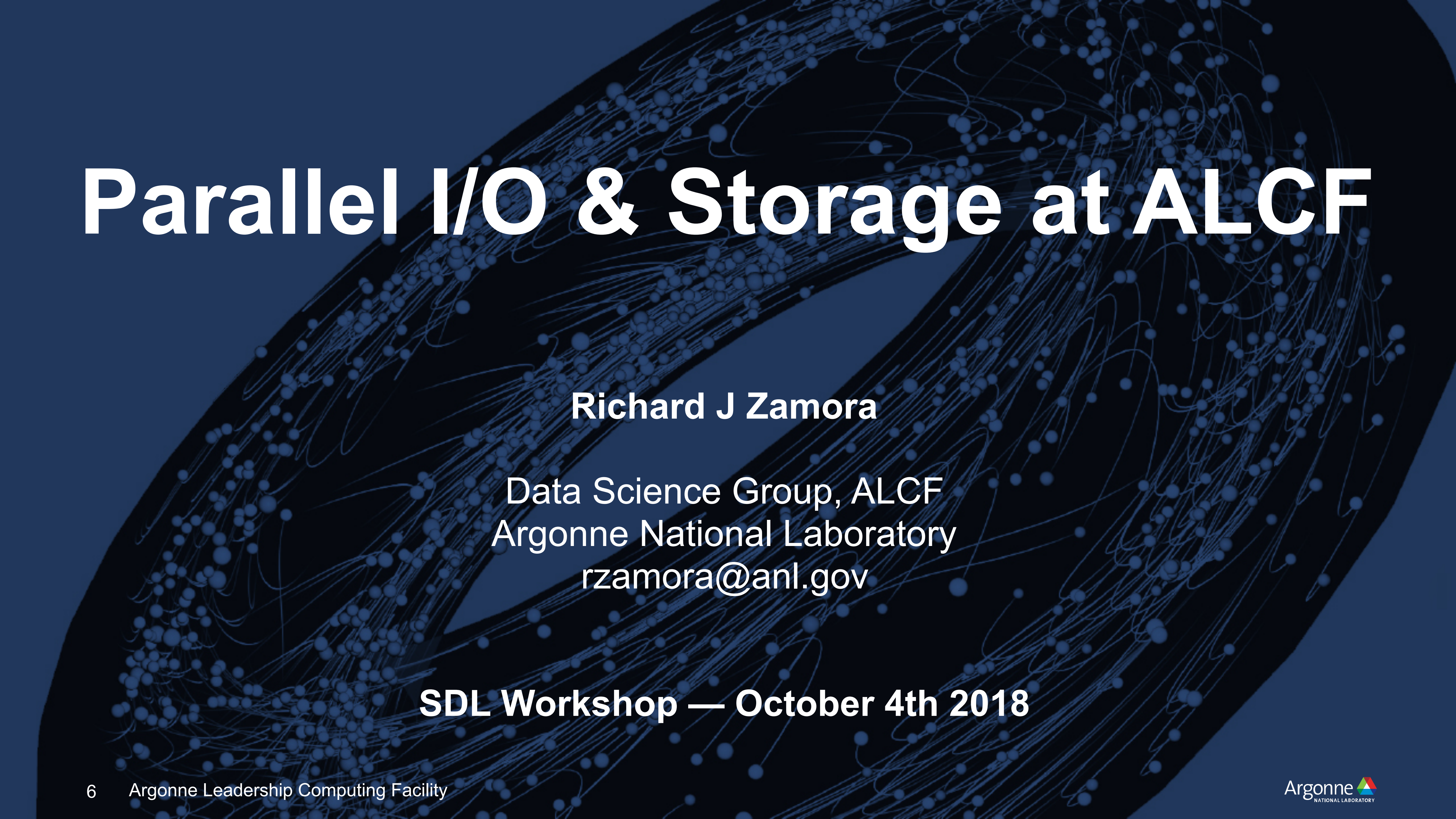


Parallel I/O & Storage at ALCF

Richard J Zamora

Data Science Group, ALCF
Argonne National Laboratory
rzamora@anl.gov

SDL Workshop — October 4th 2018

Acknowledgement

Much of the content from this talk is borrowed from similar talks by other people. A valuable source is public content from previous ALCF workshops, as well as The *Argonne Training Program on Extreme Scale Computing* (ATPESC). Others responsible for the content shared here:

- **Kevin Harms, *harms@anl.gov***
- **Venkatram Vishwanath, venkat@anl.gov**
- Paul Coffman
- Francois Tessier
- Preeti Malakar
- Rob Latham
- Rob Ross
- Phil Carnes

Preview

1. HPC I/O Basics
2. Parallel I/O Basics
3. ALCF Storage Overview
4. Optimizing I/O on **Mira** (IBM BG/Q - **GPFS** File System)
5. Optimizing I/O on **Theta** (Cray XC40 - **Lustre** File System)
6. Conclusions

HPC I/O Basics

In HPC, I/O usually corresponds to the storage and retrieval of persistent data to and from a file system (by a software application)

Key Points:

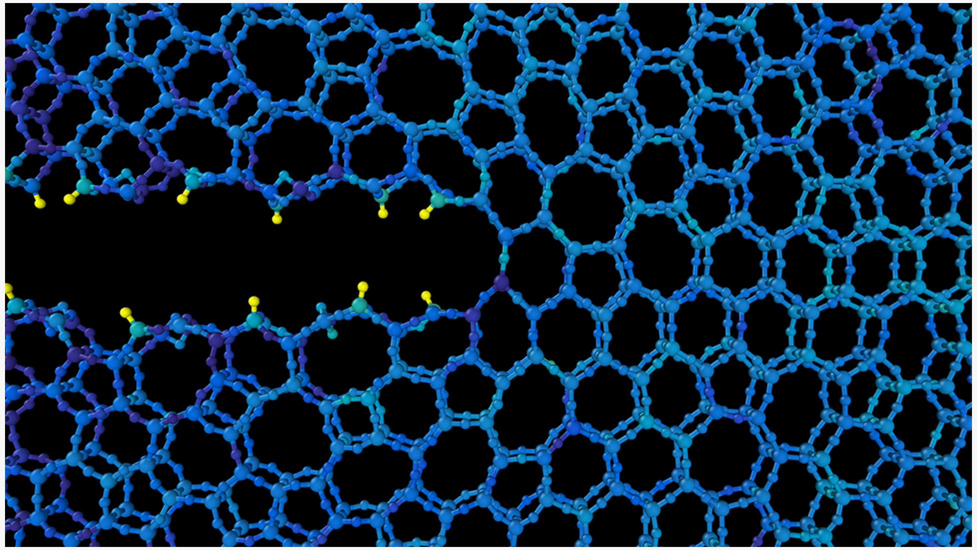
- Data is stored in **POSIX** files/directories
- HPC machines use parallel file systems (PFS) for I/O performance
- The PFS' provides **aggregate** bandwidth

Basic I/O Flavors

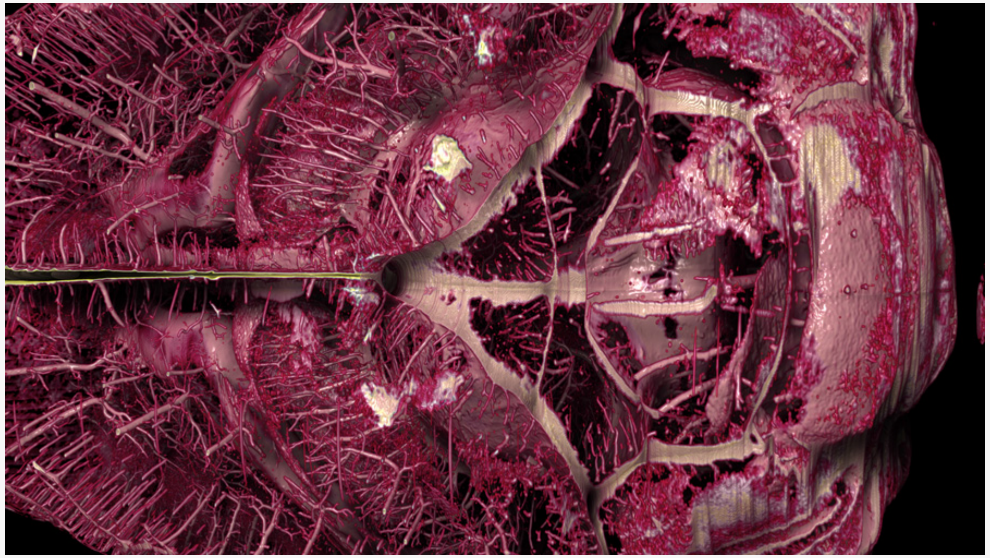
Basic flavors of I/O of concern at ALCF:

- **Defensive**
  - Writing data to protect results from software and/or system failures (a.k.a **checkpointing**)
  - Priority: Speed

- **Productive**
  - Reading and/or writing data as a necessary component of the application workflow
  - Priority: Speed, Organization, Provenance

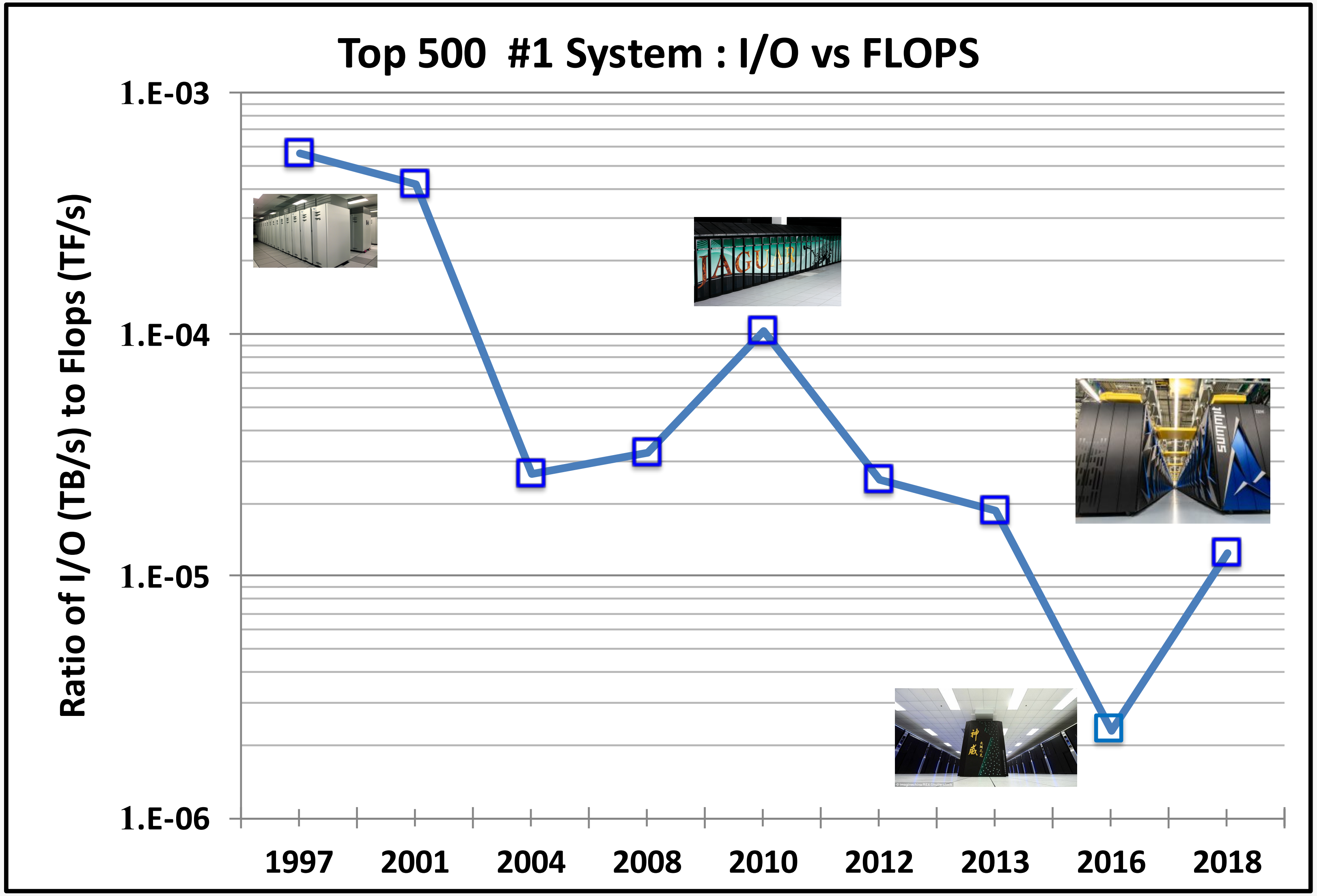


INCITE, 2016, James Kermode, University of Warwick



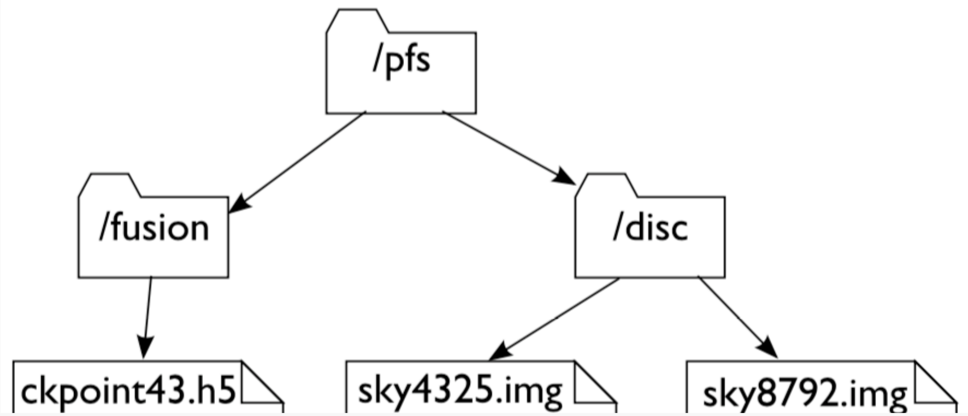
ADSP, 2017, Doga Gursoy, Argonne NAtional Laboratory

Historically: Compute has Outpaced I/O



How Data is Stored on HPC Systems (like ALCF)

- **HPC systems use the *file system* (FS) model for storing data**
  - Use a **file** (POSIX) model for data access
  - The FS is **shared** across the system
  - There can be >1 FS on each system, and FS’s can be shared between systems.
- **POSIX I/O API**
  - Lowest-level I/O API
  - **Well supported**: Fortran, C and C++ I/O calls are converted to POSIX
  - **Simple**: File is a *stream of bytes*



POSIX File: Contiguous *stream of bytes*

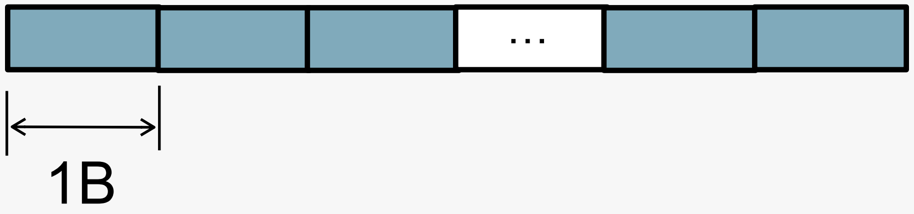
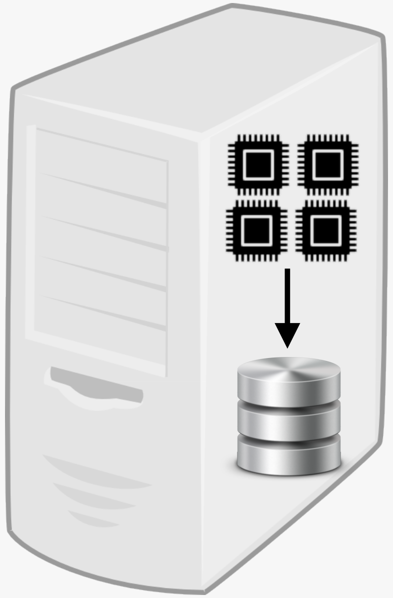


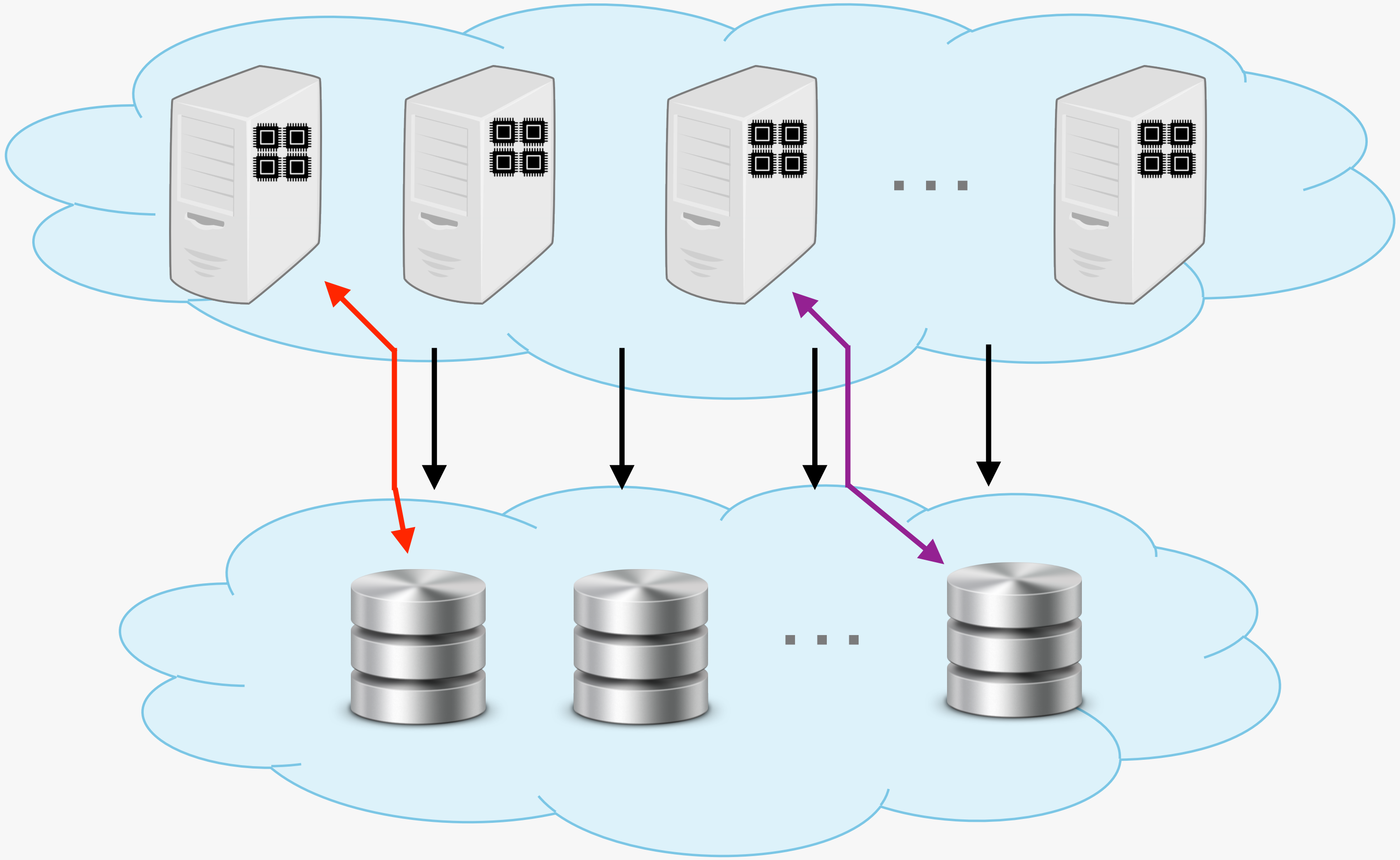
Image Source: Lantham et al. ATPESC 2014

