ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

MPI-IO

`/project/csstaff/IO_Course`

MPI-IO – the Basics

- MPI-IO provides a low-level interface to carrying out parallel I/O
- MPI-IO defines how to access a file system to store data
  - There is no metadata stored about the file
    - There are no tools to analyse what kind of data is stored in the file
- The MPI-IO API has a large number of routines
  - The MPI 2.2 standard has 64 pages for the I/O section
    - Much of this is explanation of the routines
  - There are over 50 routines in the I/O section of the standard
    - This does not include some other routines that are mainly included in the standard to provide support for MPI-IO

Compiling and Running

- As MPI-IO is part of MPI, you simply compile and link as you would any normal MPI program
  - On the Cray:-
    - `ftn -O2 mycode.f90 -o myprog`
    - `cc -O2 mycode.c -o myprog`
- As MPI-IO is part of the MPI standard, man pages for all routines are available on the Cray systems
- To run the examples grab a node from the batch system and launch the job with aprun

```
$ cp ./myprog $SCRATCH
$ cd $SCRATCH
$ salloc -N 1 --time=00:20:00
salloc: Granted job allocation XXXX
$ aprun -n 24 ./myprog
```

Opening a File –the API

```
int 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
```

- MPI_File_open is a collective call to open a file
  - The collective is defined on the communicator
    - Typically you might use MPI_COMM_WORLD as a communicator
    - To open a file just on one process use MPI_COMM_SELF as the communicator
- All processes must call MPI_File_open with the same `filename` and `amode` parameters
  - Actually processes may use different names for the `filename` as long as it actually references the same file
- If no specifics are needed for info then MPI_INFO_NULL can be used
- The file handle `fh` is then used in all subsequent file operations

File Access Modes

- A number of access modes are supported for MPI files
- The amode argument to MPI_File_open defines the access mode for the file
- Multiple access modes can be combined by
  - Using addition or the IOR function in Fortran
    - i.e. MPI_MODE_CREATE+MPI_MODE_EXCL+MPI_MODE_WRONLY
  - Using the or (|) operator in C
    - i.e. MPI_MODE_CREATE|MPI_MODE_EXCL|MPI_MODE_WRONLY

```
MPI_MODE_RDONLY              open for read only
MPI_MODE_RDWR                open for reading and writing
MPI_MODE_WRONLY              open for write only
MPI_MODE_CREATE              create the file if it does not exist
MPI_MODE_EXCL                generate an error if creating a file that already exists
MPI_MODE_DELETE_ON_CLOSE     delete the file on MPI_File_close is called
MPI_MODE_APPEND              set initial position of all file pointers to end of file

MPI_MODE_UNIQUE_OPEN
MPI_MODE_SEQUENTIAL
```

Closing a file – the API

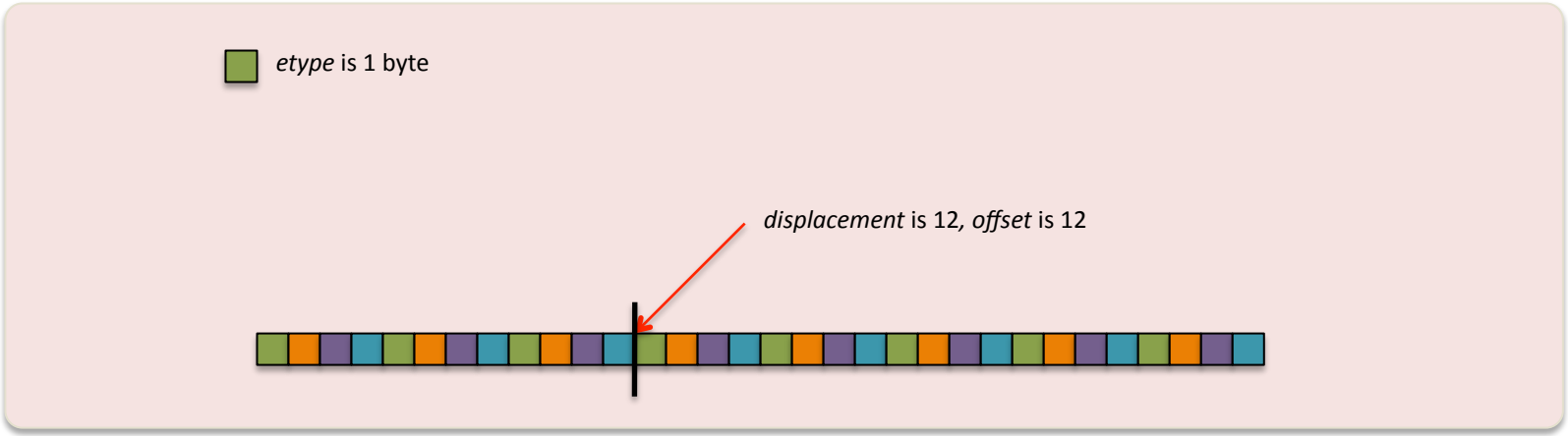
```
int MPI_File_close(MPI_File *fh)
```

```
MPI_File_close(fh, ierr)

Integer :: fh, ierr
```

