MPI-IO

Escola de Verão 2021
Supercomputador Santos Dumont

MC Introdução a Programação MPI com Extensões para E/S

André Ramos Carneiro

SERED COTIC LNCC – Laboratório Nacional de Computação Científica

Agenda

Notions

- I/O for HPC
- MPI-IO
- Terminology

File Manipulation

- Open / Create
- Access Mode (amode)
- Close
- File Views*

Individual Operations

Collective Operations

- Explicit Offsets
- Individual File Pointers
- Shared File Pointers

Hints

- File Info
- MPI-I/O Hints
- Data Seiving
- Collective Buffering

* This item will be revisited before learning individual file pointers for noncollective operations

Course Materials - slides and source code

http://www.lncc.br/~andrerc/mpi-io-lncc-2021.tar.gz

http://www.lncc.br/~andrerc/mpi-io-lncc-2021.zip

- **MPI-IO – LNCC 2021 – Slides.pdf**
- **`base.c`:** Basic template file.
- **`write-i-offsets-character.c`**: 1st hands-on to write a file using **independent** + **explicit offsets**
- **`read-i-offsets-character.c`**: 2nd hands-on to read a file using **independent** + **explicit offsets**
- **`write-i-ifp-double-buffer.c`**: 3rd hands-on to write a file using **independent** + **individual file pointers**
- **`write-c-ifp-view-subarray-datatype-double.c`**: 4th hands-on to write a file using **collective** + **individual file pointers** + **view** + **datatype**
- **`write-c-ifp-view-subarray-datatype-double-challenge.c`:** Modified version of the previous file
- **`1D_parallel_julia.c`**: 5th hands=on – Challenge to transform a **sequential** writing into a **parallel** one
- **`get-all-hints.c`:** 6th hands-on to print the **MPI-IO hints**
- **`hints.txt`**: Used on the 6th hands-on

Quick Ref.

Before we start, some quick references

- OpenMPI Documentation: https://www.open-mpi.org/doc
- MPI Standard Official Documentation: https://www.mpi-forum.org/docs
- DeinoMPI (An implementation of MPI-2 for Windows): http://mpi.deino.net/mpi_functions/index.htm
- ROMIO: https://www.mcs.anl.gov/projects/romio
- MPICH Documatation: https://www.mpich.org/static/docs/latest
- SMPI CourseWare: https://simgrid.github.io/SMPI_CourseWare
- CENAPAD-RJ: http://www.cenapad-rj.lncc.br/tutoriais/materiais-hpc

For many applications, ***I/O is a bottleneck*** *that limits scalability. Write operations* ***often do not perform well*** *because an application's processes do not write data to Lustre in an efficient manner, resulting in* ***file contention*** *and* ***reduced parallelism****.*

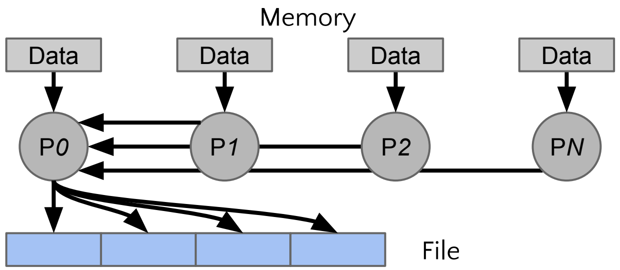
— Getting Started on MPI I/O, Cray, 2015 —

“

Notions

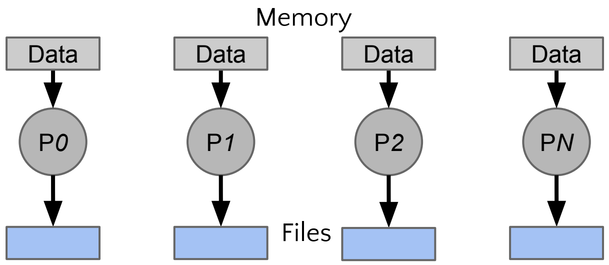
I/O for HPC
MPI-IO
Terminology

HPC & I/O



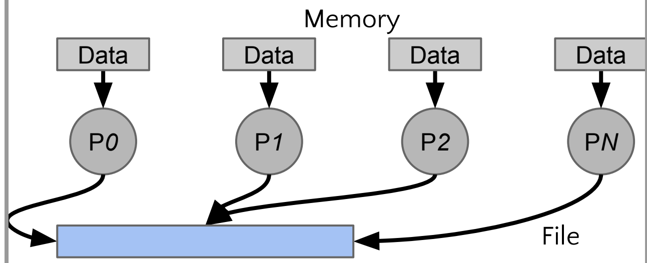
- **Serial I/O**

+ Simple for small I/O
+ No need for special I/O libraries
– Bandwidth limited by one client
– Not enough memory to hold all data
– Won't scale



- **Parallel I/O (N-N or M-N)**

+ No communication or coordination
+ Better scaling than Serial I/O
– More process -> More (small) files -> Limited Scaling
– Need for post-processing
– Uncoordinated I/O may overload filesystem (file locks!)

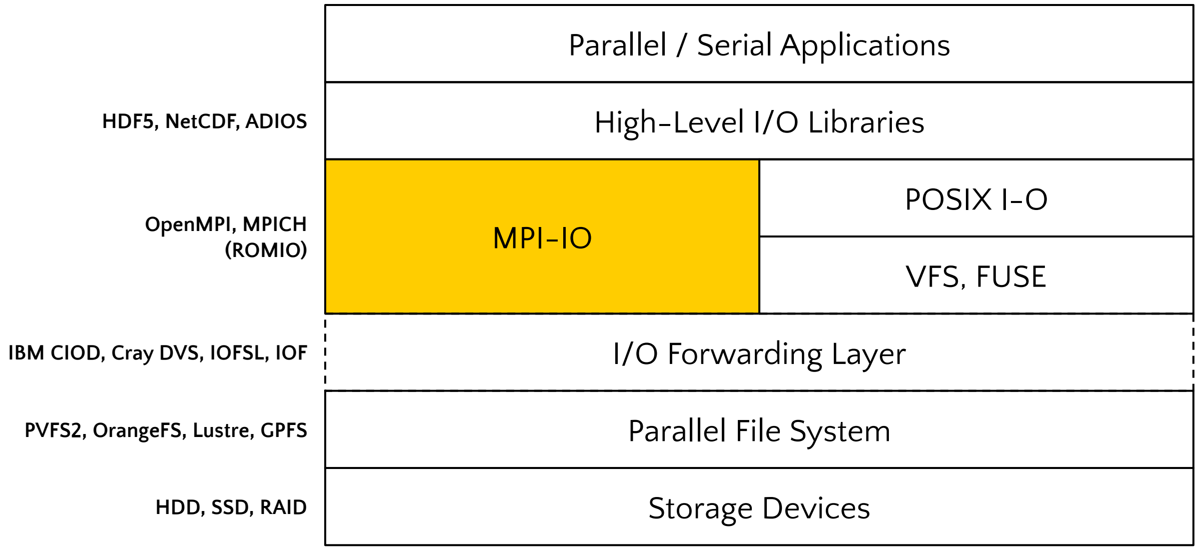


- **Parallel I/O (N-1)**

+ Only one file (visualization, management, storage)
+ No post-processing -> Scale!
– Requires more design and thought
– Uncoordinated I/O may overload filesystem (file locks!)

BUT MPI-IO CAN HELP HERE!

HPC I/O Stack



Inspired by Ohta et. a. (2010)

POSIX

- POSIX (Portable Operating System Interface for Unix)
    - Set of standards
    - Define the application programming interface (API)
    - Define some shell and utility interfaces
- POSIX I/O
    - Portion of the standard that defines the IO interface for POSIX
    - `read()`, `write()`, `open()`, `close()`, …

HPC I/O Stack

POSIX I/O

- A POSIX I/O file is simply a sequence of bytes
- POSIX I/O gives you full, low-level control of I/O operations
- There is little in the interface that inherently supports parallel I/O
- POSIX I/O does not support collective access to files
    - Programmer should coordinate access

HPC I/O Stack

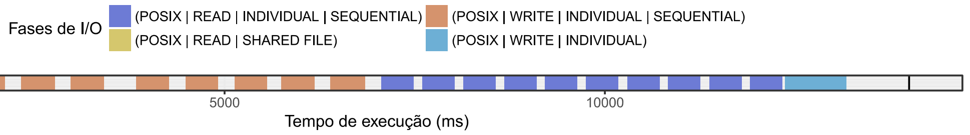
MPI-IO

- An MPI I/O file is an ordered collection of typed data items
- A higher level of data abstraction than POSIX I/O
- Define data models that are natural to your application
- You can define complex data patterns for parallel writes
- The MPI I/O interface provides independent and collective I/O calls
- Optimization of I/O functions

HPC I/O Stack

The MPI-IO Layer

- MPI-IO layer introduces an important optimization: collective I/O
- HPC programs often has distinct phases where all process:
    - Compute
    - Perform I/O (read or write checkpoint)



- Uncoordinated access is hard to serve efficiently
- Collective operations allow MPI to coordinate and optimize accesses

HPC I/O Stack

The MPI-IO Layer

Collective I/O yields four key benefits:

- "Optimizations such as data sieving and two-phase I/O rearrange the **access pattern** to be more friendly to the underlying file system"
- "If processes have overlapping requests, library can eliminate duplicate work"
- "By coalescing multiple regions, the density of the I/O request increases, making the two-phase I/O optimization more efficient"
- "The I/O request can also be aggregated down to a number of nodes more suited to the underlying file system"

MPI-IO

MPI: A Message-Passing Interface Standard

MPI-IO

Version 3.0

- This course is based on the MPI Standard Version 3.0
- The examples and exercises were created with OpenMPI 3.0.0:
- Remember to include in your C code:

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

- Remember how to compile:

```
$ mpicc code.c -o code
```

- Remember how to run:

```
$ mpirun --hostfile HOSTFILE --oversubscribe --np PROCESSES ./code
```

MPI-IO

SDumont

- You can to use Version 2.0 in SDumont Supercomputer:

```
$ module avail
$ module load openmpi/gnu/2.0.4.2
```

- Compile normally:

```
$ mpicc code.c -o code
```

- Allocate some nodes (2 nodes and 8 ranks) and run the experiment:

```
$ salloc -p treinamento -N 2 -n 8
$ mpiexec ./code
```

- Or just use `srun`

```
$ srun -p treinamento -N 2 -n 8 ./code
```

Concepts of MPI-IO

Terminology

Concepts of MPI-IO

file e *displacement*

file

An MPI file is an ordered collection of typed data items (***etype***)

MPI supports random or sequential access to any integral set of items

A file is opened collectively by a group of processes

All *Collective I/O* calls on a file are collective over this group

displacement

Absolute **byte** position relative to the beginning of a file

Defines the location where a *file view* begins

Concepts of MPI-IO

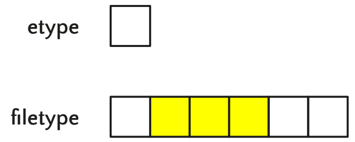
etype e *filetype*

etype

etype → elementary datatype

Unit of data access and positioning

It can be any MPI predefined or derived datatype



filetype

Basis for partitioning a file among processes

Defines a template for accessing the file

Single *etype* or derived *datatype* (multiple instances of same *etype*)

and *holes*...

