Access pattern 3: Definitions

- **File**
  - Ordered collection of typed data items
  - Opened collectively by a group of processes
- **Displacement**
  - Absolute byte position relative to the beginning of a file
  - Defines the location where a *view* begins
- ***etype* (elementary datatype)**
  - Can be a predefined or derived datatype
  - Unit of data access and positioning
  - Offsets are expressed as multiples of etypes

Access pattern 3: Definitions

- **Filetype**
  - Defines a template/pattern for a process to access the file (based on etype)
- **View**
  - View is the set of data visible to a process in a file, defined by displacement, etype and filetype
  - Different process can have different views
  - Pattern defined by filetype is repeated (in units of etypes) beginning at the displacement

Access pattern 3: File Views

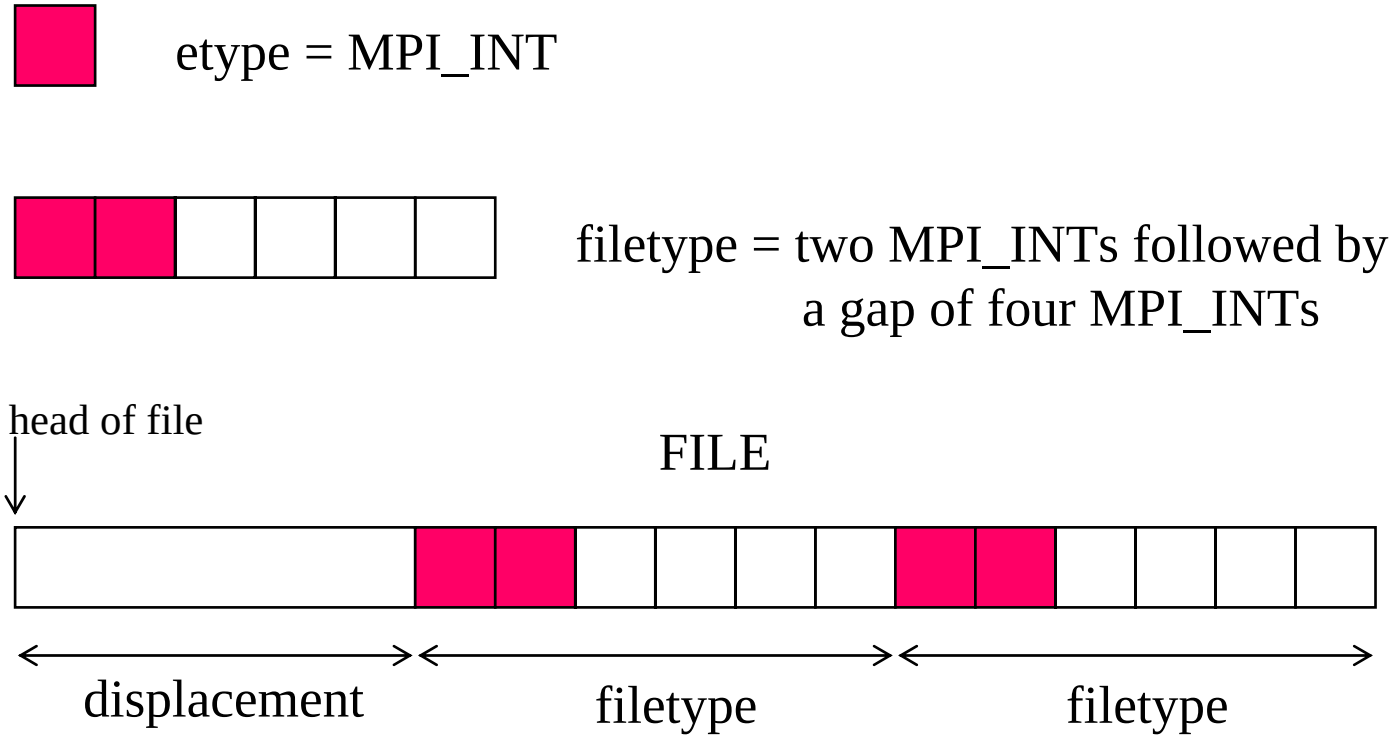
Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to `MPI_File_set_view`

***displacement* = number of bytes to be skipped from the start of the file**

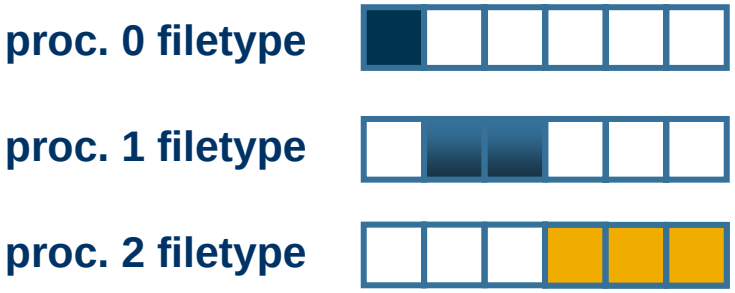
***etype* = basic unit of data access (can be any basic or derived datatype)**

***filetype* = specifies the pattern in a file accessbile to a process**

Access pattern 3: A Simple Noncontiguous File View Example



Access pattern 3: How do *views* relate to multiple processes?



