SLING

Advanced MPI

Parallel File I/O

EURO

Leon Kos

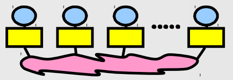
University of Ljubljana, FME, LECAD lab

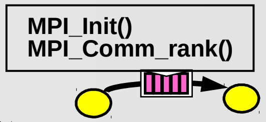
Acknowledgments

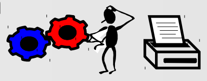
- *Parallel File I/O* is Chapter 13 from *Introduction to the Message Passing Interface (MPI)* course by Rolf Rabenseifner from University of Stuttgart and High-Performance Computing-Center Stuttgart (HLRS)
- The MPI-1.1 part of this course is partially based on the MPI course developed by the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh.
- Thanks to the EPCC, especially to Neil MacDonald, Elspeth Minty, Tim Harding, and Simon Brown.
- Course Notes and exercises of the EPCC course can be used together with this slides.
- The MPI-2.0 part is partially based on the MPI-2 tutorial at the MPIDC 2000 by Anthony Skjellum, Purushotham Bangalore, Shane Hebert (High Performance Computing Lab, Mississippi State University, and Rolf Rabenseifner (HLRS)
- Some MPI-3.0 detailed slides are provided by the MPI-3.0 ticket authors, chapter authors, or chapter working groups, Richard Graham (chair of MPI-3.0), and Torsten Hoefler (additional example about new one-sided interfaces)
- Thanks to Dr. Claudia Blaas-Schenner from TU Wien (Vienna) and many other trainers and participants for all their helpful hints for optimizing this course over so many years.

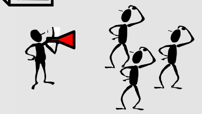
Chap.13 Parallel File I/O

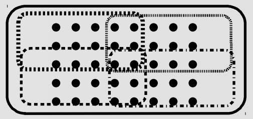
Parallel File I/O

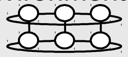


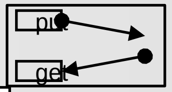


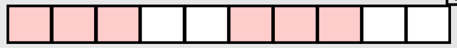


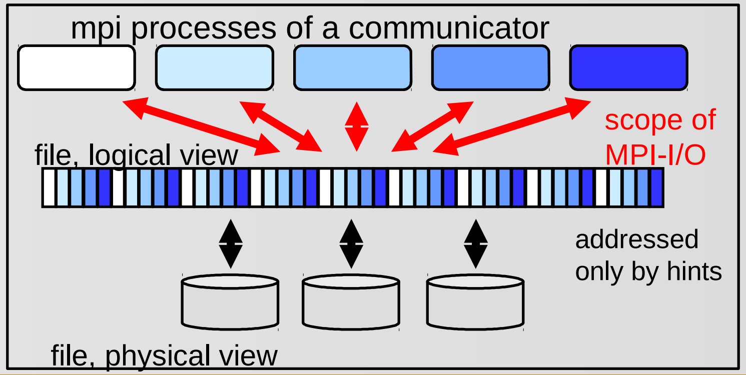












Outline

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
nonblocking operations to overlap computation & I/O	->	MPI_I..., MPI_Wait, ... new **split** collective interface
non-contiguous access	->	MPI derived datatypes