MPI datatype	C datatype	MPI datatype	C datatype
`MPI_SHORT`	signed short int	`MPI_CHAR`	char (printable character)
`MPI_INT`	signed int	`MPI_UNSIGNED_CHAR`	unsigned char (integral value)
`MPI_LONG_LONG_INT`	signed long long int	`MPI_UNSIGNED_SHORT`	unsigned short int
`MPI_LONG_LONG` (as a synonym)	signed long long int	`MPI_UNSIGNED`	unsigned int
`MPI_FLOAT`	float	`MPI_UNSIGNED_LONG`	unsigned long int
`MPI_DOUBLE`	double	`MPI_UNSIGNED_LONG_LONG`	unsigned long long int
`MPI_LONG_DOUBLE`	long double	`MPI_BYTE`	

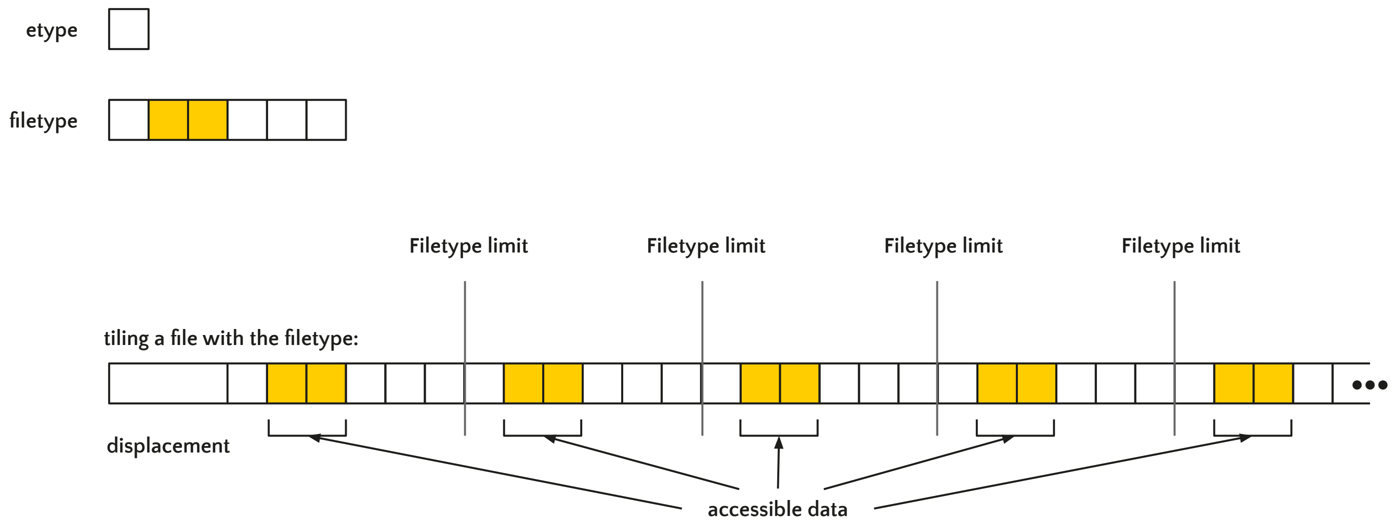
Principal MPI Datatypes

Concepts of MPI-IO

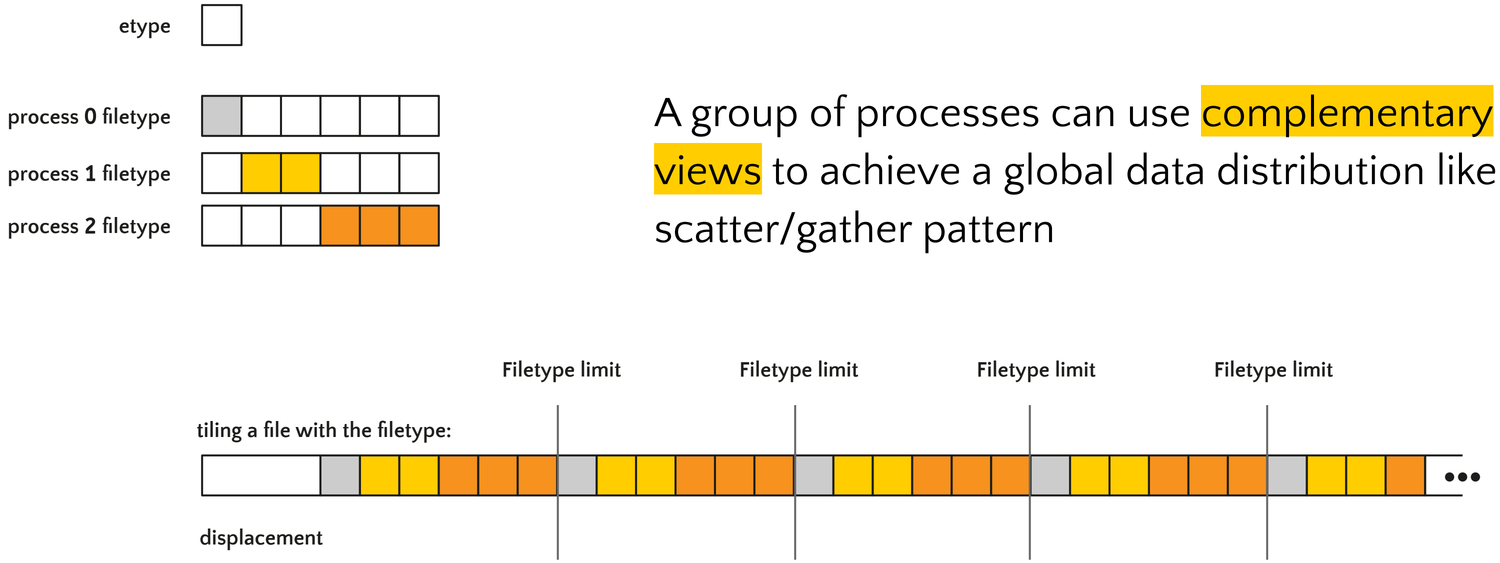
view

- Defines the current set of data visible and accessible from an open file
- Ordered set of *etypes*
- Each process has its own view, defined by:
    - a displacement
    - an *etype*
    - a filetype
- The pattern described by a filetype is repeated, beginning at the displacement, to define the view
- Default *view:* displacement 0, etype `MPI_BYTE` and filetype 1 `MPI_BYTE`

Concepts of MPI-IO
view



Concepts of MPI-IO
view



A group of processes can use complementary views to achieve a global data distribution like the scatter/gather pattern

Concepts of MPI-IO

offset e *file size*

offset

- Position in the file relative to the current *file view*
- Expressed as a count of ***etypes***
- Holes in the filetype are skipped when calculating

file size

- Size of MPI file is measured in bytes from the beginning of the file
- Newly created files have size zero

Concepts of MPI-IO

file pointer e *file handle*

file pointer

A file pointer is an implicit offset maintained by MPI

Individual pointers are local to each process

Shared pointers is shared among the group of process

file handle

Opaque object created by `MPI_FILE_OPEN`

Freed by `MPI_FILE_CLOSE`

All operation to an open file reference the file through the file handle

End of Notions

I/O for HPC
MPI-IO
Terminology

Any questions?

Next: File Manipulation

File Manipulation

Opening Files
Access Mode (`amode`)
Closing Files

File Manipulation

Opening Files

function in C

```
int MPI_File_open(
      MPI_Comm comm,           // IN       communicator (handle)
      const char *filename,    // IN       name of file to open (string)
      int amode,               // IN       file access mode (integer)
      MPI_Info info,           // IN       info object (handle)
      MPI_File *fh             //    OUT   new file handle (handle)
)
```

- `MPI_FILE_OPEN` is a collective routine
  - All process must provide the same value for filename and amode
  - `MPI_COMM_WORLD` or `MPI_COMM_SELF` (independently)
  - User must close the file before `MPI_FINALIZE`
- Initially all processes view the file as a linear byte stream

File Manipulation

Opening Files

```
int MPI_File_open(
      MPI_Comm comm,              // IN       communicator (handle)
      const char *filename,       // IN       name of file to open (string)
      int amode,                  // IN       file access mode (integer)
      MPI_Info info,              // IN       info object (handle)
      MPI_File *fh                //    OUT   new file handle (handle)
)
```

function name
it is not code!

- `MPI_FILE_OPEN` is a collective routine
  - All process must provide the same value for filename and amode
  - `MPI_COMM_WORLD` or `MPI_COMM_SELF` (independently)
  - User must close the file before `MPI_FINALIZE`
- Initially all processes view the file as a linear byte stream

File Manipulation

Opening Files

```
int MPI_File_open(
        MPI_Comm comm,              // IN       communicator (handle)
        const char *filename,       // IN       name of file to open (string)
        int amode,                  // IN       file access mode (integer)
        MPI_Info info,              // IN       info object (handle)
        MPI_File *fh                //    OUT   new file handle (handle)
)
```

- `MPI_FILE_OPEN` is a collective routine
  - All process must provide the same value for filename and amode
  - `MPI_COMM_WORLD` or `MPI_COMM_SELF` (independently)
  - User must close the file before `MPI_FINALIZE`
- Initially all processes *view* the file as a linear byte stream
- MPI_Info is used to pass hints. Pass `MPI_INFO_NULL` to use the default

File Manipulation

Access Mode

exactly one!

`MPI_MODE_RDONLY`	→ read only
`MPI_MODE_RDWR`	→ reading and writing
`MPI_MODE_WRONLY`	→ write only
`MPI_MODE_CREATE`	→ create the file if it does not exist
`MPI_MODE_EXCL`	→ error if creating file that already exists
`MPI_MODE_DELETE_ON_CLOSE`	→ delete file on close
`MPI_MODE_APPEND`	→ set initial position of all file pointers to end of file

```
*Combining access modes -> (MPI_MODE_CREATE|MPI_MODE_EXCL|MPI_MODE_RDWR)
```

File Manipulation

Closing Files

```
int MPI_File_close(
      MPI_File *fh             // IN       file handle (handle)
)
```

- `MPI_FILE_CLOSE` first synchronizes file state
  - Equivalent to performing an `MPI_FILE_SYNC`
  - For writes `MPI_FILE_SYNC` provides the only guarantee that data has been transferred to the storage device
- Then closes the file associated with `fh`
- `MPI_FILE_CLOSE` is a collective routine
- User is responsible for ensuring all requests have completed
- `fh` is set to `MPI_FILE_NULL`
- **MUST** be called **before** `MPI_FINALIZE`

Data Access

Positioning
Coordination
Synchronism

Data Access

Overview

- There are 3 aspects to data access:

positioning

Explicit offset – "We" decide where to…

Implicit file pointer – "MPI" handles it…

Individual: local to each process

Shared: … by the group of processes

synchronism

Blocking – "Stop" processing

Nonblocking – "Don't Stop"

Split Collective – "mix" of both

coordination

Noncollective

Collective – MPI "helps" here

- POSIX `read()/fread()` and `write()/fwrite()`
  - Blocking, noncollective operations with individual file pointers
  - `MPI_FILE_READ` and `MPI_FILE_WRITE` are the MPI equivalents

Data Access

Positioning

- We can use a mix of the three types in our code
- Routines that accept explicit offsets contain `_AT` in their name
- Implicit Individual file pointer routines contain no positional qualifiers
- Implicit Shared file pointer routines contain `_SHARED` or `_ORDERED` in the name
  - `_SHARED` for noncollective operations
  - `_ORDERED` for collective operations
- I/O operations leave the MPI file pointer pointing to next item (***etype***)
- In collective or split collective the pointer is updated by the call

Data Access

Synchronism

- Blocking calls
    - Will not return until the I/O request is completed
- Nonblocking calls
    - Initiates an I/O operation
    - Does not wait for it to complete
    - Need to send a request complete call (`MPI_WAIT` or `MPI_TEST`)
- Nonblocking calls are named `MPI_FILE_I...` with an `I` (immediate)
- We should not access the buffer until the operation is complete

Data Access

Coordination

- Every noncollective (*individual*) routine has a collective counterpart:
  - `MPI_FILE_...` is `MPI_FILE_..._ALL`
  - `MPI_FILE_..._SHARED` is `MPI_FILE_..._ORDERED`
- We also have a pair `MPI_FILE_..._BEGIN` and `MPI_FILE_..._END`
  - Split Collectvie routines
- Collective routines may perform much better
- Global data access have potential for automatic optimization

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	`MPI_File_read_at` `MPI_File_write_at`	`MPI_File_read_at_all` `MPI_File_write_at_all`
	nonblocking	`MPI_File_iread_at` `MPI_File_iwrite_at`	`MPI_File_iread_at_all` `MPI_File_iwrite_at_all`
	split collective	N/A	`MPI_File_read_at_all_begin/end` `MPI_File_write_at_all_begin/end`
individual file pointers	blocking	`MPI_File_read` `MPI_File_write`	`MPI_File_read_all` `MPI_File_write_all`
	nonblocking	`MPI_File_iread` `MPI_File_iwrite`	`MPI_File_iread_all` `MPI_File_iwrite_all`
	split collective	N/A	`MPI_File_read_all_begin/end` `MPI_File_write_all_begin/end`
shared file pointer	blocking	`MPI_File_read_shared` `MPI_File_write_shared`	`MPI_File_read_ordered` `MPI_File_write_ordered`
	nonblocking	`MPI_File_iread_shared` `MPI_File_iwrite_shared`	N/A
	split collective	N/A	`MPI_File_read_ordered_begin/end` `MPI_File_write_ordered_begin/end`

