MPI-IO part 2

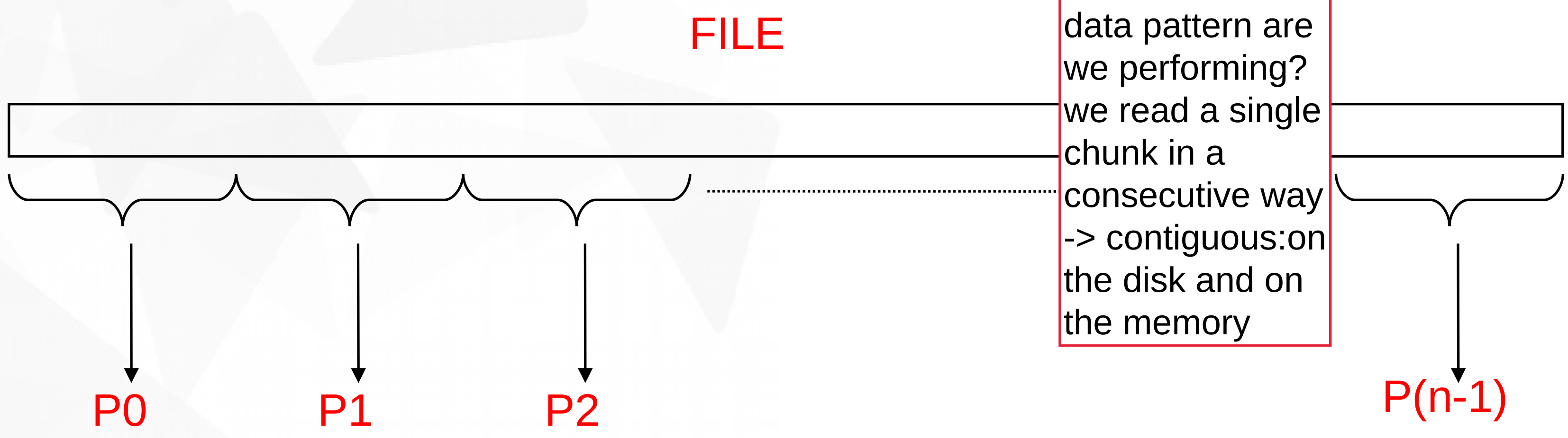
- Stefano Cozzini
- CNR-IOM and eXact lab srl

Agenda

- Short recap: again on File View
- Collective Operations
- MPI_HINTS
- A final exercise (for MHPC students)

Short recap :

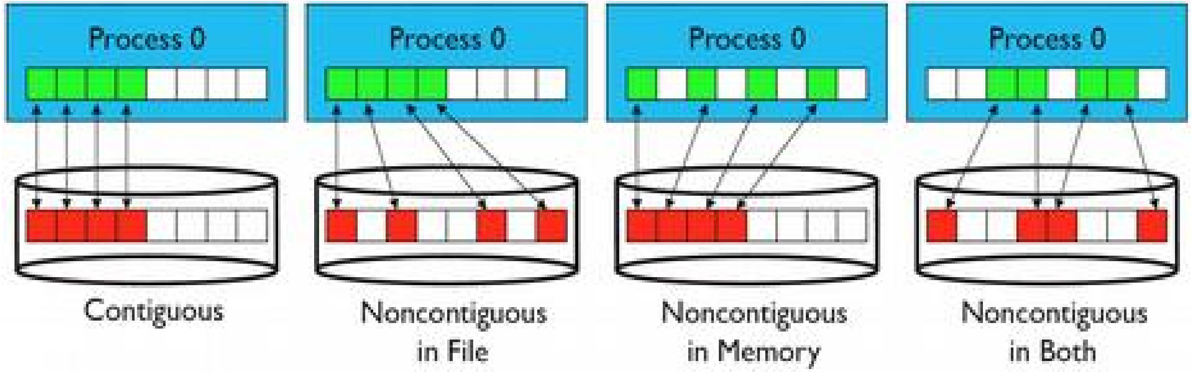
What MPI-I/O is dealing with...



a shared unique file, each processor needs to read a chunk of data from it
which kind of data pattern are we performing? we read a single chunk in a consecutive way -> contiguous:on the disk and on the memory

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

Which kind of data pattern are we performing here ?



