
Parallel programming / computation

Sultan ALPAR
s.alpar@iitu.edu.kz

IITU

Lecture 10
Parallel File I/O

Outline

- **Block 1**
 - Introduction [323]
 - **Definitions** [328]
 - Open / Close [330]
 - WRITE / **Explicit Offsets** [335]
 - Exercise 1 [336]
- **Block 2**
 - **File Views** [338]
 - **Subarray & Darray** [342]
 - I/O Routines Overview [350]
 - READ / Explicit Offsets [352]
 - **Individual File Pointer** [353]
 - Exercise 2 [355]
- **Block 3**
 - **Shared File Pointer** [358]
 - **Collective** [360]
 - Non-Blocking / Split Collective [364/365]
 - Other Routines [368]
 - Error Handling [369]
 - Implementation Restrictions [370]
 - **Summary** [371]
 - Exercise 3 [372]
 - Exercise 4 [373]

Motivation, I.

If you have **thousands of MPI processes**

& each has to write/read data to/from a file

- Opening thousands of file in a parallel file system can be **extremely slow**
- Writing 1000s of files → reading in by a different number of processes → **hard**

Motivation, I.

If you have **thousands of MPI processes**

& each has to write/read data to/from a file

- Opening thousands of file in a parallel file system can be **extremely slow**
- Writing 1000s of files → reading in by a different number of processes → **hard**

MPI parallel file I/O offers

- A method to write/read data by all processes to/from one common (large) file
- Supports disk striping for this one file
- Is also the internal basis for parallel netCDF and HDF5

Motivation, I.

If you have **thousands of MPI processes**

& each has to write/read data to/from a file

- Opening thousands of file in a parallel file system can be **extremely slow**
- Writing 1000s of files → reading in by a different number of processes → **hard**

MPI parallel file I/O offers

- A method to write/read data by all processes to/from one common (large) file
- Supports disk striping for this one file
- Is also the internal basis for parallel netCDF and HDF5

Historically

- This parallel I/O interface was included into MPI because it totally fits to the principles of message passing → next motivation slides (skipped)

Motivation, I.

If you have **thousands of MPI processes**

& each has to write/read data to/from a file

- Opening thousands of file in a parallel file system can be **extremely slow**
- Writing 1000s of files → reading in by a different number of processes → **hard**

MPI parallel file I/O offers

- A method to write/read data by all processes to/from one common (large) file
- Supports disk striping for this one file
- Is also the internal basis for parallel netCDF and HDF5

Historically

- This parallel I/O interface was included into MPI because it totally fits to the principles of message passing → next motivation slides (skipped)

Other options

- Small #processes → just send the data to process 0 for non-parallel I/O
- Use several dedicated MPI processes for asynchronous I/O → “ICON” in course chapter 8-(1)
 - fetching the data with MPI_Get, and
 - writing it to several files or one file with MPI I/O

Motivation, II.

- Many parallel applications need
 - coordinated parallel access to a file by a group of processes
 - simultaneous access
 - all processes may read/write many (small) non-contiguous pieces of the file,
i.e. the data may be distributed amongst the processes according to a partitioning scheme
 - all processes may read the same data
- Efficient collective I/O based on
 - fast physical I/O by several processors, e.g. striped
 - distributing (small) pieces by fast message passing



Motivation, III.

- Analogy: writing / reading a file is like sending/receiving a message
- Handling parallel I/O needs
 - handling groups of processes → MPI topologies and groups
 - collective operations → file handle defined like communicators
 - nonblocking operations to overlap computation & I/O → MPI_I..., MPI_Wait, ... & new **split** collective interface
 - non-contiguous access → MPI derived datatypes

MPI-I/O Features

- Provides a high-level interface to support
 - data file partitioning among processes
 - transfer global data between memory and files (collective I/O)
 - asynchronous transfers
 - strided access
- MPI derived datatypes used to specify common data access patterns for maximum flexibility and expressiveness

MPI-I/O, Principles

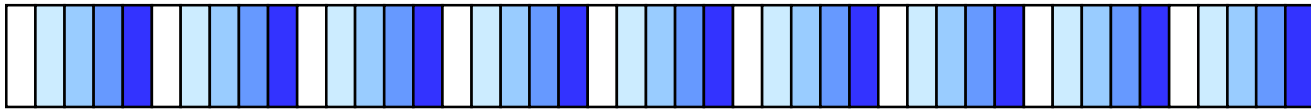
- MPI file contains elements of a single MPI datatype (etype)
- partitioning the file among processes with an access template (filetype)
- all file accesses transfer to/from a contiguous or non-contiguous user buffer (MPI datatype)
- nonblocking / blocking and collective / individual read / write routines
- individual and shared file pointers, explicit offsets
- binary I/O
- automatic data conversion in heterog. systems
- file interoperability with external representation

Logical view / Physical view

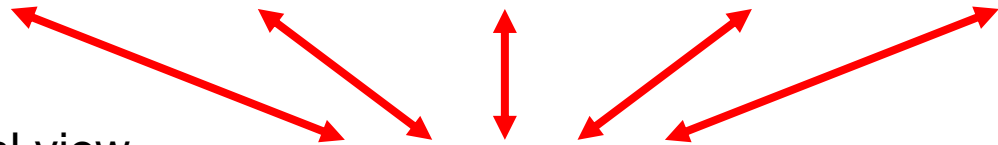
mpi processes of a communicator



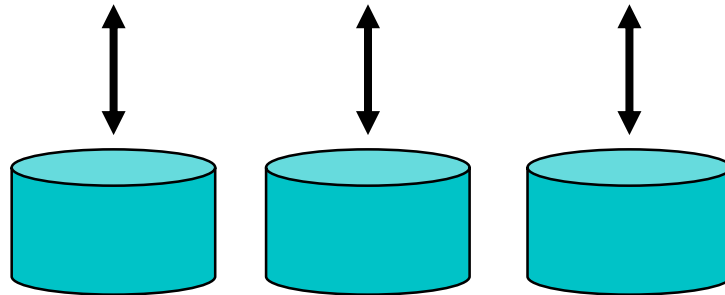
file, logical view



scope of
MPI-I/O



addressed
only by hints

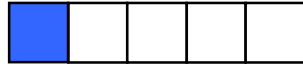


file, physical view

Definitions



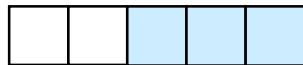
etype (elementary datatype)



