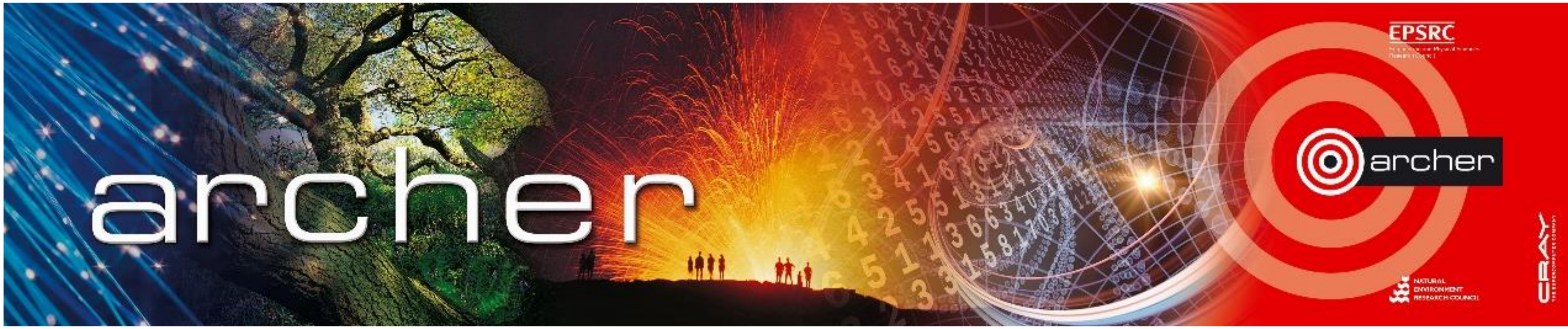



# Welcome!

## Virtual tutorial starts at 15:00 BST

# Parallel IO and the ARCHER Filesystem

ARCHER Virtual Tutorial, Wed 8th Oct 2014

David Henty <d.henty@epcc.ed.ac.uk>

# Reusing this material

# Overview

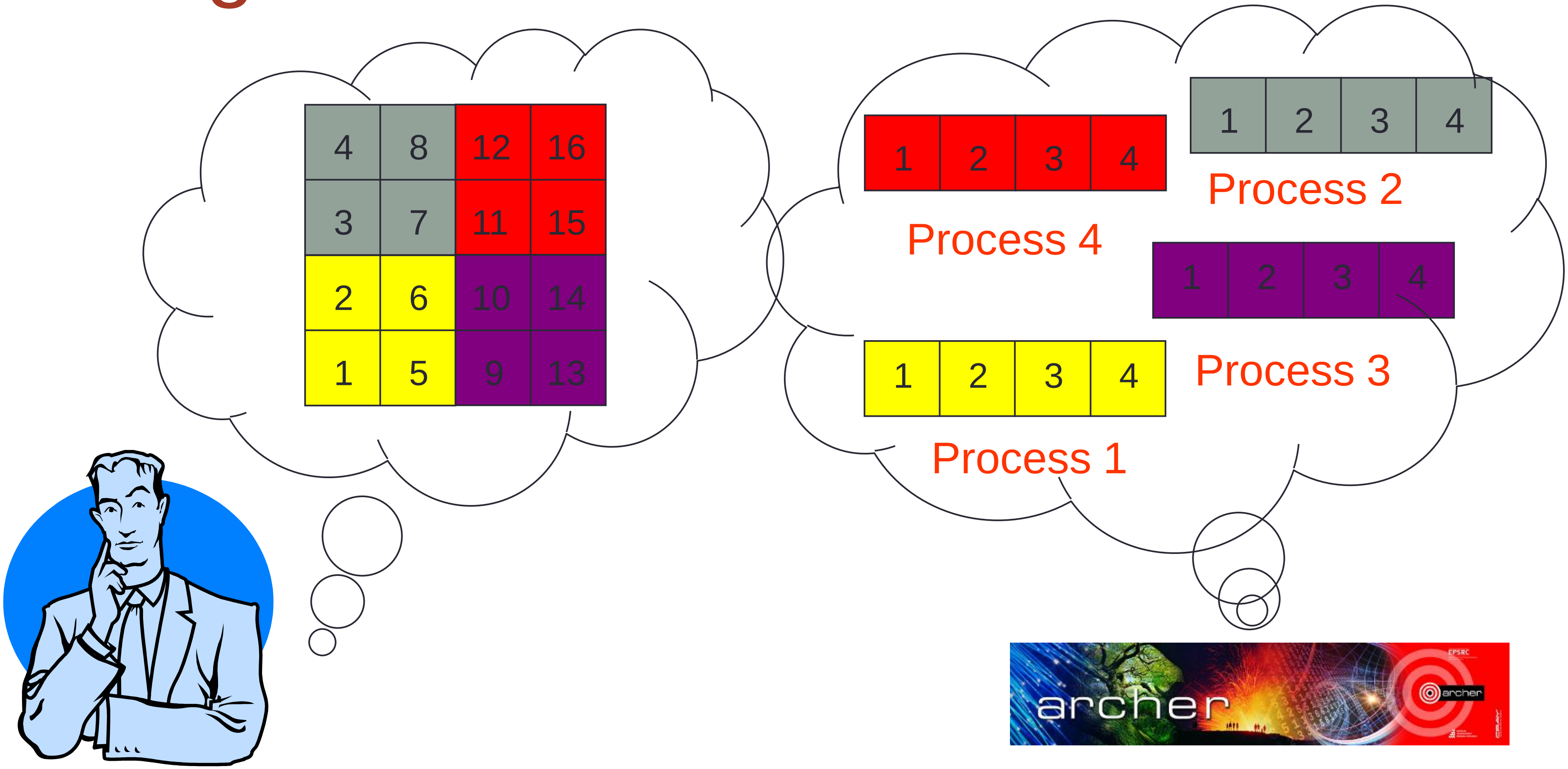
- Why parallel IO is difficult
- The Lustre file system
- Standard parallel IO strategies
- MPI-IO
- Tuning Lustre

# Why is Parallel IO Difficult?

- Difficult in principle
  - combine distributed data into a single location
  - data access patterns surprisingly complicated

- Difficult in practice
  - individual disk IO speeds are not very fast
  - file systems are complicated
  - parallel file systems are even more complicated
  - IO performance achieved by using multiple disks at once

