Hewlett Packard Enterprise

Optimizing Large Scale I/O

LUMI Advanced Workshop

March 5–7, 2025

Agenda

- Lustre Introduction
- Common application I/O patterns
- Tuning Lustre Settings
- Being nice to Lustre
- Asynchronous I/O
- MPI-IO internals and controls

Lustre

A parallel filesystem

Lustre

The Lustre® file system is an open-source, parallel file system that supports many requirements of leadership class HPC simulation environments
(from https://www.lustre.org/)

- A scalable cluster file system for Linux
  - Developed by Cluster File Systems -> Sun -> Oracle.
  - Name derives from “Linux Cluster”
  - The Lustre file system consists of software subsystems, storage, and an associated network

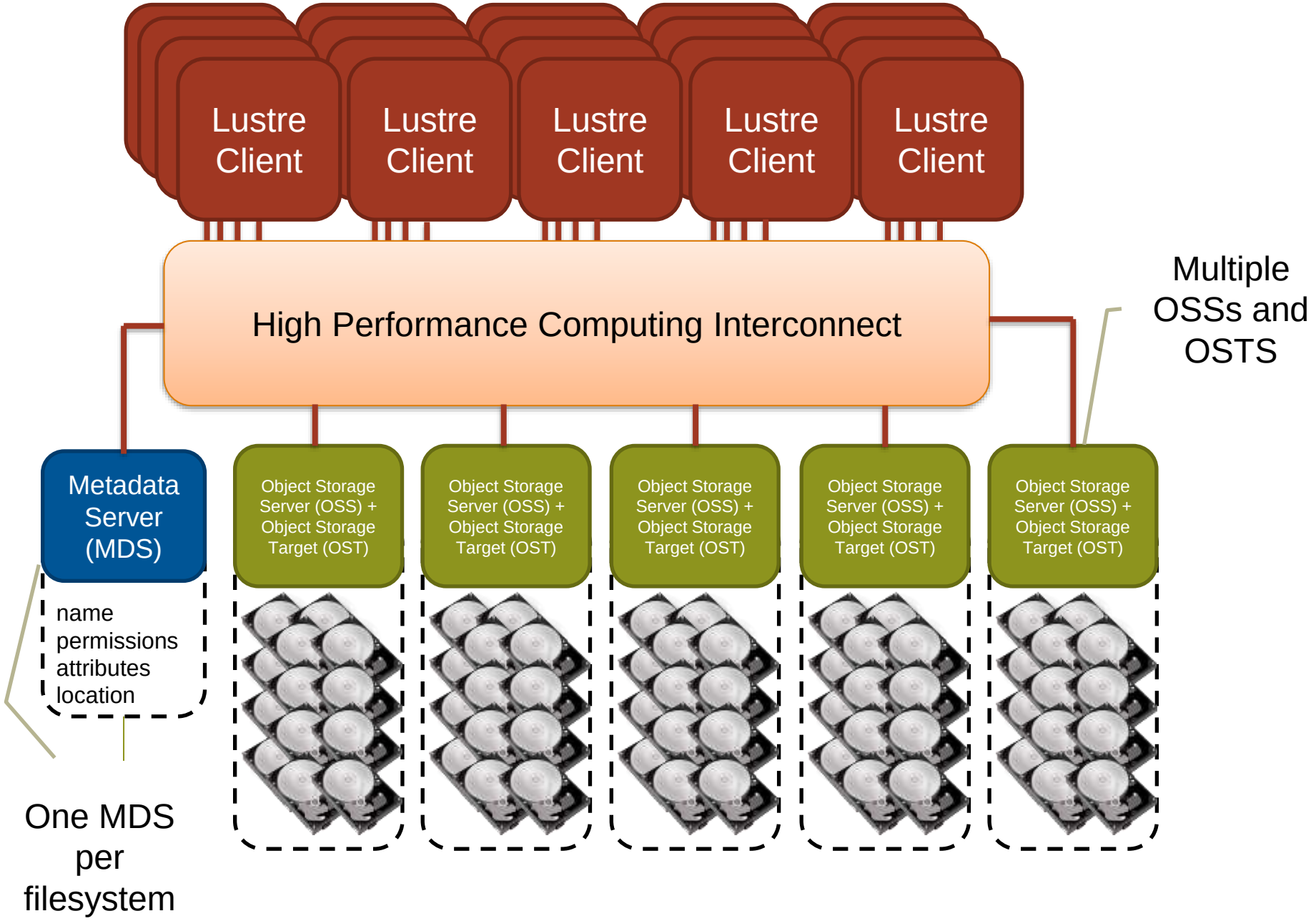
Lustre Building Blocks

A Lustre file system has three major functional units:

- One or more metadata servers (MDSes) that has one or more metadata targets (MDTs) per Lustre filesystem that stores namespace metadata, such as filenames, directories, access permissions, and file layout.

- One or more object storage servers (OSSes) that store file data on one or more object storage targets (OSTs).
  Depending on the server's hardware, an OSS typically serves between two and eight OSTs, with each OST managing a single local disk filesystem. The capacity of a Lustre file system is the sum of the capacities provided by the OSTs.

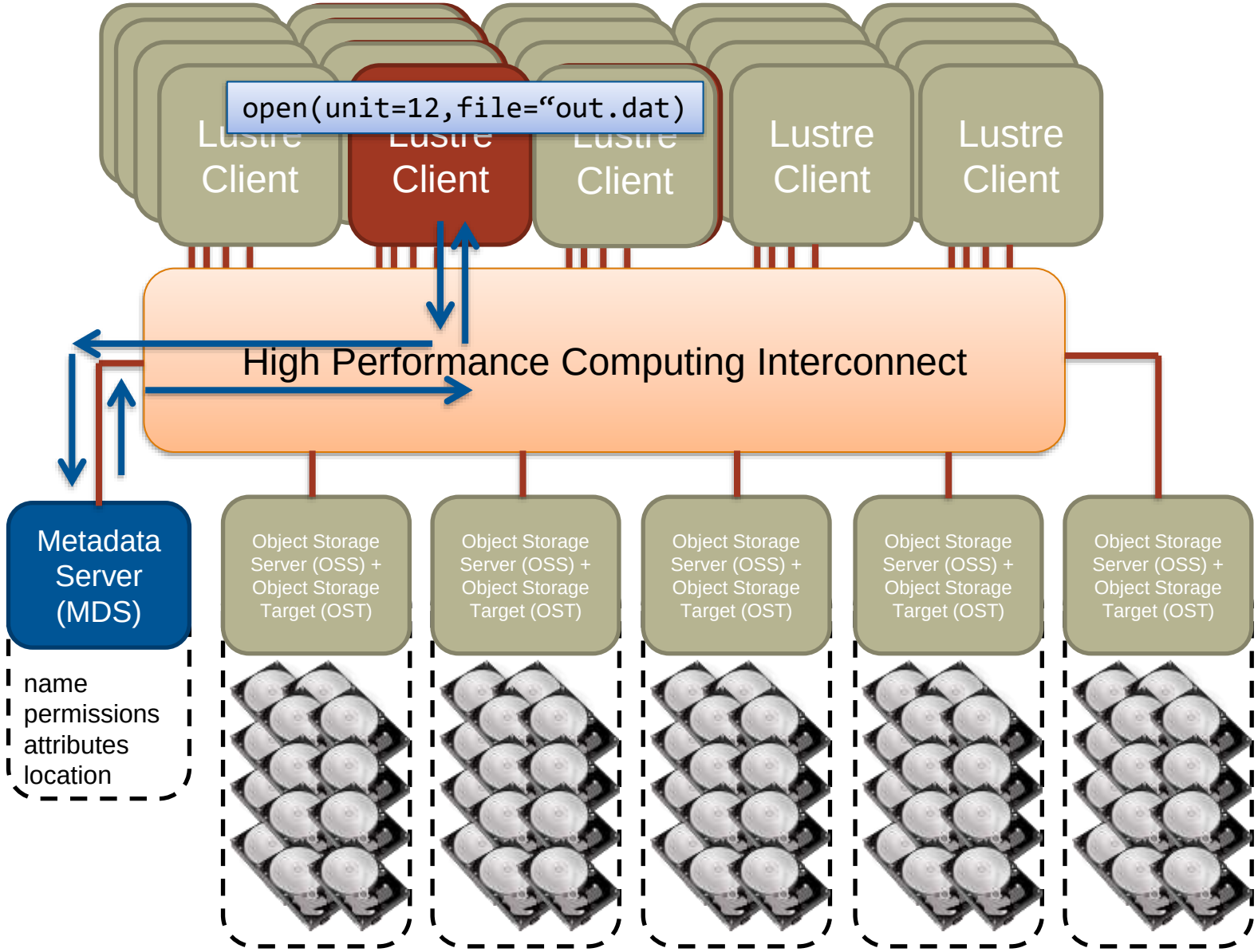
- Client(s) that access and use the data. Lustre presents all clients with a unified namespace for all of the files and data in the filesystem, using standard POSIX semantics, and allows concurrent and coherent read and write access to the files in the filesystem.