*Classification of **MPI-IO Functions** in C*

Data Access

Noncollective I/O

Explicit Offsets

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	`MPI_File_read_at` `MPI_File_write_at`	`MPI_File_read_at_all` `MPI_File_write_at_all`
	nonblocking	`MPI_File_iread_at` `MPI_File_iwrite_at`	`MPI_File_iread_at_all` `MPI_File_iwrite_at_all`
	split collective	N/A	`MPI_File_read_at_all_begin/end` `MPI_File_write_at_all_begin/end`
individual file pointers	blocking	`MPI_File_read` `MPI_File_write`	`MPI_File_read_all` `MPI_File_write_all`
	nonblocking	`MPI_File_iread` `MPI_File_iwrite`	`MPI_File_iread_all` `MPI_File_iwrite_all`
	split collective	N/A	`MPI_File_read_all_begin/end` `MPI_File_write_all_begin/end`
shared file pointer	blocking	`MPI_File_read_shared` `MPI_File_write_shared`	`MPI_File_read_ordered` `MPI_File_write_ordered`
	nonblocking	`MPI_File_iread_shared` `MPI_File_iwrite_shared`	N/A
	split collective	N/A	`MPI_File_read_ordered_begin/end` `MPI_File_write_ordered_begin/end`

*Classification of **MPI-IO Functions** in C*

Data Access - Noncollective

Explicit Offsets

```
int MPI_File_write_at(
      MPI_File fh,              // IN OUT   file handle (handle)
      MPI_Offset offset,        // IN       file offset (integer)
      const void *buf,          // IN       initial address of buffer (choice)
      int count,                // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,    // IN       datatype of each buffer element (handle)
      MPI_Status *status        //    OUT   status object (Status)
)
```

- `MPI_FILE_WRITE_AT`
  - write to a file associated with the **`fh`**
  - beginning at the position specified by **`offset`** (in *etype* units relative to the current view)
  - **`count`** number of elements, defined by the **`datatype,`** from the buffer **`buf`**
- buf should store at least as many values as **`count*sizeof(datatype)`**

Data Access - Noncollective

Explicit Offsets

```
int MPI_File_read_at(
      MPI_File fh,             // IN OUT   file handle (handle)
      MPI_Offset offset,       // IN       file offset (integer)
      void *buf,               //    OUT   initial address of buffer (choice)
      int count,               // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,   // IN       datatype of each buffer element (handle)
      MPI_Status *status       //    OUT   status object (Status)
)
```

- `MPI_FILE_READ_AT`
    - read from a file associated with the **`fh`**
    - beginning at the position specified by **`offset`** (in *etype* units relative to the current view)
    - **`count`** number of elements, defined by the **`datatype`**, to the buffer **`buf`**
- **`buf`** should store at least as many values as **`count*sizeof(datatype)`**

exercise or

experiment

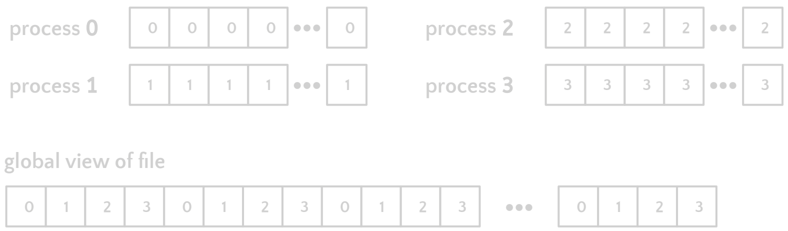
SOLUTION FILE

write-i-offsets-character.c

Hands-on!

WRITE - Explicit Offsets

Using explicit offsets (and default view) write a program where each process will print its rank, as a character, 10 times.



If we ran with 4 processes the file (you should create it) should contain:

```
$ cat my-rank.txt
0123012301230123012301230123012301230123
```

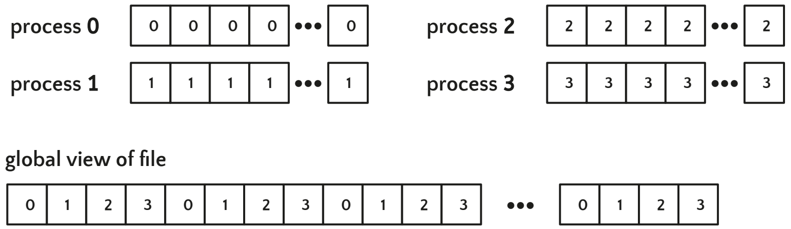
SOLUTION FILE

`write-i-offsets-character.c`

Hands-on!

WRITE - Explicit Offsets

Using explicit offsets (and default view) write a program where each process will print its rank, as a character, 10 times.



If we ran with 4 processes the file (you should create it) should contain:

```
$ cat my-rank.txt
0123012301230123012301230123012301230123
```

Hands-on!

WRITE - Explicit Offsets

SOLUTION FILE

`write-i-offsets-character.c`

```
$ vi write-i-offsets-character.c
        // Function to Open the file - HINT: verify the access modes
        // TO-DO

        int i=0;
        for (i=0; i<10; i++) {
                // Calculates the offset - HINT: offset = Position in the file expressed as a count of etypes
                // TO-DO

                // Define the character rank - HINT: Write it as character
                // TO-DO

                // Function to Write the character at the defined offset
                // TO-DO
        }

        // Function to Close the file
        // TO-DO
```

SOLUTION FILE
`write-i-offsets-character.c`

Hands-on!

WRITE - Explicit Offsets

- Compile and execute locally:

```
$ mpicc write-i-offsets-character.c -o write-i-offsets-character
$ mpirun --oversubscribe --np 4 ./write-i-offsets-character

$ cat my-rank.txt
0123012301230123012301230123012301230123
```

- Execute on SDumont (verify if the OpenMPI module is **loaded**):

```
$ srun -p treinamento -N 1 -n 4 ./write-i-offsets-character
$ cat my-rank.txt
0123012301230123012301230123012301230123
```

SOLUTION FILE

`read-i-offsets-character.c`

Hands-on!

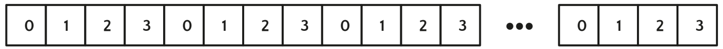
READ - Explicit Offsets

Modify your program so that each process will open the previous written file and read the printed ranks, as a character, 10 times using explicit offsets (and default view).

Remember to open the file to read only!

Each process should print to `stdout` the values.

global view of file



```
rank: 2, offset: 120, read: 2
rank: 1, offset: 004, read: 1
rank: 2, offset: 136, read: 2
rank: 2, offset: 152, read: 2
rank: 0, offset: 000, read: 0
...
```

SOLUTION FILE

read-i-offsets-character.c

Hands-on!

READ - Explicit Offsets

```
$ vi read-i-offsets-character.c
        // Function to Open the file - HINT: verify the access modes
        // TO-DO

        int i=0;
        for (i=0; i<10; i++) {
                // Calculates the offset - HINT: offset = Position in the file expressed as a count of etypes
                // TO-DO

                // Function to Read the character at the defined offset
                // TO-DO

                // Print the rank, offset, and value
                // TO-DO
        }

        // Function to Close the file
        // TO-DO
```

SOLUTION FILE
`read-i-offsets-character.c`

Hands-on!

READ - Explicit Offsets

- Compile and execute locally:

```
$ mpicc read-i-offsets-character.c -o read-i-offsets-character
$ mpirun --oversubscribe --np 4 ./read-i-offsets-character
rank: 1, offset: 004, read: 1
rank: 2, offset: 152, read: 2
rank: 0, offset: 000, read: 0
...
```

- Execute on SDumont (verify if the OpenMPI module is **loaded**):

```
$ srun -p treinamento -N 1 -n 4 ./read-i-offsets-character
$ cat slurm-JOBID.out
rank: 1, offset: 004, read: 1
rank: 2, offset: 152, read: 2
rank: 0, offset: 000, read: 0
...
```

Before we continue

Any questions?

Revisiting…

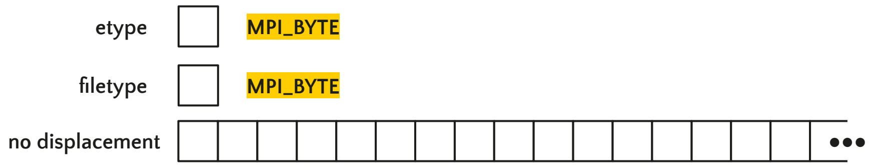
File Manipulation

File View
Data Types
Data Representation

File Manipulation

Default File View

- Unless explicitly specified, the default file view is:
  - A linear byte steam
  - `displacement` is set to zero
  - `etype` is set to `MPI_BYTE`
  - `filetype` is set to `MPI_BYTE`
  - This is the same for all the processes that opened the file
  - i.e. each process initially sees the whole file



File Manipulation

File Views

```
int MPI_File_set_view(
      MPI_File fh,              // IN OUT   file handle (handle)
      MPI_Offset disp,          // IN       displacement (integer)
      MPI_Datatype etype,       // IN       elementary datatype (handle)
      MPI_Datatype filetype,    // IN       filetype (handle)
      const char *datarep,      // IN       data representation (string)
      MPI_Info info             // IN       info object (handle)
)
```

- `MPI_FILE_SET_VIEW` is a collective operation
- Changes the process's view of the data (and resets *file pointers* to **zero**)
- Values for `disp`, `filetype` and `info` may vary
  - `datarep` and `etype` **must** be the same!
- `disp` is the absolute offset in bytes from where the view begins
- If `amode` = `MPI_MODE_SEQUENTIAL` than `disp` = `MPI_DISPLACEMENT_CURRENT`
- Multiple file views is possible

File Manipulation

Datatypes

- A general *datatype* is an opaque object that specifies two things:
  - A sequence of basic datatypes
  - A sequence of integer (byte) displacements
- e.g. `MPI_INT` is a predefined handle to a datatype with:
  - One entry of type `int`
  - Displacement equals to zero
- The other basic MPI *datatypes* are similar
- We can also create derived datatypes
- Transfer "*bulk*" data in single operation X many operations for small transfers

File Manipulation

Datatypes

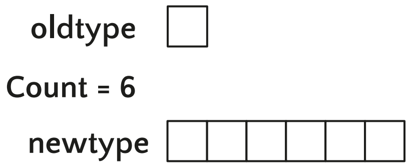
MPI Function	To create a...
`MPI_Type_contiguous`	contiguous datatype
`MPI_Type_vector`	vector (strided) datatype
`MPI_Type_create_indexed`	indexed datatype
`MPI_Type_create_indexed_block`	indexed datatype w/uniform block length
`MPI_Type_create_struct`	structured datatype
`MPI_Type_create_resized`	type with new extent and bounds
`MPI_Type_create_darray`	distributed array datatype
`MPI_Type_create_subarray`	n-dim subarray of an n-dim array

File Manipulation

Datatype Constructors

```
int MPI_Type_contiguous(
      int count,                // IN       replication count (non-negative integer)
      MPI_Datatype oldtype,     // IN       old datatype (handle)
      MPI_Datatype *newtype     //    OUT   new datatype (handle)
)
```

- `MPI_TYPE_CONTIGUOUS` is the simplest datatype constructor
- Allows replication of a datatype into contiguous locations
- `newtype` is the concatenation of `count` copies of `oldtype`



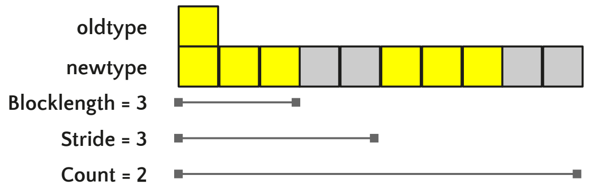
File Manipulation

Datatype Constructors

```
int MPI_Type_vector(
      int count,                // IN      number of blocks (non-negative integer)
      int blocklength,          // IN      number of elements in each block (non-negative integer)
      int stride,               // IN      number of elements between start of each block (integer)
      MPI_Datatype oldtype,     // IN      old datatype (handle)
      MPI_Datatype *newtype     //   OUT   new datatype (handle)
)
```

- `MPI_TYPE_VECTOR` replication into locations of equally spaced blocks
- Each block is the concatenation of copies of `oldtype`
- Spacing between blocks is multiple of `oldtype`