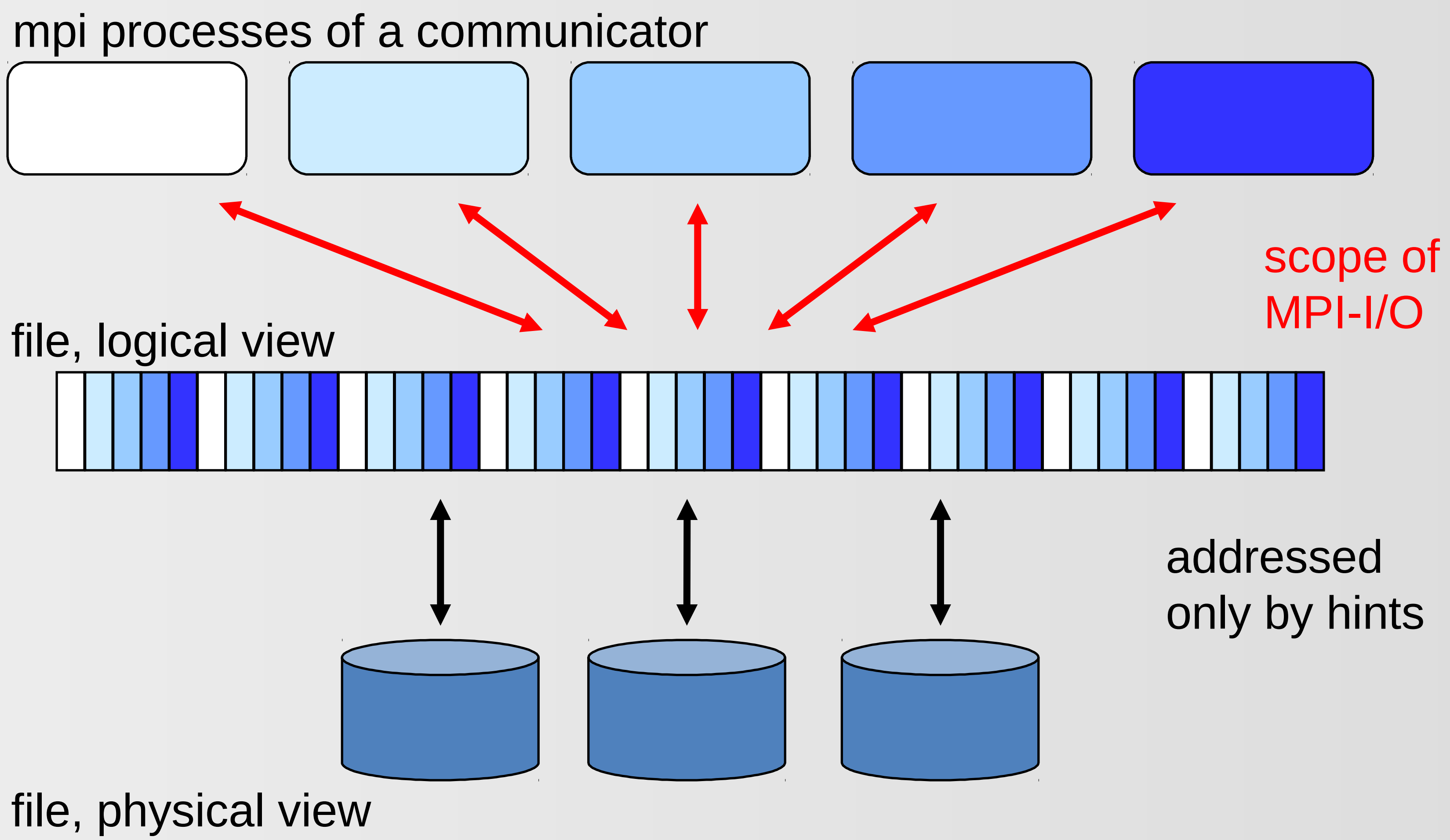
MPI-I/O Features

- 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

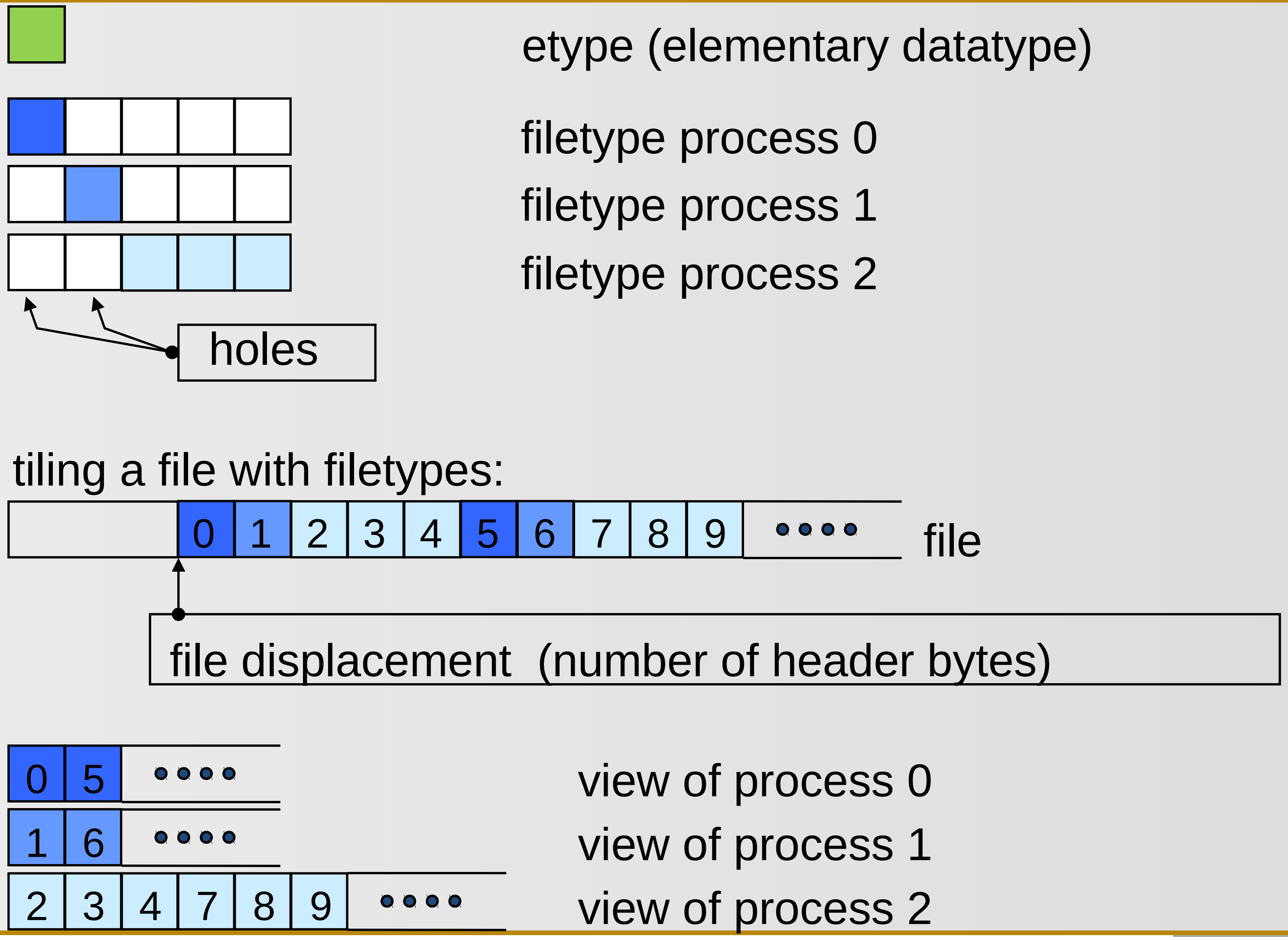
MPI-I/O Principles

- 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



Definitions



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., <u>multiples of etype extent</u>)
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)
```

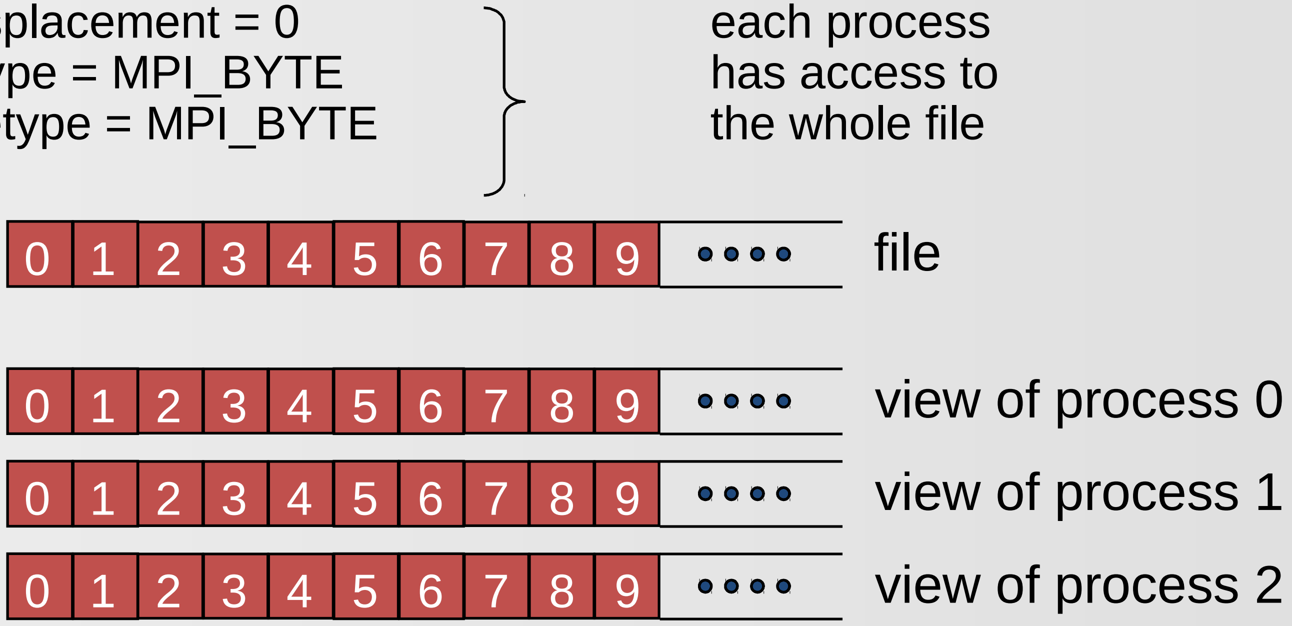
Fortran C/C++ language bindings – see MPI Standard

