Advanced Message-Passing Programming

Overview of Parallel IO

Reusing this material

Why is IO hard?

- Breaks out of the nice process/memory model
  - data in memory has to physically appear on an external device
- Files are very restrictive
  - linear access probably implies remapping of program data
  - just a string of bytes with no memory of their meaning

- Many, many system-specific options to IO calls
- Different formats
  - text, binary, big/little endian, Fortran unformatted, HDF5, NetCDF, ...
- Disk systems are very complicated
  - RAID disks, many layers of caching on disk, in memory, ...

- IO is the HPC equivalent of printing!

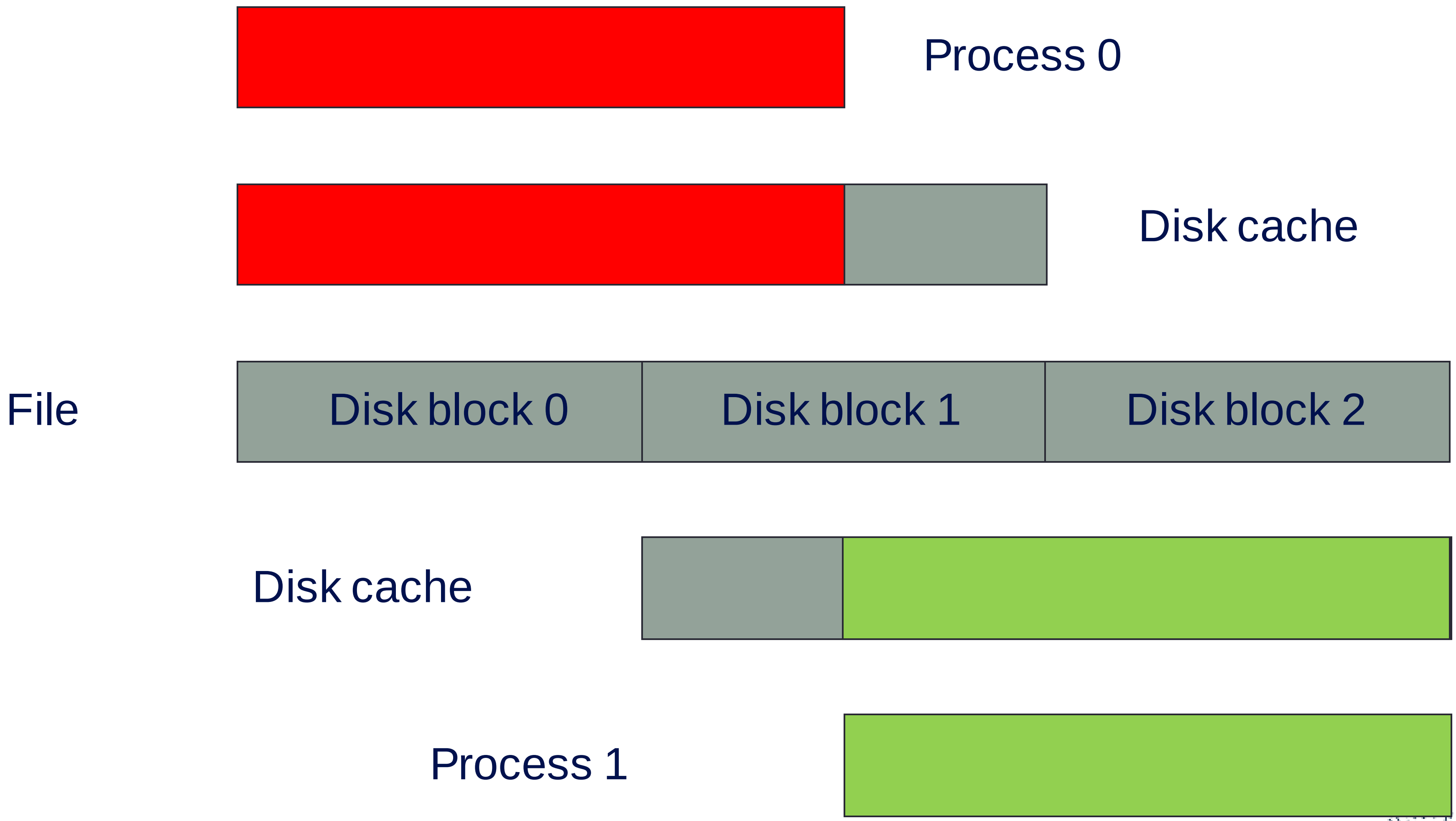
Why is Parallel IO Harder?

