22nd Summer School on PARALLEL COMPUTING

Parallel IO: basics and MPI2-IO

Giusy Muscianisi – g.muscianisi@cineca.it
Luca Ferraro – l.ferraro@cineca.it

SuperComputing Applications and Innovation Department

CINECA

Introduction

- IO is a crucial issue in the modern high performance codes:
  - 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 trascurable
  - data portability (e.g. endianism)

- Solution to avoid that IO became a bottleneck:
  - HW: parallel file-system available on all the HPC platform
  - SW: high level libraries able to manage parallel accesses to the file in efficient way (e.g. MPI2-IO, HDF5, NetCDF, …)

CINECA IO System Configuration

Both IBM BlueGene/Q (FERMI) and PLX Linux Cluster IO are based on the

General Parallel File System (GPFS)

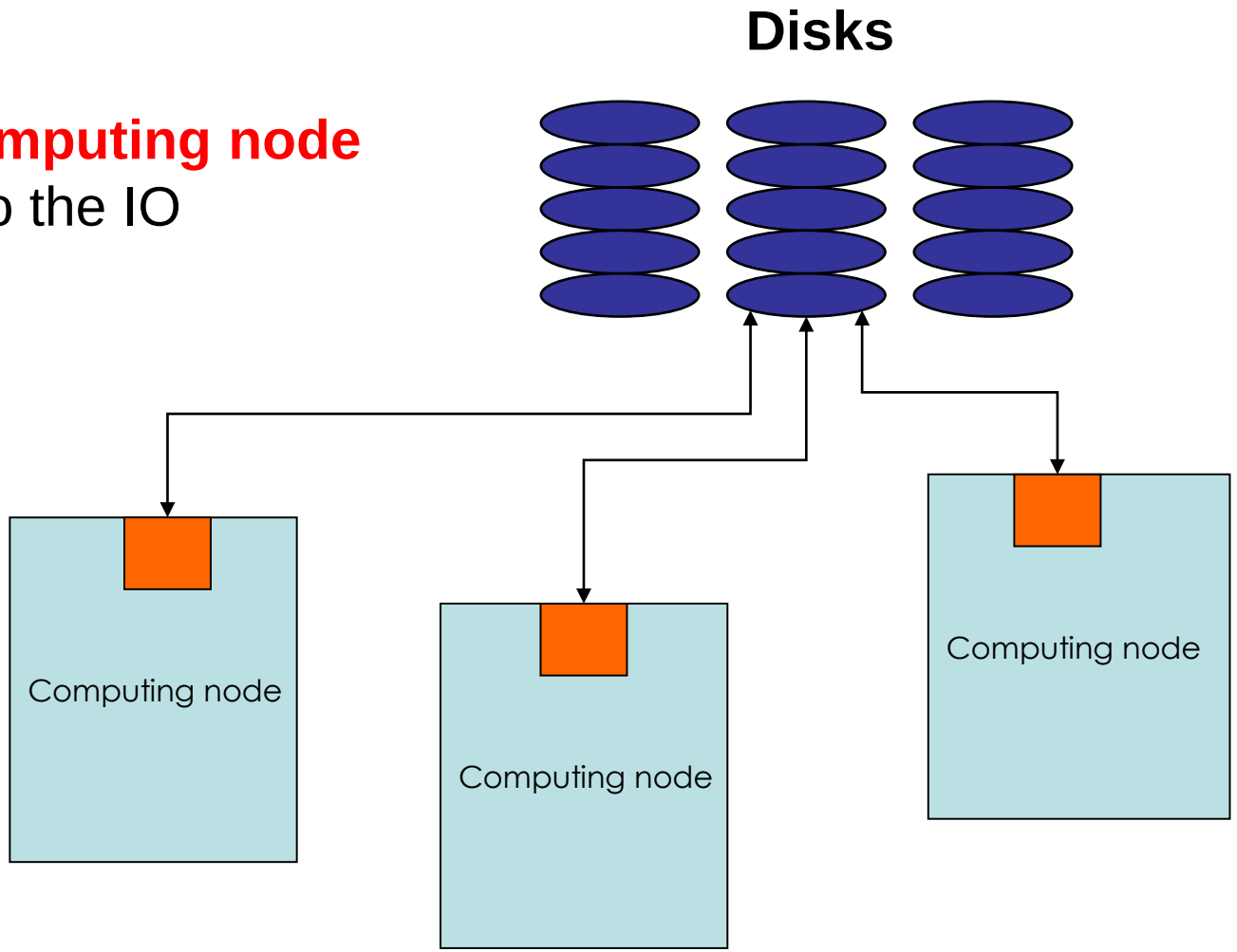
technology (IBM propretary)

GPFS is:

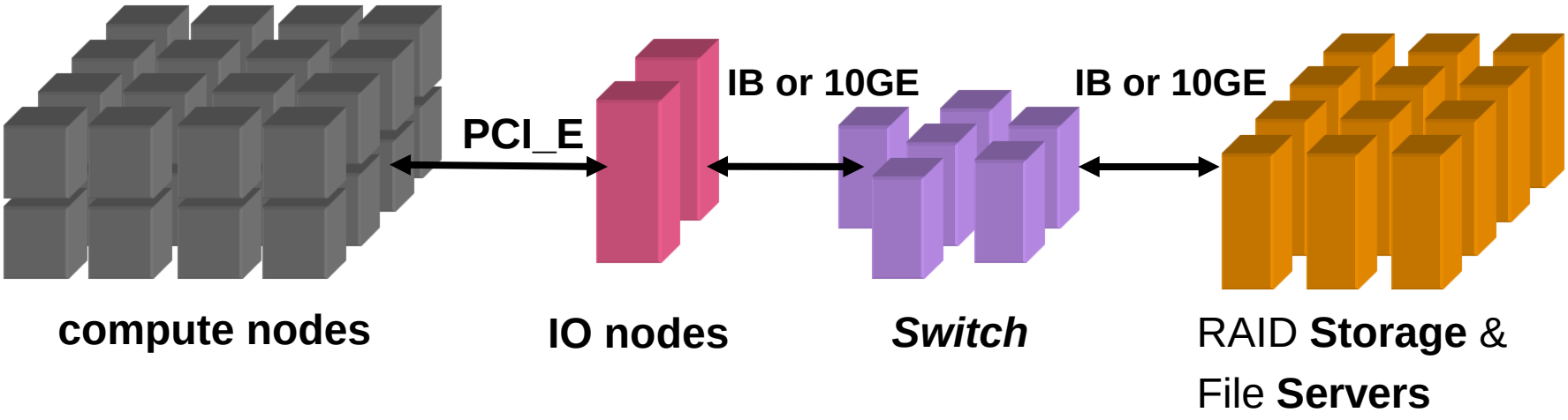
- **High performance**
- **Scalable**
- **Reliable**
- **Ported on many platforms (in particular AIX and Linux)**

