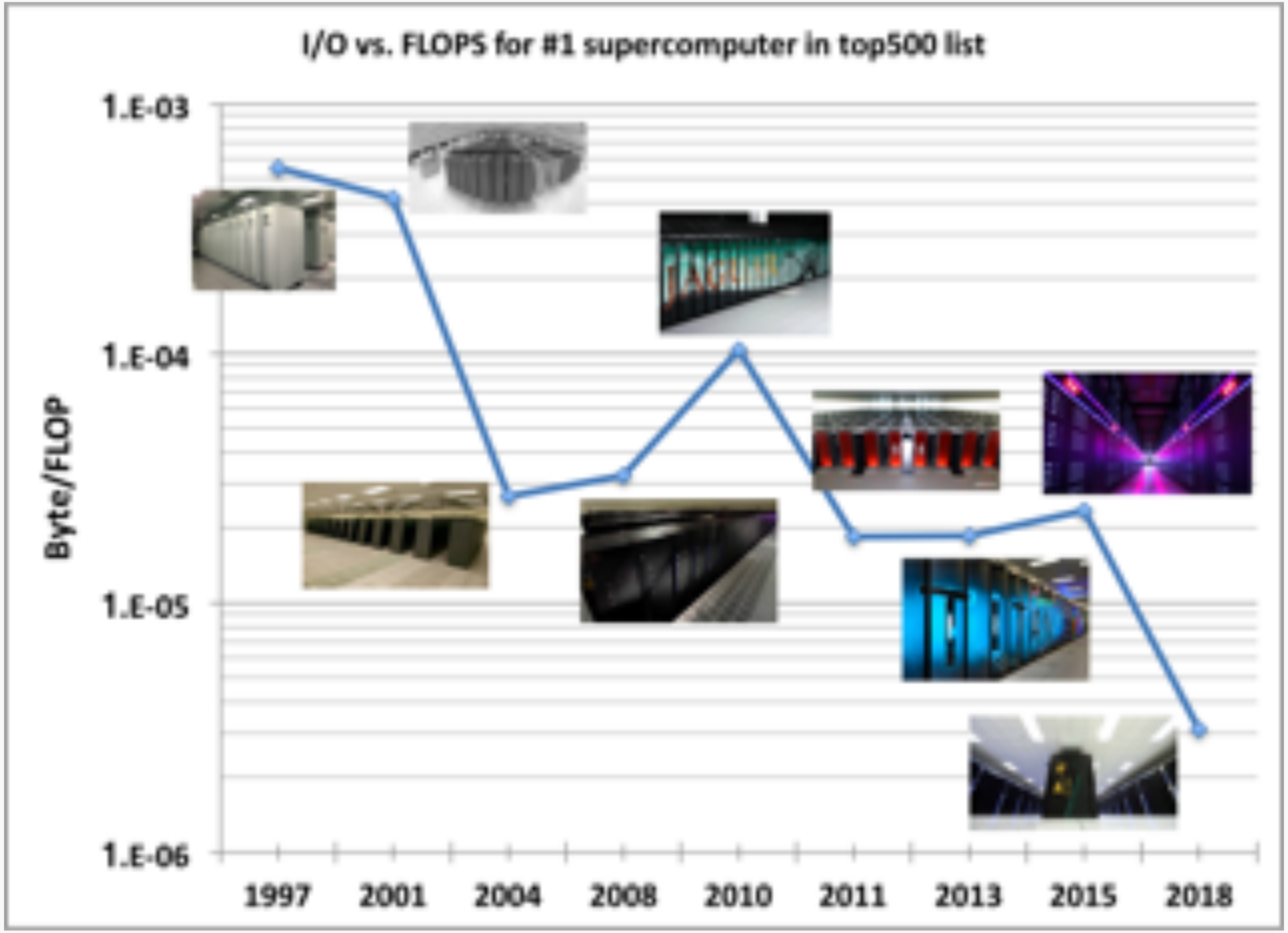
Parallel I/O on Theta with Best Practices

Paul Coffman, Francois Tessier, Preeti Malakar, George Brown
ALCF

Parallel IO Performance on Theta dependent on optimal Lustre File System utilization with potential use cases for node local SSDs

- Theta IO Architecture and Component Overview
- Theta Lustre File System Access Basics
- Cray MPI-IO, Tuning and Profiling, IO Libraries
- Lustre Performance Investigations, Analysis and Best Practices
- Theta Node Local SSD Utilization

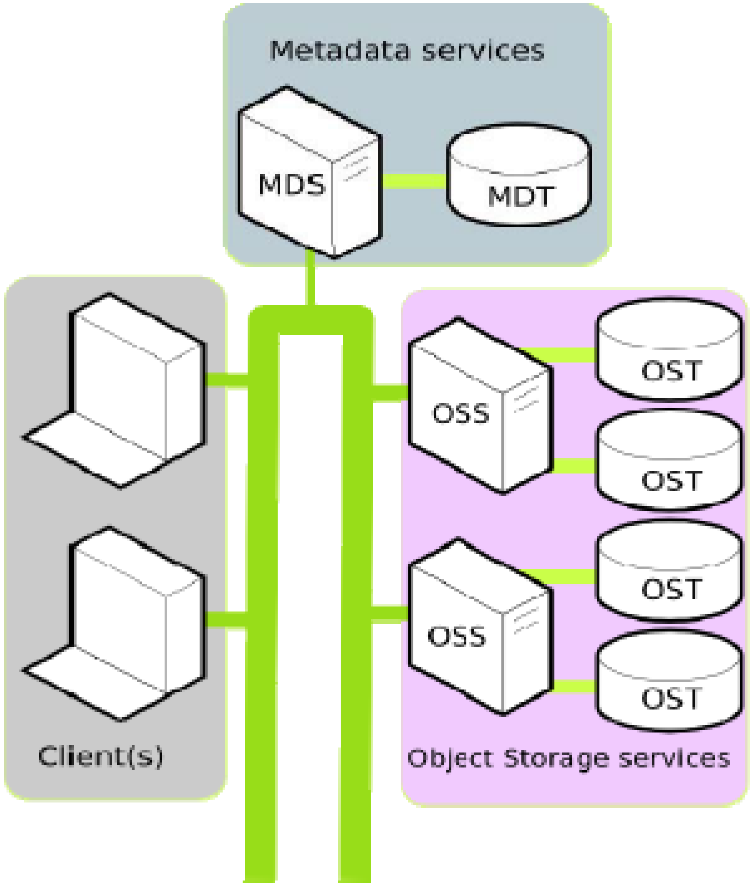
Storage vs Computation Trend



Theta IO Architecture and Component Overview

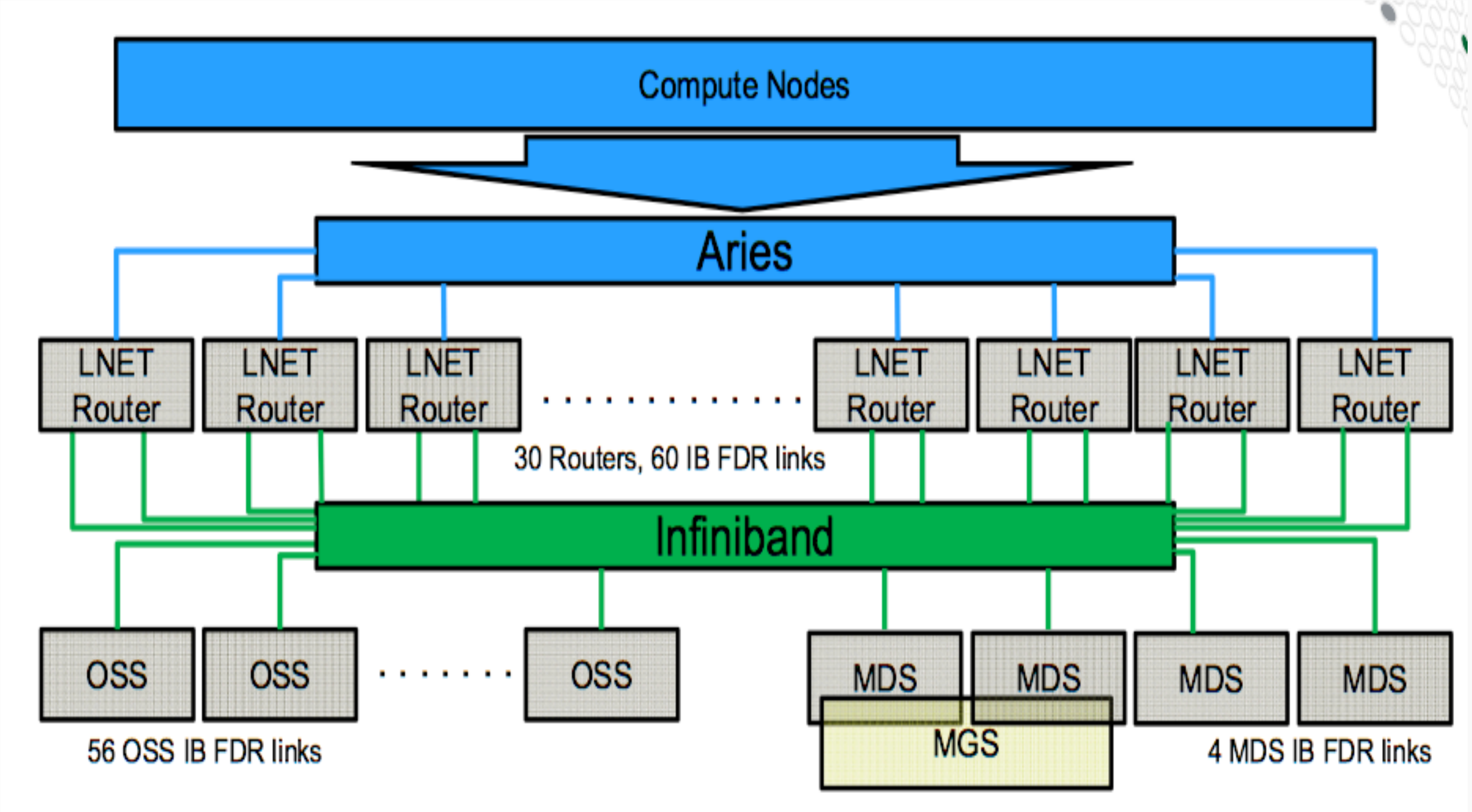
Basic overview of Lustre File System and the component configuration on theta.

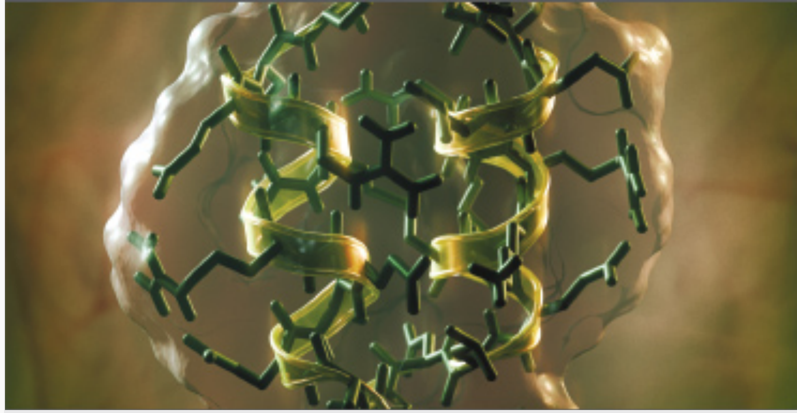
Lustre File System Basic Components



https://wiki.hpdd.intel.com/display/PUB/Components+of+a+Lustre+filesystem

Lustre Architecture On Theta



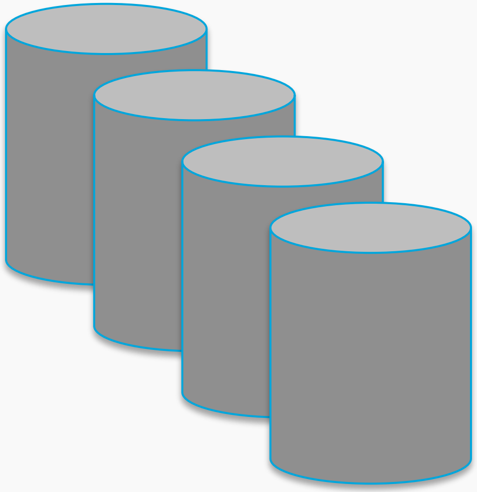


- **IO Forwarding from compute node to LNET Service Node / Router**
  - LNet Aries NIC on compute side, 2 IB links on Object Storage Server (OSS) side
  - OSS handles communication from LNet Router to Object Storage Target (OST) which is the physical storage device
  - Although there are 4 MDTs only 1 currently has directories placed on it

LUSTRE Specifications on Theta

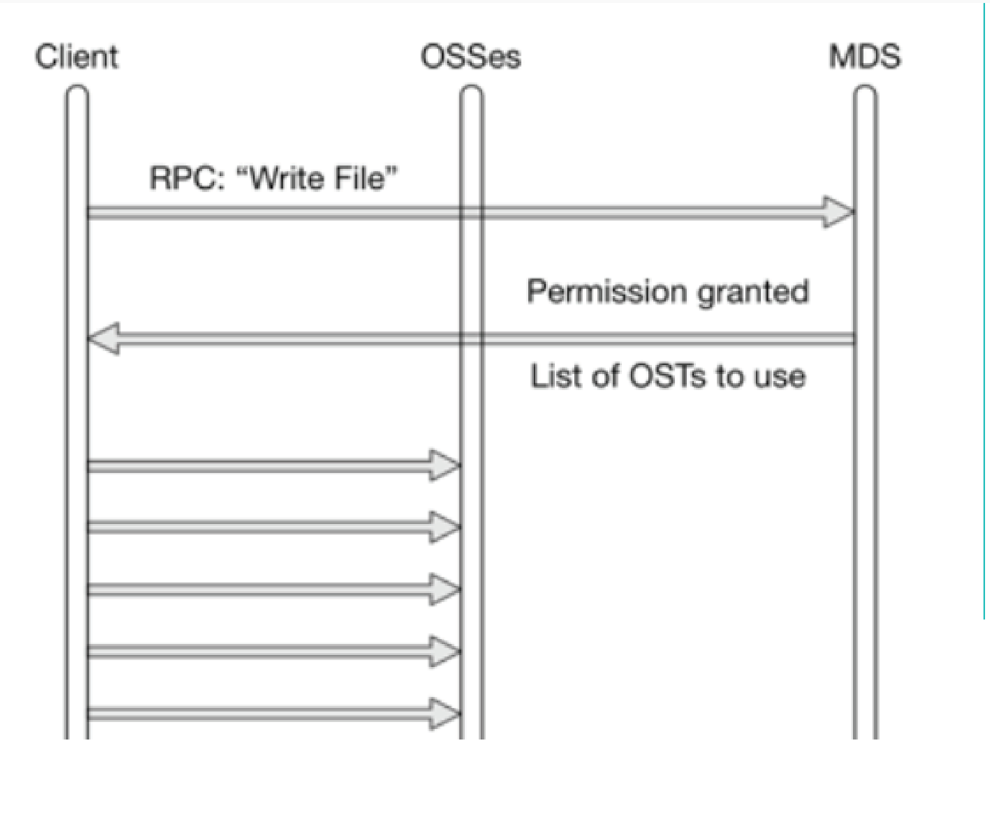
theta-fs0 - *project file system*

- /lus/theta-fs0/projects
- Sonexion Storage
  - 4 cabinets
  - 10 PB usable RAID storage
  - Total Lustre HDD Performance Write BW 172 GB/s Read BW 240 GB/s
    - 56 OSS Peak Performance 6 GB/s each
    - Node local client and OSS cache go a lot higher
- No project quotas, no backups yet available on lustre
  - /home is GPFS and backed up with 7 rolling daily snapshots
  - Group quotas sometimes enforced based on the project
    - lfs quota -h -g <group_name> /lus/theta-fs0

