Advanced Parallel Programming

Basic MPI-IO Calls

ARCHER Training Courses

Sponsors

Reusing this material

Overview

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

Comparing MPI-IO and Master IO

- Have so far defined datatypes appropriate for each process
  - and used them to do multiple sends from a master
- This requires a buffer to hold entire file on master
  - not scalable to many processes due to memory limits
- MPI-IO model
  - each process defines the datatype for its section of the file
  - 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 master 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

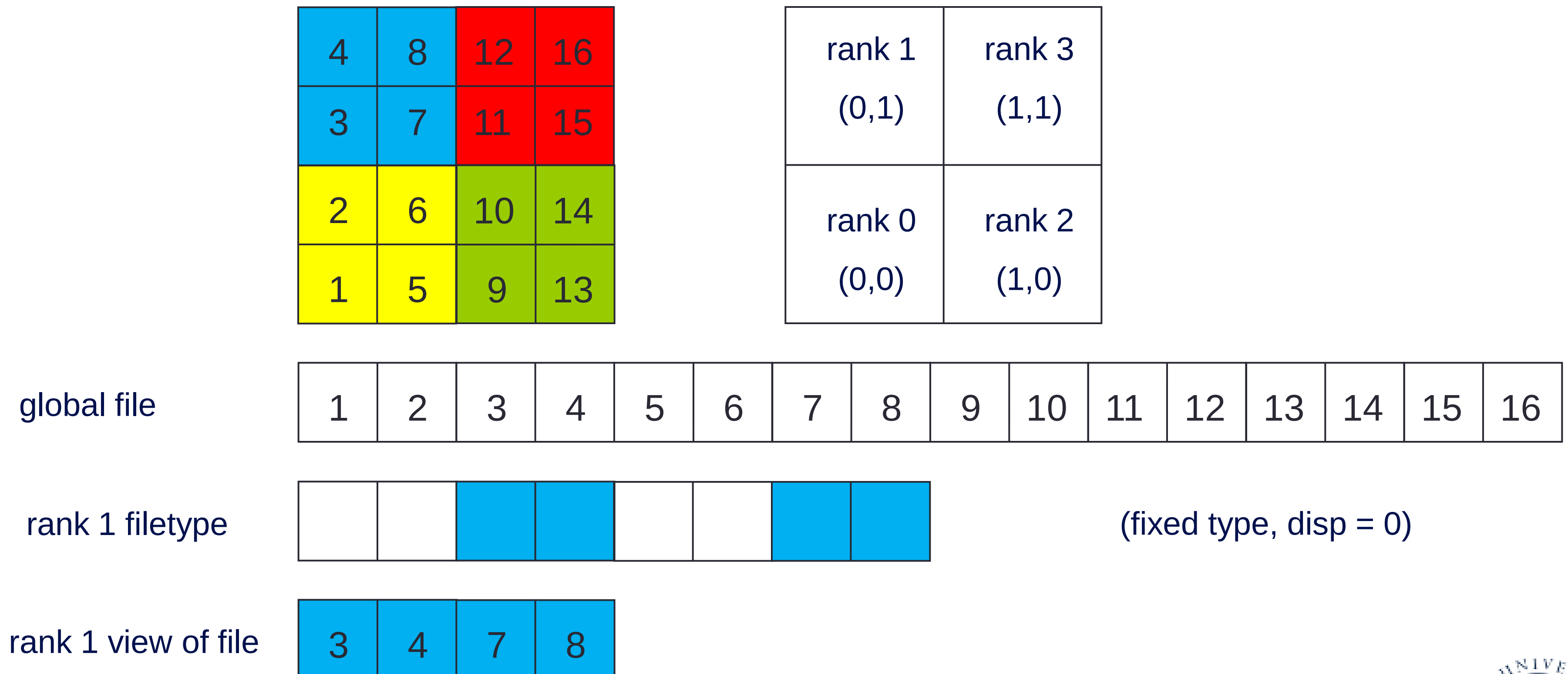
```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
                      MPI_Datatype etype, MPI_Datatype filetype,
                      char *datarep, MPI_Info info);

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
```

