21st Summer
School of
PARALLEL
COMPUTING

July 2 - 13, 2012 (Italian)
September 3 - 14, 2012 (English)

Parallel I/O: Basics and MPI2

Giusy Muscianisi – g.muscianisi@cineca.it
SuperComputing Applications and Innovation Department

CINECA

Introduction

Reading and Writing data is a problem usually underestimated.

However it can become crucial for:

1. Performance
2. Porting data on different platforms

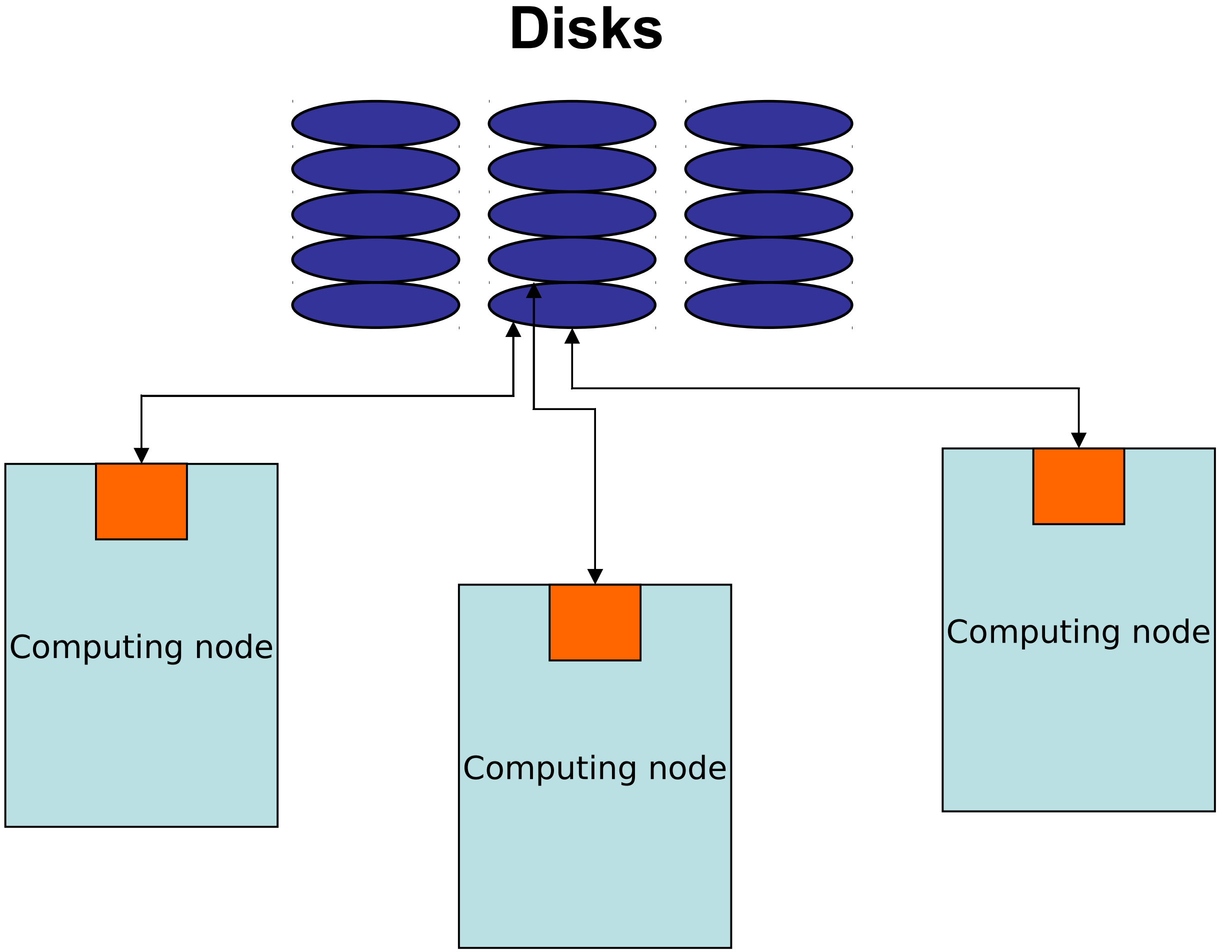
CINECA I/O system configuration

Both IBM BlueGene/Q (FERMI) and PLX Linux Cluster I/O are based on the General Parallel File System (GPFS) technology (IBM propretary)

GPFS is:

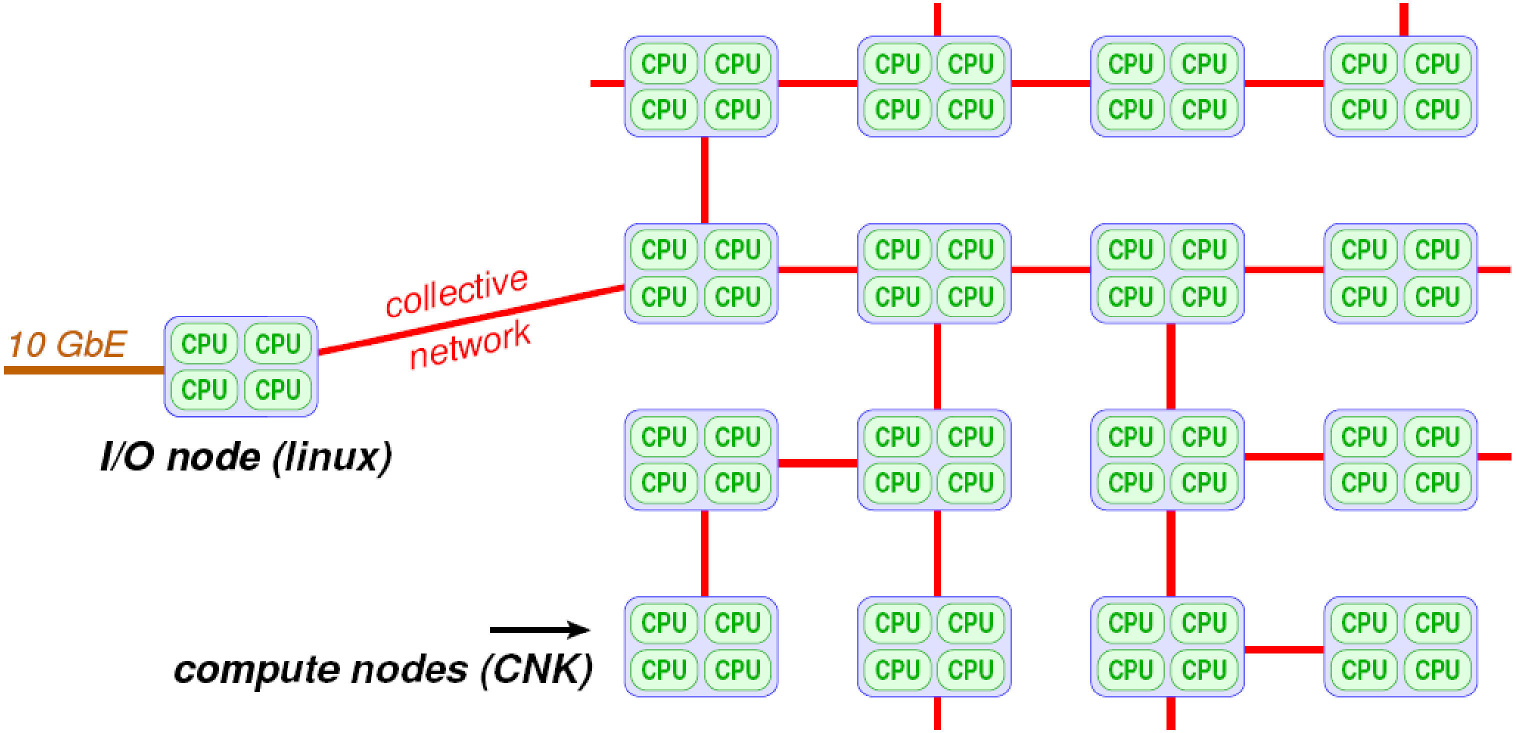
- High performance
- Scalable
- Reliable
- Ported on many platforms (in particular AIX and Linux)

GPFS architecture



Part of the computing node is dedicated to the I/O management

Blue Gene P psets



I/O nodes: each one manages groups of compute nodes

1. Performance

Optimization is platform dependent.

In general: **write large amount of data in single shots**

For example: avoid looped read/write

```
 do i=1,N
    write (10) A(i)
enddo
```

It's VERY slow

2. Data portability

This is a subtle problem, which becomes crucial only after all… when you try to use data on different platforms.