GPFS Architecture

Part of the **computing node** is dedicated to the IO management



Blue Gene/Q IO architecture

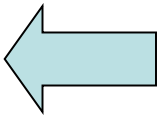


IO nodes: each one manages groups of compute nodes

Data Portability: Data Representation

There are two common different representations:

Little Endian



PC (Windows/Linux)

Byte3 Byte2 Byte1 Byte0

will be arranged in memory as follows:

Base Address+0 Byte0
Base Address+1 Byte1
Base Address+2 Byte2
Base Address+3 Byte3

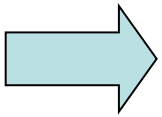
Big Endian

Byte3 Byte2 Byte1 Byte0

will be arranged in memory as follows:

Base Address+0 Byte3
Base Address+1 Byte2
Base Address+2 Byte1
Base Address+3 Byte0

Unix (IBM, SGI, SUN…)



Parallel IO

Goals:

- Improve the performance
- Ensure data consistency
- Avoid communication
- Usability

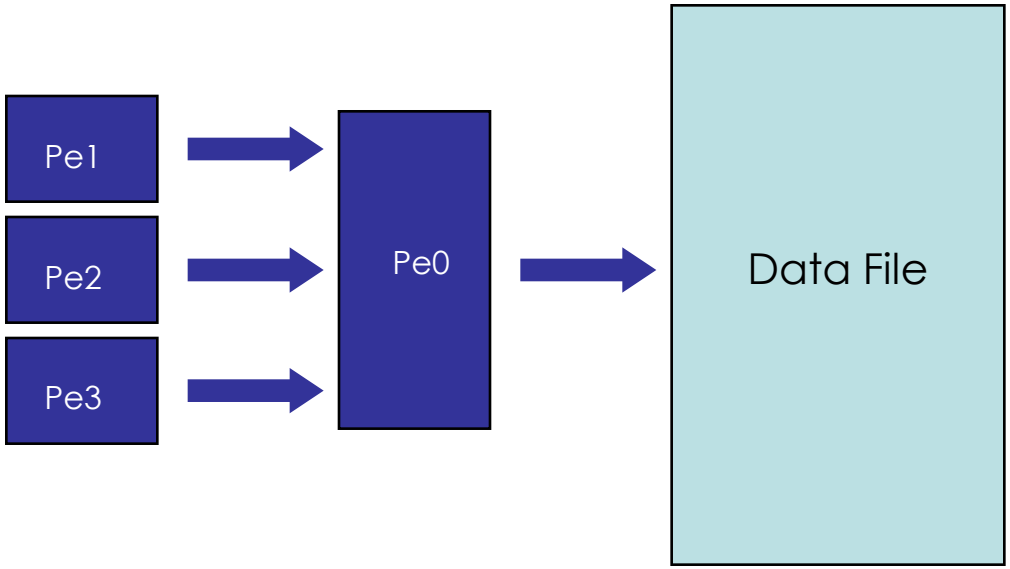
Possible solutions:

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

Parallel IO

Solution 1: Master-Slave

Only 1 processor performs IO



Goals:

Improve the performance: **NO**

Ensure data consistency: **YES**

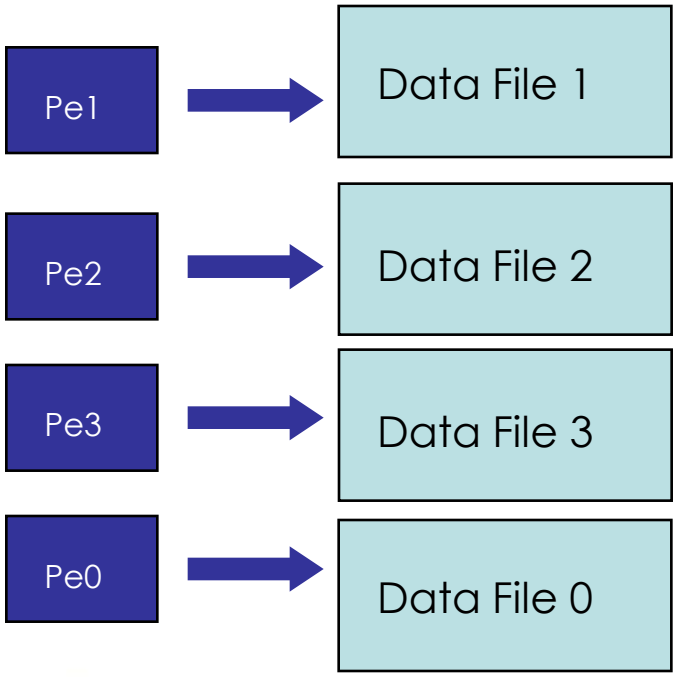
Avoid communication: **NO**

Usability: **YES**

Parallel IO

Solution 2: Distributed IO

All the processors read/writes their own files



Goals:

Improve the performance: **YES**

(but be careful)

