MPI I/O

Reusing this material

Shared Memory

- Easy to solve in shared memory
  - imagine a shared array called **x**

```
begin serial region
        open the file
        write x to the file
        close the file
end serial region
```

- Simple as every thread can access shared data
  - may not be efficient but it works
- But what about distributed memory?

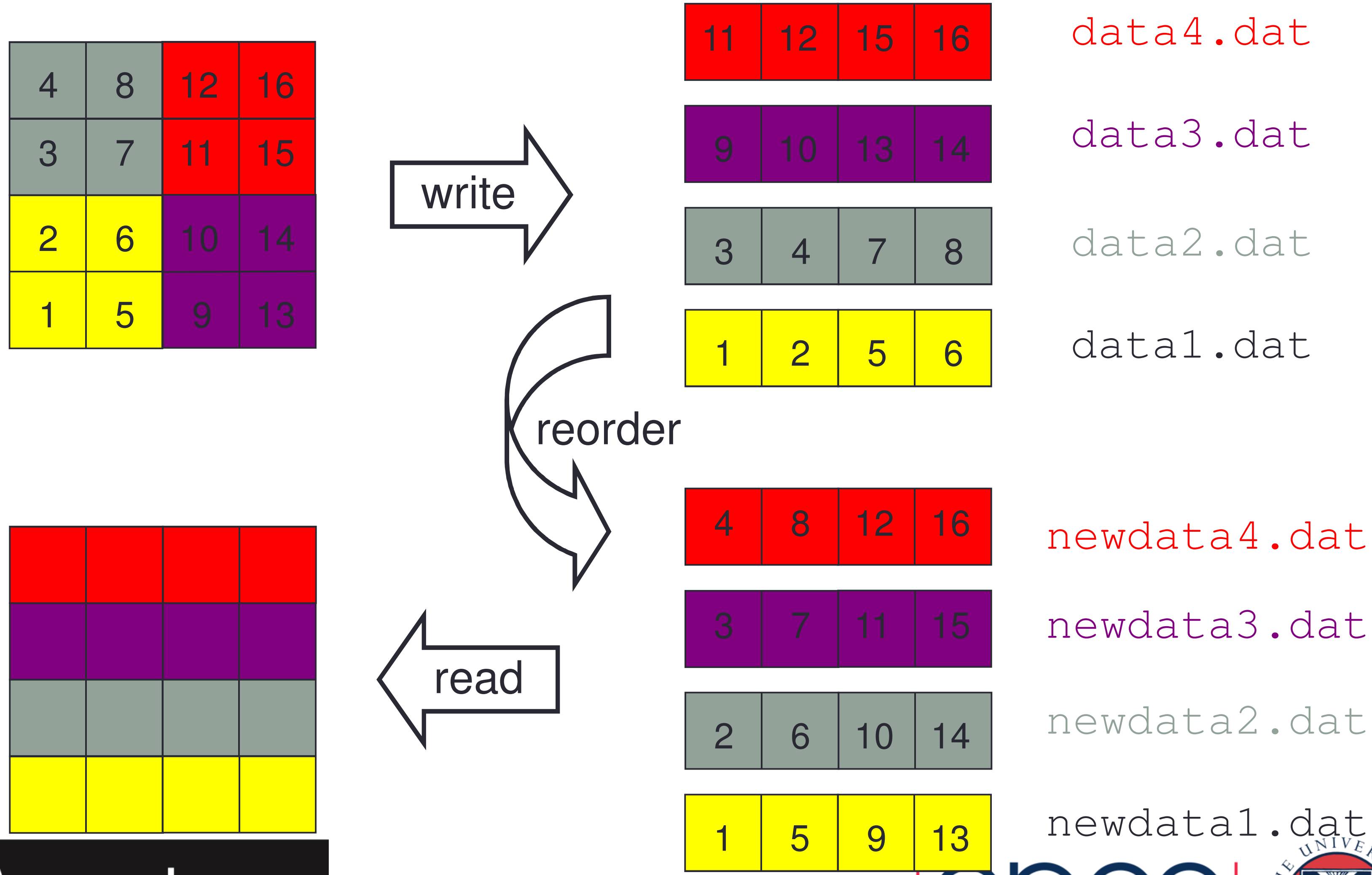
I/O Strategies

- Basic one file for a program
  - Works fine for serial
  - Most codes use this initially
  - Works for shared memory parallelism
- Distributed memory
  - Data now not in single memory space
- Master I/O
  - Use communication to get and send all data from one process
  - High overhead
  - Use single file
  - Memory issues, no access to I/O resources at scale

I/O Strategies cont.

