# Understanding and Tuning I/O Performance

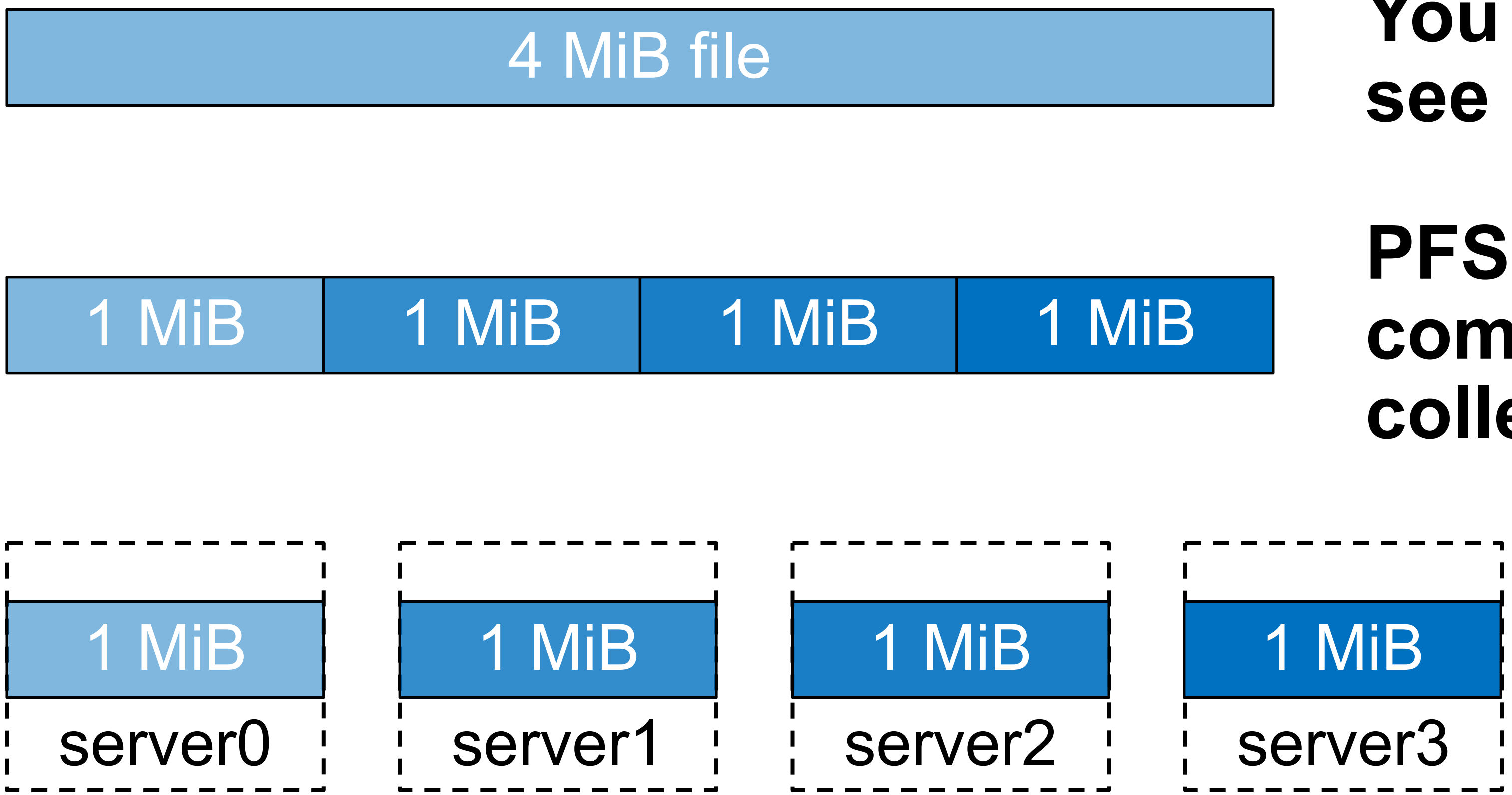
ATPESC 2019

Glenn K. Lockwood
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

Q Center, St. Charles, IL (USA)
July 28 – August 9, 2019

# Parallel file systems in principle

File system that spreads files across multiple servers (∴ many NICs and drives)