- Cannot have multiple processes writing to a single file
  - Unix generally cannot cope with this
  - data cached in units of disk blocks (eg 4K) and is *not coherent*
  - not even sufficient to have processes writing to distinct parts of file

- Even reading can be difficult
  - 1024 processes opening a file can overload the filesystem (fs)

- Data is distributed across different processes
  - processes do not in general own contiguous chunks of the file
  - cannot easily do linear writes
  - local data may have halos to be stripped off

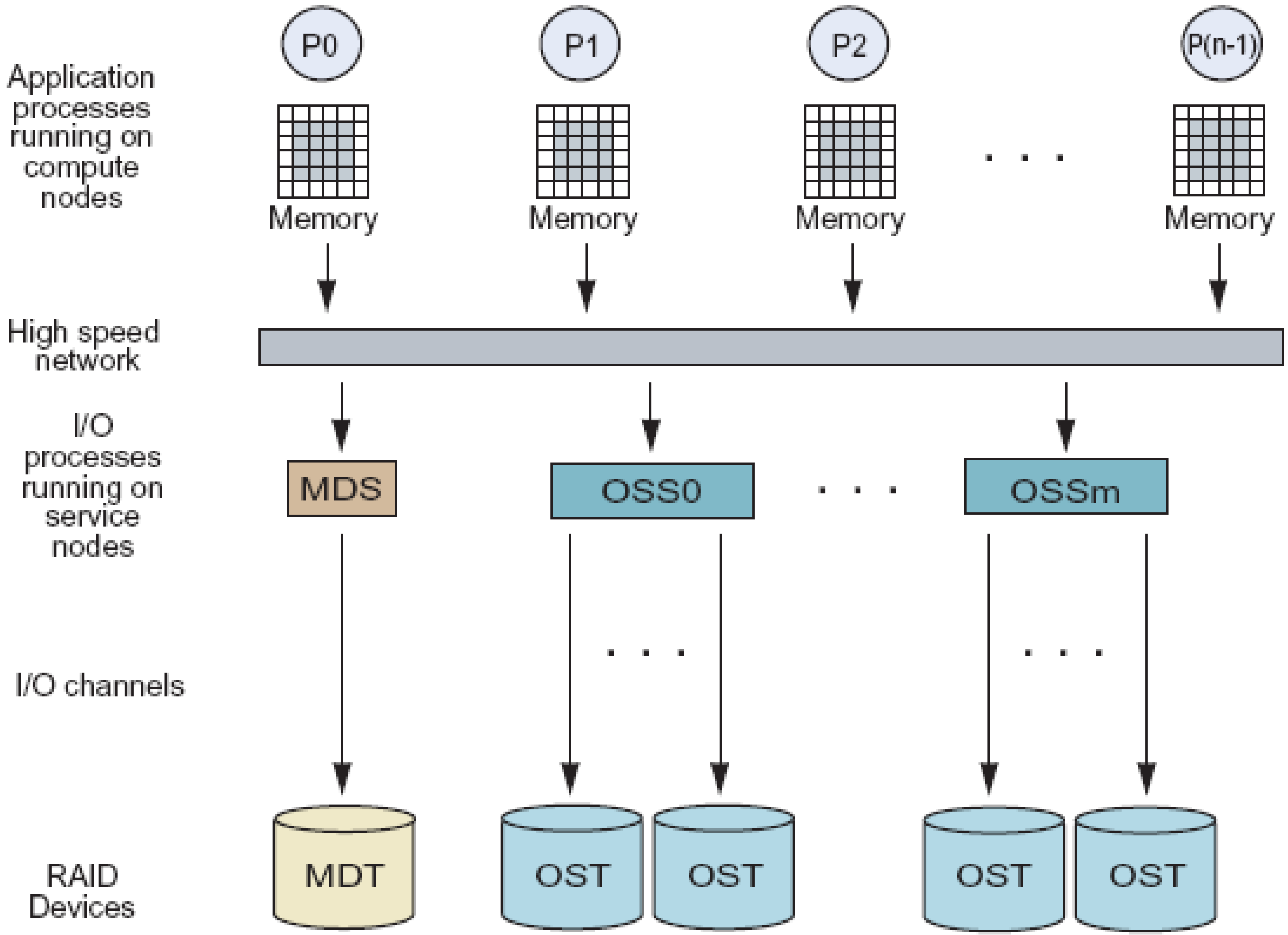
Simultaneous Access to Files



Parallel File Systems

- Parallel computer
  - constructed of many standard processors, each not particularly fast
  - performance comes from using many processors at once
  - requires manual distribution of data and calculation across processors
- Parallel file systems