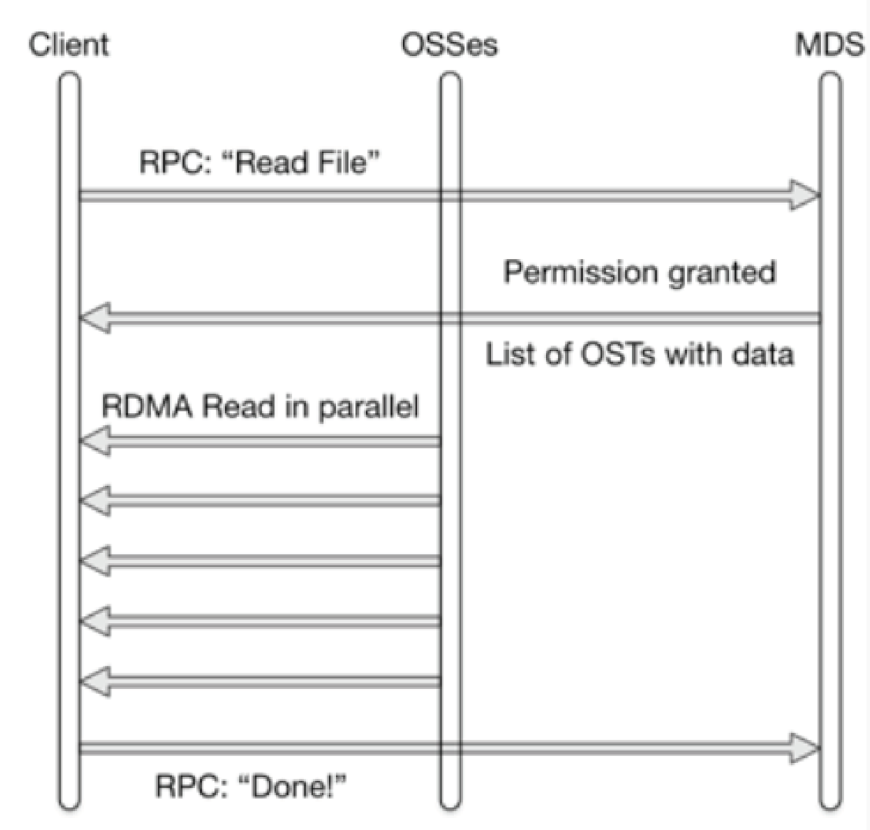


Lustre File operation flow

Posix Write



Posix Read



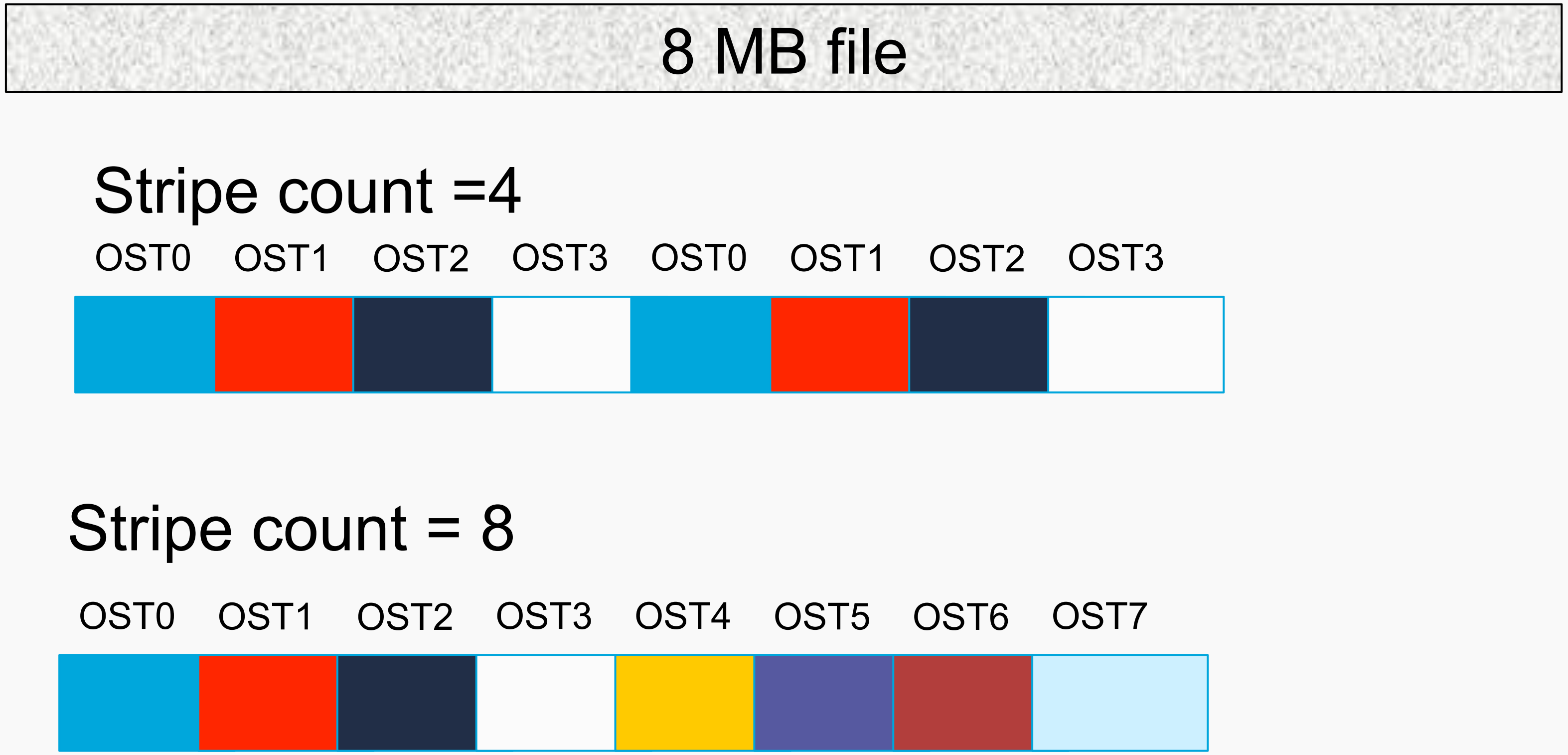
Theta Lustre File System Access Basics

Overview of basic Lustre File System interaction and key performance features .

Striping concepts – key to performance

Example:

Stripe size = 1mb, total file size being written 8mb



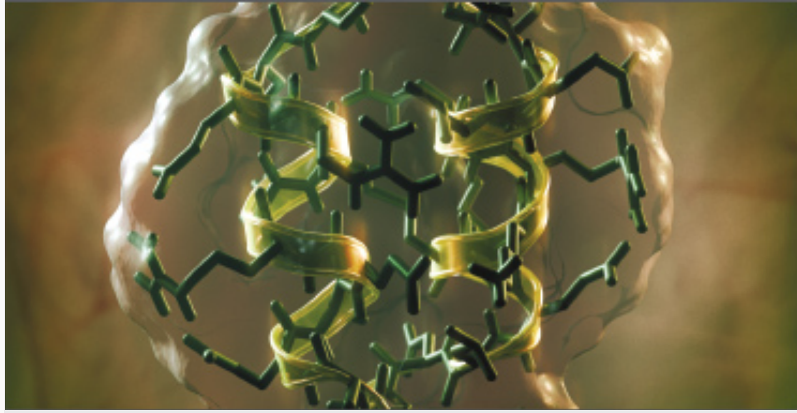
8 MB file

Stripe count =4

OST0 OST1 OST2 OST3 OST0 OST1 OST2 OST3

Stripe count = 8

OST0 OST1 OST2 OST3 OST4 OST5 OST6 OST7



Striping pattern = count and size

Count is number of OSTs (storage devices) used to store/access the file

Size is the width of each contiguous access on the OST

File is striped across OSTs

Lustre file system utility (lfs)

- http://doc.lustre.org/lustre_manual.pdf
- Run from login node
- lfs help
- List OSTs in file system
  - lfs osts <path>
- Set/Get striping information
- Search directory tree
- Check disk space usage
- lfs –version
  - lfs 2.7.2.26

lfs setstripe / getstripe example

```
lfs getstripe <file/dir name>
lfs setstripe --stripe-size <size> --count <count>
<file/dir name>
```

```
thetalogin4> mkdir stripecount4size8m
thetalogin4> lfs setstripe -c 4 -S 8m stripecount4size8m
thetalogin4> cd stripecount4size8m/
thetalogin4/stripecount4size8m> lfs getstripe .
.
stripe_count:   4 stripe_size:    8388608 stripe_offset:  -1
thetalogin4/stripecount4size8m> touch file1
thetalogin4/stripecount4size8m> touch file2
```

- Files created in same directory can be striped across different OSTs (shown in next slide)

lfs setstripe / getstripe example continued

```
thetalogin4/stripecount4size8m> lfs getstripe .
.
stripe_count:   4 stripe_size:    8388608 stripe_offset:  -1
./file1
lmm_stripe_count:   4
lmm_stripe_size:    8388608
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  20
        obdidx           objid           objid           group
            20        39462252       0x25a256c                0
            24        39465932       0x25a33cc                0
            30        39460521       0x25a1ea9                0
            38        39461956       0x25a2444                0

./file2
lmm_stripe_count:   4
lmm_stripe_size:    8388608
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  35
        obdidx           objid           objid           group
            35        39455744       0x25a0c00                0
            51        39440356       0x259cfe4                0
            13        39487313       0x25a8751                0
            47        39465748       0x25a3314                0
```

Striping implementation notes

- Make sure to use /project file system not /home
  - /project is production lustre file system, /home is GPFS for development
- Default: stripe_count: 1 stripe_size: 1048576
- Manage from command line on file or directory scope via lfs
- Manage from code
  - Cray MPI-IO info hints striping_unit, striping_factor (eg MPICH_MPIIO_HINTS=*: striping_unit=8388608: striping_factor=48) on file creation
  - Can do ioctl system call yourself passing LL_IOC_LOV_SETSTRIPE with structure for count and size -- ROMIO example:
    - https://github.com/pmodels/mpich/blob/master/src/mpi/romio/adio/ad_lustre/ad_lustre_open.c#L114
- Files and directories inherit striping patterns from the parent directory
- File cannot exist before setting striping pattern
  - Properties set in MDS on file creation
- Stripe count cannot exceed number of OSTs (56)
  - lfs osts
- Striping cannot be changed once file created
  - Need to re-create file – copy to directory with new striping pattern to change it

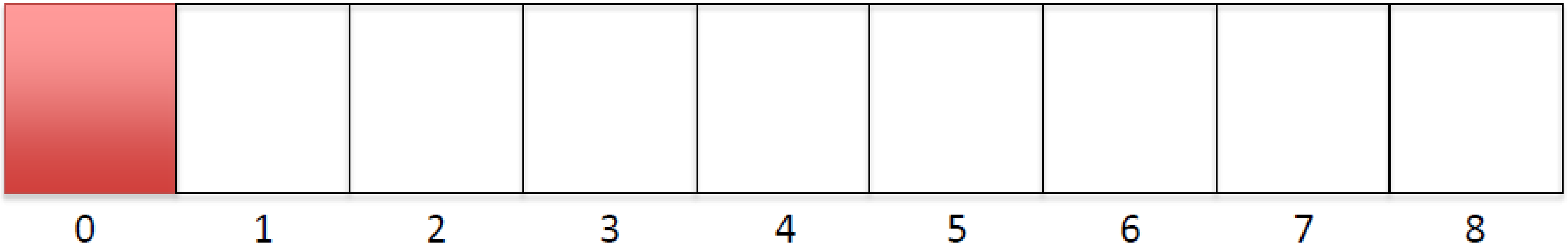
Lustre file system caching

- Lustre cache hierarchy - node client and OSS
    - Node client cache effect impacted by application, shares node DRAM
- Depending on access pattern can see widely varying performance based on cache utilization
    - For writes, MPI_File_sync, posix fsync (force write to disk) calls negate all cache effects and force write to HDD
        - Write call (mpi-io or posix) for limited data sizes may return with only local cache updated – very fast
        - OSS cache utilization impacted by other users
        - Understand when your application is fsync'ing in underlying IO libraries or directly in your code
    - For reads on a file, will have to come from HDD unless very recently written
        - Generally no cache effect for real applications – ie reading initial data file
        - IO Benchmarks such as IOR often show this cache effect though if they are run in both write and read mode
            - ior –C to re-order rank read, or write run with –K and then separate read run and eliminate cache effects