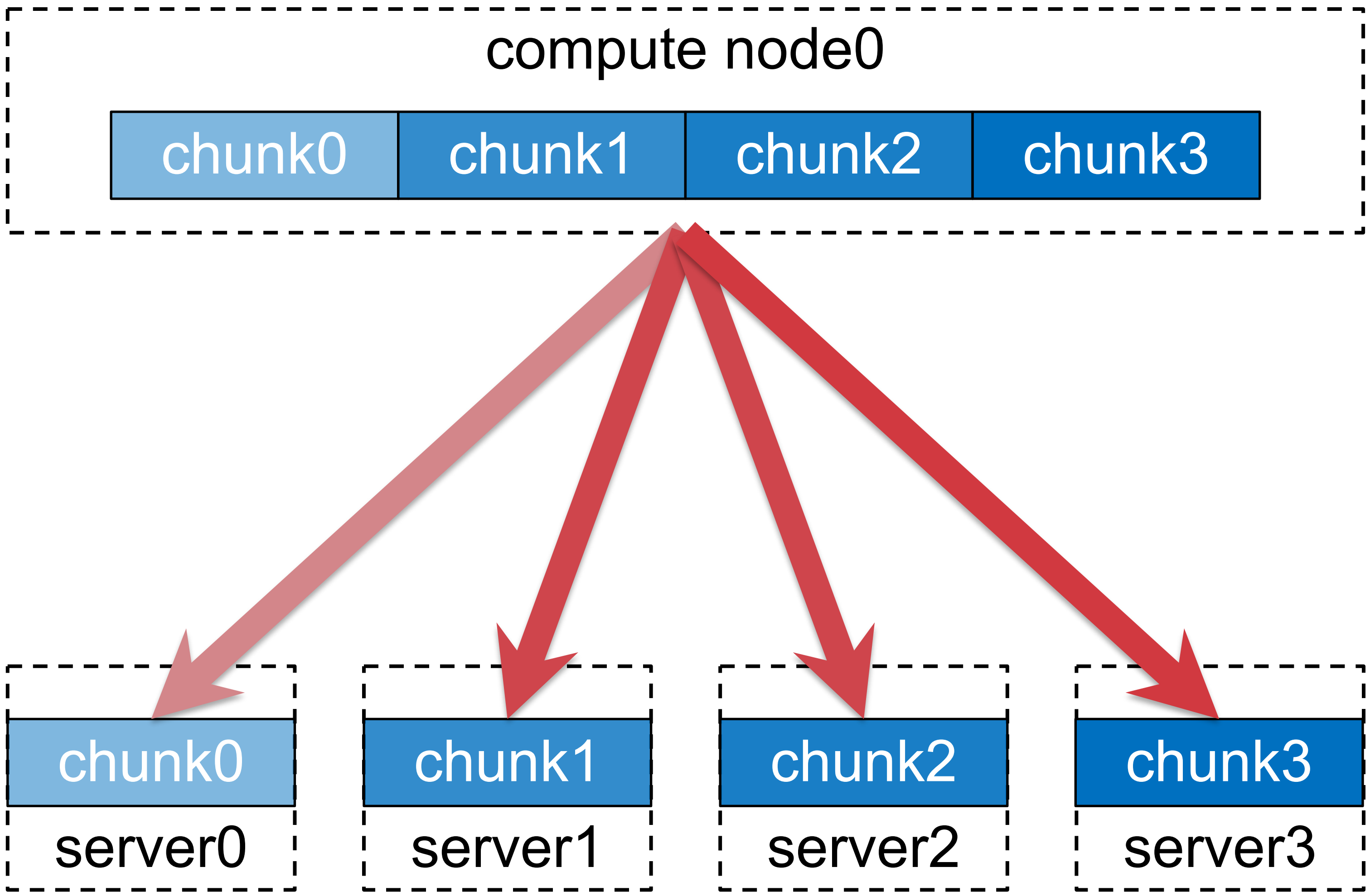



**You and your application see one big file**

**PFS driver on your compute nodes see a collection of chunks**

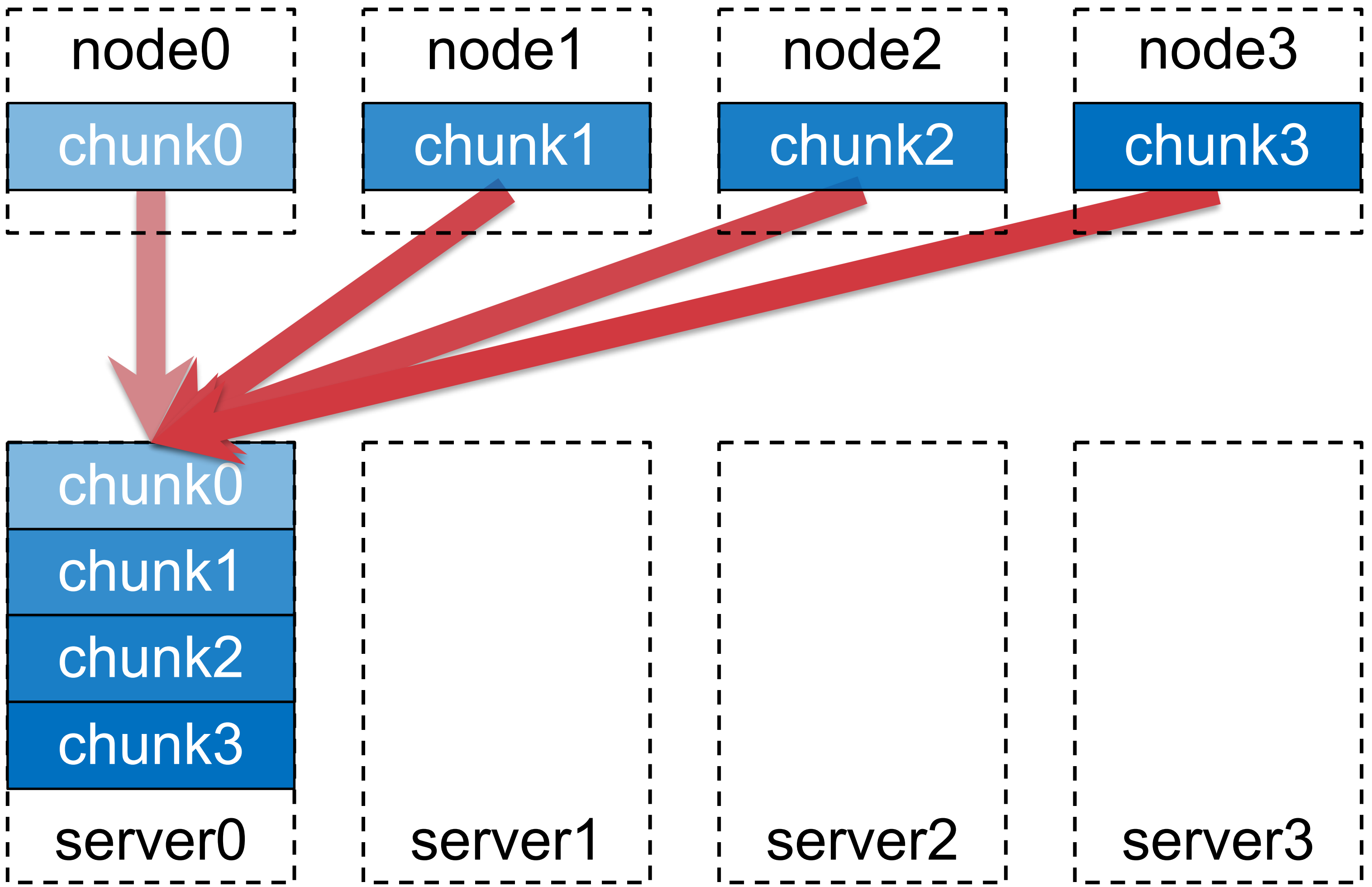
**PFS servers see individual chunks**

# The speed of light still applies



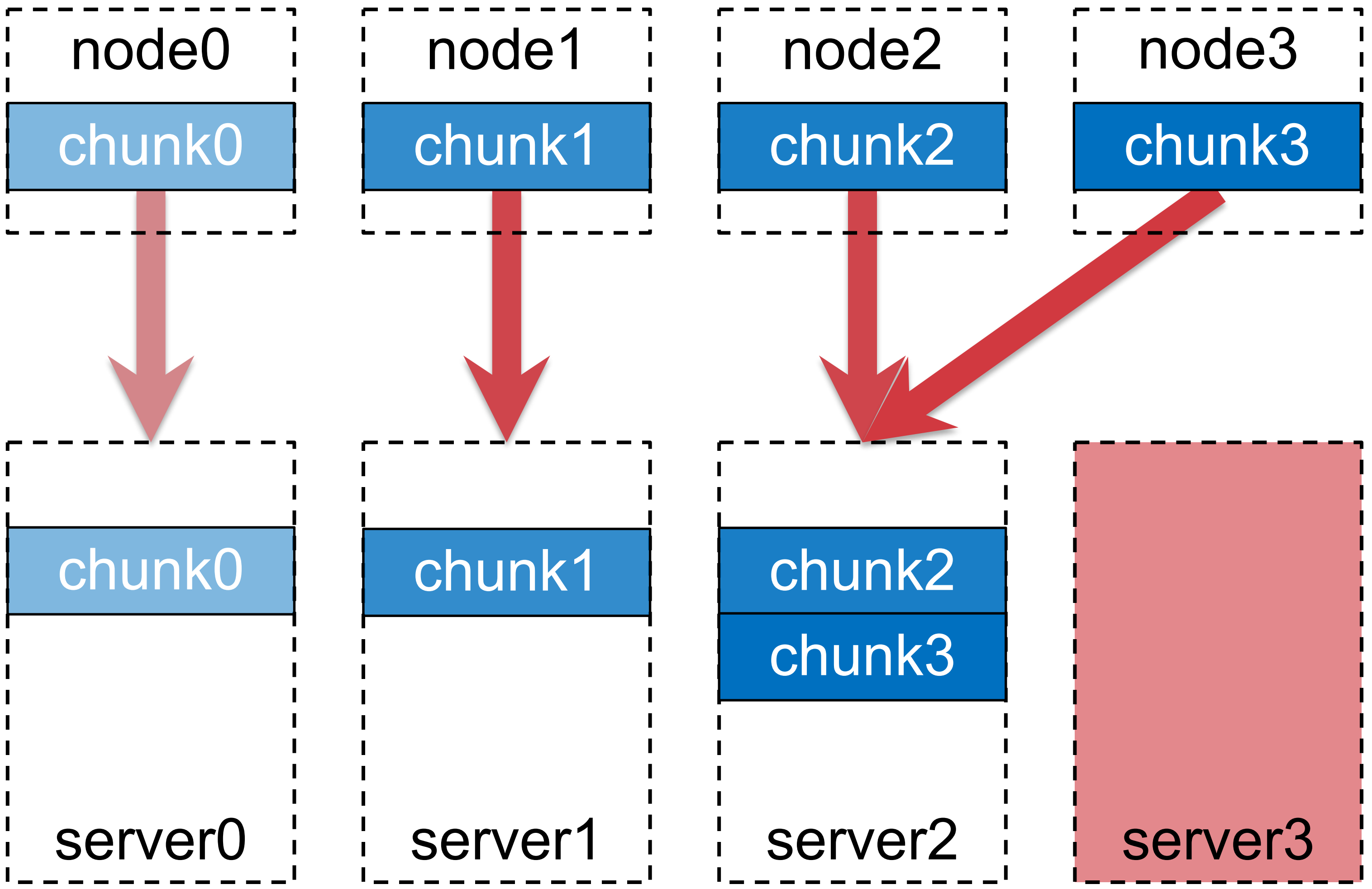

**One compute node can't talk to every storage server at full speed**

# The speed of light still applies



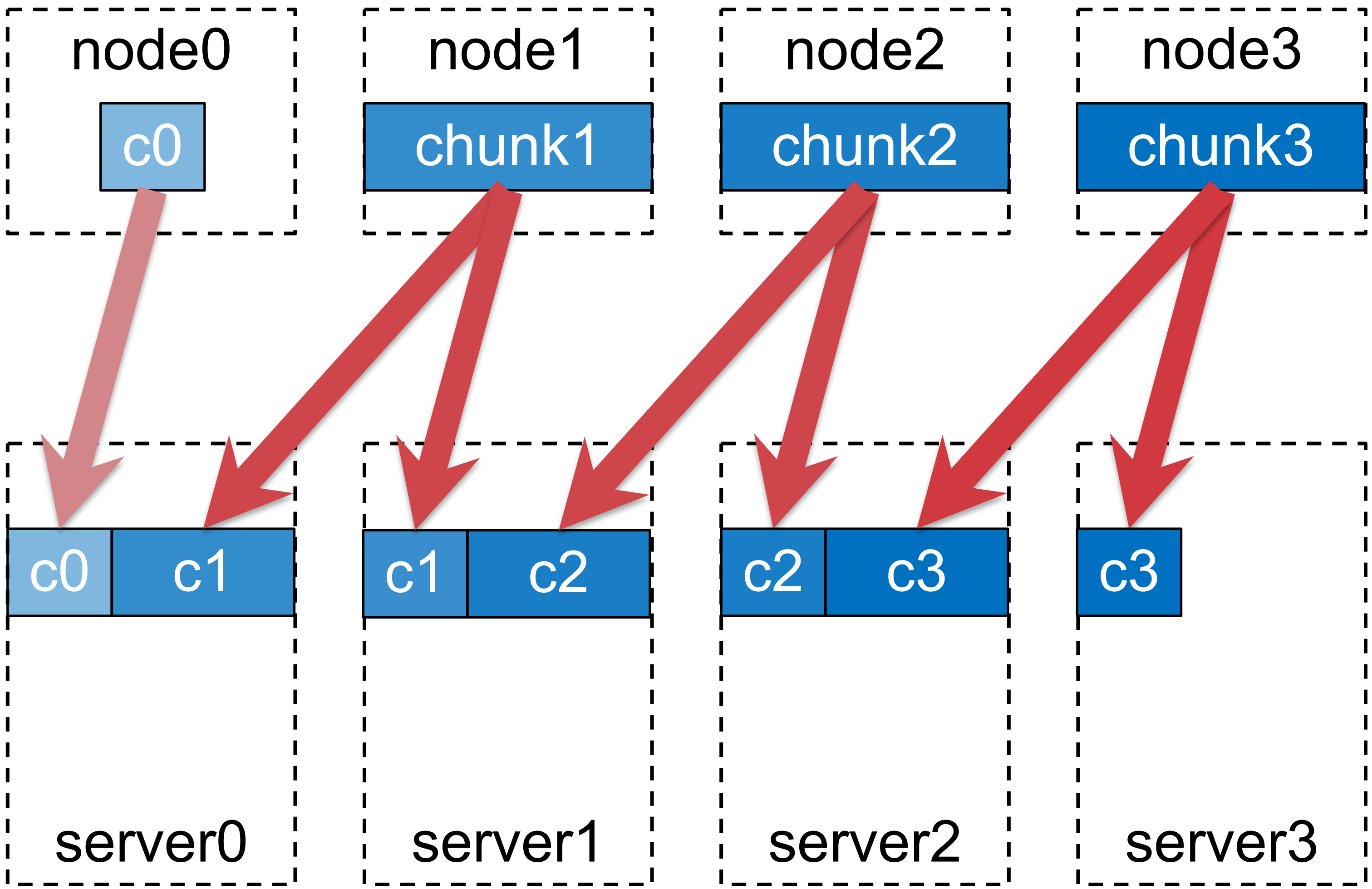

**One storage server can't talk to every compute node at full speed**

# The speed of light still applies



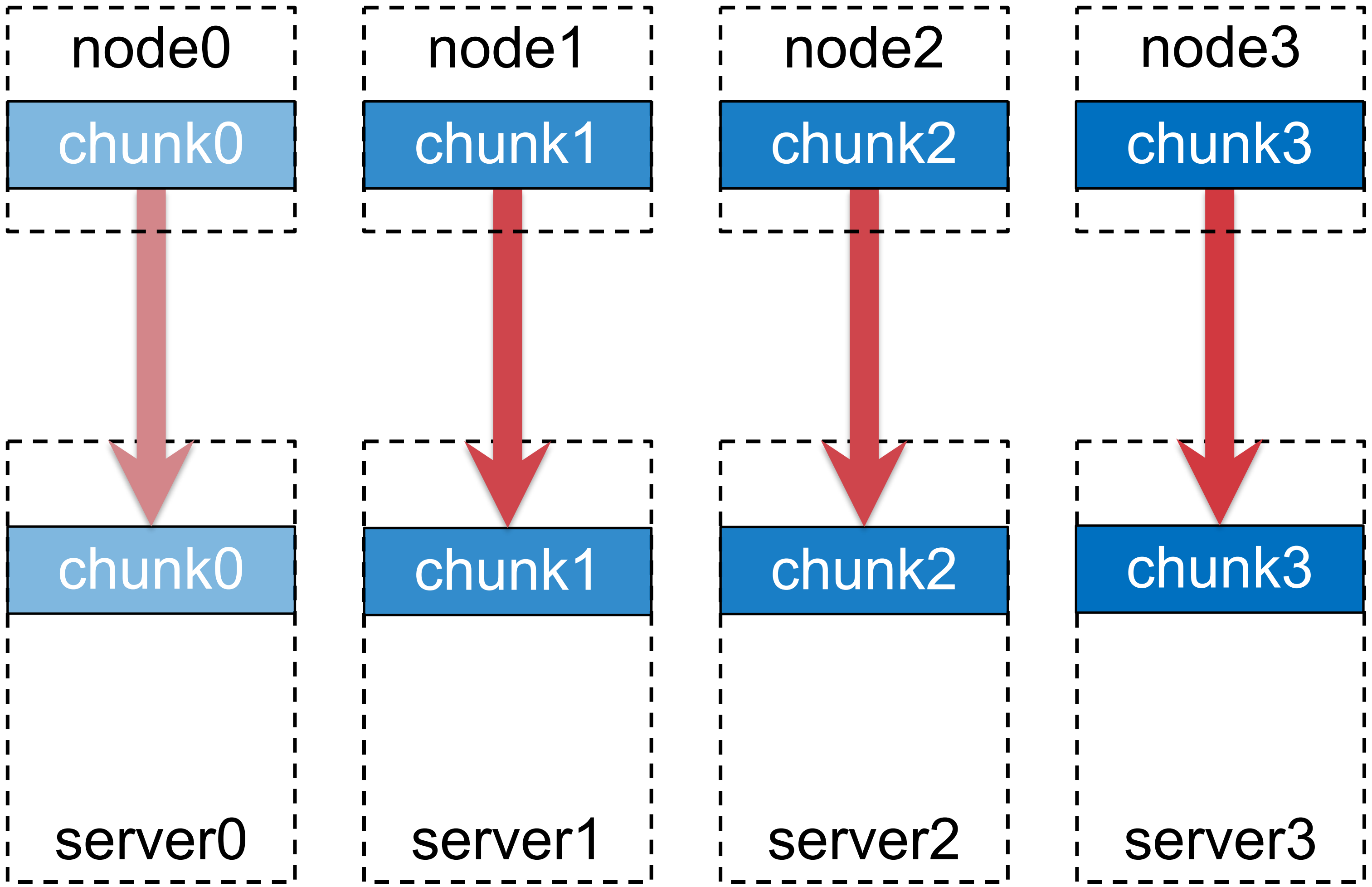

**Accidental imbalance caused by a server failure**

# The speed of light still applies




**Accidental imbalance caused by misalignment**

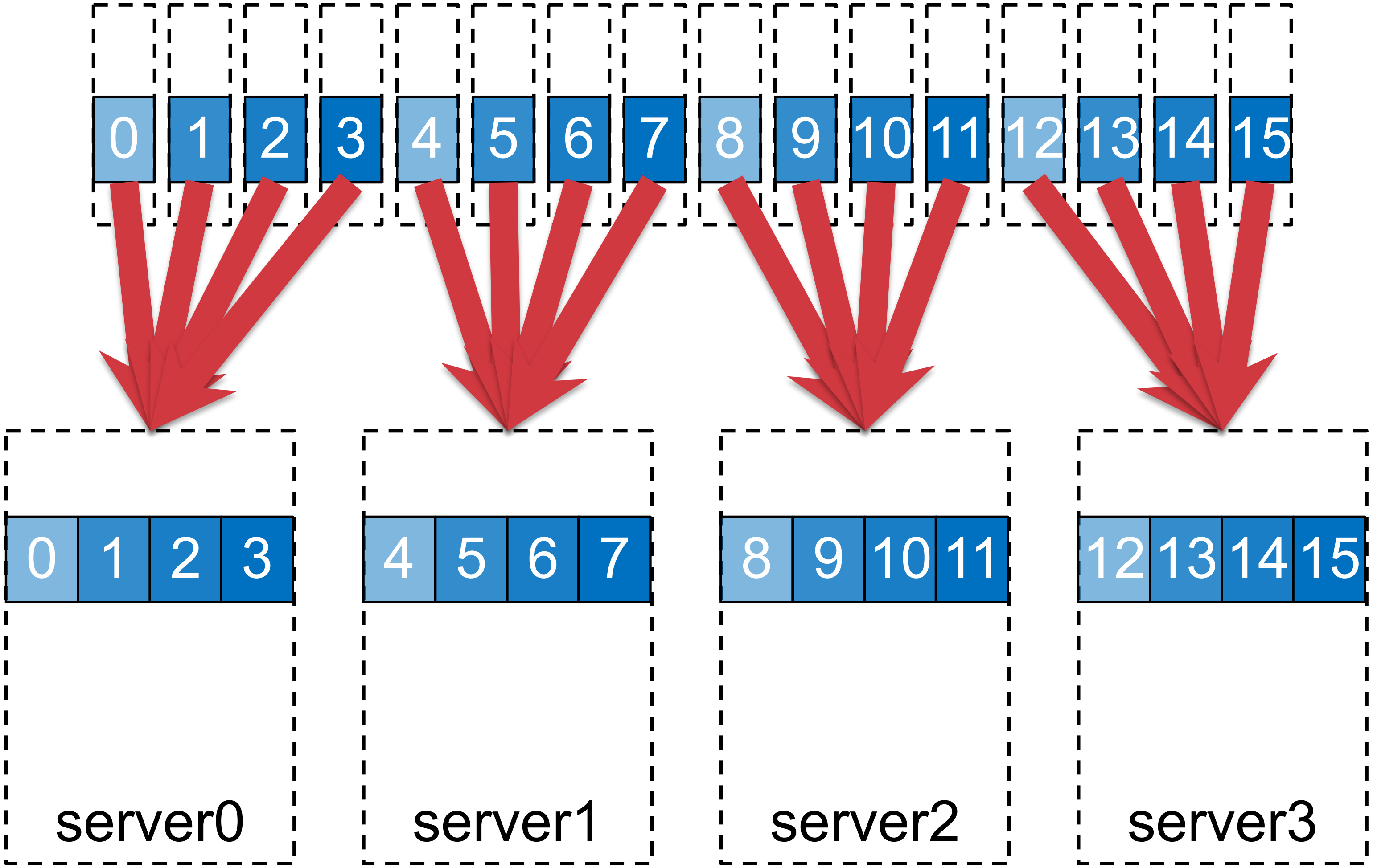
# The speed of light still applies




## Overall goals when doing I/O to a PFS:

- Each client *and server* handle the same data volume
- Work around gotchas specific to the PFS implementation

# The speed of light still applies




**Overall goals when doing I/O to a PFS:**

- Each client *and* *server* handle the same data volume
- Work around gotchas specific to the PFS implementation

