MPI-IO

Escola Supercomputador SDumont 2018

Msc. Jean Luca Bez

Hi!

I am Jean Luca Bez

Msc. Computer Science - UFRGS, Porto Alegre - RS

Computer Scientist - URI, Erechim - RS

jean.bez@inf.ufrgs.br

UFRGS - Universidade Federal do Rio Grande do Sul

INF - Instituto de Informática

GPPD - Grupo de Processamento Paralelo e Distribuído

Agenda

Notions
- I/O for HPC
- Data Striping
- Access Pattern
- MPI-IO

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

Individual Operations
- Read / Write
- Explicit Offsets
- Individual File Pointers
- Shared File Pointers

Collective Operations
- Read / Write
- Explicit Offsets
- Individual File Pointers
- Shared File Pointers

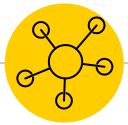
Asynchronous Operations
- Read / Write
- Explicit Offsets
- Individual File Pointers
- Shared File Pointers

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

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

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****.*

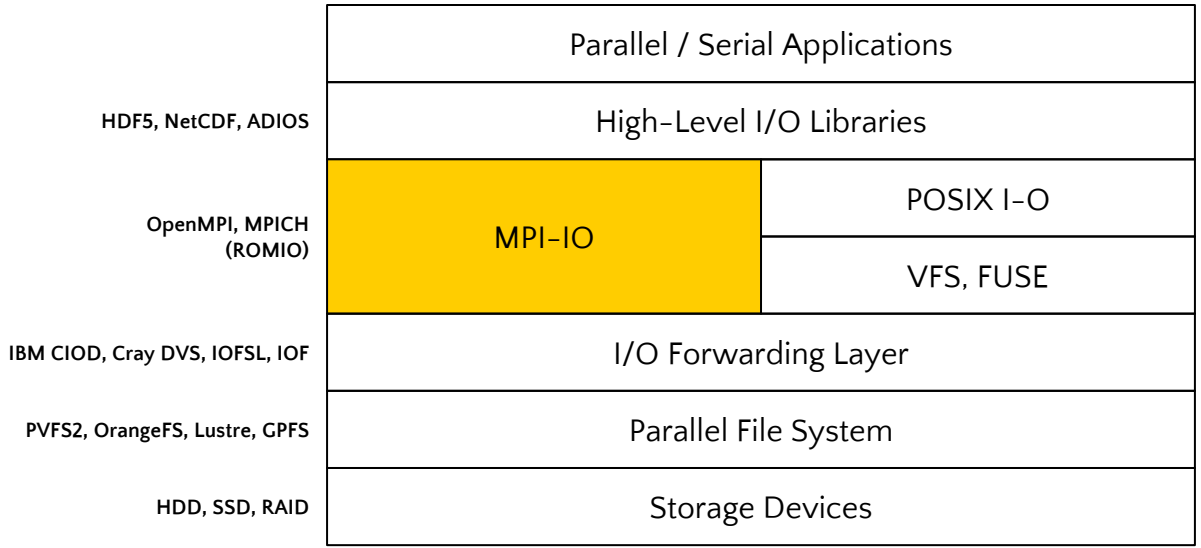
— *Getting Started on MPI I/O, Cray, 2015* —



Notions

I/O for HPC
Data Striping
Access Pattern
MPI-IO

HPC I/O Stack



Inspired by Ohta et. a. (2010)

HPC I/O Stack

POSIX I/O

- A POSIX I/O file is simply a sequence of bytes
- You use the POSIX I/O interface to transfer:
  - Contiguous regions of bytes between the file and memory
  - Noncontiguous regions of bytes from memory to a file
- 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”

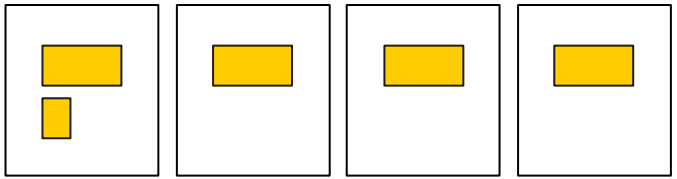
HPC I/O Stack

Physical and Logical File View

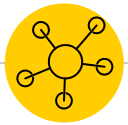
- Logically, a file is a linear sequence of bytes
- This is how the application access the file (displacement, N bytes)

file (logical view)

- Physically, a file consists of data distributed across data servers
- This operations is called file striping



file (physical view)



Notions

Access Pattern

Access Pattern

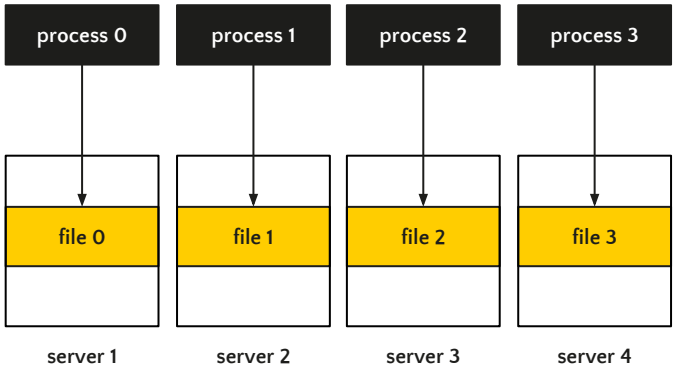
I/O Strategies

What are the most common ones?

