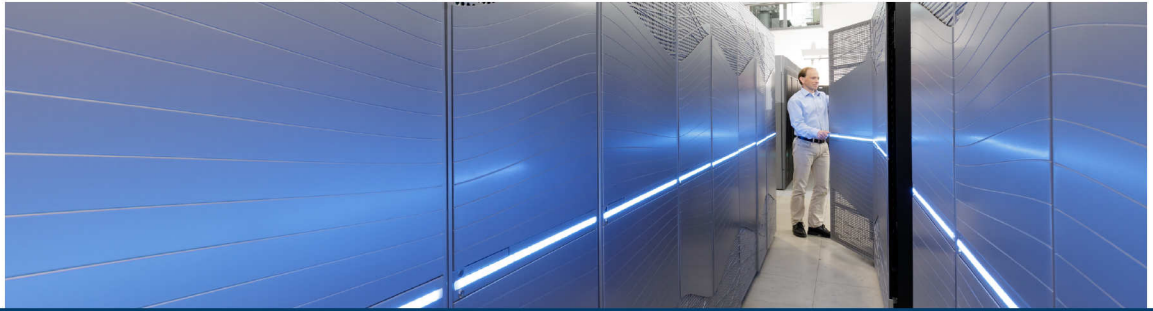
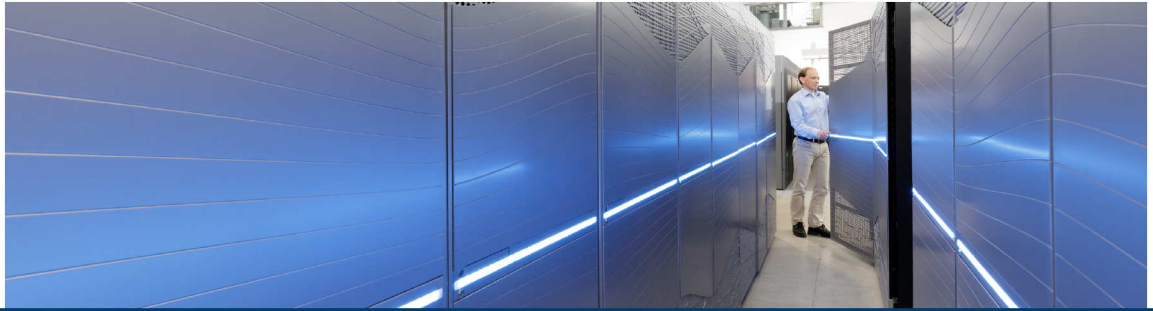


PARALLEL I/O WITH MPI

29 January 2020 | Benedikt Steinbusch | Jülich Supercomputing Centre



Part I: Introduction

MPI I/O FACT SHEET

Data model a sequence of typed data items

Self describing no

Full control of file content maybe

Use cases

- Implementing higher level formats
- Compatibility to existing formats

JÜLICH Forschungszentrum

FEATURES OF MPI I/O

- Standardized I/O API since 1997
- Available in many MPI libraries
- Language bindings for C and Fortran
- Re-uses MPI's concepts for the description of data
- Allows portable and efficient implementation of parallel I/O operations due to support for
  - multiple data representations
  - asynchronous I/O
  - non-contiguous file access patterns
  - collective file access
  - MPI Info Objects

PREREQUISITES

You should be familiar with MPI, especially with

Processes and Ranks	An MPI program is executed by multiple processes in parallel. Processes are identified by ranks (0, 1, …)
Communicator	Combines a group of processes and a context for communication.
Blocking and Nonblocking	Provide different guarantees to the user / different liberties to the MPI library.
P2P and Collective Communication	Communication among pairs or groups of processes
Derived Datatypes	Descriptions of data layouts
MPI Info Objects	Key-value maps that can be used to provide hints

LITERATURE & ACKNOWLEDGEMENTS

Literature

- Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 3.1. June 4, 2015. URL: https://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI. Portable Parallel Programming with the Message-Passing Interface*. 3rd ed. The MIT Press, Nov. 2014. 336 pp. ISBN: 9780262527392
- William Gropp et al. *Using Advanced MPI. Modern Features of the Message-Passing Interface*. 1st ed. Nov. 2014. 392 pp. ISBN: 9780262527637
- https://www.mpi-forum.org