# Understanding I/O performance and behavior using Darshan

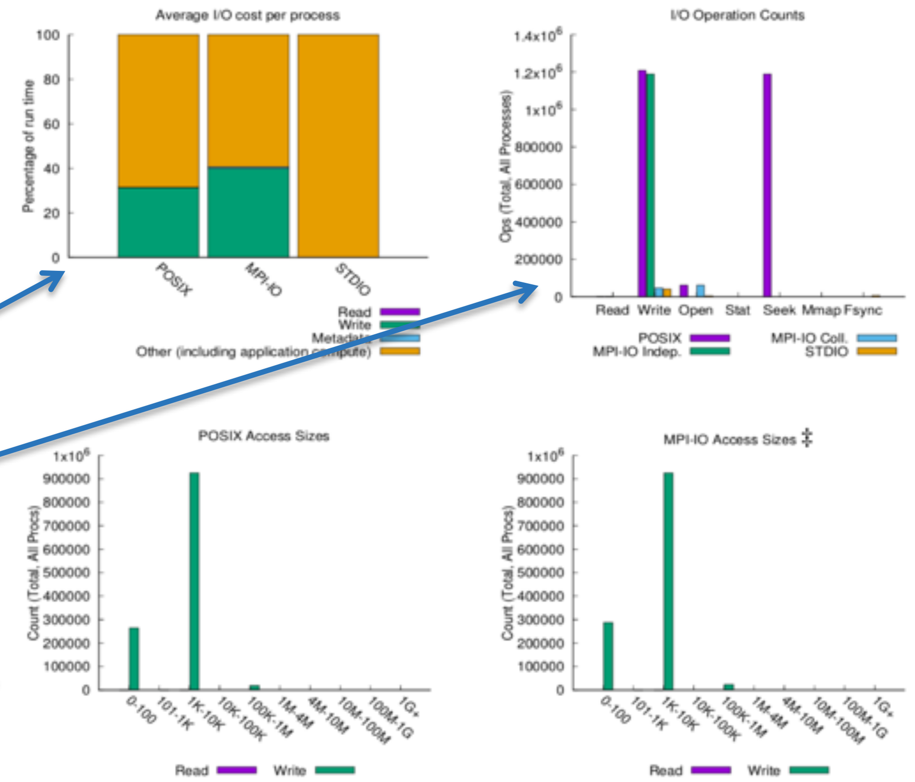
# Job-Level Performance Analysis

- Darshan provides insight into the I/O behavior and performance of a job
- darshan-job-summary.pl creates a PDF file summarizing various aspects of I/O performance
  - Percent of runtime spent in I/O
  - Operation counts
  - Access size histogram
  - Access type histogram
  - File usage

Han

| jobid: 9544193 | uid: 59902 | nprocs: 4720 | runtime: 1512 seconds |
|---|---|---|---|

I/O performance *estimate* (at the MPI-IO layer): transferred 411195 MiB at 24.77 MiB/s
I/O performance *estimate* (at the STDIO layer): transferred 0.4 MiB at 0.77 MiB/s




Most Common Access Sizes
(POSIX or MPI-IO)

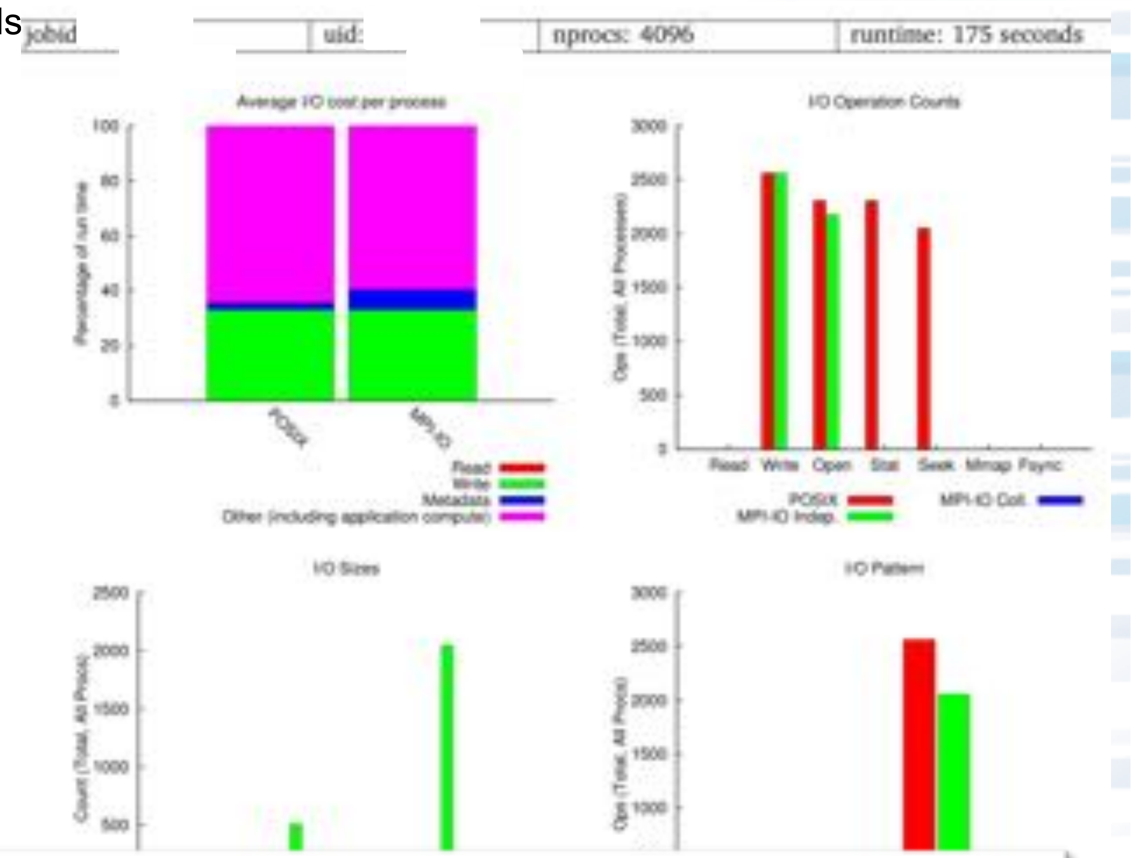
| | access size | count |
|---|---|---|
| POSIX | 4096 | 660800 |
| | 68 | 47200 |
| | 64 | 47200 |
| | 24 | 23500 |
| MPI-IO ‡ | 4096 | 660801 |
| | 64 | 47200 |
| | 68 | 47200 |
| | 24 | 23500 |

‡ NOTE: MPI-IO accesses are given in terms of aggregate datatype size.

File Count Summary
(estimated by POSIX I/O access offsets)

| type | number of files | avg. size | max size |
|---|---|---|---|
| total opened | 13 | 1.4G | 3.7G |
| read-only files | 1 | 2.1K | 2.1K |
| write-only files | 11 | 1.6G | 3.7G |
| read/write files | 0 | 0 | 0 |
| created files | 11 | 1.6G | 3.7G |

Hands




# Example: Is your app doing what you think it's doing?

- App opens 129 files (one “control” file, 128 data files)
- User expected one ~40 KiB header per data file
- Darshan showed 512 headers being written
- Code bug: header was written 4× per file

**Most Common Access Sizes**

| access size | count |
|---|---|
| 67108864 | 2048 |
| 41120 | 512 |
| 8 | 4 |
| 4 | 3 |

**File Count Summary**

| type | number of files | avg. size | max size |
|---|---|---|---|
| total opened | 129 | 1017M | 1.1G |
| read-only files | 0 | 0 | 0 |
| write-only files | 129 | 1017M | 1.1G |
| read/write files | 0 | 0 | 0 |
| created files | 129 | 1017M | 1.1G |

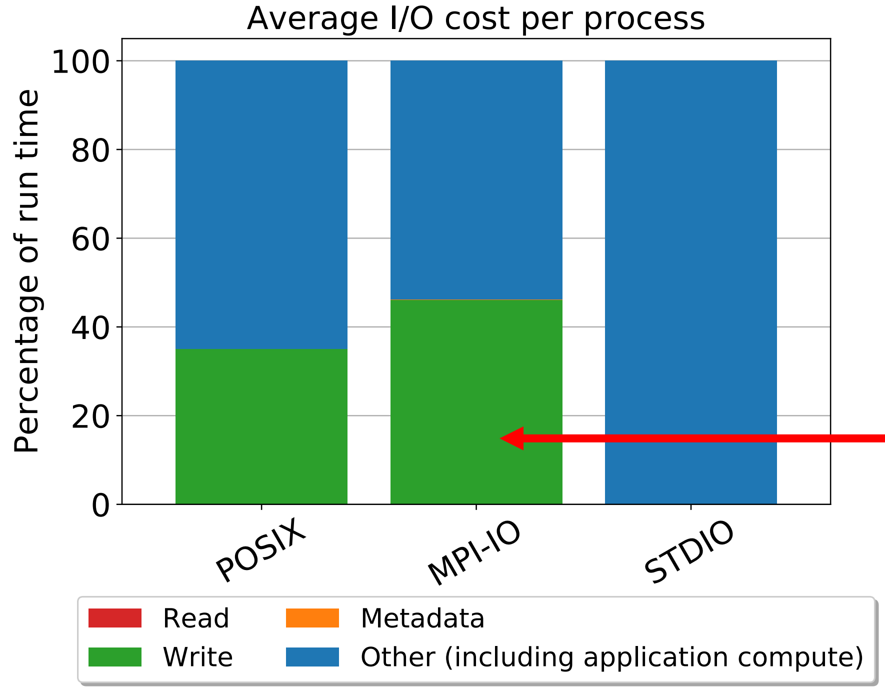
# Example: When doing the right thing goes wrong

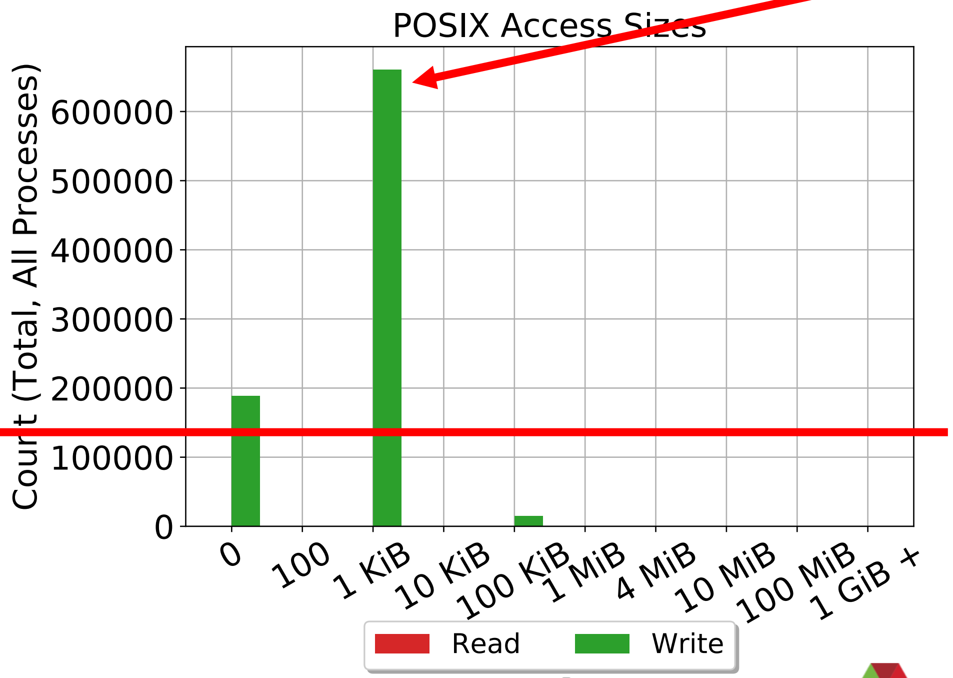
- Large-scale astrophysics application using MPI-IO
- Used 94,000,000 CPU hours at NERSC since 2015

| | | | |
|---|---|---|---|
| jobid: 1950915785 | uid: 81587417 | nprocs: 4720 | runtime: 1005 seconds |

I/O performance *estimate* (at the MPI-IO layer): transferred 17443.1 MiB at 26.35 MiB/s
I/O performance *estimate* (at the STDIO layer): transferred 0.1 MiB at 1673.25 MiB/s

**Wrote 17 GiB using 4 KiB transfers**




**40% of walltime spent doing I/O *even with MPI-IO***

Case study courtesy Jialin Liu and Quincey Koziol (NERSC)

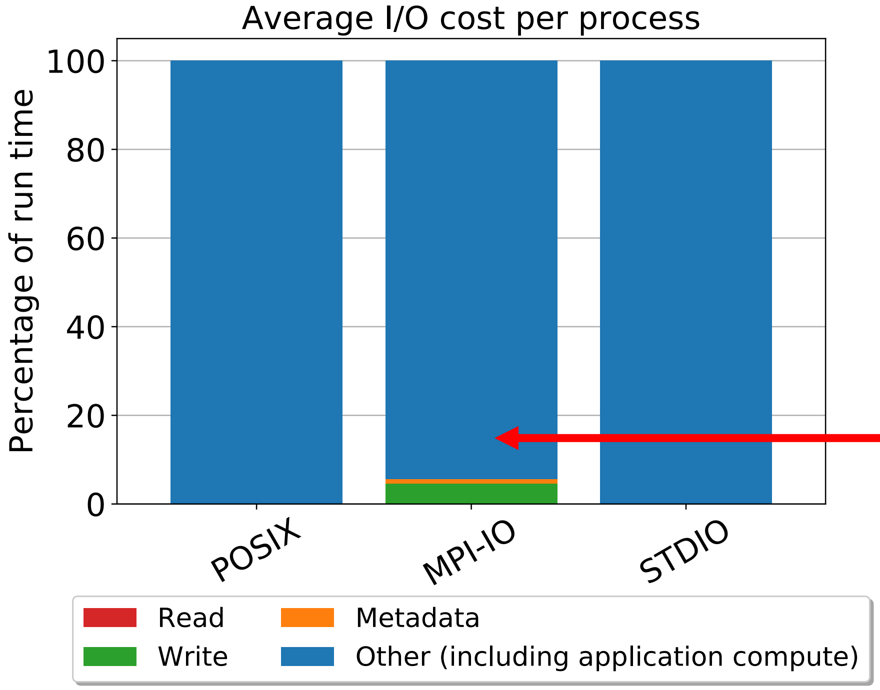
Argonne NATIONAL LABORATORY
ECP EXASCALE COMPUTING PROJECT

