Optimizing large scale I/O

Supercomputing, **n. A special branch of scientific computing that turns a computation-bound problem into an I/O-bound problem.**

Overview

- **The Cray Linux Environment and parallel libraries provide full support for common I/O standards.**
  - Serial POSIX I/O
  - Parallel MPI I/O
    - Third-party libraries built on top of MPI I/O
    - HDF5, NetCDF4

- **Cray versions provide many enhancements over generic implementations that integrate directly with Cray XC40 and Cray Sonexion hardware.**
  - Cray MPI-IO collective buffering, aggregation and data sieving.
  - Automatic buffering and direct I/O for Posix transfers via IOBUF.

Building blocks of HPC file systems

- **Modern Supercomputer hardware is typically built on two fundamental pillars:**
  1. The use of widely available commodity (inexpensive) hardware.
     - Intel CPUs, AMD CPUs, DDR3, DDR4, …
  2. Using parallelism to achieve very high performance.

- **The file systems connected to computers are built in the same way**
  1. Gather large numbers of widely available, inexpensive, storage devices
     - Can be HDDs, SSDs
  2. then connect them together in parallel to create a high bandwidth, high capacity storage device.

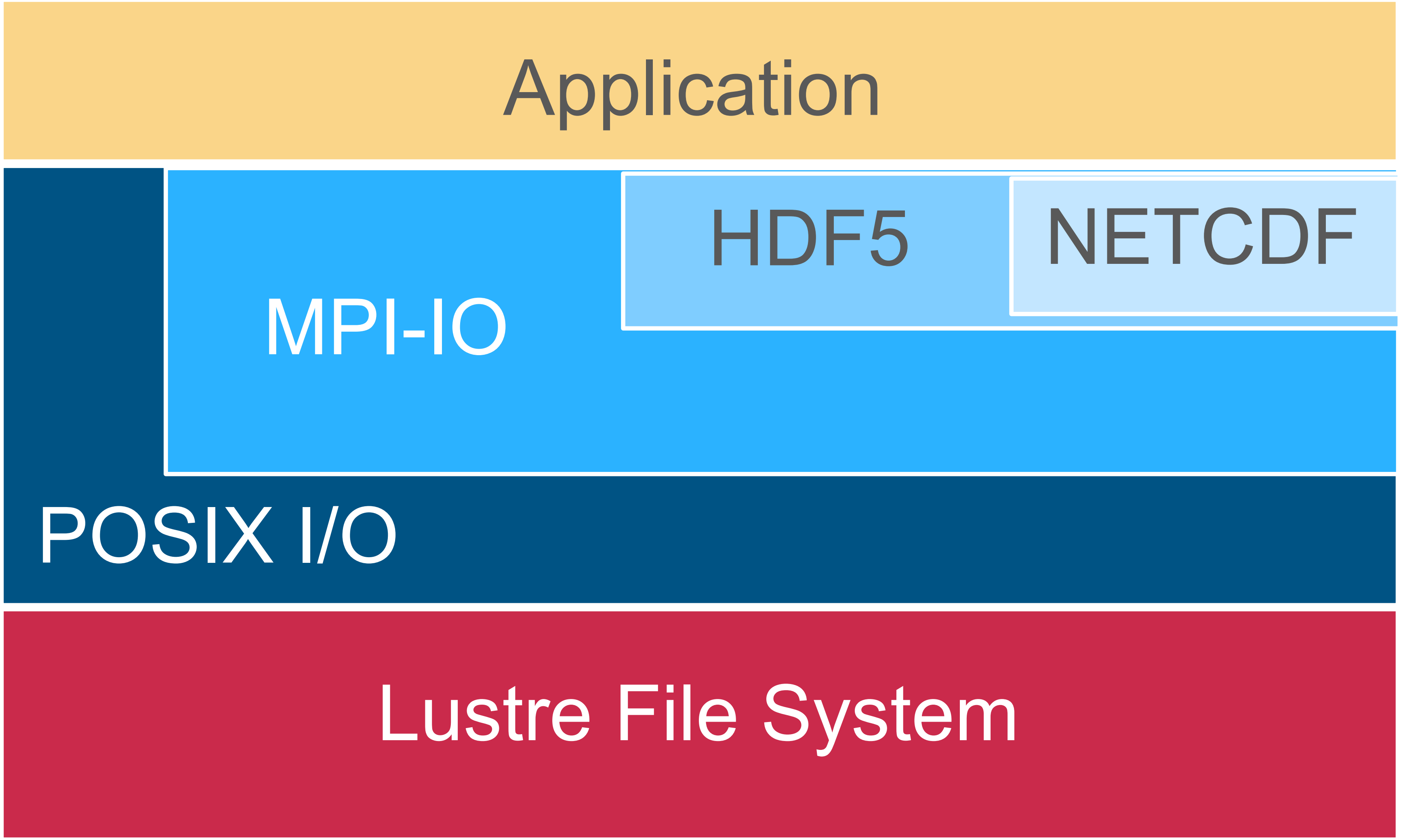
So you will have to do parallel I/O in order to get performance

Challenges in I/O

- **From an application point of view :**
  - The tasks of the applications has to be able to make use of the bandwidth the I/O system offers
  - The number of files created is also an issue
    - If your application uses more than 10,000 tasks and creates 3 files per task, you will have over 30,000 output files to deal with

- **But the ‘workflow’ is getting more and more important**
  - How is the created data to be used after the run?
  - Where is the data stored?
    - Moving XXX Tbytes of data from a fast /scratch file system to a permanent place is at best time consuming and at worst impossible
  - How do I deal with 30,000 output files?
  - Which tools are used...?

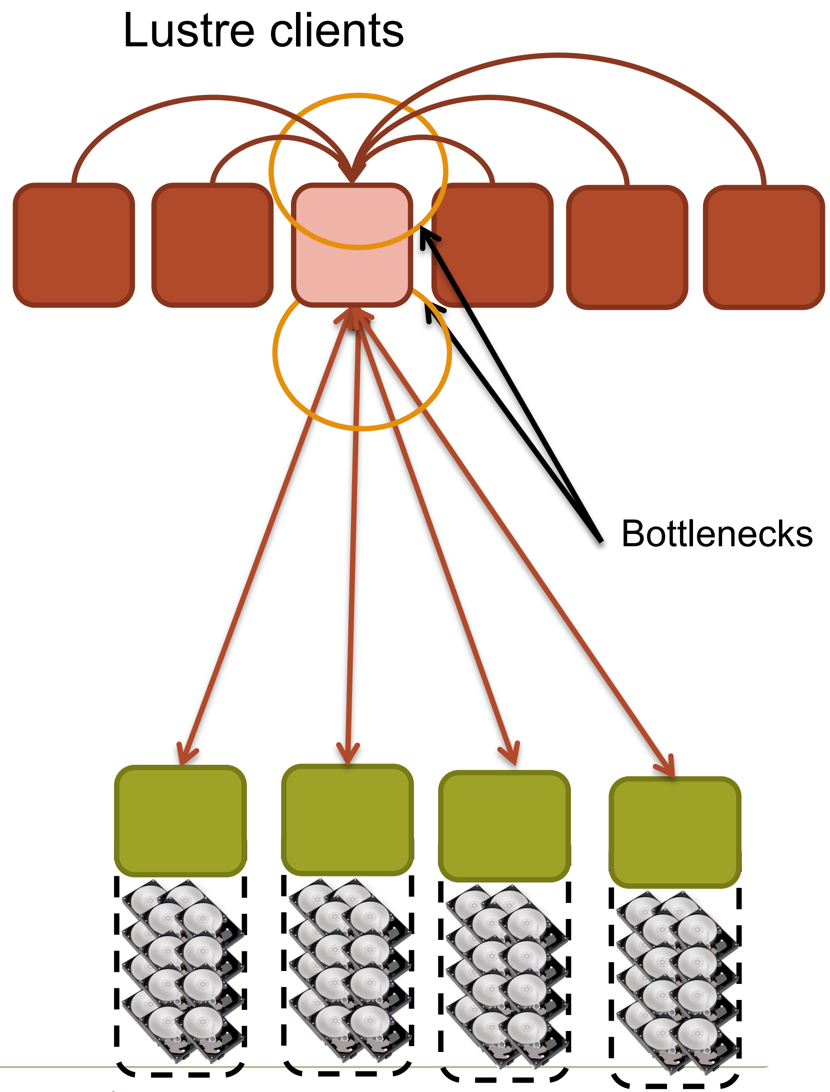
CRAY I/O stack



Common I/O Patterns found in applications

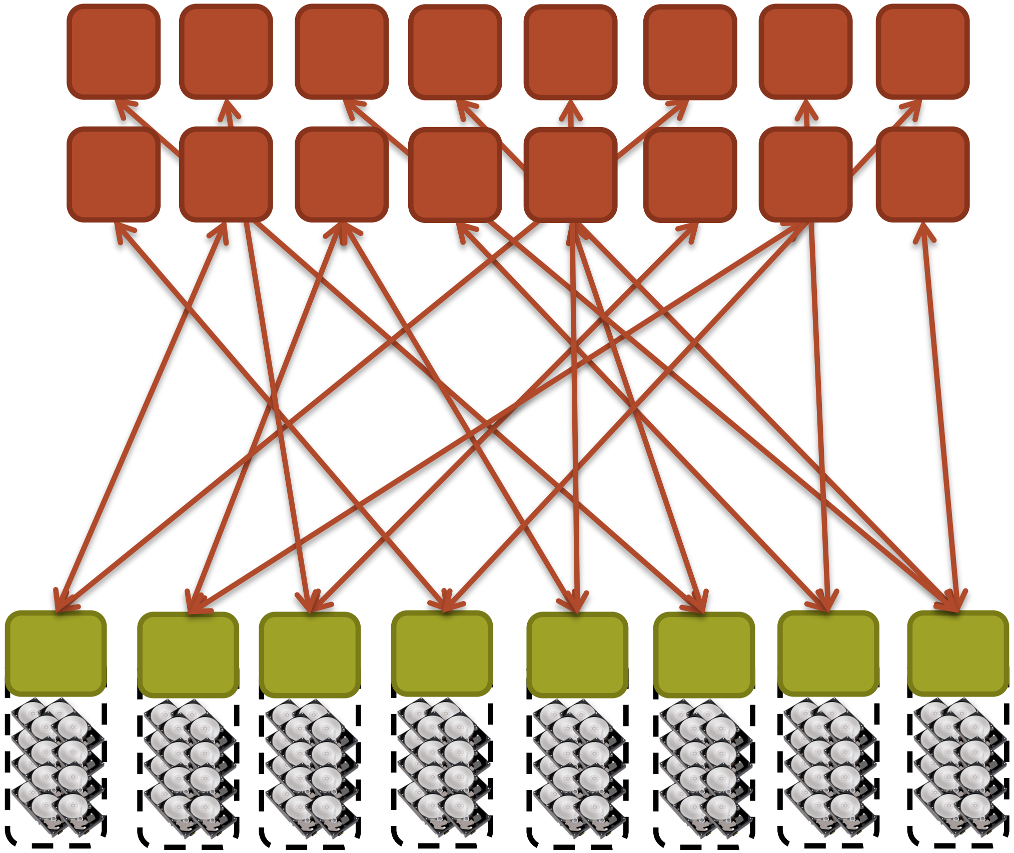
I/O strategies: Spokesperson (sequential I/O)

- **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**



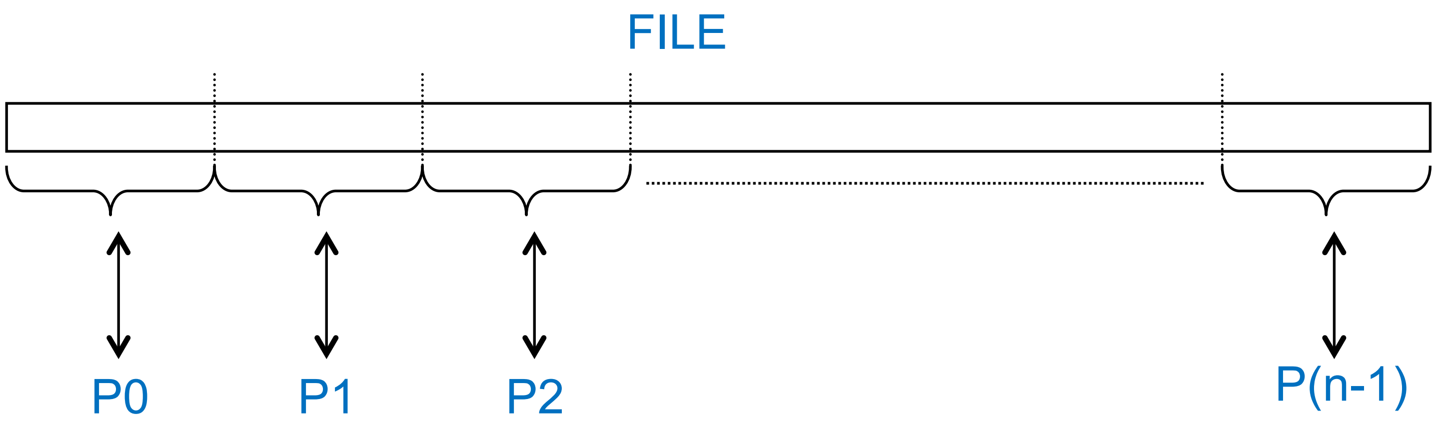
I/O strategies: Multiple Writers – Multiple Files

- **All processes perform I/O to individual files**
- **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
  - Hard to read back from diff number of processes



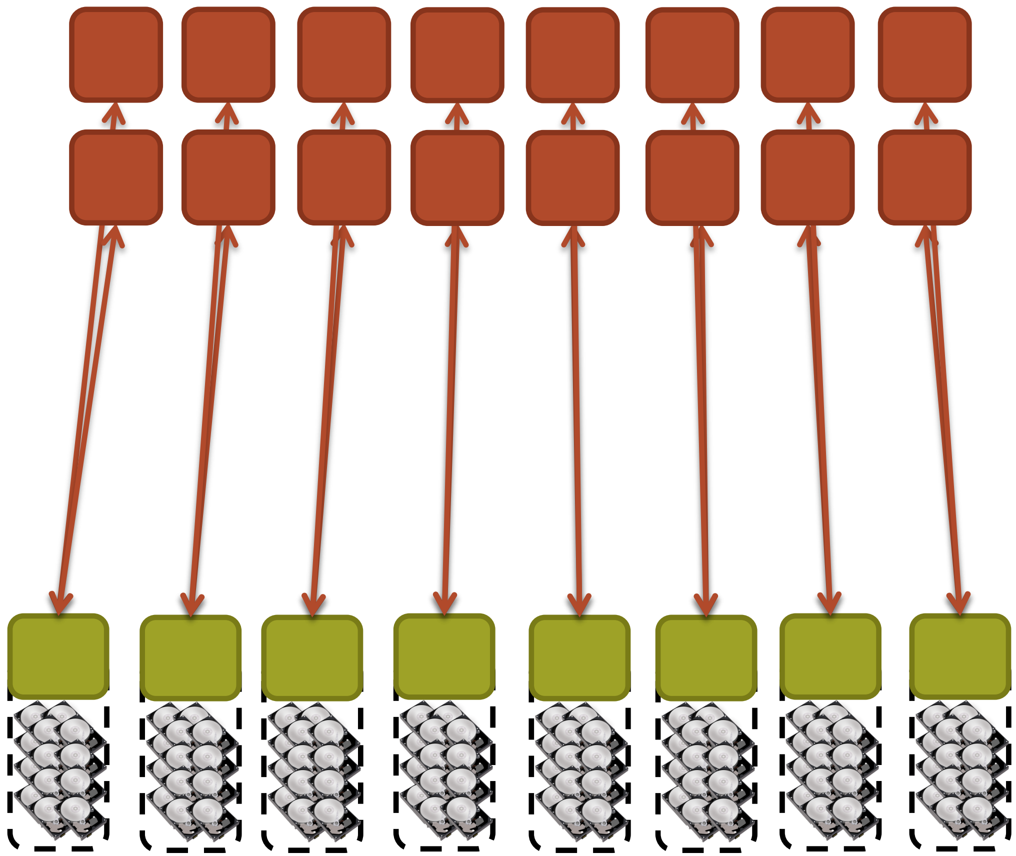
What is Parallel I/O?

