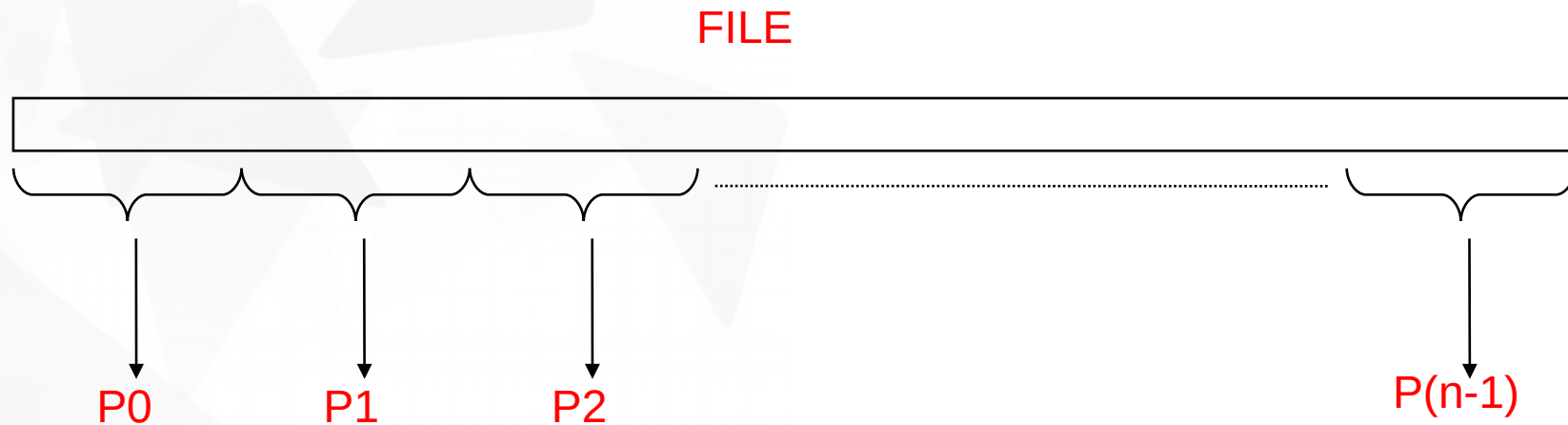# MPI-IO part 2

- Stefano Cozzini
- CNR-IOM and eXact lab srl

Agenda

- Short recap: again on File View

- Collective Operations

- Right Way to access the data
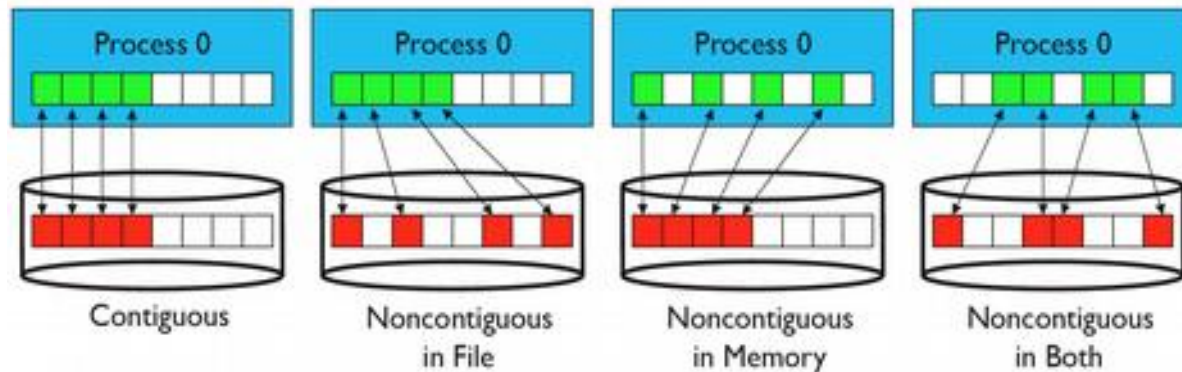
- A final exercise (for MHPC students)

Short recap:

What MPI-I/O is dealing with...