  - constructed from many standard disks, each not particularly fast
  - performance comes from reading / writing to many disks at once
  - requires many *clients* to read / write to different disks
    - each *node* appears as a separate IO client
  - data from a single file can be *striped* across many disks
- Must appear as a single file system to user
  - typically have a single *MetaData* Server (MDS)

Parallel File Systems: Lustre



ARCHER’s (**not** ARCHER2) Cray Sonexion Storage

MMU: ***Metadata Management Unit***

1

Lustre MetaData Server

- Contains server hardware and storage

SSU: ***Scalable Storage Unit***

2

2 x OSSs and 8 x OSTs (Object Storage Targets)

- Contains Storage controller, Lustre server, disk controller and RAID engine
- Each unit is 2 OSSs each with 4 OSTs of 10 (8+2) disks in a RAID6 array

3

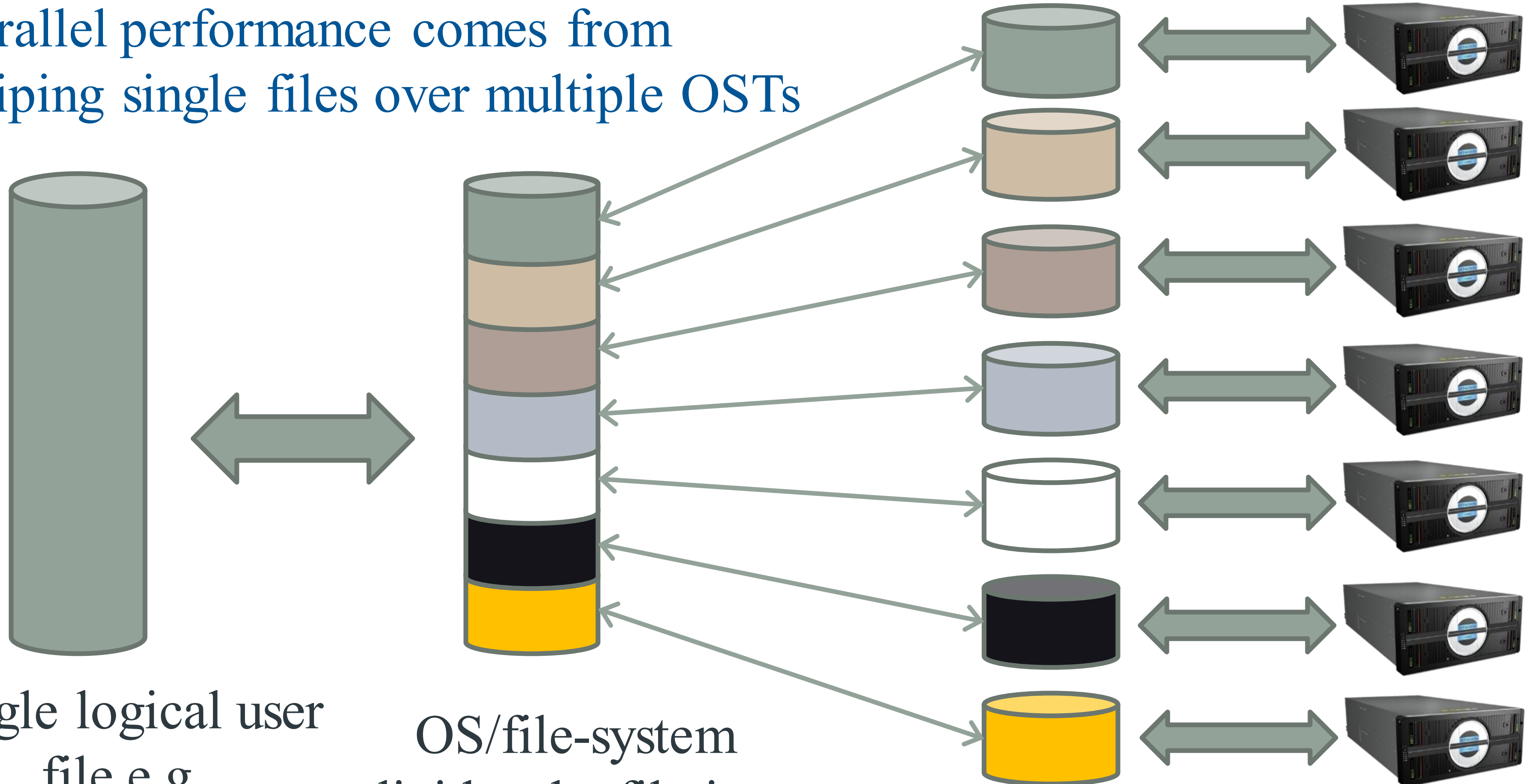
Multiple SSUs are combined to form storage racks

