Data Layout and the Describing Datatype Handle

```
struct buff_layout
    { int        i_val[3];
      double d_val[5];
  } buffer;
```

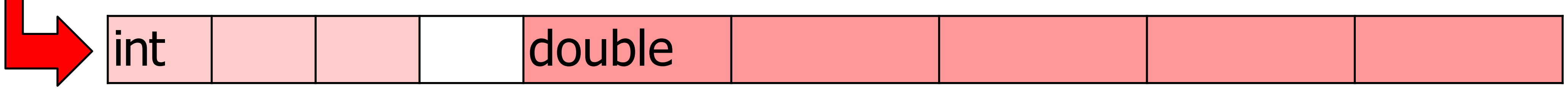
Compiler

```
array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=0;
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=…;
MPI_Type_create_struct(2, array_of_blocklengths,
    array_of_displacements, array_of_types,
                &buff_datatype);
MPI_Type_commit(&buff_datatype);
```

```
MPI_Send(&buffer, 1, buff_datatype, …)
```

&buffer = the start address of the data

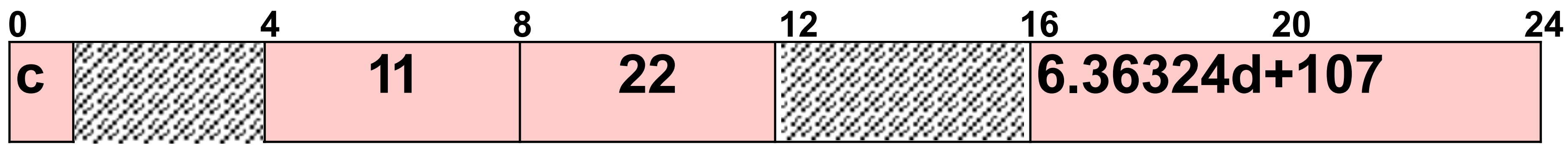
the datatype handle describes the data layout



Derived Datatypes — Type Maps

- A derived datatype is logically a pointer to a list of entries:
  - *basic datatype* at *displacement*

Example:



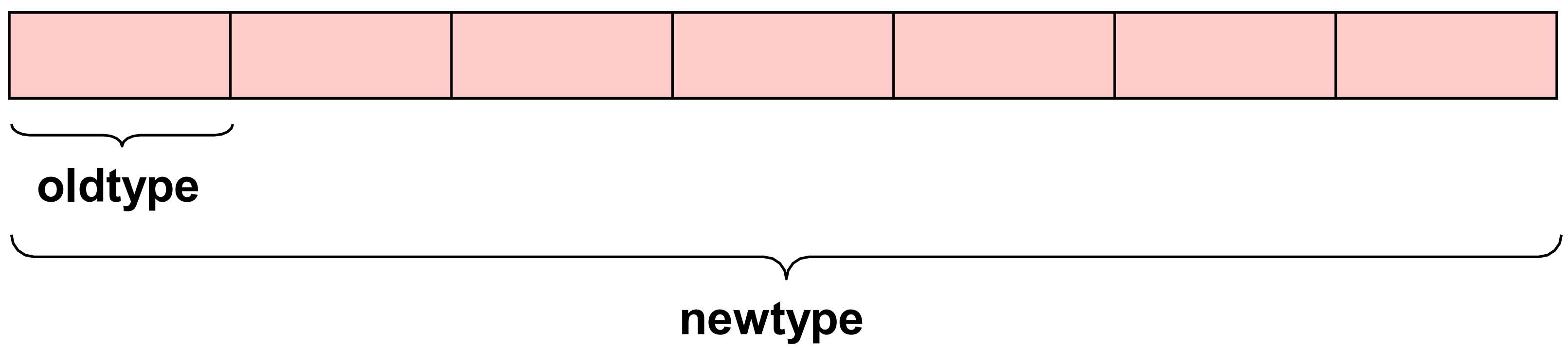
derived datatype handle

basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

A derived datatype describes the memory layout of, e.g., structures, common blocks, subarrays, some variables in the memory

Contiguous Data

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype



C

- C/C++: int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype*)

Python

- Python: *newtype* = oldtype.Create_contiguous(int count)

Committing and Freeing a Datatype

- Before a datatype handle is used in message passing communication, **it needs to be committed with MPI_TYPE_COMMIT**.