For example: unformatted data written by an IBM system cannot be read by a Linux/MS Windows PC

2. Data portability: data representation

There are two 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

Unix (IBM, SGI, SUN…)

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

Parallel I/O

Goals:

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

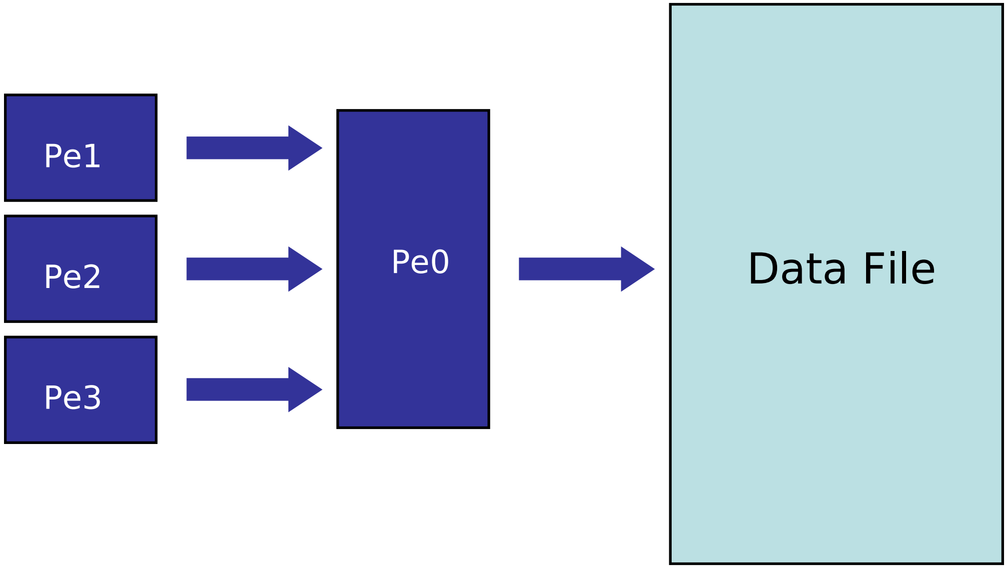
Possible solutions:

1. Master-Slave
2. Distributed
3. Coordinated
4. MPI I/O
5. High level libraries

Parallel I/O

Solution 1: Master-Slave

Only 1 processor performs I/O



Goals:

Improve the performance: **NO**

Ensure data consistency: **YES**

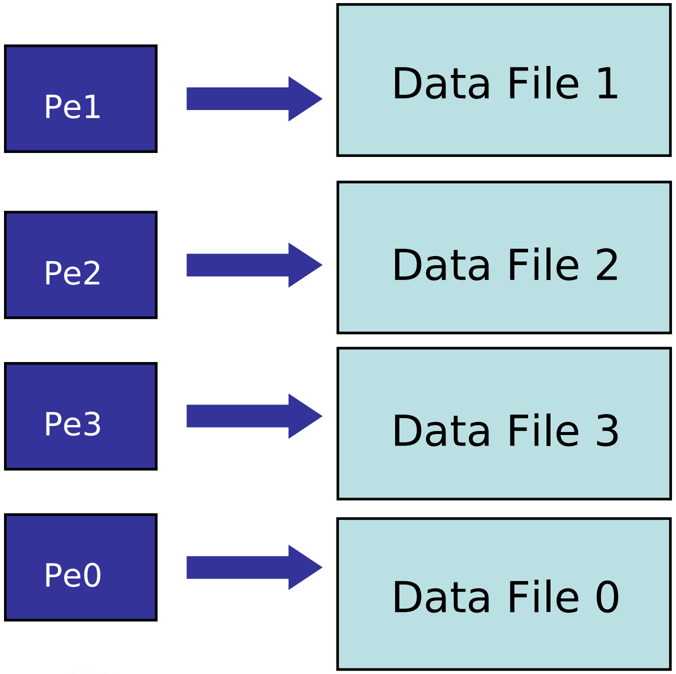
Avoid communication: **NO**

Usability: **YES**

Parallel I/O

Solution 2: Distributed I/O

All the processors read/writes their own files



Goals:

Improve the performance: **YES**
(but be careful)