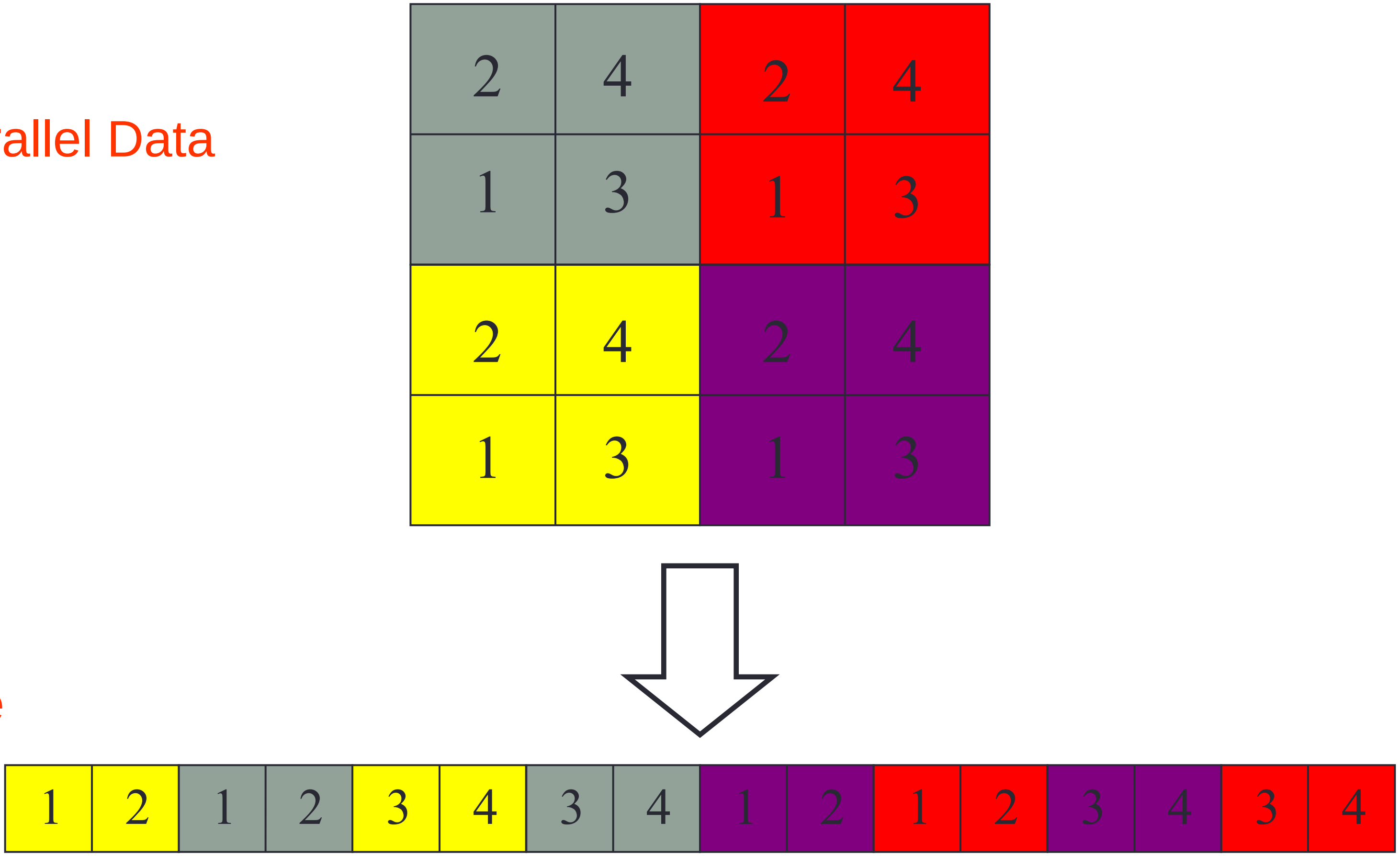
# Programmer View vs Machine View



# 4x4 array on 2x2 Process Grid

Parallel Data

File



# ARCHER’s Lustre – Cray Sonexion Storage

1

## MMU: *Metadata Management Unit*

**Lustre MetaData Server**

- Contains server hardware and storage

2

## SSU: *Scalable Storage Unit*

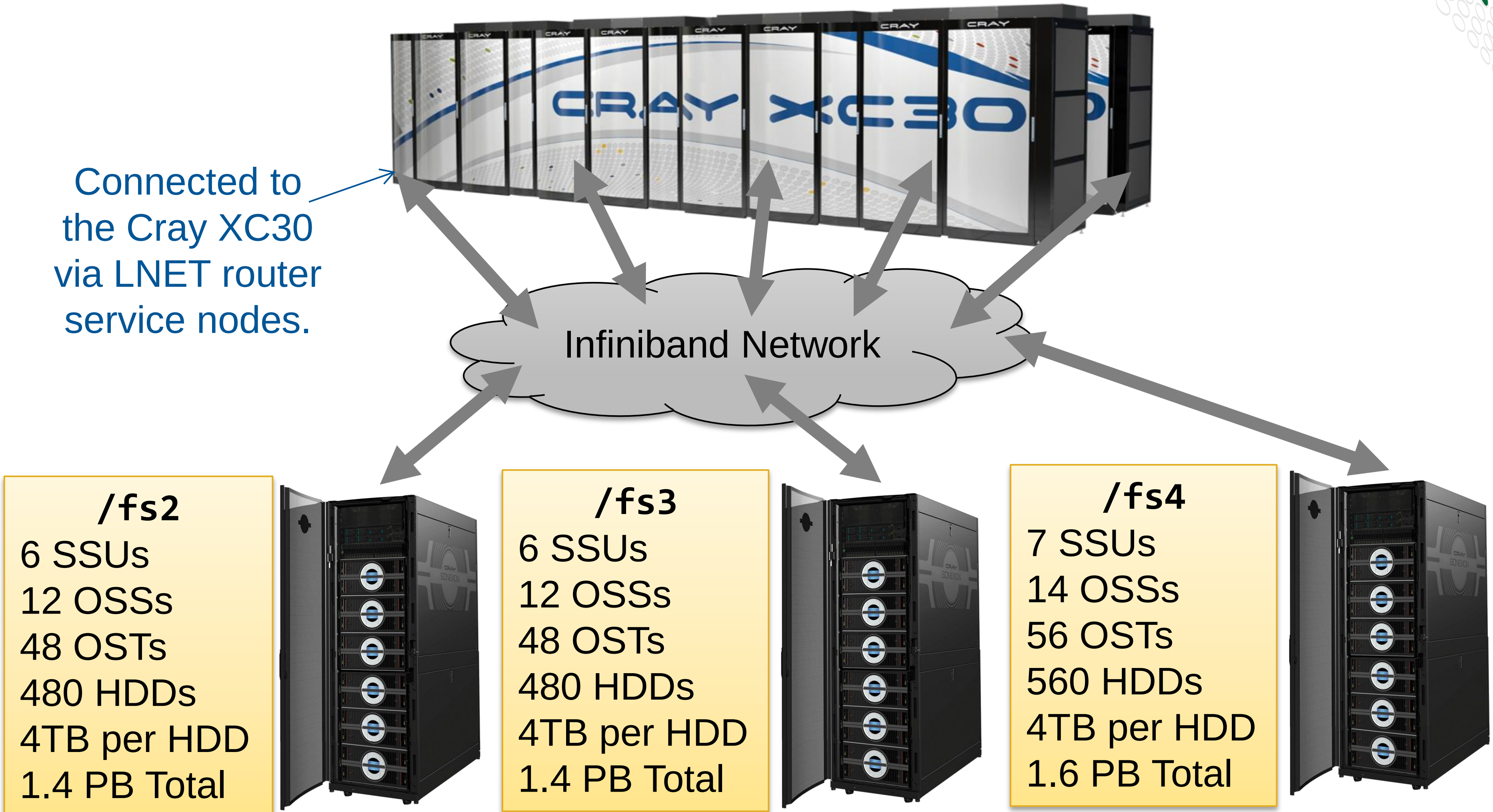
**2 x OSSs and 8 x OSTs (Object Storage Targets)**

