

# Parallel File I/O with MPI-2

Rolf Rabenseifner  
rabenseifner@hlrs.de

University of Stuttgart  
High-Performance Computing-Center Stuttgart (HLRS)  
www.hlrs.de



MPI-2 Parallel File I/O  
Slide 1 Höchstleistungsrechenzentrum Stuttgart

H L R I S 

## Acknowledgements

This course is based on the MPI-I/O chapter of the MPI-2 tutorial on the MPIDC 2000:

### MPI-2: Extensions to the Message Passing Interface

MISSISSIPPI STATE UNIVERSITY<sup>1</sup>  
**OF PERFORMANCE  
COMPUTING LAB**  
NSF ENGINEERING RESEARCH CENTER



H L R I S <sup>2</sup>

Anthony Skjellum<sup>1</sup>  
Purushotham Bangalore<sup>1</sup>, Shane Hebert<sup>1</sup>  
Rolf Rabenseifner<sup>2</sup>



MPI-2 Parallel File I/O  
Slide 2 Höchstleistungsrechenzentrum Stuttgart

H L R I S 

## Motivation, I.

- 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, II.

- 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
  - non-blocking 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)
- non-blocking / blocking and collective / individual read / write routines
- individual and shared file pointers, explicit offsets
- automatic data conversion in heterog. systems
- file interoperability with external representation



## Logical view / Physical view

mpi processes of a communicator



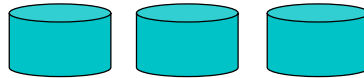
file, logical view



scope of  
MPI-I/O

addressed  
only by hints

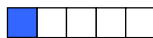
file, physical view



## Definitions



etype (elementary datatype)



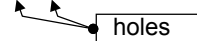
filetype process 0



filetype process 1



filetype process 2



holes

tiling a file with filetypes:



file

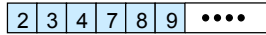
file displacement (number of header bytes)



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]

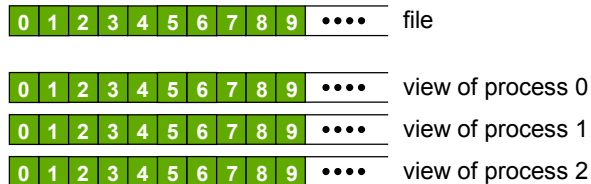
`MPI_FILE_OPEN(comm, filename, amode, info, fh)`



## 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



- MPI\_BYTE matches with any datatype



## 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

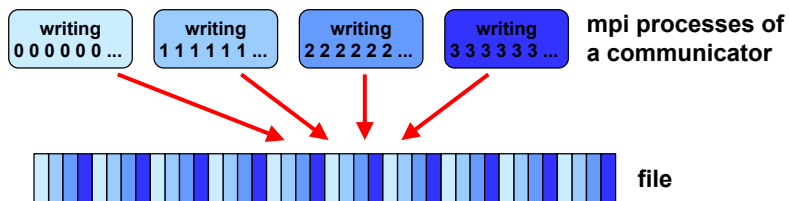
`MPI_FILE_WRITE_AT(fh,offset,buf,count,datatype,status)`

- 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 = `my_rank + i * size_of_MPI_COMM_WORLD`, `i=0..9`
- Result: "01230123012301230123012301230123012301230123"
- Each process uses the default view



- please, use skeleton:  
`cp ~/MPI/course/C/mpi_io/mpi_io_exa1_skel.c my_exa1.c`  
`cp ~/MPI/course/F/mpi_io/mpi_io_exa1_skel.f my_exa1.f`





## File Views

- Provides a set of data visible and accessible 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



## Fileview examples

- 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.

