General Interest Seminar

Parallel Programming: MPI I/O Advanced Features

Jemmy Hu
SHARCNET HPC Consultant
April 23, 2025

Parallel I/O

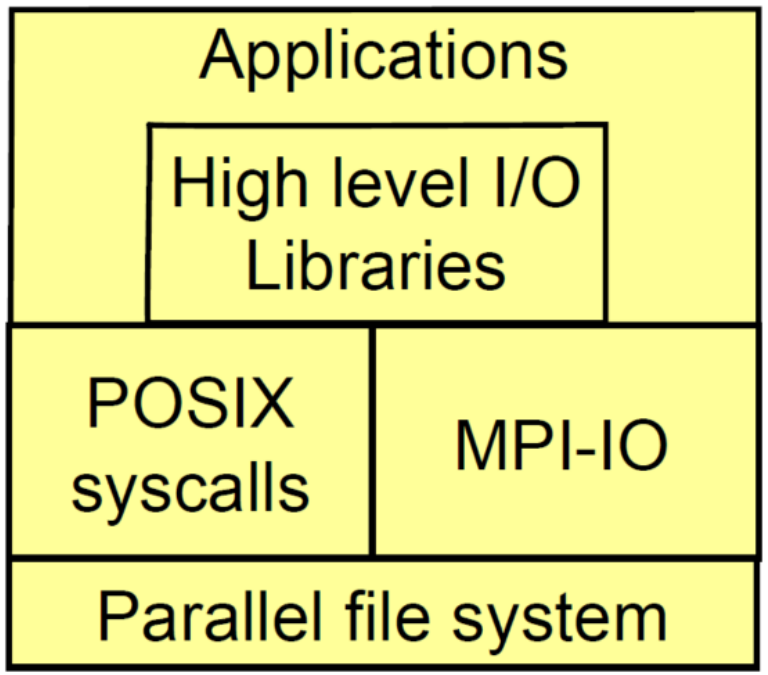
- Deal with very large datasets while running massively parallel applications on supercomputers
- Good I/O is non-trivial, the challenges are
  - performance, scalability, reliability – Ease of use of output (number of files, format)
  - amount of data saved is increased, latency to access to disks is not negligible
  - data portability

- One cannot achieve all of the above - Solutions to managing IO in parallel applications must take into account different aspects of the application and implementation, one needs to decide what is most important

- **At the program level:**
  - Concurrent reads or writes from multiple processes to a common file
- **At the system level:**
  - A parallel file system and hardware that support such concurrent access

IO Layers

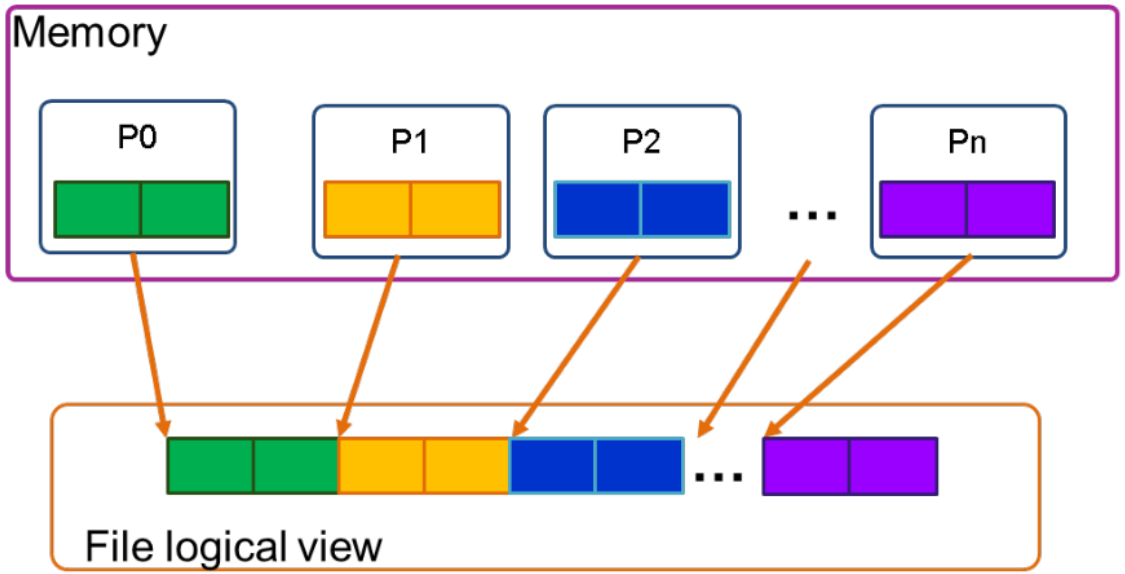
- High-level
  - application: to read or write data from disk
- Intermediate-level
  - high-level libraries
    HDF5, NETCDF
  - libraries or system tools for I/O
- Low-level
  - parallel filesystem enables the actual parallel I/O
  - Lustre, GPFS, PVFS



Managing IO in Parallel Applications: Shared File (Independent writers)

- All processes write to specific blocks of the file, but synchronization is necessary to prevent write conflicts.
- Coordination usually occurs at the parallel file system level.
- Although communication between processes is not required, multiple processes may compete for certain regions of the file, leading to lock contention, hence lower performance.

- High-level I/O libraries such as parallel HDF5 and pNetCDF are commonly used with this approach and help to encapsulate data within the file, but the complexity of managing file regions is not visible to the programmer.



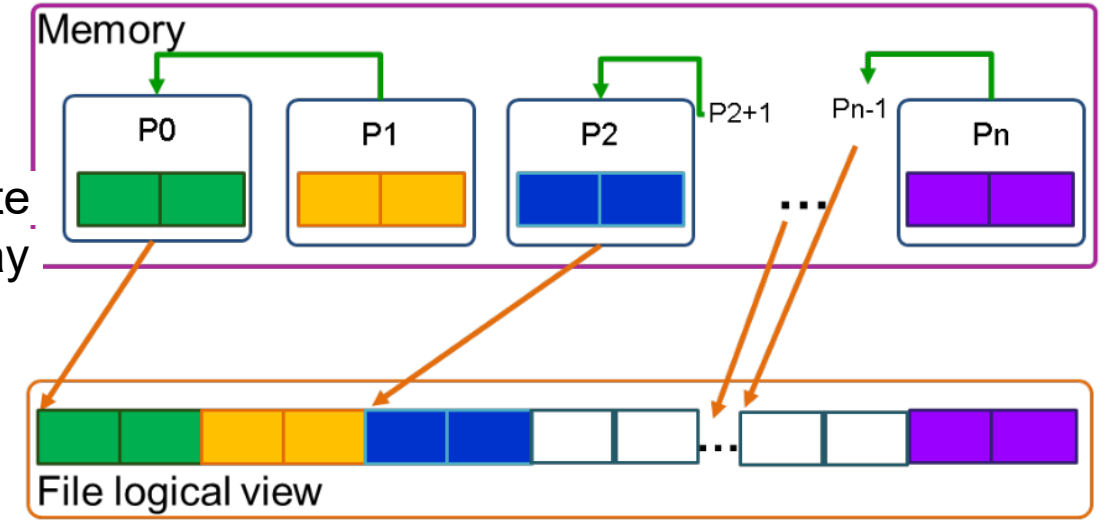
Managing IO in Parallel Applications: Shared File (Collective writers)

- Improve performance in the single file approach

- Data aggregation can be done by introducing *aggregators*

- Chosen processes that collect data from others and write it to specific sections of the file. However, this method may be complex to implement.

- Using a high-level library can make the process of data aggregation and file management much easier, as it typically handles these tasks behind the scenes and allows for a single logical view of the file while also providing control over the aggregation strategy.



MPI I/O

MPI I/O was introduced in MPI-2

- A set of extensions to the MPI library that enable parallel high-performance I/O operations
- Provides a parallel file access interface that allows multiple processes to write and read to the same file simultaneously

- Defines parallel operations for reading and writing files:
  - I/O to only one file and/or to many files
  - Contiguous and non-contiguous I/O
  - Individual and collective I/O
  - Asynchronous I/O

- Portable programming interface
- Efficient data transfer between processes, Potentially good performance
  - Enables high-performance I/O operations on large datasets
  - Used as the backbone of many parallel I/O libraries such as parallel NetCDF and parallel HDF5

Basic concepts in MPI I/O

- File handle
  - data structure which is used for accessing the file
- File pointer
  - position in the file where to read or write
  - can be individual for all processes or shared between the processes
  - accessed through file handle
- File view
  - part of a file which is visible to process
  - enables **efficient** non-contiguous access to file
- Collective and independent I/O
  - collective: MPI coordinates the reads and writes of processes
  - independent: no coordination by MPI

Basic MPI-IO Operations

- 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)
  - it is possible to know the exit status of each subroutine/function

File Open -API

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

```
MPI_File_open(comm, filename, amode, info, fh, ierr)

Character(*) :: filename
Integer :: comm, amode, info, fh, ierr
```

- A collective call for all processes in a communicator to open a file
- Comm: communicator that performs parallel I/O, typically use MPI_COMM_WORLD
- amode: file access mode, MPI_MODE_RDONLY, MPI_MODE_WRONLY,MPI_MODE_CREATE, MPI_MODE_RDWR, …
- Info: Hints to implementation for optimal performance (No hints: MPI_INFO_NULL)
- fh: parallel file handle

File seek (file pointer)

- Position in the file where to read or write
- Can be individual for all processes or shared between the processes
- Each process moves its local file pointer (individual file pointer) with

```
MPI_File_seek(fh, disp, whence)
```

fh: file handle, data structure which is used for accessing the file

disp: Displacement in bytes (with default file view)

whence: 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

File reading

- Read file at individual file pointer

```
MPI_File_read(fhandle, buf, count, datatype, status)
```

buf: Buffer in memory where to read the data

count: number of elements to read

datatype: datatype of elements to read

status: similar to status in MPI_Recv, amount of data read can be determined by MPI_Get_count

– Updates position of file pointer after reading

– Not thread safe

File writing

- Similar to reading
  - File opened with MPI_MODE_WRONLY or MPI_MODE_CREATE

- Write file at individual file pointer

```
MPI_File_write(fhandle, buf, count, datatype, status)
```

  - Updates position of file pointer after writing
  - Not thread safe

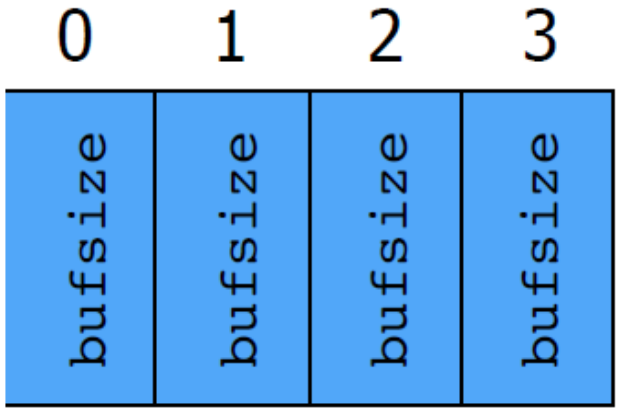
Parallel read: example in C

```
#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, “../datafile”, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

   MPI_Offset filesize;
   MPI_File_get_size(fh, &filesize);
   MPI_Offset 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);
  MPI_Finalize();
}
```



File offset determined by MPI_File_seek

MPI-IO Advanced features

Contiguous vs. Non-contiguous IO

File view (set_view, get_view)

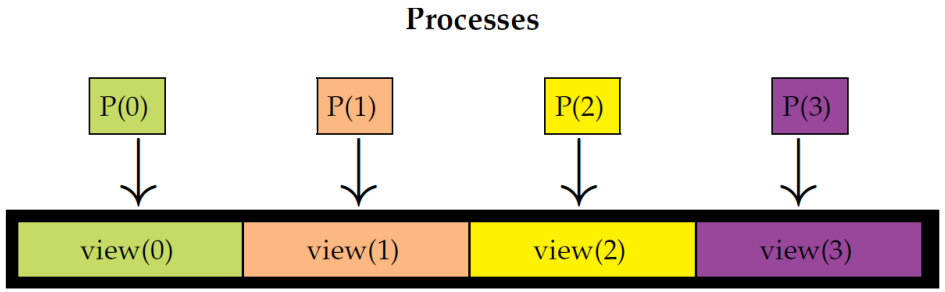
Hints (info)

Independent vs Collective IO

Blocking vs Non-blocking IO

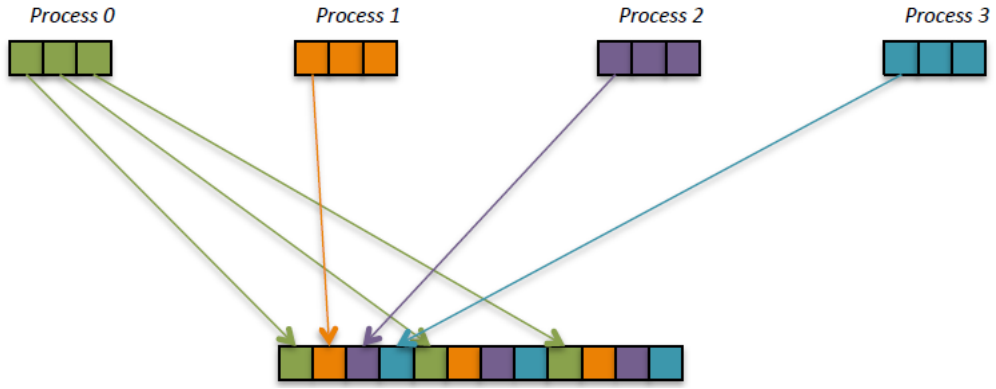
Contiguous vs Non-contiguous I/O

Contiguous I/O moves data from a single memory block into a single file region



Noncontiguous I/O has three forms:

- Noncontiguous in memory
- Noncontiguous in file
- Noncontiguous in both



Structured data leads naturally to noncontiguous I/O (e.g., block decomposition)

File view

• A file view defines which portion of a file is “visible” to a process

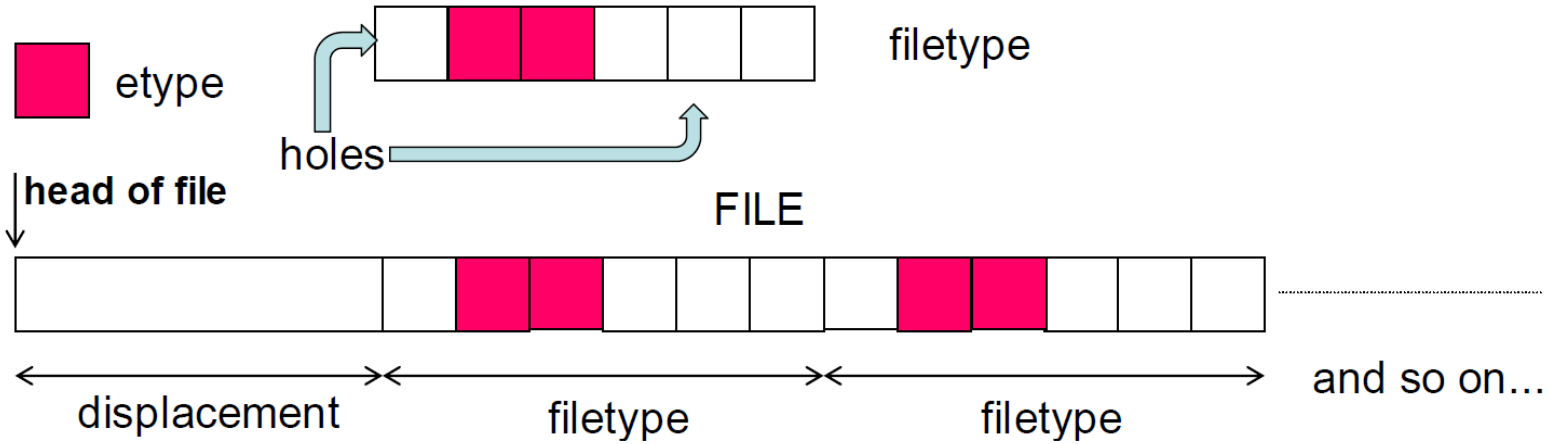
• File view defines also the type of the data in the file (byte, integer, float, …)

• By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file

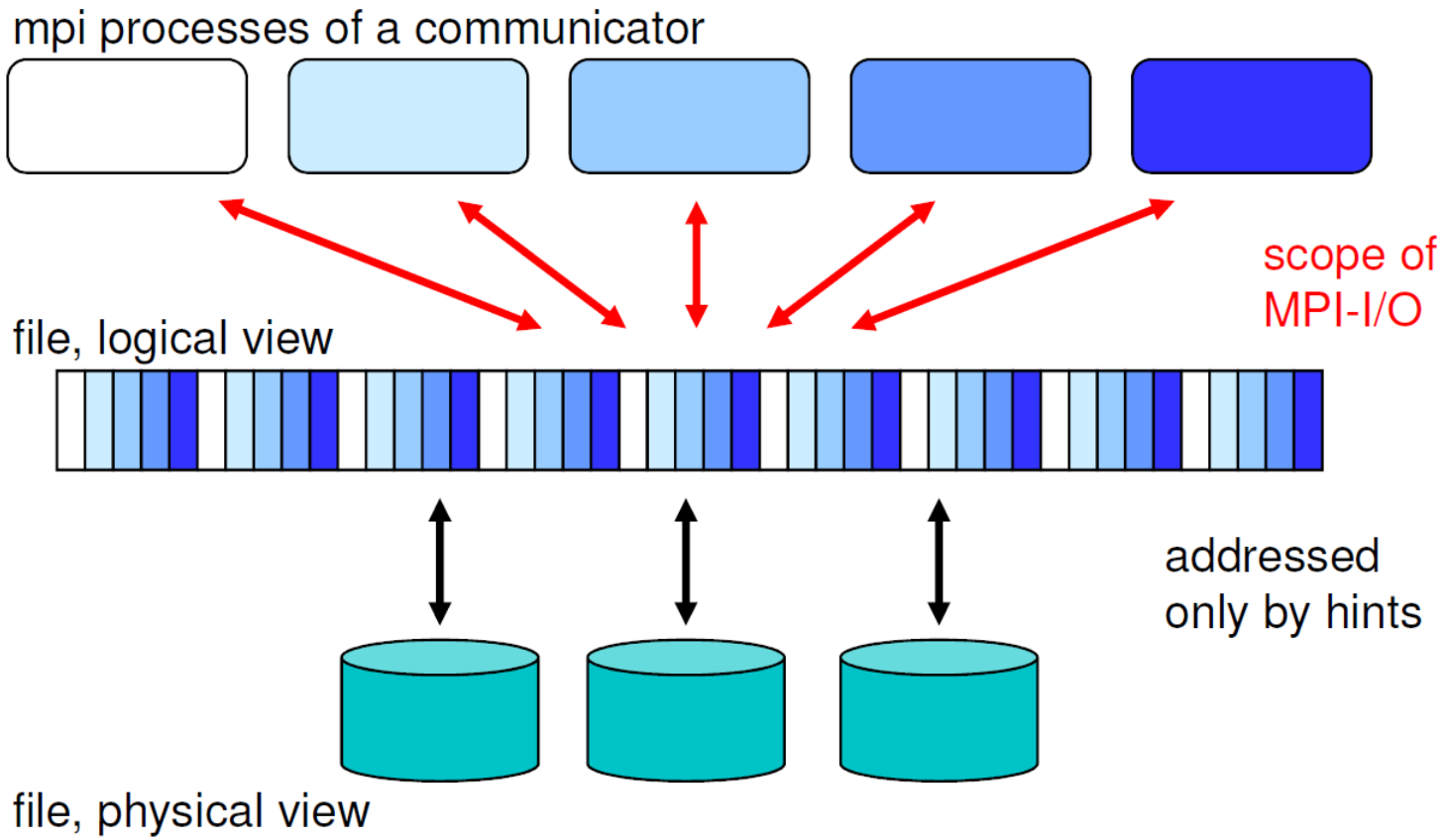
• A default view for each participating process is defined implicitly while opening the file
  – No displacement
  – The file has no specific structure (The elementary type is MPI_BYTE)
  – All processes have access to the complete file (The file type is MPI BYTE)

File view, simple example

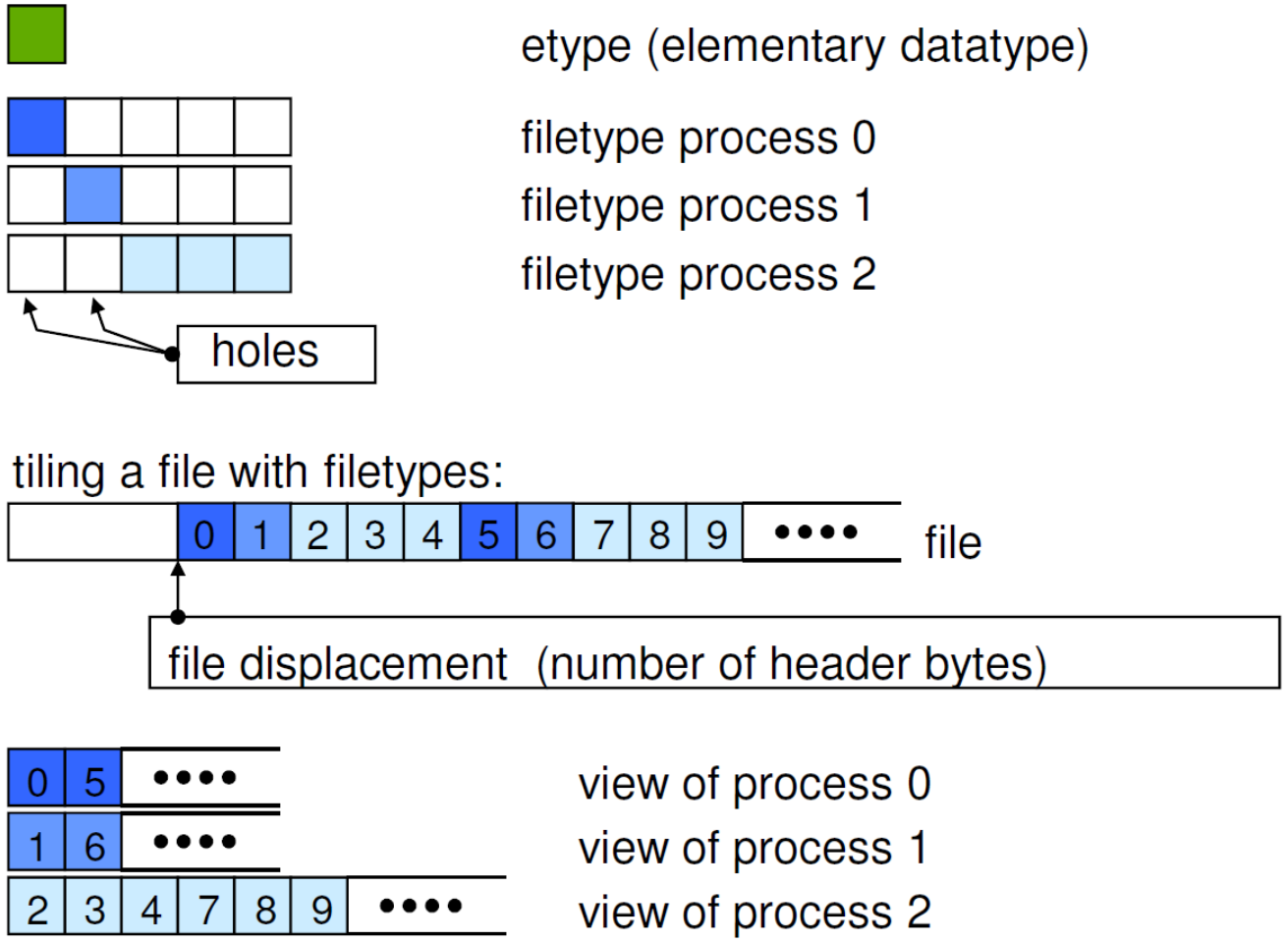
A file view consists of three components
– displacement : number of bytes to skip from the beginning of file
– etype : type of data accessed, defines unit for offsets
– filetype : base portion of file visible to process same as etype or MPI derived type consisting of etype



File View with Multi-Processes



File View with Multi-Processes Example



MPI_File_set(get)_view -API

MPI_File_set_view(fh, disp, etype, filetype, datarep, info)
MPI_FILE_get_view(fh, disp, etype, filetype, datarep)

disp Offset from beginning of file. Always in bytes
etype Basic MPI type or user defined type, Basic unit of data access Offsets in I/O commands in units of etype
filetype Same type as etype or user defined type constructed of etype, Specifies which part of the file is visible
datarep a string that specifies the format in which data is written to a file: "native", "internal", "external32", or user-defined
info Hints for implementation that can improve performance MPI_INFO_NULL: No hints

- 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

Get view – returns the process's view of the data

File set view example: contiguous data

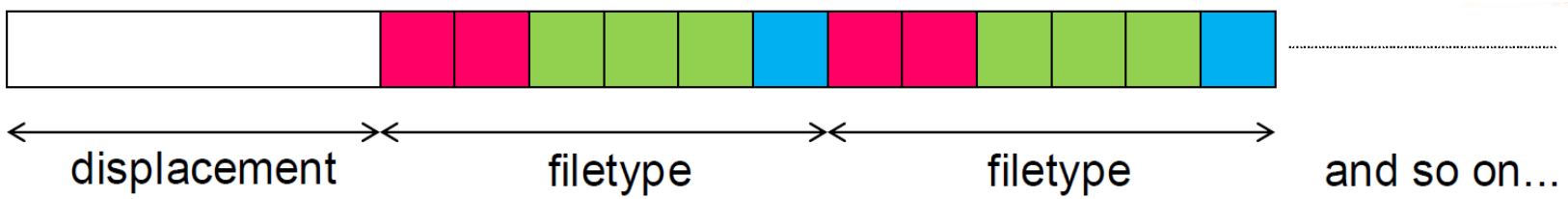
```
int rank, i; char a[10];
MPI_Offset n = 10; MPI_File fh ; MPI_Status status ;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

for (i=0; i<10; i++)
    a[i] = (char)( '0' + rank); // e.g. on processor 3 creates a[0:9]='3333333333'

MPI_File_open (MPI_COMM_WORLD, "data.out" , MPI_MODE_CREATE|MPI_MODE_WRONLY,
MPI_INFO_NULL, &fh);
MPI_Offset displace = rank*n*sizeof(char); // start of the view for each processor
MPI_File_set_view (fh , displace , MPI_CHAR, MPI_CHAR, "native" ,MPI_INFO_NULL);
MPI_File_write(fh, a, n, MPI_CHAR, &status);
MPI_File_close(&fh ) ;
```

0000000000111111111122222222223333333333

File set view example, non-contiguous data replication



– 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
  • *dis*: *dis*+0, *dis*+2 and *dis*+5

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

MPI derived filetype/data constructors

Function	Creates a . . .
MPI_Type_contiguous	contiguous datatype
MPI_Type_vector	vector (strided) datatype
MPI_Type_indexed	indexed datatype
MPI_Type_indexed_block	indexed datatype w/uniform block length
MPI_Type_create_struct	structured datatype
MPI_Type_create_resized	type with new extent and bounds
MPI_Type_create_darray	distributed array datatype
MPI_Type_create_subarray	n-dim subarray of an n-dim array
…	. . .

MPI Datatype example: MPI_Type_vector

```
int MPI_Type_vector( int count, int blocklength, int stride, MPI_Datatype old_type,
MPI_Datatype *newtype_p );
```

MPI_Type_vector is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

Count: number of blocks (nonnegative integer)

Blocklength: number of elements in each block (nonnegative integer)

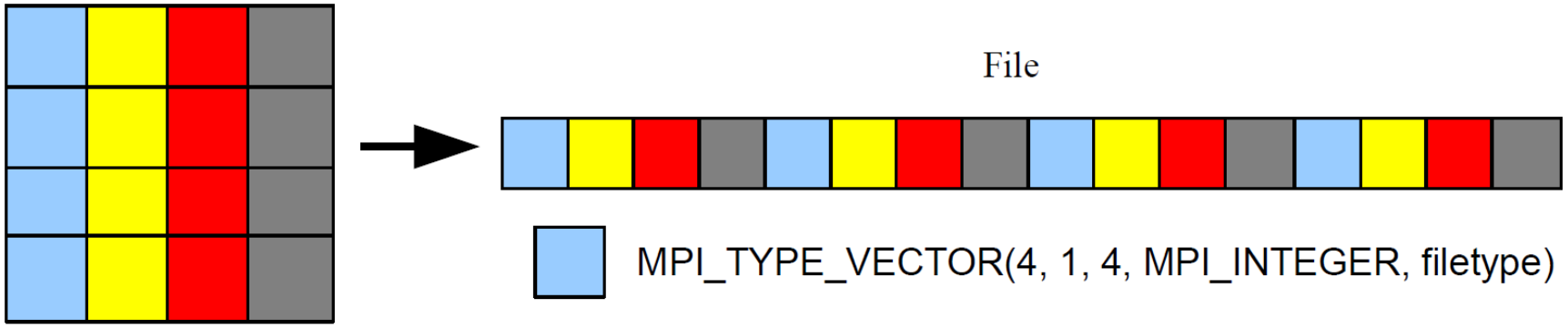
Stride: number of elements between start of each block (integer)

Oldtype: old datatype (handle)

newtype_p: new datatype (handle)

MPI vector datatype example

2D-array distributed column-wise




```
...
 INTEGER :: count = 4
 INTEGER, DIMENSION(count) :: buf
 ...
 CALL MPI_TYPE_VECTOR(4, 1, 4, MPI_INTEGER, filetype, err)
 CALL MPI_TYPE_COMMIT(filetype, err)
 disp = myid * intsize
 CALL CALL MPI_FILE_SET_VIEW(file, disp, MPI_INTEGER, filetype, “native”, &
      MPI_INFO_NULL, err)
 CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
...
```

Defining Subarray

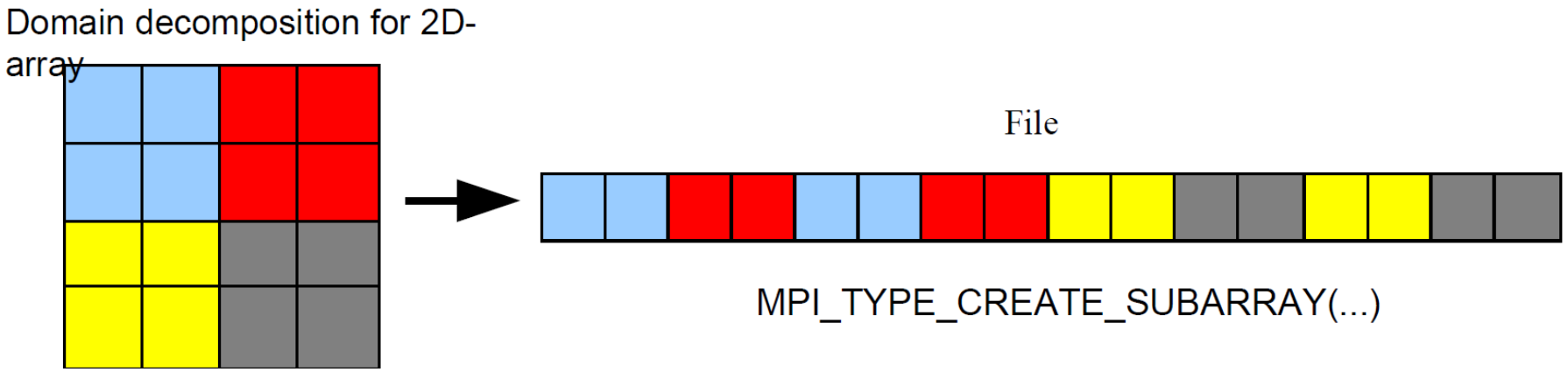
```
int MPI_Type_create_subarray(int ndims, int sizes[], int subsizes[], int starts[], int
order, MPI_Datatype basetype, MPI_Datatype *subarraytype)
```

```
MPI_Type_create_subarray(ndims, sizes, subsizes, starts, order,
basetype, subarraytype, ierr)

Integer :: dims, order, basetype, subarraytype, ierr
Integer(:) :: sizes, subsizes, starts
```

- The routine MPI_Type_create_subarray defines a new type
- The subarraytype is based on the original datatype basetype
- The creation of the subarray requires you to define the dimensions of the full array as well as the dimensions of the subarray that a particular process holds
- The starts array specifies the offset into the full array for where the subarray begins
- The order argument specifies whether the ordering is row-major (MPI_ORDER_C) or column-major (MPI_ORDER_FORTRAN) – Use the one appropriate for your programming language

Using Subarray example




```
...
 INTEGER :: sizes = (/4, 4/)
 INTEGER :: subsizes = (/2, 2/)
 INTEGER, DIMENSION(2,2) :: buf
 ...
 MPI_CART_COORDS(MPI_COMM_WORLD, myid, 2, starts, err)
 CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts,
      MPI_INTEGER, MPI_ORDER_C, filetype, err)
 CALL MPI_TYPE_COMMIT(filetype)
 CALL CALL MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype, “native”, &
      MPI_INFO_NULL, err)
 CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
...
```

MPI I/O Hints

MPI_File_open(comm, fiename, amode, info, fh)

MPI_File_set_view(fh, disp, etype, filetype, datarep, info)

- MPI_File_set_view and MPI_File_open APIs allow the user to provide information on the features of the File System employed
- Hints may enable the implementation to optimize performance
- MPI 2 standard defines several hints via MPI_Info object
  - MPI_INFO_NULL : no info
  - Functions MPI_Info_create, MPI_Info_set allow one to create and set hint
- Some implementations allow setting of hints via environment variables
  - e.g. MPICH_MPIIO_HINTS
  - Example: for file “test.dat”, in collective I/O aggregate data to 32 nodes

    export MPICH_MPIIO_HINTS=”test.dat:cb_nodes=32”
- Effect of hints on performance is implementation and application dependent

Common MPI IO Hints

	Hint Name - Usage
Controlling Parallel File System	**striping_factor** – The number of Lustre stripes to assign to a file **striping_unit** – The size (in bytes) of the Lustre stripes to use for a file **start_iodevice** - which I/O server to start with
Controlling Aggregation	**cb_config_list** - A hint as to how to select the nodes for aggregation **cb_nodes** - The number of nodes to use for aggregating data
Tuning ROMIO (most common MPI-IO implementation) optimizations	**romio_cb_read, romio_cb_write** - Flags to say whether to disable, enable or use heuristics for deciding whether to aggregate for collective reads and writes **romio_ds_read, romio_ds_write** – for data sieving on/off

Passing Hints to MPI IO

```
MPI_Info info;

MPI_Info_create(&info);

/* no. of IO devices to be used for file striping */
MPI_Info_set(info, "striping_factor", "4");

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

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
                        MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);
MPI_Info_free(&info);
```

Independent or Collective I/O

Independent I/O operations

- Specify only what a single process will do
- Do not pass on relationships between I/O on other processes

Collective I/O is coordinated access to storage by a group of processes

- Collective I/O functions are called by all processes participating in I/O
- Allows I/O layers to know more about access as a whole, more opportunities for optimization in lower software layers, better performance

Many applications have alternating phases of computation and I/O

- During I/O phases, all processes read/write data
- We can say they are collectively accessing storage

Collective operations

- I/O can be performed *collectively* by all processes in a communicator
  - MPI_File_read**_all**
  - MPI_File_write**_all**
  - MPI_File_read_at**_all**
  - MPI_File_write_at**_all**
- Same parameters as in independent I/O functions
  - MPI_File_read
  - MPI_File_write
  - MPI_File_read_at
  - MPI_File_write_at

Blocking vs Non-blocking IO

- Independent, nonblocking IO

- 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.

Collective, Non-blocking IO

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

Consistency Operations

MPI_FILE_SET_ATOMICITY (fh, flag)
INOUT fh: file handle (handle)
IN buf: true/false flag (logical)

- set consistency semantics for data access operations on file
- collective operation, all processes must pass identical values

MPI_FILE_SYNC (fh)
INOUT fh: file handle (handle)

- cause all previous write to file be transferred to storage device
- collective operation
- nonblocking requests and split collective operations must have been completed before calling MPI_FILE_SYNC

References

https://www.nhr.kit.edu/userdocs/horeka/parallel_IO/
https://hpc-forge.cineca.it/files/CoursesDev/public/2017/Parallel_IO_and_management_of_large_scientific_data/Roma/MPI-IO_2017.pdf
https://janth.home.xs4all.nl/MPIcourse/PDF/08_MPI_IO.pdf
https://events.prace-ri.eu/event/176/contributions/59/attachments/170/326/Advanced_MPI_II.pdf
https://www.cscs.ch/fileadmin/user_upload/contents_publications/tutorials/fast_parallel_IO/MPI-IO_NS.pdf