- Multiple processes of a parallel program accessing data (reading or writing) from a *common* file



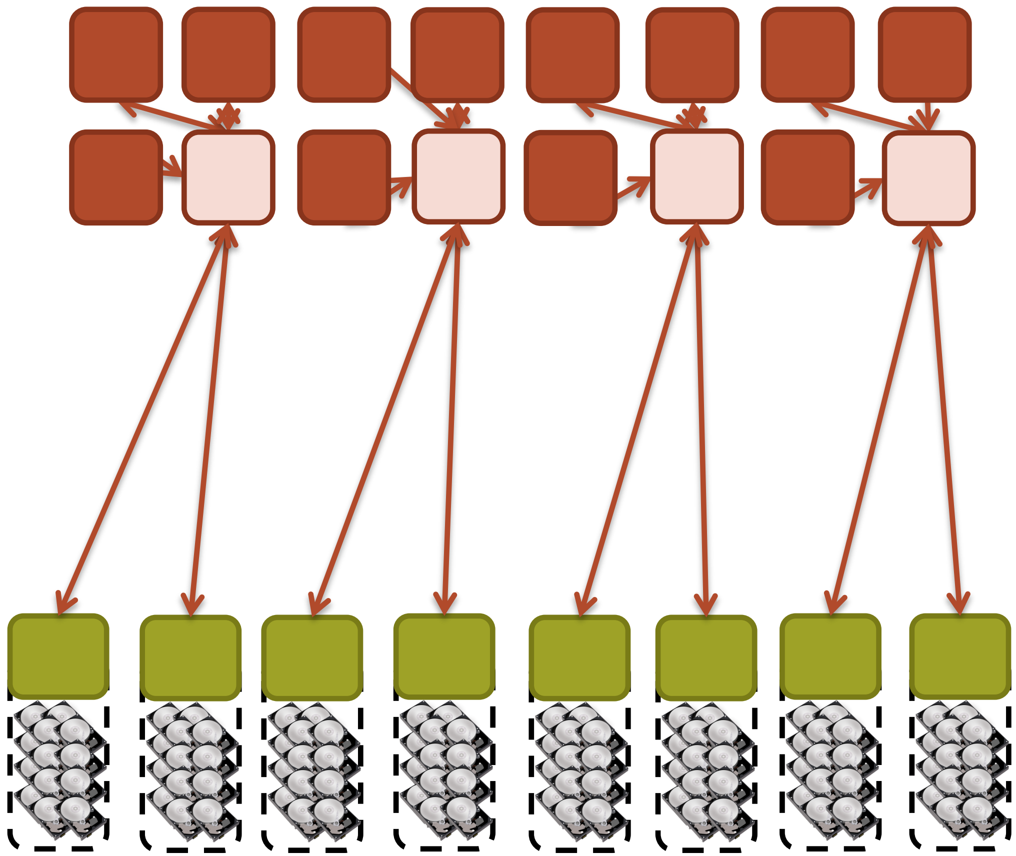
I/O strategies: Multiple Writers – Single File

- **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



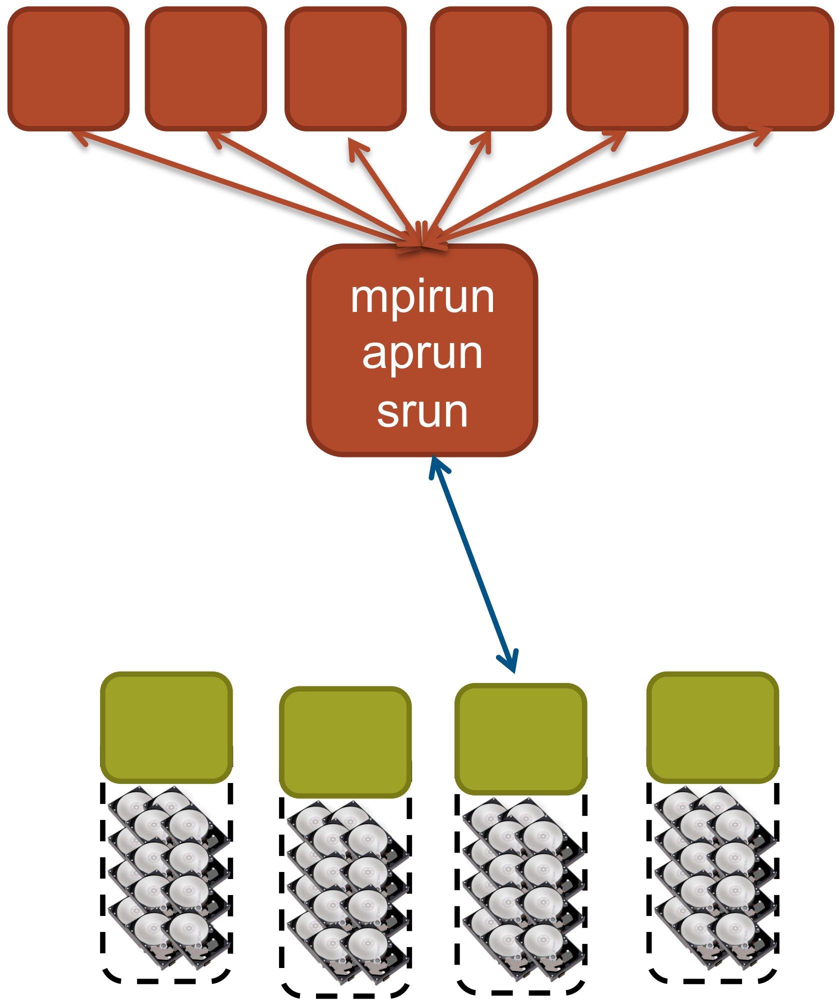
I/O strategies: Collective I/O to single or multiple files

- **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.”



COMPUTE | STORE | ANALYZE

I/O performance: to keep in mind

- There is no "One Size Fits All" solution to the I/O problem
- Many I/O patterns work well for some range of parameters
- Bottlenecks in performance can occur in many locations (application and/or filesystem)
- Going to extremes with an I/O pattern will typically lead to problems
- I/O is a shared resource: Expect timing variation

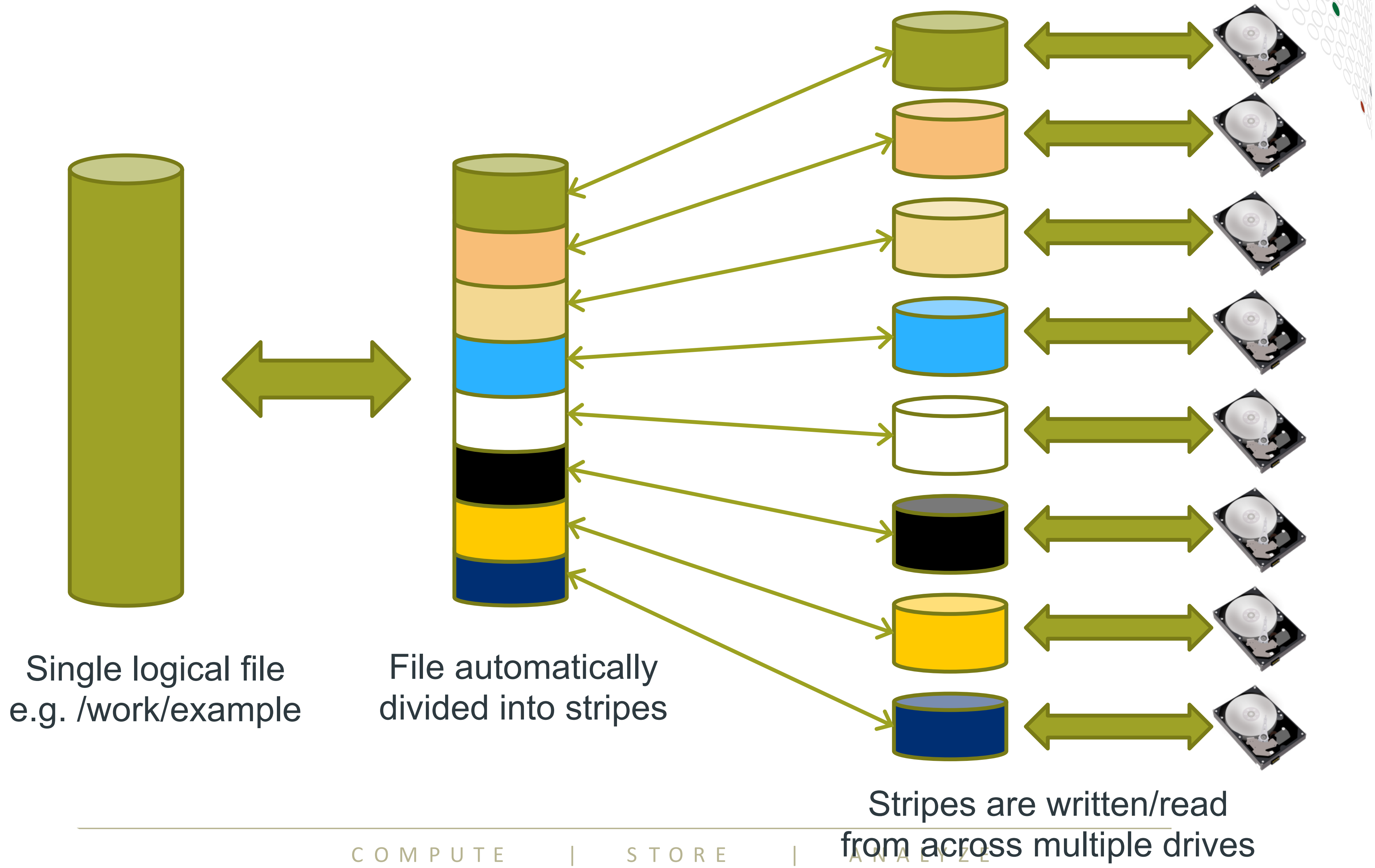
Lustre

A parallel filesystem

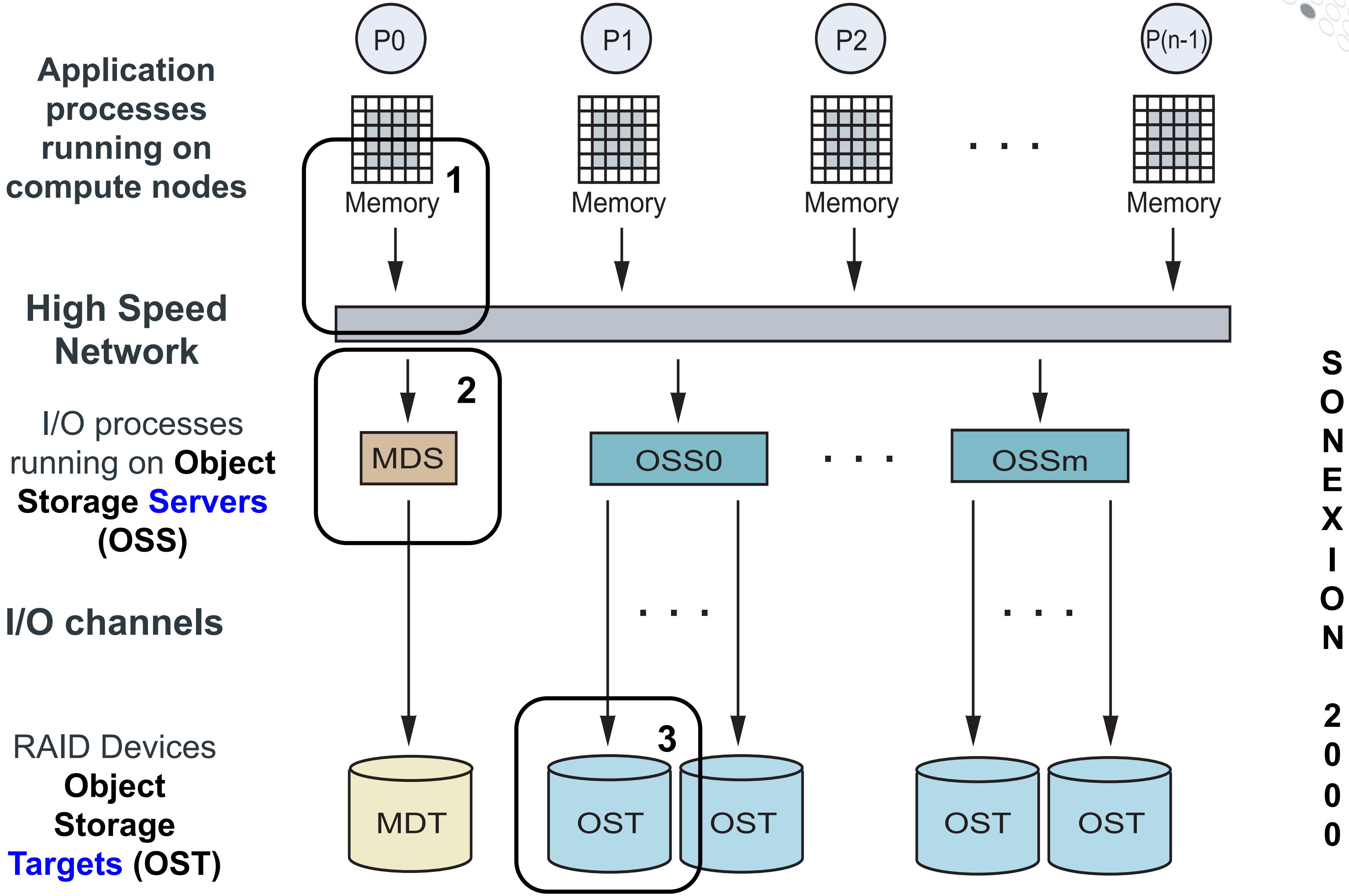
lustre®

- **A scalable cluster file system for Linux**
  - Developed by Cluster File Systems -> Sun -> Oracle.
  - Name derives from "Linux Cluster"
  - Lustre file system consists of software subsystems, storage, and an associated network
- **MDS – metadata server**
  - Handles information (metadata) about files and directories
- **OSS – Object Storage Server**
  - The hardware entity
  - The server node
  - Stores file data on and supports multiple OSTs
- **OST – Object Storage Target**
  - The 'software' entity
  - This is the software interface to the backend volume
  - Each OST manages a single local disk filesystem
- **Client**
  - Accesses and uses data