Acknowledgements

- Rolf Rabenseifner for his comprehensive course on MPI and OpenMP
- Marc-André Hermanns, Florian Janetzko and Alexander Trautmann for their course material on MPI and OpenMP

LANGUAGE BINDINGS [MPI-3.1, 17, A]

C Language Bindings

C

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

Fortran Language Bindings

Consistent with F08 standard; good type-checking; highly recommended

F08

```
use mpi_f08
```

Not consistent with standard; so-so type-checking; not recommended

F90

```
use mpi
```

Not consistent with standard; no type-checking; strongly discouraged

F77

```
include 'mpif.h'
```

FORTRAN HINTS [MPI-3.1, 17.1.2 – 17.1.4]

This course uses the Fortran 2008 MPI interface (**`use`** `mpi_f08`) which is not available in all MPI implementations. The Fortran 90 bindings differ from the Fortran 2008 bindings in the following points:

- All derived **`type`** arguments are instead **`integer`** (some are arrays of **`integer`** or have a non-default `kind`)
- Argument **`intent`** is not mandated by the Fortran 90 bindings
- The `ierror` argument is mandatory instead of **`optional`**
- Further details can be found in [MPI-3.1, 17.1]

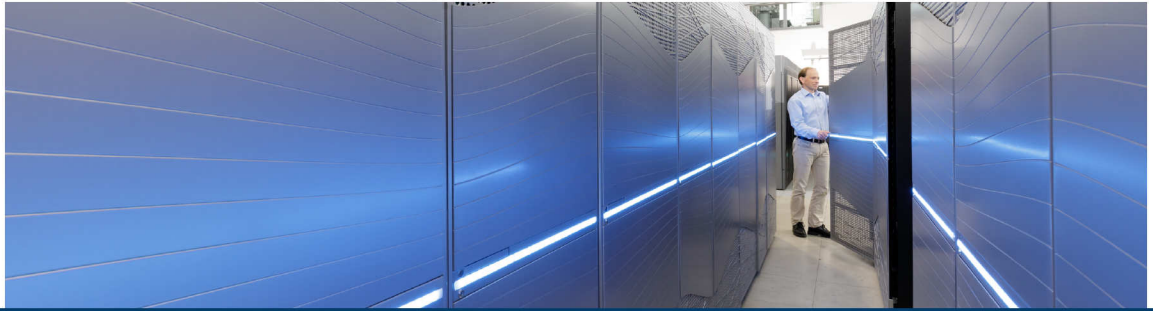
MPI4PY HINTS

All exercises can be solved using Python with the `mpi4py` package. The slides do not show Python syntax, so here is a translation guide from the standard bindings to `mpi4py`.

- Everything lives in the `MPI` module (**`from mpi4py import`** `MPI`).
- Constants translate to attributes of that module: `MPI_COMM_WORLD` is `MPI.COMM_WORLD`.
- Central types translate to Python classes: `MPI_Comm` is `MPI.Comm`.
- Functions related to point-to-point and collective communication translate to methods on `MPI.Comm`: `MPI_Send` becomes `MPI.Comm.Send`.
- Functions related to I/O translate to methods on `MPI.File`: `MPI_File_write` becomes `MPI.File.Write`.
- Communication functions come in two flavors:
  - high level, uses `pickle` to (de)serialize python objects, method names start with lower case letters, e.g. `MPI.Comm.send`,
  - low level, uses MPI Datatypes and Python buffers, method names start with upper case letters, e.g. `MPI.Comm.Scatter`.

See also `https://mpi4py.readthedocs.io` and the built-in Python `help()`.