Scalable I/O Requires a *Parallel* File System

Traditional: Serial I/O
(to *local* file system)



HPC: Parallel I/O (to *shared* file system)



Several Popular *Parallel* File Systems

IBM® GPFS

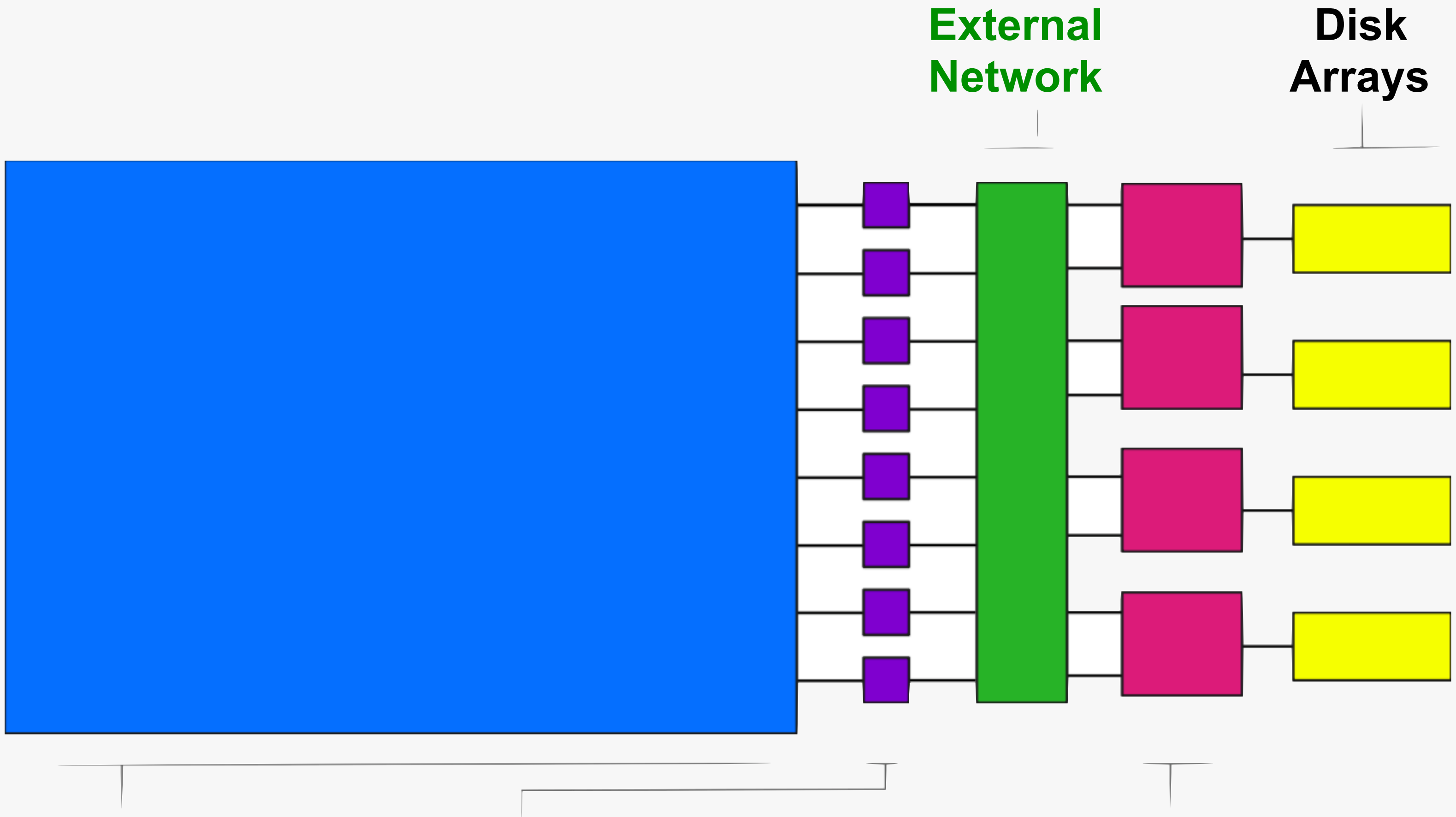
lustre™

panasas®

CRAY DataWarp™

BeeGFS®

Typical I/O Path

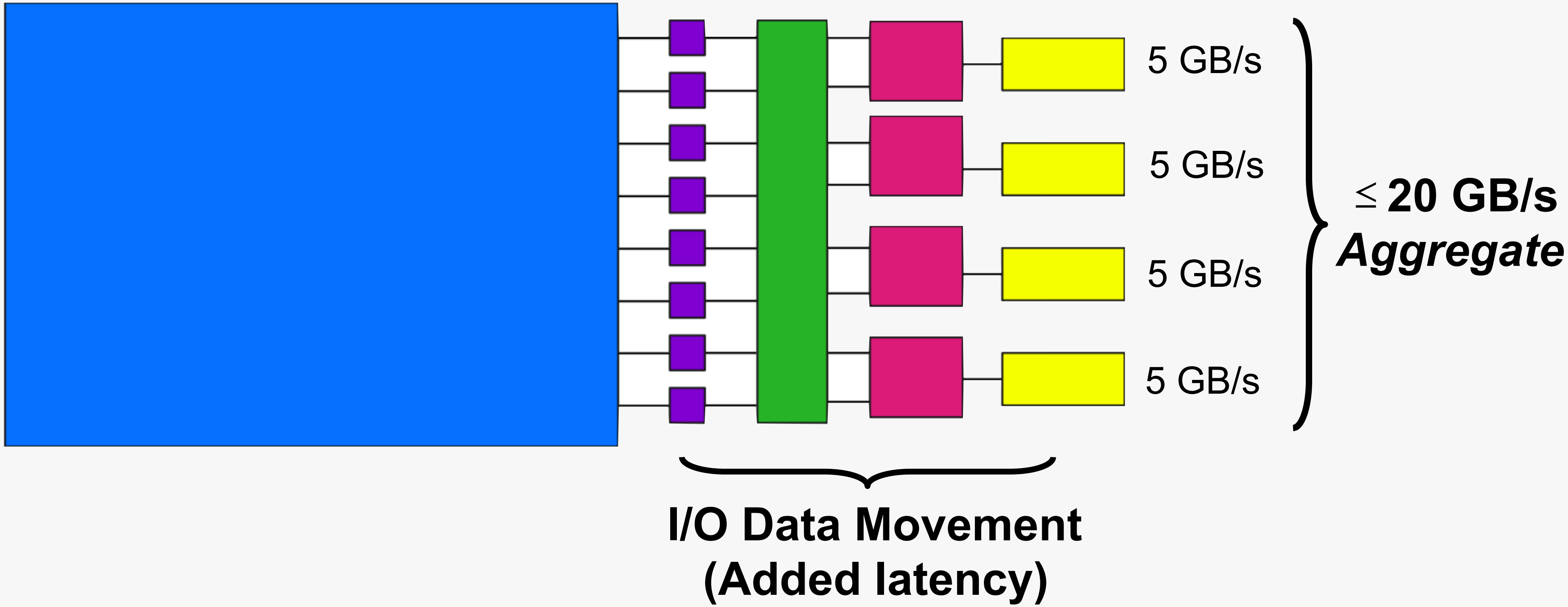


Compute nodes run user application. Data model software and some I/O transformations are also performed here.

I/O forwarding nodes (gateway nodes, LNET nodes,…) shuffle data between compute nodes and storage.

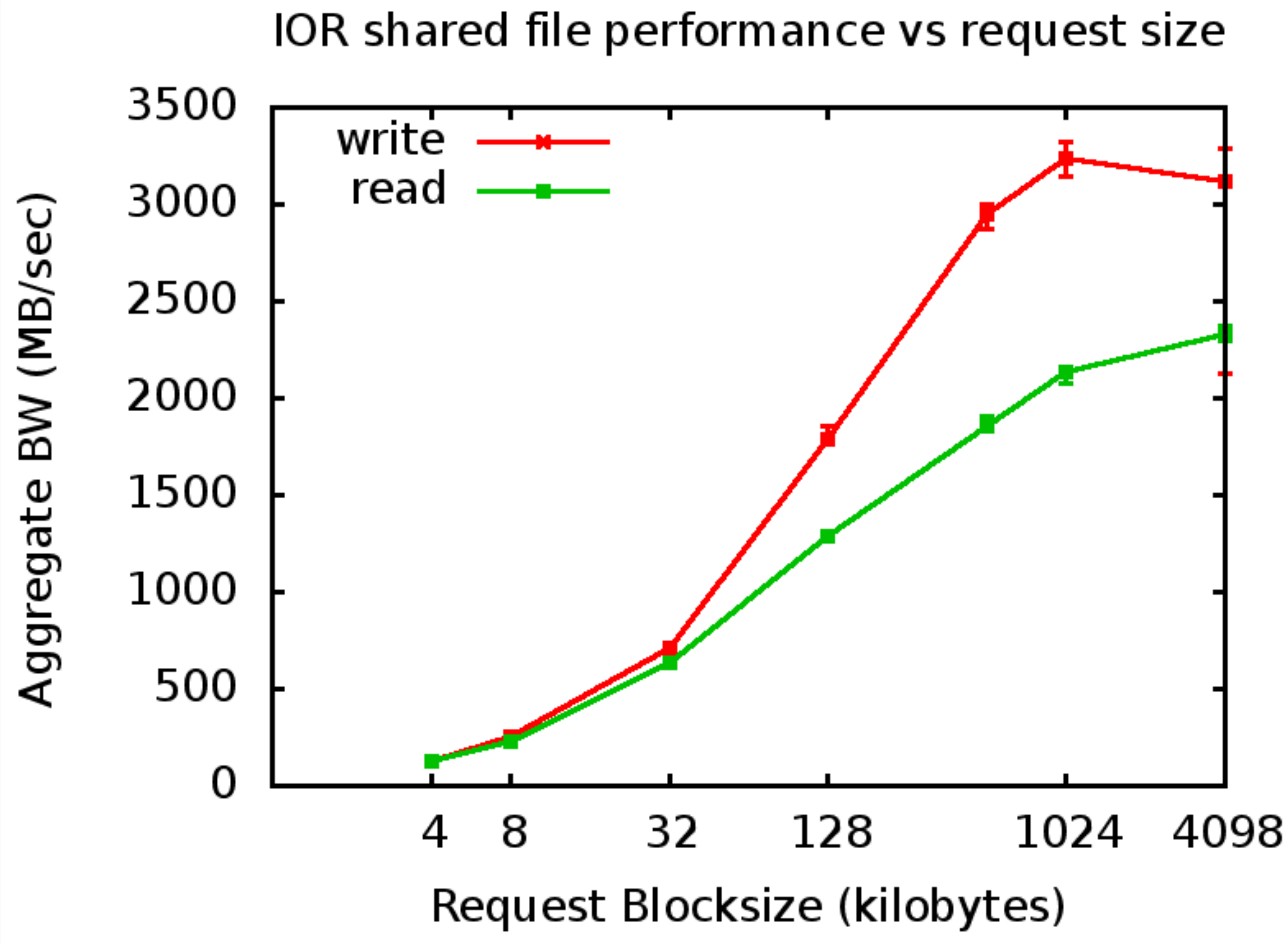
Storage nodes run the parallel file system.

Parallel I/O provides *Aggregate* Bandwidth



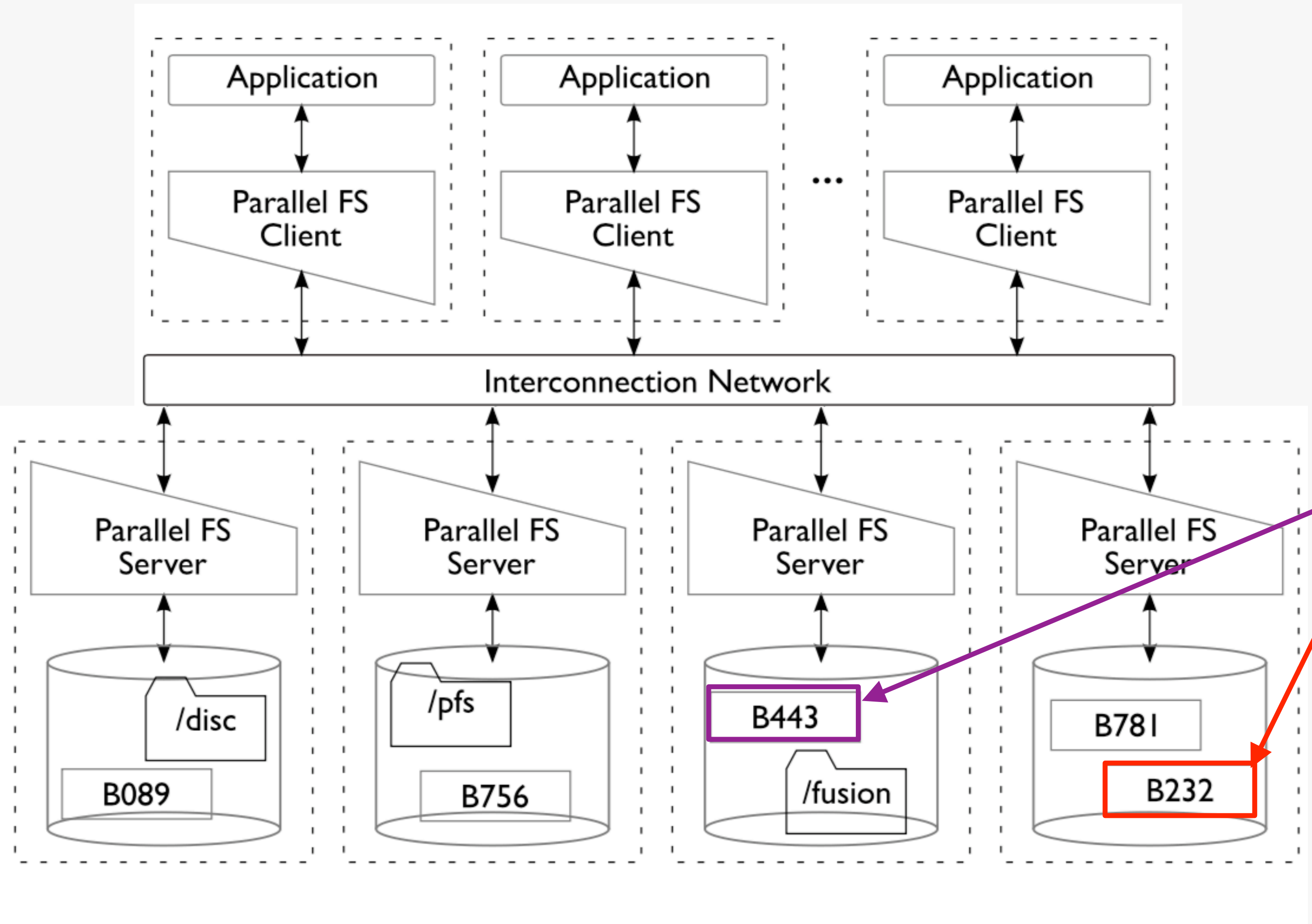
Data must take multiple hops to get to/from disk -> **high latency**
Multiple disks can be accessed concurrently -> **high aggregate bandwidth**

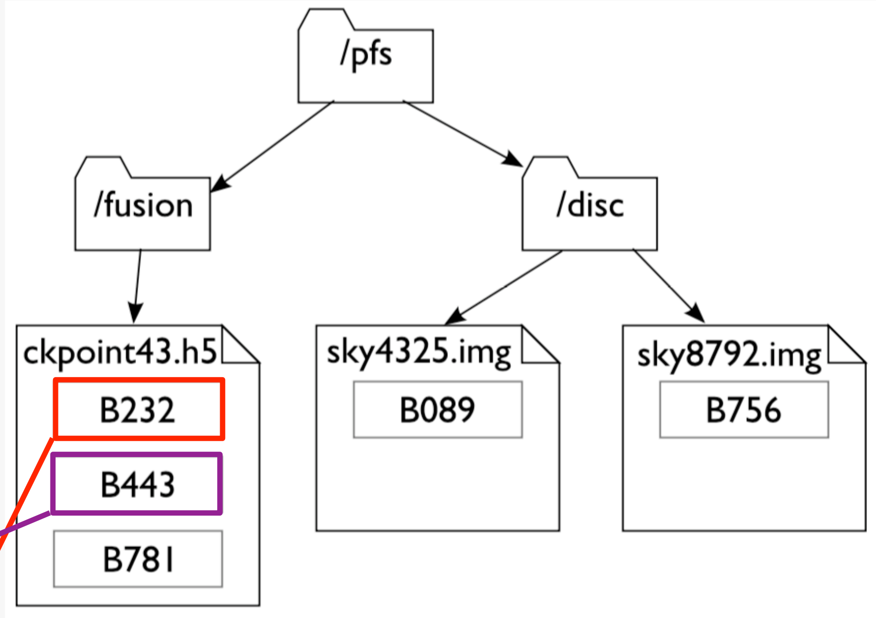
Example: Large Parallel Operations are *Faster*