- One **File per Process**
  - All Write
- Single **Shared File**
  - One Writes
  - All Write
  - Subset Writes

Which one is the best solution?

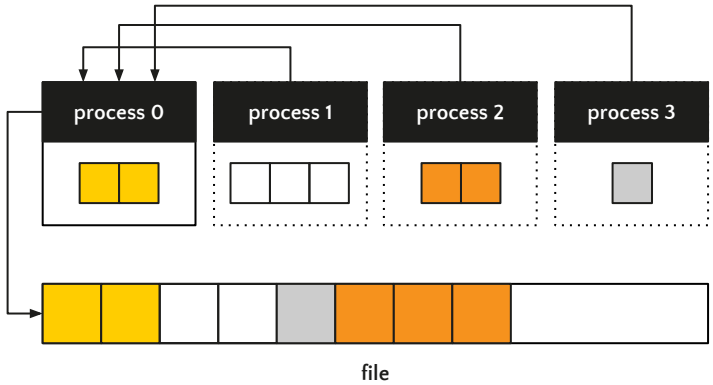
File per Process
All Write



Each process write to its **own file**

- Independent writes could bring improvements because multiple PFS nodes can support parallel I/O for multiple files
- The **metadata** server becomes a **bottleneck**
  - Centralized
  - Impacts the application
  - Impacts the system

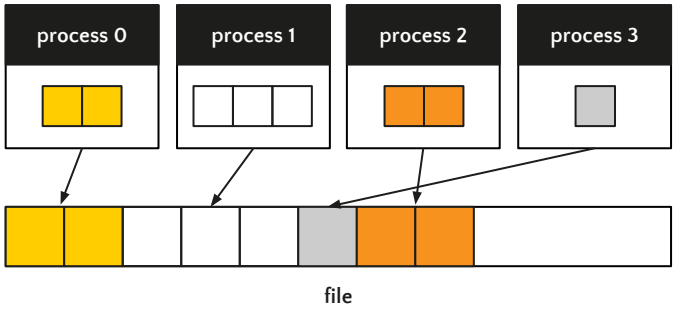
Shared File One Writes



All processes **send their data** to only **one process** which then writes the data to a single shared file

- Uses data **aggregation**
- `process 0` is able to combine the data into **larger** and **contiguous requests**
- There is no bottleneck at the metadata server
- Writes are sequential and not parallel

Shared File
All Write

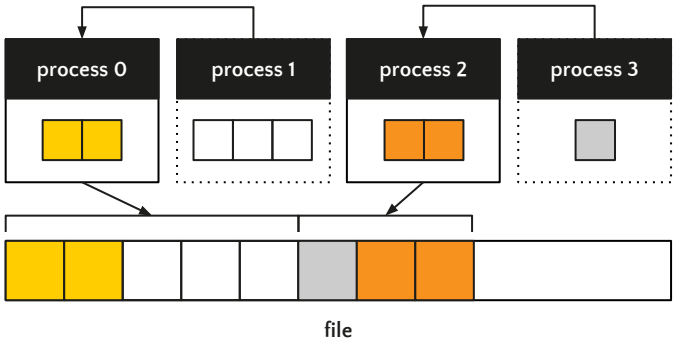


All processes **write their data** in a single shared file

- Parallel I/O
- Requires extra work by the application to coordinate and maintain the separate file offsets to avoid data override
- Optimizations with collective operations

(next slides)

Shared File
Subset Writes



Only a **subset of processes** involved in the write send their data **directly** to an unique shared file

- Also uses data **aggregation**
- There is no bottleneck in the metadata server
- Hard to find the best **balance**
  - Number of process that do I/O
  - Can change with the problem size
  - Can be different on other systems

Access Pattern

Cray's Guidelines

"The **one file per process, all write** strategy can work well if you have a small number of processes and the amount of data per process is large."

"The **single shared file, one writes** strategy can work well if the application writes a small amount of data."

"The **single shared file, all write** strategy can work well if the amount of data per process is large and the ratio of processes to OSTs is small."

"The **single shared file, subset writes** strategy can work well if there are a large number of processes and there is a large amount of data, and if the appropriate number for the subset is selected and the data is sent to the subset in the appropriate way."

small *vs.* large

How much is small and how much is large for I/O?

