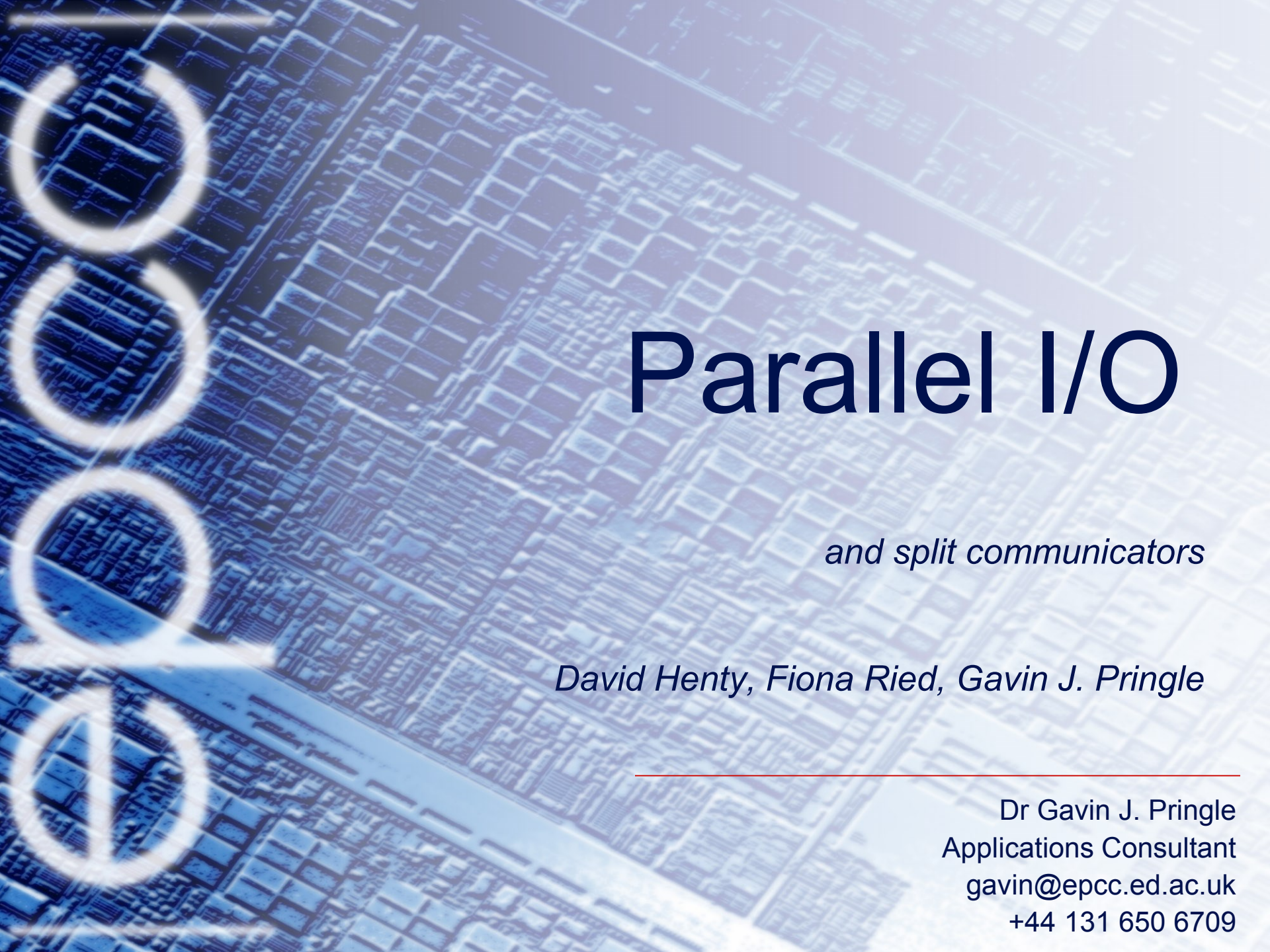


Parallel I/O

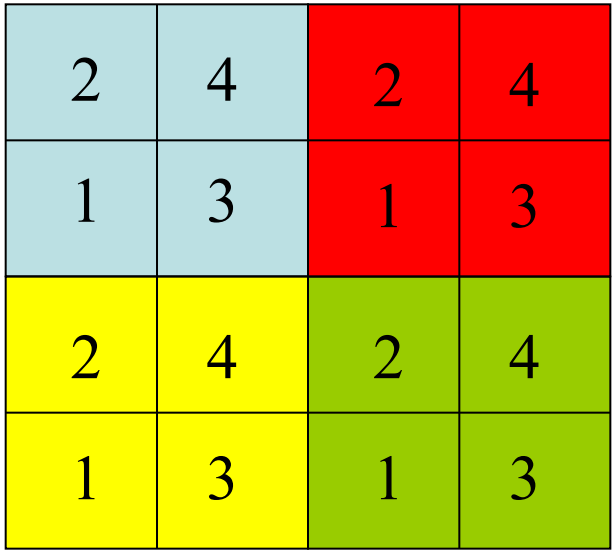
and split communicators

David Henty, Fiona Ried, Gavin J. Pringle

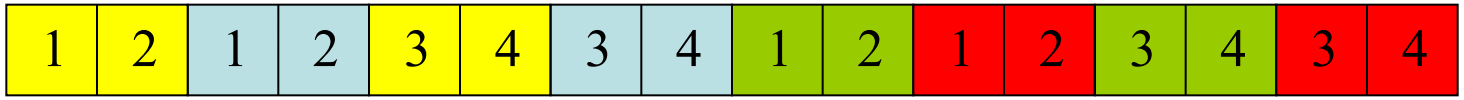
Dr Gavin J. Pringle
Applications Consultant
gavin@epcc.ed.ac.uk
+44 131 650 6709