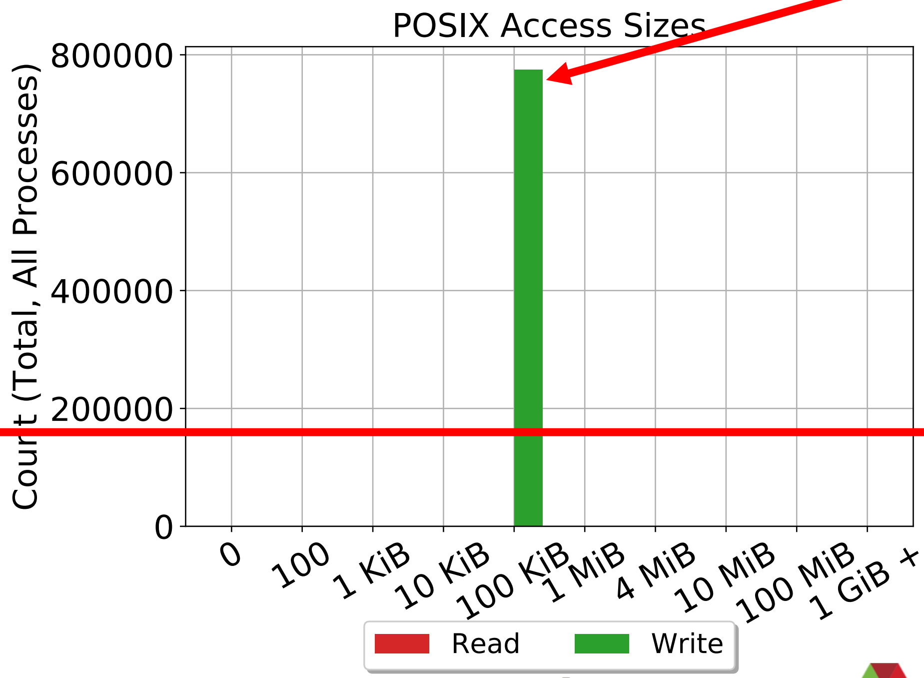
# Example: When doing the right thing goes wrong

- Collective I/O was being disabled by middleware due to type mismatch in app code
- After type mismatch bug fixed, collective I/O gave 40× speedup

| | | | |
|---|---|---|---|
| jobid: 2308034461 | uid: 81587417 | nprocs: 77312 | runtime: 11680 seconds |

I/O performance *estimate* (at the MPI-IO layer): transferred 774986.3 MiB at 1143.95 MiB/s
I/O performance *estimate* (at the STDIO layer): transferred 12298.7 MiB at 608.59 MiB/s




**Wrote 756 GiB in 1 MiB transfers (>40× speedup)**

**6% of walltime spent doing I/O**

# Example: Redundant read traffic

- Applications sometimes read more bytes from a file than the file's size
  - Can cause disruptive I/O network traffic and storage contention
  - Good candidate for aggregation, collective I/O, or burst buffering
- Common pattern in emerging AI/ML workloads
- Example:
  - Scale: 6,138 processes
  - Run time: 6.5 hours
  - Avg. I/O time per proc: 27 minutes
- 1.3 TiB of file data
- 500+ TiB read!

File Count Summary
(estimated by I/O access offsets)

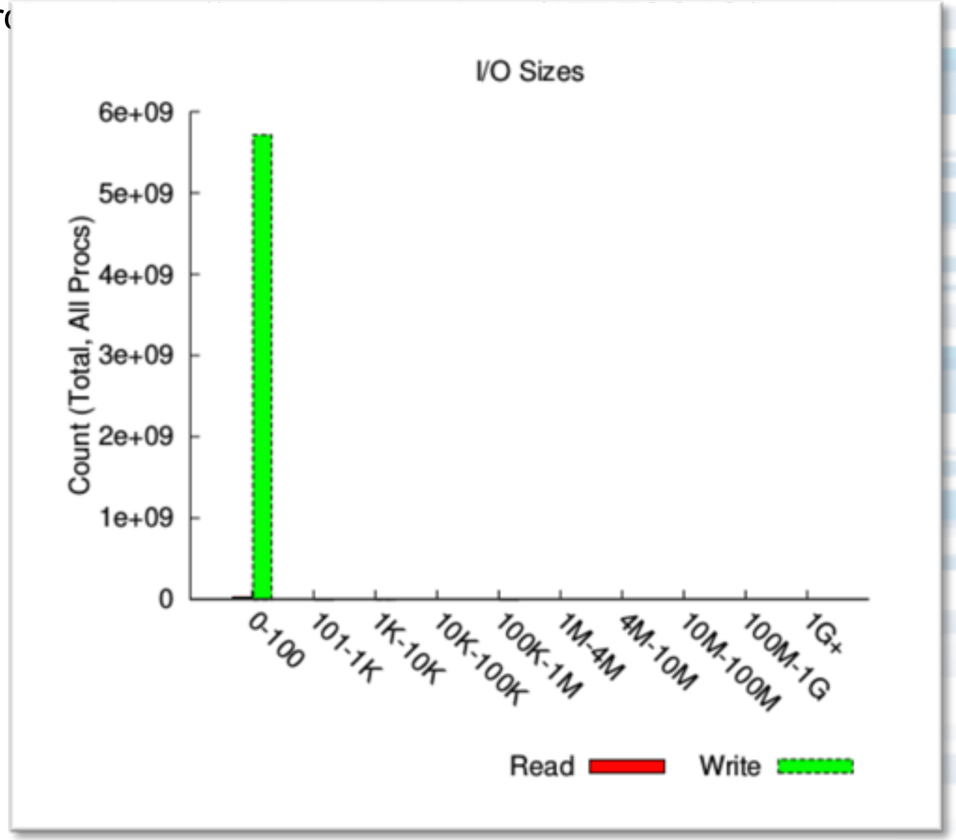
| type | number of files | avg. size | max size |
|---|---|---|---|
| total opened | 1299 | 1.1G | 8.0G |
| read-only files | 1187 | 1.1G | 8.0G |
| write-only files | 112 | 418M | 2.6G |
| read/write files | 0 | 0 | 0 |
| created files | 112 | 418M | 2.6G |

Data Transfer Per Filesystem

| File System | Write | | Read | |
|---|---|---|---|---|
| | MiB | Ratio | MiB | Ratio |
| / | 47161.47354 | 1.00000 | 575224145.24837 | 1.00000 |

# Example: Small writes to shared files

- **Scenario: Small writes can contribute to poor performance**
  - Particularly when writing to shared files
  - Candidates for collective I/O or batching/buffering of write operations
- **Example:**
  - Issued 5.7 billion writes to shared files, each less than 100 bytes in size
  - Averaged just over 1 MiB/s per process during shared write phase

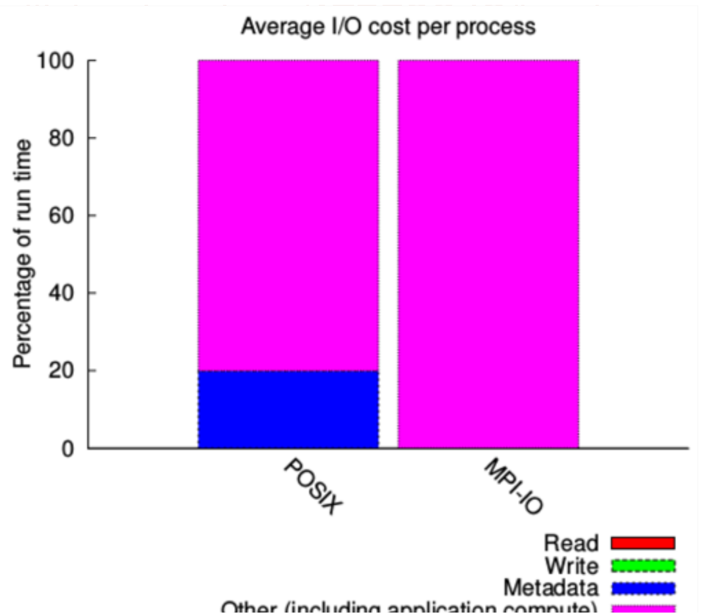


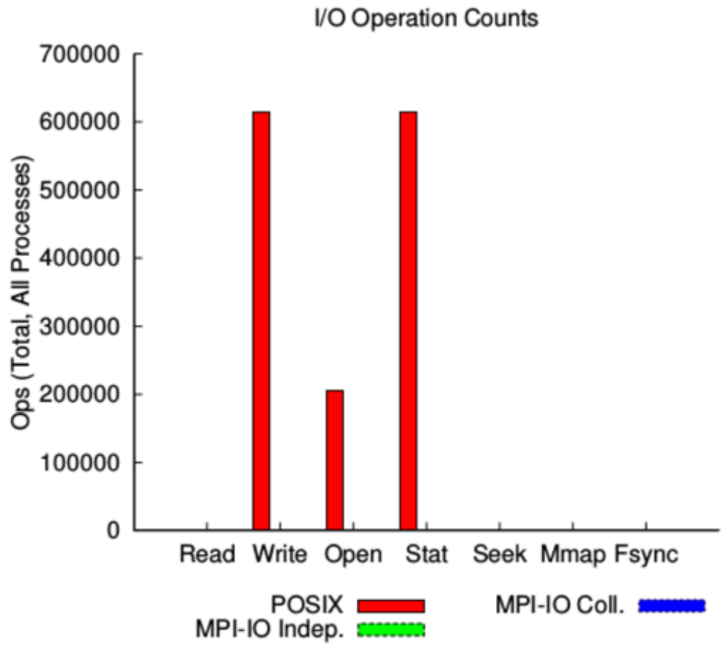

Most Common Access Sizes

| access size | count |
|---|---|
| 1 | 3418409696 |
| 15 | 2275400442 |
| 24 | 42289948 |
| 12 | 14725053 |

# Example: Excessive metadata overhead

- **Scenario: Most I/O time spent on metadata ops (open, stat, etc.)**
  - close() cost can be misleading due to write-behind cache flushing!
  - Remedy: coalescing files to eliminate extra metadata calls
- **Example:**
  - Scale: 40,960 processes, > 20% time spent in I/O
  - 99% of I/O time in metadata operations
  - Generated 200,000+ files with 600,000+ write and 600,000+ stat calls



# Available Darshan Analysis Tools

- Docs: http://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html
- Officially supported tools
  - **darshan-job-summary.pl**: Creates PDF with graphs for initial analysis
  - **darshan-summary-per-file.sh**: Similar to above, but produces a separate PDF summary for every file opened by application
  - **darshan-parser**: Dumps all information into text format
- Third-party tools
  - **darshan-ruby**: Ruby bindings for darshan-util C library
    https://xgitlab.cels.anl.gov/darshan/darshan-ruby
  - **HArshaD**: Easily find and compare Darshan logs
    https://kaust-ksl.github.io/HArshaD/
  - **pytokio**: Detect slow Lustre OSTs, create Darshan scoreboards, etc.
    https://pytokio.readthedocs.io/



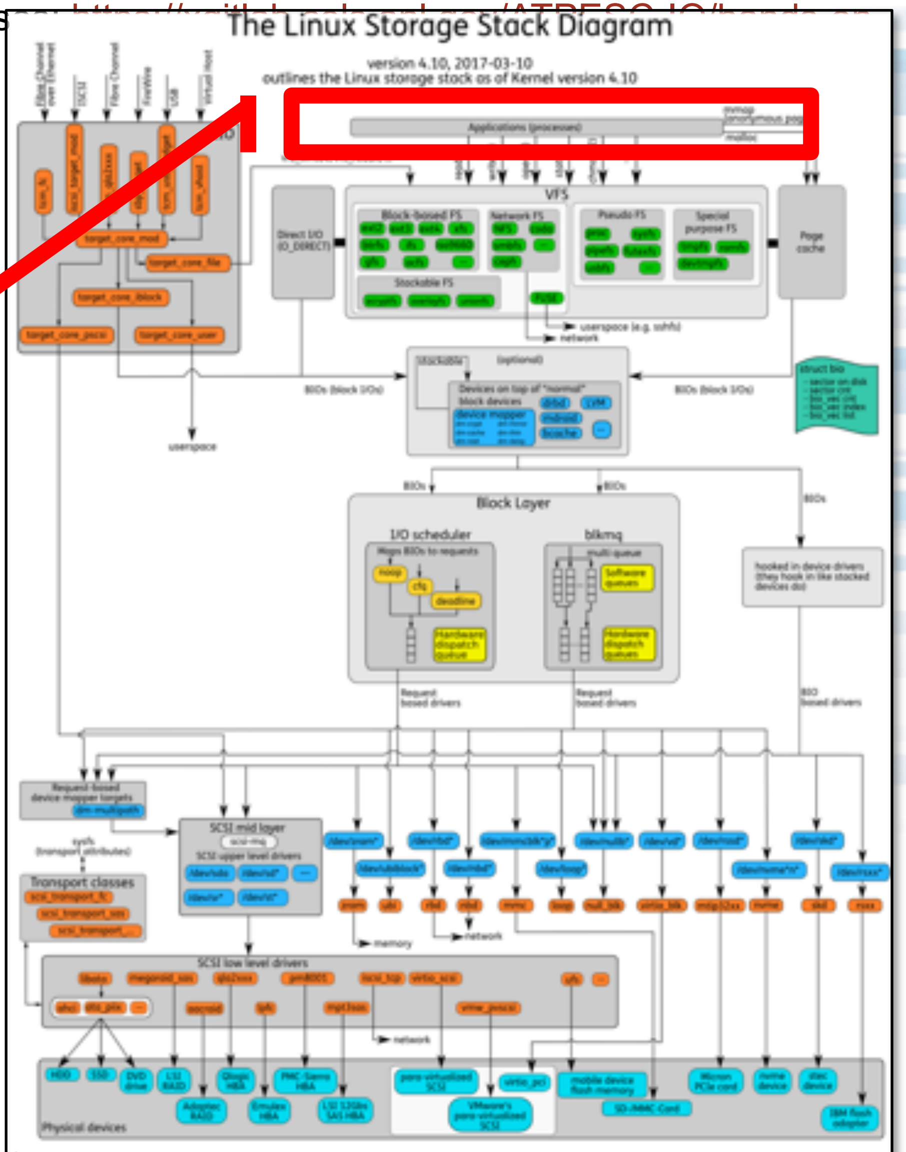
Argonne NATIONAL LABORATORY EXASCALE COMPUTING PROJECT

