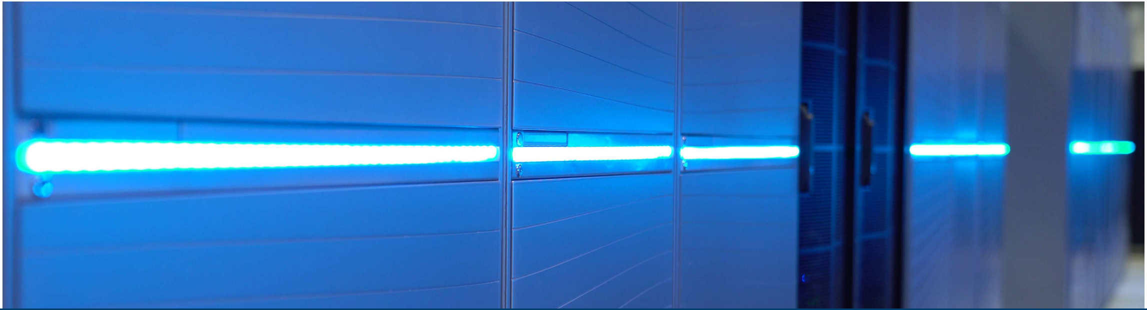


Part I: Input/Output

MOTIVATION

I/O on HPC Systems

- “This is not your parents’ I/O subsystem”
- File system is a shared resource
  - Modification of metadata might happen sequentially
  - File system blocks might be shared among processes
- File system access might not be uniform across all processes
- Interoperability of data originating on different platforms

MPI I/O

- MPI already defines a language that describes data layout and movement
- Extend this language by I/O capabilities
- More expressive/precise API than POSIX I/O affords better chances for optimization

COMMON I/O STRATEGIES

Funnelled I/O

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

All or several processes use one file

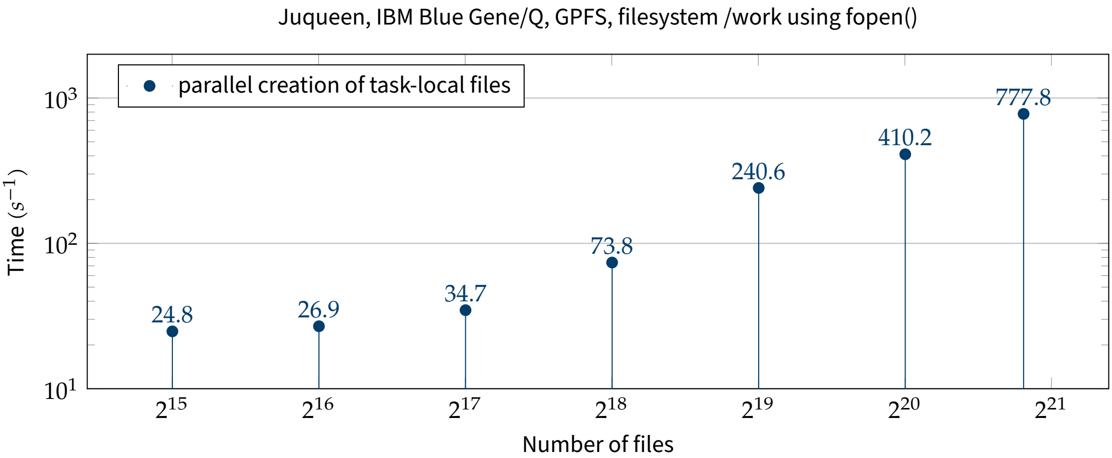
+ Number of files is independent of number of processes
+ File is in canonical representation (no post-processing)
- Uncoordinated client requests might induce time penalties
- File layout may induce false sharing of file system blocks

COMMON I/O STRATEGIES

Task-Local Files

+ Simple to implement
+ No explicit coordination between processes needed
+ No false sharing of file system blocks
- Number of files quickly becomes unmanageable
- Files often need to be merged to create a canonical dataset (post-processing)
- File system might introduce implicit coordination (metadata modification)

SEQUENTIAL ACCESS TO METADATA



FILE, FILE POINTER & HANDLE [MPI-4.0, 14.1]

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

File Pointer
A file pointer is an implicit offset into a file maintained by MPI.