Extent lock contention

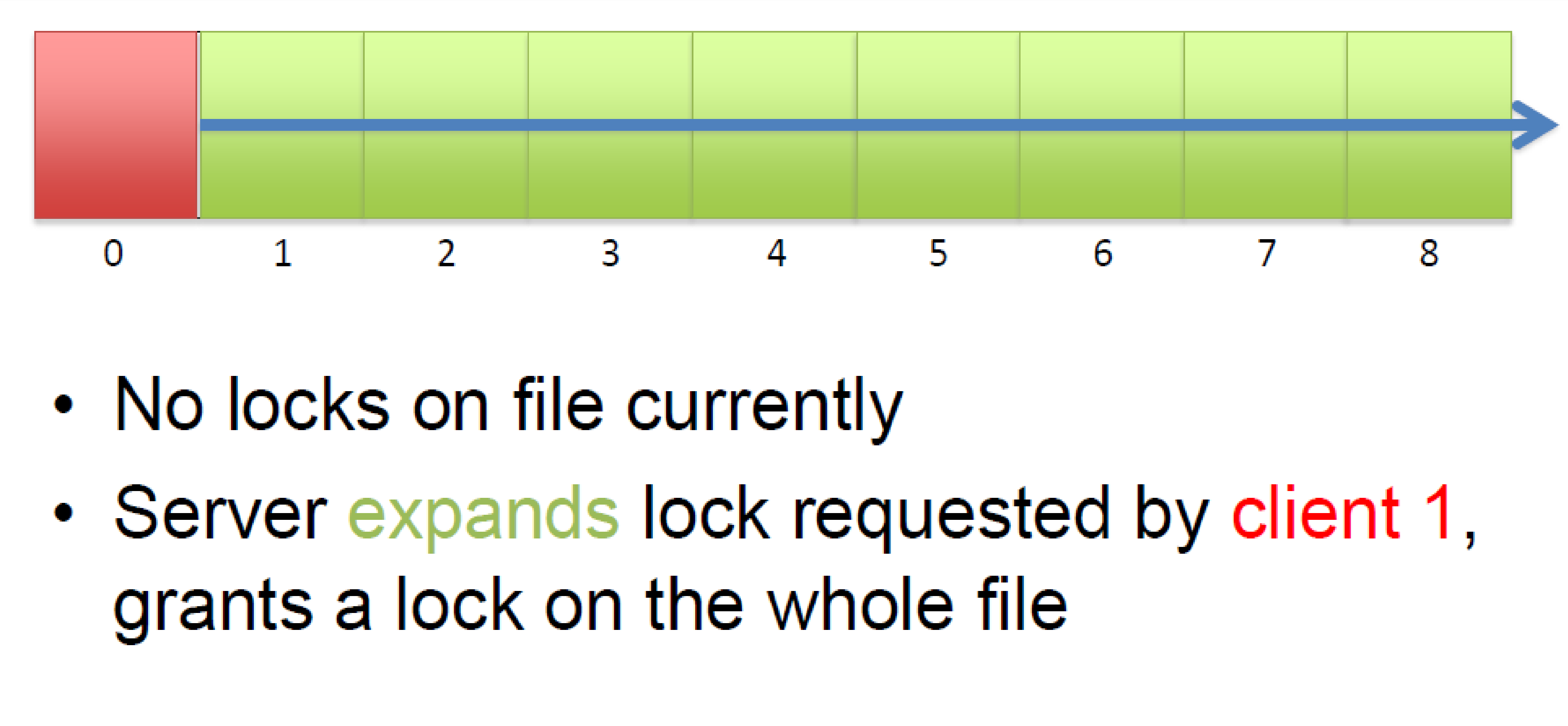
- Each rank (client) needs its own lock when accessing striped data for a given file on an OST
  - If more than one rank concurrently accesses same file on OST, causes extent lock contention
  - Concurrent access improves storage bandwidth
- Extent Locks managed by OSS with (LDLM) Lustre Distributed Lock Manager (LDLM)
  - The LDLM provides a means to ensure that data is updated in a consistent fashion across multiple OSS and OST nodes
- Following slides detail simple example illustrating this issue
  - File with 1 stripe existing entirely on 1 OST accessed by 2 ranks
- Cray MPI-IO has a current limited mitigation for this (cray_cb_write_lock_mode=1 – shared lock locking mode – will be discussed later)
- Extent locks aren't' an issue until data reaches the server
  - If all data locally cached won't see this overhead

Extent lock contention continued



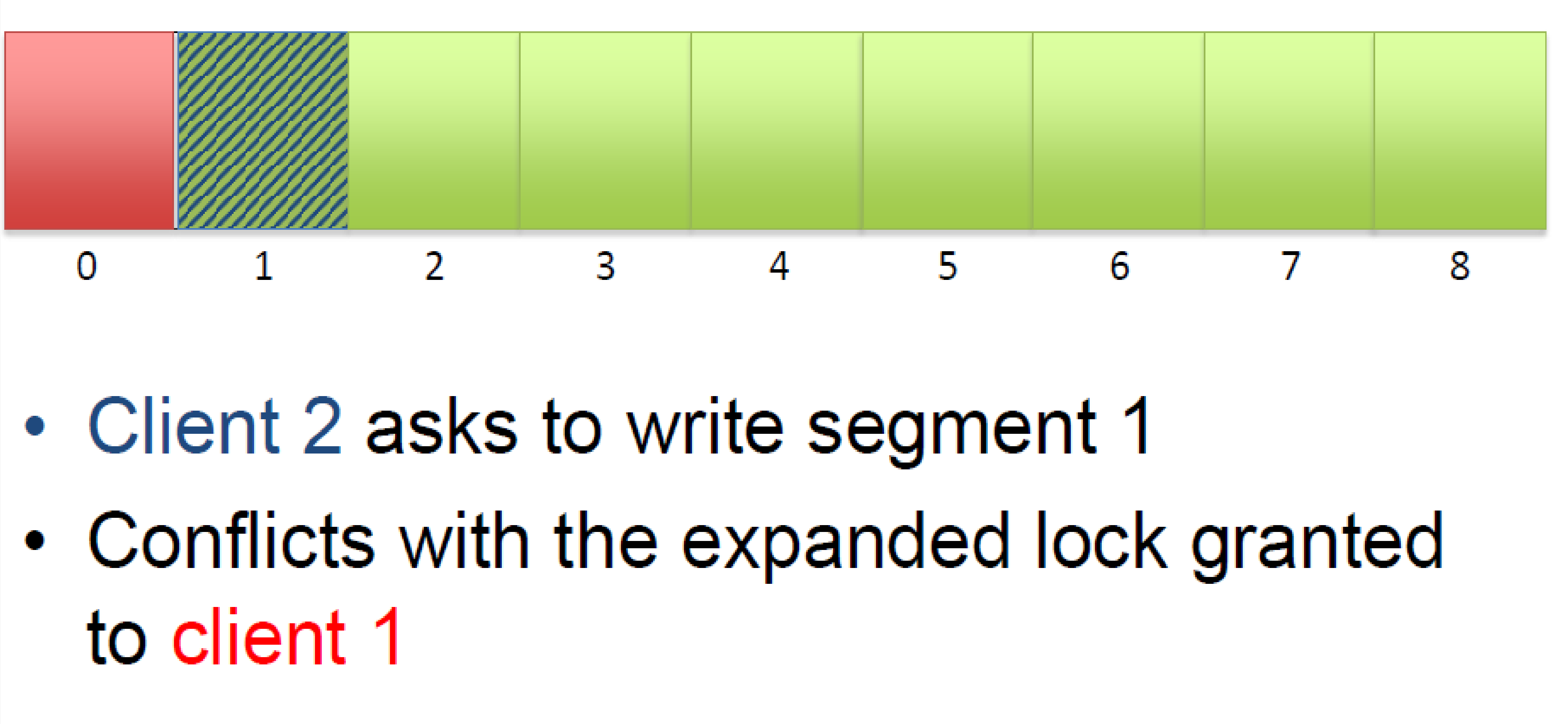
- Single OST view of a file, also applies to individual OSTs in a striped file
- Two clients, doing strided writes
- Client 1 asks to write segment 0 (Assume stripe size segments)

Extent lock contention continued



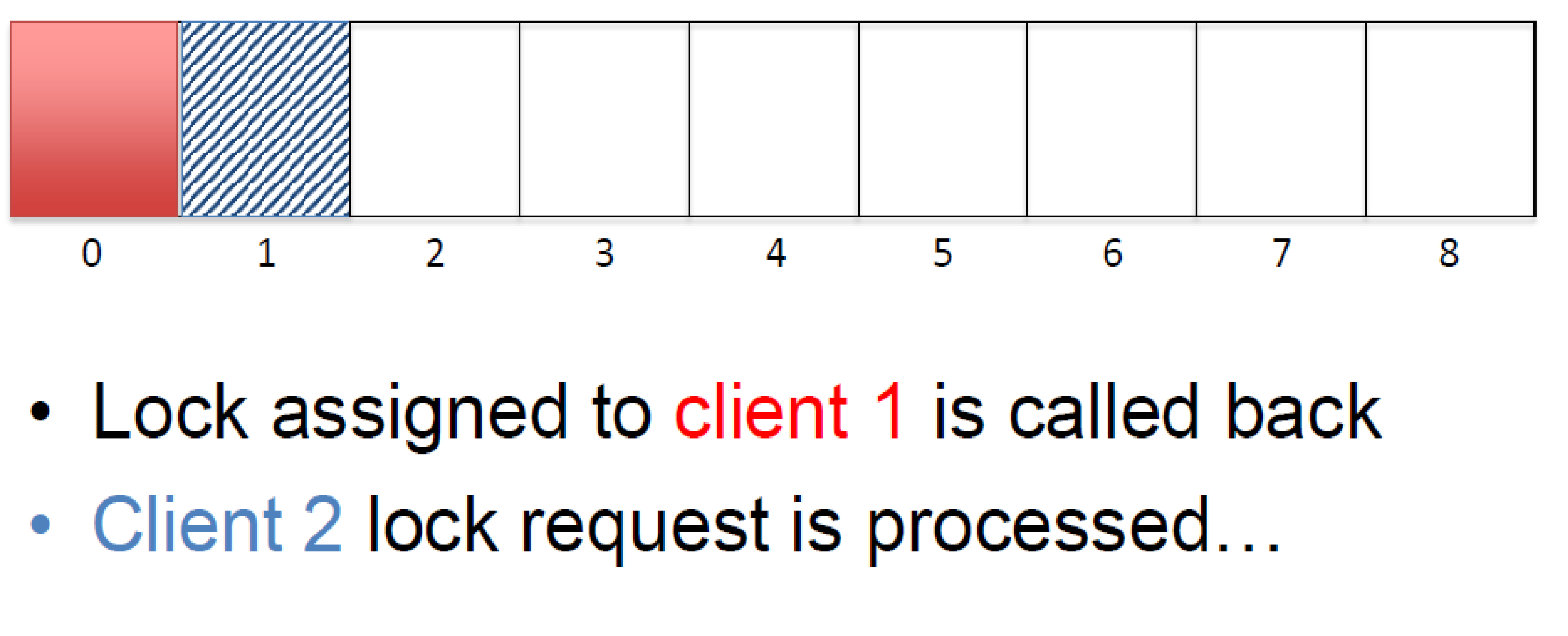
- No locks on file currently
- Server expands lock requested by client 1, grants a lock on the whole file

Extent lock contention continued



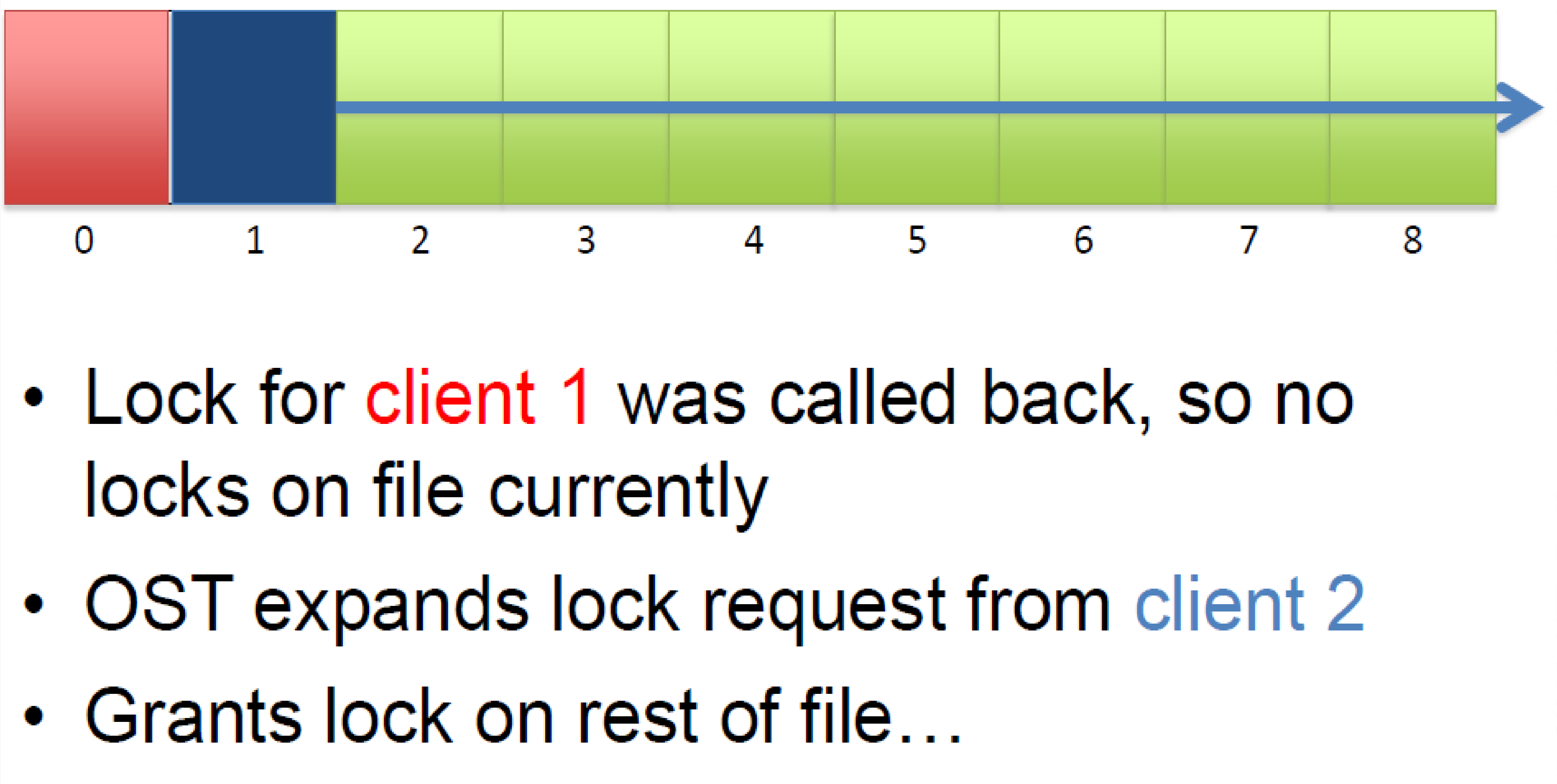
- Client 2 asks to write segment 1
- Conflicts with the expanded lock granted to client 1