Group of processes using complementary views to achieve global data distribution

Partitioning a file among parallel processes

MPI_File_set_view

Describes that part of the file accessed by a single MPI process.

```
int MPI_File_set_view( MPI_File mpi_fh, MPI_Offset disp,
MPI_Datatype etype, MPI_Datatype filetype, char *datarep,
MPI_Info info );
```

Parameters

mpi_fh : **[in] file handle (handle)**

disp : **[in] displacement (nonnegative integer)**

etype : **[in] elementary datatype (handle)**

filetype : **[in] filetype (handle)**

datarep : **[in] data representation (string)**

info : **[in] info object (handle)**

Access pattern 3: File View Example

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
                  MPI_MODE_CREATE | MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
                     MPI_INT, MPI_INT, "native",
                   MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
                   MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```

MPI_Type_create_subarray

Create a datatype for a subarray of a regular, multidimensional array

```
int MPI_Type_create_subarray( int ndims, int array_of_sizes[], int
array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype
oldtype, MPI_Datatype *newtype );
```

Parameters

ndims :[in] number of array dimensions (positive integer)

array_of_sizes :[in] number of elements of type oldtype in each dimension of the full array (array of positive integers)

array_of_subsizes :[in] number of elements of type oldtype in each dimension of the subarray (array of positive integers)

array_of_starts :[in] starting coordinates of the subarray in each dimension (array of nonnegative integers)

order :[in] array storage order flag (state)

oldtype :[in] array element datatype (handle)

newtype :[out] new datatype (handle)

Using the Subarray Datatype

```
gsizes[0] = 16;  /* no. of rows in global array */
gsizes[1] = 16;  /* no. of columns in global array*/

psizes[0] = 4; /* no. of procs. in vertical dimension */
psizes[1] = 4; /* no. of procs. in horizontal dimension */

lsizes[0] = 16/psizes[0]; /* no. of rows in local array */
lsizes[1] = 16/psizes[1]; /* no. of columns in local array*/

dims[0] = 4; dims[1] = 4;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
```

Subarray Datatype contd.

```
/* global indices of first element of local array */
start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];

MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                         MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);
```

Access pattern 3

- Each process creates a derived datatype to describe the non-contiguous access pattern
- We thus have a *file view* and independent access

```
MPI_Type_create_subarray
                    (… , &subarray, …)
MPI_Type_commit (&subarray)
MPI_File_open(… , “filename”, … , &fh)
MPI_File_set_view (fh, disp, … , subarray, …)
MPI_File_read (fh, local_array, 1, subarray,…)
MPI_File_close (&fh)
```

- Creates a datatype describing a subarray of a multi-dimentional array
- Commits the datatype (must be done before comms)
- Opens the file as before

16 independent, non-contiguous requests

Access pattern 3

- Each process creates a derived datatype to describe the non-contiguous access pattern
- We thus have a *file view* and independent access

```
MPI_Type_create_subarray
                    (… , &subarray, …)
MPI_Type_commit (&subarray)
MPI_File_open(… , “filename”, … , &fh)
MPI_File_set_view (fh, disp, … , subarray, …)
MPI_File_read (fh, local_array, 1, subarray,…)
MPI_File_close (&fh)
```

- Now changes the processes view of the data in the file using *set_view*
- *set_view* is collective
- Although the reads are still independent

16 independent, non-contiguous requests

Access pattern 4

- Each process creates a derived datatype to describe the non-contiguous access pattern
- We thus have a *file view* and collective access

```
MPI_Type_create_subarray
                    (… , &subarray, …)
MPI_Type_commit (&subarray)
MPI_File_open(… , “filename”, … , &fh)
MPI_File_set_view (fh, … , subarray, …)
MPI_File_read_all (fh, local_array, …)
MPI_File_close (&fh)
```