- 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

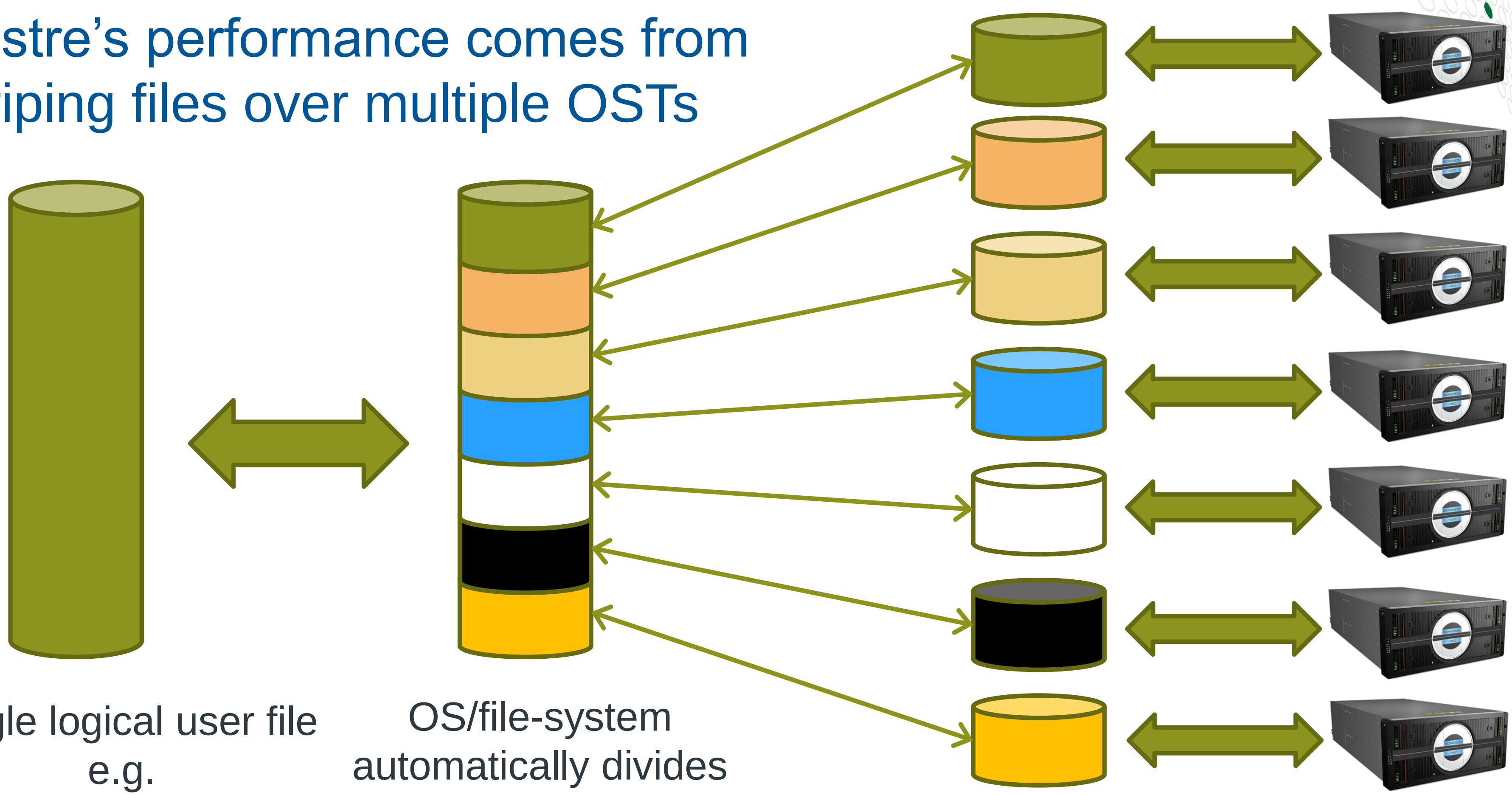

**Multiple SSUs are combined to form storage racks**

# ARCHER's File systems



# Lustre data striping

## Lustre's performance comes from striping files over multiple OSTs

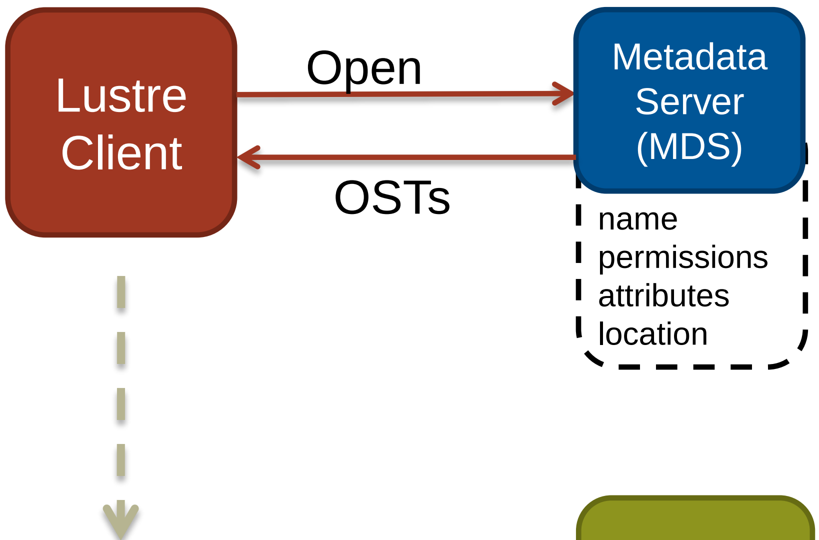


Single logical user file
e.g.
`/work/y02/y02/ted`

OS/file-system automatically divides the file into stripes

Stripes are then read/written to/from their assigned OST

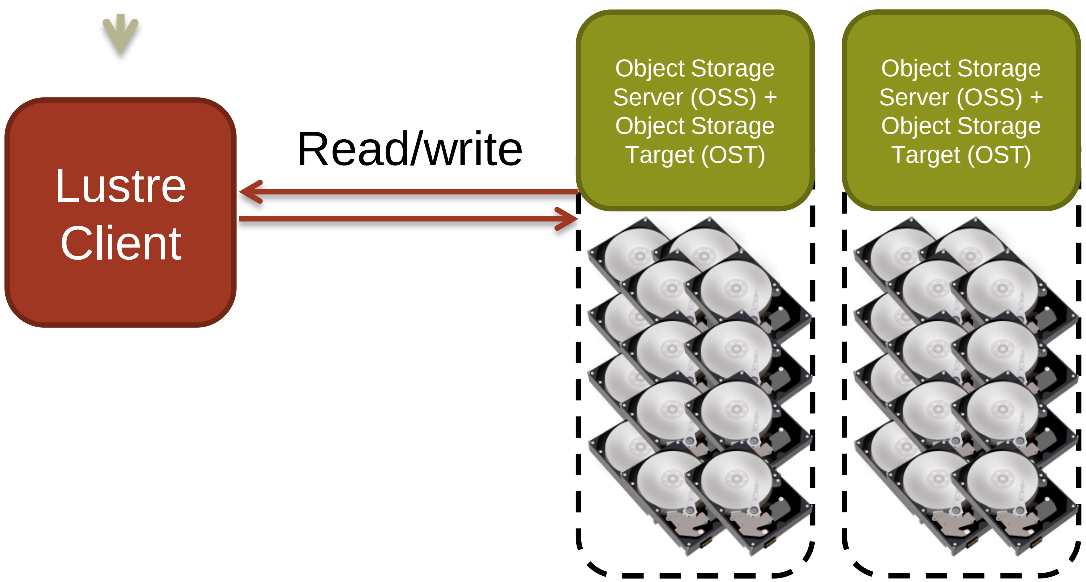
# Opening a file




The client sends a request to the MDS to opening/acquiring information about the file

The MDS then passes back a list of OSTs

- For an existing file, these contain the data stripes
- For a new files, these typically contain a randomly assigned list of OSTs where data is to be stored