- This need be done only once (by each MPI process).
(Using more than once ☹ corresponds to additional no-operations.)

C

- C/C++: int MPI_Type_commit(MPI_Datatype *datatype*);

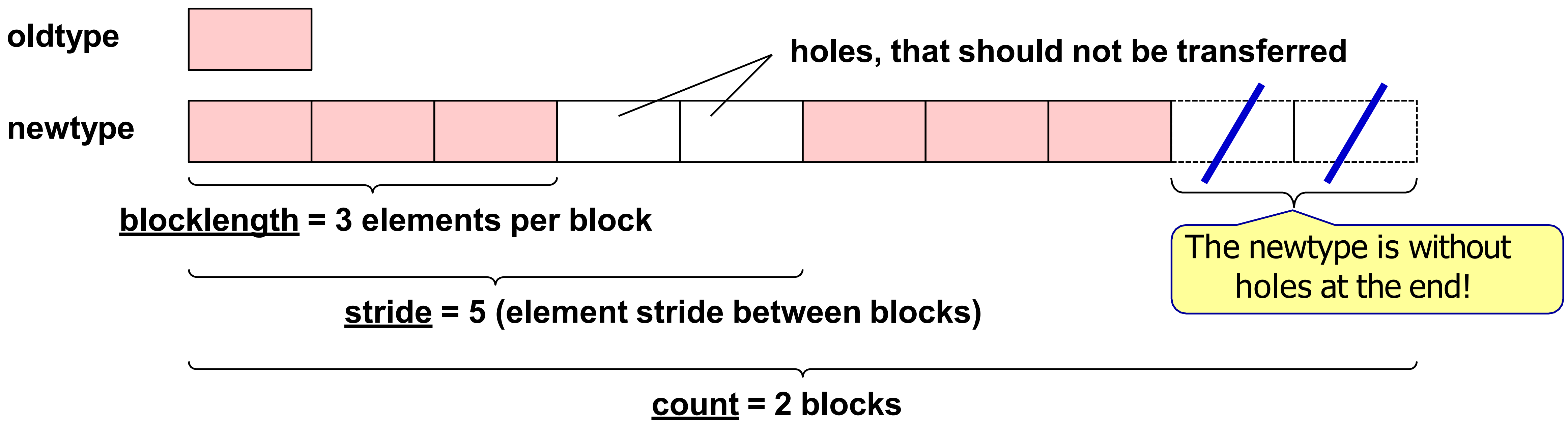
IN-OUT argument
(although handle is not modified)

Python

- Python: datatype.Commit()

- If usage is over, one may call MPI_TYPE_FREE() to free a datatype and its internal resources.