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-3.0, Section 13.6.5 on page 544 / MPI-3.1, Section 13.6.6 on page 549)
- 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

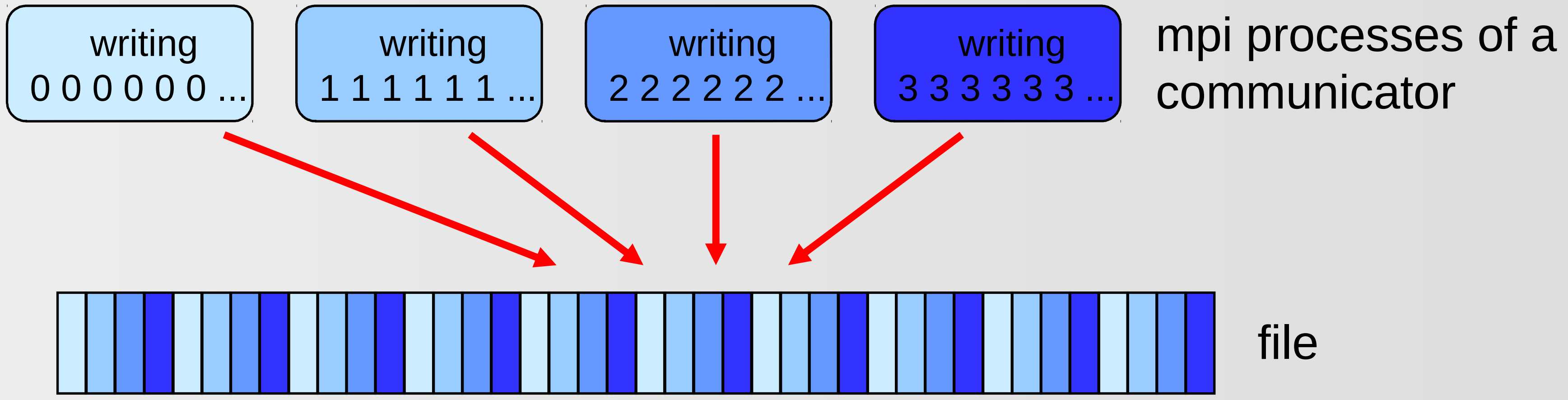
MPI_FILE_WRITE_AT(fh,offset,buf,count,datatype,*status*)

- 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

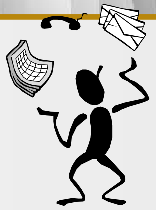
  to the offsets = my_rank + i * size_of_MPI_COMM_WORLD, i=0..9
- Result: “0123012301230123012301230123012301230123“
- Each process uses the default view



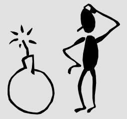
- please, use skeleton:

