ARGONNE
ATPESC2023
EXTREME-SCALE COMPUTING

How to Understand and Tune HPC I/O Performance

Shane Snyder
ssnyder@mcs.anl.gov
Argonne National Laboratory

August 10, 2023

Surveying the HPC I/O landscape

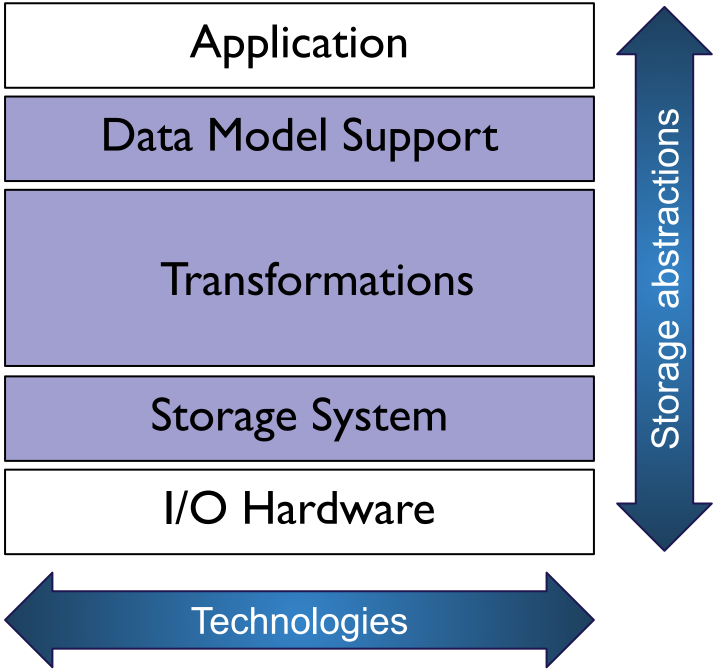
A complex data management ecosystem

As evidenced by today's presentations, the HPC I/O landscape is deep and vast

- High-level data abstractions: HDF5, PnetCDF
- I/O middleware: MPI-IO
- Storage systems: Lustre, GPFS, DAOS
- Storage hardware: HDDs, SSDs, SCM

HPC applications themselves are evolving and encountering new data management challenges

Understanding I/O behavior in this environment is difficult, much less turning observations into actionable I/O tuning decisions

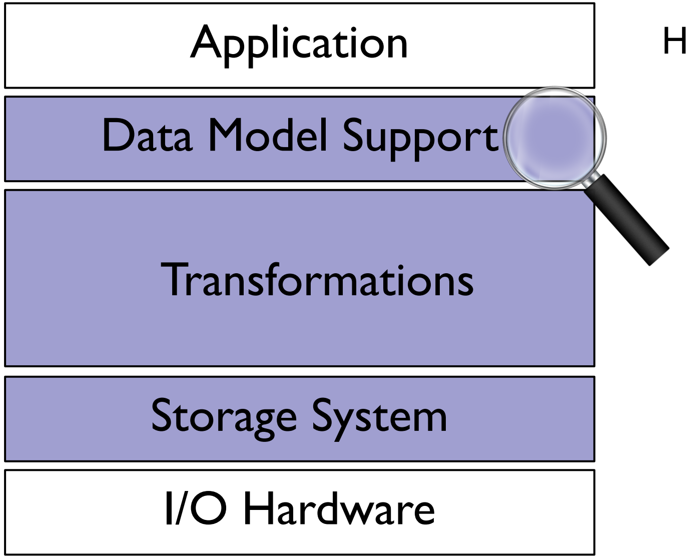


Characterizing HPC I/O workloads with Darshan

A look under the hood of an HPC application

You have already heard some basics about Darshan, a powerful tool for users to better understand and tune their I/O workloads

Darshan provides many helpful stats across multiple layers of the I/O stack that are critical to understanding application I/O behavior



HDF5 stats*:

- Accessed files/datasets
- Operation counts
- Total read/write volumes
- Common access info (including details of hyperslab accesses)
- Chunking parameters
- Dataset dimensionality and size
- MPI-IO usage
- I/O timing

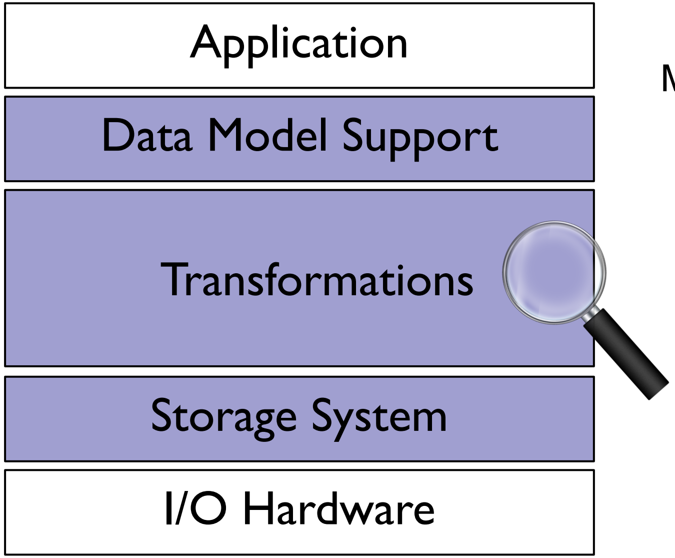
*Note: HDF5 instrumentation is not typically enabled for facility Darshan installs – you will need to install this version yourself

Characterizing HPC I/O workloads with Darshan

A look under the hood of an HPC application

You have already heard some basics about Darshan, a powerful tool for users to better understand and tune their I/O workloads

Darshan provides many helpful stats across multiple layers of the I/O stack that are critical to understanding application I/O behavior



MPI-IO stats:

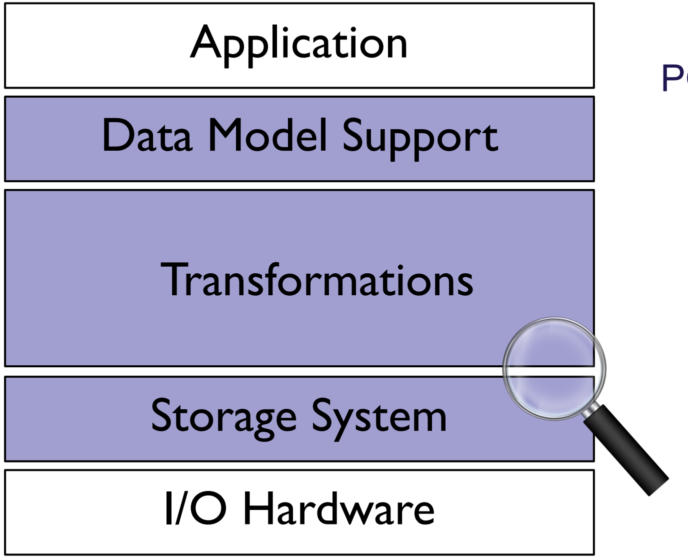
- Operation counts (open, read, write, sync)
- Collective and independent I/O usage
- Total read/write volumes
- Access size info
  - Common values
  - Histograms
- I/O timing

Characterizing HPC I/O workloads with Darshan

A look under the hood of an HPC application

You have already heard some basics about Darshan, a powerful tool for users to better understand and tune their I/O workloads

Darshan provides many helpful stats across multiple layers of the I/O stack that are critical to understanding application I/O behavior



POSIX stats:

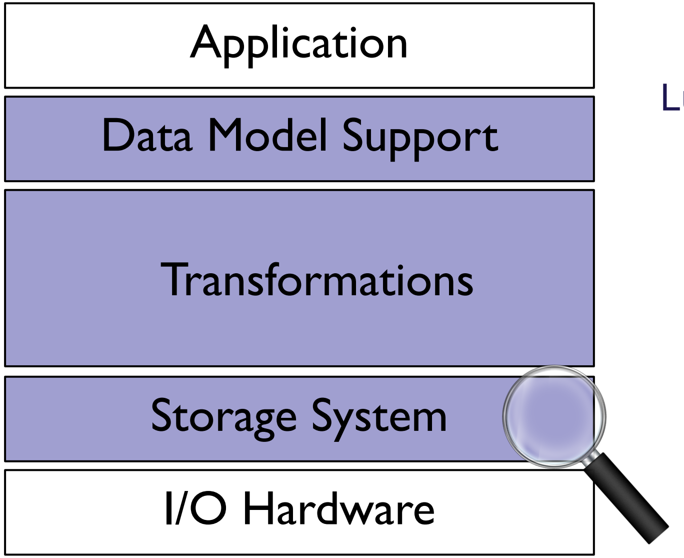
- Operation counts (open, read, write, seek, stat)
- Total read/write volumes
- File alignment
- Access size/stride info
  - Common values
  - Histograms
- I/O timing

Characterizing HPC I/O workloads with Darshan

A look under the hood of an HPC application

You have already heard some basics about Darshan, a powerful tool for users to better understand and tune their I/O workloads

Darshan provides many helpful stats across multiple layers of the I/O stack that are critical to understanding application I/O behavior



Lustre stats:

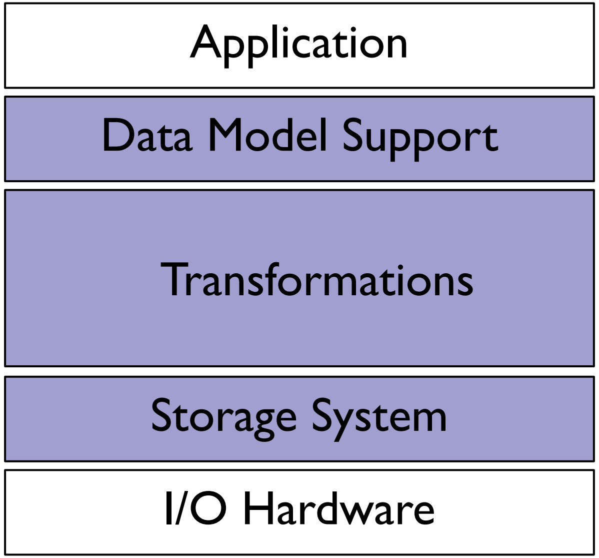
- Data server (OST) and metadata server (MDT) counts
- Stripe size/width
- OST list serving a file

Characterizing HPC I/O workloads with Darshan

A look under the hood of an HPC application

You have already heard some basics about Darshan, a powerful tool for users to better understand and tune their I/O workloads

Darshan provides many helpful stats across multiple layers of the I/O stack that are critical to understanding application I/O behavior



Let's see how Darshan can be leveraged in some practical use cases that demonstrate some general best practices in tuning HPC I/O performance

Tuning the storage system

Ensuring storage resources match application I/O needs

For some parallel file systems like Lustre, users have direct control over file striping parameters

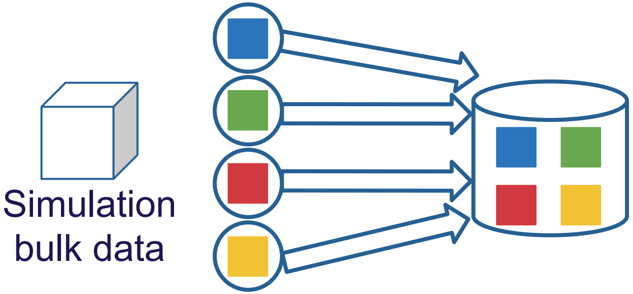
Bad news: Users may have to have some knowledge of the file system to get good I/O performance

Good news: Users can often get higher I/O performance than system defaults with thoughtful tuning -- file systems aren't perfect for every workload!