Ensure data consistency: **YES**

Avoid communication: **YES**

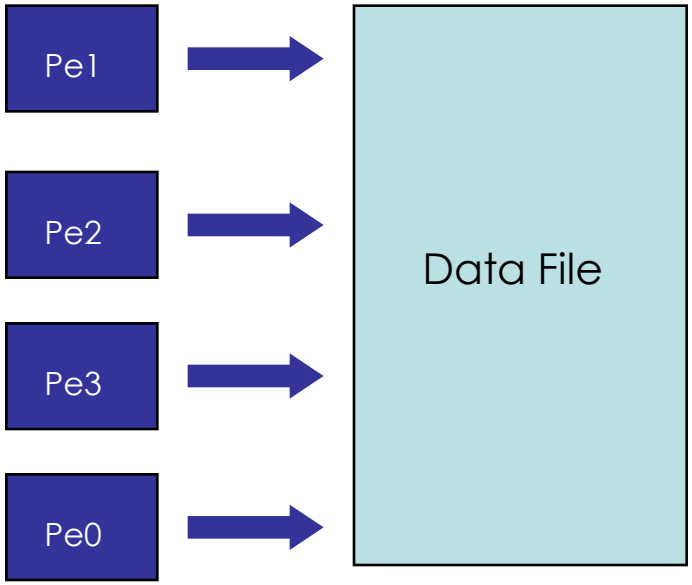
Usability: **NO**

Warning: avoid to parametrize with processors!!!

Parallel IO

Solution 3: Distributed IO on single file

All the processors read/writes on a single ACCESS = DIRECT file



Goals:

Improve the performance: **YES** for read, **NO** for write

Ensure data consistency: **NO**

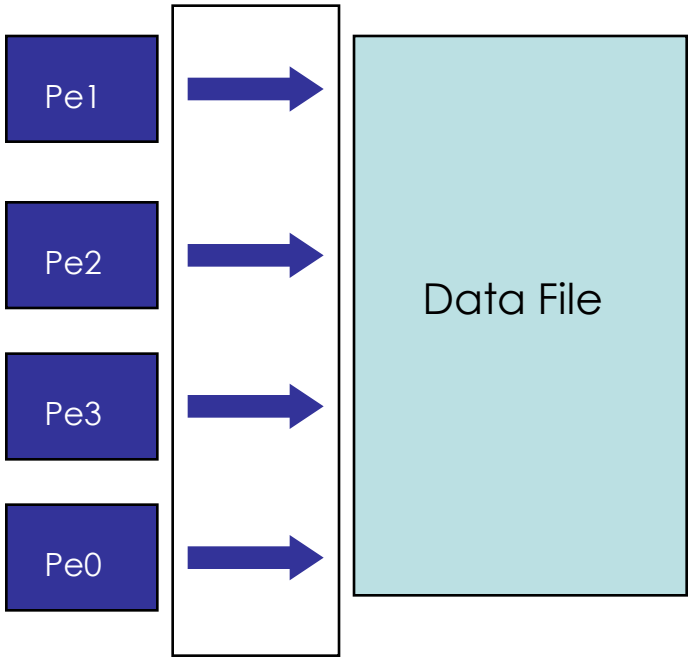
Avoid communication: **YES**

Usability: **YES (portable !!!)**

Parallel IO

Solution 4: MPI2 IO

MPI functions performs the IO. Asyncronous IO is supported.



Goals:

Improve the performance: **YES (strongly!!!)**

Ensure data consistency: **NO**

Avoid communication: **YES**

Usability: **YES**

Parallel IO

I/O Patterns in Parallel applications:

- **Different from those in sequential programs, which usually access data in contiguous chunks**
- **In many parallel programs, each program may need to access several noncontiguous pieces of data from a file**
- **In addition, groups of processes may need to access the file simulataneously, and theaccesses of different processes may be interleaved in the file**

Unix APIs for I/O 1/3

Most parallel file-system have UNIX like API

- **open, open a file -> may be expensive**
- **lseek, move the pointer to a particular offset of the file -> performance depend on the implementation**
- **read/write, read/write n bytes starting from the current position of the pointer**
  - **perform well if I/O size is large 1Mb or more, very poorly if size is small < 8Kb**
- **close, close the file -> not expensive as open, but not so cheap**

Unix APIs for I/O 2/3

Other I/O functions:

- **readv, writev**
  - **read into/write from multiple buffers in memory**
  - **in the file, however, data is assumed to be contiguously located**
  - **of limited use because the users need to specify noncontiguity in the file more often than in memory**
- **aio_read,aio_write, POSIX, asynchronous, performance is not good enough, usually**
- **list I/O in POSIX: users can specify a list of operations at a time**
  - **it doesn't treat the list as a single element**
  - **non notion of collective I/O**
  - **each operation is internally performed as a separate aio_read, aio_write**

Unix APIs for I/O 3/3

Problems with UNIX API for parallel I/O

- **Non contiguous access cannot be expressed as a single call: each contiguous piece must be accessed separately resulting in too many system calls and poor performance**
- **No notion of collective I/O**

MPI-IO can be considered as UNIX-IO plus a lot of stuff more

MPI2-IO

MPI-2.x features for Parallel IO

- MPI-IO: introduced in MPI-2.x standard(1997)
  - Non contiguous access in both memory and file
  - reading/writing a file is like send/receive a message from a MPI buffer
  - optimized access to 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 100002 (dp)
    - MPI-IO: **48**sec vs Traditional-IO: **3570**sec (dimension of written file is 24Gb)

Starting with MPI-IO

- MPI-IO provides basic IO operations:
  - open, seek, read, write, close (ecc.)
- open/close are collective operations on the same file
  - many modalities to access the file (combinabili: |,+)
- read/write are similar to send/recv of data to/from a buffer
  - Each MPI process has own local pointer to the file (individual file pointer) by 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

Open/close a file 1/3

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 operations 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 vie 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.

Open/close a file 2/3

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

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

Open/close a file 3/3

MPI_FILE_CLOSE(fh)

INOUT fh: file handle (handle)

- Collective operation
- This function is called when the file access is finished, to free the file handle.

Data Access 1/3

MPI-2 provides a large number of routines to read and write data from a file.
There are three properties which differentiate the different **data access** routines.

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**
  - This file 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!