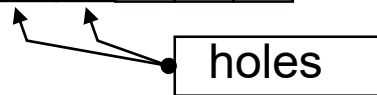
filetype process 0



filetype process 1



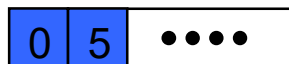
filetype process 2



tiling a file with filetypes:



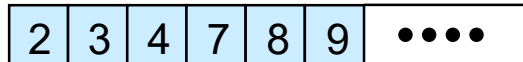
file displacement (number of header bytes)



view of process 0



view of process 1



view of process 2

Comments on Definitions

- file** - an ordered collection of typed data items
- etypes** - is the unit of data access and positioning / offsets
- can be any basic or derived datatype
(with non-negative, monotonically non-decreasing, non-absolute displacem.)
 - generally contiguous, but need not be
 - typically same at all processes
- filetypes** - the basis for partitioning a file among processes
- defines a template for accessing the file
 - different at each process
 - the etype or derived from etype (displacements:
non-negative, monoton. non-decreasing, non-abs., multiples of etype extent)
- view** - each process has its own view, defined by:
a displacement, an etype, and a filetype.
- The filetype is repeated, starting at **displacement**
- offset** - position relative to current view, in units of etype

Opening an MPI File

- **MPI_File_open** is collective over **comm**
- filename's namespace is implementation-dependent!
- filename must reference the same file on all processes
- process-local files can be opened by passing **MPI_COMM_SELF** as **comm**
- returns a file handle *fh*
[represents the file, the process group of **comm**, and the current view]

```
MPI_File_open(comm, filename, amode, info, fh)
```

Fortran

C/C++

Python

language bindings – see MPI Standard
and mpi4py

Default View

```
MPI_File_open(comm, filename, amode, info, fh)
```

- Default:

- displacement = 0
 - etype = MPI_BYTE
 - filetype = MPI_BYTE
- } each process has access to the whole file



- Sequence of MPI_BYTE matches with any datatype
(see MPI-3.1/MPI-4.0, Section 13/14.6.6 on page 549/714)
- Binary I/O (no ASCII text I/O)

Access Modes

- same value of **amode** on all processes in **MPI_File_open**
- Bit vector OR of integer constants (Fortran 77: +)
 - MPI_MODE_RDONLY - read only
 - MPI_MODE_RDWR - reading and writing
 - MPI_MODE_WRONLY - write only
 - MPI_MODE_CREATE - create if file doesn't exist
 - MPI_MODE_EXCL - error creating a file that exists
 - MPI_MODE_DELETE_ON_CLOSE - delete on close
 - MPI_MODE_UNIQUE_OPEN - file not opened concurrently
 - MPI_MODE_SEQUENTIAL - file only accessed sequentially:
mandatory for sequential stream files (pipes, tapes, ...)
 - MPI_MODE_APPEND - all file pointers set to end of file
[caution: reset to zero by any subsequent MPI_FILE_SET_VIEW]

File Info: Reserved Hints

- Argument in `MPI_File_open`, `MPI_File_set_view`, `MPI_File_set_info`
- reserved key values:
 - collective buffering
 - **“collective_buffering”**: specifies whether the application may benefit from collective buffering
 - **“cb_block_size”**: data access in chunks of this size
 - **“cb_buffer_size”**: on each node, usually a multiple of block size
 - **“cb_nodes”**: number of nodes used for collective buffering
 - disk striping (only relevant in `MPI_FILE_OPEN`)
 - **“striping_factor”**: number of I/O devices used for striping
 - **“striping_unit”**: length of a chunk on a device (in bytes)
- `MPI_INFO_NULL` may be passed

Closing and Deleting a File

- Close: collective

```
MPI_File_close(fh)
```

- Delete:

- automatically by `MPI_FILE_CLOSE`
if `amode=MPI_DELETE_ON_CLOSE` | ...
was specified in `MPI_FILE_OPEN`
- deleting a file that is not currently opened:

```
MPI_File_delete(filename, info)
```

[same implementation-dependent rules as in `MPI_FILE_OPEN`]

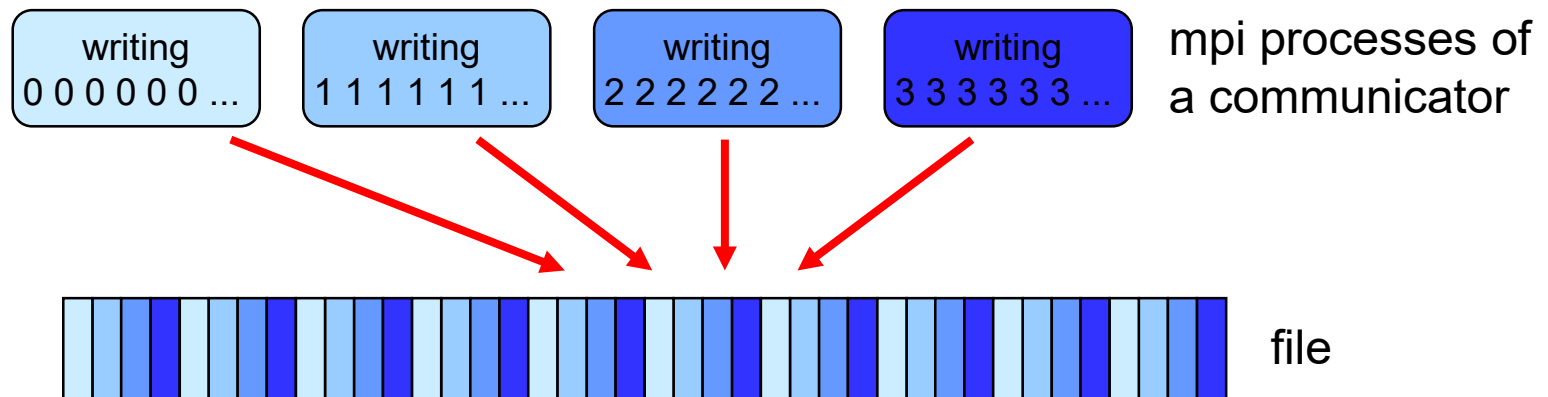
Writing with Explicit Offsets

```
MPI_File_write_at(fh, offset, buf, count, datatype, status)
```

