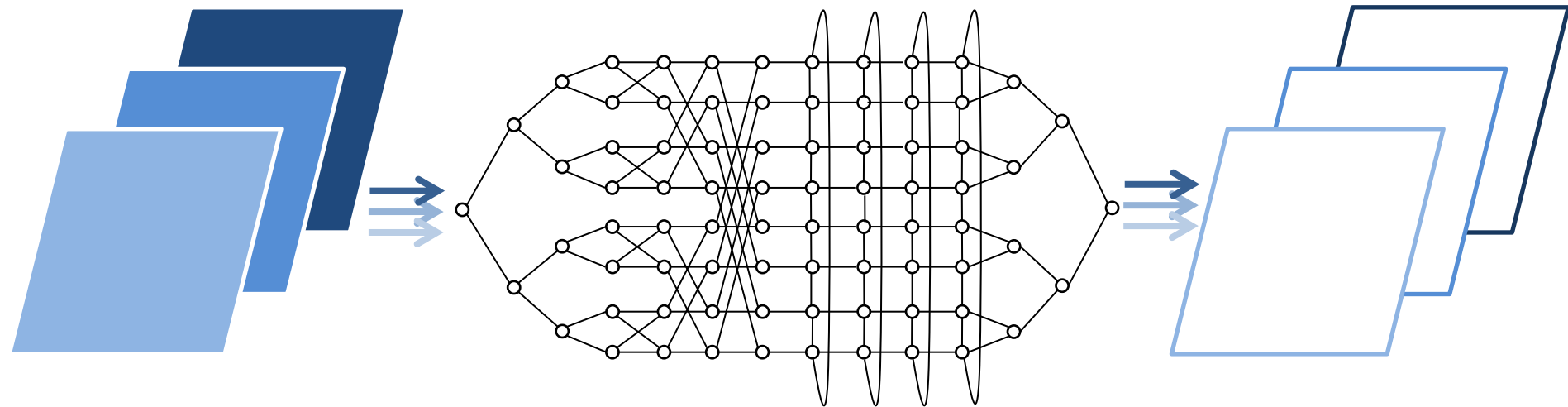


Programmation Parallèle

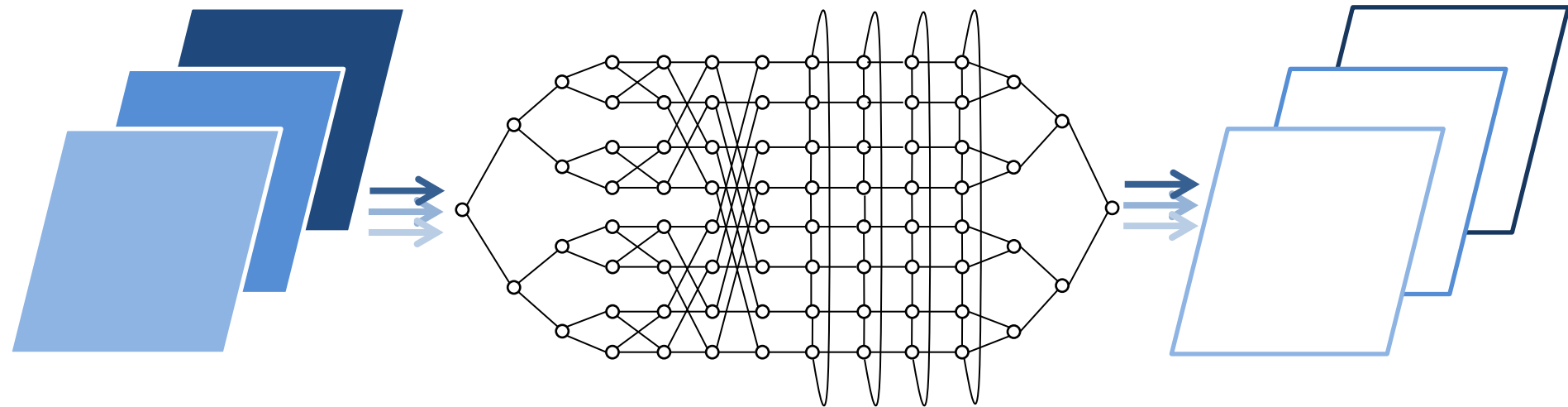
MPI: Message Passing Interface



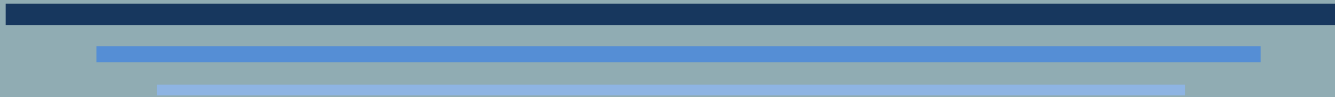
MPI I/O

Parallel file reading/writing

ENSIIE-HPC/BigData-PP-IPAR-Lecture 5



INTRODUCTION ON I/O



What are I/O ?



- I/O stands for Input/Output
- I/O usually regroups every operations applied on files by a program
 - Reading (Input)
 - Writing (Output)
 - File Management
 - Resizing
 - Splitting/Merging
 - Organizing
 - Searching

Why do we need I/O ?



- Simulations produce lot of data
 - Not meant to be directly read by a person
 - Necessary to
 - Directly translate them into human readable objects (pictures,...)
 - Store the data to be read by another “translator” program (visualizer,...)
- These data need to be stored
 - To be used by other programs
 - To keep history of simulations
 - You don’t want to replay the simulation any time you need some result
- Checkpoint/restart
 - More complex supercomputer means more failures
 - Mean time before failure (mtbf) less than some simulations time
 - Necessary to save temporary states to restart the simulation

Sequential I/O

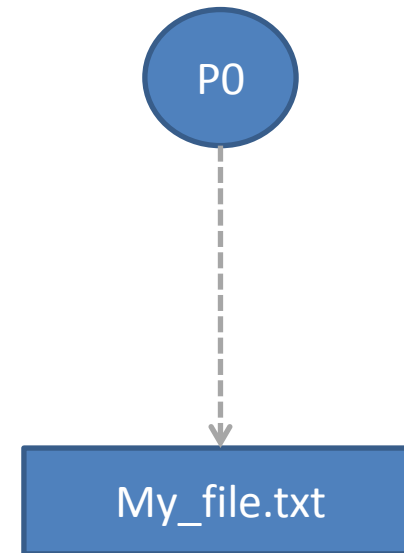


- Original paradigm
 - Example: Posix I/O
- How to do I/O?



Sequential I/O

- Original paradigm
 - Example: Posix I/O
- How to do I/O?
 - Process P1 open a file handler
 - Associated with a name



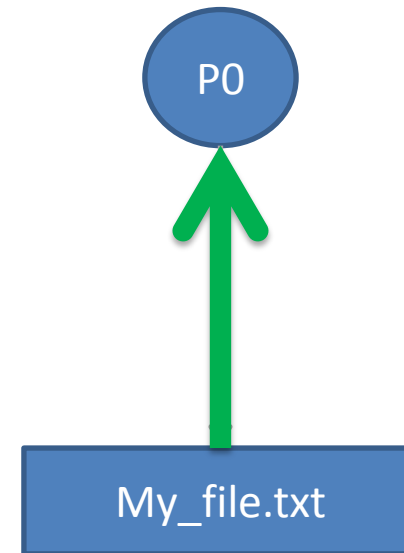
Sequential I/O

- Original paradigm

- Example: Posix I/O

- How to do I/O?

- Process P1 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P1 either read data....



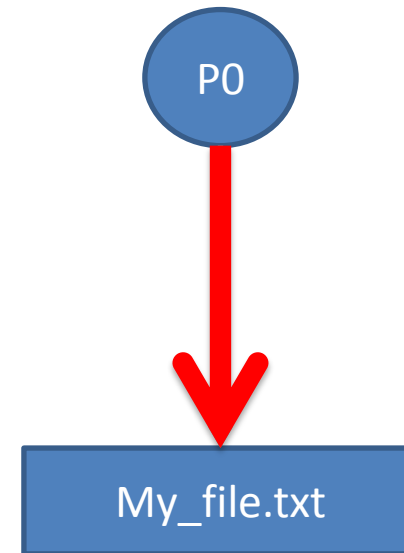
Sequential I/O

- Original paradigm

- Example: Posix I/O

- How to do I/O?

- Process P1 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P1 either read data....
 - ... or write data



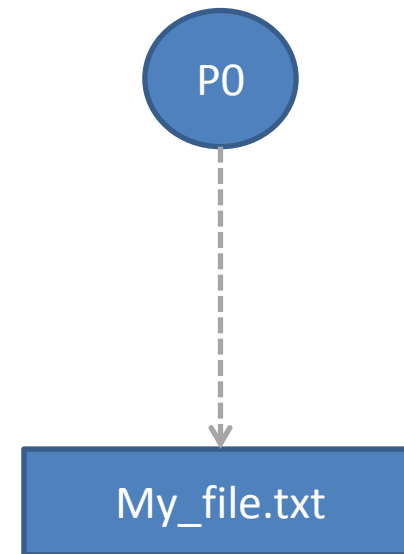
Sequential I/O

- Original paradigm

- Example: Posix I/O

- How to do I/O?

- Process P1 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P1 either read data....
 - ... or write data
 - Once it is done



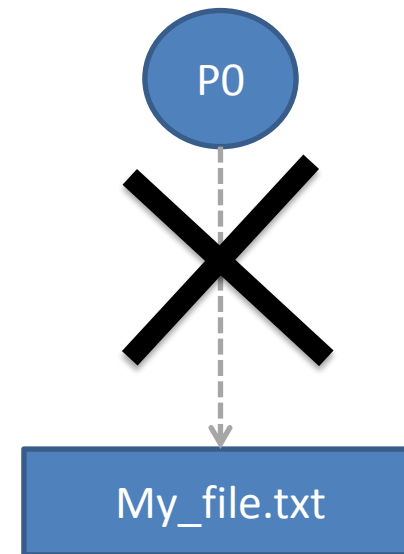
Sequential I/O

- Original paradigm

- Example: Posix I/O

- How to do I/O?

- Process P1 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P1 either read data....
 - ... or write data
 - Once it is done, P1 close file handler



Sequential I/O

- Original paradigm

- Example: Posix I/O

- How to do I/O?

- Process P1 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P1 either read data....
 - ... or write data
 - Once it is done, P1 close file handler



My_file.txt

A blue rounded rectangle containing the text "My_file.txt", representing a file.

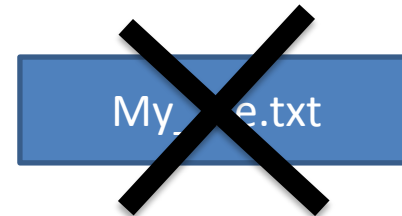
Sequential I/O

- Original paradigm

- Example: Posix I/O

- How to do I/O?

