



## Parallel I/O with MPI IO

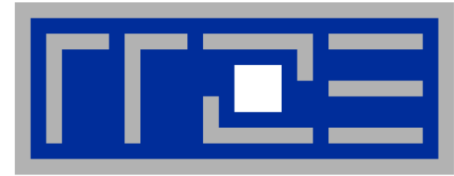
**Georg Hager, Thomas Zeiser,  
Gerhard Wellein, Markus Wittmann  
(RRZE)**

`hpc@rrze.fau.de`

**A. Skjellum, P. Bangalore, S. Herbert,  
R. Rabenseifner**



- **MPI derived data types**
- **MPI I/O**
  - File open and closing
  - Views
  - Reading and writing
- **Examples & Use Cases**



# Repetition: Derived Data Types in MPI



### Root reads configuration and broadcasts it to all others

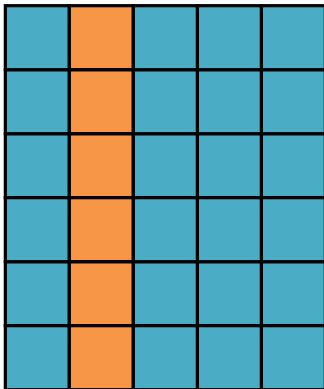
```
// root: read configuration from
// file into struct config
MPI_Bcast(&cfg.nx, 1, MPI_INT, ...);
MPI_Bcast(&cfg.ny, 1, MPI_INT, ...);
MPI_Bcast(&cfg.du, 1, MPI_DOUBLE, ...);
MPI_Bcast(&cfg.it, 1, MPI_INT, ...);
```

`MPI_Bcast(&cfg, sizeof(cfg), MPI_BYTE, ...)`

is **not** a solution. Its not portable as no data conversion can take place



```
MPI_Bcast(
    &cfg, 1, <type cfg>, ...);
```



### Send column of matrix (noncontiguous in C):

- Send each element alone?
- Manually copy elements out into a contiguous buffer and send it?

- Create in three steps

- **Construct** with

```
MPI_Type*
```

- **Commit** new data type with

```
MPI_Type_commit(MPI_Datatype * nt)
```

- After use, **deallocate** the data type with

```
MPI_Type_free(MPI_Datatype * nt)
```

- Create vector-like data type

```
MPI_Type_vector(count, int blocklength, int stride,
                MPI_Datatype oldtype,
                MPI_Datatype * newtype)
```

count            2

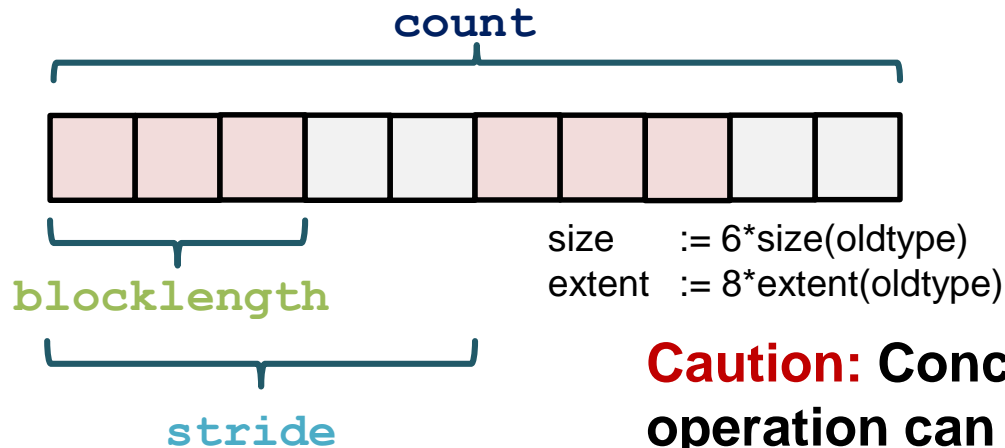
blocklength    3

stride           5

oldtype           MPI\_INT



```
MPI_Datatype nt;
MPI_Type_vector(
    2, 3, 5,
    MPI_INT, &nt);
```



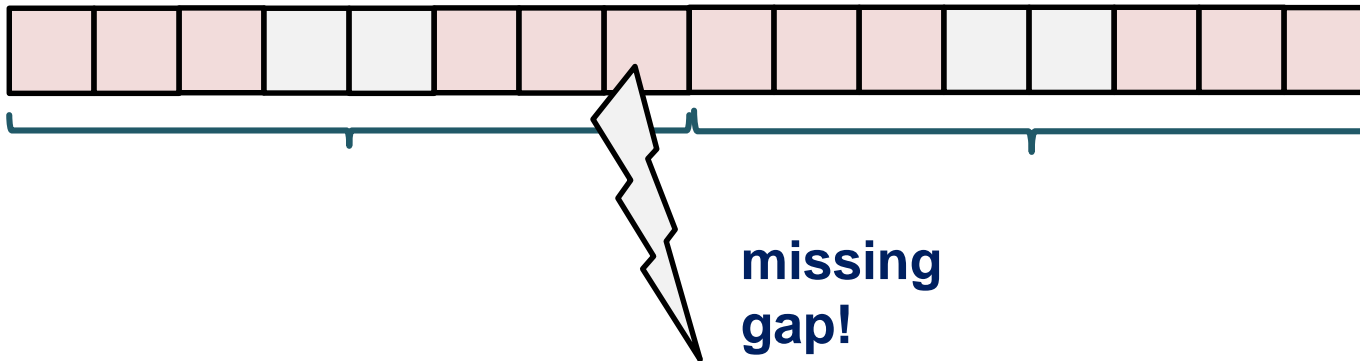
```
MPI_Type_commit(&nt);
// use nt...
MPI_Type_free(&nt);
```

**Caution:** Concatenating such types in a SEND operation can lead to unexpected results!

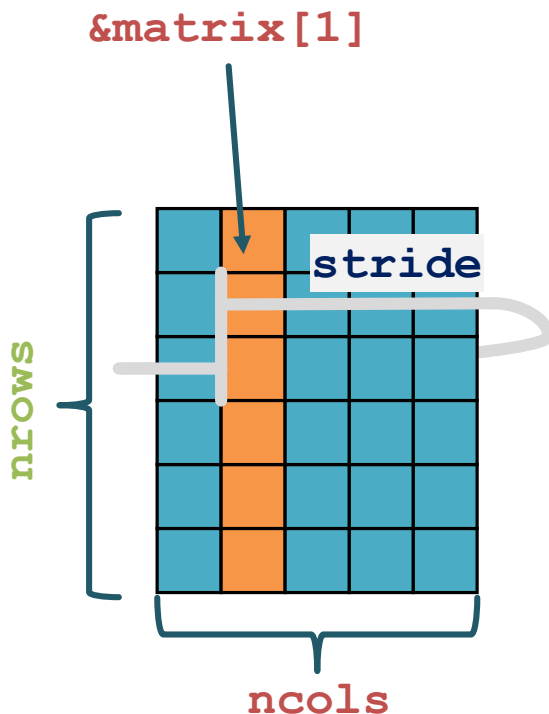
See Sec. 3.12.3 and 3.12.5 of the MPI 1.1 Standard for details.