Extent lock contention continued



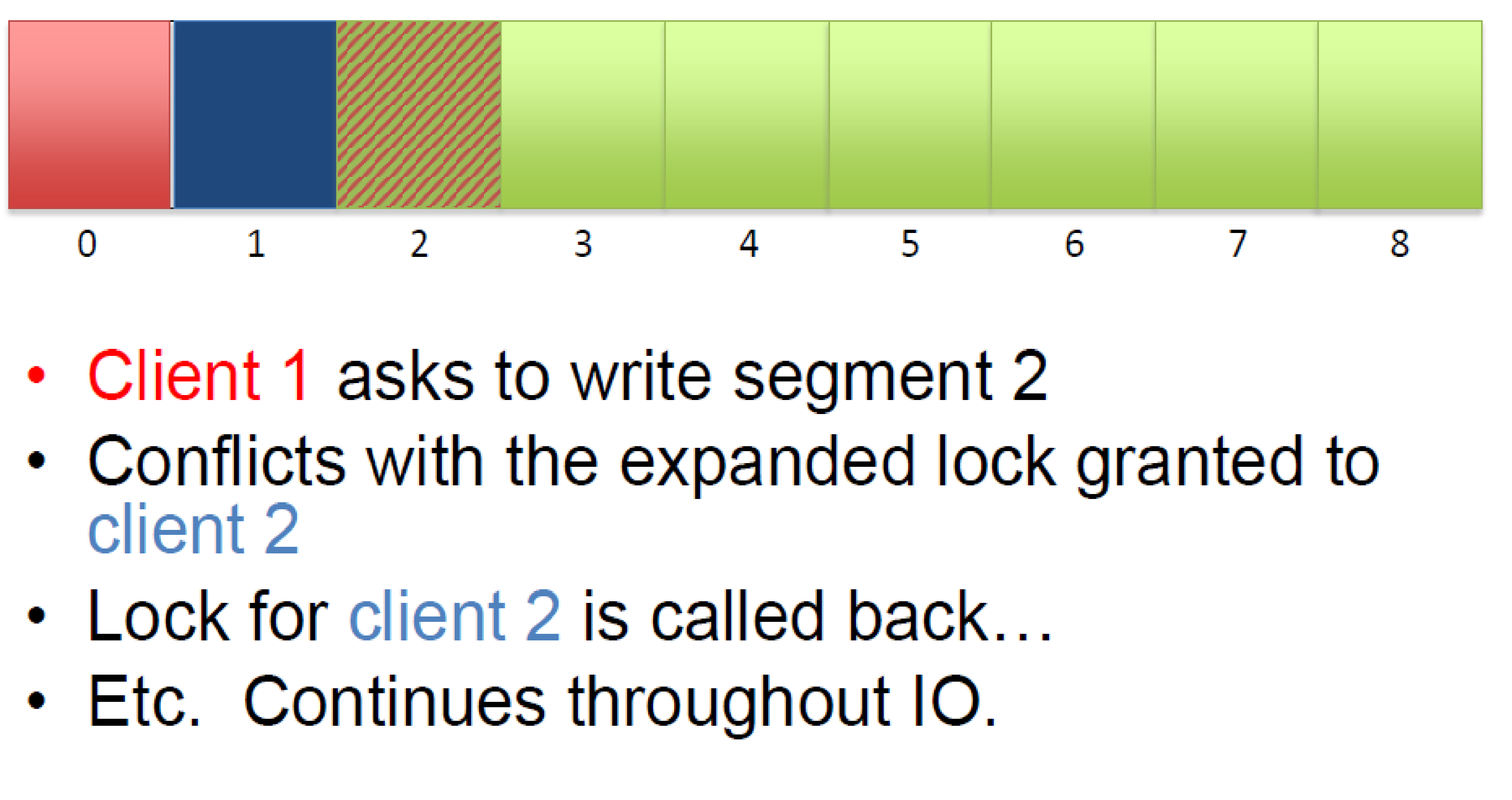
- Lock assigned to client 1 is called back
- Client 2 lock request is processed…

Extent lock contention continued



- Lock for client 1 was called back, so no locks on file currently
- OST expands lock request from client 2
- Grants lock on rest of file…

Extent lock contention continued



- Client 1 asks to write segment 2
- Conflicts with the expanded lock granted to client 2
- Lock for client 2 is called back…
- Etc. Continues throughout IO.

Cray MPI-IO, tuning and profiling, io libraries

Overview of Cray MPI-IO, tuning with Cray MPI-IO, profiling within Cray MPI-IO and externally, and sample IO libraries available on Theta.

Cray MPI-IO Overview

- Rest of presentation will focus alot on large shared file performance with Cray MPI-IO
  - Theta is Cray machine, Cray stack, Cray Programming Environment -- Cray MPI-IO commonly used
- Based on MPICH-MPIIO (ROMIO)
- Facilitates parallel single shared file access
  - Independent
    - Each process accesses file directly – eg MPI_File_write_at
  - Collective
    - Data is aggregated to or from subset of processes which optimally access the file – eg MPI_File_write_at_all
      - Can aggregate a lot of small file accesses from several ranks to single large stripe-sized (aligned) access from single rank
      - Can turn collective IO off without changing code via hints
        - romio_cb_read=disable, romio_cb_write=disable (default is auto determine at runtime)
- Many tuning parameters
  - man intro_mpi
- Underlying IO layer for many IO libraries
  - HDF5, PNetCDF
  - Tuning Cray MPI-IO improves performance of IO libraries

Cray MPI-IO Collectives

- MPI_File_*_all calls
- Facilitates optimal aligned striped access
    - Can aggregate smaller and discontiguous per-process chunks of data into contiguous stripe-size file access
- Default number of aggregators set to match stripe_count (1 client per OST) and size set to stripe size
    - Number of aggregator nodes (cb_nodes hint) defaults to the striping factor (count) and can be changed
    - cray_cb_nodes_multiplier hint increases number of aggregators per stripe (multiple clients per OST)
        - Total aggregators = cb_nodes x cray_cb_nodes_multiplier
- Collective buffer size is the stripe size --- cb_buffer_size hint is ignored
    - ROMIO collective buffer still allocated with cb_buffer_size but not used
- Can choose to run MPICH MPI-IO (ROMIO) instead – can see the open source code to know what it is doing
    - cb_align=3, all ROMIO hints apply

Key Cray MPI-IO Profiling and Tuning Environment variables

- Cray MPI-IO provides many environment variables to gain insight into performance
- MPICH_MPIIO_STATS=1
  - MPI-IO access patterns for reads and writes written to stderr by rank 0 for each file accessed by the application on file close
- MPICH_MPIIO_STATS=2
  - set of data files are written to the working directory, one file for each rank, with the filename prefix specified by the MPICH_MPIIO_STATS_FILE env var
- MPICH_MPIIO_TIMERS=1
  - Internal timers for MPI-IO operations, particularly useful for collective MPI-IO
- MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
- MPICH_MPIIO_AGGREGATOR_PLACEMENT_STRIDE
- MPICH_MPIIO_HINTS=<file pattern>:key=value:…
- MPICH_MPIIO_HINTS_DISPLAY=1

MPICH_MPIIO_STATS=1 Sample Output for IOR

```
+-------------------------------------------------------+
| MPIIO write access patterns for testFile
|   independent writes      = 0
|   collective writes       = 4096
|   independent writers     = 0
|   aggregators             = 48
|   stripe count            = 48
|   stripe size             = 1048576
|   system writes           = 4096
|   stripe sized writes     = 4096
|   aggregators active      = 0,0,0,4096 (1, <= 24, > 24, 48)
|   total bytes for writes  = 4294967296 = 4096 MiB = 4 GiB
|   ave system write size   = 1048576
|   read-modify-write count = 0
|   read-modify-write bytes = 0
|   number of write gaps    = 0
|   ave write gap size      = NA
+-------------------------------------------------------+
```

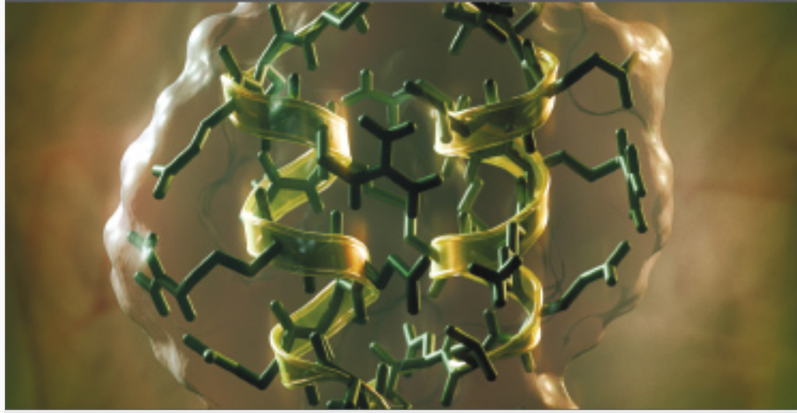
MPICH_MPIIO_TIMERS=1 Sample Output for IOR

```
| MPIIO write by phases, all ranks, for testFile
|   number of ranks writing         =    48
|   number of ranks not writing     =  4048
|   time scale: 1 = 2**1      clock ticks    min         max         ave
|                                ----------  ----------  ---------- ---
|   total                        =                       515639319
|
|   imbalance                    =     70473     4113200     4038142  0%
|   open/close/trunc             =  20449942    21682440    20547164  3%
|   local compute                =   1264081    72814196     7986327  1%
|   wait for coll                =    342868   469936675   261193401 50%
|   collective                   =   5040048     7905176     6537893  1%
|   exchange/write               =   2857647    32294255     3413887  0%
|   data send (*)                =   6250725   403979537   205475108 39%
|   file write                   =         0   235316380   226898464 44%
|   other                        =     49224     5699284     3788427  0%
|
|   data send BW (MiB/s)         =                         12967.254
|   raw write BW (MiB/s)         =                         11742.909
|   net write BW (MiB/s)         =                          5167.271
|
| (*) send and write overlap when number ranks != number of writers
+------------------------------------------------------------------------+
```

Craypat IOR Collective MPI-IO Profiling for Read

- module load perftools
- pat_build –w –g io –g mpi <binary name>
- pat_report –s pe=ALL <pat-dir-name>

```
Table 5:  File Input Stats by Filename

    Time |  Read MBytes |    Read Rate |   Reads |  Bytes/ Call | File PE

 0.645434 | 1,280.719242 | 1,984.275263 | 9,952.0 |   134,940.86 | Total

|--------------------------------------------------------------------------

| 0.585577 | 1,280.000000 | 2,185.878594 | 1,280.0 | 1,048,576.00 | testFile

||-------------------------------------------------------------------------

|| 0.076877 |   160.000000 | 2,081.242606 |   160.0 | 1,048,576.00 | pe.16

|| 0.074686 |   160.000000 | 2,142.314659 |   160.0 | 1,048,576.00 | pe.17

…..
```

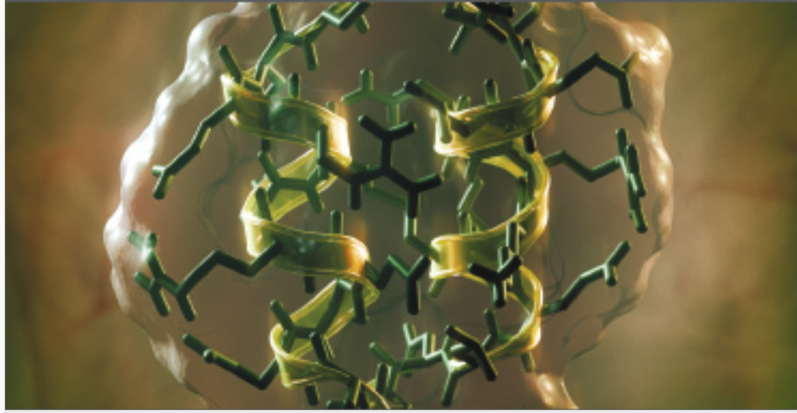


Shows the total and individual aggregator process read bandwidth in MB/s

Craypat IOR Collective MPI-IO Profiling for Write

- module load perftools
- pat_build –w –g io –g mpi <binary name>
- pat_report –s pe=ALL <pat-dir-name>

```
Table 6:  File Output Stats by Filename (maximum 15 shown)

    Time | Write MBytes | Write Rate |  Writes |  Bytes/ Call | File PE

 6.459586 | 1,280.249772 | 198.193774 | 6,612.0 |   203,030.73 | Total

|---------------------------------------------------------------------------

| 6.376338 | 1,280.000000 | 200.742172 | 1,280.0 | 1,048,576.00 | testFile

||--------------------------------------------------------------------------

|| 0.838935 |   160.000000 | 190.718093 |   160.0 | 1,048,576.00 | pe.32

|| 0.801623 |   160.000000 | 199.595064 |   160.0 | 1,048,576.00 | pe.48

…
```