- Process P1 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P1 either read data....
 - ... or write data
 - Once it is done, P1 close file handler
 - May also delete the file if asked



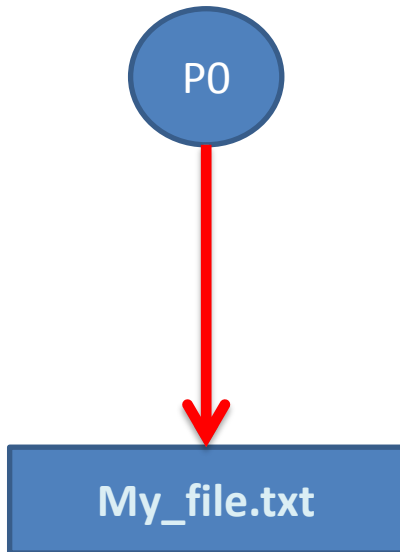
Parallel I/O



- Easy when there is only one process
- How to manage that with multiple processes which all want to read/write?
- Solution:
 - One-to-One
 - All-to-All
 - One-to-All
 - All-to-One

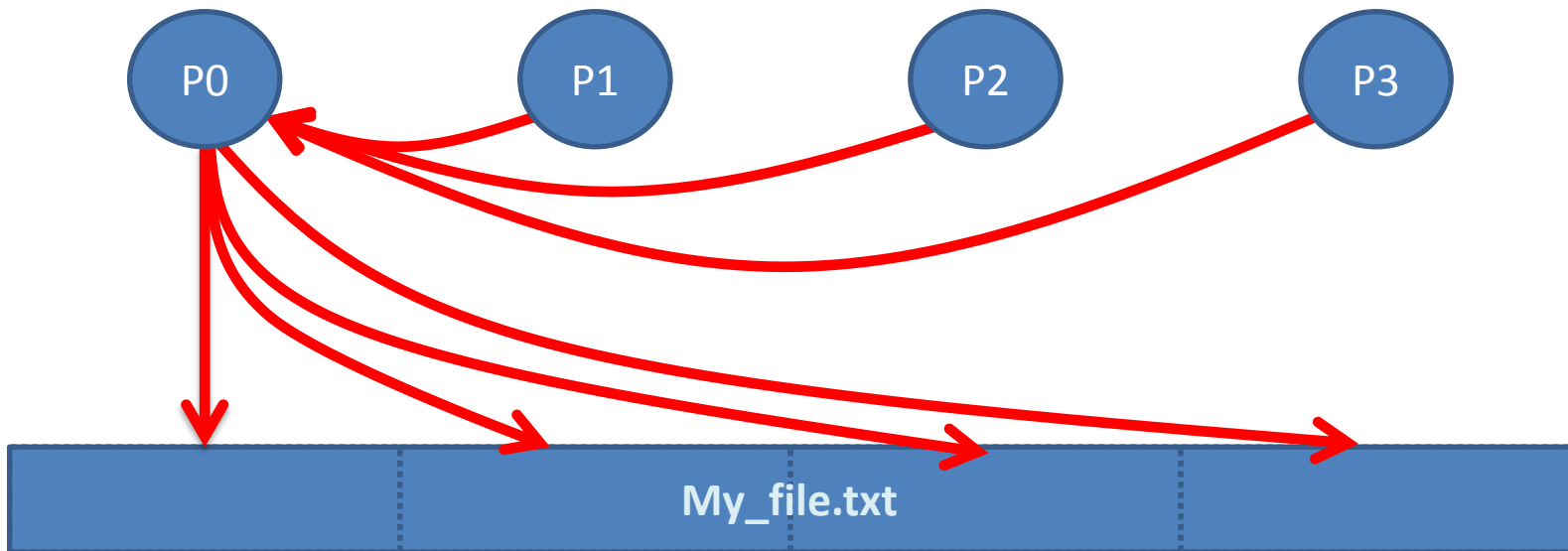
Parallel One-to-One I/O

- Based on legacy sequential I/O
 - One process manage the I/Os (Open File handler, read, write)



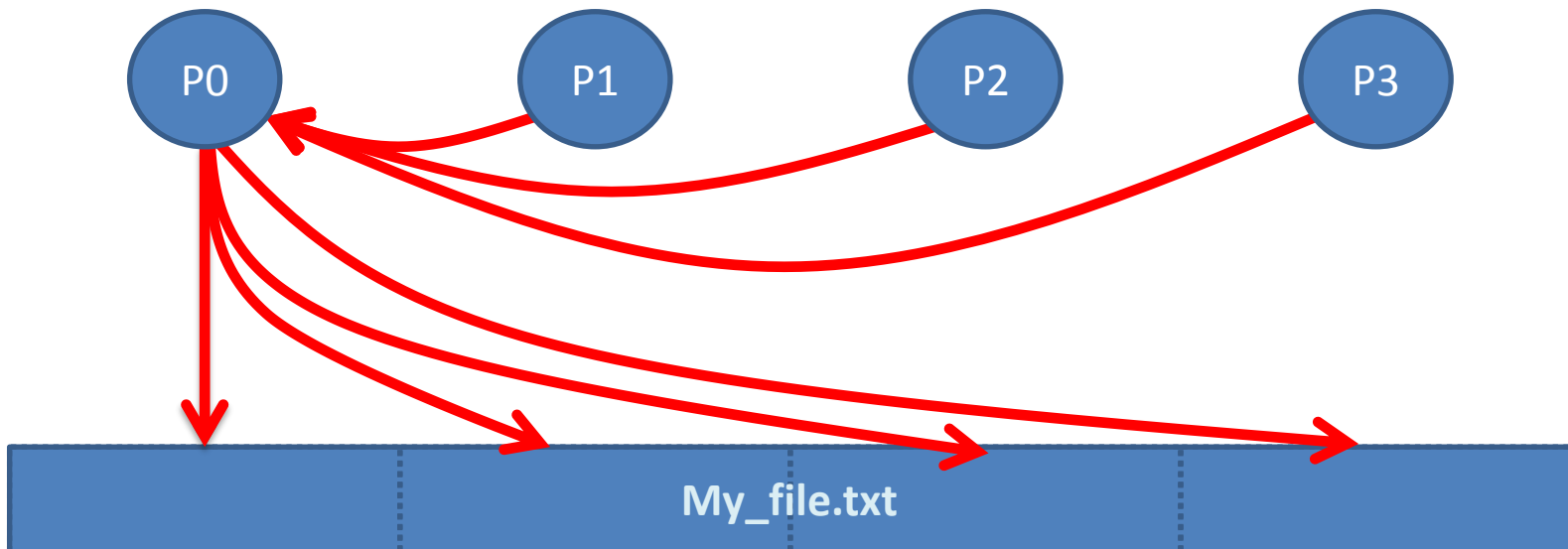
Parallel One-to-One I/O

- Based on legacy sequential I/O
 - One process manage the I/Os (Open File handler, read, write)
- The other processes “delegate” their I/Os to the chosen one



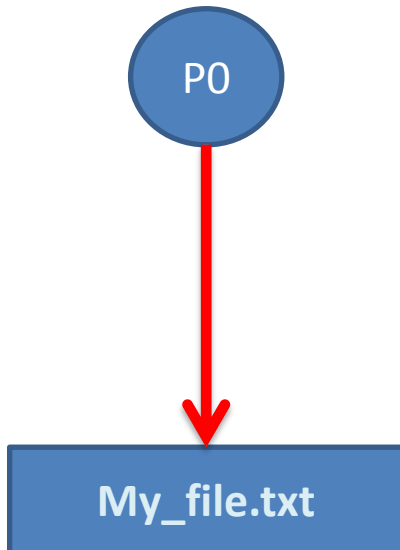
Parallel One-to-One I/O

- Based on legacy sequential I/O
 - One process manage the I/Os (Open File handler, read, write)
- The other processes “delegate” their I/Os to the chosen one
 - ✓ Fit for parallel systems and I/O library not supporting parallel I/O
 - ✓ Only one result file: easy to read, to move...
 - ✗ No parallelism: poor scalability -> poor performances



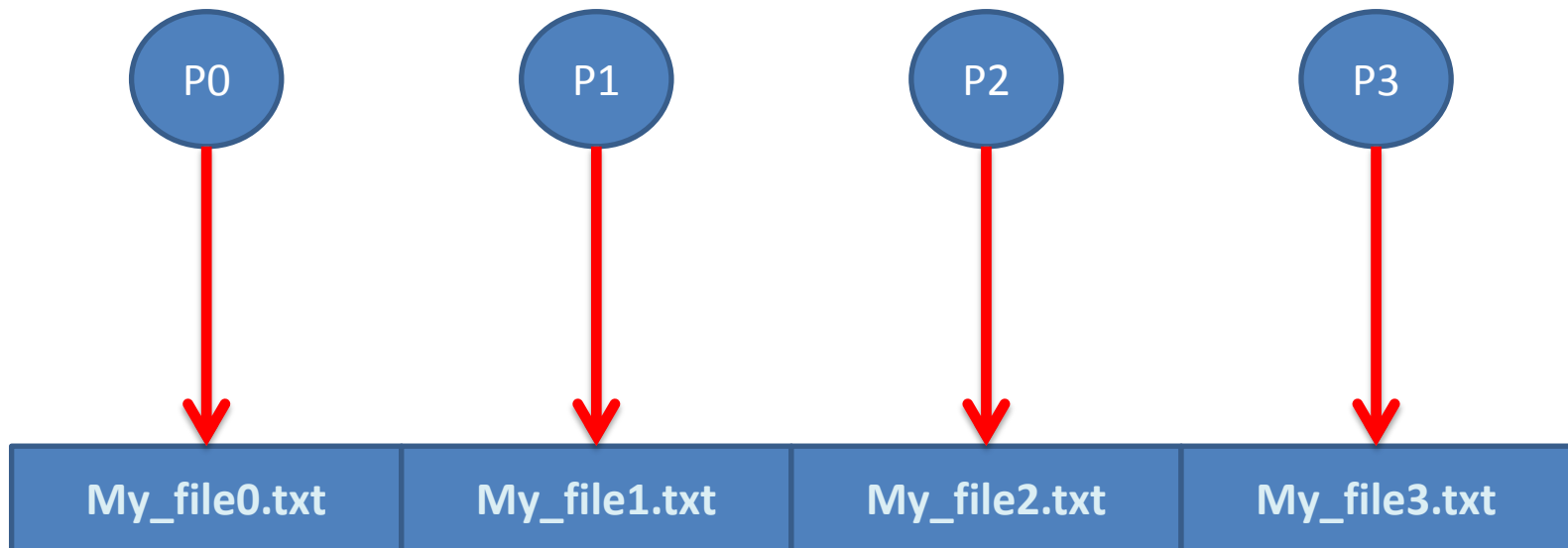
Parallel All-to-All I/O

- Based on legacy sequential I/O
 - One process manages its own I/O



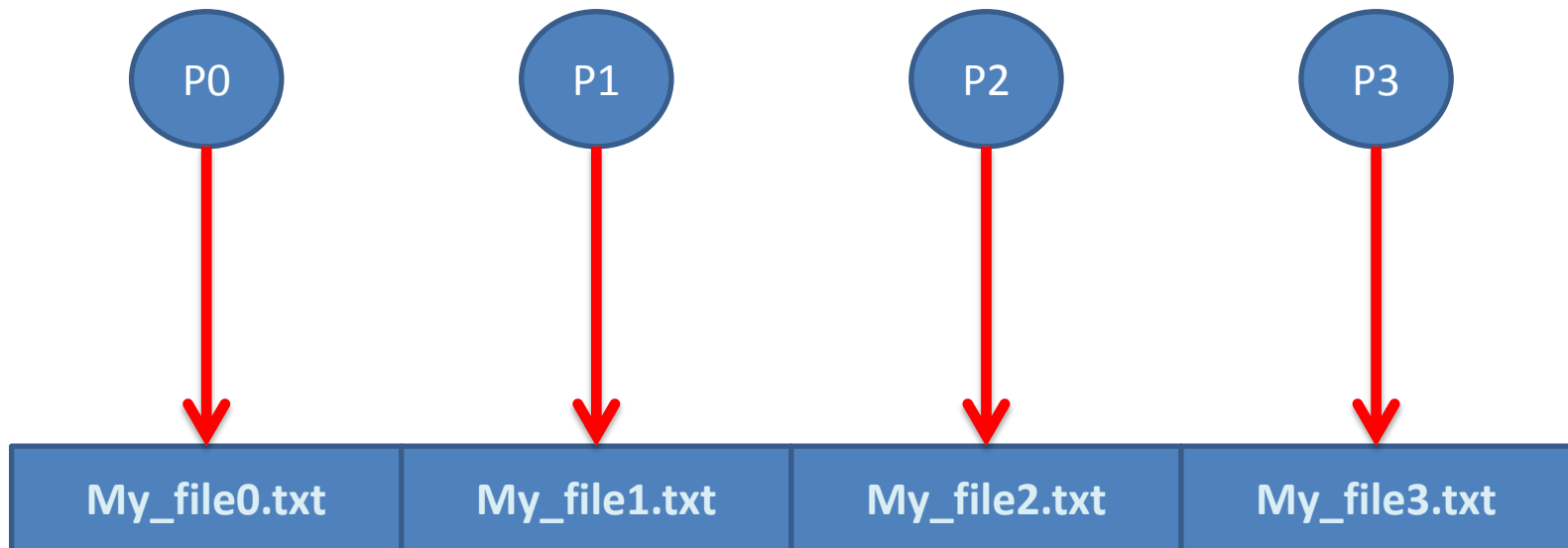
Parallel All-to-All I/O

- Based on legacy sequential I/O
 - One process manages its own I/O
- Every process only manages its own handler file