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 --np PROCESSES ./code
```

MPI-IO

Version 2.1

- You may need to use Version 2.1 in SDumont Supercomputer
- Remember how to load the version:

```
module avail
module load openmpi/3.0_gnu        # in testing (will not work with srun)
module load openmpi/2.1_gnu
```

- Compile normally:

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

SLUM

```
#!/bin/bash
#SBATCH --nodes=4             # Número de Nós
#SBATCH --ntasks-per-node=2   # Número de tarefas por Nó
#SBATCH --ntasks=8            # Número total de tarefas MPI
#SBATCH -p treinamento        # Fila (partition) a ser utilizada
#SBATCH -J mpi-io             # Nome job
#SBATCH --exclusive           # Utilização exclusiva dos nós durante a execução do job

# Exibe os nós alocados para o Job
echo $SLURM_JOB_NODELIST
nodeset -e $SLURM_JOB_NODELIST
cd $SLURM_SUBMIT_DIR

# Configura os compiladores
module load openmpi/2.1_gnu

# Configura o executavel
EXEC=/scratch/treinamento/ALUNO/MPI-IO/get-hints

# Exibe informações sobre o executável
/usr/bin/ldd $EXEC

srun --resv-ports -n $SLURM_NTASKS $EXEC
```

Concepts of MPI-IO

Terminology
Error Classes

Concepts of MPI-IO

file e *displacement*

file

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

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 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*)

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

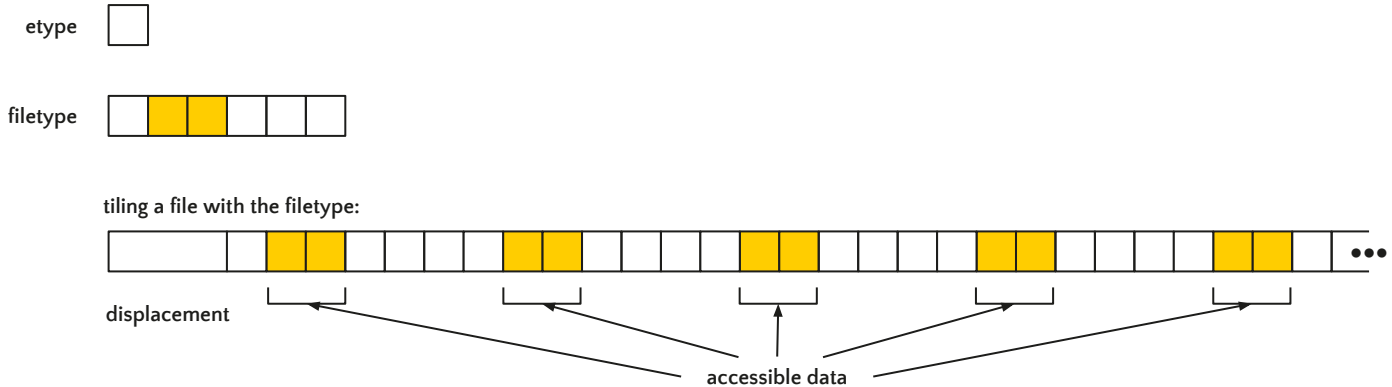
Each **process** has its **own view**, defined by:

- a displacement
- an *etype*
- a filetype

The pattern described by a filetype is repeated, beginning at the displacement, to define the view

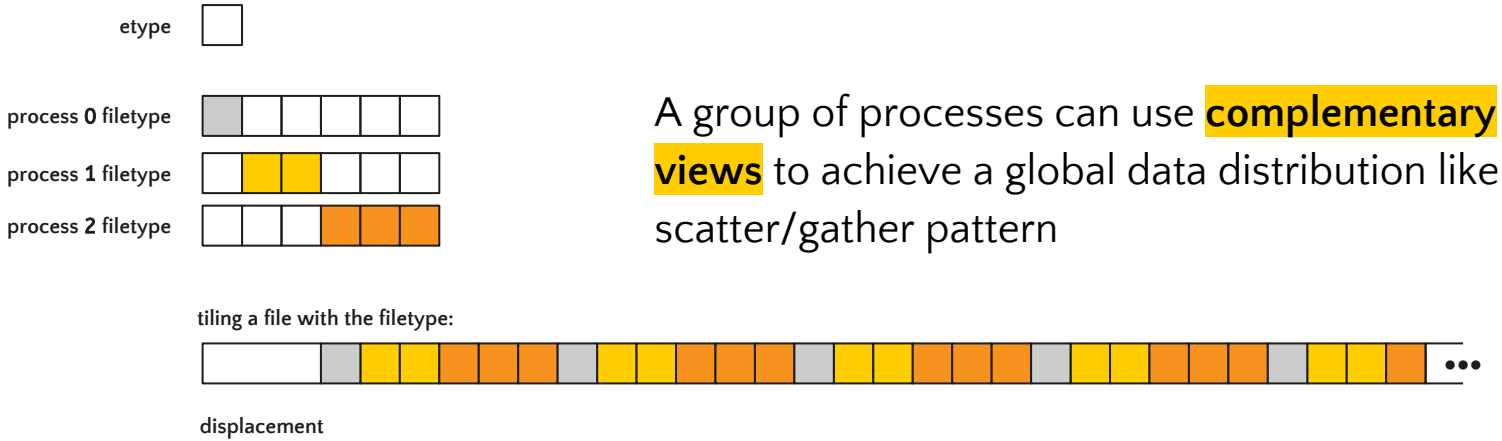
Concepts of MPI-IO

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 view

Expressed as a **count of etypes**

Holes in the filetype are **skipped** when calculating

file size

Size of MPI file is measures 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

Concepts of MPI-IO

I/O Error Classes

MPI Error	Description
`MPI_ERR_FILE`	Invalid file handle
`MPI_ERR_NOT_SAME`	Collective argument not identical on all processes
`MPI_ERR_AMODE`	Error related to the amode passed to MPI_FILE_OPEN
`MPI_ERR_NO_SUCH_FILE`	File does not exist
`MPI_ERR_FILE_IN_USE`	File operation could not be completed The file is currently open by some process
`MPI_ERR_ACCESS`	Permission denied
`MPI_ERR_READ_ONLY`	Read-only file or file system
`MPI_ERR_IO`	Other I/O error

File Manipulation

Opening Files
Access Mode (amode)
Closing Files
Removing Files

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

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*