Count = 6



File Manipulation

Datatype Constructors

```
int MPI_Type_create_subarray(
      int ndims,                    // IN       number of array dimensions (positive integer)
      const int array_sizes[],      // IN       number of elements of in each dimension
      const int array_subsizes[],   // IN       number of elements of oldtype in each dimension
      const int array_starts[],     // IN       start coordinates of subarray in each dimension
      int order,                    // IN       array storage order flag (state)
      MPI_Datatype oldtype,         // IN       array element datatype (handle)
      MPI_Datatype *newtype         //     OUT  new datatype (handle)
)
```

- Describes an *n*-dimensional subarray of an *n*-dimensional array
- Facilitates access to arrays distributed in blocks among processes to a single shared file that contains the global array (I/O)
- The order in C is `MPI_ORDER_C` (row-major order)

File Manipulation

Datatypes

- A *datatype* object has to be committed before it can be used

```
int MPI_Type_commit(
      MPI_Datatype *datatype  // IN OUT   datatype that is committed (handle)
)
```

- There is no need to commit basic datatypes!
- To free a datatype we should use:

```
int MPI_Type_free(
      MPI_Datatype *datatype  // IN OUT   datatype that is freed (handle)
)
```

File Manipulation

Data Representation

- MPI supports multiple data representations (***datarep***):
    - `“native”`
    - `“internal”`
    - `“external32”`
- `“native”` and `“internal”` are implementation dependent (OpenMPI, MPICH, etc)
- `“external32”` is common to all MPI implementations
    - Intended to facilitate file interoperability

File Manipulation
Data Representation

`“native”`

- Data in this representation is stored in a file exactly as it is in memory
- Homogeneous systems:
  - No loss in precision or I/O performance due to type conversions
- On heterogeneous systems
  - Loss of interoperability

`“internal”`

- Data stored in implementation specific format
- Can be used with homogeneous or heterogeneous environments
- Implementation will perform type conversions if necessary

File Manipulation

Data Representation

`“external32”`

- Follows standardized representation (IEEE)
- All input/output operations are converted from/to the `“external32”`
- Files can be exported/imported between different MPI environments
- I/O performance may be lost due to type conversions
- `“internal”` may be implemented as equal to `“external32”`
- Can be read/written also by non-MPI programs

End of File Manipulation

Opening Files / Access Mode (`amode`) /Closing Files

File View / Data Types /Data Representation

Any questions?

Next: Operations

Data Access

Noncollective I/O

Individual File Pointers
Shared File Pointers

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	nonblocking	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
	split collective	N/A	MPI_File_read_at_all_begin/end MPI_File_write_at_all_begin/end
individual file pointers	blocking	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	nonblocking	MPI_File_iread MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
	split collective	N/A	MPI_File_read_all_begin/end MPI_File_write_all_begin/end
shared file pointer	blocking	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	nonblocking	MPI_File_iread_shared MPI_File_iwrite_shared	N/A
	split collective	N/A	MPI_File_read_ordered_begin/end MPI_File_write_ordered_begin/end

*Classification of **MPI-IO Functions** in C*

Data Access - Noncollective

Individual File Pointers

- MPI maintains one individual file pointer per process per file handle
- Implicitly specifies the offset
- Same semantics of the explicit offset routines
- Relative to the current view of the file

“After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next `etype` after the last one that will be accessed”

Data Access - Noncollective

Individual File Pointers

```
int MPI_File_write(
      MPI_File fh,               // IN OUT   file handle (handle)
      const void *buf,           // IN       initial address of buffer (choice)
      int count,                 // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,     // IN       datatype of each buffer element (handle)
      MPI_Status *status         //    OUT   status object (Status)
)
```

No offset is passed!

```
int MPI_File_read(
      MPI_File fh,               // IN OUT   file handle (handle)
      void *buf,                 //    OUT   initial address of buffer (choice)
      int count,                 // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,     // IN       datatype of each buffer element (handle)
      MPI_Status *status         //    OUT   status object (Status)
)
```

SOLUTION FILE
`write-i-ifp-double-buffer.c`

Hands-on!

WRITE - Individual Pointers

Using individual file pointers (and a *view*) write 100 double precision values `rank + (i / 100)`, one per line, per process. Write the entire buffer at once!

Attention: to view the file you should use **`hexdump`** or similar!

```
$ mpirun --oversubscribe --np 4 write-i-ifp-double-buffer
$ hexdump -v -e '10 "%f "' -e '"\n"' write-i-ifp-double-buffer.data
0,000000 0,010000 0,020000 0,030000 0,040000 0,050000 0,060000 0,070000 0,080000 0,090000
0,100000 0,110000 0,120000 0,130000 0,140000 0,150000 0,160000 0,170000 0,180000 0,190000
0,200000 0,210000 0,220000 0,230000 0,240000 0,250000 0,260000 0,270000 0,280000 0,290000
...
0,900000 0,910000 0,920000 0,930000 0,940000 0,950000 0,960000 0,970000 0,980000 0,990000
1,000000 1,010000 1,020000 1,030000 1,040000 1,050000 1,060000 1,070000 1,080000 1,090000
```

rank   i / 100

SOLUTION FILE

`write-i-ifp-double-buffer.c`

Hands-on!

WRITE - Individual Pointers

```
$ vi write-i-ifp-double-buffer.c
        // Function to Open the file - HINT: verify the access modes
        // TO-DO

        // Fill the buffer - HINT: Rebember, it's in double value!
        int i=0;
        for (i=0; i<BUFFER_SIZE; i++) {
                // TO-DO
        }

        // Set a displacement of the buffer size for each process based on its rank - HINT: change the file view
        // TO-DO

        // Function to Write with individual file pointer the entire buffer at once
        // TO-DO

        // Function to Close the file
        // TO-DO
```

```
$ mpicc write-i-ifp-double-buffer.c -o write-i-ifp-double-buffer
$ mpirun --oversubscribe --np 4 write-i-ifp-double-buffer
$ hexdump -v -e '10 "%.2f "' -e '"\n"' write-i-ifp-double-buffer.data
```

```
0,00 0,01 0,02 0,03 0,04 0,05 0,06 0,07 0,08 0,09
0,10 0,11 0,12 0,13 0,14 0,15 0,16 0,17 0,18 0,19
0,20 0,21 0,22 0,23 0,24 0,25 0,26 0,27 0,28 0,29
0,30 0,31 0,32 0,33 0,34 0,35 0,36 0,37 0,38 0,39
0,40 0,41 0,42 0,43 0,44 0,45 0,46 0,47 0,48 0,49
0,50 0,51 0,52 0,53 0,54 0,55 0,56 0,57 0,58 0,59
0,60 0,61 0,62 0,63 0,64 0,65 0,66 0,67 0,68 0,69
0,70 0,71 0,72 0,73 0,74 0,75 0,76 0,77 0,78 0,79
0,80 0,81 0,82 0,83 0,84 0,85 0,86 0,87 0,88 0,89
0,90 0,91 0,92 0,93 0,94 0,95 0,96 0,97 0,98 0,99
1,00 1,01 1,02 1,03 1,04 1,05 1,06 1,07 1,08 1,09
1,10 1,11 1,12 1,13 1,14 1,15 1,16 1,17 1,18 1,19
1,20 1,21 1,22 1,23 1,24 1,25 1,26 1,27 1,28 1,29
1,30 1,31 1,32 1,33 1,34 1,35 1,36 1,37 1,38 1,39
1,40 1,41 1,42 1,43 1,44 1,45 1,46 1,47 1,48 1,49
1,50 1,51 1,52 1,53 1,54 1,55 1,56 1,57 1,58 1,59
1,60 1,61 1,62 1,63 1,64 1,65 1,66 1,67 1,68 1,69
1,70 1,71 1,72 1,73 1,74 1,75 1,76 1,77 1,78 1,79
1,80 1,81 1,82 1,83 1,84 1,85 1,86 1,87 1,88 1,89
1,90 1,91 1,92 1,93 1,94 1,95 1,96 1,97 1,98 1,99
2,00 2,01 2,02 2,03 2,04 2,05 2,06 2,07 2,08 2,09
2,10 2,11 2,12 2,13 2,14 2,15 2,16 2,17 2,18 2,19
2,20 2,21 2,22 2,23 2,24 2,25 2,26 2,27 2,28 2,29
2,30 2,31 2,32 2,33 2,34 2,35 2,36 2,37 2,38 2,39
2,40 2,41 2,42 2,43 2,44 2,45 2,46 2,47 2,48 2,49
2,50 2,51 2,52 2,53 2,54 2,55 2,56 2,57 2,58 2,59
2,60 2,61 2,62 2,63 2,64 2,65 2,66 2,67 2,68 2,69
2,70 2,71 2,72 2,73 2,74 2,75 2,76 2,77 2,78 2,79
2,80 2,81 2,82 2,83 2,84 2,85 2,86 2,87 2,88 2,89
2,90 2,91 2,92 2,93 2,94 2,95 2,96 2,97 2,98 2,99
3,00 3,01 3,02 3,03 3,04 3,05 3,06 3,07 3,08 3,09
3,10 3,11 3,12 3,13 3,14 3,15 3,16 3,17 3,18 3,19
3,20 3,21 3,22 3,23 3,24 3,25 3,26 3,27 3,28 3,29
3,30 3,31 3,32 3,33 3,34 3,35 3,36 3,37 3,38 3,39
3,40 3,41 3,42 3,43 3,44 3,45 3,46 3,47 3,48 3,49
3,50 3,51 3,52 3,53 3,54 3,55 3,56 3,57 3,58 3,59
3,60 3,61 3,62 3,63 3,64 3,65 3,66 3,67 3,68 3,69
3,70 3,71 3,72 3,73 3,74 3,75 3,76 3,77 3,78 3,79
3,80 3,81 3,82 3,83 3,84 3,85 3,86 3,87 3,88 3,89
3,90 3,91 3,92 3,93 3,94 3,95 3,96 3,97 3,98 3,99
```

```
1,00 1,01 1,02 1,03 1,04 1,05 1,06 1,07 1,08 1,09
1,10 1,11 1,12 1,13 1,14 1,15 1,16 1,17 1,18 1,19
1,20 1,21 1,22 1,23 1,24 1,25 1,26 1,27 1,28 1,29
1,30 1,31 1,32 1,33 1,34 1,35 1,36 1,37 1,38 1,39
1,40 1,41 1,42 1,43 1,44 1,45 1,46 1,47 1,48 1,49
1,50 1,51 1,52 1,53 1,54 1,55 1,56 1,57 1,58 1,59
1,60 1,61 1,62 1,63 1,64 1,65 1,66 1,67 1,68 1,69
1,70 1,71 1,72 1,73 1,74 1,75 1,76 1,77 1,78 1,79
1,80 1,81 1,82 1,83 1,84 1,85 1,86 1,87 1,88 1,89
1,90 1,91 1,92 1,93 1,94 1,95 1,96 1,97 1,98 1,99
```

Data Access - Noncollective

Shared File Pointers

- MPI maintains exactly one shared file pointer per collective open
- Shared among processes in the communicator group
- Same semantics of the explicit offset routines
- Multiple calls to the shared pointer behaves as if they were serialized
- All processes must have the same view
- For **noncollective** operations order is not deterministic
  - If sync is needed, it's up to the programmer

Data Access - Noncollective

Shared File Pointers

```
int MPI_File_write_shared(
      MPI_File fh,              // IN OUT   file handle (handle)
      const void *buf,          // IN       initial address of buffer (choice)
      int count,                // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,    // IN       datatype of each buffer element (handle)
      MPI_Status *status        //    OUT   status object (Status)
)
```

```
int MPI_File_read_shared(
      MPI_File fh,              // IN OUT   file handle (handle)
      void *buf,                //    OUT   initial address of buffer (choice)
      int count,                // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,    // IN       datatype of each buffer element (handle)
      MPI_Status *status        //    OUT   status object (Status)
)
```

Before we continue

Any questions?