- count argument to `send` and others must be handled with care:

`MPI_Send(buf, 2, nt, ...)` with `nt` (newtype from prev. slide)



- Create data type describing one column of a matrix
  - assuming row-major layout like in C



```
double matrix[30]
MPI_Datatype nt;

// count = nrows, blocklength = 1,
// stride = ncols
MPI_Type_vector(nrows, 1, ncols,
               MPI_DOUBLE, &nt);
MPI_Type_commit(&nt);

// send column
MPI_Send(&matrix[1], 1, nt, ...);

MPI_Type_free(&nt);
```



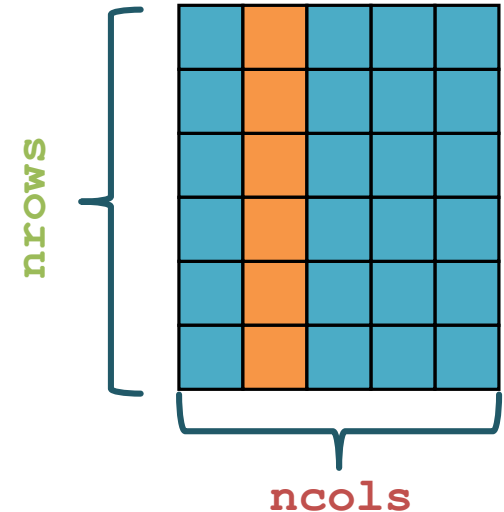
- **Create sub array data type**

```
MPI_Type_create_subarray(int dims,  
int ar_sizes[], int ar_subsizes[], int ar_starts[],  
int order, MPI_Datatype oldtype, MPI_Datatype * newtype)
```

- **dims:** dimension of the array
- **ar\_sizes:** array with sizes of array (dims entries)
- **ar\_subsizes:** array with sizes of subarray (dims entries)
- **ar\_starts:** start indices of the subarray inside array (dims entries), start at 0 (also in Fortran)
- **order**
  - **row-major:** MPI\_ORDER\_C
  - **column-major:** MPI\_ORDER\_FORTRAN
- **oldtype:** data type the array consist of
- **newtype:** data type describing a subarray

```
dims           2
ar_sizes       {ncols, nrows}
ar_subsizes    {1, nrows}
ar_starts      {1, 0}
order          MPI_ORDER_C
oldtype        MPI_DOUBLE
```

assuming row-major layout



```
MPI_Type_create_subarray(dims, ar_sizes, ar_subsizes,
ar_starts, order, oldtype, &nt)
```

```
MPI_Type_commit(&nt);
// use nt...
MPI_Type_free(&nt);
```

```

dims           1
ar_sizes       {5}
ar_subsizes    {3}
ar_starts      {0}
order          MPI_ORDER_C
oldtype        MPI_INT

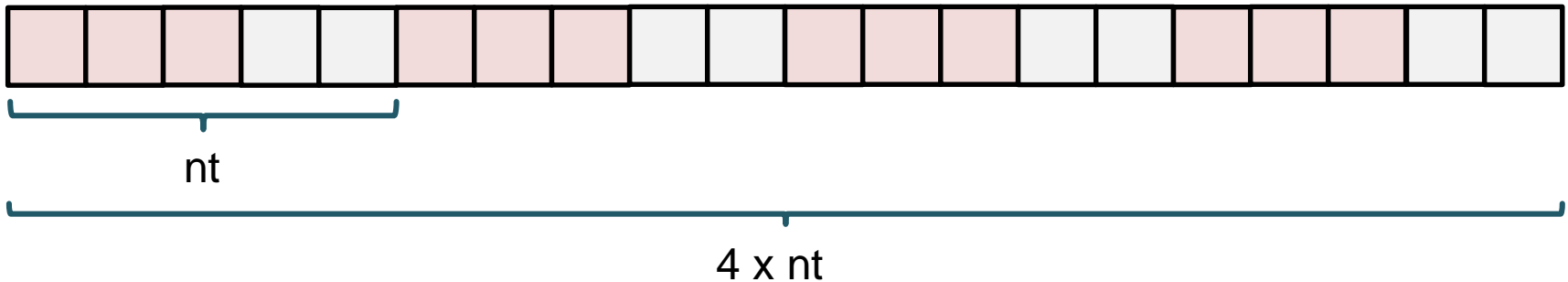
```



```

MPI_Type_create_subarray(dims, ar_sizes, ar_subsizes, ar_starts,
order, oldtype, &nt)
// commit type...
MPI_Send(buf, 4, nt, ...)

```



- Most general type constructor

- Describe blocks with arbitrary data types and arbitrary displacements

count = 2

types[0]



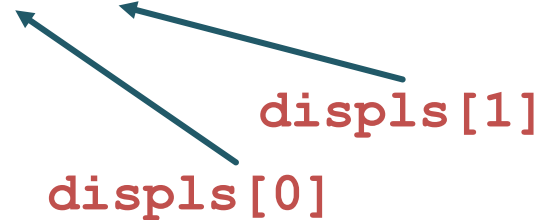
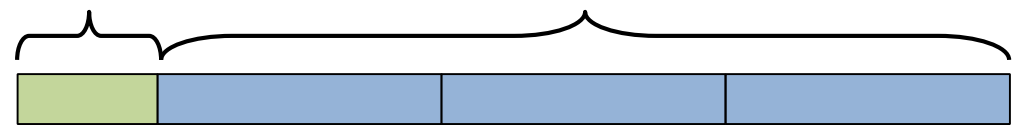
types[1]



```
MPI_Type_create_struct(count,
    int block_lengths[],
    MPI_Aint displs[],
    MPI_Datatype types[],
    MPI_Datatype * newtype)
```

block\_lengths[0]=1

block\_lengths[1]=3



The contents of **displs** are either the displacements in **bytes** of the block bases or **MPI addresses**

- What about displacements in Fortran?

```
MPI_GET_ADDRESS(location, address, ierror)
<type> location
INTEGER(KIND=MPI_ADDRESS_KIND) address
```

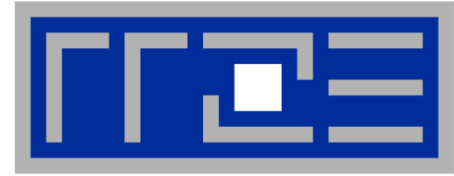
- Example:

```
double precision a(100)
integer a1, a2, disp
call MPI_GET_ADDRESS(a(1), a1, ierror)
call MPI_GET_ADDRESS(a(50), a2, ierror)
disp=a2-a1
```