JÜLICH Forschungszentrum



Part II: File Operations

FILE, FILE POINTER & HANDLE [MPI-3.1, 13.1]

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-3.1, 13.2.1]

C

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

F08

```
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-3.1, 13.2.1]

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).

JÜLICH Forschungszentrum

CLOSING A FILE [MPI-3.1, 13.2.2]

C

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

F08

```
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-3.1, 13.2.3]

C

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

F08

```
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`
- File deletion is a local operation and should be performed by a single process
- 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-3.1, 13.2.4]
`MPI_File_preallocate`	Preallocate disk space [MPI-3.1, 13.2.5]
`MPI_File_set_info`	Supply additional information [MPI-3.1, 13.2.8]

Inspecting File Parameters

`MPI_File_get_size`	Size of a file [MPI-3.1, 13.2.6]
`MPI_File_get_amode`	Acess mode [MPI-3.1, 13.2.7]
`MPI_File_get_group`	Group of processes that opened the file [MPI-3.1, 13.2.7]
`MPI_File_get_info`	Additional information associated with the file [MPI-3.1, 13.2.8]

I/O ERROR HANDLING [MPI-3.1, 8.3, 13.7]

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

```f08
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`

FILE VIEW [MPI-3.1, 13.3]

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.

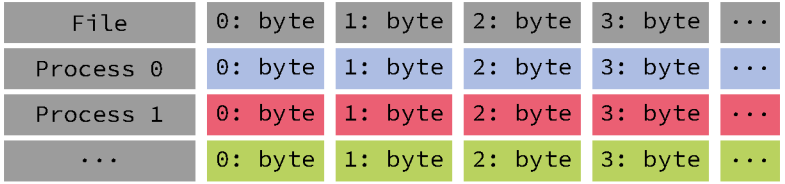
File Types and Elementary Types are Data Types

- 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-3.1, 13.3]

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



ELEMENTARY TYPE [MPI-3.1, 13.3]

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	···

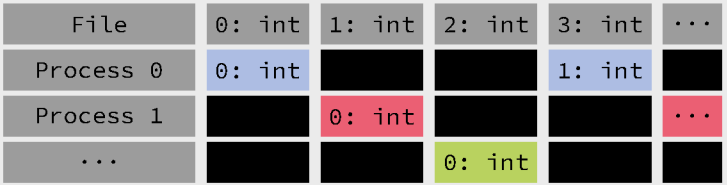
FILE TYPE [MPI-3.1, 13.3]

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. $Filetype_0 = \{(\texttt{int}, 0), (hole, 4), (hole, 8)\}$, $Filetype_1 = \{(hole, 0), (\texttt{int}, 4), (hole, 8)\}$, ...:



JÜLICH Forschungszentrum

CHANGING THE FILE VIEW [MPI-3.1, 13.3]

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

```
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 match
- `disp`, `filetype` and `info` can be distinct
- File pointers are reset to zero
- May not overlap with nonblocking or split collective operations

JÜLICH Forschungszentrum

DATA REPRESENTATION [MPI-3.1, 13.5]

- 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-3.1, 13.5]

`"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.

JÜLICH Forschungszentrum

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-3.1, 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

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

JÜLICH Forschungszentrum

POSITIONING [MPI-3.1, 13.4.1 – 13.4.4]

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-3.1, 13.4.2]

C

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

F08