Tuning the storage system

Ensuring storage resources match application I/O needs

Tuning decisions can and should be made independently for different file types



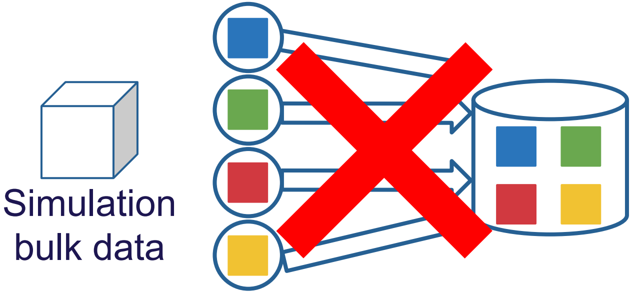
Simulation clients write
data to 1 storage server

Tuning the storage system

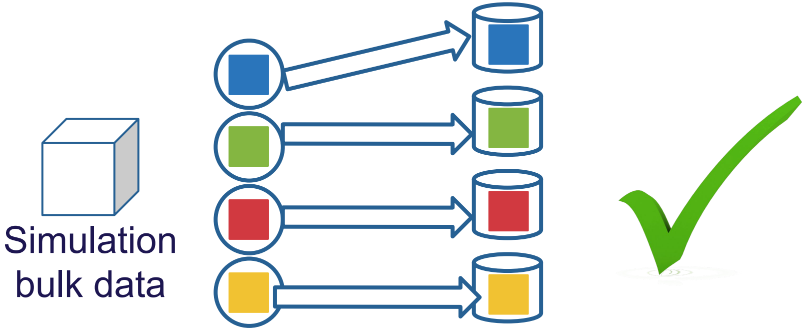
Ensuring storage resources match application I/O needs

Tuning decisions can and should be made independently for different file types

Large application datasets should ideally be distributed across as many storage resources as possible



Simulation clients write data to 1 storage server



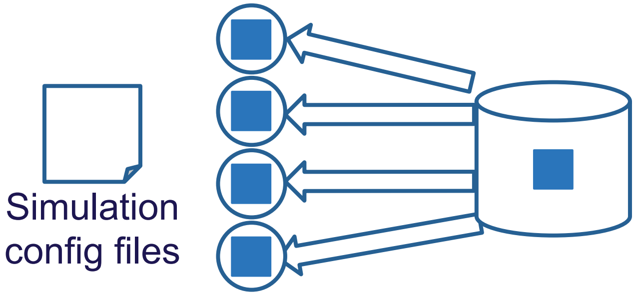
Simulation clients load balance writes across multiple servers

Tuning the storage system

Ensuring storage resources match application I/O needs

Tuning decisions can and should be made independently for different file types

On the other hand, smaller files often benefit from being stored on a single server



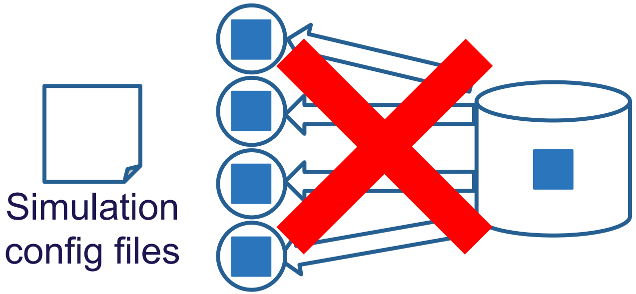
Simulation clients read config data from 1 storage server

Tuning the storage system

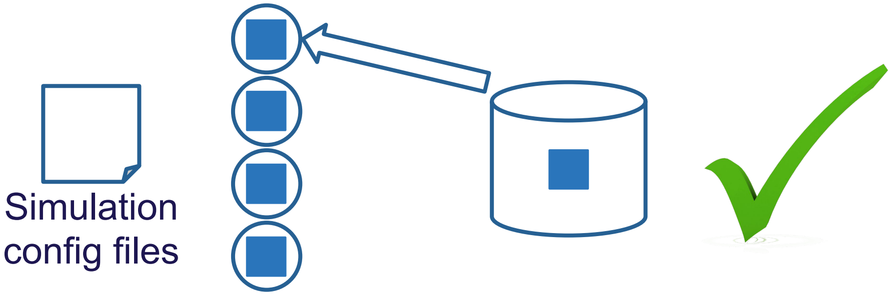
Ensuring storage resources match application I/O needs

Tuning decisions can and should be made independently for different file types

On the other hand, smaller files often benefit from being stored on a single server



Simulation clients read config data from 1 storage server



Better yet, limit storage contention by having 1 client read data and distribute using communication (e.g., MPI)

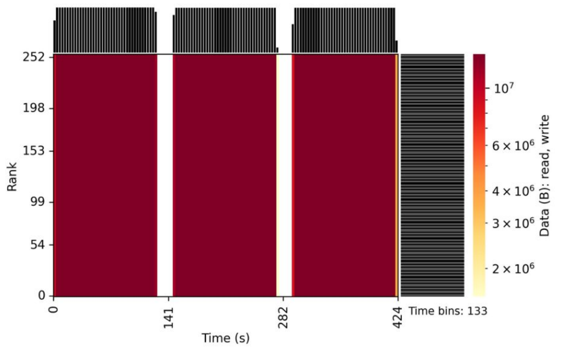
Tuning the storage system

Ensuring storage resources match application I/O needs

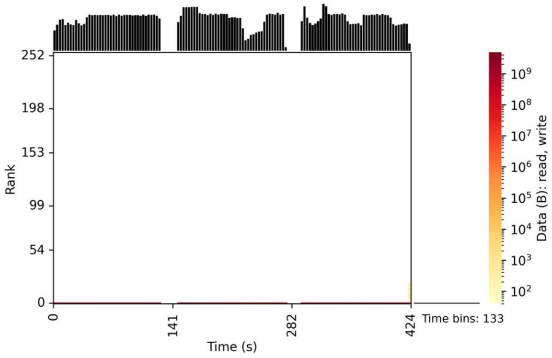
Be aware of what file system settings are available to you and don't assume system defaults are always the best… you might be surprised what you find

- ALCF Polaris/Theta and NERSC Perlmutter Lustre scratch file systems both have a default stripe width of 1 (i.e., files are stored on one server by default)

MPI-IO



POSIX



256 process (4 node) h5bench[1] runs on NERSC Perlmutter

h5bench contains lots of parameters for controlling characteristics of generated HDF5 workloads.

1. https://github.com/hpc-io/h5bench

Hands on exercises: https://github.com/radix-io/hands-on

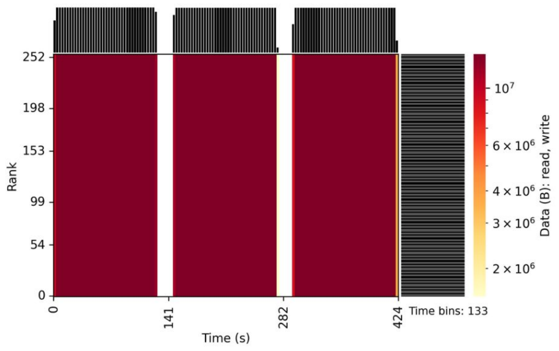
Tuning the storage system

Ensuring storage resources match application I/O needs

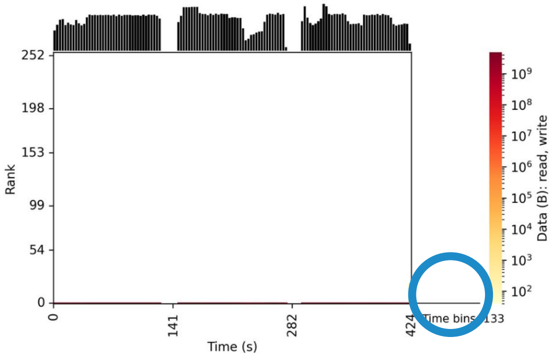
Be aware of what file system settings are available to you and don't assume system defaults are always the best… you might be surprised what you find

- ALCF Polaris/Theta and NERSC Perlmutter Lustre scratch file systems both have a default stripe width of 1 (i.e., files are stored on one server by default)

MPI-IO



POSIX



All I/O funneled through rank 0

MPI-IO collective I/O driver for Lustre assigns dedicated aggregators for each stripe, yielding a single aggregator for files of 1 stripe