Shows the total and individual aggregator process write bandwidth in MB/s

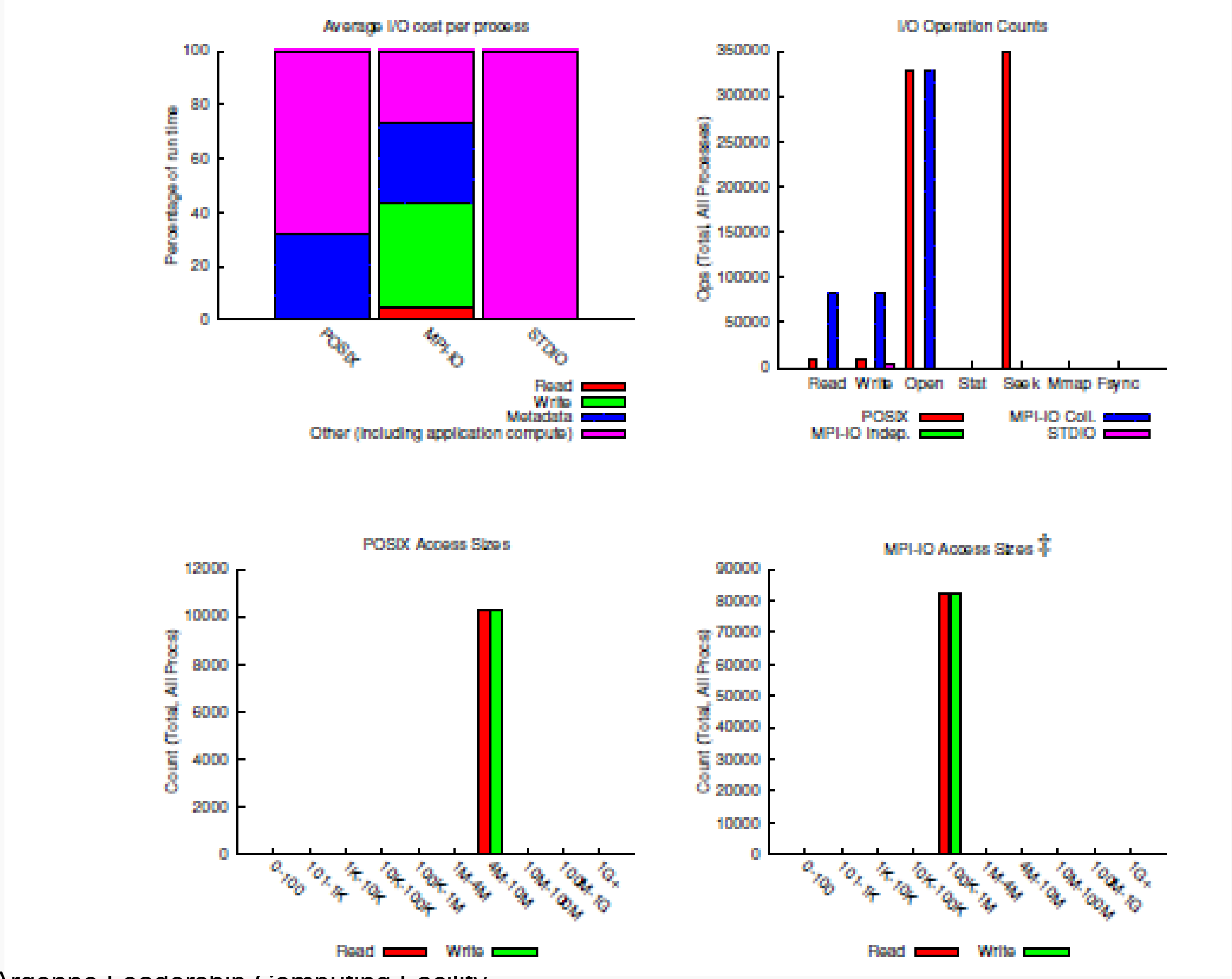
Darshan Profiling

- https://www.alcf.anl.gov/user-guides/darshan
- An open-source tool developed for statistical profiling of I/O
- Designed to be lightweight and low overhead
  - Finite memory allocation for statistics (about 2MB) done during MPI_Init
  - Overhead of 1-2% total to record I/O calls
  - Variation of I/O is typically around 4-5%
  - Darshan does not create detailed function call traces
- No source modifications
  - ‘module list’ should show darshan
  - Uses PMPI interfaces to intercept MPI calls
  - Use ld wrapping to intercept POSIX calls
  - Can use dynamic linking with LD_PRELOAD=$DARSHAN_PRELOAD instead
- Stores results in single compressed log file

Darshan Profiling Continued

- Make sure postscript-to-pdf converter is loaded
  - module load texlive
- IO characterization file placed here at job completion: /lus/theta-fs0/logs/darshan/theta/<YEAR>/<MONTH>/<DAY> with format <USERNAME>_<BINARY_NAME>_id<COBALT_JOB_ID>_<DATE>-<UNIQUE_ID>_<TIMING>.darshan
- darshan-job-summary.pl command for charts, table summaries
  - darshan-job-summary.pl <darshan_file_name> --output darshansummaryfilename.pdf
- darshan-parser for detailed text file
  - darshan-parser <darshan_file_name> > darshan-details-filename.txt

Darshan-job-summary.pl Example Using IOR Collective MPI-IO



Theta IO Libraries

Cray PE offers several pre-built I/O libraries

module avail, module list, module load

- HDF5
  - cray-hdf5-parallel/1.10.1.1
- NetCDF
  - cray-netcdf/4.4.1.1.6(default)
- PNetCDF
  - cray-parallel-netcdf/1.8.1.3(default)
- Adios

These libraries offer capabilities to make managing large parallel I/O easier

Pay attention to MPI-IO settings

- HDF5 allows user to specify independent or collective IO for raw data and metadata
  - Raw data can be written collectively via property list
    - hid_t xferPropList = H5Pcreate(H5P_DATASET_XFER);
    - H5Pset_dxpl_mpio(xferPropList, H5FD_MPIO_INDEPENDENT); or H5Pset_dxpl_mpio(xferPropList, H5FD_MPIO_COLLECTIVE);
  - Metadata can be written collectively via H5Pset_all_coll_metadata_ops, H5Pset_coll_metadata_write as of release 1.10.0

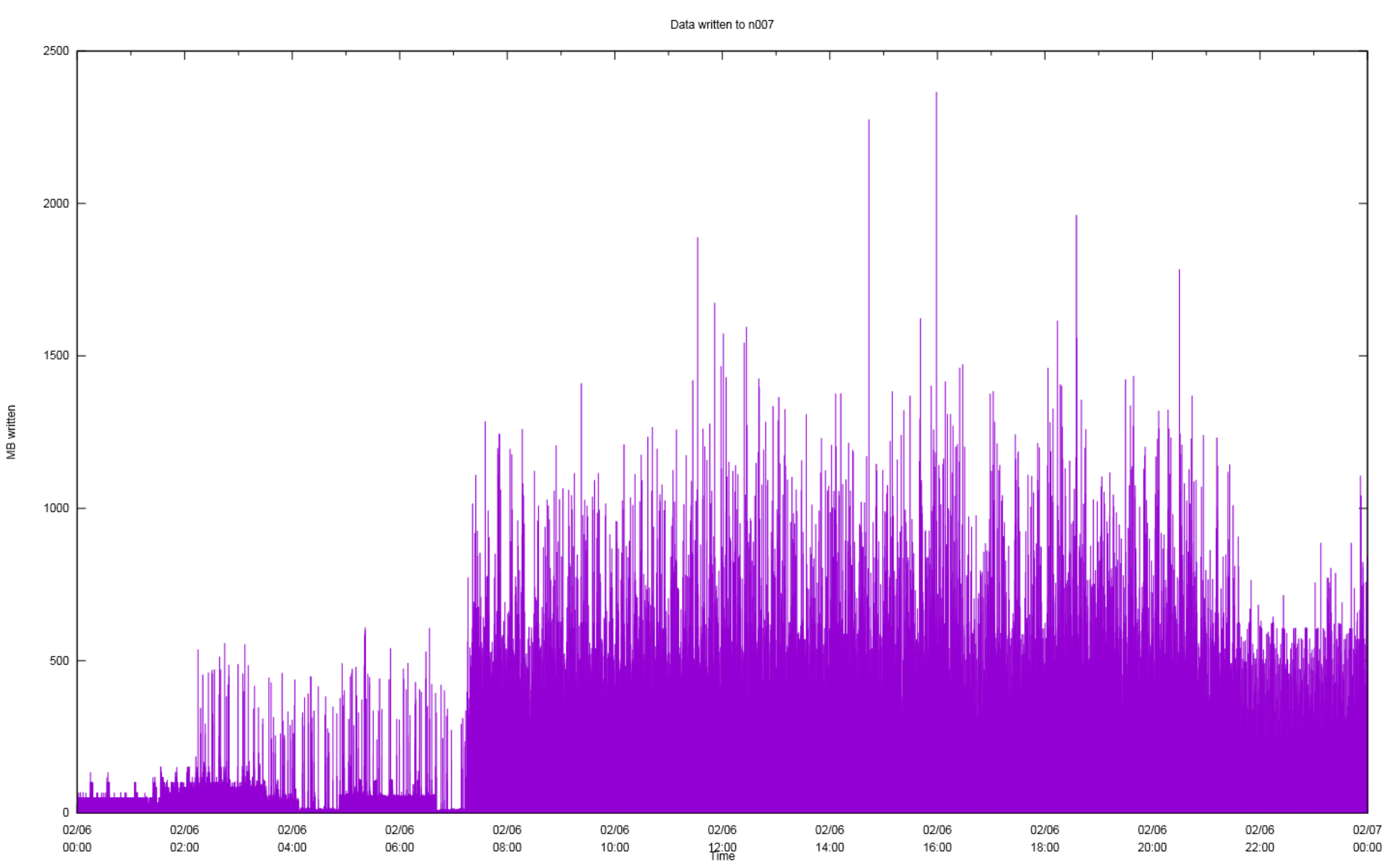
Lustre performance investigations, analysis and best practices

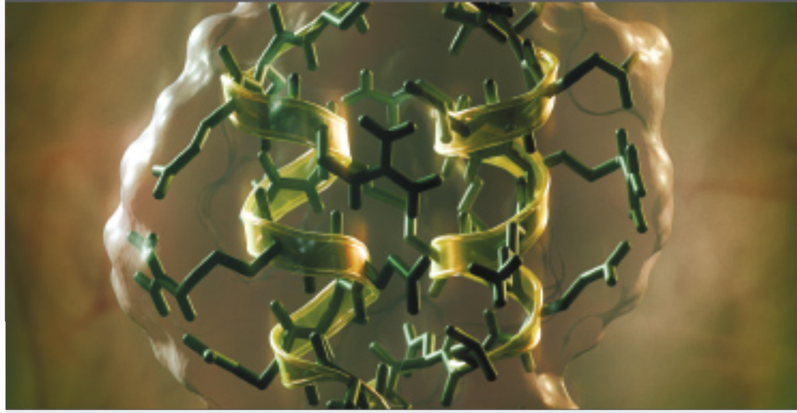
In-depth charts and explanations for various performance characteristics exhibited on Theta Lustre utilizing various benchmarks, primarily IOR.

Dragonfly Network and Lustre Jitter

- Network and Lustre shared with and impacted by other users
  - No job isolation
- Currently 1 Metadata Sever (MDS) shared by all users
  - Performance for a particular transaction easily bottlenecked by traffic surge to MDS or one of your OSSs
- Only way to truly minimize jitter is to have the system to yourself
  - No impact then from other users
- When running IO performance tests run several iterations best result usually represents least jitter
  - Best result is the one that basically had minimal interfence in network or lustre from other users

Lustre OSS N007 activity – 5 sec intervals





24-hour time period 2017-02-06 of OSS traffic on a particular server, pattern is typical of other servers. At this time lustre utilization relatively low, about 100 mb/s with spikes to 500 mb/s.

File storage approaches

- File-per-process (FPP)
  - scales well in Lustre, shared file has issues with extent lock contention and MDS overhead
  - FPP scales well as ranks exceed number of OSTs multiple ranks can write to same OST but within separate files without extent lock contention issue
  - FPP has management and consumption issues at scale with sheer number of files
- Single shared file
  - MPI-IO most common for access
  - Independent vs Collective
    - Weigh cost of collective aggregation against optimization of LFS access
      - For small discontiguous chunk data collective faster
      - For larger contiguous data independent read has no lock contention may be faster
        - Also LFS node caching may mitigate extent lock issues for write
      - If rank data is stripe aligned independent writes may also be faster
      - Experiment – implement collective calls (MPI_File_*_all) and then turn off collective aggregation via romio_cb_write and romio_cb_read hints to see which performs better