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

OPENING A FILE [MPI-4.0, 14.2.1]

C

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

F08

```fortran
MPI_File_open(comm, filename, amode, info, fh, ierror)
type(MPI_Comm), intent(in) :: comm
character(len=*), intent(in) :: filename
integer, intent(in) :: amode
type(MPI_Info), intent(in) :: info
type(MPI_File), intent(out) :: fh
integer, optional, intent(out) :: ierror
```

- Collective operation on communicator `comm`
- Filename must reference the same file on all processes
- Process-local files can be opened using `MPI_COMM_SELF`
- `info` object specifies additional information (`MPI_INFO_NULL` for empty)

ACCESS MODE [MPI-4.0, 14.2.1]

amode denotes the access mode of the file and must be the same on all processes. It must contain exactly one of the following:

`MPI_MODE_RDONLY` read only access

`MPI_MODE_RDWR` read and write access

`MPI_MODE_WRONLY` write only access

and may contain some of the following:

`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_UNIQUE_OPEN` file is not opened elsewhere

`MPI_MODE_SEQUENTIAL` access to the file is sequential

`MPI_MODE_APPEND` file pointers are set to the end of the file

Combine using bit-wise or (| operator in C, `ior` intrinsic in Fortran).

CLOSING A FILE [MPI-4.0, 14.2.2]

C

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

F08

```fortran
MPI_File_close(fh, ierror)
type(MPI_File), intent(out) :: fh
integer, optional, intent(out) :: ierror
```

- Collective operation
- User must ensure that all outstanding nonblocking and split collective operations associated with the file have completed

DELETING A FILE [MPI-4.0, 14.2.3]

C

```c
int MPI_File_delete(const char* filename, MPI_Info info)
```

F08

```fortran
MPI_File_delete(filename, info, ierror)
character(len=*), intent(in) :: filename
type(MPI_Info), intent(in) :: info
integer, optional, intent(out) :: ierror
```

- Deletes the file identified by `filename`
- If the file does not exist an error is raised
- If the file is opened by any process
  - all further and outstanding access to the file is implementation dependent
  - it is implementation dependent whether the file is deleted; if it is not, an error is raised

FILE PARAMETERS

Setting File Parameters

`MPI_File_set_size`	Set the size of a file [MPI-4.0, 14.2.4]
`MPI_File_preallocate`	Preallocate disk space [MPI-4.0, 14.2.5]
`MPI_File_set_info`	Supply additional information [MPI-4.0, 14.2.8]

Inspecting File Parameters

`MPI_File_get_size`	Size of a file [MPI-4.0, 14.2.6]
`MPI_File_get_amode`	Acess mode [MPI-4.0, 14.2.7]
`MPI_File_get_group`	Group of processes that opened the file [MPI-4.0, 14.2.7]
`MPI_File_get_info`	Additional information associated with the file [MPI-4.0, 14.2.8]

I/O ERROR HANDLING [MPI-4.0, 9.3, 14.7]

Communication, by default, aborts the program when an error is encountered. I/O operations, by default, return an error code.

C

```c
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
```

F08

```fortran
MPI_File_set_errhandler(file, errhandler, ierror)
type(MPI_File), intent(in) :: file
type(MPI_Errhandler), intent(in) :: errhandler
integer, optional, intent(out) :: ierror
```

- The default error handler for files is `MPI_ERRORS_RETURN`
- Success is indicated by a return value of `MPI_SUCCESS`
- `MPI_ERRORS_ARE_FATAL` aborts the program
- Can be set for each file individually or for all files by using `MPI_File_set_errhandler` on a special file handle, `MPI_FILE_NULL`

QUIZ

What is the correct way to handle errors when using MPI I/O?

1. Setting the error handler of `MPI_File` objects to `MPI_ERRORS_ARE_FATAL`.
2. Checking the return value of every function against `MPI_SUCCESS`.
3. It depends...?

FILE VIEW [MPI-4.0, 14.3]

File View

A file view determines what part of the contents of a file is visible to a process. It is defined by a displacement (given in bytes) from the beginning of the file, an elementary datatype and a file type. The view into a file can be changed multiple times between opening and closing.

