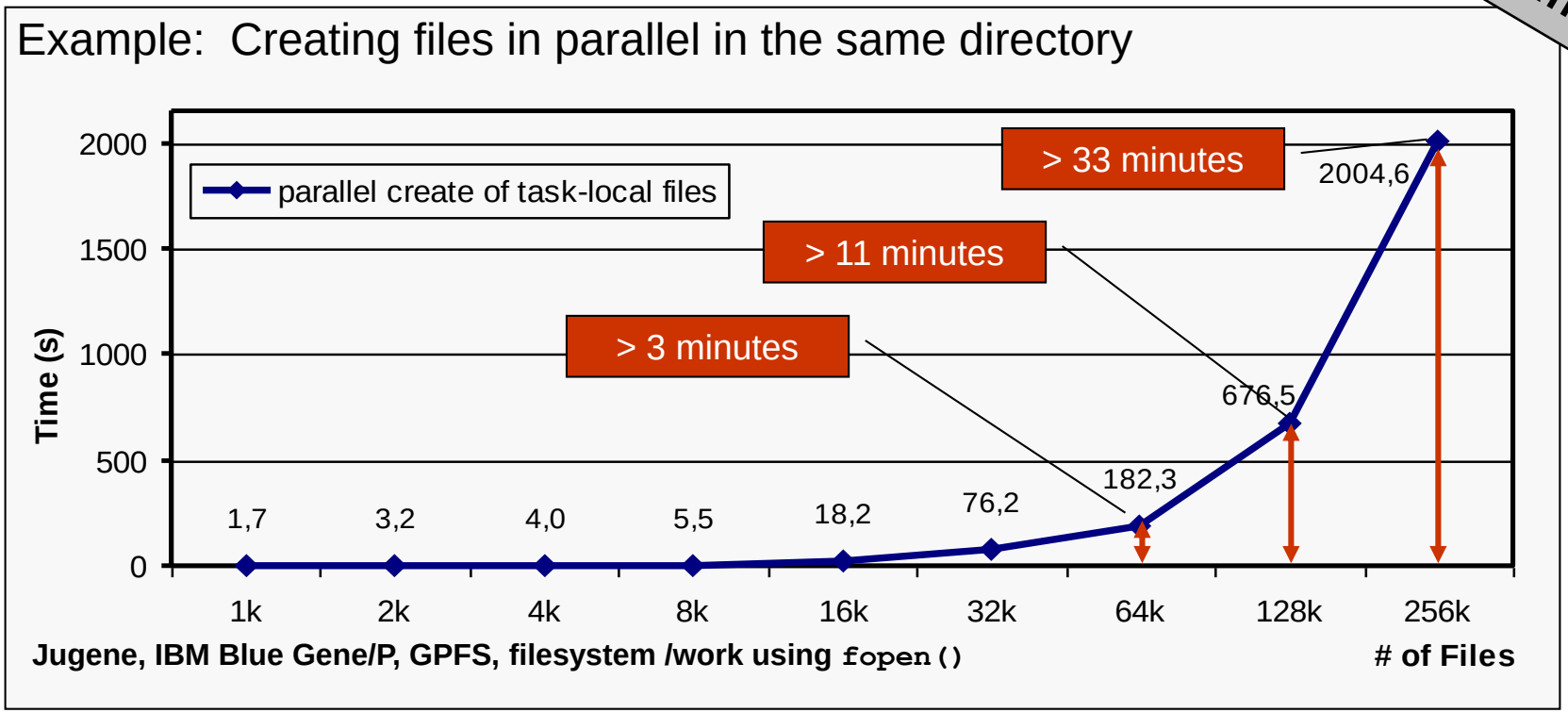
Task-local files

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

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

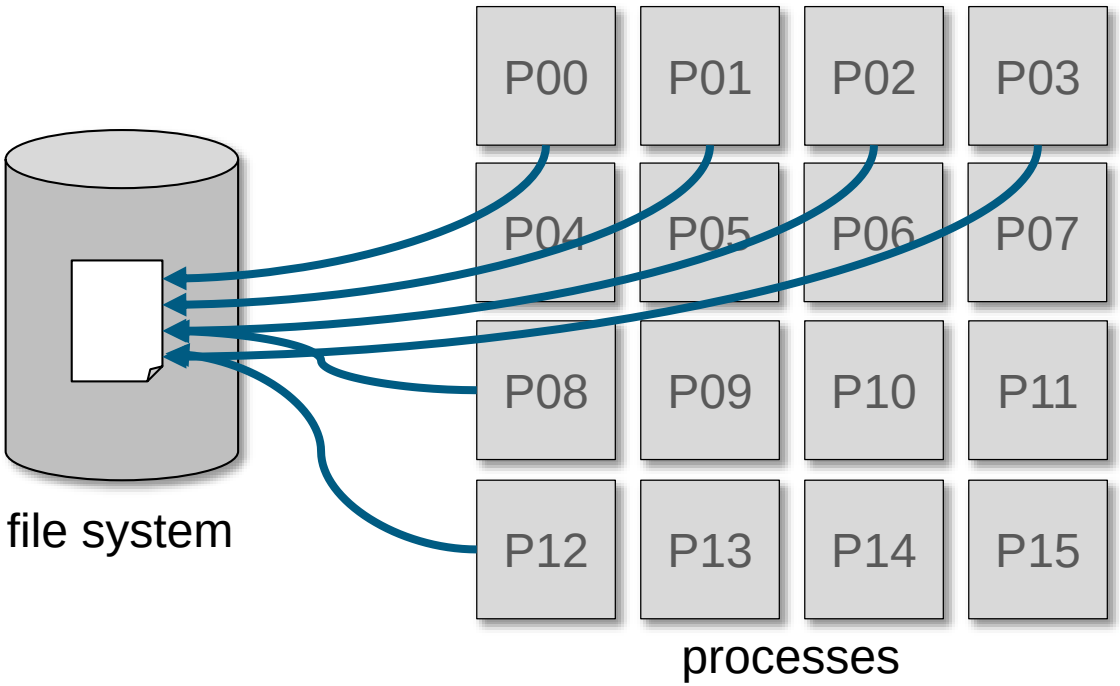
Serialization of meta data modification

Pitfall 2



The creation of 256.000 files costs 142.500 core hours!

Shared files

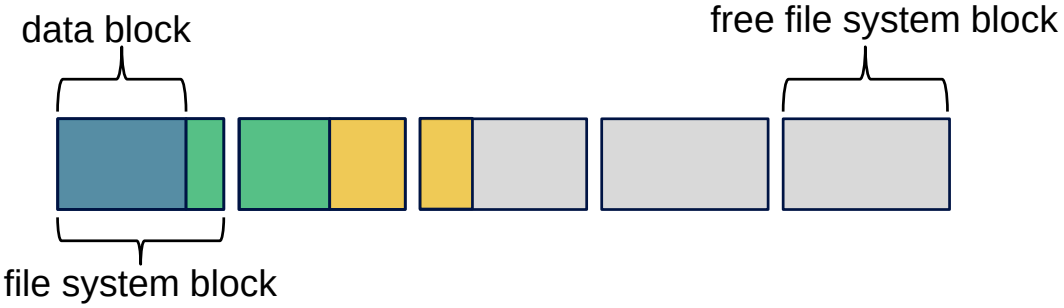


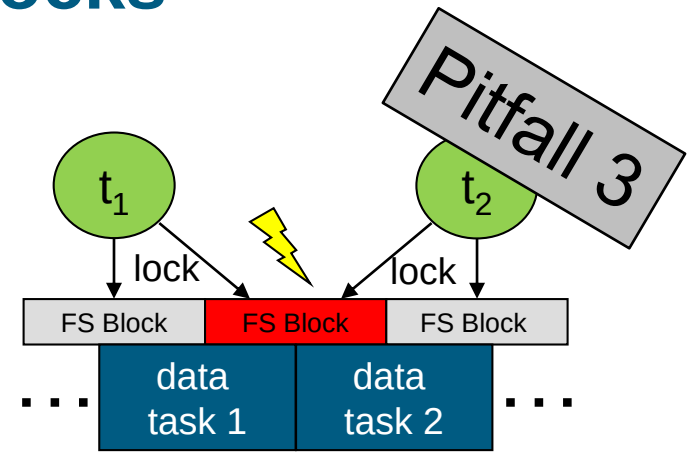
Shared files

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

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

False sharing of file system blocks



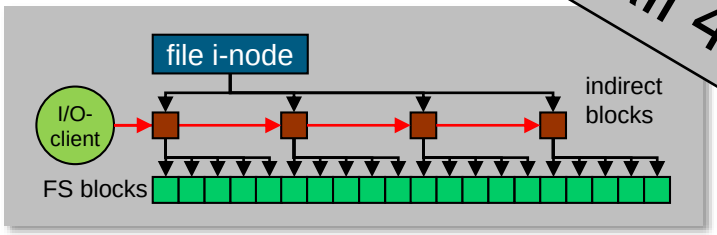


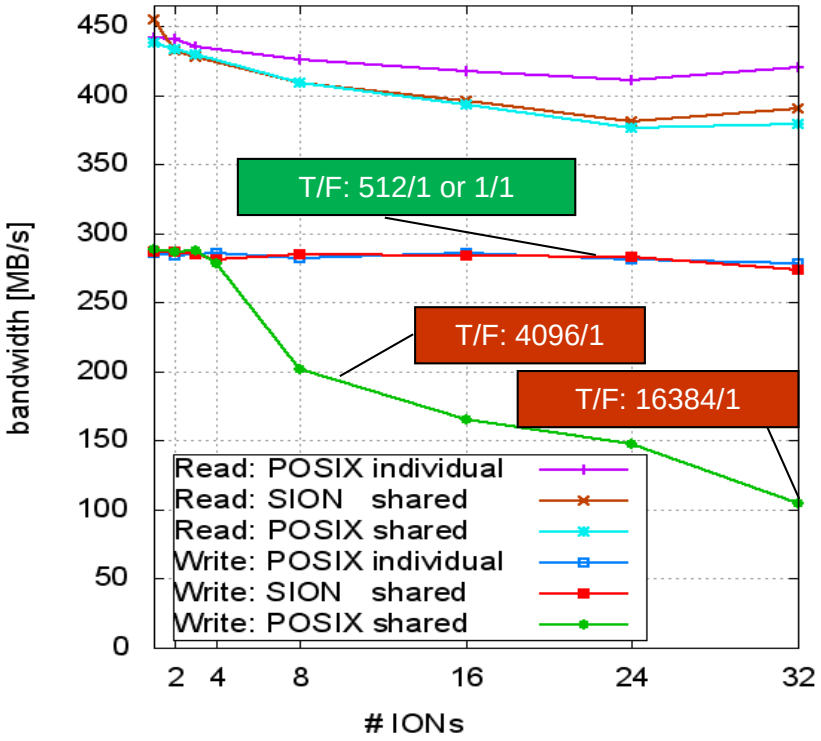
- Data blocks of individual processes **do not fill up a complete file system block**
- Several processes **share a file system block**
- Exclusive access (e.g. write) must be **serialized**
- The more processes have to synchronize the more waiting time will propagate