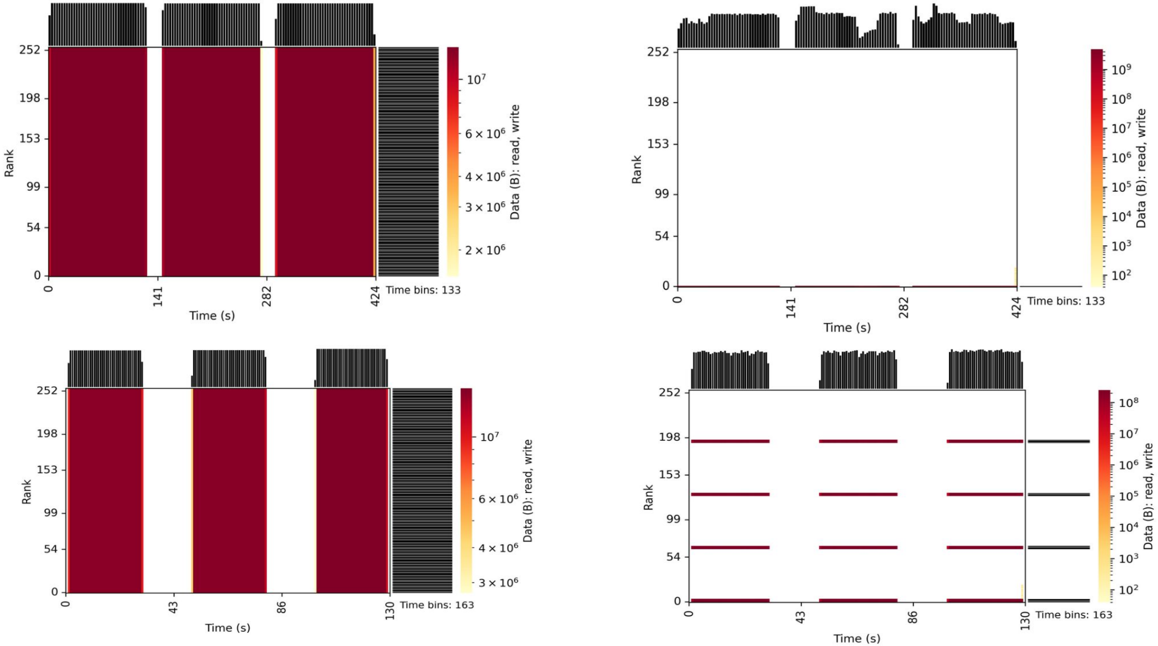
Tuning the storage system

Ensuring storage resources match application I/O needs

MPI-IO | POSIX

1 stripe

16 stripes



Manually setting the stripe width to 16 yields more I/O aggregators and better performance:

```
> lfs setstripe -c 16 testFile
```


Hands on exercises: https://github.com/radix-io/hands-on

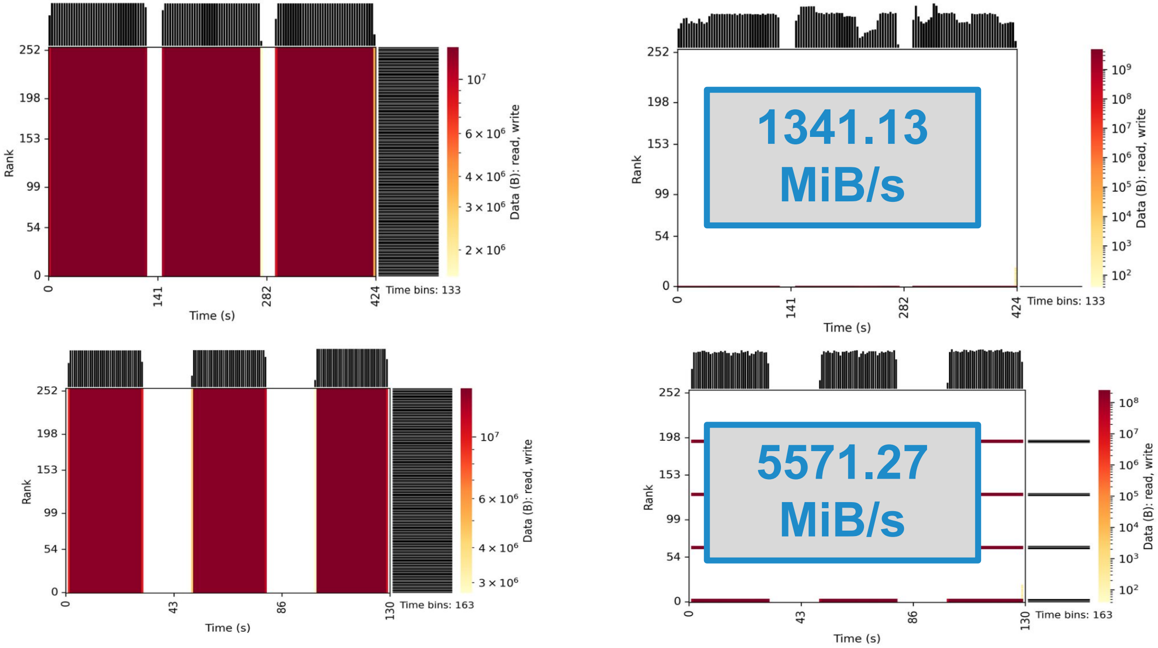
Tuning the storage system

Ensuring storage resources match application I/O needs

MPI-IO **POSIX**

1 stripe

16 stripes



Manually setting the stripe width to 16 yields more I/O aggregators and better performance:

```
> lfs setstripe -c 16 testFile
```

4x performance improvement!

Hands on exercises: https://github.com/radix-io/hands-on

Tuning the storage system

Ensuring storage resources match application I/O needs

Consult facilities documentation for established best practice!

	Single Shared-File I/O	File per Process
File size	Command	Command
< 1 GB	keep default striping	keep default striping
1 - 10 GB	`stripe_small`	keep default striping
10 - 100 GB	`stripe_medium`	keep default striping
100 GB - 1 TB	`stripe_large`	keep default striping
> 1 TB	`stripe_large`	`stripe_large`

Perlmutter (NERSC) docs providing commands to set stripe params for various file types

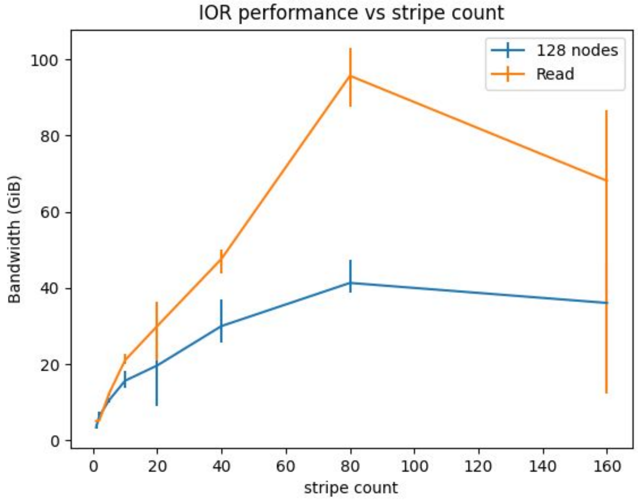
- The default striping set on Orion is targeted to work well for a variety of workloads
- In most cases, users should use this default striping. Though possible, manual striping should only occur after careful consideration and under collaboration with OLCF staff
- The default striping policy may change due to findings in production

OLCF presentation on Orion storage system detailing usage of Lustre's new progressive file layout mechanism

Tuning the storage system

Ensuring storage resources match application I/O needs

Consult facilities documentation for established best practice! Sometimes you may even need to experiment yourself.



128-node example of the IOR benchmark using various stripe counts on ALCF Polaris.

For more I/O intensive programs, it's typically better to err on the side of more storage servers. The following command stripes across all servers:

```
> lfs setstripe -c -1 testFile
```

https://github.com/radix-io/io-sleuthing/tree/main/examples/striping

Tuning low-level (POSIX) file I/O

Making efficient use of a no-frills I/O API

Users may also need to pay close attention to file system alignment when issuing I/O accesses to a file

- Accesses that are not aligned can introduce performance inefficiencies on file systems

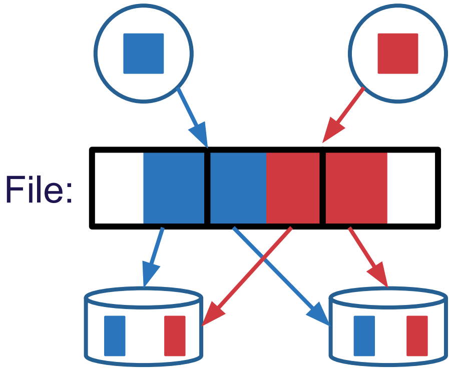
Tuning low-level (POSIX) file I/O

Making efficient use of a no-frills I/O API

Users may also need to pay close attention to file system alignment when issuing I/O accesses to a file

- Accesses that are not aligned can introduce performance inefficiencies on file systems

For Lustre, performance can be maximized by aligning I/O to stripe boundaries:



Unaligned I/O requests can span multiple servers and introduce inefficiencies in storage protocols

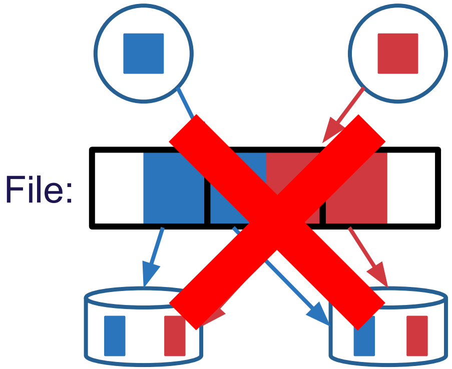
Tuning low-level (POSIX) file I/O

Making efficient use of a no-frills I/O API

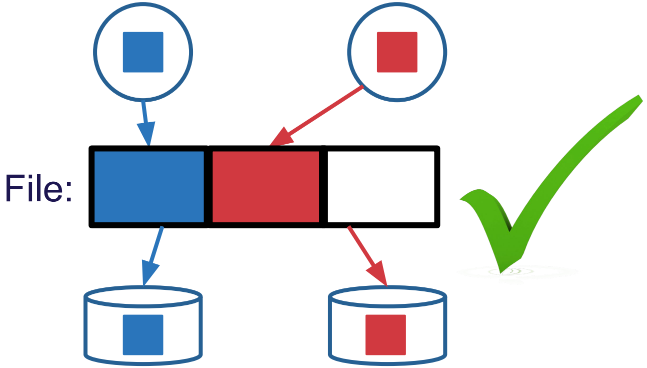
Users may also need to pay close attention to file system alignment when issuing I/O accesses to a file

- Accesses that are not aligned can introduce performance inefficiencies on file systems

For Lustre, performance can be maximized by aligning I/O to stripe boundaries:



Instead, ensure client accesses are well-aligned to avoid Lustre server contention



Tuning low-level (POSIX) file I/O

Making efficient use of a no-frills I/O API

Consider a simple 10-process (10-node) NERSC Cori example where processes write in an interleaved fashion to a single shared file

aligned

# Module	Rank	Wt/Rd	Segment	Offset	Length	Start(s)	End(s)	[OST]
X_POSIX	0	write	0	0	1048576	0.0054	0.0066	[197]
X_POSIX	0	write	1	10485760	1048576	0.0066	0.0073	[197]
X_POSIX	0	write	2	20971520	1048576	0.0073	0.0081	[197]
X_POSIX	0	write	3	31457280	1048576	0.0081	0.0088	[197]

Use Darshan's DXT tracing module to get details about each individual write access – **more details on DXT usage coming soon**