File Types and Elementary Types are Data Types

- Can be predefined or derived
- The usual constructors can be used to create derived file types and elementary types, e.g.
  - `MPI_Type_indexed`,
  - `MPI_Type_create_struct`,
  - `MPI_Type_create_subarray`
- Displacements in their typemap must be non-negative and monotonically nondecreasing
- Have to be committed before use

DEFAULT FILE VIEW [MPI-4.0, 14.3]

When newly opened, files are assigned a default view that is the same on all processes:

- Zero displacement
- File contains a contiguous sequence of bytes
- All processes have access to the entire file

File	0: byte	1: byte	2: byte	3: byte	...
Process 0	0: byte	1: byte	2: byte	3: byte	...
Process 1	0: byte	1: byte	2: byte	3: byte	...
...	0: byte	1: byte	2: byte	3: byte	...

ELEMENTARY TYPE [MPI-4.0, 14.3]

Elementary Type

An elementary type (or etype) is the unit of data contained in a file. Offsets are expressed in multiples of etypes, file pointers point to the beginning of etypes. Etypes can be basic or derived.

Changing the Elementary Type

E.g. `etype = MPI_INT`:

`File`	`0: int`	`1: int`	`2: int`	`3: int`	⋯
`Process 0`	`0: int`	`1: int`	`2: int`	`3: int`	⋯
`Process 1`	`0: int`	`1: int`	`2: int`	`3: int`	⋯
⋯	`0: int`	`1: int`	`2: int`	`3: int`	⋯

JÜLICH Forschungszentrum

FILE TYPE [MPI-4.0, 14.3]

File Type

A file type describes an access pattern. It can contain either instances of the etype or holes with an extent that is divisible by the extent of the etype.

Changing the File Type

E.g. $\mathit{Filetype}_0 = \{(\texttt{int}, 0), (\mathit{hole}, 4), (\mathit{hole}, 8)\}, \mathit{Filetype}_1 = \{(\mathit{hole}, 0), (\texttt{int}, 4), (\mathit{hole}, 8)\}, \ldots$:

File	0: int	1: int	2: int	3: int	...
Process 0	0: int			1: int	
Process 1		0: int			...
...			0: int		

CHANGING THE FILE VIEW [MPI-4.0, 14.3]

C

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

F08

```fortran
MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
type(MPI_File), intent(in) :: fh
integer(kind=MPI_OFFSET_KIND), intent(in) :: disp
type(MPI_Datatype), intent(in) :: etype, filetype
character(len=*), intent(in) :: datarep
type(MPI_Info), intent(in) :: info
integer, optional, intent(out) :: ierror
```

- Collective operation
- `datarep` and extent of `etype` must be identical across all process
- `disp`, `filetype` and `info` can be distinct
- File pointers are reset to zero
- May not overlap with nonblocking or split collective operations

DATA REPRESENTATION [MPI-4.0, 14.5]

- Determines the conversion of data in memory to data on disk
- Influences the interoperability of I/O between heterogeneous parts of a system or different systems

`"native"`

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

+ No loss of precision
+ No overhead
- On heterogeneous systems loss of transparent interoperability

DATA REPRESENTATION [MPI-4.0, 14.5]

`"internal"`

Data is stored in implementation-specific format

+ Can be used in a homogeneous and heterogeneous environment
+ Implementation will perform conversions if necessary
- Can incur overhead
- Not necessarily compatible between different implementations

`"external32"`

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

+ Can be read/written also by non-MPI programs
- Precision and I/O performance may be lost due to type conversions between `native` and `external32` representations
- Not available in all implementations

DATA ACCESS

Three orthogonal aspects

1. Synchronism
   1. Blocking
   2. Nonblocking
   3. Split collective
2. Coordination
   1. Noncollective
   2. Collective
3. Positioning
   1. Explicit offsets
   2. Individual file pointers
   3. Shared file pointers

POSIX `read()` and `write()`

These are blocking, noncollective operations with individual file pointers.

SYNCHRONISM

Blocking I/O

Blocking I/O routines do not return before the operation is completed.