Ensure data consistency: **YES**

Avoid communication: **YES**

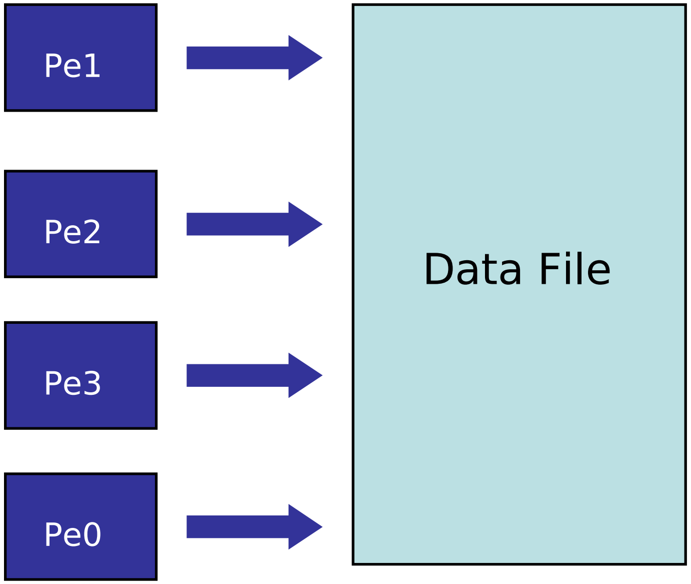
Usability: **NO**

Warning: avoid to parametrize with processors!!!

Parallel I/O

Solution 3: Distributed I/O 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**

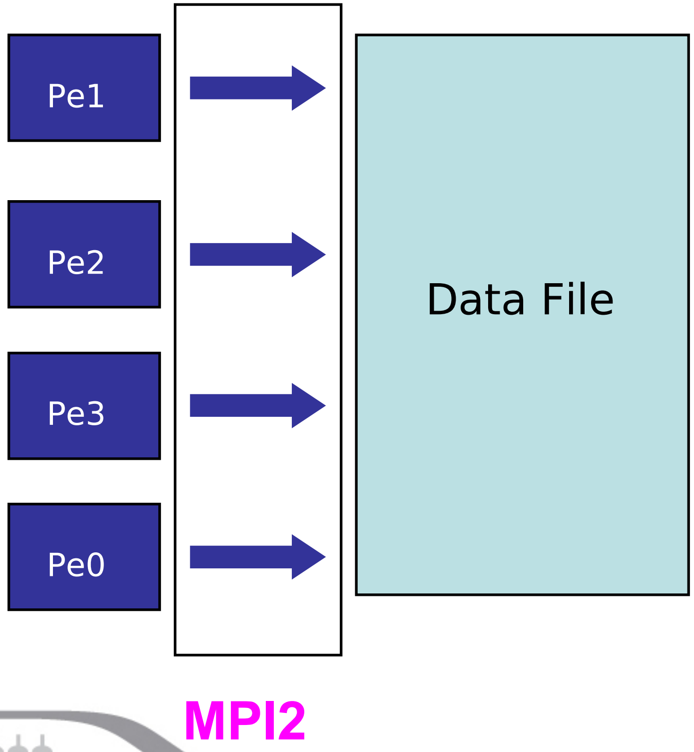
Avoid communication: **YES**

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

Parallel I/O

Solution 4: MPI2 I/O

MPI functions performs the I/O. Asyncronous I/O is supported.



Goals:

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

Ensure data consistency: **NO**

Avoid communication: **YES**

Usability: **YES**

Parallel I/O

Solution 5: High level libraries

HDF5

NetCDF

…

MPI 2 – I/O

MPI 2 - I/O

- 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
- Potentially good performance
- Easy to use
  - It blends into syntax and semantic scheme of point-to-point and collective communication of MPI.
  - Writing to a file is like sending data to another process.
- Used as the backbone of many parallel I/O libraries such as parallel NetCDF and parallel HDF5

Some definitions 1/3

File

An MPI file is an ordered collection of typed data items.

MPI supports random or sequential access to any integral set of these items.

MPI standard **does not add anything to the file**. It is not a file format.

Files written with MPI I/O can be read from **any** non **MPI application.**

Displacement

A file displacement is an absolute **byte** position relative to the beginning of a file.

The displacement defines the **location where a view begins.**

Offset

An offset is a position in the file relative to the current view.

It os expressed as a count of elementary type.

Holes in the view's filetype are skipped when calculating this position.

Some definitions 2/3

Etype (elemntary type)

An etype is unit of data access and positioning.

It must be the same on all processes with the same file handle.

It can be any MPI predefined or derived datatype.

Filetype

The filetype describes the access pattern of the processes on the file.

It defines what parts of the file are accessible by a specific process.

The processes may have different file types to access different parts of a file.