Tuning low-level (POSIX) file I/O

Making efficient use of a no-frills I/O API

Consider a simple 10-process (10-node) NERSC Cori example where processes write in an interleaved fashion to a single shared file

aligned

# Module	Rank	Wt/Rd	Segment	Offset	Length	Start(s)	End(s)	[OST]
X_POSIX	0	write	0	0	1048576	0.0054	0.0066	[197]
X_POSIX	0	write	1	10485760	1048576	0.0066	0.0073	[197]
X_POSIX	0	write	2	20971520	1048576	0.0073	0.0081	[197]
X_POSIX	0	write	3	31457280	1048576	0.0081	0.0088	[197]

Each access is aligned to the Lustre stripe size (1 MiB)

Each process interacts with a single Lustre server (OST)

Tuning low-level (POSIX) file I/O

Making efficient use of a no-frills I/O API

Consider a simple 10-process (10-node) NERSC Cori example where processes write in an interleaved fashion to a single shared file

unaligned

```
# Module      Rank  Wt/Rd   Segment        Offset       Length    Start(s)       End(s)  [OST]
 X_POSIX        0   write         0        524288      1048576      0.0065       0.0594  [ 32]  [197]
 X_POSIX        0   write         1      11010048      1048576      0.0594       0.1268  [ 32]  [197]
 X_POSIX        0   write         2      21495808      1048576      0.1268       0.2060  [ 32]  [197]
 X_POSIX        0   write         3      31981568      1048576      0.2060       0.2069  [ 32]  [197]
```

Each access spans two Lustre stripes due to unaligned offsets

Each process interacts with two Lustre servers (OSTs)

Tuning low-level (POSIX) file I/O

Making efficient use of a no-frills I/O API

Even in this small workload, we pay a nearly **20% performance penalty when I/O accesses are not aligned** to file stripes (1 MB)

aligned

# Module	Rank	Wt/Rd	Segment	Offset	Length	Start(s)	End(s)
X_POSIX	0	write	0	0	1048576	0.0054	0.0
X_POSIX	0	write	1	10485760	1048576	0.0066	0.0
X_POSIX	0	write	2	20971520	1048576	0.0073	0.0
X_POSIX	0	write	3	31457280	1048576	0.0081	0.0

310.14 MiB/s

unaligned

# Module	Rank	Wt/Rd	Segment	Offset	Length	Start(s)	End(s)	[OST
X_POSIX	0	write	0	524288	1048576	0.0065	0.0594	[
X_POSIX	0	write	1	11010048	1048576	0.0594	0.1268	[
X_POSIX	0	write	2	21495808	1048576	0.1268	0.2060	[
X_POSIX	0	write	3	31981568	1048576	0.2060	0.2069	[

380.28 MiB/s

Tuning low-level (POSIX) file I/O

Making efficient use of a no-frills I/O API

Accounting for subtle I/O performance factors like file alignment can be a painstaking process…

*As highlighted by other presentations, high-level I/O libraries like HDF5 and PnetCDF can help mask much of the complexity needed for transforming scientific computing I/O workloads into performant POSIX-level file system accesses – **don't reinvent the wheel, use high-level I/O libraries wherever you can!***

Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

Recall that HDF5 provides a chunking mechanism to partition user datasets into contiguous chunks in the underlying file

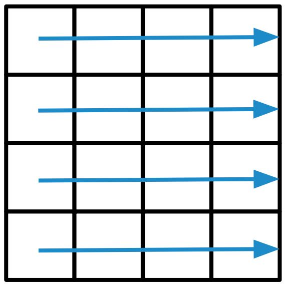
- Users can greatly improve performance of partial dataset I/O operations by choosing chunking parameters that match expected access patterns

Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

Recall that HDF5 provides a chunking mechanism to partition user datasets into contiguous chunks in the underlying file

- Users can greatly improve performance of partial dataset I/O operations by choosing chunking parameters that match expected access patterns



By default, HDF5 will store the dataset contiguously row-by-row (i.e., row-major format) in the file

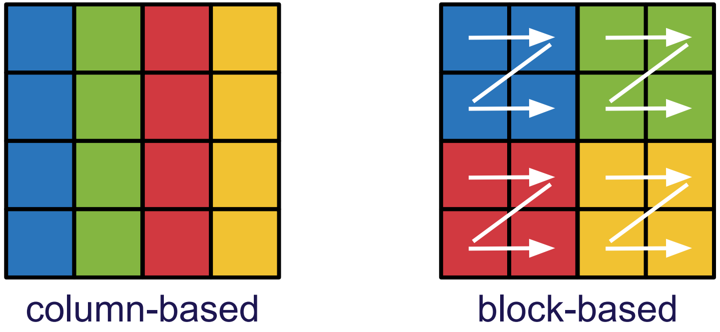
Hands on exercises: https://github.com/radix-io/hands-on

Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

Recall that HDF5 provides a chunking mechanism to partition user datasets into contiguous chunks in the underlying file

- Users can greatly improve performance of partial dataset I/O operations by choosing chunking parameters that match expected access patterns



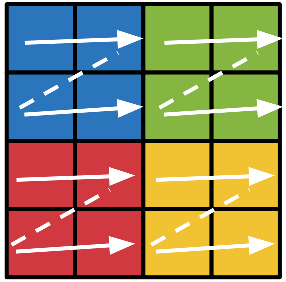
If dataset access patterns do not suit a simple row-major storage scheme, chunking can be applied to map chunks of dataset data to contiguous regions in the file

Hands on exercises: https://github.com/radix-io/hands-on

Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

Consider a 256-process (4-node) Polaris example where each process exclusively writes a 2048x2048 block of the dataset (32 MB per-process, 8 GB total)

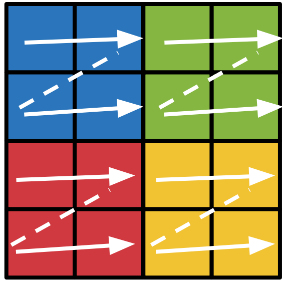


With no chunking, each process must issue many smaller non-contiguous I/O requests (solid lines) and seek around the file (dashed lines), yielding low I/O performance

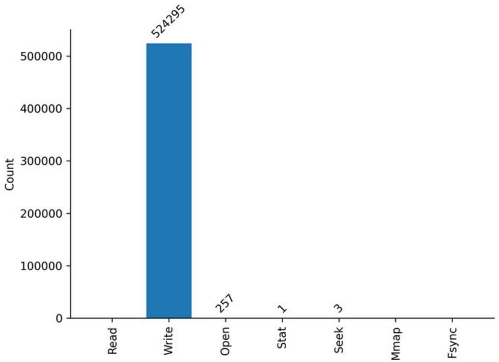
Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

Consider a 256-process (4-node) Polaris example where each process exclusively writes a 2048x2048 block of the dataset (32 MB per-process, 8 GB total)



I/O performance estimate	503.47 MiB/s (average)



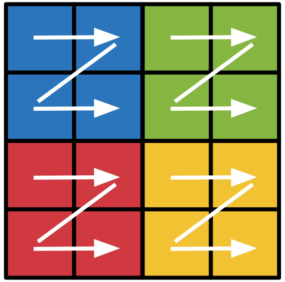
Access Size	Count
16384	524288
96	2
328	1
544	1

256 individual HDF5 writes (1-per-process) yields 500K+ POSIX writes

Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

Consider a 256-process (4-node) Polaris example where each process exclusively writes a 2048x2048 block of the dataset (32 MB per-process, 8 GB total)



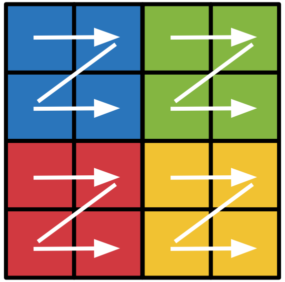
With chunking applied, each process can read their entire data block using one large, contiguous access in the file

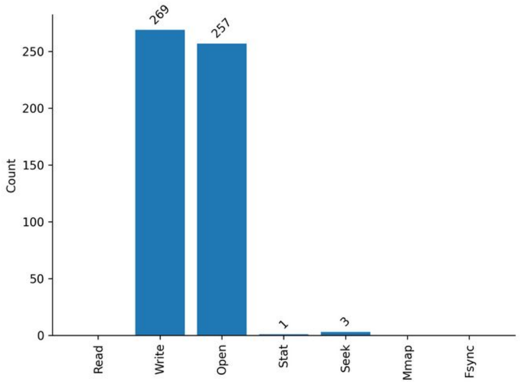
Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

Consider a 256-process (4-node) Polaris example where each process exclusively writes a 2048x2048 block of the dataset (32 MB per-process, 8 GB total)

I/O performance estimate	1450.57 MiB/s (average)





Access Size	Count
33554432	256
2616	6
96	2
544	1

Chunking results in a much more manageable POSIX workload

Nearly a 3x performance improvement!

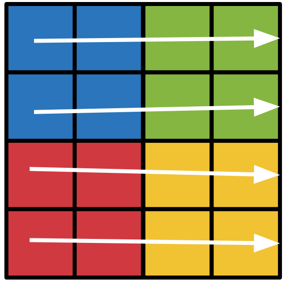
Hands on exercises: https://github.com/radix-io/hands-on

Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

An alternative optimization forgoes chunking and uses collective I/O to improve the efficiency of this block-style data access

- Rely on MPI-IO layer collective buffering algorithm to generate contiguous storage accesses and to limit number of clients interacting with storage system



With collective I/O enabled, designated aggregator processes perform I/O on behalf of their peers, and communicate their data using MPI calls

E.g., the green process sends its write data to the blue process (aggregator), who then writes both of their data in one big contiguous chunk

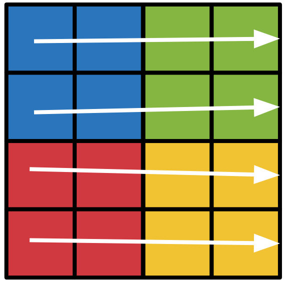
Hands on exercises: https://github.com/radix-io/hands-on

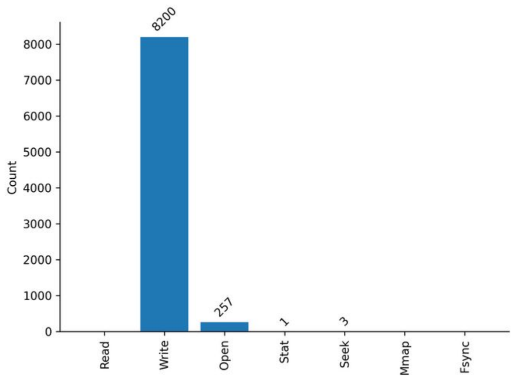
Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

Consider a 256-process (4-node) Polaris example where each process exclusively writes a 2048x2048 block of the dataset (32 MB per-process, 8 GB total)

I/O performance estimate	13124.01 MiB/s (average)