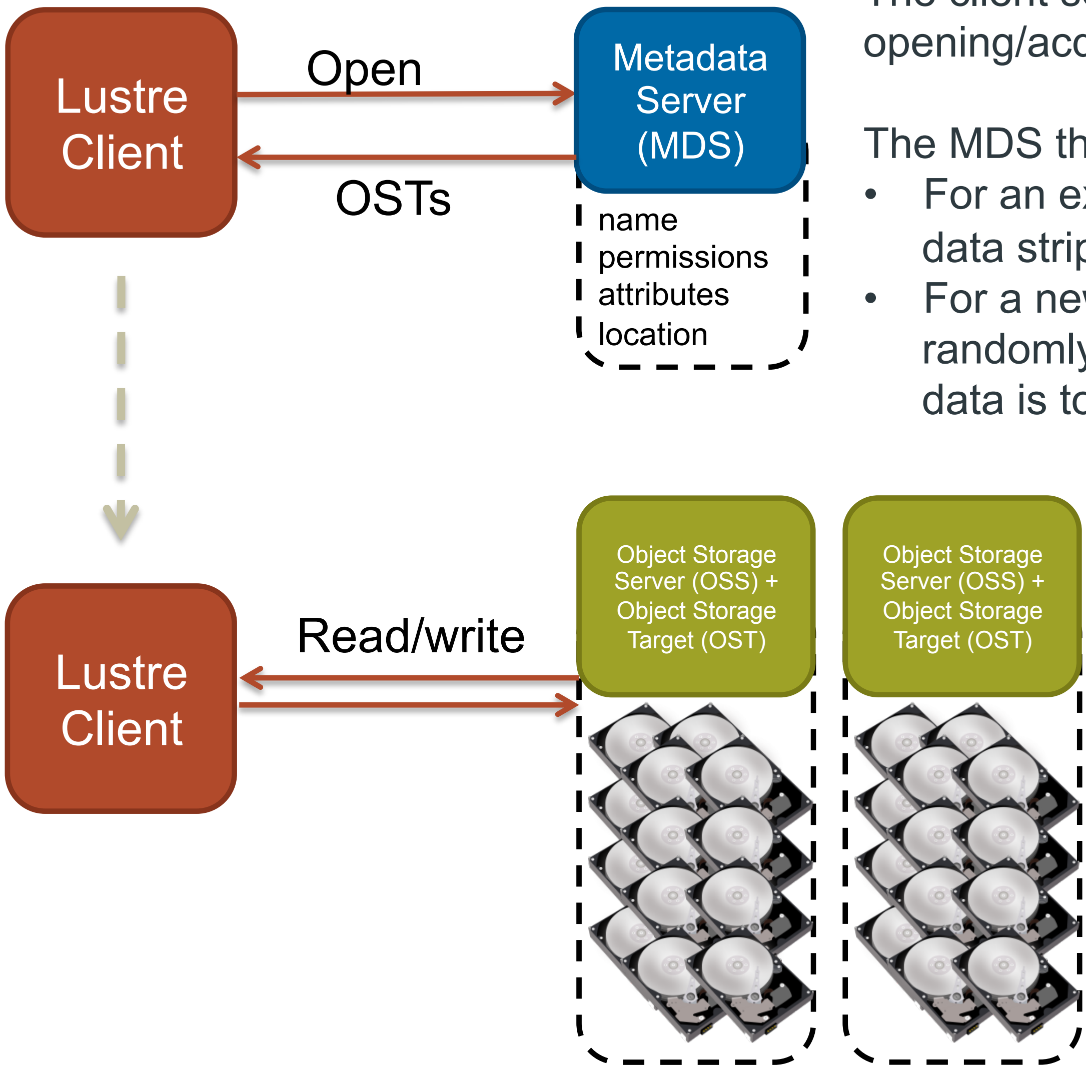
Parallel Filesystem fundamentals



Basic Lustre Overview



Opening a file

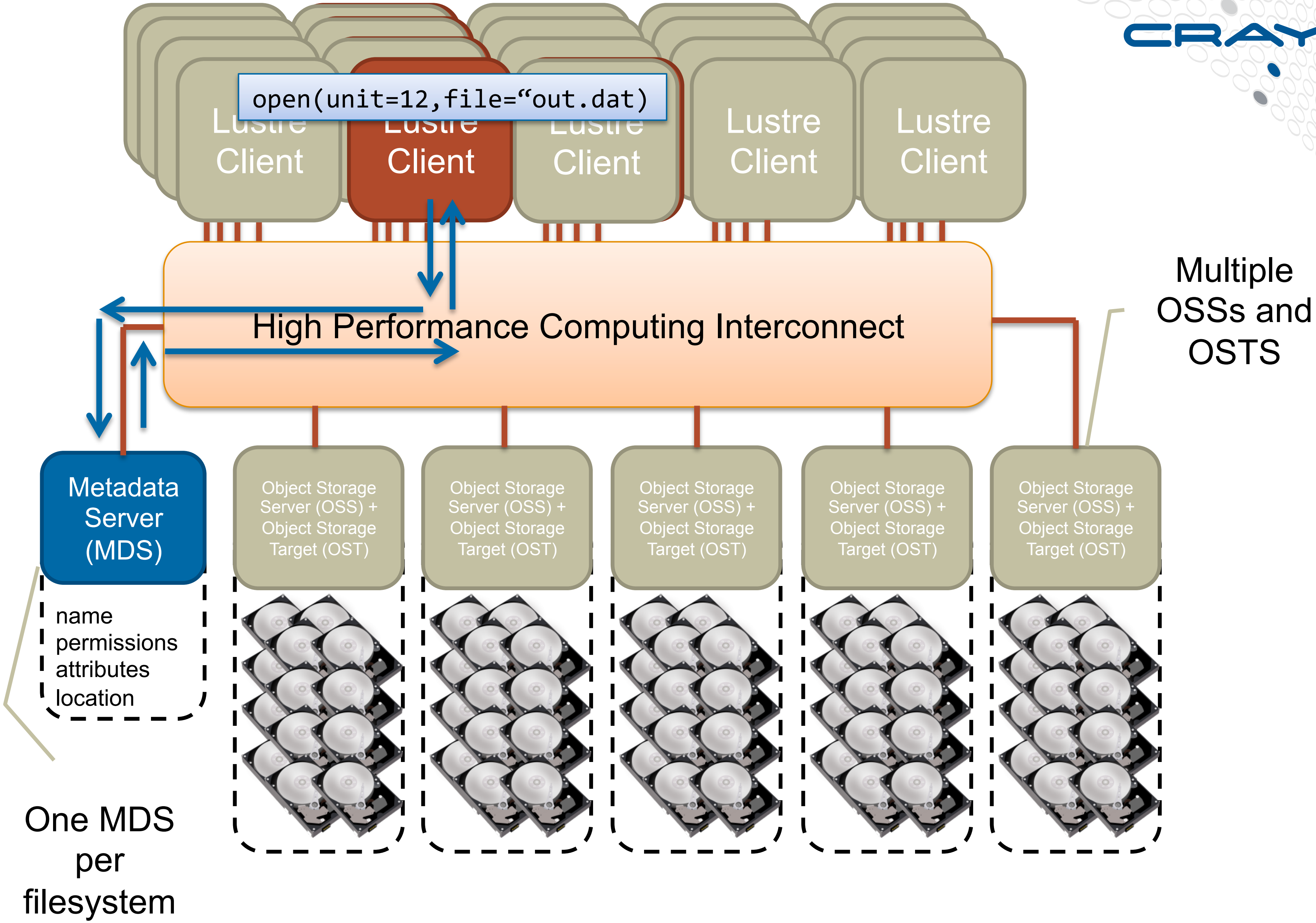


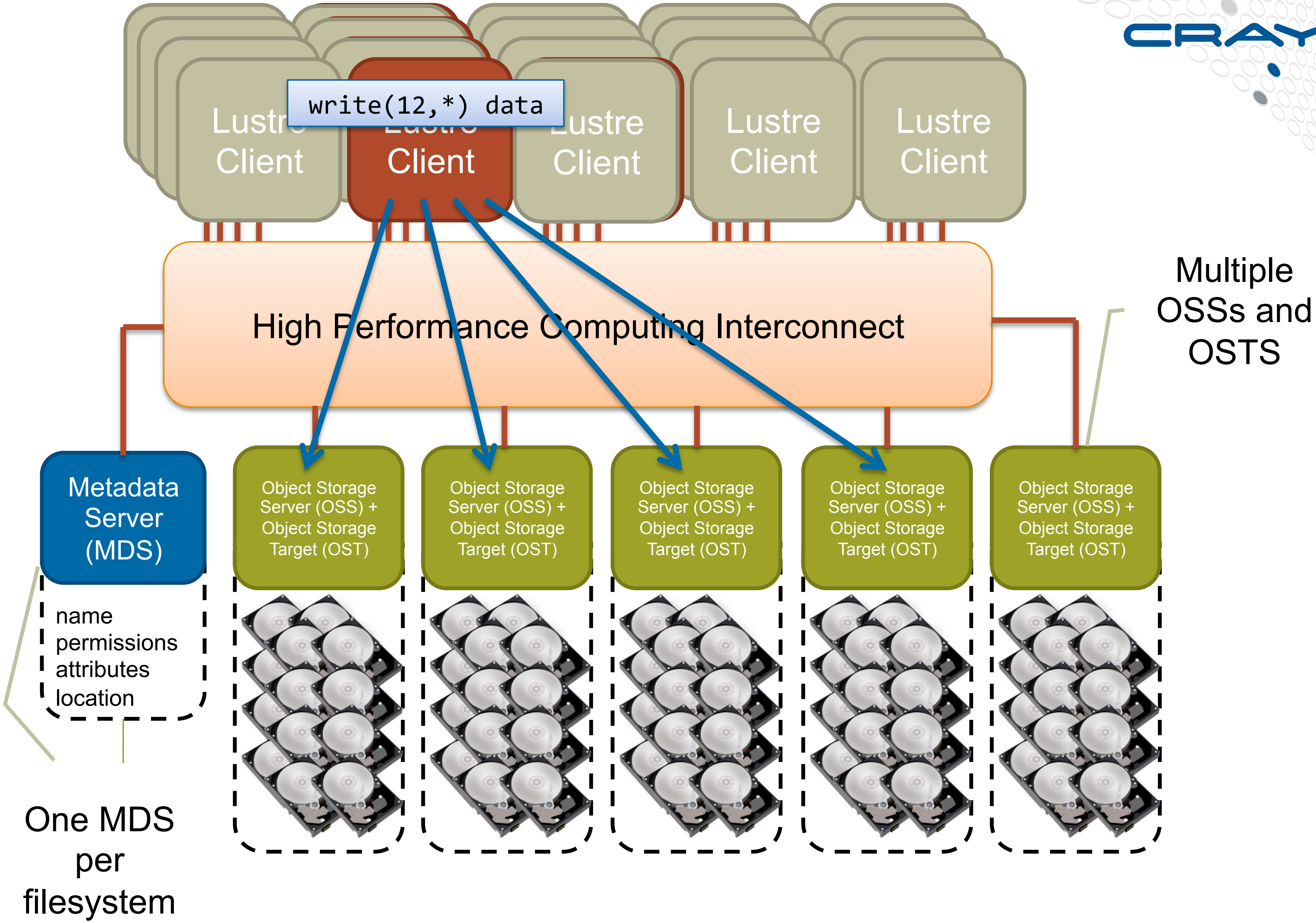
The client sends a request to the MDS to opening/acquiring information about the file

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

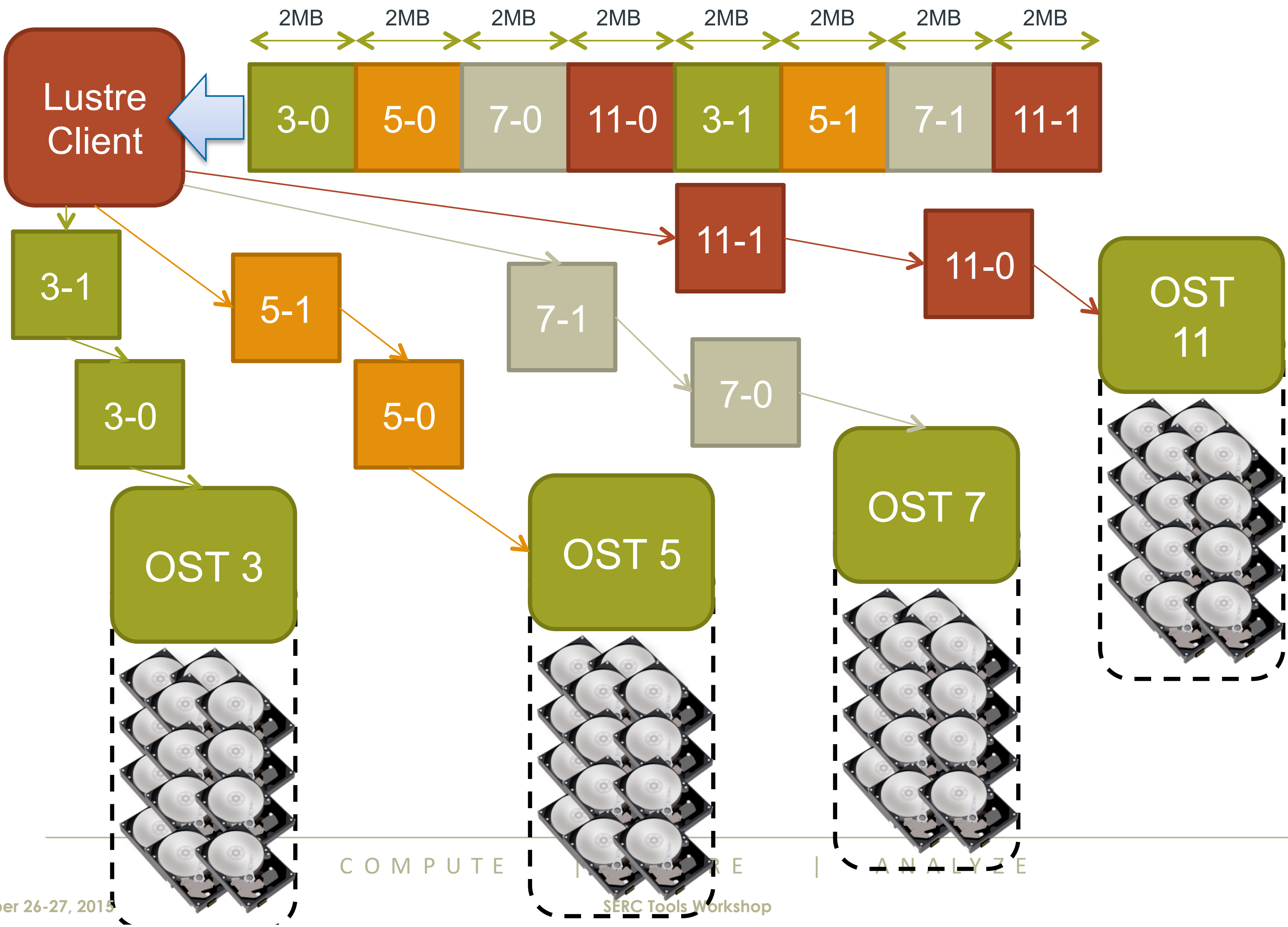
Once a file has been opened no further communication is required between the client and the MDS