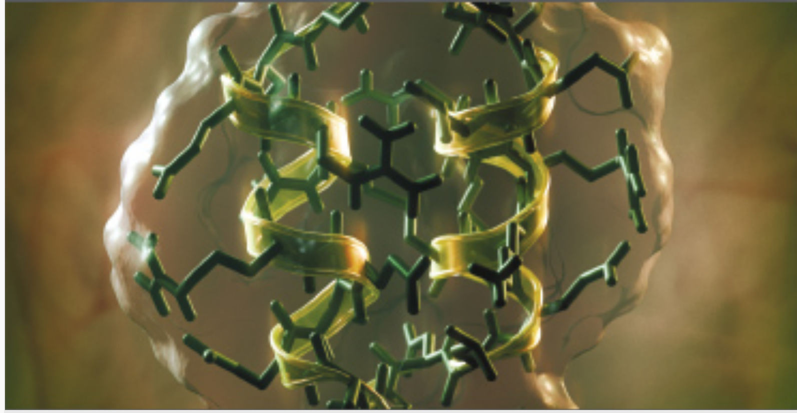
File storage approaches continued

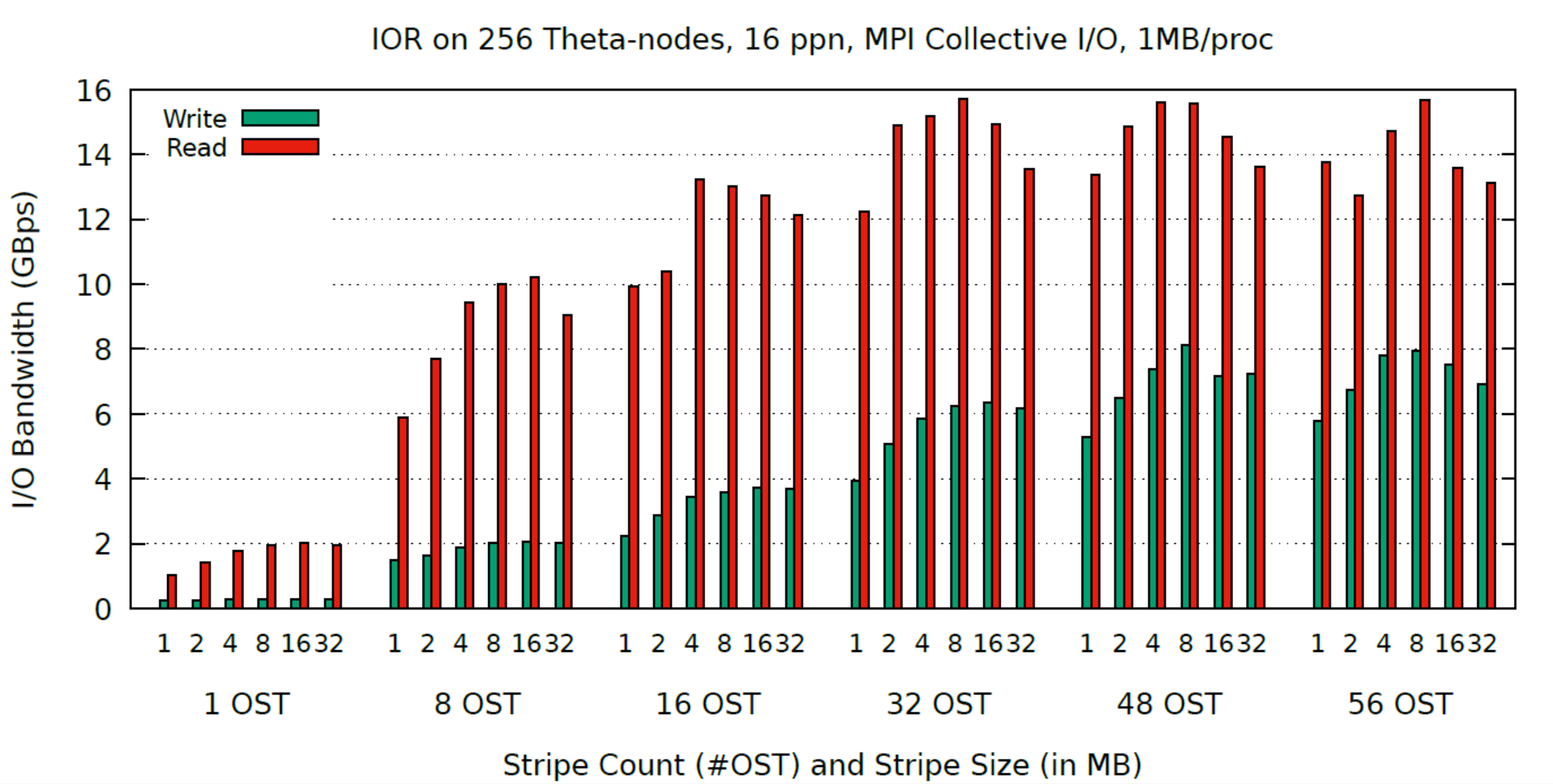
- Multiple shared files (subfiling)
  - Instead of one large shared file accessed by all processes, use multiple shared files individually accessed by subsets of processes
    - Less MDS overhead than one big shared file
    - More manageable than file-per-process
    - Typical implementation: sub-comm with collective MPI-IO for each file
- Many IO libraries currently write singe shared file (HFD5, PNetCDF)
- Shared file extent lock contention issues at the server with > 1 client per OST
  - Only 1 rank at a time can optimally access stripe set on OST
  - With > 1 client per OST, writes are serialized due to LDLM extent lock design in Lustre and performance worse than single client with lock exchange latency overhead

General Striping approaches

- For file-per-process just use 1 stripe
- For shared files, in general more stripes are better and larger stripes (to a point) are better
  - Don’t exceed node count with stripe count
  - Start with 1mb stripe size, increase stripe count to 48
    - Don’t go to full 56 as lustre will bypass assigning slow OSTs to the file at file creation time, going to 48 leaves room for this
    - For this reason don’t explicitly choose OSTs during setstripe, let lustre do it for you
  - Once using 48 OSTs, increase stripe size to between 8 and 32 mb
    - Cray MPI-IO collective buffer is stripe size – potential memory impact for large stripes and collective io
- Place small files on single OST
- Place directories containing many small files on single OSTs (Such as extracting source code distributions from tarballs)
  - Minimize MDS overhead

Shared File Stripe Size vs Count Affect on Performance




IOR on 256 Theta-nodes, 16 ppn, MPI Collective I/O, 1MB/proc
Write 🟩
Read 🟥
I/O Bandwidth (GBps)
0
2
4
6
8
10
12
14
16
1 2 4 8 16 32
1 OST
8 OST
16 OST
32 OST
48 OST
56 OST
Stripe Count (#OST) and Stripe Size (in MB)

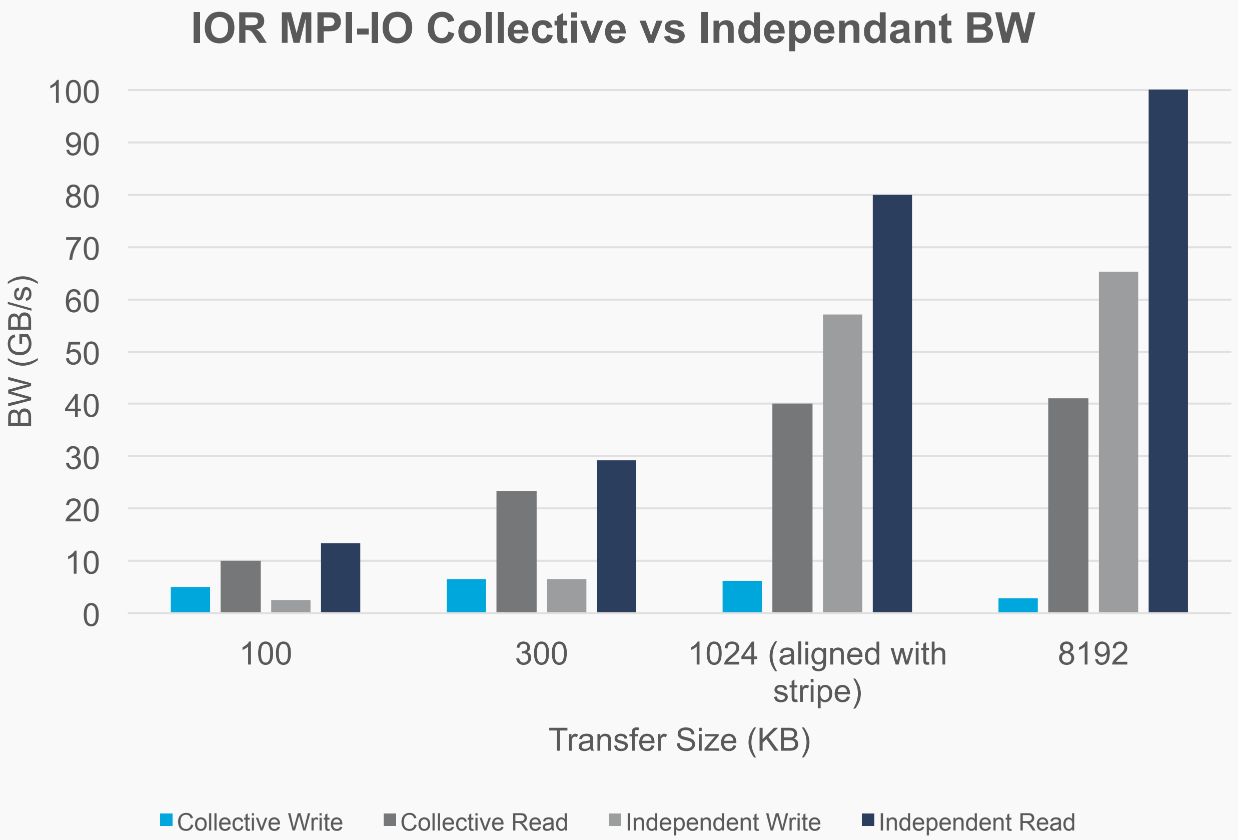
Stripe count more important than stripe size

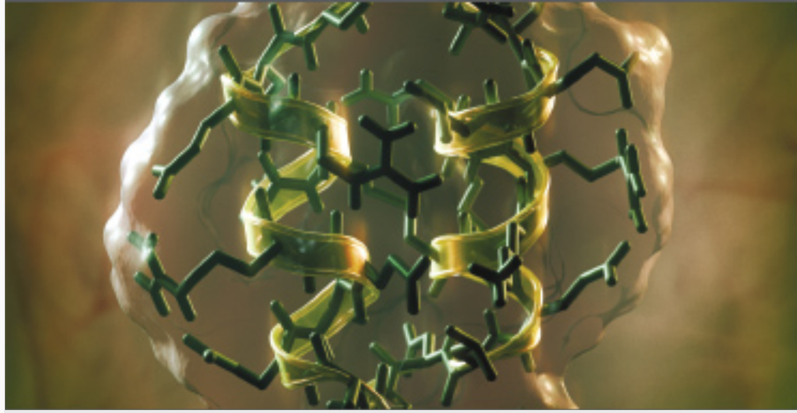
For this example, best results are 48 OST 8mb stripe

Experiment!

IOR MPI-IO Collective vs Independent 1 MB Stripe Size

IOR on 256 nodes 16 ppn 48 OSTs





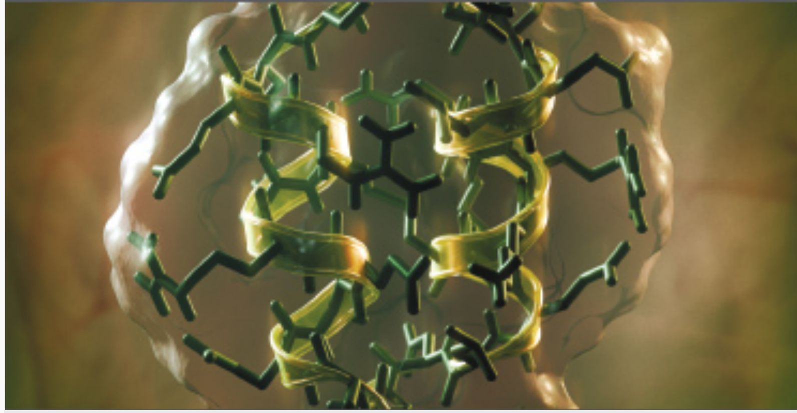
Local node client cache effect exhibited here

Collective write only better for smaller transfers (100k)

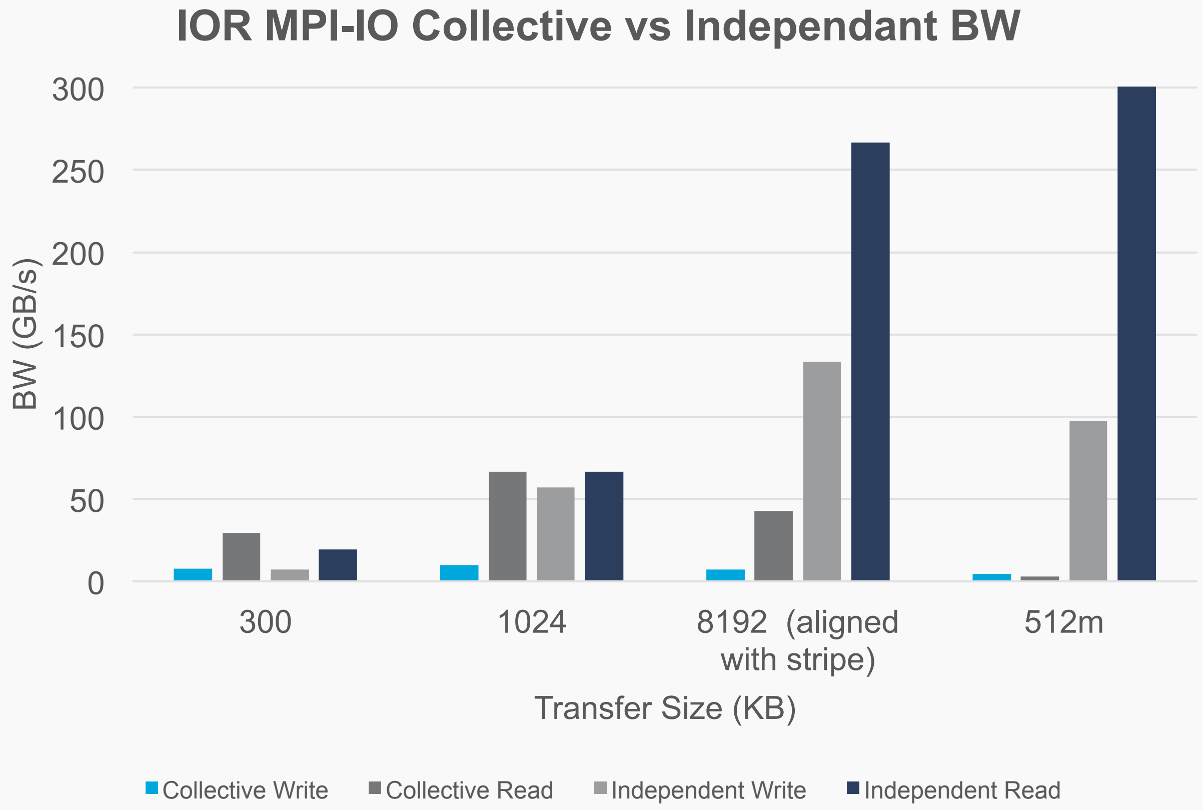
Independent better for larger transfers, independent extent lock contention mitigated by collective aggregation overhead and local cache effects

Collective write actually worse for transfer size > stripe size (8mb)

IOR MPI-IO Collective vs Independent 8 MB Stripe Size



IOR on 256 nodes 16 ppn 48 OSTs

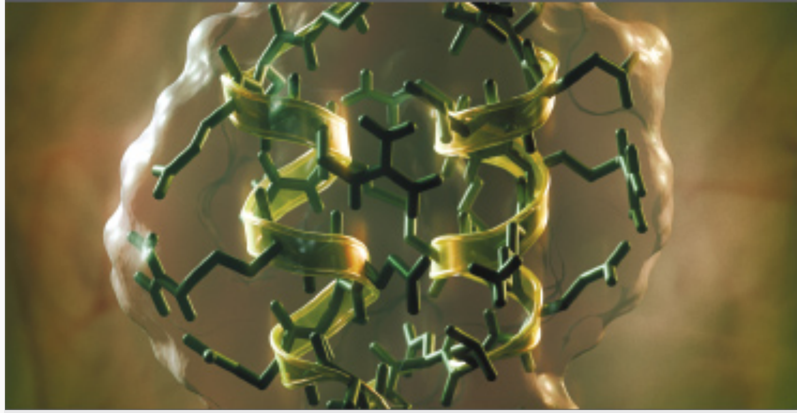


Compared with previous slide, larger stripe size (8m vs 1m) and shows results for larger data transfer (512m)