Once a file has been opened no further communication is required between the client and the MDS

All transfer is directly between the assigned OSTs and the client

# Summary

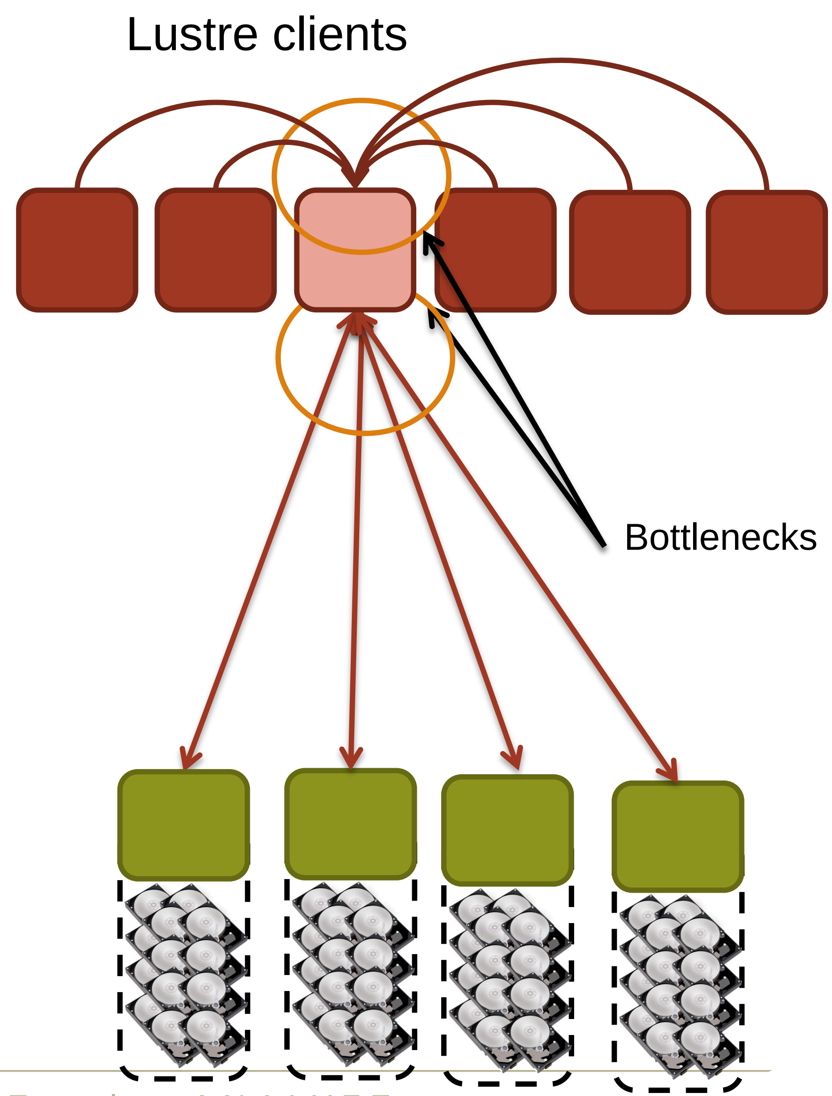
- Lustre achieves high bandwidth via multiple disks
- Single file can be striped across multiple disks
  - allows simultaneous IO from multiple Object Storage Targets
  - think of each OST as a separate IO path to disk

- Optimised for large transactions
  - “please write 100 Mb to disk”

- Meta Data Server can be a bottleneck
  - opening and closing files is serialised and can be slow
  - *not* optimised for large numbers of small files

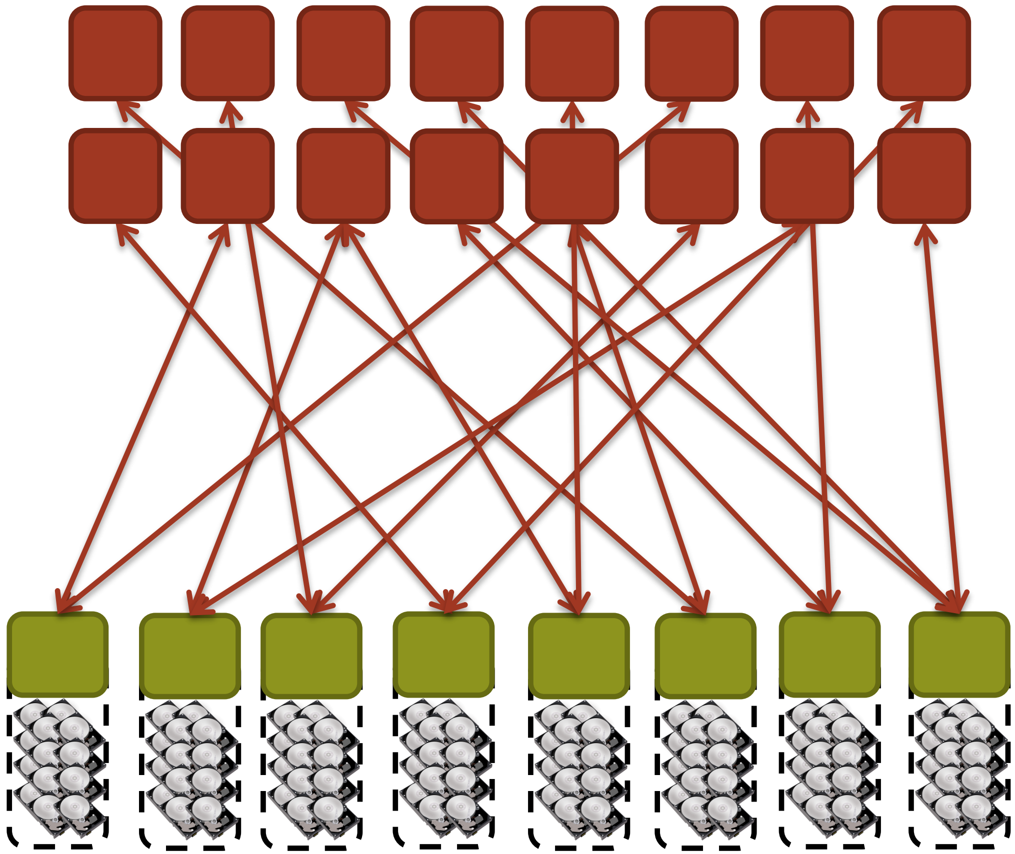
# I/O strategies: Spokesperson (master/serial IO)