Result would usually be `disp = 392 (49 x 8)`

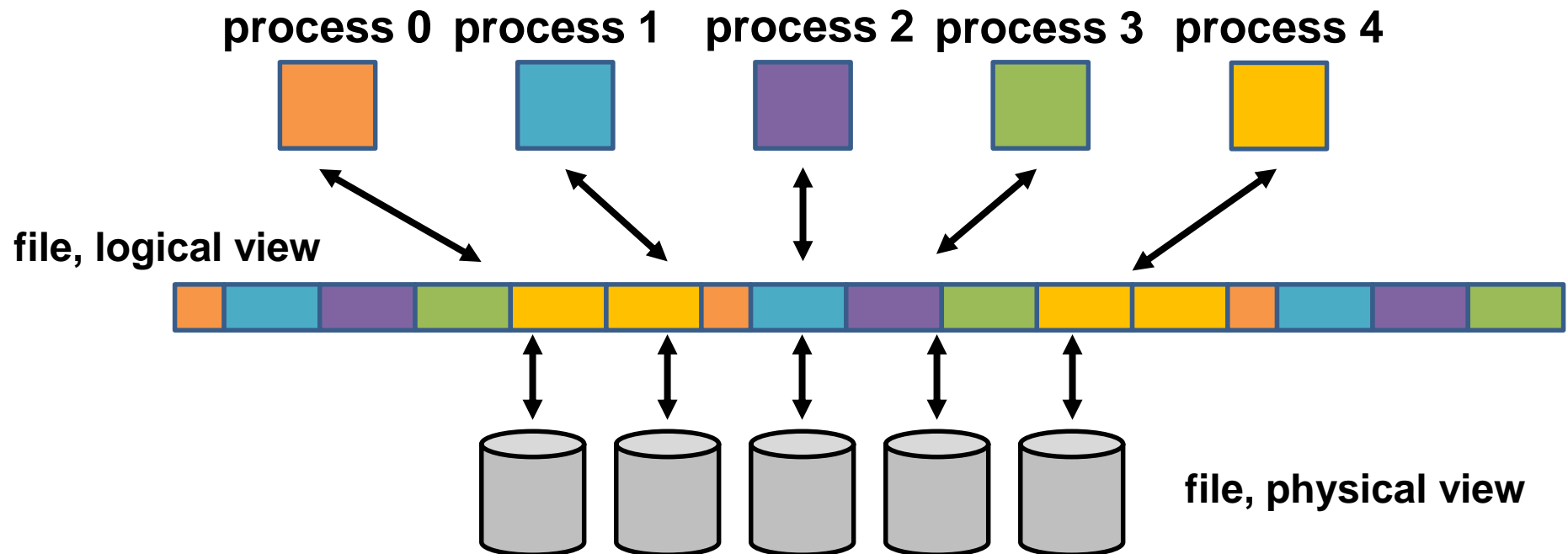
- When using absolute addresses, set buffer address = `MPI_BOTTOM`

- **Derived data types provide a flexible tool to communicate complex data structures in an MPI environment**
- **Most important calls:**
  - `MPI_Type_vector` (second simplest)
  - `MPI_Type_create_subarray`
  - `MPI_Type_create_struct` (most advanced)
  - `MPI_Type_commit/MPI_Type_free`
  - `MPI_GET_ADDRESS`
- **Other useful features:**
  - `MPI_Type_contiguous`, `MPI_Type_indexed`,  
`MPI_Type_get_extent`, `MPI_Type_size`
- **Matching rule: send and receive match if specified basic datatypes match one by one, regardless of displacements**
  - Correct displacements at receiver side are automatically matched to the corresponding data items



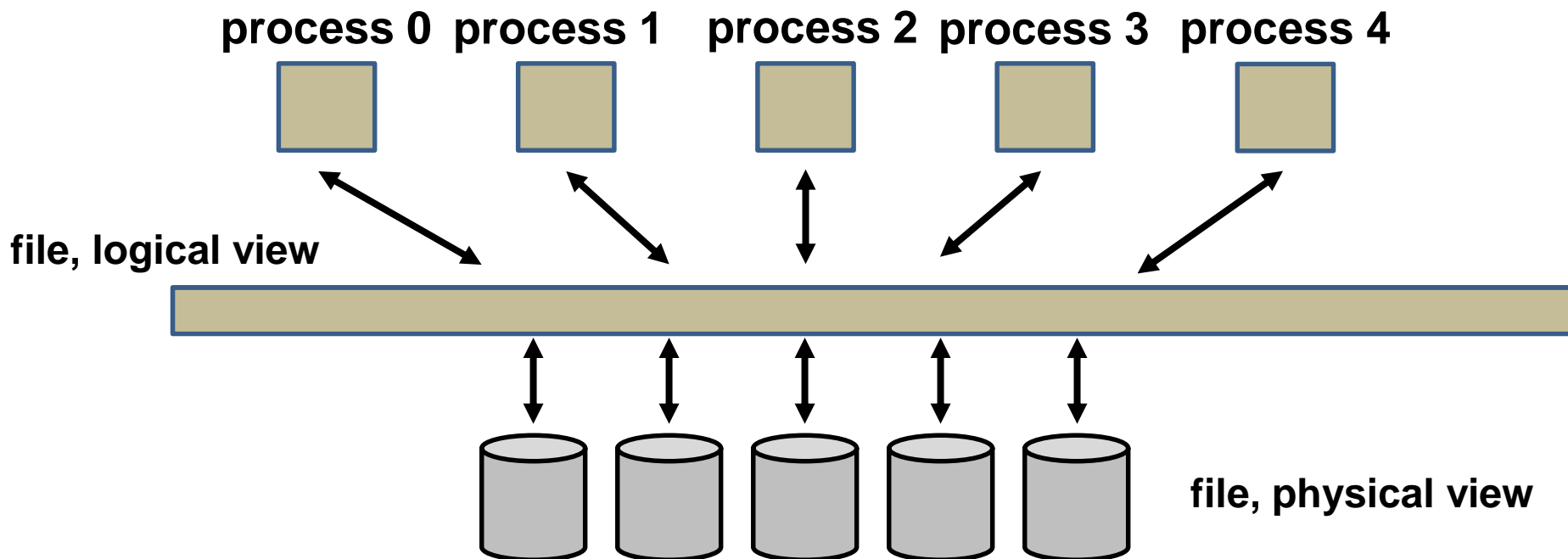
# MPI Input/Output

- **Many parallel applications need ...**
  - Coordinated parallel access to a file by a group of processes
  - Simultaneous access
  - All processes read/write many non-contiguous pieces of the file
  - i.e. the data may be distributed amongst the processes according to a partitioning scheme





- **Many parallel applications need ...**
  - 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





- 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 data types** are used to specify common **data access patterns** for maximum flexibility and expressiveness



- MPI file contains elements of a single MPI data type (**etype**)
- The file is partitioned among processes using an access template (**filetype**)
- All file accesses transfer to/from a contiguous or non-contiguous user buffer (**MPI data type**)
- Several different ways of reading/writing data:
  - non-blocking / blocking
  - collective / individual
  - individual / shared file pointers, explicit offsets
- Automatic **data conversion** in heterogeneous systems
- File **interoperability** with external representation

# Open & Closing Files



```
int MPI_File_open(  
    MPI_Comm comm, const char *filename, int amode,  
    MPI_Info info, MPI_File *fh)
```

- **Collective** call by all processes which are part of `comm`
- `filename` can be different, but must point to the same file
- `amode` describes access mode (see next slide)
- `info` object, can be `MPI_INFO_NULL` (see later)
- `fh` represents the file handle, associated to it is also `comm` and the view (see later)
  