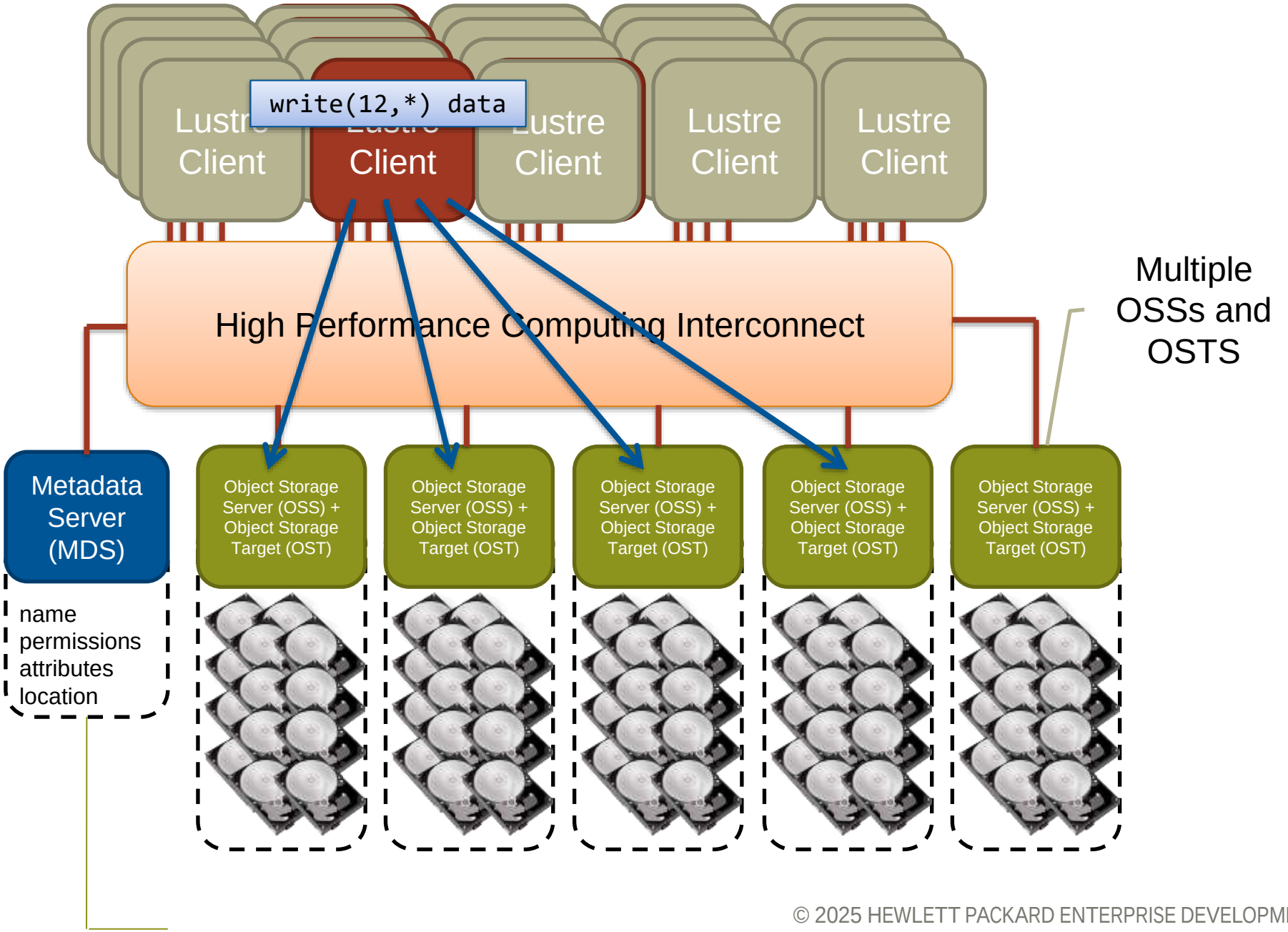


Raid Blocks vs Lustre Stripes

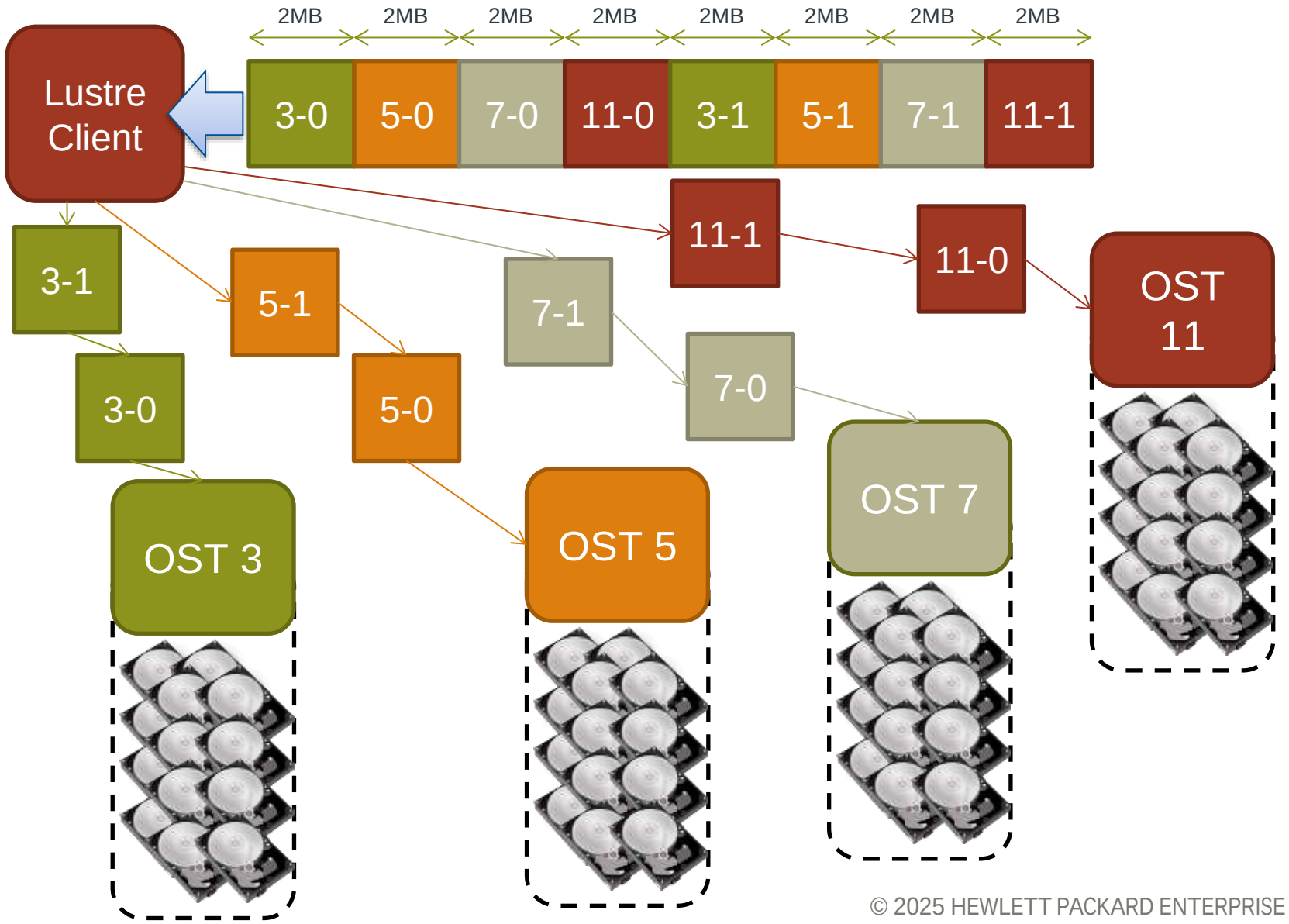
RAID blocks and Lustre stripes appear, at least on the surface, to perform the similar function, however there are some important differences.