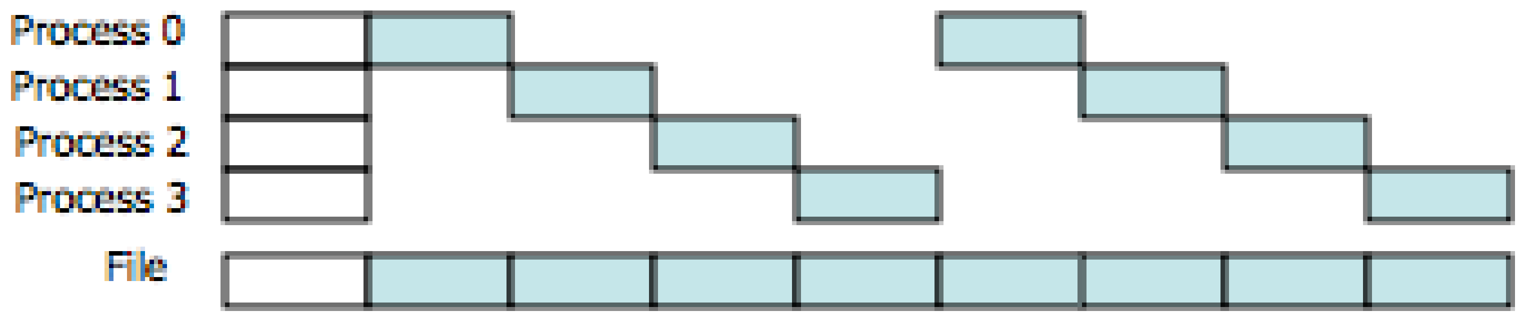
But do we need MPI for this ?

- Regular Posix I/O functions can do contiguous access…
  - `lseek` (C System Call)
    - `lseek` is a system call tha… e the location of the read/write pointer of a… he location can be set either in absolute or re…
  - Be careful however about lo…

interact with the file system, system call lseek, exactly what we did on mpi: we can do it without mpi
lseek works well

beware of lock mechanismo
posix is atomic on some operations, locks them.
in any case just contiguous access:
lock meckanism: atomic operation on the file concurrent access on the file, then processes are queued

What about this case ?



- Simple way:
  - 2 separated calls
    - Read first chunk
    - Read the second chunk

Extremely ine

mpi with different kind of approach than contiguos
here non contiguius acces of file, can be done on C opening and closing files, very inefficent: because I have to open the file twice, for each reading i read only a small chunk, dobling the latency of my iosystem, get doen to the disk for a small amount of data.
if there is a complex non.contiguous pattern it takes ages. You have to reconsider the seek operation. seeking a new position is a very expensive operation.

MPI I/O approach

strong point of open MPI

- 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 a[illegible]ee” only the visible portion of the file