# Measuring I/O performance

Hands on exercis

# Fine-tuning performance requires benchmarking

- Darshan tells you what your application is trying to do
- A lot more is happening that Darshan cannot (is not allowed to) see
- Benchmarking your I/O pattern is often a necessary part of optimization



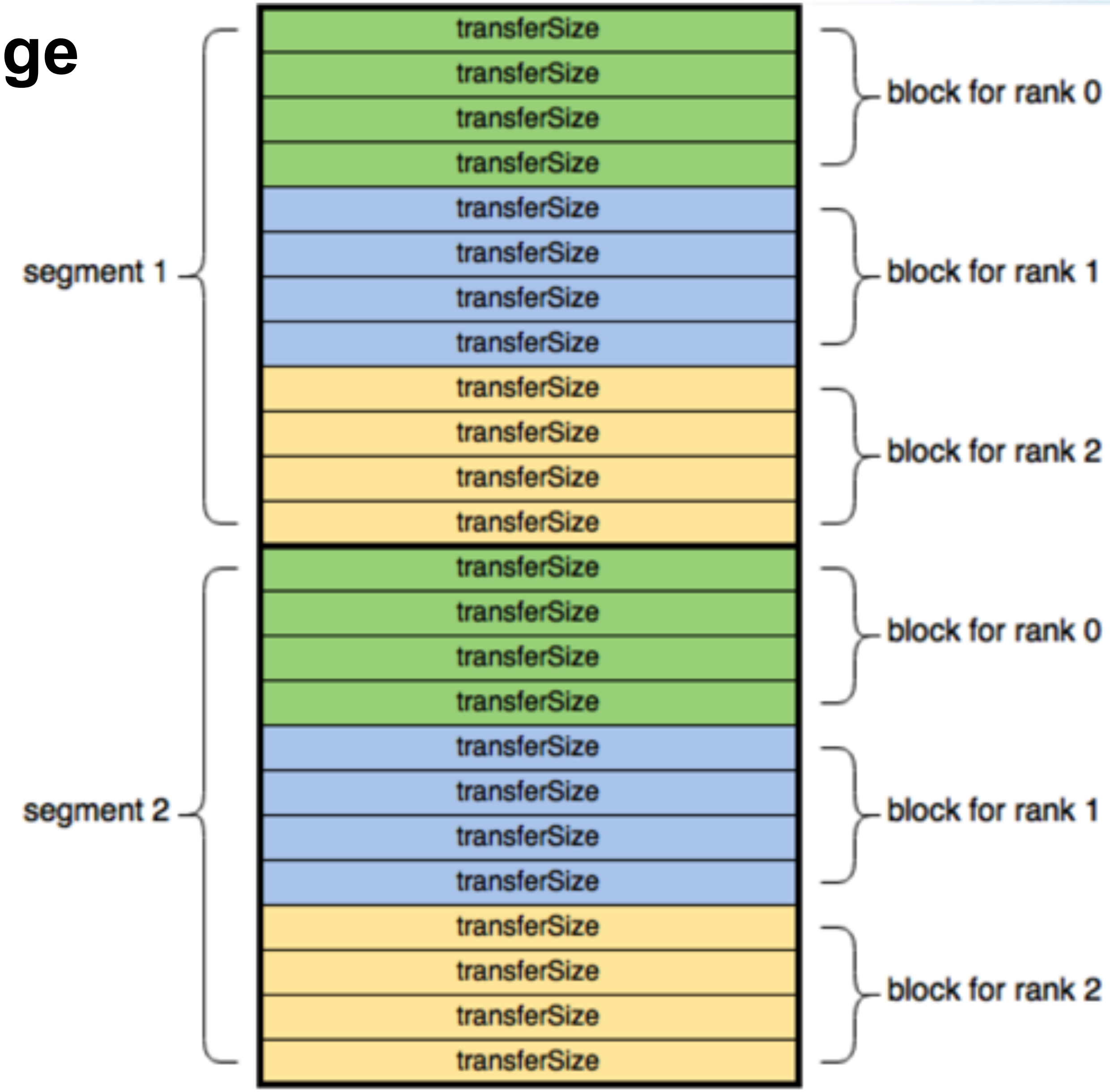

Source: Thomas Krenn
https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram


Argonne NATIONAL LABORATORY
ECP EXASCALE COMPUTING PROJECT

# IOR – A tool for measuring storage system performance

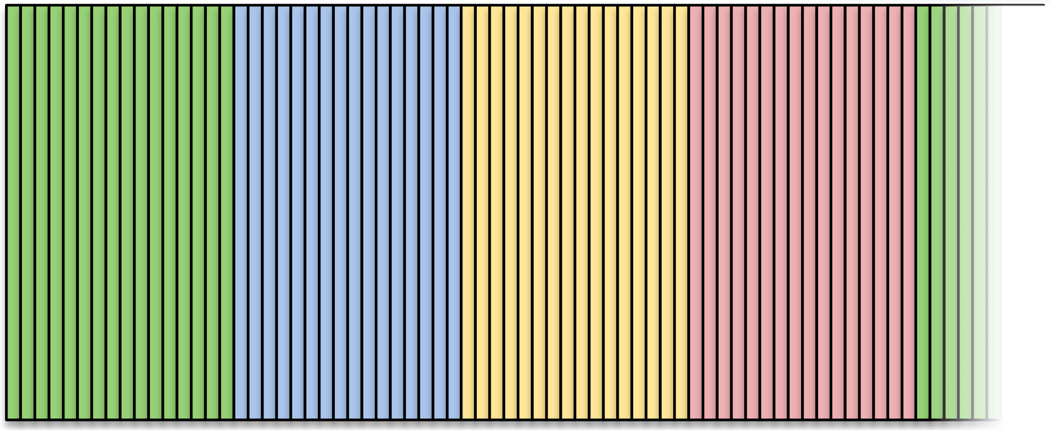
- MPI application benchmark that reads and writes data in configurable ways
- I/O pattern can be interleaved or random
- Input: Desired transfer sizes, block sizes, and segment counts
- Output: Bandwidth and IOPS
- Configurable backends
  - POSIX, STDIO, MPI-IO
  - HDF5, PnetCDF, rados

**https://github.com/hpc/ior/releases**



# First attempt at benchmarking an I/O pattern

- On a 700 GB/sec Lustre file system (Cori)
- 4 nodes, 16 ppn
- Performance makes no sense
  - write performance is awful
  - read performance is phenomenal



```
$ mpirun -n 64 ./ior -t 1m -b 16m -s 256
...
Max Write: 450.54 MiB/sec (472.43 MB/sec)
Max Read:  27982.41 MiB/sec (29341.68 MB/sec)
```

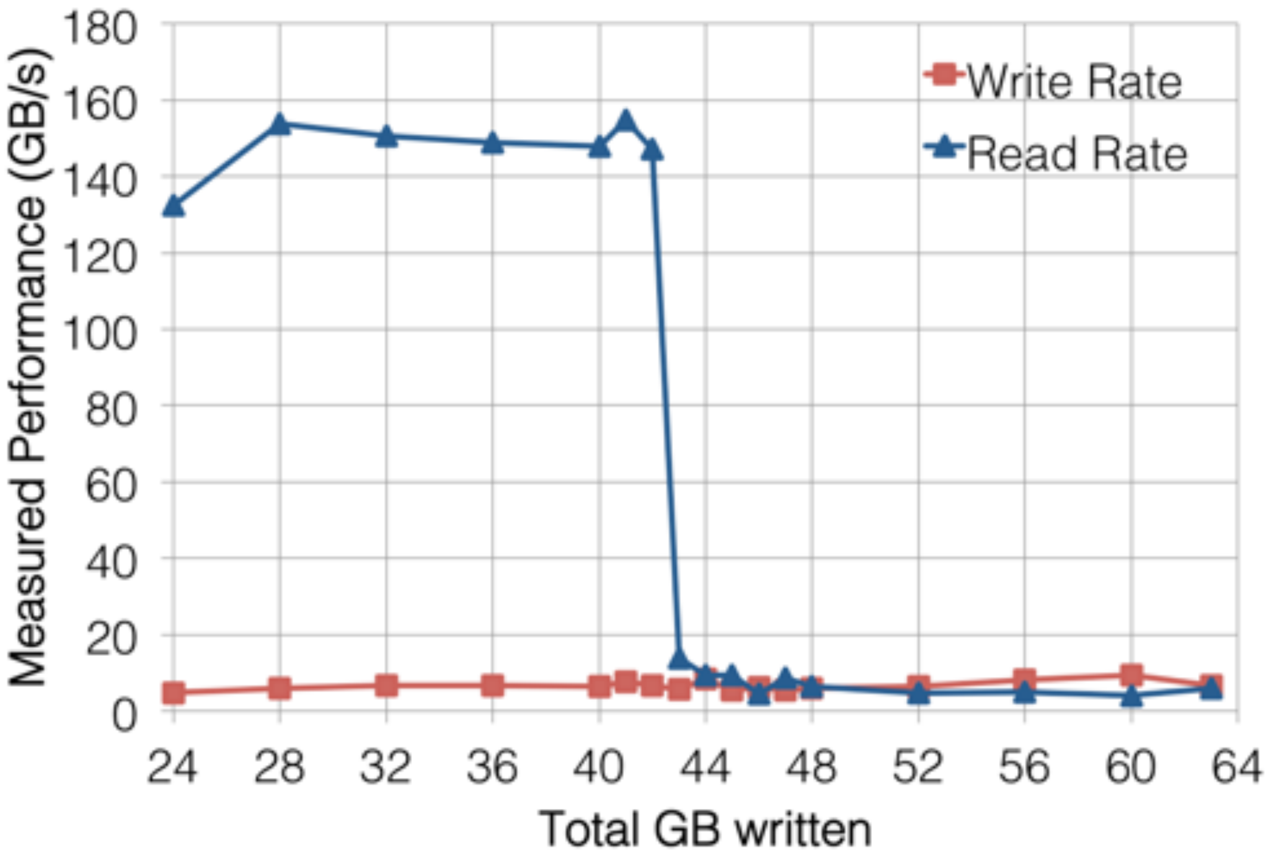
# Try breaking up output into multiple files

- IOR provides **-F** option to make each rank read/write to its own file
  - Reduces lock contention within file
  - Can cause significant metadata load at scale
- Problem: 250 GB/sec from 4 nodes is faster than light

```
$ mpirun -n 64 ./ior -t 1m -b 1m -s 256 -F
...
Max Write: 13887.92 MiB/sec (14562.54 MB/sec)
Max Read:  259730.43 MiB/sec (272347.09 MB/sec)
```

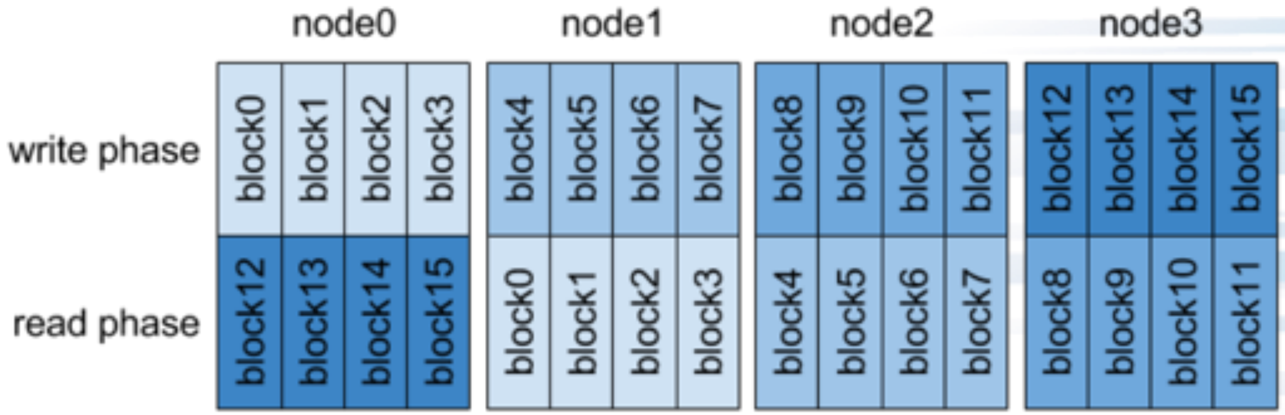
# Effect of page cache when measuring I/O bandwidth

- Uses unused compute node memory to store file contents
  - Write-back for small I/O performance
  - Read cache for re-read data
- Easy to accidentally measure memory bandwidth, not file system bandwidth!