```
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
```

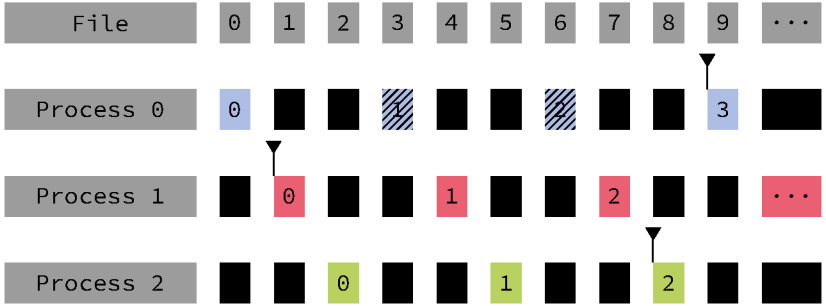
- Starting offset for access is explicitly given
- No file pointer is updated
- Writes `count` elements of `datatype` from memory starting at `buf`
- Typesig *datatype* = Typesig *etype* ... Typesig *etype*
- Writing past end of file increases the file size

JÜLICH Forschungszentrum

EXAMPLE

blocking, noncollective, explicit offset [MPI-3.1, 13.4.2]

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



JÜLICH Forschungszentrum

WRITING

blocking, noncollective, individual [MPI-3.1, 13.4.3]

C

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

F08

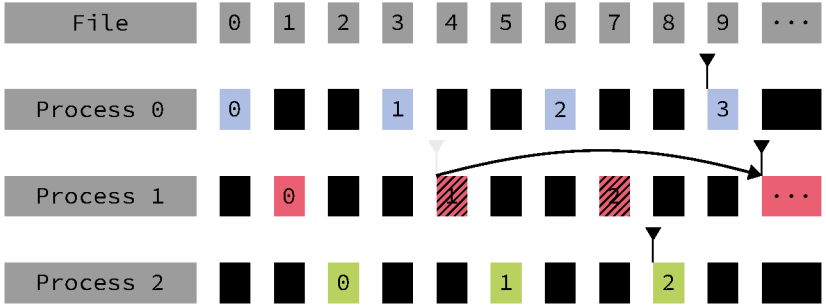
```
MPI_File_write(fh, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- 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-3.1, 13.4.3]

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



JÜLICH Forschungszentrum

WRITING

nonblocking, noncollective, individual [MPI-3.1, 13.4.3]

C

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

F08

```
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-3.1, 13.4.3]

C

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

F08

```
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
```

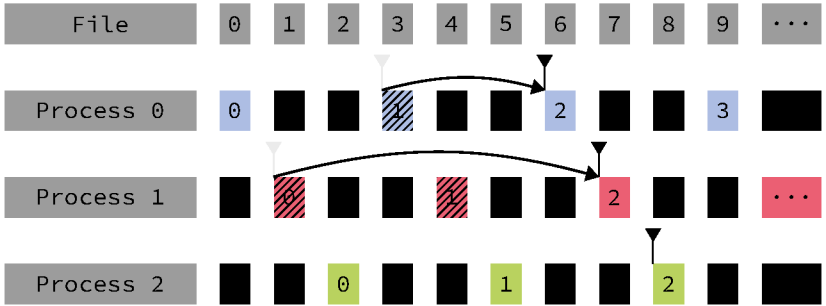
- Same signature as `MPI_File_write`, but collective coordination
- Each process uses its individual file pointer
- MPI can use communication between processes to funnel I/O

JÜLICH Forschungszentrum

EXAMPLE

blocking, collective, individual [MPI-3.1, 13.4.3]

- 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)`:



JÜLICH Forschungszentrum

WRITING

split-collective, individual [MPI-3.1, 13.4.5]

C

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

F08

```
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

JÜLICH Forschungszentrum

WRITING

split-collective, individual [MPI-3.1, 13.4.5]