- MPI_File_close is a collective call to close a file
  - The collective is defined on the same communicator use to open the file
- All outstanding operations are synced on the file before it is closed
- If the file was opened with the access mode MPI_MODE_DELETE_ON_CLOSE then the file will be deleted before the call returns

Displacements, Elementary Datatypes and Offsets

- The **displacement** of a position within a file is the number of bytes from the beginning of the file
- An **etype** or elementary datatype is an MPI datatype (predefined or a derived datatype)
  - The *etype* is used to set file views and for file access operations (reads and writes)
- An **offset** is a position in the file given in terms of multiples of *etypes*
  - Actually it is a multiple of *etypes* from the beginning of the current *view*
    - On file open the *view* begins at the start of the file
    - On file open the *etype* is a byte



Simple Independent Writing in a File – the API

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

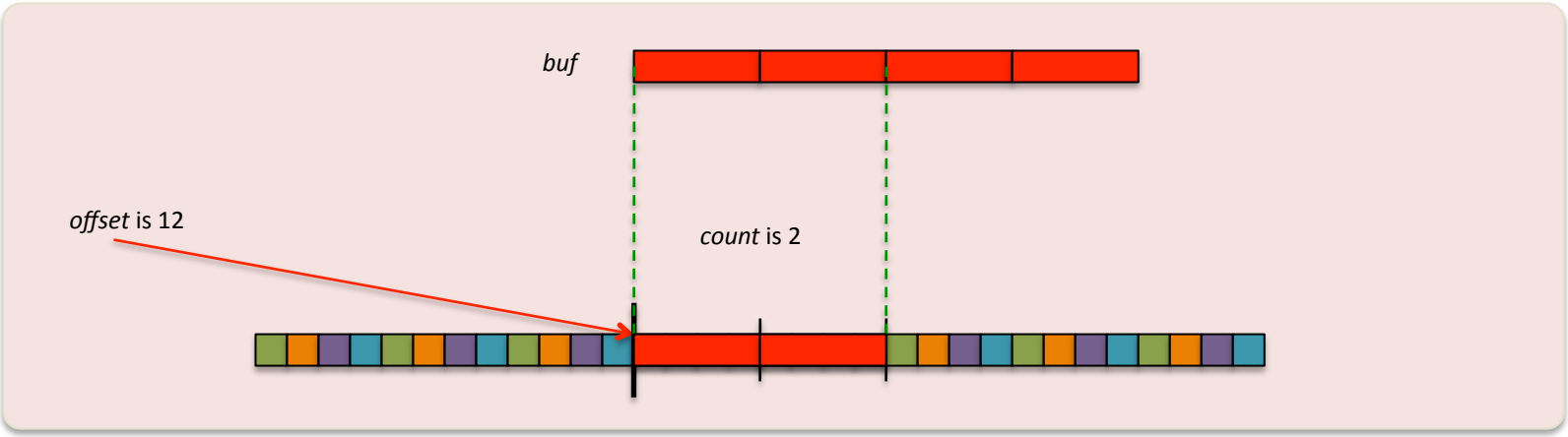
```
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierr)

Integer :: fh, count, datatype, ierr
Integer :: status(MPI_STATUS_SIZE)
Integer(Kind=MPI_OFFSET_KIND) :: offset
<type> :: buf(*)
```

- The routine MPI_File_write_at can be used to write data into a file independently by each process
- Each process specifies an explicit `offset` to write in the file
  - The `offset` is calculated in multiples of the size of the *etype*
- `count` elements of type `datatype` are then written from memory buffer `buf` into the file at the point determined by the `offset`

Independent file write example

```
offset=12;
MPI_File_write_at(fh, offset, buf, 2, MPI_INT, &status);
```

```
offset=12
Call MPI_File_write_at(fh, offset, buf, 2, MPI_INTEGER, status, ierr)
```