- RAID on OSTs
  - Typically RAID-6 or GridRAID, block and stripe size chosen when device configured
- Lustre striping
  - Can be configured by user on file-by-file basis
  - Lustre stripe sizes normally between 1 and 32 MB





File Decomposition – 2 Megabyte Stripes

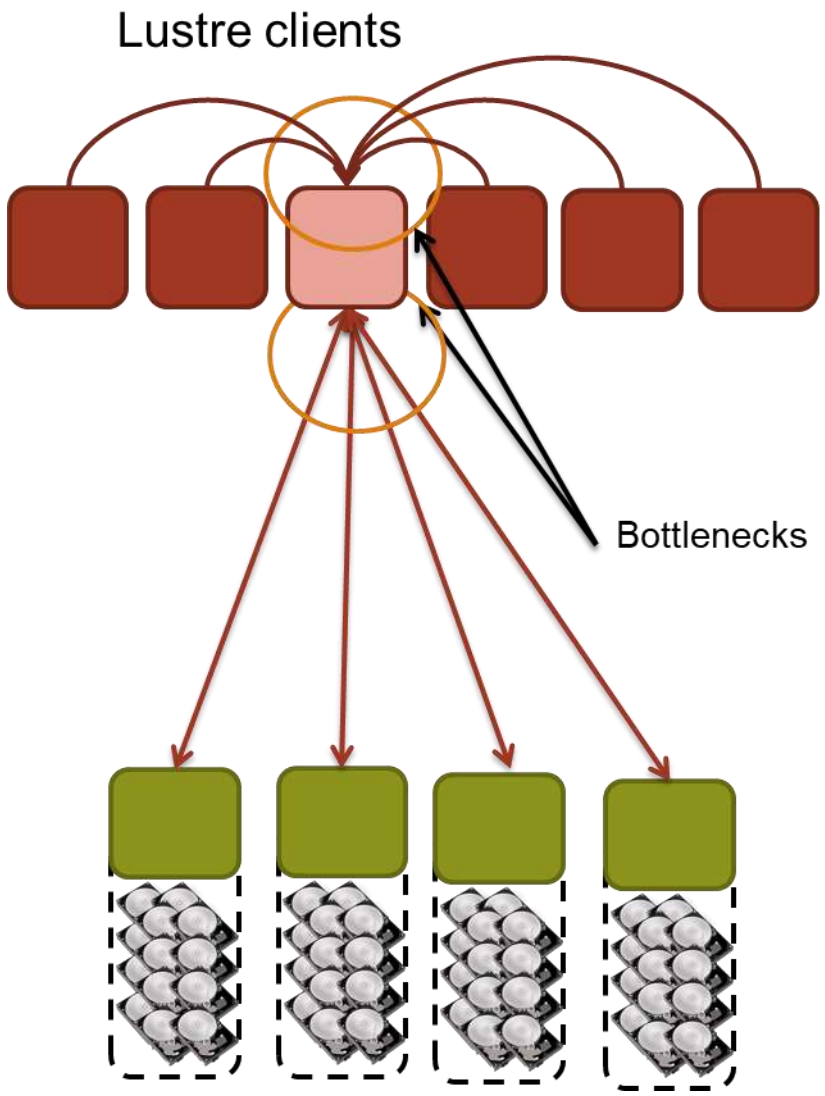


Key Points

- Lustre achieves high performance through parallelism
  - Best performance from multiple clients writing to multiple OSTs
- Lustre is designed to achieve high bandwidth to/from a small number of files
  - A typical use case is a scratch file system for HPC
  - It is a good match for scientific datasets and/or checkpoint data
- Lustre was not originally designed to handle large numbers of small files
  - Potential bottlenecks at the MDS when files are opened
  - New features have addressed this

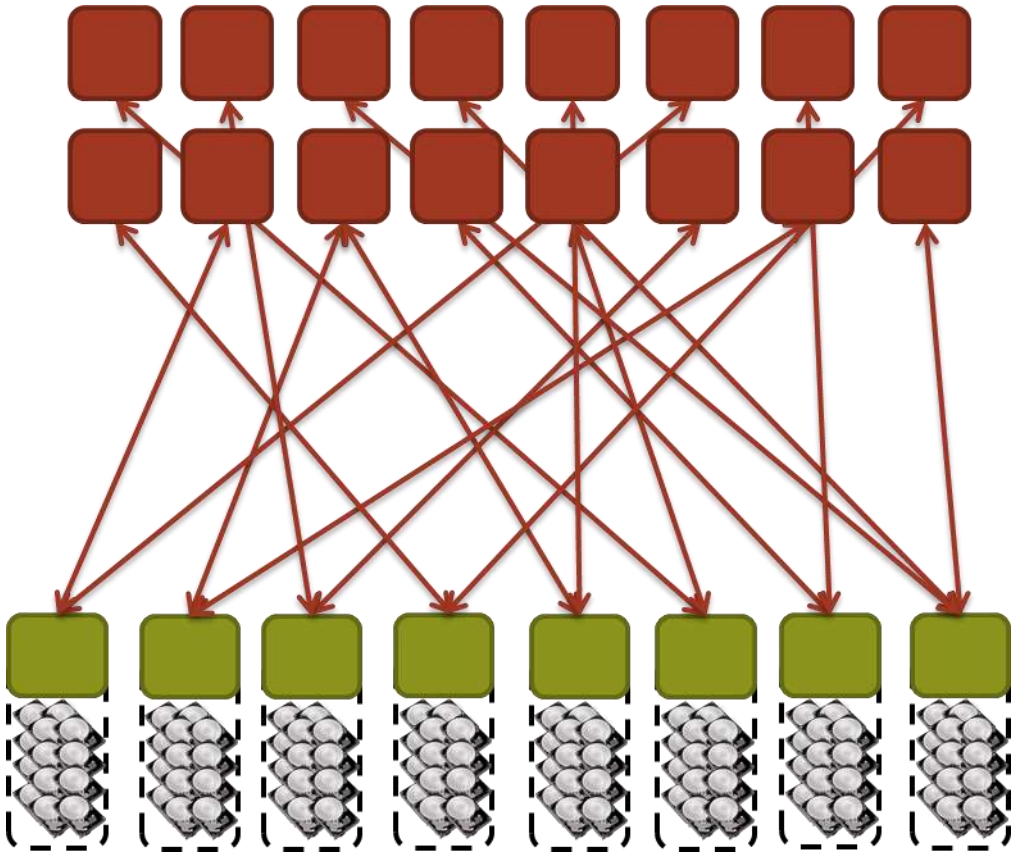
Common I/O Patterns Found in Applications