# Avoid reading back what you just wrote using shifts

- IOR provides `-C` to shift MPI ranks by one node before reading back
- Read performance looks reasonable
- But what about write cache?




```
$ mpirun -n 64 ./ior -t 1m -b 1m -s 256 -F –C
...
Max Write: 13398.16 MiB/sec (14048.98 MB/sec)
Max Read:  6950.81 MiB/sec (7288.45 MB/sec)
```



Argonne NATIONAL LABORATORY
EXASCALE COMPUTING PROJECT

# Flushing write-back cache to include time-to-persistence

- IOR provides `-e` option to force fsync(2) and write back all dirty pages
- Measures time to write data to durable media—not just page cache
- Without fsync, close(2) operation may include hidden sync time

```
$ mpirun -n 64 ./ior -t 1m -b 1m -s 256 -F -C -e
...
Max Write: 12289.16 MiB/sec (12886.11 MB/sec)
Max Read:   6274.31 MiB/sec (6579.09 MB/sec)
```



Argonne NATIONAL LABORATORY

ECP EXASCALE COMPUTING PROJECT

# How well does single-file I/O perform now?

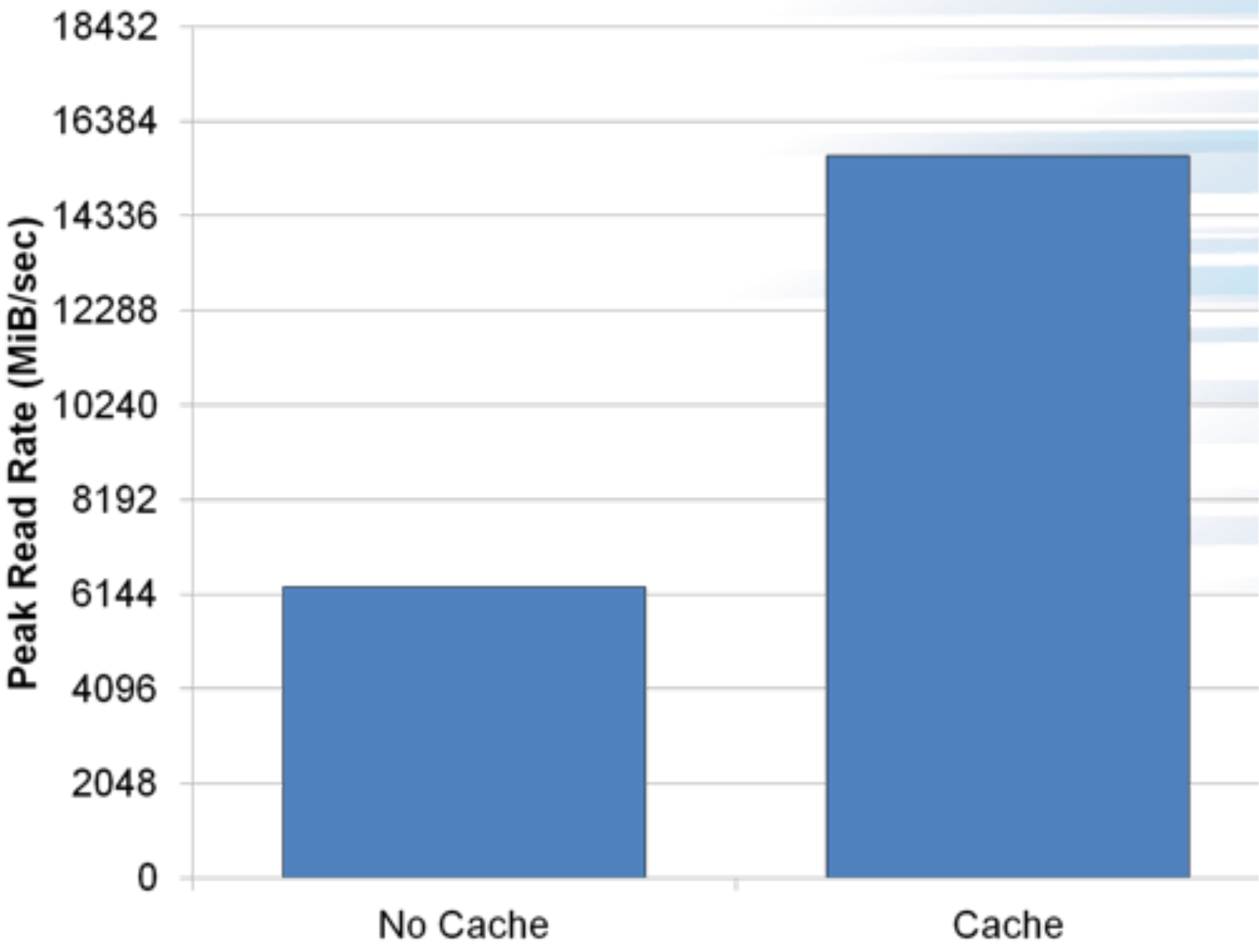
```
$ mpirun -n 64 ./ior -t 1m -b 1m -s 256 -C -e
...
Max Write: 455.49 MiB/sec (477.62 MB/sec)
Max Read:  2317.75 MiB/sec (2430.33 MB/sec)
```

**Don't forget to set Lustre striping!**

```
$ lfs setstripe -c 8 .
$ mpirun -n 64 ./ior -t 1m -b 1m -s 256 -C -e
...
Max Write: 2234.91 MiB/sec (2343.47 MB/sec)
Max Read:  4451.02 MiB/sec (4667.24 MB/sec)
```

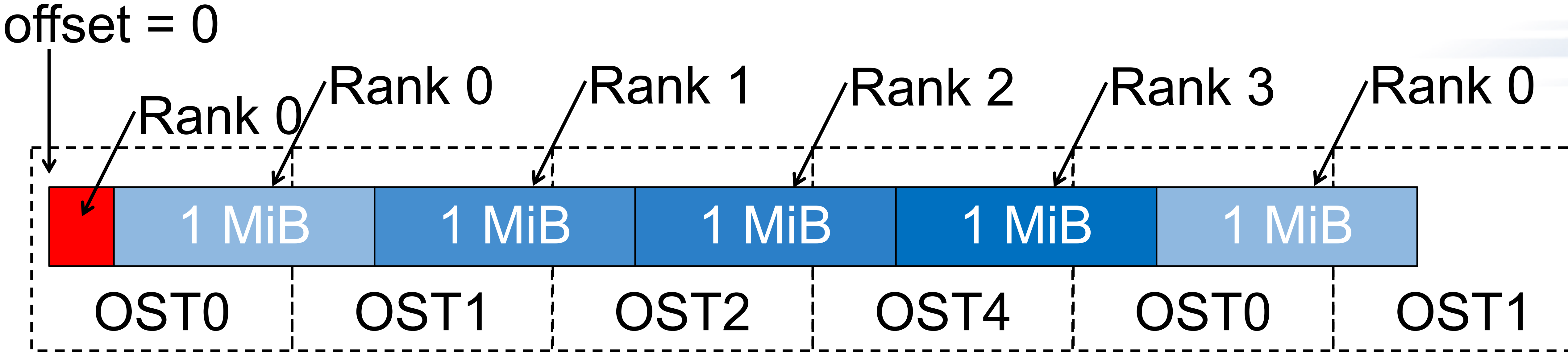
# Corollaries of I/O benchmarking

- **Understand cache effects**
  - can correct I/O problems at small scale
  - tends not to behave well at scale
- **Utilize caches *when it makes sense***
  - small but sequential writes
  - read then re-read
    (e.g., BLAST bioinformatics app)



# Effect of misaligned I/O

- Common to write a small, fixed-size header and big blobs of data
- Does not map well to parallel file systems

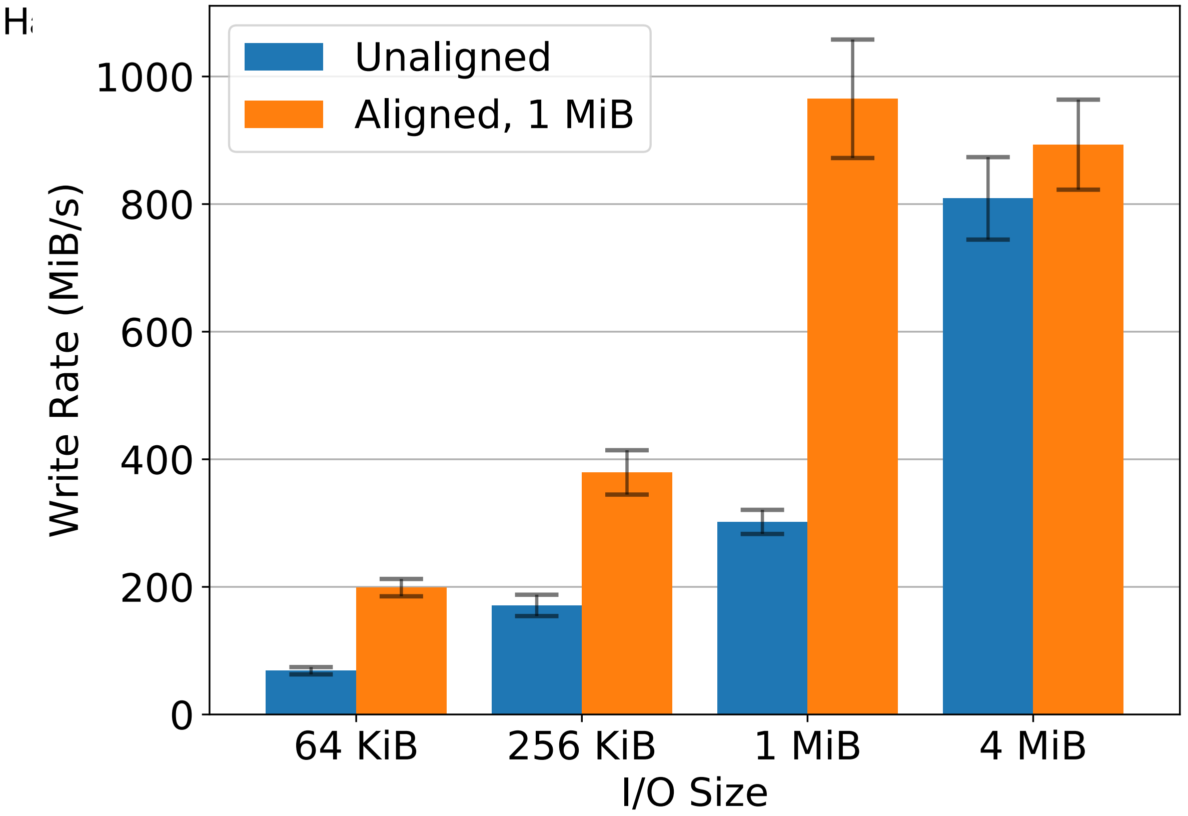


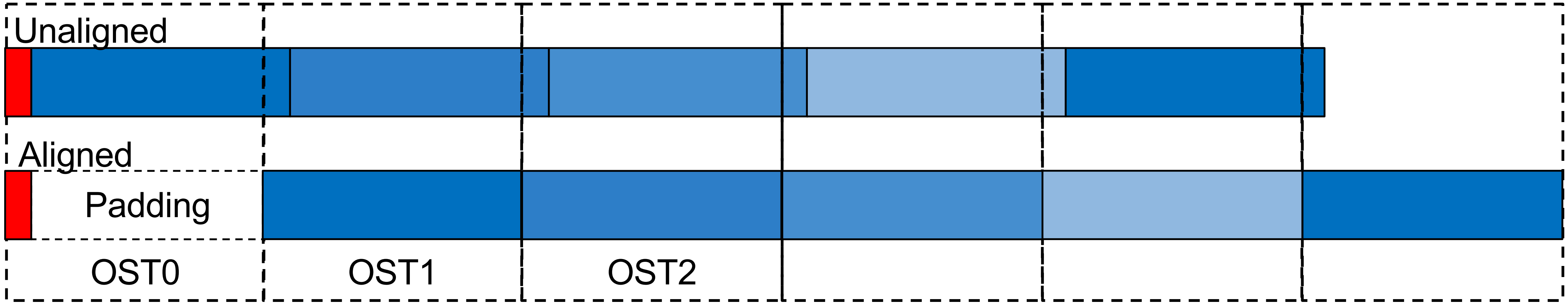



Argonne NATIONAL LABORATORY
ECP EXASCALE COMPUTING PROJECT

# Effect of misaligned I/O

- Parallel HDF5 and PnetCDF allow you to adjust alignment
  - `nc_header_align_size` and `nc_var_align_size` hints
  - `H5Pset_alignment` operation
- Consult user docs for best alignment



# Effect of misaligned I/O

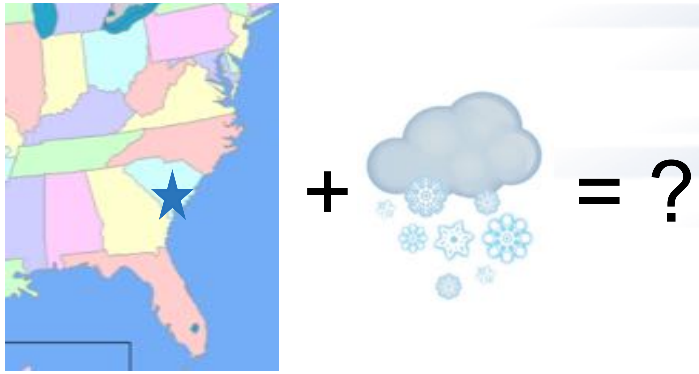
```
$ mpirun -n 128 ./ior -t 1m -b 1m -s 256 -a HDF5 -w
...
Max Write: 785.43 MiB/sec (823.58 MB/sec)

$ mpirun -n 128 ./ior -t 1m -b 1m -s 256 \
                          -a HDF5 -w -O setAlignment=1M
...
Max Write: 1318.53 MiB/sec (1382.58 MB/sec)
```