Nonblocking I/O

- Nonblocking I/O routines do not wait for the operation to finish
- A separate completion routine is necessary [MPI-4.0, 3.7.3, 3.7.5]
- The associated buffers must not be used while the operation is in flight

Split Collective

- “Restricted” form of nonblocking collective
- Buffers must not be used while in flight
- Does not allow other collective accesses to the file while in flight
- `begin` and `end` must be used from the same thread

JÜLICH Forschungszentrum

COORDINATION

Noncollective

The completion depends only on the activity of the calling process.

Collective

- Completion may depend on activity of other processes
- Opens opportunities for optimization

POSITIONING [MPI-4.0, 14.4.1 – 14.4.4]

Explicit Offset

- No file pointer is used
- File position for access is given directly as function argument

Individual File Pointers

- Each process has its own file pointer
- After access, pointer is moved to first etype after the last one accessed

Shared File Pointers

- All processes share a single file pointer
- All processes must use the same file view
- Individual accesses appear as if serialized (with an unspecified order)
- Collective accesses are performed in order of ascending rank

Combine the prefix MPI_File_ with any of the following suffixes:

		coordination	
positioning	**synchronism**	noncollective	collective
explicit offsets	blocking	read_at, write_at	read_at_all, write_at_all
	nonblocking	iread_at, iwrite_at	iread_at_all, iwrite_at_all
	split collective	N/A	read_at_all_begin, read_at_all_end, write_at_all_begin, write_at_all_end
individual file pointers	blocking	read, write	read_all, write_all
	nonblocking	iread, iwrite	iread_all, iwrite_all
	split collective	N/A	read_all_begin, read_all_end, write_all_begin, write_all_end
shared file pointers	blocking	read_shared, write_shared	read_ordered, write_ordered
	nonblocking	iread_shared, iwrite_shared	N/A
	split collective	N/A	read_ordered_begin, read_ordered_end, write_ordered_begin, write_ordered_end

WRITING

blocking, noncollective, explicit offset [MPI-4.0, 14.4.2]

C

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

F08

```fortran
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
integer(kind=MPI_OFFSET_KIND), intent(in) :: offset
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
integer, optional, intent(out) :: ierror
```

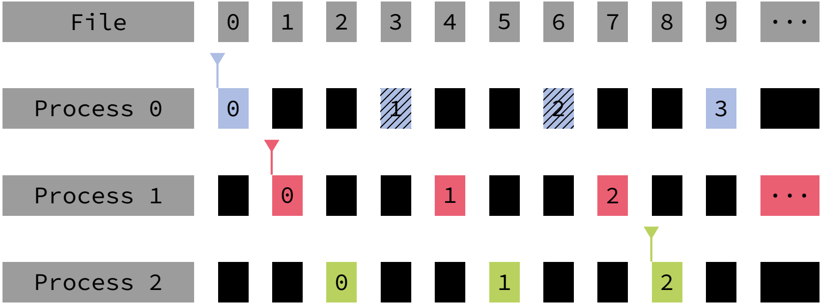
- Starting offset for access is explicitly given
- No file pointer is updated
- Writes `count` elements of `datatype` from memory starting at `buf`
- Typesig *datatype* = Typesig *etype* ... Typesig *etype*
- Writing past end of file increases the file size

JÜLICH Forschungszentrum

EXAMPLE

blocking, noncollective, explicit offset [MPI-4.0, 14.4.2]

Process 0 calls `MPI_File_write_at(offset = 1, count = 2)`:



WRITING

blocking, noncollective, individual [MPI-4.0, 14.4.3]

C

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

F08

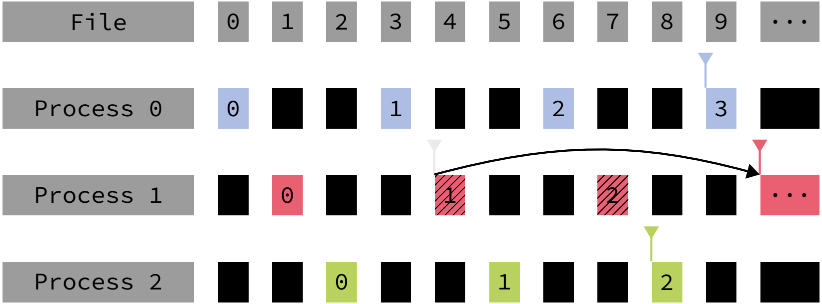
```fortran
MPI_File_write(fh, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- Starts writing at the current position of the individual file pointer
- Moves the individual file pointer by the count of etypes written

EXAMPLE

blocking, noncollective, individual [MPI-4.0, 14.4.3]

With its file pointer at element 1, process 1 calls `MPI_File_write(count = 2)`:



WRITING

nonblocking, noncollective, individual [MPI-4.0, 14.4.3]

C