2K processes of IBM Blue Gene/P at ANL

Mapping Files Onto Parallel Storage





In the PFS, files are broken up into *regions* called:

blocks in GPFS

stripes in Lustre

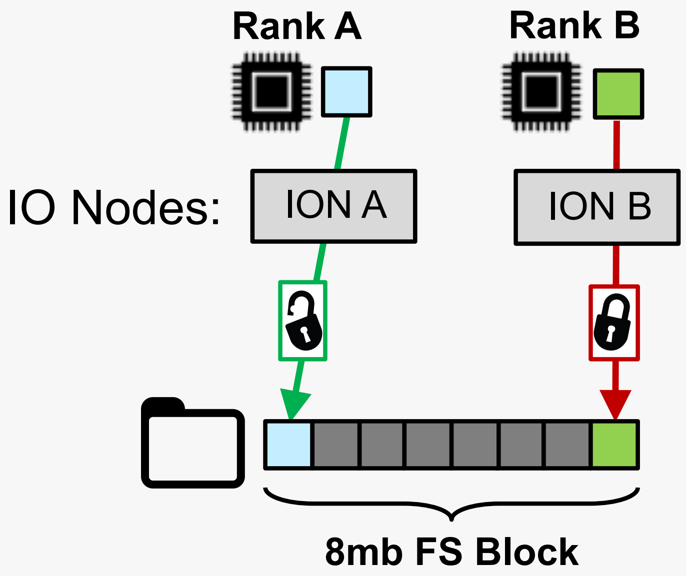
Image Source: Lantham et al. ATPESC 2014

Locking of File Blocks/Stripes

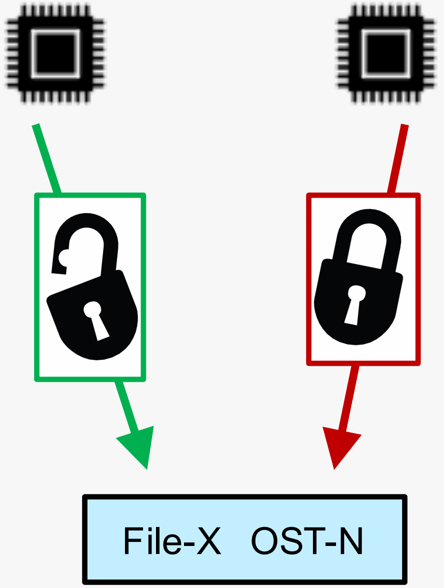
Relevant to GPFS & Lustre

– Block/Striped **aligned** I/O requests are important to avoid **lock contention** (false data sharing)

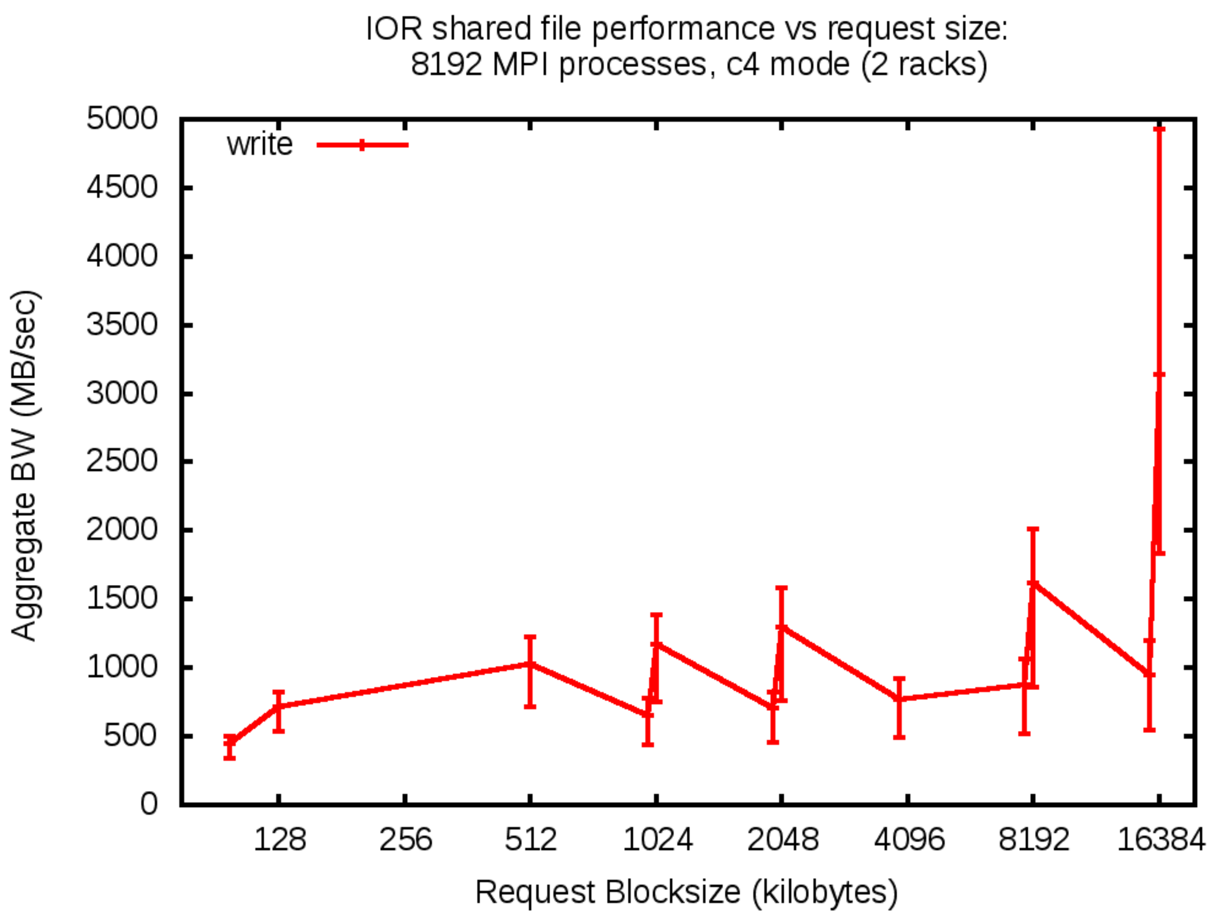
GPFS Block Locking



Lustre Stripe Locking



Example: Block-Aligned I/O is *Faster*



8K processes of IBM Blue Gene/Q at ANL

Parallel I/O Basics

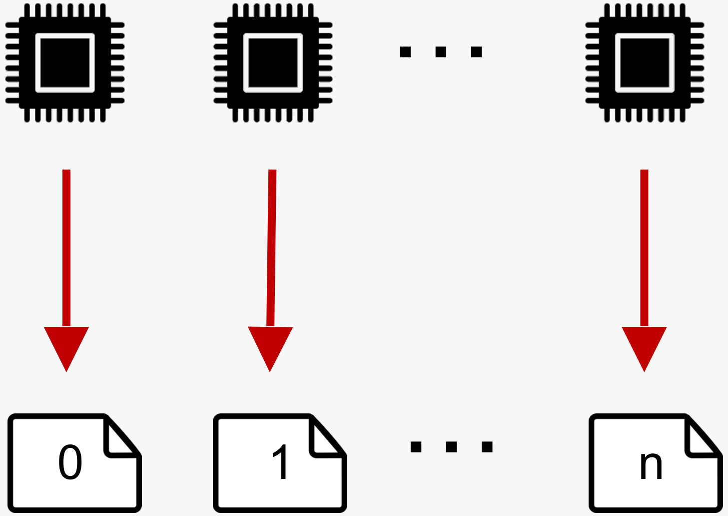
ALCF machines mostly rely on the parallel file system (PFS) for I/O performance. Parallel algorithms and optimizations are needed to efficiently move data between compute and storage hardware

Key Points:

- File-**per-process** I/O is not scalable
- **MPI-IO** (or a higher-level I/O library) is recommended
    - Libraries include optimizations for **collective I/O**

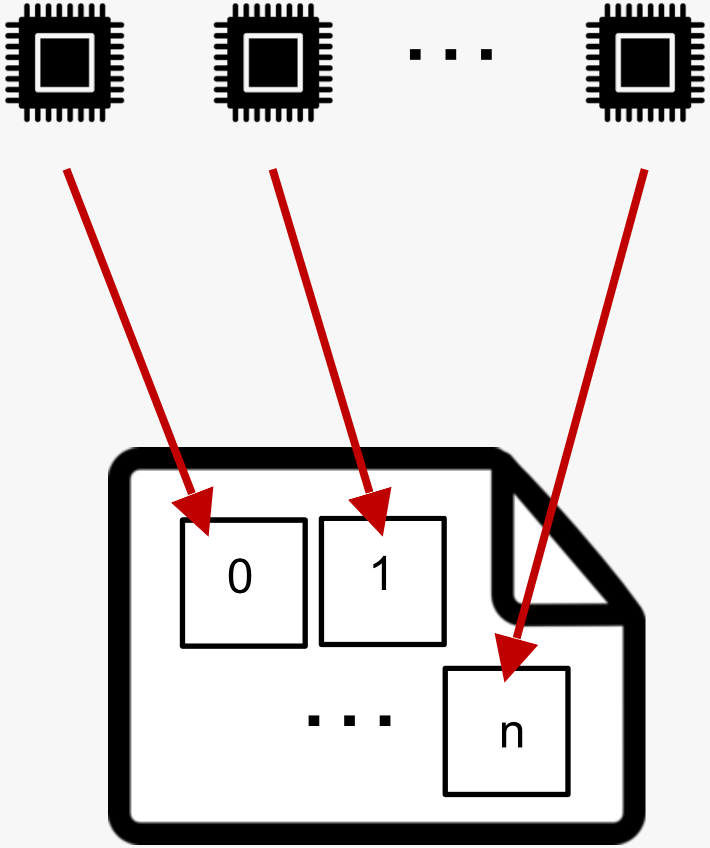
Types of Parallel I/O

File-per-process (FPP) Parallel

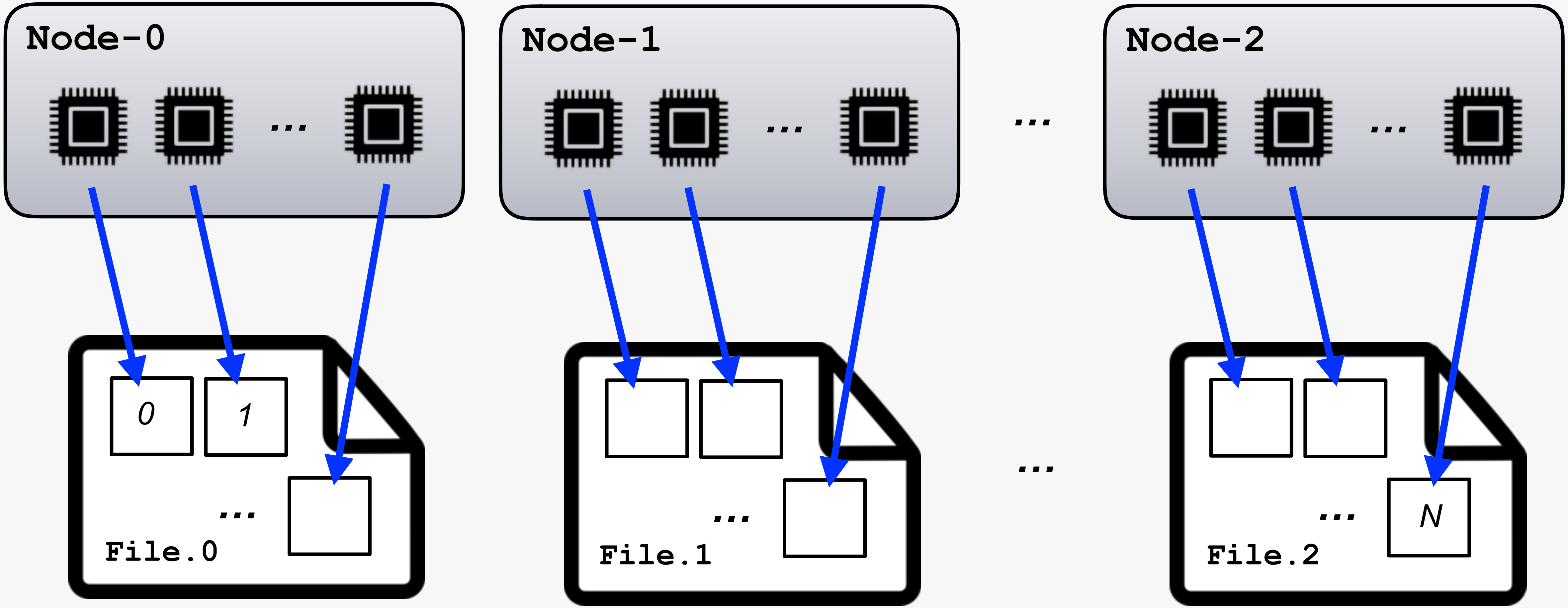


FPP can be fast for 10^1-10^3 ranks, but cannot scale to extreme scales (management and consumption issues)

Shared File Parallel



Mixing FPP with Shared Files: Sub-filing



At large scale, it can be optimal to use a shared file for each subset of processes (Ex. Per-node)

MPI-IO: MPI-Based POSIX-IO Replacement

- **POSIX has limitations**
  - Shared-file parallel I/O is possible, but complicated (parallel access, buffing, flushing, etc. must be explicitly managed)
- **Independent MPI-IO**
  - Each MPI task is handles the I/O independently using *non-collective* calls
    - Ex. `MPI_File_write()` and `MPI_File_read()`
  - Similar to POSIX I/O, but supports derived datatypes (useful for non-contiguous access)
- **Collective MPI-IO**
  - All MPI ranks (in a communicator) participate in I/O, and must call the same routines
    - Ex. `MPI_File_write`**`_all`**`()` and `MPI_File_read`**`_all`**`()`
  - Allows MPI library to perform collective I/O optimizations (often boosting performance)
- **MPI-IO (or a higher-level library leveraging MPI-IO) is recommended on Mira & Theta**
  - Python codes can use the `mpi4py` implementation of MPI-IO

*More code is needed for functionality

A Simple MPI-IO Example*

Simple MPI-IO code to concatenate local a 1-D arrays (on each rank) into a single global array in a file.

Note:* Grey *= Optional

```
// Create array to write (localbuf)
localbuf = (int *) malloc((N / size) * sizeof(int));
for(i=0; i<(N / size); i++) localbuf[i] = i;

// Determine file offset
offset = (N / size) * rank * sizeof(int);

// Let rank 0 Create the file
if(rank == 0){
  MPI_File_open( MPI_COMM_SELF, filename, MPI_MODE_CREATE|MPI_MODE_WRONLY, info, &fh );
  MPI_File_set_size( fh, filesize );
  MPI_File_close( &fh );
}

// Open the file for writing
MPI_File_open( MPI_COMM_WORLD, filename, MPI_MODE_WRONLY, info, &fh );
MPI_File_set_atomicity( fh, 0 );

// Write the file
MPI_File_write_at_all( fh, offset, localbuf, (N/size), MPI_INT, &status );

// Close the file
MPI_File_close( &fh );
```

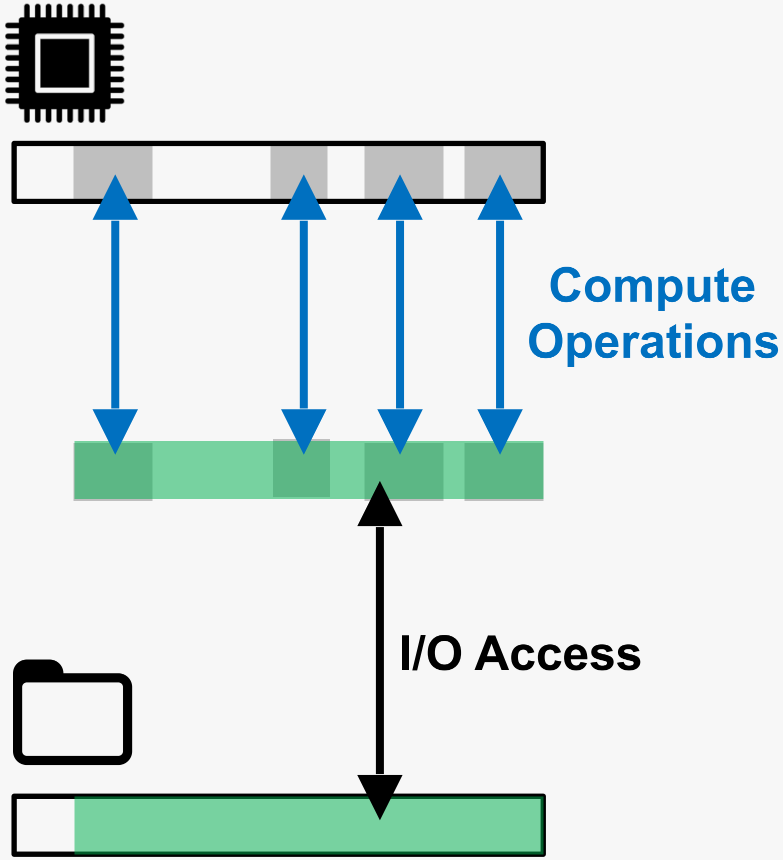
Creating and pre-allocating the file on a single rank can avoid metadata overhead