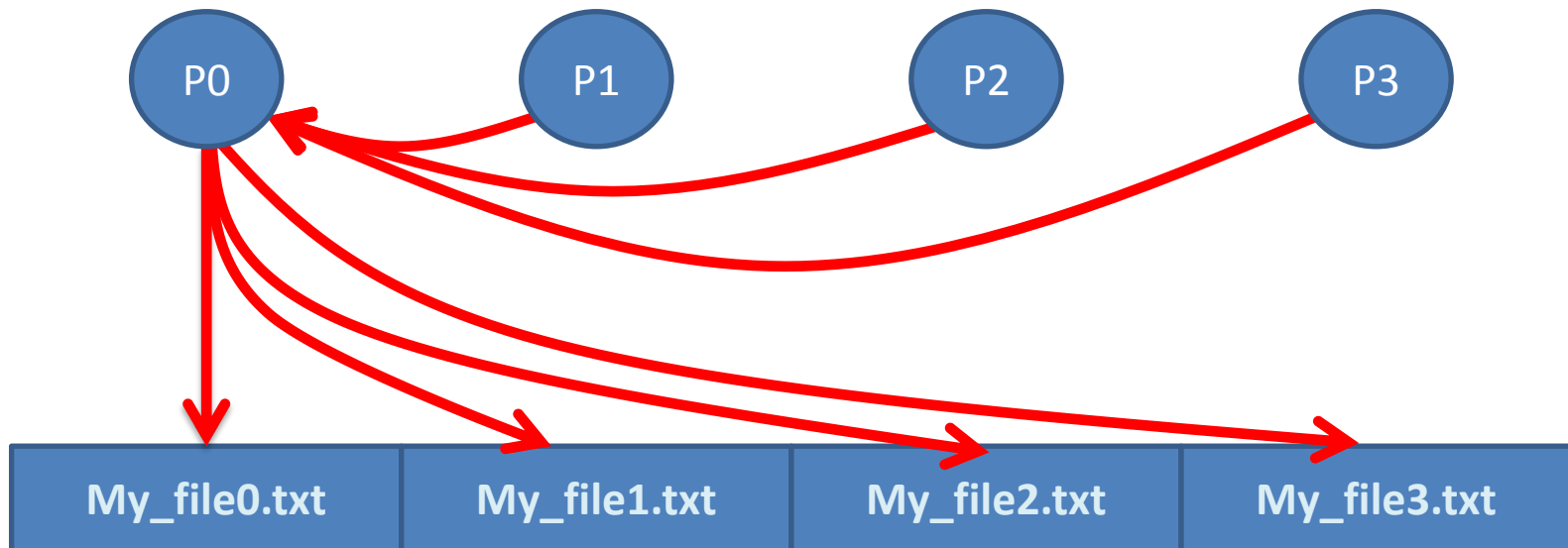
Parallel All-to-All I/O

- Based on legacy sequential I/O
 - One process manages its own I/O
- Every process only manages its own handler file
 - ✓ Parallelism: no bottlenecks on processes
 - ✗ Lots of small files to manages
 - ✗ Difficult to read back from different number of processes



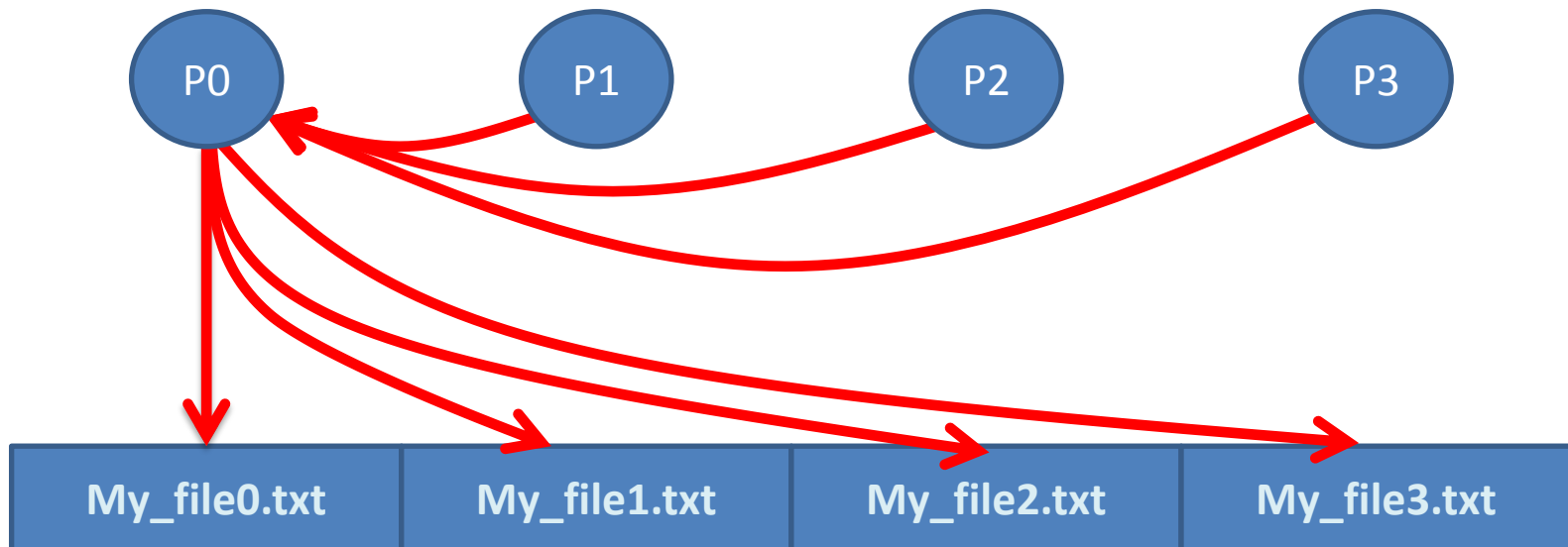
Parallel One-to-All I/O

- One process manage the I/Os (Open File handler, read, write)
- The other processes “delegate” their I/Os to the chosen one
- The I/Os are spread across multiple files



Parallel One-to-All I/O

- One process manage the I/Os (Open File handler, read, write)
- The other processes “delegate” their I/Os to the chosen one
- The I/Os are spread across multiple files
 - ❌ Bottleneck on the I/O manager process
 - ❌ Multiple file handlers to manage by chosen process
 - ❌ Multiple result files to manage



Parallel One-to-All I/O

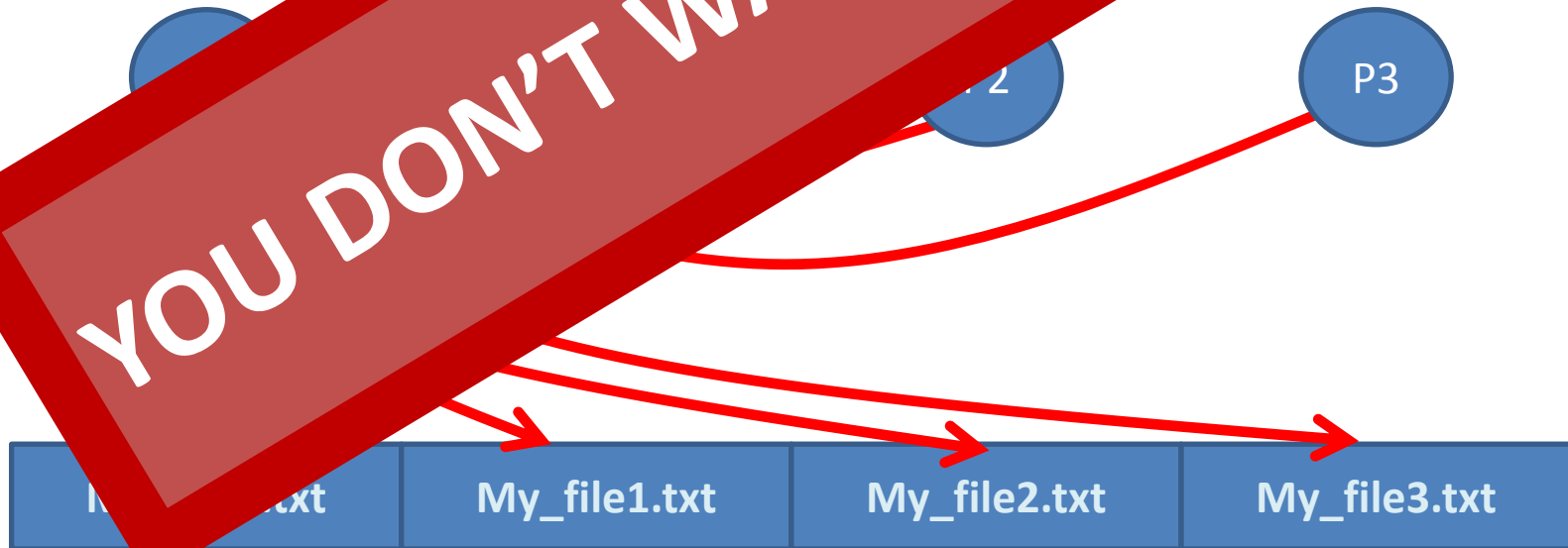
- One process manage the I/Os (Open, Read, Write, Close)
- The other processes “delegate”
- The I/Os are spread across