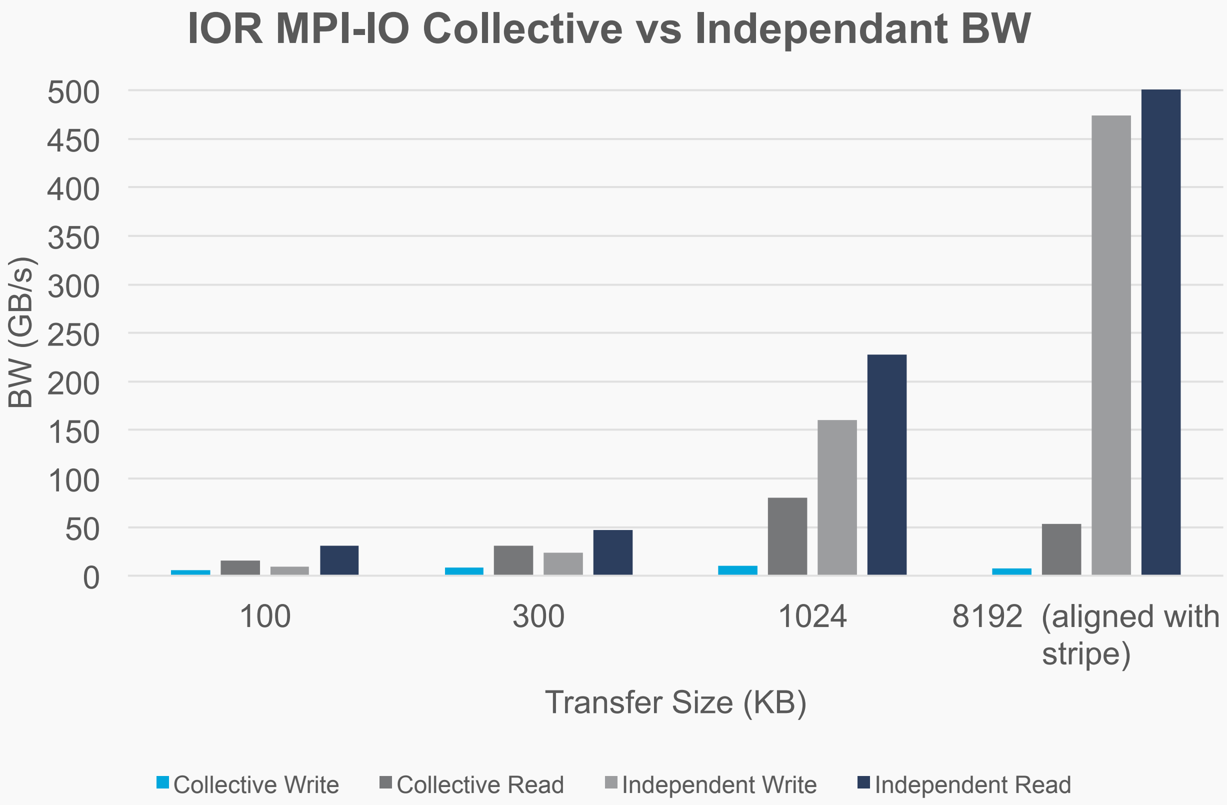
Larger stripe size helps independent more than collective for large data size

At 512m lose some of local client cache effect on independent write

IOR MPI-IO Collective vs Independent 1024 Nodes 8 MB Stripe



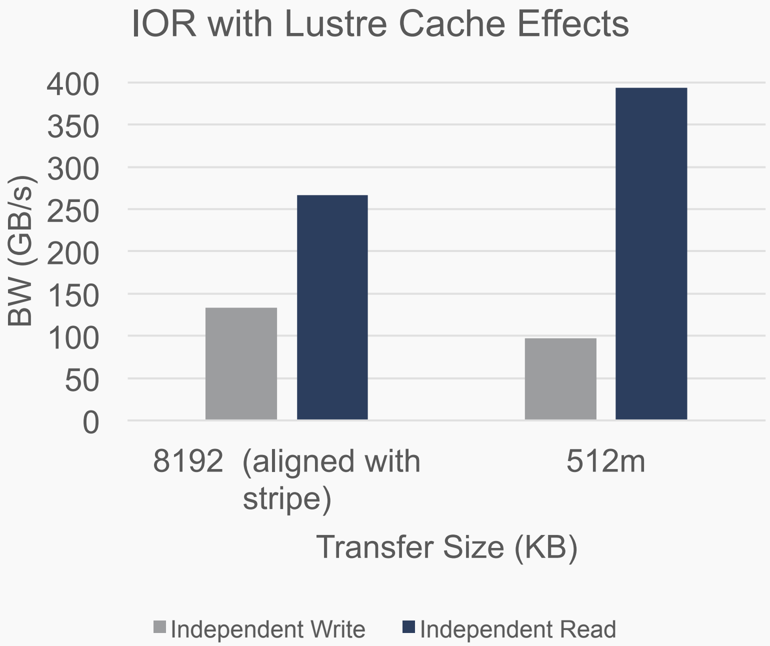
IOR on 1024 nodes 16 ppn 48 OSTs

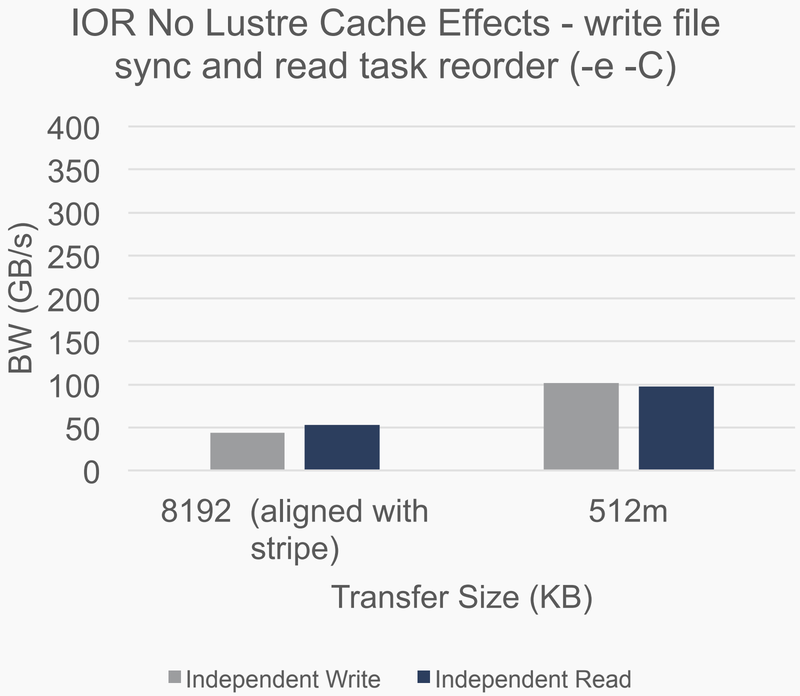


Compared with previous slide, due to node local client caching independent continues to scale well past OST HDD max BW (nearly double)

IOR Independent MPI-IO Effect of Lustre Caching

IOR 256 Nodes 16 PPN 48 OSTs 8 MB stripe size

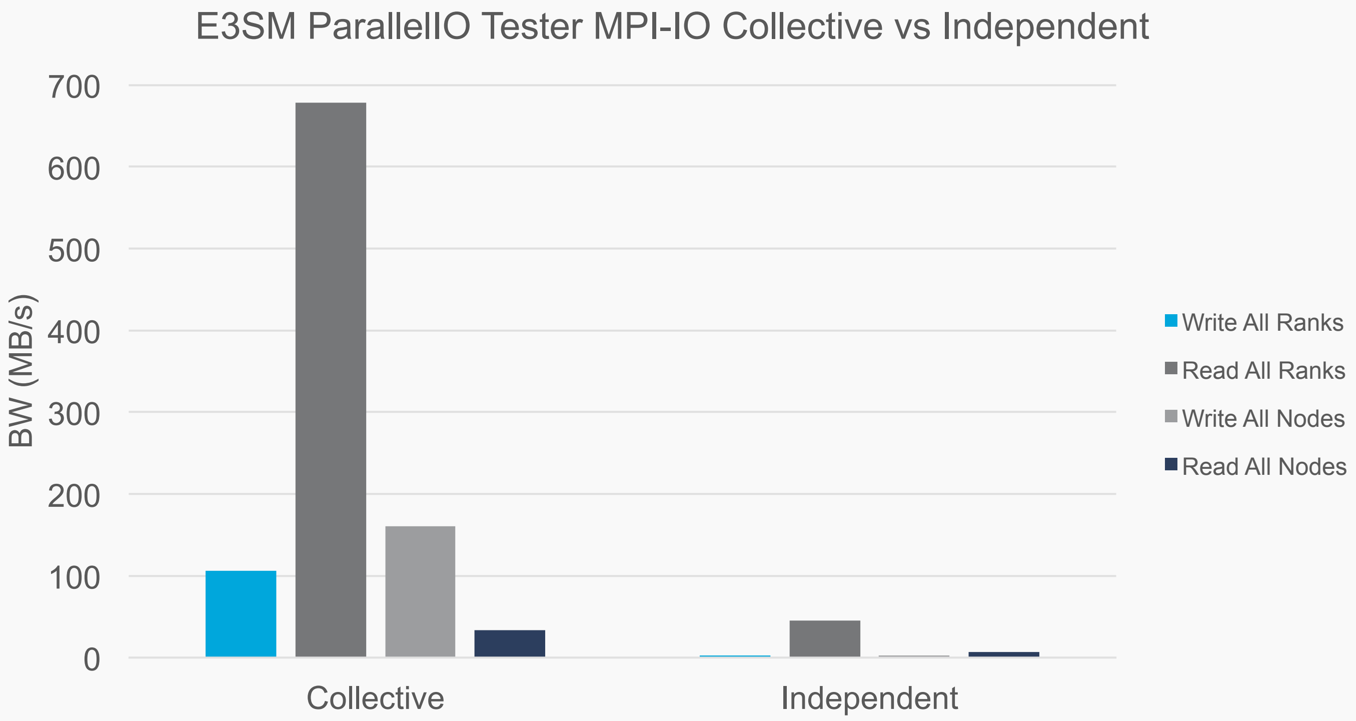


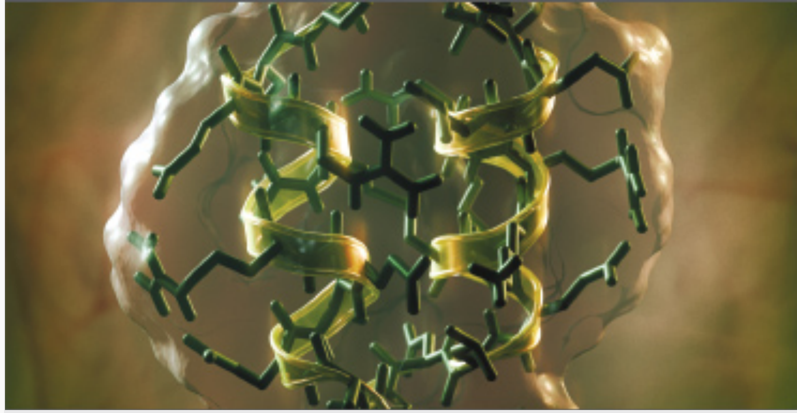


At 8mb transfer size full cache effect BW goes way down for HDD
At 512mb transfer size not getting full cache effect on write BW doesn’t go down for HDD

MPI-IO Collective vs Independent discontiguous data

pioperf on 256 nodes 32 ppn 48 OSTs 8 MB Stripe 3 GB shared file





E3SM Climate Modeling ParellelIO Library performance test tool (pioperf)

Run with data decomposition generated from E3SM running on 8192 ranks with about 350K of highly non-contiguous data

Data is non-contiguous in local buffer and non-contiguous across shared file – every rank accesses every stripe

PNetCDF interface used over MPI-IO backend

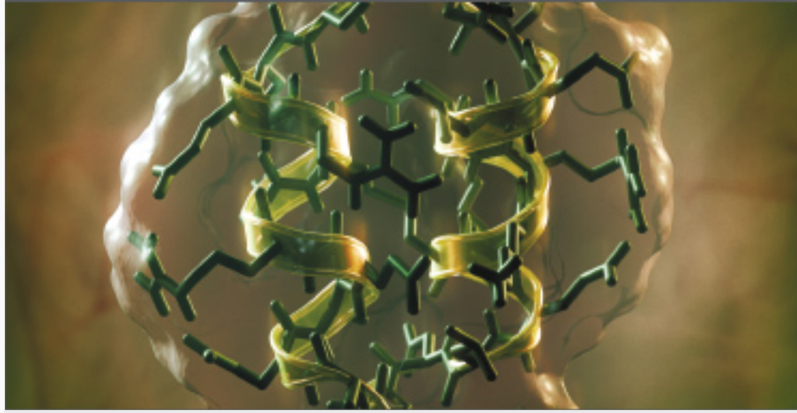
‘Write/Read All Ranks’ means no pre-aggregation before MPI-IO call

‘Write/Read All Nodes’ means pre-aggregation of 32 ranks of data to node first, MPI-IO over subcomm of 1 rank per node

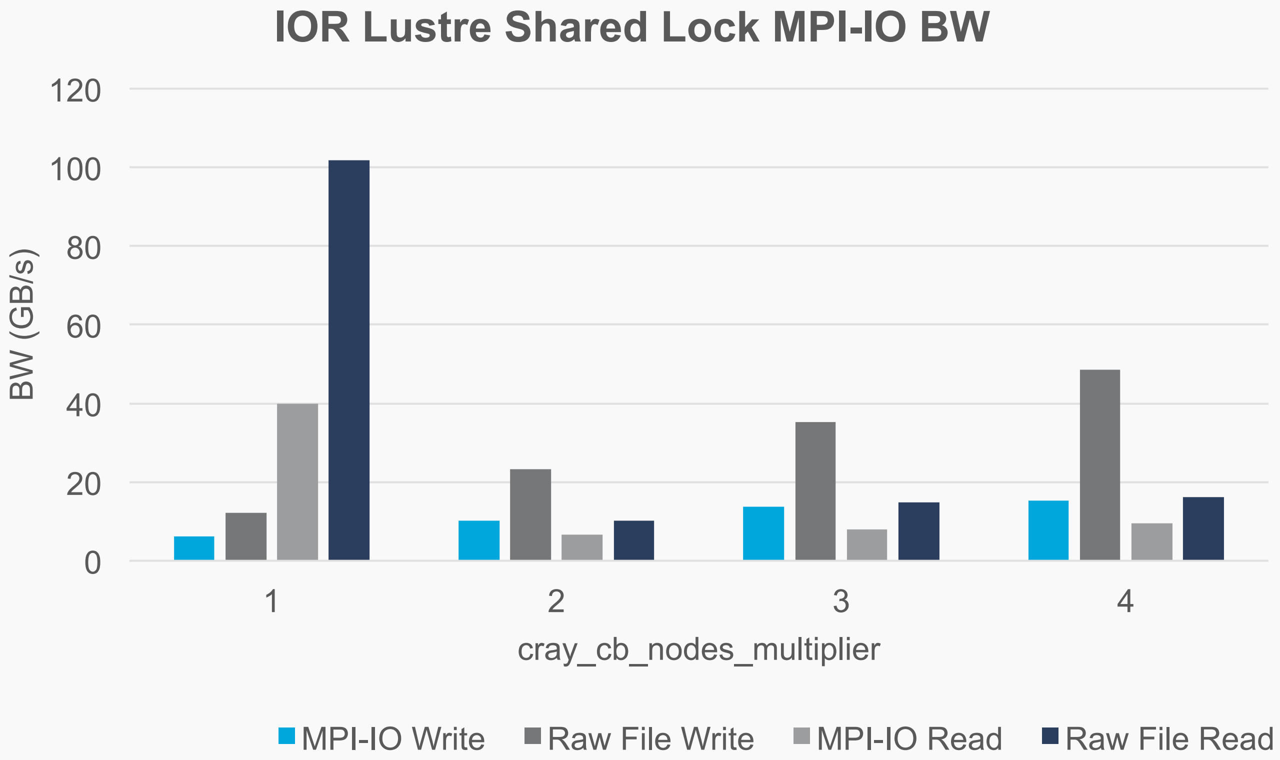
Cray Collective MPI-IO Shared Lock Utilization