Data Access

Collective I/O

Explicit Offsets
Individual File Pointers
Shared File Pointers

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	`MPI_File_read_at` `MPI_File_write_at`	`MPI_File_read_at_all` `MPI_File_write_at_all`
	nonblocking	`MPI_File_iread_at` `MPI_File_iwrite_at`	`MPI_File_iread_at_all` `MPI_File_iwrite_at_all`
	split collective	N/A	`MPI_File_read_at_all_begin/end` `MPI_File_write_at_all_begin/end`
individual file pointers	blocking	`MPI_File_read` `MPI_File_write`	`MPI_File_read_all` `MPI_File_write_all`
	nonblocking	`MPI_File_iread` `MPI_File_iwrite`	`MPI_File_iread_all` `MPI_File_iwrite_all`
	split collective	N/A	`MPI_File_read_all_begin/end` `MPI_File_write_all_begin/end`
shared file pointer	blocking	`MPI_File_read_shared` `MPI_File_write_shared`	`MPI_File_read_ordered` `MPI_File_write_ordered`
	nonblocking	`MPI_File_iread_shared` `MPI_File_iwrite_shared`	N/A
	split collective	N/A	`MPI_File_read_ordered_begin/end` `MPI_File_write_ordered_begin/end`

Classification of ***MPI-IO Functions*** *in C*

Data Access - Collective

Explicit Offsets

```
int MPI_File_write_at_all(
      MPI_File fh,              // IN OUT   file handle (handle)
      MPI_Offset offset,        // IN       file offset (integer)
      const void *buf,          // IN       initial address of buffer (choice)
      int count,                // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,    // IN       datatype of each buffer element (handle)
      MPI_Status *status        //    OUT   status object (Status)
)
```

```
int MPI_File_read_at_all(
      MPI_File fh,              // IN OUT   file handle (handle)
      MPI_Offset offset,        // IN       file offset (integer)
      void *buf,                // OUT      initial address of buffer (choice)
      int count,                // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,    // IN       datatype of each buffer element (handle)
      MPI_Status *status        //    OUT   status object (Status)
)
```

Data Access - Collective

Individual File Pointers

```
int MPI_File_write_all(
      MPI_File fh,              // IN OUT   file handle (handle)
      const void *buf,          // IN       initial address of buffer (choice)
      int count,                // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,    // IN       datatype of each buffer element (handle)
      MPI_Status *status        //    OUT   status object (Status)
)
```

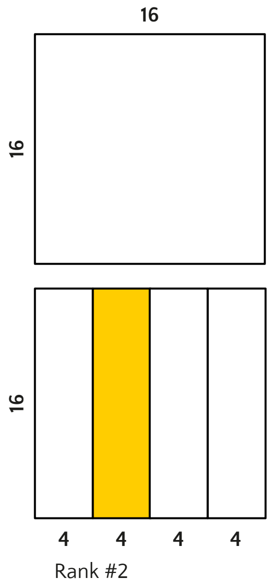
```
int MPI_File_read_all(
      MPI_File fh,              // IN OUT   file handle (handle)
      void *buf,                //    OUT   initial address of buffer (choice)
      int count,                // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,    // IN       datatype of each buffer element (handle)
      MPI_Status *status        //    OUT   status object (Status)
)
```

SOLUTION FILE

`write-c-ifp-view-subarray-datatype-double.c`

Hands-on!

WRITE - Subarray Datatype



- Consider a global matrix of 16 X 16 and 4 processes (easy to visualize)
- Divide the domain into 4 parts of 16 X 4 (local matrix)
- Make each process fill its local matrix with `(rank + (count / 100.0))`
- Where `count` is a counter of cells when iterating
- Create a new filetype to write the subarray of doubles
- Define a view based on the subarray filetype you created
- Each process should write its subarray in a collective operation using individual file pointers
- Make sure you are using `MPI_ORDER_C` to store in row-major order
- Use `hexdump` to view your file and make sure it is correct!

SOLUTION FILE

`write-c-ifp-view-subarray-datatype-double.c`

Hands-on!

WRITE - Subarray Datatype

```
$ vi write-c-ifp-view-subarray-datatype-double.c
        // Function to Open the file - HINT: verify the access modes
        // TO-DO
...
        MPI_Datatype filetype;

        // Function to Create the datatype - HINT: verify how to create a subarray datatype
        // TO-DO

        // Function to Commit the datatype
        // TO-DO

        // Function to Set the file view - HINT: attention to the etype and datatype used
        // TO-DO

#Continue in the next slide ->
```

SOLUTION FILE

`write-c-ifp-view-subarray-datatype-double.c`

Hands-on!

WRITE - Subarray Datatype

```c
	// Each process will fill its local array with the value of (myrank + (count / 100.0))
	for (i = 0; i < subsizes[0]; i++) {
		for (j = 0; j < subsizes[1]; j++) {
			// TO-DO
		}
	}

	// Function to Write the subarray - HINT: use the collective implicit offset function
	// TO-DO

	// Function to Close the file
	// TO-DO
```

```
$ mpicc write-c-ifp-view-subarray-datatype-double.c -o write-c-ifp-view-subarray-datatype-double
$ mpirun --oversubscribe --np 4 write-c-ifp-view-subarray-datatype-double
$ hexdump -v -e '16 "%f "' -e '"\n"' write-c-ifp-view-subarray-datatype-double.data

0,000000 0,010000 0,020000 0,030000 1,000000 1,010000 1,020000 1,030000 2,000000 2,010000 2,020000 2,030000 3,000000 3,010000 3,020000 3,030000
0,040000 0,050000 0,060000 0,070000 1,040000 1,050000 1,060000 1,070000 2,040000 2,050000 2,060000 2,070000 3,040000 3,050000 3,060000 3,070000
0,080000 0,090000 0,100000 0,110000 1,080000 1,090000 1,100000 1,110000 2,080000 2,090000 2,100000 2,110000 3,080000 3,090000 3,100000 3,110000
0,120000 0,130000 0,140000 0,150000 1,120000 1,130000 1,140000 1,150000 2,120000 2,130000 2,140000 2,150000 3,120000 3,130000 3,140000 3,150000
0,160000 0,170000 0,180000 0,190000 1,160000 1,170000 1,180000 1,190000 2,160000 2,170000 2,180000 2,190000 3,160000 3,170000 3,180000 3,190000
0,200000 0,210000 0,220000 0,230000 1,200000 1,210000 1,220000 1,230000 2,200000 2,210000 2,220000 2,230000 3,200000 3,210000 3,220000 3,230000
0,240000 0,250000 0,260000 0,270000 1,240000 1,250000 1,260000 1,270000 2,240000 2,250000 2,260000 2,270000 3,240000 3,250000 3,260000 3,270000
0,280000 0,290000 0,300000 0,310000 1,280000 1,290000 1,300000 1,310000 2,280000 2,290000 2,300000 2,310000 3,280000 3,290000 3,300000 3,310000
0,320000 0,330000 0,340000 0,350000 1,320000 1,330000 1,340000 1,350000 2,320000 2,330000 2,340000 2,350000 3,320000 3,330000 3,340000 3,350000
0,360000 0,370000 0,380000 0,390000 1,360000 1,370000 1,380000 1,390000 2,360000 2,370000 2,380000 2,390000 3,360000 3,370000 3,380000 3,390000
0,400000 0,410000 0,420000 0,430000 1,400000 1,410000 1,420000 1,430000 2,400000 2,410000 2,420000 2,430000 3,400000 3,410000 3,420000 3,430000
0,440000 0,450000 0,460000 0,470000 1,440000 1,450000 1,460000 1,470000 2,440000 2,450000 2,460000 2,470000 3,440000 3,450000 3,460000 3,470000
0,480000 0,490000 0,500000 0,510000 1,480000 1,490000 1,500000 1,510000 2,480000 2,490000 2,500000 2,510000 3,480000 3,490000 3,500000 3,510000
0,520000 0,530000 0,540000 0,550000 1,520000 1,530000 1,540000 1,550000 2,520000 2,530000 2,540000 2,550000 3,520000 3,530000 3,540000 3,550000
0,560000 0,570000 0,580000 0,590000 1,560000 1,570000 1,580000 1,590000 2,560000 2,570000 2,580000 2,590000 3,560000 3,570000 3,580000 3,590000
0,600000 0,610000 0,620000 0,630000 1,600000 1,610000 1,620000 1,630000 2,600000 2,610000 2,620000 2,630000 3,600000 3,610000 3,620000 3,630000
```

Output for 16 x 16 matrix and 4 processes

Data Access - Collective

Shared File Pointers

- MPI maintains exactly one shared file pointer per collective open `(MPI_FILE_OPEN)`
- Shared among processes in the communicator group
- Same semantics of the explicit offset routines
- Multiple calls to the shared pointer behaves as if they were serialized
- Order is deterministic
- Accesses to the file will be in the order determined by the **ranks**

Data Access - Collective

Shared File Pointers

```
int MPI_File_write_ordered(
      MPI_File fh,               // IN OUT   file handle (handle)
      const void *buf,           // IN       initial address of buffer (choice)
      int count,                 // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,     // IN       datatype of each buffer element (handle)
      MPI_Status *status         //    OUT   status object (Status)
)
```

```
int MPI_File_read_ordered(
      MPI_File fh,               // IN OUT   file handle (handle)
      void *buf,                 //    OUT   initial address of buffer (choice)
      int count,                 // IN       number of elements in buffer (integer)
      MPI_Datatype datatype,     // IN       datatype of each buffer element (handle)
      MPI_Status *status         //    OUT   status object (Status)
)
```

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	`MPI_File_read_at` `MPI_File_write_at`	`MPI_File_read_at_all` `MPI_File_write_at_all`
	nonblocking	`MPI_File_iread_at` `MPI_File_iwrite_at`	`MPI_File_iread_at_all` `MPI_File_iwrite_at_all`
	split collective	N/A	`MPI_File_read_at_all_begin/end` `MPI_File_write_at_all_begin/end`
individual file pointers	blocking	`MPI_File_read` `MPI_File_write`	`MPI_File_read_all` `MPI_File_write_all`
	nonblocking	`MPI_File_iread` `MPI_File_iwrite`	`MPI_File_iread_all` `MPI_File_iwrite_all`
	split collective	N/A	`MPI_File_read_all_begin/end` `MPI_File_write_all_begin/end`
shared file pointer	blocking	`MPI_File_read_shared` `MPI_File_write_shared`	`MPI_File_read_ordered` `MPI_File_write_ordered`
	nonblocking	`MPI_File_iread_shared` `MPI_File_iwrite_shared`	N/A
	split collective	N/A	`MPI_File_read_ordered_begin/end` `MPI_File_write_ordered_begin/end`

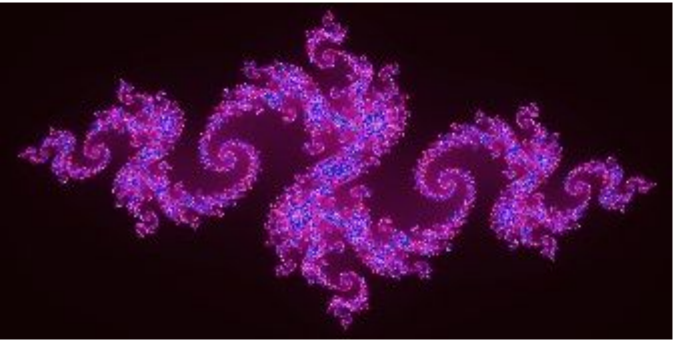
Classification of ***MPI-IO Functions*** *in C*

SOLUTION FILE

`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set



- Based on the "Activity #2" of "**Computing a Julia Set**":
  - https://simgrid.github.io/SMPI_CourseWare/topic_basics_of_distributed_memory_programming/julia_set
- Simple implementation of 1-D data distribution
  - File: `1D_parallel_julia.c`
- Each rank is responsible for calculating the entire rows, in sequence, of the Julia Set.
- The writing process is sequential, with only one rank writing its data at a time (`out_julia.bmp`).
- The challenge is to transform this sequential writing version into a parallel I/O one, using any of the previous techniques

