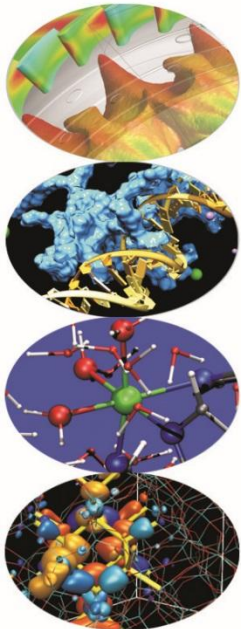
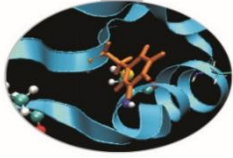


Parallel IO: basics and MPI2-IO

Super Computing Applications and Innovation Department

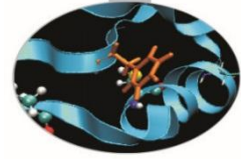
Courses Edition 2016





- IO is a crucial issue in modern HPC applications:
 - deal with very large datasets while running massively parallel applications on supercomputers
 - amount of data saved is increased
 - latency to access to disks is not negligible
 - data portability (e.g. endianness)
- Solutions to avoid that IO become a bottleneck:
 - HW: parallel file-system available on all the HPC platforms
 - SW: high level libraries able to manage parallel accesses to the file in efficient way (e.g. MPI2-IO, HDF5, NetCDF, ...)

Managing IO in Parallel Applications



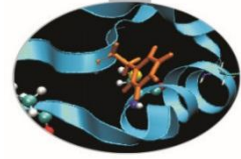
Solutions to managing IO in parallel applications must take into account different aspects of the application and implementation:

- potential performance improvements
- scaling with respect resources/system size
- ensure data consistency
- avoid communications
- strive for usability

Common approaches:

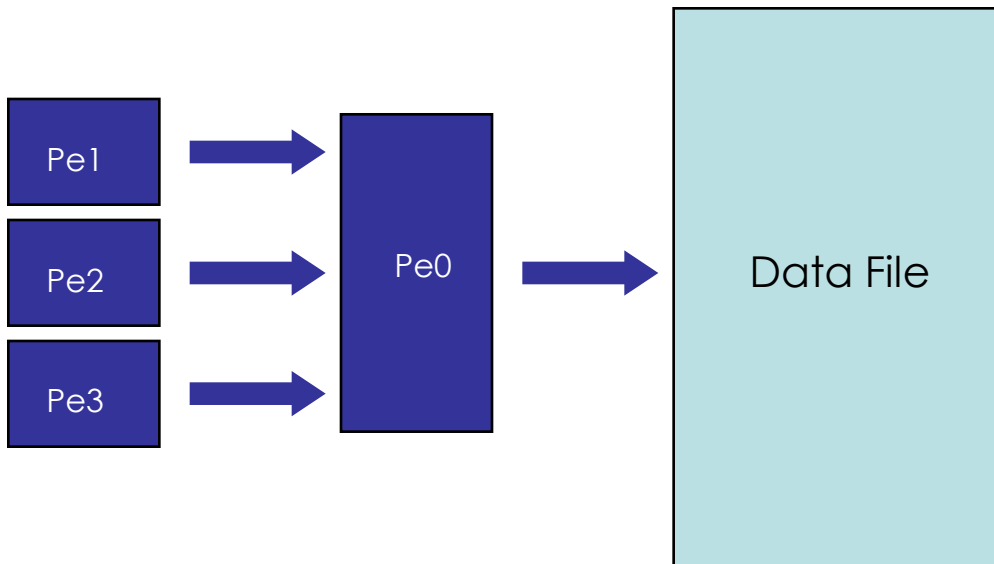
1. Master-Slave
2. Distributed IO on local files
3. Coordinated controlled accesses
4. MPI-IO or high level libraries
(e.g. HDF5, NetCDF use MPI-IO as the backbone)

Managing IO in Parallel Applications



Approach 1: Master-Slave

Only 1 processor performs IO



Goals:

Scalable: **NO**

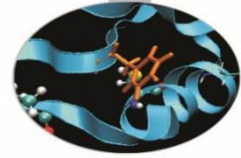
Ensure data consistency: **YES**

Avoid communication: **NO**

Usability: **YES**

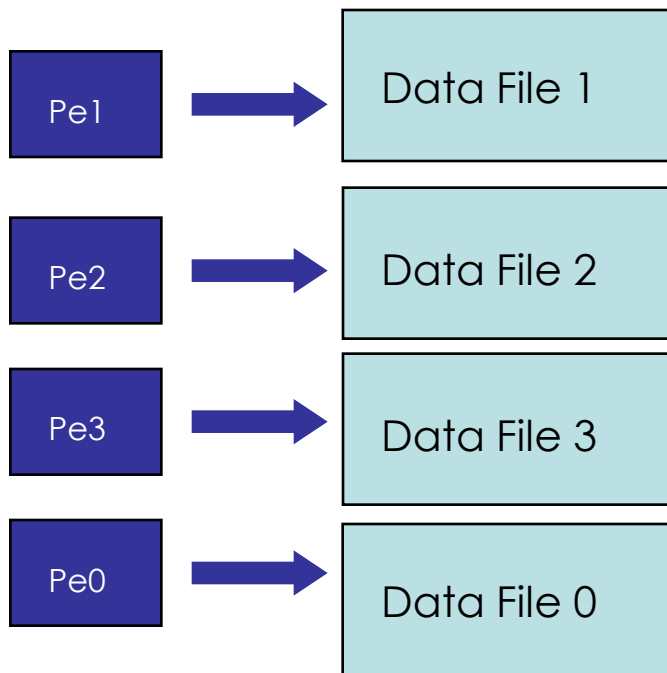
note: no parallel FS needed

Managing IO in Parallel Applications



Approach 2: Distributed IO on local files

All the processors read/writes their own files



Goals:

Scalable: **YES**

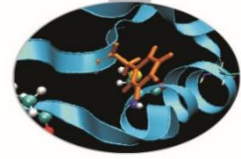
Ensure data consistency: **YES**

Avoid communication: **YES**

Usability: **NO (need extra work later)**

Warning: avoid to parametrize with processors!!!

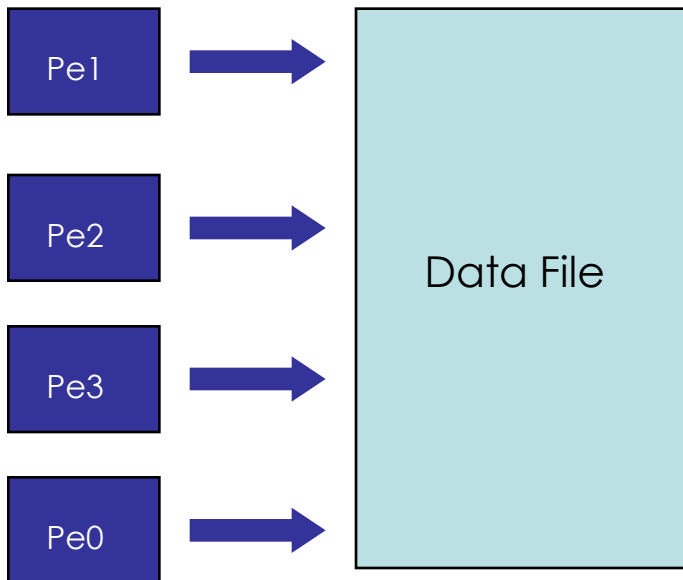
Managing IO in Parallel Applications



Approach 3: Coordinated controlled accesses

All the processors read/writes on a single file

- offsets to access data must be explicitly specified



Goals:

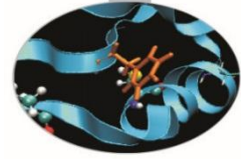
Scalable: **YES**

Ensure data consistency: **NO**

Avoid communication: **YES**

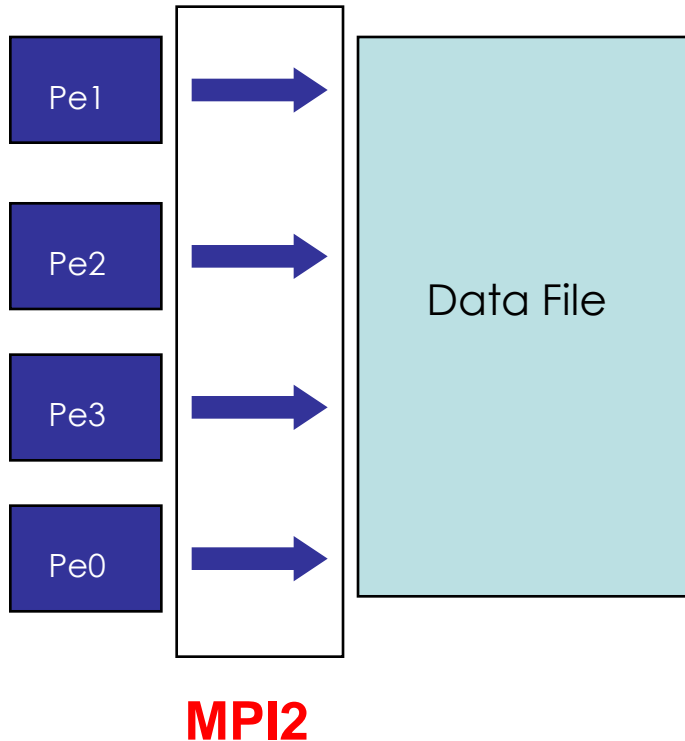
Usability: **YES**

Managing IO in Parallel Applications



Solution 4: MPI2 IO (or other parallel IO libraries)

MPI functions perform the IO. Asynchronous IO is also supported.



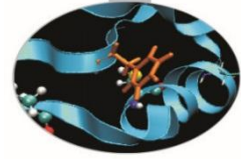
Goals:

Scalable: **YES**

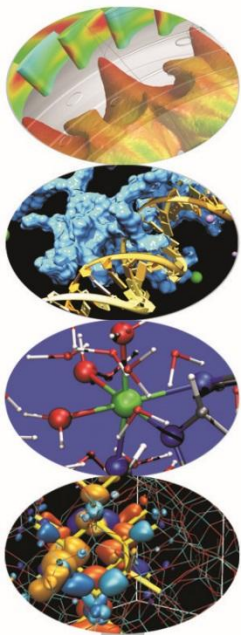
Ensure data consistency: **YES**

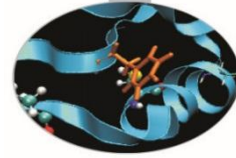
Avoid communication: **YES**

Usability: **YES**

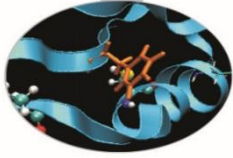


MPI2-IO



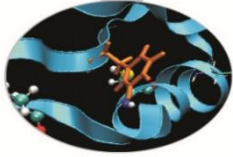


- MPI-IO: introduced in MPI-2.x standard (1997)
 - allow non-contiguous access in both memory and file
 - reading/writing a file is like send/receive a message from a MPI buffer
 - optimized access for non-contiguous data
 - collective / non-collective access operations with communicators
 - blocking / non-blocking calls
 - data portability (implementation/system independent)
 - good performance in many implementations
- Why do we start to use it???
 - syntax and semantic are very simple to use
 - performance : 32 MPI processes (4x8) with local grid 50000² (sp)
 - MPI-IO: **85**sec vs Traditional master/slave IO: **5200**sec
 - dimension of written file: 305GB
 - measured bandwidth of GPFS (single thread): 65MB/sec
 - measured bandwidth of MPI gather for Master/slave: 900MB/sec
 - MPI_File_write_all (**85**sec) vs MPI_file_write (**320**sec)



- MPI-IO provides basic IO operations:
 - open, seek, read, write, close (etc.)
- open/close are collective operations on the same file
 - many modalities to access the file (composable: |, +)
- read/write are similar to send/recv of data to/from a buffer
 - each MPI process has its own local pointer to the file (individual file pointer) for seek, read, write operations
 - offset variable is a particular kind of variable and it is given in elementary unit (etype) of access to file (default in byte)
 - error: declare offset as an integer
 - it is possible to know the exit status of each subroutine/function