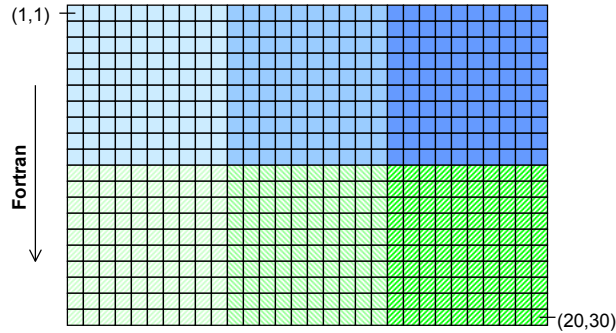
- 2-dimensional distribution scheme: (BLOCK,BLOCK)
- garray on the file 20x30:
  - Contiguous indices is language dependent:
  - in Fortran: (1,1), (2,1), (3,1), ... , (1,10), (2,20), (3,10), ..., (20,30)
  - in C/C++: [0][0], [0][1], [0][2], ... , [10][0], [10][1], [10][2], ..., [19][29]
- larray = 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 processor: 10x10

**C / C++** (contiguous indices on the file and in the memory) →



MPI-2 Parallel File I/O  
Slide 23

Rolf Rabenseifner  
Hochleistungsrechenzentrum Stuttgart

H L R I S 

## 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)
```



MPI-2 Parallel File I/O  
Slide 24

Rolf Rabenseifner  
Hochleistungsrechenzentrum Stuttgart

H L R I S 

## Example with Subarray, IV.

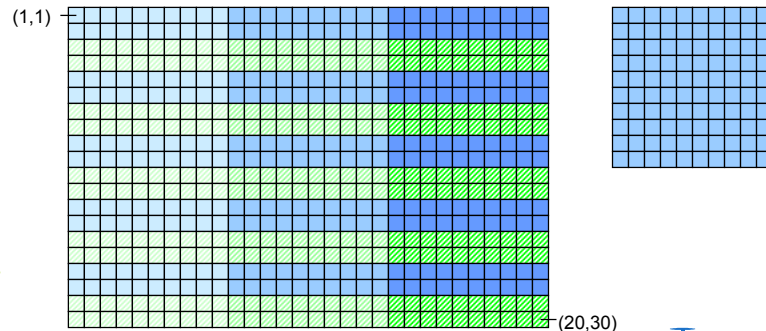
- All MPI coordinates and indices start with 0, even in Fortran, i.e. with MPI\_ORDER\_FORTRAN
- MPI indices (here **starts**) may differ (✓) from Fortran indices
- Block distribution on 2\*3 processes:

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



## 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,
call MPI_COMM_RANK(comm, rank, ierror) .TRUE., comm, 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,

```

Slide 27

Höchstleistungsrechenzentrum Stuttgart

H L R Z

## Example with Darray, III.

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

|                                                                                                                                      |                                                                                                                                       |                                                                                                                                       |
|--------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <p>rank = 0<br/>coords = ( 0, 0)</p> <p>garray( 1: 2<br/>5: 6<br/>9:10, 1:10<br/>13:14<br/>17:18)</p> <p>= larray ( 1:10, 1:10)</p>  | <p>rank = 1<br/>coords = ( 0, 1)</p> <p>garray( 1: 2<br/>5: 6<br/>9:10, 11:20<br/>13:14<br/>17:18)</p> <p>= larray ( 1:10, 1:10)</p>  | <p>rank = 2<br/>coords = ( 0, 2)</p> <p>garray( 1: 2<br/>5: 6<br/>9:10, 21:30<br/>13:14<br/>17:18)</p> <p>= larray ( 1:10, 1:10)</p>  |
| <p>rank = 3<br/>coords = ( 1, 0)</p> <p>garray( 3: 4<br/>7: 8<br/>11:12, 1:10<br/>15:16<br/>19:20)</p> <p>= larray ( 1:10, 1:10)</p> | <p>rank = 4<br/>coords = ( 1, 1)</p> <p>garray( 3: 4<br/>7: 8<br/>11:12, 11:20<br/>15:16<br/>19:20)</p> <p>= larray ( 1:10, 1:10)</p> | <p>rank = 5<br/>coords = ( 1, 2)</p> <p>garray( 3: 4<br/>7: 8<br/>11:12, 21:30<br/>15:16<br/>19:20)</p> <p>= larray ( 1:10, 1:10)</p> |