- writes `count` elements of `datatype` from memory `buf` to the file
- starting `offset * units of etype` from begin of view
- the elements are stored into the locations of the current view
- the sequence of basic datatypes of `datatype` (= signature of `datatype`) must match contiguous copies of the `etype` of the current view

MPI-IO Exercise 1: Four processes write a file in parallel

- each process should write its rank (as one character) ten times to the offsets = $\text{my_rank} + i * \text{size_of_MPI_COMM_WORLD}$, $i=0..9$
- Result: “01230123012301230123012301230123012301230123”
- Each process uses the default view



- please, use skeleton:

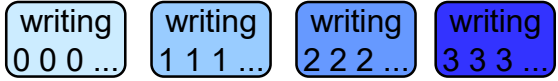
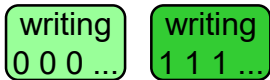
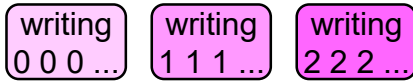
C `cp ~/MPI/tasks/C/Ch13/mpi_io_exa1_skel.c my_exa1.c`

Fortran `cp ~/MPI/tasks/F_30/Ch13/mpi_io_exa1_skel_30.f90 my_exa1_30.f90`

Python `cp ~/MPI/tasks/PY/Ch13/mpi_io_exa1_skel.py my_exa1.py`

- edit; compile; **rm -f my_test_file**; `mpirun ...` (always remove `my_test_file` before re-run)
- **cat my_test_file**; **echo**; **wc -c my_test_file** (verifying the result)

MPI-IO Advanced Exercise 1b: MPI_File_set_size

- `rm -f my_test_file`
- Run program of Exercise 1 with 4 processes: 
Expected result `"0123012301230123012301230123012301230123"`
- Do **not** remove `my_test_file` and run again with **only 2** processes: 
Expected result `"010101010101010101010101010101230123012301230123"`
- Please, make a copy of your result: `cp my_exa1.c my_exa1b.c` or `_30.f90`
- **Set the file size to 0 (zero) directly after the MPI_File_open.**
 - Use `MPI_File_set_size()`
 - For the interface, please look into the MPI standard.
- Compile and run again (**without** removing `my_test_file`), now with **3** processes:
`cat my_test_file ; echo ; wc -c my_test_file` 
Expected result: `"012012012012012012012012012012012012012"`

Outline – Block 2

- **Block 1**
 - Introduction [323]
 - **Definitions** [328]
 - Open / Close [330]
 - WRITE / **Explicit Offsets** [335]
 - Exercise 1 [336]
- **Block 2**
 - **File Views** [338]
 - **Subarray & Darray** [342]
 - I/O Routines Overview [350]
 - READ / Explicit Offsets [352]
 - **Individual File Pointer** [353]
 - Exercise 2 [355]
- **Block 3**
 - **Shared File Pointer** [358]
 - **Collective** [360]
 - Non-Blocking / Split Collective [364/365]
 - Other Routines [368]
 - Error Handling [369]
 - Implementation Restrictions [370]
 - **Summary** [371]
 - Exercise 3 [372]
 - Exercise 4 [373]

File Views

- Provides a visible and accessible set of data from an open file
- A separate view of the file is seen by each process through triple := (displacement, etype, filetype)
- User can change a view during the execution of the program - but collective operation
- A linear byte stream, represented by the triple (0, MPI_BYTE, MPI_BYTE), is the default view

Set/Get File View

- Set view
 - changes the process's view of the data
 - local and shared file pointers are reset to zero
 - collective operation
 - etype and filetype must be committed
 - datarep argument is a string that specifies the format in which data is written to a file:
 - “native”, “internal”, “external32”, or user-defined
 - same etype extent and same datarep on all processes
- Get view
 - returns the process's view of the data

```
MPI_File_set_view(fh, disp, etype, filetype, datarep, info)
```

```
MPI_File_get_view(fh, disp, etype, filetype, datarep)
```


Data Representation, I.

- “native”
 - data stored in file identical to memory
 - on homogeneous systems no loss in precision or I/O performance due to type conversions
 - on heterogeneous systems loss of interoperability
 - no guarantee that MPI files accessible from C/Fortran
- “internal”
 - data stored in implementation specific format
 - can be used with homogeneous or heterogeneous environments
 - implementation will perform type conversions if necessary
 - no guarantee that MPI files accessible from C/Fortran

Data Representation, II.

- “external32”
 - follows standardized representation (IEEE)
 - all input/output operations are converted from/to the “external32” representation
 - files can be exported/imported between different MPI environments
 - due to type conversions from (to) native to (from) “external32” data precision and I/O performance may be lost
 - “internal” may be implemented as equal to “external32”
 - can be read/written also by non-MPI programs
- user-defined

No information about the default,
i.e., `datarep` without `MPI_File_set_view()` is not defined

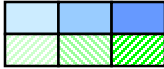
Fileview examples with SUBARRAY and DARRAY