Number of Tasks per Shared File

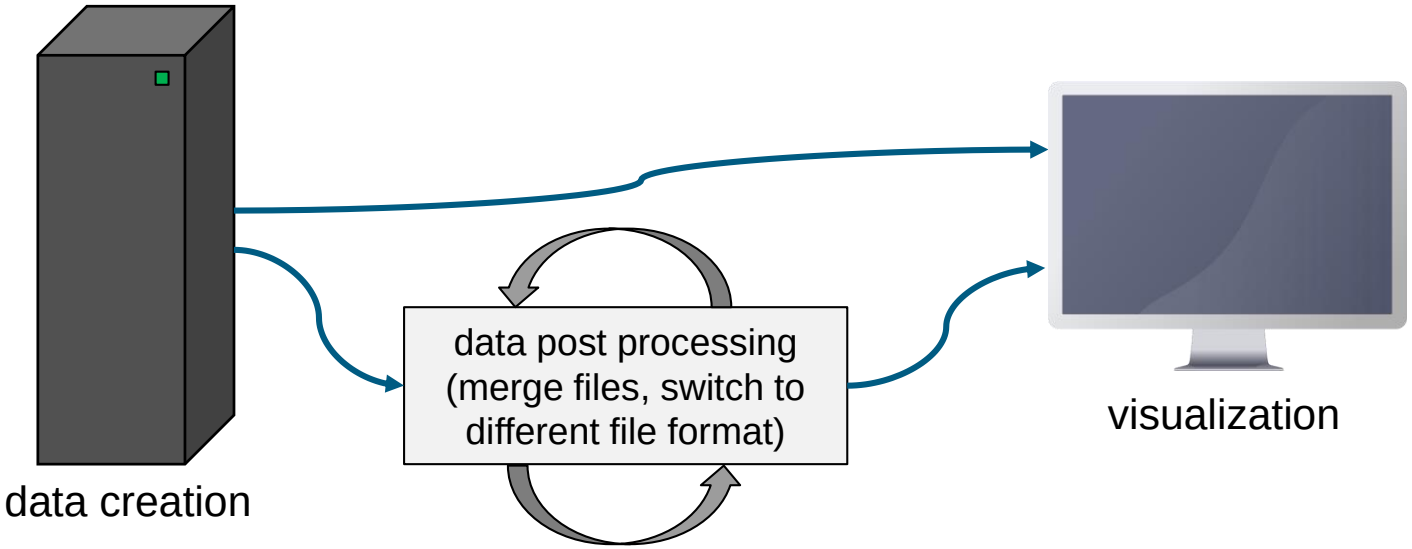
Pitfall 4

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





I/O Workflow

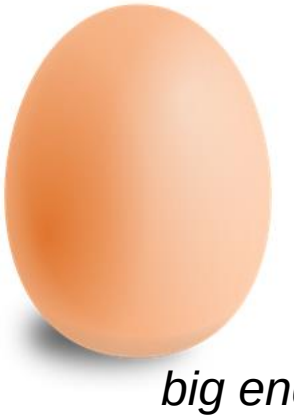


- Post processing can be very time-consuming (> data creation)
  - Widely used portable data formats avoid post processing
- Data transportation time can be long:
  - Use shared file system for file access, avoid raw data transport
  - Avoid renaming/moving of big files (can block backup)

Portability

Pitfall 5

- Endianness (byte order) of binary data



2,712,847,316

=

10100001 10110010 11000011 11010100

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

- Conversion of files might be necessary and expensive

Portability

Pitfall 6

- Memory order depends on programming language

1	2	3
4	5	6
7	8	9

→

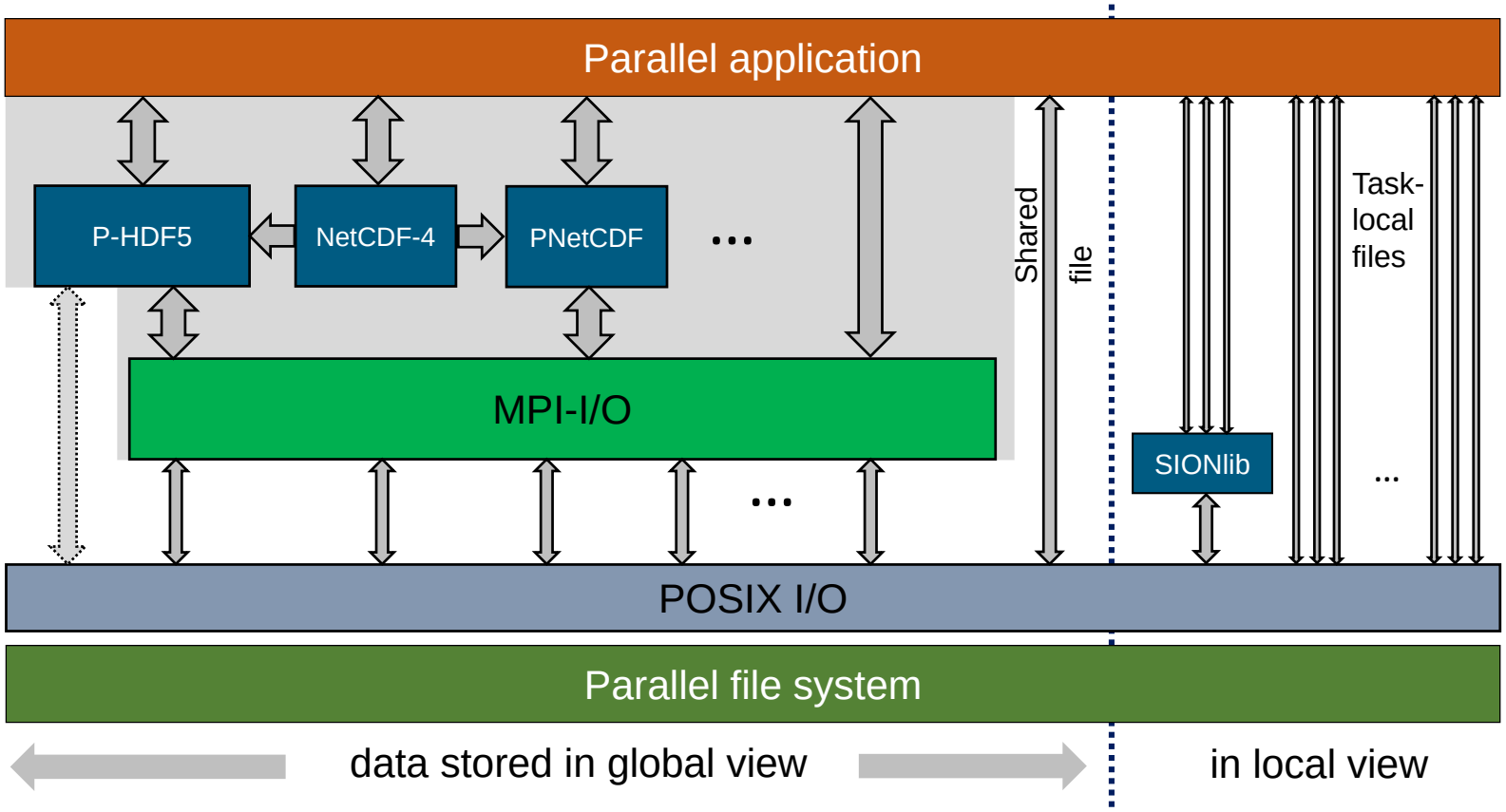
Address	row-major order (e.g. C/C++)	column-major order (e.g. Fortran)
1000	1	1
1001	2	4
1002	3	7
1003	4	2
1004	5	5
…	…	…

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

How to choose the I/O strategy?