FILE

P0   P1   P2   P(n-1)

Each process needs to read
a chunk of data from a common file

Which kind of data pattern
are we performing here ?

Process 0   Process 0   Process 0   Process 0

Contiguous   Noncontiguous   Noncontiguous   Noncontiguous
                   in File   in Memory   in Both

But do we need MPI for this ?

- Regular Posix I/O functions can do contiguous access...
  - `lseek` (C System Call)
    - `lseek` is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.
  - Be careful however about lock mechanism !

What about this case ?
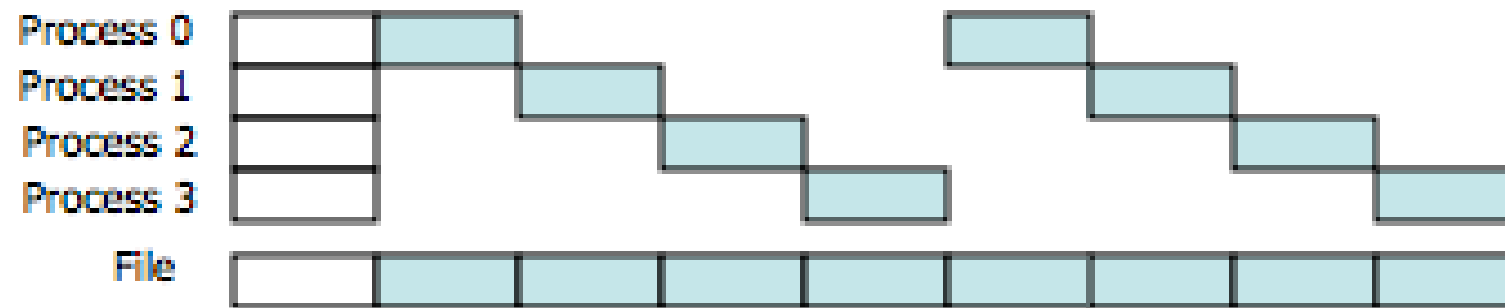
- Simple way:
  - 2 separated calls
    - Read first chunk
    - Read the second chunk

Extremely inefficient !

MPI I/O approach

- Ability to access NON contiguous data with a single function call

MPI notion of file view

- File view in MPI defines which portion of a file is *visible* to a process
- When a file is first open it is entirely visible to all processes
- The file view of each process can be changed by means of `MPI_File_set_view`
- Read/Write function will be able to "see" only the visible portion of the file
- `MPI_File_set_view` assigns regions of the file to separate processes

File Views

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

Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **MPI_File_set_view**

- *displacement* = number of bytes to be skipped from the start of the file

- *etype* = basic unit of data access (can be any basic or derived datatype)

- *filetype* = specifies which portion of the file is visible to the process (same as etype or derived type consisting of etype)

- Default view: displacement 0 / etype filetype =MPI_BYTE/

Note !

The pattern described by a filetype is repeated, beginning at the displacement, to define the view within the file..

etype = MPI_INT

filetype = two MPI_INTs followed by a gap of four MPI_INTs

head of file

FILE

displacement   filetype   filetype   and so on...

File View basic example: contiguous access

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
                MPI_MODE_CREATE | MPI_MODE_WRONLY,
                MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank *BUFSIZE * sizeof(int),
 MPI_INT, 4*MPI_INT, "native",
                    MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
                MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```

See in your github account for this complete example

Quick introduction to Derived Data Types

- What are they?
  - Data types built from the basic MPI datatypes. Formally, the MPI Standard defines a general datatype as an object that specifies two things:
    - a sequence of basic datatypes
    - a sequence of integer (byte) displacements
- How to use them ?
  - Construct the datatype using a template or constructor.
  - Allocate the datatype
  - Use the datatype.
  - Deallocate the datatype.