- **Process local file I/O** is possible by specifying `MPI_COMM_SELF` as `comm`

access mode	description
<code>MPI_MODE_RDONLY</code>	read only
<code>MPI_MODE_RDWR</code>	read and write
<code>MPI_MODE_WRONLY</code>	write only
	} one of these flags is required
<code>MPI_MODE_CREATE</code>	create if it does not exist
<code>MPI_MODE_EXCL</code>	error if file exists
<code>MPI_MODE_DELETE_ON_CLOSE</code>	file is deleted when closed
<code>MPI_MODE_UNIQUE_OPEN</code>	file is not concurrently opened by anybody else
<code>MPI_MODE_SEQUENTIAL</code>	only sequential access will occur ( <code>MPI_File_read/write_shared</code> is allowed)
<code>MPI_MODE_APPEND</code>	all file pointers are located at the end of the file

- **Flags can be or'ed together: `MPI_MODE_WRONLY` | `MPI_MODE_APPEND`**
- **Use or function in Fortran**



All processes in MPI\_COMM\_WORLD open file collectively

```
MPI_File fh;
...
MPI_File_open(MPI_COMM_WORLD, filename,
              MPI_MODE_WRONLY | MPI_MODE_CREATE,
              MPI_INFO_NULL, &fh);
...
```

Also possible to open file with only one process

```
if (rank == 0) {
    MPI_File fh;
    ...
    MPI_File_open(MPI_COMM_SELF, filename,
                  MPI_MODE_WRONLY | MPI_MODE_CREATE,
                  MPI_INFO_NULL, &fh);
    ...
}
```

```
int MPI_File_close(MPI_File *fh)
```

- **Collective** call by all processes which are part of `comm`, the file was opened with
- File state is synchronized, i.e. all data is transferred to disk storage
- File handle `fh` is set to `MPI_FILE_NULL`
- File is deleted if `MPI_MODE_DELETE_ON_CLOSE` was part of access mode
- All outstanding nonblocking requests & split collectives must have been completed

```
MPI_File fh;
```

```
...
```

```
MPI_File_open(MPI_COMM_WORLD, ..., &fh);
```

```
...
```

```
MPI_File_close(&fh);
```



---

# Info Objects



- **Opaque object, storing key/value pairs**
- **Often used to provide system-specific information**
  - via `info` argument in function calls
  - for MPI IO, process management, memory allocation, ...
- **Keys**
  - All keys might be ignored
  - MPI defines a set of reserved keys
  - Implementations may provide additional keys
- **Keys/values are strings and converted to other types as required**
- **Use `MPI_INFO_NULL` if you do not want to provide additional information**

```
MPI_Info info;
```

- **Generate new, empty info object:**

```
int MPI_Info_create(MPI_Info *info)
```

- **Add entry to existing info object:**

```
int MPI_Info_set(MPI_Info info,  
                const char *key, const char *value)
```



- **Delete entry from info object**

```
int MPI_Info_delete(MPI_Info info, const char *key)
```

- **Retrieve value associated with key**

```
int MPI_Info_get(MPI_Info info, const char *key,  
                int valuelen, char *value, int *flag)
```

- **flag = true:** a value is associated with the key and returned in **value**
- **flag = false:** no value associated with the key, **value** is unchanged
- **valuelen:** size of the buffer **value** points to,  
if associated value is larger, data is truncated

- **Free info object**

```
int MPI_Info_free(MPI_Info *info)
```

- **Length restriction:**

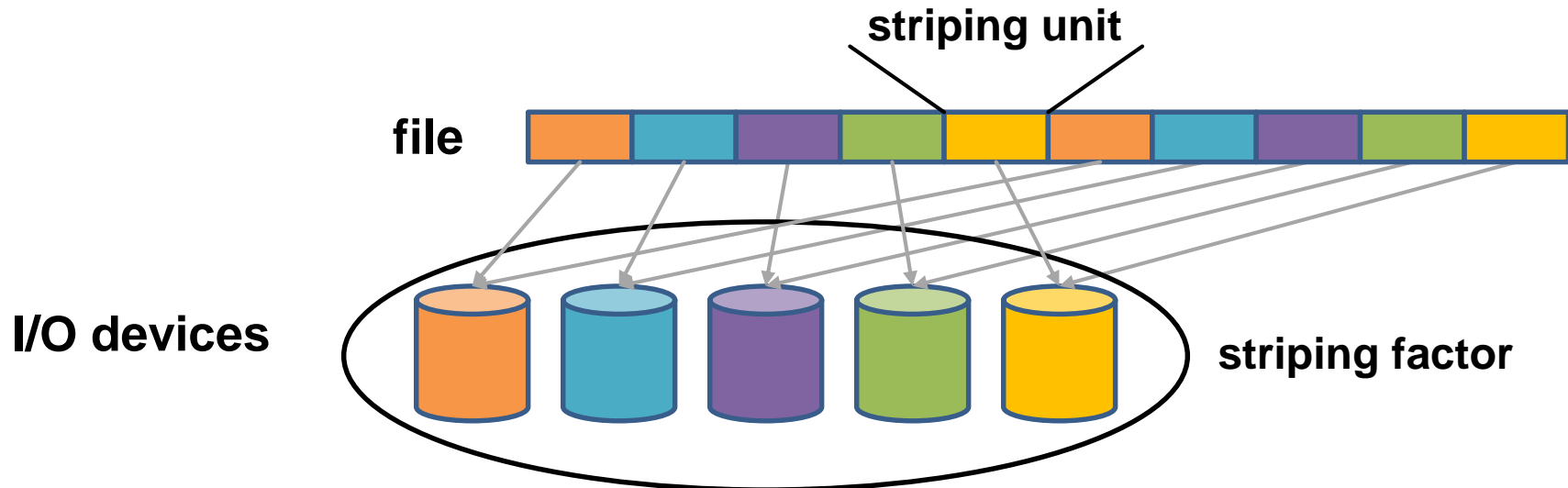
- keys: **MPI\_MAX\_INFO\_KEY**
- values: **MPI\_MAX\_INFO\_VAL**

### Striping:

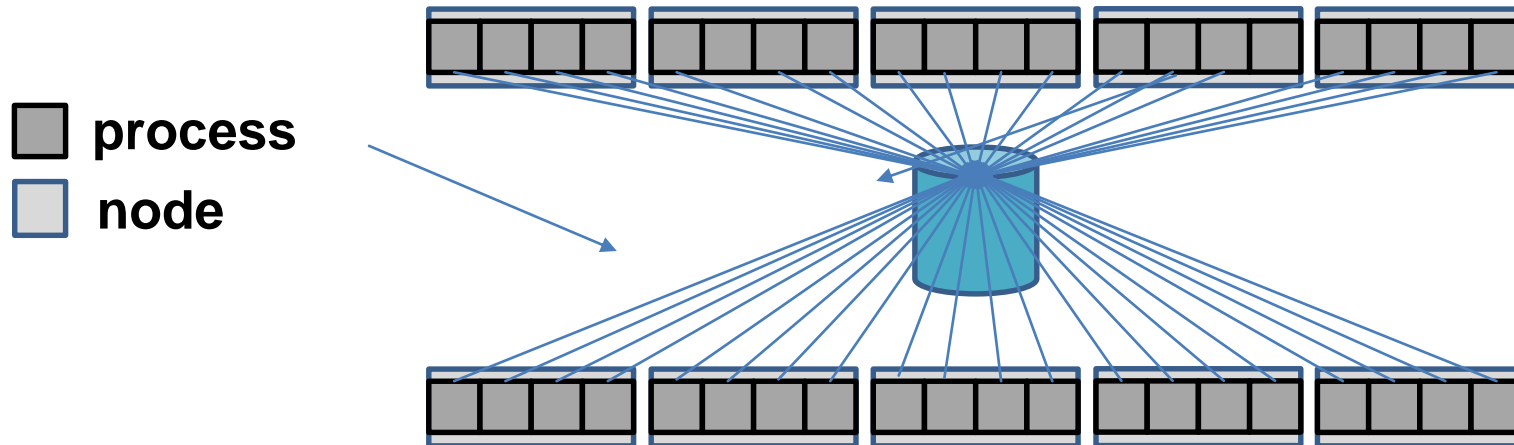
- relevant only when file is **created**, i.e. in `MPI_File_open`
- must be the same for all processes
- is only a hint