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

Parallel I/O Software Stack



JÜLICH
FORSCHUNGSZENTRUM

MPI-I/O

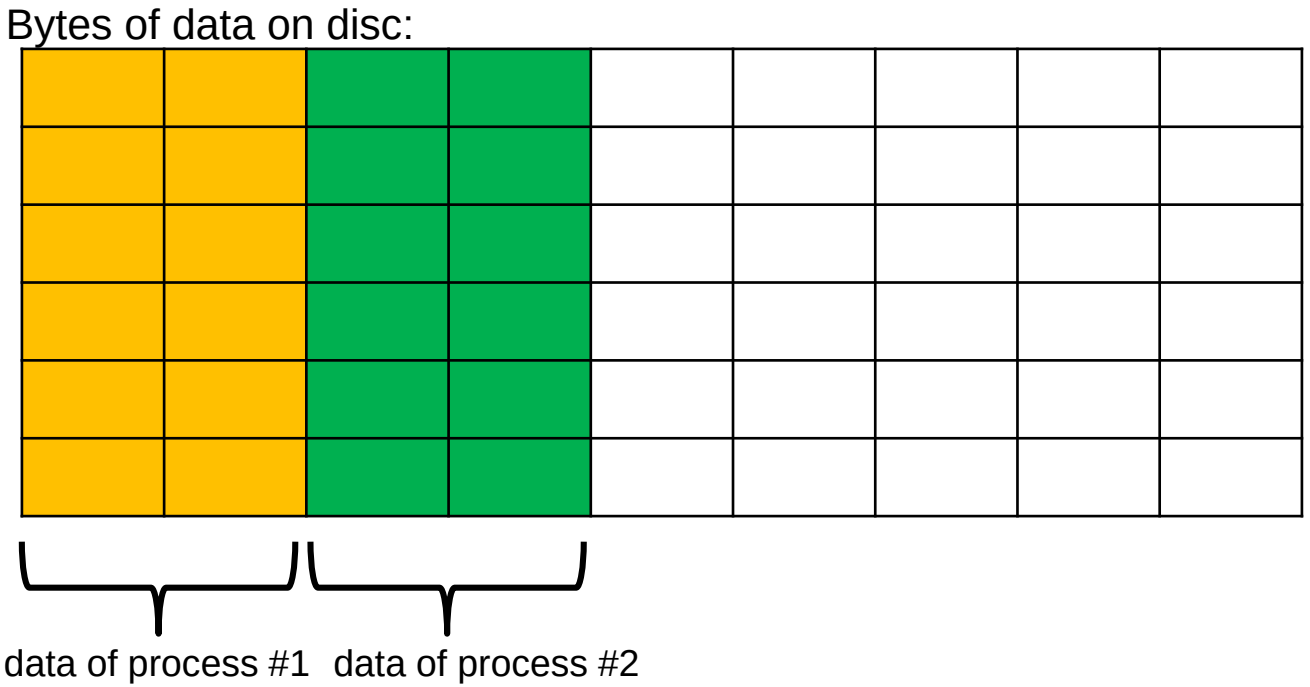
PRACE Winter School 2017, Tel Aviv, February 6th - 9th 2017

Features of MPI-I/O

- Standardized I/O API since 1997
- Available in basically all MPI implementations and as Open Source library ROMIO
- Support for C and Fortran
- Carries on the concepts of MPI communication to file I/O
- Allows portable and efficient implementation of parallel I/O operations due to support for
  - multiple data representations
  - asynchronous I/O
  - non-contiguous file access patterns
  - collective file access
  - MPI Info Objects

Idea of MPI-I/O

- Shared file is written/read by multiple processes in parallel
- File represents the **global** data view
- Process specific view can be used to write/read distributed data in parallel



MPI I/O – terminology

File

An MPI file is an ordered collection of typed data items.

File pointer

A file pointer is an implicit offset maintained by MPI.

File handle

An opaque MPI object. All operations on an open file reference the file through the file handle.

Language bindings

C	ISO C	`#include <mpi.h>`
Fortran	Fortran77	`include 'mpif.h'`
	Fortran90	`use mpi`
	Fortran08	`use mpi_f08`

Opening a file

C

```
int MPI_File_open(MPI_Comm comm, char *filename,
                  int amode, MPI_Info info,
                  MPI_File *fh)
```

Fortran

```
MPI_File_open(comm, filename, amode, info, fh)
  character(len=*) :: filename
  integer :: comm, amode, info, fh, ierror
```

- Call is collective on `comm`
- Filename must reference the same file on all processes
- Process-local files can be opened with `MPI_COMM_SELF`

Access mode

`amode` denotes the access mode of the file and is a bit vector

mandatory (mutual exclusive)

`MPI_MODE_RDONLY:`	read only access
`MPI_MODE_RDWR :`	read and write access
`MPI_MODE_WRONLY:`	write only access

optional (combinable)

`MPI_MODE_CREATE :`	create file if it doesn't exist
`MPI_MODE_EXCL :`	error if creating file that already exists
`MPI_MODE_DELETE_ON_CLOSE:`	delete file on close
`MPI_MODE_UNIQUE_OPEN :`	file can not be opened elsewhere
`MPI_MODE_SEQUENTIAL :`	sequential file access (e.g. tapes)
`MPI_MODE_APPEND :`	all file pointers are set to end of file

Combination with

C: Bit vector OR "|"

Fortran: Integer addition "+"

Closing a file

C

```
int MPI_File_close(MPI_File *fh)
```

Fortran

```
MPI_File_close(fh, ierror)
  integer :: fh, ierror
```

- Collective operation
- If `MPI_MODE_DELETE_ON_CLOSE` was specified on opening, the file is deleted after closing
- The user must ensure that all outstanding requests of a process connected with `fh` have completed before that process calls `MPI_FILE_CLOSE`

Querying file parameters – size and amode

C

```
int MPI_File_get_size(MPI_File fh,
                      MPI_Offset *size)
```

Fortran

```
MPI_File_get_size(fh, size, ierror)
  integer :: fh, ierror
  integer(kind=MPI_Offset_kind) :: size
```

C

```
int MPI_File_get_amode(MPI_File fh,
                       int *amode)
```

Fortran

```
MPI_File_get_amode(fh, amode, ierror)
  integer :: fh, amode, ierror
```

Preallocating space for a file

C

```
int MPI_File_preallocate(MPI_File fh,
                         MPI_Offset size)
```

Fortran

```
MPI_File_preallocate(fh, size, ierror)
  integer :: fh, ierror
  integer(kind=MPI_Offset_kind) :: size
```

- Collective operation that allocates `size` bytes storage space for `fh`
- Existing regions unaffected, new regions filled with undefined data
- If `size` is equal or less the current file size, the file size is unchanged

! **In some implementations, file pre-allocation might be expensive, use with care** !

Exercise

Exercise 1 – File manipulation

Open a file

- Write a program in **C or Fortran running with 8 processes** that
  - Opens a file `output.dat` for reading and writing; it creates the file if it does not exist
  - Queries the size of the file
  - Closes the file

Pre-allocating space

- Extend the program
  - Pre-allocate 1024 bytes for the file
  - Query the size of the file
  - Check the size of the file with `ls`

Compile

- `mpicc your_app.c -o your_app`
- `mpif90 your_app.f90 -o your_app`

Exercise - Template - C

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main (int argc,char *argv[]){
    int myrank, nranks;

    /* Initialize MPI */
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &nranks);

    if (myrank == 0)
        printf("Number of processors: %d\n",nranks);

    /* TODO */

    /* Finalize MPI */
    MPI_Finalize();
    return(EXIT_SUCCESS);
}
```

Exercise - Template - Fortran

```
program exercise1
  use mpi

  implicit none

  integer :: ierror,nranks,myrank

  ! Initialize MPI
  call MPI_INIT(ierror)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,nranks,ierror)
  call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierror)

  if (myrank == 0) then
    write(*,*) 'Number of processors: ', nranks
  end if

  ! TODO

  ! Finalize MPI
  call MPI_FINALIZE(ierror)

end program exercise1
```

Exercise - Job execution

Jobscript:

```
#!/bin/sh
#PBS -lnodes=1:ppn=8

cd $PBS_O_WORKDIR

. /opt/torque/etc/openmpi-setup.sh

mpirun ./a.out #use the name of your executable
```

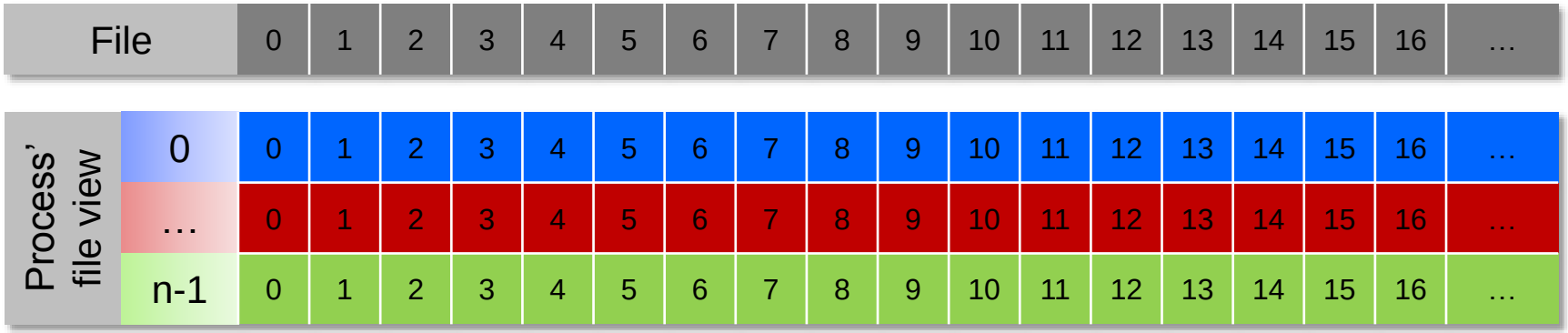
Job start:

```
qsub name_of_jobscript
```

Default file view

A default view for each participating process is defined implicitly while opening the file

- No displacement
- The file has no specific structure, it is a sequence of bytes
- All processes have access to the complete file



Changing the file view – etype

Example

- Program runs with 4 MPI processes
- Each process writes one integer

Elementary type (etype)

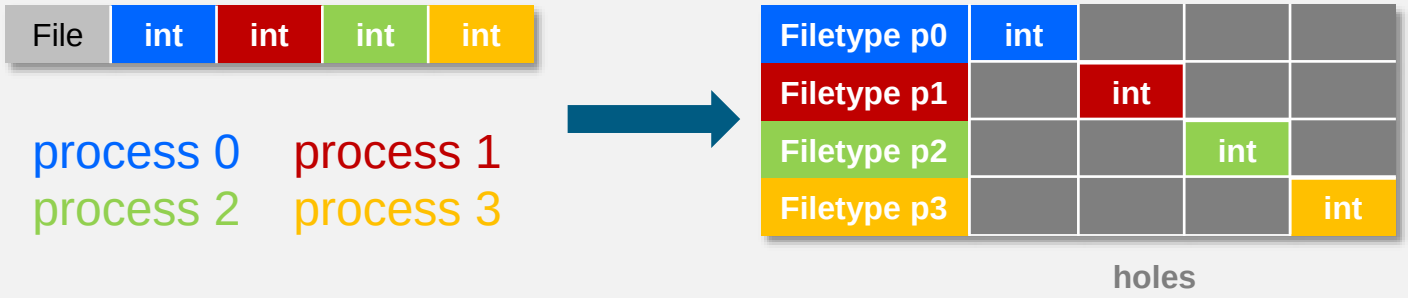
The etype is the basic entity (either an MPI basic or a derived datatype) of a file. It is the unit of data access and positioning. It must be the same on all processes with the same file handle.

Changing the file view – filetype

Filetype

The filetype is the basis for partitioning the file among processes. It defines a template for accessing the file

Example



Constructing filetypes

- A filetype is an MPI derived datatype
- All constructors for derived datatypes can be used to create a filetype, for example:
  - `MPI_Type_contiguous`
  - `MPI_Type_vector`
  - `MPI_Type_indexed`
  - `MPI_Type_create_struct`
  - `MPI_Type_create_subarray`
  - `MPI_Type_create_darray`
- After construction filetypes have to be committed

Constructing filetypes – Sub-array data

C

```
int MPI_Type_create_subarray(int ndims,
                             int array_of_sizes[],
                             int array_of_subsizes[],
                             int array_of_starts[],
                             int order,
                             MPI_Datatype etype,
                             MPI_Datatype *filetype)
```

Fortran

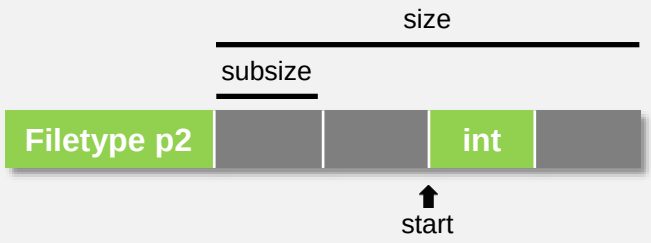
```
MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes,
                         array_of_subsizes,
                         array_of_starts, order, etype,
                         filetype, ierror)
  integer :: ndims, order, etype, filetype, ierror
  integer, dimension(*) :: array_of_sizes,
                           array_of_subsizes,
                           array_of_starts
```

Constructing filetypes – Sub-array data

C

```
int MPI_Type_create_subarray(int ndims,
                             int array_of_sizes[],
                             int array_of_subsizes[],
                             int array_of_starts[],
                             int order,
                             MPI_Datatype etype,
                             MPI_Datatype *filetype)
```

Example



ndims	1	dimension of array
array_of_sizes	4	
array_of_subsizes	1	units of etype
array_of_starts	2	

order	MPI_ORDER_C or MPI_ORDER_FORTRAN	
etype	C	MPI_INT
	Fortran	MPI_INTEGER

Committing and freeing filetypes

C

```
int MPI_Type_commit(MPI_Datatype *filetype)
```

Fortran

```
MPI_Type_commit(filetype, ierror)
  integer :: filetype, ierror
```

C

```
int MPI_Type_free(MPI_Datatype *filetype)
```

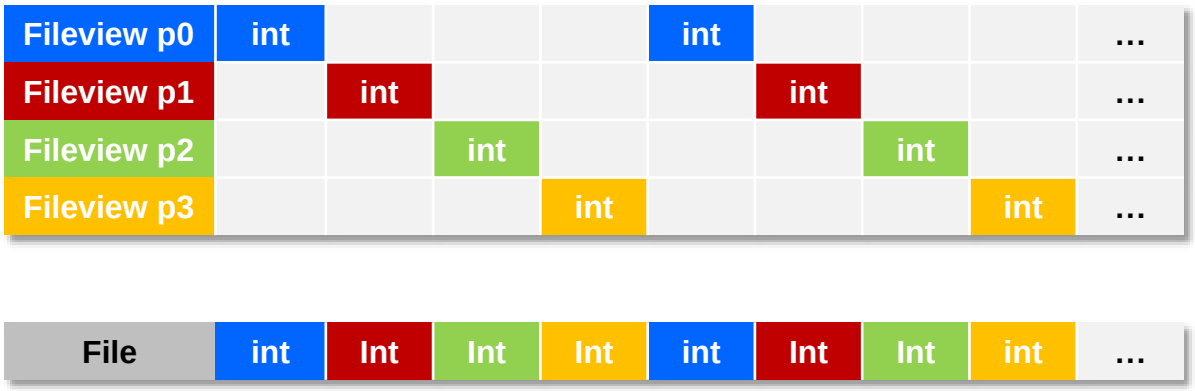
Fortran

```
MPI_Type_free(filetype, ierror)
  integer :: filetype, ierror
```

Changing the file view

File view

A view defines the file data visible to a process. Each process has an individual view of a file defined by a displacement, an elementary type and a file type. The pattern described by the filetype is repeated to define the view.



Changing the file view

C

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
                      MPI_Datatype etype, MPI_Datatype
                      filetype, const char *datarep,
                      MPI_Info info)
```

Fortran

```
MPI_File_set_view(fh, disp, etype, filetype, datarep,
                  info, ierror)
  integer :: fh, etype, filetype, info, ierror
  integer(kind=MPI_OFFSET_KIND) :: disp
  character(len=*) :: datarep
```

- Collective operation
- Local and shared file pointers are reset to zero
- `ETYPE` and `FILETYPE` must be committed
- `DATAREP` is a string specifying the data format (representation)

Data representations

native

Data is stored in the file exactly as it is in memory

+ No loss of precision
- On heterogeneous systems loss of transparent interoperability
- No guarantee that MPI files are accessible from C/Fortran

internal

Data is stored in implementation-specific format

+ Can be used in a homogeneous **and** heterogeneous environment
+ Implementation will perform file conversions if necessary
- No guarantee that MPI files are accessible from C/Fortran

external32

Data is stored in Standardized data representation (big-endian IEEE)

+ Can be read/written also by non-MPI programs
- Precision and I/O performance may be lost due to type conversions between `native` and `external32` representations

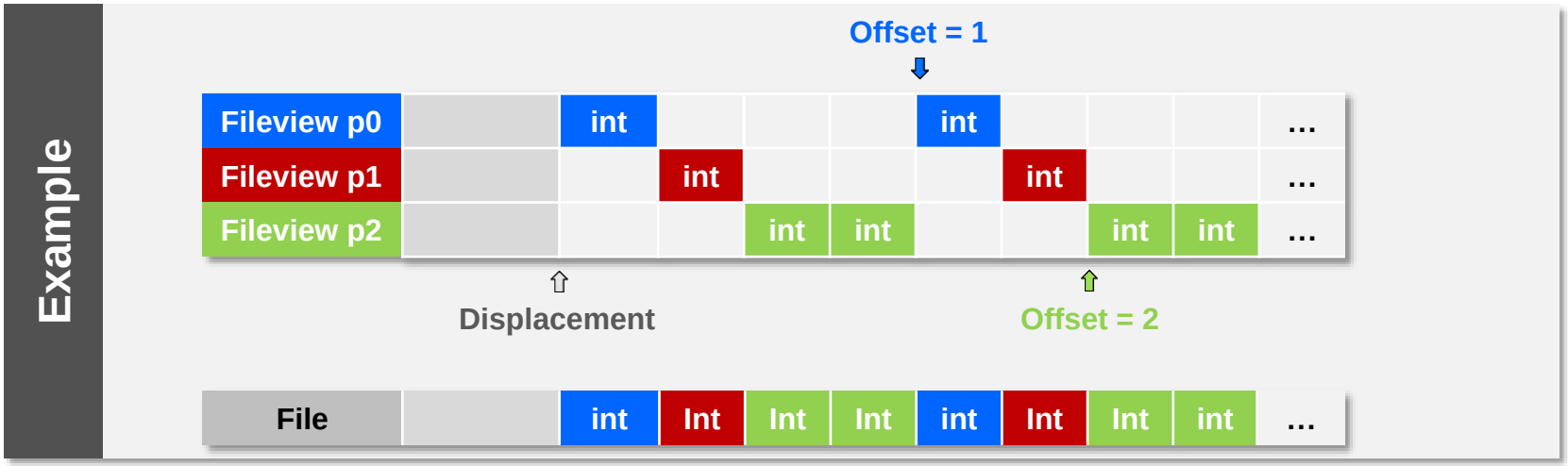
Offset and displacement

Displacement

Displacement is an absolute *byte* position relative to the beginning of a file.

Offset

Offset is a position in the file relative to the **current view**. It is expressed as a count of *etypes*. Holes in the view's filetype are skipped.