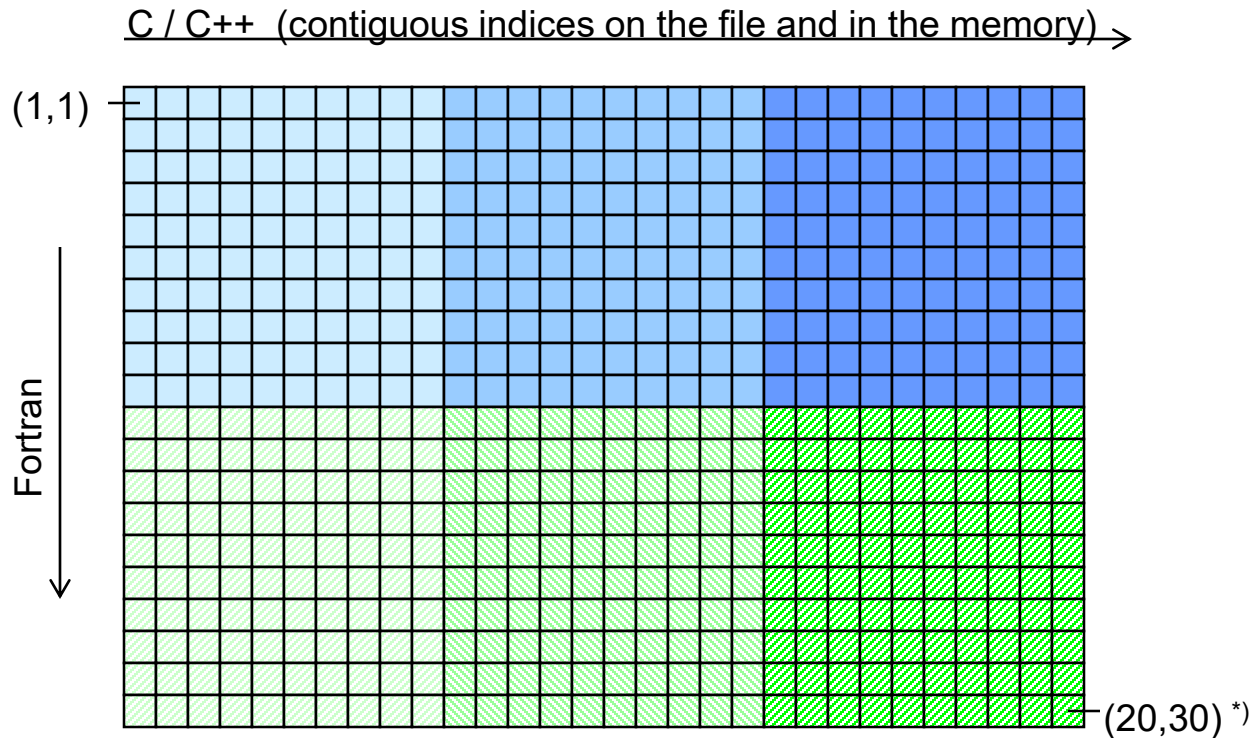
- Task
 - reading a global matrix from a file
 - storing a subarray into a local array on each process
 - according to a given distribution scheme

Example with Subarray, I.

- 2-dimensional distribution scheme: (BLOCK,BLOCK)
- garray on the file 20x30:
 - Contiguous indices is language dependent:
 - in Fortran: (1,1), (2,1), (3,1), ... , (1,10), (2,10), (3,10), ..., (20,30)
 - in C/C++: [0][0], [0][1], [0][2], ... , [10][0], [10][1], [10][2], ..., [19][29]
- larray = local array in each MPI process
= subarray of the global array
- same ordering on file (garray) and in memory (larray)

Example with Subarray, II. — Distribution

- Process topology: 2x3 
- global array on the file: 20x30
- distributed on local arrays in each process: 10x10



*) Figure: as in a math matrix, first index is vertical (i.e., not horizontal as in a x,y-diagram)

Example with Subarray, III. — Reading the file

```
!!!! real garray(20,30) ! these HPF-like comment lines !
!!!! PROCESSORS procs(2, 3) ! explain the data distribution !
!!!! DISTRIBUTE garray(BLOCK,BLOCK) onto procs ! used in this MPI program !
real larray(10,10) ; integer (kind=MPI_OFFSET_KIND) disp,offset; disp=0; offset=0
ndims=2 ; psizes(1)=2 ; period(1)=.false. ; psizes(2)=3 ; period(2)=.false.
call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, psizes, period,
call MPI_COMM_RANK(comm, rank, ierror) .TRUE., comm, ierror)
call MPI_CART_COORDS(comm, rank, ndims, coords, ierror)

gsizes(1)=20 ; lsizes(1)= 10 ; starts(1)=coords(1)*lsizes(1)
gsizes(2)=30 ; lsizes(2)= 10 ; starts(2)=coords(2)*lsizes(2)
call MPI_TYPE_CREATE_SUBARRAY(ndims, gsizes, lsizes, starts,
MPI_ORDER_FORTRAN, MPI_REAL, subarray_type, ierror)
call MPI_TYPE_COMMIT(subarray_type , ierror)


call MPI_FILE_OPEN(comm, 'exa_subarray_testfile', MPI_MODE_CREATE +
MPI_MODE_RDWR, MPI_INFO_NULL, fh, ierror)
call MPI_FILE_SET_VIEW (fh, disp, MPI_REAL, subarray_type, 'native',
MPI_INFO_NULL, ierror)
call MPI_FILE_READ_AT_ALL(fh, offset, larray, lsizes(1)*lsizes(2), MPI_REAL,
status, ierror)
```

