Introduction to Parallel Programming with MPI

Lecture #5: *Parallel I/O*

Andrea Mignone[1]

[1]Dipartimento di Fisica- Turin University, Torino (TO), Italy

Parallel I/O

- MPI-IO provides a large number of routines to read and write data from a file (I/O);
- Here we will only cover the basics.
- There are three properties which differentiate data access routines:

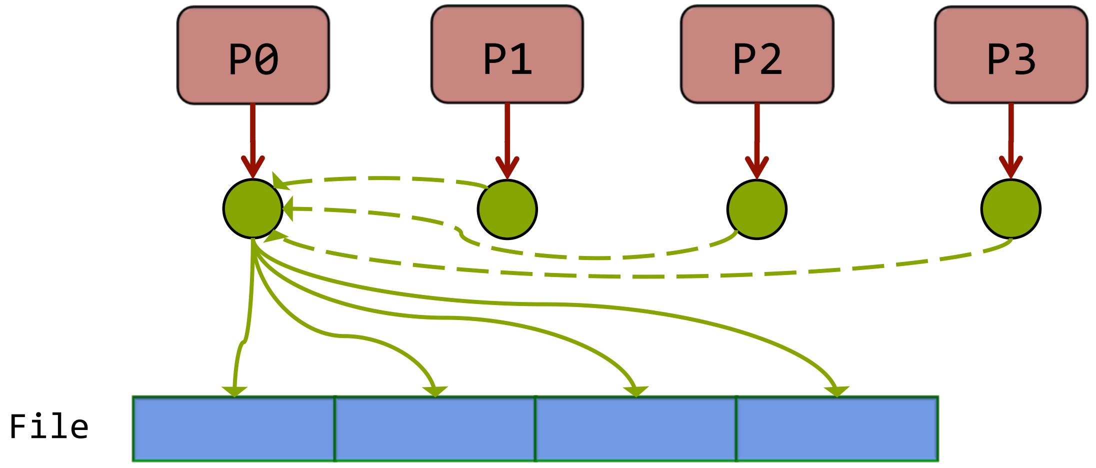
  - **Positioning:** users can either specify explicitly the offset in the file at which the data access takes place or they can use MPI file pointers;
  - **Synchronisation:** as for common communication APIs, we can use both synchronous (blocking) or asynchronous (non-blocking) function calls;
  - **Coordination:** data accesses can be done through local or collective operations.

I/O in Parallel Programs

- Input/Output (I/O) operations in parallel programs can be done in a variety of different ways.
- Solutions to managing IO in parallel applications must take into account different aspects of the application and implementation:
  - potential performance improvements (access latency to disk not neglible);
  - scaling with respect resources/system size;
  - ensure data consistency;
  - avoid communications;
  - strive for usability.
- Three are, roughly speaking, three different approaches:
  - Master-Slave (or sequential) I/O;
  - Distributed I/O on local files;
  - Fully parallel I/O.

Master-Slave Approach

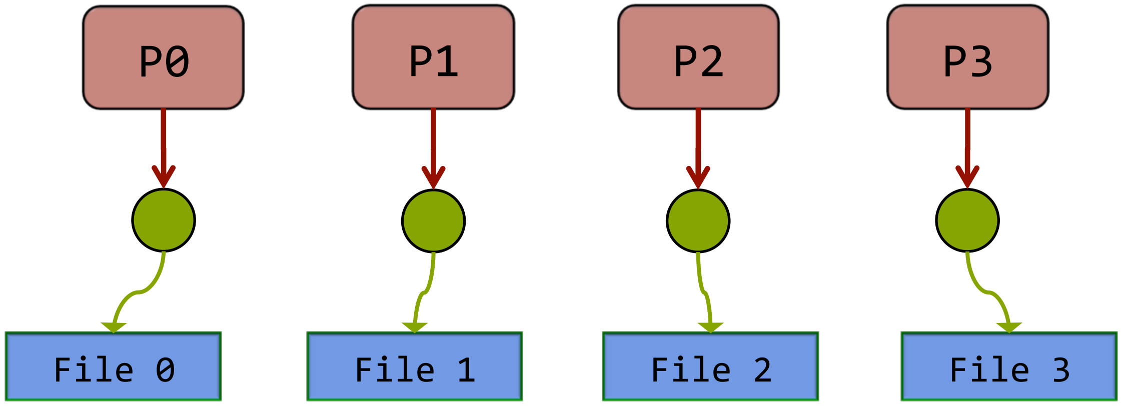
- In the sequential approach, one processor gathers the data and the does the writing:



- Pros: ensure data consistency, parallel machine may support I/O from only one process.
- Cons: lack of parallelism limit scalability, many communications involved.

Distributed I/O on Separate Files

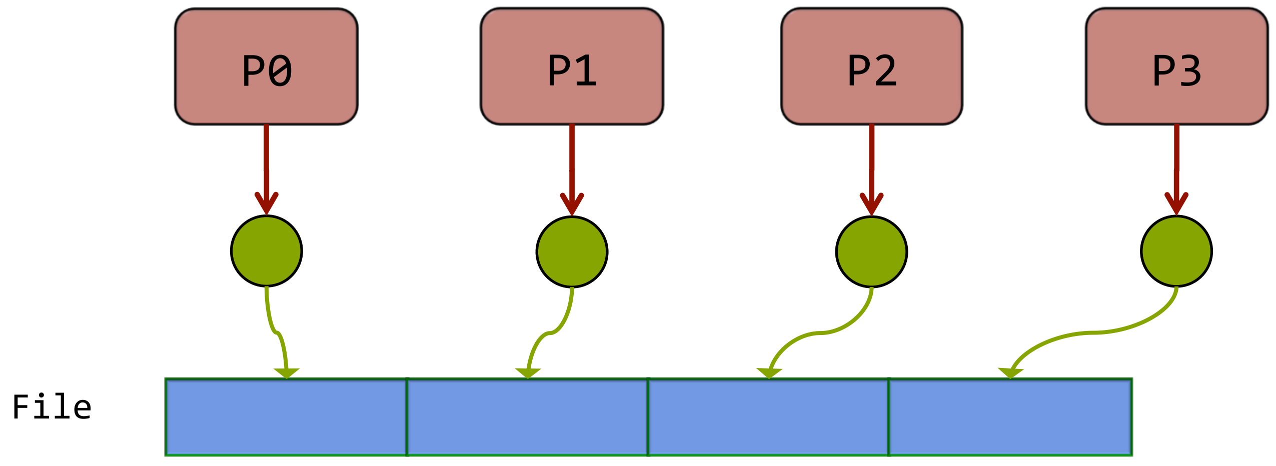
- All the processors read/writes their own files:



- Pros: scalable, and avoids communications.
- Cons: not very usable since the number of files is determined by the number of processes. End up having lots of files.

Fully Parallel I/O

- Multiple processes access data (reading / writing) from the same file



- MPI performs the output.
- Pros: High performance, avoid communication, single file provided.
- Cons: require some extra coding, depending on the data layout.

MPI I/O Functions

- MPI provides several functions for I/O.
- This table summarizes only some of the most commonly used I/O functions (see the MPI guide for a full reference):

Constructor	Purpose
`MPI_File_open()`	Opens a file on all processes in the communicator group
`MPI_File_close()`	Closes a file on all processes in the communicator group
`MPI_File_delete()`	Deletes a file

`MPI_File_write()`
`MPI_File_write_all()`
`MPI_File_write_ordered()`
`MPI_File_write_at()`
`MPI_File_write_shared()`
	Write using individual file pointer;
	Collective write using individual file pointer;
	Collective write using shared file pointer;
	Write using explicit offset.
	Write using shared file pointer

`MPI_File_read()`
`MPI_File_read_all()`
`...`
	Read using individual file pointer;
	Collective read using individual file pointer;
	…

`MPI_File_seek()`	Updates the individual file pointer
`MPI_File_set_view()`	Changes the process's view of the data in the file

Opening Files

- `MPI_File_open()` opens the file on all processes in the communicator.