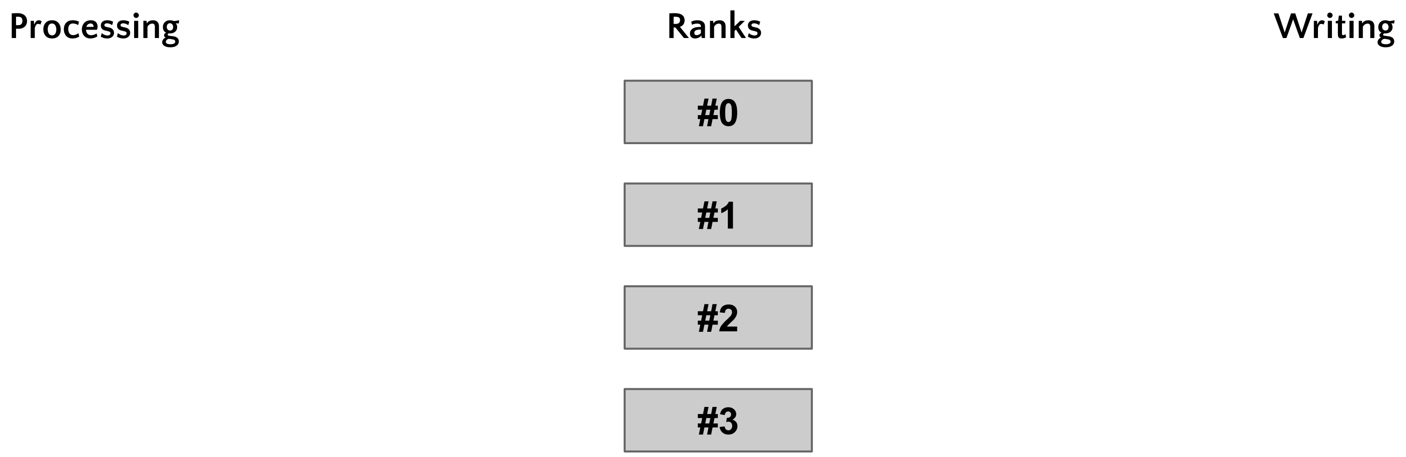
SOLUTION FILE

`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
$ mpicc 1D_parallel_julia.c -lm -o 1D_parallel_julia
$ mpirun -np 4 1D_parallel_julia 1000
[Process 0 out of 4]: I should compute pixel rows 0 to 249, for a total of 250 rows - subAREA 1500000…
[Process 1 out of 4]: I should compute pixel rows 250 to 499, for a total of 250 rows - subAREA 1500000...
[Process 2 out of 4]: I should compute pixel rows 500 to 749, for a total of 250 rows - subAREA 1500000...
[Process 3 out of 4]: I should compute pixel rows 750 to 999, for a total of 250 rows - subAREA 1500000...
```



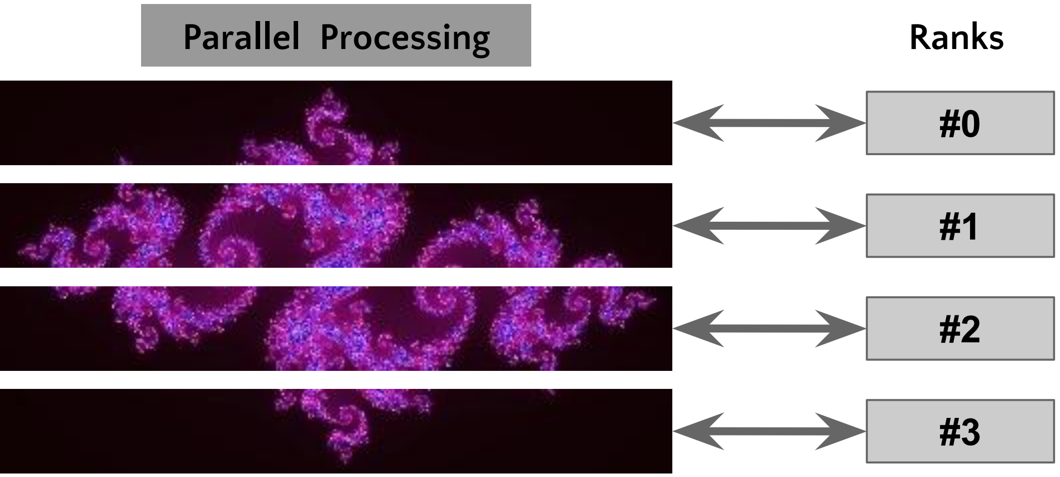
SOLUTION FILE

`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
$ mpicc 1D_parallel_julia.c -lm -o 1D_parallel_julia
$ mpirun -np 4 1D_parallel_julia 1000
[Process 0 out of 4]: I should compute pixel rows 0 to 249, for a total of 250 rows - subAREA 1500000…
[Process 1 out of 4]: I should compute pixel rows 250 to 499, for a total of 250 rows - subAREA 1500000...
[Process 2 out of 4]: I should compute pixel rows 500 to 749, for a total of 250 rows - subAREA 1500000...
[Process 3 out of 4]: I should compute pixel rows 750 to 999, for a total of 250 rows - subAREA 1500000...
```



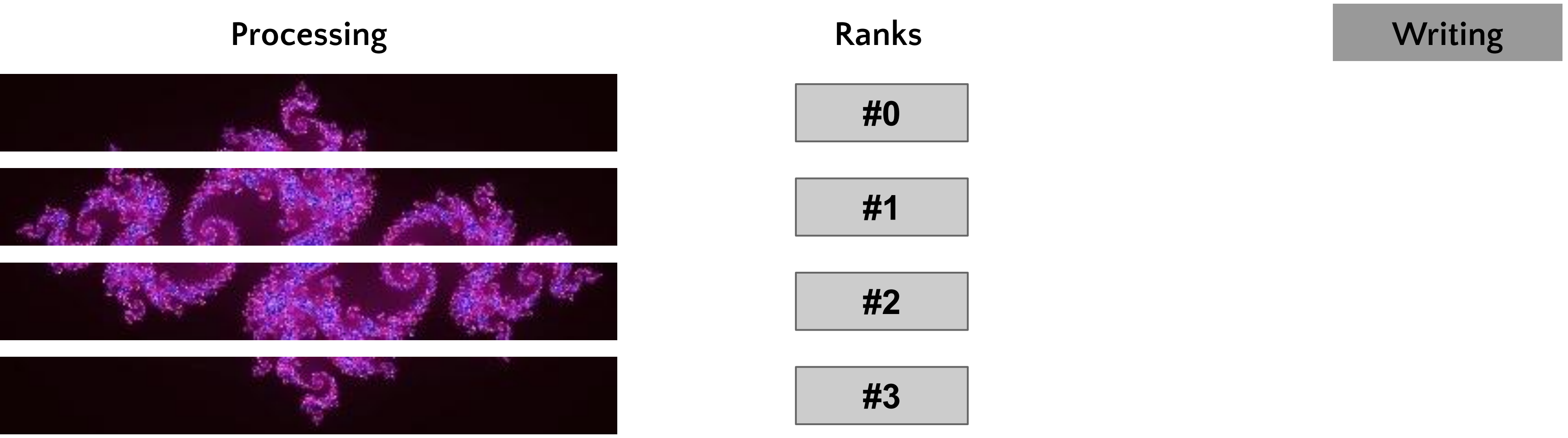
Writing

SOLUTION FILE
`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
$ mpicc 1D_parallel_julia.c -lm -o 1D_parallel_julia
$ mpirun -np 4 1D_parallel_julia 1000
[Process 0 out of 4]: I should compute pixel rows 0 to 249, for a total of 250 rows - subAREA 1500000…
[Process 1 out of 4]: I should compute pixel rows 250 to 499, for a total of 250 rows - subAREA 1500000...
[Process 2 out of 4]: I should compute pixel rows 500 to 749, for a total of 250 rows - subAREA 1500000...
[Process 3 out of 4]: I should compute pixel rows 750 to 999, for a total of 250 rows - subAREA 1500000...
```

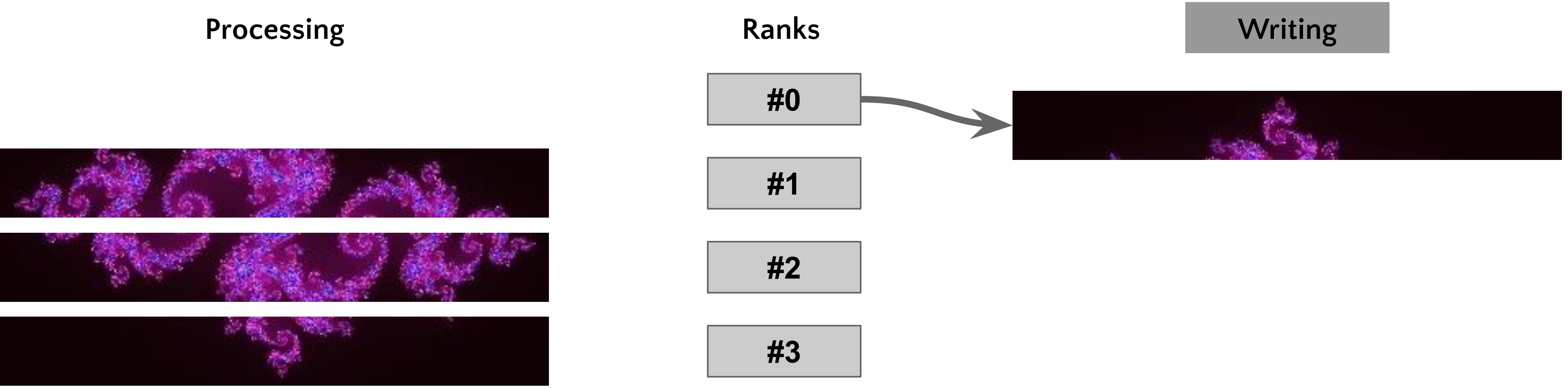


SOLUTION FILE
`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
$ mpicc 1D_parallel_julia.c -lm -o 1D_parallel_julia
$ mpirun -np 4 1D_parallel_julia 1000
[Process 0 out of 4]: I should compute pixel rows 0 to 249, for a total of 250 rows - subAREA 1500000…
[Process 1 out of 4]: I should compute pixel rows 250 to 499, for a total of 250 rows - subAREA 1500000...
[Process 2 out of 4]: I should compute pixel rows 500 to 749, for a total of 250 rows - subAREA 1500000...
[Process 3 out of 4]: I should compute pixel rows 750 to 999, for a total of 250 rows - subAREA 1500000...
```

