Advanced Message-Passing Programming

Basic MPI-IO calls

Reusing this material

Overview

- Lecture will cover
  - MPI-IO model
  - basic file handling routines
  - setting the file view
  - achieving performance

Comparing MPI-IO and Controller IO

- Controller IO uses a single process for read and write
- This requires a buffer to hold entire file on controller
  - not scalable to many processes due to memory limits
- MPI-IO model
  - each process describes what subsections of the file it owns
  - this is done using standard MPI datatypes
  - these are passed into the MPI-IO routines
  - data is automatically read and transferred directly to local memory
  - there is no single large buffer and no explicit controller process

MPI-IO Approach

- Four stages
  - open file
  - set file view
  - read or write data
  - close file
- All the complexity is hidden in setting the file view
  - this is where the derived datatypes appear
- Write is probably more important in practice than read
  - but exercises concentrate on read
  - makes for an easier progression from serial to parallel IO examples

Opening a File

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

MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERR)
CHARACTER*(*) FILENAME
INTEGER COMM, AMODE, INFO, FH, IERR
```

- Attaches a file to the File Handle
  - use this handle in all future IO calls
  - analogous to C file pointer or Fortran unit number
- Routine is collective across the communicator
  - must be called by all processes in that communicator
- Access mode specified by amode
  - common values are: **MPI_MODE_CREATE**, **MPI_MODE_RDONLY**, **MPI_MODE_WRONLY**, **MPI_MODE_RDWR**

Examples

```
MPI_File fh;
int amode = MPI_MODE_RDONLY;
MPI_File_open(MPI_COMM_WORLD, "data.in", amode,
              MPI_INFO_NULL, &fh);
```

```
integer fh
integer amode = MPI_MODE_RDONLY
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'data.in', amode,
                   MPI_INFO_NULL, fh, ierr)
```

- Must specify create as well as write for new files

```
int     amode = MPI_MODE_CREATE | MPI_MODE_WRONLY;
integer amode = MPI_MODE_CREATE + MPI_MODE_WRONLY
```

  - will return to the **info** argument later

Closing a File

```
MPI_File_close(MPI_File *fh)

MPI_FILE_CLOSE(FH, IERR)
INTEGER FH, IERR
```

- Routine is collective across the communicator
  - must be called by all processes in that communicator

Reading Data

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

MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERR)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERR
```

- Reads **`count`** objects of type **`datatype`** from the file on each process
  - this is collective across the communicator associated with **`fh`**
  - similar in operation to C **`fread`** or Fortran **`read`**
- No offsets into the file are specified in the read
  - but processes do not all read the same data!
  - actual positions of read depends on the process's own file view
- Similar syntax for write