File View

Part of the file which is visible to a process. Each process has its own view of the file, defined by a displacement, an elementary type, and a filetype.

File view enables efficient noncontiguous access to file.

Some definitions 3/3

File pointer

Position in the file where to read or write.

– **Individual file pointers:** local to each process that opened the file.

– **Shared file pointer:** it is shared by the group of processes that opened the file.

File handle

A file handle is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE.

All operations on an open file reference the file through the file handle.

File size and end of file

The size of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file.

For any given view, the end of the file is the offset of the first etype accessible in the current view starting after the last byte in the file.

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

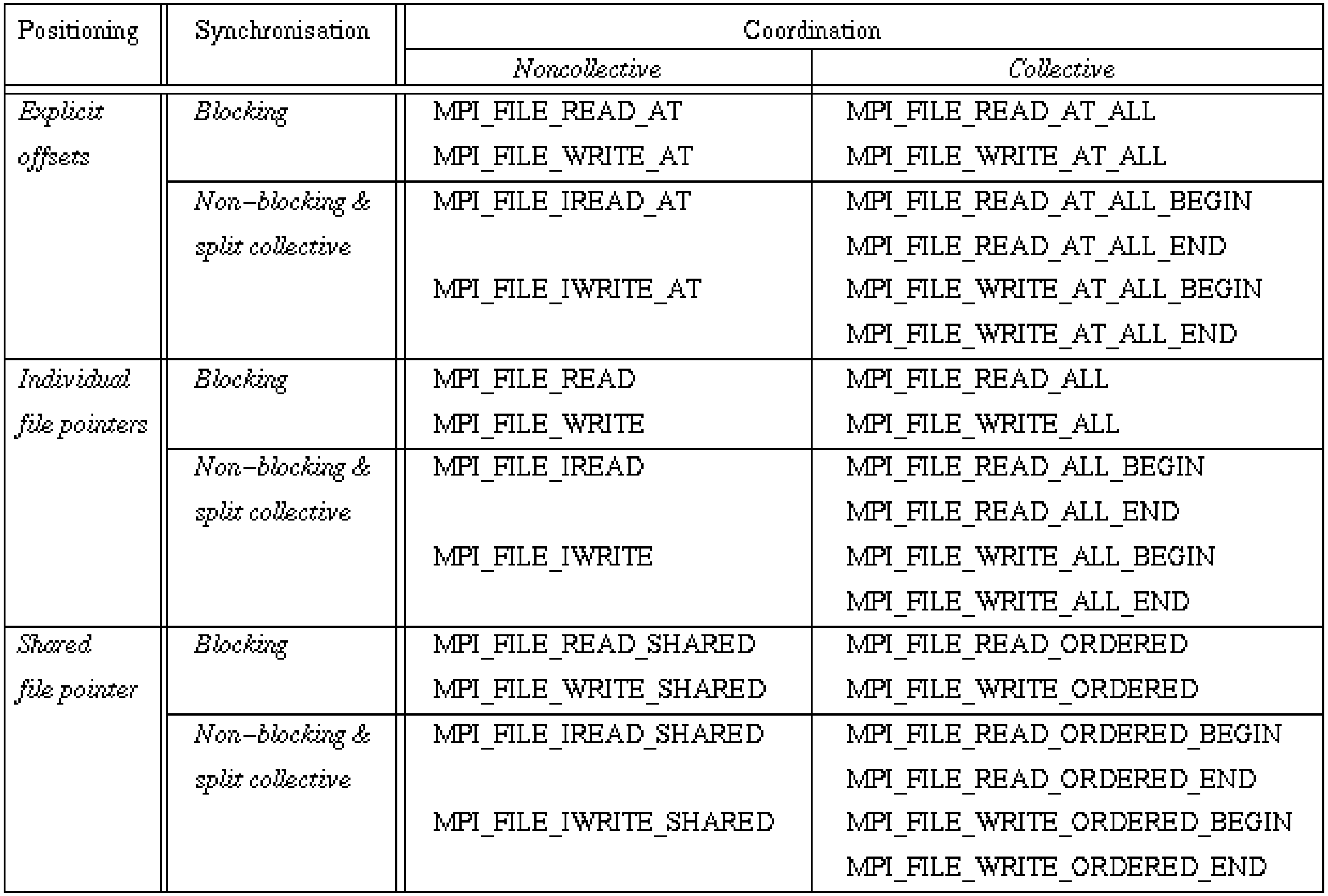
Synchronisation. MPI-2 supports both **blocking** and **non-blocking I/O** routines.