Independent I/O

- Also called File-per-process
- By default, each file stored on a single OST
  - simplest to think of each Object Storage Target as a disk
- The MDS will assign different files to different disks
  - in principle achieves maximum bandwidth
- But ...
  - a nightmare to cope with in practice (thousands of files)
  - opening and closing files is serialised via a single MDS
    - this can be a performance bottleneck
- Mitigations
  - multiple filesystems with fewer OSTs, each with their own MDS
  - write a file-per-node rather than file-per-process

Lustre data striping

Parallel performance comes from striping single files over multiple OSTs



Single logical user file e.g. `/work/q01/q01/user/bigfile.dat`

OS/file-system divides the file into stripes *if requested by the user*

Stripes read/written to/from their assigned OST

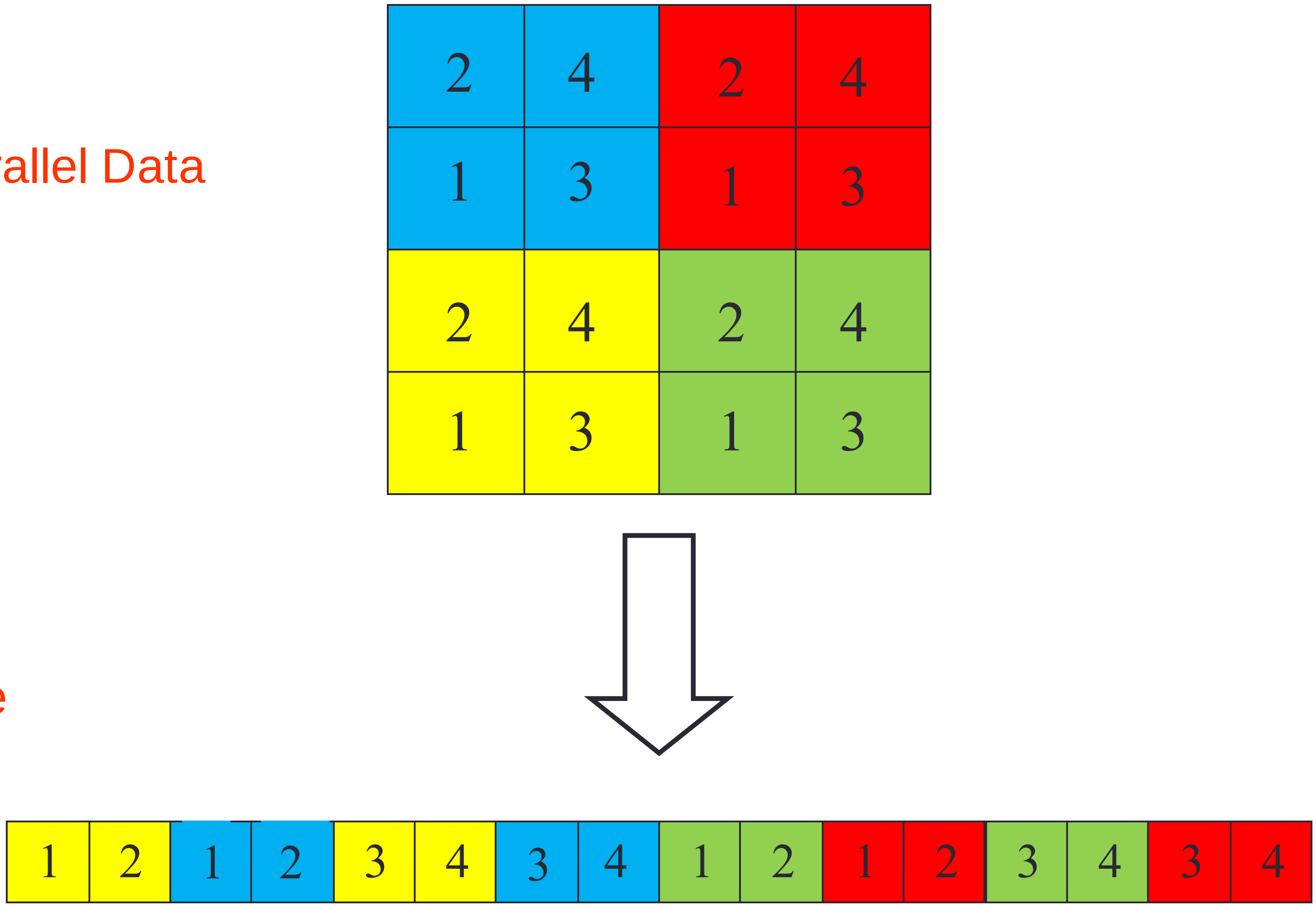
Parallel File Systems

- Allow multiple IO processes to access same file
  - increases bandwidth
- Typically optimised for bandwidth
  - not for latency
  - e.g. reading/writing small amounts of data is very inefficient
- Very difficult for general user to configure and use
  - need some kind of higher-level abstraction
  - allow user to focus on data layout across user processes
  - don’t want to worry about how file is split across IO servers

4x4 array on 2x2 Process Grid

Parallel Data

File



Message Passing: Naive Solutions

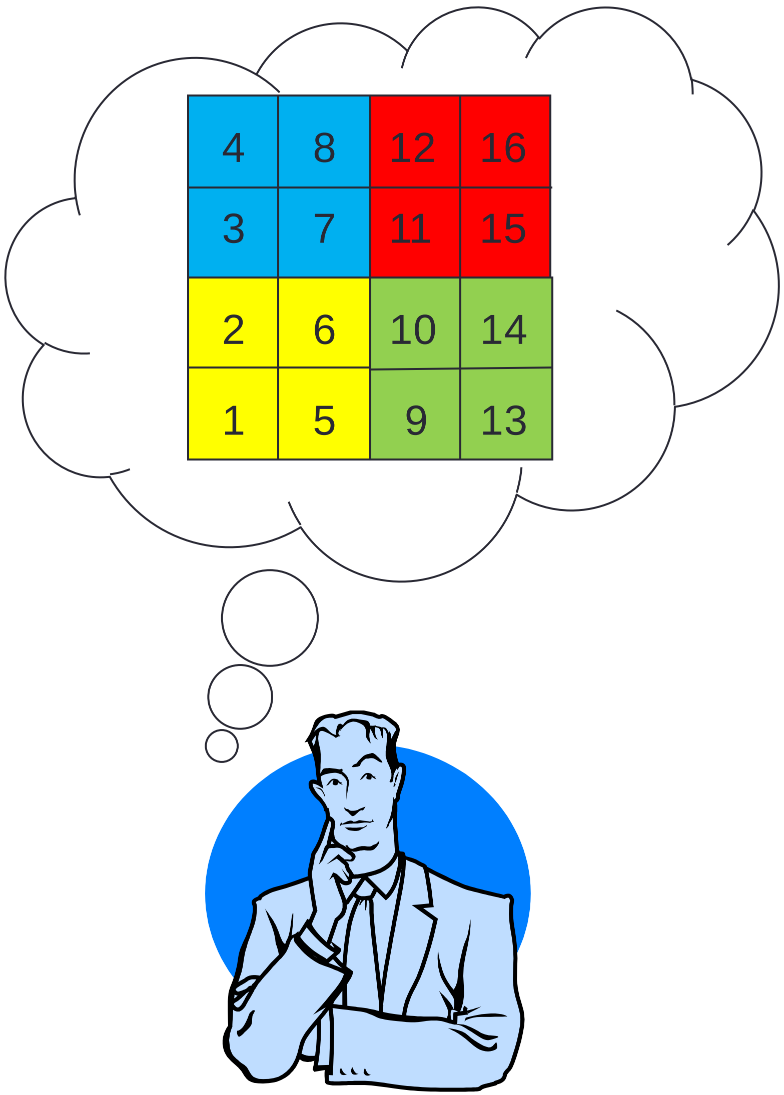
- Controller IO
  - send all data to/from single controller process and write/read a single file
  - quickly run out of memory on the controller
    - or have to write in many small chunks
  - does not benefit from a parallel fs that supports multiple write streams