Slide 28

Höchstleistungsrechenzentrum Stuttgart

H L R Z

## 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
  - non-blocking (I) 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              | Synchronization                 | Non-collective                | Collective                                                                         |
|--------------------------|---------------------------------|-------------------------------|------------------------------------------------------------------------------------|
| Explicit offsets         | blocking                        | READ_AT<br>WRITE_AT           | READ_AT_ALL<br>WRITE_AT_ALL                                                        |
|                          | non-blocking & split collective | IREAD_AT<br>IWRITE_AT         | READ_AT_ALL_BEGIN<br>READ_AT_ALL_END<br>WRITE_AT_ALL_BEGIN<br>WRITE_AT_ALL_END     |
| Individual file pointers | blocking                        | READ<br>WRITE                 | READ_ALL<br>WRITE_ALL                                                              |
|                          | non-blocking & split collective | IREAD<br>IWRITE               | READ_ALL_BEGIN<br>READ_ALL_END<br>WRITE_ALL_BEGIN<br>WRITE_ALL_END                 |
| Shared file pointers     | blocking                        | READ_SHARED<br>WRITE_SHARED   | READ_ORDERED<br>WRITE_ORDERED                                                      |
|                          | non-blocking & split collective | IREAD_SHARED<br>IWRITE_SHARED | READ_ORDERED_BEGIN<br>READ_ORDERED_END<br>WRITE_ORDERED_BEGIN<br>WRITE_ORDERED_END |

Read e.g. `MPI_FILE_READ_AT`



## 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  
(*formula is not valid if EOF is reached*)





## 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\_EOF

**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]



## MPI-IO Exercise 2: Using fileviews and individual filepointers

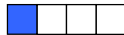
- Copy to your local directory:  
`cp ~/MPI/course/C/mpi_io/mpi_io_exa2_skel.c my_exa2.c`  
`cp ~/MPI/course/F/mpi_io/mpi_io_exa2_skel.f my_exa2.f`
- Tasks:
  - Each MPI-process of `my_exa2` should write one character to a file:
    - process “rank=0” should write an ‘a’
    - process “rank=1” should 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 / .f`
  - Compile and run your `my_exa2.c / .f`



## MPI-IO Exercise 2: Using fileviews and individual filepointers, continued



etype = MPI\_CHARACTER / MPI\_CHAR



filetype process 0



filetype process 1



filetype process 2



filetype process 3

holes

tiling a file with filetypes:

**a b c d a b c d a b c d** file

file displacement = 0 (number of header bytes)

|          |          |          |     |
|----------|----------|----------|-----|
| <b>a</b> | <b>a</b> | <b>a</b> | ... |
| <b>b</b> | <b>b</b> | <b>b</b> | ... |
| <b>c</b> | <b>c</b> | <b>c</b> | ... |
| <b>d</b> | <b>d</b> | <b>d</b> | ... |

view of process 0

view of process 1

view of process 2

view of process 3



## 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  $\text{fh}$ ”
  - ***chance for best speed!!!***
- shared file pointer:
  - accesses are ordered by the ranks of the processes
  - optimization chance:
    - **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`, 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"*]



## Non-blocking 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 non-blocking
- **non-standard** interface if ROMIO is used and not integrated:

`MPIO_Request request`  
`MPI_FILE_IREAD(fh, buf, count, datatype, request)`  
`MPIO_WAIT(request, status)`  
`MPIO_TEST(request, flag, status)`



## 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)`



## 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 `..._END` call
- Chance 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 – Split Collective

- Scenery A:
  - Task: Each process has to read the whole file
  - Solution:
    - `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_FILE_READ_ALL_END`.  
*[internally: waiting until striped-reading finished,  
then distributing the data with bcast]*