Specifying that "atomic" ordering is unnecessary

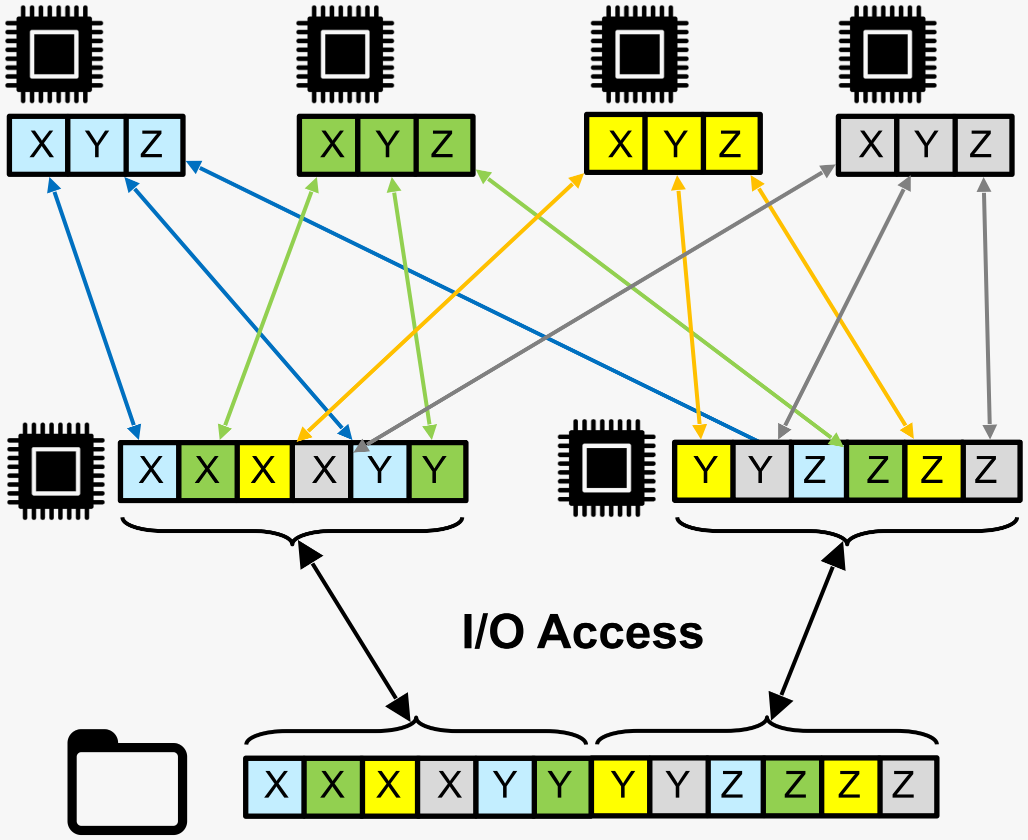
Collective write, starting at "offset"

Common Optimizations in MPI-IO

Data Sieving

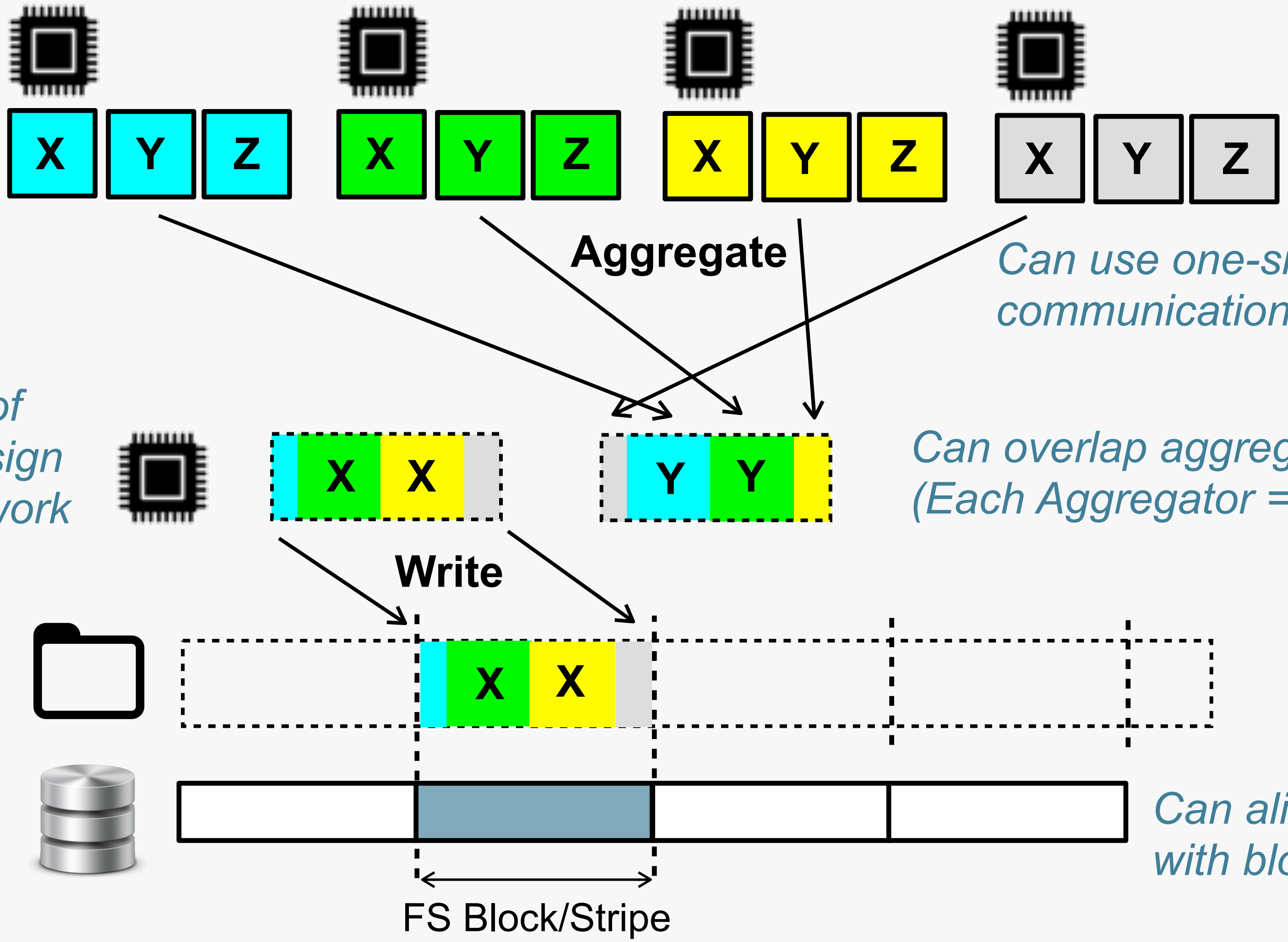


Two-Phase I/O
(Collective Aggregation)



More Detail on Two-Phase I/O

Typical collective I/O algorithm can be (internally) optimized in many ways…



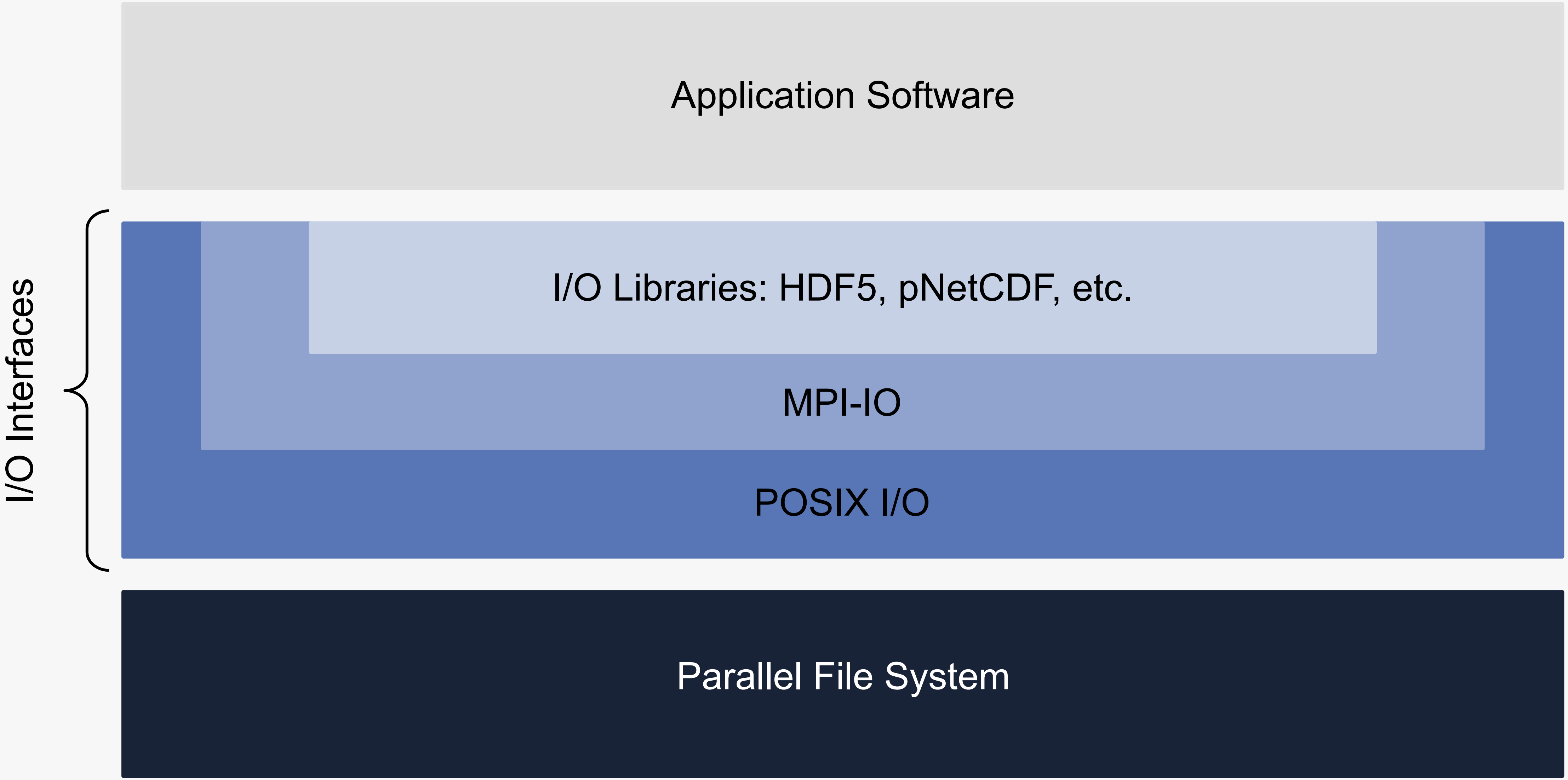
Can use one-sided (RMA) communication

Can select number of aggregators and assign ranks based on network topology

Can overlap aggregation and I/O (Each Aggregator = 2 buffers)

Can align buffers with blocks/stripes

ALCF I/O Software Stack



Application Software

I/O Interfaces

I/O Libraries: HDF5, pNetCDF, etc.

MPI-IO

POSIX I/O

Parallel File System

High-level I/O Libraries at ALCF

- High level I/O libraries provide an abstraction layer above MPI-IO/POSIX
  - Parallel **HDF5** and NetCDF leverage MPI-IO (both available on ALCF Systems)

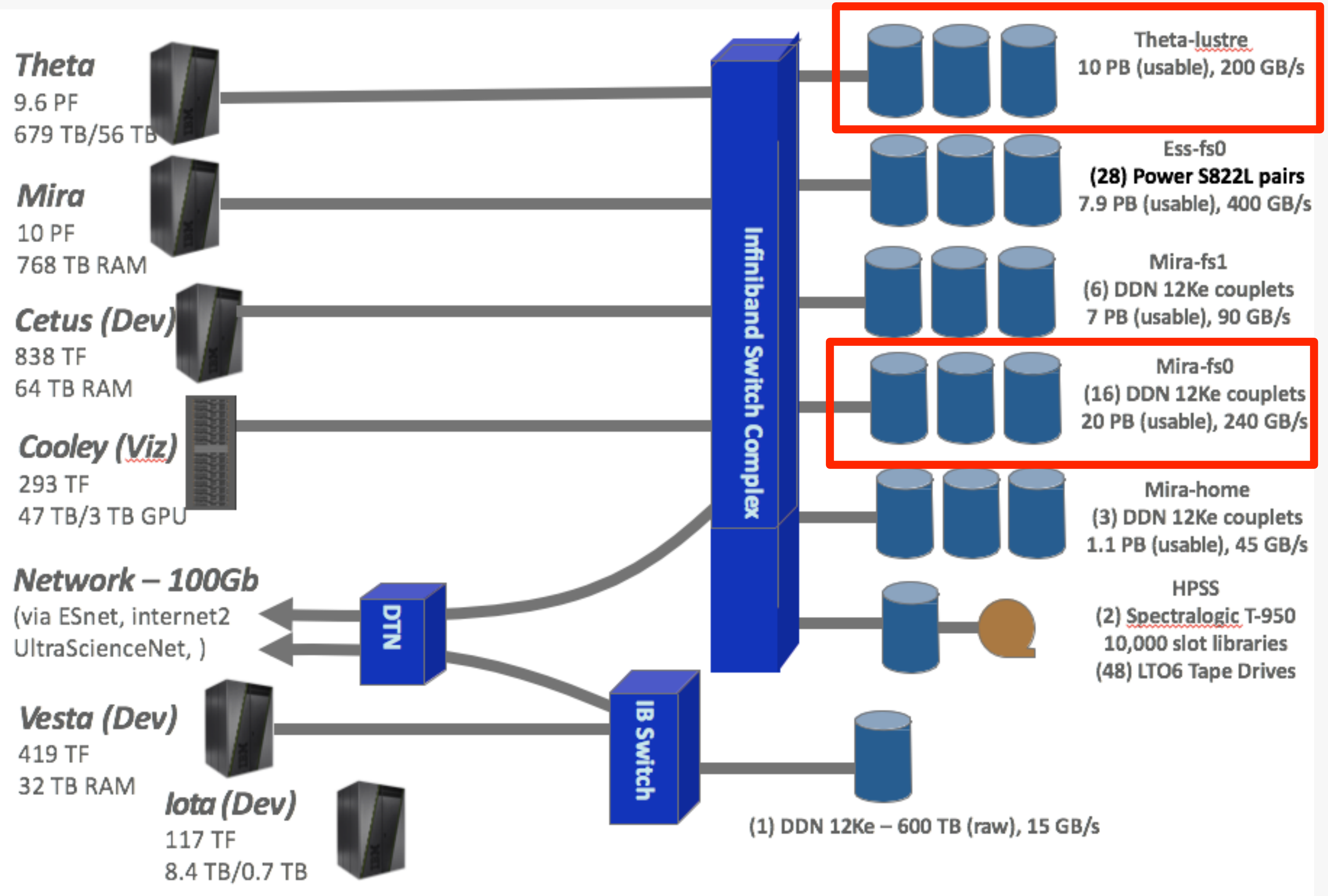
ALCF Storage Overview

ALCF machines offer both GPFS and Lustre file systems (with Lustre only supported on Theta)

Key Points:

- Use `/projects/<your-project>/` for high-performance I/O
- Theta production PFS: Lustre (`theta-fs0`)
- Mira production PFS: GPFS (`mira-fs0`)

ALCF Resources



ALCF Storage *Details*

Property	mira-fs0	theta-fs0	mira-fs1	mira-home
File system	GPFS	Lustre	GPFS	GPFS
Capacity	20 PB	10 PB	7 PB	1 PB
Performance	240 GB/s	210 GB/s	90 GB/s	45 GB/s
MD Perf	5000 creat/s	40000 c/s	5000 c/s	5000 c/s
Block Size	8 MiB	1 MiB	8 MiB	256 KiB
# targets	896	56	336	72
# drives	8960	4592 + 40	3360	720
# ssd	512	112	12	120

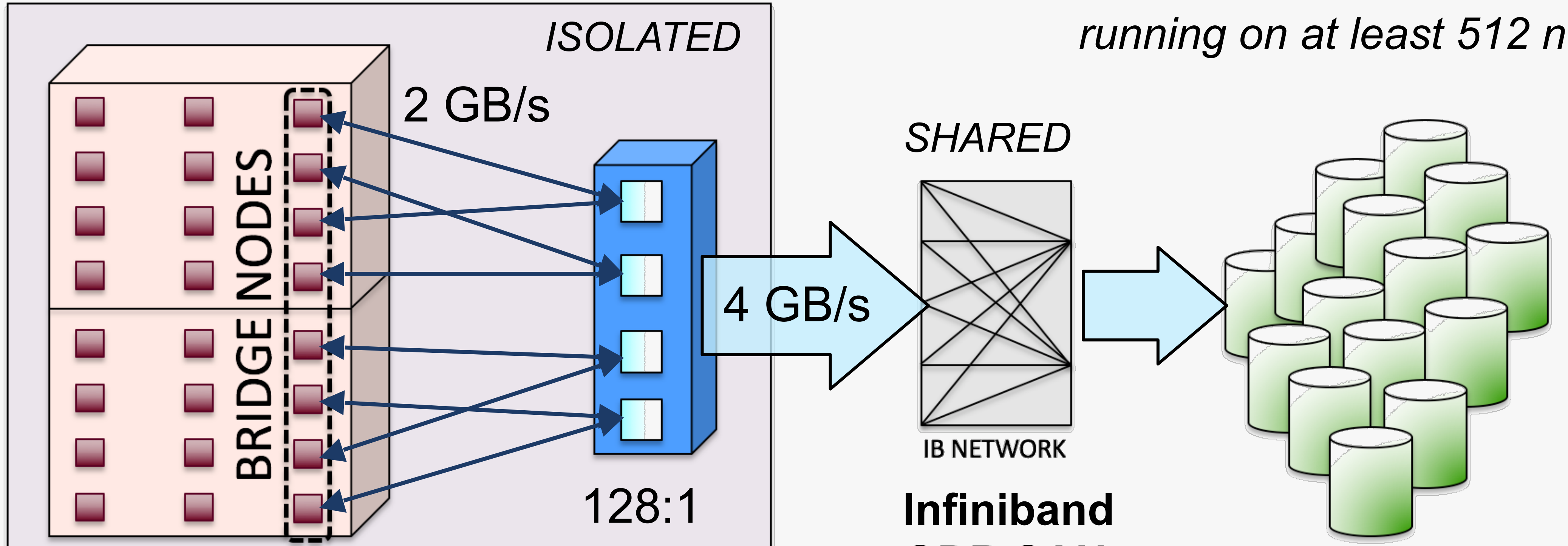
Important Considerations

- Use `/projects/<your-project>/` for high-performance IO
  - Automatically assigned to the appropriate parallel file system
  - Note: Files are **not** backed up in `/projects`
- Storage space is managed by project quotas
  - Files are not purged
  - Users must manage their own space