Using individual file pointers

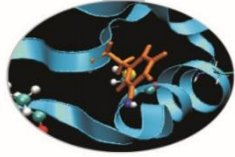


```
#include "mpi.h"
```

```
int main(int argc, char **argv){  
    int rank, nprocs;  
  
    MPI_Init(&argc, &argv);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);  
  
    MPI_File fh; MPI_Status status;  
    MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,  
        MPI_INFO_NULL, &fh);  
  
    int filesize;  
    MPI_File_get_size(fh, &filesize);  
    int bufsize = filesize/nprocs;  
    int nints = bufsize/sizeof(int);  
    int *buf = (int*) malloc(nints);  
  
    MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);  
    MPI_File_read(fh, buf, nints, MPI_INT, &status);  
  
    MPI_File_close(&fh);  
    ...  
}
```

File offset
determined by
MPI_File_seek

Using individual file pointers



PROGRAM Output

```
USE MPI
IMPLICIT NONE
INTEGER :: err, i, myid, file, intsize
INTEGER :: status(MPI_STATUS_SIZE)
INTEGER, PARAMETER :: count=10000
INTEGER DIMENSION(count) :: buf
INTEGER, INTEGER(KIND=MPI_OFFSET_KIND) :: disp

CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, err)

DO i = 1, count
    buf(i) = myid * count + i
END DO
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', MPI_MODE_WRONLY + &
    MPI_MODE_CREATE, MPI_INFO_NULL, file, err)

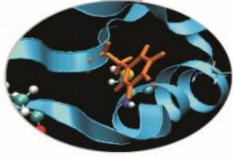
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize, err)
disp = myid * count * intsize

CALL MPI_FILE_SEEK(file, disp, MPI_SEEK_SET, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)

CALL MPI_FILE_CLOSE(file, err)

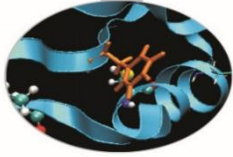
CALL MPI_FINALIZE(err)
END PROGRAM Output
```

File offset
determined by
MPI_File_seek



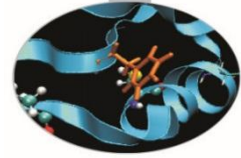
```
MPI_FILE_OPEN(comm, filename, amode, info, fh)
  IN comm: communicator (handle)
  IN filename: name of file to open (string)
  IN amode: file access mode (integer)
  IN info: info object (handle)
  OUT fh: new file handle (handle)
```

- **Collective** operation across processes within a communicator.
- Filename must reference the same file on all processes.
- Process-local files can be opened with **MPI_COMM_SELF**.
- Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation. The file view can be changed via the **MPI_FILE_SET_VIEW** routine.
- Additional information can be passed to MPI environment via the MPI_Info handle. The info argument is used to provide extra information on the file access patterns. The constant **MPI_INFO_NULL** can be specified as a value for this argument.



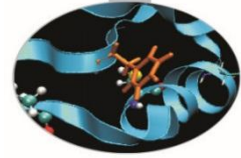
Each process within the communicator must specify the same filename and access mode (amode):

<code>MPI_MODE_RDONLY</code>	read only
<code>MPI_MODE_RDWR</code>	reading and writing
<code>MPI_MODE_WRONLY</code>	write only
<code>MPI_MODE_CREATE</code>	create the file if it does not exist
<code>MPI_MODE_EXCL</code>	error if creating file that already exists
<code>MPI_MODE_DELETE_ON_CLOSE</code>	delete file on close
<code>MPI_MODE_UNIQUE_OPEN</code>	file will not be concurrently opened elsewhere
<code>MPI_MODE_SEQUENTIAL</code>	file will only be accessed sequentially
<code>MPI_MODE_APPEND</code>	set initial position of all file pointers to end of file



```
MPI_FILE_CLOSE(comm)
  INOUT fh: file handle (handle)
```

- Collective operation
- Call this function when the file access is finished to free the file handle.



- Several parallel file system can benefit from “hints” given to MPI-IO
 - optimization may be possible with performance benefits
- Info to MPI are opaque objects (MPI_Info in C or integer in FORTRAN)
- hints can be provided as (key, value) pairs with MPI_Info_set function

```
MPI_Info info;
```

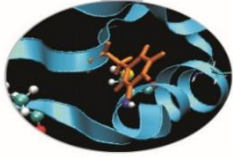
```
MPI_Info_create(&info);
```

```
// set number of I/O devices across which the file should be striped  
MPI_Info_set(info, "striping_factor", "4");
```

```
// set the striping unit in bytes  
MPI_Info_set(info, "striping_unit", "65536");
```

```
// buffer size of collective I/O  
MPI_Info_set(info, "cb_buffer_size", "8388608");
```

```
// number of processes that should perform disk accesses during collective I/O  
MPI_Info_set(info, "cb_nodes", "4");
```

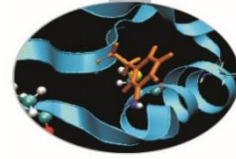
- Info can also be retrieved from the implementation
 - which hints were used for a file?
 - which defaults are actually in use?

```
char key[MPI_MAX_INFO_KEY], value[MPI_MAX_INFO_VAL];
MPI_Info info_used;

MPI_File_get_info(fh, &info_used);

int nkeys;
MPI_Info_get_nkeys(info_used, &nkeys);

for (int i=0; i<nkeys; i++) {
    MPI_Info_get_nthkey(info_used, i, key);
    int flag; // return true if key was set
    MPI_Info_get(info_used, key, MPI_MAX_INFO_VAL, value, &flag);
    printf("key = %s, value = %s\n", key, value);
}
```



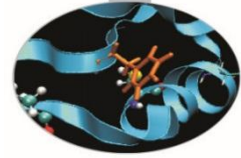
```
MPI_FILE_WRITE (fh, buf, count, datatype, status)
  INOUT fh: file handle (handle)
  IN buf: initial address of buffer (choice)
  IN count: number of elements in buffer (integer)
  IN datatype: datatype of each buffer element (handle)
  OUT status: status object (status)
```

- Write `count` elements of `datatype` from memory starting at `buf` to the file
- Starts writing at the current position of the file pointer
- **status** will indicate how many **bytes** have been written
- Updates position of file pointer after writing
- Blocking, independent (local, not collective) call.

– **Individual file pointers are used:**

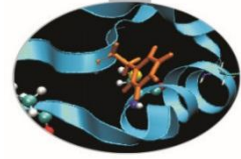
Each processor has **its own pointer** to the file

Pointer on a processor **is not influenced** by any other processor



```
MPI_FILE_READ (fh, buf, count, datatype, status)
  INOUT fh: file handle (handle)
  OUT buf: initial address of buffer (choice)
  IN count: number of elements in buffer (integer)
  IN datatype: datatype of each buffer element (handle)
  OUT status: status object (status)
```

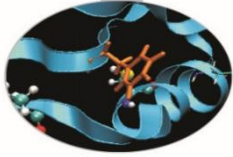
- Read count elements of datatype from the file to memory starting at buf
- Starts reading at the current position of the file pointer
- status will indicate how many **bytes** have been read
- Updates position of file pointer after writing
- Blocking, independent (local, not collective) call.
- **Individual file pointers are used:**
 - Each processor has **its own pointer** to the file
 - Pointer on a processor **is not influenced** by any other processor



```
MPI_FILE_SEEK (fh, offset, whence)
  INOUT fh: file handle (handle)
  IN offset: file offset in byte (integer)
  IN whence: update mode (state)
```

- Updates the individual file pointer according to whence, which can be:
 - MPI_SEEK_SET: the pointer is set to offset
 - MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
 - MPI_SEEK_END: the pointer is set to the end of the file plus offset
- offset can be negative, which allows seeking backwards
- It is erroneous to seek to a negative position in the view

Querying the position



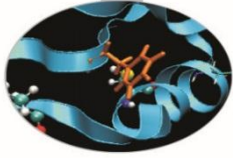
```
MPI_FILE_GET_POSITION (fh, offset)
```

```
IN fh: file handle (handle)
```

```
OUT offset: offset of the individual file pointer  
(integer)
```

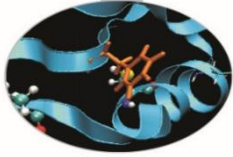
- Returns in offset the current position of the individual file pointer
 - offset is in etype units relative to the current view
- The value can be used to return to this position or calculate a displacement
 - do not forget to convert from offset to byte displacement if needed using MPI_FILE_GET_BYTE_OFFSET

Querying file size

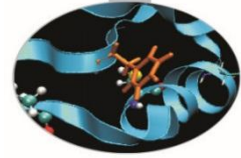


```
MPI_FILE_GET_SIZE (fh, size)
  IN fh: file handle (handle)
  OUT size: file size in bytes (integer)
```

- Returns the file size in bytes
- The value is returned in a variable type of proper kind:
 - integer MPI_KIND_OFFSET in FORTRAN
 - MPI_Offset in C



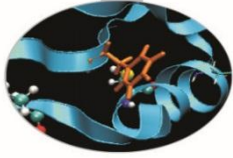
- MPI-IO provides a large number of routines to read and write data from a file.
- There are three properties which differentiate data access routines:
 - **Positioning:** users can either specify explicitly the offset in the file at which the data access takes place or they can use MPI file pointers
 - **Synchronisation:** as for common communication APIs, we can use both synchronous (blocking) or asynchronous (non-blocking) function calls
 - **Coordination:** data accesses can be local or collective operations



Positioning:

Users can either specify the offset in the file at which the data access takes place or they can use MPI file pointers:

- **Individual file pointers:** each process has its own file pointer that is only altered on accesses of that specific process
- **Shared file pointer:** pointer is shared among all processes in the communicator used to open the file
 - It is modified by any shared file pointer access of any process
 - Shared file pointers can only be used if file type gives each process access to the whole file!
- **Explicit offset:** no file pointer is used or modified
 - An explicit offset is given to determine access position
 - This can not be used with MPI MODE SEQUENTIAL!



Synchronisation:

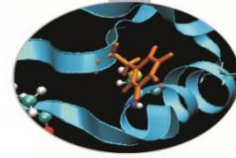
MPI-2 supports both **blocking** and **non-blocking IO** routines:

- A **blocking IO** call will not return until the IO request is completed.
- A **nonblocking IO** call initiates an IO operation, but not wait for its completion. It also provides 'split collective routines' which are a restricted form of non-blocking routines for collective data access.

Coordination:

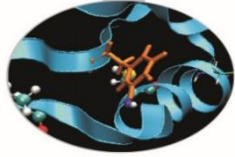
Data access can either take place from individual processes or collectively across a group of processes:

- **collective:** MPI coordinates the reads and writes of processes
- **independent:** no coordination by MPI



Positioning	Synchronisation	Coordination	
		<i>Noncollective</i>	<i>Collective</i>
<i>Explicit offsets</i>	<i>Blocking</i>	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	<i>Non-blocking & split collective</i>	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
<i>Individual file pointers</i>	<i>Blocking</i>	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	<i>Non-blocking & split collective</i>	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
<i>Shared file pointer</i>	<i>Blocking</i>	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	<i>Non-blocking & split collective</i>	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Using explicit offsets



PROGRAM main

```
include 'mpif.h'

parameter (FILESIZE=1048576, MAX_BUFSIZE=1048576, INTSIZE=4)
integer buf(MAX_BUFSIZE), rank, ierr, fh, nprocs, nints
integer status(MPI_STATUS_SIZE), count
integer (kind=MPI_OFFSET_KIND) offset

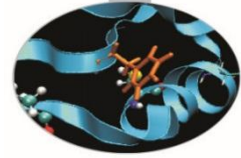
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
    MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)

nints = FILESIZE/(nprocs*INTSIZE)
offset = rank * nints * INTSIZE

call MPI_FILE_READ_AT(fh, offset, buf, nints, MPI_INTEGER, status, ierr)
call MPI_FILE_CLOSE(fh, ierr)
call MPI_FINALIZE(ierr)
```

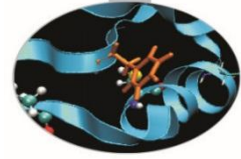
END PROGRAM main



```
MPI_FILE_WRITE_AT (fh, offset, buf, count, datatype, status)
  IN fh: file handle (handle)
  IN offset: file offset in byte (integer)
  IN buf: destination buffer
  IN count: number of read elements
  IN datatype: MPI type of each element
  OUT status: MPI status
```

- An explicit offset is given to determine access position
- The file pointer is neither used or incremented or modified
- Blocking, independent.