C

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

F08

```
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

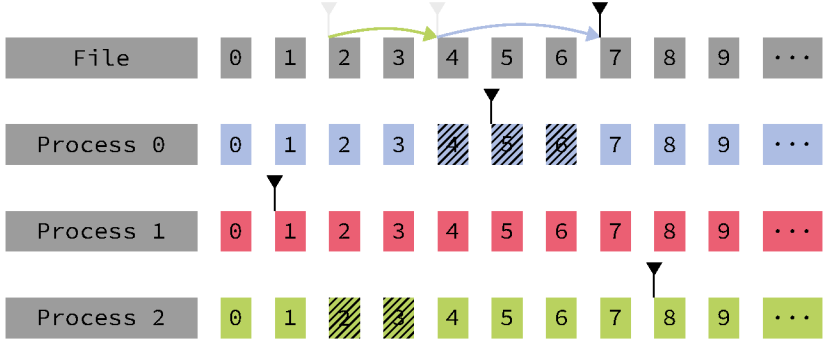
JÜLICH Forschungszentrum

EXAMPLE

blocking, noncollective, shared [MPI-3.1, 13.4.4]

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)`:



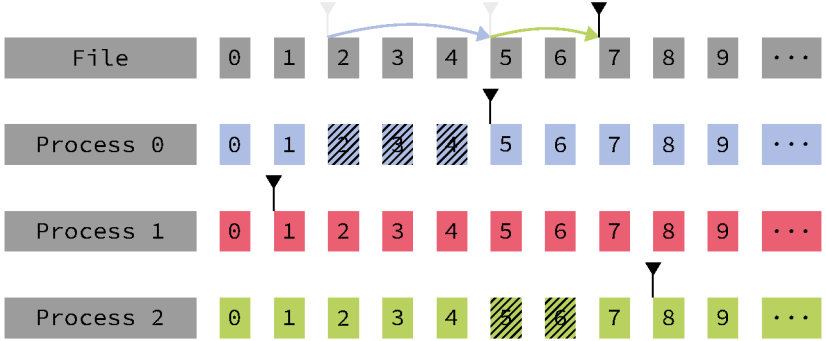
JÜLICH Forschungszentrum

EXAMPLE

blocking, noncollective, shared [MPI-3.1, 13.4.4]

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)`:



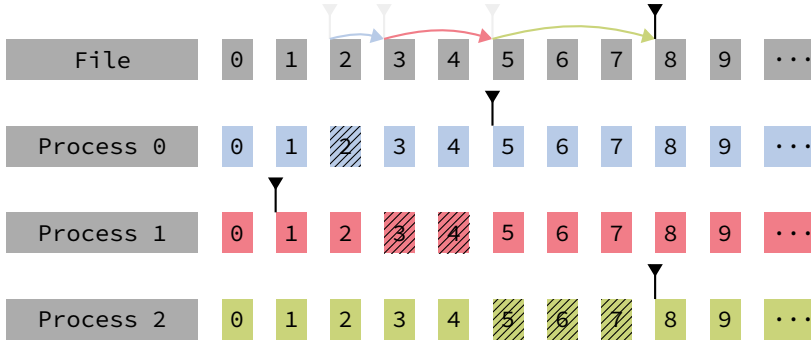
JÜLICH Forschungszentrum

EXAMPLE

blocking, collective, shared [MPI-3.1, 13.4.4]

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-3.1, 13.4.3]

C

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

F08

```
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

JÜLICH Forschungszentrum

FILE POINTER POSITION [MPI-3.1, 13.4.3]

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

F08
```
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-3.1, 13.4.3]

C

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

F08

```f08
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

CONVERTING OFFSETS [MPI-3.1, 13.4.3]

C

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

F08

```
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-3.1, 13.6.1]

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-3.1, 13.6.1]

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

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

F08

```
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-3.1, 13.6.1]

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

```f90
MPI_File_sync(fh, ierror)
type(MPI_File), intent(in) :: fh
integer, optional, intent(out) :: ierror
```

NONATOMIC MODE [MPI-3.1, 13.6.1]

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

JÜLICH Forschungszentrum

LOGIN & PROGRAMMING ENVIRONMENT

JURECA Login

1. In a terminal on your PC, enter:

```
$ ssh name1@jureca.fz-juelich.de
```

2. Load modules and activate the training project:

```
$ module load intel-para
$ jutil env activate -p training2000
  ↪  -A training2000
```

Course Material

Copy the course material by running:

```
$ $PROJECT/mpi-io/copy.sh
```

MPI Infrastructure

C

```
$ mpicc
```

Fortran

```
$ mpif90
```

C++

```
$ mpicxx
```

Process startup

```
$ srun -n <numprocs> <program>
```


JÜLICH Forschungszentrum

RUNNING PARALLEL PROGRAMS ON JURECA

Interactive Mode

1. Start an interactive session

```
$ salloc --reservation=pario
 ↪  --nodes=1 --time=08:00:00
```

2. Wait for the prompt…
3. Start applications with n processes

```
$ srun --ntasks=<n>
 ↪  <application>
```

Batch Mode

To start an application with n processes, submit the following job script with

```
sbatch --reservation=pario <script>
```