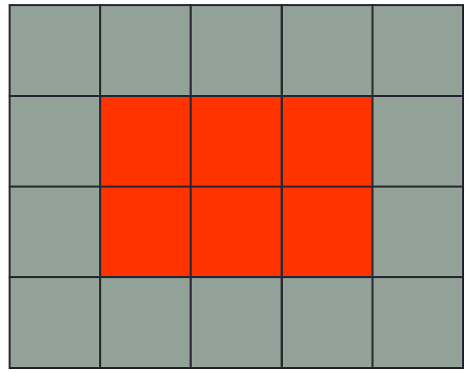
Setting the File View

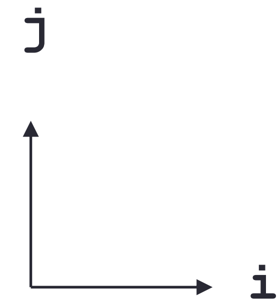
```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
                      MPI_Datatype etype, MPI_Datatype filetype,
                      char *datarep, MPI_Info info);
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO,IERR)
INTEGER FH, ETYPE, FILETYPE, INFO, IERR
CHARACTER*(*) DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

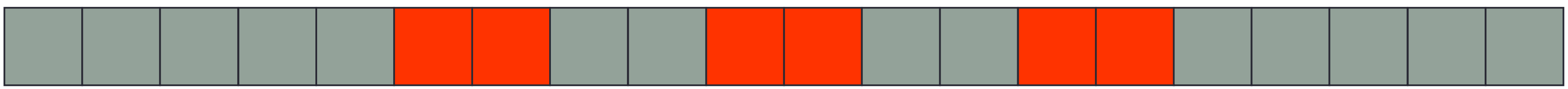
- **`disp`** specifies the starting point in the file *in bytes*
- **`etype`** specifies the elementary datatype which is the building block of the file
- **`filetype`** specifies which subsections of the global file each process accesses
- **`datarep`** specifies the format of the data in the file
- **`info`** contains hints and system-specific information – see later

3x2 Subsections of 5x5 Array in Memory

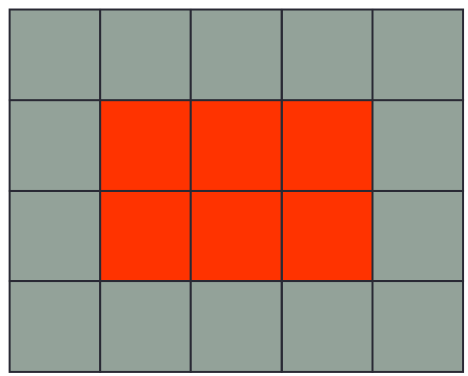
C: `x[5][4]`



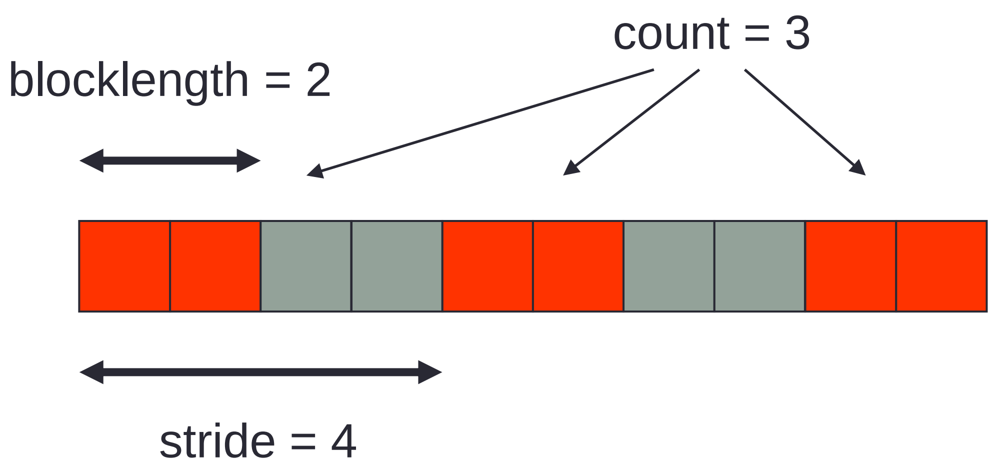


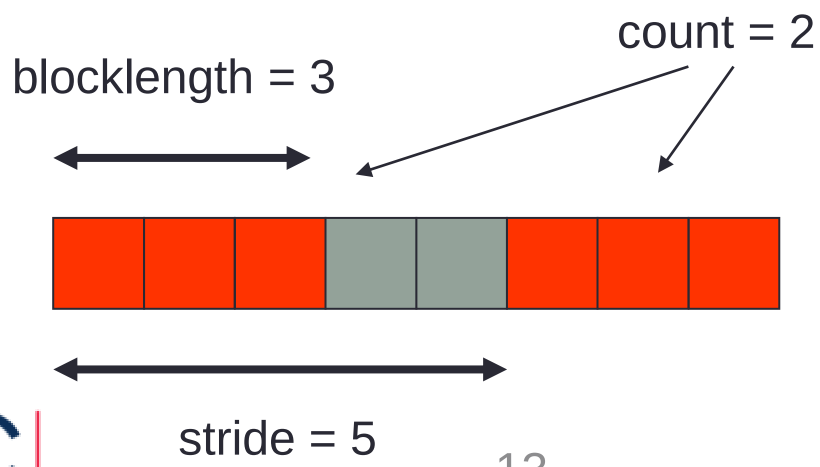


F: `x(5,4)`



Equivalent Vector Datatypes





Subarray Datatype

- A single call that defines multi-dimensional subsections
  - much easier than vector types for 3D arrays

```
MPI_Type_create_subarray(int ndims, int array_of_sizes[],
 int array_of_subsizes[], int array_of_starts[],
 int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES,
 ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER,
 OLDTYPE, NEWTYPE, IERR)
```