Exercise 1

- Take a skeleton MPI code from /project/csstaff/IO_Course/MPI-IO/skeletons
- Add instructions to open and close a file
  - Use file creation and read/write access modes
  - Collectively open the file on all processes
- Use `MPI_File_write_at` to write the Integer value of each *rank* into the file at a displacement of *rank* Integers into the file
  - Check that the file was written correctly using the following command which will display the integer values in the file
    - `od -i <filename>`

```
prompt$ od -i myfile.dat
0000000           0           1           2           3
0000020           4           5           6           7
0000040           8           9          10          11
0000060          12          13          14          15
0000100          16          17          18          19
0000120          20          21          22          23
0000140
```

File Views

- You can define a **view** of the file in order to make it natural for you to deal with your data
- The *view* is defined in terms of a displacement into the file and an *etype*, *filetype* and data representation

Setting a File View – the API

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

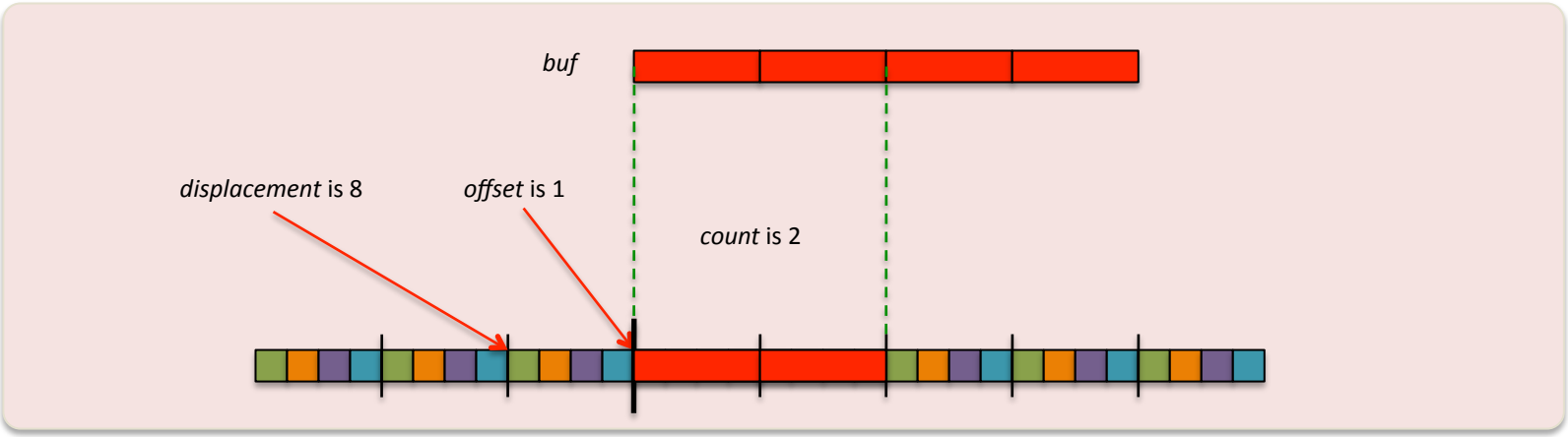
```
MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierr)

Integer :: fh, etype, filetype, info, ierr
Integer(Kind=MPI_OFFSET_KIND) :: disp
Character(*) :: datarep
```

- The routine MPI_File_set_view is used to change the view for a process
- Each process specifies an explicit `disp` that determines where it sees the file beginning
  - The value of `disp` is in bytes
- `etype` defines the new basic type of the file view, and *filetype* must be *etype* or some type derived from multiple copies of *etype*
- `datarep` can be one of “native”, “internal” or “external32”
  - “external32” is a data representation that is supposed to be portable across acrhitectures
    - Many MPI libraries don’t implement “external32”

Set View and Independent File Write Example

```
disp=8;
MPI_File_set_view(fh, disp, MPI_INT, MPI_INT, “native”, MPI_INFO_NULL);
offset=1;
MPI_File_write_at(fh, offset, buf, 2, MPI_INT, &status);
```

```
disp=8
Call MPI_File_set_view(fh, disp, MPI_INTEGER, MPI_INTEGER, “native”, MPI_INFO_NULL, ierror)
offset=1
Call MPI_File_write_at(fh, offset, buf, 2, MPI_INTEGER, status, ierr)
```