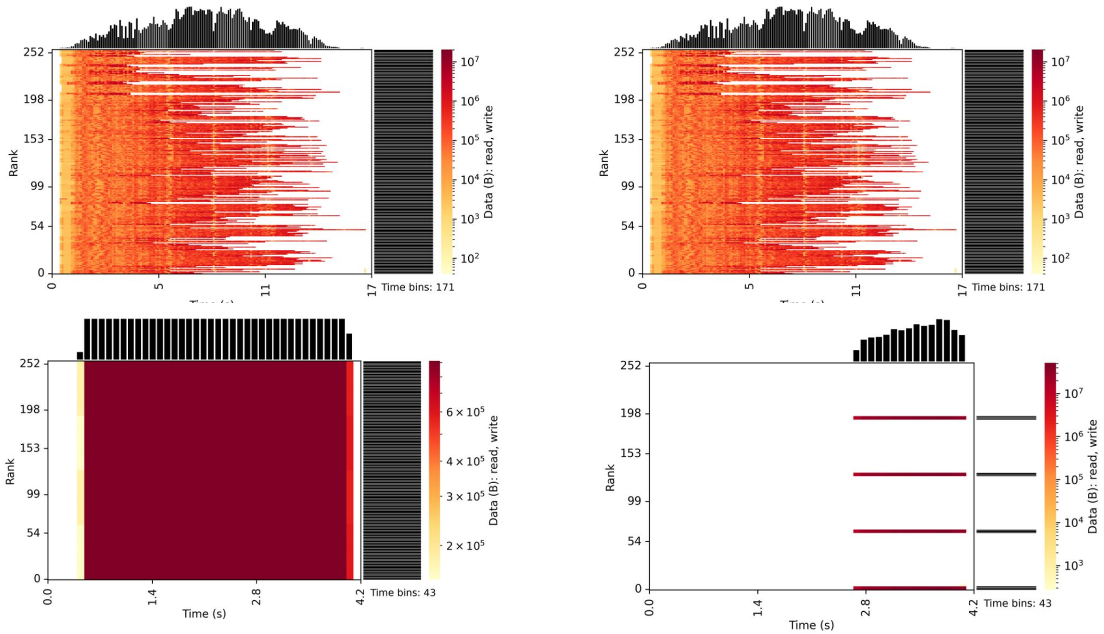
Access Size	Count
1048576	8191
96	2
2048	1
1046528	1

Collective I/O yields 26x improvement over no chunking, and 9x improvement over chunking!!!

Tuning high-level (HDF5) data access

Optimizing application interactions with the I/O stack

MPI-IO **POSIX**



Darshan I/O activity heatmaps illustrate how different the I/O behavior is for the unoptimized independent configuration (**top**) and the most performant collective I/O configuration (**bottom**)

Summarizing I/O tuning options

As a user of I/O interface X, what tuning vectors do I have?

I/O Interface	Striping	Alignment	Collective I/O	Chunking
HDF5	✓	✓	✓	✓
PnetCDF	✓	✓	✓	X
MPI-IO	✓	✓	✓	X
POSIX	✓	✓-	X	X

Hands on exercises: https://github.com/radix-io/hands-on

Summarizing I/O tuning options

As a user of I/O interface X, what tuning vectors do I have?

I/O Interface	Striping	Alignment	Collective I/O	Chunking
HDF5	✓	✓	✓	✓
PnetCDF	✓	✓	✓	X
MPI-IO	✓	✓	✓	X
POSIX	✓	✓-	X	X

Automatically align application data and library metadata, if user requests so

Collective I/O can be automatically aligned

POSIX I/O requires manually aligning every access

Summarizing I/O tuning options

As a user of I/O interface X, what tuning vectors do I have?

I/O Interface	Striping	Alignment	Collective I/O	Chunking
HDF5	✓	✓	✓	✓
PnetCDF	✓	✓	✓	X
MPI-IO	✓	✓	✓	X
POSIX	✓	✓-	X	X

Just another reminder that high-level I/O libraries are here to make your life easier

- I/O optimization strategies like collective I/O & chunking can net large performance gains, especially when combined with striping and alignment optimizations

Adapting to a changing HPC landscape

Adapting to a changing HPC landscape

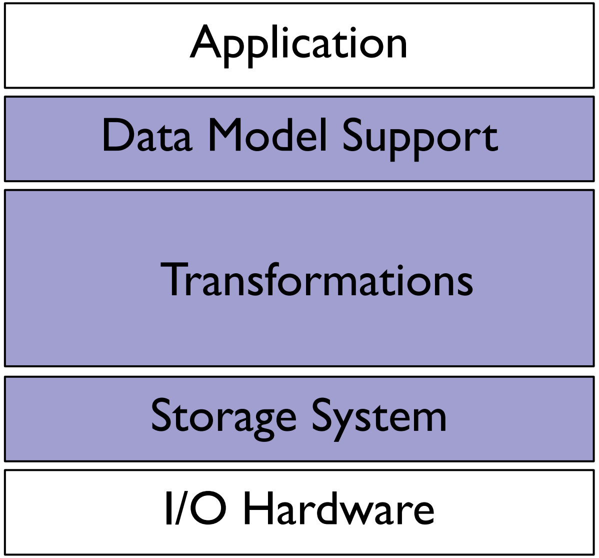
The various technologies covered today form much of the foundation of the traditional HPC data management stack

- Variations on this stack have been deployed at HPC facilities and leveraged by users for high-performance parallel I/O for decades

But, the HPC computing landscape is changing, even if slowly

Changes driven at both ends of the stack

- Newly embraced compute paradigms
- Emerging storage technologies

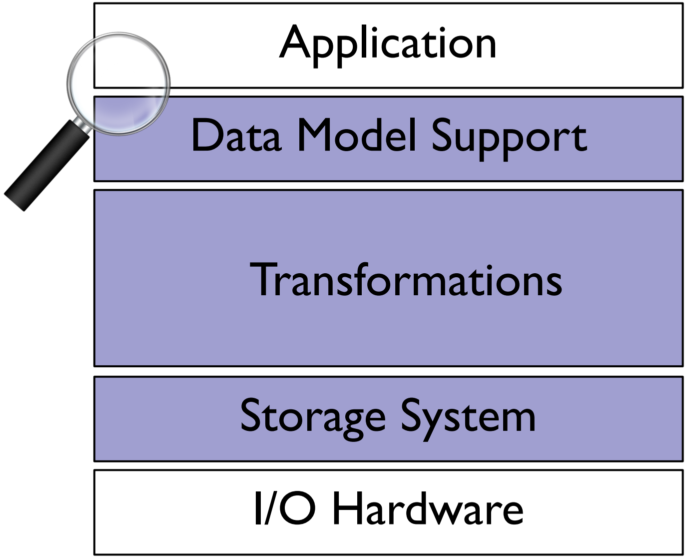


New computing paradigms

Large-scale MPI applications are still the norm at most HPC centers, but other non-MPI compute frameworks are gaining traction:

- AI/ML (TensorFlow, Keras, PyTorch)
- Data analytics frameworks (Spark, Dask)
- Other non-MPI distributed computing frameworks (Legion, UPC)

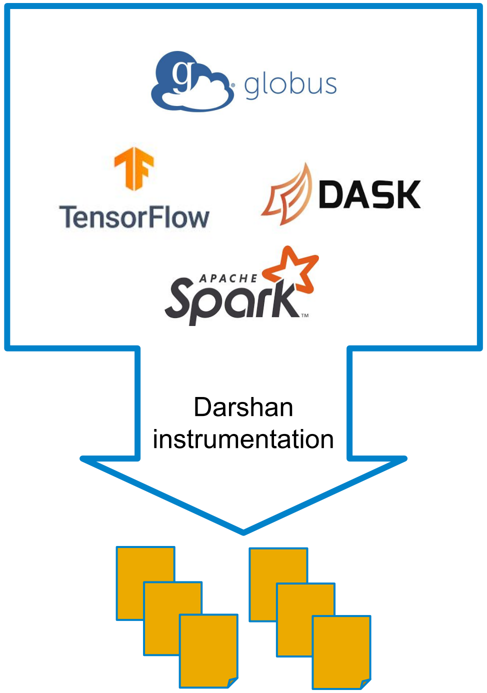
Many of these frameworks define their own data models and have their own mechanisms for managing distributed tasks



Darshan instrumentation beyond MPI

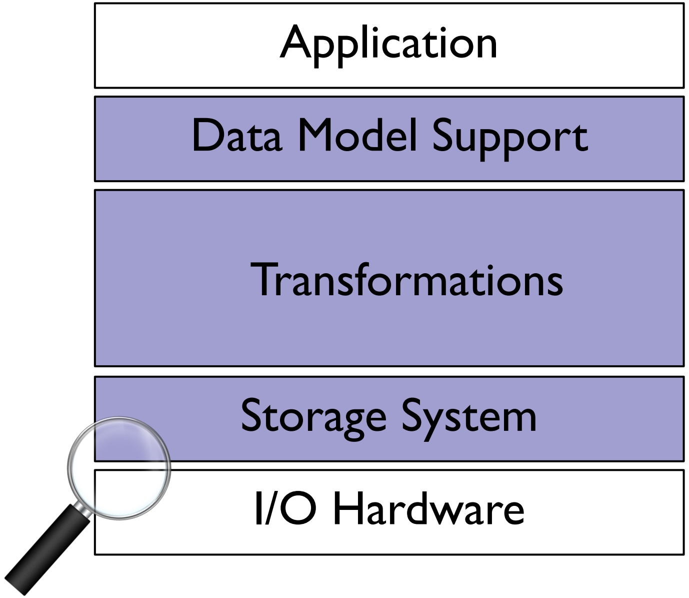
- Historically, Darshan has only worked with MPI applications
  - `MPI_Init/MPI_Finalize` used to bootstrap/shutdown Darshan
- Darshan has been modified to use a secondary bootstrapping mechanism that enables its use outside of MPI
  - Based on GCC-specific library constructor/destructor attributes
  - **Only works for dynamically-linked executables!**
- To enable non-MPI mode, users must explicitly opt-in by setting the **`DARSHAN_ENABLE_NONMPI`** environment variable
  - A unique log will be generated for every process that executes
  - Often best to limit instrumentation scope to the target executable:

```
$ LD_PRELOAD=/path/to/libdarshan.so \
  DARSHAN_ENABLE_NONMPI=1 \
  ./exe <args>
```



Emerging storage technologies

- HPC storage technology is changing to meet needs of diverse application workloads
  - Users typically have more options than a traditional parallel file system over HDDs
- Hardware trends enabling low-latency, high-bandwidth I/O to applications
  - E.g., SSDs, SCM
- Novel storage services offer compelling alternatives to traditional file systems
  - E.g., **DAOS**, Unify