```
(MPI_MODE_CREATE|MPI_MODE_EXCL|MPI_MODE_CREATE)
```

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`

File Manipulation

Removing Files

```
int MPI_File_delete(
      const char *filename,   // IN       name of file to delete (string)
      MPI_Info info           // IN       info object (handle)
)
```

- `MPI_FILE_DELETE` deletes the file
- If the file does not exists, raises `MPI_ERR_NO_SUCH_FILE`
- We can pass info regarding file system specifics
- When no info needs to be specified we should use `MPI_INFO_NULL`
- If the file is being used, the behavior depends of the implementation
  - If the file is not removed
  - Raises `MPI_ERR_FILE_IN_USE` or `MPI_ERR_ACCESS`

File Manipulation

Resizing a File
File Information

File Manipulation

Resizing a File

```
int MPI_File_set_size(
      MPI_File fh,            // IN OUT   file handle (handle)
      MPI_Offset size         // IN       size to truncate or expand file (integer)
)
```

- `MPI_FILE_SET_SIZE` is used to resize the file
- `size` is measured in bytes from the beginning of the file
- This operation is collective
- All process must provide the same value for `size`
- File can be truncated or expanded

File Manipulation

File Information

```
int MPI_File_get_size(
      MPI_File fh,            // IN       file handle (handle)
      MPI_Offset *size        //    OUT   size of the file in bytes (integer)
)
```

```
int MPI_File_get_group(
      MPI_File fh,            // IN       file handle (handle)
      MPI_Offset *size        //    OUT   group which opened the file (handle)
)
```

```
int MPI_File_get_amode(
      MPI_File fh,            // IN       file handle (handle)
      int *amode              //    OUT   file access mode used to open the file (integer)
)
```

exercise or experiment

Hands-on!

File Creation

Write a program that:

- Opens a file named `code.out` for reading and writing
  - If the file does not exist, it should be created
- Queries the size of the file
- Closes the file

Hands-on!

File Creation

SOLUTION FILE

`create-get-size-of-file.c`

Write a program that:

- Opens a file named `code.data` for reading and writing
  - If the file does not exist, it should be created
- Queries the size of the file
- Closes the file

Hands-on!

Preallocate Space

SOLUTION FILE

`create-set-and-get-size-of-file.c`

Upgrade the program from the previous exercise to:

- Open a file named `code.data` for reading and writing
  - If the file does not exist, it should be created
- Preallocate `1MB` of space for the file
- Query the size of the file

Check the file size with `ls` to make sure your solution is correct!

Data Access

Positioning
Coordination
Synchronism

Data Access

Overview

- There are 3 aspects to data access:

positioning	synchronism	coordination
explicit offset	blocking	noncollective
implicit file pointer (individual)	nonblocking	collective
implicit file pointer (shared)	split collective	

- 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
- Individual file pointer routines contain no positional qualifiers
- Shared pointer routines contain `_SHARED` or `_ORDERED` in the name
- I/O operations leave the MPI file pointer pointing to next item
- 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`)
- The nonblocking calls are named `MPI_FILE_IXXX` with an `I` (immediate)
- We should not access the buffer until the operation is complete

Data Access

Coordination

- Every noncollective routine has a collective counterpart:
  - `MPI_FILE_XXX` is `MPI_FILE_XXX_ALL`
  - a pair `MPI_FILE_XXX_BEGIN` and `MPI_FILE_XXX_END`
  - `MPI_FILE_XXX_SHARED` is `MPI_FILE_XXX_ORDERED`
- 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)
)
```

```
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)
)
```