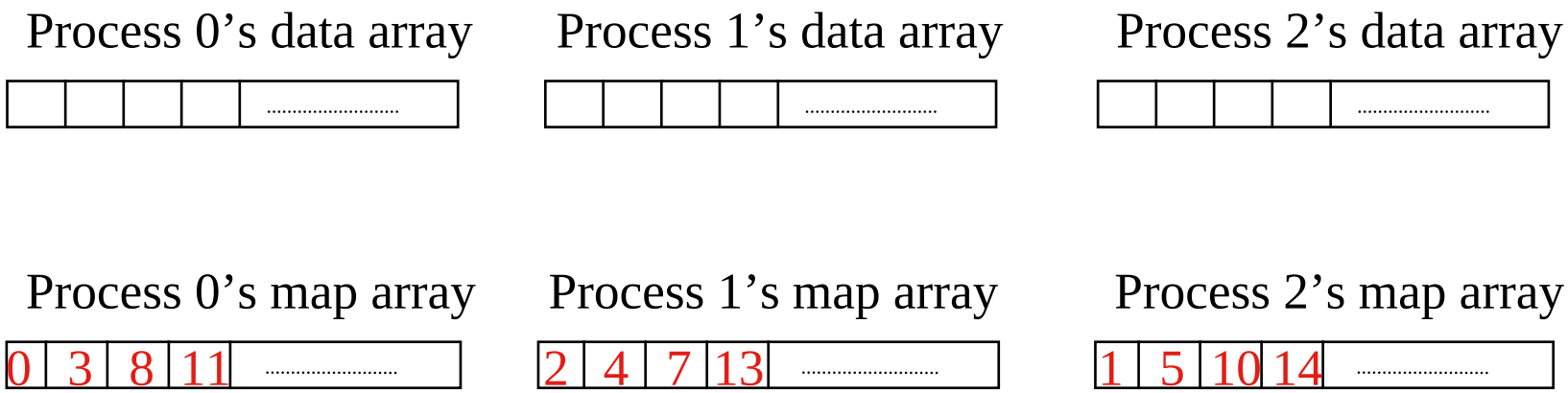
Single collective, non-contiguous request

- Creates and commits datatype as before
- Now changes the processes view of the data in the file using *set_view*
- *set_view* is collective
- Reads are now collective

Access patterns

- We discussed four different style of parallel I/O
- You should choose your access pattern depending on the application
- *Larger the size of the I/O request, the better performance*
- Collectives are going to do better than individual reads
- Pattern 4 therefore offers (potentially) the best performance

Accessing Irregularly Distributed Arrays



The map array describes the location of each element of the data array in the common file

Accessing Irregularly Distributed Arrays

int MPI_Type_create_indexed_block(int *count*, int *blocklength*, int *array_of_displacements*[], MPI_Datatype *oldtype*, MPI_Datatype **newtype*);

Parameters:

count

- **[in] length of array of displacements (integer)**

blocklength

- **[in] size of block (integer)**

array_of_displacements

- **[in] array of displacements (array of integer)**

oldtype

- **[in] old datatype (handle)**

newtype

- **[out] new datatype (handle)**

File Info

```
int MPI_File_set_view( MPI_File mpi_fh, MPI_Offset
disp, MPI_Datatype etype, MPI_Datatype filetype, char
*datarep, MPI_Info info );
```

MPI_Info allows a user to provide hints to the MPI system about file access patterns and file system details to direct optimization.

→MPI_Info is a (key, value) pair

→For example,

- (access_style, read_once)
- (striping_unit, 512)

→Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources.

→an implementation is free to ignore all hints.

→`MPI_File_set_info(MPI_File mpi_fh, MPI_Info info );`

Data Access Routines

- There are three aspects to data access:
  - positioning (explicit offset *vs.* implicit file pointer),
  - synchronism (blocking *vs.* nonblocking),
  - coordination (noncollective *vs.* collective)

Positioning

→three types of positioning for data access routines:

- explicit offsets,
- individual file pointers,
- shared file pointers.

→Three positioning methods do not affect each other

→The data access routines that accept explicit offsets contain _AT in their name

- E.g. `MPI_File_write_at`(MPI_File *mpi_fh*, MPI_Offset *offset*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*);

→The names of the individual file pointer routines contain no positional qualifier

- E.g. `MPI_File_write`(MPI_File *mpi_fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*);

→The data access routines that use shared file pointers contain _SHARED

- E.g. `MPI_File_write_shared`(MPI_File *mpi_fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*);

Open a file collectively

int MPI_File_open(MPI_Comm *comm*, char **filename*, int *amode*, MPI_Info *info*, MPI_File **mpi_fh*);

Opens the file identified by the file name filename on all processes in the comm communicator

All processes share a file pointer

Synchronism

→A *blocking* I/O call will not return until the I/O request is completed

→A *nonblocking* I/O call initiates an I/O operation and then returns immediately

→Format: `MPI_FILE_IXXX(MPI_File` ***`mpi_fh`***`, void` ***`*buf`***`, int` ***`count`***`, MPI_Datatype` ***`datatype`***`, MPI_Request` ***`*request`*** `);`

→Using MPI_WAIT, MPI_TEST or any of their variants to test whether the I/O operation has been completed.