You must construct and allocate a datatype before using it.
You are not required to use it or deallocate it, but it is recommended

Main functions

- Datatype constructors:
  - MPI_TYPE_CONTIGUOUS (count, oldtype, newtype)
    - Simplest constructor. Makes count copies of an existing datatype(oldtype) into newtype
  - MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype)
    - Make count copies of a block of length blocklength and allows for regular gaps (stride) in the displacements.
- Allocate and deallocate
  - int MPI_Type_commit (MPI_datatype *datatype)
  - int MPI_Type_free (MPI_datatype *datatype)

Let us solve exercises

- Take a look at the F90 code where file_set_view routine is introduced

- Compile the code and compare results with output of writeFile_pointer.f90

- Modify the code to use file_set_view using mpi derived data_type (MPI_TYPE_CONTIGUOUS or MPI_TYPE_VECTOR) and get the same results as the code you start from.

Exercise 4 optional

- Write contiguous data into a contiguous block using file view
- Use derived data type to define filetype in the file view.

P0	1	2	3	4
P1	11	12	13	14
P2	21	22	23	24
P3	31	32	33	34

1	2	3	4	11	12	13	14	21	22	23	24	31	32	33	34

Solution to exercise 4:

```fortran
integer ierr, i, myrank, BUFSIZE, thefile
parameter (BUFSIZE=4)

….

!!define a contiguous derived datatype
call MPI_TYPE_CONTIGUOUS(BUFSIZE,MPI_INTEGER, filetype,
ierr)
call MPI_TYPE_COMMIT(filetype, ierr)

disp = myrank * BUFSIZE * intsize

write(*,*) "myid ",myrank,"disp ", disp

call MPI_FILE_SET_VIEW(thefile, disp, etype, &
                       filetype, 'native', &
                       MPI_INFO_NULL, ierr)
```
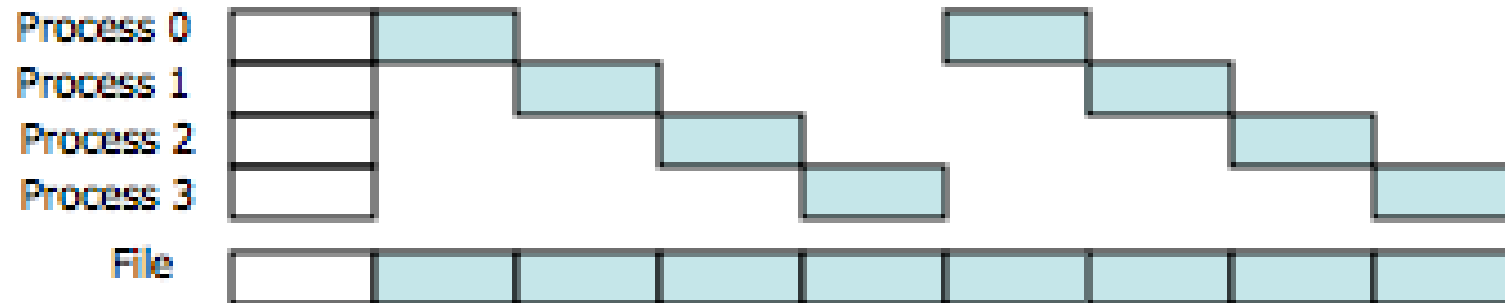
….

File view example/exercise 4b

- Write a file with the following layout:

1	2	11	12	21	22	31	32	3	4	13	14	23	24	33	34

File view example/exercise 4b

```fortran
!!define a different pattern by means of MPI_TYPE_VECTOR
!!    first parameter: number of global element
!!    second parameter: number of blocks
!!    third parameter:  stride between block
!!
call MPI_TYPE_VECTOR(bufsize/2,nprocs/2,2*nprocs, &
    MPI_INTEGER,filetype,ierr)
call MPI_TYPE_COMMIT(filetype, ierr)

disp = (bufsize/2) * myrank * intsize
 write(*,*) "myid ",myrank," disp ", disp


...
call MPI_FILE_SET_VIEW(thefile, disp, etype, &
                        filetype, 'native', &
                        MPI_INFO_NULL, ierr)
```

MPI properties

POSITIONING

SYNCHRONIZATION

COORDINATION

MP-IO properties : positioning

- Positioning
    - Use individual file pointers:
        - call MPI_File_seek/read
    - Calculate byte offsets:
        - call MPI_File_read_at
    - Access a shared file pointer:
        - call MPI_File_seek_shared/read_shared