- A blocking I/O call will not return until the I/O request is completed.
- A nonblocking I/O call initiates an I/O operation, but not wait for its completitition. 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`** element 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 heve 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 of the file pointer

```
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

Read from a common file 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

Write in a common file using individual file pointers

File offset determined by MPI_File_seek

```
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
```

Explicit offset - Write

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

– An explicit offset is given to determine access position

– The file pointer is neither used or incremented or modified

– Blocking, independent.

– Explicit seek can be avoided…

– 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 - Write

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

- An explicit offset is given to determine access position
- The file pointer is neither used or incremented or modified
- Blocking, independent.
- Explicit seek can be avoided…

- 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

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

– An explicit offset is given to determine access position

– The file pointer is neither used or incremented or modified

– Blocking, independent.

– Explicit seek can be avoided…

– Reads **COUNT** elements of **DATATYPE** from the file into memory

– **DATATYPE** defines where the data is placed in memory

– **EOF** is reaches when elements read is different from **COUNT**

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

Read from a common file 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

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

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

```
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int 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

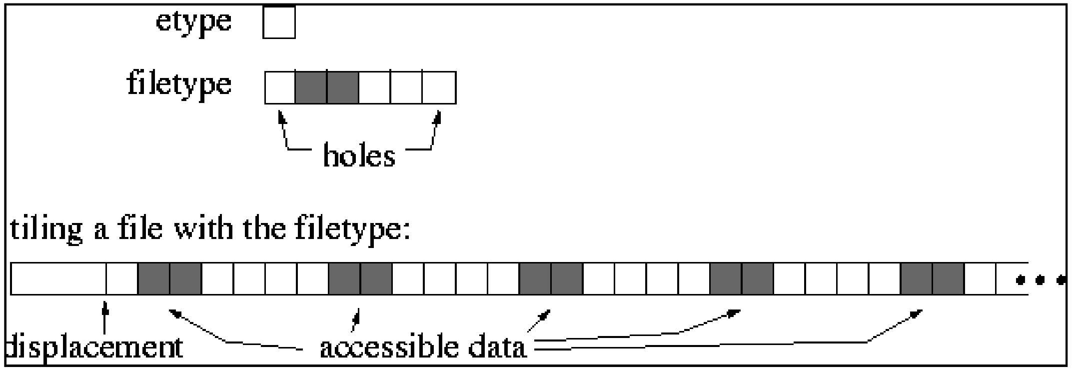
```
int MPI_File_get_position_shared(MPI_File fh, MPI_Offset* offset)
```

- 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

File view

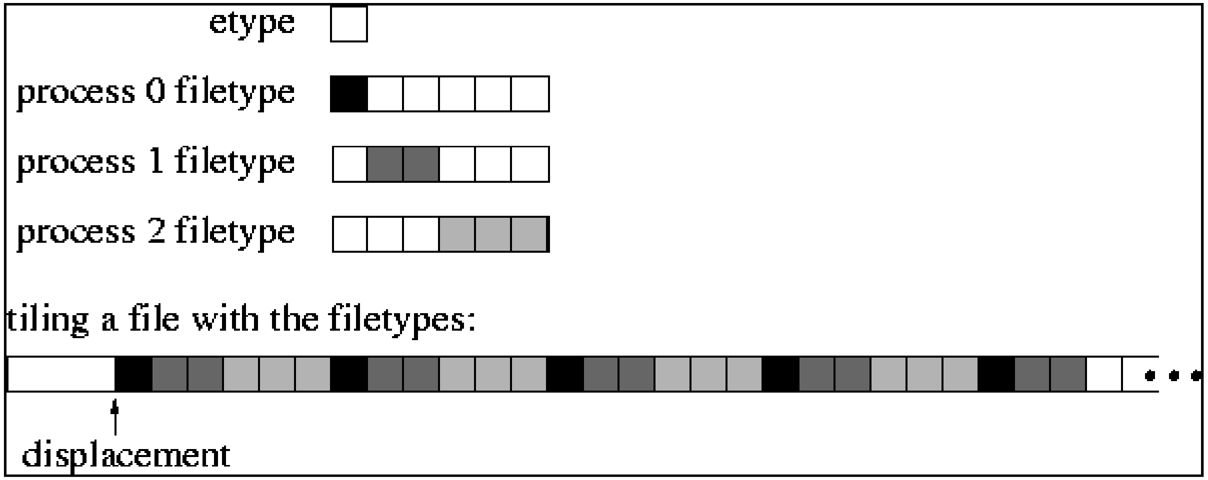
- A file view defines which portion of a file is “visible” to a process
- File view defines also the type of the data in the file (byte, integer, float, …)
- By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file
- A file view consists of three components
  - displacement : number of bytes to skip from the beginning of file
  - etype : type of data accessed, defines unit for offsets
  - filetype : portion of file visible to process same as etype or MPI derived type consisting of etypes
- 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



Etype
An etype is unit of data access and positioning.