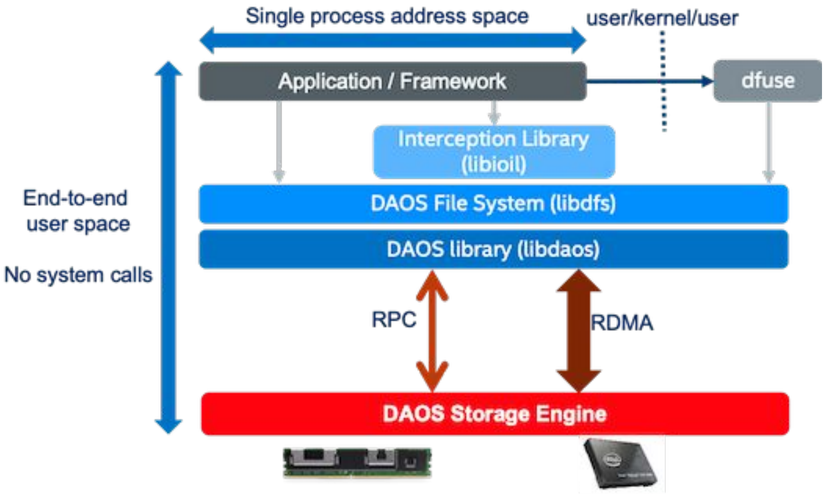


Darshan instrumentation of DAOS

ALCF Aurora will feature Intel's DAOS storage system, a first-of-a-kind object-based storage system for large-scale HPC platforms

- Leverages both SCM and SSDs for storage

Development of Darshan instrumentation modules is underway to provide valuable insights into the various ways apps and I/O middleware utilize DAOS



Various access methods for DAOS users.

Figure courtesy of Intel

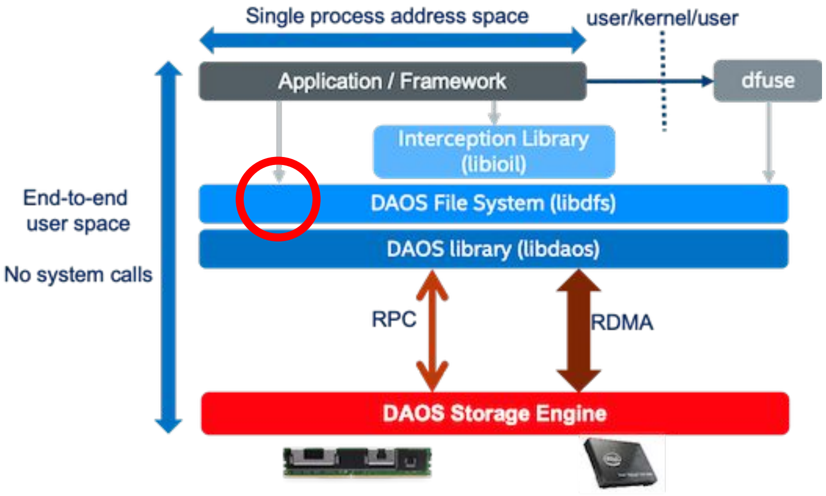
Darshan instrumentation of DAOS

ALCF Aurora will feature Intel's DAOS storage system, a first-of-a-kind object-based storage system for large-scale HPC platforms

- Leverages both SCM and SSDs for storage

Development of Darshan instrumentation modules is underway to provide valuable insights into the various ways apps and I/O middleware utilize DAOS

- **Direct usage of POSIX-like DAOS file system (libdfs) interface**



Various access methods for DAOS users.

Figure courtesy of Intel

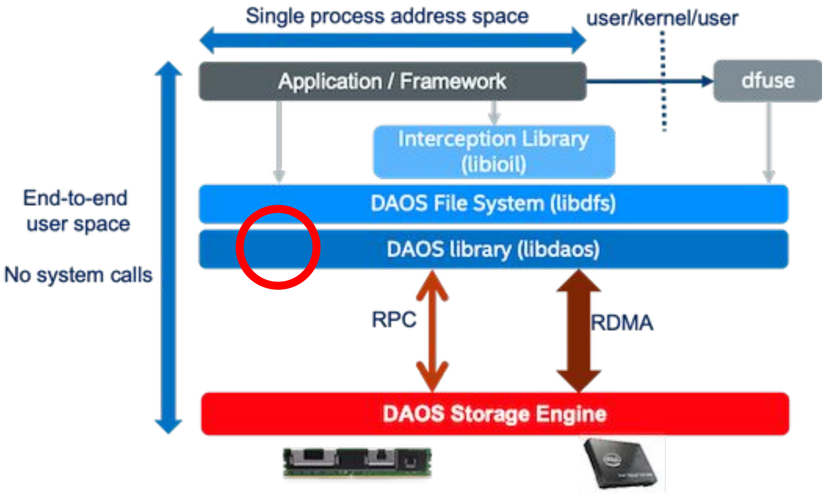
Darshan instrumentation of DAOS

ALCF Aurora will feature Intel's DAOS storage system, a first-of-a-kind object-based storage system for large-scale HPC platforms

- Leverages both SCM and SSDs for storage

Development of Darshan instrumentation modules is underway to provide valuable insights into the various ways apps and I/O middleware utilize DAOS

- Direct usage of POSIX-like DAOS file system (libdfs) interface
- **Direct usage of native DAOS object (libdaos) interface**



Various access methods for DAOS users.

Figure courtesy of Intel

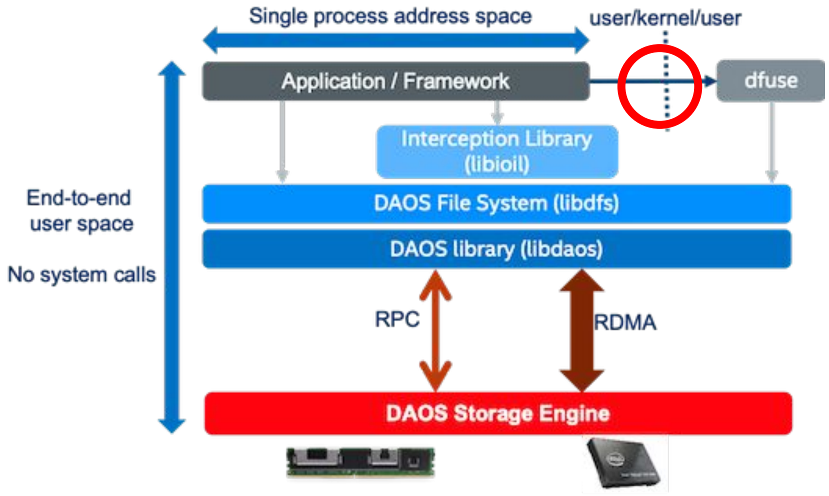
Darshan instrumentation of DAOS

ALCF Aurora will feature Intel's DAOS storage system, a first-of-a-kind object-based storage system for large-scale HPC platforms

- Leverages both SCM and SSDs for storage

Development of Darshan instrumentation modules is underway to provide valuable insights into the various ways apps and I/O middleware utilize DAOS

- Direct usage of POSIX-like DAOS file system (libdfs) interface
- Direct usage of native DAOS object (libdaos) interface
- **Legacy POSIX support using FUSE**



Various access methods for DAOS users.

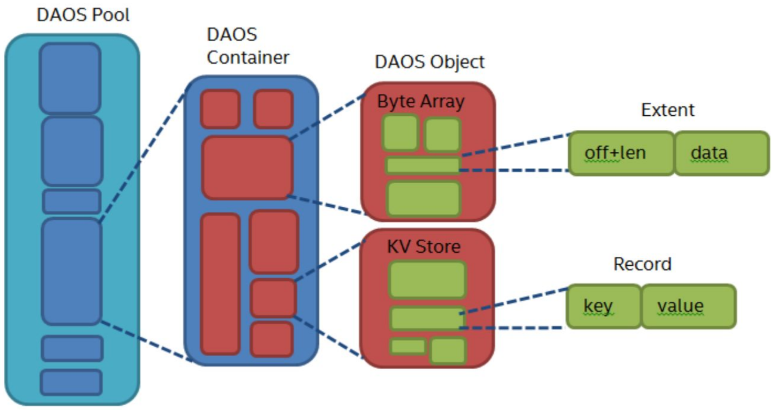
Figure courtesy of Intel

Darshan instrumentation of DAOS

DAOS will provide new-to-HPC interfaces that can yield attractive performance characteristics if used to their full potential

- Array objects
  - Extent-based access, similar to files
- Key-val objects
  - Data accessed using arbitrary keys
  - Keys are split into a dkey (distribution key) and an akey (attribute key) to offer users control over data locality
    - All keys with same dkey are co-located on the same DAOS storage target

Darshan can play an important role in understanding application and I/O library usage of DAOS objects



DAOS storage model. DAOS objects can be accessed using either key-val or array interfaces.

Figure courtesy of Intel

Additional Darshan tips and tricks

Finer-grained details with Darshan: DXT tracing

- By default, Darshan captures a fixed set of counters for each file
- With DXT, Darshan additionally traces every read/write operation (for POSIX and MPI-IO interfaces)
- Enable by setting **DXT_ENABLE_IO_TRACE** env variable
- Finer grained instrumentation data comes at a cost of additional overhead and larger logs

```
export DXT_ENABLE_IO_TRACE=1

mpiexec -n 256 --ppn 64 ./helloworld
```


Hands on exercises: https://github.com/radix-io/hands-on

Finer-grained details with Darshan: DXT tracing

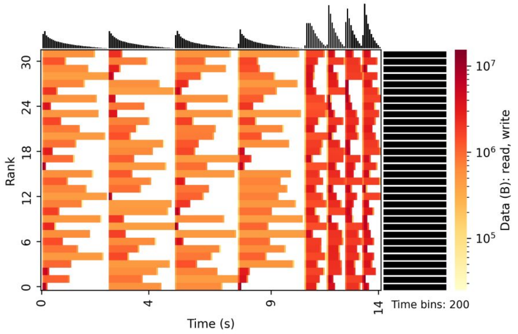
- By default, Darshan captures a fixed set of counters for each file
- With DXT, Darshan additionally traces every read/write operation (for POSIX and MPI-IO interfaces)
- Enable by setting **`DXT_ENABLE_IO_TRACE`** env variable
- Finer grained instrumentation data comes at a cost of additional overhead and larger logs

```
# DXT, file_id: 1163774110118722858, file_name: /grand/projects/ATPESC2023/usr/snyder/hello
# DXT, rank: 0, hostname: x3202c0s1b0n0
# DXT, write_count: 160, read_count: 0
# DXT, mnt_pt: /, fs_type: overlay
# Module    Rank  Wt/Rd  Segment          Offset       Length    Start(s)      End(s)
 X_POSIX       0  write        0               0      1048576      3.9347      3.9468
 X_POSIX       0  write        1       167772160      1048576      4.2503      4.2575
 X_POSIX       0  write        2       335544320      1048576      4.5495      4.5564
 X_POSIX       0  write        3       503316480      1048576      4.8632      4.8707
```