Optimizing I/O on Mira (Blue Gene/Q – GPFS)

Mira I/O Infrastructure: Overview

Note: I/O Nodes are dedicated resources when running on at least 512 nodes



Compute Node Rack
1024 compute nodes
16 bridge nodes

I/O Nodes (Gateway)
2 bridge nodes connect to each I/O node

Infiniband QDR SAN
Connected to Cooley & Cetus

GPFS File system
`/mira-fs0`
`/mira-fs1`
`/mira-home`

Note: 384 I/O Nodes

IBM's General Parallel File System (GPFS)

IBM's GPFS is used for all parallel file systems on Mira

- Uses client-side and server-side caching
- Metadata is replicated on all file systems
- Quotas are enabled
  - `myquota` (home)
  - `myprojectquotas` (project)
  - Overrun quota error: `-EQUOTA`

Name	Type	Blocksize	Capacity	Speed
`mira-fs0`	project	8 MB	19 PB	240 GB/s
`mira-fs1`	project	8 MB	7 PB	90 GB/s
`mira-home`	home	256 K	1 PB	-

MPI-IO on Mira

Mira has great support for MPI-IO

- Leveraged by other major I/O libraries
  - Look in `/soft/libraries`
  - HDF5, NetCDF, pNetCDF, Adios, etc.
- Uses BG/Q-specific Optimizations
  - Handles alignment on block boundaries
  - Leverages Mira 5D Torus network

Important Note

MPI-IO scales well, but may run out of memory at full-machine scales

Usually related to MPI all-to-all calls and discontinuous data types (Workarounds discussed soon)

Important MPI-IO Recommendations

- Use collective routines (eg. `MPI_File_write_at`**`_all`**`()`)
- Disable locking within the Blue Gene ADIO layer for non-overlapping writes using the following environment variable:
  - `--env BGLOCKLESSMPIO_F_TYPE=0x47504653`

MPI-IO BG/Q Driver Tuning

Advanced Options:

- Environment variable `BGMPIO_NAGG_PSET=16` (default 8)
- Hint: `cb_buffer_size=16m` (change the collective aggregation buffer size)
- Hint: `romio_no_indep_rw` can improve collective file open/close performance
  - Only does file open on aggregator ranks during `MPI_File_open`, for independent I/O (eg `MPI_File_write_at`) non-aggregator nodes file open at write time (deferred)

BGQ driver variables for memory-issue workarounds (often hurts performance):

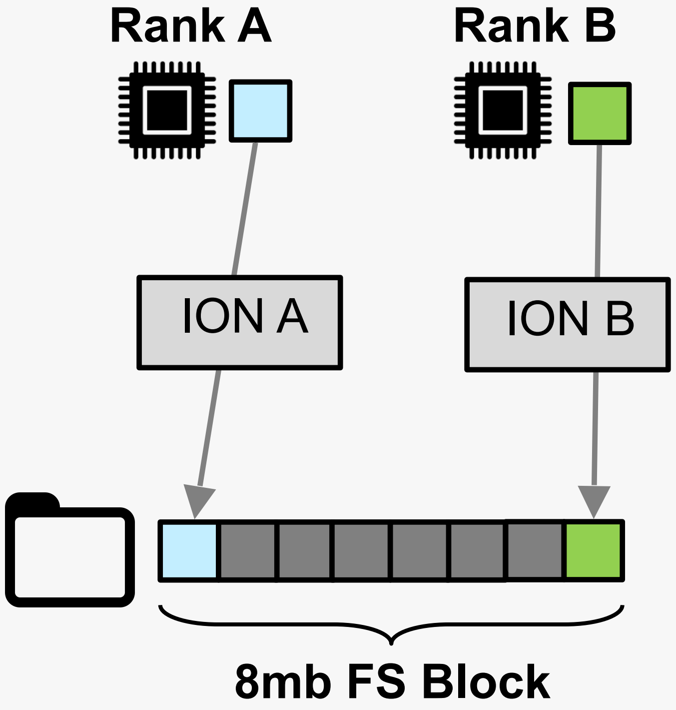
- No MPI_Alltoall(v) calls: `--envs BGMPIO_COMM=1`
- Tune routing protocol and avoid heap fragmentation:

```
--envs PAMID_SHORT=0
--envs PAMID_DISABLE_INTERNAL_EAGER_TASK_LIMIT=1
```

GPFS Block Alignment

Use block-aligned I/O when using shared files

- The GPFS project file systems are all 8 MB
  - Unaligned access will be punished by GPFS locking
  - Larger, block-aligned accesses will perform best
- Collective MPI-IO should take care of this for you



Example:

- *MPI rank A and B happen to use two different I/O nodes*
- *Rank A writes the first MB of an 8 MB block*
  - *Rank A must acquire the lock for this fs block*
- *Rank B writes the last MB of an 8 MB block*
  - *Rank B tries to acquire the block for this block but must wait because it is in use*
- *Parallel I/O becomes serial for this workload*

Performance Tools on Mira

Darshan (https://www.alcf.anl.gov/user-guides/darshan)

- Stores I/O profiling summary in single compressed log file
  - Look in: `/gpfs/mira-fs0/logs/darshan/mira/<year>/<month>/<day>`

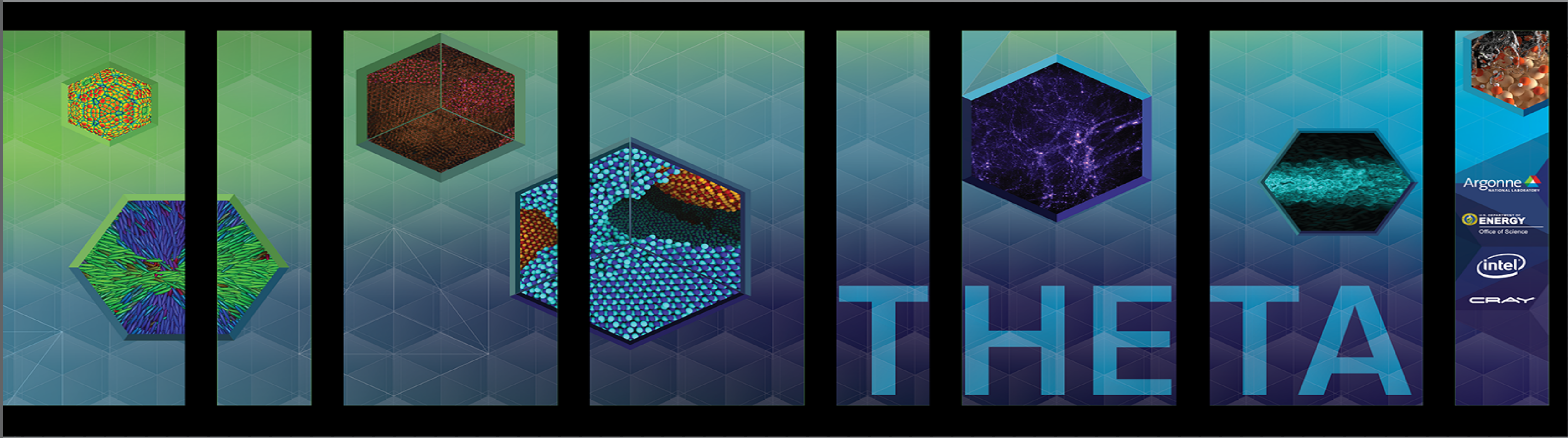
TAU (https://www.alcf.anl.gov/user-guides/tuning-and-analysis-utilities-tau)

- “`-optTrackIO`” in `TAU_OPTIONS`

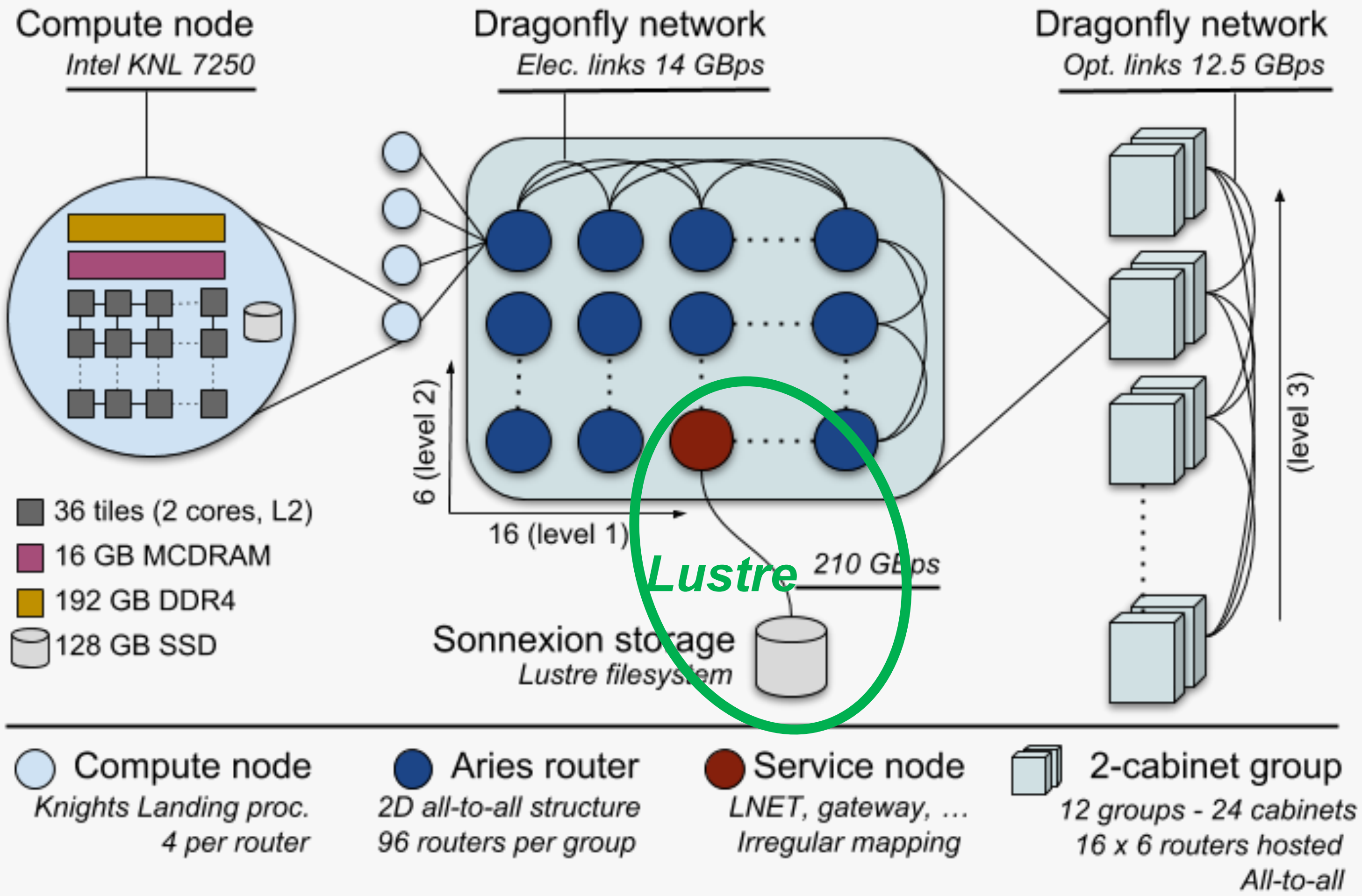
mpitrace (http://www.alcf.anl.gov/user-guides/hpctw)

- List performance of `MPI_File*` calls
  - Show performance of underlying MPI-IO for IO libraries such as HDF5

Optimizing I/O on Theta (Cray XC40 – Lustre)



Theta system Overview



Note: 30 LNET Nodes

Architecture: Cray XC40
Processor: 1.3 GHz Intel Xeon Phi 7230 SKU
Peak performance of 11.69 petaflops
Racks: 24
Nodes: 4,392
Total cores: 281,088
Cores/node: 64
Memory/node: 192 GB DDR4 SDRAM
(Total DDR4: 843 TB)
High bandwidth memory/node: 16 GB MCDRAM
(Total MCDRAM: 70 TB)

10 PB Lustre file system
SSD/node: 128 GB
(Total SSD: 562 TB)
Aries interconnect - Dragonfly configuration

Lustre File System Basics

Clients = LNET Router Nodes
MDS = Metadata Server
MDT = Metadata Target
OSS = Object Storage Server
OST = Object Storage Target

Each file is distributed over 1+ OSTs, depending on the size and striping settings for the specific file.

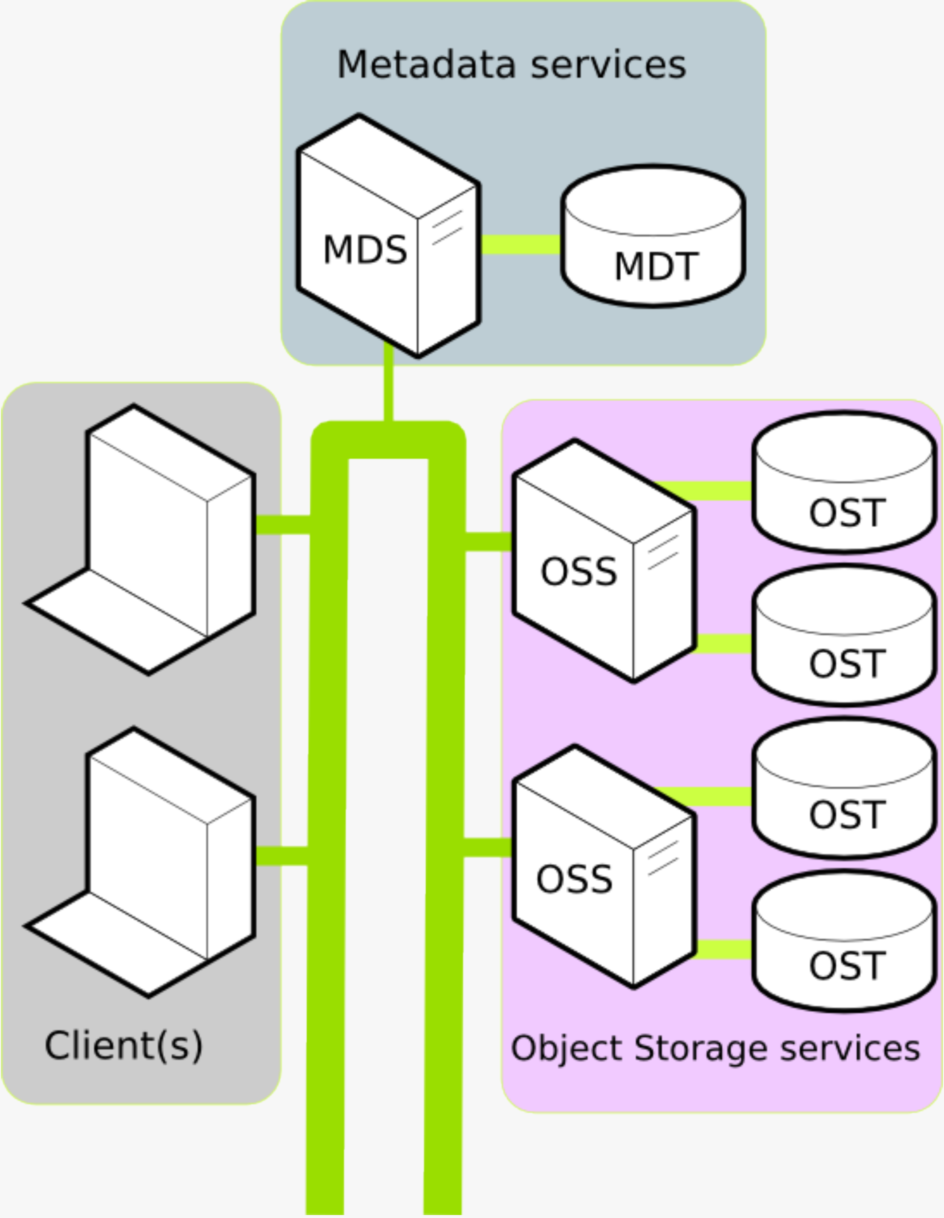


Image: https://wiki.hpdd.intel.com/display/PUB/Components+of+a+Lustre+filesystem

Theta – Lustre Specification

Current Version: lfs 2.7.2.26

Hardware: 4 Sonexion Storage Cabinets

- 10 PB usable RAID storage
- 56 OSS (1 OST per OSS)

Note: OSS cache currently disabled by hardware (Sonexion)