Filetype
The filetype describes the access pattern of the processes on the file.



File View
Part of the file which is visible to a process.

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.

– MPI provides functions for creating datatypes for subarrays which can be used in the filetype argument.

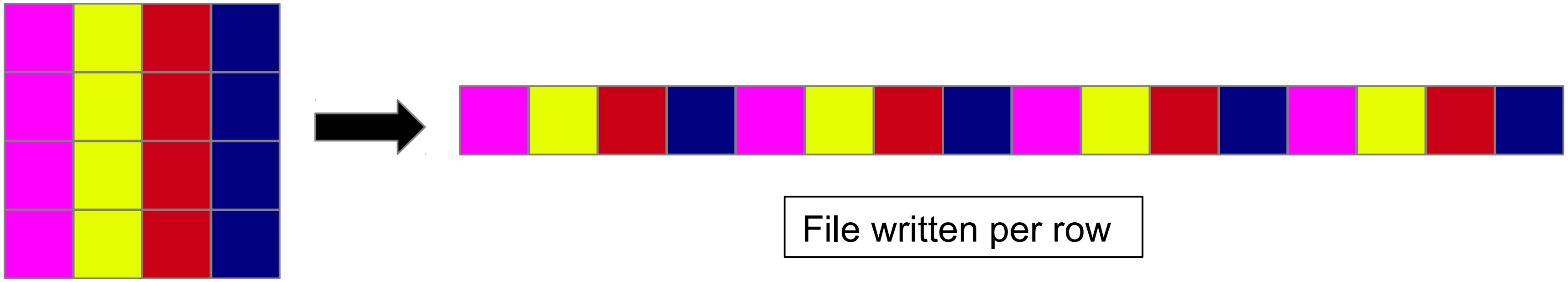
Data representation in the file view

native: highest performance – data are written as they are in memory

internal: implementation-defined. If necessary data are converted – useful for heterogeneous distributed computing platforms

external32: highest portability: All floating point values are in big-endian 32-bit IEEE format

File view for non-contiguous data: filetype



File written per row

2D-array distributed column-wise

- 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

File view for non-contiguous data: filetype

2D-array distributed column-wise

File written per row

```
...
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)
```

Collective, blocking I/O

I/O can be performed collectively by all processes in a communicator

Same parameters as in independent I/O 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 I/O

```
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 I/O ALL the processors defined in a communicator execute the I/O operation

This permits to optimize the read/write procedure

It is particularly effective for non atomic operations

Non-contiguous access with a single collective I/O function 1/2

```
#include “mpi.h”
#define FILESIZE      1048576
#define INTS_PER_BLK  16

int main(int argc, char **argv){
   int *buf, rank, nprocs, nints, bufsize;
   MPI_File fh;
   MPI_Datatype filetype;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   bufsize = FILESIZE/nprocs;
   buf = (int *) malloc(bufsize);
   nints = bufsize/sizeof(int);

   MPI_File_open(MPI_COMM_WORLD, “/pfs/datafile”, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
   MPI_Type_vector(nints/INT_PER_BLK, INTS_PER_BLK, INTS_PER_BLK*nprocs, MPI_INT,
    &filetype);
   MPI_Type_commit(&filetype);
```

Non-contiguous access with a single collective I/O function 2/2

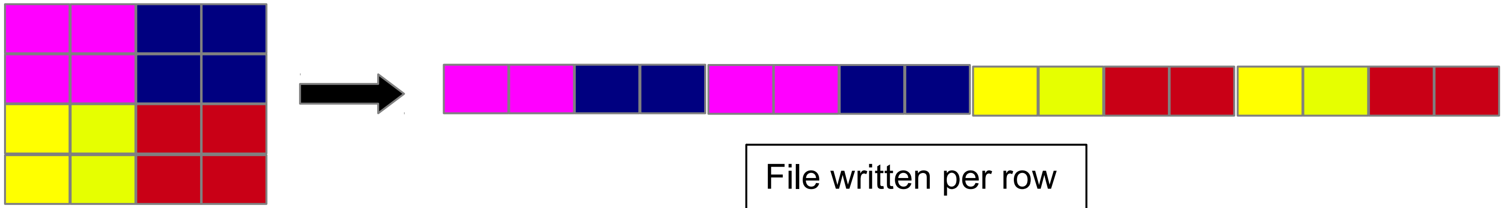
```
#include "mpi.h"
#define FILESIZE      1048576
#define INTS_PER_BLK  16