# Understanding I/O Behavior and Performance in Complex Storage Systems

# "I observed performance XYZ.  Now what?"

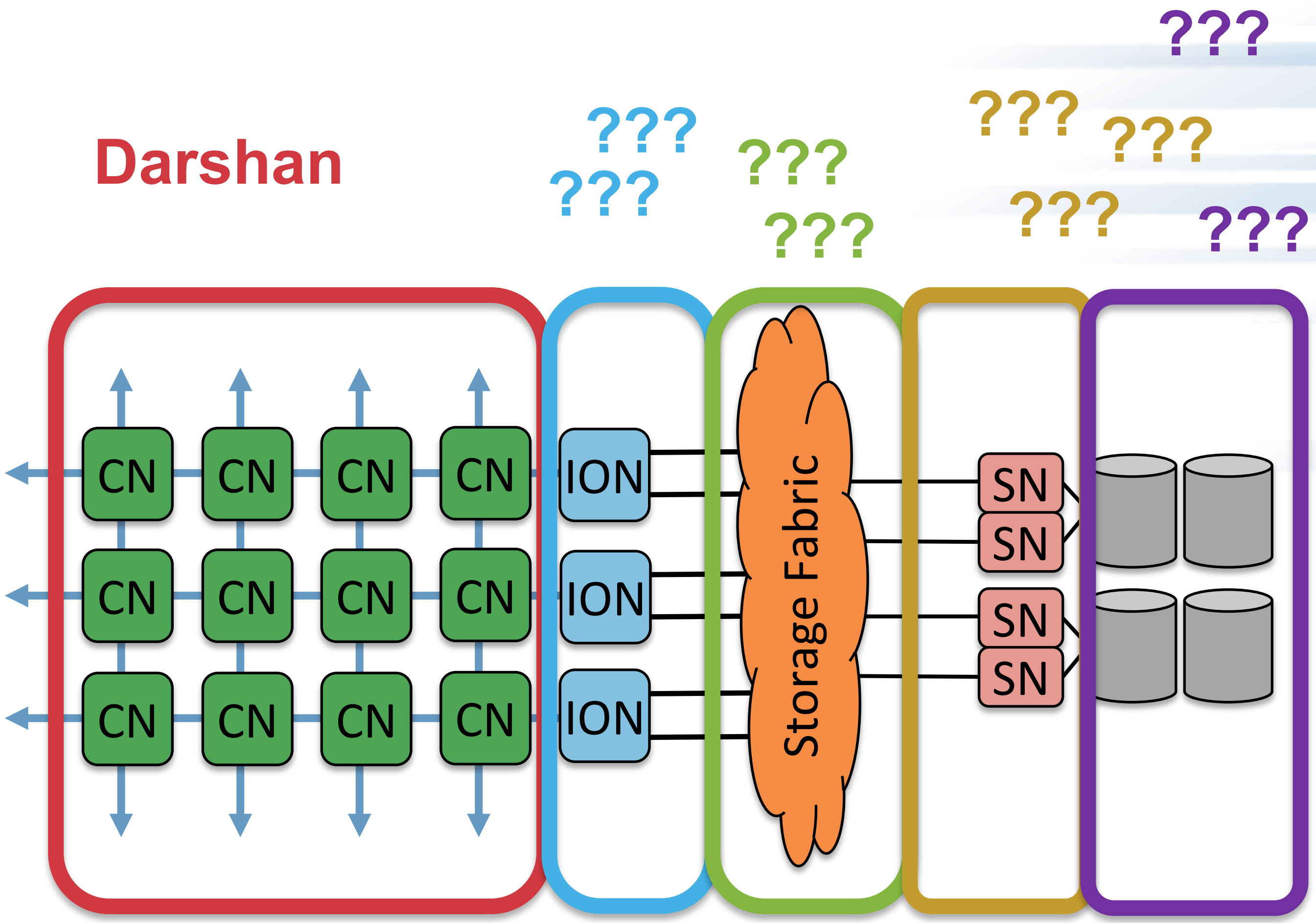
- A climate vs. weather analogy: It is snowing outside. Is that normal?
- You need context to know:
  - Does it ever snow there?
  - What time of year is it?
  - What was the temperature yesterday?
  - Do your neighbors see snow too?
  - Should you look at it first hand?




- It is similarly difficult to understand a single application performance measurement without broader context: How do we differentiate typical I/O climate from extreme I/O weather events?

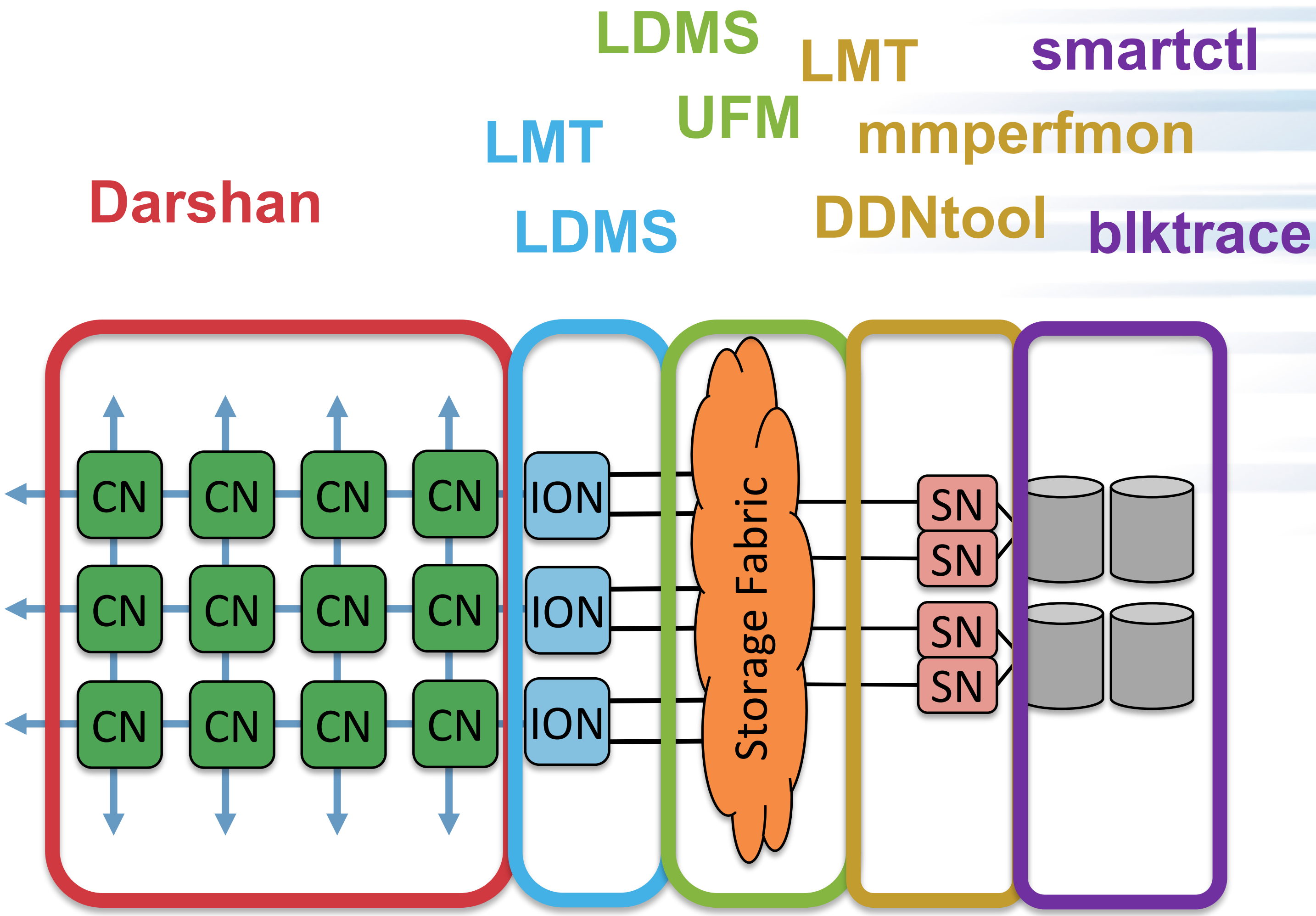
# "I observed performance XYZ.  Now what?"

- If Darshan doesn't flag any problems, what next?



# "I observed performance XYZ. Now what?"

- If Darshan doesn't flag any problems, what next?
- Need a big picture view
- Many component-level tools
- Each uses its own data formats, resolutions, and scopes

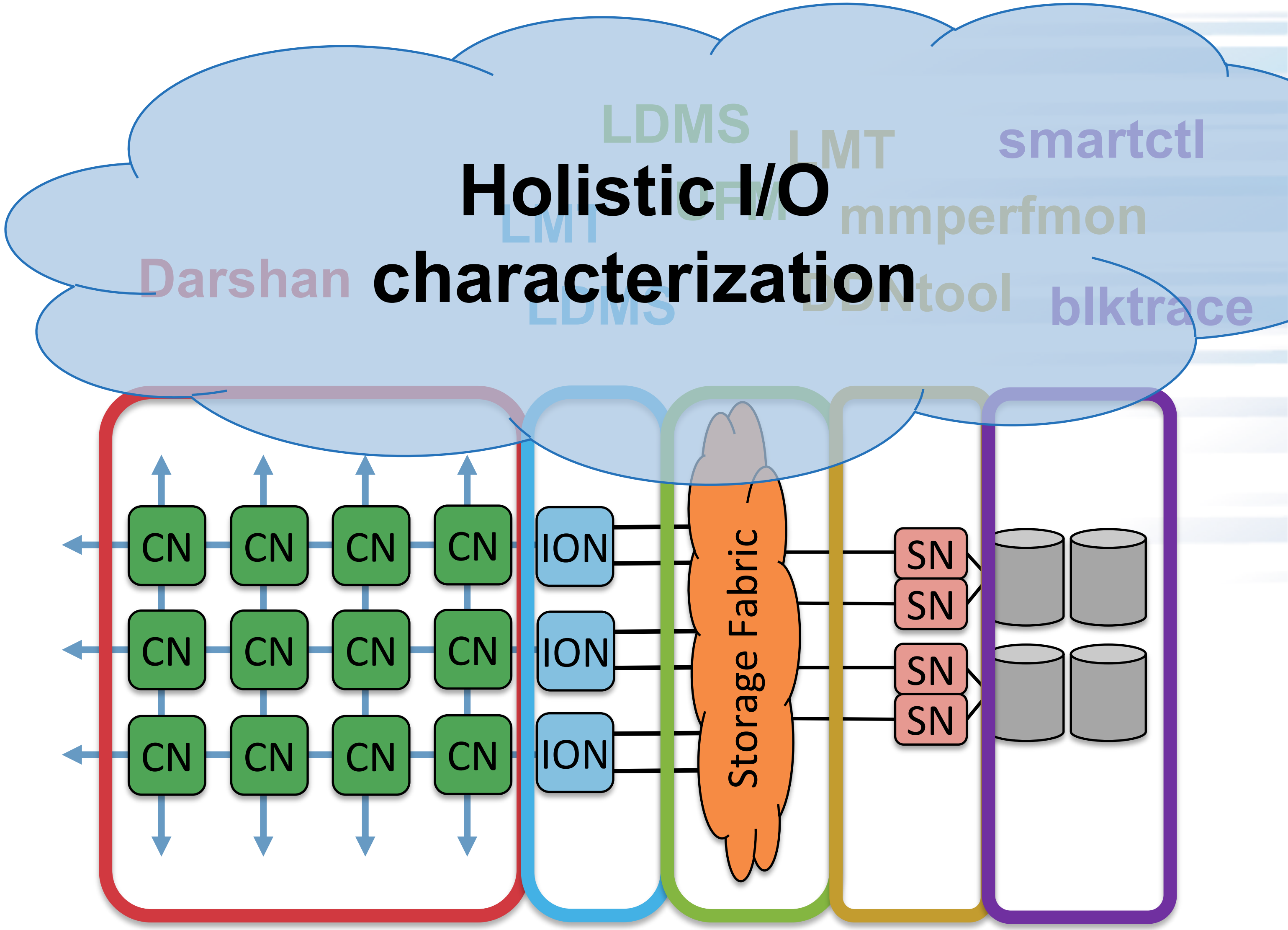





Argonne NATIONAL LABORATORY
ECP EXASCALE COMPUTING PROJECT

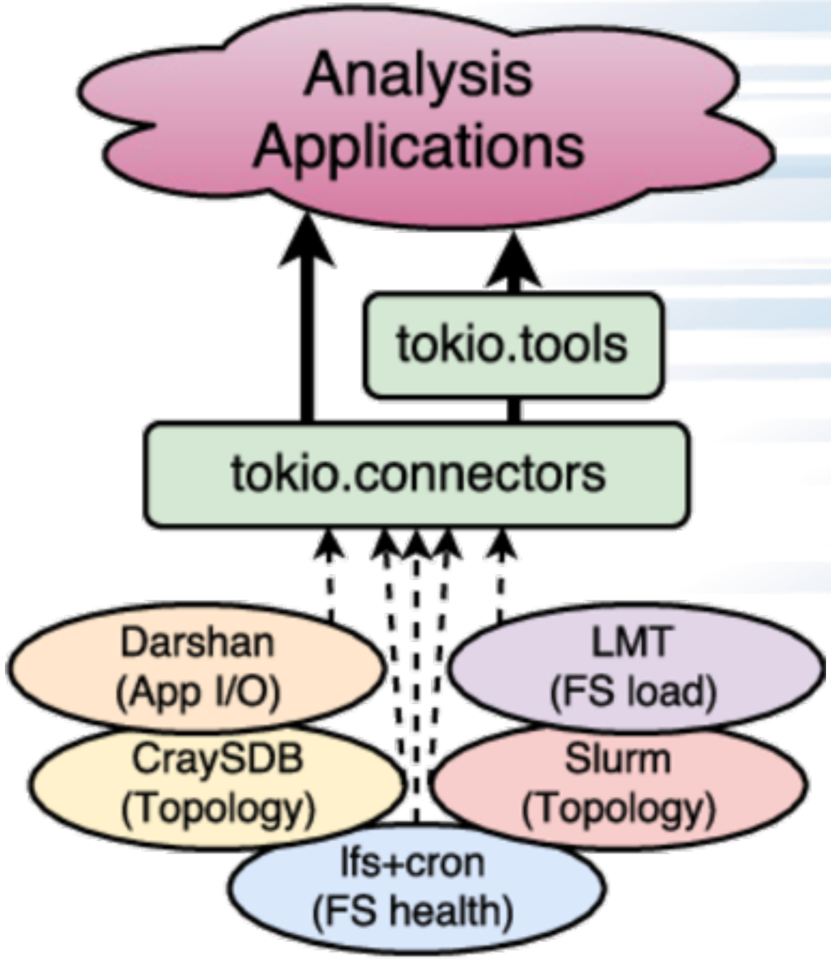
# "I observed performance XYZ.  Now what?"