Data access – general properties

- Positioning
  - Explicit offsets
  - File pointers
    - shared
    - individual
- Synchronism
  - Blocking
  - Nonblocking
- Coordination
  - Noncollective
  - Collective

Example

`POSIX read()/write()`

These are blocking, noncollective operations with individual file pointers

Positioning

File pointers

Individual file pointers

- Each process has its own file pointer that is only altered on accesses of that specific process with using the corresponding MPI routines

Shared file pointers

- This file pointer is shared among all processes in the communicator used to open the file
- It is modified by any shared file pointer access of any process
- Shared file pointers can only be used if each process has access to the whole file
- Deterministic outcome when **collective** routines are used: data is written in the order of process ranks

Explicit offset

- No file pointer is used or modified
- An explicit offset is given to determine access position
- This cannot be used with `MPI_MODE_SEQUENTIAL`

Data access – overview

Positioning	Synchronism	Coordination	
		Noncollective (nc)	Collective (c)
Individual file pointers (ifp)	Blocking (b)	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	Nonblocking (nb) / Split collective (sc)	MPI_File_iread MPI_File_iwrite	MPI_File_read_all_begin MPI_File_read_all_end MPI_File_write_all_begin MPI_File_write_all_end
Shared file pointers (sfp)	Blocking (b)	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	Nonblocking (nb) / Split collective (sc)	MPI_File_iread_shared MPI_File_iwrite_shared	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end
Explicit offsets (ep)	Blocking (b)	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	Nonblocking (nb) / Split collective (sc)	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_read_at_all_begin MPI_File_read_at_all_end MPI_File_write_at_all_begin MPI_File_write_at_all_end

Writing data – blocking, noncollective

C

```
int MPI_File_write(MPI_File fh, void *buf,
                   int count, MPI_Datatype datatype,
                   MPI_Status *status)
```

Fortran

```
MPI_File_write(fh, buf, count, datatype, status,
               ierror)
  integer :: fh, count, datatype, ierror
  integer, dimension(MPI_STATUS_SIZE) :: status
  type, dimension(*) :: buf
```

- Starts writing at the current position of the (individual) file pointer
- Writes `count` elements of `datatype` from memory starting at `buf`
- The sequence of basic datatypes of `datatype` (*type signature*) must match contiguous copies of the `etype` of the current view
- `status` will indicate how many bytes have been written
- Default status: `MPI_STATUS_IGNORE`

Writing data – blocking, collective

C

```
int MPI_File_write_all(MPI_File fh, void *buf, int
                       count, MPI_Datatype datatype,
                       MPI_Status *status)
```

Fortran

```
MPI_File_write_all(fh, buf, count, datatype, status,
                   ierror)
  integer :: fh, count, datatype, ierror
  integer, dimension(MPI_STATUS_SIZE) :: status
  type, dimension(*) :: buf
```

- Starts writing at the current position of the (individual) file pointer with the same properties as the `MPI_File_write` command, except for being collective
- MPI can use communication between ranks to optimize I/O
- False share of file system blocks can be minimized as access pattern can be communicated an rearranged

Writing data – nonblocking, noncollective

C

```
int MPI_File_iwrite(MPI_File fh, void *buf, int
                    count, MPI_Datatype datatype,
                    MPI_Resquest *request)
```

Fortran

```
MPI_File_iwrite(fh, buf, count, datatype, request,
                ierror)
  integer :: fh, count, datatype, request, ierror
  type, dimension(*) :: buf
```

- Starts writing at the current position of the (individual) file pointer with the same properties as the `MPI_File_write` command, except for being nonblocking
- `buf` should not be altered in between
- Must be completed by a call to an `MPI_Wait` routine

File access using explicit offsets

- The offset is specified explicitly in the data access routines by the user
- These routines do not use nor update any file pointer

C

```
int MPI_File_write_at(MPI_File fh,
                      MPI_Offset offset,
                      void *buf, int count,
                      MPI_Datatype datatype,
                      MPI_Status *status)
```

Fortran

```
MPI_File_write_at(fh, offset, buf, count, datatype,
                  status, ierror)
  integer :: fh, count, datatype, ierror
  integer, dimension(MPI_STATUS_SIZE) :: status
  integer(kind=MPI_OFFSET_KIND) :: offset
  type, dimension(*) :: buf
```

Reading data – blocking, noncollective

C

```
int MPI_File_read(MPI_File fh, void *buf, int count,
                  MPI_Datatype datatype,
                  MPI_Status *status)
```

Fortran

```
MPI_File_read(fh, buf, count, datatype, status,
              ierror)
  integer :: fh, count, datatype, ierror
  type, dimension(*) :: buf
  integer, dimension(MPI_STATUS_SIZE) :: status
```

- Starts reading at the current position of the (individual) file pointer
- Reads `count` elements of `datatype` from memory starting at `buf`
- `status` will indicate how many bytes have been read

Querying current file pointer position

C

```
int MPI_File_get_position(MPI_File fh,
                          MPI_Offset *offset)
```

Fortran

```
MPI_File_get_position(fh, offset, ierror)
  integer :: fh, ierror
  integer(kind=MPI_OFFSET_KIND) :: offset
```

- Returns the current position of the individual file pointer in `offset` (units of etype relative to current file view)
- The value can be used to
  - return to this position (seek)
  - calculate a displacement

Seeking a file position

C
```
int MPI_File_seek(MPI_File fh, MPI_Offset *offset,
                  int *whence)
```

Fortran
```
MPI_File_seek(fh, offset, whence, ierror)
  integer :: fh, whence, ierror
  integer(kind=MPI_OFFSET_KIND) :: offset
```

- Updates the individual file pointer according to `WHENCE`, which can have the following values:
  - `MPI_SEEK_SET`: pointer is set to **`OFFSET`**
  - `MPI_SEEK_CUR`: pointer is set to the current position plus `OFFSET`
  - `MPI_SEEK_END`: pointer is set to the end of file plus `OFFSET`
- `OFFSET` can be negative, which allows seeking backwards
- It is erroneous to seek to a negative position in the view

Converting offsets to absolute positions

C

```
int MPI_File_get_byte_offset(MPI_File fh,
                             MPI_Offset *offset,
                             MPI_Offset *disp)
```

Fortran

```
MPI_File_get_byte_offset(fh, offset, disp, ierror)
  integer :: fh, ierror
  integer(kind=MPI_OFFSET_KIND) :: offset, disp
```

- Converts an file-view-related `offset` (in units of etype) to an absolute byte position `disp`

Exercise

Exercise 2 – Data access

Writing and reading data

- Write a program in C or Fortran running with n processes
  - Each MPI process writes its own rank to the common file `rank.dat`
  - The ranks in the file should be in the order 0…(*n*-1)
  - Rank 0 reads the complete file and prints its content to the screen

Accessing parts of files

- Take the file `rank.dat`
  - The processes read the integers in reverse order:
    - Process 0 reads the n^{th} integer from the file
    - Process 1 reads the $(n\text{-}1)^{st}$ integer from the file
    - …

Help

- There are different solutions to solve these exercises
- You can use explicit offsets, file views, seeks or shared filepointers
- Try to use different approaches for the different exercises
- You can use `MPI_Type_size` if you need the size of a particular MPI datatype

MPI_Info object

Used to pass hints for optimization to MPI

- Consist of (key,value) pairs, key and value being strings
- A key may only have one value
- `MPI_INFO_NULL` is always a valid MPI_Info object
- The maximum key size is `MPI_MAX_INFO_KEY`
- The maximum value size is `MPI_MAX_INFO_VAL` (implementation dependent)

! `MPI_MAX_INFO_VAL` **might be very large! It is not advisable to declare strings of that size!** !

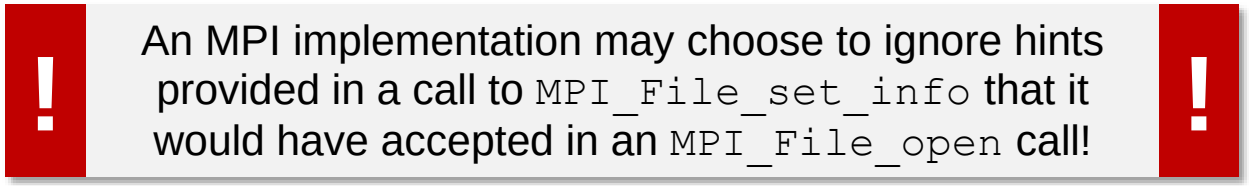
reference

```
MPI_Info_create, MPI_Info_free,
MPI_Info_set, MPI_Info_delete,
MPI_Info_get, MPI_Info_get_nkeys,
MPI_Info_get_valuelen, MPI_Info_get
```

MPI_Info object – Usage for MPI I/O

Passing hints to MPI-I/O

- Passing hints when opening the file
  - The corresponding MPI_Info object is provided in the MPI_File_open call
- Setting hints for files which are already open
  - The corresponding MPI_Info object is provided using the MPI_File_set_info routine
- Passing hints using a file with (key,value) pairs
  - ROMIO offers the possibility to specify (key,value) pairs using an ASCII file



Associate/retrieve info objects of open files

C
```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

Fortran
```
MPI_File_set_info(fh, info, ierror)
  integer :: fh, info, ierror
```

C
```
int MPI_File_get_info(MPI_File fh, MPI_Info *info)
```

Fortran
```
MPI_File_get_info(fh, info, ierror)
  integer :: fh, info, ierror
```

Passing hints using a file

Command	`export ROMIO_HINTS=<absolute_path>/hintfile`

- Variable must be exported to the CN
  - Use the appropriate option of MPI starters like `mpiexec` or `runjob` to export `ROMIO_HINTS` to the CNs
- Hints are valid for all files handled by MPI I/O in the application through
  - direct MPI I/O calls
  - libraries using MPI I/O (e.g., HDF5)

Example

`hintfile`