- **One process performs I/O**
  - Data Aggregation or Duplication
  - Limited by single I/O process
- **Easy to program**
- **Pattern does not scale**
  - Time increases linearly with amount of data
  - Time increases with number of processes
- **Care has to be taken when doing the all-to-one kind of communication at scale**
- **Can be used for a dedicated I/O Server**

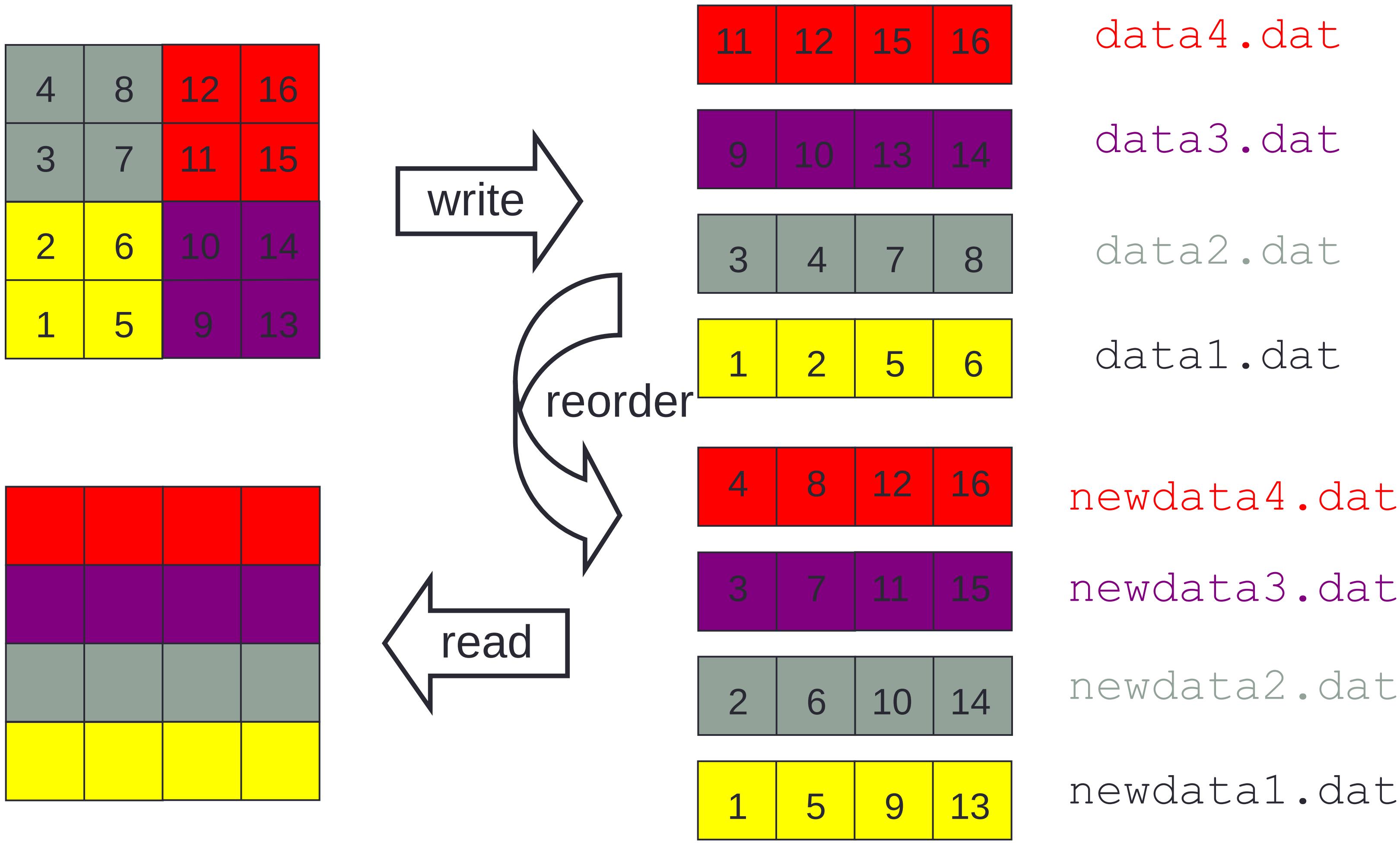


# I/O strategies: Multiple Writers – Multiple Files

- **All processes perform I/O to individual files**
  - Limited by file system
- **Easy to program**
- **Pattern may not scale at large process counts**
  - Number of files creates bottleneck with metadata operations
  - Number of simultaneous disk accesses creates contention for file system resources