you see non-contiguos acces to file, reduce latency with just one command

- `MPI_File_set_view` assigns regions of the file to separate processes

File Views

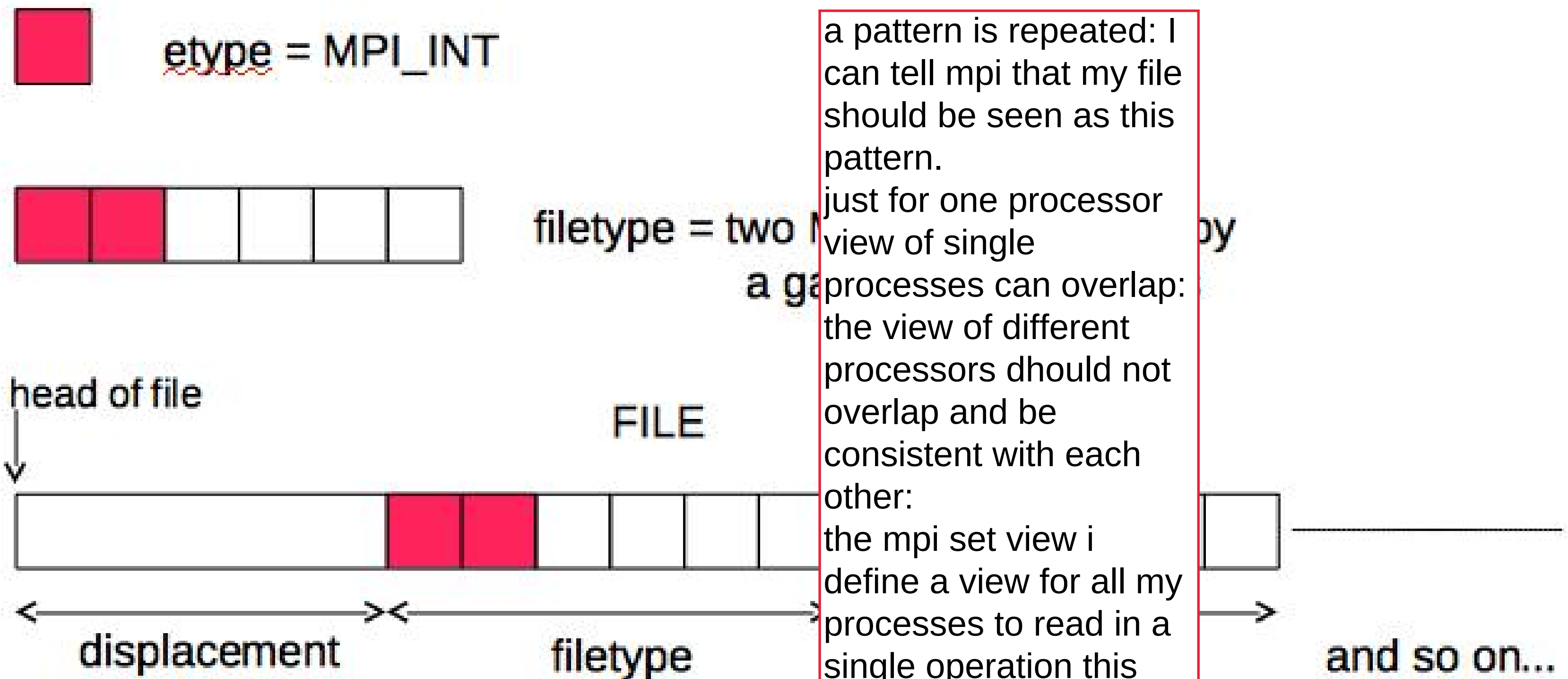
```
int MPI_File_set_view(MPI_File fh,
MPI_Offset displacement,
MPI_Datatype etype, MPI_Datatype filetype,
char *datarep, MPI_Info info)
```

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..



a pattern is repeated: I can tell mpi that my file should be seen as this pattern.
just for one processor view of single processes can overlap: the view of different processors dhould not overlap and be consistent with each other:
the mpi set view i define a view for all my processes to read in a single operation this non contiguous access pattern

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, "tes
                MPI_MODE_CREATE | M          RONLY,
                MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
                    MPI_INT, MPI_INT, "native",
                  MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
                 MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```

we open a file: each processor sees exacty what portion of data they must see mpi_file_set_view-->ordered by rank, mpi int

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 dataty… I Standard defines a general datatype as an… two things:
    - a sequence of basic datatypes
    - a sequence of integer (byte) displacements
- How to use them ?
  - Construct the datatype using a template
  - Allocate the datatype
  - Use the datatype.
  - Deallocate the datatype.