- Individual files
  - Each process writes own file (either on shared filesystem or local scratch space)
  - Use as much of I/O system as possible
  - file contents dependent on number of CPUs and decomposition
  - pre / post-processing steps needed to change number of processes
  - Filesystem breaks down for large numbers of processors
  - File handles or number of files a problem
- Look to better solution
  - I/O libraries

2x2 to 1x4 Redistribution



I/O options

- I/O to single file
  - Everyone involved in I/O
    - Processes write their own data
  - I/O Server/I/O Writers
    - Subset of processes do I/O
- Choice depends on scale and operations to be done and filesystem characteristics
- All I/O
  - Good up to reasonable scale for standard parallel filesystems (10,000s processes)
- Sub I/O
  - Good for very large scale applications or where processing of data is required
  - Enables collection of data and in-situ analytica

Files vs Arrays

- Think of the file as a large array
  - forget that I/O actually goes to disk
  - imagine we are recreating a single large array on a master process
- The I/O system must create this array and save to disk
  - without running out of memory
    - never actually creating the entire array
    - ie without doing naive master I/O
  - and by doing a small number of large I/O operations
    - merge data to write large contiguous sections at a time
  - utilising any parallel features
    - doing multiple simultaneous writes if there are multiple I/O nodes
    - managing any coherency issues re file blocks

MPI-I/O

- Aim to provide distributed access to single file
  - File shared
  - Control by programmer
  - Look like a serial program has written the data
- Part of MPI-2 standard
  - http://www.mpi-forum.org/docs/docs.html
  - Typically available in modern MPI libraries, but if not can use ROMIO (MPI-IO built on MPI-1 calls)
  - Performance dependent on implementation
- Built on MPI collective operations
  - Data structure defined by programmer

MPI-I/O cont.

- Array based I/O
  - Each process creates description of subset it holds (derived datatype)
  - No checking of correctness
- Library handles read and write to files
  - Don’t ever have all in memory
  - Everything done with MPI calls
  - Scale as well as MPI communications
  - Best performance for big reads/writes
- Info object for passing system specific information
  - Lots of optimisations, tweaking, etc…

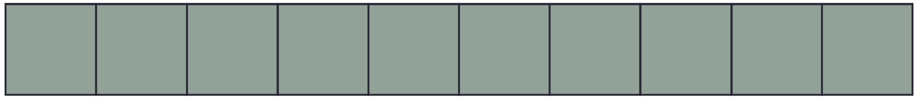
Basic Datatypes

- MPI has a number of pre-defined datatypes
  - eg **MPI_INT** / **MPI_INTEGER, MPI_FLOAT** / **MPI_REAL**
  - user passes them to send and receive operations

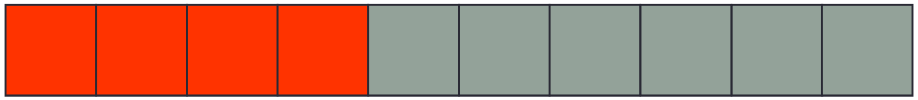
- For example, to send 4 integers from an array x

C: `int[10];`

F: `INTEGER x(10)`



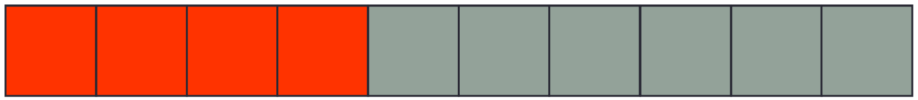
```
MPI_Send(x, 4, MPI_INT, ...);

MPI_SEND(x, 4, MPI_INTEGER, ...)
```



