

IO Thoughts and Optimization



Topics

- Why am I here?
 - ◆ Why is I/O always the last consideration?
- What should I be able to achieve?
- I/O Architecture
- Lustre and Cray implications
- I/O Styles and Methodologies
- Performance Studies
 - ◆ Serial Performance
 - ◆ Achieving maximum performance
- Lustre Optimizations
- Lustre Performance scaling

Target is to give an understanding and framework to choose the correct I/O performance for your code.

Topics – what isn't here

- Target is to give an understanding and framework to choose the correct I/O performance for your code.

- This talk does not cover the immense area of MPI-IO
 - nor netCDF
 - or HDF5
 - but maybe one day...

Why Should I be Interested in I/O performance

- If I were to ask 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 here to fix it
 - I/O has never really been a problem until I got on the Cray system
 - ▶ Oh, and I also upped my job size to 1024 from 128
 - 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

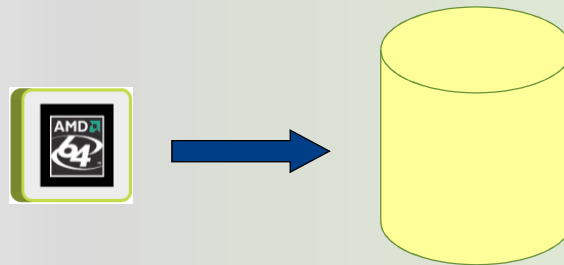
- I/O is a shared resource unless the disk is a dedicated resource
 - ⚙️ On Cray XT series no disk resource is dedicated – remember that /tmp is memory so not very big nor permanent... nor important.

What Should I Target

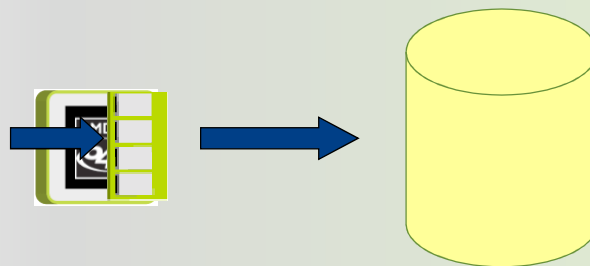
- I don't care about what order data reaches disk or when, how it is split and it will be done in parallel. All that matters is performance.
 - ⚙️ Excellent – the greatest performances await ☺️
- I want performance but ensuring the data is on disk is important.
 - ⚙️ Good – measure in GB/s (maybe higher)
- Format and structure and portability matter but I've tried to make my code use large contiguous blocks
 - ⚙️ Measure I/O in 100's MB/s
- None of the above apply
 - ⚙️ 10's MB/s – maybe lower.
 - ⚙️ You should look at the I/O pattern in your code
- I have no control – I use an external library that I have no control over.
 - ⚙️ You always have control over how you use the library.
 - ⚙️ Choose another library or the optimized version.
 - ⚙️ Optimize the use of Lustre.

What is I/O

- Input is the need to load data into my program/data space
- Output is the need to move data out of my data space
 - ⚙ This could either be from my program
 - ⚙ Or to disk

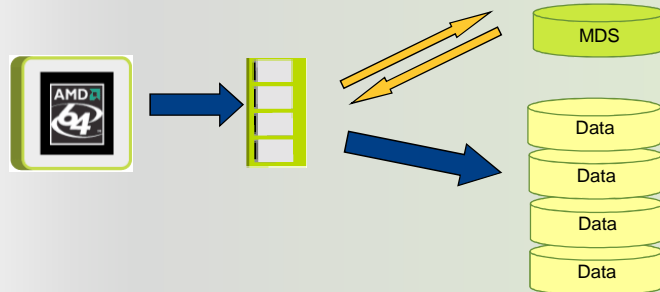


- That looks really simple but the real situation is:



- Linux is really good at using buffer cache.
 - ⚙ Much better than catamount

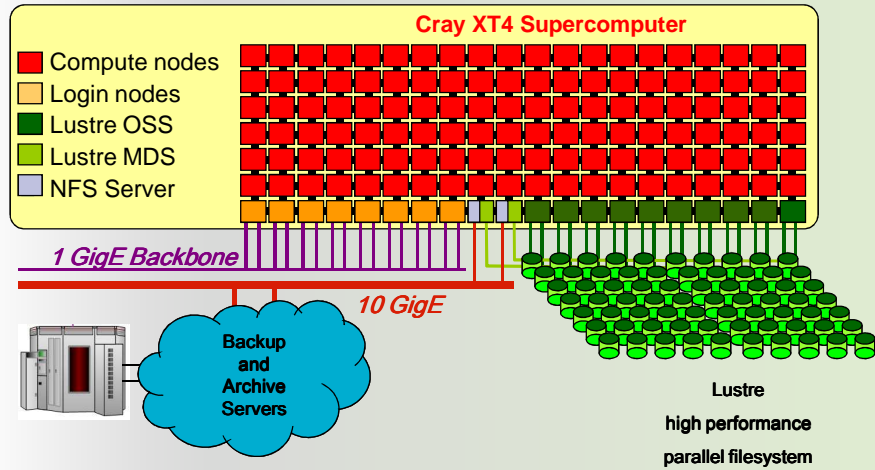
- Actually it is a little more complicated than that with Lustre...
 - ✿ The interaction with the disk consists of two phases



The Complexity of I/O

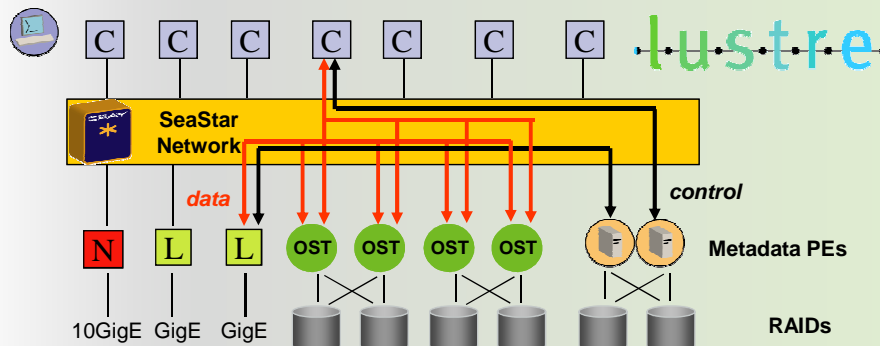
- There are so many levels within I/O that there are more questions than answers.
- Often great performance is masking some underlying problems that are exposed when we scale beyond our current levels (in terms of total required size).
- Levels:
 - ✿ User array (Size)
 - ✿ Write statement (size of each write, open method, file type)
 - ✿ Fortran & User level buffering (How much is there, how much is available)
 - ✿ Kernel Buffering (How much, how frequently flushed)
 - ✿ I/O nodes (How many, how much memory, how is data shared amongst the I/O nodes)
- Some of these questions will be answered here. Not all but sufficient to understand I/O levels and how to decide on a strategy for your code.

Machine Layout



I/O nodes

- I/O (lustre) nodes: run Lustre processes (OST, metadata server)
- Lustre terminology:
 - OST: Object Storage Target, software interface to back-end storage volumes
 - OSS: Object Storage Server, I/O node that host OSTs
 - MDS: Metadata server, handles namespace operations



IO Styles and Methodologies

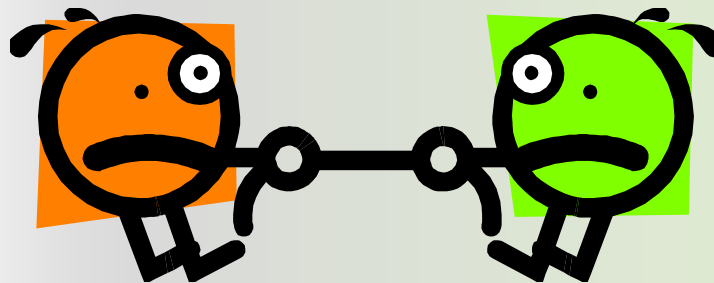
November 08

Cray Inc. Proprietary

Slide 13

“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

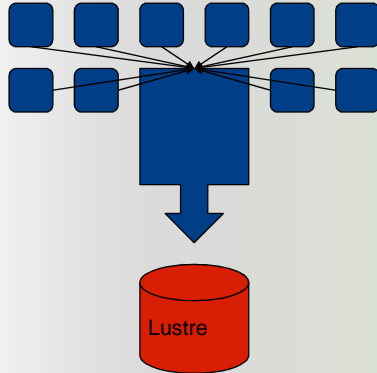
Efficient

November 08

Cray Inc. Proprietary

Slide 14

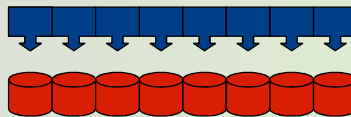
Spokesperson Method



- The Plan
 - ⚙ All processors send to 1 I/O node for output
 - ⚙ File striped to maximum OSTs
- The Problem
 - ⚙ Even with maximum striping, 1 node will never achieve maximum bandwidth.
 - ⚙ single node IO bandwidth is extremely limited. The network bandwidth in and out of the node is limited.
 - ⚙ reading/writing a terabyte would require more than 1 hour at current I/O rates.