```
INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
  ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERR
```

C Definition

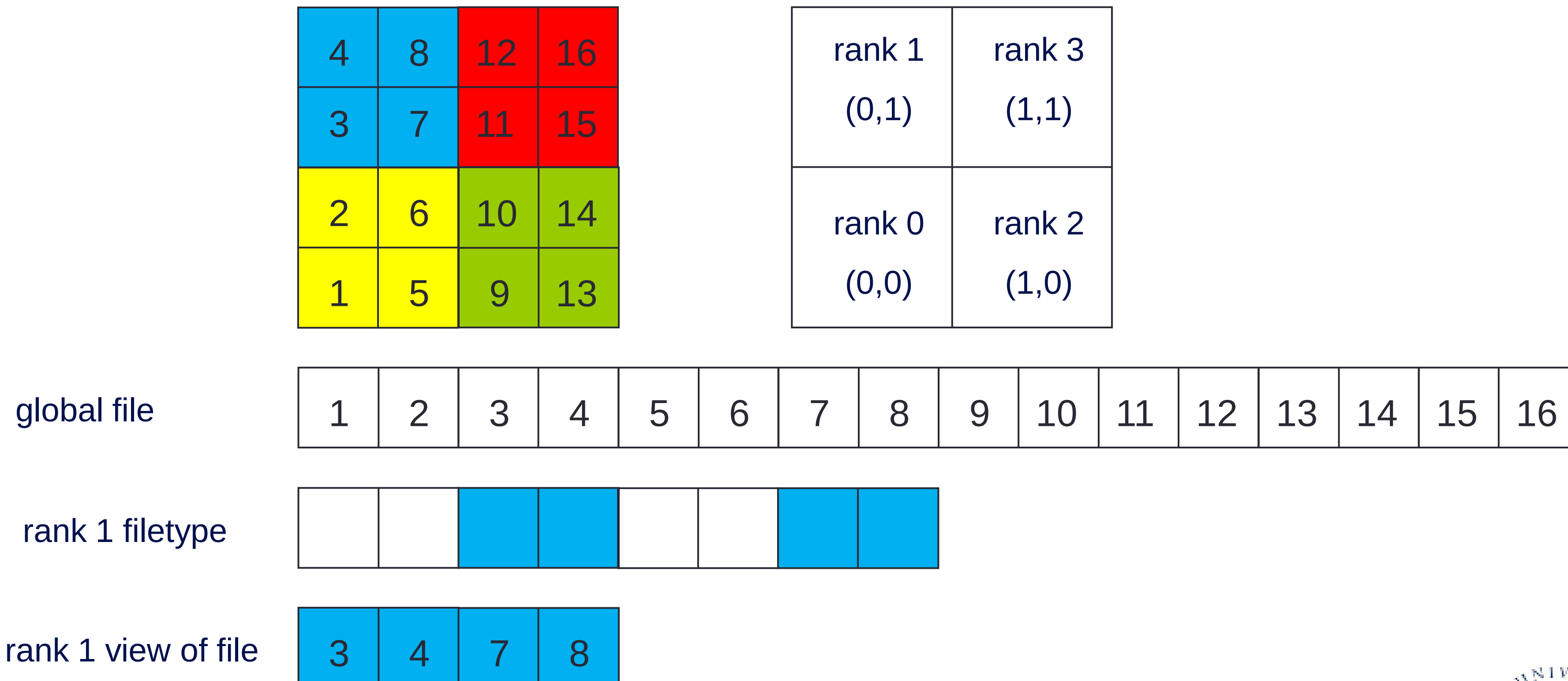
```
#define NDIMS 2
MPI_Datatype subarray3x2;
int array_of_sizes[NDIMS], array_of_subsizes[NDIMS],
    arrays_of_starts[NDIMS];

array_of_sizes[0]    = 5; array_of_sizes[1]    = 4;
array_of_subsizes[0] = 3; array_of_subsizes[1] = 2;
array_of_starts[0]   = 2; array_of_starts[1]   = 1;

order = MPI_ORDER_C;