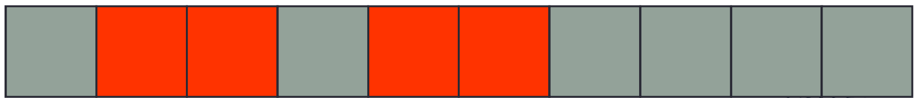
Simple Example

- Contiguous type

```
MPI Datatype my_new_type;
MPI_Type_contiguous(count=4, oldtype=MPI_INT, newtype=&my_new_type);
MPI_Type_commit(&my_new_type);

INTEGER MY_NEW_TYPE
CALL MPI_TYPE_CONTIGUOUS(4, MPI_INTEGER, MY_NEW_TYPE, IERROR)
CALL MPI_TYPE_COMMIT(MY_NEW_TYPE, IERROR)

MPI_Send(x, 1, my_new_type, ...);

MPI_SEND(x, 1, MY_NEW_TYPE, ...)
```



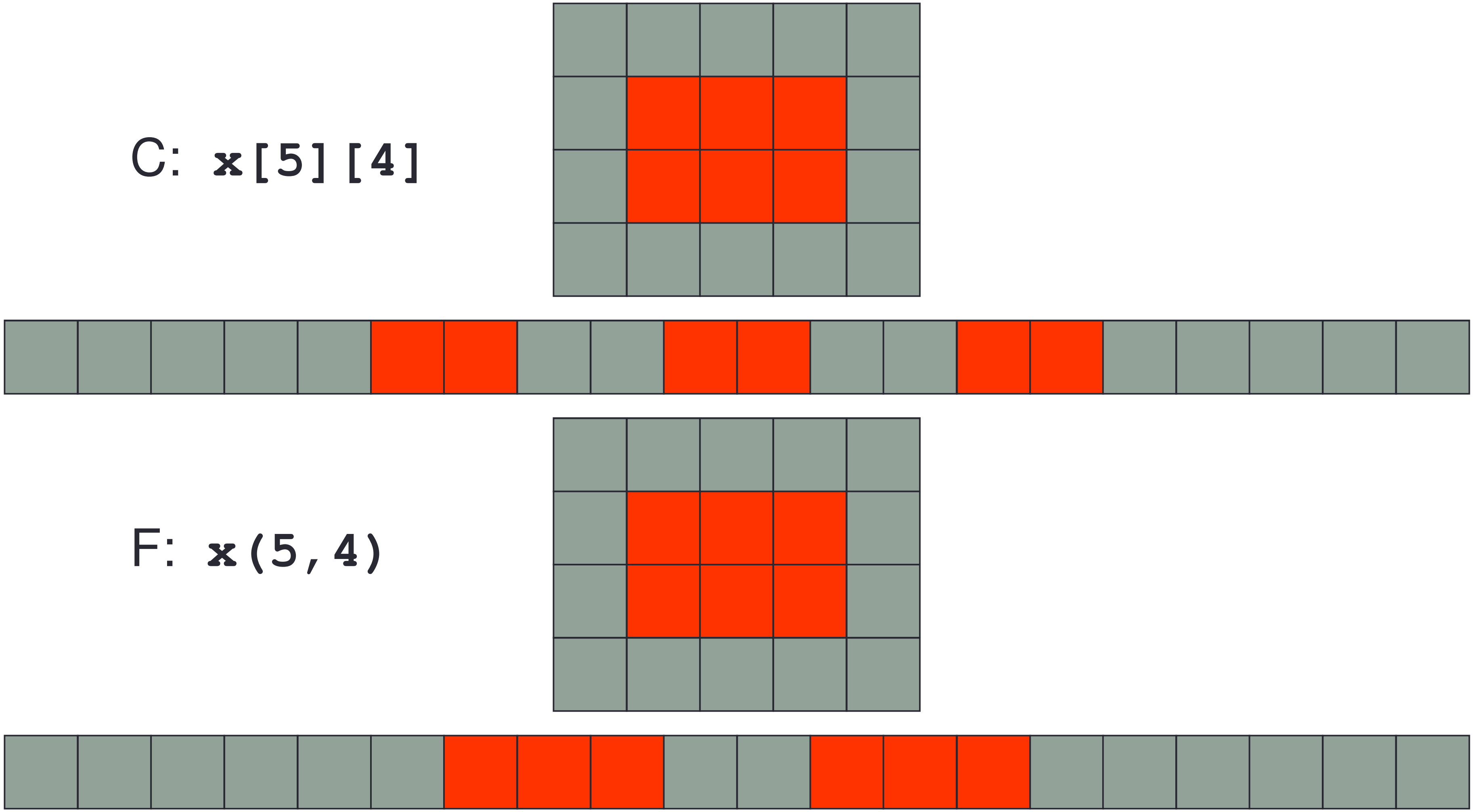
- Vector types correspond to patterns such as