## Other File Manipulation Routines

- Pre-allocating space for a file *[may be expensive]*  
`MPI_FILE_PREALLOCATE(fh, size)`
- Resizing a file *[may speed up first writing on a file]*  
`MPI_FILE_SET_SIZE(fh, size)`
- 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)`  
`MPI_FILE_GET_INFO(fh, info_used)`



## 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`:  
`MPI_File_set_errhandler(MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL);`  
`CALL MPI_FILE_SET_ERRHANDLER(MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL, ierr)`
- MPI is *undefined* after first erroneous MPI call
- but a **high quality** implementation  
will support I/O error handling facilities



## Implementation-Restrictions, I.

- MPICH 1.1.2 , mpt.1.3.0.1 on CRAY-T3E, HP MPI 1.4, ...
  - with ROMIO 1.0.1
- ROMIO 1.0.1
  - without shared file pointer routines
  - without `MPI_MODE_SEQUENTIAL` (as `amode` in `MPI_FILE_OPEN`)
  - without split collective routines
  - “status” is not filled in any function
  - EOF is not detected while reading and file pointer is set behind EOF
  - i.e. no chance to detect EOF via status or `MPI_FILE_GET_POSITION`
  - nonblocking with `MPI○_Request`, `MPI○_Wait`, `MPI○_Test`
  - returns only `ierror = MPI_SUCCESS` or `MPI_ERR_UNKNOWN`
  - only “native” data representation
  - without registering user-defined representations





## Implementation-Restrictions, II.

- mpt.1.3.0.1 on T3E
  - only with I/O chapter, based on ROMIO
  - problems with MPI\_TYPE\_CREATE\_DARRAY
    - empty blocks allowed but erroneous implementation
  - without parallel, striped file I/O --> only ~30 MB/s RAID
  - Work around for empty blocks and striped file I/O:
    - see [www.hlrs.de/mpi/mpi\\_t3e.html#StripedIO](http://www.hlrs.de/mpi/mpi_t3e.html#StripedIO)
    - up to 200 MByte/sec parallel striped file I/O but only for applications using parallel collective I/O on large files ( > 3 Mbyte / process)
    - with benchmark example `exa_block.f` [www.hlrs.de/mpi/exa\\_block.f](http://www.hlrs.de/mpi/exa_block.f)



## Implementation-Restrictions, III.

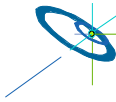
- MPI-2 on the Fujitsu VPP
  - needs extra PE for a server process
    - to handle shared file pointers
    - to run on MPIFS caching file system
  - does not support Unix file system — implementation is underway
  - collective file access — optimization is underway

(state 1/2000)



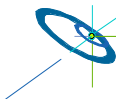
## MPI-I/O: Summary

- 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
- Initial implementations of MPI I/O available (eg. ROMIO from Argonne)
- Available nearly on every MPI implementation



## MPI-IO Exercise 3: Collective ordered I/O

- Copy to your local directory:  
`cp ~/MPI/course/C/mpi_io/mpi_io_exa3_skel.c my_exa3.c`  
`cp ~/MPI/course/F/mpi_io/mpi_io_exa3_skel.f my_exa3.f`
- Tasks:
  - Substitute the write call with individual filepointers by a collective write call with shared filepointers
  - Compile and run your `my_exa3.c / .f`



## MPI-IO Exercise 4: I/O Benchmark

- Copy to your local directory:  

```
cp ~/MPI/course/F/mpi_io/[am]* .
```
  - You receive:  

```
mpi_io_exa4.f  
ad_ufs_open.o, ad_ufs_read.o, ad_ufs_write.o *)
```
  - Tasks:
    - compile and execute `mpi_io_exa4` on 4 PEs
    - compile and link with `ad_ufs*.o` and execute on 4 PEs \*)
    - duplicate lines 65 –93 three times and substitute “WRITE\_ALL” by “WRITE”, “READ\_ALL”, “READ” and execute on 4 PEs
    - double the value of `gsize` and compile and execute on 8 PEs
    - link without `ad_ufs*.o` and execute on 8 PEs \*)
- \*) `ad_ufs` only on T3Es with striped file system