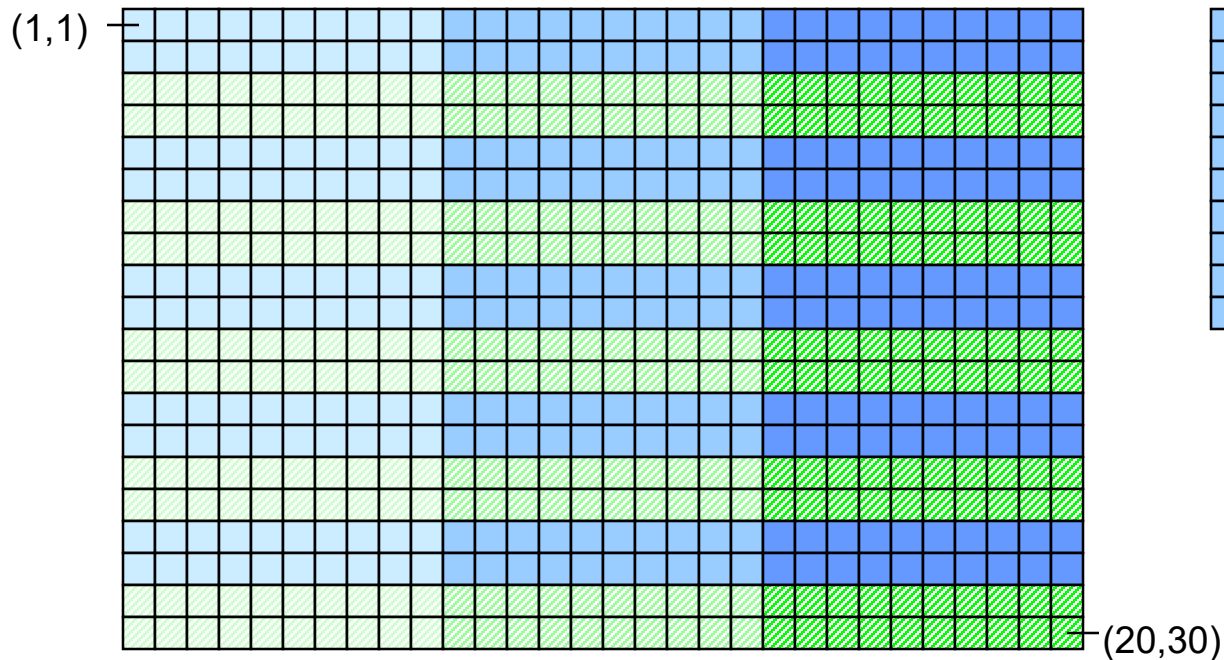
Example with Subarray, IV.

- All MPI coordinates and indices start with 0, even in Fortran, i.e. with MPI_ORDER_FORTRAN
- MPI indices (here `starts`) may differ (✓) from Fortran indices
- Block distribution on 2*3 processes:

<pre>rank = 0 coords = (0, 0) starts = (0, 0) garray(1:10, 1:10) = larray (1:10, 1:10)</pre>	<pre>rank = 1 coords = (0, 1) starts = (0, 10) garray(1:10, 11:20) = larray (1:10, 1:10)</pre>	<pre>rank = 2 coords = (0, 2) starts = (0, 20) garray(1:10, 21:30) = larray (1:10, 1:10)</pre>
<pre>rank = 3 coords = (1, 0) starts = (10, 0) garray(11:20, 1:10) = larray (1:10, 1:10)</pre>	<pre>rank = 4 coords = (1, 1) starts = (10, 10) garray(11:20, 11:20) = larray (1:10, 1:10)</pre>	<pre>rank = 5 coords = (1, 2) starts = (10, 20) garray(11:20, 21:30) = larray (1:10, 1:10)</pre>

Example with Darray, I.

- Distribution scheme: (CYCLIC(2), BLOCK)
- Cyclic distribution in first dimension with strips of length 2
- Block distribution in second dimension
- distribution of global garray onto the larray in each of the 2x3 processes 
- garray on the file:
 - e.g., larray on process (0,1):



Example with Darray, II.

```
!!!! real garray(20,30) ! these HPF-like comment lines !
!!!! PROCESSORS procs(2, 3) ! explain the data distribution!
!!!! DISTRIBUTE garray(CYCLIC(2),BLOCK) onto procs !used in this MPI program!
real larray(10,10); integer (kind=MPI_OFFSET_KIND) disp, offset; disp=0; offset=0

call MPI_COMM_SIZE(comm, size, ierror)
ndims=2 ; psizes(1)=2 ; period(1)=.false. ; psizes(2)=3 ; period(2)=.false.
call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, psizes, period,
                    .TRUE., comm, ierror)
call MPI_COMM_RANK(comm, rank, ierror)
call MPI_CART_COORDS(comm, rank, ndims, coords, ierror)

gsizes(1)=20 ; distribs(1)= MPI_DISTRIBUTE_CYCLIC; dargs(1)=2
gsizes(2)=30 ; distribs(2)= MPI_DISTRIBUTE_BLOCK; dargs(2)=
                    MPI_DISTRIBUTE_DFLT_DARG

call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, gsizes, distribs, dargs,
                          psizes, MPI_ORDER_FORTRAN, MPI_REAL, darray_type, ierror)
call MPI_TYPE_COMMIT(darray_type , ierror)

call MPI_FILE_OPEN(comm, 'exa_subarray_testfile', MPI_MODE_CREATE +
                  MPI_MODE_RDWR, MPI_INFO_NULL, fh, ierror)
call MPI_FILE_SET_VIEW (fh, disp, MPI_REAL, darray_type, 'native',
                      MPI_INFO_NULL, ierror)
call MPI_FILE_READ_AT_ALL(fh, offset, larray, 10*10, MPI_REAL, istatus, ierror)
```

Example with Darray, III.

- Cyclic distribution in first dimension with strips of length 2
- Block distribution in second dimension
- Processes' tasks:

<p>rank = 0 coords = (0, 0) garray($\begin{bmatrix} 1:2 \\ 5:6 \\ 9:10 \\ 13:14 \\ 17:18 \end{bmatrix}$, 1:10) = larray (1:10, 1:10)</p>	<p>rank = 1 coords = (0, 1) garray($\begin{bmatrix} 1:2 \\ 5:6 \\ 9:10 \\ 13:14 \\ 17:18 \end{bmatrix}$, 11:20) = larray (1:10, 1:10)</p>	<p>rank = 2 coords = (0, 2) garray($\begin{bmatrix} 1:2 \\ 5:6 \\ 9:10 \\ 13:14 \\ 17:18 \end{bmatrix}$, 21:30) = larray (1:10, 1:10)</p>
<p>rank = 3 coords = (1, 0) garray($\begin{bmatrix} 3:4 \\ 7:8 \\ 11:12 \\ 15:16 \\ 19:20 \end{bmatrix}$, 1:10) = larray (1:10, 1:10)</p>	<p>rank = 4 coords = (1, 1) garray($\begin{bmatrix} 3:4 \\ 7:8 \\ 11:12 \\ 15:16 \\ 19:20 \end{bmatrix}$, 11:20) = larray (1:10, 1:10)</p>	<p>rank = 5 coords = (1, 2) garray($\begin{bmatrix} 3:4 \\ 7:8 \\ 11:12 \\ 15:16 \\ 19:20 \end{bmatrix}$, 21:30) = larray (1:10, 1:10)</p>