❌ Bottleneck

❌ Multiple

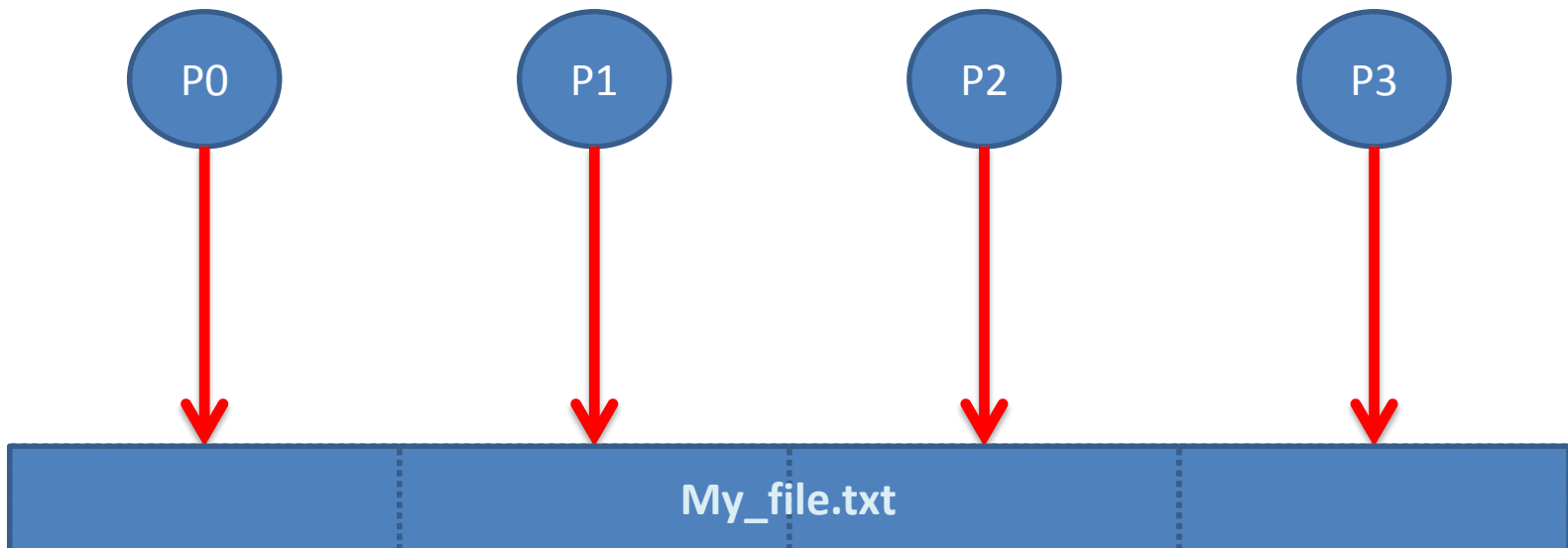
❌

YOU DON'T WANT TO DO THAT!



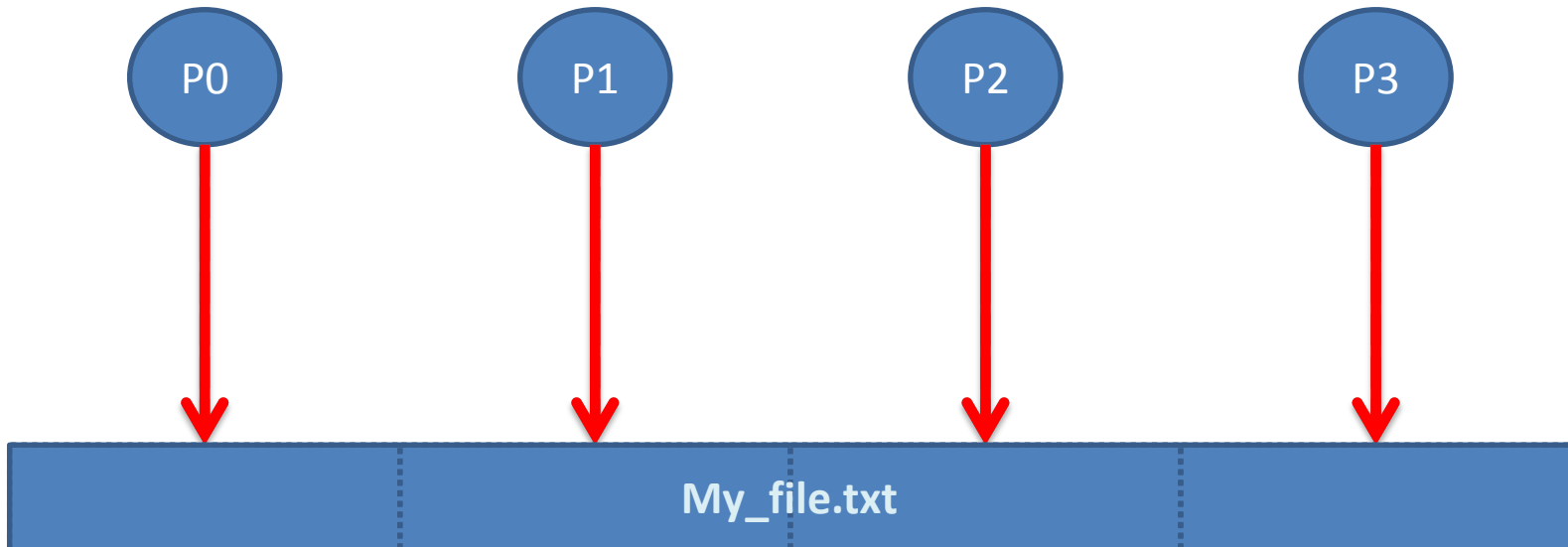
Parallel All-to-one I/O

- Every process manages its own I/Os
- Cooperative access to the same file to write data in parallel



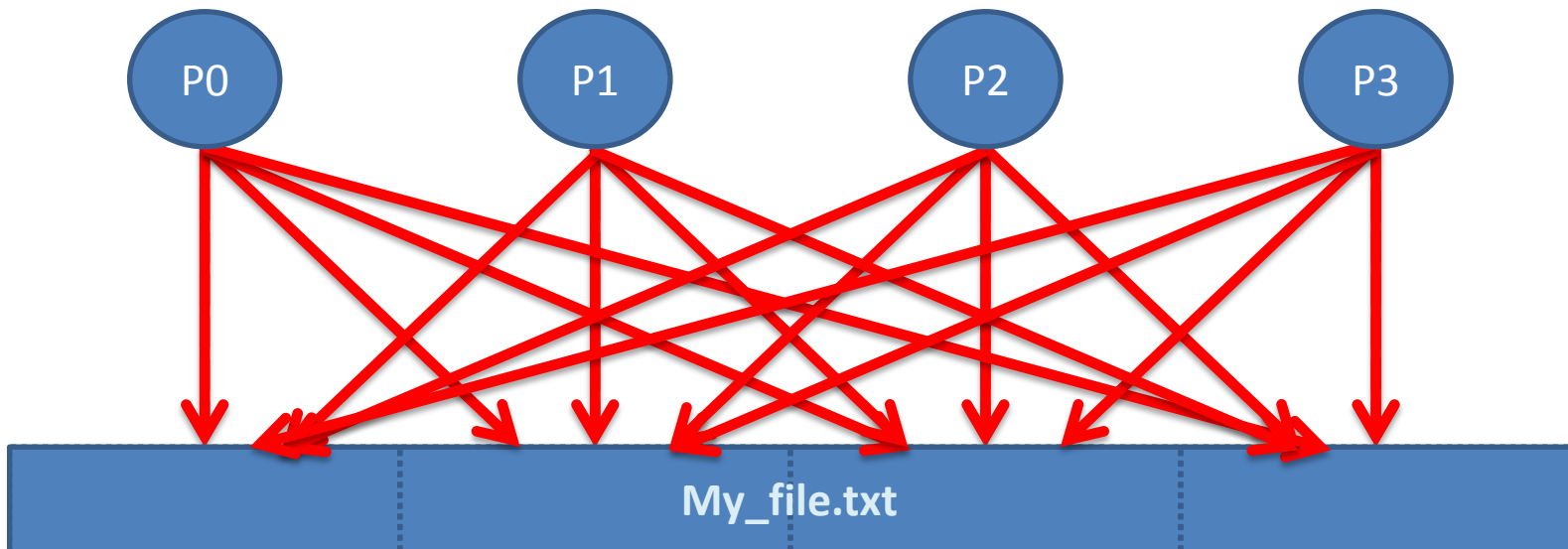
Parallel All-to-one I/O

- Every process manages its own I/Os
- Cooperative access to the same file to write data in parallel
 - ✓ Parallelism: no bottlenecks on processes
 - ✓ Only one result file to manage
 - ✗ `/!\` ordering when multiple processes are writing to the same address



Parallel All-to-one I/O

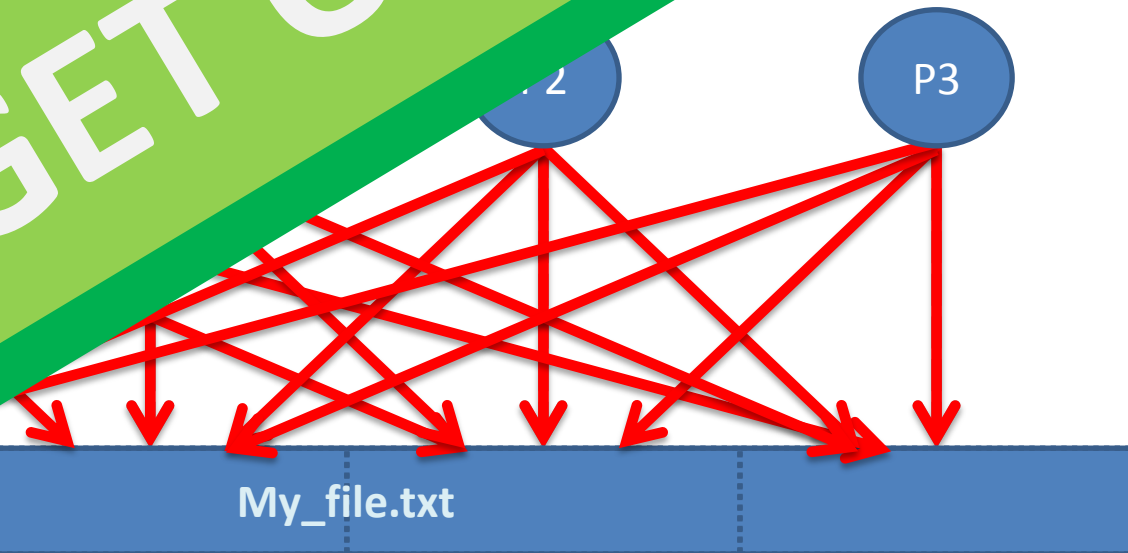
- Every process manages its own I/Os
- Cooperative access to the same file to write data in parallel
 - ✓ Parallelism: no bottlenecks on processes
 - ✓ Only one result file to manage
 - ✗ /!\ ordering when multiple processes are writing to the same address
 - ✓ If handled correctly, all processes should be able to write all over the file



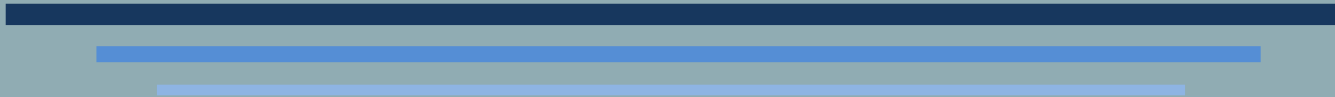
Parallel All-to-one I/O

- Every process manages its own I/Os
- Cooperative access to the same

- ✓ Parallelism: no bottlenecks
- ✓ Only one request per process
- ✗ /!\ order of access is not the same address
- ✓ If processes write to the same file, they must write to different locations to avoid overwriting



MPI I/O VIEW SYSTEM



What are MPI I/O views?