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

  where

  - `comm`: communicator
  - `filename`: name of the input/output file
  - `amode`: the mode used to open the file. Modes can be combined by bitwse OR operations (see next slide).
  - `info`: used to provide additional information to the MPI-IO system. System dependent, so here we just use MPI_INFO_NULL
  - `fh`: file pointer.

- `MPI_File_open()` is a collective routine: all processes must provide the same value for `amode`, and all processes must provide filenames that reference the same file.

- Important: only Binary I/O (no ASCII text I/O)

Access Modes

- Files can be opened using a variety of modes,

Mode	Purpose
`MPI_MODE_RDONLY`	Open in read only mode
`MORE_MODE_RDWR`	Open for read/write modes
`MPI_MODE_WRONLY`	Open in write only mode
`MPI_MODE_CREATE`	Create file if it does not exist
`MPI_MODE_EXCL`	Generate error if creating an existing file
`MPI_MODE_DELETE_ON_CLOSE`	Delete file when closed (used for temporary files).
`MPI_MODE_UNIQUE_OPEN`	File will not be opened elsewhere by the system
`MPI_MODE_SEQUENTIAL`	File will not have file pointer moved manually
`MPI_MODE_APPEND`	Move file pointer to end of tile when opening.

- Modes can be combined together, e.g., `MPI_MODE_CREATE | MPI_MODE_WRONLY` will create a file and open it for write only.

Shared and Individual File Pointers

- MPI allows reading / writing of files using two different kind file pointers:
- *Shared file pointer*: file pointer is shared among all processes in the communicator used to open the file. Same pointer for all processors.
  - Only one processor can "own" shared pointer for writing or reading at a time.
  - May lead to a performance drops.
  - Functions are collective.
  - Examples: `MPI_Write_shared()`, `MPI_Write_ordered()`, `MPI_File_seek_shared()` and the corresponding `MPI_Read_...()` functions.
- *Individual file pointer*: each process has its own local file pointer for seek, read and write operations;
  - Non-collective version (e.g. `MPI_File_write()`, `MPI_File_read()`);
  - Collective version (e.g. `MPI_File_write_all()`): generally more efficiency in HPC.
- Finally, there's the concept of *file view*: maps data from multiple processors to the file representation on disk.

I/O Using Shared Pointers

- The function `MPI_Write_ordered()` provides a collective access using a shared file pointer.
- Accesses to the file will be in the order determined by the ranks of the processes within the group.
- For each process, the access location in the file is the position at which the shared file pointer would be after all processes whose ranks within the group less than that of this process had accessed their data.

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

- Shared file pointers require that the same view is used on all processes. Also, these operations are less efficient because of the need to maintain the shared pointer.
- Reading done using the corresponding function `MPI_File_read_ordered()`.

I/O Using Individual Pointers

- The same result can be obtained using a combination of `MPI_File_seek()` and `MPI_File_write()`.

- The function `MPI_File_seek()` updates the individual file pointer:

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

  where

  - `fh`: file handle, `offset`: file offset (in bytes), `whence`: update mode:
    - `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.