```
collective_buffering true
cb_buffer_size       33554432
cb_block_size        4194304
```

Reserved keys for file hints

File properties		
	`filename`	
	Specifies the file name used when the file was opened.	`string` *implementation dependent*
	`file_perm`	
	Specifies the files permissions when to use for file creation.	`string` *same value on all processes, implementation dependent values*
	`access_style`	

	This hint specifies the manner in which the file will be accessed until the file is closed or until the value of this key is altered	*comma separated list of strings*
		`read_once`
		`write_once`
		`read_mostly`
		`write_mostly`
		`sequential`
		`reverse_sequential`
		`random`

Reserved keys for file hints

Environment/setup		
	`nb_proc`	
	Specifies the number of parallel processes that will be assigned typically to run programs that access this file.	`integer` *same value on all processes*
	`num_io_nodes`	
	Specifies the number of I/O devices available to access the file.	`integer` *same value on all processes*
	`io_node_list`	
	This hint specifies the a list of I/O devices that should be used to store the file.	*comma separated list of strings, same value on all processes, Implementation/system dependent values*

Reserved keys for file hints

Data properties		
	`chuncked`	
	Specifies that the file consists of a multidimensional array that is often accessed by subarrays.	*Comma separated list of array dimensions, starting with the most significant one , same value on all processes*
	`chunked_item`	
	Specifies the size of each array entry in bytes.	*Comma separated list of integers, same value on all processes*
	`chunked_size`	
	Specifies the dimensions of the subarrays.	*Comma separated list of array dimensions, starting with the most significant one , same value on all processes*

Most significant dimension		
	C	First dimension (row-major)
	Fortran	Last dimension (column-major)

Reserved keys for file hints

Collective buffering		
	`collective_buffering`	
	Specifies whether the application might benefit from collective buffering.	`boolean` *same value on all processes*
	`cb_nodes`	
	Specifies the number of target nodes to be used for collective buffering.	`integer` *same value on all processes*
	`cb_block_size`	
	Specifies the block size to be used for collective buffering. Data is accessed in chunks of this size.	`integer` *same value on all processes*
	`cb_buffer_size`	
	Specifies the total buffer space that can be used for collective buffering on each node, usually a multiple of `cb_block_size`.	`integer` *same value on all processes*

Reserved keys for file hints

Lustre file system		
	`striping_factor`	
	Specifies the number of I/O devices (object storage target, OST) that the file should be sriped across.	`integer` *same value on all processes*
	`stiping_unit`	
	Specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device..	`integer` *same value on all processes*

JÜLICH
FORSCHUNGSZENTRUM

HDF5

PRACE Winter School 2017, Tel Aviv, February 6th - 9th 2017

Outline

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

What is HDF5?

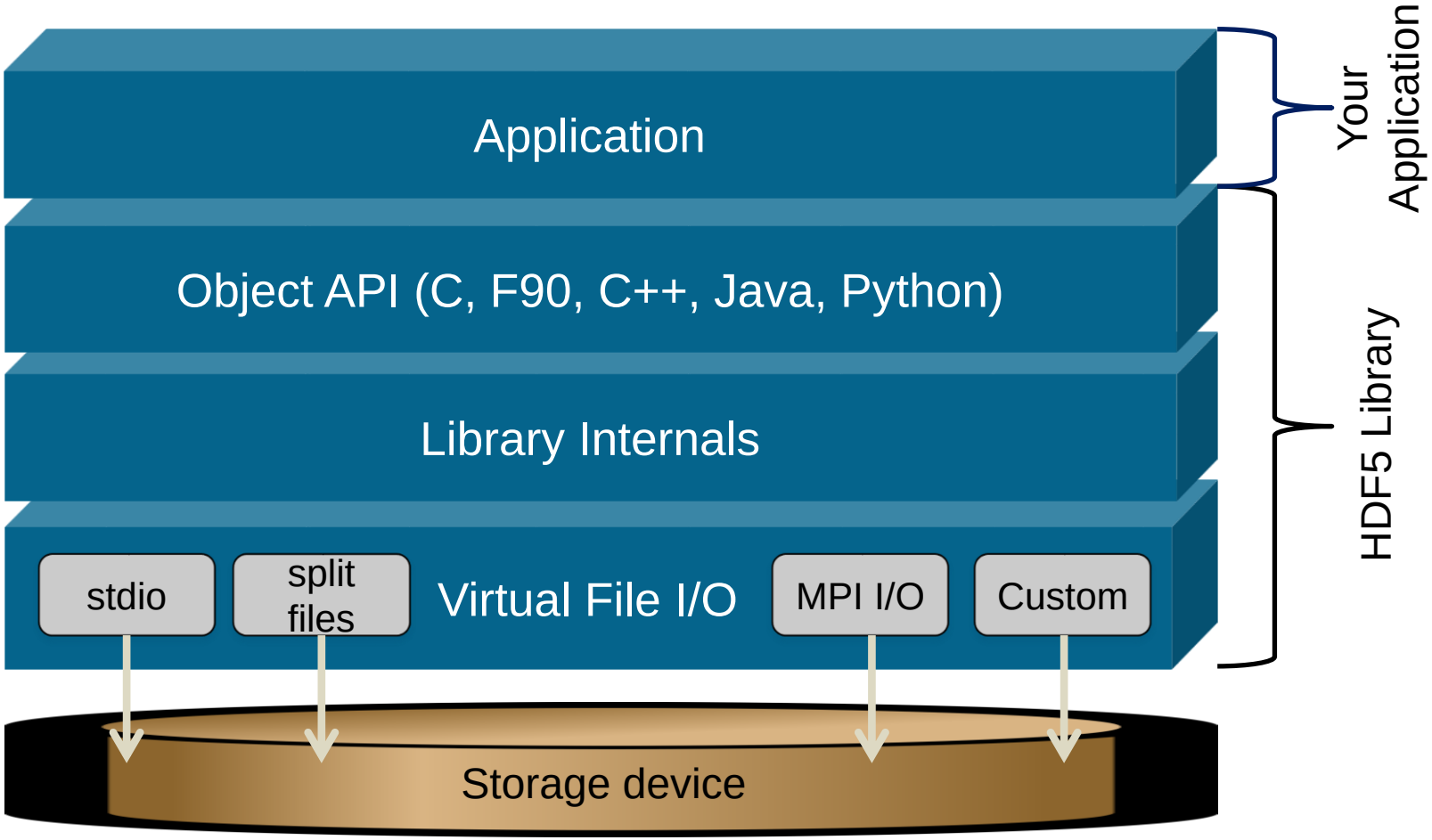
Hierarchical Data Format

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

HDF5 - Features

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

Layers of the HDF5 Library



File organization

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

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

/DataSet1

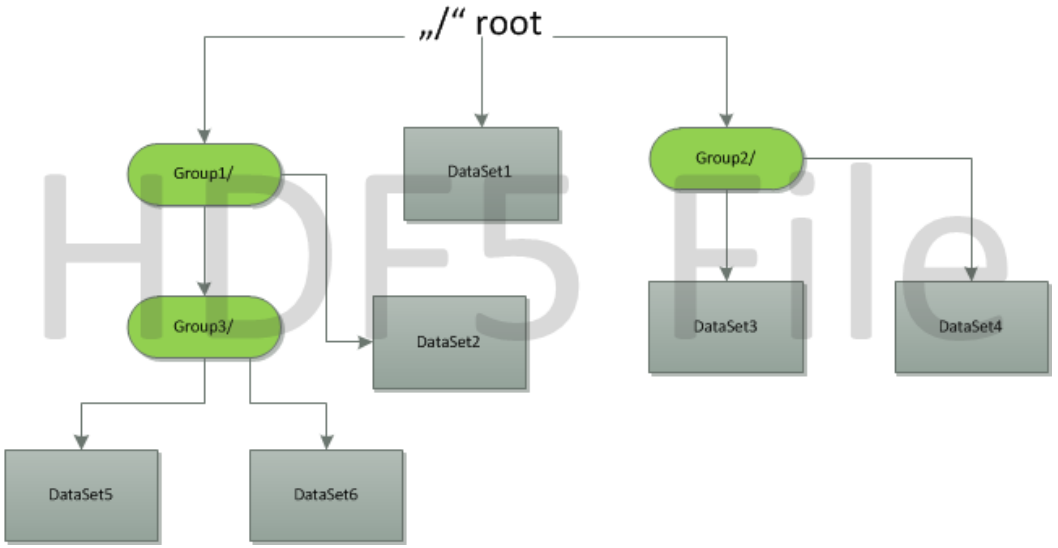
/Group1/DataSet2

/Group1/Group3/DataSet5

/Group1/Group3/DataSet6

/Group2/DataSet3

/Group2/Dataset4



Terminology

File

Container for storing data

Group

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

Attribute

Can be used to describe datasets and is attached to them

Dataspace

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

Dataset

Multi-dimensional array of data elements

Library specific types

C

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

Fortran

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

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

Fortran HDF5 open

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

Fortran

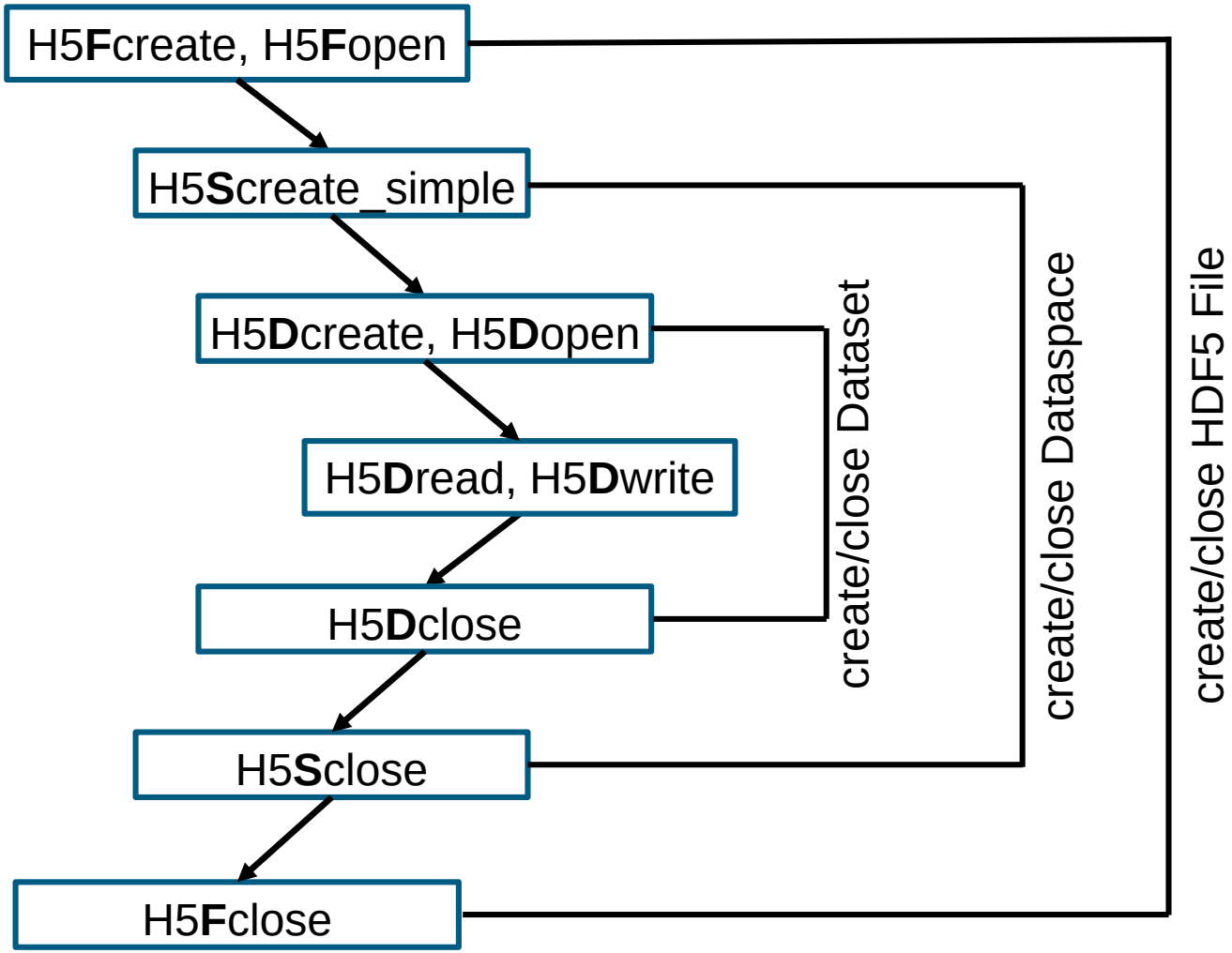
```
H5OPEN_F(STATUS)
  INTEGER, INTENT(OUT) :: STATUS