4x4 array on 2x2 Process Grid

Parallel Data



File



Shared Memory

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

```
begin serial region
      open the file
      write data 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 message-passing?

Message Passing

- Single master IO processor
- Multiple IO processors
  - All processors write their own files
  - A subset of all process write their own files

Master IO

- All processors send their data to the Master
- If the master has large enough memory
  - Create a single array
  - Write to a single file
- If master memory is too small
  - Receive data from each process in turn
  - Append data to file
    - Order will be important
- But does not benefit from a parallel fs that supports multiple write streams

Multiple IO processors, single file

- Cannot have multiple processors writing to a single file
- Unix cannot cope with this
- Not even sufficient to have processes writing to distinct parts of a file
- Even reading can be difficult
  - 1024 processes opening a file can over load the file system
- Data is typically distributed across different processes
  - Processes do not in general own contiguous chunks of the file
  - Cannot easily do linear writes
  - Local data my have ghost cells which need to be ignored.
- Parallel file systems may allow multiple access
  - but complicated and difficult for the user to manage

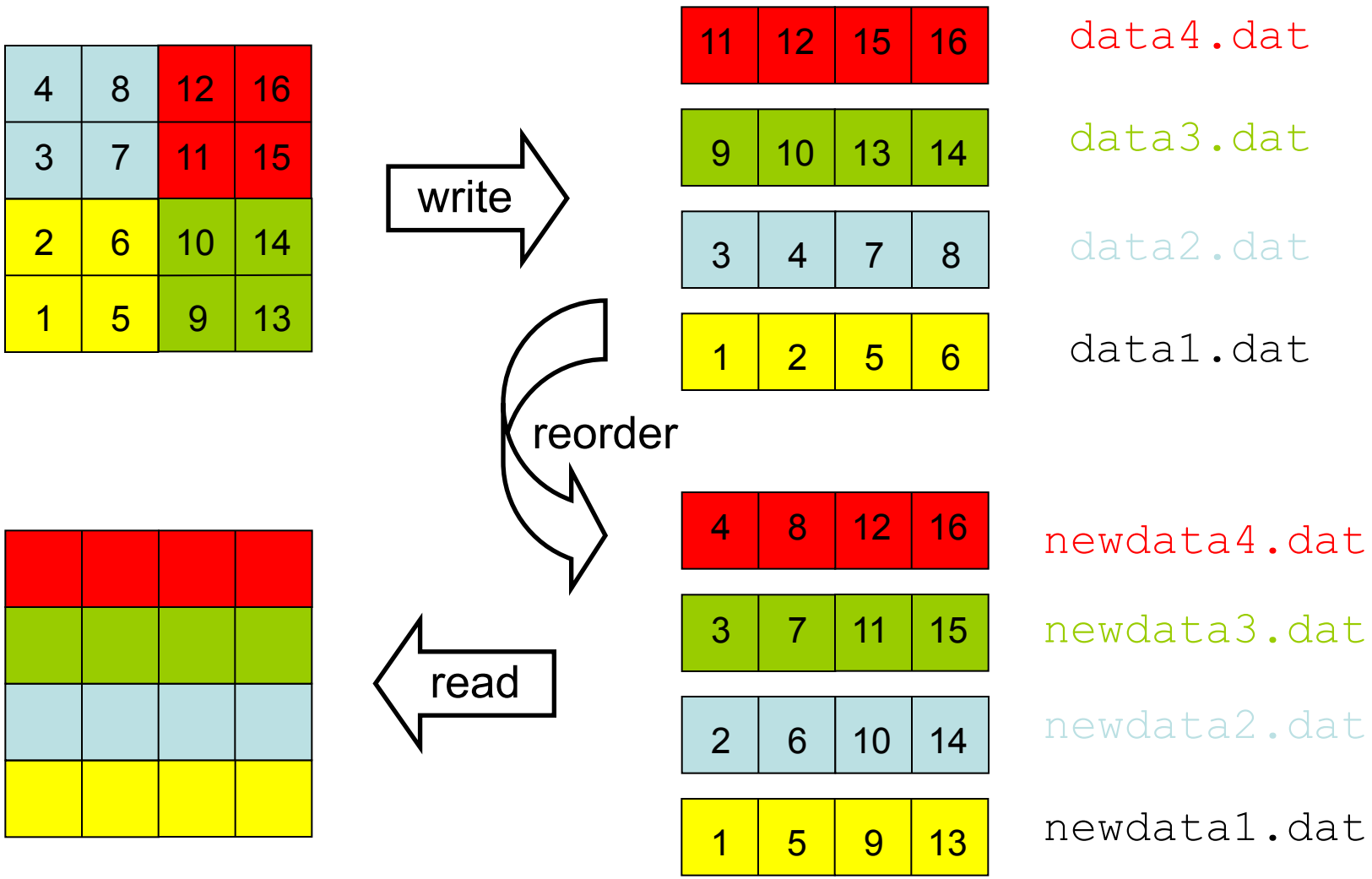
- Solution is to have Multiple IO processors with multiple files

Multiple IO processors, multiple files

- All processors write their own data to their own file
  - N processors create N files

- Major problem is reassembling data
  - contents of the file are dependent on the decomposition
  - pre and post-processing steps to change number of processes
    - Each process writes to a local file system and user copies back to home
    - or each process opens a unique file (dataXX.dat) on shared fs
  - but at least this approach means that reads and writes are in parallel
    - but may overload file system for many processes

2x2 to 1x4 Redistribution

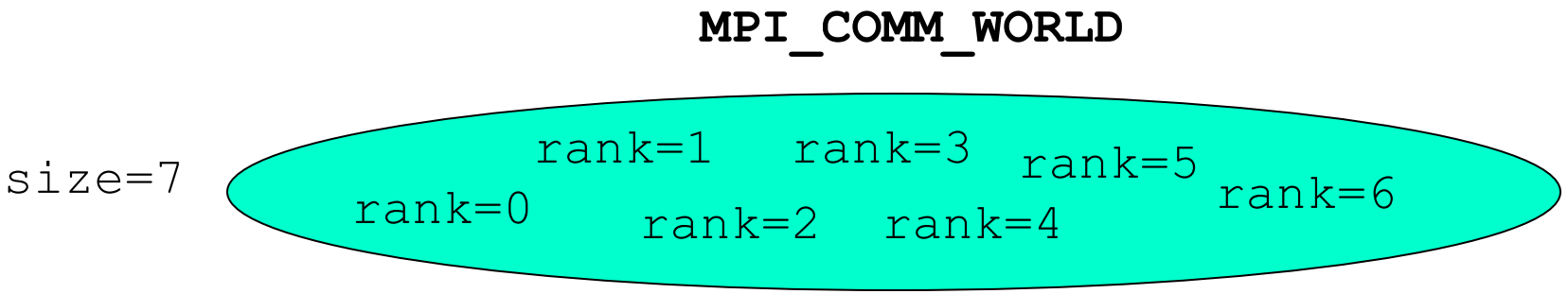