- The function `MPI_File_write()` does the writing at the file pointer position:

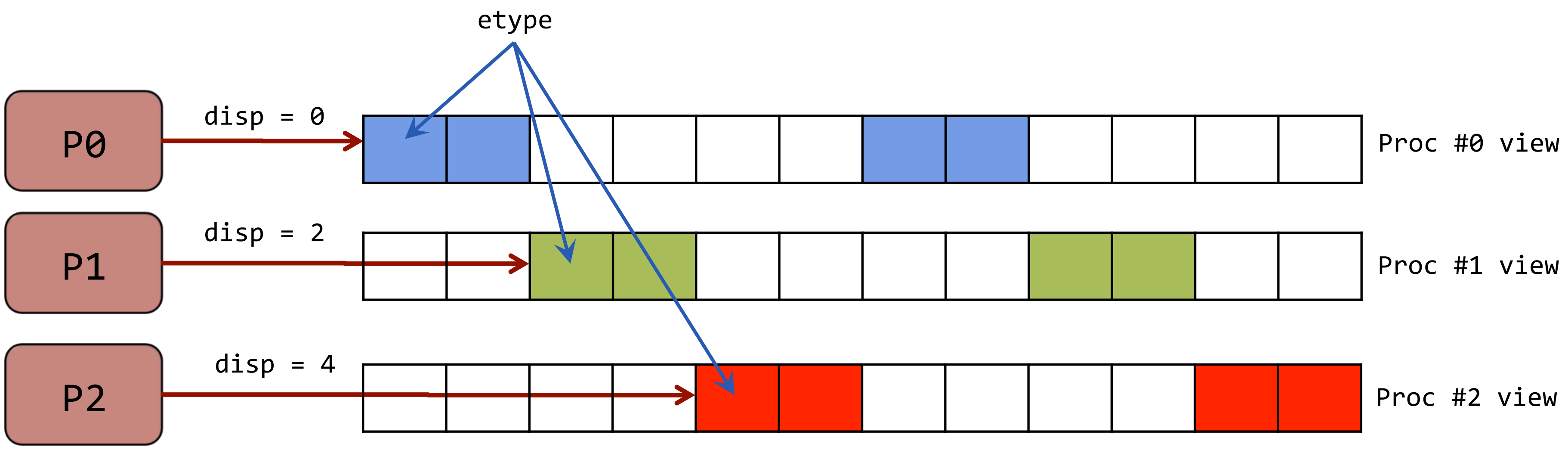
```
int MPI_File_write(MPI_File mpi_fh, void *buf, int count,
                   MPI_Datatype datatype, MPI_Status *status);
```

- Note that `MPI_File_write()` is non-collective (the I/O library has to process individual requests). The *collective version* (more efficient for large datasets) is

```
int MPI_File_write_all();
```

I/O Using File Views

- A file view defines which portion of a file is "visible" to a process as well as the type of the data in the file (byte, integer, float, ...).
- By default, the file is treated as consisting of bytes and process can access (read or write) any byte in the file.
- A view consists of:
  - `displacement`: number of bytes to skip from beginning of file;
  - `etype`: Basic unit of data access
  - `filetype`: portion of file visible to process



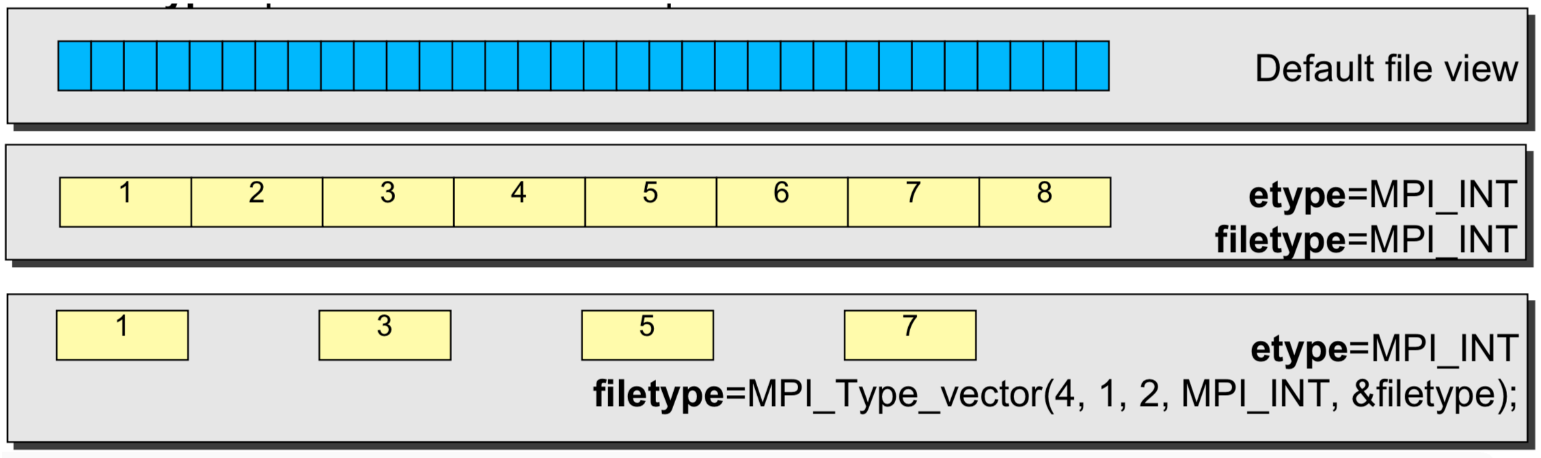
Setting the File View: `MPI_File_set_view()`