All transfer is directly between the assigned OSTs and the client

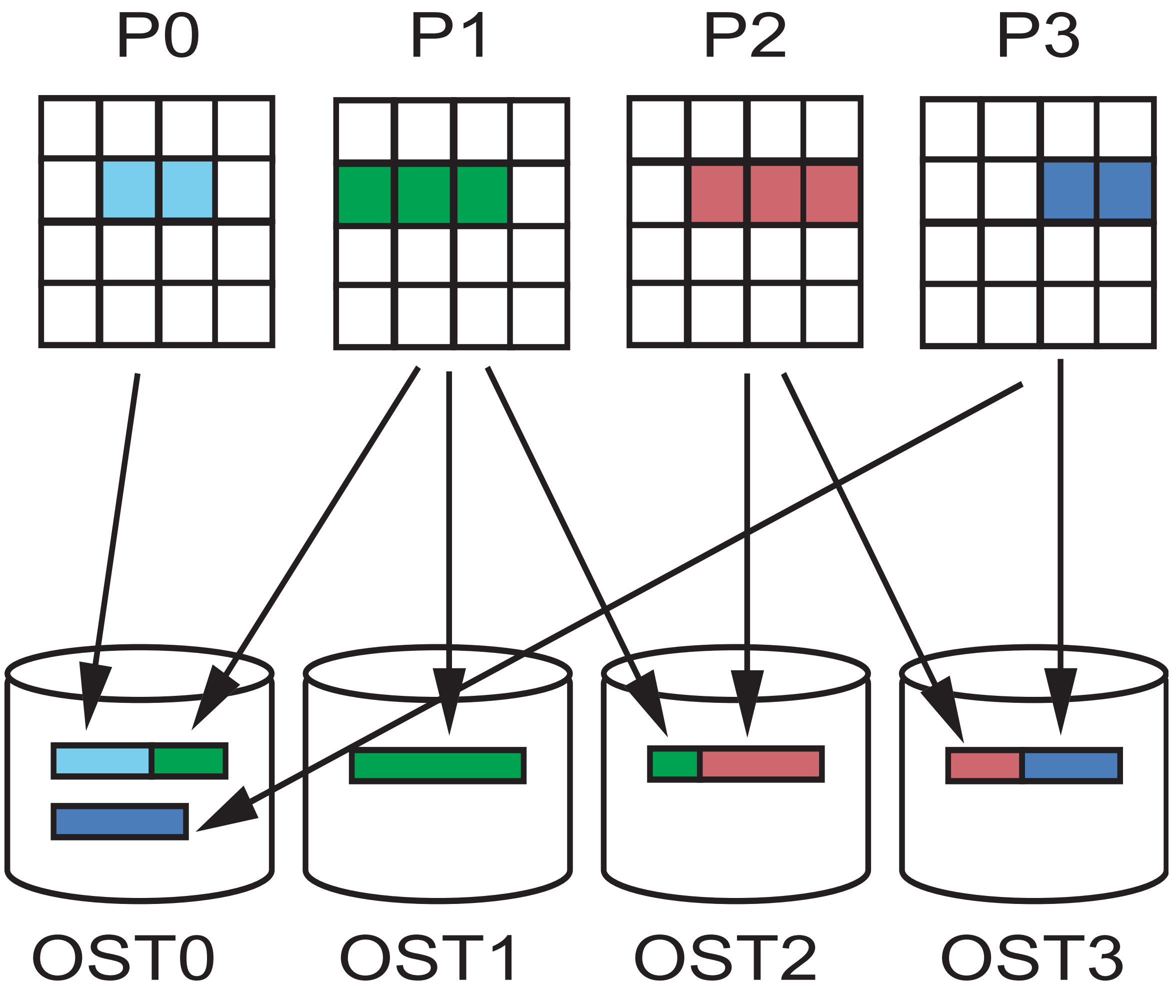

open(unit=12,file="out.dat)
Client
Client
Client
Lustre
Client
Lustre
Client
High Performance Computing Interconnect
Metadata
Server
(MDS)
Object Storage
Server (OSS) +
Object Storage
Target (OST)
name
permissions
attributes
location
Multiple
OSSs and
OSTS
One MDS
per
filesystem


write(12,*) data
Lustre Client
Lustre Client
Lustre Client
Lustre Client
Lustre Client
High Performance Computing Interconnect
Multiple OSSs and OSTS
Metadata Server (MDS)
Object Storage Server (OSS) + Object Storage Target (OST)
Object Storage Server (OSS) + Object Storage Target (OST)
Object Storage Server (OSS) + Object Storage Target (OST)
Object Storage Server (OSS) + Object Storage Target (OST)
Object Storage Server (OSS) + Object Storage Target (OST)
name
permissions
attributes
location
One MDS per filesystem

File decomposition – 2 Megabyte stripes



Physical View of Striping



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**
  - Typically use case is a scratch file system for HPC
  - It is a good match for scientific datasets and/or checkpoint data
- **Lustre is not designed to handle large numbers of small files**
  - Potential bottle necks at the MDS when files are opened
  - Data will not be spread over multiple OSTs
  - Not a good choice for compilation
- **Lustre is NOT a bullet-proof file system.**
  - If an OST fails, all files using that OST are basically inaccessible
  - **BACKUP important data elsewhere!**
  - Deleting files is also a greater good – full OSTs start to slow down – get rid of those huge unwanted output data files!

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
  - `osts` – List the number of OSTs associated with this file system
  - `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

```
crystal:ior% lfs df -h
UUID                       bytes        Used   Available Use% Mounted on
snx11014-MDT0000_UUID       2.1T       47.5G        2.0T   2% /lus/sonexion[MDT:0]
snx11014-OST0000_UUID      20.8T        4.6T       16.0T  22% /lus/sonexion[OST:0]
snx11014-OST0001_UUID      20.8T        4.3T       16.3T  21% /lus/sonexion[OST:1]
snx11014-OST0002_UUID      20.8T        4.3T       16.3T  21% /lus/sonexion[OST:2]
snx11014-OST0003_UUID      20.8T        4.0T       16.6T  20% /lus/sonexion[OST:3]
snx11014-OST0004_UUID      20.8T        4.3T       16.3T  21% /lus/sonexion[OST:4]
snx11014-OST0005_UUID      20.8T        4.6T       16.0T  22% /lus/sonexion[OST:5]
snx11014-OST0006_UUID      20.8T        3.9T       16.7T  19% /lus/sonexion[OST:6]
snx11014-OST0007_UUID      20.8T        4.0T       16.6T  20% /lus/sonexion[OST:7]
snx11014-OST0008_UUID      20.8T        4.4T       16.2T  22% /lus/sonexion[OST:8]
snx11014-OST0009_UUID      20.8T        5.1T       15.5T  25% /lus/sonexion[OST:9]
snx11014-OST000a_UUID      20.8T        4.9T       15.8T  24% /lus/sonexion[OST:10]
snx11014-OST000b_UUID      20.8T        4.5T       16.2T  22% /lus/sonexion[OST:11]
snx11014-OST000c_UUID      20.8T        4.8T       15.8T  23% /lus/sonexion[OST:12]
…
snx11014-OST001d_UUID      20.8T        4.1T       16.5T  20% /lus/sonexion[OST:29]
snx11014-OST001e_UUID      20.8T        3.6T       17.0T  18% /lus/sonexion[OST:30]
snx11014-OST001f_UUID      20.8T        3.6T       17.0T  18% /lus/sonexion[OST:31]

filesystem summary:       666.9T      137.2T      522.9T  21% /lus/sonexion
```

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 ~ 1MB?)
  - count: Number of OSTs to stripe over (0 default; -1 all OSTs)

- **Comments**
  - The striping of a file is given 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 (“touch” command)
  - Can apply striping settings to a directory, any children will inherit parent’s stripe settings on creation.
  - Don’t use the ‘index’ option (-i)

Select best Lustre striping values

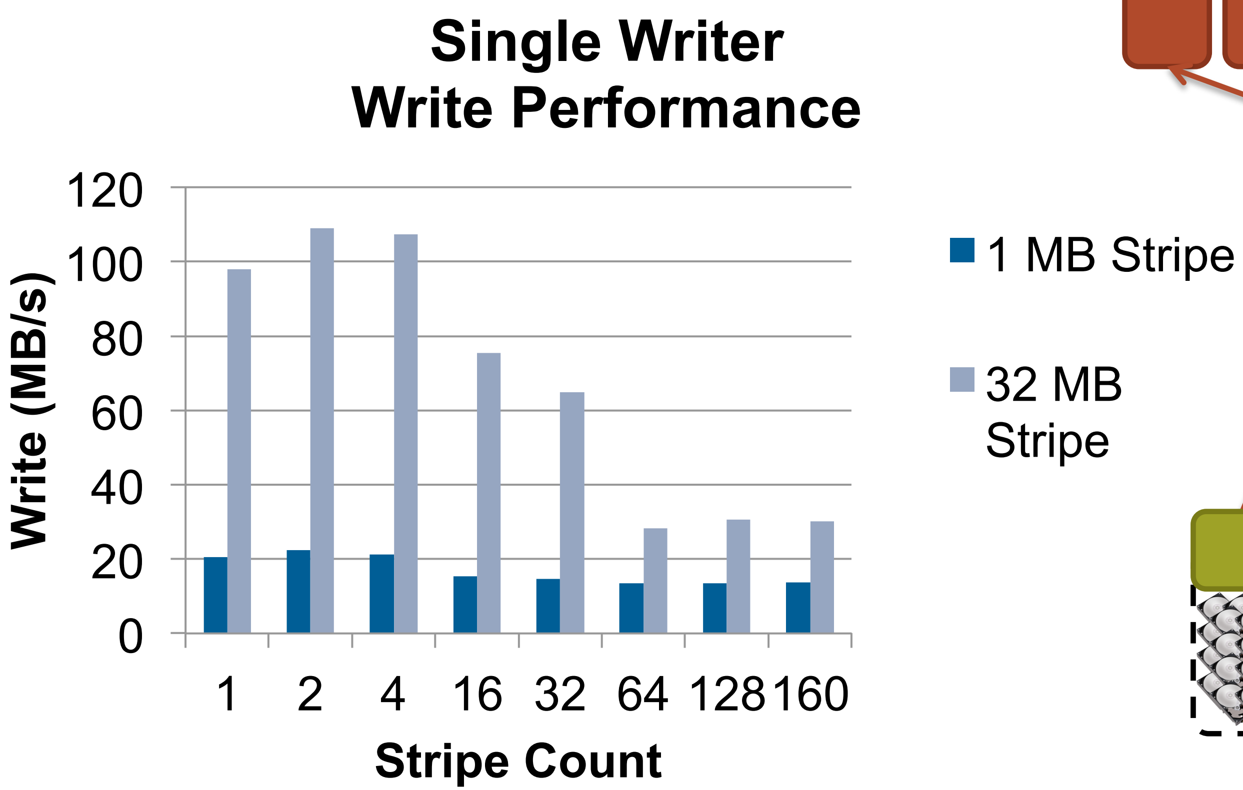
- **Selecting the striping values can have a large impact on the I/O performance of your application**
- **Rules of thumb: Try to use all OSTs**
  1. # files > # OSTs => Set stripe_count=1
     You will reduce the lustre contention and OST file locking this way and gain performance
  2. #files==1 => Set stripe_count=#OSTs
  3. #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!**

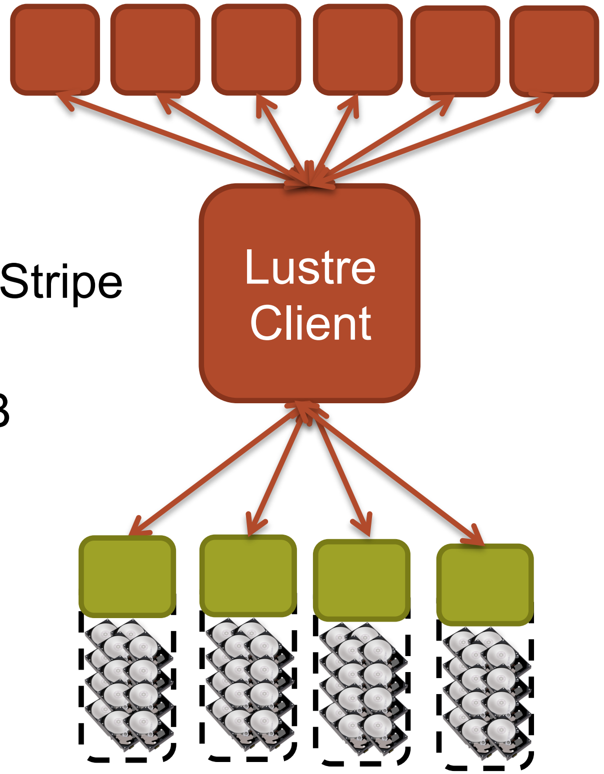
Sample Lustre commands: striping

```
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
```