Vector DataType



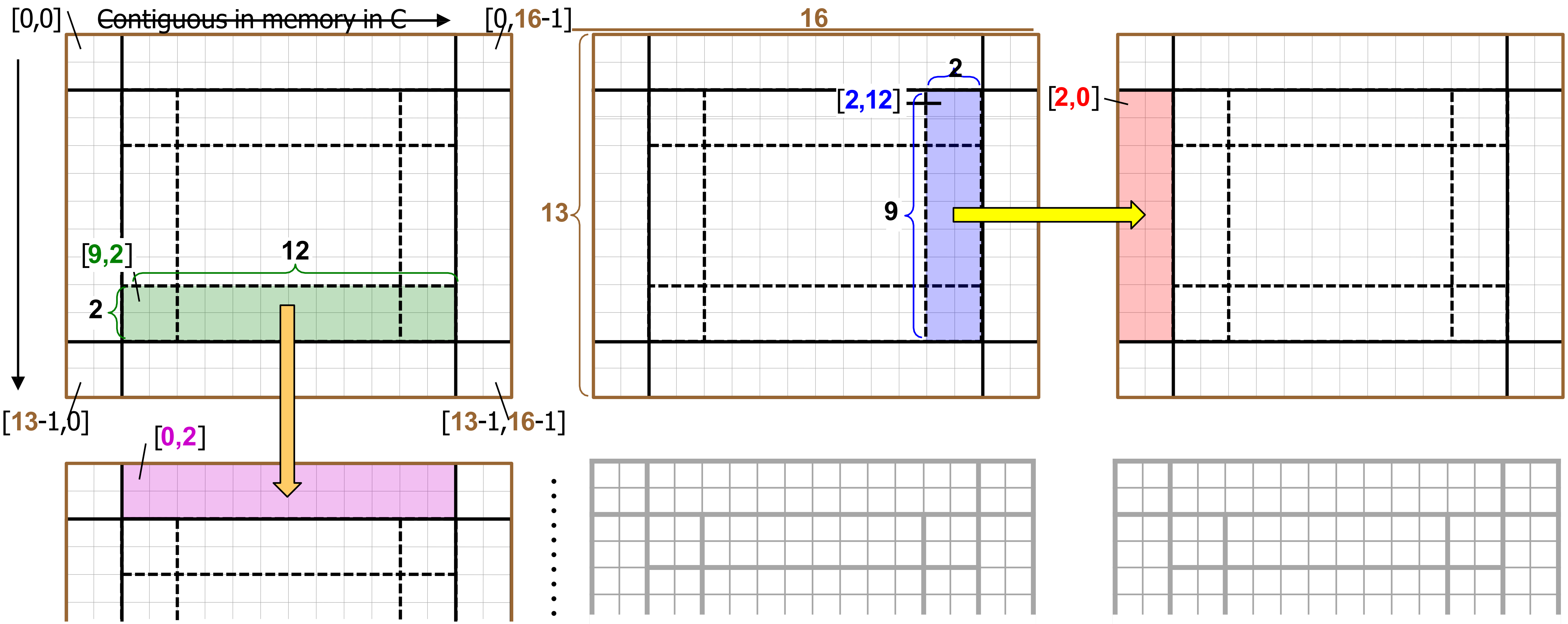
C

- C/C++: int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Python

- Python: newtype = oldtype.Create_vector(int count, int blocklength, int stride)

Example with MPI_Type_vector



C

MPI_Type_vector(**2**, **12**, **16**, etype, &*newt*);
MPI_Send(**a**[**9,2**], 1, newt, ...);

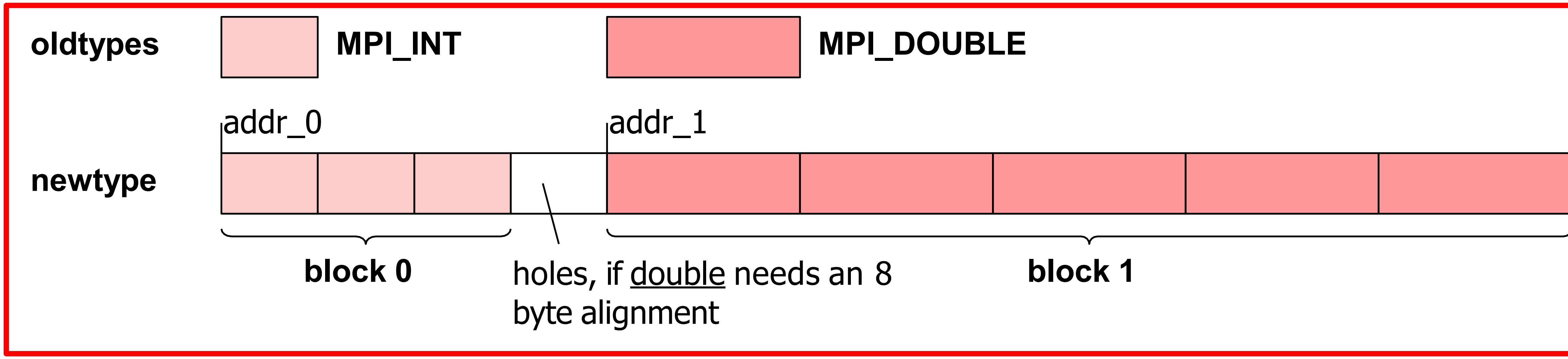
MPI_Type_vector(**2**, **12**, **16**, etype, &*newt*);
MPI_Recv(**a**[**0,2**], 1, newt, ...);

MPI_Type_vector(**9**, **2**, **16**, etype, &*newt*);
MPI_Send(**a**[**2,12**], 1, newt, ...);

MPI_Type_vector(**9**, **2**, **16**, etype, &*newt*);
MPI_Recv(**a**[**2,0**], 1, newt, ...);

Python Same numbering as with C

Struct Datatype



C

Python

- C/C++: int MPI_Type_create_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)
- Python: newtype = MPI.Datatype.Create_struct(array_of_blocklengths, array_of_displacements, array_of_types)

count = 2
array_of_blocklengths = (3, 5)
array_of_displacements = (0, addr_1 – addr_0) [1)]
array_of_types = (MPI_INT, MPI_DOUBLE)