I/O Strategies: Spokesperson



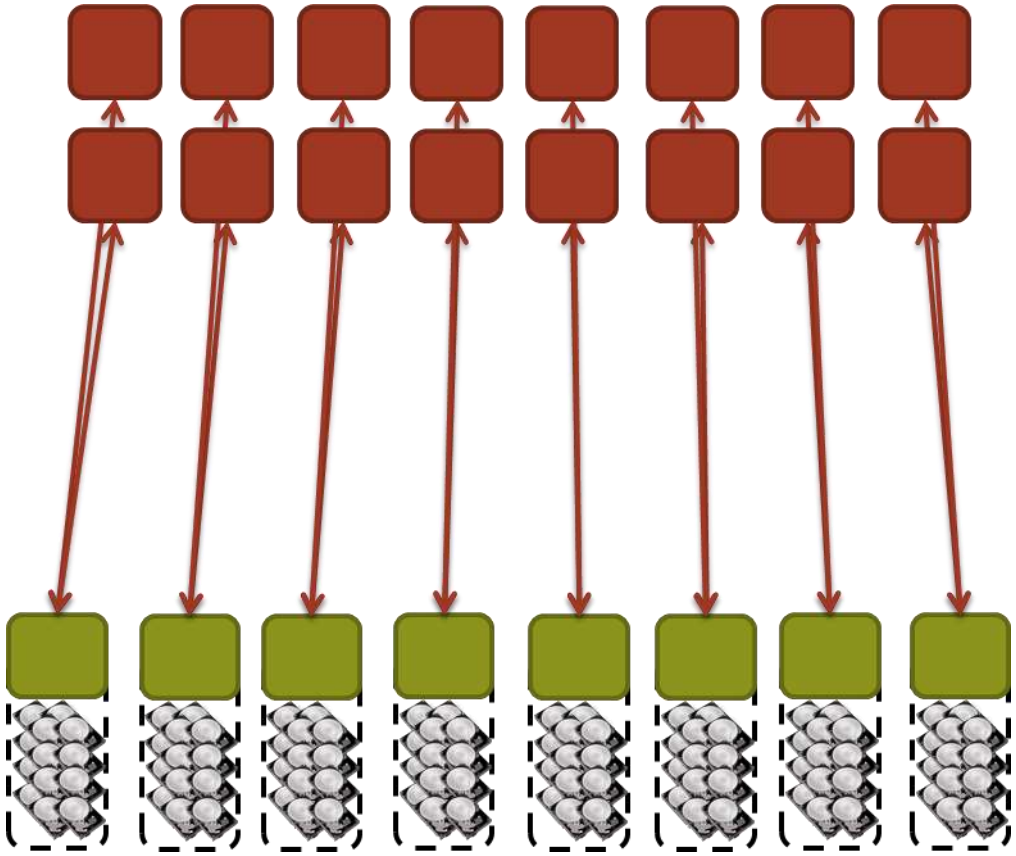
- One process performs I/O
  - Data Aggregation or Duplication
  - Limited by single I/O process
- Easy to program
- Pattern does not scale
  - Time increases linearly with amount of data
  - Time increases with number of processes
- Care must be taken when doing the all-to-one kind of communication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

I/O Strategies: Multiple Writers – Multiple Files



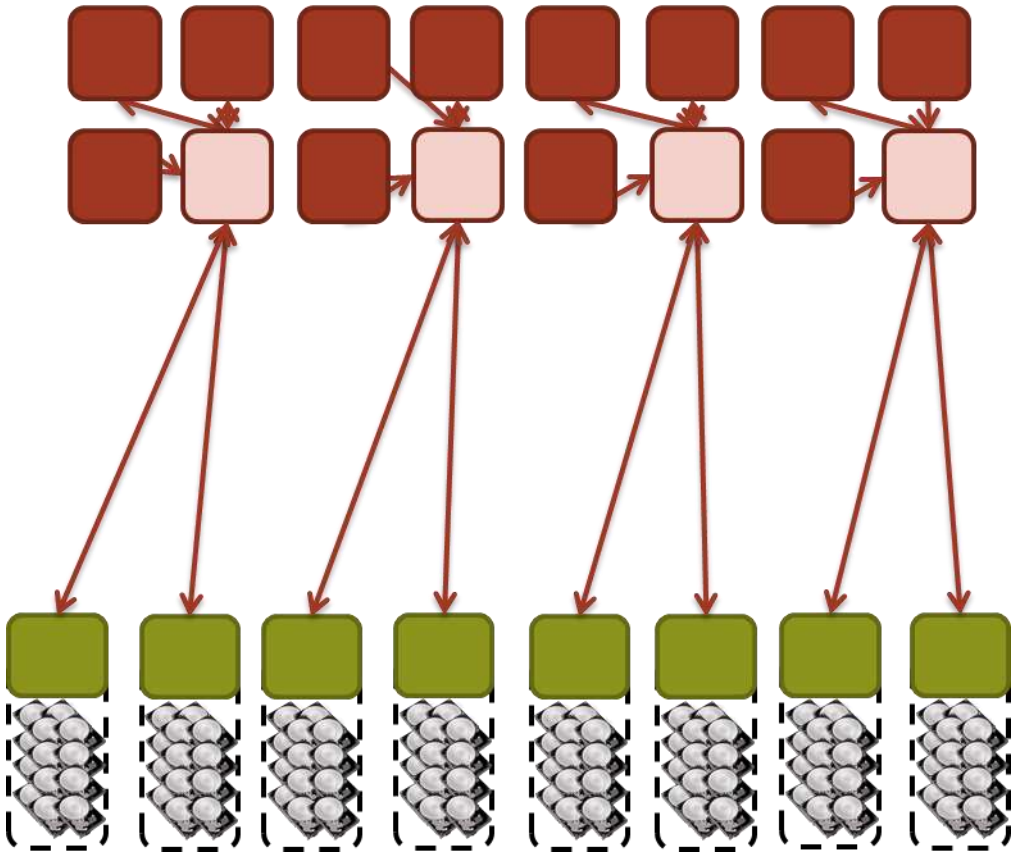
- All processes perform I/O to individual files
  - Limited by file system
- Easy to program
- Pattern may not scale at large process counts
  - Number of files creates bottleneck with metadata operations
  - Number of simultaneous disk accesses creates contention for file system resources