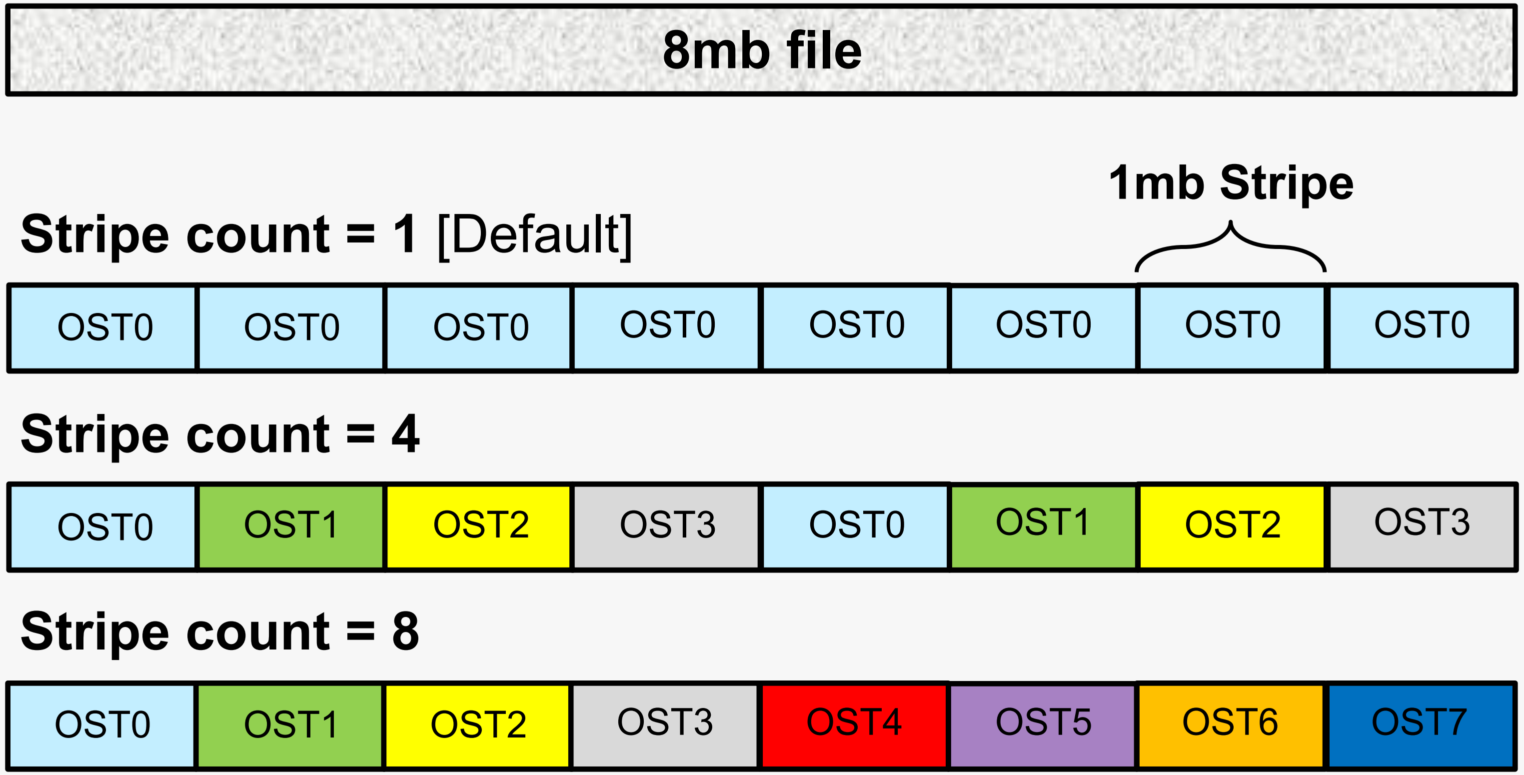
Performance:

- Total Write BW **172 GB/s**, Total Read BW **240 GB/s**
- <u>Peak Performance of 1 OST is 6 GB/s</u>
  - **Lustre client-cache effects can allow much higher BW**

Lustre File Striping Basics

Key to Parallel Performance

Example: Consider a single **8mb file** with **1mb stripe size…**



8mb file

1mb Stripe

Stripe count = 1 [Default]

OST0	OST0	OST0	OST0	OST0	OST0	OST0	OST0

Stripe count = 4

OST0	OST1	OST2	OST3	OST0	OST1	OST2	OST3

Stripe count = 8

OST0	OST1	OST2	OST3	OST4	OST5	OST6	OST7

Basic Idea
Files are *striped* across OSTs using a predefined striping pattern (pattern = count & size)

Stripe count
The number of OSTs (storage devices) used to store/access the file
`[Default = 1]`

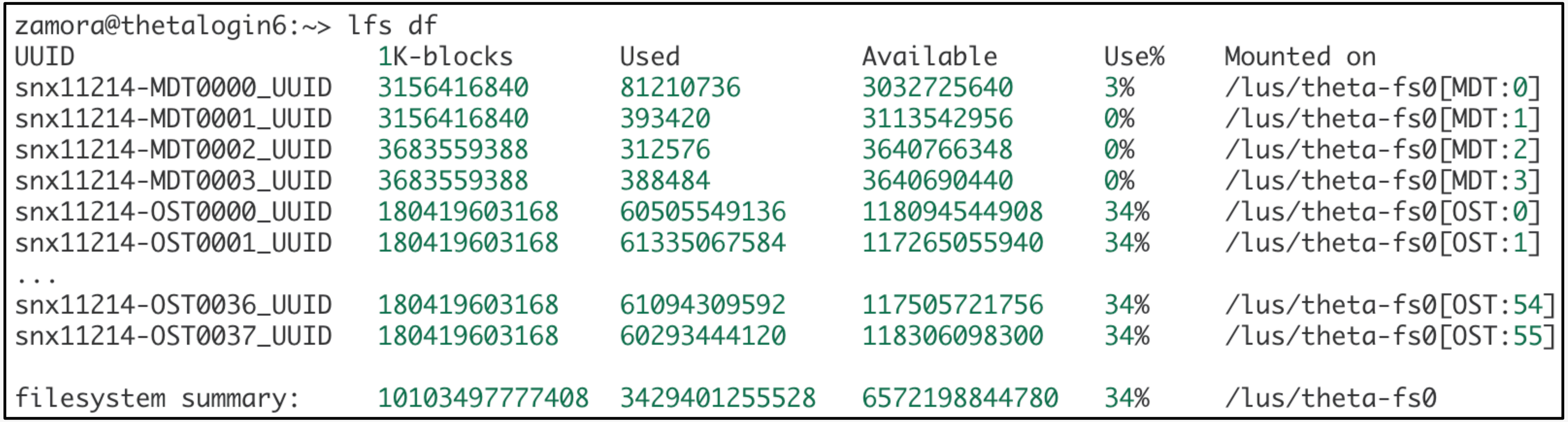
Stripe size
The width of each contiguous OST access
`[Default = 1m]`

Note: `1m = 1048576`

Lustre File System Utility: `lfs`

Manual: http://doc.lustre.org/lustre_manual.pdf

- List available `lfs` arguments: `lfs help`
- List name and status of the various OSTs: `lfs osts <path>`
- Set/Get striping information: `lfs getstripe <path>`
- Set/Get striping information: `lfs setstripe <args> <path>`
- Check disk space usage: `lfs df`



```
zamora@thetalogin6:~> lfs df
UUID                   1K-blocks       Used           Available       Use%     Mounted on
snx11214-MDT0000_UUID  3156416840      81210736       3032725640      3%       /lus/theta-fs0[MDT:0]
snx11214-MDT0001_UUID  3156416840      393420         3113542956      0%       /lus/theta-fs0[MDT:1]
snx11214-MDT0002_UUID  3683559388      312576         3640766348      0%       /lus/theta-fs0[MDT:2]
snx11214-MDT0003_UUID  3683559388      388484         3640690440      0%       /lus/theta-fs0[MDT:3]
snx11214-OST0000_UUID  180419603168    60505549136    118094544908    34%      /lus/theta-fs0[OST:0]
snx11214-OST0001_UUID  180419603168    61335067584    117265055940    34%      /lus/theta-fs0[OST:1]
...
snx11214-OST0036_UUID  180419603168    61094309592    117505721756    34%      /lus/theta-fs0[OST:54]
snx11214-OST0037_UUID  180419603168    60293444120    118306098300    34%      /lus/theta-fs0[OST:55]

filesystem summary:    10103497777408  3429401255528  6572198844780   34%      /lus/theta-fs0
```

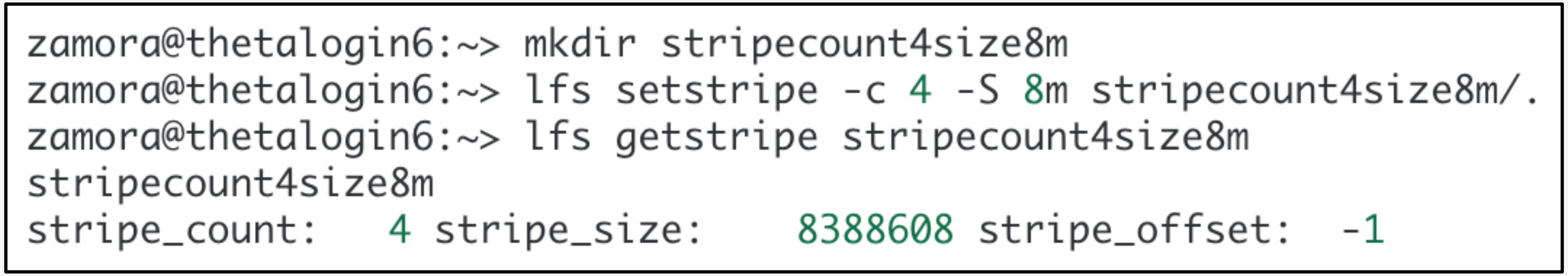
Example: `lfs setstripe` (IMPORTANT)

The stripe settings are critical to performance

- Defaults are **not** optimal for large files

Command syntax:

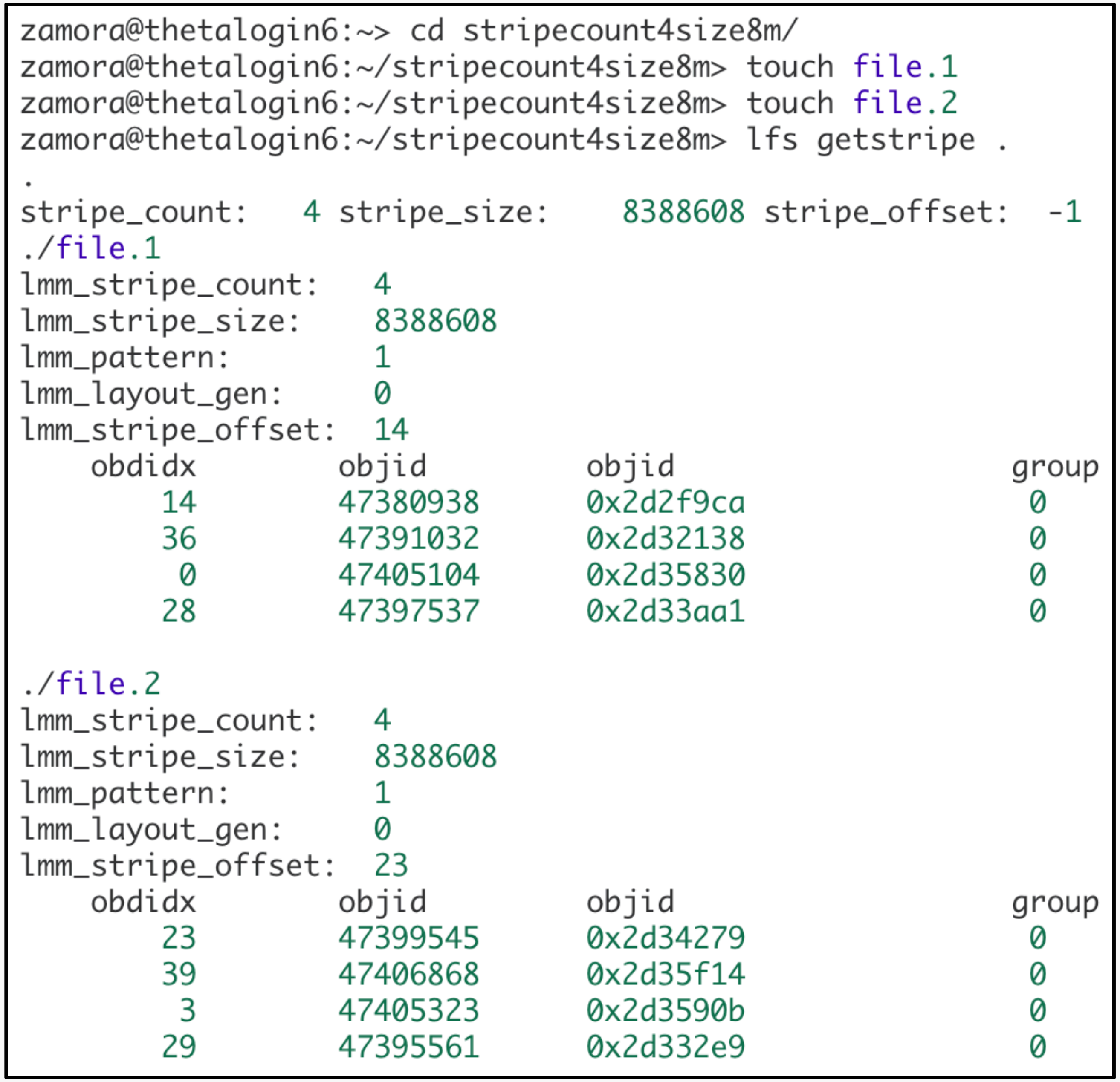
```
lfs setstripe --stripe-size <size> --count <count> <file/dir name>
lfs setstripe –S <size> -c <count> <file/dir name>
```



```
zamora@thetalogin6:~> mkdir stripecount4size8m
zamora@thetalogin6:~> lfs setstripe -c 4 -S 8m stripecount4size8m/.
zamora@thetalogin6:~> lfs getstripe stripecount4size8m
stripecount4size8m
stripe_count:   4 stripe_size:    8388608 stripe_offset:  -1
```

Example:

`lfs getstripe`



```
zamora@thetalogin6:~> cd stripecount4size8m/
zamora@thetalogin6:~/stripecount4size8m> touch file.1
zamora@thetalogin6:~/stripecount4size8m> touch file.2
zamora@thetalogin6:~/stripecount4size8m> lfs getstripe .
.
stripe_count:   4 stripe_size:    8388608 stripe_offset:  -1
./file.1
lmm_stripe_count:   4
lmm_stripe_size:    8388608
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  14
    obdidx        objid         objid                   group
        14        47380938      0x2d2f9ca                0
        36        47391032      0x2d32138                0
         0        47405104      0x2d35830                0
        28        47397537      0x2d33aa1                0

./file.2
lmm_stripe_count:   4
lmm_stripe_size:    8388608
lmm_pattern:        1
lmm_layout_gen:     0
lmm_stripe_offset:  23
    obdidx        objid         objid                   group
        23        47399545      0x2d34279                0
        39        47406868      0x2d35f14                0
         3        47405323      0x2d3590b                0
        29        47395561      0x2d332e9                0
```

Important Notes about File Striping

- Make sure to use the `/project` file system (NOT `/home`)
  - `/project` is Lustre, `/home` is NOT
- Don't set the `stripe_offset` yourself (let Lustre choose *which* OSTs to use)
- Default Striping is `stripe_count=1` and `stripe_size=1048576`
- Files and directories inherit striping patterns from the parent directory
- Stripe count cannot exceed number of OSTs (56)
- Striping cannot be changed once file created
  - Need to re-create file – copy to directory with new striping pattern to change it

Non-`lfs` Options:

- Can set stripe settings in **Cray MPI-IO** (striping_unit=*size*, striping_factor=*count*)
  - Ex: `MPICH_MPIIO_HINTS=*:striping_unit=<SIZE>:striping_factor=<COUNT>`
- 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

General Luster Striping Guidelines

Large Shared Files:

- More than 1 stripe (default) usually best
  - Keep stripe count below the node count
  - ~8-48 usually good (not 56 - let Lustre avoid slow OSTs)
- Larger than a 1mb stripe (default) usually best
  - ~8-32 usually good
  - Note: large stripe sizes can require memory-hungry collective I/O

File-per-process: Use 1 stripe

Small files: Use 1 stripe

Cray MPI-IO

Cray MPI-IO Overview

Cray MPI-IO is recommended on Theta

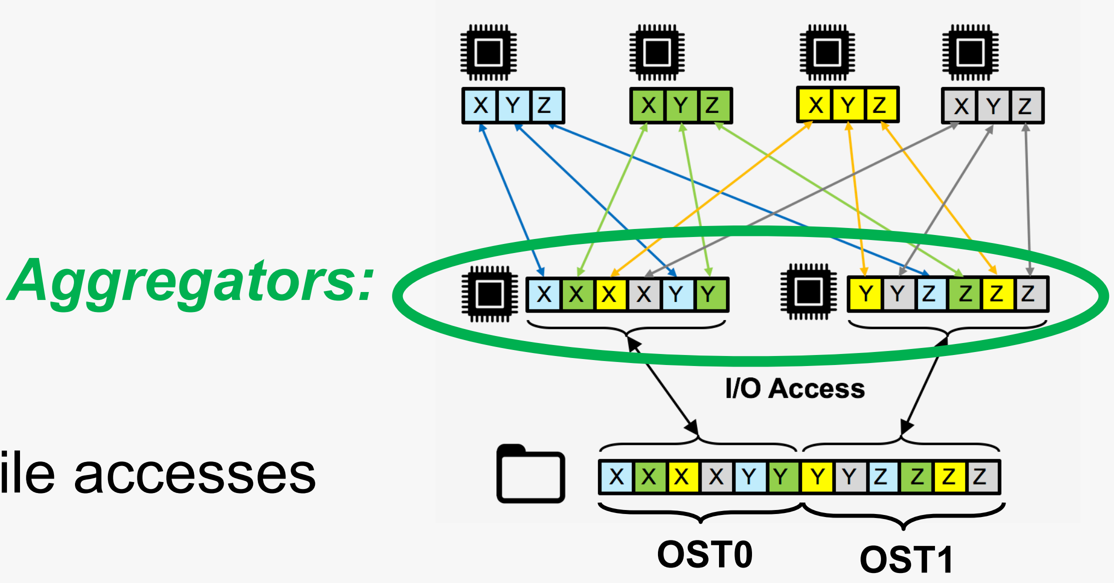
- Used by Cray-MPICH (default MPI environment on Theta - `cray-mpich` module)
- Based on MPICH-MPIIO (ROMIO)
- Optimized for Cray XC40 & Lustre
- Many tuning parameters: `man intro_mpi`