- MPI I/O targets the One-to-All parallel I/O model
 - Each process should be able to write anywhere in the file
- Problem with this model: race conditions if multiple processes try to write at the same address
 - At least contention if they try to read the same address
- Each MPI process should have its own accessible area
- A *view* defines, for a process, the current set of data visible and accessible from an open file

MPI I/O definitions: etype



- **etype**: An *etype* (*elementary* datatype) is the unit of data access and positioning
 - From MPI I/O point of view, a file is a contiguous set of etypes

MPI I/O definitions: etype



- **etype**: An *etype* (*elementary* datatype) is the unit of data access and positioning
 - From MPI I/O point of view, a file is a contiguous set of etypes
- Example: consider the following memory area of a file



MPI I/O definitions: etype

- **etype**: An *etype* (*elementary datatype*) is the unit of data access and positioning

- From MPI I/O point of view, a file is a contiguous set of etypes

- Example: consider the following memory area of a file



- if etypes=MPI_DOUBLE, this file is composed of 9 elements



MPI I/O definitions: etype

- **etype**: An *etype* (*elementary* datatype) is the unit of data access and positioning

- From MPI I/O point of view, a file is a contiguous set of etypes

- Example: consider the following memory area of a file



- if etypes=MPI_DOUBLE, this file is composed of 9 elements



- If etypes=MPI_FLOAT, the same file is composed of 18 elements



MPI I/O definitions: filetype



- **filetype**: A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype, or a derived MPI datatype constructed from multiple instances of the same etype (including the holes).

MPI I/O definitions: filetype



- **filetype**: A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype, or a derived MPI datatype constructed from multiple instances of the same etype (including the holes).
- Example: consider the following etype:



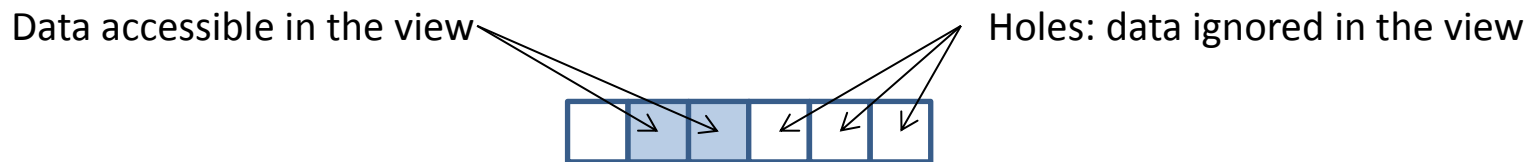
MPI I/O definitions: filetype

- **filetype**: A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype, or a derived MPI datatype constructed from multiple instances of the same etype (including the holes).

- Example: consider the following etype:



- An example of filetype constructed from this etype:



MPI I/O definitions: displacement



- **displacement**: A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a view begins.

MPI I/O definitions: displacement



- **displacement**: A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a view begins.
- Example: consider the following memory area of a file



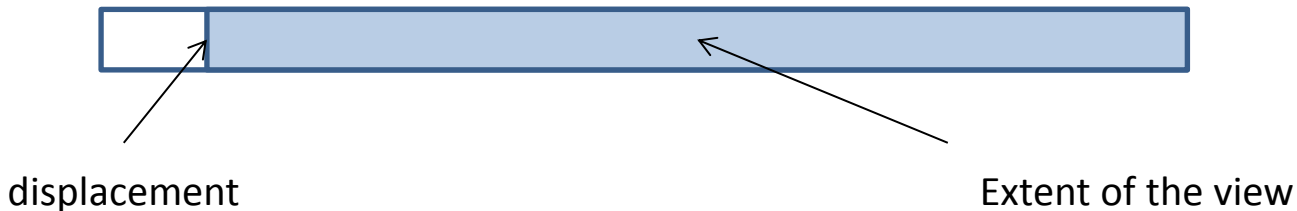
MPI I/O definitions: displacement

- **displacement**: A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a view begins.

- Example: consider the following memory area of a file



- Example of displacement



MPI I/O definitions: view



- **view**: A *view* is defined by three quantities: a displacement, an etype and a filetype. The pattern described by the filetype is repeated, beginning at the displacement, to define the complete view.

MPI I/O definitions: view





- **view**: A *view* is defined by three quantities: a displacement, an etype and a filetype. The pattern described by the filetype is repeated, beginning at the displacement, to define the complete view.
- Example from the MPI standard (Chapter 13 MPI I/O)

MPI I/O definitions: view



- **view**: A *view* is defined by three quantities: a displacement, an etype and a filetype. The pattern described by the filetype is repeated, beginning at the displacement, to define the complete view.
- Example from the MPI standard (Chapter 13 MPI I/O)
 - etype: ☐


MPI I/O definitions: view

- **view**: A *view* is defined by three quantities: a displacement, an etype and a filetype. The pattern described by the filetype is repeated, beginning at the displacement, to define the complete view.
- Example from the MPI standard (Chapter 13 MPI I/O)
 - etype: 
 - filetype: 

MPI I/O definitions: view

- **view**: A *view* is defined by three quantities: a displacement, an etype and a filetype. The pattern described by the filetype is repeated, beginning at the displacement, to define the complete view.

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: 


- filetype: 

- file: 

MPI I/O definitions: view

- **view**: A *view* is defined by three quantities: a displacement, an etype and a filetype. The pattern described by the filetype is repeated, beginning at the displacement, to define the complete view.

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: 

- filetype: 


- file: 

- displacement: 8

MPI I/O definitions: view

- **view**: A *view* is defined by three quantities: a displacement, an etype and a filetype. The pattern described by the filetype is repeated, beginning at the displacement, to define the complete view.

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: 

- filetype: 

- file: 

- displacement: 8


- **view**:



MPI I/O definitions: view

- **view**: A *view* is defined by three quantities: a displacement, an etype and a filetype. The pattern described by the filetype is repeated, beginning at the displacement, to define the complete view.

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: 

- filetype: 

- file: 

- displacement: 8


- **view**:



MPI I/O definitions: view

- **view**: A *view* is defined by three quantities: a displacement, an etype and a filetype. The pattern described by the filetype is repeated, beginning at the displacement, to define the complete view.

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: 

- filetype: 

- file: 

- displacement: 8

- **view**:



Sharing a file



- A group of processes sharing the same file may use complementary views to achieve a global data distributio

Sharing a file





- A group of processes sharing the same file may use complementary views to achieve a global data distribution
- Example from the MPI standard (Chapter 13 MPI I/O)

Sharing a file

- A group of processes sharing the same file may use complementary views to achieve a global data distribution
- Example from the MPI standard (Chapter 13 MPI I/O)
 - etype: ☐

Sharing a file

- A group of processes sharing the same file may use complementary views to achieve a global data distribution
- Example from the MPI standard (Chapter 13 MPI I/O)
 - etype: 
 - Process 0 filetype: 

Sharing a file

- A group of processes sharing the same file may use complementary views to achieve a global data distribution

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: 

- Process 0 filetype:



- Process 1 filetype:



Sharing a file

- A group of processes sharing the same file may use complementary views to achieve a global data distribution

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: ☐

- Process 0 filetype:



- Process 1 filetype:



- Process 2 filetype:





- [illegible]

Sharing a file

- A group of processes sharing the same file may use complementary views to achieve a global data distribution

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: 

- Process 0 filetype:



- Process 1 filetype:



- Process 2 filetype:



- Data covered by all the views:




Sharing a file

- A group of processes sharing the same file may use complementary views to achieve a global data distribution

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: 

- Process 0 filetype: 

- Process 1 filetype: 

- Process 2 filetype: 

- Data covered by all the views:




Sharing a file

- A group of processes sharing the same file may use complementary views to achieve a global data distribution

- Example from the MPI standard (Chapter 13 MPI I/O)

- etype: 

- Process 0 filetype: 

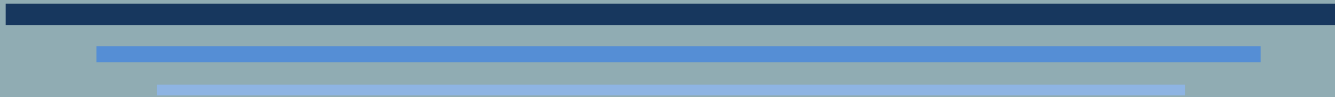
- Process 1 filetype: 

- Process 2 filetype: 

- Data covered by all the views:



BUILDING FILETYPES



Building filetypes with holes



- Filetypes are datatypes, and they can be build from with any datatype constructors, as long as it follows these rules:
 - Filetypes are only composed of a single datatype (etype)
 - Holes in the datatypes are multiple of the extent of this etype
- However, from functions seen in PP-C3, we can only build datatypes with no holes at the beginning and the end
- How to build the filetypes from the MPI standard example?

Resizing datatypes



- `MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype);`
 - IN oldtype input datatype (handle)
 - IN lb new lower bound of datatype (integer)
 - IN extent new extent of datatype (integer)
 - OUT newtype output datatype (handle)
- `MPI_Type_create_resized` takes an old datatype and changes the lower bound and the size
- To insert holes before the datatype, lower bound must be negative
- To insert holes after the datatype, extent must be greater than the original datatype size.
- Lower bound and extent are given in bytes

Example from MPI I/O chapter





```
MPI_Datatype datatmp, filetype;  
MPI_Type_contiguous(2, MPI_INT, &datatmp)  
MPI_Aint lower_bound=-1*sizeof(int)  
MPI_Aint extent = 6*sizeof(int)  
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);  
MPI_Type_commit(&filetype);
```

■ oldtype = int ■

Example from MPI I/O chapter






```
MPI_Datatype datatmp, filetype;  
MPI_Type_contiguous(2, MPI_INT, &datatmp)  
MPI_Aint lower_bound=-1*sizeof(int)  
MPI_Aint extent = 6*sizeof(int)  
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);  
MPI_Type_commit(&filetype);
```

- oldtype = int 
- datatmp: 





Example from MPI I/O chapter

```
MPI_Datatype datatmp, filetype;  
MPI_Type_contiguous(2, MPI_INT, &datatmp)  
MPI_Aint lower_bound=-1*sizeof(int)  
MPI_Aint extent = 6*sizeof(int)  
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);  
MPI_Type_commit(&filetype);
```

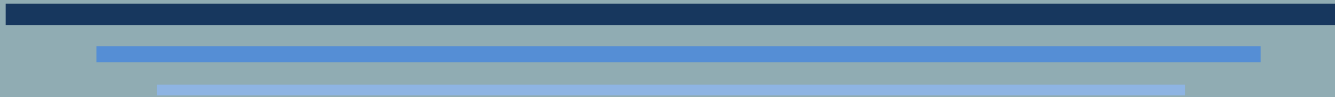
- oldtype = int 
- datatmp: 
- datatmp + lower_bound: 

Example from MPI I/O chapter

```
MPI_Datatype datatmp, filetype;  
MPI_Type_contiguous(2, MPI_INT, &datatmp)  
MPI_Aint lower_bound=-1*sizeof(int)  
MPI_Aint extent = 6*sizeof(int)  
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);  
MPI_Type_commit(&filetype);
```

- oldtype = int 
- datatmp: 
- datatmp + lower_bound: 
- filetype: 

FILE MANAGEMENT



Opening a file

- `MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh);`
 - IN comm communicator (handle)
 - IN filename name of file to open (string)
 - IN amode file access mode (integer)
 - IN info info object (handle)
 - OUT fh new file handle (handle)
- Collective function
 - Each process must specify same amode, and referencing the same file
- Amode defines the access mode when opening the file
 - Read only, write only, read-write,...
- When opening a file, the default view is a linear byte stream
 - Displacement=0, etype=filetype=MPI_BYTE)