Multiple IO processors, multiple files (cont.)

- Only some processors perform IO
  - More efficient than using all processors or just one IO processor
- Most efficient number of IO processors is
  - Problem dependant
  - System dependant
- Highly beneficial to employ split communicators

Communicators

- All MPI communications take place within a communicator
  - a group of processes with necessary information for message passing
  - there is one pre-defined communicator: **`MPI_COMM_WORLD`**
  - contains all the available processes
- Messages move within a communicator
  - E.g., point-to-point send/receive must use same communicator
  - Collective communications occur in single communicator



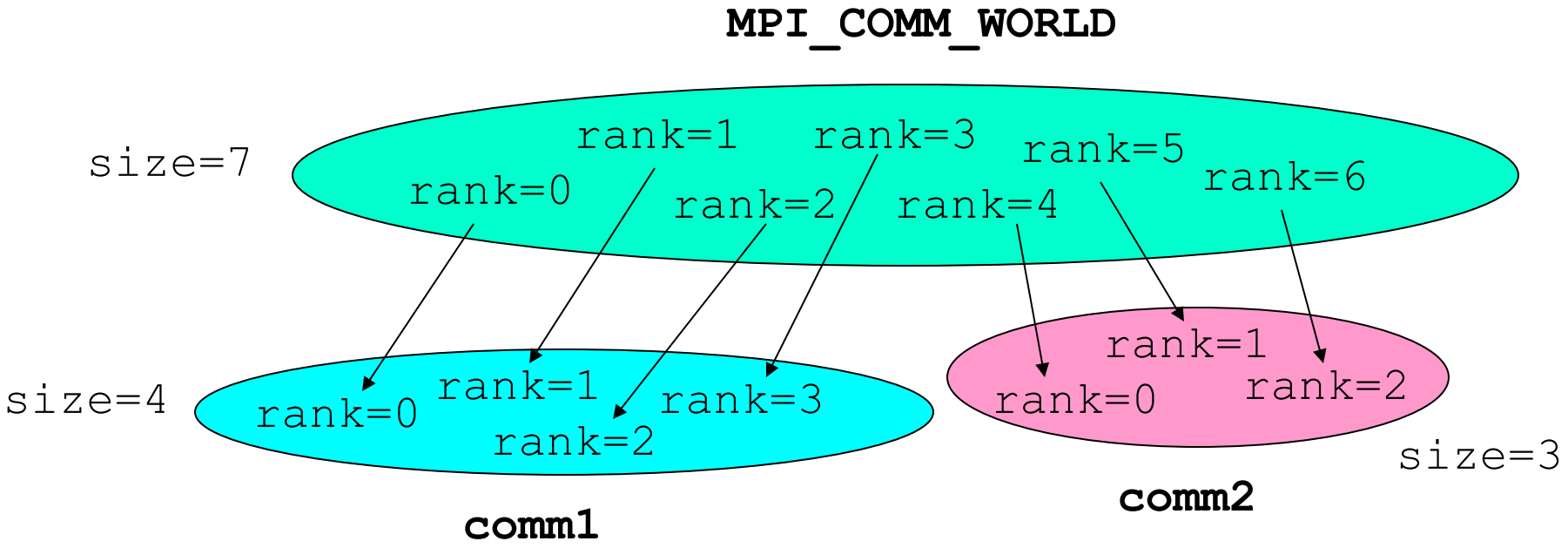
Use of communicators

- Question: Can I just use **`MPI_COMM_WORLD`** for everything?
- Answer: Yes
  - many people use **`MPI_COMM_WORLD`** everywhere in their MPI programs
- Better programming practice suggests
  - abstract the communicator using the MPI handle
  - such usage offers very powerful benefits