Underlying I/O layer for other I/O libraries

- HDF5 (`module load cray-hdf5-parallel`)
- PNetCDF (`module load cray-netcdf-hdf5parallel`)
- Python mpi4py (Ex: `module load miniconda-3.6/conda-4.5.4`)
- Python h5py (Ex: `module load miniconda-2.7/conda-4.4.10-h5py-parallel`)

CRAY
THE SUPERCOMPUTER COMPANY

Tuning Cray-MPI-IO: Collective Buffering



Collectives (two-phase I/O): MPI_File_*_all calls

– Aggregate data into large/contiguous stripe-size file accesses

Tuning aggregation settings:

- Number of aggregator nodes (`cb_nodes` hint) defaults to the striping factor (*count*)
  - `cray_cb_nodes_multiplier` hint will multiply the number of aggregators
  - Total aggregators = `cb_nodes` x `cray_cb_nodes_multiplier`
  - Collective buffer size defaults to the stripe size
    - `cb_buffer_size` hint (in ROMIO) is ignored by Cray
  - ROMIO's collective buffer is allocated (according to this setting), but not used

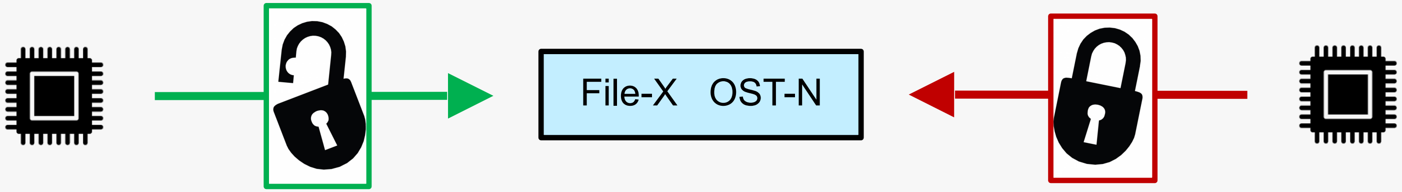
Note: To use open-source MPICH MPI-IO (ROMIO), use `cb_align=3`

Or env: `MPICH_MPIIO_CB_ALIGN=3`

Tuning Cray-MPI-IO: Extent-lock Contention

Each rank (client) needs its own lock to access a given file on an OST

- When 2+ ranks access same file-OST combination: extent lock contention



Mitigation: `cray_cb_write_lock_mode=1` (shared lock locking mode)

- A single lock is shared by all MPI ranks that are writing the file.
- Lock-ahead locking mode (`cray_cb_write_lock_mode=2`) not yet supported by Sonexion
  - **All file accesses MUST be collective**
    - `romio_no_indep_rw` must be set to true
    - HDF5, PNetCDF, and Darshan wont work (rely on some independent access)

Example:

```
MPICH_MPIIO_HINTS=*:cray_cb_write_lock_mode=1:cray_cb_nodes_multiplier
=<N>:romio_no_indep_rw=true
```

I/O Profiling Tools on Theta

Cray-MPI: Environment Variables for Profiling

- `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 variable
- `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_CB_ALIGN=3`
  - Turn off Cray’s MPI-IO Lustre File driver (not recommended for production)

Darshan I/O Profiling

Open-source statistical I/O profiling tool (https://www.alcf.anl.gov/user-guides/darshan)

- No source modifications, lightweight and low overhead
- Finite memory allocation (about 2MB) - Overhead of 1-2% total

USE:

- Make sure postscript-to-pdf converter is loaded: `module load texlive`
- `darshan` module should be loaded by default
- I/O characterization file placed here at job completion:

```
/lus/theta-fs0/logs/darshan/theta/<YEAR>/<MONTH>/<DAY>
```

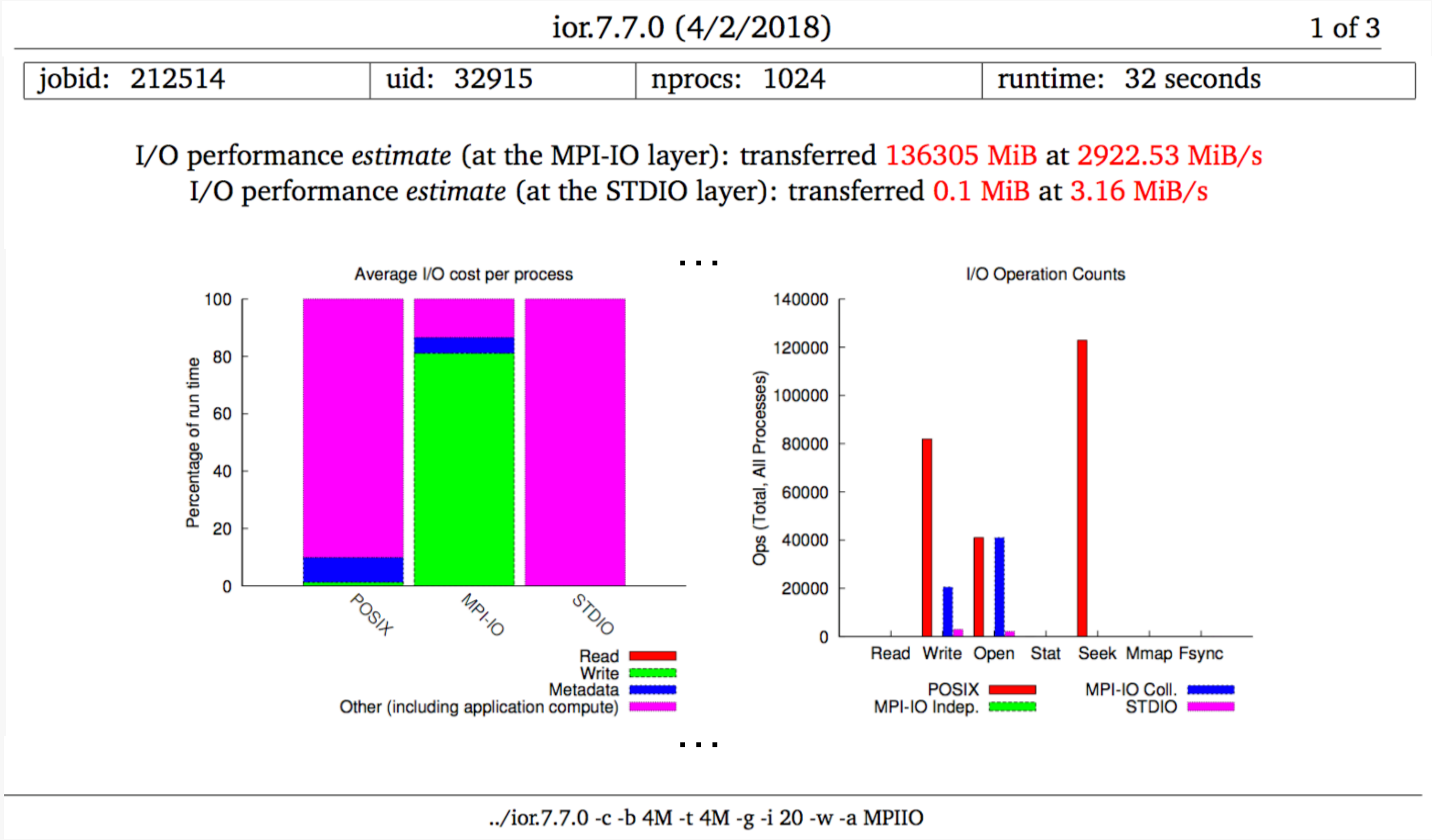
- Use `darshan-job-summary.pl` command for charts, table summaries:

```
darshan-job-summary.pl <darshan_file_name> --output darshansummaryfilename.pdf
```

- Use `darshan-parser` for detailed text file:

```
darshan-parser <darshan_file_name>  > darshan-details-filename.txt
```

Darshan Output Example



ior.7.7.0 (4/2/2018) 1 of 3

jobid: 212514	uid: 32915	nprocs: 1024	runtime: 32 seconds

I/O performance *estimate* (at the MPI-IO layer): transferred 136305 MiB at 2922.53 MiB/s
I/O performance *estimate* (at the STDIO layer): transferred 0.1 MiB at 3.16 MiB/s

…

…

../ior.7.7.0 -c -b 4M -t 4M -g -i 20 -w -a MPIIO

Lustre Performance on Theta

Dragonfly Network and Lustre Jitter

Communication is over shared networks (No job isolation)

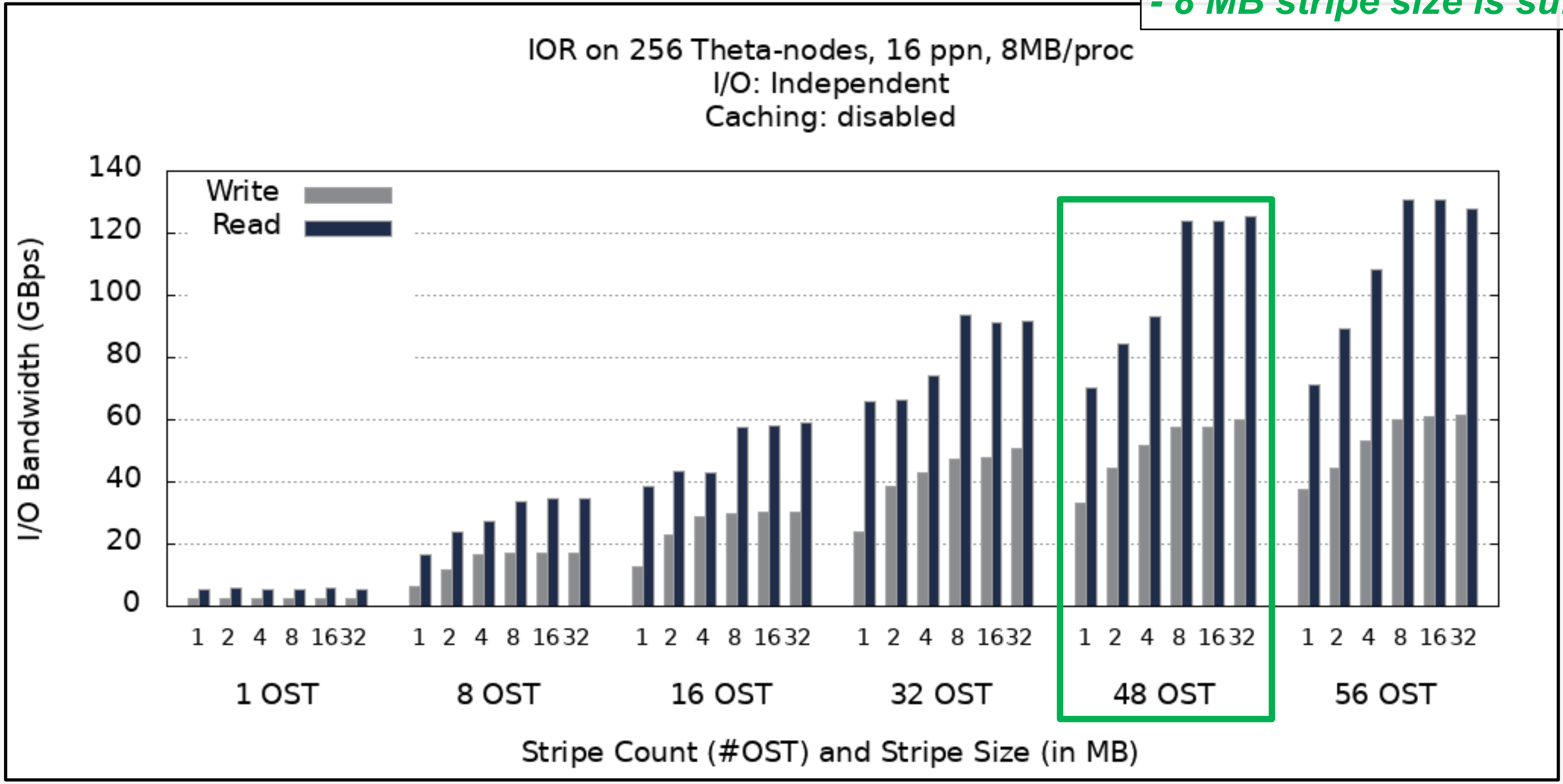
- Currently 1 Metadata Sever (MDS) shared by all users
- MDS and/or OSS traffic surge can dramatically effect performance

When running IO performance tests, want either:

- run-time statistics (max, min, mean, median, etc.)
- Best of multiple trials (typically used here)
- Dedicated system

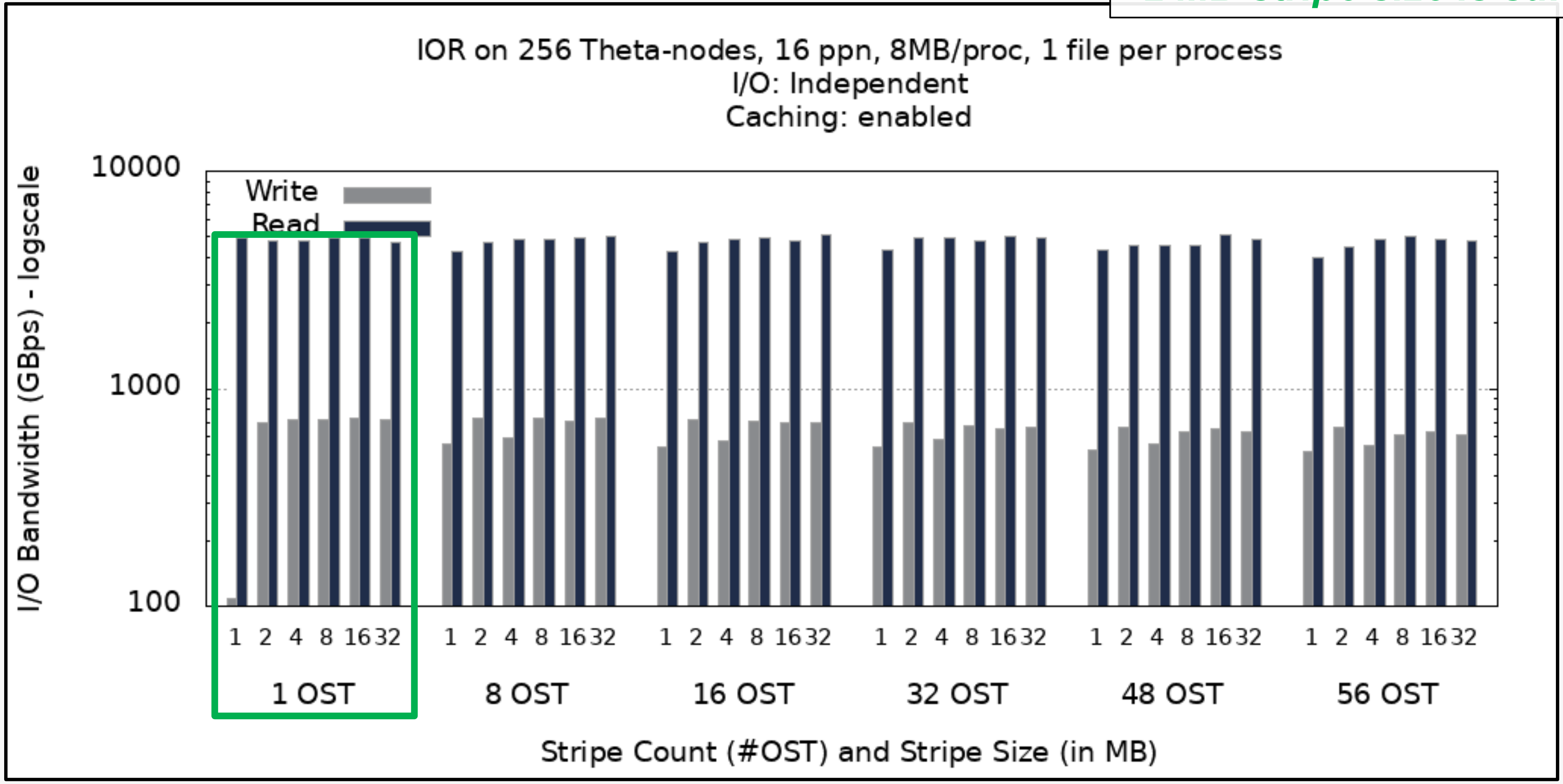
Shared File – 8MB/proc – Independent I/O
Client-side Caching DISABLED