- Total Knowledge of I/O (TOKIO) designed to address this
- Integrate, correlate, and analyze I/O behavior from the system as a whole for holistic understanding



# TOKIO approach to I/O performance analysis

- Integrate existing instrumentation tools
- Index data sources in their native format
  - Tools to align and link data sets
  - Connectors to produce coherent views on demand
- Develop analysis methods that integrate data
- Produce tools that share a common interface and data format




https://www.nersc.gov/research-and-development/tokio/

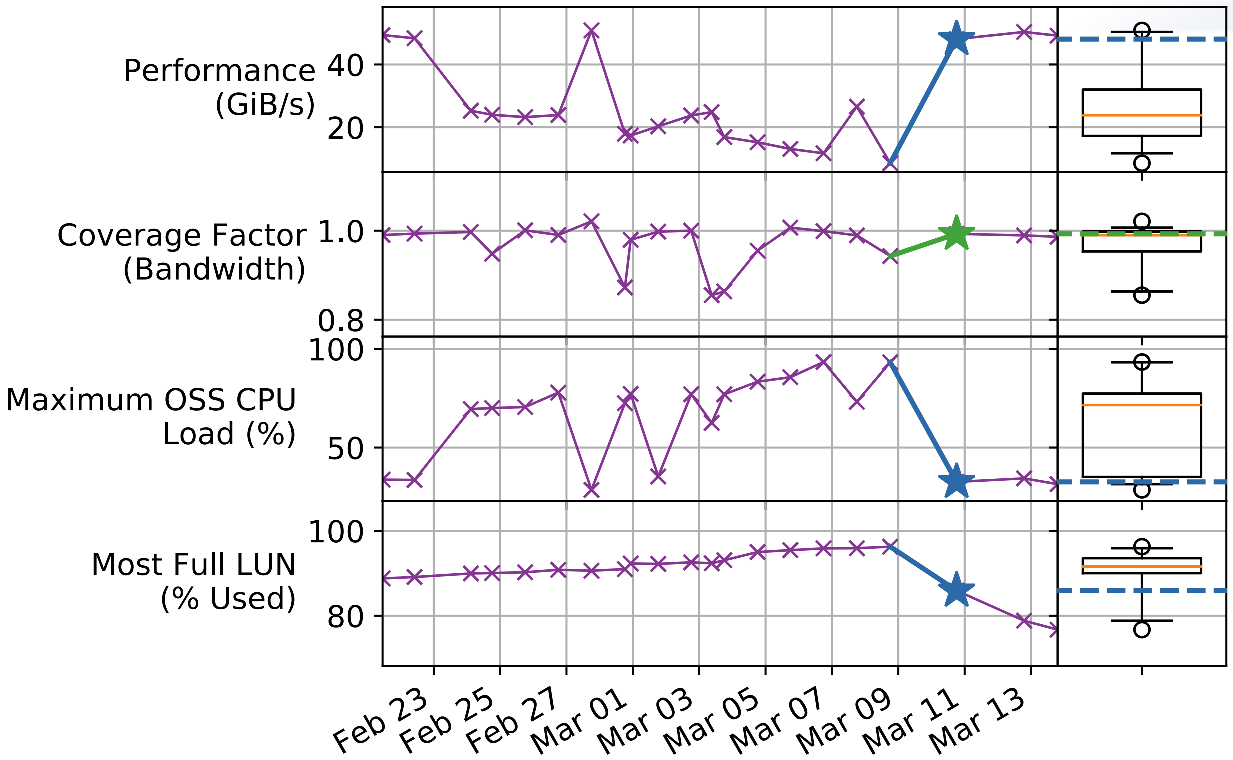
# Example: Unified Monitoring and Metrics Interface (UMAMI)

- Pluggable dashboard that displays the I/O performance of an app in context of
- other system telemetry
- historical records

Historical samples (for a given application) are plotted over time

Each metric is shown in a separate row

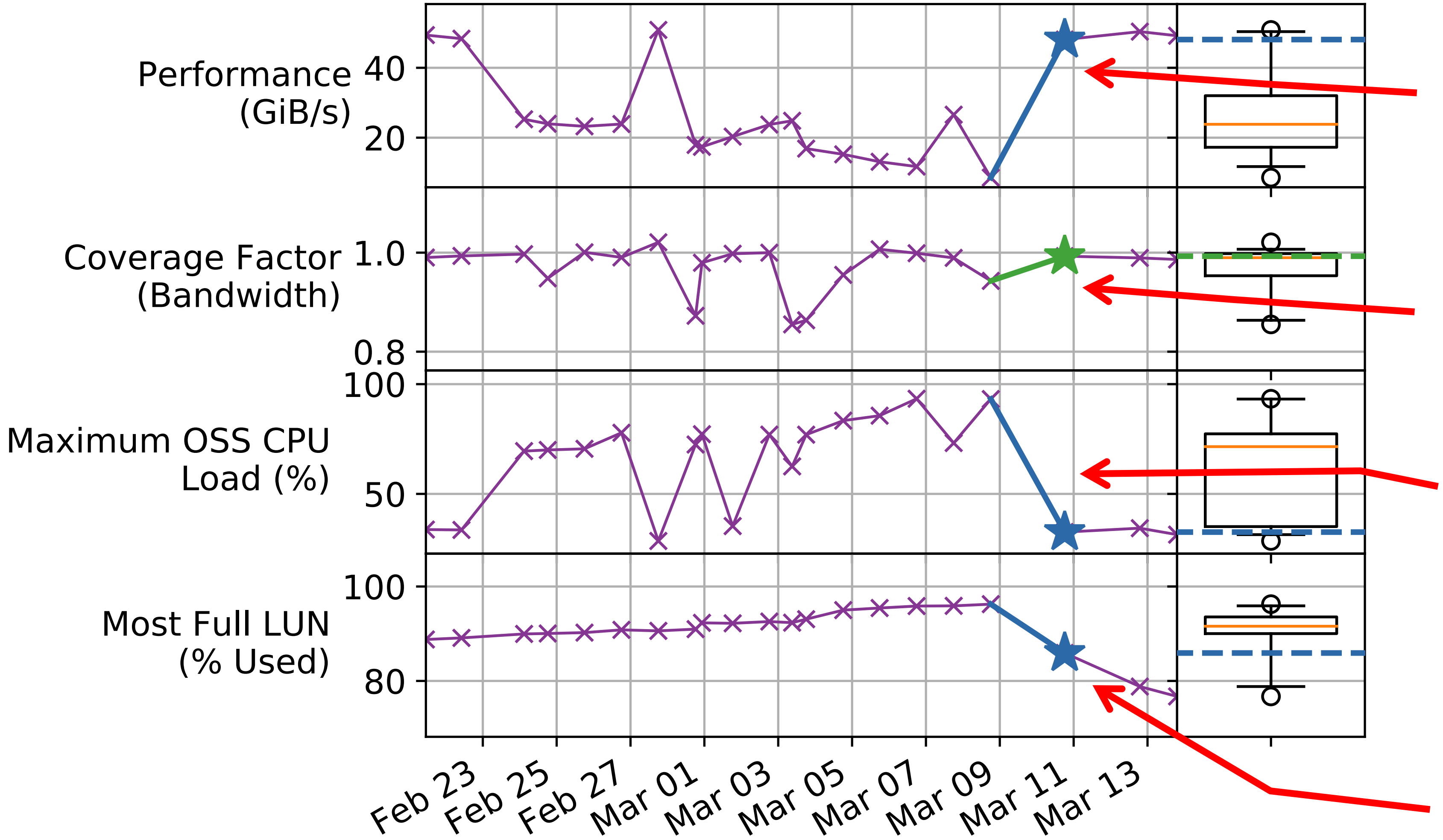
Box plots relate current values to overall variance






Argonne NATIONAL LABORATORY
ECP EXASCALE COMPUTING PROJECT

# UMAMI Example: What Made Performance Increase?




Performance for job is 2.5x higher than usual

System background load is typical

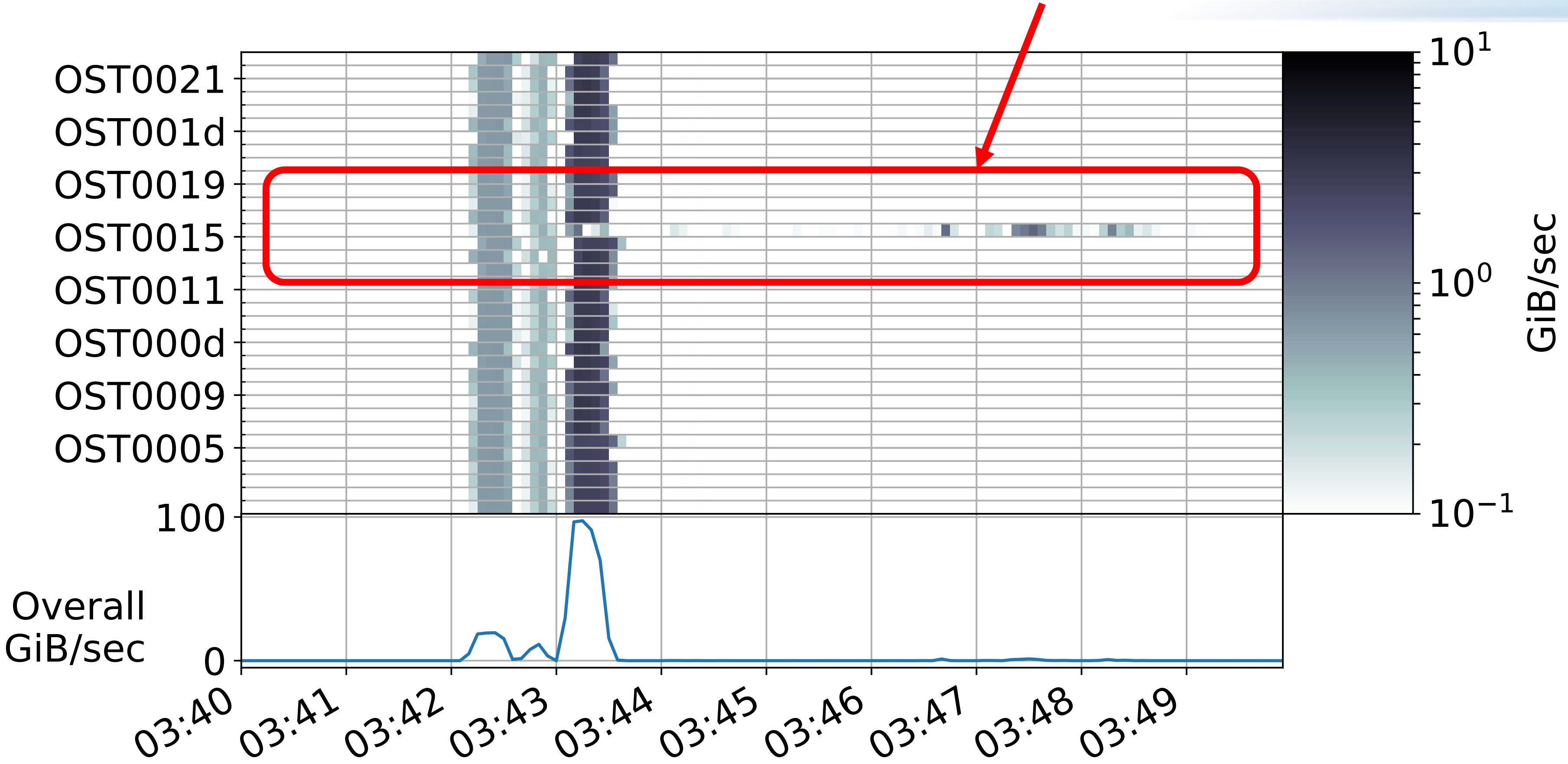
Server CPU load is low after long-term steady climb

Corresponds to data purge that freed up disk blocks

# Example: Identifying Straggling OSTs

**TOKIO utility to generate heatmaps of parallel file system servers**

OST0015 didn't finish writing until 3:49 and caused 3x slowdown!




Most servers wrote all data between 3:42 and 3:44 at ~100 GiB/sec

# Example: TOKIO darshan_bad_ost to find stragglers

- Combine file:OST mappings with file:performance mappings in Darshan logs
- Correlate slow files with slow OSTs over any number of Darshan logs

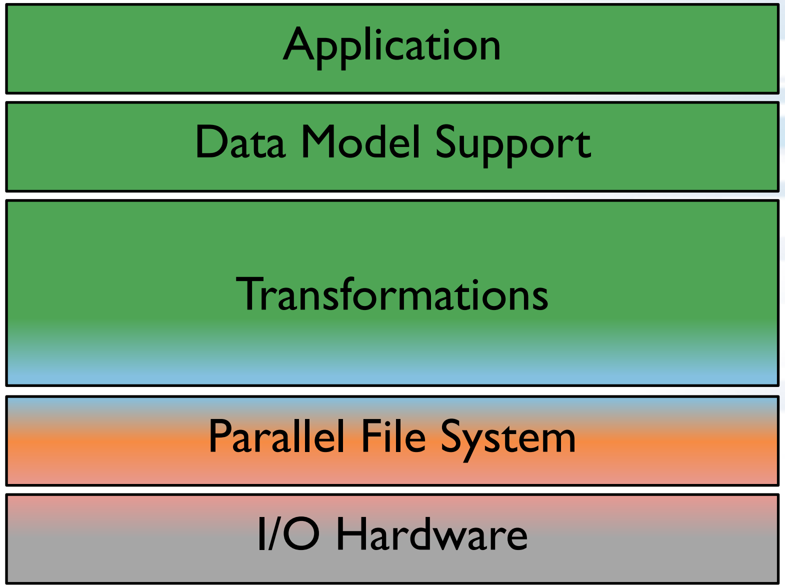
```
$ darshan_bad_ost ./glock_ior_id1149144_4-9-65382-13265564257186991065_1.darshan

OST Name     Correlation     P-Value
OST#21             -0.703 2.473e-305 <-- this OST looks unhealthy
OST#8              -0.102  4.039e-06
OST#7              -0.067    0.00246
...
```

https://pytokio.readthedocs.io/en/latest/api/tokio.cli.darshan_bad_ost.html

# I/O understanding in complex systems takeaway

- Complex storage architectures require an holistic approach to performance analysis
  - I/O problems sometimes out of users' control!
  - More storage tiers, more things can go wrong
- Software stacks to address this complexity are emerging
  - Darshan for application and runtime library
  - Implementation-specific tools for file system and storage devices
  - TOKIO to connect everything together
- For more information, see:
  - https://www.mcs.anl.gov/research/projects/darshan/
  - https://www.nersc.gov/research-and-development/tokio/



# Thank you!