```c
int MPI_File_iwrite(MPI_File fh, const void* buf, int count, MPI_Datatype
 ↪  datatype, MPI_Request* request)
```

F08

```fortran
MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Request), intent(out) :: request
integer, optional, intent(out) :: ierror
```

- Starts the same operation as `MPI_File_write` but does not wait for completion
- Returns a `request` object that is used to complete the operation

JÜLICH Forschungszentrum

WRITING

blocking, collective, individual [MPI-4.0, 14.4.3]

C

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

F08

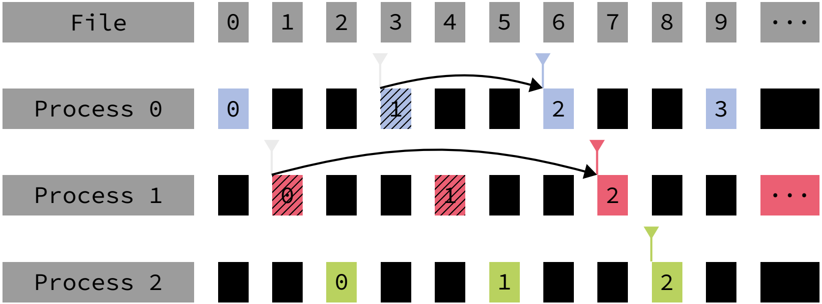
```fortran
MPI_File_write_all(fh, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- Same signature as `MPI_File_write`, but collective coordination
- Each process uses its individual file pointer
- MPI can use communication between processes to funnel I/O

EXAMPLE

blocking, collective, individual [MPI-4.0, 14.4.3]

- With its file pointer at element 1, process 0 calls `MPI_File_write_all(count = 1)`,
- With its file pointer at element 0, process 1 calls `MPI_File_write_all(count = 2)`,
- With its file pointer at element 2, process 2 calls `MPI_File_write_all(count = 0)`:



WRITING

split-collective, individual [MPI-4.0, 14.4.5]

C

```c
int MPI_File_write_all_begin(MPI_File fh, const void* buf, int count,
 ↪  MPI_Datatype datatype)
```

F08

```fortran
MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
integer, optional, intent(out) :: ierror
```

- Same operation as `MPI_File_write_all`, but split-collective
- `status` is returned by the corresponding end routine

WRITING

split-collective, individual [MPI-4.0, 14.4.5]

C

```c
int MPI_File_write_all_end(MPI_File fh, const void* buf, MPI_Status*
↪  status)
```