```
MPI_Comm comm;          /* or INTEGER for Fortran */
comm = MPI_COMM_WORLD;
...
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
....
```

Split Communicators

- It is possible to sub-divide communicators
- E.g.,split **`MPI_COMM_WORLD`**
  - Two sub-communicators can have the same or differing sizes
  - Each process has a new rank within each sub communicator
  - Messages in different communicators guaranteed not to interact



MPI interface

- **`MPI_Comm_split()`**
  - splits an existing communicator into disjoint (i.e. non-overlapping) subgroups
- Syntax, C:

```
int MPI_Comm_split(MPI_Comm comm, int colour, int
                                        key, MPI_Comm *newcomm)
```

- Fortran:

```
MPI_COMM_SPLIT(COMM, COLOUR, KEY, NEWCOMM, IERROR)

INTEGER COMM, COLOUR, KEY, NEWCOMM, IERROR
```

- **`colour`** – controls assignment to new communicator
- **`key`** – controls rank assignment within new communicator

What happens…

- **`MPI_Comm_split()`** is collective
  - must be executed by **all** processes in group associated with **`comm`**
- New communicator is created
  - for each unique value of **`colour`**
  - All processes having the same **`colour`** will be in the same sub-communicator
- New ranks 0…size-1
  - determined by the (ascending) value of the key
  - If keys are same, then MPI determines the new rank
  - Processes with the same **`colour`** are ordered according to their **`key`**
- Allows for arbitrary splitting

Split Communicators – Fortran example

```
integer :: comm, newcomm
integer :: colour, rank, size, errcode
comm = MPI_COMM_WORLD
call MPI_COMM_RANK(comm, rank, errcode)

! Again, set colour according to rank

colour = mod(rank,2)
call MPI_COMM_SPLIT(comm, colour, rank, newcomm,&
errcode)
MPI_COMM_SIZE(newcomm, size, errcode)
MPI_COMM_RANK(newcomm, rank, errcode)
```

Split Communicators – C example

```
MPI_Comm comm, newcomm;
int colour, rank, size;
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &rank);
/* Set colour depending on rank: Even numbered ranks have
   colour = 0, odd have colour = 1 */
colour = rank%2;
MPI_Comm_split(comm, colour, rank, &newcomm);
MPI_Comm_size (newcomm, &size);
MPI_Comm_rank (newcomm, &rank);
```

Diagrammatically

- Rank and size of the new communicator

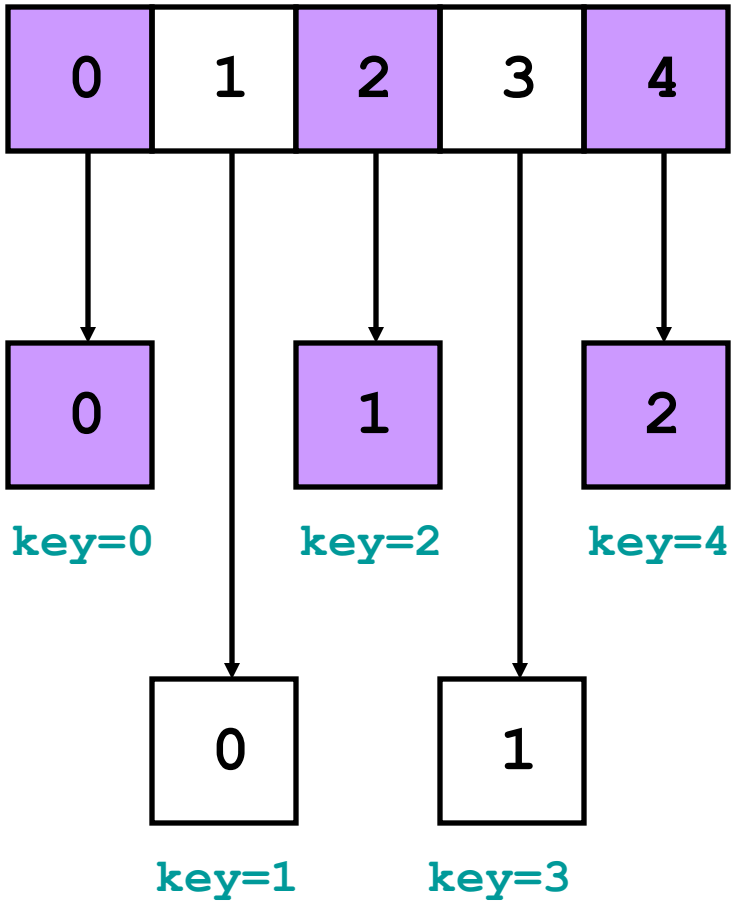
```
MPI_COMM_WORLD, size=5

color = rank%2;

key = rank;
```

```
newcomm, color=0, size=3
```

```
newcomm, color=1, size=2
```



Freeing Communicators

- **`MPI_Comm_free()`**
  - a **collective** operation which destroys an unwanted communicator
- Syntax, C:

```
int MPI_Comm_free(MPI_Comm * comm)
```

- Fortran:

```
MPI_COMM_FREE(COMM, IERROR)