Case Study 1: Spokesman

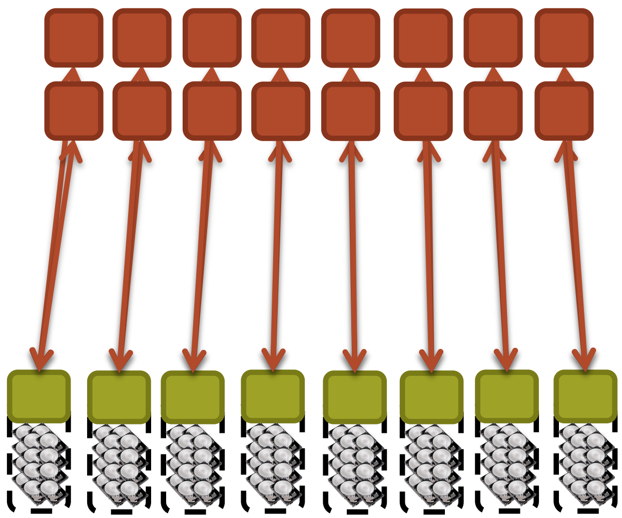
- **32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size**
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance





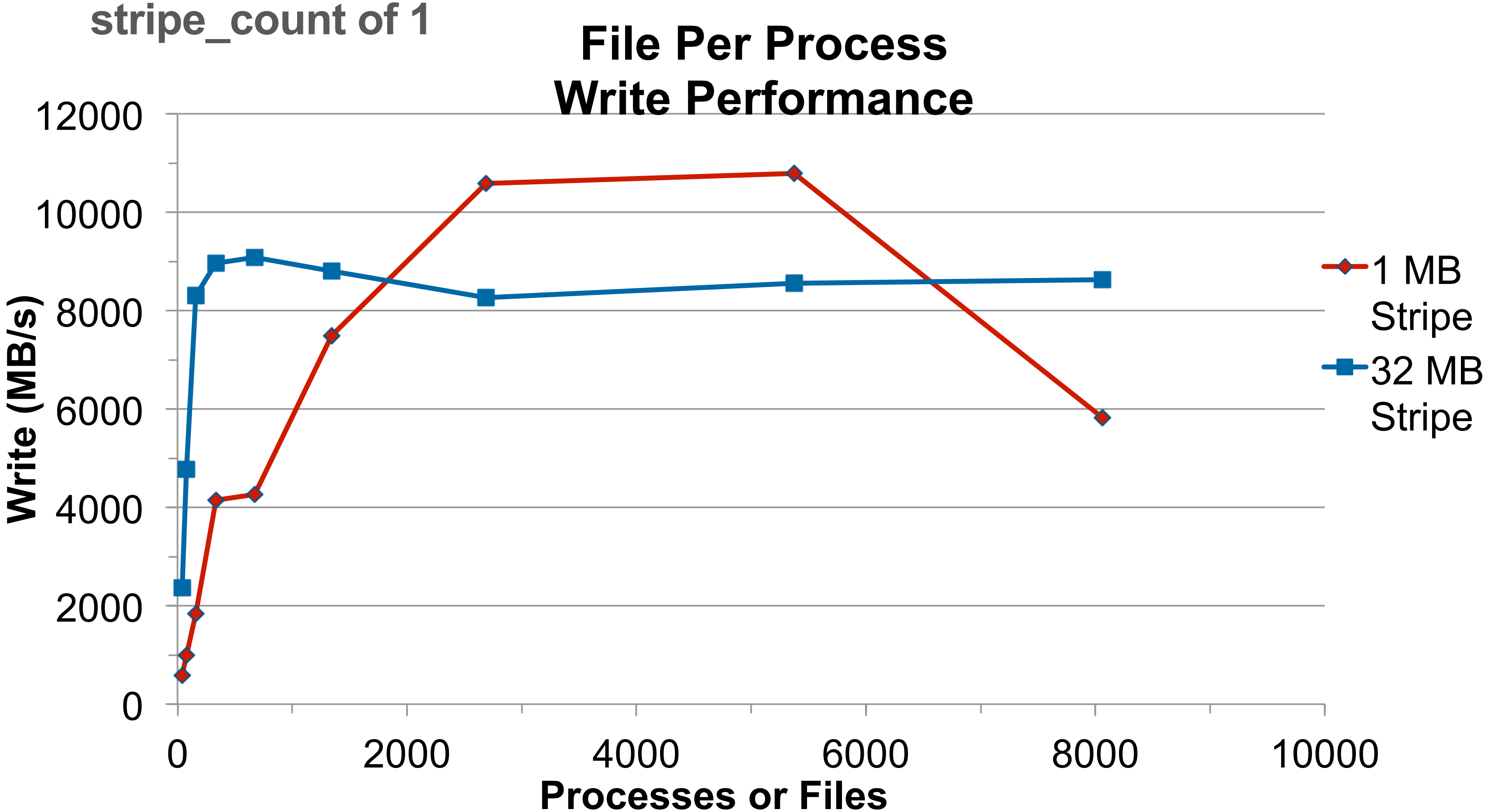
Case Study 2: Parallel I/O into a single file

- **A particular code both reads and writes a 377 GB file. Runs on 6000 cores.**
  - Total I/O volume (reads and writes) is 850 GB.
  - Utilizes parallel HDF5
- **Default Stripe settings: count =4, size=1M, index =-1.**
  - 1800 s run time (~ 30 minutes)
- **Stripe settings: count=-1, size=1M, index = –1.**
  - 625 s run time (~ 10 minutes)
- **Results**
  - 66% decrease in run time.



Case Study 3: Single File Per Process

- 128 MB per file and a 32 MB Transfer size, each file has a stripe_count of 1



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

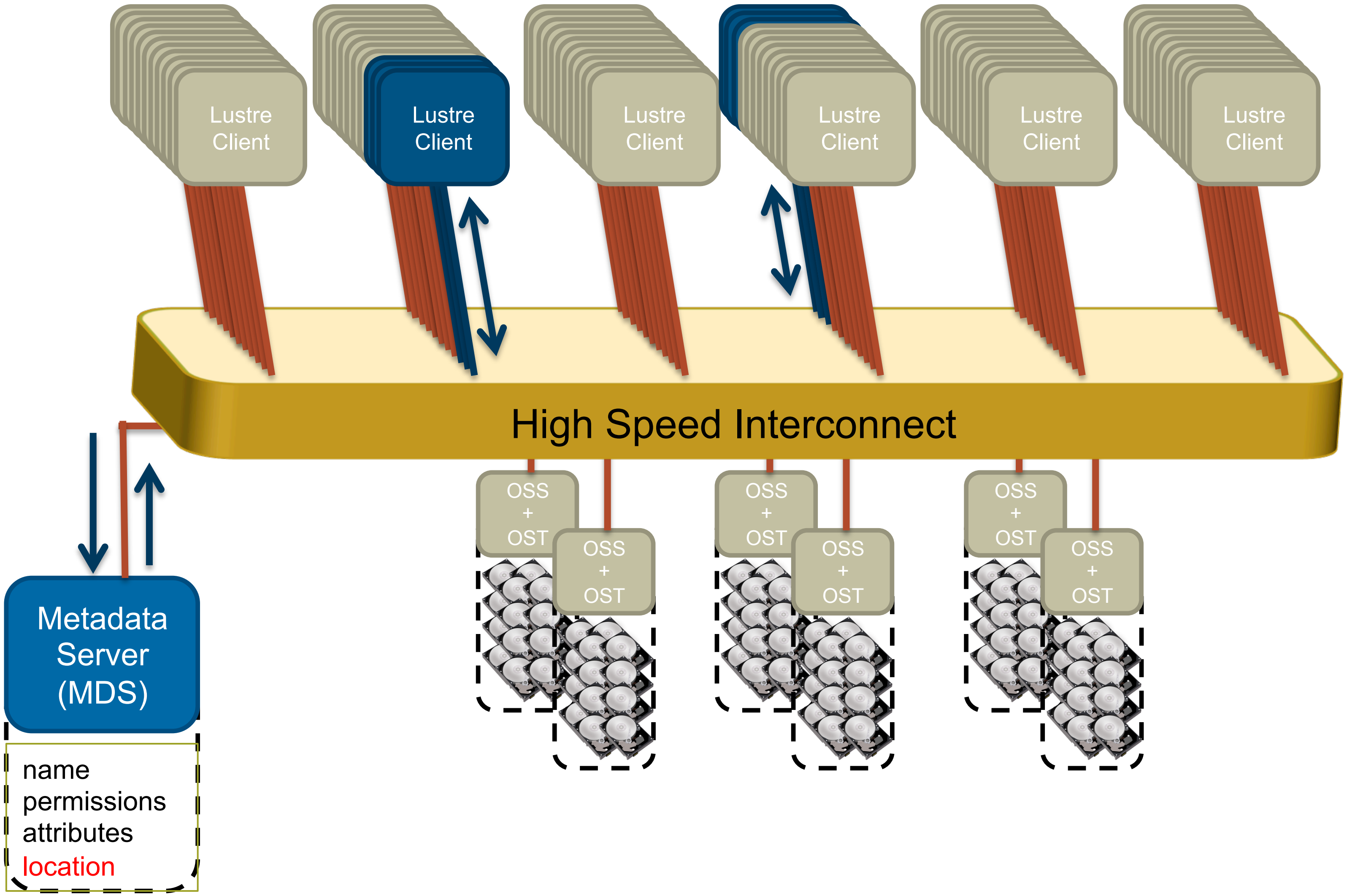
- **There are two characteristics we typically use to talk about storage or filesystem performance**
  - BANDWIDTH
  - OPERATIONS PER SECOND (IOPS)

- **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 – look after it!



Metadata Operations

- The Metatada 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

Being nice - Overview

- There are various approaches we can take to minimize the metadata server load
- Be aware of usage patterns that are not appropriate for Lustre
- Be aware of usage patterns that are most problematic
- Note that an individual application run may seem fine but in combination with other similar runs can add up to a significant problem
  - So watch for ensemble runs – many copies of the same program running simultaneously

Use Lustre for what it is designed for

- Lustre aggregates multiple storage devices providing scalable I/O for very large systems
- Sweet-spot is writing of large files
- Lustre is designed to provide a consistent (POSIX) view of the filesystem and this requires extra work to maintain

So

- Don't use Lustre for local TMPDIR
- This can be particularly problematic for large compilations

Some expensive metadata operations

stat()

- The stat operations return information on file ownerships, permissions, size, update times etc.
- To obtain the file size requires a lookup on the MDS and an enquiry for file size on each OST owning a stripe

So

- Avoid ls -l (and colour ls)
- Avoid file completion in shells
- Open and fail instead of stat/INQUIRE
- Don't stripe small files (you may have to check every OST that *might* own a part of the file)

Unnecessary file operations

Only ask Lustre for what you want

- Open a file read-only if that is what you will do

There are tools optimized for (or aware of) Lustre

- e.g. lfs find, lfs df, lustre_rsync

Some large applications read the same files on every task

- This generates a lot of metadata and data load
- Better to read on one task and use the High Speed Interconnect to move data to other tasks
  - e.g. replace "all ranks read namelist data" with "rank 0 reads namelist data and broadcasts it to all other ranks"

Shared access to single file

- **There is no problem in opening a file from multiple clients**
- **Also fine if multiple clients write to parts of file on different OSTs**
- **But expensive if multiple clients access parts of the file on the same OST**
  - New write (*or read*) causes previous client owning lock to flush
  - New client has to get lock
  - OST grants lock for portion of the file

So

- **Avoid multiple clients writing to same OST**
- **Use software (Cray MPI MPI-IO aggregation) that does this for you**

File creation and large directories

- To create a new file in a directory needs a lock on the directory
- If the directory has thousands of files then a linear search is required to check if file exists
- This search holds the lock for longer for a big directory
- Once open for a client, contents are hashed and operations are fast
- A new open on another client will force a flush and get new lock

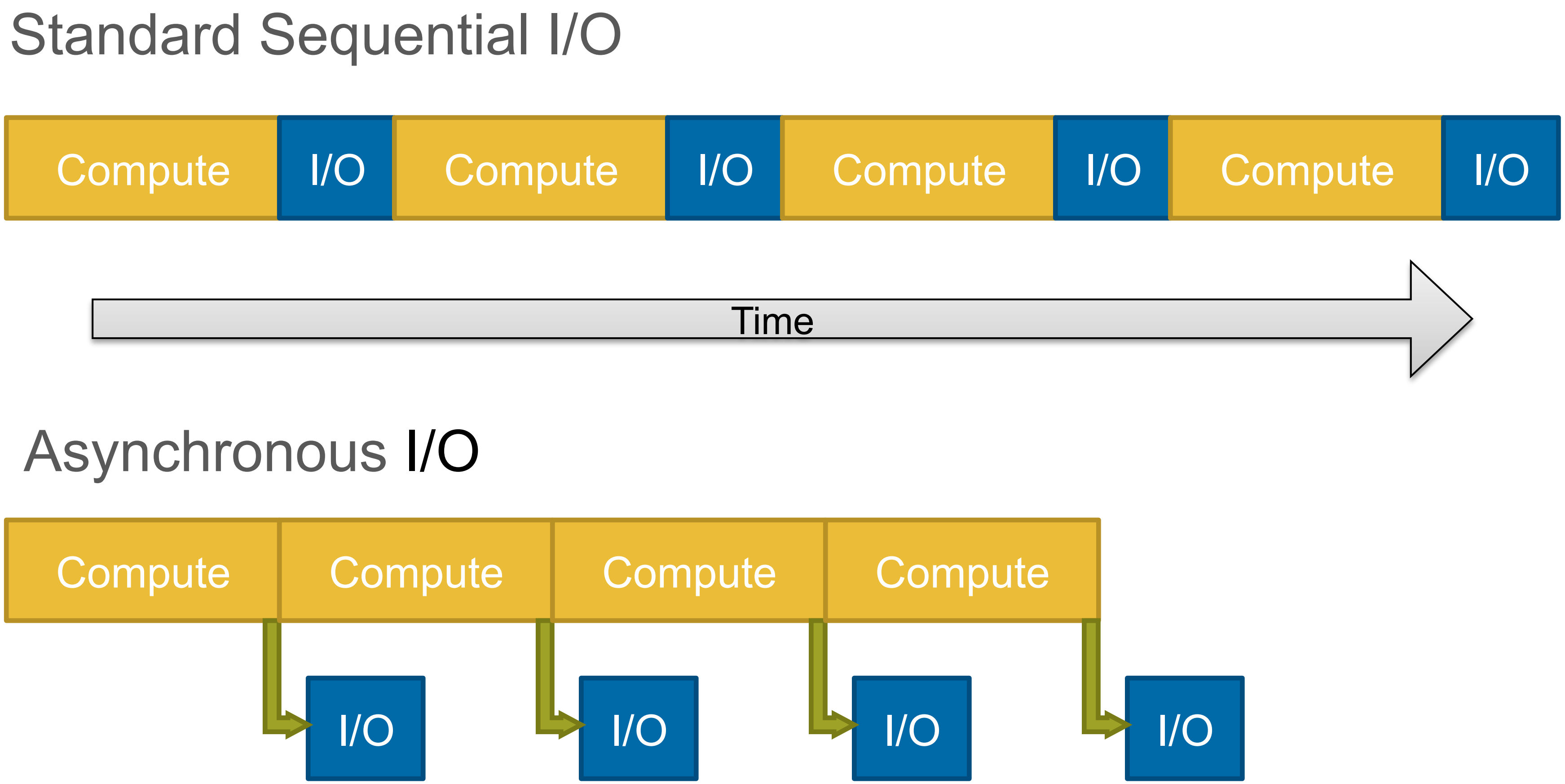
So

- Avoid large directories
- Perhaps organize directory structure by client

Asynchronous I/O

A Good Idea!