Exercise 2

- Take your code from exercise 1 and modify it to set the file view in terms of Integers
  - First use a displacement of 0 and use an offset when writing to put the data into the file
  - Then use an appropriate displacement in setting the view so that you can write the data with 0 offset
- Check your results with od

Routines for Reading and Writing Without Offsets

- Routines are also provided so that you can write into the file without using explicit offsets
- In these cases you set the view prior to carrying out the write so that the view begins where you wish to write the data
- Note that for all of these write routines there are corresponding routines for reading data from existing files

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

```
MPI_File_write(fh, buf, count, datatype, status, ierr)

Integer :: fh, count, datatype, ierr
Integer :: status(MPI_STATUS_SIZE)
<type> :: buf(*)
```

Independent or Collective Data Accesses

- The routines that we have seen so far all provide for **independent** writing of data
  - Each process accesses the file independently of the other processes in the communicator
- There are also corresponding routines that provide for **collective** writing of data
  - Each of these calls is a collective communication
- For large data accesses, collective data accesses can offer good performance improvements
  - The underlying MPI library can aggregate data and implement optimisations tuned to the file system
- The collective routines have the same APIs as the independent routines except that the name of the routine ends in "`_all`"
  - Independent: `MPI_File_write_at`
  - Collective: `MPI_File_write_at_all`

Exercise 3

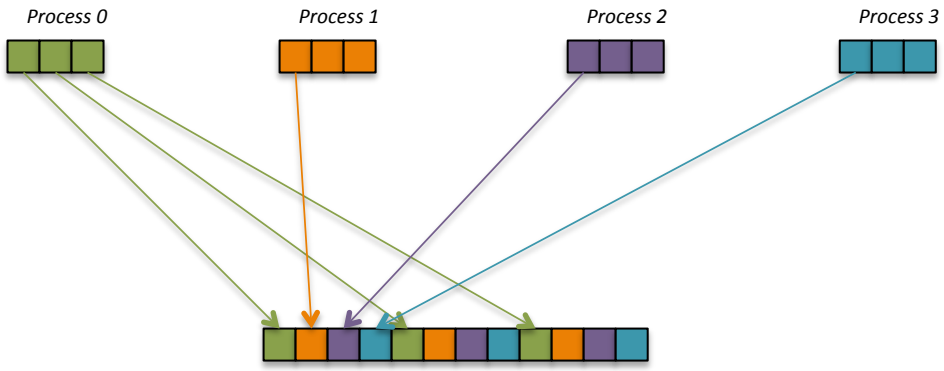
- Modify your code to carry out a collective write without an explicit offset
- Check your results with “od”

Writing Structured Data in Slabs

- One section of the MPI user defined types in MPI 2 is specifically designed to improve data access for MPI-IO
- MPI-2 introduced the idea of a subarray for data from a structured grid distributed on a set of processes
  - Typically the domain decomposition defines a small cuboid inside a larger cuboid
    - Possibly including Halo regions
- By defining a subarray type, we can use a collective call to write our data into one file

Using Subarray Types

- By using subarray types we can construct patterns of how to put data from one process into the file
- An individual process can define a datatype to map data in memory into non-contiguous patterns in the file
- The API for how to define subarrays is not in the 64 pages of MPI-IO routines, as it is just a type definition routine



Defining Subarrays – the API

```
int MPI_Type_create_subarray(int ndims, int sizes[], int subsizes[], int starts[], int
order, MPI_Datatype basetype, MPI_Datatype *subarraytype)
```

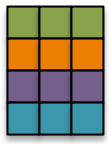
```
MPI_Type_create_subarray(ndims, sizes, subsizes, starts, order,
basetype, subarraytype, ierr)

Integer :: dims, order, basetype, subarraytype, ierr
Integer(:) :: sizes, subsizes, starts
```