5 Aspects of Data Access

- Direction: Read / Write
- Positioning [realized via routine names]
 - explicit offset (`_AT`)
 - individual file pointer (no positional qualifier)
 - shared file pointer (`_SHARED` or `_ORDERED`)
(different names used depending on whether non-collective or collective)
- Coordination
 - non-collective
 - collective (`_ALL`)
- Synchronism
 - blocking
 - nonblocking (l) and split collective (`_BEGIN`, `_END`)
- Atomicity, [realized with a separate API: `MPI_File_set_atomicity`]
 - non-atomic (default)
 - atomic: to achieve sequential consistency for conflicting accesses on same fh in different processes

All Data Access Routines

positioning	synchronism	coordination		split collective
		noncollective	collective	
explicit offsets	blocking	READ_AT WRITE_AT	READ_AT_ALL WRITE_AT_ALL	READ_AT_ALL_BEGIN READ_AT_ALL_END
	nonblocking	IREAD_AT IWRITE_AT	IREAD_AT_ALL IWRITE_AT_ALL	WRITE_AT_ALL_BEGIN WRITE_AT_ALL_END
individual file pointers	blocking	READ WRITE	READ_ALL WRITE_ALL	READ_ALL_BEGIN READ_ALL_END
	nonblocking	IREAD IWRITE	IREAD_ALL IWRITE_ALL	WRITE_ALL_BEGIN WRITE_ALL_END
shared file pointer	blocking	READ_SHARED WRITE_SHARED	READ_ORDERED WRITE_ORDERED	READ_ORDERED_BEGIN READ_ORDERED_END
	nonblocking	IREAD_SHARED IWRITE_SHARED	N/A	WRITE_ORDERED_BEGIN WRITE_ORDERED_END

Read e.g. **MPI_FILE_READ_AT**

New in MPI-3.1

Explicit Offsets

e.g. `MPI_File_read_at(fh, offset, buf, count, datatype, status)`

- attempts to read `count` elements of `datatype`
- starting `offset * units of etype` from begin of view (= `displacement`)
- the sequence of basic datatypes of `datatype` (= signature of `datatype`) must match contiguous copies of the `etype` of the current view
- EOF can be detected by noting that the amount of data read is less than `count`
 - i.e. EOF is no error!
 - use `MPI_Get_count(status, datatype, recv_count)`

Individual File Pointer, I.

e.g. `MPI_File_read(fh, buf, count, datatype, status)`

- same as “*Explicit Offsets*”, except:
- the offset is the current value of the **individual file pointer** of the calling process
- the individual file pointer is updated by

$$\text{new_fp} = \text{old_fp} + \frac{\text{elements}(\text{datatype})}{\text{elements}(\text{etype})} * \text{count}$$

i.e. it points to the next etype after the last one that will be accessed
(if EOF is reached, then *recv_count* is used, see previous slide)

Individual File Pointer, II.


```
MPI_File_seek(fh, offset, whence)
```

- set individual file pointer fp:
 - set fp to offset – if whence=MPI_SEEK_SET
 - advance fp by offset – if whence=MPI_SEEK_CUR
 - set fp to EOF+offset – if whence=MPI_SEEK_END

```
MPI_File_get_position(fh, offset)
```

```
MPI_File_get_byte_offset(fh, offset, disp)
```

- to inquire offset
- to convert offset into byte displacement
[e.g. for *disp* argument in a new view]

 language bindings – see MPI Standard
and mpi4py

MPI-IO Exercise 2:

Using fileviews and individual filepointers

- Copy to your local directory:

C

```
cp ~/MPI/tasks/C/Ch13/mpi_io_exa2_skel.c my_exa2.c
```

Fortran

```
cp ~/MPI/tasks/F_30/Ch13/mpi_io_exa2_skel_30.f90 my_exa2_30.f90
```

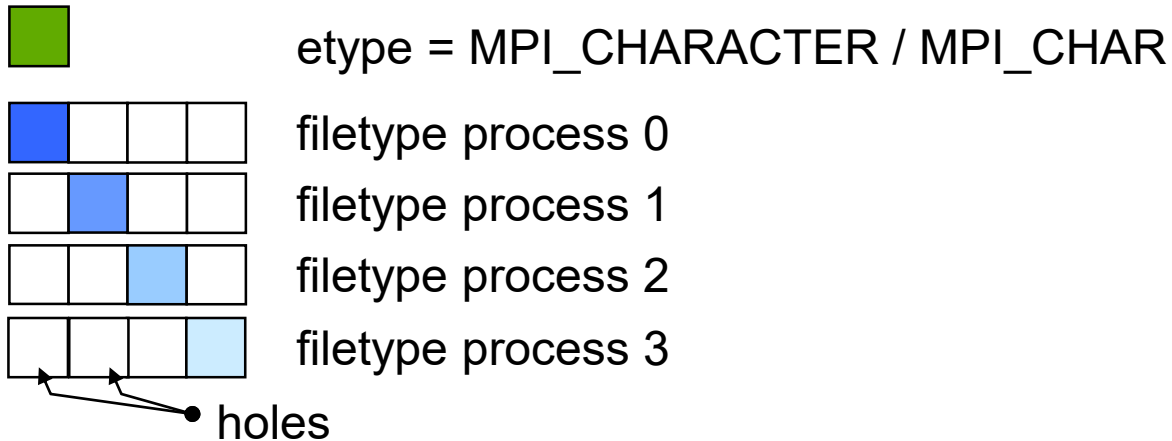
Python

```
cp ~/MPI/tasks/PY/Ch13/mpi_io_exa2_skel.py my_exa2.py
```