Asynchronous I/O



Asynchronous I/O

- **Good when majority of the data is output, which allows overlap with computation**
- **Double buffer arrays to allow computation to continue while data is flushed to disk**
  1. Use asynchronous POSIX calls such as aio_read, aio_write etc.
     - Only covers the I/O call itself, any packing/gathering/encoding still has to be done by the compute processors
     - Not currently supported by Lustre but calls will still function
  2. Use third party libraries
     - e.g., MPI I/O, HDF5, parallel NetCDF, IOBUF
     - Again, packing/gathering/encoding still done by compute processors
  3. 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)

I/O servers

- **Successful strategy deployed in several codes such as WRF, UM**
- **Has become more successful as number of nodes has increased**
  - Extra nodes only cost few extra 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

Naive 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
```

IOBUF

Controlling I/O Buffering in Traditional Serial I/O

Problem to be addressed

- **Application produces massive serial I/O on Lustre**
- **A generic solution for serial I/O is buffering.**
  - Temp storage of results of I/O operation in user space before writing (minimize system calls, block-align I/O operations)
  - Default Linux buffering offers no control.to the user
- **Other possible solutions:**
  - Moving part of the I/O to /tmp, which resides in the memory or is local
    - This generally involves changing the source code or namelist
    - With CCE, options for assign available
  - Changing the I/O pattern
  - Rewriting the algorithm
- **Buffering solutions (even if only .o files are available):**
  - Using buffering flags to the Intel Compiler
  - IOBUF

IOBUF

- **IOBUF** is an **I/O buffering library** officially supported by Cray that can reduce the I/O wait time for programs that read or write large files sequentially. IOBUF **intercepts I/O system calls** such as read and open and adds a layer of buffering, thus improving program performance by enabling asynchronous prefetching and caching of file data.
- **IOBUF** can also gather runtime statistics and print a summary report of I/O activity for each file (***verbose*** option)
- In general, **no program source changes** are needed in order to take advantage of **IOBUF**.
  - ***module load iobuf***
  - Relink the program
  - Set the **IOBUF_PARAMS** environment variable, for example:
    export IOBUF_PARAMS =
    "*.mtc:size=4M:count=3:verbose,*.bin:size=250K:count=3:verbose”
  - Run the program
  - For a detailed output use: `export IOBUF_PARAMS='*:verbose'\`
  - See the iobuf man page for full details

IOBUF Sample

```
IOBUF parameters: file="FILE.dat":size=2:count=0:
                                  vbuffer_count=-2147483648:prefetch=1:verbose
PE 0: File "OPTINFO.DAT"
                Calls         Seconds      Megabytes   Megabytes/sec  Avg Size
Write           19107        0.194701       1.631562        8.379836        85
Open                1        0.000317
Close               1        0.000261
Buffer Write    19107        0.187175       1.631562        8.716794        85
```

```
IOBUF parameters: file="FILE.dat":size=1048576:count=4:
                                  vbuffer_count=4096:prefetch=1:verbose
PE 0: File "OPTINFO.DAT"
                Calls         Seconds      Megabytes   Megabytes/sec  Avg Size
Write           19107        0.004624       1.631562      352.836660        85
Open                1        0.000235
Close               1        0.003174
Buffer Write        2        0.002823       1.631562      577.929822    815781
I/O Wait            2        0.002913       1.631562      560.097154
Buffers used            2 (2 MB)
Preflushes              1
```

IOBUF sample output 2

```
IOBUF parameters: file="defstriped/
serial.dat":size=1048576:count=4:vbuffer_count=4096:prefetch=1:verbose
PE 0: File "defstriped/serial.dat"
                Calls        Seconds      Megabytes   Megabytes/sec   Avg Size
Write            2048       0.580566     402.653184      693.552615     196608
Open                1       0.001288
Close               1       0.006056
Buffer Write      384       0.533518     402.653184      754.713968    1048576
I/O Wait          384       0.530056     402.653184      759.643408
Buffers used              4 (4 MB)
Preflushes            384
```

- Each file accessed on each PE will print a summary when closed.
- Users set a “buffer size” (default 1MB), transactions that are smaller are cached into one of the buffers
- Larger transactions are performed directly, bypassing the buffers.

IOBUF configuration

- Users can increase the size of buffers (`size=#[KMG]`)
- They can also add more buffers (`count=#`) this allows for more access points
- Data is automatically pre-fetched. More buffers can be pre-fetched (`count=#`) or disabled completely (`count=0`)
- Data can also be written “direct”, i.e., bypassing the OS’s internal buffering process.
- Settings controlled on a file by file basis or via pattern matching, e.g:

```
export IOBUF_PARAMS="input.dat:count=8:size=64M:direct2,\
                    out*.dat:size=1M:count=4:prefetch=0"
```

Alternative: Buffering of the Intel Compiler

- **Compiler Flag: -assume <options>**
  - [no]buffered_io
    - Equivalent to **OPEN** statement BUFFERED='YES'
    - or **environment variable** FORT_BUFFERED=TRUE
  - [no]buffered_stdout

- More control with the **OPEN** statements
  - BLOCKSIZE
    - size of the disk block I/O buffer
    - default=8192 (or 1024 if –fscomp general or all is set)
  - BUFFERCOUNT:
    - number of buffers used
    - default=1
  - Actual Memory used for buffer = BLOCKSIZE × BUFFERCOUNT

- BUFFERED=yes has precedence over –assume buffered_io, which has precedence over FORT_BUFFERED=TRUE

- Source code has to be changed for fine tuning.

Cray PAT can give I/O stats too

```
 Write Time |       Write | Write Rate | Writes |      Bytes/ |File Name[max15]
            |      MBytes | MBytes/sec |        |        Call | PE

 185.711637 | 1506.987655 |   8.114665 | 2012.0 |   785383.24 |Total
|-----------------------------------------------------------------------------
| 185.711149 | 1506.964413 |   8.114561 | 1000.0 | 1580166.72 |testit
||----------------------------------------------------------------------------
||   7.396177 |   45.639641 |   6.170707 |   30.0 | 1595221.07 |pe.20
||   7.306253 |   46.559864 |   6.372605 |   30.0 | 1627385.20 |pe.21
||   7.089236 |   45.474480 |   6.414581 |   30.0 | 1589448.27 |pe.30
||   7.014675 |   48.505741 |   6.914895 |   32.0 | 1589436.12 |pe.16
||   6.950223 |   46.723812 |   6.722635 |   30.0 | 1633115.60 |pe.17
||   6.808180 |   48.438625 |   7.114769 |   32.0 | 1587236.88 |pe.22
||   6.754414 |   47.045898 |   6.965208 |   32.0 | 1541600.00 |pe.5
||   6.703325 |   48.626842 |   7.254138 |   32.0 | 1593404.38 |pe.10
||   6.647510 |   48.204975 |   7.251584 |   32.0 | 1579580.62 |pe.26
||   6.544040 |   49.069084 |   7.498286 |   32.0 | 1607895.75 |pe.29
||   6.492357 |   49.122524 |   7.566208 |   32.0 | 1609646.88 |pe.14
||   6.314911 |   48.830154 |   7.732517 |   32.0 | 1600066.50 |pe.24
||   6.193225 |   49.240063 |   7.950634 |   32.0 | 1613498.38 |pe.15
||   6.137744 |   45.870838 |   7.473566 |   32.0 | 1503095.62 |pe.19
||   6.057450 |   47.024872 |   7.763147 |   32.0 | 1540911.00 |pe.11
||   6.027708 |   49.454807 |   8.204579 |   33.0 | 1571428.00 |pe.6
||   6.021351 |   49.716827 |   8.256757 |   33.0 | 1579753.70 |pe.27
||   6.013654 |   46.878239 |   7.795301 |   32.0 | 1536106.12 |pe.31
||   5.989393 |   49.970413 |   8.343151 |   33.0 | 1587811.39 |pe.3
||   5.893607 |   49.569782 |   8.410772 |   33.0 | 1575081.33 |pe.9
......
......
```

Cray PAT