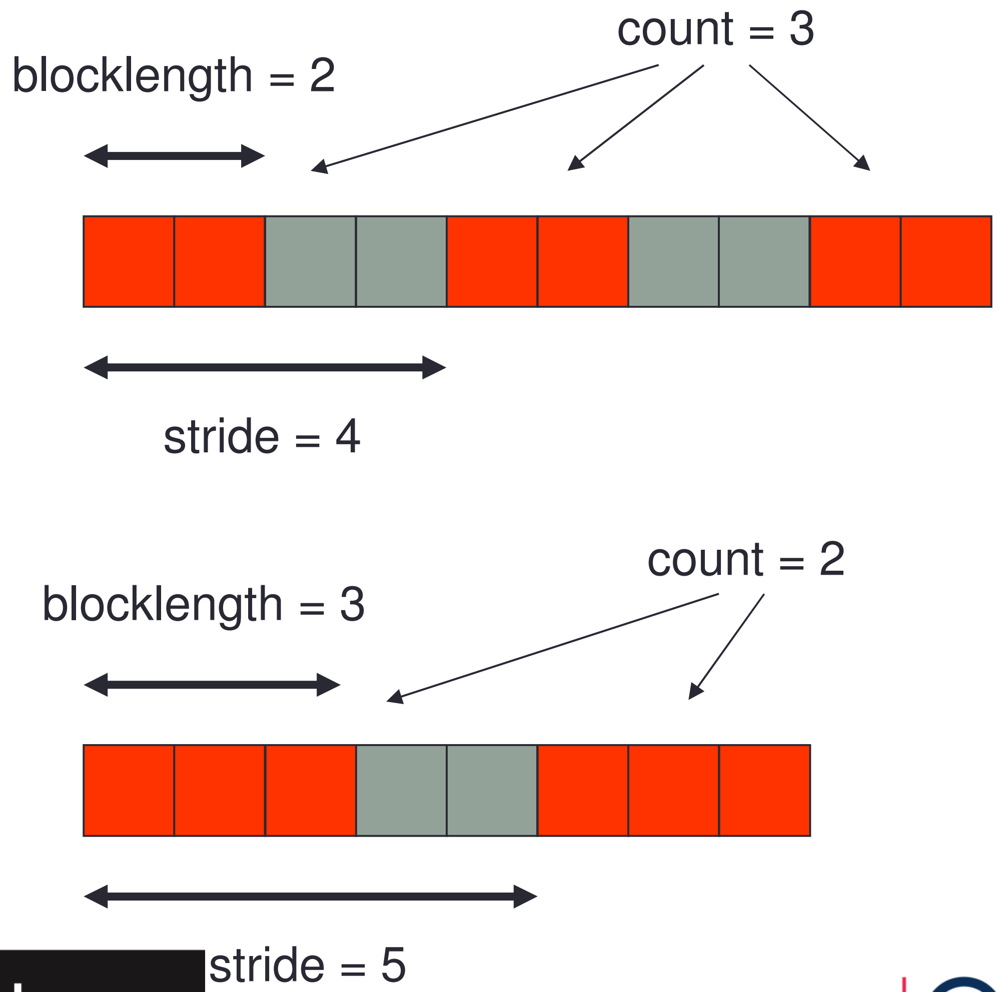
Array Subsections in Memory

C: **`x[5][4]`**

F: **`x(5,4)`**



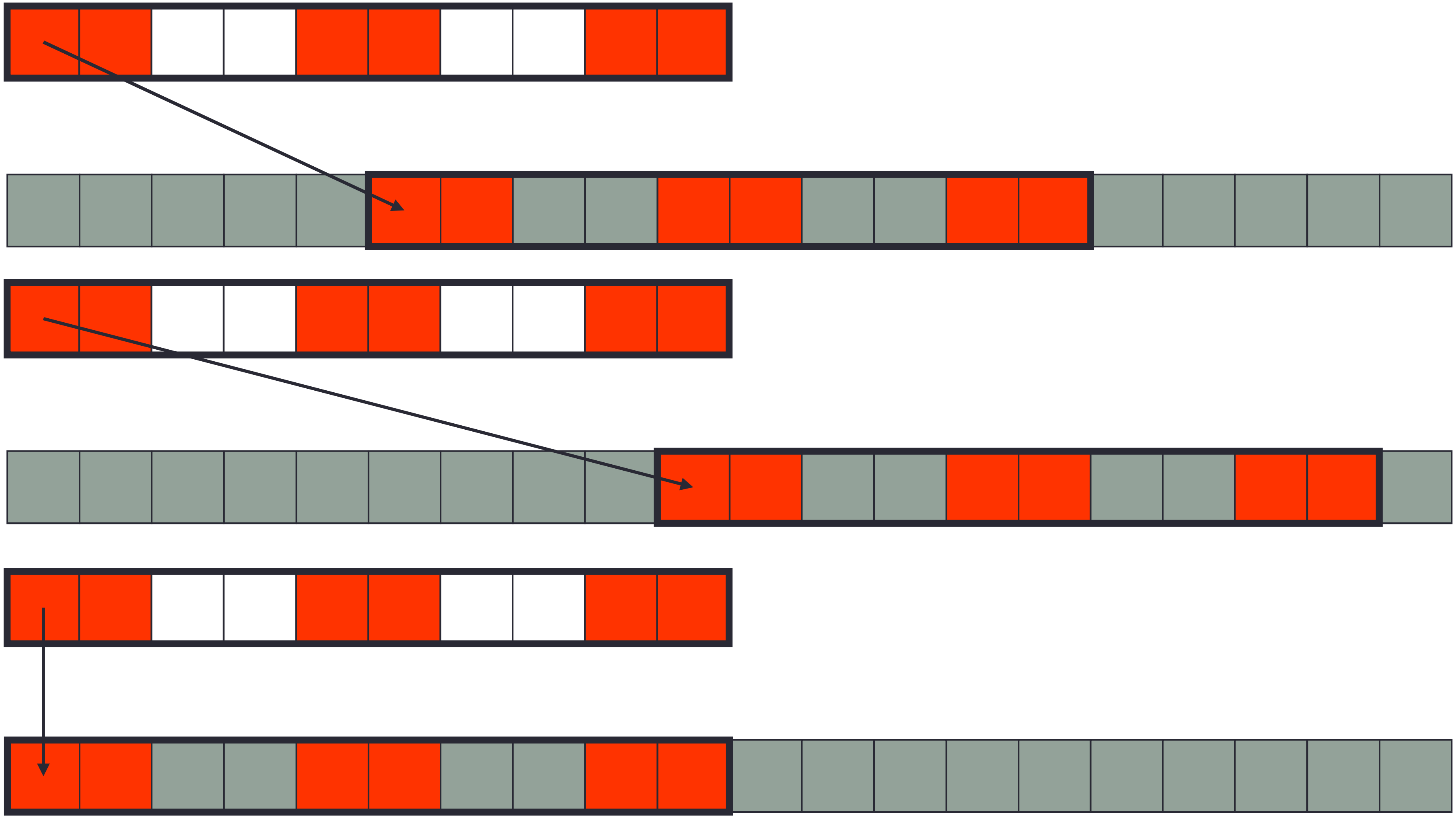
Equivalent Vector Datatypes



Definition in MPI

```
MPI_Type_vector(int count, int blocklength, int stride,
            MPI_Datatype oldtype, MPI_Datatype *newtype);

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE,
                  OLDTYPE, NEWTYPE, IERR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE
INTEGER NEWTYPE, IERR

MPI_Datatype vector3x2;
MPI_Type_vector(3, 2, 4, MPI_FLOAT, &vector3x2)
MPI_Type_commit(&vector3x2)

integer vector3x2
call MPI_TYPE_VECTOR(2, 3, 5, MPI_REAL, vector3x2, ierr)
call MPI_TYPE_COMMIT(vector3x2, ierr)
```