- Tasks:
 - Each MPI-process of `my_exa2` should write one character to a file:
 - process “rank=0” should write an ‘a’
 - process “rank=1” should write an ‘b’
 - ...
 - Use a 1-dimensional fileview with `MPI_TYPE_CREATE_SUBARRAY`
 - The pattern should be repeated 3 times, i.e., four processes should write: “abcdabcdabcd”
 - Please, substitute “_____” in your `my_exa2.c` / `_30.f90`
- Edit; compile; **rm -f my_test_file**; `mpirun...` (always remove `my_test_file` before re-run)
- **cat my_test_file**; **echo**; **wc -c my_test_file** (verifying the result)

MPI-IO Exercise 2:

Using fileviews and individual filepointers, continued

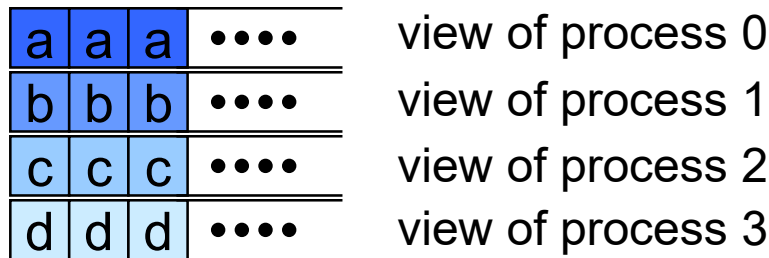


tiling a file with filetypes:



↑ file displacement = 0 (number of header bytes), **identical on all processes**

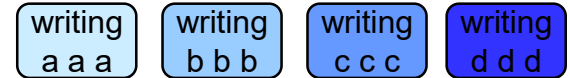
Otherwise optimization may be impossible




MPI-IO Advanced Exercise 2b+c: Append

- `rm -f my_test_file`

- Run program of Exercise 1 with 4 processes:
`cat my_test_file ; echo ; wc -c my_test_file`

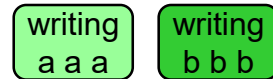


Expected result “” (12 characters)

- **2b)** Please, make a copy of your result: `cp my_exa2.c my_exa2b.c` or `_30.f90`

- Set the displacement **disp** to the current filesize: Use **MPI_File_get_size()**
(For the interface, please look into the MPI standard)
- Compile and run again (**without** removing `my_test_file`), now with 2 processes:

Expected result: “” (18 characters)

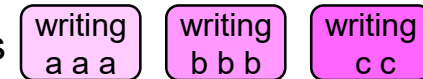


- **2c)** Please, make a copy of your **original** result: `cp my_exa2.c my_exa2c.c` or `_30.f90`

- Use **MPI_File_seek()** to move all individual file pointers to the **end of the file**
(For the interface, please look into the MPI standard)

- Again (**without** removing `my_test_file`), now with 3 processes

Expected result: “” (27 characters)



- **Caution:**– Existing file size should be a multiple of the new filetype size
– Both OpenMPI and mpich may have a bug.

Outline – Block 3

- **Block 1**
 - Introduction [323]
 - **Definitions** [328]
 - Open / Close [330]
 - WRITE / **Explicit Offsets** [335]
 - Exercise 1 [336]
- **Block 2**
 - **File Views** [338]
 - **Subarray & Darray** [342]
 - I/O Routines Overview [350]
 - READ / Explicit Offsets [352]
 - **Individual File Pointer** [353]
 - Exercise 2 [355]
- **Block 3**
 - **Shared File Pointer** [358]
 - **Collective** [360]
 - Non-Blocking / Split Collective [364/365]
 - Other Routines [368]
 - Error Handling [369]
 - Implementation Restrictions [370]
 - **Summary** [371]
 - Exercise 3 [372]
 - Exercise 4 [373]

Shared File Pointer, I.

- same view at all processes mandatory!
- the offset is the current, *global* value of the **shared file pointer** of `fh`
- multiple calls [*e.g. by different processes*] **behave as** if the calls were **serialized**
- non-collective, e.g.

```
MPI_File_read_shared(fh, buf, count, datatype, status)
```

- collective calls are *serialized* in the **order** of the processes' ranks, e.g.:

```
MPI_File_read_ordered(fh, buf, count, datatype, status)
```

Shared File Pointer, II.

`MPI_File_seek_shared(fh, offset, whence)`

`MPI_File_get_position_shared(fh, offset)`

`MPI_File_get_byte_offset(fh, offset, disp)`

- same rules as with individual file pointers

Collective Data Access

- Explicit offsets / individual file pointer:
 - same as non-collective calls by all processes “of `fh`”
 - ***opportunity for best speed!!!***
- shared file pointer:
 - accesses are ordered by the ranks of the processes
 - optimization opportunity:
 - **first, locations within the file for all processes can be computed**
 - **then parallel physical data access by all processes**

Application Scenery, I.

- Scenery A:
 - Task: Each process has to read the whole file
 - Solution: **MPI_File_read_all**
= collective with individual file pointers,
with same view (displacement+etype+filetype)
on all processes
*[internally: striped-reading by several process, only once
from disk, then distributing with bcast]*
- Scenery B:
 - Task: The file contains a list of tasks,
each task requires different compute time
 - Solution: **MPI_File_read_shared**
= non-collective with a shared file pointer
(same view is necessary for shared file p.)

Application Scenery, II.

- Scenery C:
 - Task: The file contains a list of tasks, each task requires **the same** compute time
 - Solution: **MPI_File_read_ordered**
= **collective** with a **shared** file pointer
(same view is necessary for shared file p.)
 - or: **MPI_File_read_all**
= **collective** with **individual** file pointers,
different views: *filetype* with
MPI_Type_create_subarray(1, nproc,
1, myrank, ..., datatype_of_task, *filetype*)
*[internally: both may be implemented the same
and equally with following scenery D]*

Application Scenery, III.