```
||   4.854732 |   47.821522 |   9.850496 |    33.0 | 1519530.30 |pe.25
||   4.209574 |   49.621185 |  11.787696 |    33.0 | 1576714.67 |pe.28
||   3.608060 |   51.816326 |  14.361272 |    34.0 | 1598039.88 |pe.7
||   0.000000 |    0.000000 |         -- |     0.0 |         -- |pe.0
||
==============================================================================
|    0.000355 |     0.022888 |  64.504298 | 1000.0 |       24.00 |testit_index
||-----------------------------------------------------------------------------
-
||   0.000355 |    0.022888 |  64.504298 | 1000.0 |      24.00 |pe.0
||   0.000000 |    0.000000 |         -- |    0.0 |         -- |pe.31
||   0.000000 |    0.000000 |         -- |    0.0 |         -- |pe.30
||   0.000000 |    0.000000 |         -- |    0.0 |         -- |pe.29
||   0.000000 |    0.000000 |         -- |    0.0 |         -- |pe.28
||   0.000000 |    0.000000 |         -- |    0.0 |         -- |pe.27
||   0.000000 |    0.000000 |         -- |    0.0 |         -- |pe.26
||   0.000000 |    0.000000 |         -- |    0.0 |         -- |pe.25
||   0.000000 |    0.000000 |         -- |    0.0 |         -- |pe.24
||   0.000000 |    0.000000 |         -- |    0.0 |         -- |pe.23
```

pat_build –w –g io –g mpi io_tester
pat_report –s pe=ALL *.xf

Cray MPI-IO Layer

Data Aggregation and Data Sieving

MPI I/O

- **The MPI-2.0 standard provides a standardised interface for reading and writing data to disk in parallel. Commonly referred to as MPI I/O**
- **Full integration with other parts of the MPI standard allows users to use derived types to complete complex tasks with relative ease.**
- **Can automatically handle portability issues such as byte-ordering and native and standardised data formats.**
- **Available as part of the cray-mpich library on XC40, commonly referred to as Cray MPI-IO.**
  - Fully optimised and integrated with underlying Lustre file-system.

Collective Buffering & Data Sieving

Two Techniques: Sieving and Aggregation

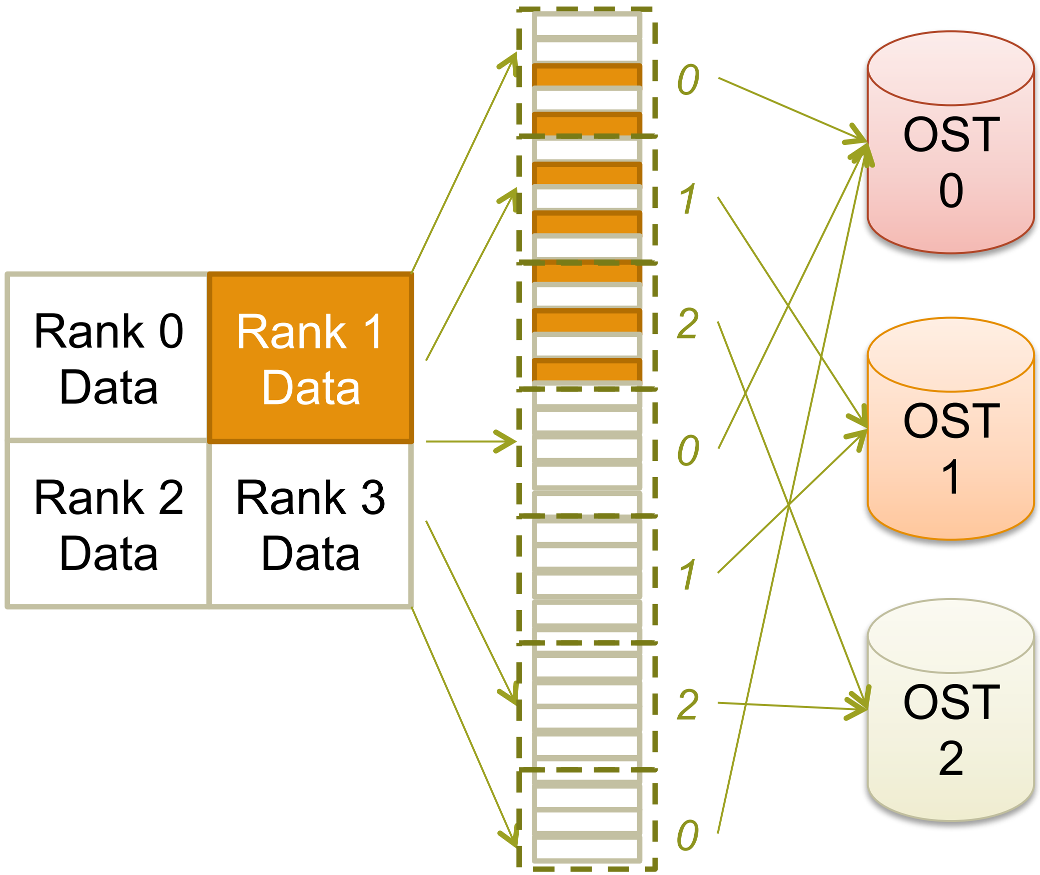
- **Data sieving is used to combine lots of small accesses into a single larger one**
  - Reducing # of operations important (latency)
  - A system buffer/cache is one example
- **Aggregation/Collective Buffering refers to the concept of moving data through intermediate nodes**
  - Different numbers of nodes performing I/O (transparent to the user)
- **Both techniques are used by MPI-IO and triggered with HINTS**

Data Sieving

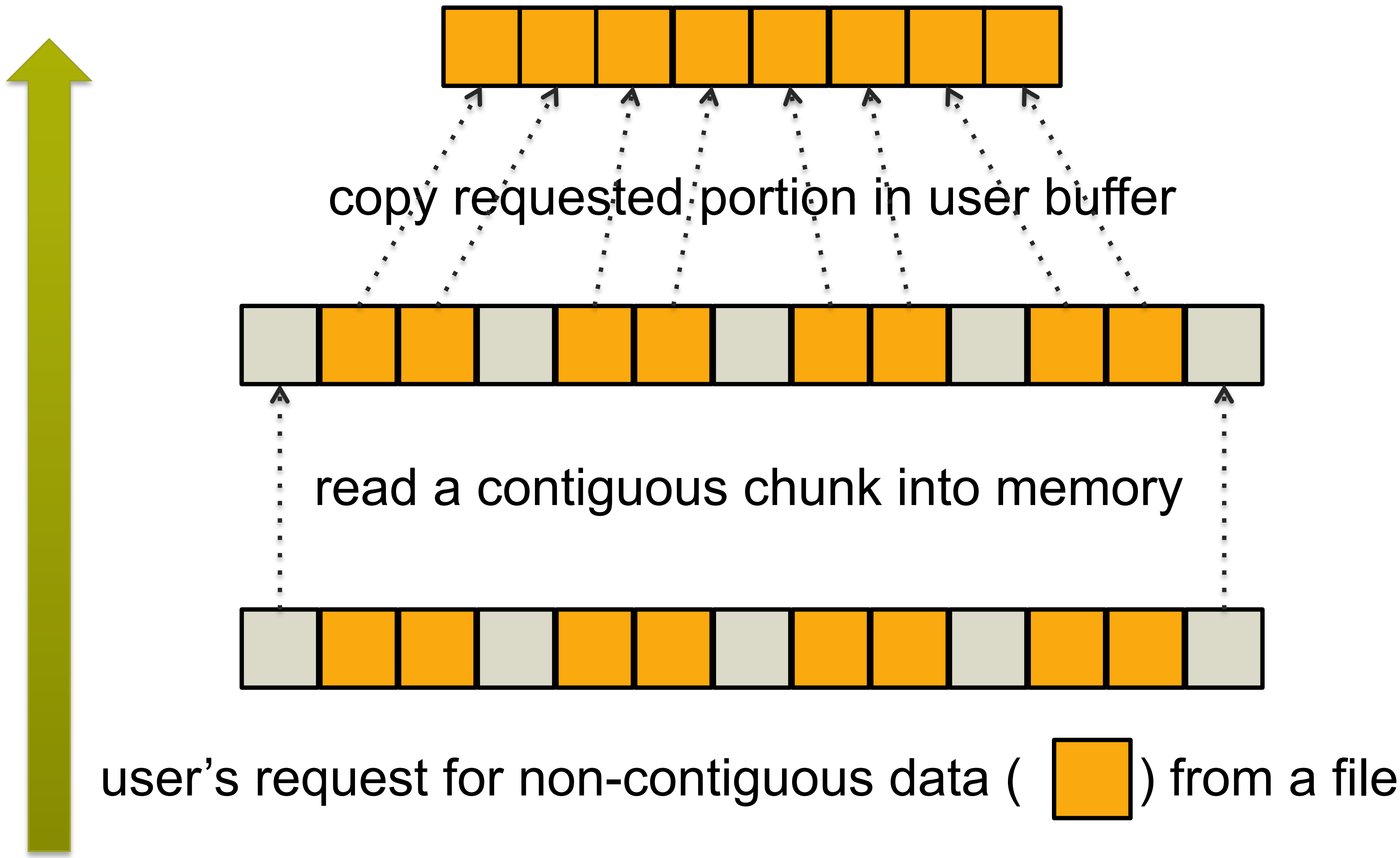
- "Read/Write Gaps" occur when the data is not accessed contiguously from the file.
- This limits the total bandwidth rate as each access requires separate calls and may cause additional seek time on HDD storage.
- Overall performance can be improved by minimising the number of read/write gaps.
- The Cray MPI-IO library will attempt to use data sieving to automatically combine multiple smaller operations into fewer larger operations.

Strided file access

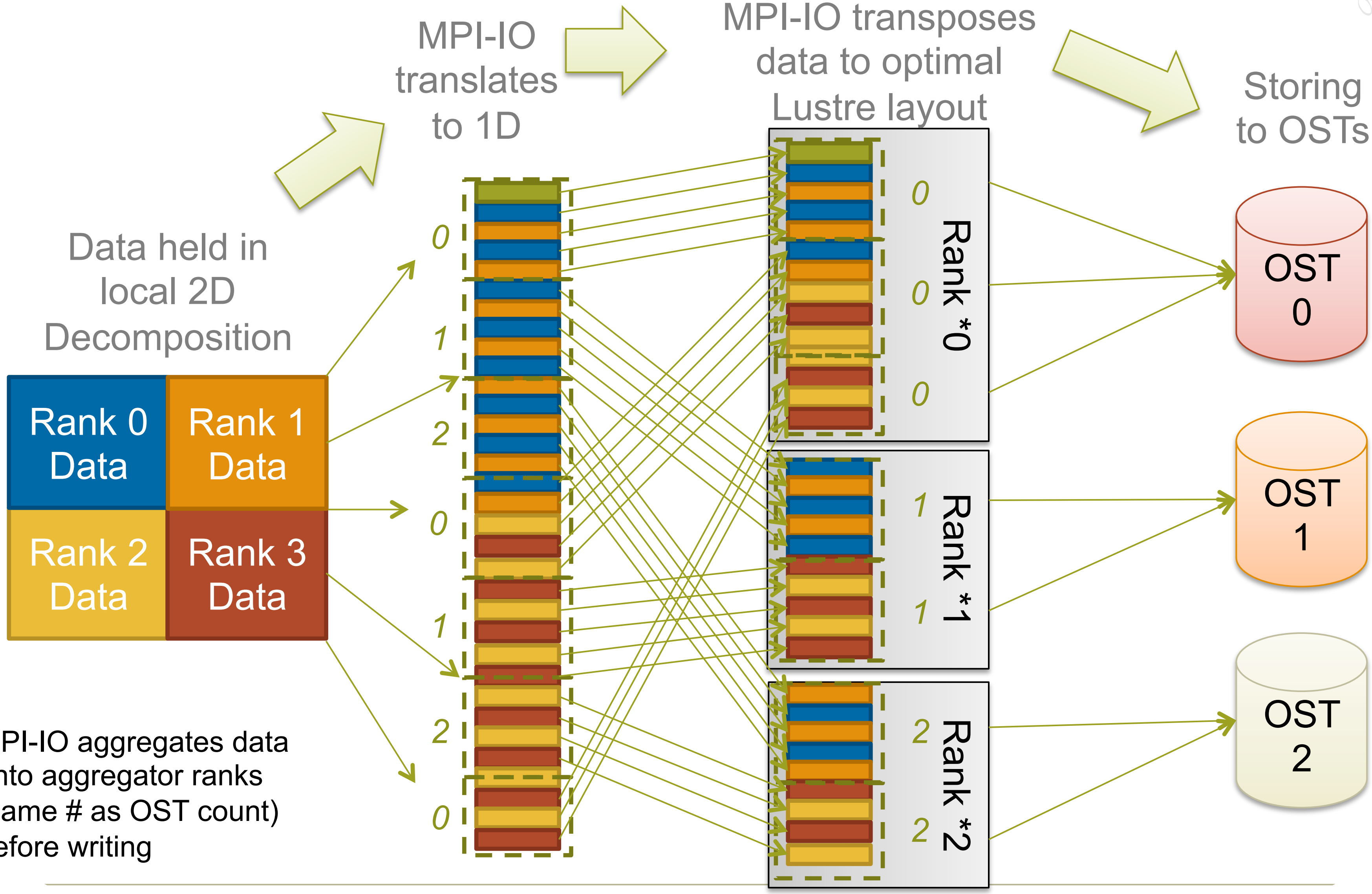
- Focusing on a rank we can see that it will potentially end up writing strided data to each OST.
- This is likely to incur penalties due to extent locking on each of the OSTs.
- It also prevents optimal performance of HDD block devices that comes from writing contiguous blocks of data



Data Sieving

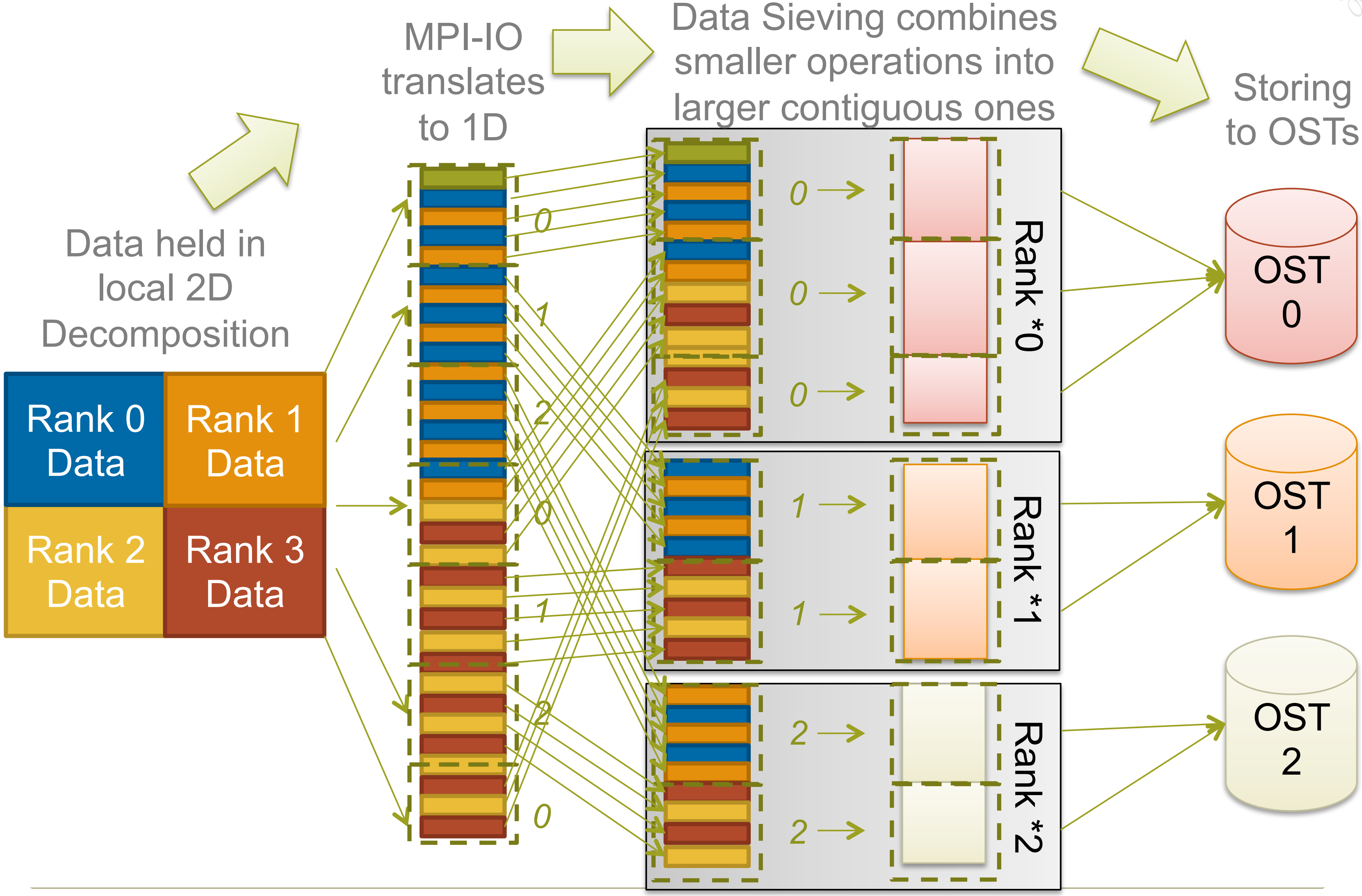


Writing structured data to disk



MPI-IO aggregates data onto aggregator ranks (same # as OST count) before writing

Data Sieving



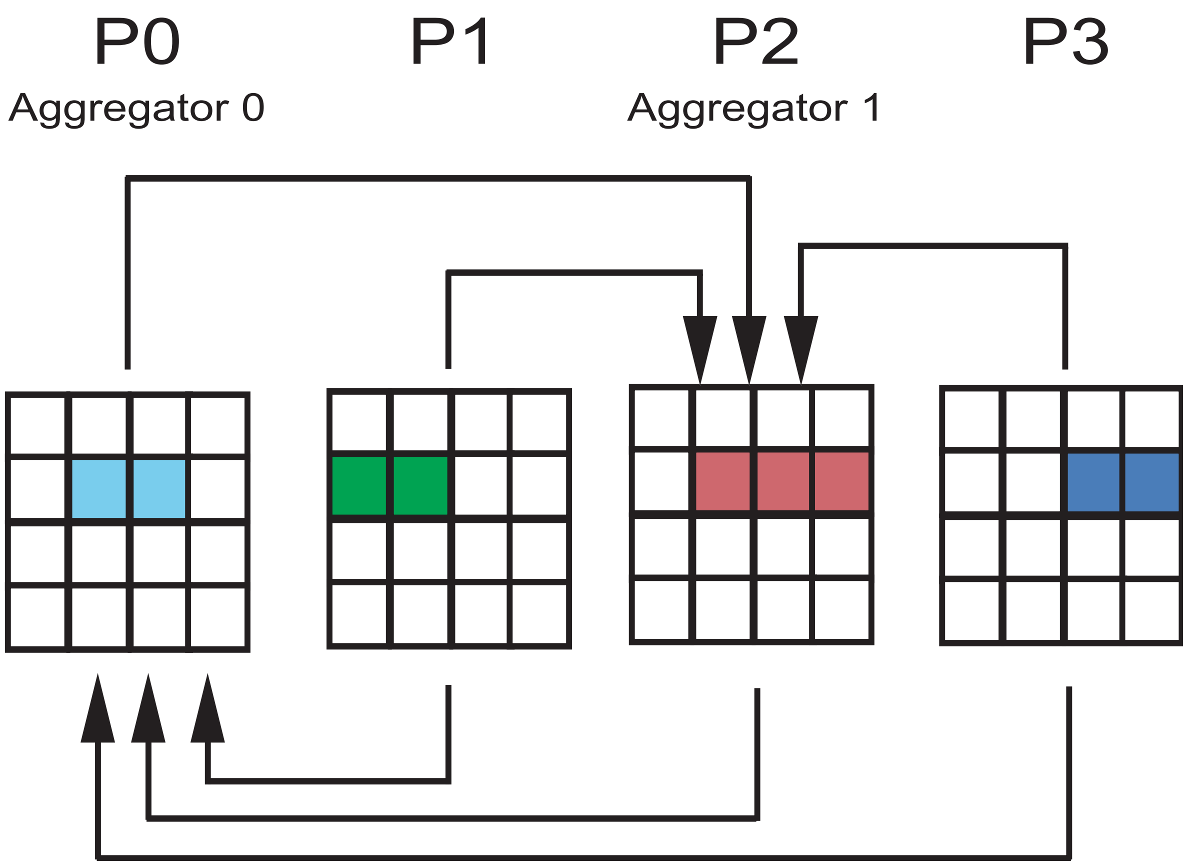
Managing Collective Buffering

- **The Cray MPI-IO library will automatically perform collective buffering of collective MPI-IO calls. There are two algorithms controlled by the value of `MPICH_MPIIO_CB_ALIGN=[0|2]`**
  - `0` : distribute data equally across all aggregators regardless of Lustre stripe settings (inefficient if data in a single stripe or small number of stripes)
  - `2` (default): Divides data into Lustre stripe-sized pieces and assigns them to collective buffering nodes such that each node always and exclusively accesses the same set of stripes.

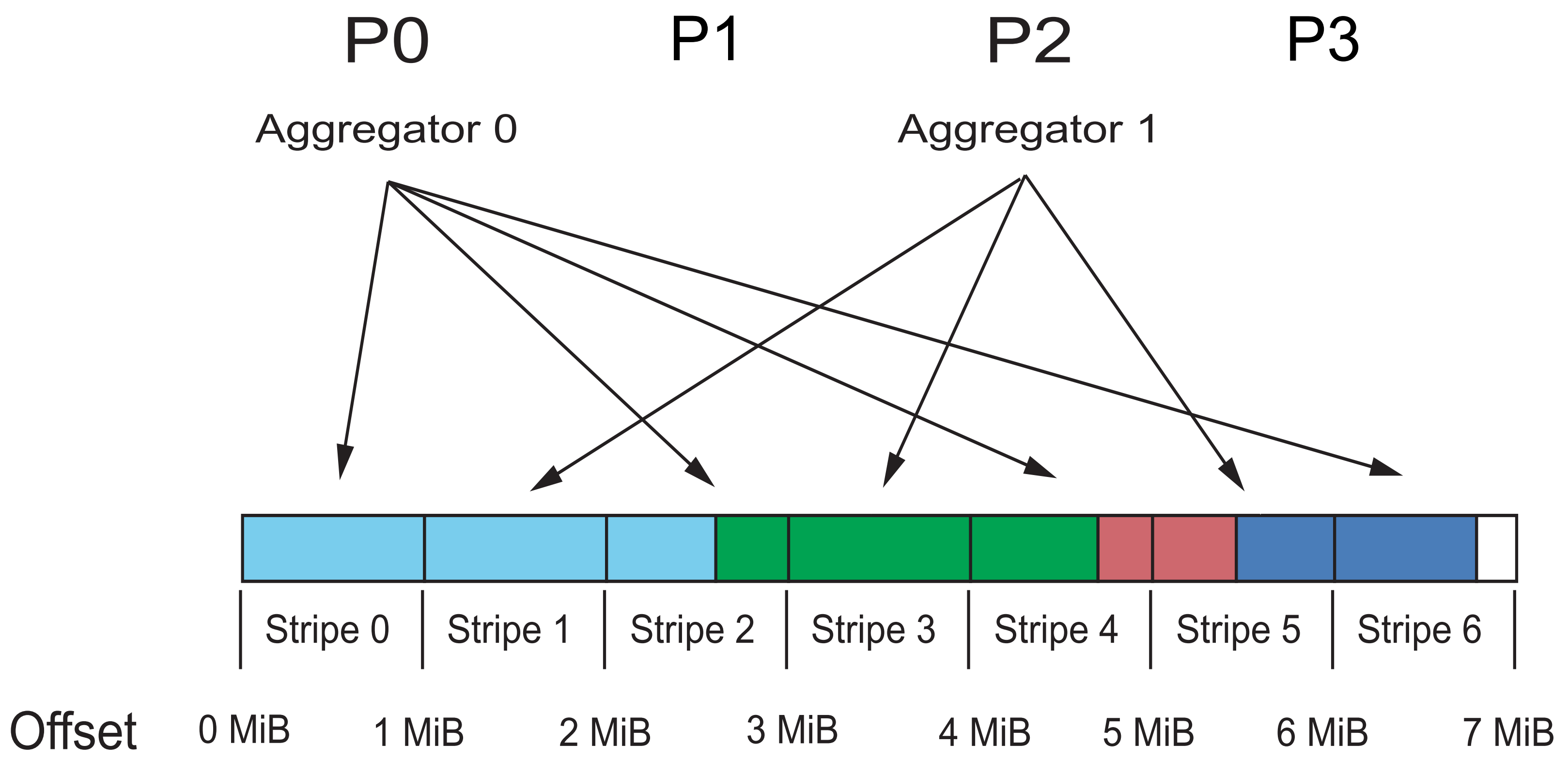
- **The default behaviour (`MPICH_MPIIO_CB_ALIGN=2`) will:**
  - Automatically set the number of aggregators to the number of stripes
  - Attempt to place each aggregator on its own node
  - Our experience is that the default aligned algorithm achieves best performance in most circumstances.

- **So in most cases it is only necessary to change the Lustre stripe settings to optimise performance**

Collective buffering: aggregating data

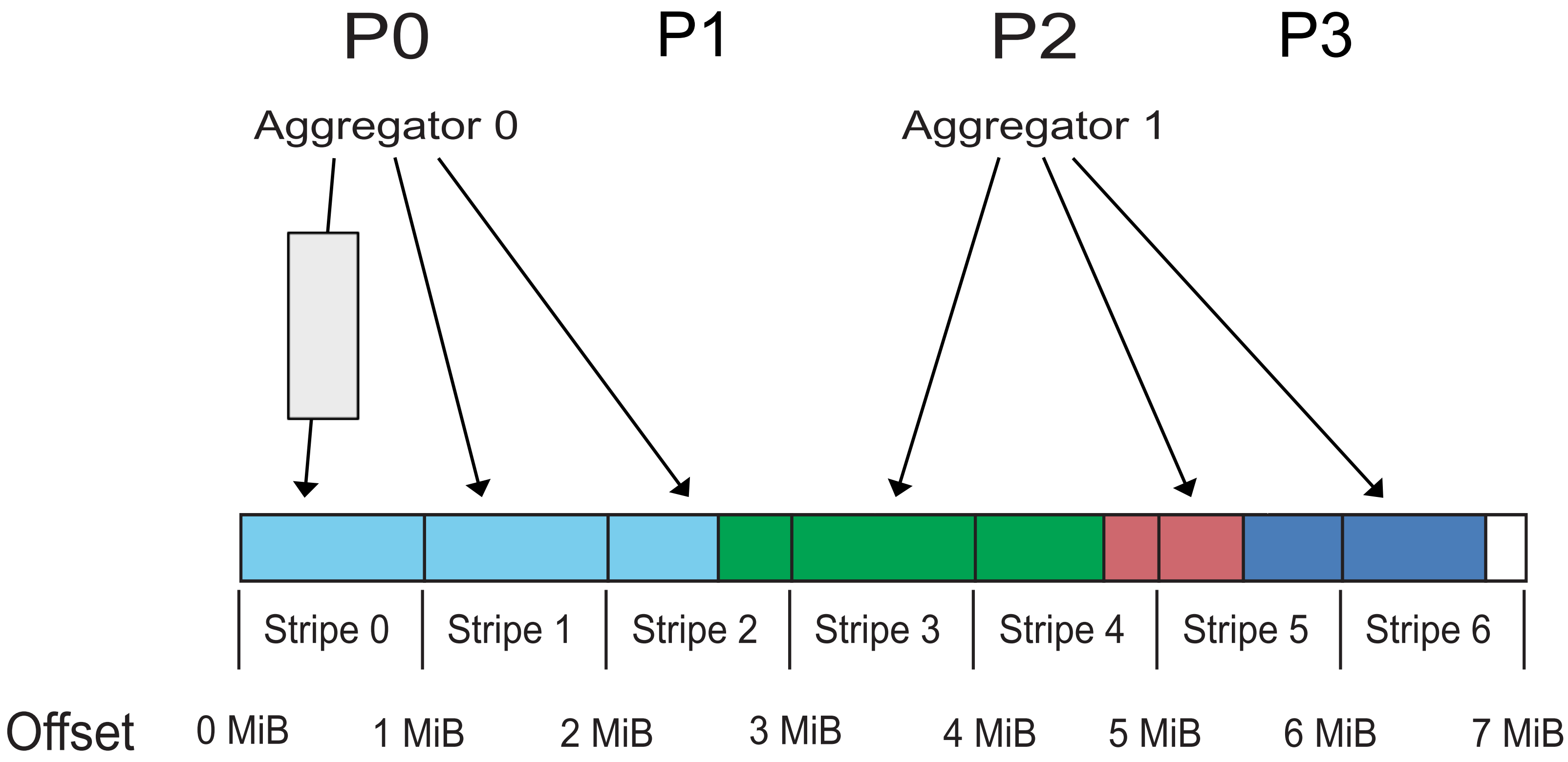


Collective Buffering: writing data CB=0



`CB=0` : distribute data equally across all aggregators regardless of Lustre stripe settings

Collective Buffering writing data CB=2



CB=2 : Divides data into Lustre stripe-sized pieces & assigns them to collective buffering nodes so each node always and exclusively accesses the same set of OSTs

Collective vs independent calls

- **Opening a file via MPI I/O is a collective operation that must be performed by all members of a supplied communicator.**

- **However, many individual file operations have two versions:**
  - A collective version which must be performed by all members of the supplied communicator
  - An independent version which can be performed ad-hoc by any processor at any time. This is akin to standard POSIX I/O, however includes MPI data handling syntactic sugar.

- **It is only during collective calls that the MPI-IO library can perform required optimisations. Independent I/O is usually no more (or less) efficient than POSIX equivalents.**

MPI I/O interaction with Lustre

- **Included in the Cray MPT library (man intro_mpi)**
- **Environmental variables used to help MPI-IO optimize I/O performance:**
  - MPICH_MPIIO_CB_ALIGN (default 2) sets collective buffering behavior
  - MPICH_MPIIO_HINTS can set striping_factor and striping_unit for files created with MPI I/O
  - 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**

MPI-IO Hints

The MPI I/O interface provides a mechanism for providing additional information about how to the MPI-IO layer should access files.

These are controlled via MPI-IO HINTS, either via calls in the MPI API or passed via an environment variable. All hints can be set on a file-by-file basis.

On the Cray XC40 the first most useful are:

- `striping_factor` – Number of lustre stripes
- `striping_unit` – Size of lustre stripes in bytes

These set the file's Lustre properties when it is created by an MPI-IO API call.

* Note these require `MPICH_MPIIO_CB_ALIGN` to be set to its default value of 2.

Setting hints via environment variables

Hints can be applied to all files, specific files, or pattern files, e.g.