Datatypes as Floating Templates



MPI-IO vs Master IO

- Can use MPI-I/O derived types to do master I/O
  - 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-I/O model
  - each process defines the datatype for its section of the file
  - these are passed into the MPI-I/O routines
  - data is automatically read and transferred directly to local memory
  - there is no single large buffer and no explicit master process

MPI-I/O 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 I/O 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
```

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

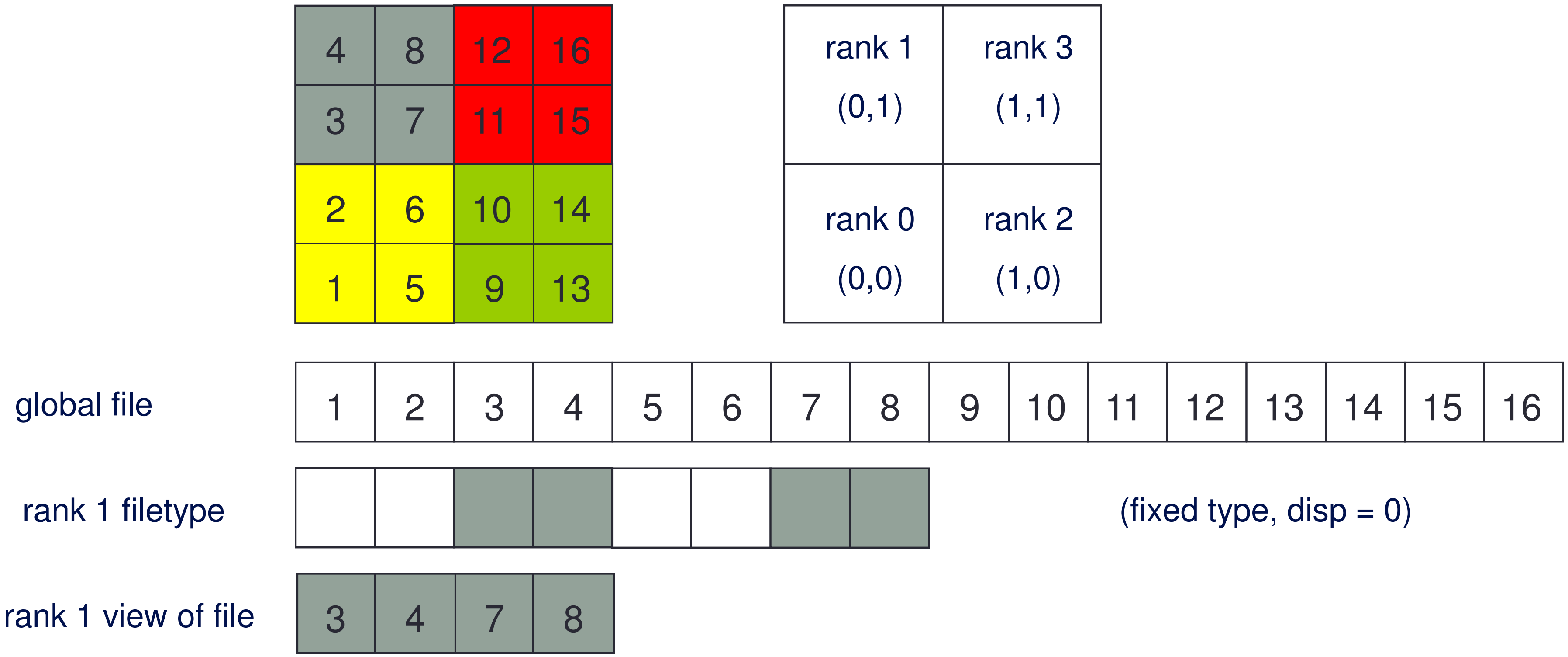
Setting the File View

```
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,
  IERROR)
INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
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