1) Via MPI_Get_address and MPI_Aint_diff, see following slides

How to compute the displacement (1)

- array_of_displacements[i] := address(block_i) – address(block_0)

Retrieve an absolute address:

C

Python

- C/C++: int MPI_**Get_**address(void* location, MPI_Aint **address*)
- Python: *address* = MPI.**Get_**address(location)

How to compute the displacement (2)

New in MPI-3.1

Relative displacement:= absolute address 1 – absolute address 2

C

Python

- C/C++: *MPI_Aint* MPI_Aint_diff(MPI_Aint addr1, MPI_Aint *addr2*)
- Python: *int* MPI.Aint_diff(addr1, *addr2*)

Python's int allows 64 bit

New in MPI-3.1

New absolute address := existing absolute address + relative displacement:

C

Python

- C/C++: *MPI_Aint* MPI_Aint_add(MPI_Aint base, MPI_Aint *disp*)
- Python: *int* MPI.Aint_add(base, *disp*)

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type MPI_Aint and result in unexpected values on some platforms. The MPI_AINT_ADD and MPI_AINT_DIFF functions can be used to safely perform address arithmetic with MPI_Aint displacements. (*End of advice to users.*)

Example for
array_of_displacements[i] := address(block_i) – address(block_0)

C

```
struct buff
   {       int i[3];
           double d[5];
   }snd_buf;
MPI_Aint iaddr0, iaddr1, disp;
MPI_Get_address( &snd_buf.i[0], &iaddr0);  // the address value &snd_buf.i[0] is stored into variable iaddr0
MPI_Get_address(&snd_buf.d[0], &iaddr1);
disp = MPI_Aint_diff(iaddr1, iaddr0);                 // MPI-3.0 & former: disp = iaddr1–iaddr0
```

New in MPI-3.1

Python

```
np_dtype = np.dtype([('i', np.intc, 3), ('d', np.double, 4)])
snd_buf = np.empty((),dtype=np_dtype)
addr0 = MPI.Get_address(snd_buf['i'])
addr1 = MPI.Get_address(snd_buf['d'])
disp = MPI.Aint_diff(addr1, addr0)
```

Scope & Performance options

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods

Which is the fastest neighbor communication with strided data?

- **Copying** the strided data in a contiguous scratch send-buffer,
  communicating this send-buffer into a contiguous recv-buffer,
  and copying the recv-buffer back into the strided application array
- Using derived datatype handles
- And which of the communication routines should be used?

No answer by the MPI standard, because:

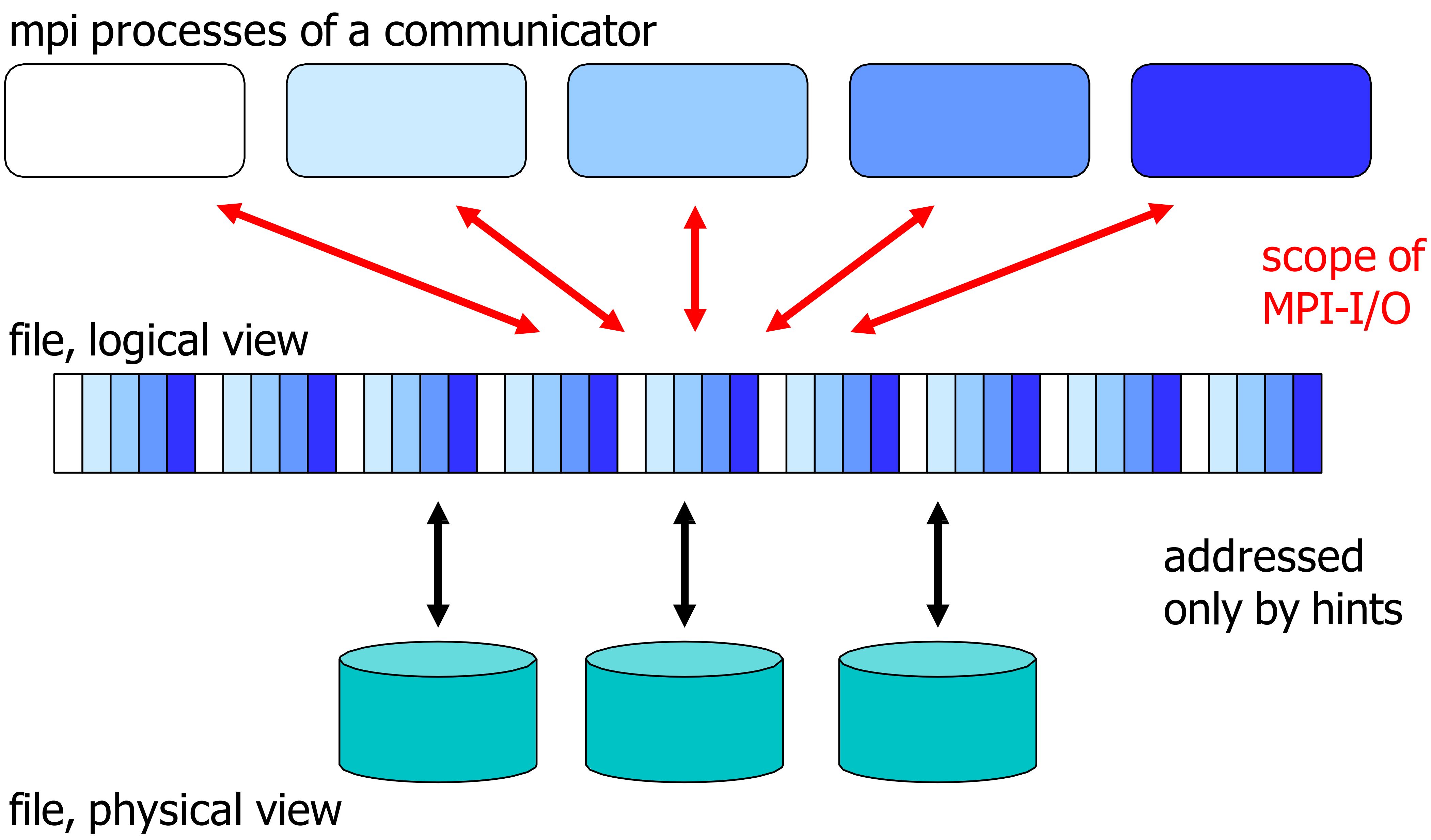
MPI targets portable and efficient message-passing programming
but
efficiency of MPI application-programming is not portable!