- The function setting the view is

```
int MPI_File_set_view(MPI_File mpi_fh, MPI_Offset disp, MPI_Datatype etype,
                      MPI_Datatype filetype, char *datarep, MPI_Info info);
```

where

- `fh`: file handle (handle)
- `disp`: displacement from the start of the file, in bytes (integer)
- `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)
- `filetype`: datatype describing each processes view of the file. (handle)
- `datarep`: data representation (string)
- `info`: info object (handle)

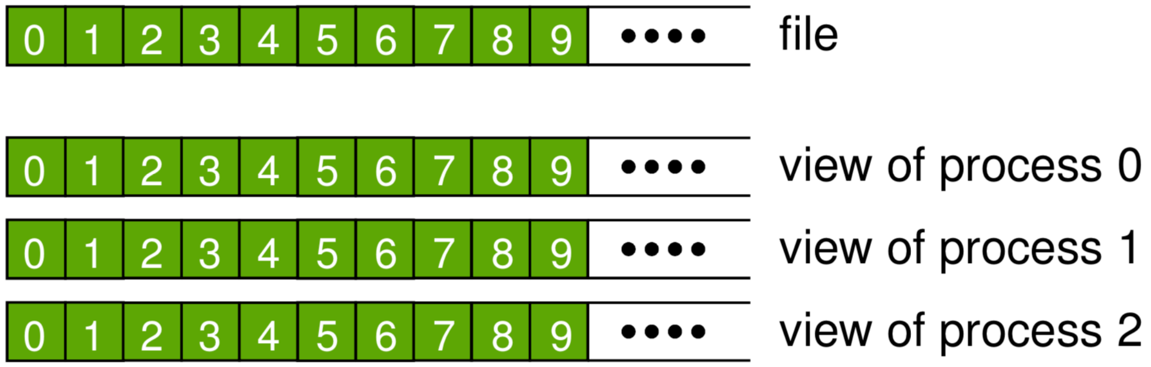


File View: Data representation

- The data representation (`datarep`) defines the layout and data access modes (byte order, type sizes, etc.):
  - **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 using the correct MPI datatypes (not using MPI_BYTE)

Default File View

- A default file view for each participating process is defined implicitly with `MPI_File_open()`:
- This view has no displacement, the file has no specific structure and all processes have access to the complete file. In other words:
  - `disp = 0;`
  - `etype = MPI_BYTE`
  - `filetype = MPI BYTE`



Example #1: writing contiguous array

- Write a program (`write_1Darr.c`) that writes a double-precision buffer with `NELEM` elements all set equal to the process rank.