```
cp ~/MPI/tasks/C/Ch13/mpi_io_exa1_skel.c  my_exa1.c
cp ~/MPI/tasks/F_30/Ch13/mpi_io_exa1_skel_30.f90   my_exa1_30.f90
```

During the Exercise



Please **stay here in the main room** while you do this exercise



And have fun with this short exercise

Please do not look at the solution before you finished this exercise,

otherwise,

90% of your learning outcome may be lost

As soon as you finished the exercise,

please **go to your breakout room**

and continue your discussions with your fellow learners:

Who of you uses NetCDF or HDF5?

As far as I know, both use MPI-I/O.

Outline – Block 2

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

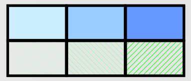
Example with Subarray, I.

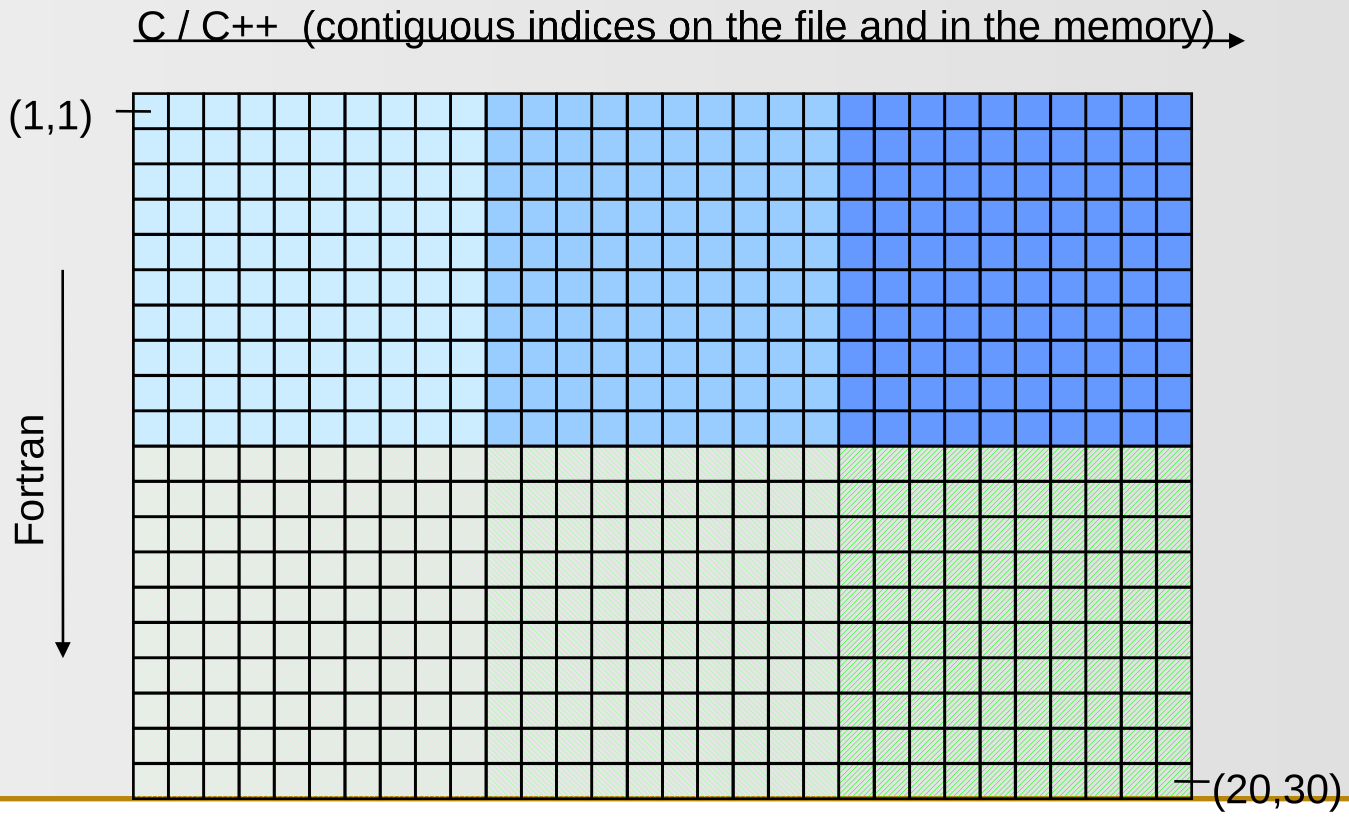
- 2-dimensional distribution scheme: (BLOCK,BLOCK)
- garray on the file 20x30:
  - Contiguous indices is language dependent:
  - in Fortran: (1,1), (2,1), (3,1), ... , (1,10), (2,20), (3,10), ..., (20,30)