MPI_Type_create_subarray(NDIMS, array_of_sizes,
 array_of_subsizes, array_of_starts, order,
 MPI_FLOAT, &subarray3x2);
MPI_TYPE_COMMIT(&subarray3x2);
```

Fortran Definition

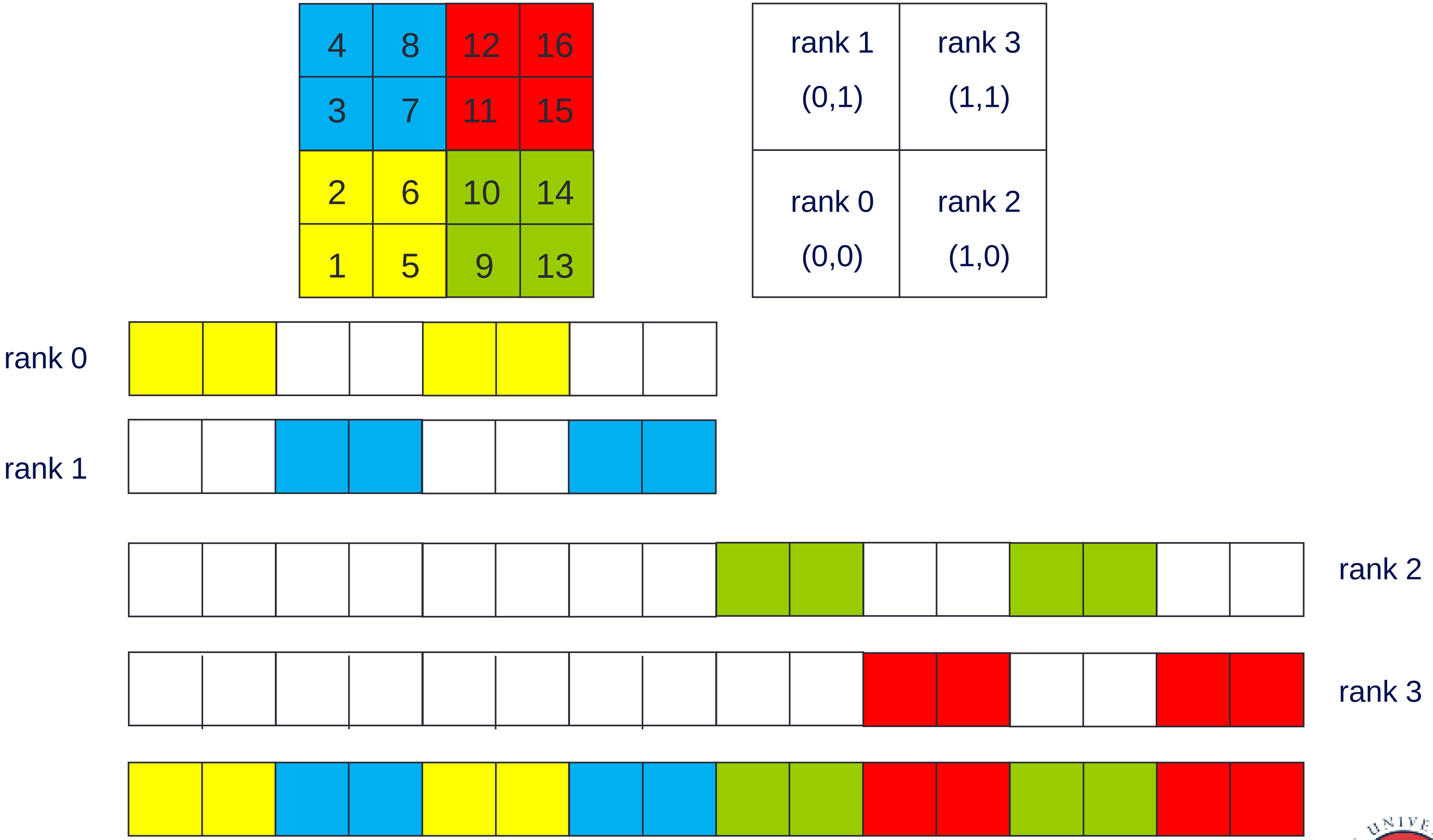
```
integer, parameter :: ndims = 2
integer subarray3x2
integer, dimension(ndims) :: array_of_sizes,
 array_of_subsizes,
                                arrays_of_starts
! Indices start at 0 as in C !

array_of_sizes(1)    = 5; array_of_sizes(2)    = 4
array_of_subsizes(1) = 3; array_of_subsizes(2) = 2
array_of_starts(1)   = 2; array_of_starts(2)   = 1