MP-IO properties : SYNCHRONIZATION

- Synchronization:
  - MPI-2 supports both blocking and non-blocking IO routines
    - A blockingIO call will not return until the IO request is completed.
    - A non blocking IO call initiates an IO operation, but not wait for its completion

Nonblocking I/O

```
MPI_Request request;
MPI_Status status;

MPI_File_iwrite_at(fh, offset, buf, count,
datatype,
                        &request);


for (i=0; i<1000; i++) {
    /* perform computation */
}


MPI_Wait(&request, &status);
```

MP-IO properties : 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
        - Collective I/O functions must be called by all processes participating in I/O
        - Allows I/O layers to know more about access as a whole

    - independent: no coordination by MPI
        - No apparent order or structure to accesses

Collective I/O operations (1)

Independent I/O
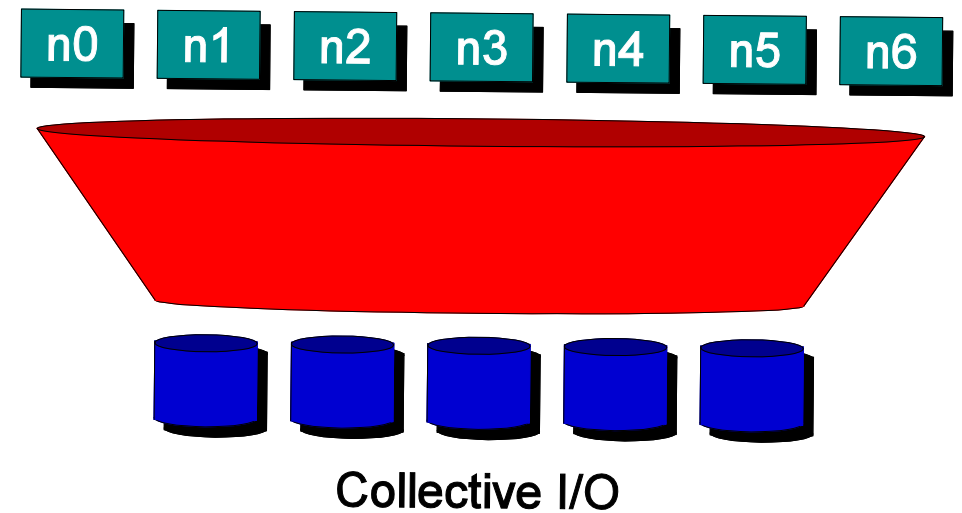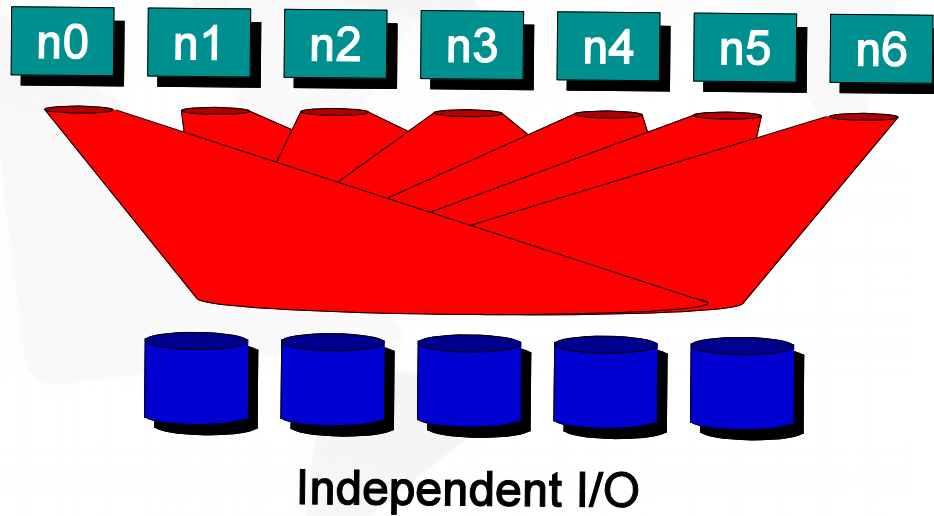
Collective I/O