You must construct and allocate a datatype before u…
You are not required to use it or deallocate it, but it is…

to allow contiguous approach: use derived data types
(we dont really need so just to show an alternative to the past slide)
datatypes derived from datatypes
defined by sequences of basic datatypes, sequence of integer displacements
each process

castings make the program unreadable

read integers
optimization not introduced when developing

dont do sophisticated operations, don't introduce additional opeartions, do them when you need them

premature optimization is the root of all evils

IO is the slowes operation on the computer, come to the io as last
computer turns cpu-bound problem to IO-bound problem

Main functions

- Datatype constructors:
  - MPI_TYPE_CONTIGUOUS (count, e)
    - Simplest constructor. Makes count cop atype(oldtype) into newtype
  - MPI_TYPE_VECTOR (count, blockl dtype, newtype)
    - Make count copies of a block of length vs for regular gaps (stride) in the displacements.
- Allocate and deallocate
  - C
    - int MPI_Type_commit (MPI_datatype *
    - int MPI_Type_free (MPI_datatype *dat

two different datatypes
strides, regular gaps

mpi _type_commit: i can use the new datatype, free to deallocate

i define a new datatype from an old datatype, contiguous types of old datatype
I take a standard data type and i make a sequence of them
Vector, there is stride in between
stride:space between two readings
I define a 2d array i wanna built a datatype to read just rows, fortran reads by column, implementationbehind resolves all location in memory

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 outp writeFile_pointer.f90
- Modify the code to use file_set_view using mpi d data_type (MPI_TYPE_CONTIGUOUS or MPI_TY d get the same results as the code you start from.

derived datatypes:same thing as building structures in C++
etype specifies the size, the second type is filetype explains the way you are reading
when i use file set view you read an integers and read four of them, specify file type

Exercise 4 optional

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

read 4 elements one after the other

P0	1	2	3	
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:

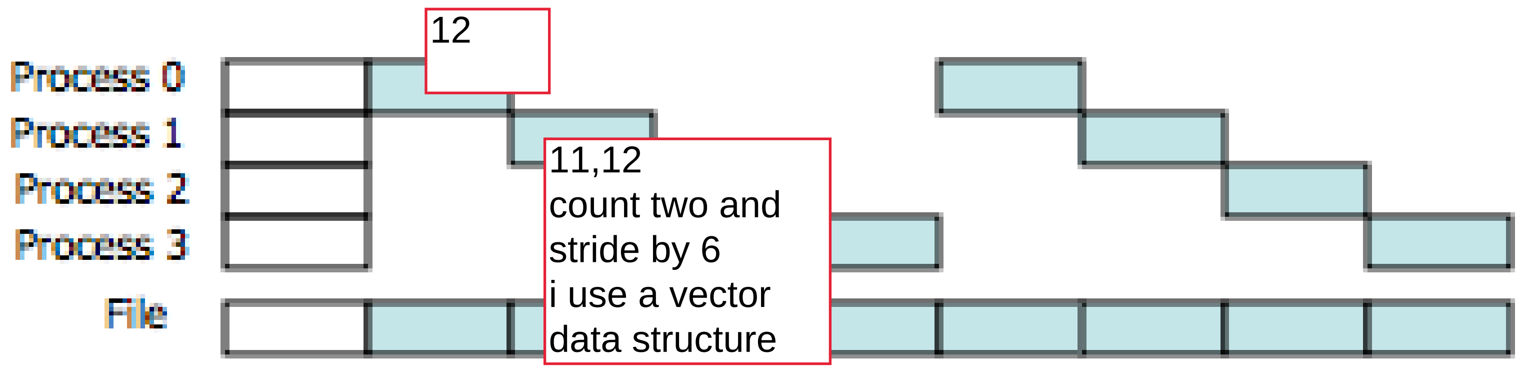
countiguous elemen is four elements one after the other disp must be computed in bytes

```
integer ierr, i, myrank, BUFSIZE, t
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

```
!!define a different pattern by mea TYPE_VECTOR
!!    first parameter: number of gl nt
!!    second parameter: number of b
!!    third parameter:  stride betw
!!
call MPI_TYPE_VECTOR(bufsize/2,npro ocs, &
    MPI_INTEGER,filetype,ierr)
call MPI_TYPE_COMMIT(filetype, ierr