  - in C/C++: [0][0], [0][1], [0][2], ... , [10][0], [10][1], [10][2], ..., [19][29]
- larray =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





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

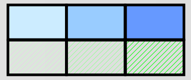
- Block distribution on 2*3 processes:

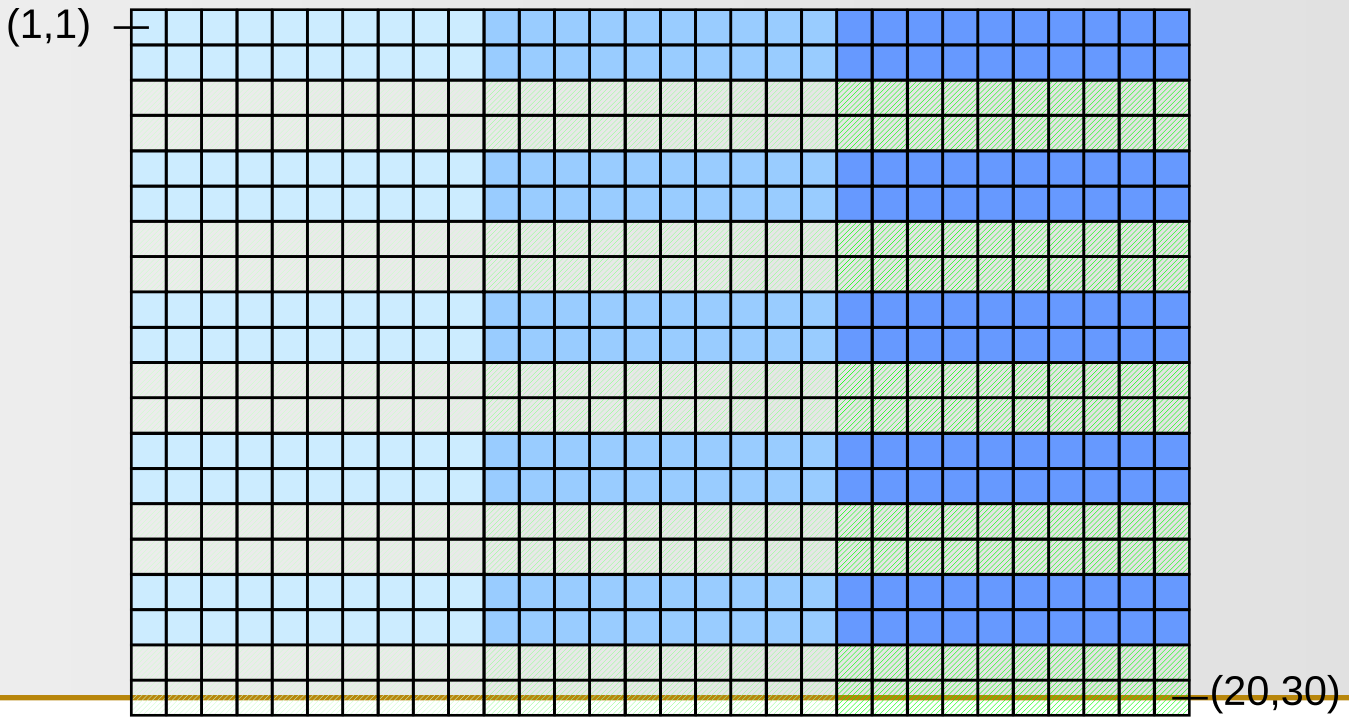
rank = 0
coords = (0, 0)
starts = (0, 0)
garray(1:10, 1:10)
= larray (1:10, 1:10)
	rank = 1
	coords = (0, 1)
	starts = (0, 10)
	garray(1:10, 11:20)
	= larray (1:10, 1:10)
		rank = 2
		coords = (0, 2)
		starts = (0, 20)
		garray(1:10, 21:30)
		= larray (1:10, 1:10)

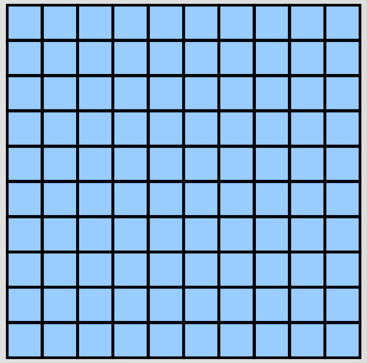
rank = 3
coords = (1, 0)
starts = (10, 0)
garray(11:20, 1:10)
= larray (1:10, 1:10)
	rank = 4
	coords = (1, 1)
	starts = (10, 10)
	garray(11:20, 11:20)
	= larray (1:10, 1:10)
		rank = 5
		coords = (1, 2)
		starts = (10, 20)
		garray(11:20, 21:30)
		= larray (1:10, 1:10)

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,
                                                        .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:

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)
	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)
		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)

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)
	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)
		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)

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 (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	synchronism	coordination		
		noncollective	collective	split 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

$$\text{new\_fp} = \text{old\_fp} + \frac{\text{elements(datatype)}}{} * \text{count}$$

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]

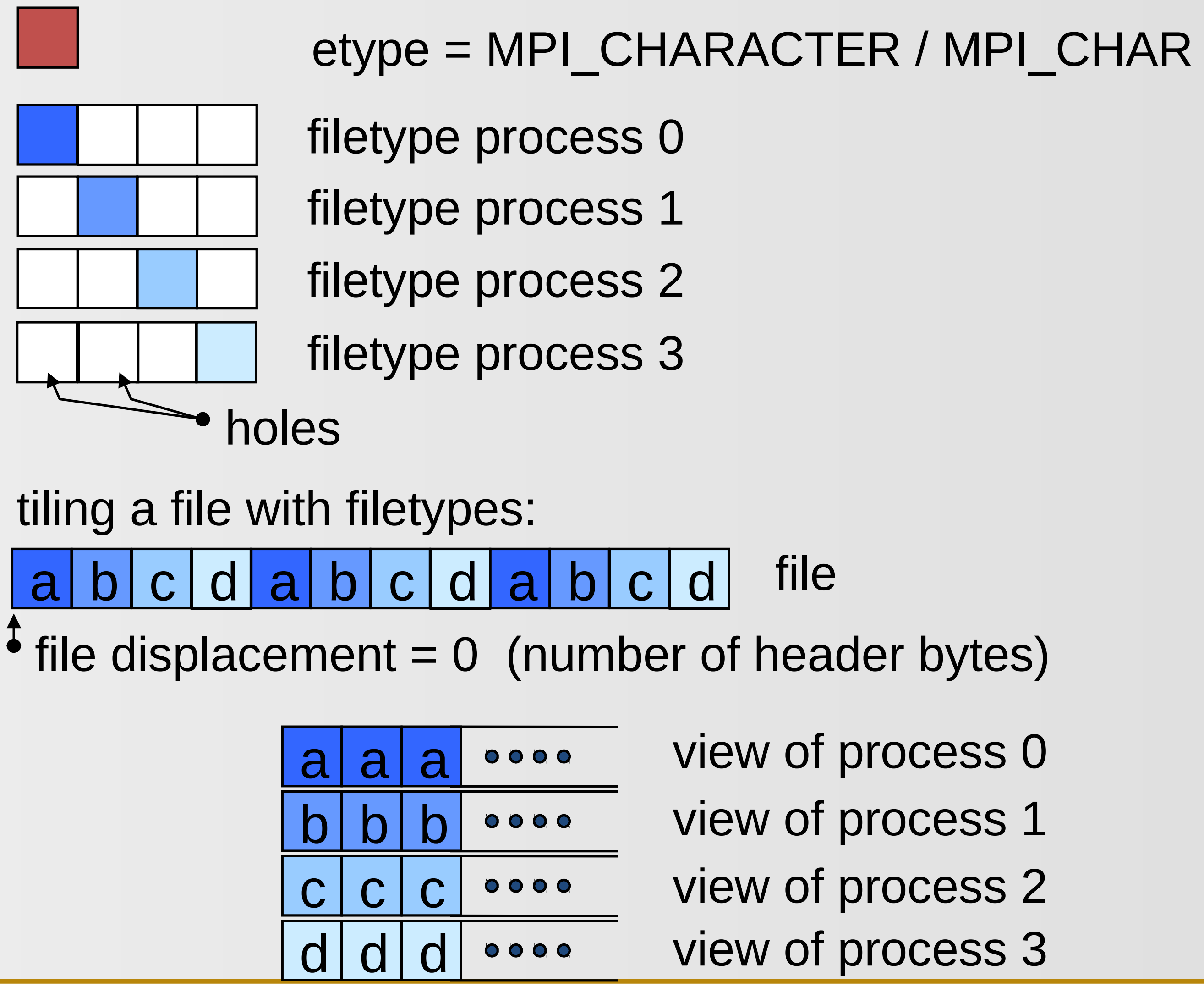
MPI–IO Exercise 2: Using fileviews and individual filepointers

- Copy to your local directory:

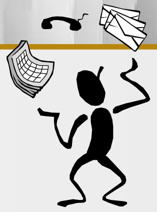
```
cp ~/MPI/tasks/C/Ch13/mpi_io_exa2_skel.c  my_exa2.c