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



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
- 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, ...)
```

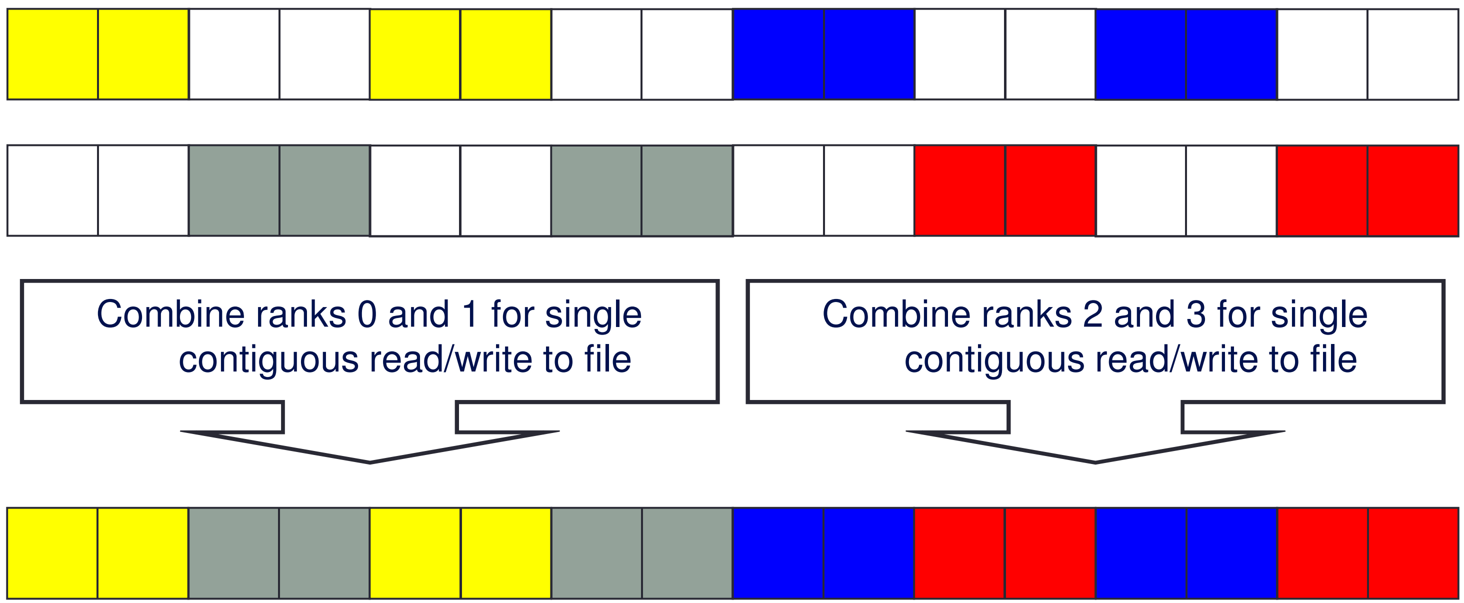
- Recommend setting the view with fixed datatypes
  - and zero displacements

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
- **`etype`** is normally something like **`MPI_REAL`** or **`MPI_FLOAT`**
  - **`datatype`** in read/write calls is usually the same as the **`etype`**

Collective I/O

- For read and write, "`_all`" means operation is collective
  - all processes attached to the file are taking part
- Other I/O 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



Other individual operations

- Alternative approach
  - let everyone see the whole file (i.e. do not set a view)
  - manually seek to correct location using, e.g., **`MPI_File_write_at()`**
  - displacement is in units of the extent of the **`datatype`**

- Disadvantages
  - a very low-level, manual approach less amenable to I/O optimisation
  - danger that each request is handled individually with no aggregation
  - can use **`MPI_File_write_at_all()`** but might still be slow

INFO Objects and Performance

- Used to pass optimisation hints to MPI-I/O
  - 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
  - **`MPI_Info_create`**
  - **`MPI_Info_set`**
  - **`MPI_Info_free`**
  - see man pages for details
- 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-I/O hints

Non-blocking I/O in MPI-I/O

- Two forms
  - General non-blocking
    - `MPI_File_iwrite(fh, buf, count, datatype, request)`
    - finish by waiting on `request`
    - but no collective version
  - Split collective
    - `MPI_File_write_all_begin(fh, buf, count, datatype)`
    - `MPI_File_write_all_end(fh, buf, status)`
    - only a single outstanding I/O operation at any one time
    - allows for collective version

MPI-I/O

- MPI-I/O 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 I/O library to merge reads and writes
  - enables a smaller number of larger I/O operations from/to disk