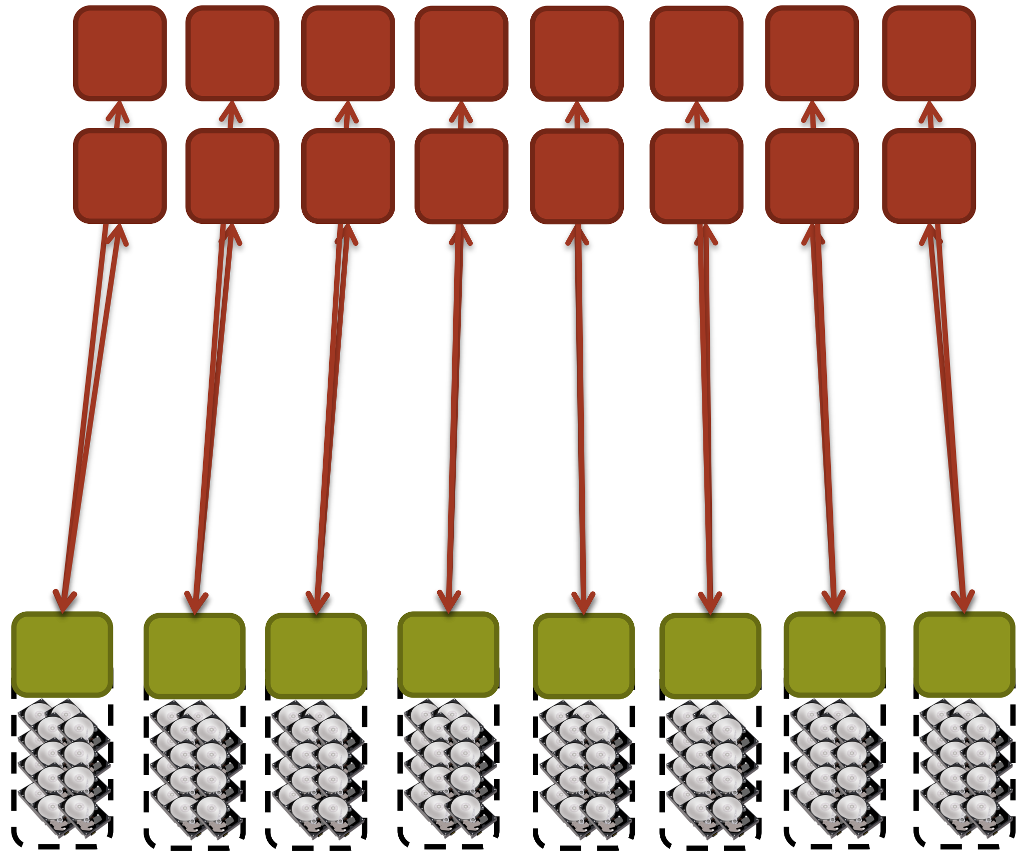


# 2x2 to 1x4 Redistribution



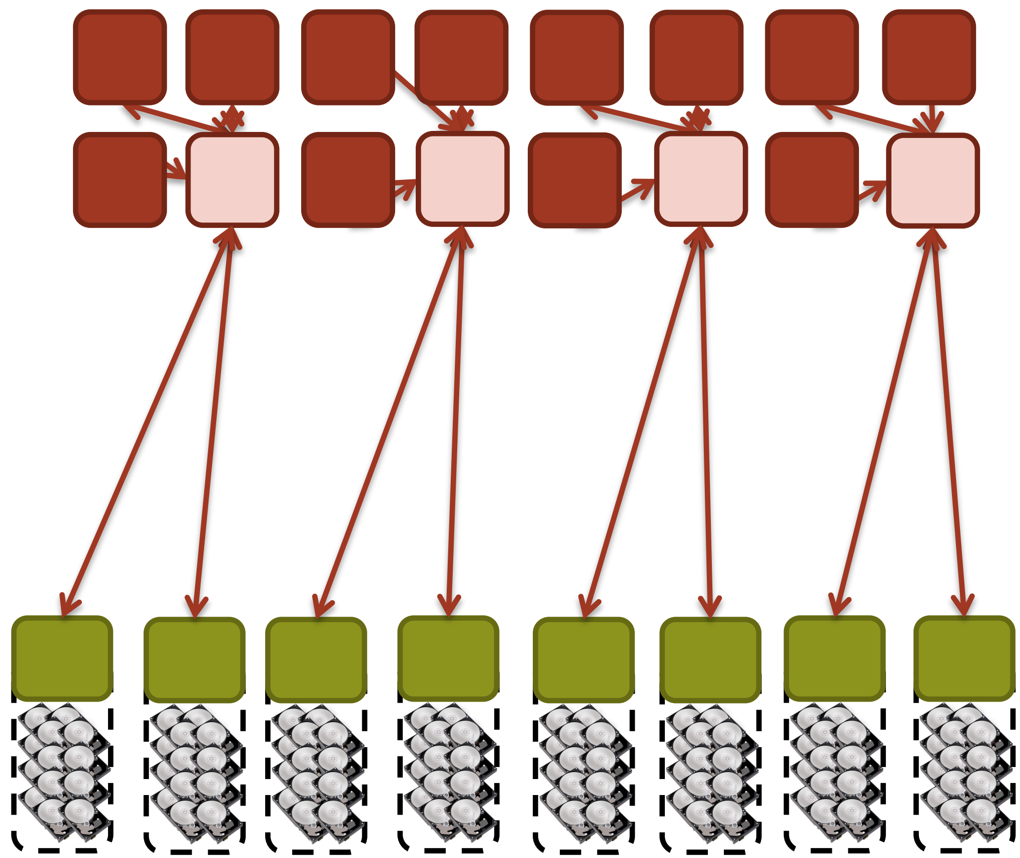
# I/O strategies: Multiple Writers – Single File

- **Each process performs I/O to a single file which is shared.**
- **Performance**
  - Data layout within the shared file is very important.
  - At large process counts contention can build for file system resources.
- **Not all programming languages support it**
  - C/C++ can work with fseek
  - No real Fortran standard



# I/O strategies: Collective IO to single or multiple files

- **Aggregation to a processor in a group which processes the data.**
  - Serializes I/O in group.
- **I/O process may access independent files.**
  - Limits the number of files accessed.
- **Group of processes perform parallel I/O to a shared file.**
  - Increases the number of shares to increase file system usage.
  - Decreases number of processes which access a shared file to decrease file system contention.



# Summary

- Need subgroups of IO processes to do IO simultaneously
  - too many and there is contention for file system resources
  - too few and we do not use all the OSTs
- Far too complicated to do this ourselves
  - need some library to help us out
- For example, MPI-IO
  - part of MPI standard since MPI-2.0