Available amode

- MPI_MODE_RDONLY : file will only be read
- MPI_MODE_RDWR : file will be read and written
- MPI_MODE_WRONLY : file will only be written
- MPI_MODE_CREATE : file is created if it does not exist
- MPI_MODE_EXCL : error if creating file that already exists
- MPI_MODE_DELETE_ON_CLOSE : delete file when closing the file handler
- MPI_MODE_UNIQUE_OPEN : file will not be concurrently opened elsewhere,
- MPI_MODE_SEQUENTIAL : file will only be accessed sequentially
- MPI_MODE_APPEND : set initial position of all file pointers to end of file

Changing file view

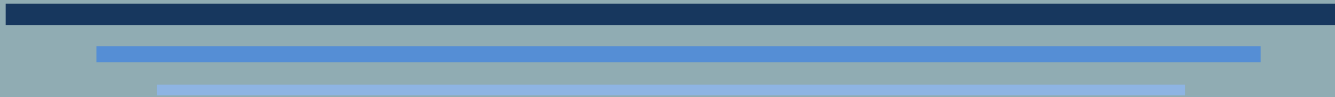
- `MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, const char *datarep, MPI_Info info);`
 - INOUT fh file handle (handle)
 - IN disp displacement (integer)
 - IN etype elementary datatype (handle)
 - IN filetype filetype (handle)
 - IN datarep data representation (string)
 - IN info info object (handle)
- Attach a new view to an opened file
- Collective function
 - Each process must specify same etype and datarep
- Datarep specify how the data will be written in memory
 - Native, internal and extern32

Closing and deleting file



- `MPI_File_close(MPI_File *fh);`
- Close the specified file handler
 - If file was opened with `MPI_MODE_DELETE_ON_CLOSE`, file is deleted
- Collective function
- `MPI_File_delete(const char *filename, MPI_Info info);`
- Delete the specified file
 - If file does not exist, an error class `MPI_ERR_NO_SUCH_FILE` is raised

READING AND WRITING ROUTINES



Individual read

- `MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);`
 - INOUT fh file handle (handle)
 - OUT buf initial address of buffer to put read data
 - IN count number of elements in buffer (integer)
 - IN datatype datatype of each buffer element (handle)
 - OUT status status object (Status)
- Individual function
- *count* and *datatype* specifies the number of elements and the type of elements to read and put in buffer *buf*
 - Much like in a MPI_Recv function
- Read contiguous elements according to the view associated with the file

individual reading example

/*filetype from previous example “sharing a file”*/

MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);

MPI_Type_commit(&filetype);

Int buf[7];

MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);

■ File:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----



```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
Int buf[7];
```

```
MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);
```

■ View: 

individual reading example

/*filetype from previous example "sharing a file"*/

MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);

MPI_Type_commit(&filetype);

Int buf[7];

MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);

- File:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----
- View:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
- To read: 7 MPI_INT

individual reading example

/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[7];
```

```
MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);
```

■ File:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

■ View:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

■ To read: 7 MPI_INT

■ File+view:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

individual reading example

/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[7];
```

```
MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);
```

- File:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----
- View:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
- To read: 7 MPI_INT
- File+view:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----
- buffer:

--	--	--	--	--	--	--

individual reading example

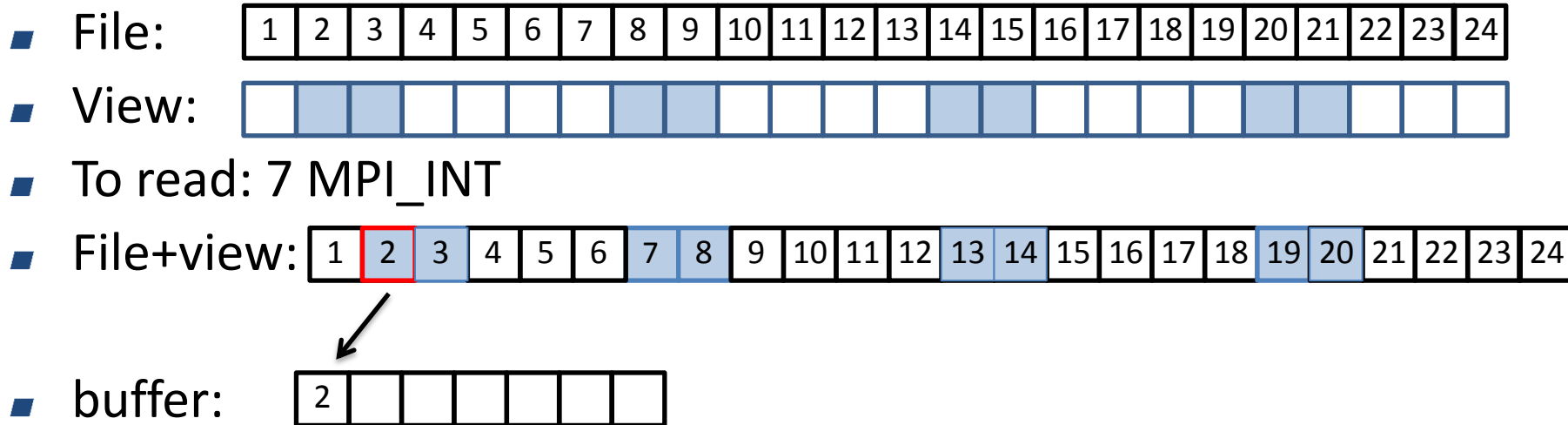
/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[7];
```

```
MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);
```



individual reading example

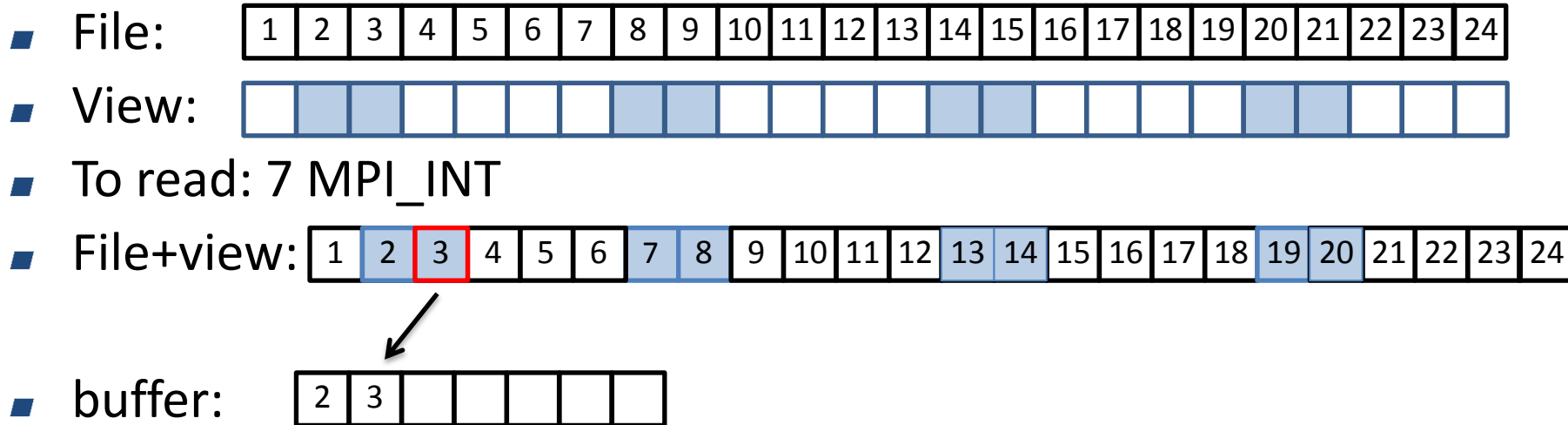
/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[7];
```

```
MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);
```



individual reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[7];
```

```
MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);
```

- File:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----
- View:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
- To read: 7 MPI_INT
- File+view:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----
- buffer:

2	3	7				
---	---	---	--	--	--	--

individual reading example

/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[7];
```

```
MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);
```

- File:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----
- View:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
- To read: 7 MPI_INT
- File+view:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----
- buffer:

2	3	7	8			
---	---	---	---	--	--	--

individual reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[7];
```

```
MPI_File_read(fh, &buf, 7, MPI_INT, MPI_STATUS_IGNORE);
```

- File:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----
- View:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
- To read: 7 MPI_INT
- File+view:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----
- buffer:

2	3	7	8	13	14	19
---	---	---	---	----	----	----

Individual write

- `MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status);`
 - INOUT fh file handle (handle)
 - IN buf initial address of buffer (choice)
 - IN count number of elements in buffer (integer)
 - IN datatype datatype of each buffer element (handle)
 - OUT status status object (Status)
- Individual function
- *count* and *datatype* specifies the number of elements and the type of elements to write from buffer *buf*
 - Much like in a MPI_Send function
- Write contiguous elements according to the view associated with the file



```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```

[illegible]

individual writing example

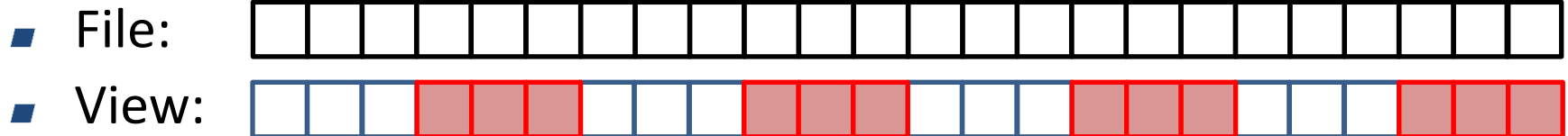
/*filetype from previous example “sharing a file”*/

MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);

MPI_Type_commit(&filetype);

Int buf[10] = {1,2,3,4,5,6,7,8,9,10};

MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);



individual writing example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```

- File:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
- View:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
- To read: 10 MPI_INT