Data Access 2/3

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 completition. 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

Data Access 3/3

Positioning	Synchronisation	Coordination	
		Noncollective	*Collective*
Explicit offsets	*Blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	Non-blocking & split collective	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
Individual file pointers	*Blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	Non-blocking & split collective	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
Shared file pointer	*Blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	Non-blocking & split collective	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

Individual file pointers - Write

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 elemnt (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.
- **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

Individual file pointers - Read

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.
- **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

Seeking to a file position

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 have the following values:
  - 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 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

Using individual file pointers

```
#include "mpi.h"
#define FILESIZE(1024*1024)
int main(int argc, char **argv){
     int *buf, rank, nprocs, nints, bufsize;
     MPI_File fh; MPI_Status status;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
     bufsize = FILESIZE/nprocs;
     nints =bufsize/sizeof(int);
     buf = (int*) malloc(nints);

     MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,
        MPI_INFO_NULL,&fh);
     MPI_File_seek(fh, rank*bufsize,MPI_SEEK_SET);
     MPI_File_read(fh, buf, nints, MPI_INT, &status);
     MPI_File_close(&fh);
     free(buf);
     MPI_Finalize();
     return 0;
}
```

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=100
     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

Explicit offset - Write

MPI_FILE_WRITE_AT (fh, offset, buf, count, datatype, status)

- IN fh: file handle (handle)
- IN offset: file offset in byte (integer)
- IN buf: source buffer
- IN count: number of written 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
- The sequence of basic datatypes of **DATATYPE** (= signature of DATATYPE) must match contiguous copies of the etype of the current view

Explicit offset – Read

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
- The sequence of basic datatypes of **DATATYPE** (= signature of DATATYPE) must match contiguous copies of the etype of the current view