### Keys for info object

<code>striping_factor</code>	int	number of I/O devices the file should be striped across
<code>striping_unit</code>	int	number of consecutive bytes stored on one I/O device before the next is used



- Each process might access I/O devices
- Can generate high load
- Collective buffering to mitigate this problem

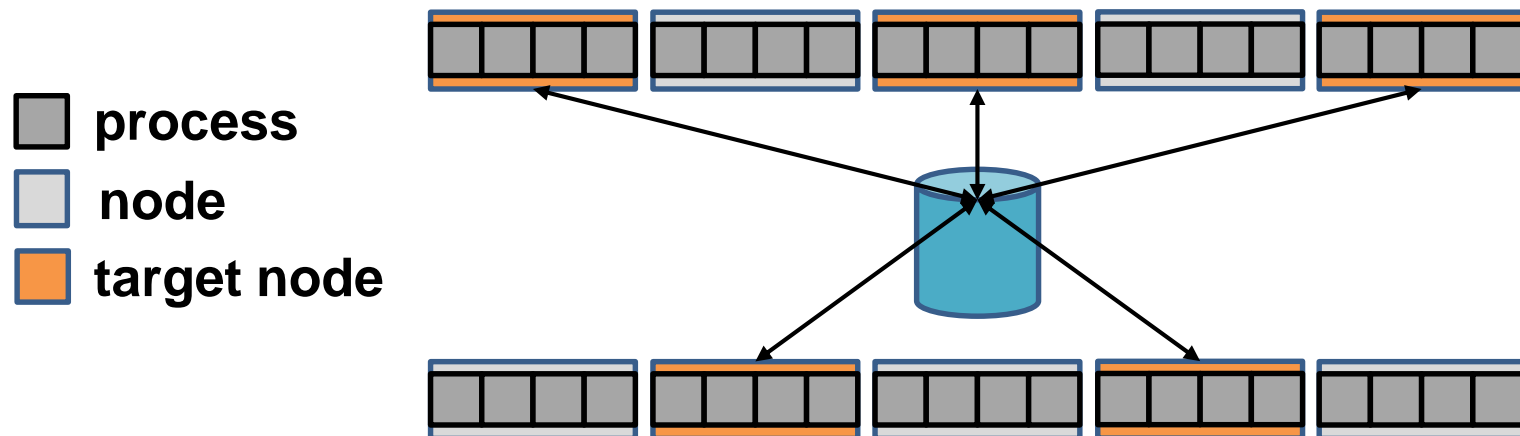


## Collective buffering

- Optimization for collective accesses
- Access performed on behalf of all processes by some target nodes
- Key/value pairs must be the same on all processes

## Keys for info object

<code>collective_buffering</code>	<code>bool</code>	true if application might benefit from collective buffering, false if not
<code>cb_block_size</code>	<code>int</code>	target nodes access data in chunks of this size
<code>cb_buffer_size</code>	<code>int</code>	buffer size on target node, that can be used for collective buffering, typically a multiple of the block size
<code>cb_nodes</code>	<code>int</code>	number of target nodes that should be used





## Example: create MPI info object for MPI\_File\_open

```
MPI_Info info;

MPI_Info_create(&info);
// Hint: stripe over 10 I/O devices
MPI_Info_set(info, "striping_factor", "10");
// Hint: enable collective buffering
MPI_Info_set(info, "collective_buffering", "true");
// Hint: use 4 target nodes for buffering
MPI_Info_set(info, "cb_nodes", "4");

...

MPI_File_open(comm, filename, amode, info, &fh);

...

MPI_Info_free(&info);
```



- **Pre-allocating space for a file (may be expensive)**

```
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
```

- **Resizing a file (may speed up first writing on a file)**

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
```

- **Querying file size**

```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
```

- **Querying file access mode**

```
int MPI_File_get_amode(MPI_File fh, int *amode)
```

- **File info object**

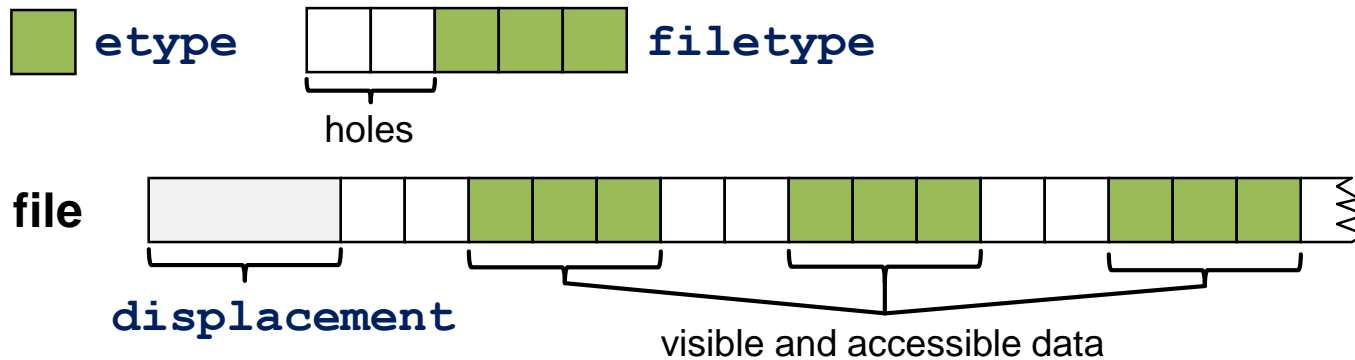
```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

```
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
```