- *More OSTs is better*
- *8 MB stripe size is sufficient*



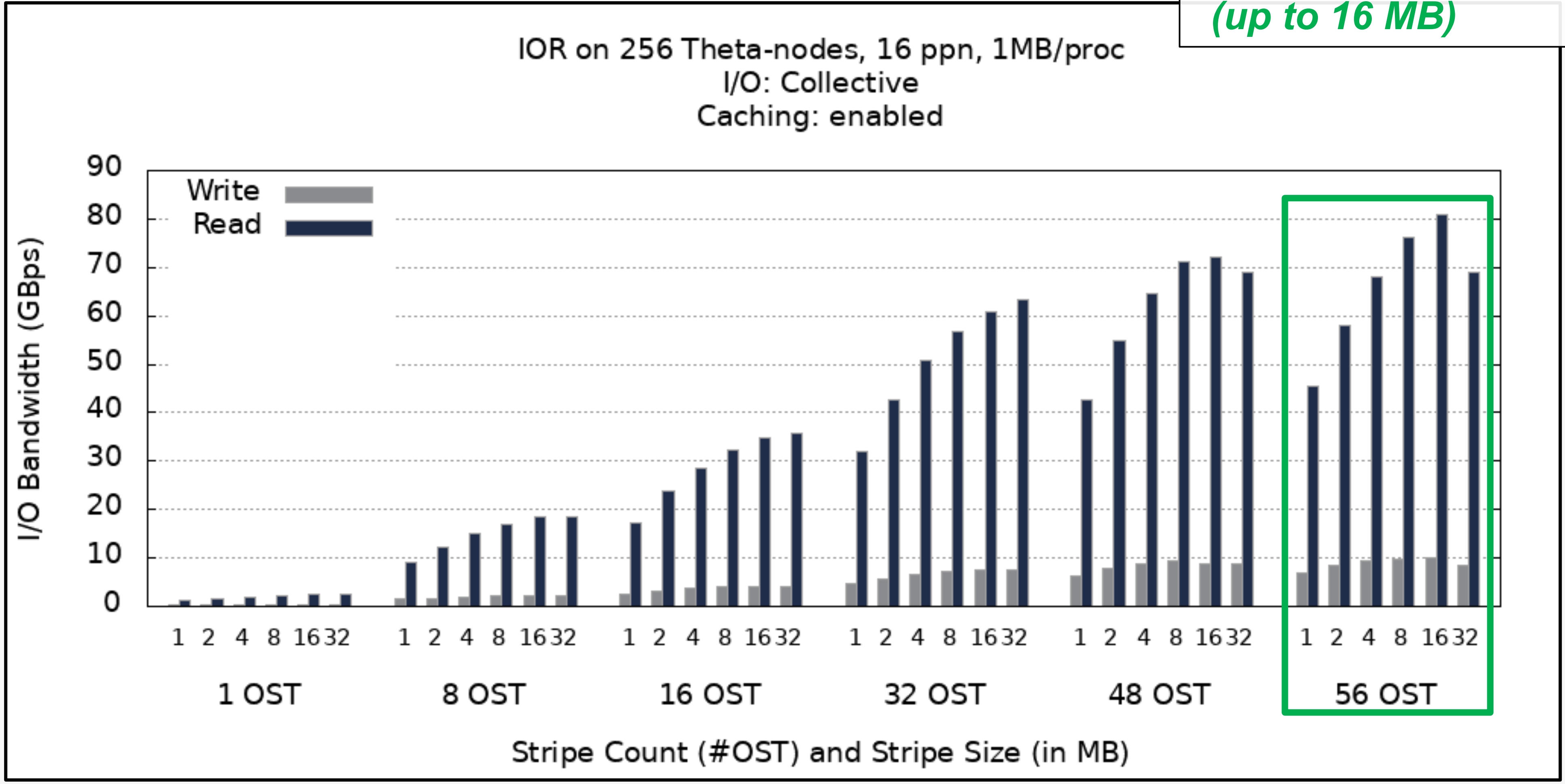
Shared File – 8MB/proc – Independent I/O
Client-side Caching ENABLED

- *Many OSTs are NOT necessary*
- *2 MB stripe size is sufficient*

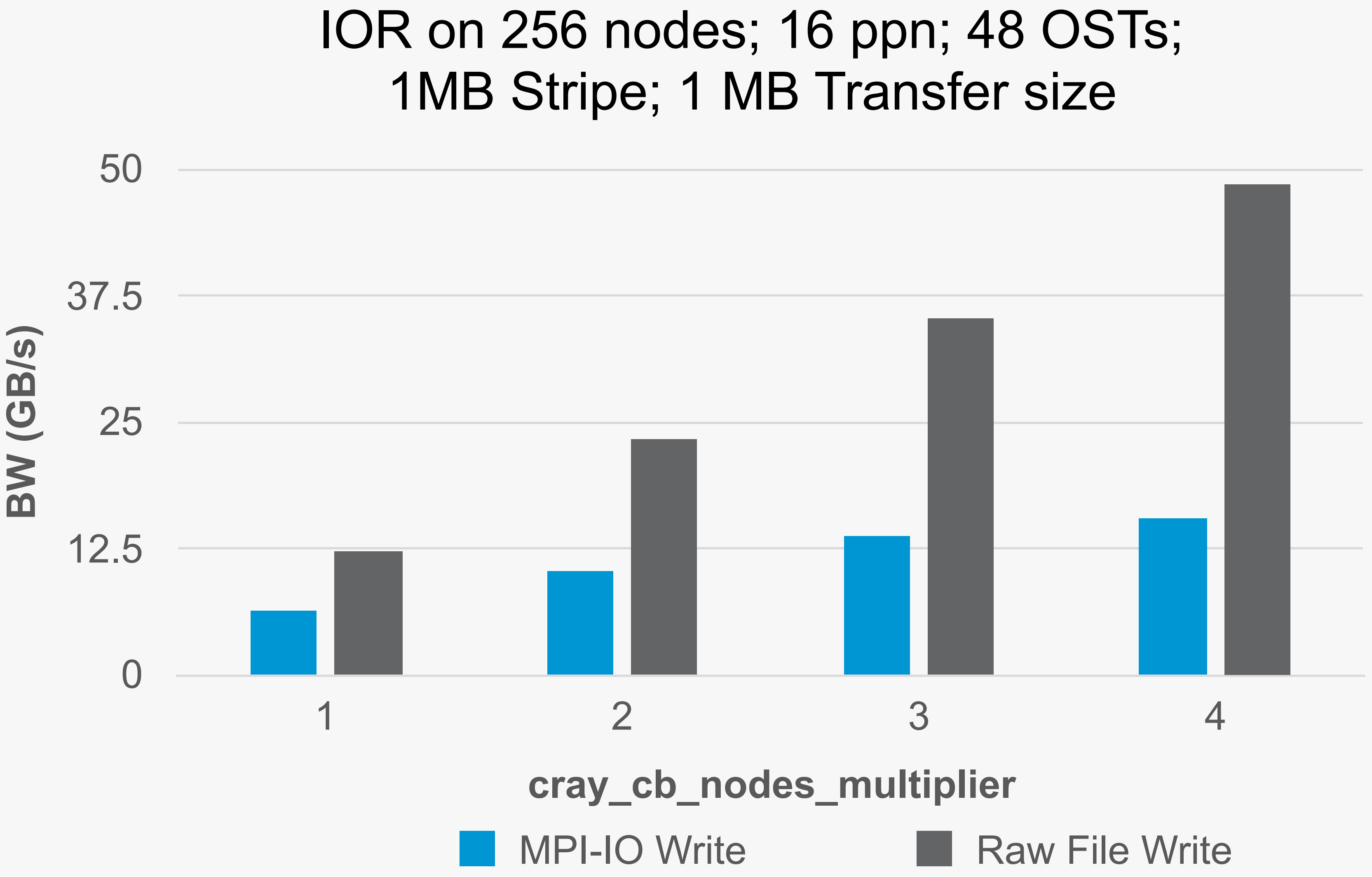


Shared File – 1MB/proc – Collective I/O
Client-side Caching ENABLED

- *More OSTs is better*
- *Larger stripe size is better (up to 16 MB)*



Collective I/O Shared-lock Performance



'Raw File Write' times taken from `MPICH_MPIIO_TIMERS=1` trace

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

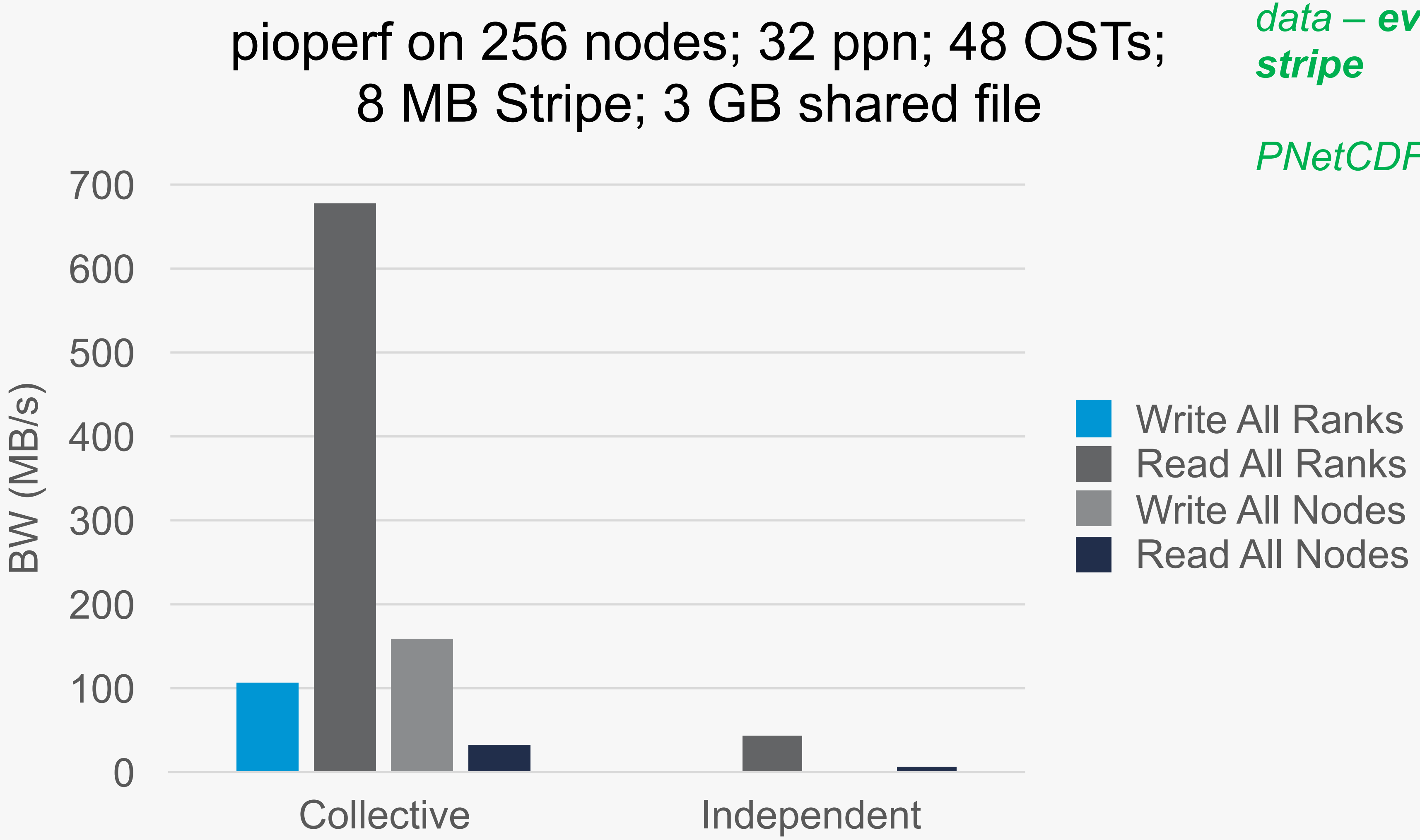
Collective I/O vs Independent I/O

Discontiguous Data

E3SM Climate Modeling Parellel I/O Library performance test tool (pioperf)

*8192 ranks with highly non-contiguous data – **every rank accesses every stripe***

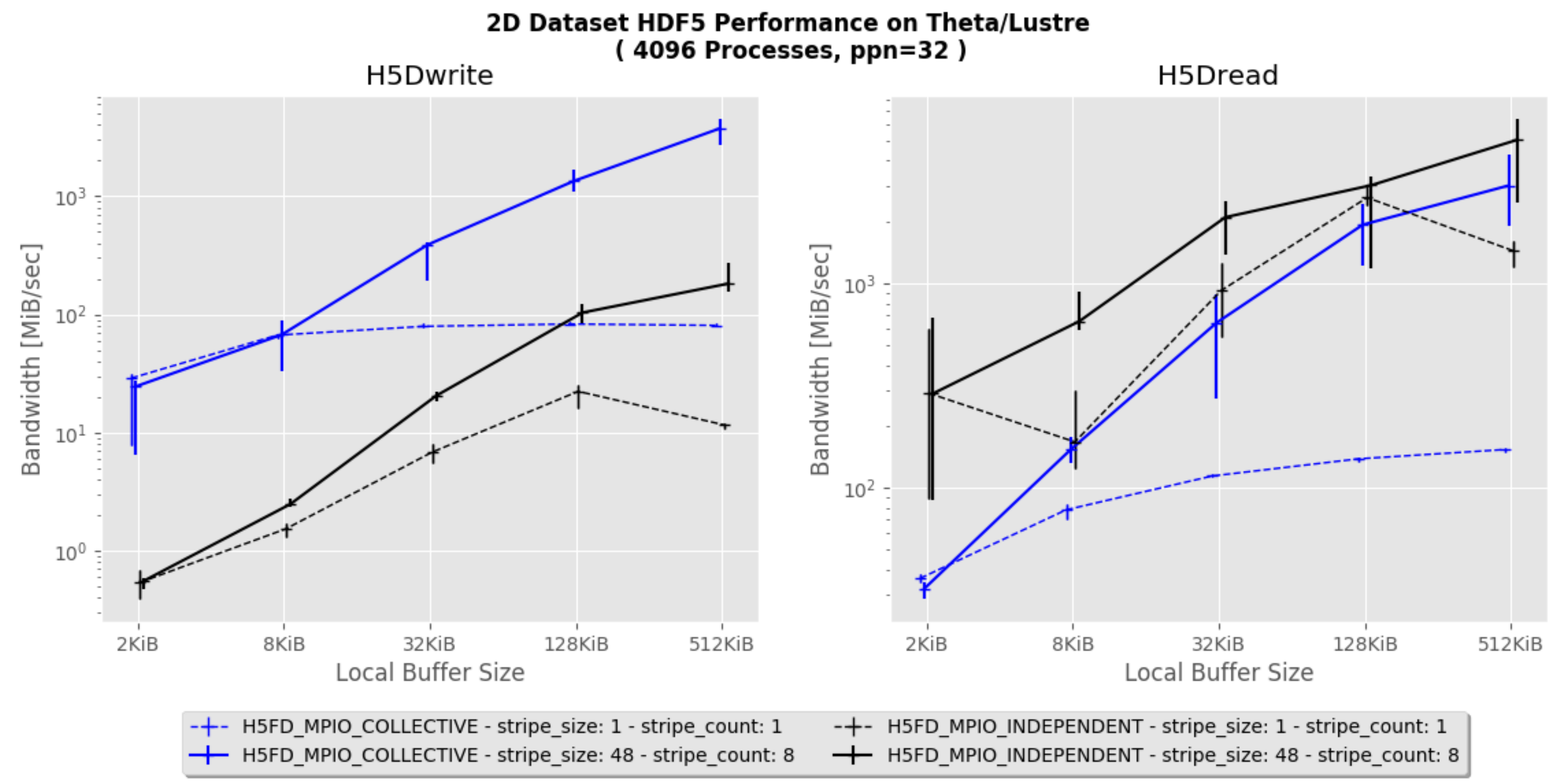
PNetCDF interface (MPI-IO backend)



Collective I/O vs Independent I/O

Parallel HDF5

Collective I/O is often more important for write operations on Theta



HDF5 *Exerciser* Benchmark

Node-Local SSD Utilization on Theta

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

SSD Use Cases:

- Store local intermediate files (scratch)
- Legacy code initialization with lots of small data files – every rank reads
- Untar into local ssd

Tiered storage utility currently unavailable (Under investigation)

Using the SSDs on Theta

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 need
  - `ssd_size` is optional

The SSD are mounted on `/local/scratch` on each node

- Data deleted when cobalt job terminates

SSD I/O Performance (per process): Read **1.1 GB/s** – Write **175 MB/s**

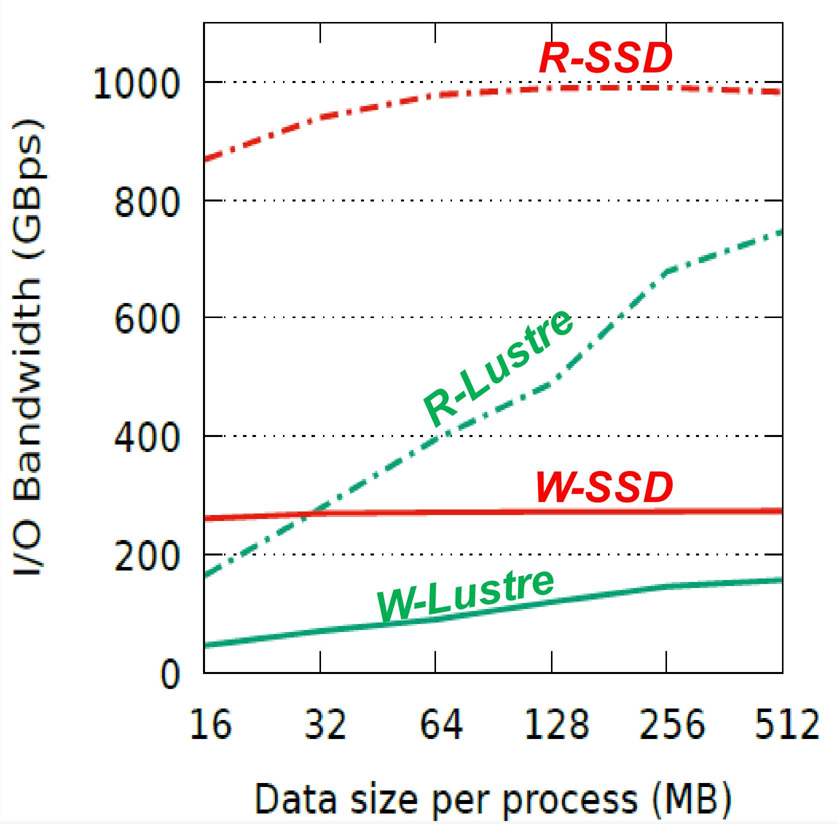
- Can scale to two processes
- Outperforms Lustre at scale (aggregated bandwidth)
- Node-limited scope
- Requires explicit manual programming

Node-Local SSD Performance

SSD performance is more scalable than Lustre. Beyond 256 nodes, SSD's can provide a significant advantage

Node-local SSD vs Lustre

IOR, 1024 nodes, 2 ppn, 1 fpp



Model SM961 drives

Capacity	128 GB
Sequential Read	3100 MB/s
Sequential Write	700 MB/s

Model SM951 drives

Capacity	128 GB
Sequential Read	2150 MB/s
Sequential Write	1550 MB/s

Conclusions

- High-performance I/O on both Mira and Theta often require MPI-IO (or an I/O library)

- Key to Theta is efficient Lustre access
  - Choose appropriate striping
  - Use optimized Cray MPI-IO
  - Use I/O libraries (HDF5, PNetCDF)

- Use local SSDs on Theta to scale small-file I/O, etc

ALCF Staff is available to help!

Thank You!