- For `NELEM = 3` and 4 processors, the output (binary) file should consist of

0	0	0	1	1	1	2	2	2	3	3	3

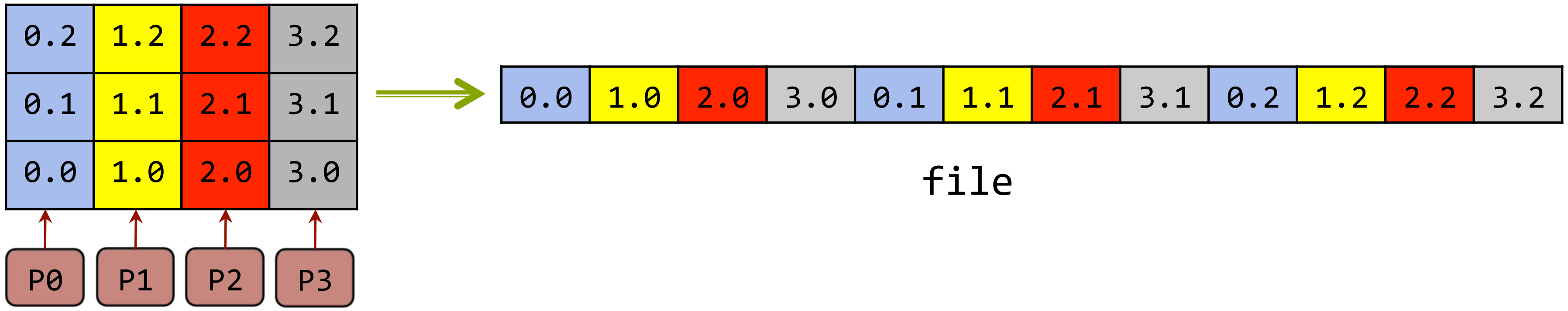
- Explore 3 different strategies:
  - Shared file pointer (`MPI_File_write_ordered()`);
  - Individual file pointer (`MPI_File_seek() + MPI_File_write()`);
  - Using the file view (`MPI_File_set_view + MPI_File_write()`);

- To check that the file has been written correctly you can use the "**od**" command:

```
> od -Fv <file.bin>
0000000      0.000000000000000e+00    0.000000000000000e+00
0000020      0.000000000000000e+00    1.000000000000000e+00
0000040      1.000000000000000e+00    1.000000000000000e+00
0000060      2.000000000000000e+00    2.000000000000000e+00
0000100      2.000000000000000e+00    3.000000000000000e+00
0000120      3.000000000000000e+00    3.000000000000000e+00
0000140
```

Non-Contiguous Data

- File views are particularly useful when data has to be written non-contiguously to disk.
- Consider, for instance, the following 2D array distributed column-wise:



- We create a vector type with `count=3, blocklen=1, stride=4` and use it to set the file view:

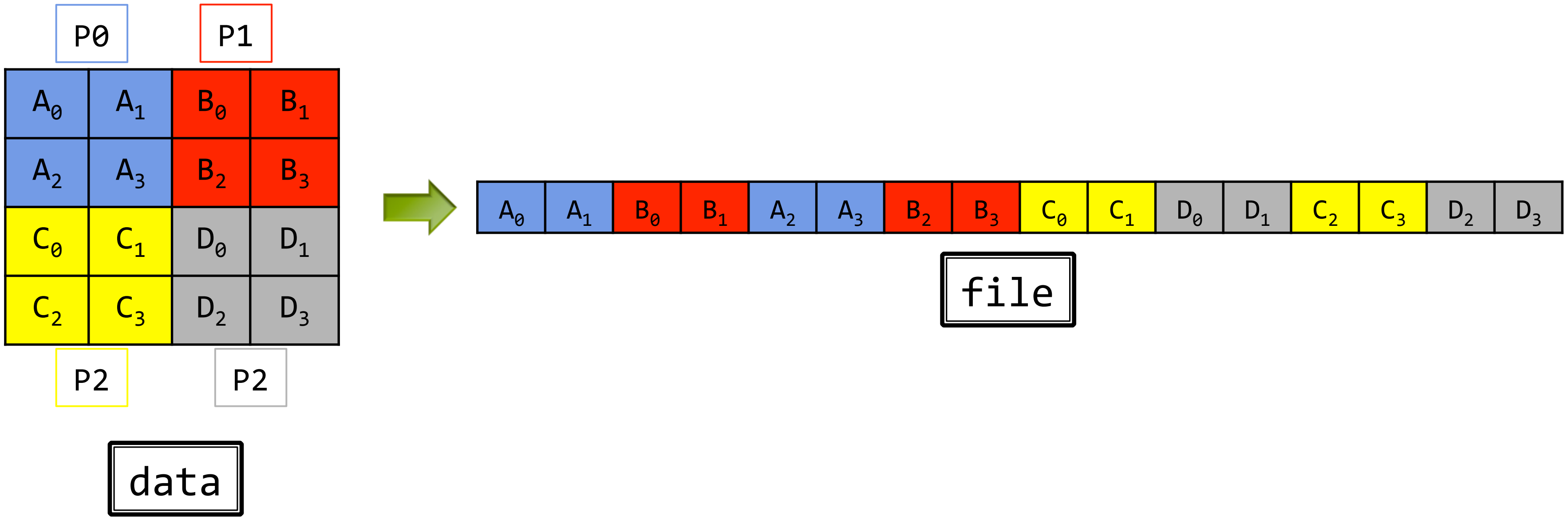
```
for (i = 0; i < NELEM; i++) buf[i] = rank + 0.1*i;    // Fill buffer

MPI_Datatype vec_type;
MPI_Type_vector(NELEM, 1, size, MPI_DOUBLE, &vec_type); // Create vector type
MPI_Type_commit(&vec_type);

disp = rank*sizeof(double);                               // Compute offset (in bytes)
MPI_File_set_view(fh, disp, MPI_DOUBLE, vec_type, "native", MPI_INFO_NULL);   // Set view
MPI_File_write(fh, buf, NELEM, MPI_DOUBLE, MPI_STATUS_IGNORE);                // Write
MPI_Type_free(&vec_type);
```

Multidimensional Arrays

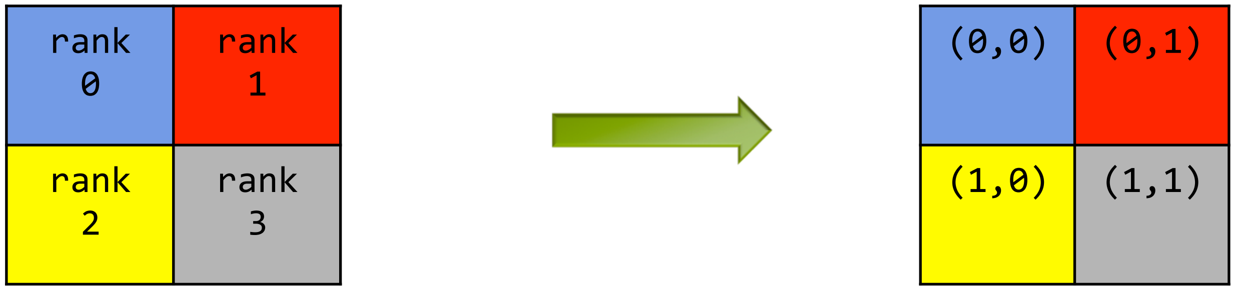
- I/O of multi-dimensional arrays should be handled in a way which is independent of the decomposition.
- Datafiles should be written according to a usual serial order: row major order (C) or column major order (Fortran).



- The **subarray** datatype may easily handle these situations.
- However, a Cartesian decomposition is needed for this situation.