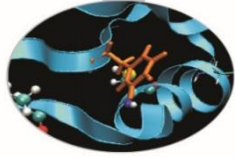
- Writes COUNT elements of DATATYPE from memory BUF to the file
- Starts writing at OFFSET units of etype from begin of view



```
MPI_FILE_READ_AT (fh, offset, buf, count, datatype, status)
  IN fh: file handle (handle)
  IN offset: file offset in byte (integer)
  IN buf: destination buffer
  IN count: number of read elements
  IN datatype: MPI type of each element
  OUT status: MPI status
```

- An explicit offset is given to determine access position
- The file pointer is neither used or incremented or modified
- Blocking, independent.

- reads COUNT elements of DATATYPE from FH to memory BUF
- Starts reading at OFFSET units of etype from begin of view

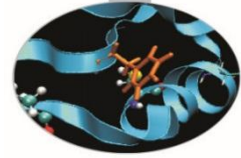


```
MPI_FILE_WRITE_SHARED (fh, buf, count, datatype, status)
```

```
MPI_FILE_READ_SHARED (fh, buf, count, datatype, status)
```

- Blocking, independent write/read using the shared file pointer
- Only the shared file pointer will be advanced accordingly
- DATATYPE is used as the access pattern to BUF
- Middleware will serialize accesses to the shared file pointer to ensure collision-free file access

Seeking and querying the shared file pointer position

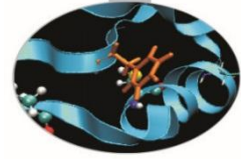


`MPI_FILE_SEEK_SHARED(fh, offset, whence)`

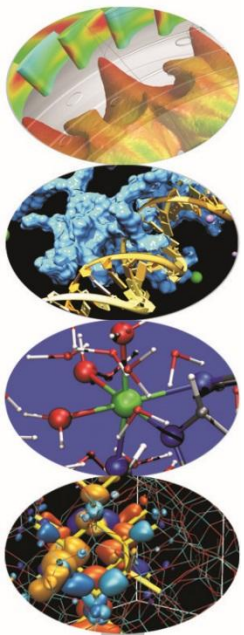
- Updates the individual file pointer according to `WHENCE` (`MPI_SEEK_SET`, `MPI_SEEK_CUR`, `MPI_SEEK_END`)
- `OFFSET` can be negative, which allows seeking backwards
- It is erroneous to seek to a negative position in the view
- The call is collective : all processes with the file handle have to participate

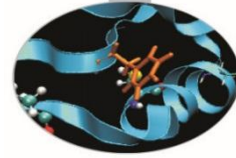
`MPI_FILE_GET_POSITION_SHARED(fh, offset)`

- Returns the current position of the individual file pointer in `OFFSET`
- The value can be used to return to this position or calculate a displacement
 - Do not forget to convert from offset to byte displacement if needed
- Call is not collective



MPI Derived Datatypes



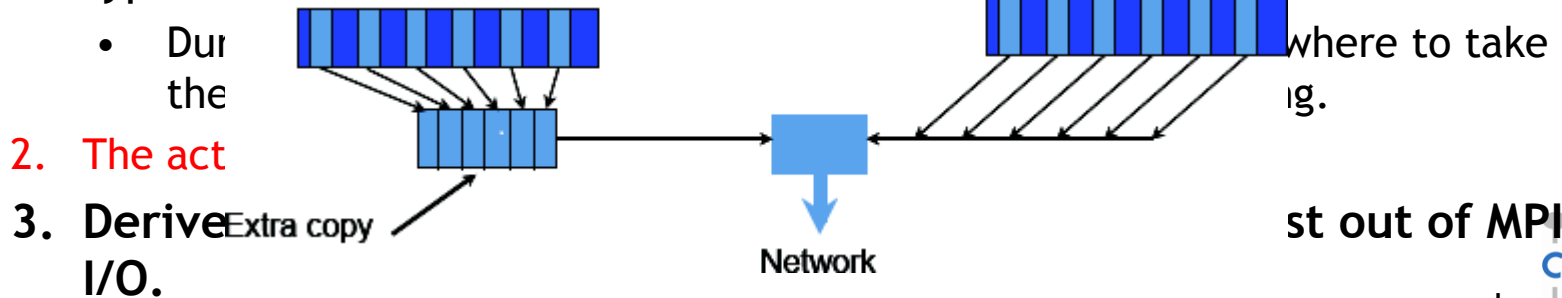


You may need to send messages that contain

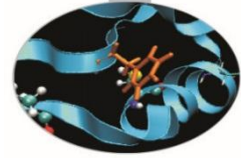
1. non-contiguous data of a single type (e.g. a sub-block of a matrix)
2. contiguous data of mixed types (e.g., an integer count, followed by a sequence of real numbers)
3. non-contiguous data of mixed types

Datatype solution:

1. The idea of MPI derived datatypes is to provide a simple, portable, elegant and efficient way of communicating non-contiguous or mixed types in a message.



<ggd



A **general datatype** is an **opaque object** able to describe a buffer layout in memory by specifying:

- A sequence of basic datatypes
- A sequence of integer (byte) displacements.

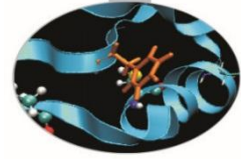
Typemap = $\{(\text{type } 0, \text{displ } 0), \dots (\text{type } n-1, \text{displ } n-1)\}$
- pairs of basic types and displacements (in byte)

Type signature = $\{\text{type } 0, \text{type } 1, \dots \text{type } n-1\}$

- list of types in the typemap
- gives size of each elements
- tells MPI how to interpret the bits it sends and received

Displacement:

- tells MPI where to get (when sending) or put (when receiving)



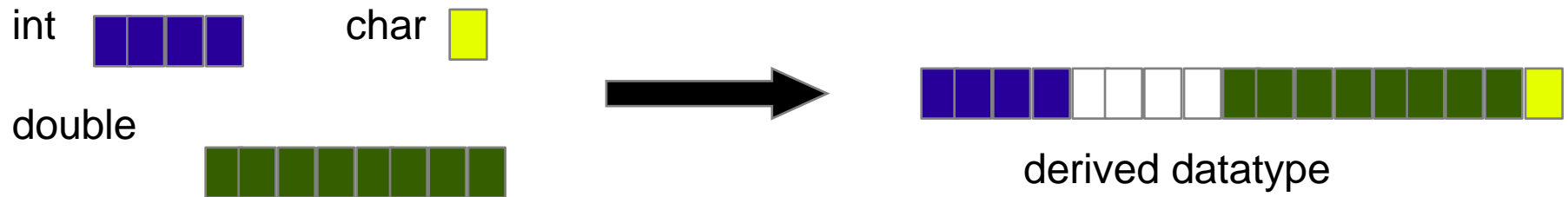
Example:

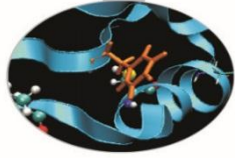
Basic datatypes are particular cases of a general datatype, and are predefined:

$MPI_INT = \{(int, 0)\}$

General datatype with typemap

$Typemap = \{(int,0), (double,8), (char,16)\}$



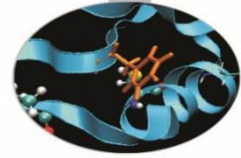


General datatypes are created and destroyed at run-time through calls to MPI library routines.

Implementation steps are:

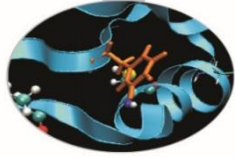
1. Creation of the datatype from existing ones with a **datatype constructor**.
2. Allocation (**committing**) of the datatype before using it.
3. **Usage of the derived datatype** for MPI communications and/or for MPI-I/O
4. Deallocation (**freeing**) of the datatype after that it is no longer needed.

Some derived datatype constructors



- `MPI_TYPE_CONTIGUOUS` contiguous datatype
- `MPI_TYPE_VECTOR` regularly spaced datatype
- `MPI_TYPE_CREATE_HVECTOR` like vector, but the stride is specified in byte
- `MPI_TYPE_INDEXED` variably spaced datatype
- `MPI_TYPE_CREATE_HINDEXED` like indexed, but the stride is specified in byte
- ... (many other)
- `MPI_TYPE_CREATE_SUBARRAY` subarray within a multidimensional array
- `MPI_TYPE_CREATE_STRUCT` fully general datatype

MPI_TYPE_CONTIGUOUS



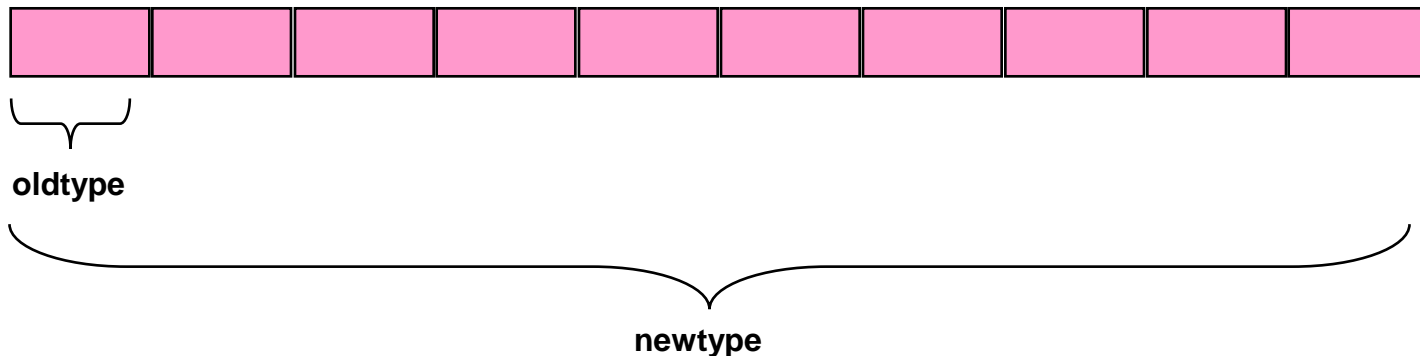
`MPI_TYPE_CONTIGUOUS (count, oldtype, newtype)`

IN `count`: replication count (non-negative integer)

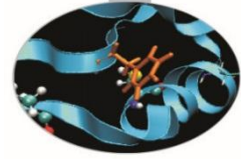
IN `oldtype`: old datatype (handle)

OUT `newtype`: new datatype (handle)

- `MPI_TYPE_CONTIGUOUS` constructs a typemap consisting of the **replication** of a **datatype** into contiguous locations.
- `newtype` is the datatype obtained by concatenating `count` copies of `oldtype`.