INTEGER COMM, IERROR
```

  - Any pending communications which use the communicator will complete normally
  - Deallocation occurs only if there are no more active references to the communication object

Advantages of Communicators

- Many requirements can be met by using communicators
  - Can't I just do this all with tags?
  - Possibly, but difficult, painful and error-prone
- Easier to use collective communications than point-to-point
  - Where subsets of **`MPI_COMM_WORLD`** are required
  - For example
    - averages over coordinate directions in Cartesian grids
    - parallel IO
- In dynamic problems
  - Allows controlled assignment of different groups of processors to different tasks at run time

Applications, for example

- Linear algebra
  - row or column operations or act on specific regions of a matrix (diagonal, upper triangular etc)
- Hierarchical problems
  - Multi-grid problems e.g. overlapping grids or grids within grids
  - Adaptive mesh refinement
    - E.g. complexity may not be known until code runs, can use split comms to assign more processors to a part of the problem
- Taking advantage of locality
  - Especially for communication (e.g. group processors by node)
- Multiple instances of same parallel problem
  - Task farms

Parallel IO communicators

- Create M sets of processors
  - Each set will have its own master IO
  - Writes/reads from M files in total
- Each set is a new communicators
- All processor then send their data to the master IO processes
  - If master has enough memory, then master can contain all data and then perform a single read/write operation
  - If master has limited memory, then master can receive and write chunks of the data.f
- The problems of multiple data files remain
  - But at least the number of data files has been reduced

What do we really need?

- Using Parallel IO with MPI communicators is a good start
- But we really need a way to do parallel IO efficiently
  - where the IO system deals with all the system specifics

- Want a single file format
- We already have one: the serial format
  - all files should have same format as a serial file
  - entries stored according to position in global array
    - not dependent on which process owns them
  - order should always be 1, 2, 3, 4, ..., 15, 16

Information on Machine

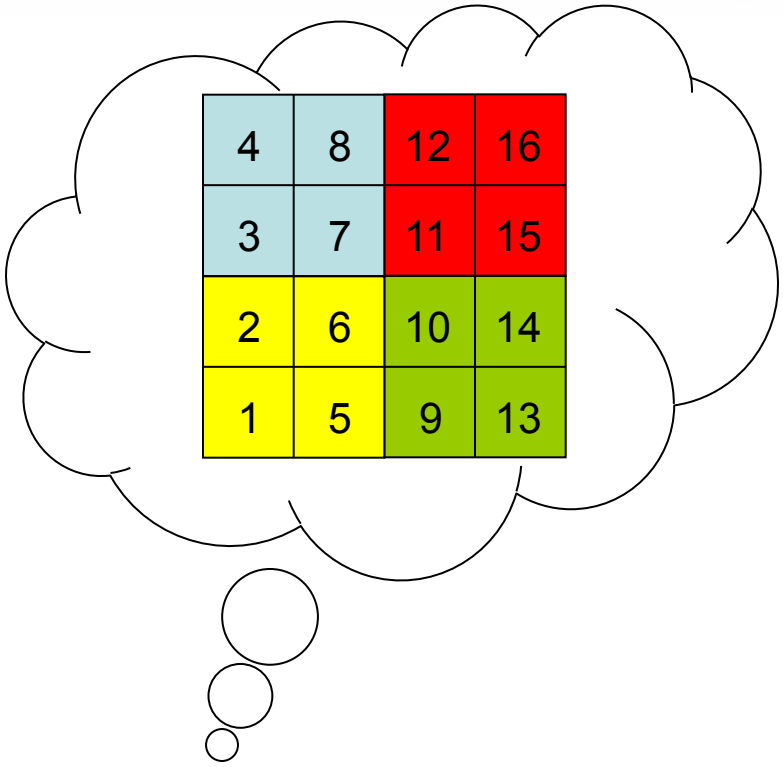