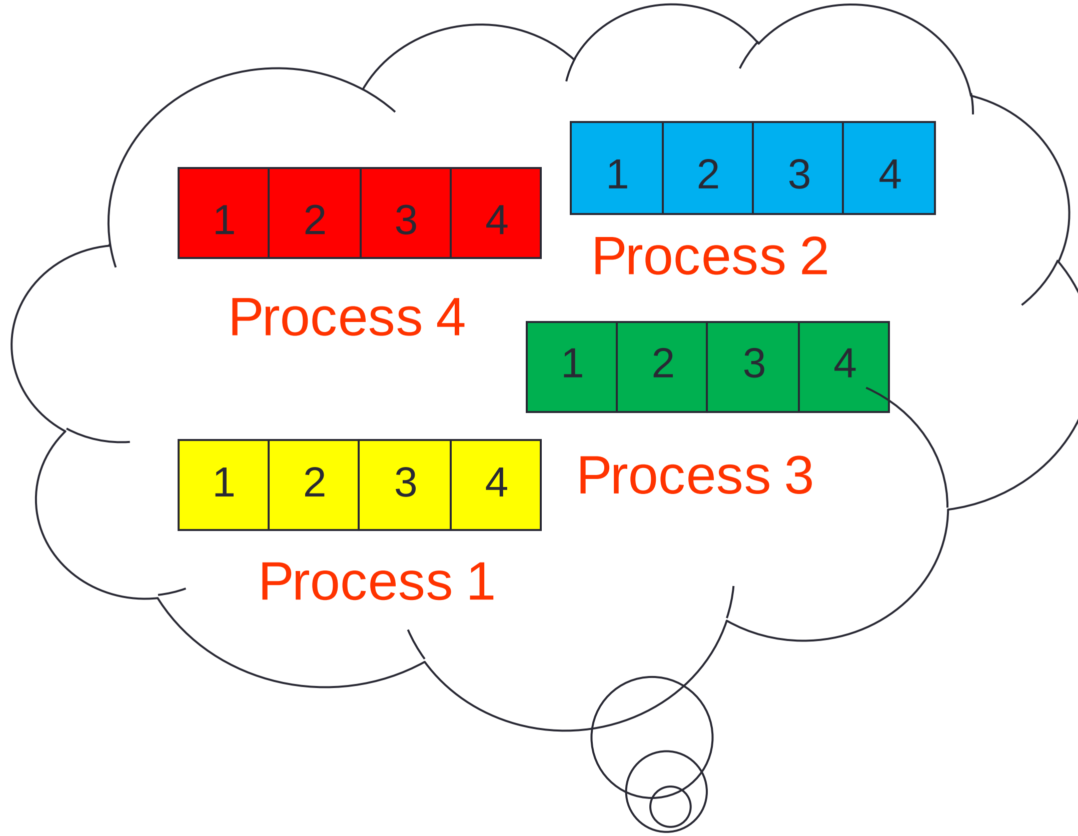
- Separate files
  - each process writes to a local fs and user copies back to home
  - or each process opens a unique file (dataXXX.dat) on shared fs