Collective I/O operations (2)

- `MPI_File_read_all`, `MPI_File_read_at_all`, etc
- `_all` indicates that all processes in the group specified by the communicator passed to `MPI_File_open` will call this function
- Each process specifies only its own access information -- the argument list is the same as for the non-collective functions

Collective I/O operations (3)

- Collective I/O is a critical optimization strategy for reading from, and writing to, the parallel file system
- The collective read and write calls force all processes in the communicator to read/write data simultaneously and to wait for each other
- The MPI implementation optimizes the read/write request based on the combined requests of all processes and can merge the requests of different processes for efficiently complete the requests

Optiming MPI operations..

- Given complete access information, an implementation can perform optimizations such as:
  - Data Sieving: Read large chunks and extract what is really Needed
  - Collective I/O: Merge requests of different processes into larger requests

Collective MPI operations: collective buffering

- breaks the IO operation into two stages.
  - first stage uses a subset of MPI tasks (called aggregators) to communicate with the IO servers and read a large chunk of data into a temporary buffer.
  - second stage, the aggregators ship the data from the buffer to its destination among the remaining MPI tasks using point-to-point MPI calls.
  - PRO/Cons
    - fewer nodes are communicating with the IO servers, which reduces contention while still attaining high performance through concurrent I/O transfers.
    - Two stages operation: not so ead

Collective MPI operations: data sieving

- For independent noncontiguous requests
- ROMIO makes large I/O requests to the file system and, in memory, extracts the data requested
- For writing, a read-modify-write is required
- Pro/Cons
  - data is always accessed in large chunks, although at the cost of reading more data than needed. For many common access patterns, the holes between useful data are not unduly large, and the advantage of accessing large chunks far outweighs the cost of reading extra data.
  - In some access patterns, however, the holes could be so large that the cost of reading the extra data outweighs the cost of handling such cases as well.
  - The implementation can decide whether to perform data sieving or access each contiguous data segment separately.

A final summary:

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

Right way to access data

Using the Right MPI-IO Function

- Any application as a particular "I/O access pattern" based on its I/O needs
- The same access pattern can be presented to the I/O system in different ways depending on what I/O functions are used and how
- We classify the different ways of expressing I/ O access patterns in MPI-IO into four levels: level 0 – level 3
- We show how the user's choice of level affects performance

Distribute array access

Large array distributed among 16 processes

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Each square represents a subarray in the memory of a single process

Access Pattern in the file

| P0 | P1 | P2 | P3 | P0 | P1 | P2 |

| P4 | P5 | P6 | P7 | P4 | P5 | P6 |

| P8 | P9 | P10 | P11 | P8 | P9 | P10 |

| P12 | P13 | P14 | P15 | P12 | P13 | P14 |

Level 0 access

- Each process makes one independent read request for each row in the local array (as in Unix)

```
MPI_File_open(..., file, ..., &fh);
for (i=0; i<n_local_rows; i++)
    { MPI_File_seek(fh, ...);
      MPI_File_read(fh, &(A[i][0]), ...); }
MPI_File_close(&fh);
```

Level 1 access

- Each process makes one independent read request for each row in the local array (as in Unix) but each process uses collective I/O functions

```
MPI_File_open(..., file, ..., &fh);
for (i=0; i<n_local_rows; i++)
   { MPI_File_seek(fh, ...);
     MPI_File_read_all(fh, &(A[i[0]), ...); }
MPI_File_close(&fh);
```

Level 2 access

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

```
MPI_Type_create_subarray(...,
  &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(..., file, ..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read(fh, A, ...);
MPI_File_close(&fh);
```

Level 3 access

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

```
MPI_Type_create_subarray(...,
  &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(..., file, ..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read_all(fh, A, ...);
MPI_File_close(&fh);
```