- Shared lock locking mode. A single lock is shared by all MPI ranks that are writing the file
    - Hint: cray_cb_write_lock_mode=1
    - Default is 0 for standard lock locking mode.
- Enables multiple clients (aggregators) to simultaneously write to the same file with no extent lock contention
    - Use cray_cb_nodes_multiplier
- Limitations
    - All accesses to the file must be via collective io
        - romio_no_indep_rw must be set to true
        - Any non-collective mpi-io will cause abort or hang
        - HDF5 and PNetCDF currently rely on at least some indepenant access (eg HDF5 meta-data) and therefore cannot use this setting
        - Darshan won’t work because of independent write.
- Sample hints settings: export MPICH_MPIIO_HINTS=*:cray_cb_write_lock_mode=1:cray_cb_nodes_multiplier=4:romio_no_indep_rw=true

IOR MPI-IO Collective Shared Lock Performance Tests



IOR on 256 nodes 16 ppn 48 OSTs 1MB Stripe 1 MB Transfer size

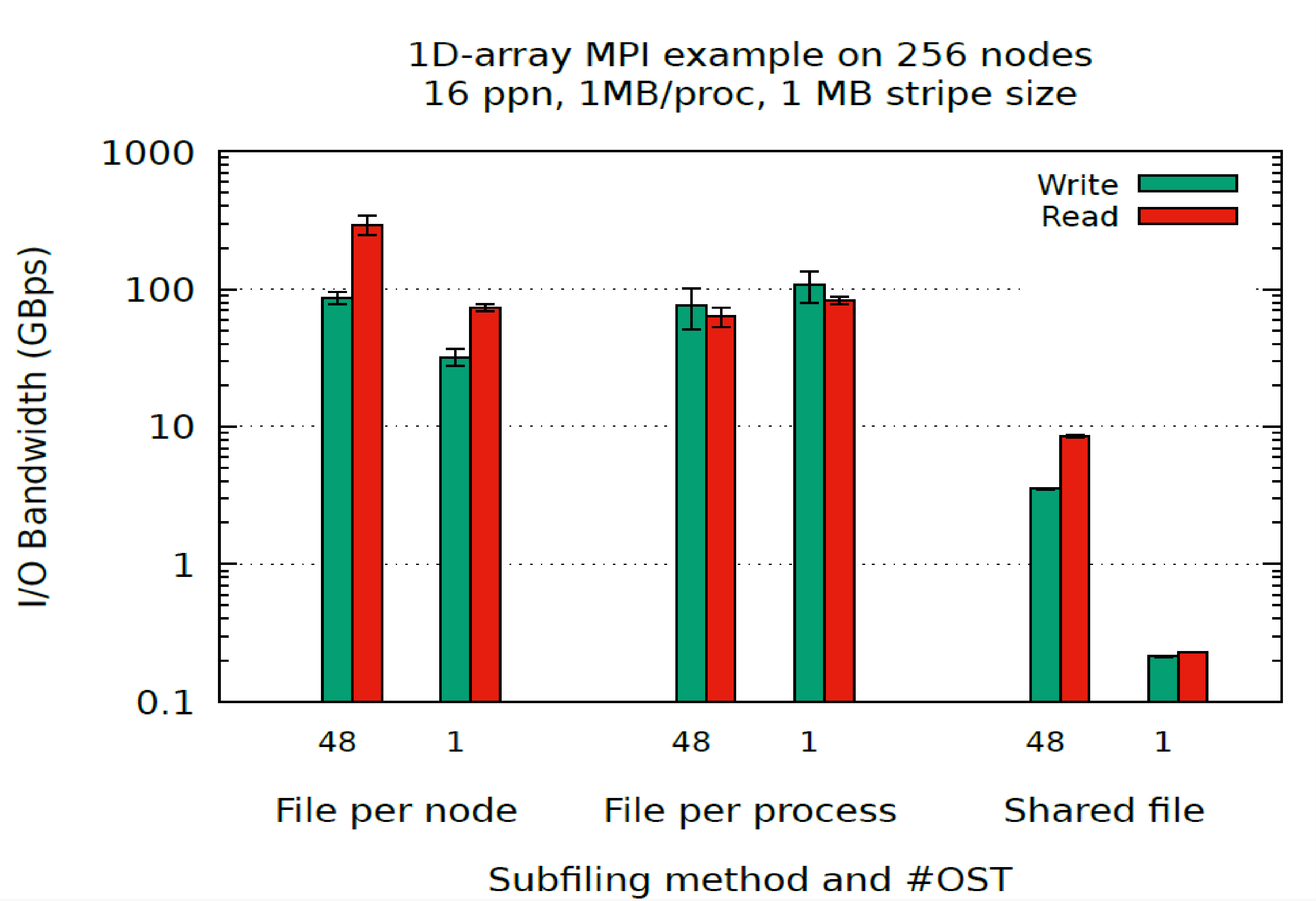


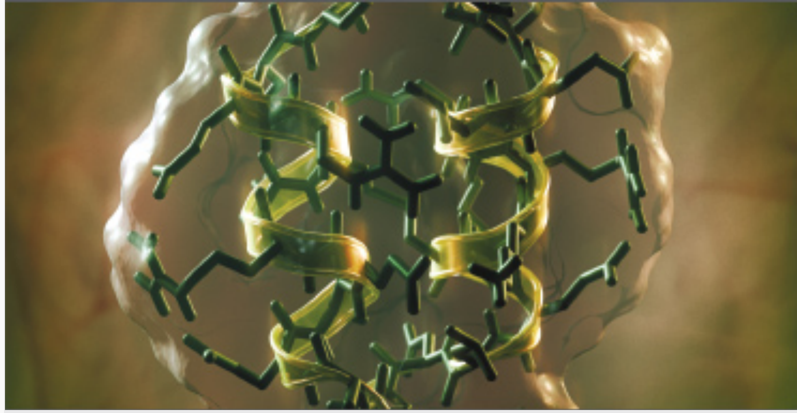
'Raw File Write' and 'Raw File Read' times taken from MPICH_MPIIO_TIMERS=1 trace

Raw File write linearly better - MPI-IO 1.5x faster at 4

Raw File read gets worse - Cray issue with cache reads being investigate by Cray

FPN Subfiling --- equivalent FPP performance but more manageable





Note the Log scaling on Y axis
Shared file is collective MPI-IO
File-Per-Node (FPN) and File-Per-Process (FPP) much faster but have to manage a lot of files

Theta Node-Local SSD Utilization

Description of methodology for SSD utilization and performance charts.

Node Local SSDs on Theta – NOT a Burst Buffer

- Local 128 GB SSD attached to each node
- Need to be granted access – PI contact support@alcf.anl.gov
  - https://www.alcf.anl.gov/user-guides/running-jobs-xc40#requesting-local-ssd-requirements
- Cray Datawarp requires burst buffer nodes
  - Flash storage attached to specialized nodes in the fabric
  - Allows for shared files to be striped across multiple burst buffer nodes as tiered storage in front of Lustre - eg Cori at NERSC
- No utility currently in place for tiered storage
  - Under investigation
- Requires explicit manual programming
  - Most useful to store local intermediate files (scratch)
  - Data deleted with cobalt job terminates

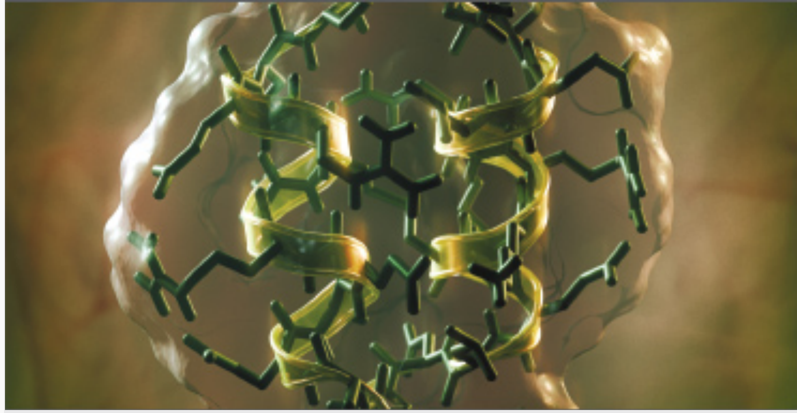
Node Local SSD Usage

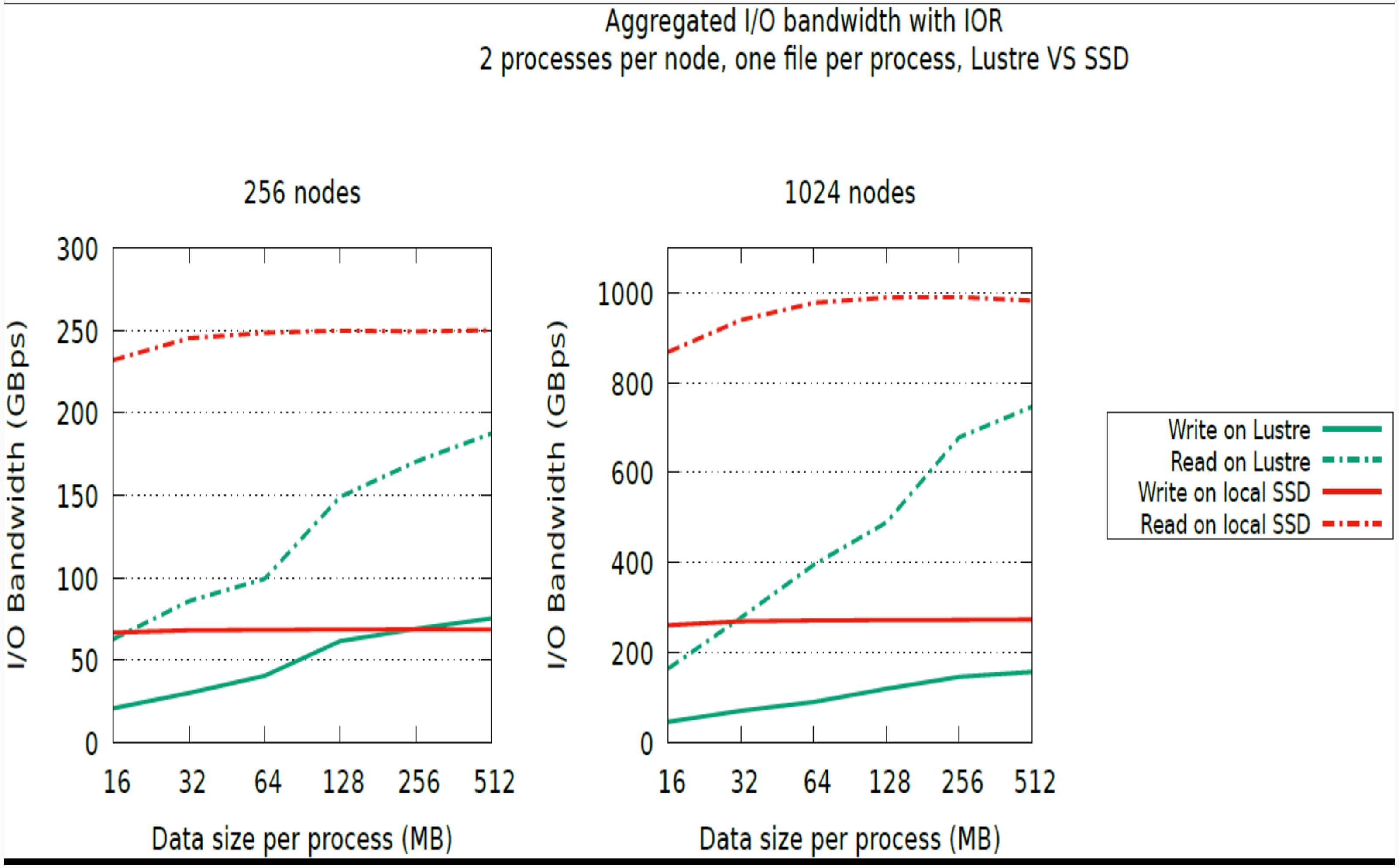
- To request SSD, add the following in your qsub command line:

```
--attrs ssds=required:ssd_size=128
```

  - This is in addition to any other attributes that you are setting for a job, including MCDRAM and NUMA modes. ssd_size is optional and may be omitted.
- The SSD are mounted on /local/scratch on each node
- SSD are emptied between allocations (job lifetime persistency)
- I/O Performance (One process): Read 1.1 GB/s – Write 175 MB/s
  - Can scale to two process: Read 2.2 GB/s, Write 350 MB/s
- Outperforms the Lustre file-system as scale (aggregated bandwidth)
- Node-limited scope, so may imply some work: sub-communicator per node, subfiling

Node SSD vs Lustre File-Per-Process Performance





Need 2 processes to drive SSD Max Bandwidth

Bandwidth number includes time to open, write/read, and close the file

After MDS overhead mitigated with larger data size SSD write is equal and read is close at 256 nodes

SSD performance scales past Lustre at 1024 nodes

CONCLUSION

Currently key to Io performance on theta is optimal Lustre File System access

- No tiered storage burst buffer implementation yet
- Understand and tune how your application is using lustre
  - striping
  - Cray MPI-IO
  - IO Libraries

ALCF Staff is available to help!