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

- 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`
  - Compile and run your `my_exa2.c / _30.f90`

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

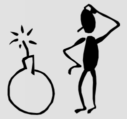


During the Exercise



Please **stay here in the main room** while you do this exercise

And have fun with this short exercise



Please do not look at the solution before you finished this exercise,

otherwise,

90% of your learning outcome may be lost

As soon as you finished the exercise,

please **go to your breakout room**

and continue your discussions with your fellow learners:

Ask yourself, whether the datatype is a 1- or higher-dimensional array?

And don't forget that counts are normally elements and not bytes!

And to look at the declaration of the buffer is also helpful

to answer tha last ____ question

Outline – Block 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 **`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,
    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

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
- 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 – Split Collective

- Scenery A:
  - Task:Each process has to read the whole file
  - Solution: o 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]

    o then computing some other initialization,

    o 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 [*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)
- 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:

```
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

- 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
- 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/tasks/**C**/Ch13/mpi_io_exa3_skel.c **my_exa3.c**

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

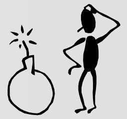
- Tasks:
  - Substitute the write call with individual filepointers

    by a collective write call with shared filepointers
  - Compile and run your `my_exa3.c / _30.f90`

During the Exercise

Please **stay here in the main room** while you do this exercise

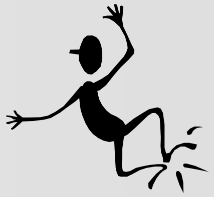
And have fun with this short exercise



Please do not look at the solution before you finished this exercise,

otherwise,

90% of your learning outcome may be lost

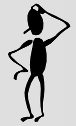


As soon as you finished the exercise,

please **go to your breakout room**

and continue your discussions with your fellow learners:

This exercise is mainly removing all about the fileview.



With the shared file pointer and collective writing, this exercise

is a one-line problem, isn't it?

Good luck!

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.

Chapter 12-(1), Exercise 1:
MPI_TYPE_CONTIGUOUS Scenery, III.

C

MPI/tasks/C/Ch12/solutions/derived-contiguous.c

```
struct buff{
     int   i;
     int   j;
} snd_buf, rcv_buf, sum;
```

Provided in the skeleton

```
TYPE(MPI_Datatype) :: send_recv_type
```

```
CALL MPI_Type_contiguous(2, MPI_INTEGER, send_recv_type)
CALL MPI_Type_commit(send_recv_type)
```

```
sum.i = 0;                    sum.f = 0;
snd_buf.i = my_rank;  snd_buf.j = 10*my_rank;
for( i = 0; i < size; i++)
{ MPI_Issend(&snd_buf,1,send_recv_type,right,17,MPI_COMM_WORLD, &request);
  MPI_Recv ( &rcv_buf,1,send_recv_type,left, 17,MPI_COMM_WORLD, &status);
  MPI_Wait(&request, &status);
  snd_buf = rcv_buf;
  sum.i += rcv_buf.i;  sum.j += rcv_buf.j;
}

printf ("PE %i: Sum = %i and %i \n", my_rank, sum.i, sum.j);
```

APPENDIX: Solution to exercises

Chapter 13-(1):

Parallel file I/O exercise 1 – explicit file-pointer

C

MPI/tasks/C/Ch13/solutions/mpi_io_exa1.c