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

- `status` returns `0` if successful

API naming scheme (excerpt)

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

General Procedure



Creating an HDF5 file

C

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

Fortran

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

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

Opening an existing HDF5 file

C

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

Fortran

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

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

Access modes

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

Group creation

C

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

Fortran

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

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

Closing an HDF5 file

C

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

Fortran

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

Exercise

Exercise 1 – 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

Execute your program by using:

```
export LD_LIBRARY_PATH=/share/apps/hdf5/lib:$LD_LIBRARY_PATH
./your_app
```

Check the resulting file using:

```
/share/apps/hdf5/bin/h5dump output_file
```

Compile

- ```
  mpicc your_app.c -o your_app \
  -I/share/apps/hdf5/include \
  -L/share/apps/hdf5/lib -lhdf5
  ```
- ```
  mpif90 your_app.f90 -o your_app \
  -I/share/apps/hdf5/include \
  -L/share/apps/hdf5/lib -lhdf5_fortran
  ```

Exercise - Template - C

```
#include <stdlib.h>
#include <hdf5.h>

int main(int argc, char** argv)
{

    /* TODO */

    return(EXIT_SUCCESS);
}
```

Exercise - Template - Fortran

```
program helloworld
    use hdf5

    implicit none

    integer :: status

    ! initialize fortran interface
    call H5open_f(status)

    ! TODO

    ! shut down fortran interface
    call H5close_f(status)
end
```

HDF5 pre-defined datatypes (excerpt)

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

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

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

Dataspace

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

Simple dataspace

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

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

Creating a dataspace

C

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

Fortran

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

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

Creating a dataspace

C

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

Fortran

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

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

Creating an Attribute

C

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

Fortran

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

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

Writing an Attribute

C

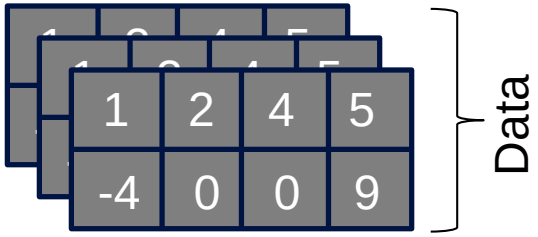
```
herr_t H5Awrite(hid_t attr_id, hid_t mem_type_id,
                const void *buf)
```

Fortran

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

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

Dataset (metadata + data)



Metadata

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

Datatype
- Integer

Attributes
- Time = 2.1
- Temp = 122

Storage
- Contiguous

Creating a Dataset

C

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

Fortran

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

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

Creating a Dataset

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

Property Lists

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

Recipe: Creating an empty dataset

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

Exercise

Exercise 2 – 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 integer attribute connected to this dataset
- Write a integer value into this attribute

Check the resulting file using:

```
h5dump
```

Writing to a dataset

C

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

Fortran

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

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

Writing to a dataset

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

Open a existing dataset

C

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

Fortran

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

- `dapl_id`: Dataset access property list

Dataspace inquiry

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

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

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

Reading a dataset

C

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

Fortran

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

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

Exercise

Exercise 3 – HDF5 write data

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

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

- Write this array into the existing empty HD5 dataset

Check the resulting file using:

```
h5dump
```

Excursion: row-major / column-major order

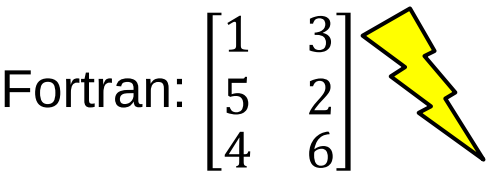
"Logical" data view:

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

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

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

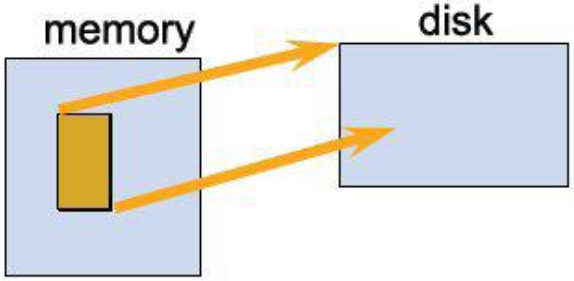
$$\text{C: } \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad \text{Fortran: } \begin{bmatrix} 1 & 3 \\ 5 & 2 \\ 4 & 6 \end{bmatrix}$$



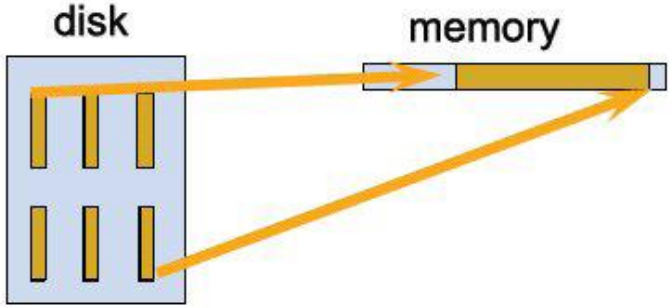
Storing data in a `2x3` dimensional dataset:

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

Partial I/O - Hyperslabs

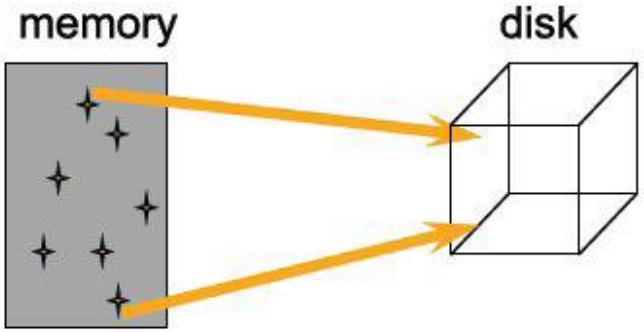


(a) Hyperslab from a 2D array to the corner of a smaller 2D array

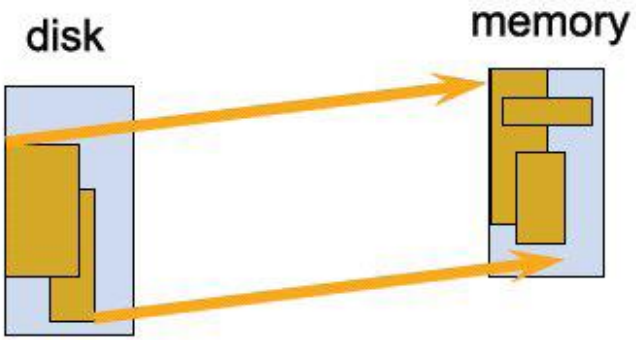


(b) Regular series of blocks from a 2D array to a contiguous sequence at a certain offset in a 1D array