int main(int argc, char **argv){
   /* declaration part */
   /* MPI initialization */
   /* settings of buf size */

   MPI_File_open(...);
   MPI_Type_vector( filetype ); MPI_Type_commit(&filetype);

   MPI_File_set_view(fh, INTS_PER_BLK*sizeof(int)*rank, MPI_INT, filetype,
        "native", MPI_INFO_NULL);
   MPI_File_read_all(fh, buf, nints, MPI_INT, MPI_STATUS_IGNORE);
   MPI_File_close(&fh);
   MPI_Type_free(&filetype);
   free(buf);
   MPI_Finalize();
   return 0;
}
```

Storing multidimensional arrays

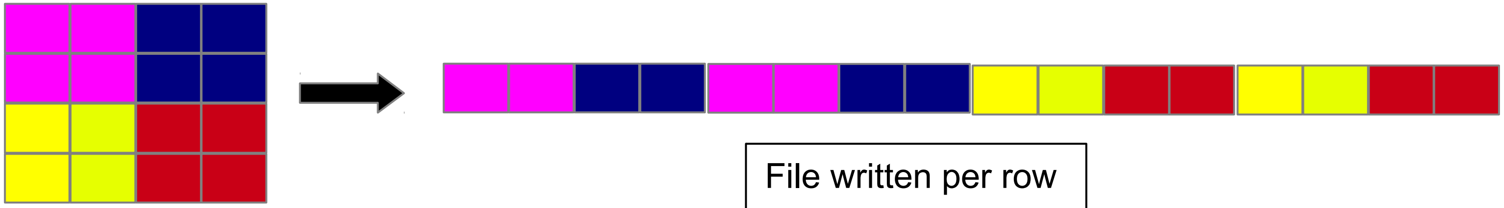


Domain decomposiztion for 2D-array

File written per row

```
...
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 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)
```

Storing multidimensional arrays: Collective I/O



Domain decomposiztion for 2D-array

File written per row

Collective write can be over hundred times faster than the individual for large arrays!

```
...
INTEGER :: sizes = (/4, 4/)
INTEGER :: subsizes = (/2, 2/)
INTEGER, DIMENSION(2,2) :: buf
...
CALL 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 MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype,&
         ‘native’, MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE_ALL(file, buf, count, MPI_INTEGER, status, err)
...
```

Darray and collective I/O 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) */
```

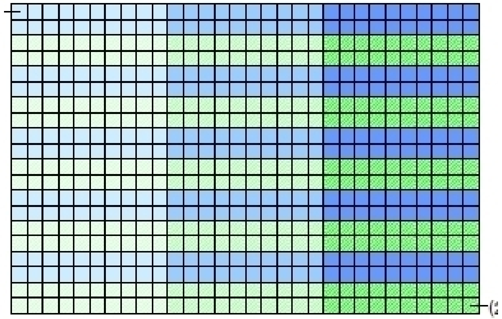
```
int gsizes[2], distribs[2], dargs[2], psizes[2];

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 */
```



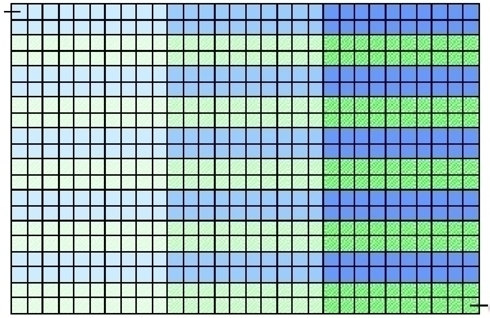
Darray and collective I/O 2/2

```
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 I/O

This is just like non blocking communication.

Same parameters as in blocking I/O 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 I/O with computation

Collective, nonblocking I/O

For collective I/O only a restricted form of nonblocking I/O 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

QUESTIONS ???