I/O Strategies: Multiple Writers – Single File



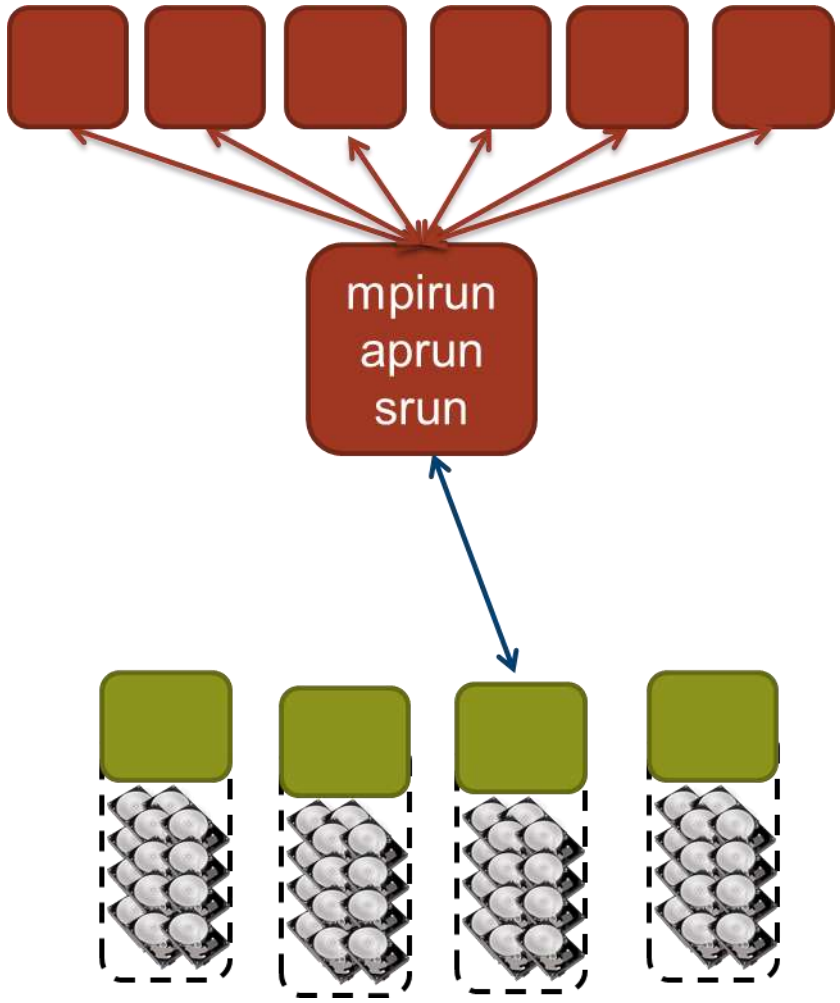
- Each process performs I/O to a single file which is shared.
- Performance
  - Data layout within the shared file is very important.
  - At large process counts contention can build for file system resources.
- Not all programming languages support it
  - C/C++ can work with fseek
  - No real Fortran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

Special Case: Standard Output and Error



- On most clusters/MPPs all STDIN, STDOUT, and STDERR I/O streams serialize through mpirun/aprun/srun
- Disable debugging messages when running in production mode.
  - "Hello, I'm task 32,000!"
  - "Task 64,000, made it through loop."

Tuning Lustre Settings

Matching Lustre Striping to an Application

Controlling Lustre Striping

- **`lfs`** is the Lustre utility for setting the stripe properties of new files, or displaying the striping patterns of existing ones

- The most used options are
  - `setstripe` – Set striping properties of a directory or new file
  - `getstripe` – Return information on current striping settings
  - `df` – Show disk usage of this file system

- For help execute lfs without any arguments

```
$ lfs
lfs > help
Available commands are:
        setstripe
        find
        getstripe
        check
      ...
```

Sample Lustre Commands: lfs df

```
% lfs df -h .
UUID                       bytes        Used   Available Use% Mounted on
clscls01-MDT0000_UUID       10.0T       79.4G        9.8T   1% /lus/clscls01[MDT:0]
clscls01-MDT0001_UUID       10.0T       19.7M        9.9T   1% /lus/clscls01[MDT:1]
clscls01-OST0000_UUID      278.0T      120.5T      154.7T  44% /lus/clscls01[OST:0]
clscls01-OST0001_UUID      278.0T      107.2T      168.0T  39% /lus/clscls01[OST:1]
clscls01-OST0002_UUID      278.0T      116.1T      159.1T  43% /lus/clscls01[OST:2]
clscls01-OST0003_UUID      278.0T      109.8T      165.4T  40% /lus/clscls01[OST:3]
clscls01-OST0004_UUID      278.0T      110.2T      165.0T  41% /lus/clscls01[OST:4]
clscls01-OST0005_UUID      278.0T      113.8T      161.4T  42% /lus/clscls01[OST:5]
clscls01-OST0006_UUID      278.0T      117.8T      157.4T  43% /lus/clscls01[OST:6]
clscls01-OST0007_UUID      278.0T      120.6T      154.6T  44% /lus/clscls01[OST:7]
clscls01-OST0008_UUID      278.0T      116.4T      158.8T  43% /lus/clscls01[OST:8]
clscls01-OST0009_UUID      278.0T      121.9T      153.3T  45% /lus/clscls01[OST:9]
clscls01-OST000a_UUID      278.0T      108.5T      166.7T  40% /lus/clscls01[OST:10]
clscls01-OST000b_UUID      278.0T      112.2T      163.1T  41% /lus/clscls01[OST:11]

filesystem_summary:         3.3P        1.3P        1.9P  42% /lus/clscls01
```

Lfs setstripe

- Sets the stripe for a file or a directory

```
lfs setstripe    <--stripe-size |-S size>
                 <--stripe-count|-c count> <file|dir>
```

  - size: Number of bytes on each OST (0 filesystem default)
  - count: Number of OSTs to stripe over (0 default, -1 all)

- Comments
  - Striping policy is set when the file is created. It is not possible to change it afterwards.
  - Can use lfs to create an empty file with the stripes you want (like the touch command)
  - Can apply striping settings to a directory,
    any children will inherit parent's stripe settings on creation.

  - There is an index option to choose the first OST, don't use this in normal circumstances.

Select Best Lustre Striping Values

- Selecting the striping values will have a large impact on the I/O performance of your application
- Rules of thumb: Try to use all OSTs
  - # files > # OSTs => Set stripe_count=1
    You will reduce the lustre contention and OST file locking this way and gain performance
  - #files==1 => Set stripe_count=#OSTs or a number where your performance plateaus
    Assuming you have more than 1 I/O client
  - #files<#OSTs => Select stripe_count so that you use all OSTs
    Example : You have 8 OSTs and write 4 files at the same time, then select stripe_count=2

- Always allow the system to choose OSTs at random!

Sample Lustre Commands: Striping

```
crystal:ior% mkdir tigger
crystal:ior% lfs setstripe -S 2m -c 4 tigger
crystal:ior% lfs getstripe tigger
tigger
stripe_count:   4 stripe_size:    2097152 stripe_offset:  -1
crystal% cd tigger
crystal:tigger% ~/tools/mkfile_linux/mkfile 2g 2g
crystal:tigger% ls -lh 2g
-rw------T 1 harveyr criemp 2.0G Sep 11 07:50 2g
crystal:tigger% lfs getstripe 2g
2g
lmm_stripe_count:   4
lmm_stripe_size:    2097152
lmm_layout_gen:     0
lmm_stripe_offset:  26
        obdidx           objid           objid           group
            26        33770409      0x2034ba9                 0
            10        33709179      0x2025c7b                 0
            18        33764129      0x2033321                 0
            22        33762112      0x2032b40                 0
```

Conclusions

- Lustre is a high performance, high bandwidth parallel file system.
  - It requires many multiple writers to multiple stripes to achieve best performance

- There is large amount of I/O bandwidth available to applications that make use of it. However, users need to match the size and number of Lustre stripes to the way files are accessed.
  - Large stripes and counts for big files
  - Small stripes and count for smaller files