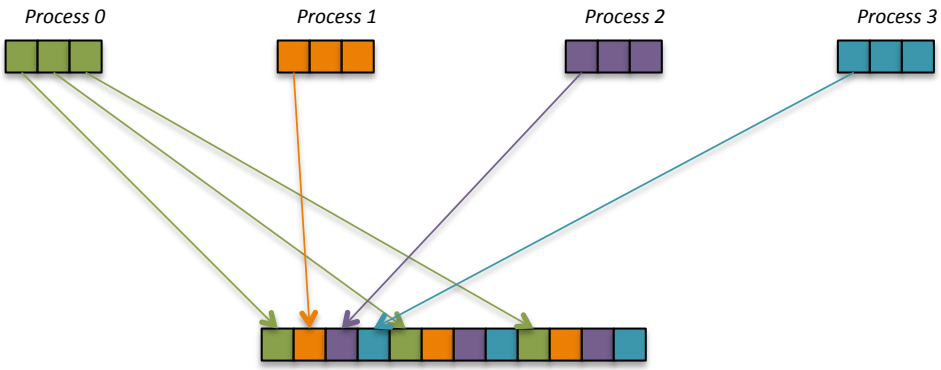
- The routine MPI_Type_create_subarray defines a new type
- The *subarraytype* is based on the original datatype *basetype*
- The creation of the subarray requires you to define the dimensions of the full array as well as the dimensions of the subarray that a particular process holds
- The starts array specifies the offset into the full array for where the subarray begins
- The *order* argument specifies whether the ordering is row-major (MPI_ORDER_C) or column-major (MPI_ORDER_FORTRAN)
  - Use the one appropriate for your programming language

Subarray Constructor Example

```
Integer :: ranknum(4)
.
.
.
sizes=(/ 3, 4 /)
subsizes=(/ 1, 4 /)
starts=(/ wrank, 0 /)
Call MPI_Type_create_subarray(2,sizes,subsizes,starts,MPI_ORDER_FORTRAN,MPI_INTEGER,subarray,ierror)
Call MPI_Type_commit(subarray,ierror)
ranknum(:)=wrank
displacement=0
Call MPI_File_set_view(fh,displacement,MPI_INTEGER, subarray, "native", MPI_INFO_NULL, ierror)
Call MPI_File_write_all(fh,ranknum,4,MPI_INTEGER,status,ierror)
```

Conceptually, the full array is 2D with one element from each column on each processor





We map the data into the file so that it is in column-major order

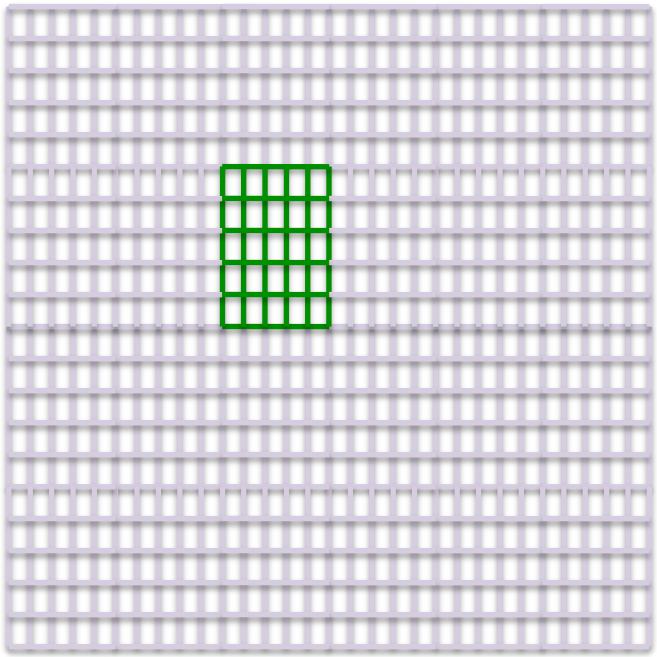
Exercise 4

- Change your code to use a subarray type to write an array of 8 elements, so that each number is *rank* elements apart in the file
- Check your results with “od”
- For 3 processes the output should look as shown
- Verify that it works with any number of processes

```
prompt$ od -i myfile.dat
0000000           0           1           2           0
0000020           1           2           0           1
0000040           2           0           1           2
0000060           0           1           2           0
0000100           1           2           0           1
0000120           2           0           1           2
0000140
```