Trace includes the timestamp, file offset, and size of every I/O operation on every rank. `darshan-dxt-parser` utility can provide a raw text dump of the trace

Finer-grained details with Darshan: DXT tracing

- By default, Darshan captures a fixed set of counters for each file
- With DXT, Darshan additionally traces every read/write operation (for POSIX and MPI-IO interfaces)
- Enable by setting **`DXT_ENABLE_IO_TRACE`** env variable
- Finer grained instrumentation data comes at a cost of additional overhead and larger logs



Traces can be visualized using summary report heatmaps or custom tools (**more on this shortly**)

Finer-grained details with Darshan: disabling shared file reductions

- To reduce log file size, globally shared file records are reduced into a single instrumentation record by default
  - However, this slightly masks per-rank contributions to I/O
- This behavior can be disabled by setting **`DARSHAN_DISABLE_SHARED_REDUCTION`** environment variable
- Allows for full accounting of per-rank contributions to shared files, if these details are important (e.g., for understanding collective I/O algorithms)

```
export DARSHAN_DISABLE_SHARED_REDUCTION=1
mpiexec -n 256 --ppn 64 ./helloworld $SCRATCHDIR
```

Finer-grained details with Darshan: disabling shared file reductions

- To reduce log file size, globally shared file records are reduced into a single instrumentation record by default
  - However, this slightly masks per-rank contributions to I/O
- This behavior can be disabled by setting **`DARSHAN_DISABLE_SHARED_REDUCTION`** environment variable
- Allows for full accounting of per-rank contributions to shared files, if these details are important (e.g., for understanding collective I/O algorithms)

```
$ darshan-parser ./snyder_helloworld_id565659-63984_8-3-68717-7310522192037150959_1.dar
>    grep POSIX_BYTES_WRITTEN
POSIX   -1      1163774110118722858     POSIX_BYTES_WRITTEN     26214400000     /grand/
```

Rank -1 indicates a shared file record, with counters containing a reduced value access all ranks (e.g., ~24.5 GiB total bytes written across all ranks)

Finer-grained details with Darshan: disabling shared file reductions

- To reduce log file size, globally shared file records are reduced into a single instrumentation record by default
  - However, this slightly masks per-rank contributions to I/O
- This behavior can be disabled by setting **DARSHAN_DISABLE_SHARED_REDUCTION** environment variable
- Allows for full accounting of per-rank contributions to shared files, if these details are important (e.g., for understanding collective I/O algorithms)

```
$ darshan-parser ./snyder_helloworld_id566419-55186_8-4-65788-342089402740504 1227_1.dar
> grep POSIX_BYTES_WRITTEN
POSIX  0     1163774110118722858     POSIX_BYTES_WRITTEN     164626432       /grand/
POSIX  255   1163774110118722858     POSIX_BYTES_WRITTEN     0        /grand/projects
```

With shared reductions disabled, each rank retains their own record giving full insight into per-rank contributions (rank 0 writes 157 MiB and rank 255 writes nothing)

Darshan runtime library configuration

- To bound memory overheads, Darshan imposes several internal memory limits (total memory usage, per-module record limits, etc.)
- For some workloads, default limits may be exceeded resulting in partial instrumentation data
- To offer user's more control over memory limits and instrumentation scope, Darshan provides a comprehensive runtime configuration system
  - Environment variables or config files

```
#    KEY                 VALUE                 MODULES
NAME_EXCLUDE        ^/home                      *
NAME_EXCLUDE        .pyc$                       *
NAME_EXCLUDE        .so$                        *
NAME_INCLUDE        .h5$                        *

MODMEM      8

MAX_RECORDS         4000                      POSIX

MOD_ENABLE          DXT_POSIX,DXT_MPIIO

APP_EXCLUDE         git,ls,sed
```

Regular expressions can be specified to control whether matching record name patterns are included/excluded in Darshan instrumentation

Darshan runtime library configuration

- To bound memory overheads, Darshan imposes several internal memory limits (total memory usage, per-module record limits, etc.)
- For some workloads, default limits may be exceeded resulting in partial instrumentation data
- To offer user's more control over memory limits and instrumentation scope, Darshan provides a comprehensive runtime configuration system
  - Environment variables or config files

```
#   KEY                VALUE              MODULES
NAME_EXCLUDE         ^/home                  *
NAME_EXCLUDE         .pyc$                   *
NAME_EXCLUDE         .so$                    *
NAME_INCLUDE         .h5$                    *

MODMEM        8

MAX_RECORDS          4000                 POSIX

MOD_ENABLE           DXT_POSIX,DXT_MPIIO

APP_EXCLUDE          git,ls,sed
```

Settings are also offered to control total per-process memory usage (8 MiB) and per-module maximum record counts (4000 POSIX records)

Hands on exercises: https://github.com/radix-io/hands-on

Darshan runtime library configuration

- To bound memory overheads, Darshan imposes several internal memory limits (total memory usage, per-module record limits, etc.)
- For some workloads, default limits may be exceeded resulting in partial instrumentation data
- To offer user's more control over memory limits and instrumentation scope, Darshan provides a comprehensive runtime configuration system
  - Environment variables or config files

```
#   KEY                 VALUE              MODULES
NAME_EXCLUDE        ^/home                  *
NAME_EXCLUDE        .pyc$                   *
NAME_EXCLUDE        .so$                    *
NAME_INCLUDE        .h5$                    *

MODMEM      8

MAX_RECORDS         4000                   POSIX

MOD_ENABLE          DXT_POSIX,DXT_MPIIO

APP_EXCLUDE         git,ls,sed
```

Additional settings allow control over enabled/disabled modules, as well as application names that should be included/excluded from instrumentation

Other I/O analysis tools

Darshan-based analysis tools

Using Darshan as a starting point for developing new I/O analysis tools is attractive for a couple of reasons:

1. Darshan is commonly deployed in production at many HPC sites, making its I/O characterization data generally accessible to custom tools
2. Recent PyDarshan work has enabled much more agile development of Darshan-based I/O analysis tools in Python

We will start by considering a couple of Darshan-based I/O analysis tools: **DXT Explorer** and **Drishti**

DXT Explorer

- Darshan does not offer much in terms of DXT trace analysis tools beyond general I/O activity heatmaps
- **DXT Explorer★** is an interactive web-based trace analysis tool for DXT data that was developed to provide:
  - Combined views of MPI-IO and POSIX activity
  - Zoom in/out capabilities to focus on subsets of ranks or specific time slices
  - Contextual information about I/O calls
  - Views based on operation type, size, and spatiality
- Interactive trace analysis with DXT Explorer can enable interesting new insights into app I/O behavior

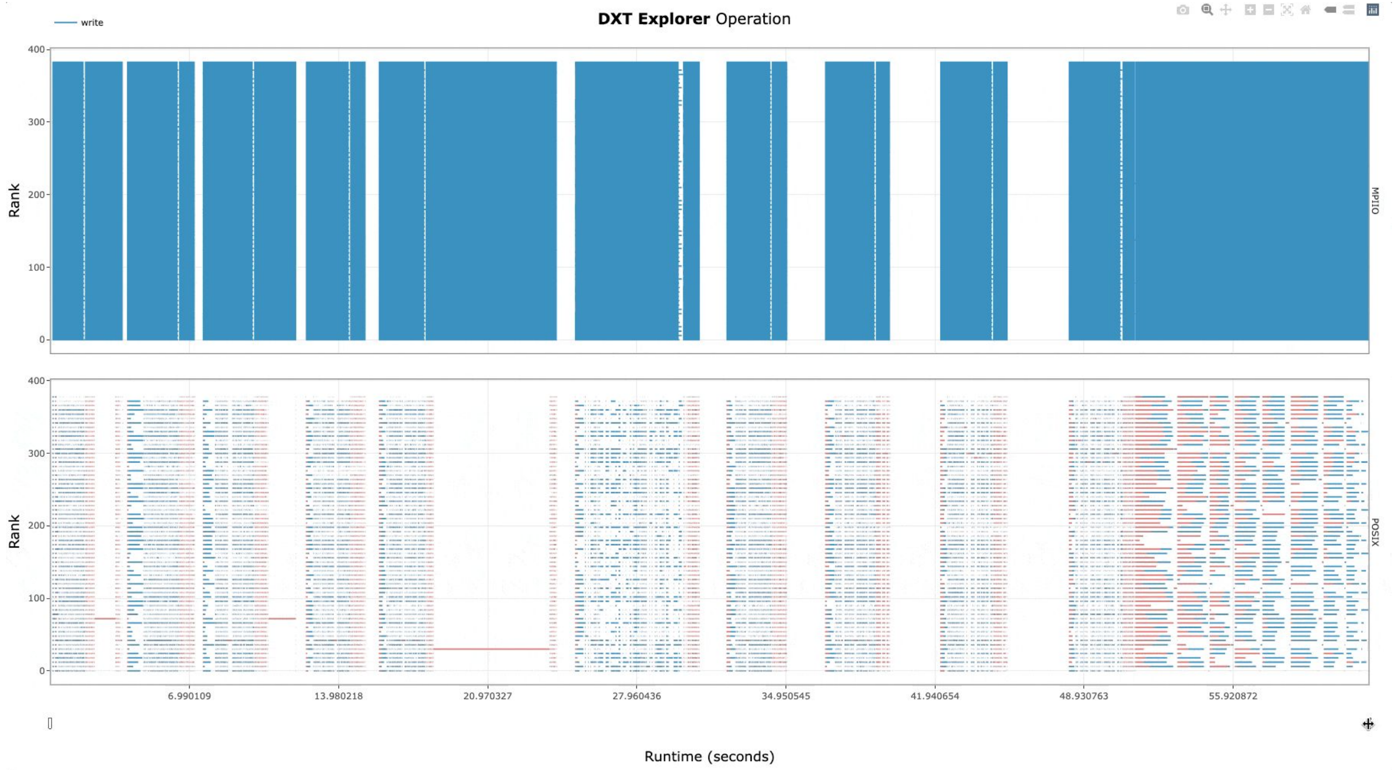
github.com/hpc-io/dxt-explorer

docker pull hpcio/dxt-explorer

★ DXT Explorer was developed by Jean Luca Bez (LBL). Slide content also provided courtesy of Jean Luca.

