

# How To write (101) data

on a large system like a Cray XT called HECToR

Martyn Foster  
mfoster@cray.com

# Topics

- Why does this matter?
- IO architecture on the Cray XT
  - ✦ Hardware Architecture
  - ✦ Language layers
  - ✦ OS layers
  - ✦ Node layers
- Lustre
- Tuning
  - ✦ Application patterns
  - ✦ Lustre Optimizations
  - ✦ MPI-IO (brief notes)

# Common mindsets

- Asking who is interested in I/O optimization people will fall into one (or more) camps:
  - ✿ It doesn't affect me – I compute for 12hrs on 8192 cores and to compute “42” thus IO is not important to me!
  - ✿ Disks are slow so there is nothing I can do about it so optimization is irrelevant
  - ✿ I do I/O but I have no idea how long it takes nor do I care.
  - ✿ I know I/O does not scale and I'm not here to fix it
  - ✿ I/O has never really been a problem until now
    - ▶ though I have tried to run on 8192 cores before....
  - ✿ I run for 12 hrs and it takes 20 minutes to create a checkpoint file and this seems insignificant.
  - ✿ If it is expensive I will do it less often.
  - ✿ My I/O works well – I dump my 2GB dataset in 2 minutes this is better than I see elsewhere.

# Answers

## ■ Everyone should care

- ✿ Either you affect everyone
- ✿ Or others affect you
- ✿ Being a good citizen on a multiuser system is important!

## ■ I/O is a shared resource

- ✿ On Cray XT4 disk resources are shared
  - ▶ /tmp is memory so not very big nor permanent.
  - ▶ Timings may vary due to other activity

# What Should I Target

- I don't care about what order data reaches disk and how it is delivered.
  - ✿ All that matters is performance
  - ✿ Good – measure performance in GB/s
- Format and structure and portability matter but I've tried to make my code use large contiguous blocks
  - ✿ Typical - measure performance in 100's MB/s
- None of the above apply
  - ✿ Poor – You may see rates in 10's MB/s
  - ✿ You should look at the I/O pattern in your code
- I have no control of my IO
  - ✿ Modify library/app behaviour via the environment/config files
    - ▶ Especially 3<sup>rd</sup> party libraries
  - ✿ Can HECToR CSE help?

# IO Architecture

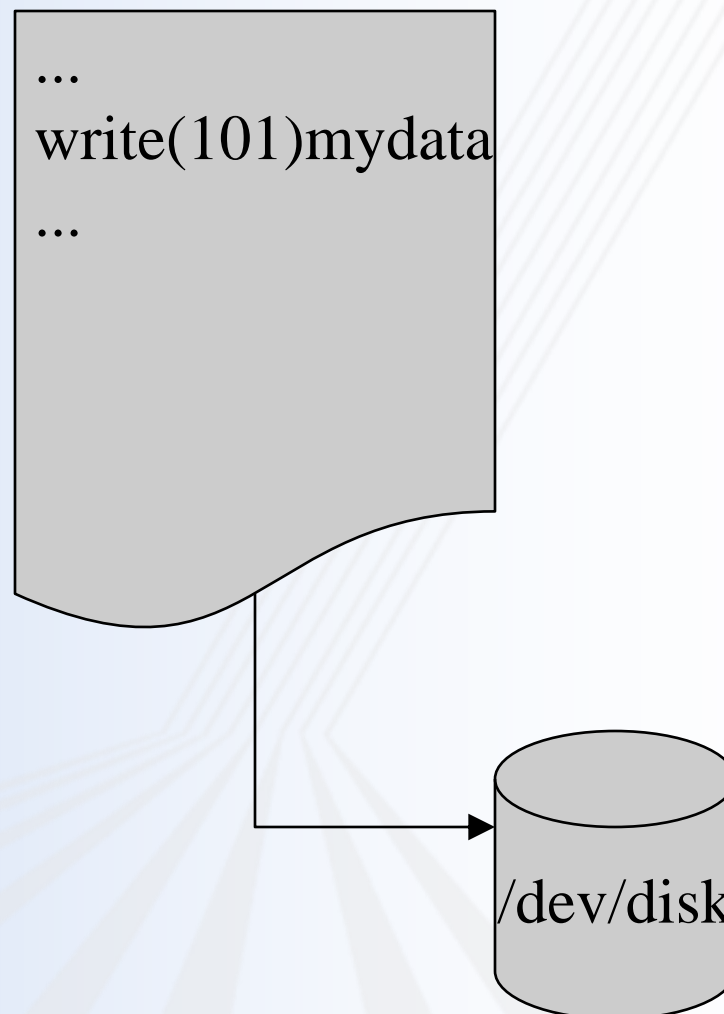
- A combination of multiple hardware and software entities.
  - ✦ Lets unravel the complexity
- Focus on the specific configuration of HECToR

# UNIX

- Unix presents a simple abstraction for IO
  - ✿ Everything is a file- even disks
  - ✿ Virtualization layers provide abstraction
  - ✿ All IO devices look pretty much the same
  - ✿ Your code just 'works'