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
```

Shared file pointer - Write, Read

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 quering 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

Advanced features of MPI-IO

- Basic MPI-IO features are not useful when
  - Data distibution 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 datatype
  - collective access: grouping multiple near accesses in one or more single accesses (decreasing the latency time)

File view

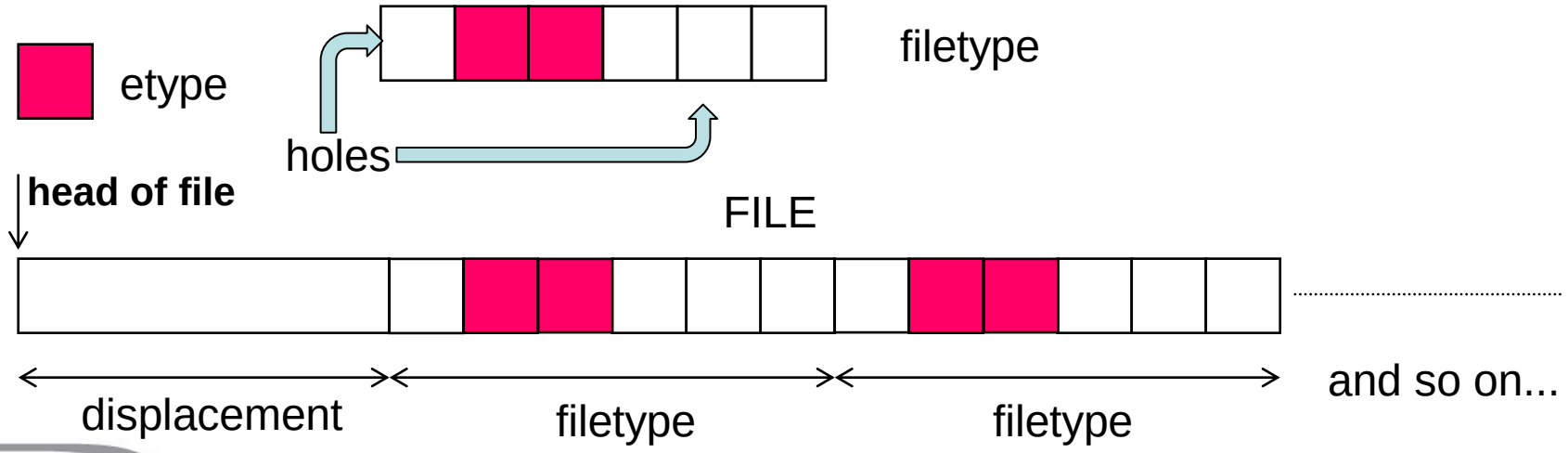
- A file view defines which portion of a file is “visible” to a process
- File view defines also the type of the data in the file (byte, integer, float, …)
- By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file
- A default view for each participating 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

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

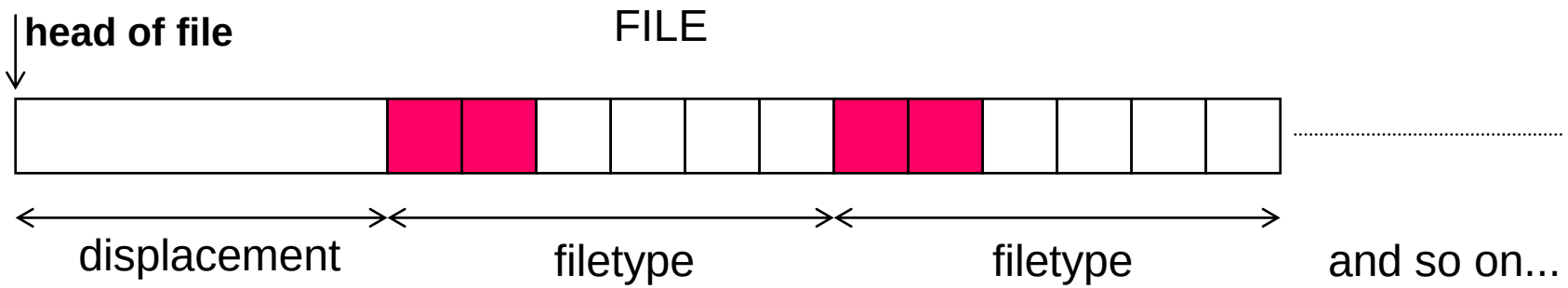
The pattern described by a filetype is repeated, beginning at the displacement, to define the view, as it happens when creating MPI_CONTIGUOUS or when sending more than one MPI datatype element: HOLES are important!



File View

etype = MPI_INT

filetype = two MPI_INTs followed by
a gap of four MPI_INTs



- Define a file-view in order to have
  - fundamental access unit (etype) is MPI_INT
  - access pattern (fileytpe) is given by:
    - first 2 fundamental units
    - skips the next 4 fundamental units
  - skips the first part (5 integers) of the file (displacement)

File View

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 in File View

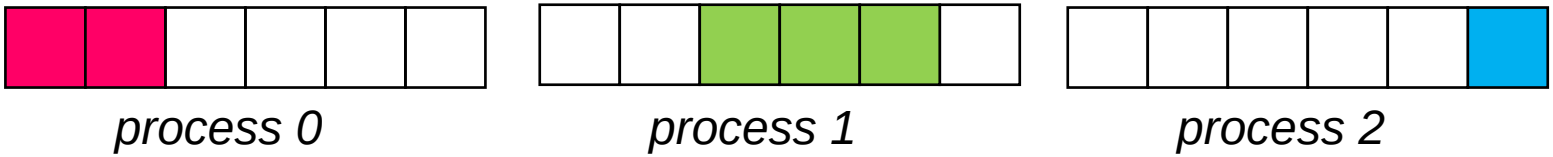
- Data representation: define the layout and data access modes (byte order, type sizes, ecc)
  - **native:** (default) use the memory layout with no conversion
  - no precision loss or conversion effort
    - not portable
  - **internal:** layout implementation-dependent
    - portable for the same MPI 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 usiing 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**

Passing hints to Filesystem

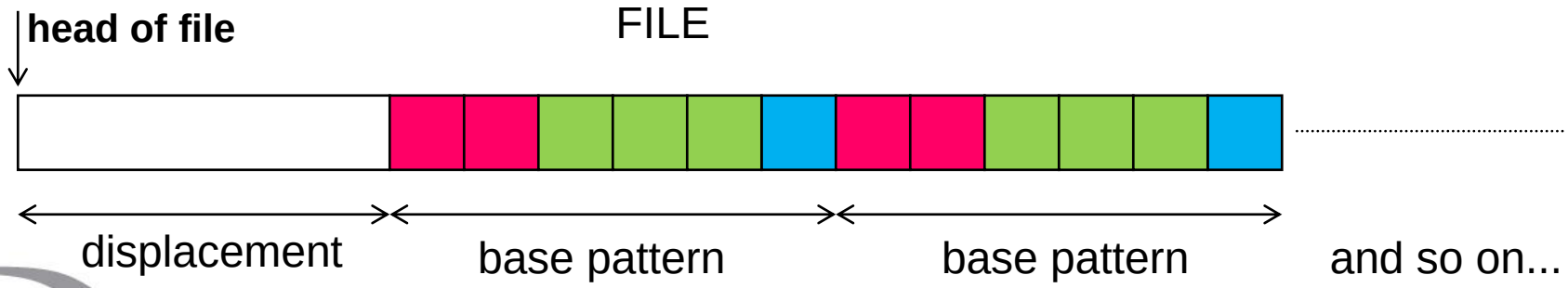
- MPI allows the user to provide information on the features of the File System employed
  - optionals
  - may improve performances
  - depend on the MPI implementation
  - default: use MPI_INFO_NULL if you are not very expert
- Infos are objects created by MPI_Info_create
  - elements key-value
  - use MPI_Info_set to add elements
- ... refer to standard for more information and to manuals
  - e.g., consider ROMIO implemenation 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

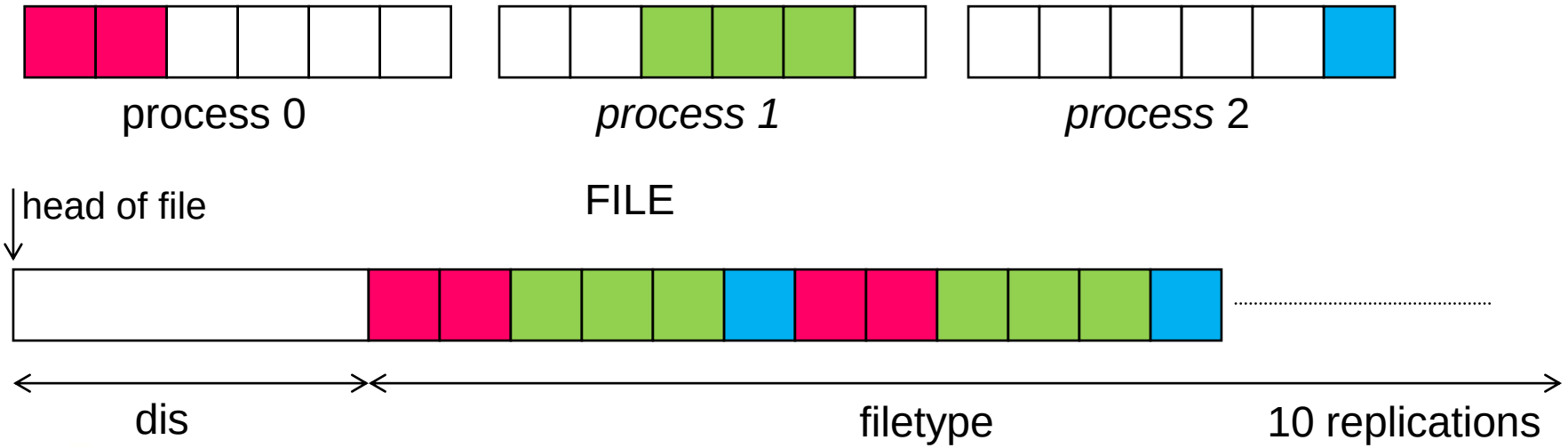


to be replicated a certain amount of (unknown?) times

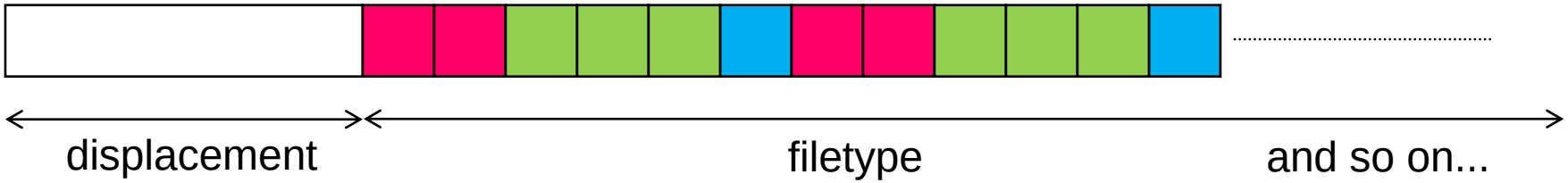


I strategy: data-type replication

- 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 and *dis*+5



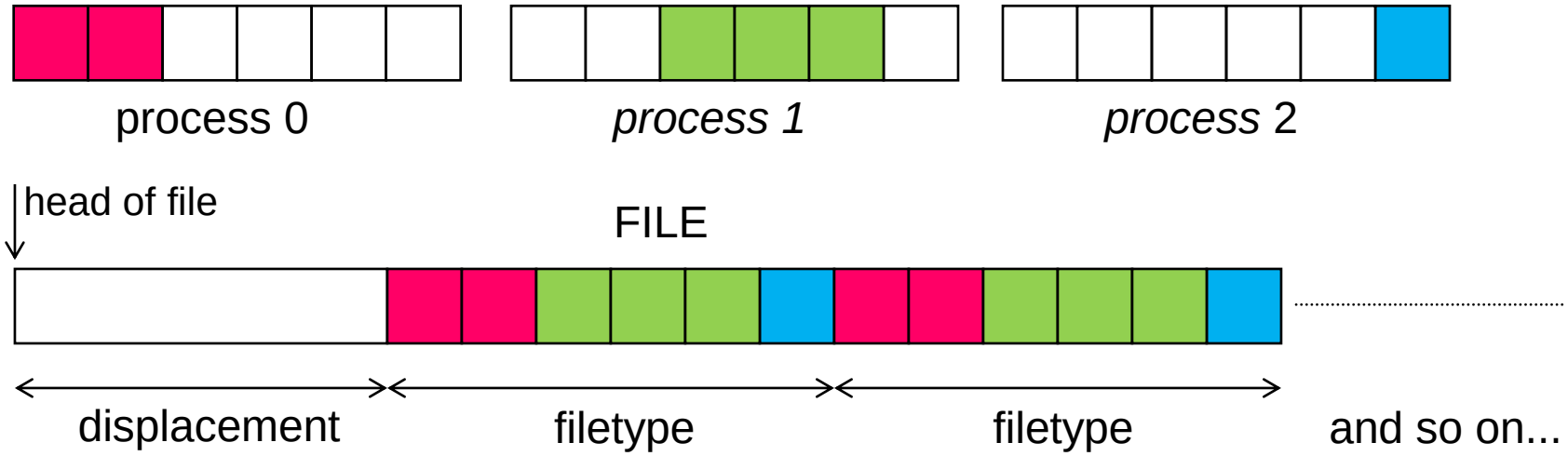
Use data-type replication