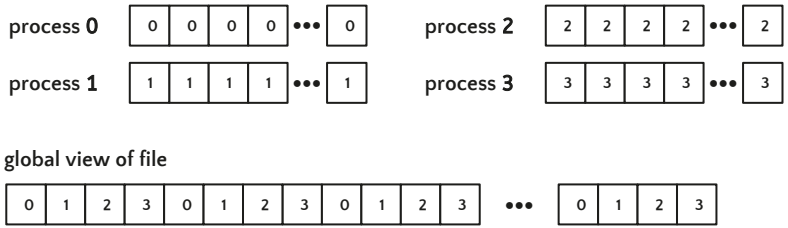
SOLUTION FILE

`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
01230123012301230123012301230123012301230123
```

Hands-on!

READ - Explicit Offsets

SOLUTION FILE

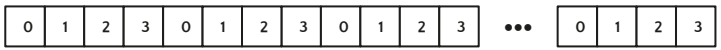
`read-i-offsets-character.c`

Modify your program so that each process will 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
...
```

Hands-on!

WRITE - Explicit Offsets

SOLUTION FILE

write-i-offsets-integer.c

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

Since MPI-IO will write in binary format, unless we manually format the data (as we did in the previous exercise), we need another way to visualize:

```
$ hexdump -v -e '"%d"' my-rank.txt
01230123012301230123012301230123012301230123
```

```
$ hexdump -v -e 'NP "%2d "' -e '"\n"' my-rank.txt     // NP = number of processes
```

Hands-on!

READ - Explicit Offsets

SOLUTION FILE

`read-i-offsets-integer.c`

Modify your program so that each process will read the printed ranks, as an integer, 10 times using explicit offsets (and default view).

Remember to open the file as read only!

Each process should print the values to `stdout`.

```
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
...
```

Revisiting...

File Manipulation

File View
Data Types
Data Representation

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` changes the process's view of the data
- This is a collective operation
- Values for `disp`, `filetype` and `info` may vary
- `disp` is the absolute offset in bytes from where the view begins

File Manipulation

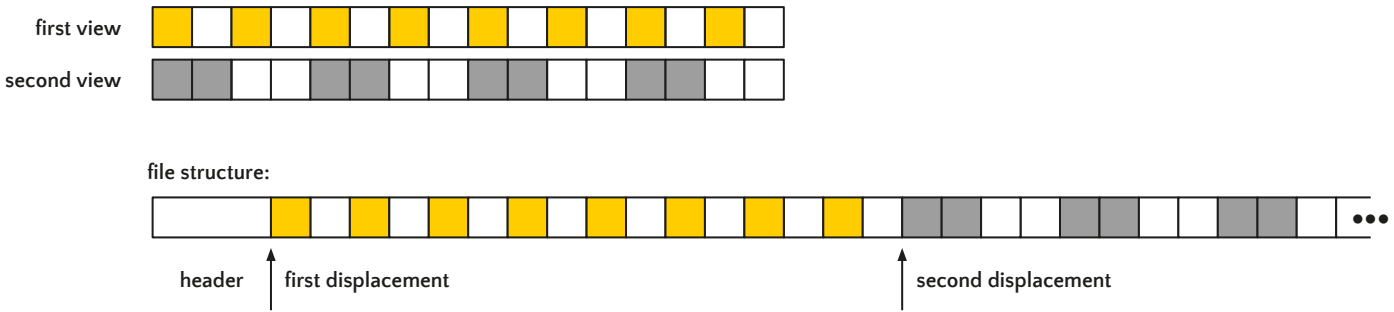
Default File View

- 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

Multiple File Views

- Define different patterns in the file
- Separate views, each using a different displacement and filetype
- Can be used to access each segment of the file



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

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

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`

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

Data Representation

- MPI supports multiple data representations:
  - `“native”`
  - `“internal”`
  - `“external32”`
- `“native”` and `“internal”` are implementation dependent
- `“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

Data Access

Noncollective I/O

Individual File Pointers
Shared File Pointers

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_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)
)
```

```
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)
)
```

Hands-on!

WRITE - Individual Pointers

SOLUTION FILE

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

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

Remember to view the file you should use `hexdump` or similar!