```
# Set all MPI-IO files to 4 x 4m stripes
MPICH_MPIIO_HINTS="*:striping_factor=4:striping_unit=4194304"

# Set all .dat files to 8 x 1m stripes
MPICH_MPIIO_HINTS="*.dat:striping_factor=8:striping_unit=1048576"

# Set default to 4 x 4m and all *.dat files to 8 x 1
MPICH_MPIIO_HINTS="*:striping_factor=4:striping_unit=4194304, \
                 =*.dat:striping_factor=8:striping_unit=1048576"
```

Displaying hints

The MPI-IO library can print out the “hint” values that are being using by each file when it is opened during a run. This is controlled by setting the runtime environment variable:

```
export MPICH_MPIIO_HINT_DISPLAY=1
```

The reported is generated by the PE with rank 0 in the relevant communicator and is printed to `stderr`.

```
PE 0: MPICH/MPIIO environment settings:
PE 0:   MPICH_MPIIO_HINTS_DISPLAY   = 1
PE 0:   MPICH_MPIIO_HINTS           = NULL
PE 0:   MPICH_MPIIO_ABORT_ON_RW_ERROR = disable
PE 0:   MPICH_MPIIO_CB_ALIGN        = 2
PE 0:   MPIIO hints for file1:
…
          direct_io                 = false
          aggregator_placement_stride = -1
…
```

More diagnostics

```
export MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
```

```
Aggregator Placement for /lus/scratch/myfile
              RankReorderMethod=3  AggPlacementStride=-1
               AGG    Rank       nid
               ----  ------  --------
                  0       0  nid00578
                  1       4  nid00579
                  2       1  nid00606
                  3       5  nid00607
                  4       2  nid00578
                  5       6  nid00579
                  6       3  nid00606
                  7       7  nid00607
```

Understanding MPI-IO Stats

The MPI library can provide stats on how many reads and writes were performed in system sized gaps. Adding:

```
export MPICH_MPIIO_STATS=1
```

to runtime environment variables will generate summary output on each PE.

```
+------------------------------------------------------+
| MPIIO write access patterns for file1
|   independent writes      = 0
|   collective writes       = 24
|   system writes           = 4916
|   stripe sized writes     = 4915
|   total bytes for writes  = 25769803776 = 24576 MiB = 24 GiB
|   ave system write size   = 5242026
|   number of write gaps    = 0
|   ave write gap size      = NA
+------------------------------------------------------+
```

In more detail

- *Independent writes* – the number of writes performed by independent call to the MPI-IO library
- *Collective writes* – the number of writes performed in collective MPI-IO calls.
- *System writes* – the number of POSIX write operations the MPI-IO translated the calls into
- *Total bytes for writes* – The amount of data written to the file
- *Avg system write size* – The average size of each POSIX write operation
- *Number of write gaps* – the number of gaps/seeks between POSIX write operations
- *Avg write gap size* – the average size of jumps/seek operations.

Recognising Poor Performance

```
+-----------------------------------------------------------+
| MPIIO write access patterns for unstriped/mpiionative.dat
|   independent writes      = 64
|   collective writes       = 0
|   system writes           = 1048576
|   stripe sized writes     = 0
|   total bytes for writes  = 1073741824 = 1024 MiB = 1 GiB
|   ave system write size   = 1024
|   number of write gaps    = 1048512
|   ave write gap size      = 15264
+-----------------------------------------------------------+
```

No Collective writes

Large numbers of system writes

No stripe sized writes

Ave system write size is small

Large number of write gaps

This is a simple example for 3D decomposed array. Independent MPI-IO writes are used in place of collectives.

0.005 GiB/s

Recognising Good Performance

```
+---------------------------------------------------------+
| MPIIO write access patterns for striped/mpiionative.dat
|   independent writes     = 0
|   collective writes      = 64
|   system writes          = 1024
|   stripe sized writes    = 1024
|   total bytes for writes = 1073741824 = 1024 MiB = 1 GiB
|   ave system write size  = 1048576
|   number of write gaps   = 0
|   ave write gap size     = NA
+---------------------------------------------------------+
```

No Independent writes

High % of stripe sized writes

Ave system ~= stripe size

No write gaps

This same simple example for 3D decomposed array. Now using collective MPI-IO writes:

1.41 GiB/s

Next steps with MPI/IO

- Cray document: **“Getting Started with MPI-IO”** S–2490–40
- Google search gives great tutorials/guides on using MPI-IO
- Parallel NetCDF and HDF5 are both built on top of MPI-IO
- More detailed information coming up....if we have time!