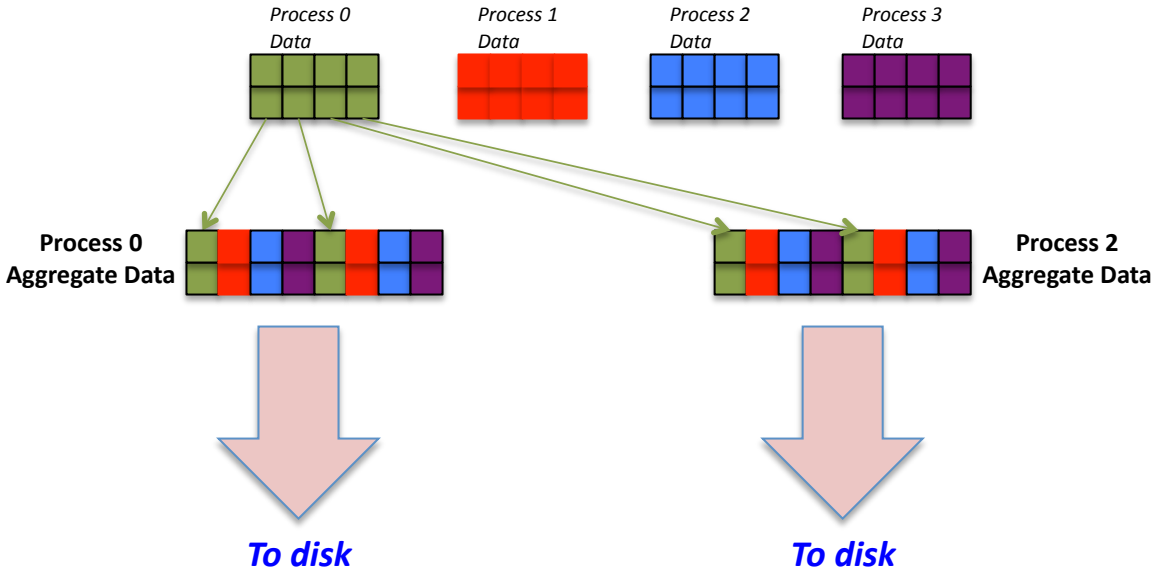
Domain Decomposition Example

- In this example a process holds the data for a small 2D grid inside a larger 2D grid
- We can define a subarray so that this data is also mapped on the file in a linear fashion
- This can be extended to 3 or more dimensions



MPI Collective Writes and Optimisations

- When writing in collective mode, the MPI library carries out a number of optimisations
  - It uses fewer processes to actually do the writing
    - Typically one per node
  - It aggregates data in appropriate chunks before writing



Exercise 5

- Change your code to use a 2 x 2 data array and subarray to write 4 elements in a *numranks* x 4 global array
- The file should have 2 elements on one row and 2 on the row below
- The code will only run with even numbers of elements
- Check your results with “od”
  - Adjust the output width by adding the output flag “-v -width= <4*numranks>”
- For 4 processes the output should look as shown below

```
prompt$ od -v -width=16 -i myfile.dat
0000000           0           0           2           2
0000020           0           0           2           2
0000040           1           1           3           3
0000060           1           1           3           3
```

Using Hints to Improve Performance

- The *info* flag when opening a file can be used to pass optimisation *hints* to the MPI library
  - The hints are provided in (key,value) pairs
- Several predefined hints are reserved in the MPI standard and are available in most MPI libraries
- Selecting appropriate hints can improve performance
- On the Cray systems, the hints in use for a file can be seen by setting the following environment variable
  - export MPICH_MPIIO_HINTS_DISPLAY=1
- The Cray implementation allows you to set hints on files using the environment variable MPICH_MPIIO_HINTS
  - This avoids the need to use the MPI routines to set the *info* flag at file open time
    - e.g. export MPICH_MPIIO_HINTS="*:cb_buffer_size=67108864"

Useful Hints to set for Performance

Hint name	Usage
`cb_buffer_size`	The amount of buffer space reserved for aggregating messages before writing data to the file system (default is 16MB)
`cb_nodes`	The number of nodes to use for aggregating data
`cb_config_list`	A hint as to how to select the nodes for aggregation
`romio_cb_read, romio_cb_write`	Flags to say whether to disable, enable or use heuristics for deciding whether to aggregate for collective reads and writes
`striping_factor`	The number of Lustre stripes to assign to a file
`striping_unit`	The size (in bytes) of the Lustre stripes to use for a file

Several of the hints related to collective buffering on the Cray require that the environment variable MPICH_MPIIO_CB_ALIGN be changed

See the `mpi` man page on the Cray systems for more details

Exercise 6

- Take a copy of the Fortran code in `/project/csstaff/IO_Course/MPI-IO/examples/mpiio_large_grid.f90` and compile it
  - `ftn -O2 mpiio_large_grid.f90 -o mpiio_large_grid`
- Grab 192 cores and launch the job
  - Use `salloc -n 192 --time=00:20:00` to get access to the cores
- Introduce some timing and bandwidth measurement routines
- Use the MPICH_MPIIO_HINTS environment variable to change the striping pattern of the file it creates to see if you can make it go faster
- Do any of the other hints make a difference ?