F08

```fortran
MPI_File_write_all_end(fh, buf, status, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..), intent(in) :: buf
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- `buf` argument must match corresponding `begin` routine

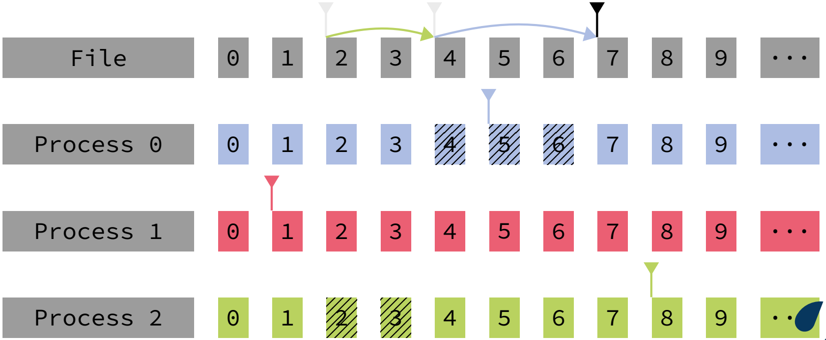
EXAMPLE

blocking, noncollective, shared [MPI-4.0, 14.4.4]

All process must share the same file view for shared file pointer data accesses!

With the shared pointer at element 2,

- process 0 calls `MPI_File_write_shared(count = 3)`,
- process 2 calls `MPI_File_write_shared(count = 2)`:



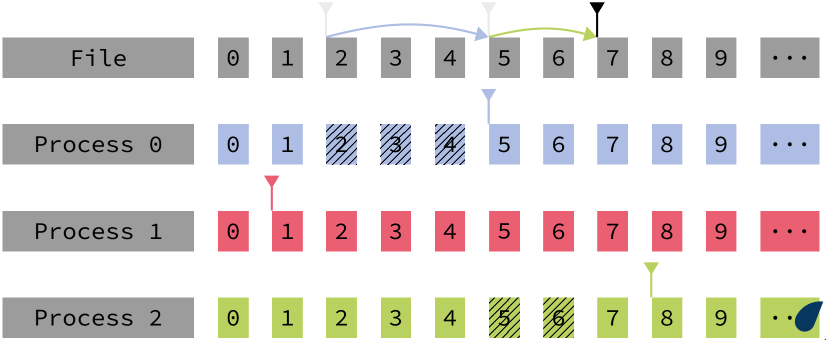
EXAMPLE

blocking, noncollective, shared [MPI-4.0, 14.4.4]

All process must share the same file view for shared file pointer data accesses!

With the shared pointer at element 2,

- process 0 calls `MPI_File_write_shared(count = 3)`,
- process 2 calls `MPI_File_write_shared(count = 2)`:

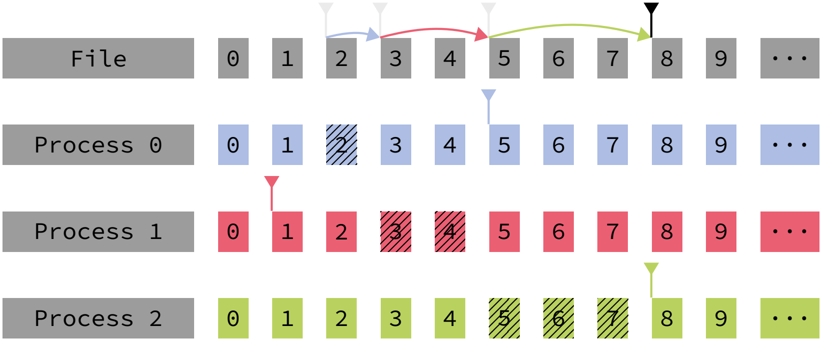


EXAMPLE

blocking, collective, shared [MPI-4.0, 14.4.4]

With the shared pointer at element 2,

- process 0 calls `MPI_File_write_ordered(count = 1)`,
- process 1 calls `MPI_File_write_ordered(count = 2)`,
- process 2 calls `MPI_File_write_ordered(count = 3)`:



READING

blocking, noncollective, individual [MPI-4.0, 14.4.3]

C

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

F08

```fortran
MPI_File_read(fh, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- Starts reading at the current position of the individual file pointer
- Reads up to `count` elements of `datatype` into the memory starting at `buf`
- `status` indicates how many elements have been read
- If `status` indicates less than `count` elements read, the end of file has been reached

FILE POINTER POSITION [MPI-4.0, 14.4.3]

C

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

F08

```fortran
MPI_File_get_position(fh, offset, ierror)
type(MPI_File), intent(in) :: fh
integer(kind=MPI_OFFSET_KIND), intent(out) :: offset
integer, optional, intent(out) :: ierror
```

- Returns the current position of the individual file pointer in units of etype
- Value can be used for e.g.
  - return to this position (via seek)
  - calculate a displacement
- `MPI_File_get_position_shared` queries the position of the shared file pointer

SEEKING TO A FILE POSITION [MPI-4.0, 14.4.3]