- What does the IO system need to know about the parallel machine?
  - all the system-specific file system details
  - block sizes, number of IO nodes, etc.

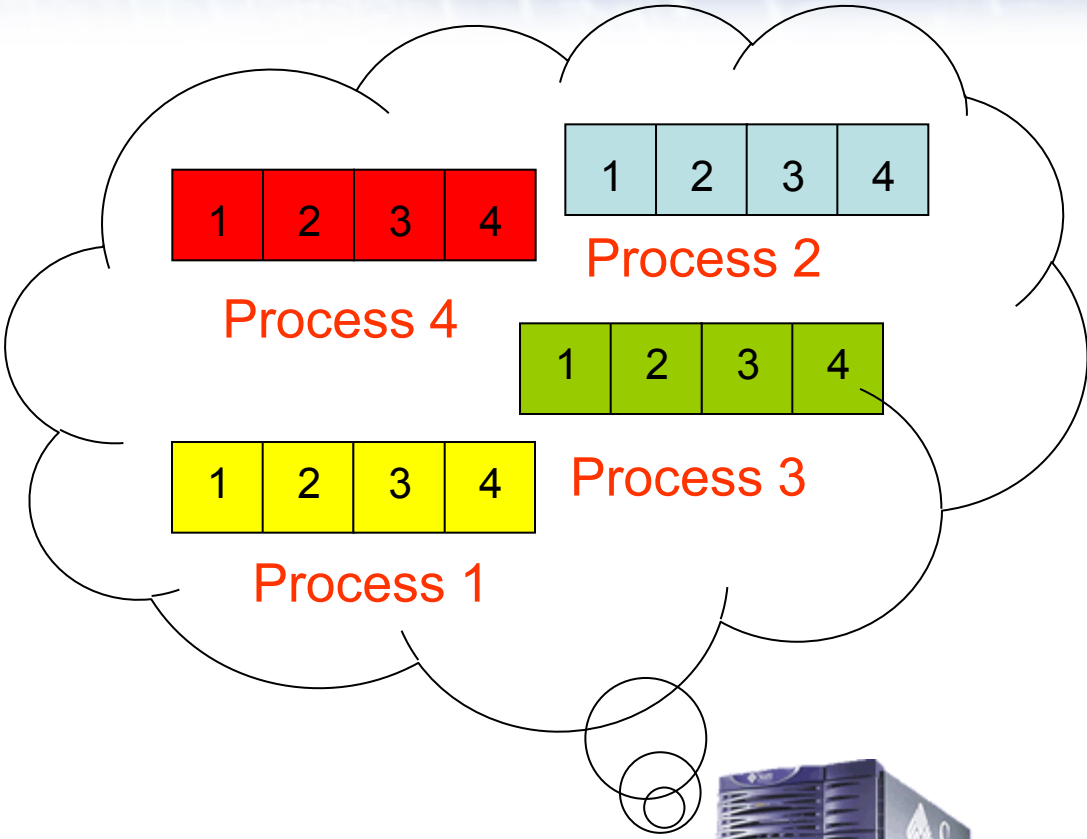
- All this should be hidden from the user
  - but the user may still wish to pass system-specific options
  - how can this be done in a portable manner?

Information on Data Layout

- What does the IO system need to know about the data?
  - how the local arrays should be stitched together to form the file
- But ...
  - mapping from local data to the global file is only in the mind of the programmer!
  - the program does not know that we imagine the processes to be arranged in a 2D grid
- How do we describe data layout to the IO system
  - without introducing a whole new concept to MPI?
  - cartesian topologies are not sufficient
    - do not distinguish between block and block-cyclic decompositions

Programmer View vs Machine View





Files vs Arrays

- Think of the file as a large array
  - forget that IO actually goes to disk
  - imagine that we are simply recreating a single large array on some master process
- The IO 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 IO
  - and by doing big writes
    - try to create and write large contiguous sections at a time
  - utilising any parallel features
    - doing multiple simultaneous writes if there are multiple IO nodes

Solution is to use Parallel IO libraries

- MPI-IO
  - Part of the MPI-2 standard
  - You don't have to have MPI-2 to have MPI-IO
    - ROMIO is an MPI-IO implementation that uses MPI-1 calls
    - Builds on most MPI systems
    - see: www-unix.mcs.anl.gov/**romio**/
  - MPI-IO now comes with most MPI's by default
  - Very difficult to use
- Better still to use a self-describing IO format and library
  - HDF5
    - HDF5 files contain complete information on their structure
    - http://hdf.ncsa.uiuc.edu/HDF5/
  - Parallel NETCDF
    - http://trac.mcs.anl.gov/projects/parallel-netcdf
  - Both employ MPI-IO

Summary

- Parallel IO is difficult
- Single IO process is easiest to construct
  - Highly inefficient
- Multiple IO processors is more efficient
- Split Communicators are extremely useful
  - Not just for parallel IO but for many HPC codes
- Issues of multiple data files remain
- Libraries may hold the solution
  - Can be very complex to use

Thanks you

- Any questions?

- gavin@epcc.ed.ac.uk