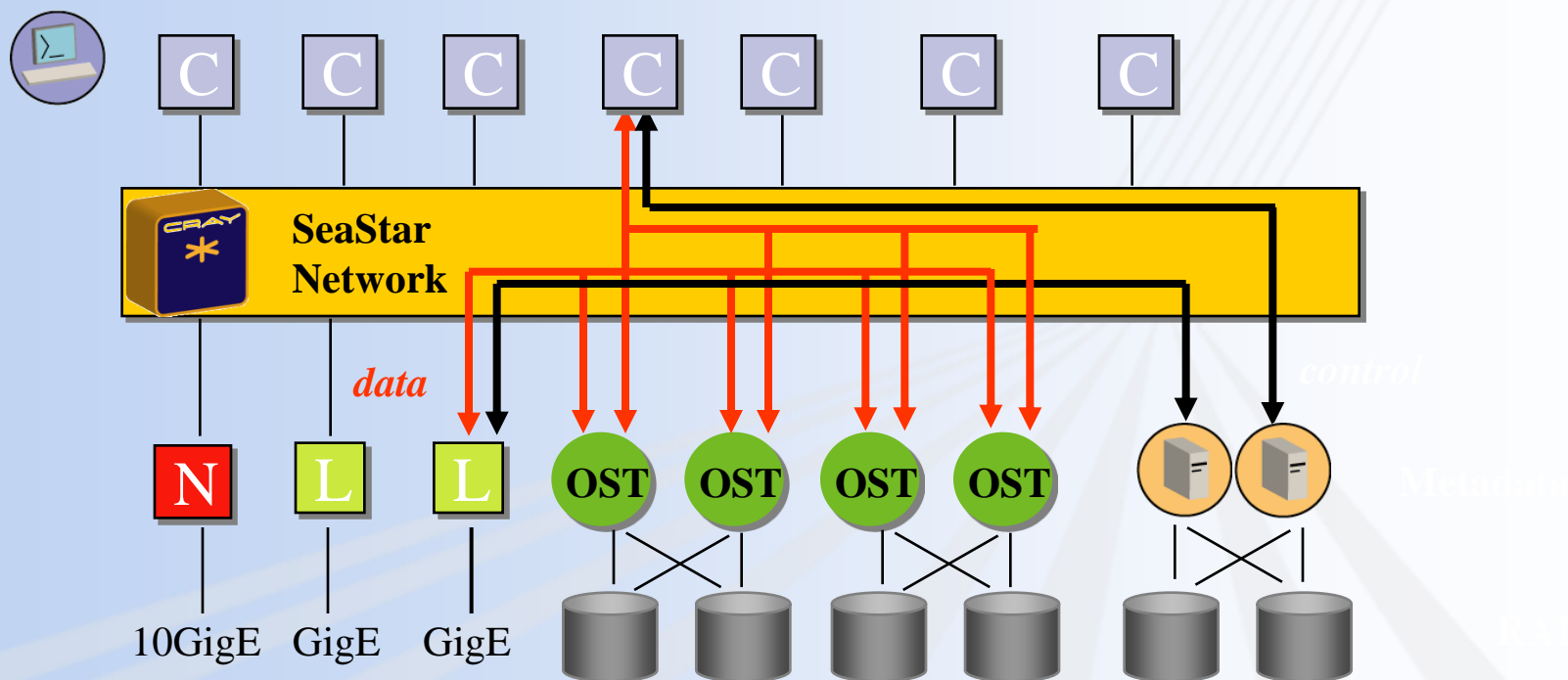
## ■ BUT

- ✿ Complexity is present, just hidden
- ✿ POSIX Semantics: Not much said about parallel file systems or concurrent access



# Terminology

- I/O nodes: Cray XT Service nodes that run Lustre processes (OST - Object Storage Target)
  - ⚙️ The node is also called an OSS (Object Storage Server)
- MDS: Metadata server, a server that runs 'somewhere' that provides metadata (file sizes, permissions), and arbitrates access.



# The on node software environment

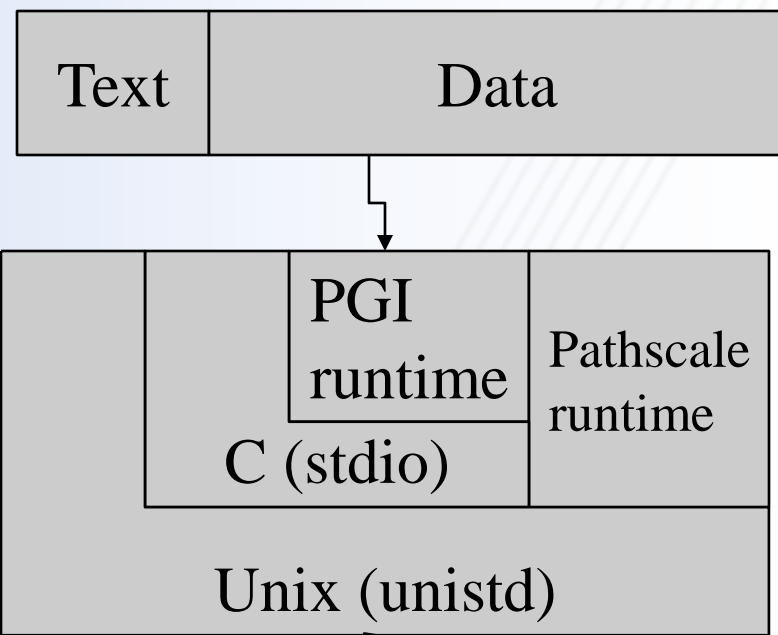
Most programmers write IO in one of 3 portable interfaces available to the programmer.

- F, C or Unix

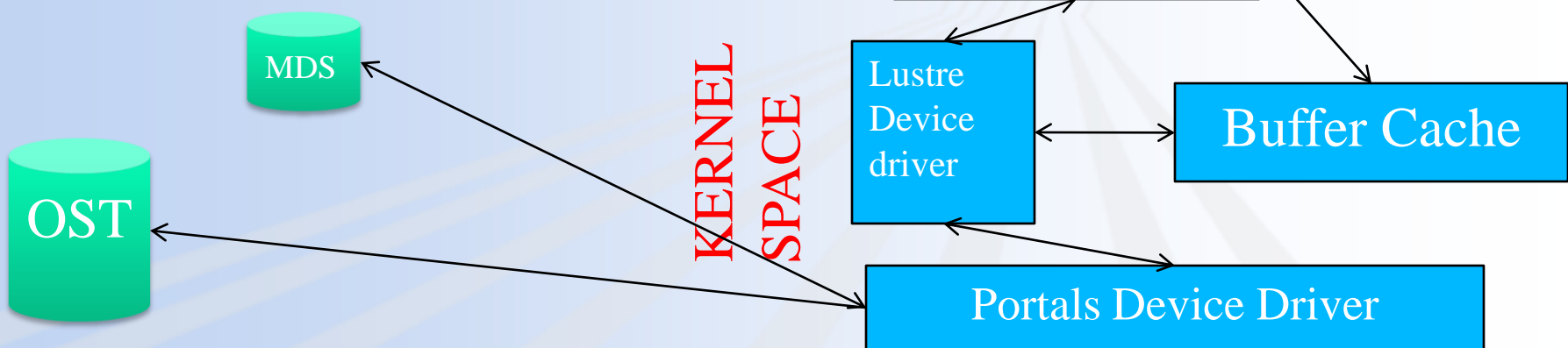
- Further abstraction from C++, MPI-IO, helper libraries.

Data flows through libraries, buffer cache, and device drivers

USER SPACE



KERNEL SPACE



# Other layers...

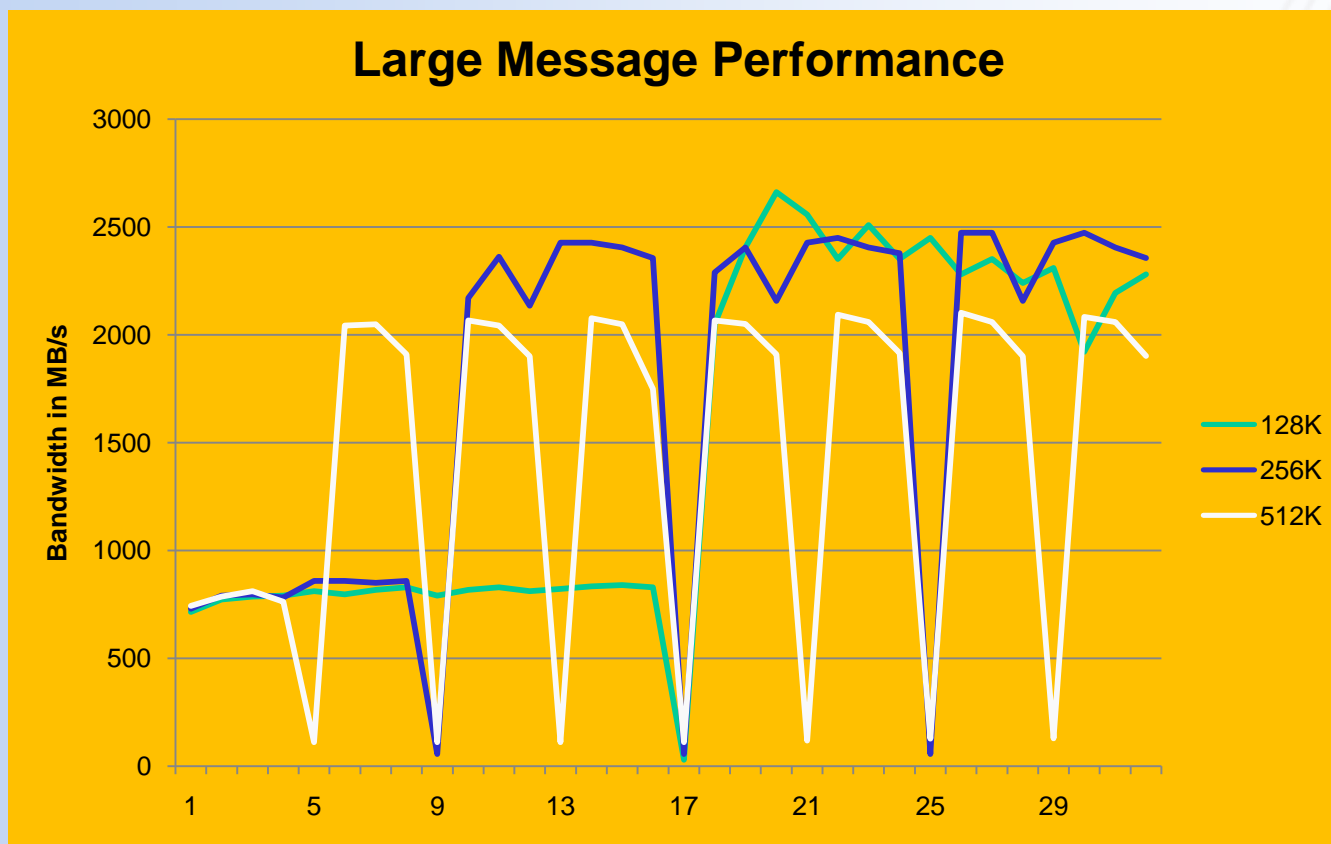
- Additional layers can be added
- IOBUFF
  - ✱ Less important on Linux, but historically valuable on catamount
  - ✱ May migrate into MPI-IO over time
- MPI-IO
  - ✱ Doesn't use language based layers
  - ✱ MPI-IO
    - ▶ Does data shuffling
    - ▶ Is responsible for locking and concurrency issues
    - ▶ Ultimately calls UNIX IO layer
- Asynchronous UNIX interface
  - ✱ `iread/iwrite` replace `read/write` etc
  - ✱ Quick return, with completion determined by subsequent "wait" call

# Library issues (Fortran)

- Performance affected by the parameters given to open
  - ✿ Use large records
- Fortran implements its own buffering policy to avoid multiple small writes resulting in multiple system calls.
  - ✿ This can be unhelpful when we are performing large IOs
  - ✿ Implies additional data copy
- PGI:
  - ✿ 2MB is the default system buffer for the unit.
  - ✿ Use `setvbuf` (and `setvbuf3f`) to specify more, less or no buffering
    - ▶ Surprising performance effect....

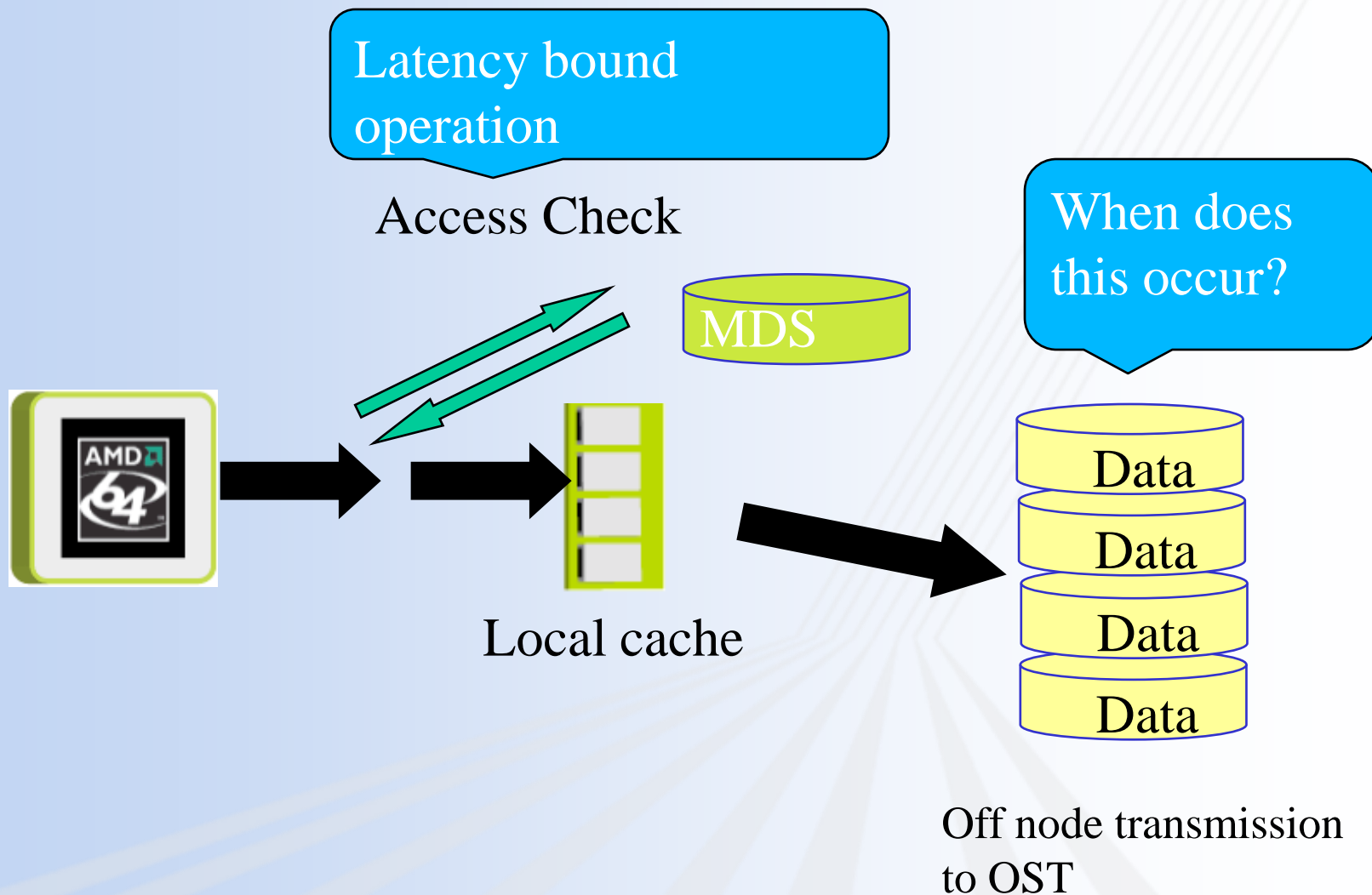
# Fortran buffering: illustratively

- Performance of repeated IOs of different sizes



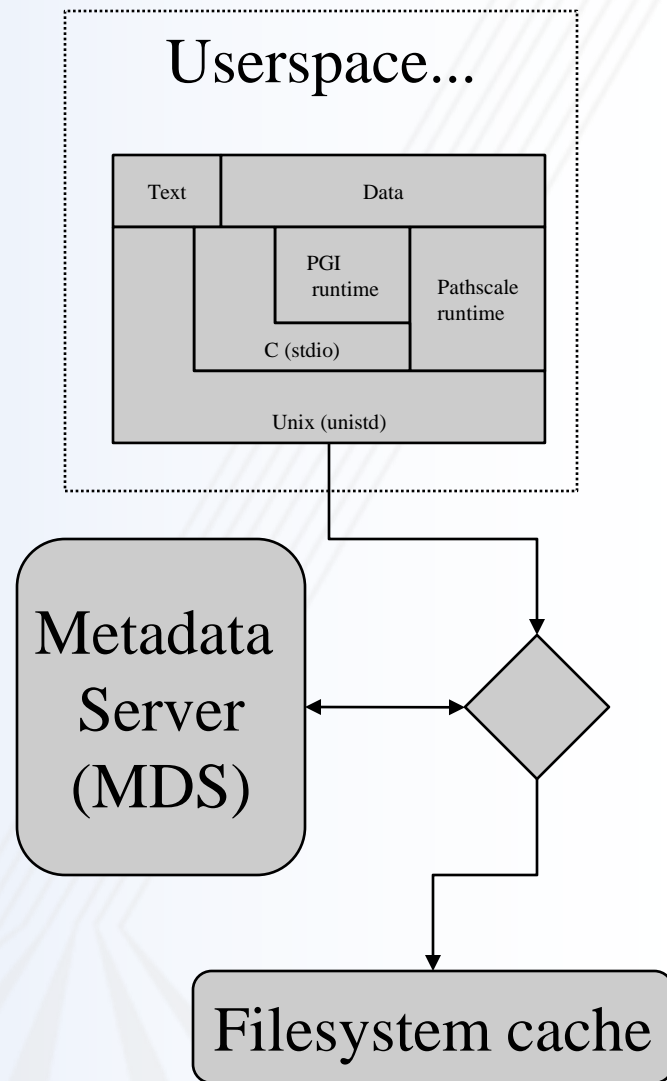
- 2 MB language level buffering becomes visible!

# The slightly more complicated view



# Metadata

- When using a parallel filesystem access is arbitrated
  - ✿ “I want to... “ requests are sent to the MDS
  - ✿ Transaction takes place with a server on another XT node via portals.
  - ✿ MDS is involved with open/close/... Operations
- Operations are latent
- Operations serialize
- Minimize operations that involve MDS
- This only applies to Lustre IO
  - ✿ stdin/stdout/stderr are separately handled and sent to ALPS/aprun

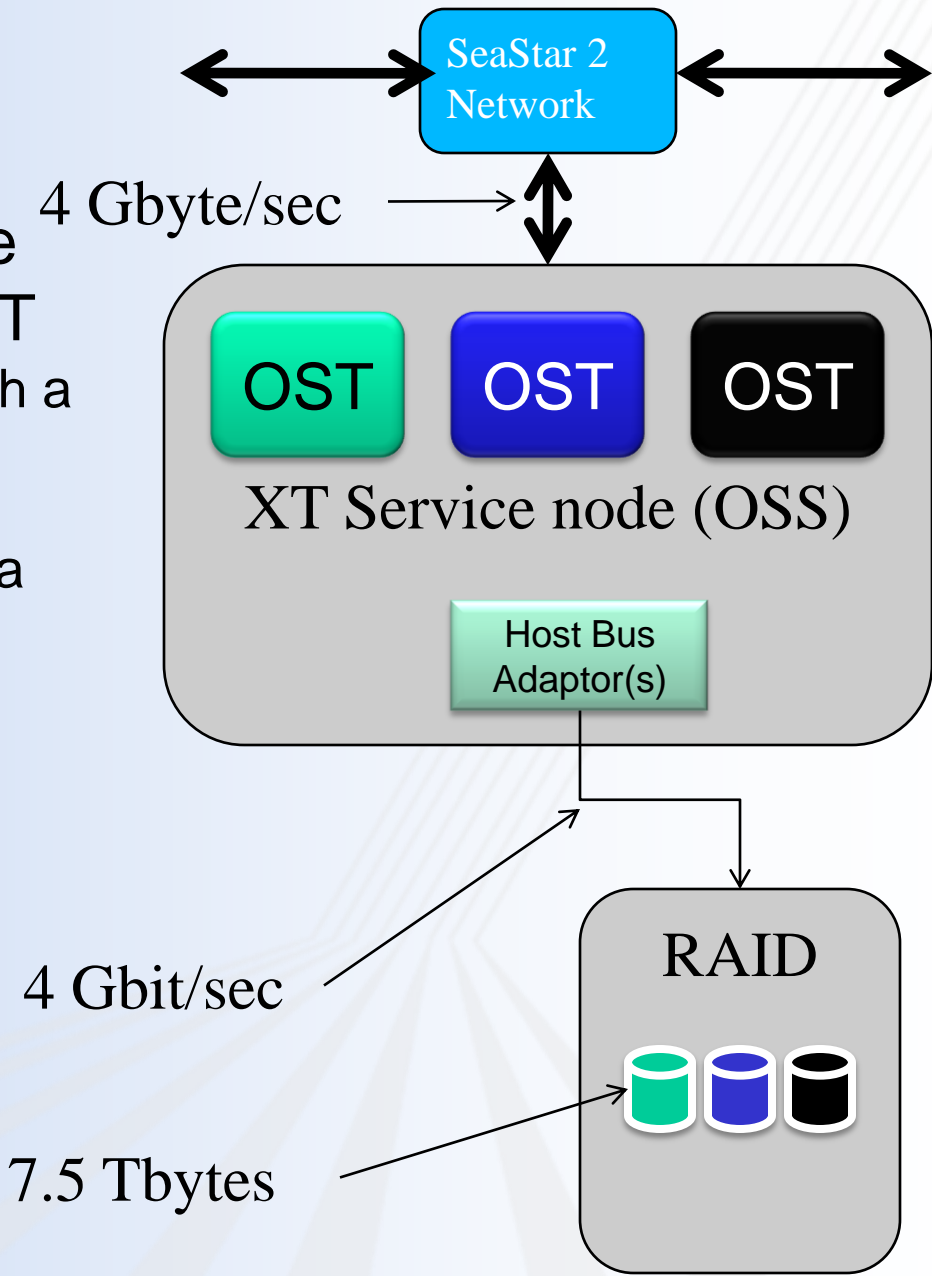


# The OS cache

- The cache will grow to fill available memory
  - ✿ Linux systems look like they have no free memory most of the time
  - ✿ OS will reclaim cache pages if applications require them
- No separate caches for different devices
  - ✿ Cache the whole VFS
- The cache operates on a deviceid:block\_number basis
  - ✿ A read() looks like
    - ▶ Get the dev:blk for the data we want to read
    - ▶ Check if present in the cache, if not
    - ▶ Get a clean buffer from the cache from the free buffer list
    - ▶ Call the device driver to read data into the buffer and place it in the cache
- It is possible to bypass the OS cache
  - ✿ Use O\_DIRECT flag to UNIX open()
    - ▶ Limitations that transfers are 512b aligned and multiples of 512b

# OST Detail

- An XT service (OSS) node may host one or more OST
  - ⚙ Each OST is associated with a block device
    - ▶ Ext3 filesystem
  - ⚙ The block device abstracts a LUN that is part of a RAID device
- HECToR:
  - ⚙ 3 OST per OSS
  - ⚙ 1 4 Gbit HBA per OSS
  - ⚙ 7.5TB per OST



# OST Detail

- 5664 Clients (compute nodes)
  - ✿ Potentially 27 TB of distributed cached data
- 24 OSS x 3 OST
  - ✿ 72 OST total
- We ignored some detail;
  - ✿ The OSS also has filesystem cache
  - ✿ The RAID also has cache, as well as its own internal network topology
- The 4GBit fiber channel link is the immediate performance bottleneck
  - ✿ One compute node writing to one OST, has a peak of 0.5GB/s
  - ✿ All OSS writing to disk, gives theoretical machine peak of 12 GB/s
    - ▶ 5GB/s demonstrated for machine acceptance

# Hector Raid devices

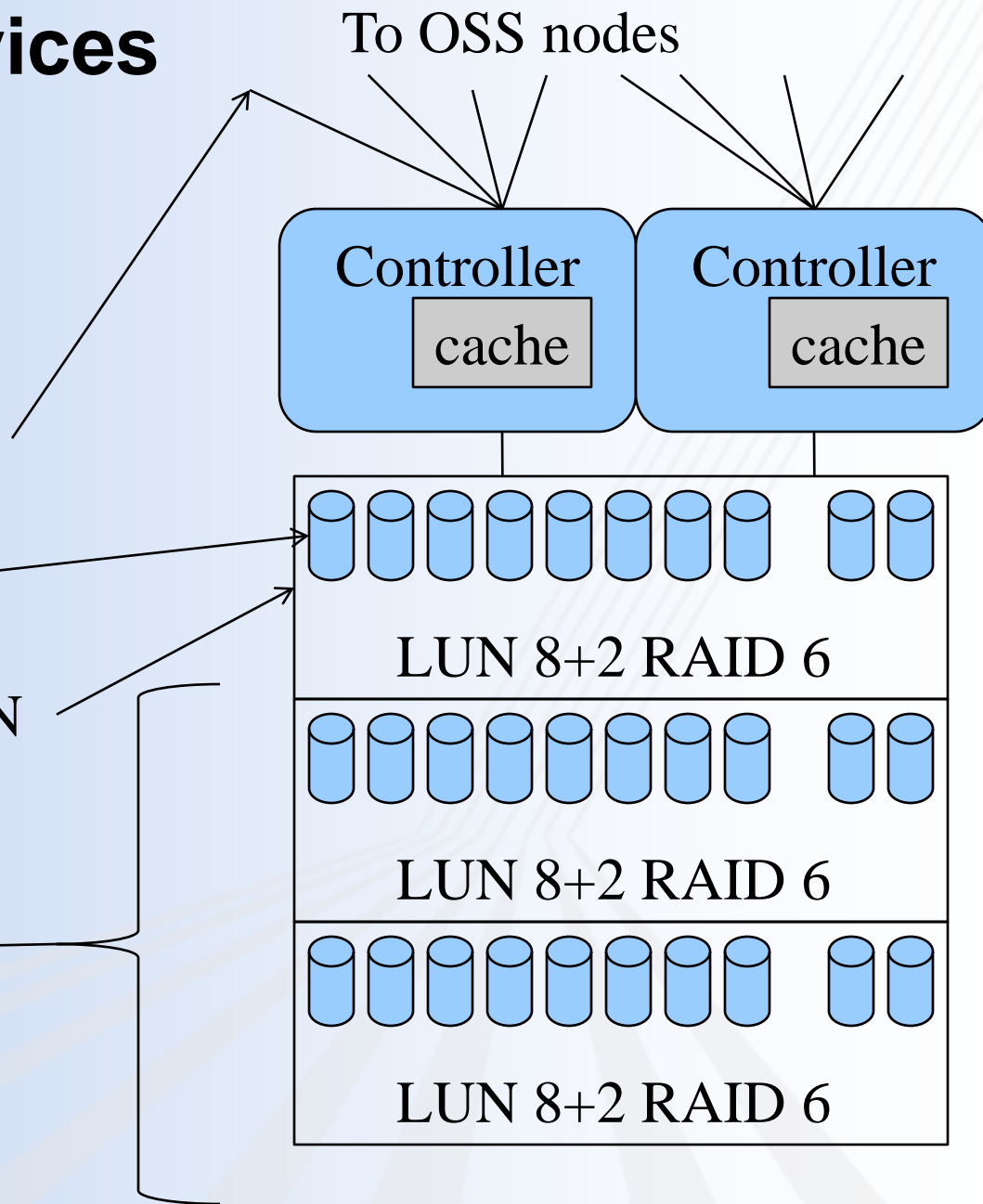
- Hector has 3 devices supporting /work

4x4Gb links per DDN controller

500GB SATA drive

7.5TB usable space/LUN

24 LUNS per controller couplet



# Safety First

- At what point is data written from an application 'safe' from a component failure?
  - ✿ close/fclose commits data from userspace
    - ▶ Protection against an application crash
    - ▶ Data potentially cached (somewhere)
  - ✿ fsync/sync ensures the cache is coherent with the filesystem,
    - ▶ Protection against a node failure
    - ▶ MPI IO equivalent exists
  - ✿ Other components are not user manageable
    - ▶ OSS failures are rare, and ext3 features help with resiliency
    - ▶ RAID caches have backup power
- Best practice
  - ✿ Close files with reasonable frequency (see later notes)
  - ✿ Use A/B images for checkpoints
  - ✿ Sync when data \*must\* be committed

# Lustre on a Cray....

- Lustre is tuned for the Cray XT platform
  - ✿ Intranode communications and data payload takes place on the Cray SeaStar 2 network
  - ✿ Portals is employed for all communications (metadata, messages and payload)
  - ✿ Otherwise similar to other Lustre environments (architecture, commands, etc)
  - ✿ X2 nodes access the same Lustre environment, as the hybrid architecture allows access to the same high performance network.

# Tuning (Application patterns)

## ■ Ideal

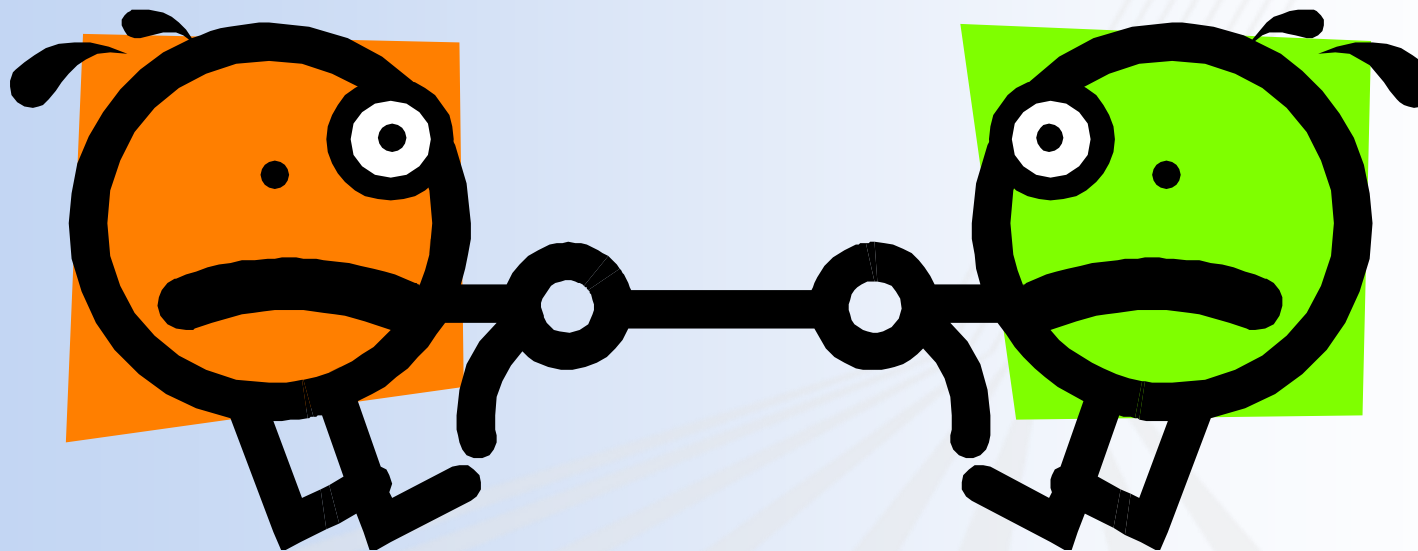
- ⚙ All OSTs are kept busy
- ⚙ MDS activity is kept to a minimum
- ⚙ Data transfers are large

## ■ Limiting factors

- ⚙ Application/library manipulation of data
- ⚙ Inject bandwidth (compute node)
- ⚙ Fiber channel bandwidth to disk (Already discussed)
  - ▶ We don't see OSS inject issues as the fiberchannel dominates on the hector configuration, similarly back end disk performance.
- ⚙ MDS
  - ▶ There is only one MDS server for the whole system
  - ▶ High request rates will serialize
    - e.g. 10000 tasks open a file simultaneously....

# “Typical” Application I/O

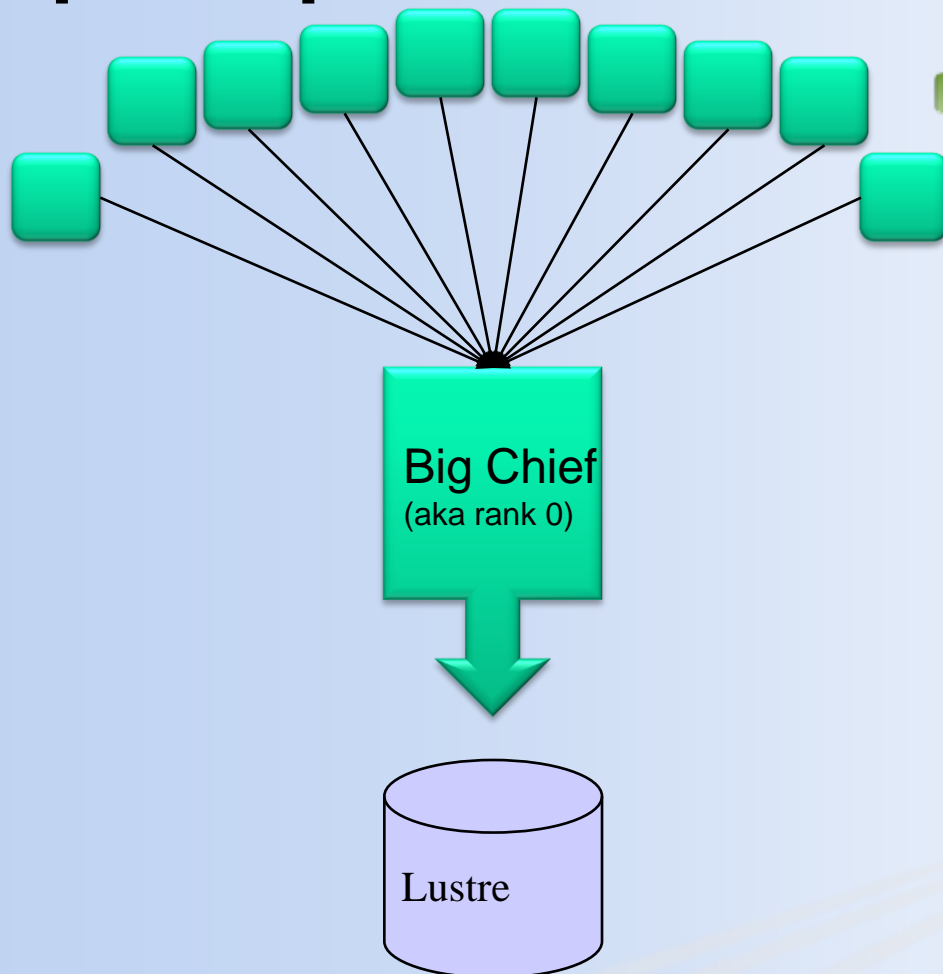
- THERE IS NO TYPICAL APPLICATION I/O
- There are several common methods, but 2 are very common and problematic
  - ✱ Spokesperson
  - ✱ Every man for himself



**Simple**

**Efficient**

# Spokesperson Method



## ■ The Plan

- ✿ All processors send to a single chief for output
- ✿ Chief arranges the inbound buffers into contiguous data for IO

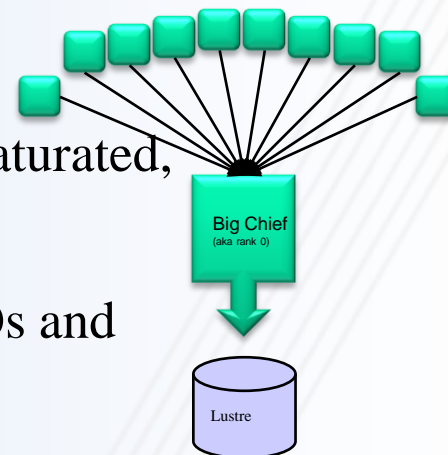
## ■ I'm a good citizen because:

- ✿ I Send all data in one big write.
  - ▶ Minimal MDS activity
  - ▶ Minimal OS activity
- ✿ File striped to maximum OSTs

# Spokesperson Method

## ■ Amdahl says;

- ⚙️ Once the (inject) bandwidth to the big chief is saturated, performance of IO is a serial part of the parallel execution.
- ⚙️ Node bandwidth is shared between outbound IOs and inbound MPI
- ⚙️ At best: a serialisation



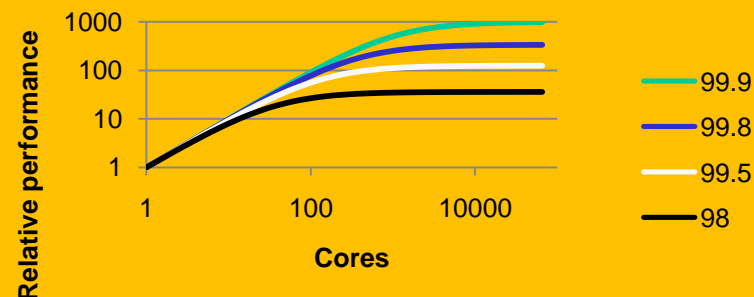
## ■ Systems architect says;

- ⚙️ You want to handle an **arbitrary** number of inbound messages without penalty and hardware acceleration?
- ⚙️ At worst: pathological at scale

## ■ The Problem

- ⚙️ To scale everything must be distributed

### Relative performance for different serial fractions



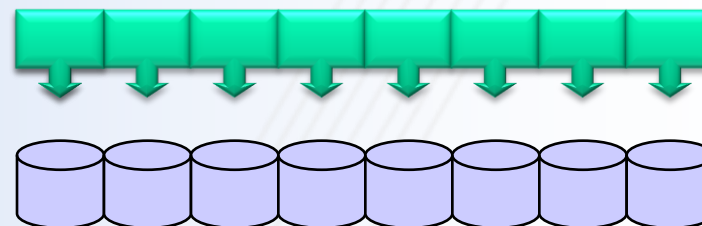
# Every Man for Himself Method

## ■ The Plan

- ✿ Every process opens a file and dumps its data
- ✿ Files striped to 1 OST

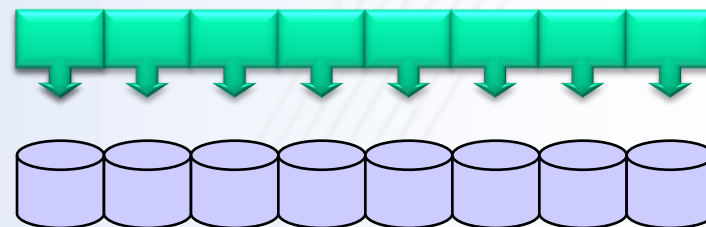
## ■ I'm a good citizen because

- ✿ I'm about as parallel as I can be (given the hardware I have)
- ✿ I don't have any communications overhead

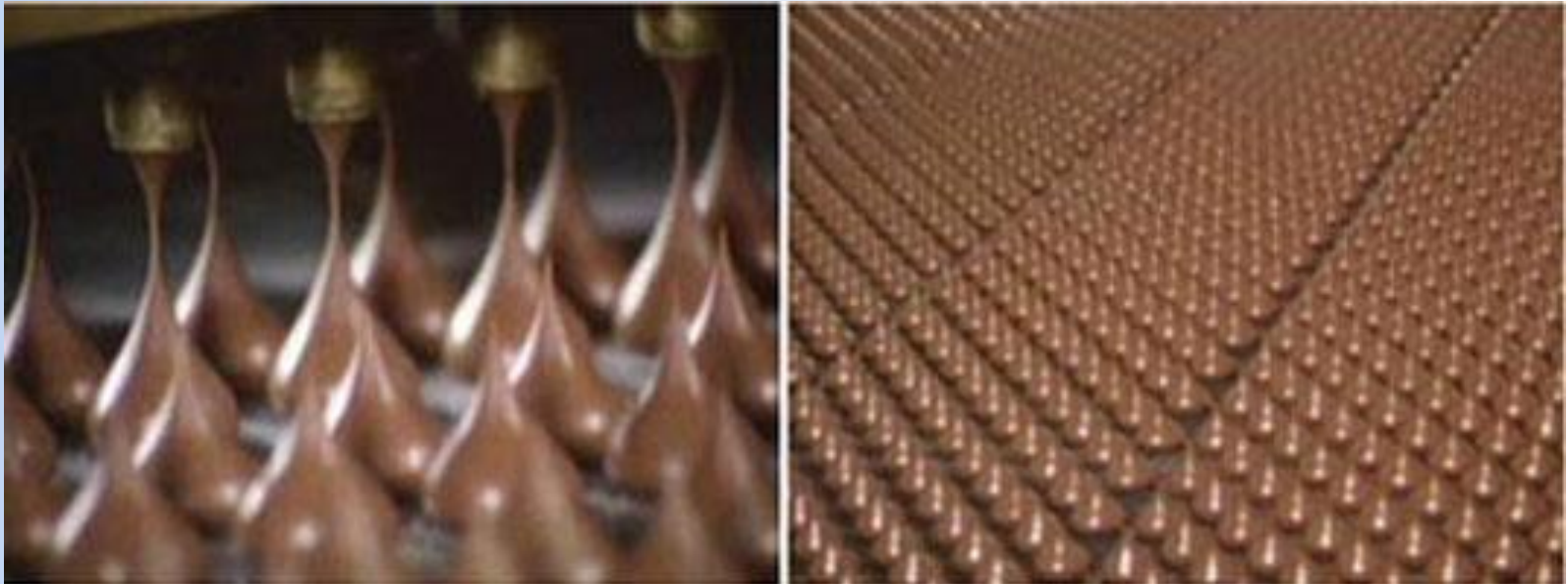


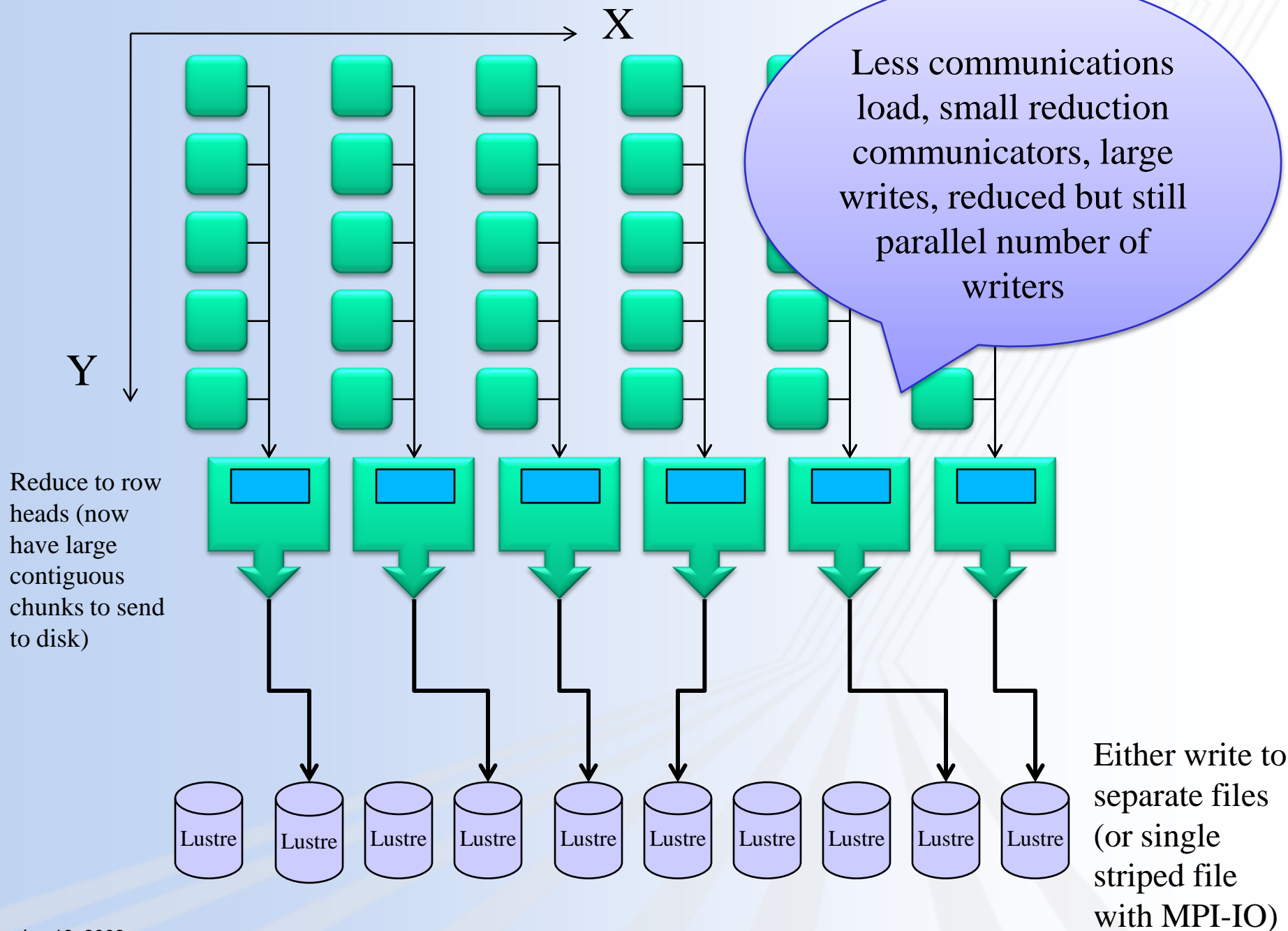
# Every Man for Himself Method

- The Problem
- MDS Load:
  - ✱ Every process opens a file and dumps its data
  - ✱ Huge load on the MDS at scale
    - ▶ Which is a serialisation
- Efficiency at scale
  - ✱ For a strong scaling problem, the IOs become smaller with increasing size
    - ▶ Individually less efficient
  - ✱ I have 10 thousand files to work with now ☹



# What does efficient I/O look like?





# Tuning (Application patterns - summary)

- All gather with one writer
  - ✱ Inject bandwidth bound
- All write to separate files
  - ✱ Potentially MDS bound
  - ✱ OSTs have multiple clients
  - ✱ Small writes
- Ideally
  - ✱ A subset of tasks write
    - ▶  $\sqrt{\text{numprocs}}$  is a good rule of thumb.
    - ▶ Application convenience is a guide (e.g. decomposition leads to natural choices)
    - ▶ Use the number of OST as a guide
- Note that:
  - ✱ Many-write-to-one-file isn't supported by a Fortran/Unix parallel environment - use MPI-IO to achieve this

# Tuning Luster

- From the Unix shell, files and directories can be given attributes.
  - ✿ Stripe count
    - ▶ How many OST to distribute files across
  - ✿ Stripe size
    - ▶ How many bytes go to each OST before moving onto the next
  - ✿ Start OST
    - ▶ Which OST shall I start on
    - ▶ Best left default on a multiuser system.
- `lfs setstripe /path/to/big/output 1048576 -1 8`
  - ✿ 8 way stripe, don't care which OSTs (-1), 1MB stripe size
- Aim for  $\text{num\_writers} * \text{stripe\_count} \sim O(\text{number\_OST})$ 
  - ✿ If everyone writes, stripe count should be 1.

# MPI IO

- MPI v2 provides a portable interface to parallel filesystems
  - ✱ Lustre, pvfs, ...
- Hides complexity and semantics of the hardware
  - ✱ Allows vendors to do the tuning
- Manages data movement on behalf of the user to obtain more optimal IO performance
- Many use cases (to many to cover here, but)

# MPI IO (generic)

- Simplest mapping of Unix -> MPI IO
  - ✱ Write() → mpi\_file\_write()
    - ▶ Individual file pointers
- Some MPI calls allow for collective buffering
  - ✱ Allows MPI to throttle, who actually writes, which nodes are responsible for what, coalescing of small IOs into larger
  - ✱ Aggregators take ownership of coarse grain chunks of the file
  - ✱ This requires collective operations
  - ✱ E.g. MPI\_FILE\_WRITE vs. MPI\_FILE\_WRITE\_ALL
    - ▶ Generally an \_ALL version for most data movers
    - ▶ Also split collectives etc
- Some MPI calls are asynchronous
  - ▶ Use them where possible

# MPI IO (specific)

- Hints normally hardcoded can be passed as environment variables
  - ✿ Avoid recompilation for tuning;
    - ▶ `export MPICH_MPIIO_HINTS="..."`
    - ▶ `export MPICH_MPIIO_HINTS_DISPLAY=1` (to show them)
- Hints to consider:
  - ✿ `direct_io=true` (`O_DIRECT`, user responsible for alignment)
  - ✿ `romio_ds_read=enable` (for noncontiguous file reads)
  - ✿ `romio_cb_read=enable` (collective buffering of reads)
  - ✿ `romio_cb_write=enable` (collective buffering of writes)
  - ✿ `cb_nodes=n` (how many nodes for collective buffering)
    - ▶ Tip: for IO intensive apps,  $n \geq 4 * \text{num\_ost}$
  - ✿ `cb_buffer_size=n`
- The items above may require recent releases of MPT.

# Notes/References

- Jeff Larkin and Mark Fahey's paper characterizing IO performance on the XT 3/4/5  
[http://pdsi.nersc.gov/IOBenchmark/Larkin\\_paper.pdf](http://pdsi.nersc.gov/IOBenchmark/Larkin_paper.pdf)
- MPI-2 IO reference: <http://www.mpi-forum.org/docs/mpi-20-html/node171.htm#Node171>
- HECToR's internal guidance on IO:  
<http://www.hector.ac.uk/support/cse/Resources/bestpractice/IO/>
- Use the HECToR CSE service for specific guidance/advice (mailto:[helpdesk@hector.ac.uk](mailto:helpdesk@hector.ac.uk)) for hector users.