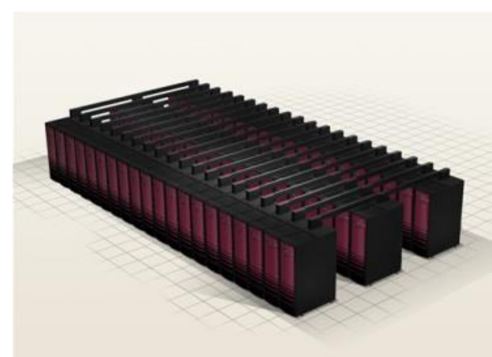
- Major problem with separate files is reassembling data
  - file contents dependent on number of CPUs and decomposition
  - pre / post-processing steps needed to change number of processes
  - but at least this approach means that reads and writes are in parallel

- But may overload filesystem for many processes
  - e.g. MDS cannot keep up with requests

Programmer View vs Machine View







epcc

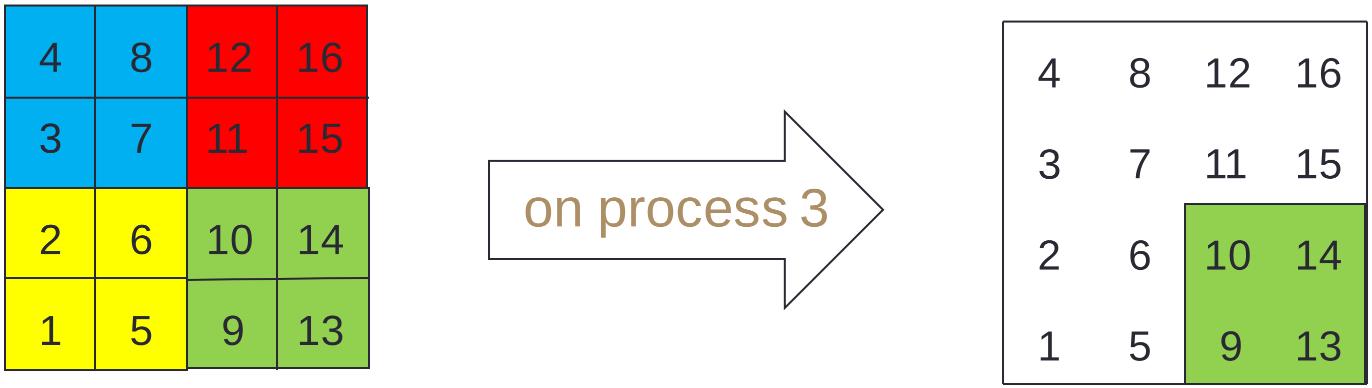
MPI-IO Approach

- MPI-IO is part of the MPI standard
  - http://www.mpi-forum.org/docs/docs.html

- Each process needs to describe what subsection of the global array it holds
  - it is entirely up to the programmer to ensure that these do not overlap for write operations!

- Programmer needs to be able to pass system-specific information
  - pass an `info` object to all calls

Data Sections



- Describe 2x2 subsection of 4x4 array
- Using standard MPI derived datatypes
- A number of different ways to do this
  - we will cover three methods in the course

Other Parallel IO Libraries

- MPI-IO is usually the lowest level
  - you may never call it directly
- Higher level libraries exit
  - HDF5
  - NetCDF
  - ADIOS
- Approach is the same
  - some way of describing what portion(s) of file each process owns
  - these libraries usually use MPI-IO to do their reads and writes

Summary

- Parallel IO is difficult
  - in theory and in practice
- MPI-IO provides a higher-level abstraction
  - user describes global data layout using derived datatypes
  - MPI-IO hides all the system specific filesystem details …
  - … but (hopefully) takes advantage of them for performance
- More flexible formats like NetCDF and HDF5 exist
  - they gain performance by layering on top of MPI-IO
- User requires a good understanding of derived datatypes
  - see next lecture