individual writing example

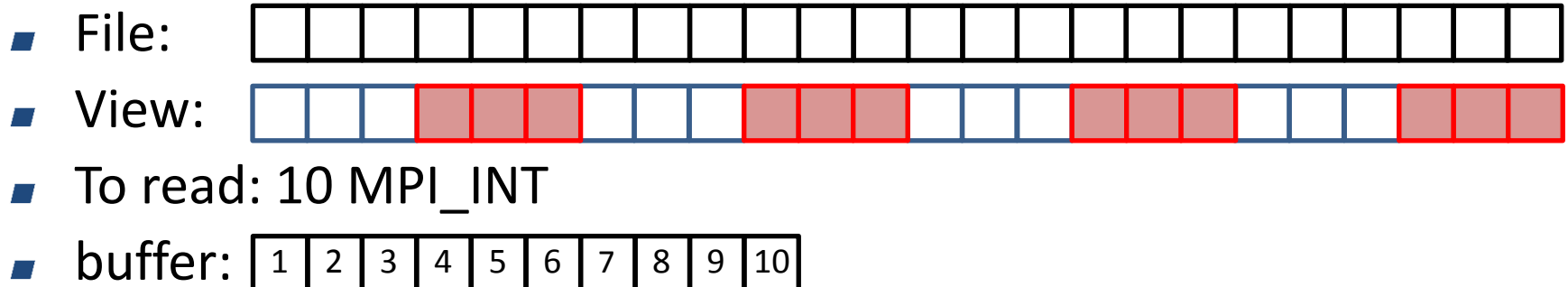
/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```









```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```

- File: 
- View: 
- To read: 10 MPI_INT
- buffer: 
- File: 

individual writing example

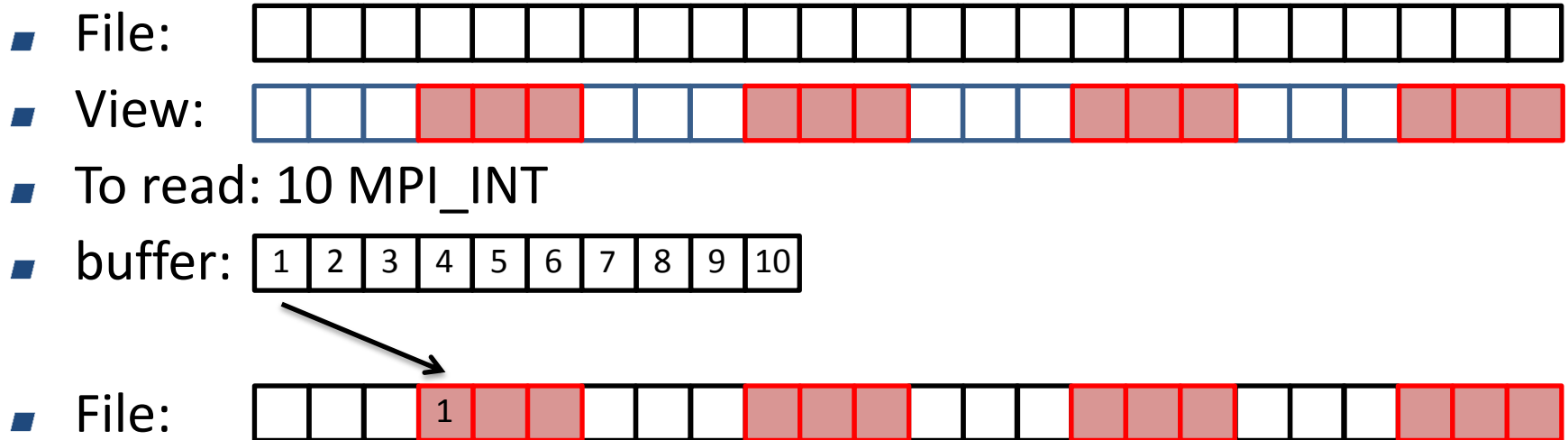
/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```



individual writing example

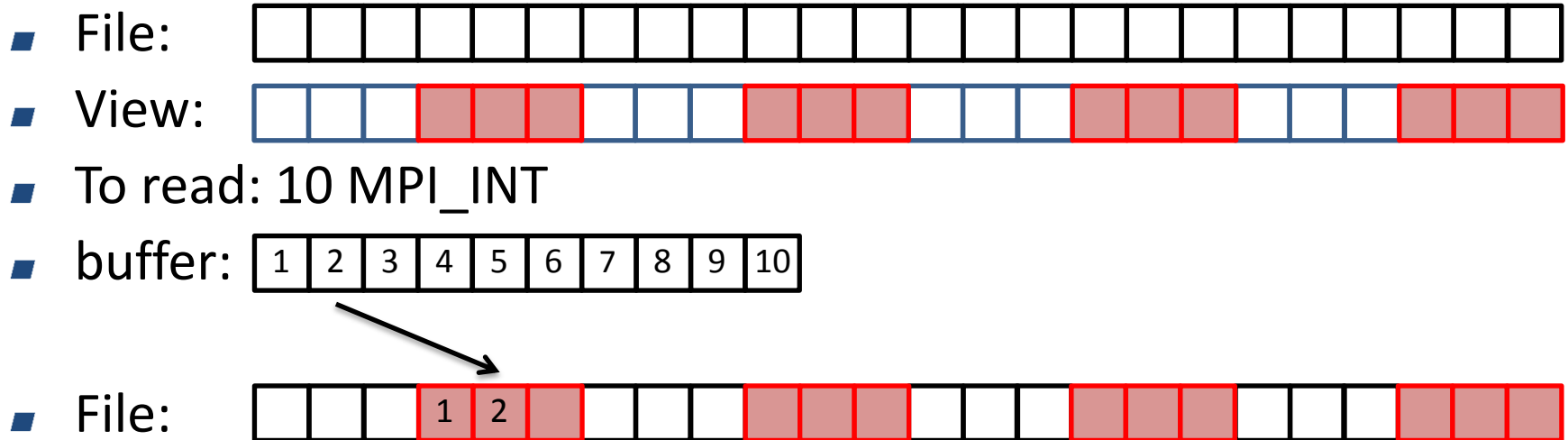
/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```



individual writing example

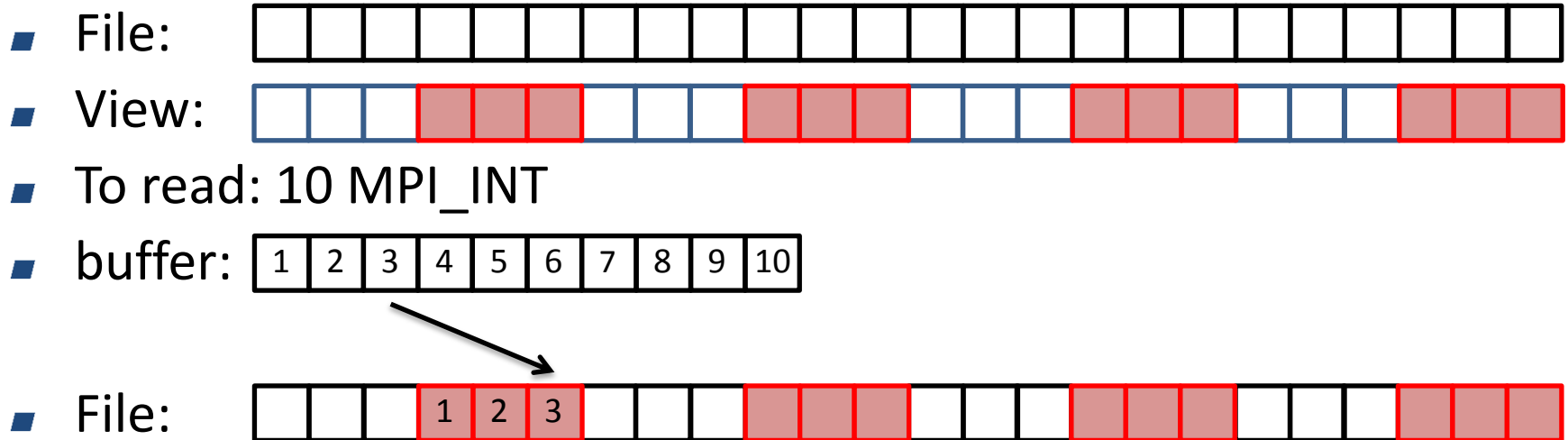
/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```



individual writing example

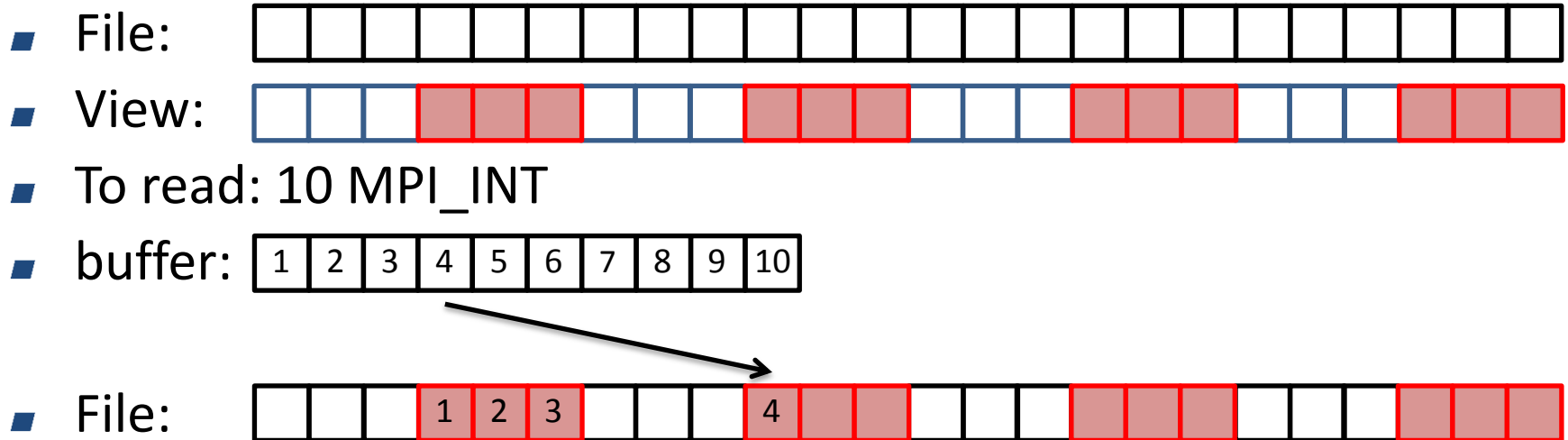
/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```



individual writing example

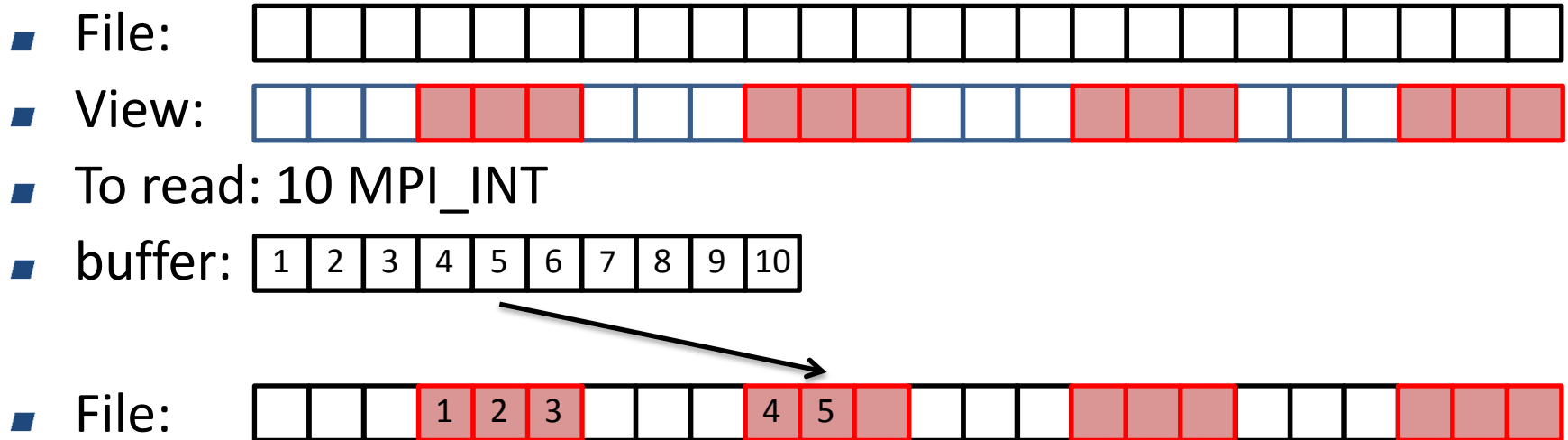
/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```



individual writing example

/*filetype from previous example "sharing a file"*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf[10] = {1,2,3,4,5,6,7,8,9,10};
```

```
MPI_File_write(fh, &buf, 10, MPI_INT, MPI_STATUS_IGNORE);
```

- File:



--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
- View:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
- To read: 10 MPI_INT
- buffer:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----
- File:

			1	2	3				4	5	6				7	8	9				10		
--	--	--	---	---	---	--	--	--	---	---	---	--	--	--	---	---	---	--	--	--	----	--	--