C

```c
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
```

F08

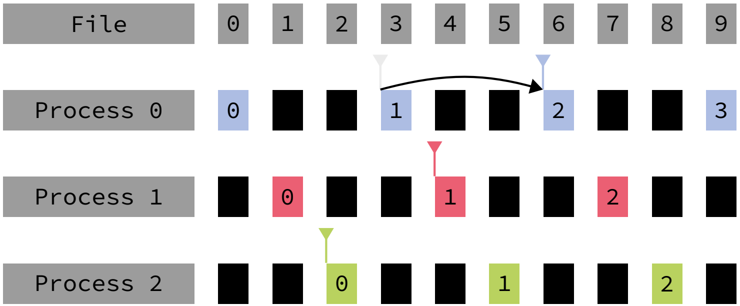
```fortran
MPI_File_seek(fh, offset, whence, ierror)
type(MPI_File), intent(in) :: fh
integer(kind=MPI_OFFSET_KIND), intent(in) :: offset
integer, intent(in) :: whence
integer, optional, intent(out) :: ierror
```

- whence controls how the file pointer is moved:

  `MPI_SEEK_SET` sets the file pointer to `offset`
  `MPI_SEEK_CUR` `offset` is relative to the current value of the pointer
  `MPI_SEEK_END` `offset` is relative to the end of the file
- `offset` can be negative but the resulting position may not lie before the beginning of the file
- `MPI_File_seek_shared` manipulates the shared file pointer

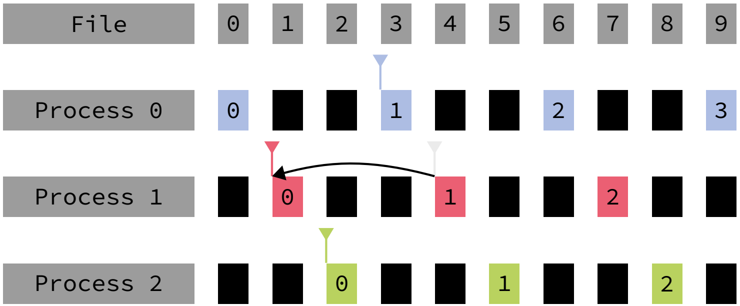
EXAMPLE

Process 0 calls `MPI_File_seek(offset = 2, whence = MPI_SEEK_SET)`:



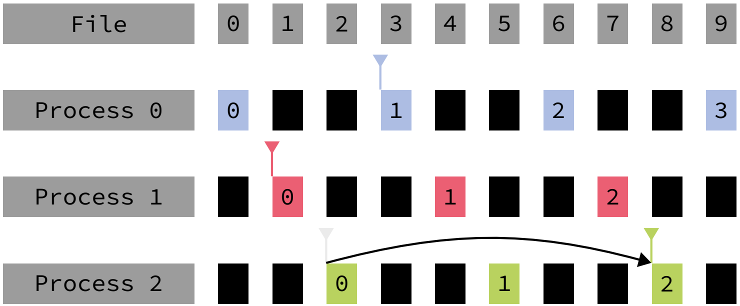
EXAMPLE

Process 1 calls `MPI_File_seek(offset = -1, whence = MPI_SEEK_CUR)`:



EXAMPLE

Process 2 calls `MPI_File_seek(offset = -1, whence = MPI_SEEK_END)`:



CONVERTING OFFSETS [MPI-4.0, 14.4.3]

C

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

F08

```fortran
MPI_File_get_byte_offset(fh, offset, disp, ierror)
type(MPI_File), intent(in) :: fh
integer(kind=MPI_OFFSET_KIND), intent(in) :: offset
integer(kind=MPI_OFFSET_KIND), intent(out) :: disp
integer, optional, intent(out) :: ierror
```

- Converts a view relative offset (in units of etype) into a displacement in bytes from the beginning of the file

CONSISTENCY [MPI-4.0, 14.6.1]

Sequential Consistency

If a set of operations is sequentially consistent, they behave as if executed in some serial order. The exact order is unspecified.

- To guarantee sequential consistency, certain requirements must be met
- Requirements depend on access path and file atomicity

Result of operations that are not sequentially consistent is implementation dependent.