---

# Views



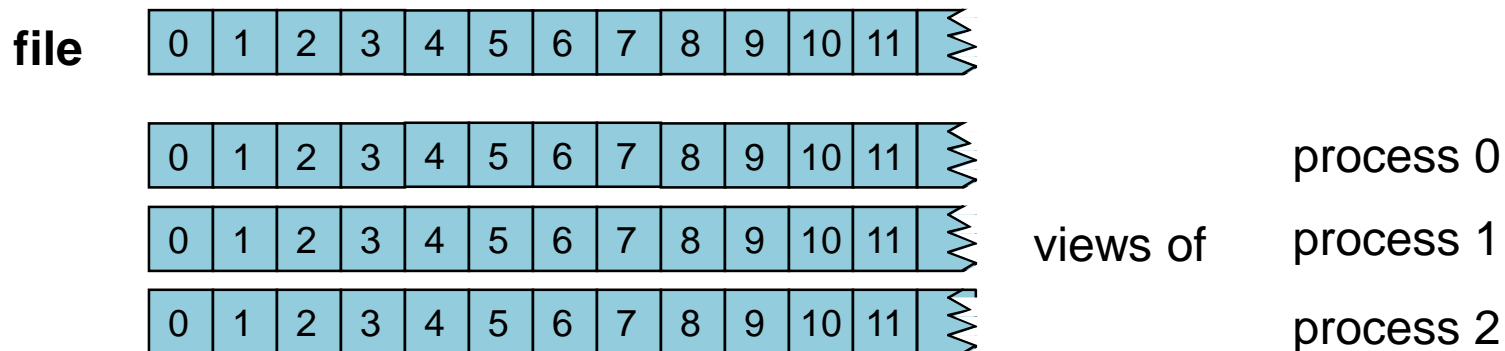
example from MPI 3.1  
standard document

- Visible and accessible data from a file
- Each process has its **own** view
- View is described via (`displacement`, `etype`, `filetype`)
- Pattern of `filetype` is repeated beginning at `displacement`
- Views can be changed, but this is a **collective** operation
- Default view: linear byte stream (`0`, `MPI_BYTE`, `MPI_BYTE`)



- After file open each file has the default view
- Default view: linear byte stream
  - `displacement = 0`
  - `etype = MPI_BYTE`
  - `filetype = MPI_BYTE`
- `MPI_BYTE` matches with any datatype

■ `etype = filetype = MPI_BYTE`



 **etype** elementary datatype

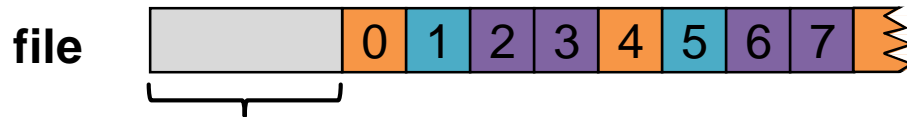
    **filetype** process 0

    **filetype** process 1

    **filetype** process 2

holes



**tiling a file with filetypes:**







**displacement**

displacement in bytes

**view** of process 0

**view** of process 1

**view** of process 2

example from MPI 3.1 standard document

<code>file</code>	<ul style="list-style-type: none"><li>▪ ordered collection of data items</li></ul>
<code>displacement</code>	<ul style="list-style-type: none"><li>▪ position from the beginning of the file</li><li>▪ marks the start of the view</li><li>▪ <b>unit: byte</b></li></ul>
<code>etype</code>	<ul style="list-style-type: none"><li>▪ elementary data type</li><li>▪ unit of data access and positioning</li><li>▪ type displacements must be:<ul style="list-style-type: none"><li>▪ nonnegative, monot. nondecreasing, and nonabsolut</li></ul></li><li>▪ <b>same for all processes</b></li></ul>
<code>filetype</code>	<ul style="list-style-type: none"><li>▪ single or multiple <b>etypes</b></li><li>▪ size of holes must be multiples of <b>etype</b> extent</li><li>▪ repeated pattern after <b>displacement</b></li><li>▪ type displacements must be:<ul style="list-style-type: none"><li>▪ nonnegative, monot. nondecreasing, nonabsolut,</li></ul></li><li>▪ <b>can be different for all processes</b></li></ul>
<code>view</code>	<ul style="list-style-type: none"><li>▪ accessible data of a file by a process</li><li>▪ defined by <b>displacement, etype, filetype</b></li></ul>
<code>offset</code>	<ul style="list-style-type: none"><li>▪ position in file relative to current view</li><li>▪ type <code>MPI_Offset</code> in C, <code>INTEGER(KIND=MPI_OFFSET_KIND)</code> in Fortran</li><li>▪ <b>uint: etype</b></li></ul>



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

- Changes the process's view of the data
- **Collective** operation
- Local and shared file pointers are reset to zero
- `etype` and `filetype` must be committed types
- `datarep` is a string specifying the format data is written to a file:  
`native`, `internal`, `external32`, or `user-defined`
- Same `etype` extent and same `datarep` on all processes
- `disp`: `MPI_Offset` in C, `INTEGER(KIND=MPI_OFFSET_KIND)` in Fortran

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp,
                     MPI_Datatype *etype, MPI_Datatype *filetype,
                     char *datarep)
```

- Returns process's view of the data

**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

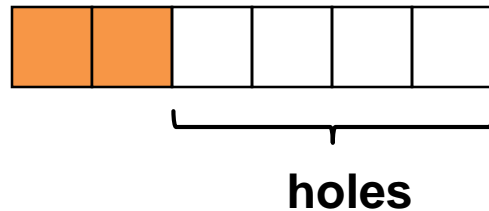
**external32**

- follows standardized representation (big endian IEEE)
- all input/output operations are converted from/to **external32**
- 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

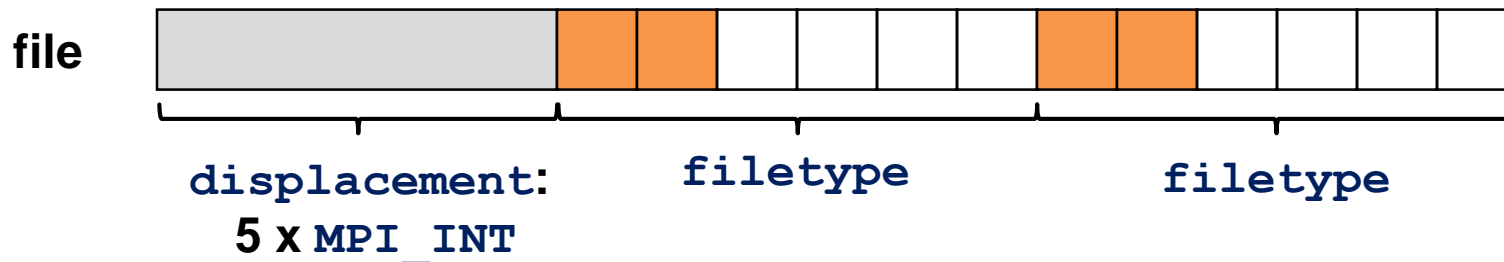
- Basic example: File view for one process
  - View contains holes with respect to original file



`etype = MPI_INT`



`filetype`: two `MPI_INT`s followed by a gap of four `MPI_INT`s







```

MPI_Offset disp;
MPI_Datatype etype, filetype;
int sizes[]      = { 6 };
int sub_sizes[]  = { 2 };
int start_idxes[] = { 0 };

```

```

MPI_Type_create_subarray( 1, sizes, sub_sizes, start_idxes,
                          MPI_ORDER_C, MPI_INT, &filetype);
MPI_Type_commit(filetype);

```

```

disp = 5 * 4; ! 4 = size of MPI_INT in bytes
etype = MPI_INT;

```

```

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR,
              MPI_INFO_NULL, &fh);

```