Example



```
count = 4;  
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

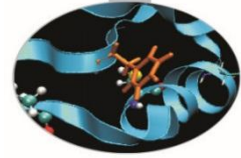
a[4][4]

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

9.0	10.0	11.0	12.0
-----	------	------	------

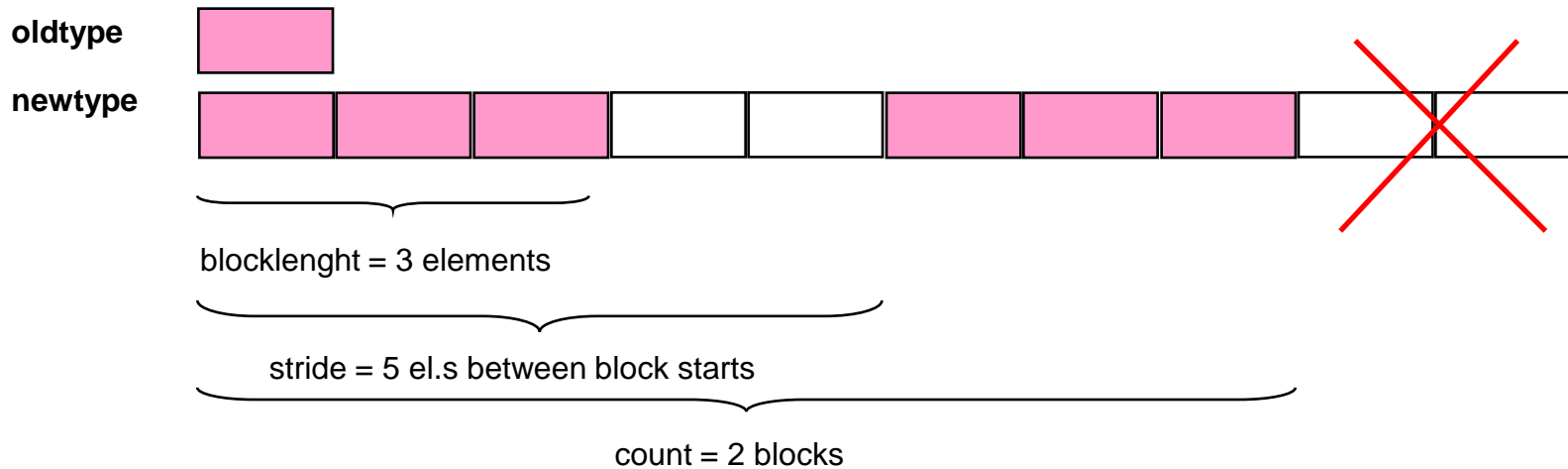
1 element of rowtype

MPI_TYPE_VECTOR

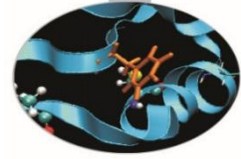


MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype)
 IN count: Number of blocks (non-negative integer)
 IN blocklen: Number of elements in each block
 (non-negative integer)
 IN stride: Number of elements (NOT bytes) between start of
 each block (integer)
 IN oldtype: Old datatype (handle)
 OUT newtype: New datatype (handle)

- Consist of a number of elements of the same datatype repeated with a certain stride



Example



```
count = 4; blocklength = 1; stride = 4;  
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,  
                &column_type);
```

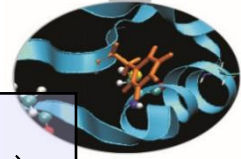
1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[0][1], 1, column_type, dest, tag, comm);
```

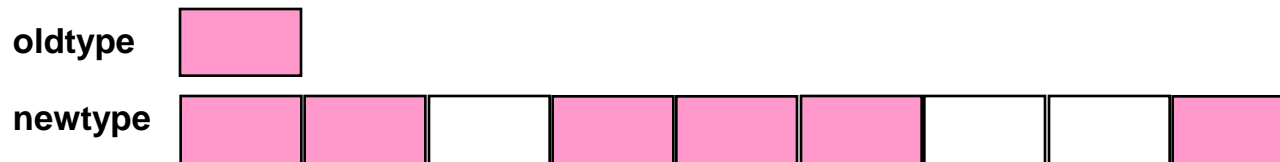
2.0	6.0	10.0	14.0
-----	-----	------	------

1 element of
column_type



MPI_TYPE_INDEXED (count, array_of_blocklengths, array_of_displacements, oldtype, newtype)
 IN count: number of blocks - also number of entries in
 array_of_blocklengths and array_of_displacements
 (non-negative integer)
 IN array_of_blocklengths: number of elements per block
 (array of non-negative integers)
 IN array_of_displacements: displacement for each block, in multiples of oldtype extent
 (array of integer)
 IN oldtype: old datatype (handle)
 OUT newtype: new datatype (handle)

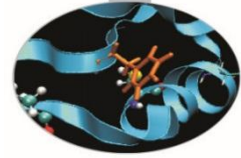
- Creates a new type from blocks comprising identical elements
- The size and displacements of the blocks can vary



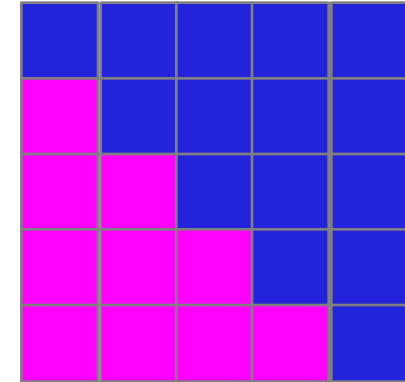
count=3, array_of_blocklengths=(/2,3,1/), array_of_displacements=(/0,3,8/)

<ggd

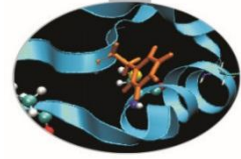
Example 2



```
/* upper triangular matrix */  
double a[100][100];  
int displ[100], blocklen[100], int i;  
MPI_Datatype upper;  
/* compute start and size of the rows */  
for (i=0; i<100; i++){  
    displ[i] = 100*i+i;  
    blocklen[i] = 100-i;  
}  
/* create and commit a datatype for upper triangular matrix */  
MPI_Type_indexed (100, blocklen, displ, MPI_DOUBLE, &upper);  
MPI_Type_commit (&upper);  
/* ... send it ... */  
MPI_Send (a, 1, upper, dest, tag, MPI_COMM_WORLD);  
MPI_Type_free (&upper);
```



Size vs. extent of a datatype



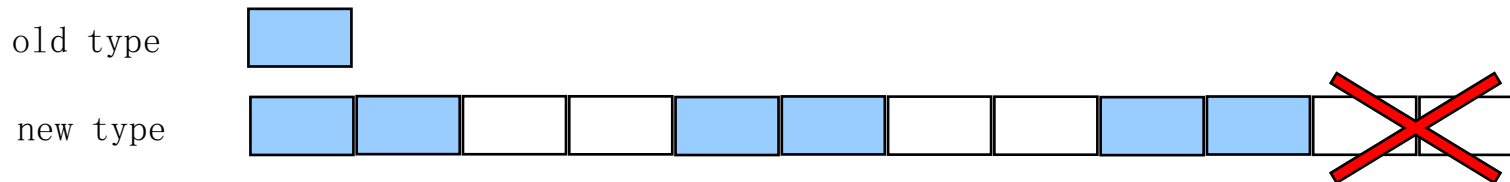
Basic datatypes:

- size = extent = number of bytes used by the compiler

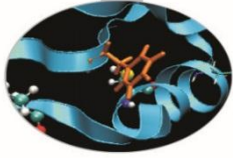
Derived datatypes:

- extent include holes but...
- ... final holes are a figment of our imagination

`MPI_TYPE_VECTOR (3, 2, 4, old_type, new_type)`



- size = 6 x size of “old type”
- extent = 10 x extent of “old type”



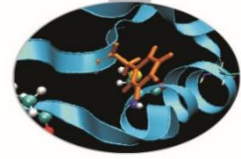
```
MPI_TYPE_SIZE (datatype, size)
  IN datatype: datatype (handle)
  OUT size: datatype size (integer)
```

- Returns the total number of bytes of the entry datatype

```
MPI_TYPE_GET_EXTENT (datatype, lb, extent)
  IN datatype: datatype to get information on(handle)
  OUT lb: lower bound of datatype (integer)
  OUT extent: extent of datatype (integer)
```

- Returns the lower bound and the extent of the entry datatype

Use of Datatype in Communications



- The extent of a datatype controls how the datatype is used with the `count > 1` field in MPI communications

```
call MPI_Send(buf,count,datatype,...)
```

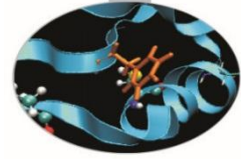
- What does actually send?

```
do i=0,count-1  
  call MPI_Send(bufb(1+i*extent(datatype)),1,datatype,...)  
enddo
```

where *bufb* is a byte type like *integer*1*

- *extent* is used to decide where to send from or receive to for `count>1` datatype communications
 - Normally, this is right after the last byte used for `(i-1)`

Use of Datatype in Communications

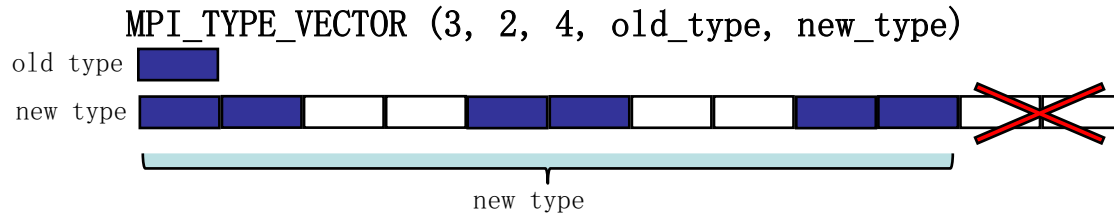


- Were we to use a derived datatype (i.e: MPI_TYPE_VECTOR) in a communication with count>1 of this derived datatype

`MPI_Send (buffer, 1000, new_type,)`
...what we want



- Remember that trailing holes are discarded in datatype creation:

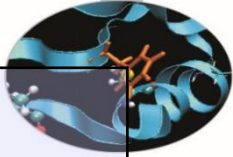


- beware of using communication of derived datatypes with count > 1

...what we get ☹️

`MPI_Send (buffer, 1000, new_type,)`





```
MPI_TYPE_CREATE_RESIZED (oldtype, newlb, newextent, newtype)
```

```
IN oldtype: input datatype (handle)
```

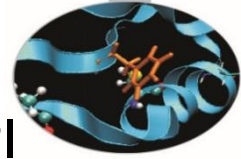
```
IN newlb: new lower bound of datatype (integer, in terms of bytes)
```

```
IN newextent: new extent of datatype (integer, in term of bytes)
```

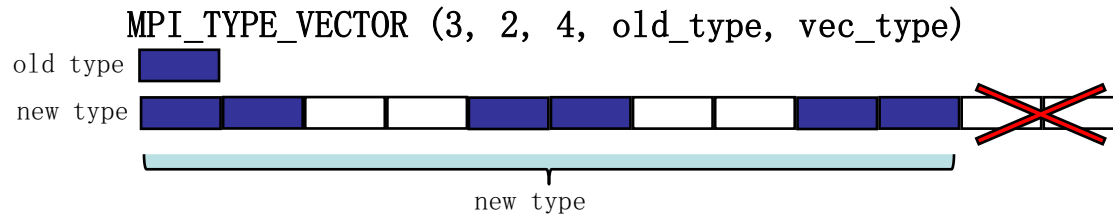
```
OUT newtype: output datatype (handle)
```

- Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be “lb”, and its upper bound is set to be “lb + extent”.
- Modifying extent is useful to handle alignments of the last items of structs
 - crucial when communicating more than one derived data-type
- Modifying also the lower bound can be confusing, use with particular care

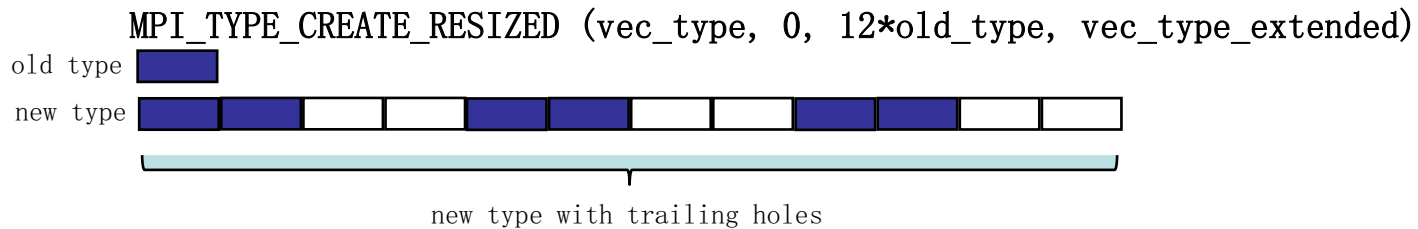
Extend Datatype for Communications



- If you want to use a derived datatype as a basic template in MPI communications, remember to take extent into account
 - trailing holes are discarded in datatype creation