Cartesian Decomposition

- A Cartesian decomposition is a parallelization method whereby different portions of the domain are assigned to individual processes;
- In other words, it maps a rank to a coordinate:



- To create a new communicator with the chosen decomposition we use

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
                    const int periods[], int reorder, MPI_Comm * comm_cart)
```

where

- `comm_old`: input communicator (handle)
- `ndims`: number of dimensions of Cartesian grid (integer)
- `dims`: integer array of size `ndims` specifying the number of procs in each dimension;
- `periods`: logical array of size `ndims` specifying periodicity (true) or not (false) in each dimension;
- `reorder`: ranking may be reordered (true) or not (false)
- `comm_cart`: communicator with new Cartesian topology (handle)

A worked example: 2D Domain Decomposition with Distributed I/O

- We now decompose the domain in 2x2 processors and create the corresponding Cartesian topology:

```
MPI_Comm MPI_COMM_CART;  // Declare new Cartesian communicator

int periods[2] = {0,0};   // No periodicity
int nprocs[0]  = {2,2};   // Number of processes in the x- and y-directions
// ! MAKE SURE nprocs[0]*nprocs[1] == size

MPI_Cart_create(MPI_COMM_WORLD, NDIM, nprocs, periods, 0, &MPI_COMM_CART);  // Cart decomposition
MPI_Cart_get(MPI_COMM_CART, NDIM, nprocs, periods, coords); // Obtain coordinates from rank
```

- If the total computation domain has dimensions Nx_g and Ny_g, each process owns a sub-portion of $nx=Nx_g/nprocs[0]$ and $ny=Ny_g/nprocs[1]$ points.

```
gsizes[0] = NX_GLOB;        // Global domain size in the x-direction ( = Nxg)
gsizes[1] = NY_GLOB;        // Global domain size in the y-direction ( = Nyg)

lsizes[0] = nx = NX_GLOB/nprocs[0];  // Local domain size in the x-direction ( = nx)
lsizes[1] = ny = NY_GLOB/nprocs[1];  // Local domain size in the y-direction ( = ny)

/* -- Allocate memory and fill 2D array on local domain -- */

A = Allocate_2DdblArray(ny,nx);   // Allocate memory on local grid  (i = fastest running index)
for (j = 0; j < ny; j++) for (i = 0; i < nx; i++) A[j][i] = rank; // Fill array
```

A worked example: 2D Domain Decomposition with Distributed I/O

- We can now create the desired subarray type from the previous decomposition:

```
MPI_Datatype subarr_type;

start[0] = coords[0]*lsizes[0];   // Offsets of local array into global array
start[1] = coords[1]*lsizes[1];   // Offsets of local array into global array

MPI_Type_create_subarray (NDIM, gsizes, lsizes, start, MPI_ORDER_FORTRAN,
                          MPI_DOUBLE, &subarr_type);
MPI_Type_commit (&subarr_type);
```

- We use `MPI_ORDER_FORTRAN` because the array is column-oriented.
- In the example we use $Nx_g = 16$ and $Ny_g = 8$
- Output can then be done using `MPI_File_set_view()`:

```
MPI_File fh;
MPI_Status status;

MPI_File_open(MPI_COMM_CART, fname,
              MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_DOUBLE, subarr_type, "native", MPI_INFO_NULL);
MPI_File_write_all(fh, A[0], nx*ny, MPI_DOUBLE, &status);
MPI_File_close(&fh);
```

Visualizing Data with Gnuplot

- Binary data can be visualized using, e.g., gnuplot.
- Commands may be entered at the gnuplot prompt,

```
gnuplot> reset
gnuplot> set autoscale xfixmin
gnuplot> set autoscale xfixmax
gnuplot> set autoscale yfixmin
gnuplot> set autoscale yfixmax

gnuplot> set pm3d map
gnuplot> set palette defined
gnuplot> splot "arr2D.bin" bin array=16x8 format='%lf' with image
```

- Alternatively, you may create a new file, e.g. “`arr2d.gp`”, with the instruction and then load it at the gnuplot prompt:

```
gnuplot> load “arr2d.gp”
```