```

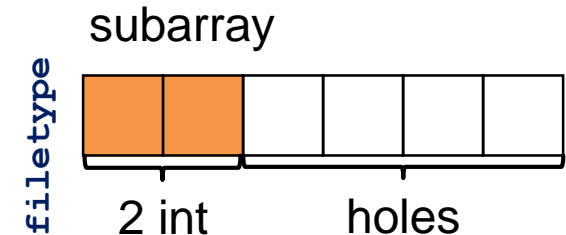
MPI_File_set_view(fh, disp, etype, filetype, "native", MPI_INFO_NULL);

```

```

MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);

```



---

# Reading and Writing



- **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`)**
  - atomic
  - non-atomic

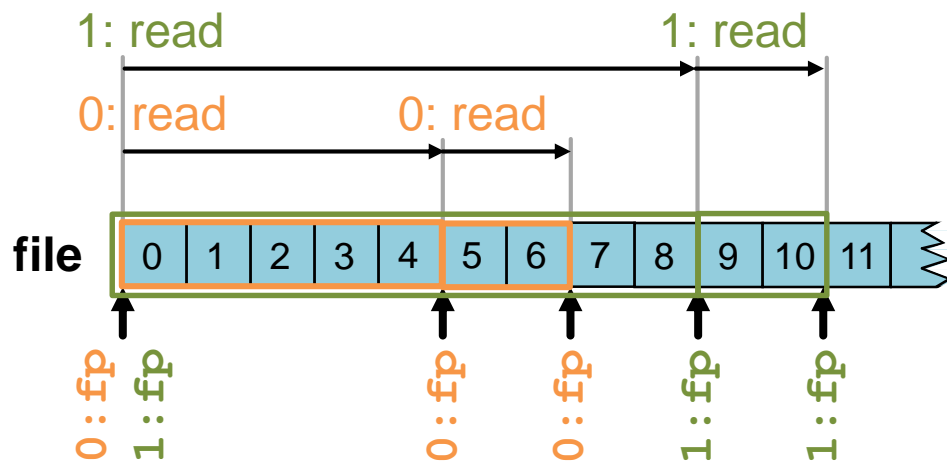
Positioning	Synchronization	Non-collective	Collective
Explicit offsets	blocking	Reat_at	Read_at_all
		Write_at	Write_at_all
	non-blocking	Iread_at	Iread_at_all
		Iwrite_at	Iwrite_at_all
	split collective		Read_at_all_(begin end)
			Write_at_all_(begin end)
Individual file pointers	blocking	Read	Read_all
		Write	Write_all
	non-blocking	Iread	Iread_all
		Iwrite	Iwrite_all
	split collective		Read_all_(begin end)
			Write_all_(begin end)
Shared file pointers	blocking	Read_shared	Read_ordered
		Write_shared	Write_ordered
	non-blocking	Iread_shared	
		Iwrite_shared	
	split collective		Read_ordered_(begin end)
			Write_ordered_(begin end)



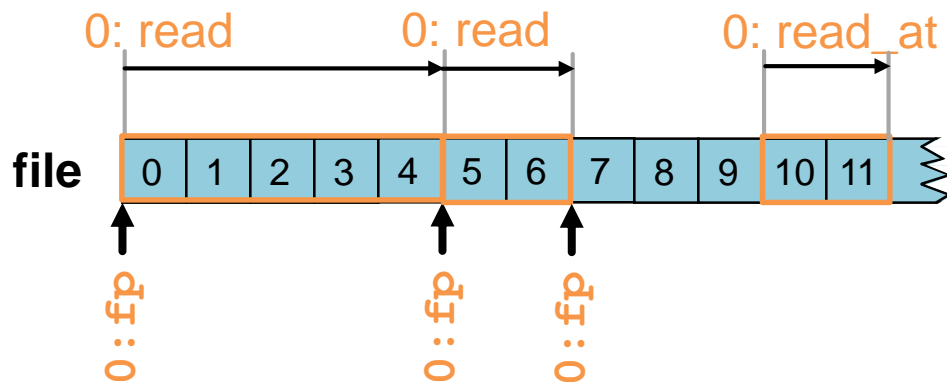
```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset,  
    void *buf, int count, MPI_Datatype datatype,  
    MPI_Status *status)
```

- Read data starting at `offset`
- Read `count` elements of `datatype`
- Starting `offset` \* units of `etype` from begin of view (displacement)
- 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)`
- Explicit offset routines do **not** alter file pointer

- Each process maintains its own individual file pointer



- Explicit offsets do not affect file pointers





```
int MPI_File_read(MPI_File fh,  
                 void *buf, int count, MPI_Datatype datatype,  
                 MPI_Status *status)
```

- Arguments have same meaning as for `MPI_File_read_at`
- `offset` is individual file pointer of calling process
- Individual file pointer is automatically incremented by  
`fp = fp + count * elements(datatype) / elements(etype)`
- I.e. it points to the next `etype` after the last one that will be accessed (formula is not valid if EOF is reached)
  
- Behaves nearly like standard serial file I/O



- **Set offset of individual file pointer fp:**

```
int MPI_File_seek(MPI_File fh,  
                 MPI_Offset offset, int whence)
```

whence	description
<code>MPI_SEEK_SET</code>	set fp to offset
<code>MPI_SEEK_CUR</code>	set fp to fp + offset
<code>MPI_SEEK_END</code>	set fp to EOF + offset

- **Get offset of individual file pointer:**

```
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
```

- **Get absolute byte position from offset for current view**

```
int MPI_File_get_byte_offset(MPI_File fh,  
                             MPI_Offset offset, MPI_Offset *disp)
```

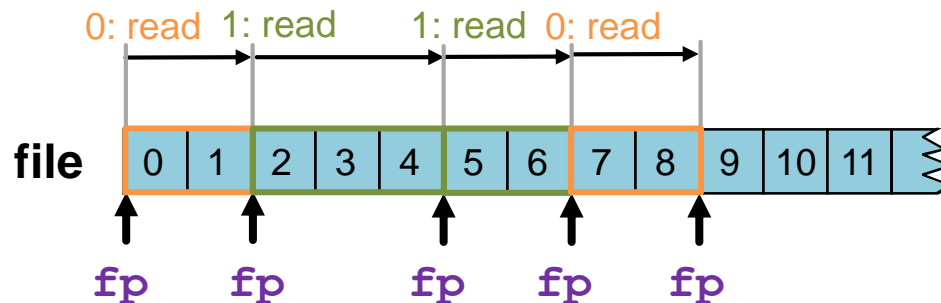
- **Retrieve `offset` and convert it into byte displacement, e.g. for usage in new view**