```
MPI_Offset offset;
…

    MPI_File_open(MPI_COMM_WORLD, "my_test_file",
                  MPI_MODE_RDWR | MPI_MODE_CREATE,
                  MPI_INFO_NULL, &fh);

    for (i=0; i<10; i++) {
      buf = '0' + (char)my_rank;
      offset = my_rank + size*i;
      MPI_File_write_at(fh, offset, &buf, 1, MPI_CHAR, &status);
    }
```

or MPI_MODE_**WRONLY**

Fortran

```
        INTEGER (KIND=MPI_OFFSET_KIND) offset
        …

        CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'my_test_file',
     &                       IOR(MPI_MODE_RDWR, MPI_MODE_CREATE),
     &                       MPI_INFO_NULL, fh, ierror)

        DO I=1,10
          buf = CHAR( ICHAR('0') + my_rank )
          offset = my_rank + size*(i-1)
          CALL MPI_FILE_WRITE_AT(fh, offset, buf, 1, MPI_CHARACTER,
     &                             status, ierror)
        END DO
```

or MPI_MODE_**WRONLY**

MPI/tasks/F_30/Ch13/solutions/mpi_io_exa1.f

Chapter 13-(2):Parallel file I/O exercise 2 – with fileview

C

MPI/tasks/C/Ch13/solutions/mpi_io_exa2.c

```
MPI_Offset disp;
…
    ndims = 1;
    array_of_sizes[0]    = size;
    array_of_subsizes[0] = 1;
    array_of_starts[0]   = my_rank;
    …
    MPI_Type_create_subarray(…);
    MPI_Type_commit(&filetype);
    MPI_File_open(…, MPI_MODE_RDWR | MPI_MODE_CREATE, …);
    disp = 0;
    MPI_File_set_view(…);
    for (i=0; i<3; i++) {
      buf = 'a' + (char)my_rank;
      MPI_File_write(fh, &buf, 1, etype, &status);
    }
```

or MPI_MODE_**WRONLY**

or **MPI_CHAR**

Fortran

MPI/tasks/F_30/Ch13/solutions/mpi_io_exa2.f

```
INTEGER (KIND=MPI_OFFSET_KIND) disp
…
ndims = 1
array_of_sizes(1)    = size
array_of_subsizes(1) = 1
array_of_starts(1)   = my_rank
…
CALL MPI_TYPE_CREATE_SUBARRAY(…)
CALL MPI_TYPE_COMMIT(filetype, ierror)
CALL MPI_FILE_OPEN( …, IOR(MPI_MODE_RDWR, MPI_MODE_CREATE), …)
disp = 0
CALL MPI_FILE_SET_VIEW(…)
DO I=1,3
  buf = CHAR( ICHAR('a') + my_rank )
  CALL MPI_FILE_WRITE(fh, buf, 1, etype, status, ierror)
END DO
```

or MPI_MODE_**WRONLY**

or **MPI_CHARACTER**

Chapter 13-(3): Parallel file I/O exercise 3 – shared filepointer

MPI/tasks/C/Ch13/solutions/mpi_io_exa3.c

C

```
~~MPI_Datatype etype;~~
~~MPI_Datatype filetype;~~
~~MPI_Offset disp;~~
~~   etype = MPI_CHAR;~~
~~   ndims = 1;~~
~~   array_of_sizes[0]    = size;~~
~~   array_of_subsizes[0] = 1;~~
~~   array_of_starts[0]   = my_rank;~~
~~   order = MPI_ORDER_C;~~
~~   MPI_Type_create_subarray(ndims, array_of_sizes,~~
~~                            array_of_subsizes, array_of_starts, order, etype, &filetype);~~
~~   MPI_Type_commit(&filetype);~~
~~   disp = 0;~~
~~   MPI_File_set_view(fh, disp, etype, filetype, "native", MPI_INFO_NULL);~~
  MPI_File_open(MPI_COMM_WORLD, "my_test_file",
                MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh);
  for (i=0; i<3; i++) {
    buf = 'a' + (char)my_rank;
    MPI_File_write_ordered(fh, &buf, 1, MPI_CHAR, &status);
  }
  MPI_File_close(&fh);
```

Fortran

MPI/tasks/F_30/Ch13/solutions/mpi_io_exa3.f

```
TYPE(MPI_Datatype) :: etype
…
  ~~CALL MPI_FILE_OPEN~~(MPI_COMM_WORLD, 'my_test_file', &
     ~~&~~                        IOR(MPI_MODE_RDWR, MPI_MODE_CREATE), &
     &                        MPI_INFO_NULL, fh, ierror)
  DO I=1,3
    buf = CHAR( ICHAR('a') + my_rank )
    CALL MPI_FILE_WRITE_ORDERED(fh, buf, 1, MPI_CHARACTER, status, ierror)
  END DO
  CALL MPI_FILE_CLOSE(fh, ierror)
```

Thanks!

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union’s Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro