MPI-IO

- Stefano Cozzini
- CNR-IOM and eXact lab srl

MPI-I/O

- I/O interface specification for use in MPI apps
- Available in MPI-2.0 standard on
- Data model is a stream of bytes in a file
  - Same as POSIX and stdio
- Features:
  - Noncontiguous I/O with MPI datatypes and file views
  - Collective I/O
  - Nonblocking I/O
  - Fortran/C bindings (and additional languages)
- API has a large number of routines..

NOTE: you simply compile and link as you would any normal MPI program.

Why MPI is good for I/O ?

- Writing is like sending a message and reading is like receiving one.
- Any parallel I/O system will need to
    - define collective operations (*MPI communicators*)
    - define noncontiguous data layout in memory and file (*MPI datatypes*)
    - Test completion of nonblocking operations (*MPI request objects*)
- i.e., lots of MPI-like machinery needed

Parallel I/O in MPI

- Why do I/O in MPI?
  - Why not just POSIX?
    - Parallel performance
      - Single file (instead of one file / process)
- MPI has replacement functions for POSIX I/O
  - Provides migration path
- Multiple styles of I/O can all be expressed in MPI
  - Including some that cannot be expressed without MPI

The basic: an example

Just like POSIX I/O, you need to

- Open the file
- Read or Write data to the file
- Close the file

In MPI, these steps are almost the same:

- Open the file: `MPI_File_open`
- Write to the file: `MPI_File_write`
- Close the file: `MPI_File_close`

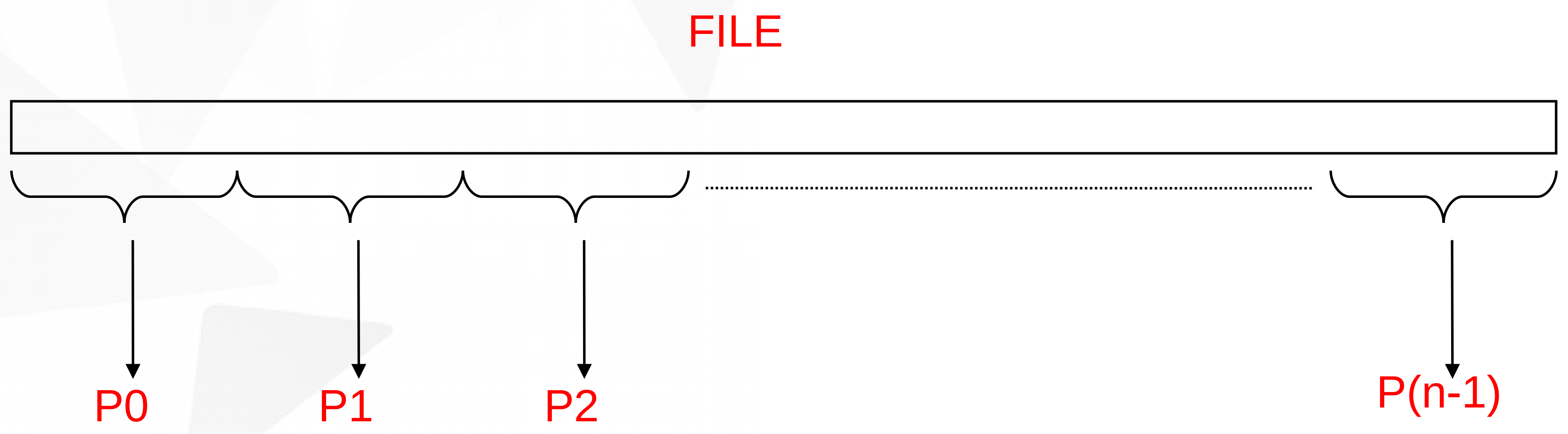
A simple C example

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
MPI_File fh;
int buf[1000], rank;
MPI_Init(0,0);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_File_open(MPI_COMM_WORLD, "test.out",MPI_MODE_CREATE|MPI_MODE_WRONLY,
MPI_INFO_NULL, &fh);
if (rank == 0) {MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);}
MPI_File_close(&fh);
MPI_Finalize();
return 0;
}
```

Some comments

- File Open is collective over the communicator
  - Will be used to support collective I/O, which is important for performance
  - Modes similar to Unix open
  - MPI_Info provides additional hints for performance
- File Write is independent (hence the test on rank)
  - Many important variations covered in later slides
- File close is collective; similar in style to `MPI_Comm_free`

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



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

How to do that ?

```
/* from Gropp's book: page 189*/
#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;
   buf= (int *) malloc(bufsize);
   nints = bufsize/sizeof(int);

   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;
}
```

MPI-I/O API Opening/Closing file

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

- Collective operation
  - All processes have to provide the same amode
  - comm must be an intra-communicator
- To close the file:

```
int MPI_File_close(MPI_File *fh)
```

MPI-I/O amode values

- MPI_MODE_RDONLY : read only
- MPI_MODE_WRONLY : write only
- MPI_MODE_RDWR: read and write
- MPI_MODE_CREATE : create file if it doesn’t 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

- Combination of several amodes possible, e.g
- – C: (MPI_MODE_CREATE | MPI_MODE_WRONLY)
- – Fortran: MPI_MODE_CREATE + MPI_MODE_WRONLY

Some more observations

- Collective operations across processes within an MPI communicator.
  - Filename must be unique for 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 via 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.

File pointer and offset

- In simple MPI-I/O, each MPI process reads or writes a single block.
- We have three means of positioning where the read or write takes place for each process:
  - Use individual file pointers:
    - call MPI_File_seek/read
  - Calculate byte offsets:
    - call MPI_File_read_at/File_write_at
  - Access a shared file pointer:
    - call MPI_File_seek_shared/read_shared

- Techniques 1 and 2 are naturally associated with C and Fortran, respectively

MPI-I/O API for reading files

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset,
int whence)
```

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

`whence` in **`MPI_File_seek`** updates the individual file pointer according to

`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 file plus offset (offset could negative)

Reading a file by using individual pointers (C code)

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "your_filename",
              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);
```

See in your github account for this complete example

Reading a file using explicit offset

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

Reading a file using explicit offset (F90)

```
include 'mpif.h'

integer status(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) offset

call MPI_FILE_OPEN(MPI_COMM_WORLD, 'my_output_file', &
          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_GET_COUNT(status, MPI_INTEGER, count, ierr)
print *, 'process ', rank, 'read ', count, 'integers'

call MPI_FILE_CLOSE(fh, ierr)
```

See in your github account for this complete example

MPI-I/O API for writing files

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

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

- To write a file:
  - Set the appropriate flag/s in MPI_File_open: MPI_MODE_WRONLY Or MPI_MODE_RDWR and if needed, MPI_MODE_CREATE
  - Use MPI_File_write or MPI_File_write_at

Write files with offset..

```
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
int i, rank, size, offset, nints, N=16 ;
MPI_File fhw;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int buf[N];
for ( i=0;i<N;i++){
  buf[i] = i ;
  }
offset = rank*(N/size)*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, "datafile",
 & MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fhw);
printf("Rank: %d, Offset: %d\n", rank, offset);
MPI_File_write_at(fhw, offset, buf, (N/size), MPI_INT, &status);
MPI_File_close(&fhw);
MPI_Finalize();
return 0;
}
```

See in your github account for this complete example

Summarizing so far:

MPI_File_open

MPI_File_seek

MPI_File_read

MPI_File_write

Individual pointer functions

MPI_File_close

+

ALL INDEPENDENT I/O OPERATION

MPI_File_read_at

MPI_File_read_at

Explicit offset functions

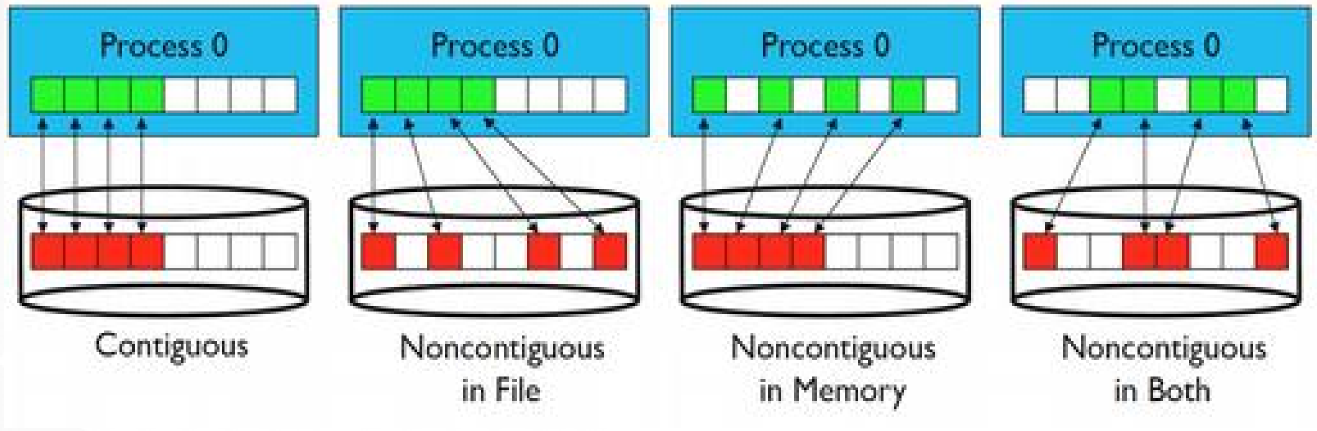
Why do we use indipendent I/O ?

- Sometimes the synchronization of collective calls is not natural
- Sometimes the overhead of collective calls outweighs theirbenefits
  - Example: very small I/O duringv header reads

Are we done ?

- YES:
  - These five routines are enough to write any parallel I/O program
- NO:
  - Other MPI-IO routines are for
    - Performance/ Portability/ Convenience
  - Real benefits comes from:
    - non contiguous access
    - collective I/O

Data pattern: contiguous vs non/contiguous



- Contiguous I/O: from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous accesses with a single operation passes more knowledge to I/O system

Considerations on contiguous vs non contiguous

- Best performance comes from situations when the data is accessed contiguously in memory and on disk.
- Commonly, data access is contiguous in memory but noncontiguous on disk.
    - i.e: reconstruct a global data structure via parallel I/O.
- Sometimes, data access may be contiguous on disk but noncontiguous in Memory
    - i.e: writing out a list of neighbors in MD codes.
- A large impact on I/O performance would be observed if data access was noncontiguous both in memory and on disk.

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 enterely visible to all processes
- The file view of each process can be changed by means of `MPI_File_set_view`
- It is possible to do this operation as many time as you want in a program

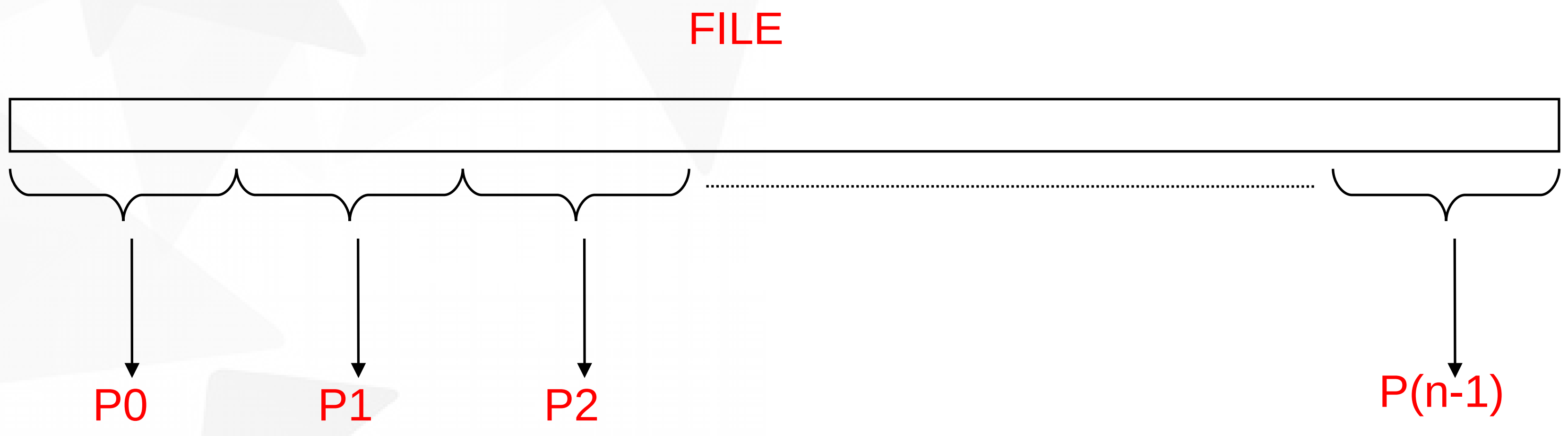
Why do we want to change File view ?

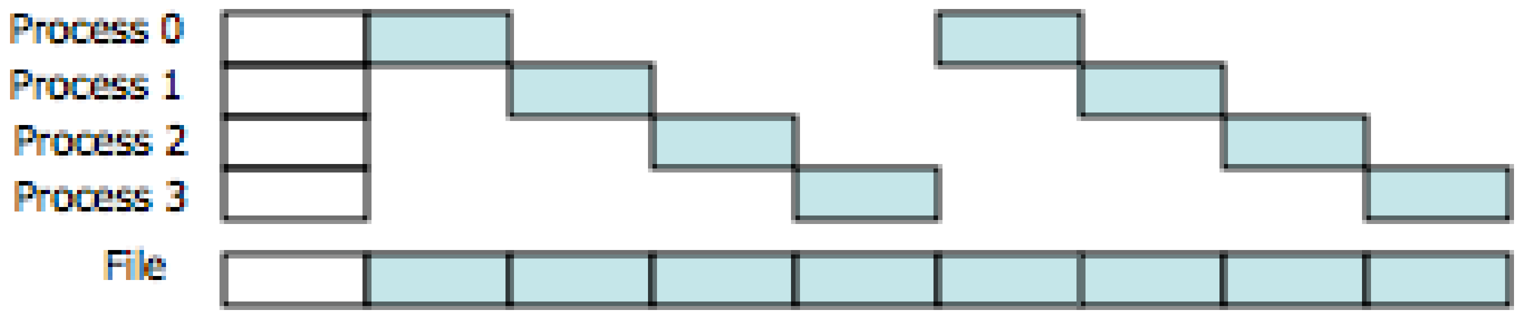
- To indicate which kind of data we are going to read.
  - By default just a bit of stream..
  - We need to use this to ensure portability

- To indicate which part of the file should be skipped… to specify non contiguous access

Using File Views

- Processes write to shared file





File Views

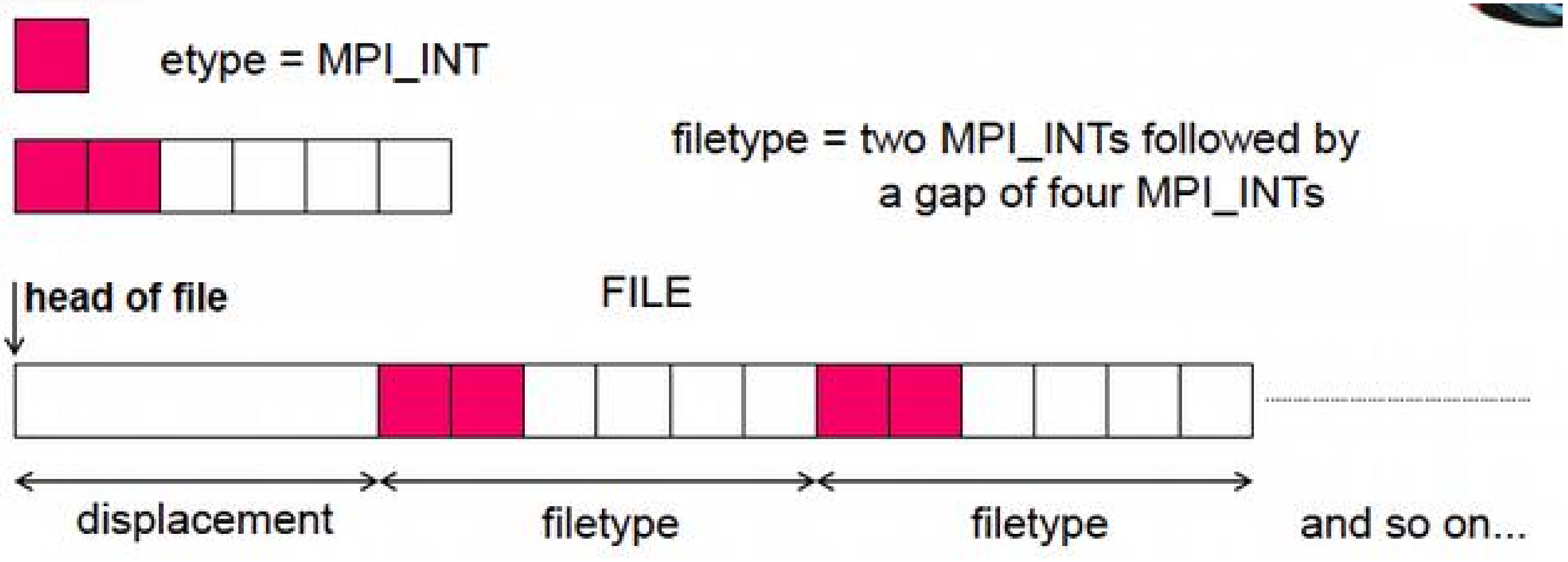
```
int MPI_File_set_view(MPI_File fh,
MPI_Offset displacement,
MPI_Datatype etype, MPI_Datatype filetype,
char *datarep, MPI_Info info)
```

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..



File View

- File view: portion of a file visible to a process
  - Processes can share a common view
  - Views can overlap or be disjoint
  - Views can be changed during runtime
- A process can have multiple instances of a file open using different file views

File View basic example (and usage)

```
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, 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

File Interoperability

- Fifth parameter of MPI_File_set_view sets the data
- representation used:
  - native: data is stored in a file exactly as it is in memory
  - internal: data representation for heterogeneous environments using the same MPI I/O implementation
  - external32: portable data representation across multiple platforms and MPI I/O libraries.
- User can also register her own data representation
  - appropriate conversion functions (MPI_Register_datarep) should be provided.

Exercises

- Play with the two examples provided (see directory)
- Write two simple MPI programs to write files using set_view function accordingly to examples/exercise 1 and 2 of the previous slides.. (optional # 1)

File view example: exercise 1 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

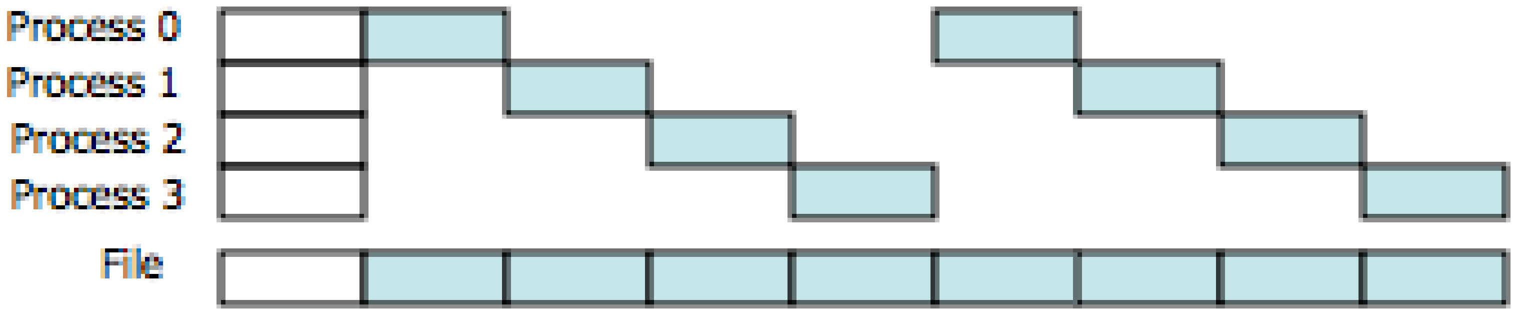
1	2	3	4	11	12	13	14	21	22	23	24	31	32	33	34

```
MPI_Type_contiguous(N, MPI_INT, &arraytype);
MPI_Type_commit(&arraytype);
```

File view Example/exercise 2

- Write a file with the following layout:

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



```
MPI_Type_vector(2,, MPI_INT, &fileblk);
MPI_Type_commit(                        &fileblk);
```

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