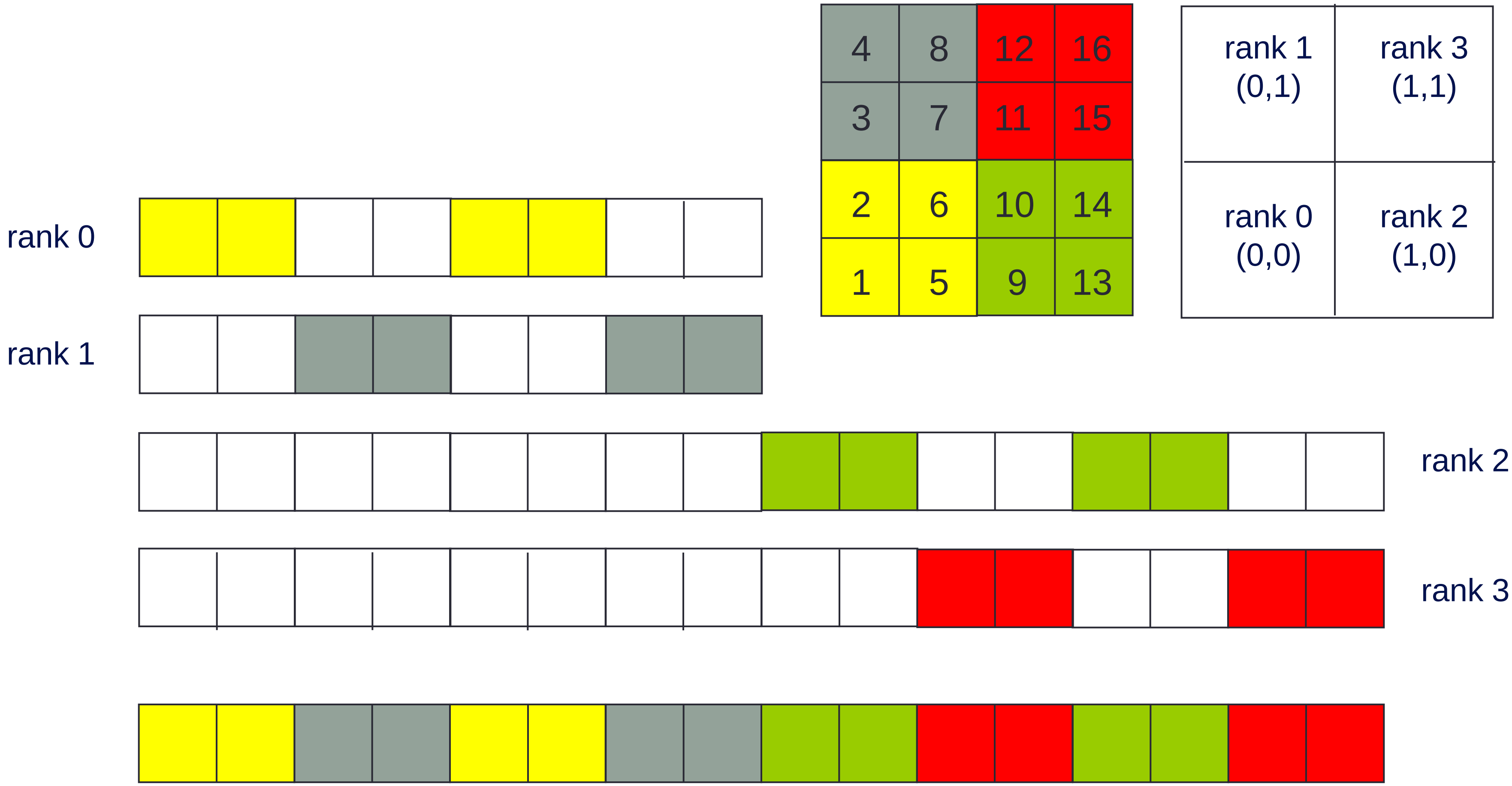
# MPI-IO Approach

- Each process / rank tells MPI-IO what portion(s) of the file it wants to read / write
  - uses MPI Derived Datatypes
  - this is called the *File View*

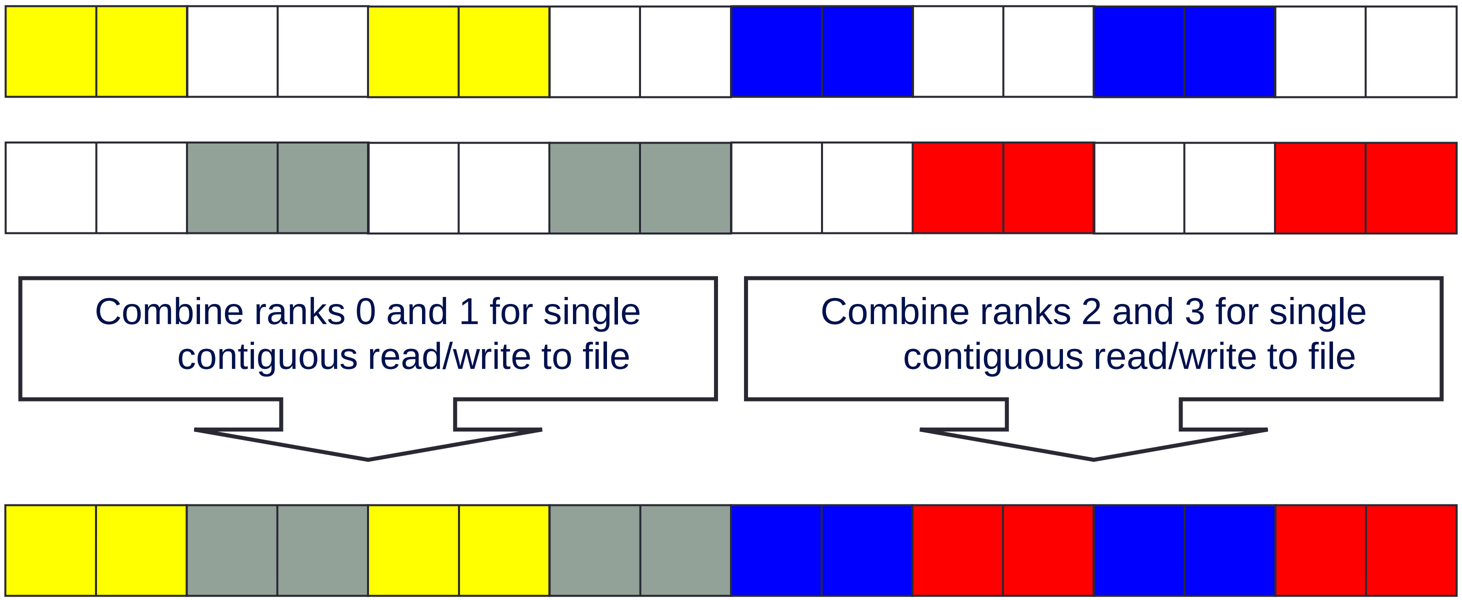
- Tell MPI-IO what data to write
  - automatically goes to the position(s) selected by the file view
  - all the communications / buffering / aggregation handled by MPI-IO

- Allows for collective IO
  - MPI-IO has a global view
  - can aggregate data for a small number of large IO transactions

# Combining File Views



# Collective IO



# MPI-IO on ARCHER

- Optimised for the Lustre file system
- Scales the number of IO processes appropriate to the number of OSTs and the total number of processes
- But ...
  - the striping of a file (the number of OSTs) is set by the user
  - important to get this right