Being Nice to Lustre

From Bandwidth to Filesystem operations

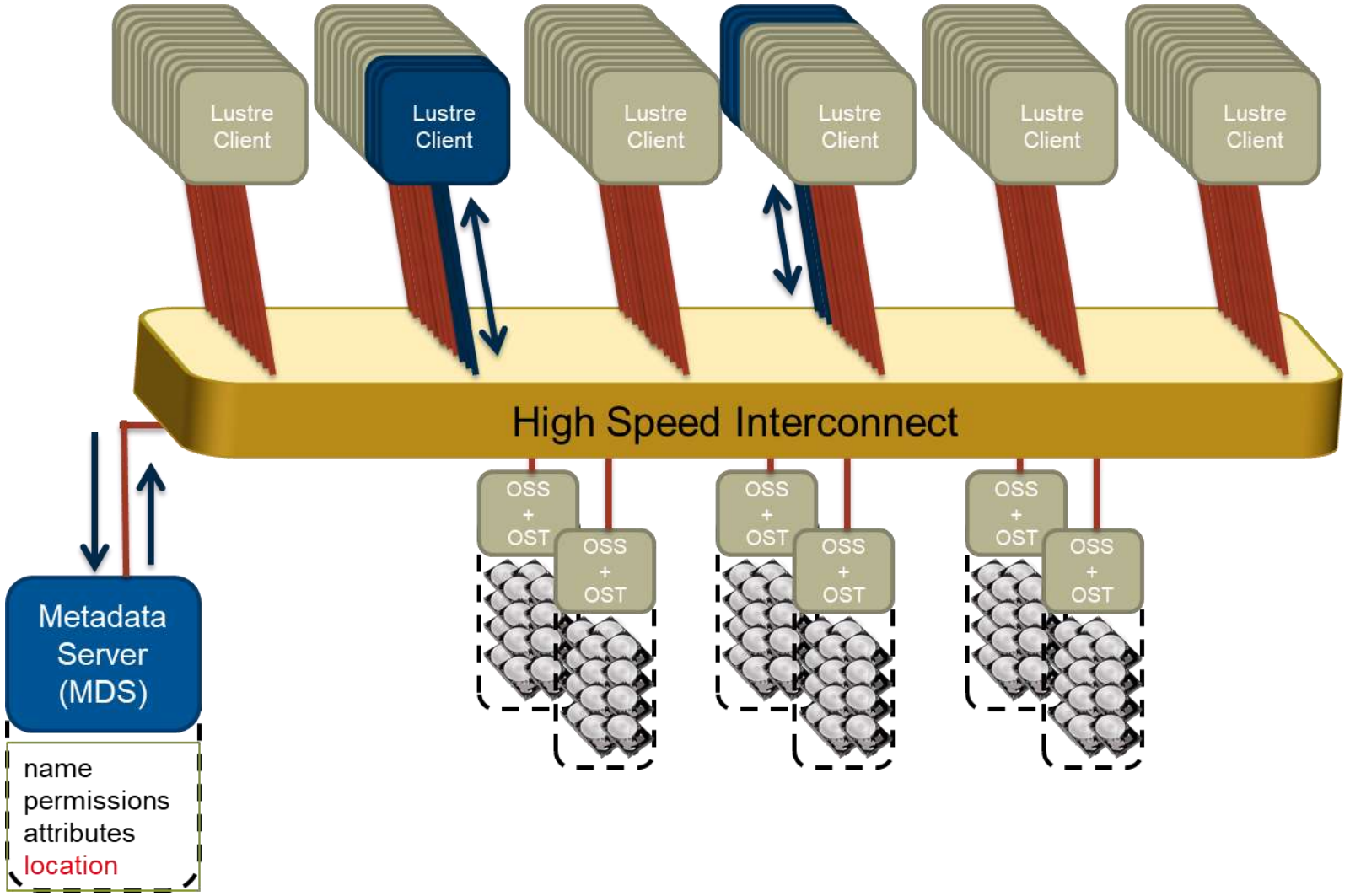
Being Nice to Lustre

- There are two characteristics we typically use to talk about storage or filesystem performance
  - BANDWIDTH
  - OPERATIONS PER SECOND (IOPS)
- Lustre is a parallel distributed filesystem, so we have two further aspects
  - Performance of data I/O (accessing OSTs)
  - Performance of metadata I/O (filesystem operations via MSS/MDT)
- We have already considered advice on optimizing for data throughput

- We now concentrate more on performance of filesystem operations

The Metadata Server is a Finite Shared Resource

One or more



Metadata Operations

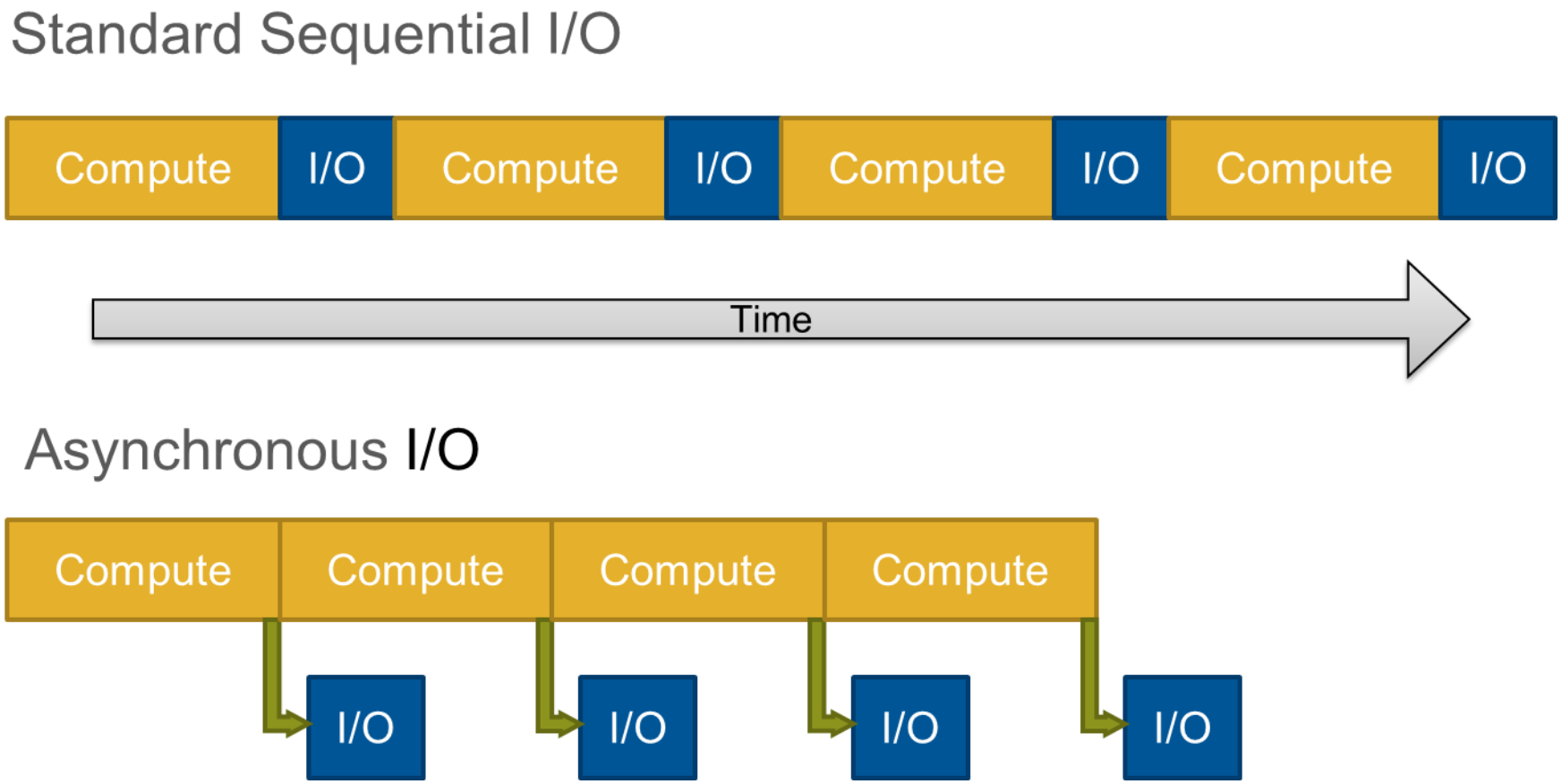
- The Metadata Server (MDS) provides access to each filesystem's metadata stored on Metadata Storage Targets (MDTs)
- It is involved in many filesystem operations
  - Create, Open, Close, get attributes etc.
  - Managing locks
  - (note Read/Write of file DATA go direct to OSSs/OSTs)
- It is a shared resource so can be stressed in large systems by some workloads
- Result may be slow or variable filesystem performance
- An older generation MDS might support metadata rates from 20,000 to 60,000 operations per second depending on operation type