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

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

bufsize=4
i wnat 2 elements, then i want two blocks
i have 4 processors and i divide by 2
now jump away two elements, two blocks, now the stride from two blocks, stride is actually 8, between blocks nprocs, means we have 4 blocks between two blocks

MPI properties

three main properties positioning seek and wathever synchronization: blocking and non blocking coordination collecting operations together

POSITIONING

SYNCHRONIZATION

COORDINATION

MP-IO properties : positioning

positioning in files

- 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 block on-blocking IO routines
    - A blockingIO call will no ntil the IO request is completed.
    - A non blocking IO call in IO operation, but not wait for its completion

write ina
blocking and
non blocking
way

mpi_file_iwrite_at is nonblocking

MPI_wait
waiting for
cpomplection of
mpi file wrtie at

Nonblocking I/O

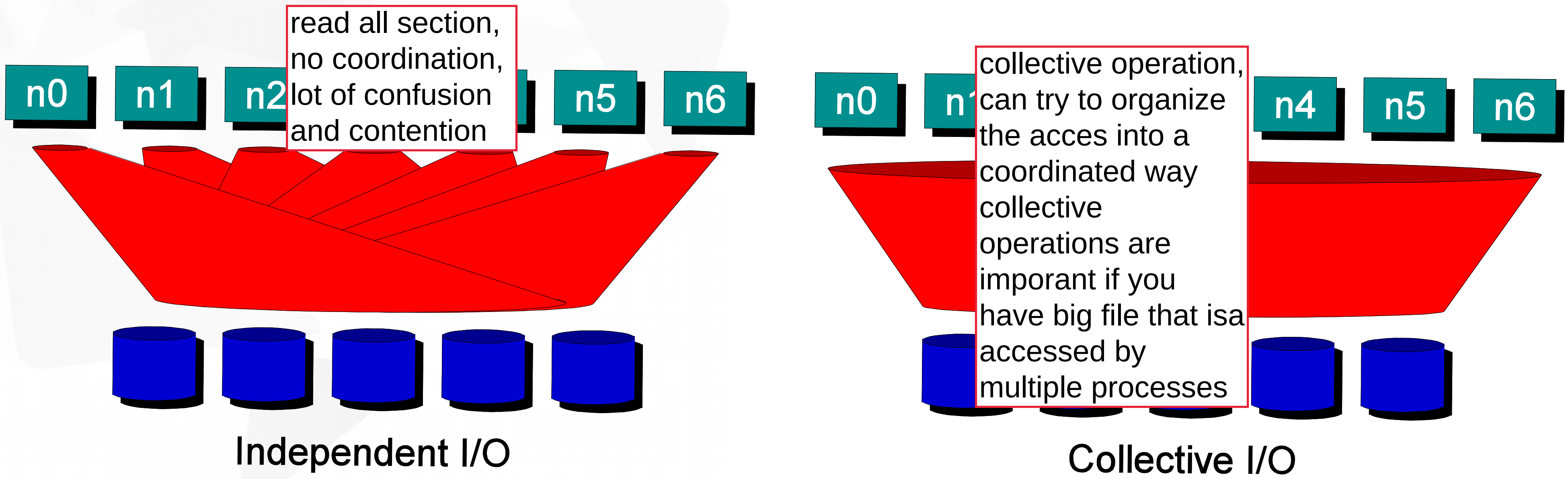
```
MPI_Request request;
MPI_Status status;

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

when we opena file is a colective opearation
every process opens it
coordinates read and write of dfferent
processoes. no appaarent order to acces

- 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)



Collective I/O operations (2)

pass throught mpi_pi_open tries to read this file in a coordinated way
attach all at the end:

- **`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)

io operations, to optimize program access collective read allows processors to read data altogether and wait ina coordinated way