# Lustre Striping

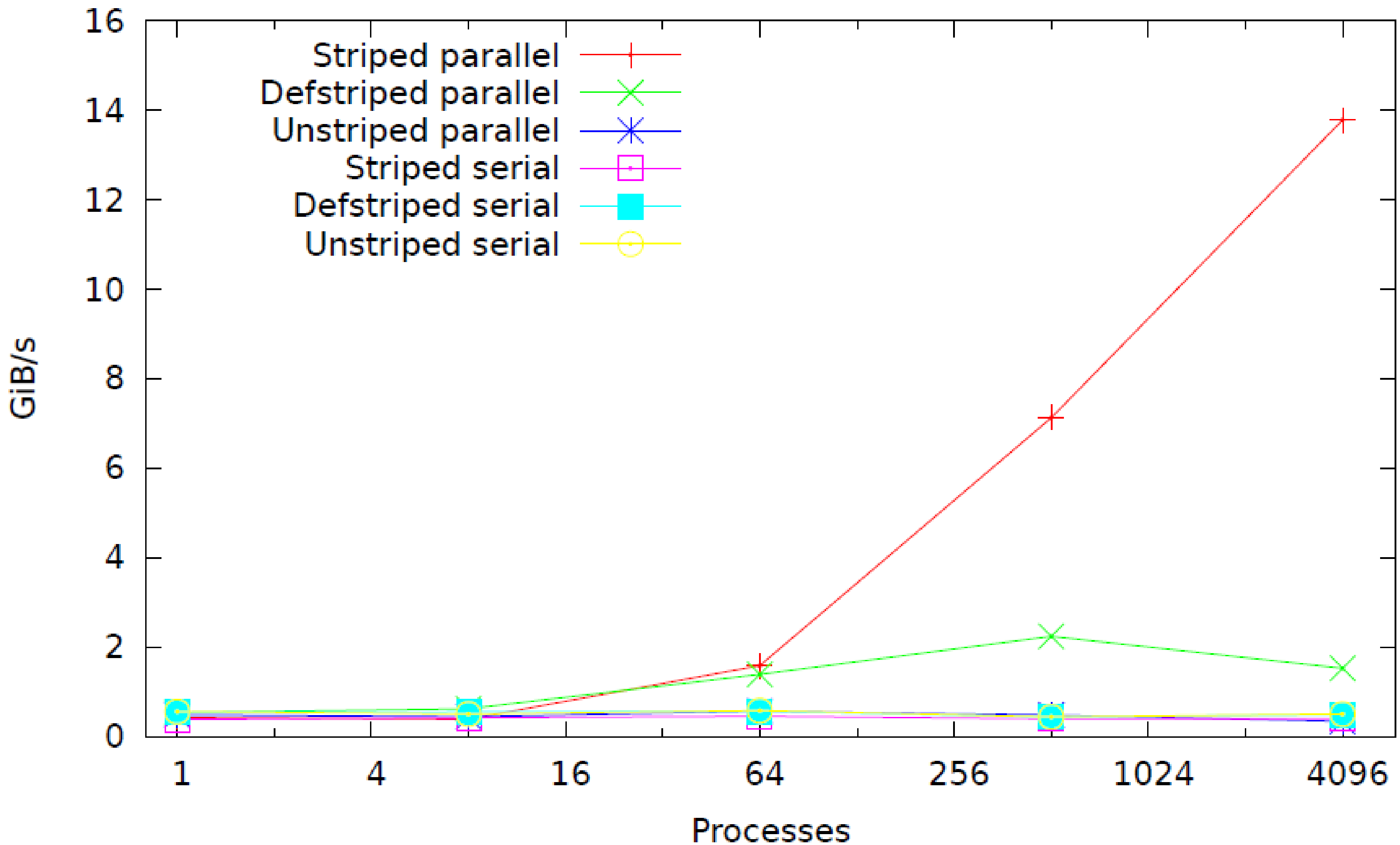
- Essential to stripe large files across multiple disks
  - but striping small files across many disks is bad

- Default striping on ARCHER is across 4 OSTs

- Can set this yourself using:
  - lfs setstripe -c <nstripe> <directory>
  - to use all the OSTs: nstripe = -1
  - to enquire: lfs getstripe <directory>

- Test case: large 3D dataset across 3D process grid
  - IO done using MPI-IO

# $128^3$ per proc: 16 MiB to 64 GiB



# Summary

- Master IO unaffected by striping
- Same bandwidth as parallel IO with no striping
- Around 400 MiB/s independent of process count

- With parallel IO and striping
  - bandwidth scales with process count (until all OSTs are used)

  - achieve around 2 GiB/s for default striping (4 OSTs)
  - 10's of GiB/s for full striping (all OSTs, nstripe = -1)

# Collective vs Independent IO

- Replace **`MPI_File_write_all`** with **`MPI_File_write`**
  - identical functionality
  - different performance

- Results with full striping

| Processes | Individual | Collective |
|---|---|---|
| 1 | 49.5 MiB/s | 441 MiB/s |
| 8 | 5.9 MiB/s | 404 MiB/s |
| 64 | 2.4 MiB/s | 1630 MiB/s |

# Conclusions

- Good IO requires three things to be true

- A sensible number of IO processes
  - not a single process, not all processes
  - MPI-IO does this for you

- File striped across multiple disks
  - in Lustre, use multiple OSTs via: lfs setstripe

- Collective IO
  - IO processes aggregate data: small number of large IO operations

# How good is my IO?

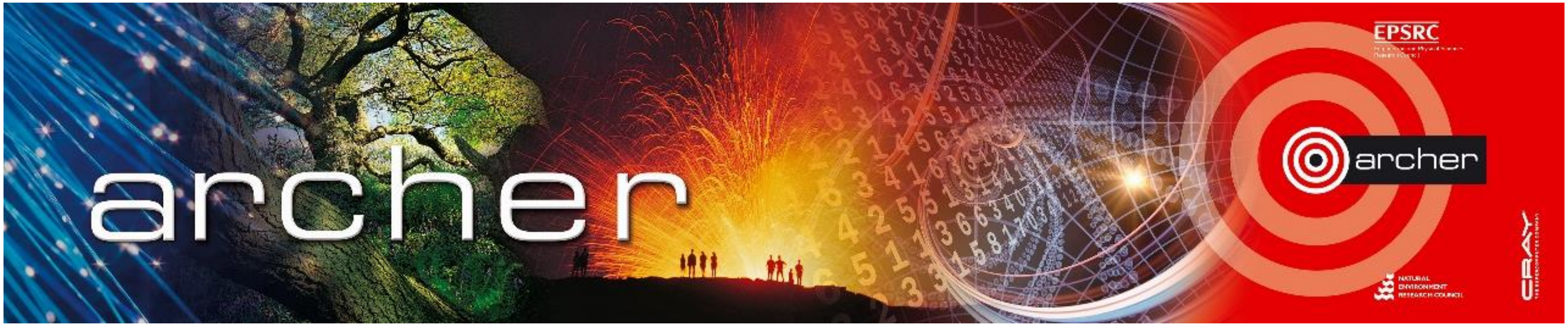
- Essential to quantify in terms of GiB/s
  - look at the size of your files
  - time the IO operations
  - hundreds of MiB/s: bad
  - tens of GiB/s: good
- Performance tools may help here
  - eg Cray performance tools can report IO rates

# Other libraries

- What if I use NetCDF / HDF5 / ... ?
- Use a version that layers on top of MPI-IO
- Ensure that it is doing collective IO
- Set the Lustre striping for the files

# Help with IO

- Contact the CSE support team!
- email: support@archer.ac.uk
- Working on a white paper on parallel IO
  - plan to have first version available in November




# Goodbye!

# Thanks for attending