```
$ mpirun -np 4 random-rank-fileview-buffer
$ hexdump -v -e '10 "%f "' -e '"\n"' random-rank-fileview-buffer.txt
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

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

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)
)
```

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_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)
)
```

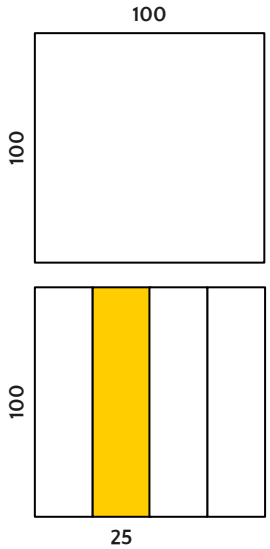
```
int MPI_File_read_at_all(
      MPI_File fh,             // IN OUT   file handle (handle)
      MPI_Offset offset,       // IN       file offset (integer)
      void *buf,               //    OUT   initial address of buffer (choice)
      int count,               // IN       number of elements in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

SOLUTION FILE

`write-c-ipf-view-subarray-datatype-character.c`

Hands-on!

WRITE - Subarray Datatype



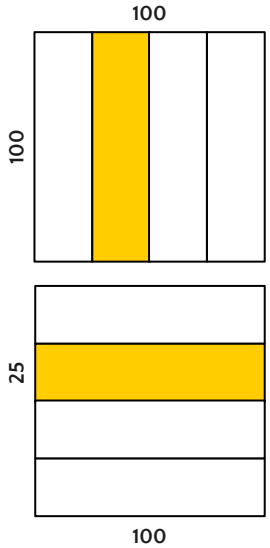
- Consider a global matrix of `100 X 100` and 4 processes
- Divide the domain into 4 parts of `100 X 25` (local matrix)
- Make each process fill its local matrix with `('a' + rank)`
- Create a new filetype to write the subarray
- Define a view based on the subarray filetype you created
- Each process should write its subarray in a collective operation
- Make sure you are using `MPI_ORDER_C` to store in row-major order
- Use `cat` to view your file and make sure it is correct!

SOLUTION FILE

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

Hands-on!

WRITE - Subarray Datatype



- Modify the previous solution to write doubles
- 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)
- Each cell should be filled with `(rank + (count / 100))`
- Where `count` is a counter of cells when iterating
- First try writing using `MPI_ORDER_C` to store in row-major order
- Try writing using `MPI_ORDER_FORTRAN` to store in column-major order
- Use `hexdump` to view your file and make sure it is correct!

```
hexdump -v -e '16 "%f "' -e '"\n"' write-individual-file-pointer-view-subarray-datatype.txt

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
- 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)
)
```

Hands-on!

Individual *vs.* Shared Pointers

SOLUTION FILE

`write-i-ifp-rank.c` / `write-i-sfp-rank.c`

Using collective I/O operations:

Create one program that uses individual file pointers to:

- Using a `for` loop write 10 times the letter equivalent to `'a' + rank`.
- Run it with less than 27 characters (i.e. processes)

Create a second code that uses shared file pointers to:

- Using a `for` loop write 10 times the letter equivalent to `'a' + rank`.
- Run it with less than 27 characters (i.e. processes)

Compare the behavior (output) of both codes. Are they the same?

Hands-on!

Noncollective *vs.* Collective

SOLUTION FILE

USE YOUR CODE

Using your solution for the previous exercises, increase the total generated file size and run two sets of experiments on SDumont supercomputer:

- Take at least 5 measurement using noncollective operations
- Take at least 5 measurements using collective operations

Compare both alternatives! Which one is faster?

```
start = MPI_Wtime();
MPI_File_write(...);
finish = MPI_Wtime();
io_time = finish - start;
```

Data Access

Asynchronous 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

Asynchronous I/O

- MPI can start a data access and associate a request handle, `request`
- Nonblocking operations are completed via:

```
int MPI_Wait(
      MPI_Request *request,   // IN       request object (Request)
      MPI_Status *status      //    OUT   status object (Status)
)
```

```
int MPI_Test(
      MPI_Request *request,   // IN       request object (Request)
      int *flag,              //    OUT   true if operation completed (logical)
      MPI_Status *status      //    OUT   status object (Status)
)
```

- Operation must be completed before it is safe to reuse data buffers

Data Access - Asynchronous and Noncollective

Explicit Offsets

```
int MPI_File_iwrite_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_Request *request     //    OUT   request object (handle)
)
```

```
int MPI_File_iread_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_Request *request     //    OUT   request object (Status)
)
```

Data Access - Asynchronous and Noncollective

Individual File Pointers

```
int MPI_File_iwrite_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_Request *request     //    OUT   request object (handle)
)
```

```
int MPI_File_iread_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_Request *request     //    OUT   request object (handle)
)
```

Data Access - Asynchronous and Noncollective

Shared File Pointers

```
int MPI_File_write_ishared(
      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_Request *request     //    OUT   request object (handle)
)
```