ATOMIC MODE [MPI-4.0, 14.6.1]

Requirements for sequential consistency

Same file handle: always sequentially consistent
File handles from same open: always sequentially consistent
File handles from different open: not influenced by atomicity, see nonatomic mode

- Atomic mode is not the default setting
- Can lead to overhead, because MPI library has to uphold guarantees in general case

C

```c
int MPI_File_set_atomicity(MPI_File fh, int flag)
```

F08

```fortran
MPI_File_set_atomicity(fh, flag, ierror)
type(MPI_File), intent(in) :: fh
logical, intent(in) :: flag
integer, optional, intent(out) :: ierror
```

NONATOMIC MODE [MPI-4.0, 14.6.1]

Requirements for sequential consistency

Same file handle: operations must be either nonconcurrent, nonconflicting, or both
File handles from same open: nonconflicting accesses are sequentially consistent, conflicting accesses have to be protected using `MPI_File_sync`
File handles from different open: all accesses must be protected using `MPI_File_sync`

Conflicting Accesses

Two accesses are conflicting if they touch overlapping parts of a file and at least one is writing.

C

```c
int MPI_File_sync(MPI_File fh)
```

F08

```fortran
MPI_File_sync(fh, ierror)
type(MPI_File), intent(in) :: fh
integer, optional, intent(out) :: ierror
```

NONATOMIC MODE [MPI-4.0, 14.6.1]

The Sync-Barrier-Sync construct

```c
// writing access sequence through
  ↪  one file handle
MPI_File_sync(fh0);
MPI_Barrier(MPI_COMM_WORLD);
MPI_File_sync(fh0);
// ...
```

```c
// ...
MPI_File_sync(fh1);
MPI_Barrier(MPI_COMM_WORLD);
MPI_File_sync(fh1);
// access sequence to the same
  ↪  file through a different file
  ↪  handle
```

- `MPI_File_sync` is used to delimit sequences of accesses through different file handles
- Sequences that contain a write access may not be concurrent with any other access sequence

LARGE COUNT EXAMPLE

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

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

LARGE COUNT EXAMPLE

F08

```
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
integer(KIND=MPI_OFFSET_KIND), intent(in) :: offset
type(*), dimension(..) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

LARGE COUNT EXAMPLE

F08

```
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
integer(KIND=MPI_OFFSET_KIND), intent(in) :: offset
type(*), dimension(..) :: buf
integer(KIND=MPI_COUNT_KIND), intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

EXERCISES

Exercise 1 – Data Access

1.1 Writing Data

In the file `rank_io.{c|cxx|f90|py}` write a function `write_rank` that takes a communicator as its only argument and does the following:

- Each process writes its own rank in the communicator to a common file `rank.dat` using `"native"` data representation.
- The ranks should be in order in the file: $0 \dots n-1$.

Use: `MPI_File_open`, `MPI_File_set_errhandler`, `MPI_File_set_view`, `MPI_File_write_ordered`, `MPI_File_close`

EXERCISES

Exercise 1 – Data Access

1.2 Reading Data

In the file `rank_io.{c|cxx|f90|py}` write a function `read_rank` that takes a communicator as its only argument and does the following:

- The processes read the integers in the file in reverse order, i.e. process 0 reads the last entry, process 1 reads the one before, etc.
- Each process returns the rank number it has read from the function.

Careful: This function might be run on a communicator with a different number of processes. If there are more processes than entries in the file, processes with ranks larger than or equal to the number of file entries should return `MPI_PROC_NULL`.

Use: `MPI_File_seek`, `MPI_File_get_position`, `MPI_File_read`

EXERCISES

Exercise 1 – Data Access

1.3 Phone Book

The file phonebook.dat contains several records of the following form:

C

```c
struct dbentry {
  int key;
  int room_number;
  int phone_number;
  char name[200];
}
```

F08

```fortran
type :: dbentry
  integer :: key
  integer :: room_number
  integer :: phone_number
  character(len=200) :: name
end type
```

In the file phonebook.{c|cxx|f90|py} write a function look_up_by_room_number that uses MPI I/O to find an entry by room number. Assume the file was written using "native" data representation. Use MPI_COMM_SELF to open the file. Return a bool or logical to indicate whether an entry has been found and fill an entry via pointer/intent out argument.

JÜLICH Forschungszentrum