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=<n>
#SBATCH --ntasks-per-node=<n>
#SBATCH --time=00:05:00
module load intel-para
srun <application>
```


JÜLICH Forschungszentrum

EXERCISE STRATEGIES

Solving

- Do not have to solve all exercises, one per section would be good
- Exercise description tells you what MPI functions/OpenMP directives to use
- Work in pairs on harder exercises
- If you get stuck
  - ask us
  - peek at solution
- `Makefile` is included

Solutions

`exercises/mpi-io/{C|C++|Fotran|Python}/`:

- Most of the algorithm is there, you add MPI

`hard` Almost empty files you add algorithm and MPI

`solutions` Fully solved exercises, if you are completely stuck or for comparison

EXERCISES

Exercise 1 – Data Access

1.1 Writing and Reading 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`.
- The ranks should be in order in the file: $0 \ldots n-1$.

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

JÜLICH Forschungszentrum

EXERCISES

Exercise 1 – Data Access

1.2 Accessing Parts of Files

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 program 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`

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];
}
```

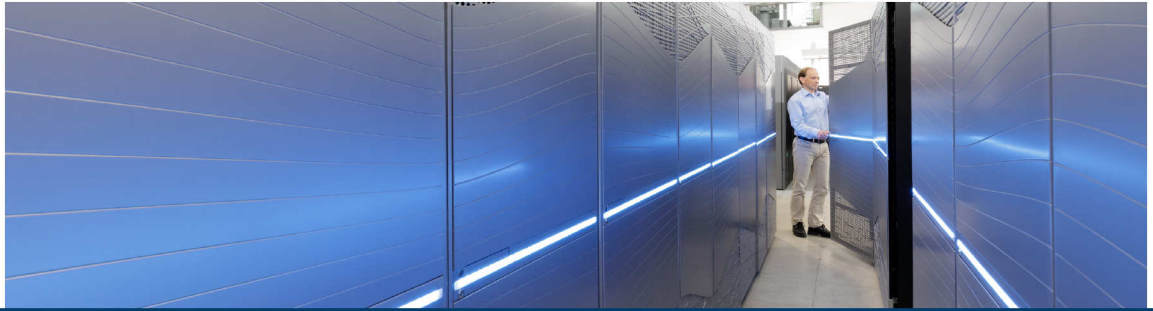
F08

```f90
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. Return a `bool` or `logical` to indicate whether an entry has been found and fill an entry via pointer/intent out argument.

Use: `MPI_File_read`

JÜLICH Forschungszentrum



Part III: The Info Object

THE INFO OBJECT [MPI-3.1, 9]

A Gentle Reminder

Used to pass hints for optimization to MPI

- Consists of (`key`, `value`) pairs, where both `key` and `value` are strings
- Each `key` must appear only once
- `MPI_INFO_NULL` can be used in place of an actual info object
- Keys must not be larger than `MPI_MAX_INFO_KEY`
- Values must not be larger than `MPI_MAX_INFO_VAL`

Info Object API

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

INFO OBJECTS FOR I/O [MPI-3.1, 13.2, 13.2]

Info objects can be associated with files that MPI I/O operates on using several mechanisms:

- When opening a file: the info object is passed to the `MPI_File_open` routine
- While the file is open:
  - When setting a file view using `MPI_File_set_view`
  - Explicitly using `MPI_File_set_info`
- When deleting a file using `MPI_File_delete`
- Globally using a ROMIO hint file

Some info items can only be reasonably used e.g. when opening a file and will be ignored when later used with `MPI_File_set_info`.

FILE INFO OBJECT ACCESSORS [MPI-3.1, 13.2.8]

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

F08
```
MPI_File_set_info(fh, info, ierror)
type(MPI_File), intent(in) :: fh
type(MPI_Info), intent(in) :: info
integer, optional, intent(out) :: ierror
```

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

F08
```
MPI_File_get_info(fh, info, ierror)
type(MPI_File), intent(in) :: fh
type(MPI_Info), intent(out) :: info
integer, optional, intent(out) :: ierror
```

PASSING HINTS USING A FILE

Specify Hint File via Environment Variable