Collective read

- 
- 
- `MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);`
 - INOUT fh file handle (handle)
 - OUT buf initial address of buffer (choice)
 - IN count number of elements in buffer (integer)
 - IN datatype datatype of each buffer element (handle)
 - OUT status status object (Status)
 - Collective function
 - Each process must reference the same file
 - *count* and *datatype* specifies the number of elements and the type of elements to read and put in buffer *buf*
 - Much like in a `MPI_Recv` function
 - Read contiguous elements according to the each process's view associated with the file

Collective reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf...;
```

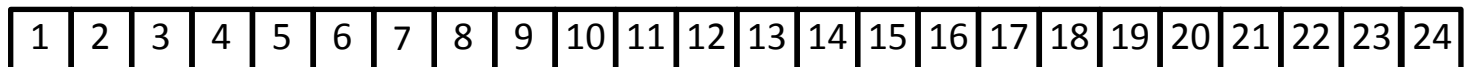
```
MPI_File_write(fh, &buf, count, MPI_INT, MPI_STATUS_IGNORE);
```

■ Process 0 filetype:  count=4

■ Process 1 filetype:  count=5

■ Process 2 filetype:  count=10

■ File:



■ bufs



Collective reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf...;
```

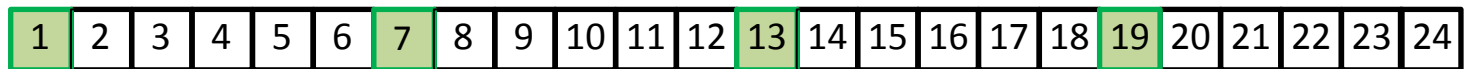
```
MPI_File_write(fh, &buf, count, MPI_INT, MPI_STATUS_IGNORE);
```

■ Process 0 filetype:  count=4

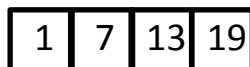
■ Process 1 filetype:  count=5

■ Process 2 filetype:  count=10

■ File:



■ bufs



Collective reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf...;
```

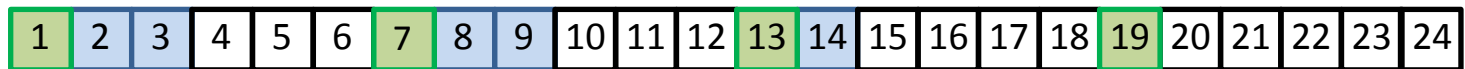
```
MPI_File_write(fh, &buf, count, MPI_INT, MPI_STATUS_IGNORE);
```

■ Process 0 filetype:  count=4

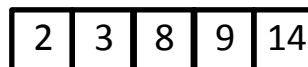
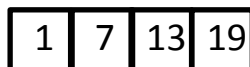
■ Process 1 filetype:  count=5

■ Process 2 filetype:  count=10

■ File:



■ bufs



Collective reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf...;
```

```
MPI_File_write(fh, &buf, count, MPI_INT, MPI_STATUS_IGNORE);
```

■ Process 0 filetype:  count=4

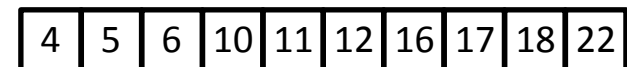
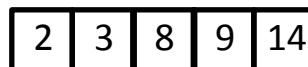
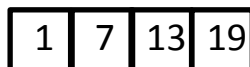
■ Process 1 filetype:  count=5

■ Process 2 filetype:  count=10

■ File:



■ bufs



Collective write

- `MPI_File_write_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status);`
 - INOUT fh file handle (handle)
 - IN buf initial address of buffer (choice)
 - IN count number of elements in buffer (integer)
 - IN datatype datatype of each buffer element (handle)
 - OUT status status object (Status)
- Collective function
 - Each process must reference the same file
- *count* and *datatype* specifies the number of elements and the type of elements to write from buffer *buf*
 - Much like in a `MPI_Send` function
- Write contiguous elements according to the each process's view associated with the file



```
MPI_File_write(fh, &buf, count, MPI_INT, MPI_STATUS_IGNORE);
```

- File

Collective reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf...;
```

```
MPI_File_write(fh, &buf, count, MPI_INT, MPI_STATUS_IGNORE);
```

- Process 0 filetype:

--	--	--	--	--	--

 count=4
- Process 1 filetype:

--	--	--	--	--	--

 count=5
- Process 2 filetype:

--	--	--	--	--	--

 count=10

- bufs:

1	2	3	4
---	---	---	---

20	21	22	23	24
----	----	----	----	----

40	41	42	43	44	45	46
----	----	----	----	----	----	----

- File

1						2						3						4							
---	--	--	--	--	--	---	--	--	--	--	--	---	--	--	--	--	--	---	--	--	--	--	--	--	--

Collective reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf...;
```

```
MPI_File_write(fh, &buf, count, MPI_INT, MPI_STATUS_IGNORE);
```

- Process 0 filetype:

--	--	--	--	--	--

 count=4
- Process 1 filetype:

--	--	--	--	--	--

 count=5
- Process 2 filetype:

--	--	--	--	--	--

 count=10

- bufs:

1	2	3	4
---	---	---	---

20	21	22	23	24
----	----	----	----	----

40	41	42	43	44	45	46
----	----	----	----	----	----	----

- File

1	20	21				2	22	23				3	24					4				
---	----	----	--	--	--	---	----	----	--	--	--	---	----	--	--	--	--	---	--	--	--	--

Collective reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf...;
```

```
MPI_File_write(fh, &buf, count, MPI_INT, MPI_STATUS_IGNORE);
```

- Process 0 filetype:

1	2	3	4		
---	---	---	---	--	--

 count=4
- Process 1 filetype:

	20	21	22	23	24
--	----	----	----	----	----

 count=5
- Process 2 filetype:

				40	41
--	--	--	--	----	----

 count=10

- bufs:

1	2	3	4
---	---	---	---

20	21	22	23	24
----	----	----	----	----

40	41	42	43	44	45	46
----	----	----	----	----	----	----

- File

1	20	21		40	41	2	22	23		42	43	3	24			44	45	4				46	
---	----	----	--	----	----	---	----	----	--	----	----	---	----	--	--	----	----	---	--	--	--	----	--

Collective reading example

/*filetype from previous example “sharing a file”*/

```
MPI_Type_create_resized(datatmp, lower_bound, extent, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
Int buf...;
```

```
MPI_File_write(fh, &buf, count, MPI_INT, MPI_STATUS_IGNORE);
```

- Process 0 filetype:

--	--	--	--	--	--

 count=4
- Process 1 filetype:

--	--	--	--	--	--

 count=5
- Process 2 filetype:

--	--	--	--	--	--

 count=10

- bufs:

1	2	3	4
---	---	---	---

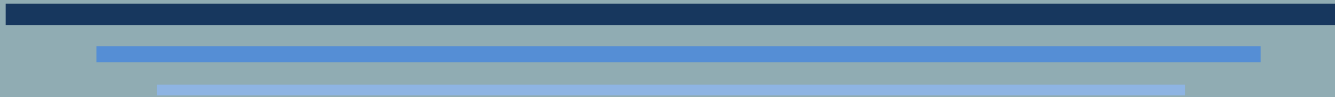
20	21	22	23	24
----	----	----	----	----

40	41	42	43	44	45	46
----	----	----	----	----	----	----

- File

1	20	21		40	41	2	22	23		42	43	3	24			44	45	4				46	
---	----	----	--	----	----	---	----	----	--	----	----	---	----	--	--	----	----	---	--	--	--	----	--

ADVANCED MPI I/O CALLS



Shared pointer routine



- The previous functions have individual file pointers
 - Each process has its own file pointer locating where to write in the file
 - The file pointer is updated to the next location each time a data is read/written
- The same functions working on a shared file pointer exist
 - The file pointer is shared among all processes involved in the file opening
 - A read/write in the file update the pointer for all processes
 - Multiple parallel calls to shared pointer routine are serialized
- `/!\` all processes involved must use the same file view
- Collective calls are ordered according to their rank
- Same function call + `_shared` suffix

Individual file pointer individual read



- `MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);`
 - INOUT fh file handle (handle)
 - OUT buf initial address of buffer to put read data
 - IN count number of elements in buffer (integer)
 - IN datatype datatype of each buffer element (handle)
 - OUT status status object (Status)

Shared file pointer individual read

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

Shared file pointer individual write



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

Shared file pointer collective read

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

Shared file pointer collective write



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

Data access with explicit offset



- The previous functions use file pointers to “know” the offset where to write
- Read/write and collective functions also exist with explicit offset to specify
 - No file pointer is used nor updated
 - Offset is explicitly given as an argument of the function
- Same function call + `_at` suffix

Individual file pointer individual read

- 
- 
- `MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);`
 - INOUT fh file handle (handle)
 - OUT buf initial address of buffer to put read data
 - IN count number of elements in buffer (integer)
 - IN datatype datatype of each buffer element (handle)
 - OUT status status object (Status)

Explicit offset individual read

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

Explicit offset individual read

- `MPI_File_read_at(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);`
 - **IN** **fh** **file handle (handle)**
 - **IN** **offset** **file offset integer)**
 - **OUT** **buf** initial address of buffer (choice)
 - **IN** **count** number of elements in buffer (integer)
 - **IN** **datatype** datatype of each buffer element (handle)
 - **OUT** **status** status object (Status)
- *Fh* is not out anymore because no file pointer is updated
- *Offset* gives the beginning where to read data in the file

Explicit offset individual write

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

Explicit offset collective read

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

Explicit offset collective write

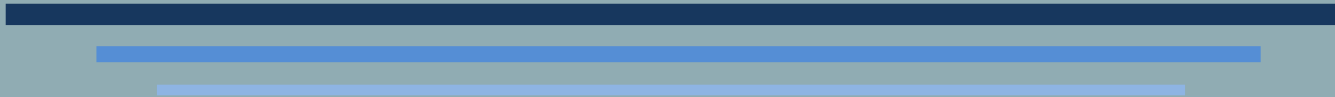
- `MPI_File_write_at_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);`
 - IN fh file handle (handle)
 - IN offset file offset integer)
 - OUT buf initial address of buffer (choice)
 - IN count number of elements in buffer (integer)
 - IN datatype datatype of each buffer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

Non-blocking routines



- All routines exist in non-blocking versions with the usual semantics
 - Function calls begin with an “I”
 - A request is given in argument, to be later passed to completion calls
- Another non-blocking semantics exist for MPI I/O collective calls: split call
 - A single collective operation is split in two: a begin and an end function
 - The operation is initiated by the `***_begin` call (much like `Iread_all` call)
 - The completion call is done by the `***_end` call (much like `wait` call)
- Rules to use split calls
 - No more that one active split collective at any moment
 - Begin calls are collective
 - No collective I/O are permitted within an active split collective access

DATA REPRESENTATION



How to write the data?



- So far, we have seen functions allowing to read and write data from files
- But what is the binary representation of these data?
 - Big indian? Little indian?
 - 16bits? 32bits? 64bits?
- If it is the binary representation of the current computer, the file may not be portable across platforms
- If it is a fixed representation, need to translate chosen representation to current computer representation

3 data representations in MPI I/O



- Native
 - Data in this representation is stored in a file exactly as it is in memory
 - data precision and I/O performance are not lost in type conversions with a purely homogeneous environment
 - Files are not portable across platforms
- internal
 - This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary.
 - Portable across platforms, but not across MPI implementations
- External32
 - read and write operations convert all data from and to the external32 representation completely defined in MPI I/O (Section 13.5.2)
 - Portable across platforms and MPI implementations
 - May be less performant due to all the conversions