Advice to Mitigate Metadata Operation Load

- Avoid using Lustre for local TMPDIR
- Avoid stat() calls (from colour ls, shell file completion, do open/fail instead of stat/INQUIRE)
- Open files read-only if that is the intention
- Read on rank-0 and broadcast instead of reading small files from every task
- If you need to access many files from many processes (for example python environments) consider moving into /tmp or embedding into a container
- For shared file access from many nodes
  - Avoid writing from multiple clients to part of a file on the same OST
  - MPI-IO does this for you
- Avoid very large directories
  - Organize by client
- Avoid appending to a file from many nodes(clients)
  - Either seek to specific location or have only one rank append to the file
- Try to write aligned data (1MB)

Asynchronous I/O

Asynchronous I/O



Approaches to Asynchonous I/O

- Double buffer arrays to allow computation to continue while data is flushed to disk
- Possible approaches to do this are:

  - Use 3rd party libraries
    - Typically, MPI I/O
      It has split phase collective I/O (check if implementation does do the I/O asynchronously)
  - Do I/O in a thread
  - Add I/O servers to application
    - Dedicated processes to perform time consuming operations
    - More complicated to implement than other solutions
    - Portable solution (works on any parallel platform)

  - Active work in this area, For example ExaIO project
    https://www.exascaleproject.org/research-project/exaio/
    Transparent Asynchronous Parallel I/O Using Background Threads,
    https://doi.org/10.1109/TPDS.2021.3090322

I/O Servers

- Successful strategy deployed in several codes
- Has become more successful as number of nodes has increased
  - Extra nodes only cost few percent of resources
- Requires additional development that can pay off for codes that generate large files
- Typically, still only one or a small number of writers performing I/O operations
  - May not reach full I/O bandwidth

I/O Server pseudo code

Compute Node

```
do i=1,time_steps
  compute(j)
  checkpoint(data)
end do

subroutine checkpoint(data)
  MPI_Wait(send_req)
  buffer = data
  MPI_Isend(IO_SERVER, buffer)
end subroutine
```

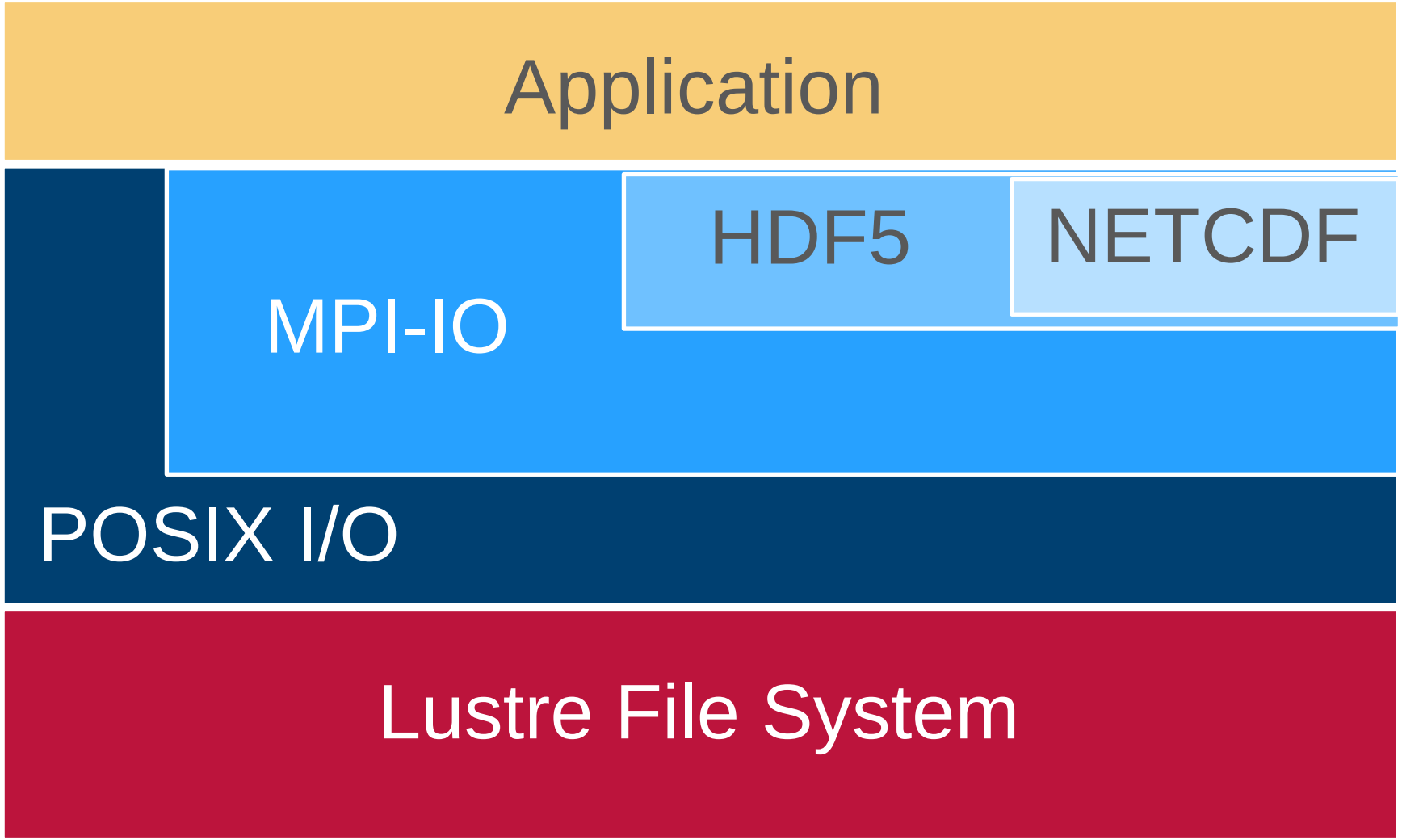
I/O Server

```
do i=1,time_steps
  do j=1,compute_nodes
    MPI_Recv(j, buffer)
    write(buffer)
  end do
end do
```

MPI-IO

Internals and profiling

MPI-IO Stack



MPI I-O Data Movement Optimizations

- The MPI-IO implementation uses two data movement optimizations
- Data sieving is used to combine lots of small accesses into a single larger one
  - Reducing # of operations important (latency)
  - A system buffer/cache is one example
- Aggregation/Collective Buffering refers to the concept of moving data through intermediate nodes
  - Different numbers of nodes performing I/O (transparent to the user)
- Triggered with MPI HINTS
- See intro_mpi and look for “cb“

MPI I/O Interaction with Lustre

- Included in the HPE Cray MPI library

- Environment variables used to help MPI-IO optimize I/O performance
  - MPICH_MPIIO_HINTS can set striping_factor and striping_unit for files created with MPI I/O:

    MPICH_MPIIO_HINTS="T.nc:striping_factor=4:striping_unit=2097152"
  - If writes and/or reads utilize collective calls, collective buffering can be utilized (romio_cb_read/write) to approximately stripe align I/O within Lustre
- HDF5 and NetCDF are both implemented on top of MPI I/O and thus are also affected by these environment variables