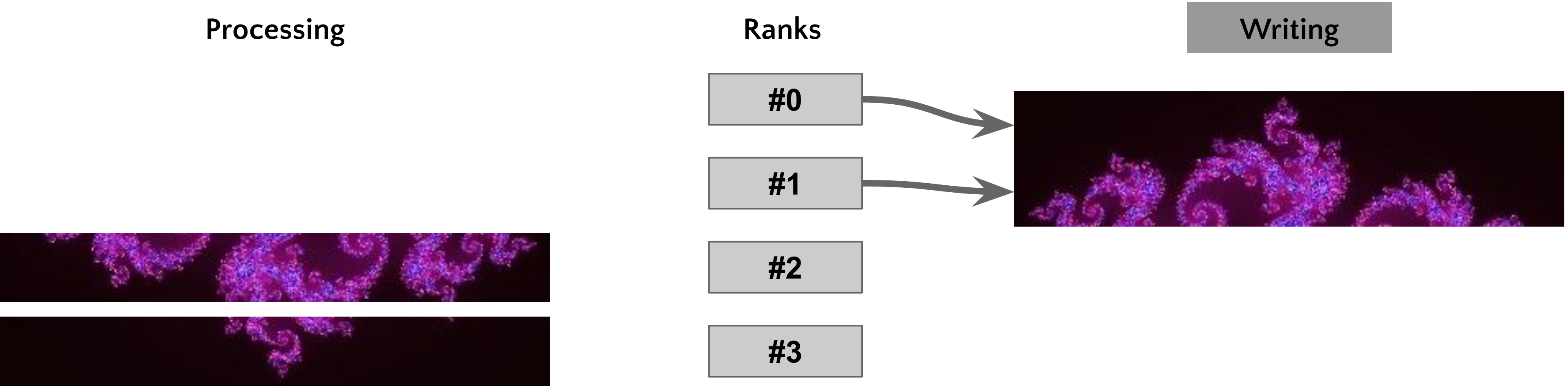


SOLUTION FILE
`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
$ mpicc 1D_parallel_julia.c -lm -o 1D_parallel_julia
$ mpirun -np 4 1D_parallel_julia 1000
[Process 0 out of 4]: I should compute pixel rows 0 to 249, for a total of 250 rows - subAREA 1500000…
[Process 1 out of 4]: I should compute pixel rows 250 to 499, for a total of 250 rows - subAREA 1500000...
[Process 2 out of 4]: I should compute pixel rows 500 to 749, for a total of 250 rows - subAREA 1500000...
[Process 3 out of 4]: I should compute pixel rows 750 to 999, for a total of 250 rows - subAREA 1500000...
```



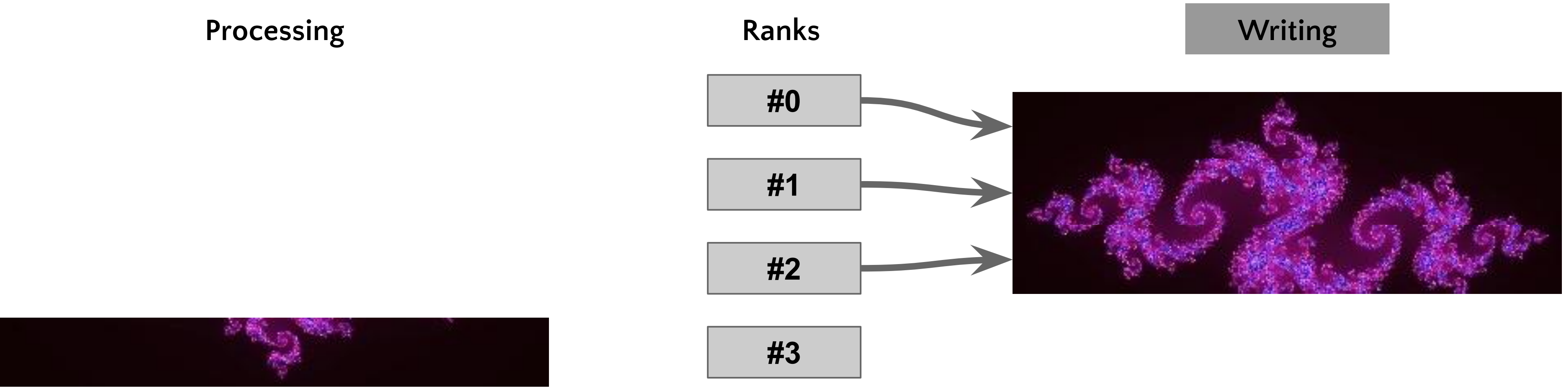
SOLUTION FILE

`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
$ mpicc 1D_parallel_julia.c -lm -o 1D_parallel_julia
$ mpirun -np 4 1D_parallel_julia 1000
[Process 0 out of 4]: I should compute pixel rows 0 to 249, for a total of 250 rows - subAREA 1500000…
[Process 1 out of 4]: I should compute pixel rows 250 to 499, for a total of 250 rows - subAREA 1500000...
[Process 2 out of 4]: I should compute pixel rows 500 to 749, for a total of 250 rows - subAREA 1500000...
[Process 3 out of 4]: I should compute pixel rows 750 to 999, for a total of 250 rows - subAREA 1500000...
```



SOLUTION FILE

`1D_parallel_julia_mpiio.c`

Hands-on!

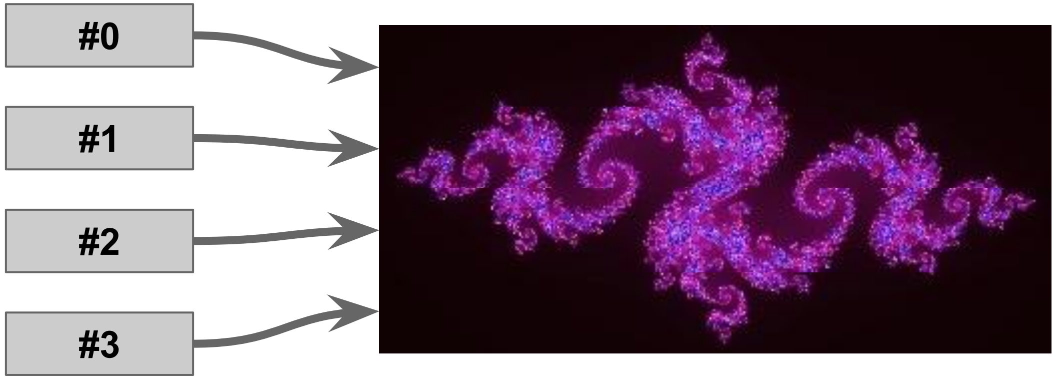
Challenge - Computing (*and writing*) a Julia Set

```
$ mpicc 1D_parallel_julia.c -lm -o 1D_parallel_julia
$ mpirun -np 4 1D_parallel_julia 1000
[Process 0 out of 4]: I should compute pixel rows 0 to 249, for a total of 250 rows - subAREA 1500000…
[Process 1 out of 4]: I should compute pixel rows 250 to 499, for a total of 250 rows - subAREA 1500000...
[Process 2 out of 4]: I should compute pixel rows 500 to 749, for a total of 250 rows - subAREA 1500000...
[Process 3 out of 4]: I should compute pixel rows 750 to 999, for a total of 250 rows - subAREA 1500000...
```

Processing

Ranks

Writing



SOLUTION FILE

`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
$ cp 1D_parallel_julia.c 1D_parallel_julia_mpiio.c
$ vi 1D_parallel_julia_mpiio.c
    //WRITE THE OUTPUT FILE

    if (myrank == 0) {
        // the first rank writes the header of the file
        char done[1]= {1};
        //write the file header
        output_file= fopen(OUTFILE, "w");
        write_bmp_header(output_file, WIDTH, HEIGHT);

        //write the local array in the file
        fwrite(local_pixel_array, sizeof(unsigned char), subAREA, output_file);
        fclose(output_file);
        free(local_pixel_array);

        //send the token to the next rank to write the output file
        MPI_Send(&done, 1, MPI_CHAR, myrank+1, 1, MPI_COMM_WORLD);

#continue in the next slide ->
```

SOLUTION FILE

1D_parallel_julia_mpiio.c

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
    } else {
        char done[1]= {1};
        //every other rank enters in a "wait" state, wating for its turn to write the data
        MPI_Recv(&done, 1, MPI_CHAR, myrank-1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        //only write if the rank actually processed some data
        if (subAREA > 0){

            //write the local array in the file
            output_file= fopen(OUTFILE, "a");
            fwrite(local_pixel_array, sizeof(unsigned char), subAREA, output_file);
            fclose(output_file);
            free(local_pixel_array);
        }
        if (myrank < size-1){
            //send the token to the next rank to write the output file
            MPI_Send(&done, 1, MPI_CHAR, myrank+1, 1, MPI_COMM_WORLD);
        }
    }
```

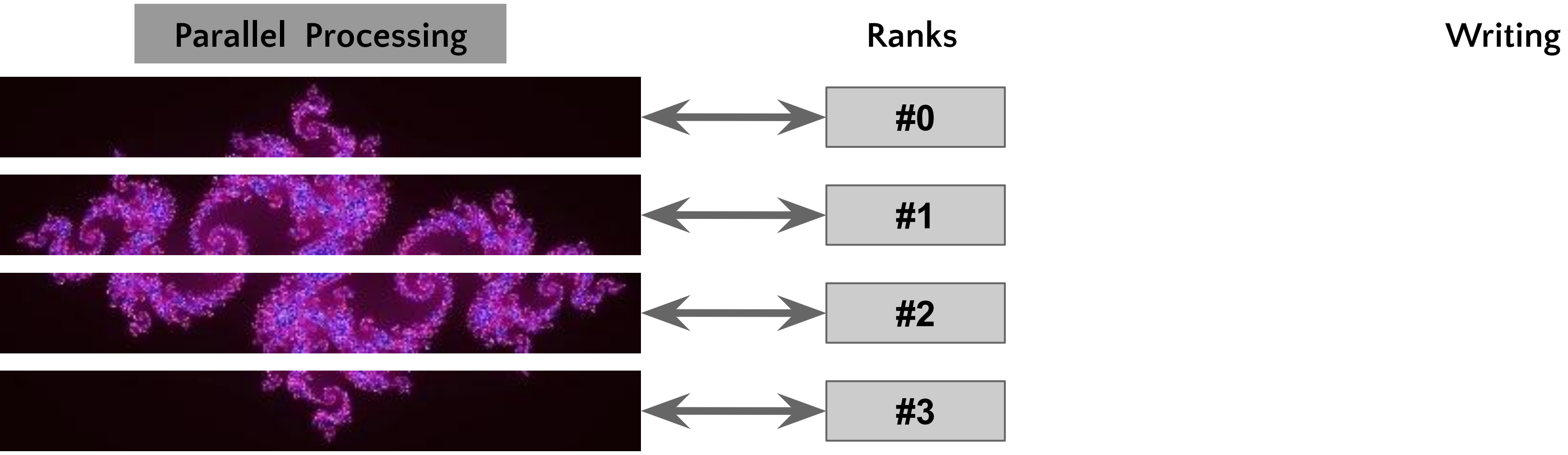
SOLUTION FILE

`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
$ mpicc 1D_parallel_julia_mpiio.c -lm -o 1D_parallel_julia_mpiio.c
$ mpirun -np 4 1D_parallel_julia_mpiio 1000
[Process 0 out of 4]: I should compute pixel rows 0 to 249, for a total of 250 rows - subAREA 1500000…
[Process 1 out of 4]: I should compute pixel rows 250 to 499, for a total of 250 rows - subAREA 1500000...
[Process 2 out of 4]: I should compute pixel rows 500 to 749, for a total of 250 rows - subAREA 1500000...
[Process 3 out of 4]: I should compute pixel rows 750 to 999, for a total of 250 rows - subAREA 1500000...
```



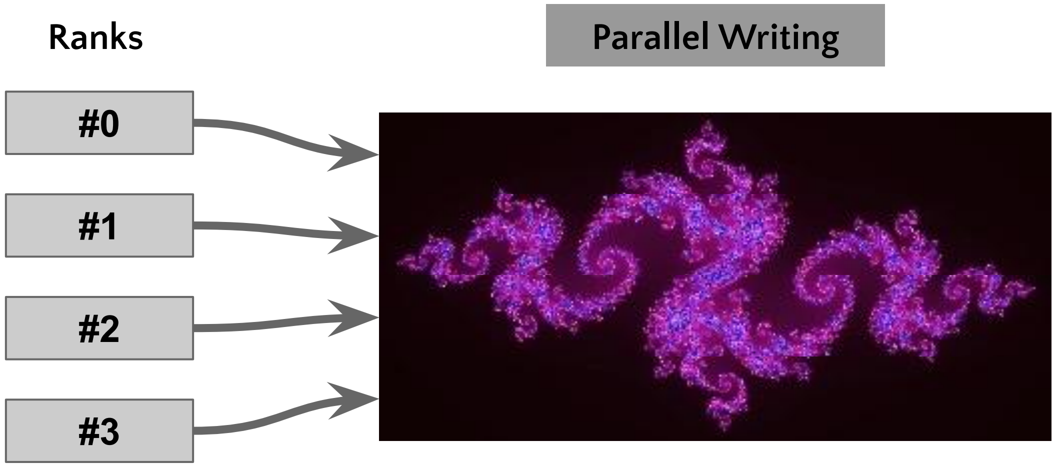
SOLUTION FILE
`1D_parallel_julia_mpiio.c`

Hands-on!

Challenge - Computing (*and writing*) a Julia Set

```
$ mpicc 1D_parallel_julia_mpiio.c -lm -o 1D_parallel_julia_mpiio.c
$ mpirun -np 4 1D_parallel_julia_mpiio 1000
[Process 0 out of 4]: I should compute pixel rows 0 to 249, for a total of 250 rows - subAREA 1500000…
[Process 1 out of 4]: I should compute pixel rows 250 to 499, for a total of 250 rows - subAREA 1500000...
[Process 2 out of 4]: I should compute pixel rows 500 to 749, for a total of 250 rows - subAREA 1500000...
[Process 3 out of 4]: I should compute pixel rows 750 to 999, for a total of 250 rows - subAREA 1500000...
```

Processing



End of Operations

Noncollective
Collective

Any questions?

Next: Hints

Hints

MPI-IO

Hints (ROMIO)

- Hints are `(key,value)` pairs
- Hints allow users to provide information on:
    - The access pattern to the files
    - Details about the file system
- The goal is to direct possible optimizations
- Applications may choose to ignore this hints

MPI-IO

Hints

- Hints are provided via `MPI_INFO` objects
- When no hint is provided you should use `MPI_INFO_NULL`
- Hints are informed, per file, in operations such as:

  `MPI_FILE_OPEN`, `MPI_FILE_DELETE`, `MPI_FILE_SET_VIEW` and `MPI_FILE_SET_INFO`

- Some hints cannot be overridden in operations such as:

  `MPI_FILE_SET_VIEW` and `MPI_FILE_SET_INFO`

Hints

Procedure

1. Create an info object with `MPI_INFO_CREATE`
2. Set the hint(s) with `MPI_INFO_SET`
3. Pass the info object to the I/O layer

   → through **`MPI_FILE_OPEN`**, `MPI_FILE_SET_VIEW` or `MPI_FILE_SET_INFO`
4. Free the info object with `MPI_INFO_FREE`

   → can be freed as soon as passed!
5. Issue the I/O operations

   → `MPI_FILE_WRITE_ALL` ...

Hints - Info

Creating and Freeing