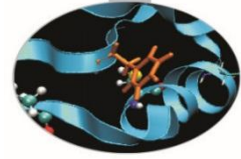
- extend the created datatype with the `MPI_TYPE_CREATE_RESIZED`



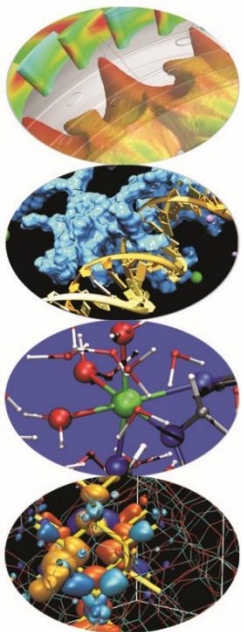
- performe MPI communication as usual

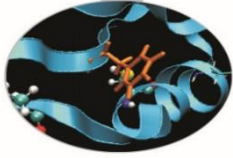
`MPI_Send (buffer, 1000, vec_type_extended,)`



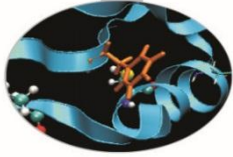


MPI2-IO Advanced Features

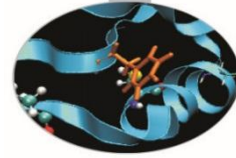




- Basic MPI-IO features are not useful when
 - Data distribution is non contiguous in memory and/or in the file
 - e.g., ghost cells
 - e.g., block/cyclic array distributions
 - Multiple read/write operations for segmented data generate poor performances
- MPI-IO allow to access to data in different way:
 - non contiguous access on file: providing the access pattern to file (fileview)
 - non contiguous access in memory: setting new MPI derived datatype
 - collective access: grouping multiple near accesses in one or more single accesses (decreasing the latency time)



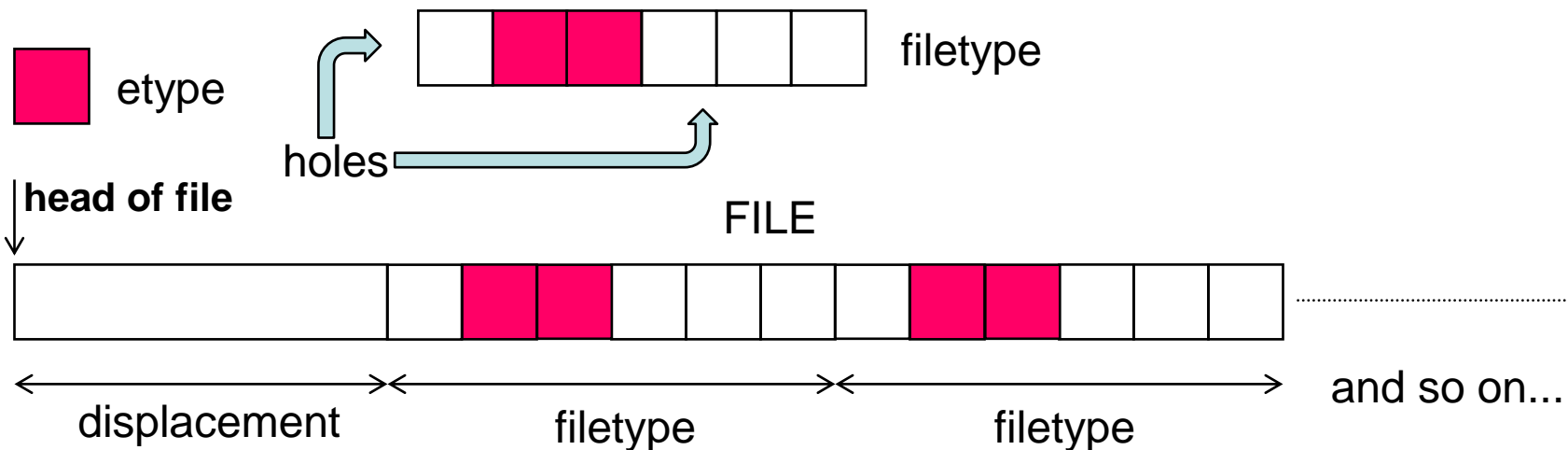
- A file view defines which portion of a file is “visible” to a process
- File view defines also the type of the data in the file (byte, integer, float, ...)
- By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file
- A default view for each participating process is defined implicitly while opening the file
 - No displacement
 - The file has no specific structure (The elementary type is `MPI_BYTE`)
 - All processes have access to the complete file (The file type is `MPI_BYTE`)



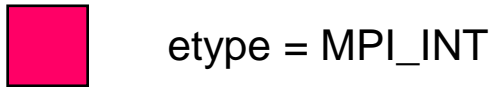
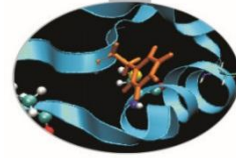
A file view consists of three components

- **displacement** : number of **bytes** to skip from the beginning of file
- **etype** : type of data accessed, defines unit for offsets
- **filetype** : base portion of file visible to a process

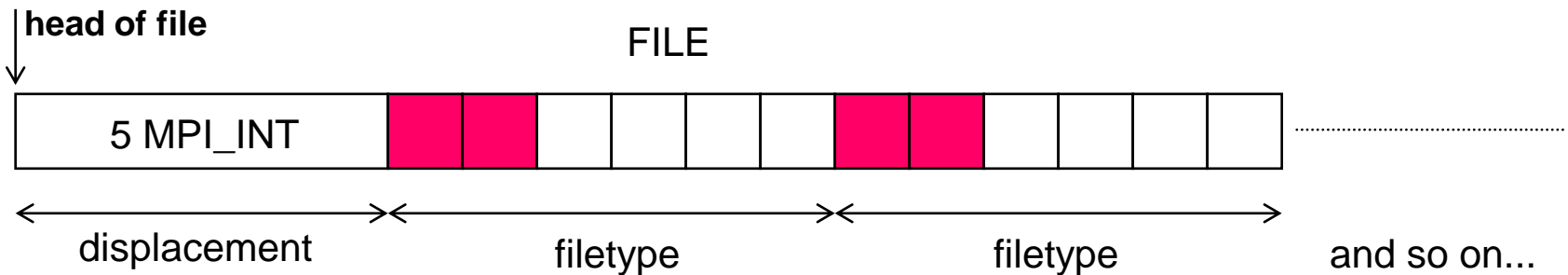
The pattern described by a filetype is repeated, beginning at the displacement, to define the view, as it happens when creating `MPI_CONTIGUOUS` or when sending more than one MPI datatype element: HOLES are important!



File View Example



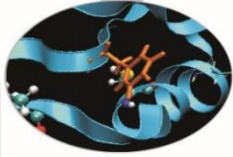
filetype = two MPI_INTs followed by a gap of four MPI_INTs



```
MPI_Datatype contiguous_t;
MPI_Type_contiguous(2, MPI_INT, &contiguous_t);
MPI_Type_commit(&contiguous_t);
```

```
MPI_Datatype filetype;
MPI_Type_create_resized(cont_t, 0, 6*sizeof(int), &filetype);
MPI_Type_commit(&filetype);
```

```
MPI_Offset offset = (MPI_Offset) 5 * size_int; // displacement is in bytes
MPI_File_set_view(fh, offset, MPI_INT, filetype, "native", MPI_INFO_NULL);
```



```
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
```

```
INOUT fh: file handle (handle)
```

```
IN disp: displacement from the start of the file, in bytes  
(integer)
```

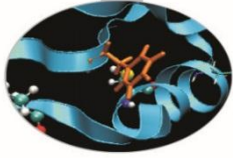
```
IN etype: elementary datatype. It can be either a pre-defined  
or a derived datatype but it must have the same value on  
each process. (handle)
```

```
IN filetype: datatype describing each processes view of the  
file. (handle)
```

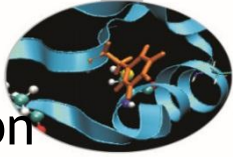
```
IN datarep: data representation (string)
```

```
IN info: info object (handle)
```

- It is used by each process to describe the layout of the data in the file
- All processes in the group must pass identical values for datarep and provide an etype with an identical extent
- The values for disp, filetype, and info may vary

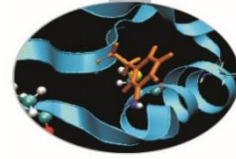


- Data representation: define the layout and data access modes (byte order, type sizes, etc.)
 - **native:** (default) use the memory layout with no conversion
 - no precision loss or conversion effort
 - not portable
 - **internal:** layout implementation-dependent
 - portable for the same MPI implementation
 - **external32:** standard defined by MPI (32-bit big-endian IEEE)
 - portable (architecture and MPI implementation)
 - some conversion overhead and precision loss
 - not always implemented (e.g. Blue Gene/Q)
- Using or internal and external32, the portability is guaranteed only if using the correct MPI datatypes (not using MPI_BYTE)
- **Note: to be portable the best and widespread choice is to use high-level libraries, e.g. HDF5 or NetCDF**

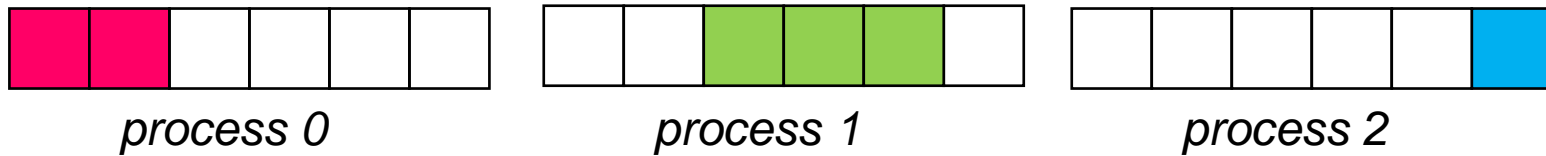


- `MPI_File_set_view` API allows the user to provide information on the features of the File System employed
 - optional (default: use `MPI_INFO_NULL` if you are not very expert)
 - may improve performances
 - depend on the MPI implementation
- Infos are objects created by `MPI_Info_create`
 - elements key-value
 - use `MPI_Info_set` to add elements
- ... refer to standard for more information and to manuals
 - e.g., consider ROMIO implementation of MPICH
 - specific infos for different file-systems (PFS, PVFS, GPFS, Lustre, ...)