→It is erroneous to access the application buffer of a nonblocking data access operation between the initiation and completion of the operation.

Coordination

→Every non-collective data access routine MPI_File_XXX has a collective counterpart

- Blocking collective version: MPI_File_XXX_all
- Non-blocking collective version: a pair of MPI_File_XXX_all_begin and MPI_File_XXX_all_end

→The counterparts to the MPI_File_XXX_shared routines are `MPI_File_XXX_Ordered(MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status );`

Split Collective I/O

- nonblocking collective I/O
- The begin routine begins the operation, much like a nonblocking data access (e.g., MPI_File_iwrite)
- The end routine completes the operation, much like the matching wait (e.g., MPI_Wait)

```
MPI_File_write_all_begin(fh, buf, count, datatype);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_File_write_all_end(fh, buf, &status);
```

File Interoperability

- file interoperability is the ability to read the information previously written to a file

- MPI supports full interoperability
  - within a single MPI environment
    - file data written by one MPI process can be read and recognized by any other MPI process
  - outside that environment through the external data representation
    - File data can be shared between any two applications, regardless of whether they use MPI, and regardless of the machine architectures on which they run

File Interoperability

Three data representations

- **Native**
  - Data in this representation is stored in a file exactly as it is in memory
  - pros: data precision and I/O performance (used in a MPI homogeneous environment)
  - cons: the loss of transparent interoperability (cannot be used in a heterogeneous MPI environment)
- **Internal**
  - the implementation will perform type conversions if necessary
  - can be used in a homogeneous or heterogeneous environment
  - The data can be recognised by other processes in the MPI environment
- **External32**
  - The data on the storage medium is always in the standard representation (big-endian IEEE format)
  - The data can be recognised by other applications

```
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int), MPI_INT,
MPI_INT, "native", MPI_INFO_NULL);

MPI_File_write(thefile, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
```

I/O Consistency

The consistency semantics specify the results when multiple processes access a common file; among multiple processes, one or more processes write to the file

The user can take steps to ensure consistency when MPI does not automatically do so

Example 1

File opened with `MPI_COMM_WORLD`. Each process writes to a *separate* region of the file and reads back only what it wrote.

Process 0	Process 1
`MPI_File_open(MPI_COMM_WORLD,…)`	`MPI_File_open(MPI_COMM_WORLD,…)`
`MPI_File_write_at(off=0,cnt=100)`	`MPI_File_write_at(off=100,cnt=100)`
`MPI_File_read_at(off=0,cnt=100)`	`MPI_File_read_at(off=100,cnt=100)`

MPI guarantees that the data will be read correctly

Example 2

Same as example 1, except that each process wants to read what the *other* process wrote (overlapping accesses)

In this case, MPI does *not* guarantee that the data will automatically be read correctly

Process 0

```
/* incorrect program */
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=0,cnt=100)
MPI_Barrier
MPI_File_read_at(off=100,cnt=100)
```

Process 1

```
/* incorrect program */
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=100,cnt=100)
MPI_Barrier
MPI_File_read_at(off=0,cnt=100)
```

In the above program, the read on each process is not guaranteed to get the data written by the other process!

Example 2 contd.

The user must take extra steps to ensure correctness

There are three choices:

- **set atomicity to true**
- **close the file and reopen it**
- **Use MPI_File_sync**

Example 2, Option 1 - Set atomicity to true

Process 0

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_set_atomicity(fh1,1)
MPI_File_write_at(off=0,cnt=100)
MPI_Barrier
MPI_File_read_at(off=100,cnt=100)
```

Process 1

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_set_atomicity(fh2,1)
MPI_File_write_at(off=100,cnt=100)
MPI_Barrier
MPI_File_read_at(off=0,cnt=100)
```

Example 2, Option 2 - Close and reopen file

Process 0

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=0,cnt=100)
MPI_File_close
MPI_Barrier
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_read_at(off=100,cnt=100)
```

Process 1

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=100,cnt=100)
MPI_File_close
MPI_Barrier
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_read_at(off=0,cnt=100)
```

Example 2, Option 3 - Using MPI_File_sync

Process 0

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=0,cnt=100)
MPI_File_sync

MPI_Barrier

MPI_File_sync /*collective*/

MPI_File_sync /*collective*/

MPI_Barrier

MPI_File_sync
MPI_File_read_at(off=100,cnt=100)
MPI_File_close
```

Process 1

```
MPI_File_open(MPI_COMM_WORLD,…)

MPI_File_sync /*collective*/

MPI_Barrier

MPI_File_sync
MPI_File_write_at(off=100,cnt=100)
MPI_File_sync

MPI_Barrier

MPI_File_sync /*collective*/
MPI_File_read_at(off=0,cnt=100)
MPI_File_close
```