```
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);
```

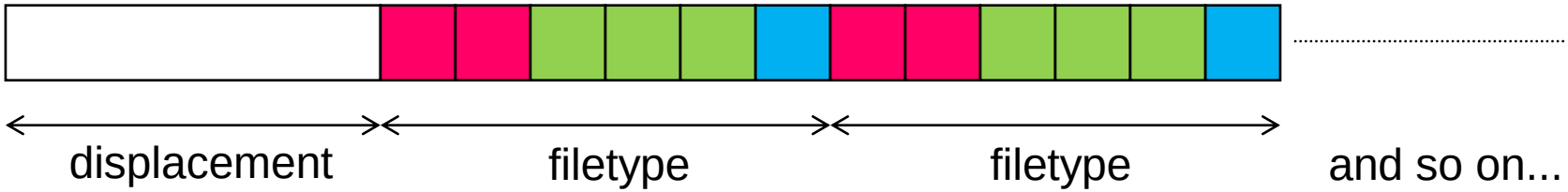
II strategy: file view replication

- 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 left and on the right)
  - set the file view with displacements to balance the left holes



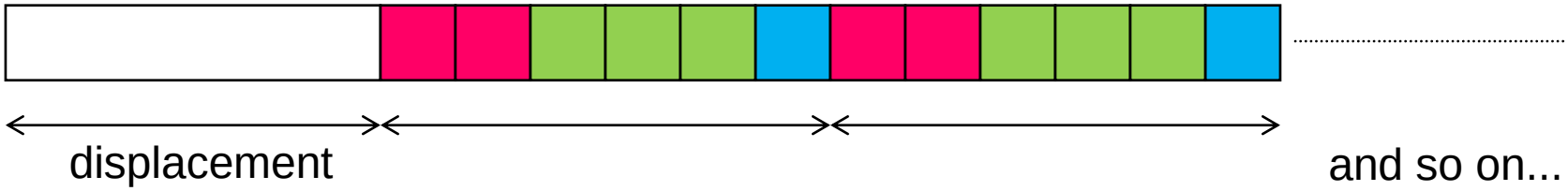
- When writing more than a filetype, a replication occurs; as it happens when sending more than one data, setting the holes is crucial!

Use file view replication



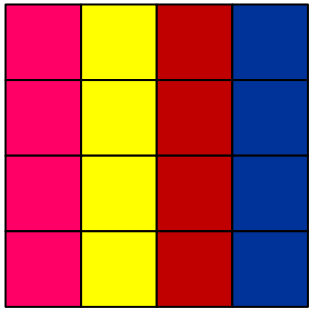

```
MPI_Datatype cont_t;
MPI_Type_contiguous(count_proc[myrank], MPI_INT, &cont_t);
MPI_Type_commit(&cont_t);

MPI_Aint lb_int, extent_int;
MPI_Type_get_extent(MPI_INT,&lb_int,&extent_int);
MPI_Aint lb_proc[]={0,-2*extent_int,-5*extent_int};
MPI_Datatype filetype;
MPI_Type_create_resized(cont_t, lb_proc[myrank], (MPI_Aint)
                                  6*extent_int, &filetype);
MPI_Type_commit(&filetype);

offset = -lb_proc[myrank];
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?