The four level
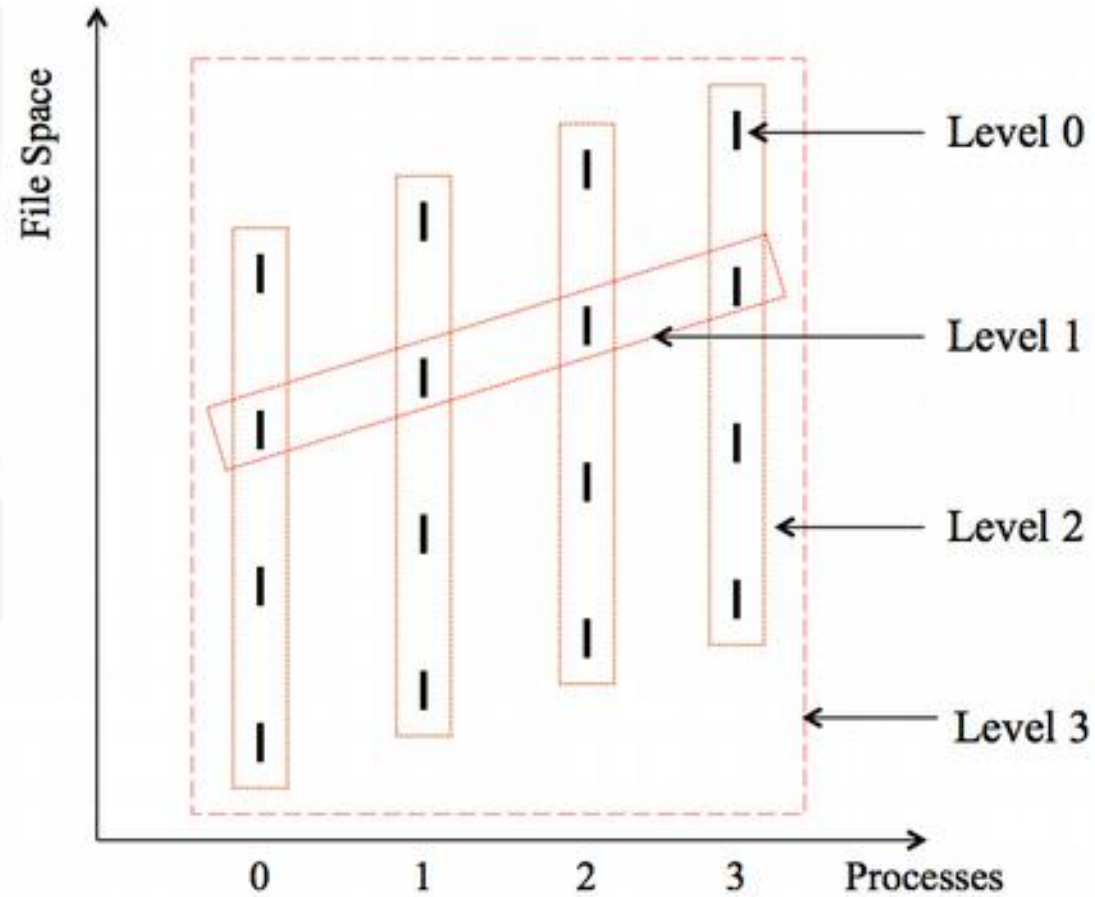
Conclusions

A very short summary:

- MPI-I/O important features are:
  - The ability to specify noncontiguous accesses
  - The collective I/O functions
  - The ability to pass hints to the implementation

Links/Reference

- MPI –The Complete Reference vol.2, The MPI Extensions (W.Gropp, E.Lusketal. -1998 MIT Press)

- Using MPI-2: Advanced Features of the Message- Passing Interface (W.Gropp, E.Lusk, R.Thakur-1999 MIT Press)

- Standard MPI-2.x (or the last MPI-3.x) (http://www.mpi-forum.org/docs)

- Users Guide for ROMIO (Thakur, Ross, Lusk, Gropp, Latham)

  (http://www.mcs.anl.gov/research/projects/romio/doc/users-guide.pdf)

- http://beige.ucs.indiana.edu/I590/node86.html