- Collective I/O is a crit ization strategy for reading from, and writing to, the p system
- The collective read a alls force all processes in the communicator to rea ata 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 of different processes into larger requests

read the file in large chunks and throw away what you dont need. I read a lot of stuff and consider a subsection of them

1.combine the requests
2.do input output operations in a collective way

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 ea

a node has a certain number of processors, if i do a collective operation

Collective MPI operations: data sieving

- For independent nonconti quests
- ROMIO makes large I/O re the file system and, in memory, extracts the data d
- For writing, a read-modify- equired
- 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.

try to read large request from file, then on memory dump to the disk what you need

data accessed in large chunks: read more than what you need

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
	…cking & …ective	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

we didnt discuss this

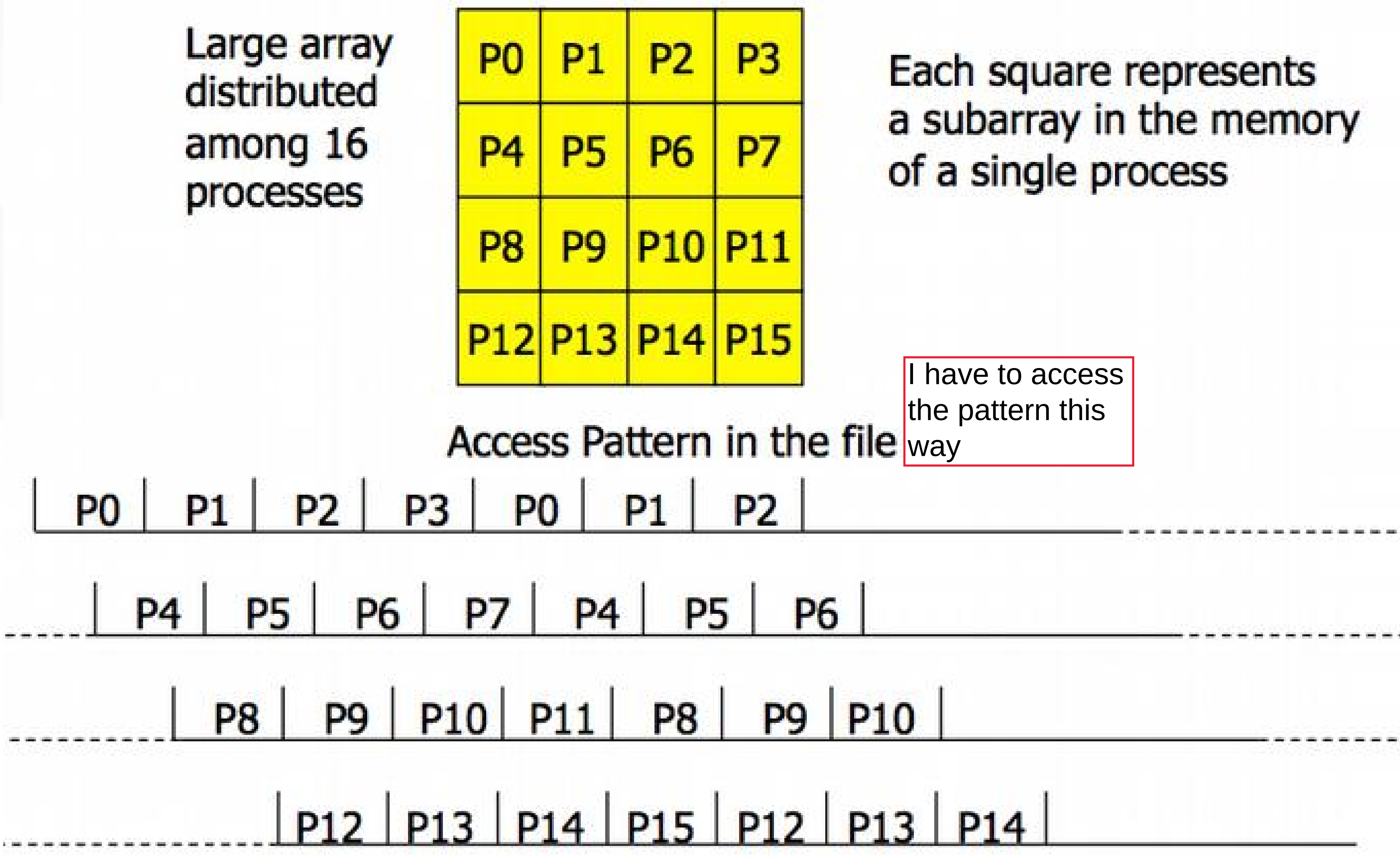
Right way to access data

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

example taken from the book of will gropp chapter 7, understand how to read fie

Distribute array access



I have to access the pattern this way

Level 0 access

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

I open the file and each of the process reads his portion of file, everybody reads by themselves