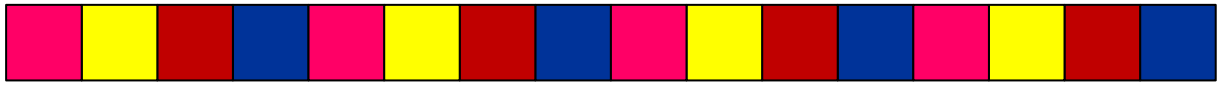


- **Which is the best replication strategy?**
  - **If possible, data-type replication is probably better (just one operation)**
  - **Surely, easier to be implemented**
  - **But exploiting file view replication is mandatory when then number of read/writes is not known *a priori***

Non-contiguous access: with known replication pattern



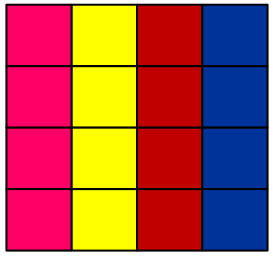
2D-array distributed column-wise



File written per row

- Each process has to access small pieces of data scattered throughout a file
- Very expensive if implemented with separate reads/writes
- Use file type to implement the non-contiguous access
- Again, employ data-type replication mechanism

Non-contiguous access: with known replication pattern

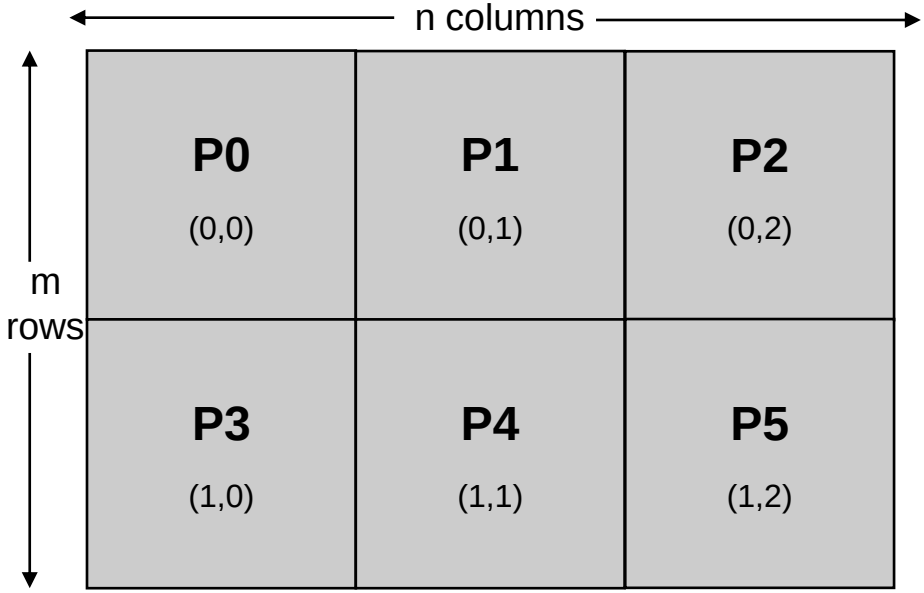


File written per row

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 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)
```

Non-contiguous access: distributed matrix



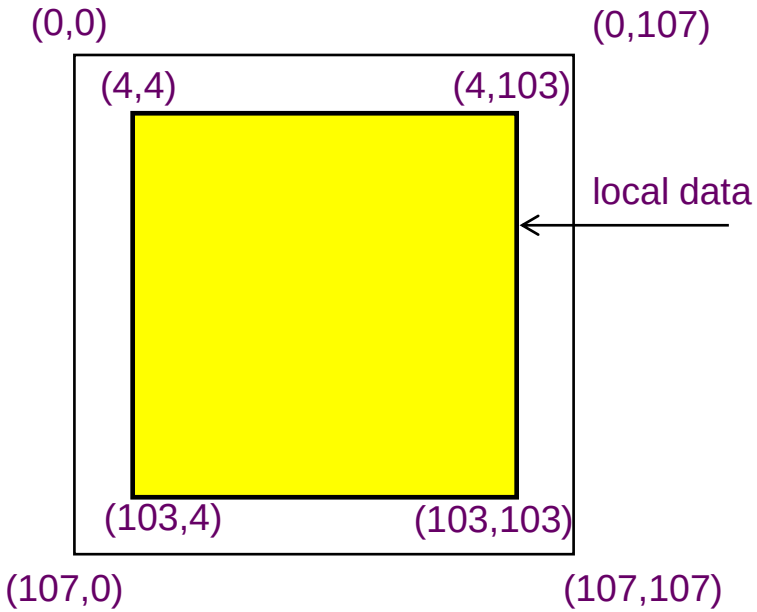
- 2D array, size (m,n) distributed among six processes
- cartesian layout 2x3

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

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 */
dims[0] = 2; dims[1] = 3;
periods[0] = periods[1] = 1;

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
/* 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);
```