```
int MPI_Info_create(
      MPI_Info *info          // OUT      info object created (handle)
)
```

- `MPI_INFO_CREATE` creates a new info object
- The info object may be different on each process
- Hints that are required to be the same must be the same

```
int MPI_Info_free(
      MPI_Info *info          // IN OUT   info object (handle)
)
```

- `MPI_INFO_FREE` frees info and sets it to `MPI_INFO_NULL`

Hints - Info

Setting and Removing

```
int MPI_Info_set(
      MPI_Info info,           // IN OUT   info object (handle)
      const char *key,         // IN       key (string)
      const char *value        // IN       value (string)
)
```

- `MPI_INFO_SET` adds `(key,value)` pair to `info`
- This will override existing values for that key

```
int MPI_Info_delete(
      MPI_Info info,           // IN OUT   info object (handle)
      const char *key          // IN       key (string)
)
```

- Deletes a `(key,value)` pair or raises `MPI_ERR_INFO_NOKEY`

Hints - Info

Fetching Information

```
int MPI_Info_get_nkeys(
      MPI_Info info,           // IN      info object (handle)
      int *nkeys               //    OUT  number of defined keys (integer)
)
```

- Retrieves the number of keys sets in the `info` object
- We can also get each of those keys, i.e. the nth key, using:

```
MPI_Info_get_nthkey(
      MPI_Info info,           // IN      info object (handle)
      int n,                   // IN      key number (integer)
      char *key                //    OUT  key (string)
)
```

Hints - Info

Fetching Information

```
int MPI_Info_get(
      MPI_Info info,          // IN      info object (handle)
      const char *key,        // IN      key (string)
      int length,             // IN      length of value arg (integer)
      char *value,            // OUT     value (string)
      int *flag               // OUT     true if key defined, false if not (boolean)
)
```

- Retrieves the value set in key in a previous call to `MPI_INFO_SET`
- `length` is the number of characters available in `value`

```
int MPI_Info_get_valuelen(
      MPI_Info info,          // IN      info object (handle)
      const char *key,        // IN      key (string)
      int *length,            // OUT     length of value arg (integer)
      int *flag               // OUT     true if key defined, false if not (boolean)
)
```

Hints

Setting Hints

```
int MPI_File_set_info(
      MPI_File fh,              // IN OUT   file handle (handle)
      MPI_Info info             // IN       info object (handle)
)
```

- `MPI_FILE_SET_INFO` define new values for the hints of `fh`
- It is a collective routine
- The info object may be different on each process
- Hints that are required to be the same must be the same

Hints

Reading Hints

```
int MPI_File_get_info(
      MPI_File fh,              // IN       file handle (handle)
      MPI_Info *info_used       // OUT      new info object (handle)
)
```

- `MPI_FILE_GET_INFO` return a new `info` object
- Contain hints associated by `fh`
- Lists the actually used hints
  - Remember that some of them may be ignored!
- Only returns active hints

SOLUTION FILE
`get-all-hints.c`

Hands-on!

Which Hints?

Create a very simple code to:

- Open a file (you should create new empty file)
- Read all the default hints
  - Get the `info` object associated with the `fh` you just opened
  - Get the total number of keys sets
  - Iterate and get each of the keys
  - Get the value of the keys
  - Print these hints and its flag to the standard output
- Run your solution on your machine or on SDumont!
- CHALLENGE: modify/create some hints and get the values again!

SOLUTION FILE

get-all-hints.c

Hands-on!

Which Hints?

```
$ vi get-all-hints.c
        // Open the file
        // TO-DO
        // Get the info object associated with the file handle
        // TO-DO
        if (myrank == 0) {
                // Get the number of defined keys from the info object used on the file handle
                // TO-DO
                printf("there are %d hints set:\n", nkeys);
                int i= 0;
                for (i = 0; i < nkeys; i++) {
                        // Get the nth key to access its value
                        // TO-DO
                        // Get the key value
                        // TO-DO
                        printf("  %s: %s (%s)\n", key, value, flag ? "true" : "false");
                }
        }
        MPI_Barrier(MPI_COMM_WORLD);
        // Close the file
        // TO-DO
```

SOLUTION FILE

`get-all-hints.c`

Hands-on!

Which Hints?

- If you run on your machine, you need to use **ROMIO**!
- To find the version use:

```
$ ompi_info | grep romio

MCA io: romio314 (MCA v2.1.0, API v2.0.0, Component v3.0.0)
```

- Then you can run with ROMIO:

```
$ mpicc get-all-hints.c -o get-all-hints
$ mpirun -np 1 --oversubscribe --mca io romio314 ./get-all-hints
```

- Try to run using the ROMIO_HINTS variable. What is the difference?

```
$ ROMIO_HINTS=$PWD/hints.txt mpirun -np 1 --oversubscribe --mca io romio314 ./get-all-hints
```

```
there are 25 hints set:
  direct_read: false (true)
  direct_write: false (true)
  romio_lustre_co_ratio: 1 (true)
  romio_lustre_coll_threshold: 0 (true)
  romio_lustre_ds_in_coll: enable (true)
  cb_buffer_size: 16777216 (true)
  romio_cb_read: automatic (true)
  romio_cb_write: automatic (true)
  cb_nodes: 1 (true)
  romio_no_indep_rw: false (true)
  romio_cb_pfr: disable (true)
  romio_cb_fr_types: aar (true)
  romio_cb_fr_alignment: 1 (true)
  romio_cb_ds_threshold: 0 (true)
  romio_cb_alltoall: automatic (true)
  ind_rd_buffer_size: 4194304 (true)
  ind_wr_buffer_size: 524288 (true)
  romio_ds_read: automatic (true)
  romio_ds_write: automatic (true)
  cb_config_list: *:1 (true)
  romio_filesystem_type: LUSTRE: (true)
  romio_aggregator_list: 0  (true)
  striping_unit: 1048576 (true)
  striping_factor: 1 (true)
  romio_lustre_start_iodevice: 0 (true)
```

Output of the exercise on SDumont

SOLUTION FILE

NOT AVAILABLE

Hands-on!

Stripes!

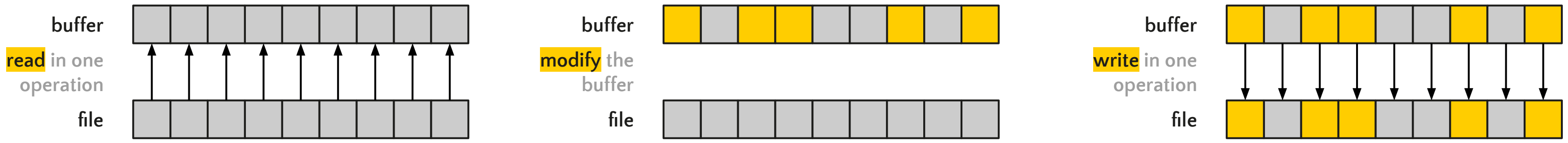
Take one of the codes from the previous exercises and:

- Make sure you are writing something
- Increase the total generated file size
- Define the `striping_factor` and/or `striping_unit` hints
- Measure the performance of your code on SDumont!
- You can use `lfs getstripe filename` to make sure of the striping data you used
- Specially usefull with the Parallel I/O version of the Julia Set

Optimization

Data Sieving

- I/O performance suffers when making many small I/O requests
- Access on small, non-contiguous regions of data can be optimized:
    - Group requests
    - Use temporary buffers
- This optimisation is local to each process (non-collective operation)



Hints

Data Sieving

`ind_rd_buffer_size` → size (in bytes) of the intermediate buffer used during read

`Default is 4194304 (4 Mbytes)`

`ind_wr_buffer_size` → size (in bytes) of the intermediate buffer used during write

`Default is 524288 (512 Kbytes)`

`romio_ds_read` → determines when ROMIO will choose to perform data sieving

`enable, disable, or` **`automatic`** `(ROMIO uses heuristics)`

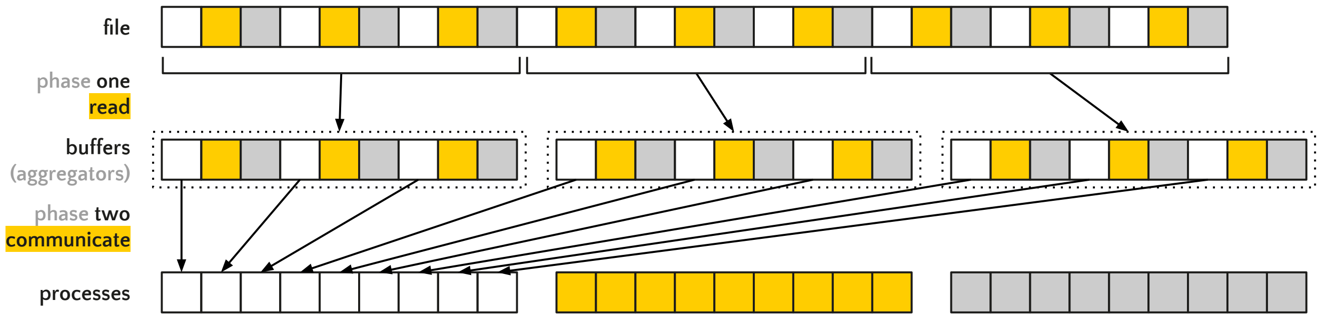
`romio_ds_write` → determines when ROMIO will choose to perform data sieving

`enable, disable, or` **`automatic`** `(ROMIO uses heuristics)`

Optimization

Collective Buffering

- Collective buffering, a.k.a. two-phase collective I/O
- Re-organises data across processes to match data layout in file
- Involves communication between processes
- Only the aggregators perform the I/O operation



Hints

Collective Buffering

`cb_buffer_size` → size (in bytes) of the buffer used in two-phase collective I/O

`Default is 4194304 (4 Mbytes)`
`Multiple operations could be used if size is greater than this value`

`cb_nodes` → maximum number of aggregators to be used

`Default is the number of unique hosts in the communicator used when opening the file`

`romio_cb_read` → controls when collective buffering is applied to collective read

`enable, disable, or` **`automatic`** `(ROMIO uses heuristics)`

`romio_cb_write` → controls when collective buffering is applied to collective write

`enable, disable, or` **`automatic`** `(ROMIO uses heuristics)`

Conclusion

Review
Final Thoughts

Conclusion

- MPI-IO is powerful to express complex data patterns
- Library can automatically optimize I/O requests
- But there is no "magic"
- There is also no "best solution" for all situations
- Modifying an existing application or writing a new application to use collective I/O optimization techniques is not necessarily easy, but the payoff can be substantial
- Prefer using MPI collective I/O with collective buffering

Thank You!

Any *questions*?

Get in touch!

andrerc@lncc.br

arcarneiro@inf.ufrgs.br

References

Cray Inc. **Getting Started on MPI I/O**, report, 2009; Illinois. (docs.cray.com/books/S-2490-40/S-2490-40.pdf: accessed February 17, 2018).

Message Passing Interface Forum. **MPI: A Message-Passing Interface Standard Version 3.0**, report, September 21, 2012; (mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf: accessed February 17, 2018), University of Tennessee, Knoxville, Tennessee.

Robert Latham, Robert Ross. (2013) **Parallel I/O Basics**. In: Earth System Modelling - Volume 4. SpringerBriefs in Earth System Sciences. Springer, Berlin, Heidelberg,

Thakur, R.; Lusk, E. & Gropp, W. **Users guide for ROMIO: A high-performance, portable MPI-IO implementation**, report, October 1, 1997; Illinois. (digital.library.unt.edu/ark:/67531/metadc695943/: accessed February 17, 2018), University of North Texas Libraries, Digital Library, digital.library.unt.edu; crediting UNT Libraries Government Documents Department.

William Gropp; Torsten Hoefler; Rajeev Thakur; Ewing Lusk, **Parallel I/O**, in Using Advanced MPI: Modern Features of the Message-Passing Interface, 1, MIT Press, 2014, pp.392.

SOLUTIONS
http://www.lncc.br/~andrerc/mpi-io-lncc-2021-solutions.tar.gz
http://www.lncc.br/~andrerc/mpi-io-lncc-2021-solutions.zip