```
MPI_File_open(..., file[illegible]&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

I can do it in a coordinated way, add an all, if we have some contentions the net effect is sometimes negligible

- Each process make pendent 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

no more loop
i can tell read
this file this way
using file set
view

- Each process creates datatype to describe the noncontiguous acces 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);
```

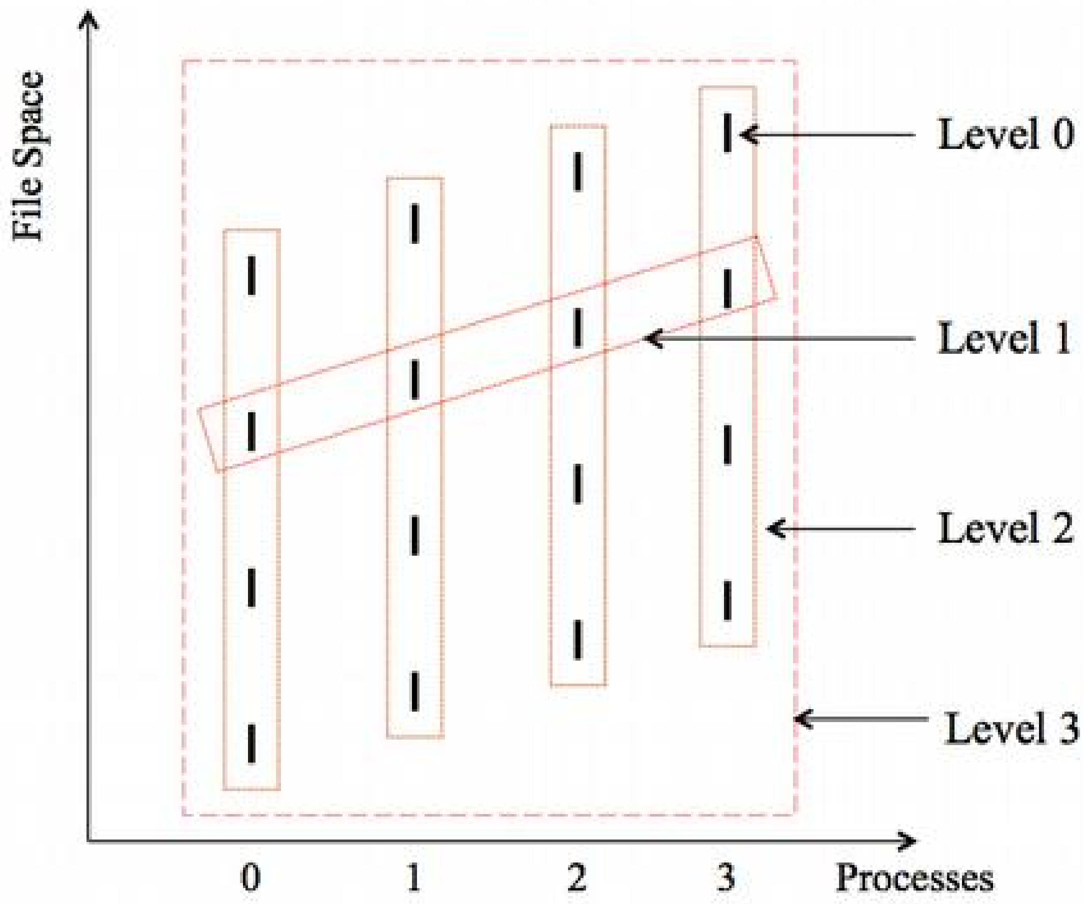
Level 3 access

- 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, ...);
MPI_File_close(&fh);
```

four operations
done altogether

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