order = MPI_ORDER_FORTRAN
call MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes,
 array_of_subsizes, array_of_starts, order,
 MPI_REAL, subarray3x2, ierr)
```

File Views

- Once set, the process only sees the data in the view
  - data starts at different positions in the file depending on the displacement and/or leading gaps in fixed datatype
  - can then do linear reads – holes in datatype are skipped over



Filetypes Should Tile the File

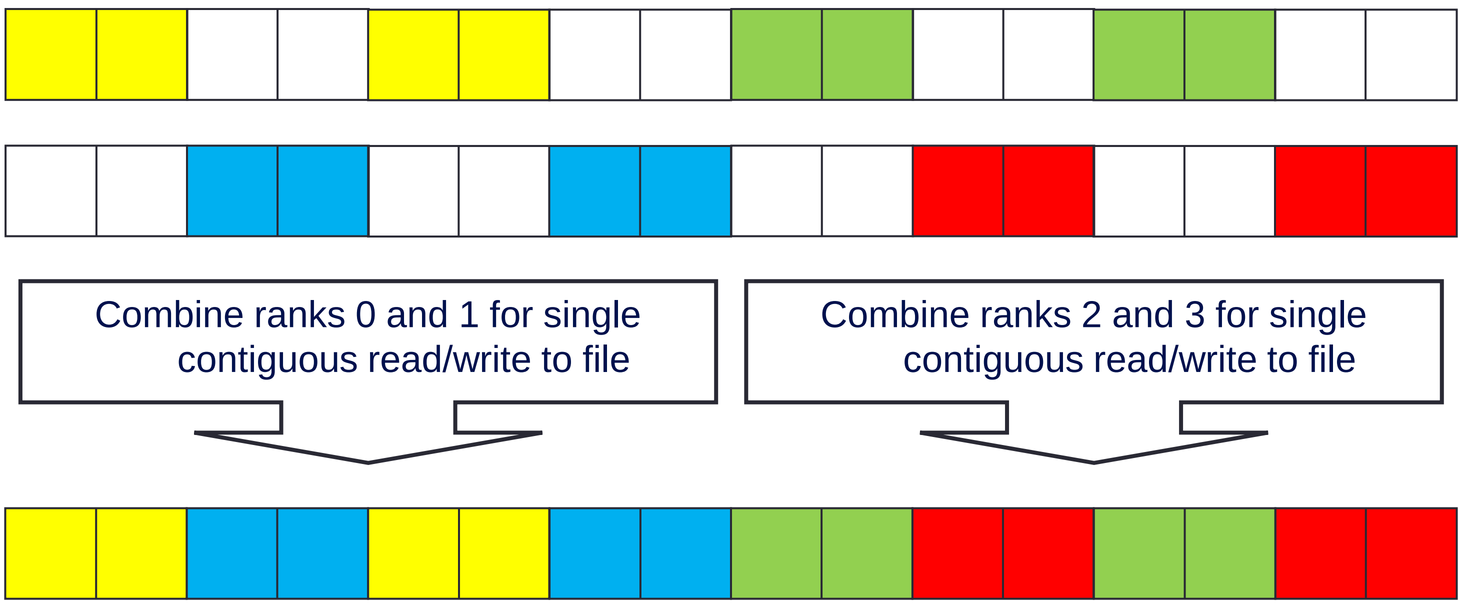


Data Representation

- **`datarep`** is a string that can be
  - **`"native"`**
  - **`"internal"`**
  - **`"external32"`**
- Fastest is **`"native"`**
  - raw bytes are written to file exactly as in memory
- Most portable is **`"external32"`**
  - should be readable by MPI-IO on any platform
- Middle ground is **`"internal"`**
  - portability depends on the implementation
- I would recommend **`"native"`**
  - convert file format by hand as and when necessary

Collective IO

- For read and write, "`_all`" means operation is collective
  - all processes attached to the file are taking part
- Other IO routines exist which are individual (delete "_all")
  - functionality is the same but performance will be slower
  - collective routines can aggregate reads/writes for better performance



`INFO` Objects and Performance

- Used to pass optimisation hints to MPI-IO
  - implementations can define any number of allowed values
  - these are portable in as much as they can be ignored!
  - can use the default value **`info = MPI_INFO_NULL`**

- Info objects can be created, set and freed (see manual for details)
  - **`MPI_Info_create`**
  - **`MPI_Info_set`**
  - **`MPI_Info_free`**
- Using appropriate values may be key to performance
  - e.g. setting buffer sizes, blocking factors, number of IO nodes, ...
  - but is dependent on the system and the MPI implementation
  - need to consult the MPI manual for your machine
  - on ARCHER2, easier to tune Lustre file system than use MPI-IO hints

Summary

- MPI-IO calls deceptively simple

- User must define appropriate filetypes so file view is correct on each process
  - this is the difficult part!

- Use collective calls whenever you can
  - enables IO library to merge reads and writes
  - enables a smaller number of larger IO operations from/to disk