Every Man for Himself Method

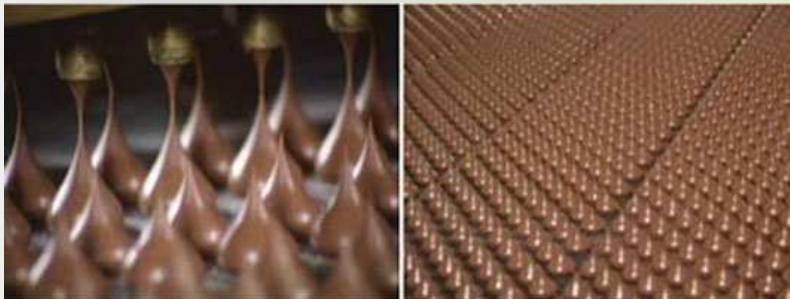
- The Plan
 - ⚙ Every process opens a file and dumps its data
 - ⚙ Files striped to 1 OST
- The Problem
 - ⚙ Can lead to slow opens and general filesystem slowness
 - ⚙ If the writes are not large, performance will suffer
 - ⚙ Many files can be inconvenient
- One Modification
 - ⚙ Use MPI-I/O for just 1 file
 - ⚙ Suffers when i/o results in small buffers
 - ⚙ Data shows is as non-optimal



The Problem

- **These methods are fine until you scale-up**
 - ❁ Proof forthcoming
- **Without user-intervention you will not get practical peak I/O bandwidth**
- **For example**
 - ❁ Spokesperson
 - ▶ Even with maximum striping on file, effective bandwidth is limited by the 1 compute node
 - ❁ Every man for himself
 - ▶ Slow metadata operations (all hit the MDS at the same time)
 - ▶ Overwhelm OSTs and/or IO service nodes
 - ▶ Possibly inconvenient to users

What does efficient I/O look like?



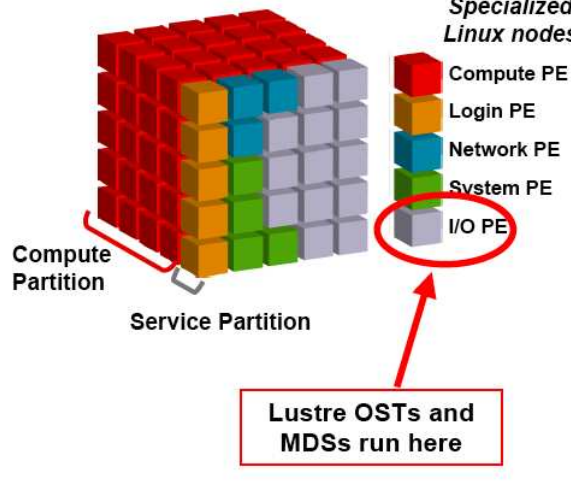
Striking a Balance

Application Needs

Filesystem Limits



Benchmark System



- Compute partition has
 - 11,508 AMD dual-core processors
 - Running Catamount
 - 46 TB of memory
- Lustre filesystems
 - Serviced by 72 I/O nodes
 - /lustre/scr144
 - ▶ 144 OSTs
 - ▶ Peak is 72 GB/s
 - ▶ Practical peaks
 - Read 45 GB/s
 - Write 25 GB/s
 - /lustre/scr72[a,b]
 - ▶ 72 OSTs each
 - ▶ Default scratch

Achieving good performance in serial...

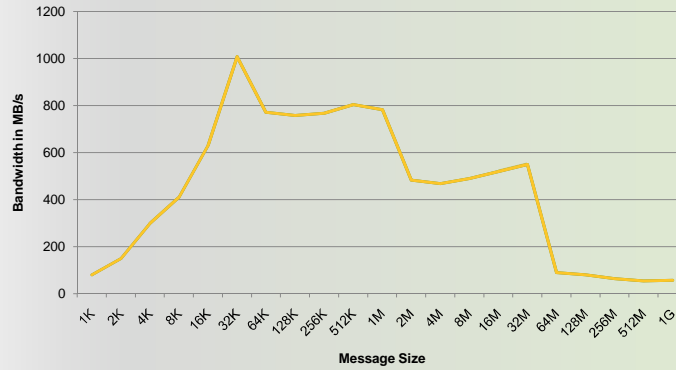
The code

- A serial code was developed that
 - ✿ Writes large blocks of data (N) to disk.
 - ✿ Can repeat this a number of times (R)
 - ✿ We measure the write performance in terms of the bandwidth.
 - ✿ Change the lustre properties of the file to achieve implicit parallelism

Experiment I - Serial Performance

- Basic IO measurement. This is symptomatic of the spokesperson method, where data is accumulated to one rank and sent to disk as one big message.

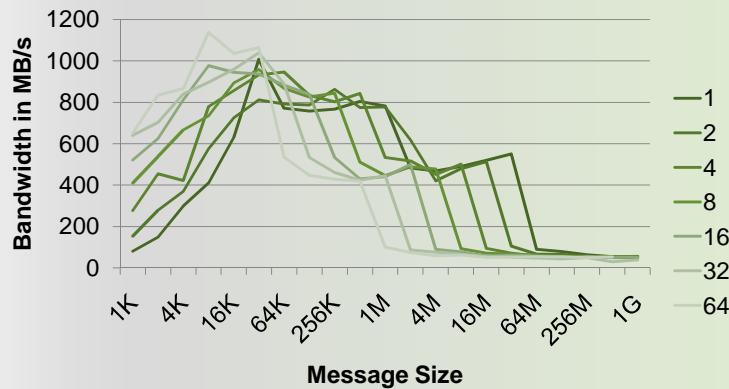
Achieved Bandwidth for Single Message



Experiment II - Repeated Performance

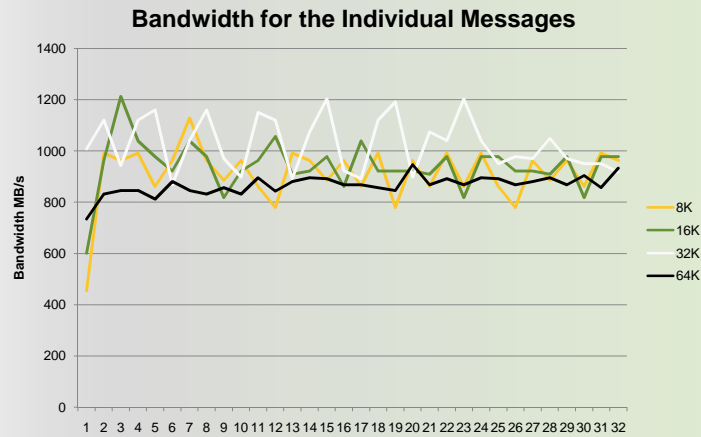
- Typically our I/O is not one contiguous block but a number of consecutive IO statements
 - Perhaps a row of the data at a time

Bandwidth for Multiple Messages



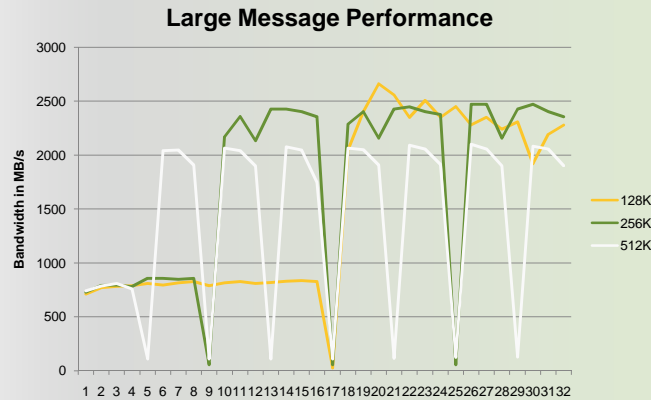
Experiment III – Per message bandwidth

- But what effect is the on the measured bandwidth across each of the write statements.



Experiment III – Per message bandwidth

- But what about larger messages

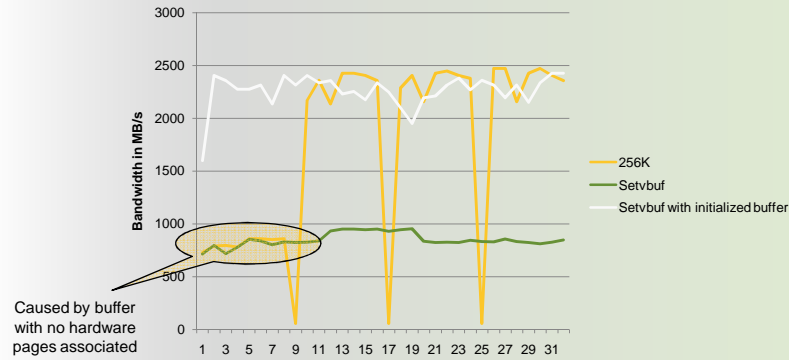


- Looks like a buffer issue when it hits 2MB

Why 2MB?

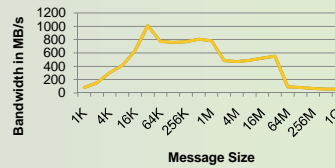
- 2MB is the system buffer for the unit.
- PGI has a nice feature called setvbuf (and setvbuf3f) which allow you to allocate buffers for the I/O

Buffering for 256K Messages



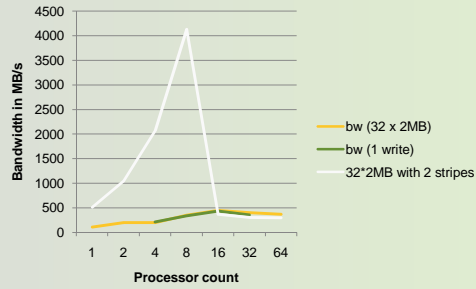
- This 2MB buffer explains the first drop in performance, but there is a further drop at 64MB
 - Caused by a drop every 64MB
 - Drops to approx. ~10MB/s

Achieved Bandwidth for Single Message



Unbuffered vs Buffered I/O

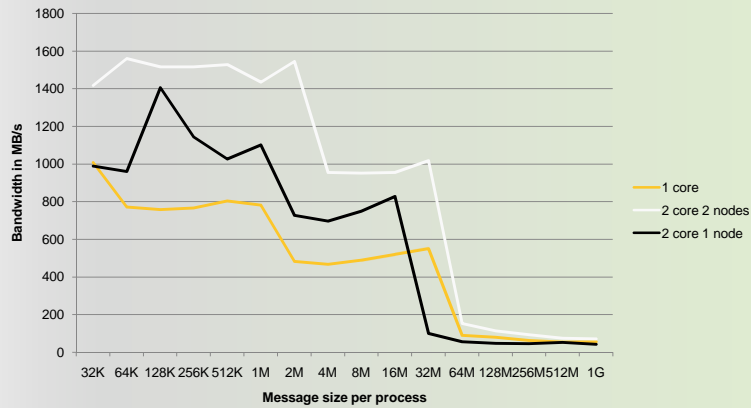
- Buffered I/O can get some very fast performance for any given instance but the time is spent somewhere
 - ◆ Either in an expensive close operation
 - ◆ Or when the buffers are full
 - ◆ The bigger the buffer the more expensive the operation
- Unbuffered I/O removes the Fortran buffering layer.
 - ◆ We can begin to see some more consistent performance
 - ◆ Some writes may slow down.
 - ◆ Close is now pretty fast.
- Becoming scalable too!



Experiment IV – Multiple processors

- We now do this with multiple processors all doing the same thing.

Saturating the Cores on the Node



Part 3

LUSTRE

Lustre Review

- The concept of object storage is basic to Lustre
 - Objects can be thought of as inodes and are used to store file data. Lustre inodes simply contain references to the object storage target (OST) that stores the file data
 - Access to these objects occurs through object storage servers (OSSs), which provide the file I/O service
 - The OSTs perform the block allocation for data objects, which results in distributed and scalable allocation
- The namespace is managed by metadata services that manage the Lustre inodes
 - The services perform file lookups, file creation, file and directory attribute manipulation
 - Such inodes can be directories, symbolic links, or special devices
 - **The associated data and metadata is stored on the metadata servers**

MDS

- Meta Data Server
- All requests and changes that do not involve the data use the metadata server
 - File Open
 - File Close
- There is (currently) only one MDS
 - This can be a bottleneck
- Avoid a lot of processors trying to communicate with the MDS
 - All processors opening and closing files simultaneously

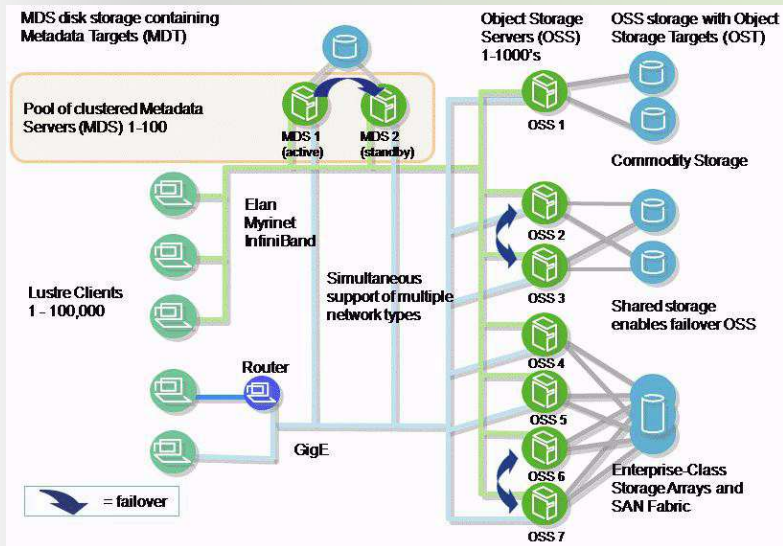
Lustre as a parallel Filesystem

- Striping and achieving parallelism at the IO level
 - Files are split across a subset of the servers
 - Data is written to the servers in a round robin fashion

Lustre

- Lustre has a utility (lfs) that can be used to create a file with a specific striping pattern, displays file striping patterns, and find file locations
- The maximum file size is 320 TB (on any lustre)
 - Maximum number of stripes per file is 160 (Lustre limit)
 - ▶ 2 TB x 160 = 320 TB (2 TB is max LUN size)

Lustre Architecture



Lustre File Striping

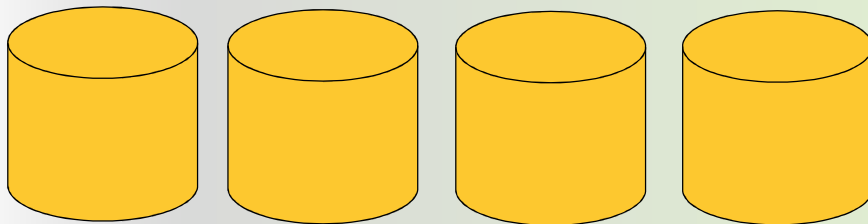
- Stripes defines the number of OSTs to write the file across
 - ✦ Can be set on a per file or directory basis
- CRAY recommends that the default be set to
 - ✦ not striping across all OSTs, but
 - ✦ set default stripe count of one to four
- But not always the best for application performance.

As a general rule of thumbs :

 - ✦ If you have one large file => stripe over all OSTs
 - ✦ If you have a large number of files (~2 times #OSTs) => turn off striping (#stripes=1)
- Common default
 - ✦ Stripe size = 1 MB
 - ✦ Stripe count = 2

File creation

- Let's assume 2MB write statements and 1MB stripesize with a stripe count of 3.
 - ✦ On open MDS chooses the three OSTs to be used. Local metadata and all further interaction involves OST only



Lustre lfs command

- **lfs** is a lustre utility that can be used to create a file with a specific striping pattern, displays file striping patterns, and find file locations
- The most used options are :
 - setstripe
 - getstripe
 - df
- For help execute **lfs** without any arguments

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

lfs setstripe

- Sets the stripe for a file or a directory
- **lfs setstripe <file|dir> <size> <start> <count>**
 - stripe size: Number of bytes on each OST (0 filesystem default)
 - stripe start: OST index of first stripe (-1 filesystem default)
 - stripe count: Number of OSTs to stripe over (0 default, -1 all)
- Comments
 - The stripes of a file is given when the file is created. It is not possible to change it afterwards.
 - If needed, use lfs to create an empty file with the stripes you want (like the touch command)

lfs getstripe

- Shows the stripe for a file or a directory
- Syntax : lfs getstripe <filename|dirname>
- Use `--verbose` option to get stripe size

```

louhi> lfs getstripe --verbose /lus/nid00131/roberto/pippo
OBDS:
0: ost0_UUID ACTIVE
<lines removed>
31: ost31_UUID ACTIVE
/lus/nid00131/roberto/pippo
lmm_magic:          0x0BD10BD0
lmm_object_gr:      0
lmm_object_id:      0x697223e
lmm_stripe_count:   2
lmm_stripe_size:    1048576
lmm_stripe_pattern: 1

```

obdidx	objid	objid	group
14	42575	0xa64f	0
15	42585	0xa659	0

lfs df

- shows the current status of a lustre filesystem

```

kroy@nid00004:~/lustre> lfs df
UUID          1K-blocks      Used Available  Use% Mounted on
mds1_UUID    249964396    14848316 235116080    5% /work[MDT:0]
ost0_UUID    1922850100   108527440 1814322660    5% /work[OST:0]
ost1_UUID    1922850100   110297980 1812552120    5% /work[OST:1]
ost2_UUID    1922850100   114369912 1808480188    5% /work[OST:2]
ost3_UUID    1922850100   104407112 1818442988    5% /work[OST:3]
ost4_UUID    1922850100   111024884 1811825216    5% /work[OST:4]
ost5_UUID    1922850100   105603904 1817246196    5% /work[OST:5]
ost6_UUID    1922850352   106531460 1816318892    5% /work[OST:6]
ost7_UUID    1922850352   109677076 1813173276    5% /work[OST:7]
ost8_UUID    1922850352   1442137764 480712588    75% /work[OST:8]

filesystem summary: 17305651656 975429728 16330221928 5% /work

```

Artificially increased to show data being prioritised in one ost

Lustre striping hints

- For maximum aggregate performance: **Keep all OSTs occupied**
- Many clients, many files: **Don't stripe**
If number of clients and/or number of files \gg number of OSTs:
Better to put each object (file) on only a **single** OST.
- Many clients, one file: **Do stripe**
When multiple processes are all accessing one large file:
Better to stripe that single file over **all** of the available OSTs.

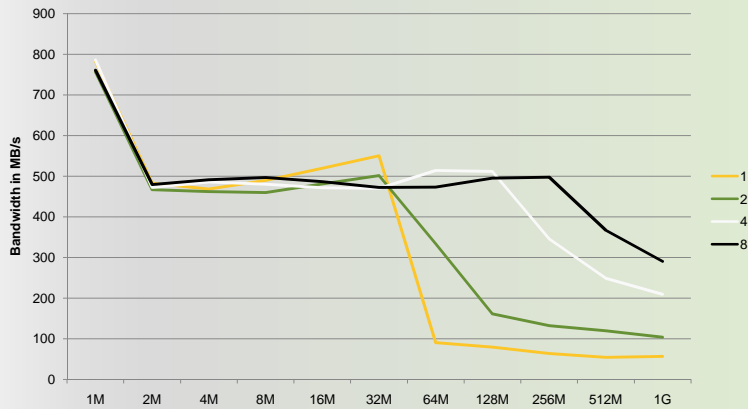
Part IV

PARALLEL I/O WITHIN LUSTRE

Experiment V - Striping

- Lets go back to our serial code and explore the Lustre parameters. Firstly the number of stripes within the file...

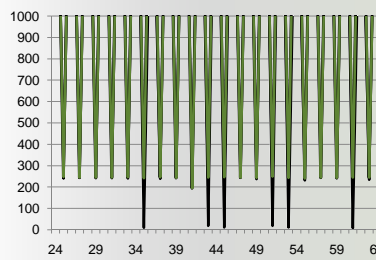
Affecting the Lustre Parallelisation



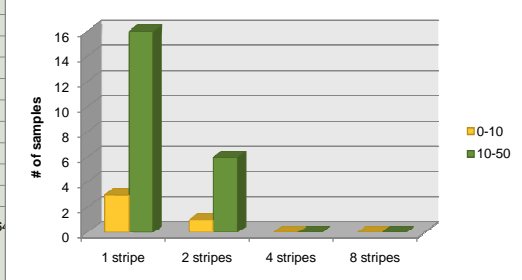
Striping

- Why does this improve things?
- We see less bad results because the distribution of our packets across OSTs allow them more time to recover from previous requests.
 - All samples are now pushed into higher bins

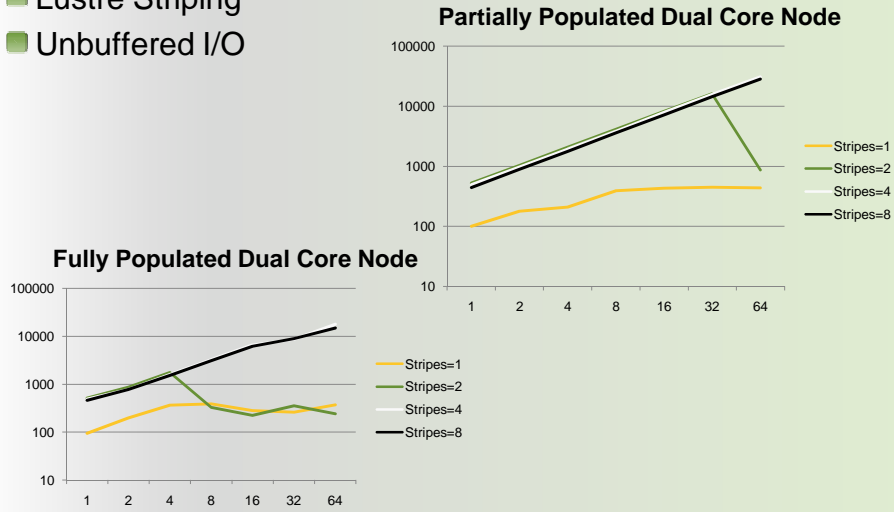
A Selection of those Samples



Binned samples



- Lustre Striping
- Unbuffered I/O



All Other Applications

- Most applications do not fit this model
- However there is enough here to make all applications a pencil and paper exercise.
- If you are going to scale down your application run for benchmarking remember to scale down your I/O expectations.
 - So when we scale back up it should still work.

I/O hints

■ Cray PAT

- Use Cray PAT options to collect I/O information
- Select proper buffer size and match it to Lustre striping parameters

■ Striping

- Select the striping according to the I/O pattern
- Experiment with different solutions

■ Performance

- One single I/O task is limited to about 1 GB/sec
- Increase I/O tasks if lustre filesystem can sustain more
- If too many tasks access the filesystem at the same time, the performance per task will drop
- It might be better to use a few tasks doing the IO (IO Servers).

What can I do?

THE GOOD STUFF

First Steps to Improving I/O Performance

- Understand your what your application needs and what your system can provide
 - ⚙ How much data I am writing and how often?
 - ⚙ What's the peak bandwidth of the system?
 - ⚙ Ok, what's the real bandwidth?
- Determine where you have a problem
 - ⚙ Am I performing a lot of small writes that could be combined?
 - ⚙ Am I overwhelming the FS with too much at once?
 - ⚙ Do I really need to save all of this data every single timestep?
- Try different Lustre parameters
 - ⚙ Could more or fewer OSTs help?
 - ⚙ Can I improve performance with larger stripes?
- If possible, make a small test program out of the I/O portion of your code
 - ⚙ Sometimes it's easier to test parameters with a smaller kernel than a full application.
 - ⚙ Bear in mind that continual I/O will swamp the system and that padding maybe required to allow this to happen.
- Seek help

Using MPI I/O Hints

- The MPI specification provides a way to give "hints" to the MPI-I/O layer for better performance.
- Hints to try
 - ⚙ `cb_nodes` -> Built-in subsetting
 - ⚙ `cb_read/write` -> Enable/Disable "collective buffering"
 - ⚙ `cb_buffer_size` -> Controls the size of the intermediate buffer used in collective buffering
 - ⚙ `ds_read/write` -> Enable/Disable "Data sieving", used with non-contiguous I/O requests
 - ▶ Generally not recommended
 - ⚙ `direct_io` -> Enables direct I/O, bypassing kernel buffers
 - ▶ Requires `xt-mpt/3.0.0.8` (pre-release as of April 3, 2008)
 - ▶ Requires data buffer to be aligned to page boundary
 - ▶ Can help with very large I/O requests
- Can be set via API or via `MPICH_MPIIO_HINTS` environment variable
 - ⚙ Example: `export MPICH_MPIIO_HINTS="${FILE}:direct_io=true:romio_cb_read=disable:romio_cb_write=disable:romio_ds_read=disable:romio_ds_write=disable"`
- Setting `MPICH_MPIIO_HINTS_DISPLAY=1` will print your MPI-IO hints when a program is run.

Best Practices

- Remember, this is not your laptop, I/O for HPC has many challenges
 - ✿ Unfortunately, I/O rarely scales at the same rate as FLOPS
- Do not open a file from hundreds/thousands of nodes at the same time
 - ✿ Metadata operations are slow, do them infrequently
 - ✿ Too many will overwhelm the MDS at very large scales
- Do not try to do all of your I/O through 1 node, unless you have a little data or a lot time.
 - ✿ Unable to saturate bandwidth
- Do not do I/O from every node for nodes over ~1K processes
 - ✿ Performance degradation
 - ✿ Where this degradation occurs varies by system

Best Practices (cont.)

- Buffer so that you can do large I/O operations
 - ✿ Bigger writes/reads perform better
 - ✿ Subsetting can help improve buffering
- Many files can perform better than 1, but can be less convenient
 - ✿ Try doing operations on many thousands of files, non-trivial
 - ✿ As discussed, too many at once files can lead to MDS overload
 - ▶ Subsetting can help with this too!
- Stripe Appropriately to Your I/O
 - ✿ Many nodes to individual files: Stripe to 1 OST
 - ✿ Many nodes to 1 file: Stripe to many OSTs
 - ✿ 1 node to 1 file: Stripe to few OSTs
 - ✿ Set your stripe size to larger than 1MB, 4MB suggested