- Scenery D:
 - Task: The file contains a matrix, block partitioning, each process should get a block
 - Solution: generate different filetypes with **MPI_Type_create_darray** or **..._subarray**, the view on each process represents the block that should be read by this process, **MPI_File_read_at_all** with offset=0 (= collective with explicit offsets) reads the whole matrix collectively
[internally: striped-reading of contiguous blocks by several process, then distributed with “alltoall”]

Nonblocking Data Access

e.g. `MPI_File_iread(fh, buf, count, datatype, request)`

`MPI_Wait(request, status)`

`MPI_Test(request, flag, status)`

- analogous to MPI-1 nonblocking

Fortran C/C++ language bindings – see MPI Standard
Python and mpi4py

May be deprecated in MPI-4.1
and removed in MPI-5

Split Collective Data Access, I.

- collective operations may be **split** into two parts:
 - start the split collective operation

```
e.g. MPI_File_read_all_begin(fh, buf, count, datatype)
```

- complete the operation and return the **status**

```
MPI_File_read_all_end(fh, buf, status)
```



language bindings – see MPI Standard
and mpi4py

Split Collective Data Access, II.

- Rules and Restrictions:
 - the **MPI_..._begin** calls are collective
 - the **MPI_..._end** calls are collective, too
 - only one active (pending) split or regular collective operation per file handle at any time
 - split collective does not match ordinary collective
 - same **buf** argument in **MPI_..._begin** and **MPI_..._end** call
- opportunity to overlap file I/O and computation
- but also a valid implementation:
 - does all work within the **MPI_..._begin** routine, passes status in the **MPI_..._end** routine
 - passes arguments from **MPI_..._begin** to **MPI_..._end**, does all work within the **MPI_..._end** routine

Scenery – Nonblocking or Split Collective

- Scenery A:
 - Task: Each process has to read the whole file
 - Solution:
 - `MPI_File_iread_all` or `MPI_File_read_all_begin`
= collective with individual file pointers,
with same view (displacement+etype+filetype)
on all processes
*[internally: starting asynchronous striped-reading
by several process]*
 - then computing some other initialization,
 - `MPI_Wait` or `MPI_File_read_all_end`.
*[internally: waiting until striped-reading completed,
then distributing the data with bcast]*

Other File Manipulation Routines

- Pre-allocating space for a file [*collective call, may be expensive*]

`MPI_File_preallocate(fh, size)`

- Resizing a file [*collective call, may speed up first writing on a file*]

`MPI_File_set_size(fh, size)`

size = 0 → current file content is erased.
Recommended, if the whole file should be overwritten.

- Querying file size

`MPI_File_get_size(filename, size)`

- Querying file parameters

`MPI_File_get_group(fh, group)`

`MPI_File_get_amode(fh, amode)`

- File info object

`MPI_File_set_info (fh, info) [collective call]`

`MPI_File_get_info(fh, info_used)`

Returns a new info object that contains the current setting of **all hints** used by the system related to this open file:

- provided by the application, and
- provided by the system

MPI I/O Error Handling

- File handles have their own error handler
- Default is `MPI_ERRORS_RETURN`,
i.e. **non-fatal**
[vs message passing: `MPI_ERRORS_ARE_FATAL`]
- Default is associated with `MPI_FILE_NULL`
[vs message passing: with `MPI_COMM_WORLD`]
- Changing the default, e.g., after `MPI_Init`:
 - `C/C++` `MPI_File_set_errhandler(MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL);`
 - `Fortran` `CALL MPI_FILE_SET_ERRHANDLER(MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL, ierr)`
 - `Python` `MPI.FILE_NULL.Set_errhandler(MPI.ERRORS_ARE_FATAL)`
- MPI is *undefined* after first erroneous MPI call
- but a **high quality** implementation
will support I/O error handling facilities

Implementation-Restrictions

- ROMIO based MPI libraries:
 - datarep = “internal” and “external32” is still not implemented
 - User-defined data representations are not supported

MPI-I/O: Summary

- Rich functionality provided to support various data representation and access
- MPI I/O routines provide flexibility as well as portability
- Collective I/O routines can improve I/O performance
- ROMIO from Argonne was an initial implementation of MPI I/O
- Available (nearly) on every MPI implementation

- Parallel MPI I/O also used as basis for important I/O packages:
 - Parallel HDF5
<https://portal.hdfgroup.org/display/HDF5/Introduction+to+Parallel+HDF5>
 - Parallel NetCDF, e.g.,
<https://en.wikipedia.org/wiki/NetCDF#Parallel-NetCDF>

MPI-IO Exercise 3: Collective ordered I/O

- Copy to your local directory:

C

```
cp ~/MPI/tasks/C/Ch13/mpi_io_exa3_skel.c my_exa3.c
```

Fortran

```
cp ~/MPI/tasks/F_30/Ch13/mpi_io_exa3_skel_30.f90 my_exa3_30.f90
```

Python

```
cp ~/MPI/tasks/PY/Ch13/mpi_io_exa3_skel.py my_exa3.py
```

- Tasks:
 - Substitute the write call with individual filepointers by a collective write call with shared filepointers
 - Edit your `my_exa3.c` / `_30.f90`
- Compile; **rm -f my_test_file**; `mpirun ...` (always remove `my_test_file` before re-run)
- **cat my_test_file**; **echo**; **wc -c my_test_file** (verifying the result)

MPI-IO Exercise 4: I/O Benchmark

- Use:

```
MPI/tasks/F_30/Ch13/mpi_io_exa4_30.f90
```

(my apologies that there is only a Fortran version)

- Tasks:

- Compile and execute `mpi_io_exa4` on 2, 4 and 8 MPI processes.
- Duplicate “WRITE_ALL & READ_ALL” block and substitute by non-collective “WRITE & READ”.
- Compare collective and non-collective I/O.
- Double the value of `gsize` and compile and execute again.