```
int MPI_File_read_ishared(
      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_Request *request     //    OUT   request object (handle)
)
```

Data Access - Asynchronous and Collective

Explicit Offsets

```
int MPI_File_iwrite_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_Request *request     //    OUT   request object (handle)
)
```

```
int MPI_File_iread_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_Request *request     //    OUT   request object (Status)
)
```

Data Access - Asynchronous and Collective

Individual File Pointers

```
int MPI_File_iwrite_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_Request *request     //    OUT   request object (handle)
)
```

```
int MPI_File_iread_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_Request *request     //    OUT   request object (handle)
)
```

Hands-on!

Overlap Computation and I/O

SOLUTION FILE

`write-async-ifp-overlap.c`

Create a program to:

- Populate a buffer of size 10000 the character `'a' + rank`
- Write said buffer to a shared file `overlap-letter.txt`
- Use asynchronous I/O with individual file pointers
- Use `MPI_TYPE_VECTOR` to create a view of the size of the buffer
- Include a `sleep(seconds)` call to simulate computation time
- Make sure you wait all process finish to write before closing the file
- Make sure you wrote the same amount of letters

```
cat overlap-letter.txt | grep -o a | wc -l
cat overlap-letter.txt | grep -o b | wc -l
```

Hands-on!

Overlap Computation and I/O

SOLUTION FILE

`write-async-ifp-overlap-continue.c`

Modify your code of the previous exercise to:

- Instead of waiting the I/O to finish to check if it finished
- If it did, your program should close the file and stop
- If it did not, "continue computation", and then wait for the I/O
- Change `sleep(seconds)` to simulate shorter and longer computations

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*

Hints

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

MPI-IO

Hints

- 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` e `MPI_FILE_SET_INFO`

- Some hints cannot be overridden in operations such as:

  `MPI_FILE_SET_VIEW` e `MPI_FILE_SET_INFO`

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 n^{th} 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

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. Do the I/O operations

   → `MPI_FILE_WRITE_ALL`...

Hands-on!

Which Hints?

SOLUTION FILE

`get-all-hints.c`

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 SDumont!

```
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

Hints

MPI-IO Reserved Hints

`access_style` → how the file will be accessed (until close or change)

`read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, and random`

`chunked*` → file is a multidimensional array accessed by subarrays

`comma separated list of array dimensions, starting by the most significant one`

`chunked_item*` → the size of each array entry in bytes

`chunked_size*` → dimensions of the subarray

`comma separated list of array dimensions, starting by the most significant one`

* The same hint should be passed by all processes in the collective call

Hints

MPI-IO Reserved Hints

Hints relevant for `MPI_FILE_CREATE`:

`file_perm`* → permissions to use for creation

`io_node_list`* → list of I/O devices that should store the file (relevant on create)

`num_io_nodes`* → number of I/O devices in the system

`striping_factor`* → number of I/O devices the file should be striped across

`striping_unit`* → suggested striping unit (amount of consecutive data)

* The same hint should be passed by all processes in the collective call

Hands-on!
Stripes!

SOLUTION FILE

`write-c-ipf-view-subarray-datatype-character-stripe.c`

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 filename` to make sure of the striping data you used