```
int MPI_File_read_shared(MPI_File fh,
    void *buf, int count, MPI_Datatype datatype,
    MPI_Status *status)
```

- One shared file pointer per `MPI_File_open`
- All processes must have the same view
- Individual file pointers are not affected
- Ordering during serialization is not deterministic
- Use ordered (collective call) if determinism is required
- Use `*shared` routines to get/set file pointer

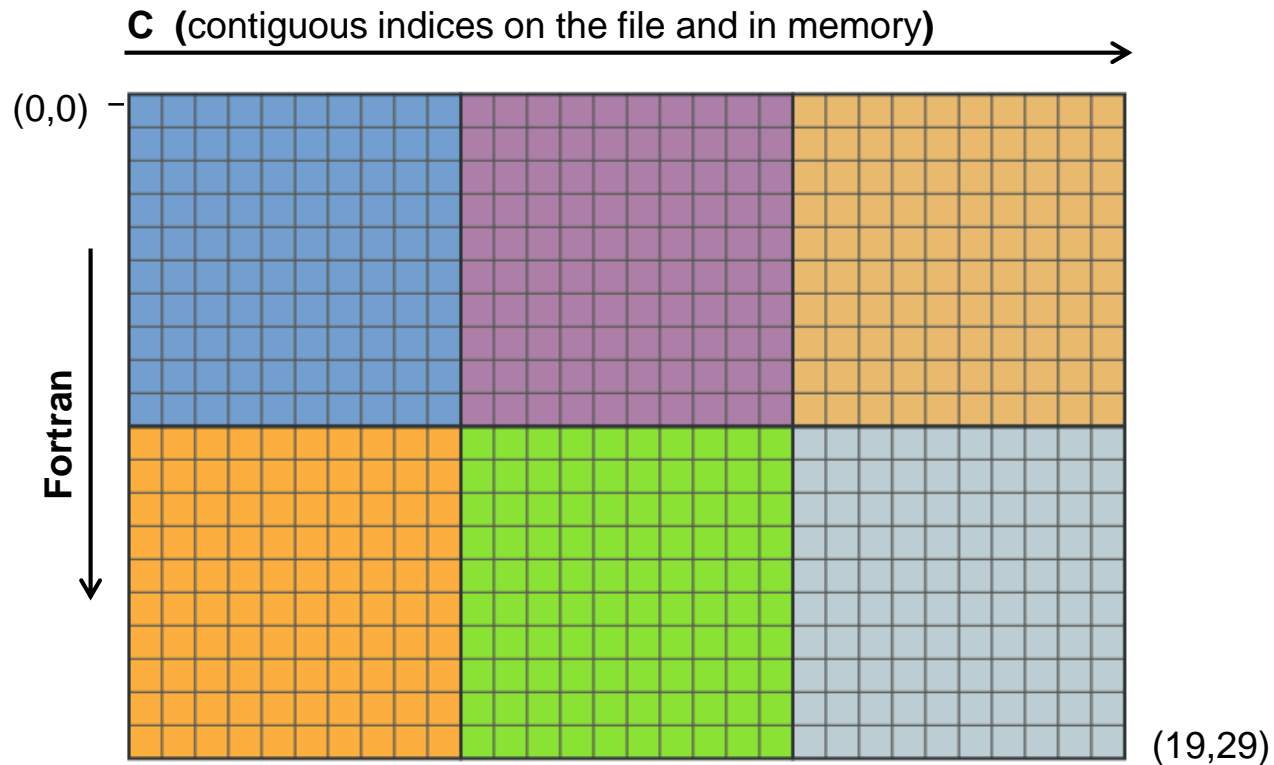


# Examples & Use Cases



- **Task**
  - read a **global matrix** from a file
  - store a subarray into a local array on each process
  - according to a given distribution scheme
- **2-dimensional distribution scheme: (BLOCK,BLOCK)**
- ***garray* on the file 20x30:**
  - Contiguous indices is language dependent:
  - Fortran: (1,1), (2,1), (3,1), ... , (1,10), (2,10), (3,10), ..., (20,30)
  - in C/C++:[0][0], [0][1], [0][2], ... , [10][0], [10][1], [10][2], ..., [20][30]
- ***larray* = local array in each MPI process**  
**= subarray of the global array**
- **same ordering on file (*garray*) and in memory (*larray*)**

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





```
// error handling omitted for brevity
// distribute garray[20,30] onto the processors [2,3]
```

```
double larray[10][10];
MPI_Offset disp, offset, disp = 0, offset = 0;
```

```
ndims=2;
psizes[0]=2; period[0]=0;
psizes[1]=3; period[1]=0;
MPI_Cart_create(MPI_COMM_WORLD, ndims, psizes, period, 1, &comm);
```

Create  
virtual  
topology

```
MPI_Comm_rank(comm, &rank) ;
MPI_Cart_coords(comm, rank, ndims, coords);
```

```
gsizes[0]=20; lsizes[0]=10; starts[0]=coords[0]*lsizes[0];
gsizes[1]=30; lsizes[1]=10; starts[1]=coords[1]*lsizes[1];
MPI_Type_create_subarray(ndims, gsizes, lsizes, starts,
                        MPI_ORDER_C, MPI_DOUBLE, &stype);
```

Create  
custom  
datatype

```
MPI_Type_commit(&stype);
```

```
MPI_File_open(comm, file_name, MPI_MODE_READ, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_DOUBLE, stype, "native", MPI_INFO_NULL);
MPI_File_read_at_all(fh, offset,
                    larray, lsizes[0]*lsizes[1], MPI_DOUBLE,
                    &status);
```

- Open file
- create view
- read data

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

<pre>rank = 0 coords = ( 0, 0) starts = ( 0, 0) garray( 0:9, 0:9) = larray( 0:9, 0:9)</pre>	<pre>rank = 1 coords = ( 0, 1) starts = ( 0, 10) garray( 0:9, 10:19) = larray ( 0:9, 0:9)</pre>	<pre>rank = 2 coords = ( 0, 2) starts = ( 0, 20) garray( 0:9, 20:29) = larray( 0:9, 0:9)</pre>
<pre>rank = 3 coords = ( 1, 0) starts = (10, 0) garray(10:19, 0:9) = larray( 0:9, 0:9)</pre>	<pre>rank = 4 coords = ( 1, 1) starts = (10, 10) garray(10:19, 10:19) = larray( 0:9, 0:9)</pre>	<pre>rank = 5 coords = ( 1, 2) starts = (10, 20) garray(10:19, 20:29) = larray( 0:9, 0:9)</pre>



- **Scenery A:** Each process has to read the whole file
- **Solution 1:** `MPI_File_read_all` **blocking**  
  
collective with individual file pointers, with same view (`displacement+etype+filetype`) on all processes
- **Solution 2:** `MPI_File_read_all_begin` **nonblocking**  
  
collective with individual file pointers, with same view (`displacement+etype+filetype`) on all processes, then computing some other initialization,  
  
`MPI_File_read_all_end.`



- **Scenery B:** 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 pointer)
- **Scenery C:** 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 pointer)
- or: `MPI_File_read_all`  
collective with individual file pointers,  
different views: `filetype` with  
`MPI_Type_create_subarray(..., &filetype)`





- **File handles have their own error handler**
- **Default is `MPI_ERRORS_RETURN`, i.e. non-fatal**
  - message passing: `MPI_ERRORS_ARE_FATAL`
- **Default is associated with `MPI_FILE_NULL`**
  - 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**

- 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
- Full implementation of MPI I/O available on / in
  - Intel MPI
  - Open MPI
  - MVAPICH
  - ...
- **Generally, use of MPI I/O is often limited to special file systems; do not expect it to work on your average NFS-mounted \$HOME**
  - If it works at all data loss might occur!