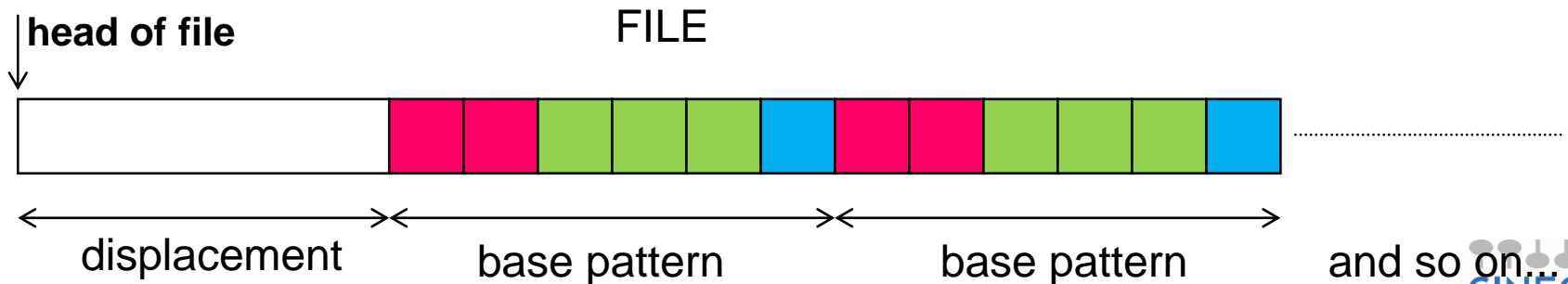
Devising the I/O strategy



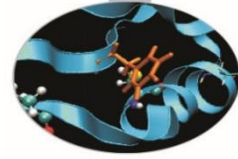
- Three main tasks:
 - let each process write to a different area without overlapping
 - repeat (indefinitely?) a certain basic pattern
 - write after an initial displacement
- Consider the following I/O pattern



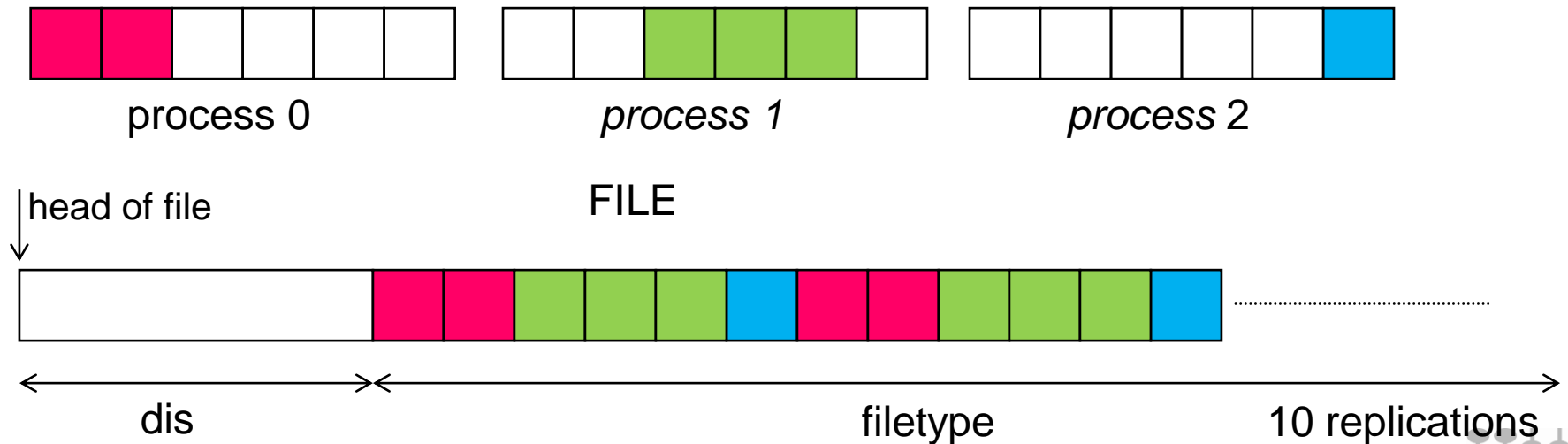
to be replicated a certain amount of (unknown?) times



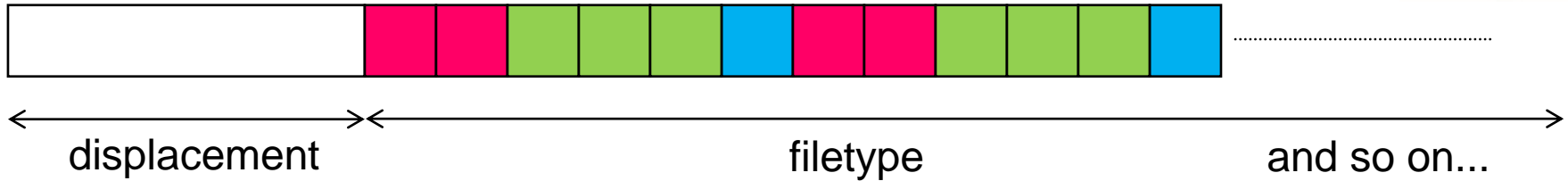
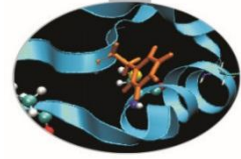
I strategy: data-type replication



- If the whole amount of basic patterns is known (e.g. 10)
 - define MPI vector with count=10, stride=6 and blocklength depending on the process:
 - P0 has 2 elements, P1 has 3 elements, and P2 has 1 element
 - define the file view using different displacements in addition to the base displacement *dis*:
 - *dis+0*, *dis+2*, *dis+5*



Use data-type replication



```

int count_proc[]={2,3,1};
int count_disp[]={0,2,5};
  
```

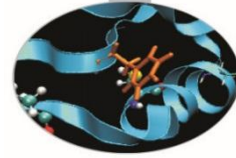
```

MPI_Datatype vect_t;
MPI_Type_vector(DIM_BUF, count_proc[myrank], 6, MPI_INT, &vect_t);
MPI_Type_commit(&vect_t);

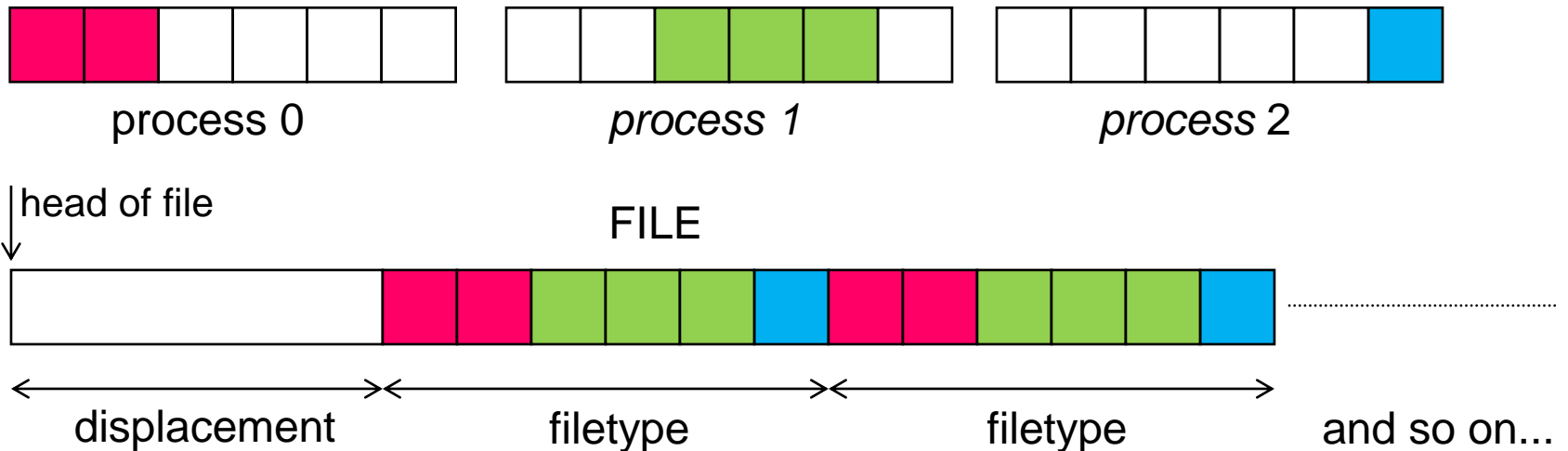
int size_int;
MPI_Type_size(MPI_INT,&size_int);
offset = (MPI_Offset)count_disp[myrank]*size_int;

MPI_File_set_view(fh,offset,MPI_INT,vect_t, "native",MPI_INFO_NULL);
MPI_File_write(fh, buf, my_dim_buf, MPI_INT, &mystatus);
  
```

II strategy: file view replication

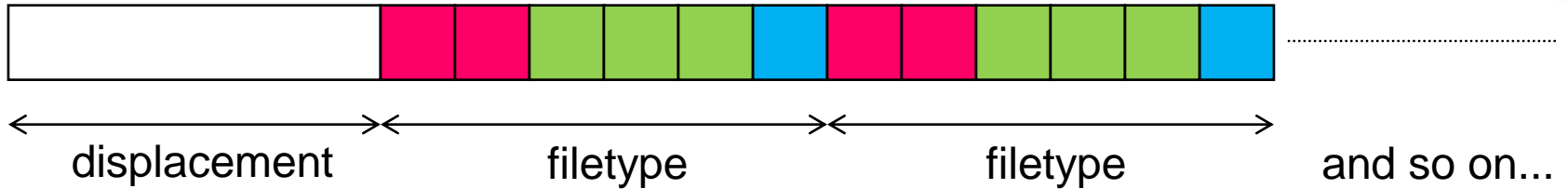
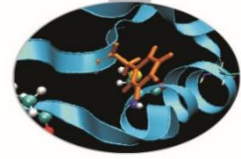


- If the whole amount of basic patterns is unknown, it is possible to exploit the replication mechanism of the MPI file view
 - define MPI contiguous with lengths 2, 3 and 1, respectively
 - resize the types adding holes (on the right only)
 - set the file view with displacements to balance the left holes



- When writing more than a filetype, a replication occurs; as it happens when sending more than one data, setting the holes is crucial!

Use file view replication



```

MPI_Datatype cont_t;
MPI_Type_contiguous(count_proc[myrank], MPI_INT, &cont_t);
MPI_Type_commit(&cont_t);

int size_int;
MPI_Type_size(MPI_INT, &size_int);

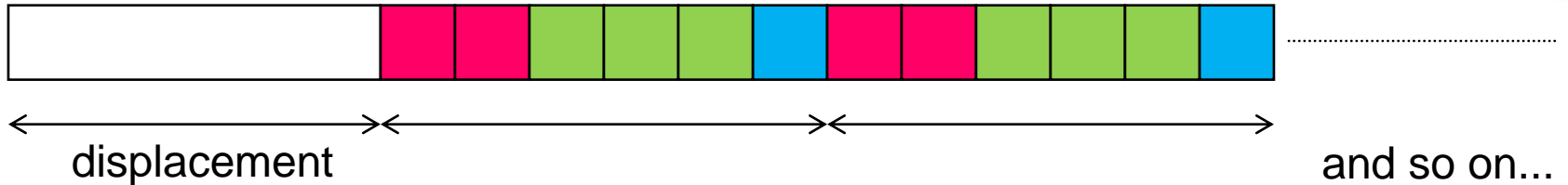
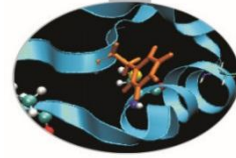
MPI_Datatype filetype;
MPI_Type_create_resized(cont_t, 0, 6*size_int, &filetype);
MPI_Type_commit(&filetype);

offset = (MPI_Offset)count_disp[myrank]*size_int;

MPI_File_set_view(fh, offset, MPI_INT, filetype, "native", MPI_INFO_NULL);

MPI_File_write(fh, buf, my_dim_buf, MPI_INT, &mystatus);
  
```

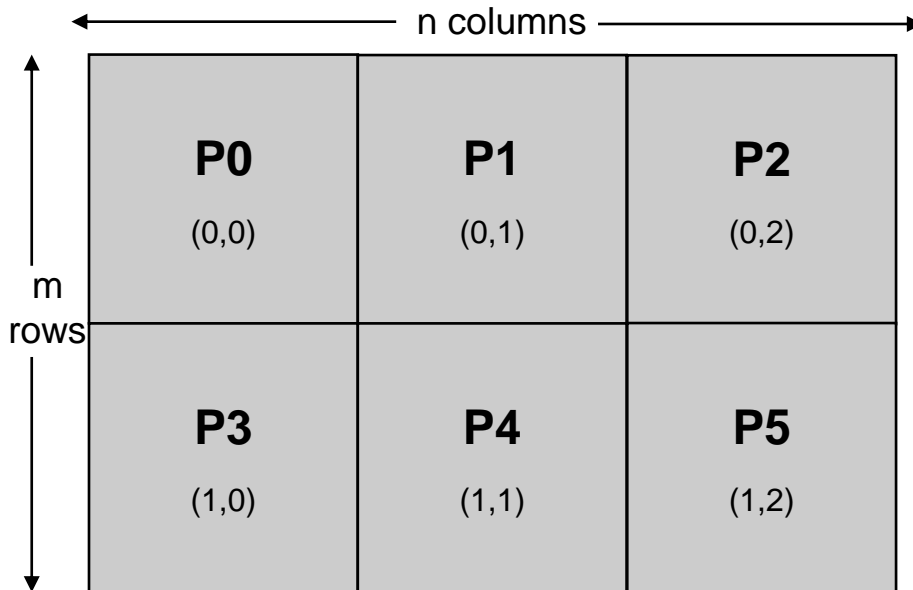
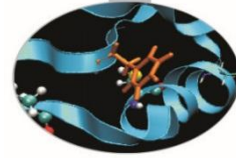
How to replicate patterns?



Which is the best replication strategy?

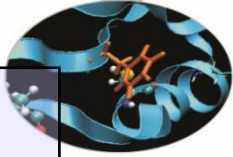
- **If possible, data-type replication is probably better**
 - Surely, easier to be implemented
- **But exploiting file view replication is mandatory when the number of read/writes is not known *a priori***

Non-contiguous access: distributed matrix



- 2D array, size (m,n) distributed among six processes
- cartesian layout 2x3

- When distributing multi-dimensional arrays among processes, we want to write files which are independent of the decomposition
 - written according to a usual serial order
 - in row major order (C) or column major order (Fortran)
- The datatype SUBARRAY may easily handle this situation



`MPI_TYPE_CREATE_SUBARRAY` (`ndims`, `array_of_sizes`, `array_of_subsizes`,
`array_of_starts`, `order`, `oldtype`, `newtype`)

IN `ndims`: number of array dimensions (positive integer)

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

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

IN `array_of_starts`: starting coordinates of the subarray in each
dimension (array of non-negative integers)

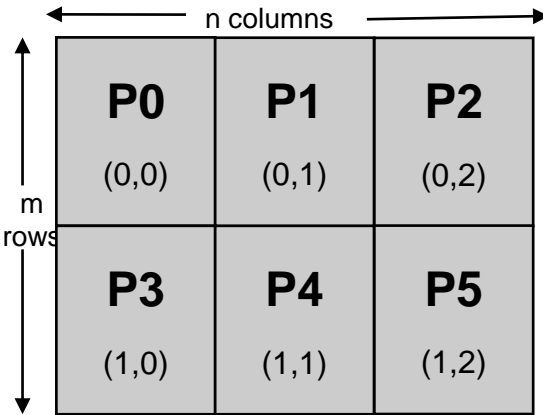
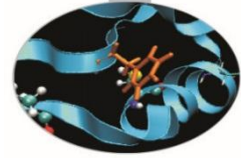
IN `order`: array storage order flag
(state: `MPI_ORDER_C` or `MPI_ORDER_FORTRAN`)

IN `oldtype`: array element datatype (handle)

OUT `newtype`: new datatype (handle)

The subarray type constructor creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array.

Non-contiguous access: distributed matrix



`MPI_Type_create_subarray` (ndims, array_of_sizes,
array_of_subsizes, array_of_starts, order, oldtype, newtype)

```

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

psizes[0] = 2; /* no. of procs. in vertical dimension */
psizes[1] = 3; /* no. of procs. in horizontal dimension */

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

MPI_Cart_coords(comm, rank, 2, coords);

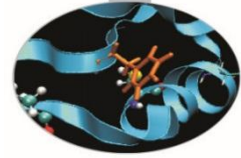
/* global indices of first element of local array */
start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];

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

```

The subarray type constructor creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array.

Non-contiguous access: distributed matrix



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

psizes[0] = 2; /* no. of procs. in vertical dimension */
psizes[1] = 3; /* no. of procs. in horizontal dimension */

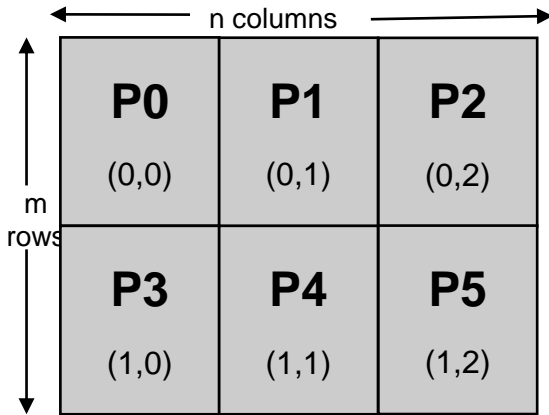
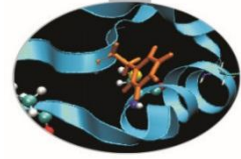
lsizes[0] = m/psizes[0]; /* no. of rows in local array */
lsizes[1] = n/psizes[1]; /* no. of columns in local array */

MPI_Cart_coords(comm, rank, 2, coords);

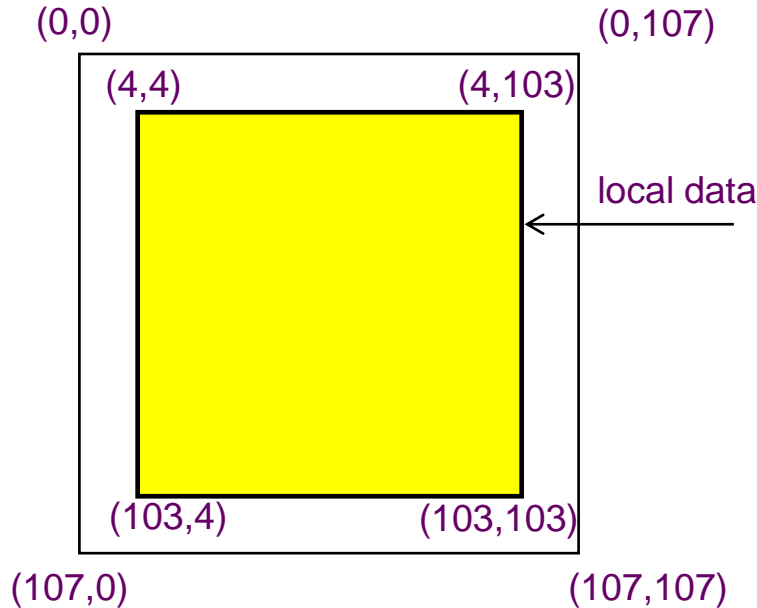
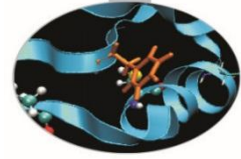
/* global indices of first element of local array */
start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];

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

Non-contiguous access: distributed matrix

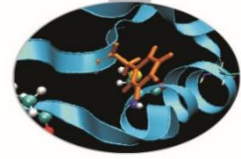


```
/* create filetype and set fileview as in subarray example */  
MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,  
                        MPI_ORDER_C, MPI_FLOAT, &filetype);  
MPI_Type_commit(&filetype);  
  
/* write local data as one big new datatype */  
MPI_File_write(fh, local_array, 1, MPI_FLOAT, &status);  
  
...  
  
MPI_File(&fh, &status);
```



- local array with sizes (100,100) allocated with sizes (108,108) to store
- ghost areas along edges
- ghost area are filled with neighbouring processes data
- local data are stored from position (4,4)
- non-contiguous memory access is needed

- Local data may be considered as a subarray
- Using `MPI_Type_create_subarray` we can filter the local data creating a subarray for accessing only required data to be taken from memory
- This type will be used as access basic type to communicate or to perform I/O



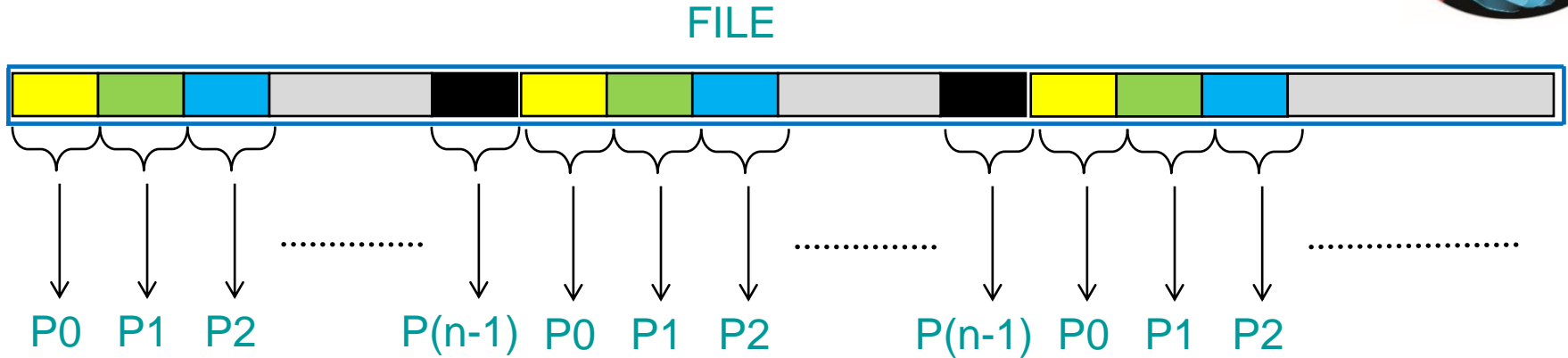
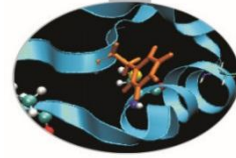
```
/* create a derived datatype describing the layout of local array in memory
   buffer that includes ghosts .This is just another sub-array datatype! */
memsizes[0] = lsizes[0] + 8; /* rows in allocated array */
memsizes[1] = lsizes[1] + 8; /* columns in allocated array */

/* indices of first local elements in the allocated array */
start_indices[0] = start_indices[1] = 4;

MPI_Type_create_subarray(2, memsizes, lsizes, start_indices,
                        MPI_ORDER_C, MPI_FLOAT, &memtype);
MPI_Type_commit(&memtype);

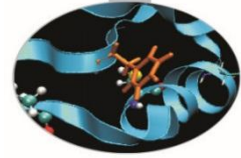
/* create filetype and set fileview as in subarray example */
...
/* write local data as one big new datatype */
MPI_File_write(fh, local_array, 1, memtype, &status);
```

Efficient non-contiguous IO with collective operations



- In general, accessing non-contiguous chunk of data from each process to storage results in very expensive operation and poor performance
 - many access requests can really knock out file systems servers
 - although MPI fileview ease the way of accessing to a file, if the fileview is not-contiguous, also accesses will be not-contiguous

- MPI-IO provides collective operation to gather fragmented accesses into fewer contiguous access operations, improving performance



IO can be performed **collectively** by all processes in a communicator

Same parameters as in independent IO functions (MPI_File_read etc)

- MPI_File_read_all
- MPI_File_write_all

- MPI_File_read_at_all
- MPI_File_write_at_all

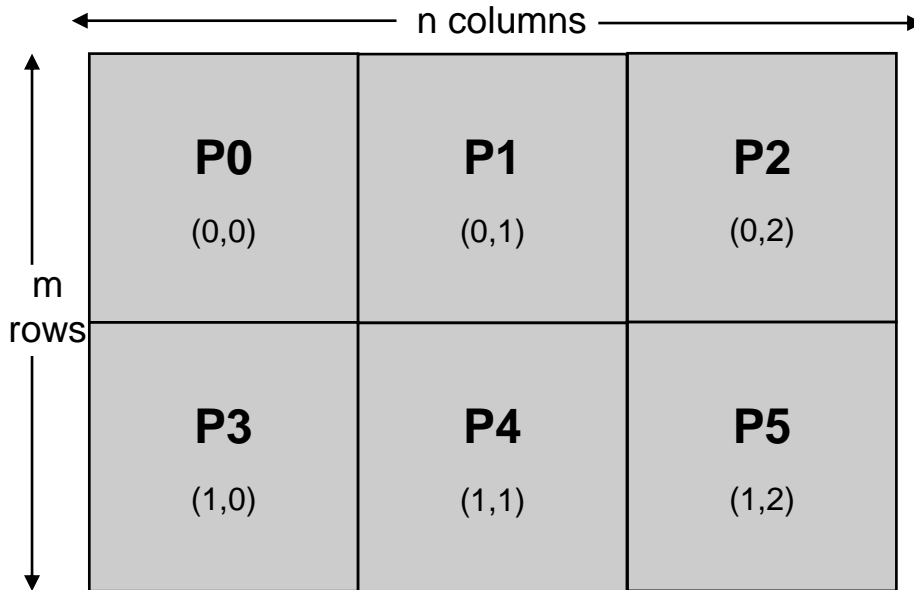
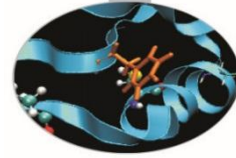
- MPI_File_read_ordered
- MPI_File_write_ordered

All processes in communicator that opened file must call function

Performance are often better than for individual functions

- Even if each processor reads a non-contiguous segment, in total the read is contiguous

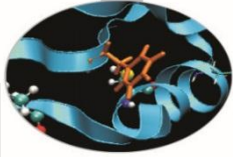
Simple Benchmark



- **Traditional I/O:** master process gathers data and perform I/O
- **MPI-IO:** use `MPI_Type_create_subarray` to define the view for each process and perform a collective call
- local grid (per process): 10000x10000 double-precision

processi	1	2	8	16	32
filesize (Mb)	763	1526	6103	12207	24414
Traditional-IO (s)	8	22	86	1738	3570
MPI-IO (s)	2	4	18	33	48

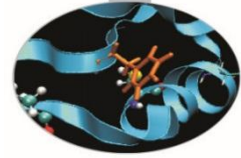
I/O performances are strongly affected by file-system, storage infra-structure, MPI implementation, network,...



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

```
int MPI_File_read_all( MPI_File mpi_fh, void *buf, int count,  
MPI_Datatype datatype, MPI_Status *status )
```

- With collective IO **ALL** the processors defined in a communicator execute the IO operation
- This allows to optimize the read/write procedure
- It is particularly effective for non atomic operations



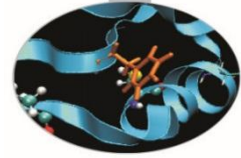
This is just like non blocking communication.

Same parameters as in blocking IO functions (MPI_File_read etc)

- MPI_File_iread
- MPI_File_iwrite
- MPI_File_iread_at
- MPI_File_iwrite_at
- MPI_File_iread_shared
- MPI_File_iwrite_shared

MPI_Wait must be used for synchronization.

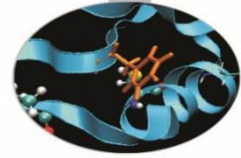
Can be used to overlap IO with computation



For collective IO only a restricted form of nonblocking IO is supported, called Split Collective.

```
MPI_File_read_all_begin( MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype )  
  
    ...computation...  
  
MPI_File_read_all_end( MPI_File mpi_fh, void *buf, MPI_Status *status );
```

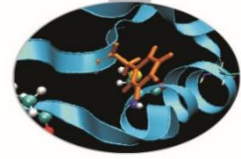
- Collective operations may be split into two parts
- Only one active (pending) split or regular collective operation per file handle at any time
- Split collective operations do not match the corresponding regular collective operation
- Same BUF argument in `_begin` and `_end` calls



- No consistency problems rise when there are no overlapping regions (bytes) accessed by any two processes
- MPI does *not* guarantee that data will automatically read correctly, when accesses of any two processes overlap in the file

```
MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh1)      MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh2)
MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ... )      MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ... )
MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ... )     MPI_File_read_at (fh2, 0, buf, 100, MPI_BYTE, ... )
```

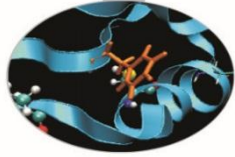
- The user must take care of consistency. There are three choices:
 - using *atomic* accesses
 - close and reopen the file
 - ensure that no “*write sequence*” on any process is concurrent with “*any sequence (read/write)*” on another process



- Change file-access mode to atomic before the write on each process
- MPI guarantees that written data can be read immediately by another process

```
MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh1)      MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh2)
MPI_File_set_atomicity(fh1, 1)                          MPI_File_set_atomicity(fh2, 1)
MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ... )     MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ... )
MPI_Barrier(MPI_COMM_WORLD)                             MPI_Barrier(MPI_COMM_WORLD)
MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ... )    MPI_File_read_at (fh2, 0, buf, 100, MPI_BYTE, ... )
```

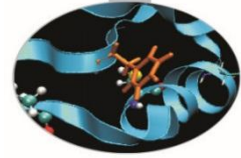
- note: the *barrier* after the writes to ensure each process has completed its write before the read is issued from the other process



- Close the file and reopen just after write operations if other processes need data just written by other processes

```
MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh1)      MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh2)
MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ... )      MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ... )
MPI_File_close(&fh1)                                     MPI_File_close(&fh2)
MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh1)      MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh2)
MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ... )     MPI_File_read_at (fh2, 0, buf, 100, MPI_BYTE, ... )
```

- note: each file open operation will create a different MPI context
context will be cleared after each close operation



- An IO sequence is defined as a set of file operations bracketed by any pair of the functions `MPI_File_open`, `MPI_File_close`, `MPI_File_sync`
- A sequence is a “*write sequence*” if contains *write* operations
- MPI guarantees that the data written by a process can be read by another process if the “*write sequence*” is not concurrent (in time) with any sequence on any other process

```

MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh1)
MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ... )
MPI_File_sync(&fh1)

```

```

MPI_Barrier(MPI_COMM_WORLD)

```

```

MPI_File_sync(&fh1)

```

```

MPI_File_sync(&fh1)

```

```

MPI_Barrier(MPI_COMM_WORLD)

```

```

MPI_File_sync(&fh1)
MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ... )
MPI_File_close(&fh1)

```

```

MPI_File_open(MPI_COMM_WORLD, "file", ....., &fh2)

MPI_File_sync(&fh2)

```

```

MPI_Barrier(MPI_COMM_WORLD)

```

```

MPI_File_sync(&fh2)
MPI_File_write_at(fh2, 0, buf, 100, MPI_BYTE, ... )
MPI_File_sync(&fh2)

```

```

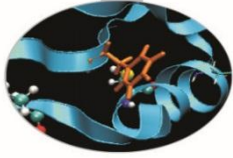
MPI_Barrier(MPI_COMM_WORLD)

```

```

MPI_File_sync(&fh2)
MPI_File_read_at(fh2, 0, buf, 100, MPI_BYTE, ... )
MPI_File_close(&fh2)

```

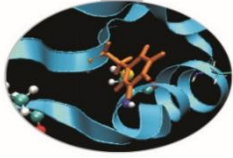



1. Each process has to read in the complete file

- Solution: `MPI_FILE_READ_ALL`
 - Collective with individual file pointers, same view (displacement, etype, filetype) on all processes
 - Internally: read in once from disk by several processes (striped), then distributed broadcast

2. The file contains a list of tasks, each task requires a different amount of computing time

- Solution: `MPI_FILE_READ_SHARED`
 - Non-collective with a shared file pointer
 - Same view on all processes (mandatory)



3. The file contains a list of tasks, each task requires the same amount of computing time

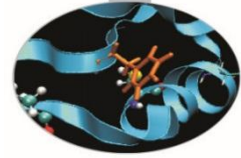
Solution A : `MPI_FILE_READ_ORDERED`

- Collective with a shared file pointer
- Same view on all processes (mandatory)

Solution B : `MPI_FILE_READ_ALL`

- Collective with individual file pointers
- Different views: filetype with `MPI_TYPE_CREATE_SUBARRAY`

Internally: both may be implemented in the same way.



4. The file contains a matrix, distributed block partitioning, each process reads a block

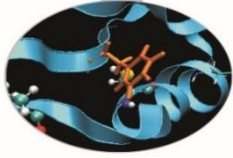
Solution: generate different filetypes with `MPI_TYPE_CREATE_DARRAY`

- The view of each process represents the block that is to be read by this process
- `MPI_FILE_READ_AT_ALL` with `OFFSET=0`
- Collective with explicit offset
- Reads the whole matrix collectively
- Internally: contiguous blocks read in by several processes (striped), then distributed with all-to-all.

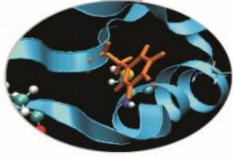
5. Each process has to read the complete file

Solution: `MPI_FILE_READ_ALL_BEGIN/END`

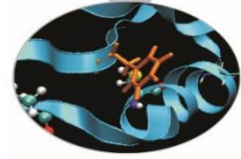
- Collective with individual file pointers
- Same view (displacement, etype, filetype) on all processes
- Internally: asynchronous read by several processes (striped) started, data distributed with bcast when striped reading has finished



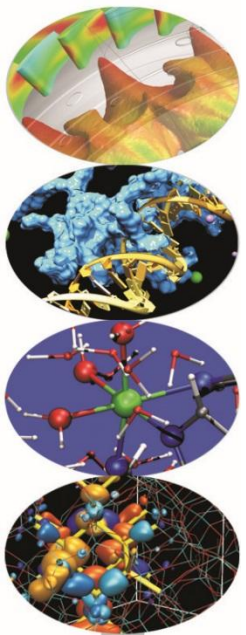
- When designing your code, think I/O carefully!
 - maximize the parallelism
 - if possible, use a single file as restart file and simulation output
 - minimize the usage of formatted output (do you actually need it?)
- Minimize the latency of file-system access
 - maximize the sizes of written chunks
 - use collective functions when possible
 - use derived datatypes for non-contiguous access
- If you are patient, read MPI standards, MPI-2.x or MPI-3.x
- Employ powerful and well-tested libraries based on MPI-I/O:
 - HDF5 or NetCDF



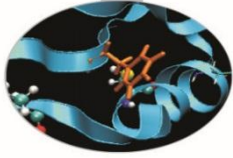
- MPI - The Complete Reference vol.2, The MPI Extensions (W.Gropp, E.Lusk et al. - 1998 MIT Press)
- Using MPI-2: Advanced Features of the Message-Passing Interface (W.Gropp, E.Lusk, R.Thakur - 1999 MIT Press)
- Standard MPI-3.x
(<http://www.mpi-forum.org/docs>)
- Users Guide for ROMIO (Thakur, Ross, Lusk, Gropp, Latham)
- ... a bit of advertising:
corsi@cineca.it (<http://www.hpc.cineca.it>)
- ...practice practice practice



QUESTIONS ???



Hands-on 2: MPI-I/O & subarrays



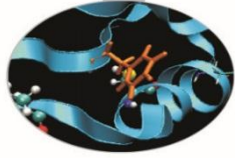
- Write a program which decomposes an integer matrix (m x n) using a 2D MPI Cartesian grid

- Handle the remainders for non multiple sizes
- Fill the matrix with the row-linearized indexes

$$A_{ij} = m \cdot i + j$$

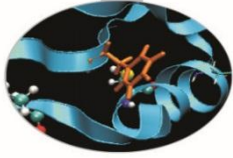
11	12	13
14	15	16
17	18	19
20	21	22

- Reconstruct the absolute indexes from the local ones
 - Remember that in C the indexes of arrays start from 0
- Writes to file the matrix using MPI-I/O collective write and using MPI data-types
 - Which data-type do you have to use?



- Check the results using:
 - Shell Command

```
od -i output.dat
```
 - Parallel MPI-I/O read functions (similar to write structure)
 - Serial standard C and Fortran check
 - only rank=0 performs check
 - read row-by-row in C and column-by-column in Fortran and check each element of the row/columns
 - use binary files and fread in C
 - use unformatted and access='stream' in Fortran
- Which one is the most scrupulous check?
 - is the Parallel MPI-I/O check sufficient?



These slides are copyrighted CINECA 2014 and are released under the Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) Creative Commons license, version 3.0.

Uses not allowed by the above license need explicit, written permission from the copyright owner. For more information see:

<http://creativecommons.org/licenses/by-nc-nd/3.0/>

Slides and examples were authored by:

- Luca Ferraro
- Francesco Salvatore