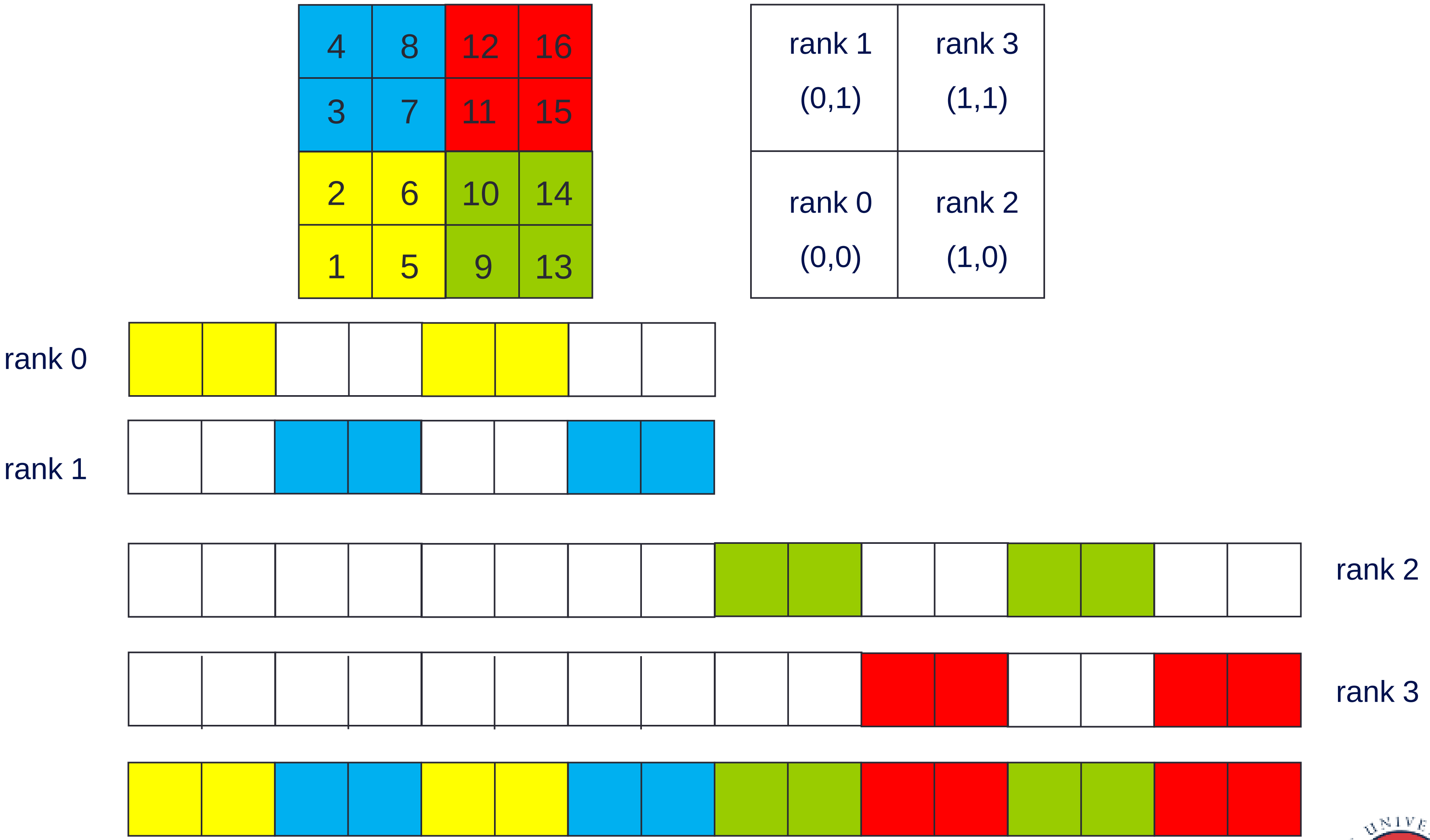
- **`disp`** specifies the starting point in the file *in bytes*
- **`etype`** specifies the elementary datatype which is the building block of the file
- **`filetype`** specifies which subsections of the global file each process accesses
- **`datarep`** specifies the format of the data in the file
-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

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

Choice of Parameters (1)

- Many different combinations are possible
  - choices of displacements, filetypes, etypes, datatypes, ...