Parallel I/O

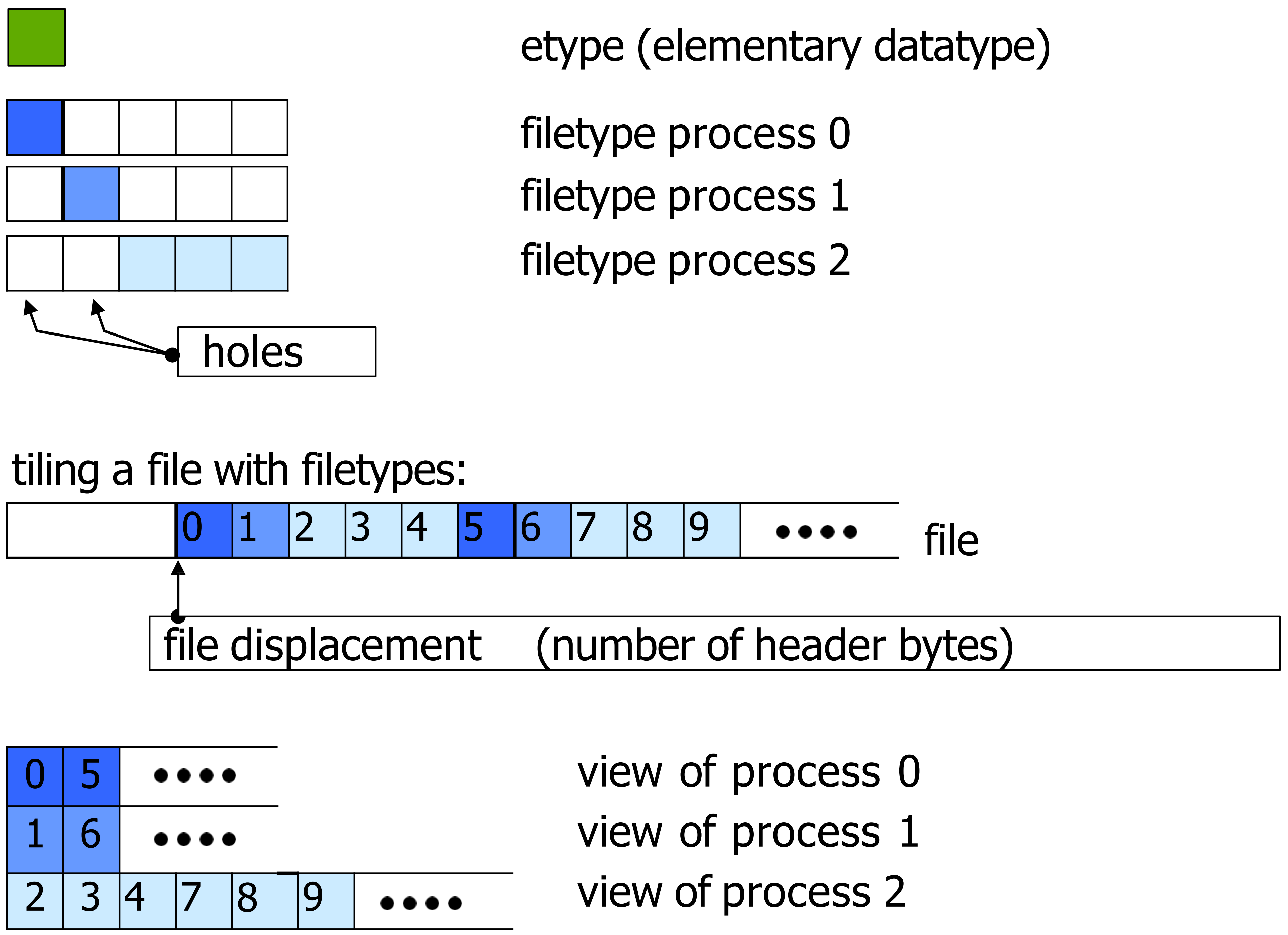
MPI-I/O Features

- 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

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
  filemon 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*]

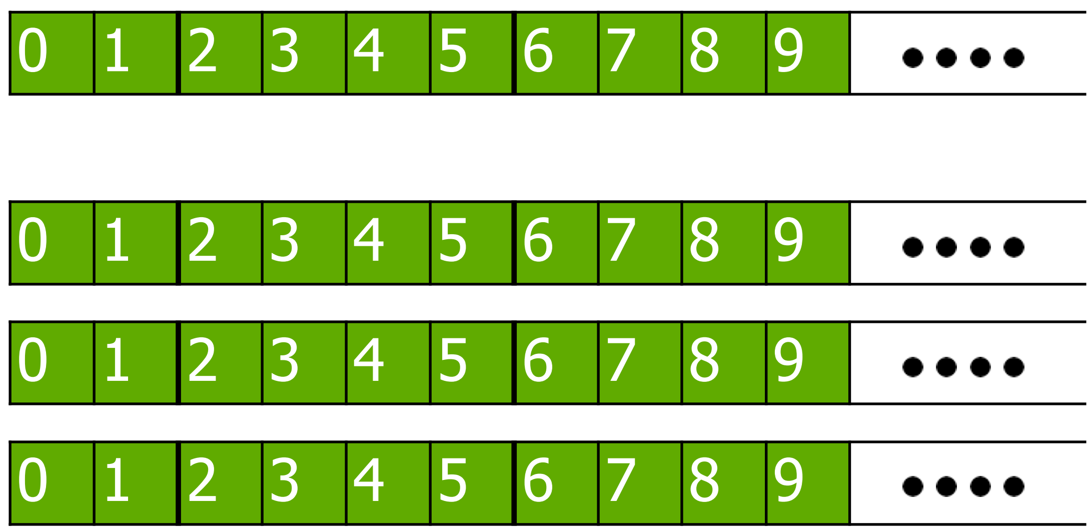
MPI_File_open(comm, filename, amode, info, *fh*)

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



file

view of process 0

view of process 1

view of process 2

- Sequence of MPI_BYTE matches with any datatype
- Binary I/O (no ASCII text I/O)

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*]

Access Modes

- same value of **`amode`** on all processes in **MPI_File_open**
- Bit vector OR of integer constants
  - 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 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

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

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

Reading with 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

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

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*]

Shared File Pointer

- 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)
```

```
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

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

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"]

Scenery – Nonblocking or Split Collective

- Scenery E:
  - 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 finished, then distributing the data with bcast*]