Ghost cells, typical case



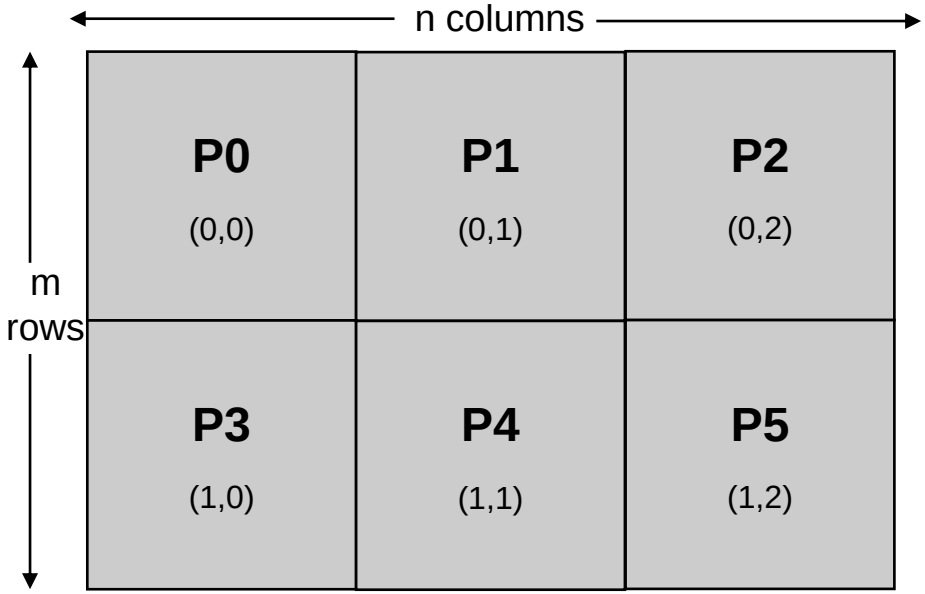
- local array with sizes (100,100) allocated with sizes (108,108) to store
- ghost areas along edges
- ghost area are filled with neighbouring processes data
- local data are stored from position (4,4)
- non-contiguous memory access is needed

- Local data may be considered as a subarray
- Using MPI_Type_create_subarray we can filter the local data creating a subarray
- This type will be used as access basic type to communicate or to perform I/O

Ghost cells, typical case

```
/* 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_all(fh, local_array, 1, memtype, &status);
```

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)	1	2	18	33	48

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

Collective, blocking IO

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_oredered
- MPI_File_write_ordered

All processes in communicator that opened file must call function

Performance potentially better than for individual functions

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

Collective, blocking IO

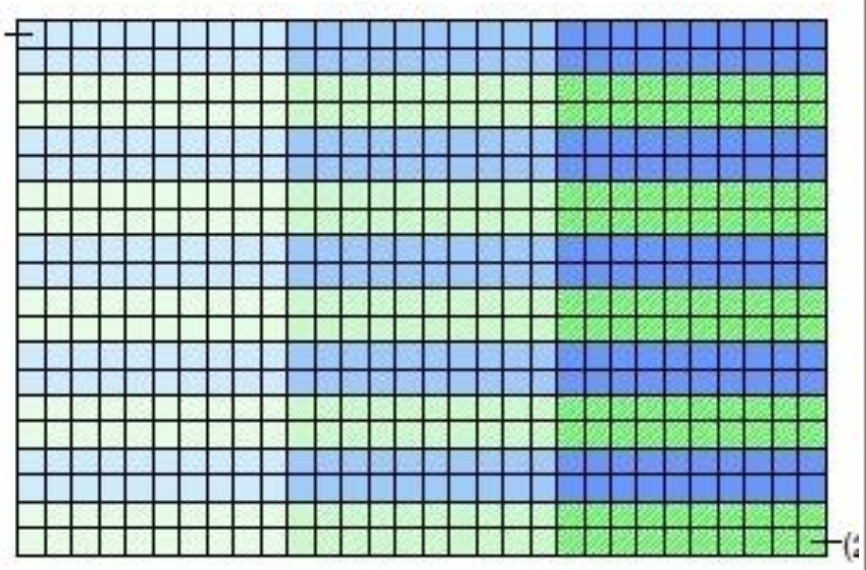
```
int MPI_File_write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
    MPI_Status *status)

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

Darray and collective IO 1/2

```
/* int MPI_Type_create_darray (int size, int rank, int ndims, int array_of_gsizes[],
      int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

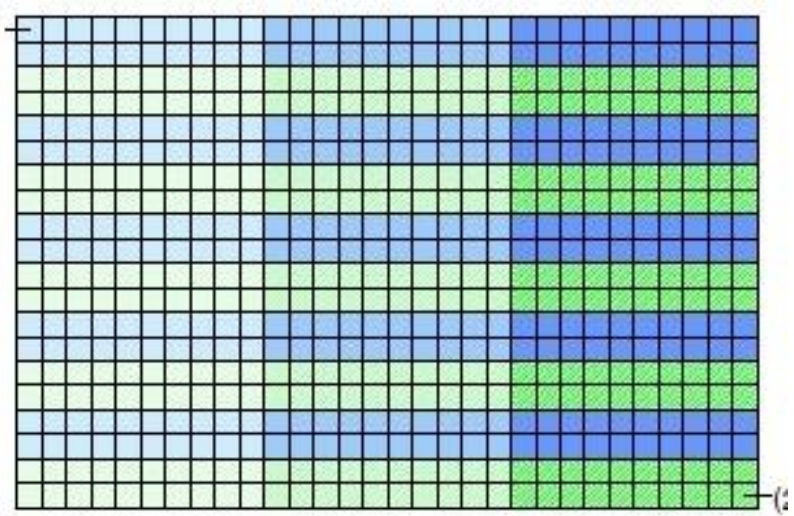
```
int gsizes[2], distribs[2], dargs[2], psizes[2];

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

distribs[0] = MPI_DISTRIBUTE_BLOCK;
distribs[1] = MPI_DISTRIBUTE_BLOCK;

dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;
```



```
psizes[0] = 2; /* no. of processes in vertical dimension of process grid */
psizes[1] = 3; /* no. of processes in horizontal dimension of process grid */
```

Darray and collective IO 2/2

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs,
          psizes, MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
          MPI_MODE_CREATE | MPI_MODE_WRONLY,
          MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
             MPI_INFO_NULL);

local_array_size = num_local_rows * num_local_cols;
MPI_File_write_all(fh, local_array, local_array_size,
        MPI_FLOAT, &status);

MPI_File_close(&fh);
```



Independent, nonblocking IO

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 syncronization.

Can be used to overlap IO with computation

Collective, nonblocking 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

Use cases

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)

Use cases

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.

Use cases

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

Best Practices

- When designing your code, include I/O!
  - 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

QUESTIONS ???