- Simplest approach is to set **`disp`** = 0 everywhere
  - then specify offsets into files using fixed datatypes when setting view
    - non-zero **`disp`** could be useful for skipping global header (eg metadata)
  - **`disp`** must be of the correct type in Fortran (NOT a default integer)
  - **CANNOT** specify ‘0’ for the displacement: need to use a variable

```
INTEGER(KIND=MPI_OFFSET_KIND) DISP = 0
CALL MPI_FILE_SET_VIEW(FH, DISP, ...)
```

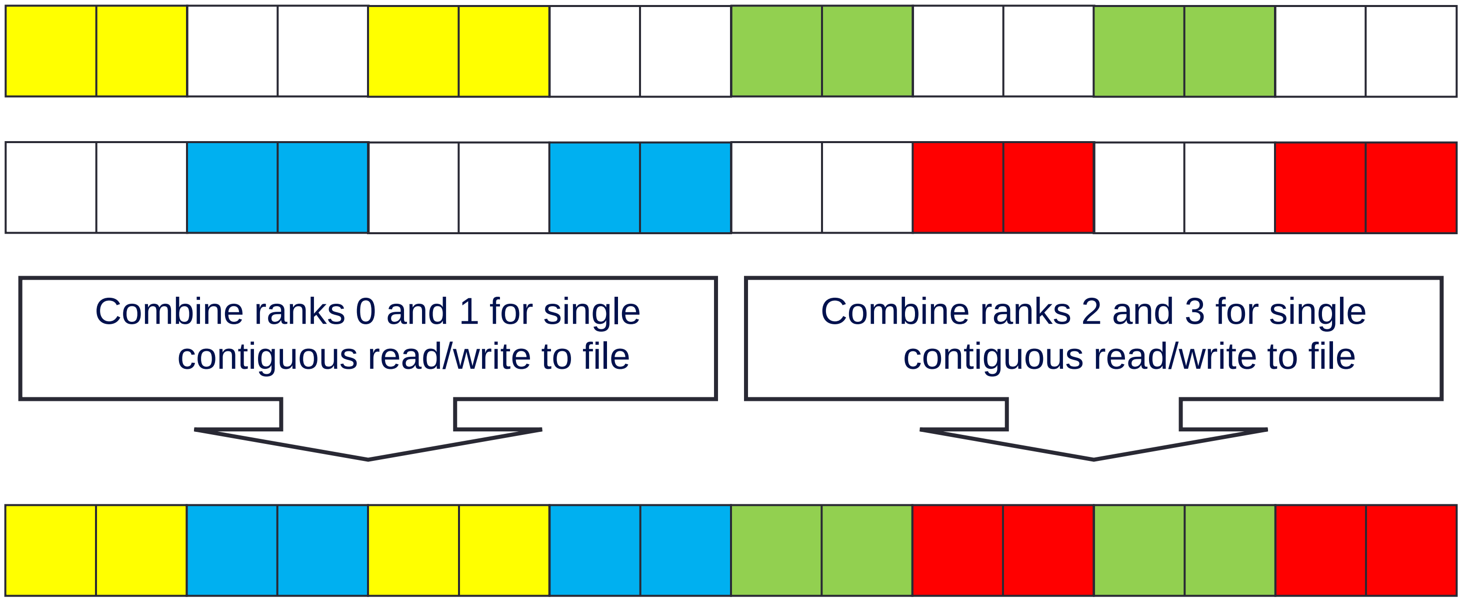
- I would recommend setting the view with fixed datatypes
  - and zero displacements

Choice of Parameters (2)

- Can also use floating datatypes in the view
  - each process then specifies a different, non-zero value of disp
- Problems
  - disp is specified in bytes so need to know the size of the **`etype`**
  - files are linear 1D arrays
    - need to do a calculation for displacement of element of 2D array
    - something like **`i*NY + j`** (in C) or **`j*NX + i`** (in Fortran)
    - then multiply by the number of bytes in a float or REAL
- Using vector types + displacements is one of the exercises
- **`etype`** normally something like **`MPI_REAL`** or **`MPI_FLOAT`**
  - **`datatype`** in read/write calls is usually the same as the **`etype`**
  - however, can play some useful tricks (see extra exercises re halos)

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 ARCHER, 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