Supplying MPI-IO hints via info

- Create an MPI_Info object and pass to file creation routine

```
integer info, stripe_count, stripe_size
character(len=20) value

! Usual MPI setup here

if (set_hints) then
   call MPI_Info_create(info,ierr)
   stripe_count = 4
   stripe_size = 1024*1024 * 1
   write(value,'(i12)')stripe_count
   call
MPI_Info_set(info,"striping_factor",value,ierr)
   write(value,'(i12)')stripe_size
   call
MPI_Info_set(info,"striping_unit",value,ierr)
  else
   info = MPI_INFO_NULL
  end if

! Pass info to MPI_file_open() or nf90_create()
```

MPI-IO Hints

- MPICH_MPIIO_HINTS_DISPLAY – Rank 0 displays the name and values of the MPI-IO hints
- MPICH_MPIIO_HINTS – Sets the MPI-IO hints for files opened with the MPI_File_Open routine
  - Overrides any values set in the application by the MPI_Info_set routine
  - Following hints are supported:

direct_io	cb_nodes	romio_ds_write
romio_cb_read	cb_config_list	ind_rd_buffer_size
romio_cb_write	romio_no_indep_rw	Ind_wr_buffer_size
	romio_ds_read	striping_factor
		striping_unit

- Can't be changed once you have called MPI_File_open()

Setting up MPI-IO for shared-file access

- It can be tricky to get performance in this case even if you have set striping…
- You can set the number of aggregators per node with the hint
  cray_cb_nodes_multiplier
- There is another hint which can improve locking performance on a shared file
  cray_cb_write_lock_mode=2
- MPI will print the aggregator locations if you set
  MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
- So, in a job script for example:

```
mult_N=$((SLURM_NNODES/4))
echo "cb_nodes multiplier $mult_N“
export
MPICH_MPIIO_HINTS="*:cray_cb_write_lock_mode=2,*:cray_cb_nodes_mu
ltiplier=$mult_N"
```

HPE Cray MPI-IO Performance Metrics

- Many times MPI-IO calls are “Black Holes” with little performance information available.
- Cray’s MPI-IO library attempts collective buffering and stripe matching to improve bandwidth and performance.
- User can help performance by favouring larger contiguous reads/writes to smaller scattered ones.
- The Cray MPI-IO library now provides a way of collecting statistics on the actual read/write operations performed after collective buffering
  - Simple report enabled with `MPICH_MPIIO_STATS=1`
  - A more complicated report enabled with: `export MPICH_MPIIO_STATS=2`
  - Will also provide some csv files which can be analysed by a provided tool called cray_mpiio_summary

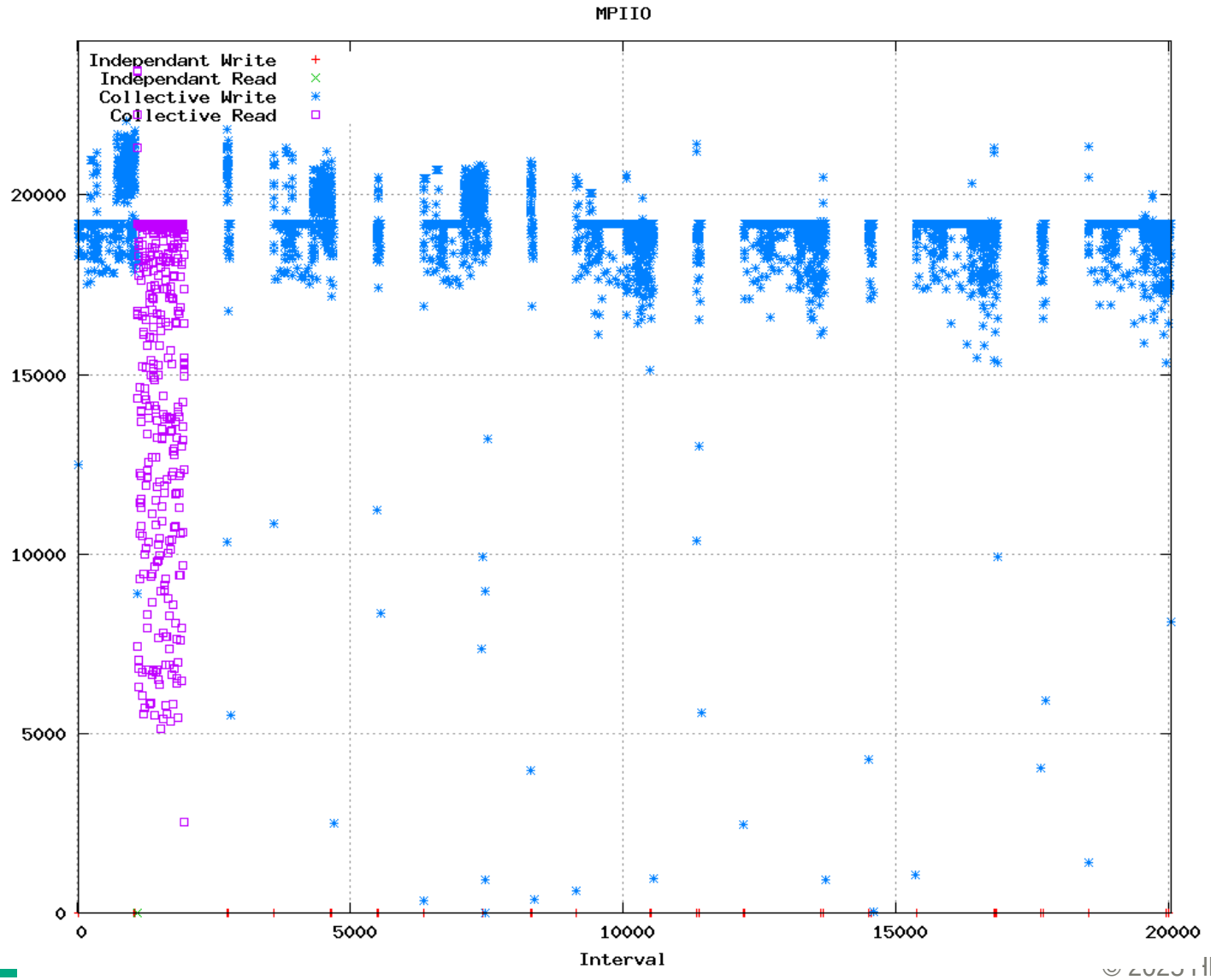
Performance Statistics Example

Running wrf on 19200 cores :

```
| MPIIO write access patterns for wrfout_d01_2013-07-01_01_00_00
|   independent writes      = 2
|   collective writes       = 5932800
|   system writes           = 99871
|   stripe sized writes     = 99291
|   total bytes for writes  = 104397074583 = 99560 MiB = 97 GiB
|   ave system write size   = 1045319
|   number of write gaps    = 2
|   ave write gap size      = 524284
| See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations.
```

Best performance when avg write size > 1MB and few gaps. Careful selection of MPI types, file views and ordering of data on disk can improve this.

WRF, 19200 cores run Number of MPI-IO calls over time



Rules for Good Application I/O Performance

1. Use parallel I/O
2. Try to hide I/O (use asynchronous I/O)
3. Tune file system parameters
4. Use I/O buffering for all sequential I/O

Summary

- I/O is always a bottleneck when scaling out
  - You may have to change your I/O implementation when scaling up

- Take-home messages on I/O performance
  - Performance is limited for single process I/O
  - Parallel I/O utilizing a file-per-process or a single shared file is limited at large scales
  - Potential solution is to utilize multiple shared file or a subset of processes which perform I/O
  - A dedicated I/O Server process (or more) might also help
  - Use MPI I/O and/or high-level libraries (HDF5, NETCDF, ..)

- Set the Lustre striping parameters!