Partial I/O - Hyperslabs



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



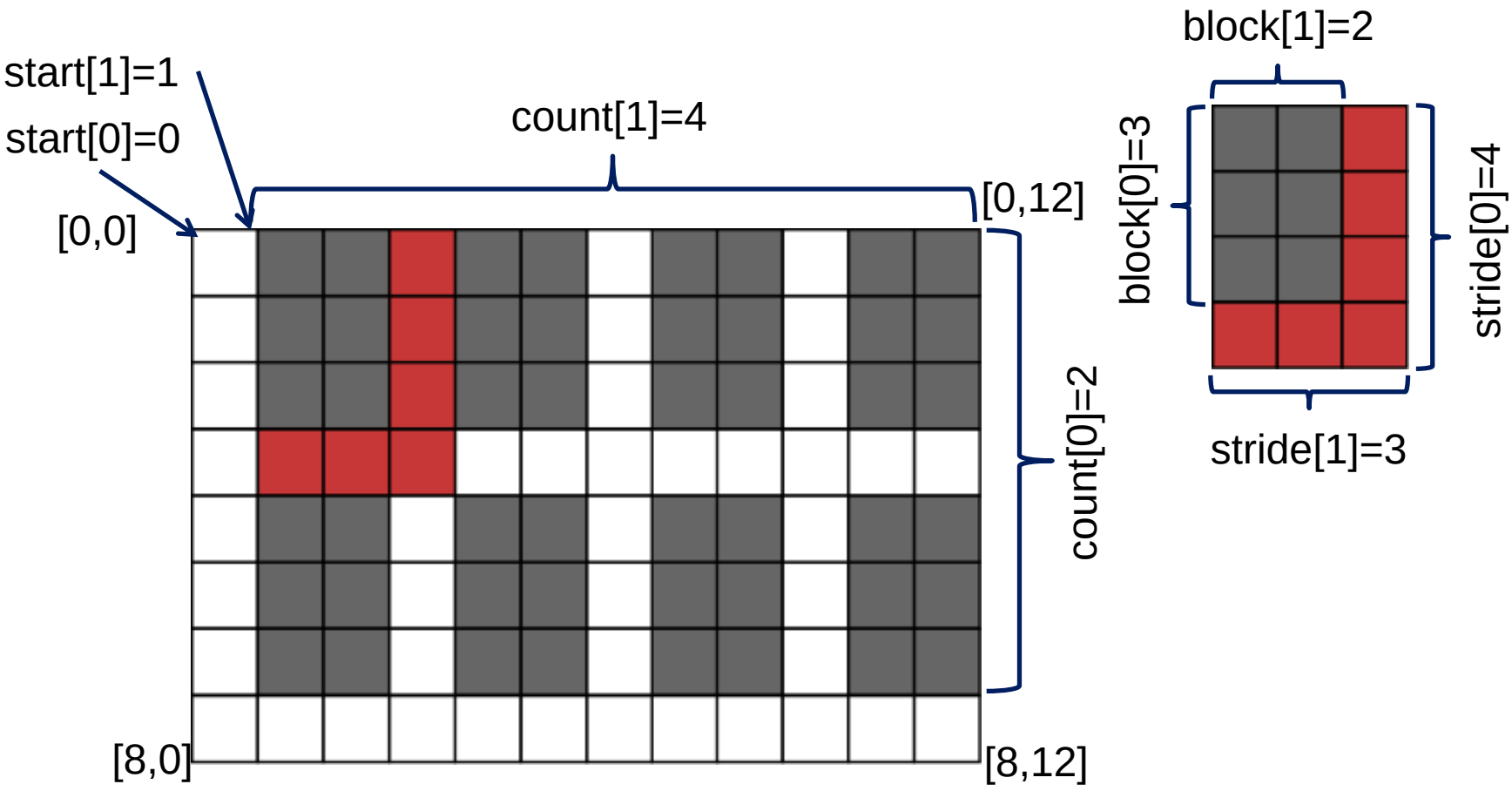
(d) Union of hyperslabs in file to union of hyperslabs in memory.

Partial I/O - Hyperslabs

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

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

Hyperslab example



Creating hyperslabs

C

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

Fortran

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

Creating hyperslabs

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

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

JÜLICH
FORSCHUNGSZENTRUM

Parallel I/O and Portable Data Formats
Parallel HDF5

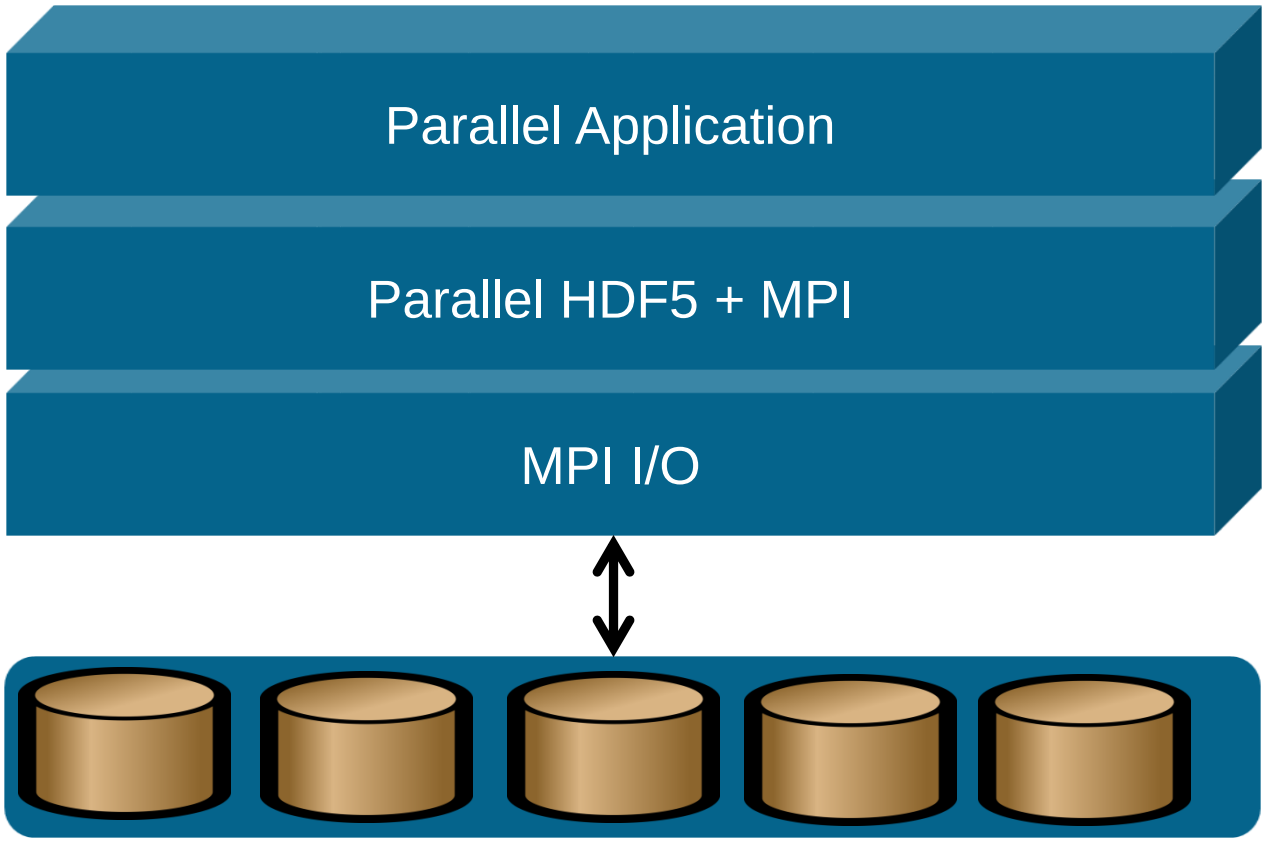
Sebastian Lührs
s.luehrs@fz-juelich.de
Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH

Jülich, March 15th, 2016

Factoids

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

Implementation layers



User Application

HDF5 library

Parallel I/O layer

Parallel file systems
(GPFS, Lustre, …)

Important to know

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

MPI-IO access template

C

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

Fortran

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

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

Dataset transfer property

C

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

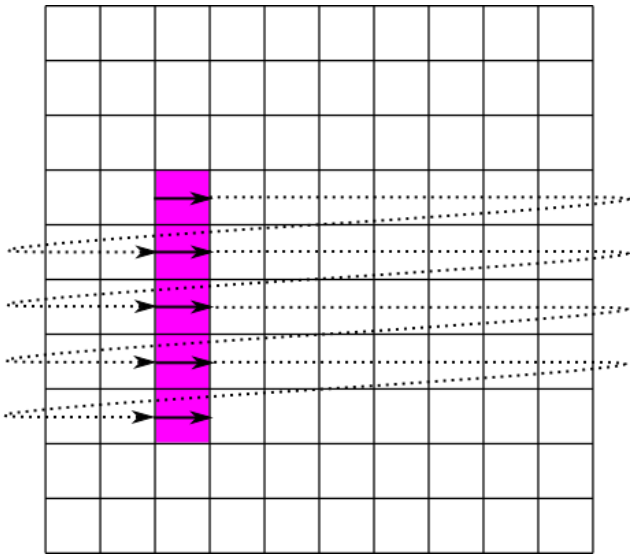
Fortran

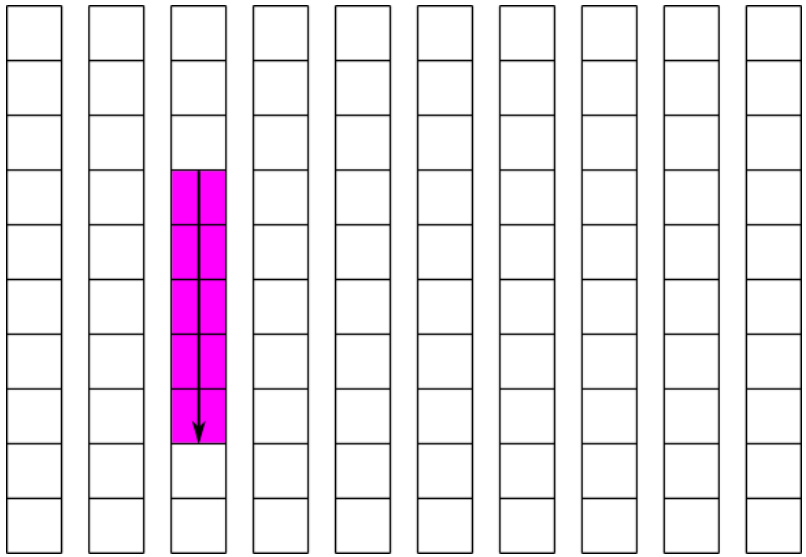
```
H5PCREATE_F(CLASSTYPE, PRP_ID, HDFERR)
  INTEGER, INTENT(IN) :: CLASSTYPE
  INTEGER(HID_T), INTENT(OUT) :: PRP_ID
  INTEGER, INTENT(OUT) :: HDFERR
H5PSET_DXPL_MPIO_F(PRP_ID, DATA_XFER_MODE, HDFERR)
  INTEGER(HID_T), INTENT(IN) :: PRP_ID
  INTEGER, INTENT(IN) :: DATA_XFER_MODE
  INTEGER, INTENT(OUT)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

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

Performance hints

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





C

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

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

Exercise

Exercise 4 – 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`

Exercise - Job execution

Jobscript:

```
#!/bin/sh
#PBS -lnodes=1:ppn=8

cd $PBS_O_WORKDIR

. /opt/torque/etc/openmpi-setup.sh

export LD_LIBRARY_PATH=/share/apps/hdf5/lib:$LD_LIBRARY_PATH
mpirun -x LD_LIBRARY_PATH ./a.out #use the name of your executable
```

Job start:

```
qsub name_of_jobscript
```