```
$ export ROMIO_HINTS=<absolute path>/hintfile
```

- Environment variable must be exported to the compute nodes. Use appropriate mechanisms provided by process starters like `mpiexec` or `runjob`.
- Hints are used for all MPI I/O operations in the application through
  - direct use of MPI I/O routines
  - use of libraries that use MPI I/O

Example Hint File Content

```
collective buffering true
cb_buffer_size       33554432
cb_block_size        4194304
```

RESERVED KEYS FOR I/O [MPI-3.1, 13.2.8]

- An MPI implementation is not required to support these hints
- If a hint is supported by an implementation, it must behave as described by the standard
- Additional keys may be supported

JÜLICH Forschungszentrum

RESERVED KEYS FOR I/O [MPI-3.1, 13.2.8]

`filename: string` *implementation dependent*

Can be used to inspect the file name of an open file.

`file_perm: string` *same, implementation dependent*

Specifies the file permissions to set on file creation.

`access_style: [string]` *comma separated*

Specifies the manner in which the file will be accessed until it is closed or this info key is changed. Valid list elements are:

- `read_once`
- `write_once`
- `read_mostly`
- `write_mostly`
- `sequential`
- `reverse_sequential`
- `random`

JÜLICH Forschungszentrum

RESERVED KEYS FOR I/O [MPI-3.1, 13.2.8]

`nb_proc: integer`	*same*
Specifies how many parallel processes usually run the application that accesses this file.	

`num_io_nodes: integer`	*same*
Specifies the number of I/O devices in the system.	

`io_node_list: [string]`	*comma separated, same, implementation dependent*
Specifies a list of I/O devices that should be used to store the file.	

RESERVED KEYS FOR I/O [MPI-3.1, 13.2.8]

`chunked: [integer]`	*comma separated, same*
Specifies that the file consists of a multidimensional array that is often accessed by subarrays. List entries are array dimensions in order of decreasing significance.	

`chunked_item: [integer]`	*comma separated, same*
Specifies the size of one array entry in bytes.	

`chunked_size: [integer]`	*comma separated, same*
Specifies the dimensions of the subarrays.	

RESERVED KEYS FOR I/O [MPI-3.1, 13.2.8]

`collective_buffering: boolean`	*same*
Specifies whether the application may benefit from collective buffering.	

`cb_nodes: integer`	*same*
Specifies the number of target nodes to be used for collective buffering.	

`cb_block_size: integer`	*same*
Specifies the block size to be used for collective buffering. Data access happens in chunks of this size.	

`cb_buffer_size: integer`	*same*
Specifies the size of the buffer space that can be used on each target node.	

RESERVED KEYS FOR I/O [MPI-3.1, 13.2.8]

`striping_factor: integer`	*same*
Specifies the number of I/O devices that the file should be striped across. Relevant only on file creation.	

`striping_unit: integer`	*same*
Specifies the striping unit – the amount of consecutive data assigned to one I/O device – to be used for this file. Only relevant on file creation.	

GOOD CHOICES FOR GPFS

`romio_ds_write: string`	*default: `automatic`*
Specifies whether to use data sieving for write access. Good choice: `enable`	

`romio_ds_read: string`	*default: `automatic`*
Specifies whether to use data sieving for read access. Good choice: `automatic`	

`cb_buffer_size: integer`	*default: 16777216*
Specifies the size of the buffer space that can be used on each target node. Good choice: 33554432	

- Default keys already seem to be a good setting
- Collective buffering is switched on by default (`collective_buffering` is ignored, but `romio_cb_read`/`romio_cb_write` are available)
- Data sieving is only important for writing with shared file pointers and for small amounts of data.
- `cb_nodes` is set automatically and cannot be changed by the user

COLOPHON

This document was typeset using

- LuaLaTeX and a host of macro packages,
- Adobe Source Sans Pro for body text and headings,
- `Adobe Source Code Pro` for listings,
- TeX Gyre Pagella Math for mathematical formulae,
- icons from Font Awesome .