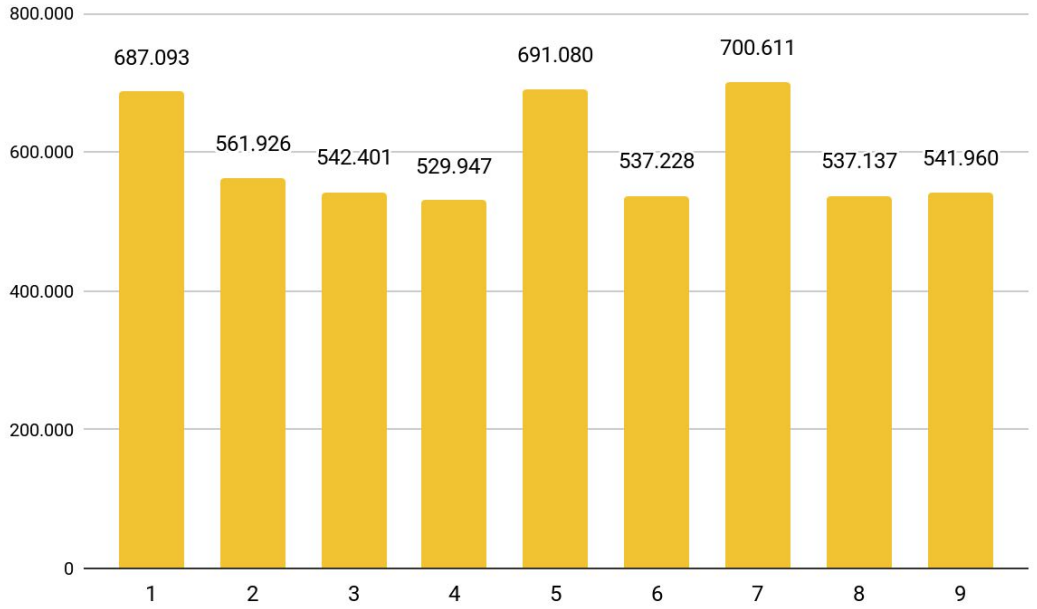
```
$ lfs getstripe striping-factor-1.txt
striping-factor-1.txt
lmm_stripe_count:   1
lmm_stripe_size:    1048576
lmm_layout_gen:     0
lmm_stripe_offset:  8
        obdidx             objid             objid             group
             8         627435435       0x2565e7ab                 0
```

```
$ lfs getstripe striping-factor-2.txt
striping-factor-2.txt
lmm_stripe_count:   2
lmm_stripe_size:    1048576
lmm_layout_gen:     0
lmm_stripe_offset:  7
        obdidx             objid             objid             group
             7         627456996       0x25663be4                 0
             5         626079511       0x25513717                 0
```

```
$ lfs getstripe striping-factor-4.txt
striping-factor-4.txt
lmm_stripe_count:   4
lmm_stripe_size:    1048576
lmm_layout_gen:     0
lmm_stripe_offset:  8
        obdidx             objid             objid             group
             8         627435439       0x2565e7af                 0
             2         626779178       0x255be42a                 0
             9         627429489       0x2565d071                 0
             1         627009698       0x255f68a2                 0
```

```
$ lfs getstripe striping-factor-8.txt
striping-factor-8.txt
lmm_stripe_count:   8
lmm_stripe_size:    1048576
lmm_layout_gen:     0
lmm_stripe_offset:  6
        obdidx             objid             objid             group
             6         627172680       0x2561e548                 0
             4         627663700       0x25696354                 0
             0         627437154       0x2565ee62                 0
             8         627435440       0x2565e7b0                 0
             2         626779179       0x255be42b                 0
             9         627429490       0x2565d072                 0
             1         627009699       0x255f68a3                 0
             7         627456999       0x25663be7                 0
```

Checking the output of the exercise on SDumont



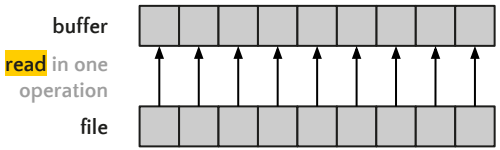
Output of the exercise on SDumont

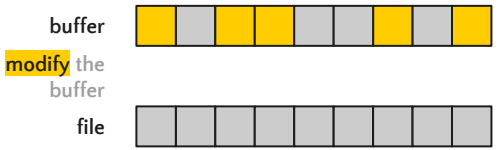
Time is presented as the average of 10 repetitions

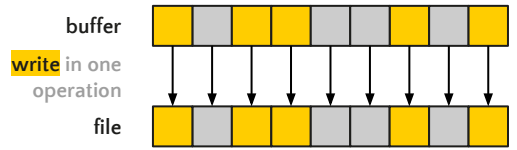
Optimization

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)

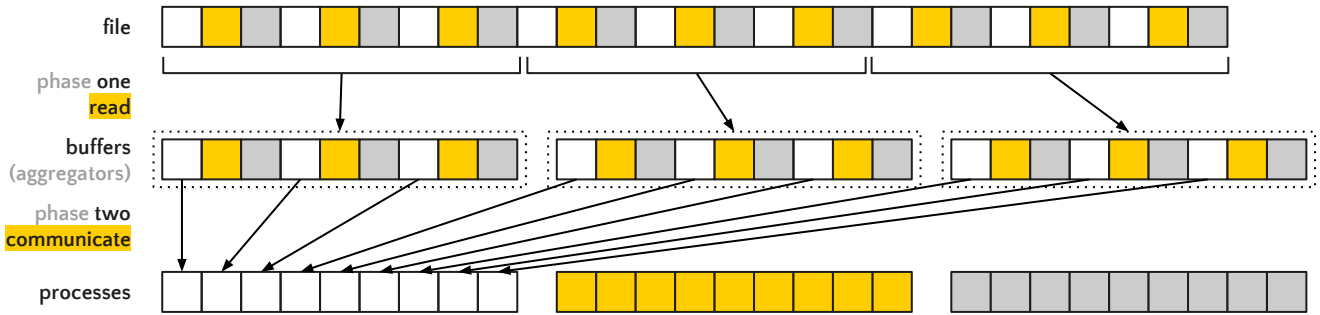
`romio_ds_write` → determines when ROMIO will choose to perform data sieving

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!

jean.bez@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
https://goo.gl/nhkZdy
2018 - Jean Luca Bez