Bez, Jean Luca, et al. "I/O bottleneck detection and tuning: connecting the dots using interactive log analysis." *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*. IEEE, 2021.

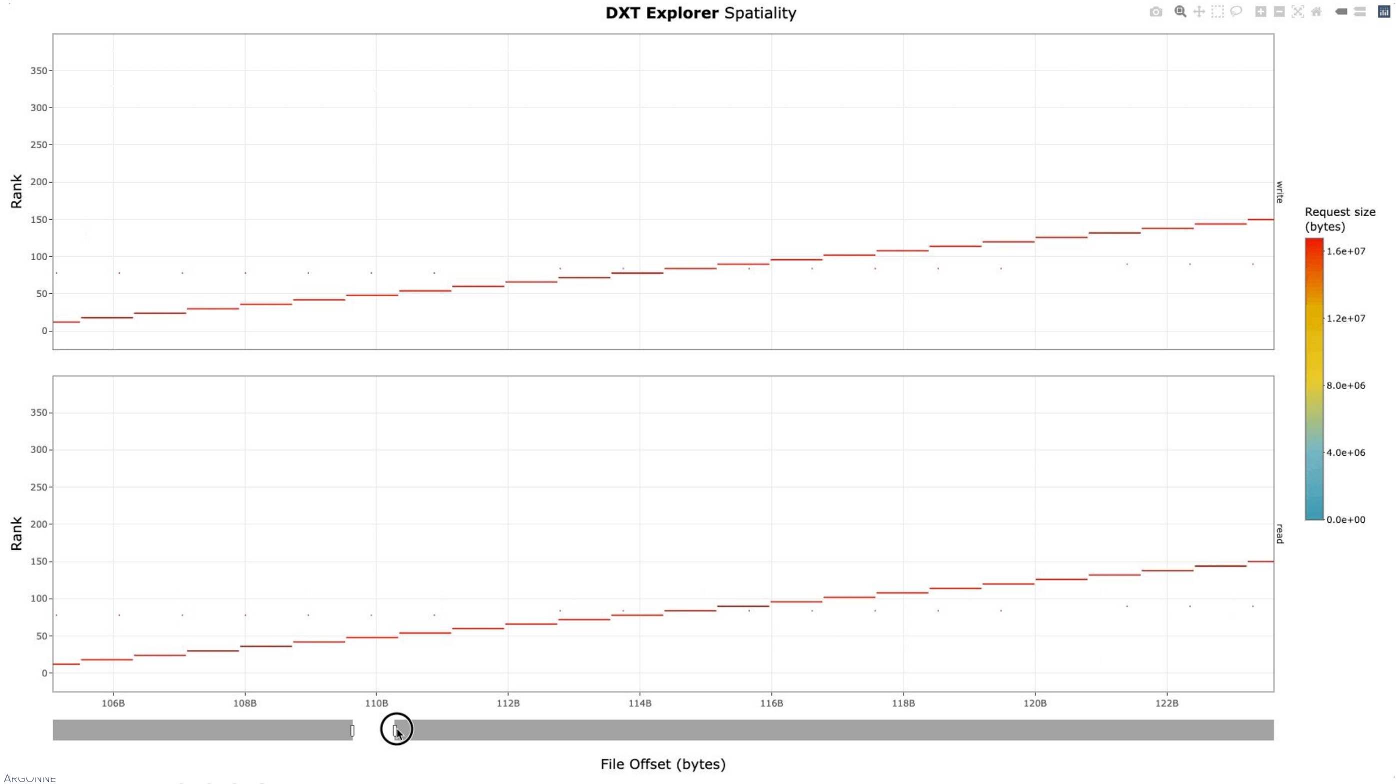
DXT Explorer



Explore the timeline by zooming in and out and observing how the MPI-IO calls are translated to the POSIX layer. For instance, you can use this feature to detect stragglers.

Hands on exercises: https://github.com/radix-io/hands-on

DXT Explorer



Explore the spatiality of accesses in file by each rank with contextual information. Understand how each rank is accessing each file.

Drishti

- Darshan can capture detailed I/O characterization data for an app, but translating this raw data to actionable tuning feedback is a significant challenge
- **Drishti★** is a command-line tool to guide end-users in optimizing I/O in their applications by detecting typical I/O performance pitfalls and providing a set of recommendations
- Drishti checks each given Darshan log against 30+ heuristic triggers for various I/O issues and suggests actions to take to resolve them
  - 4 levels of triggers: *high, warning, ok, info*

github.com/hpc-io/drishti-io

docker pull hpcio/drishti

★ Drishti was developed by Jean Luca Bez (LBL). Slide content also provided courtesy of Jean Luca.

Bez, Jean Luca, Hammad Ather, and Suren Byna. "Drishti: guiding end-users in the I/O optimization journey." 2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW). IEEE, 2022.

Drishti

```
Drishti

┌─ DRISHTI v.0.3 ──────────────────────────────────────────────────────────────┐
│                                                                              │
│ JOB:            1190243                                                      │
│ EXECUTABLE:     bin/8_benchmark_parallel                                     │
│ DARSHAN:        jlbez_8_benchmark_parallel_id1190243_7-23-45631-11755726114084236527_1.darshan
│ EXECUTION DATE: 2021-07-23 16:40:31+00:00 to 2021-07-23 16:40:32+00:00 (0.00 hours)
│ FILES:          6 files (1 use STDIO, 2 use POSIX, 1 use MPI-IO)             │
│ PROCESSES       64                                                           │
│ HINTS:          romio_no_indep_rw=true cb_nodes=4                            │
│                                                                              │
└─ 1 critical issues, 5 warnings, and 5 recommendations ───────────────────────┘

┌─ METADATA ───────────────────────────────────────────────────────────────────┐
│                                                                              │
│ ▶ Application is read operation intensive (6.34% writes vs. 93.66% reads)    │
│ ▶ Application might have redundant read traffic (more data was read than the highest read offset)
│ ▶ Application might have redundant write traffic (more data was written than the highest write offset)
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘

┌─ OPERATIONS ─────────────────────────────────────────────────────────────────┐
│                                                                              │
│ ▶ Application issues a high number (285) of small read requests (i.e., < 1MB) which represents 37.11% of all
│read/write requests
│   ↪ 284 (36.98%) small read requests are to "benchmark.h5"
│   ↪ Recommendations:
│     ↪ Consider buffering read operations into larger more contiguous ones
│     ↪ Since the appplication already uses MPI-IO, consider using collective I/O calls (e.g. MPI_File_read_all() or
│MPI_File_read_at_all()) to aggregate requests into larger ones
```

Overall information about the Darshan log and execution

Number of critical issues, warning, and recommendations

Details on metadata and data operations

Critical issue and corresponding recommendation for `benchmark.h5`

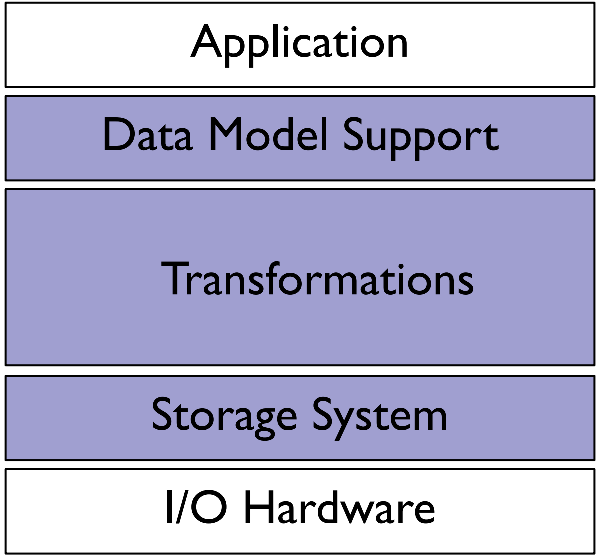
Other I/O analysis tools

- There are some other notable tools that may be of use for gaining more insights into the I/O behavior of an application:
  - **TAU**: http://www.cs.uoregon.edu/research/tau/
    - General call profiling/tracing toolkit for HPC applications, including I/O routines
    - Tools for visualizing profiles/traces and detecting bottlenecks, etc.
    - See: https://hps.vi4io.org/_media/events/2019/sc19-analyzing-tau.pdf
  - **Recorder**: https://github.com/uiuc-hpc/Recorder
    - Multi-level detailed traces and corresponding trace viz tools
    - More detail than DXT but not as production hardened
  - **LDMS**: https://hmdsa.github.io/hmdsa/pages/tools/ldms
    - Beyond the application, includes detailed system metrics collection
    - Not typically available to users in general, but maybe another resource at some facilities

Wrapping up

- Hopefully this material proves useful in providing a deeper understanding of the different layers of the HPC I/O stack covered today, as well as potential tuning vectors available to you as user

- **Some key takeaways:**
  - Optimizing your I/O workload can be challenging, but can potentially offer large performance gains
  - Use high-level I/O libraries where you can
  - Don’t always count on I/O libraries or file systems to automatically provide you the best performance out-of-the-box



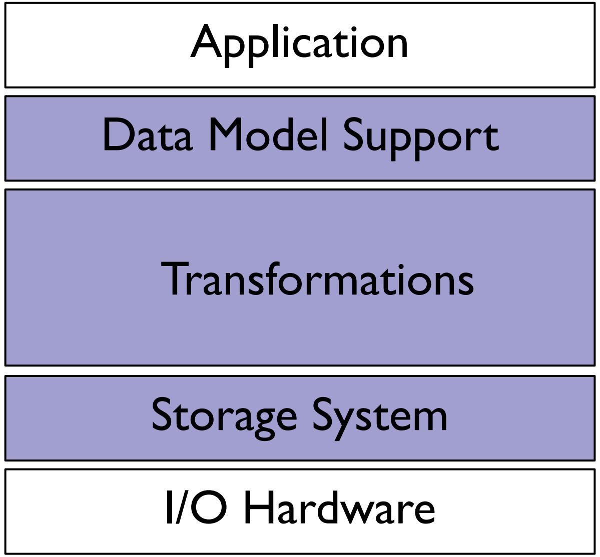
Wrapping up

- Darshan is an invaluable tool for providing understanding of application I/O behavior and informing potential tuning decisions – use it to experiment with different tuning options and measure resulting I/O performance!
- Please reach out with questions, feedback, etc.

DARSHAN **https://www.mcs.anl.gov/research/projects/darshan/**

github.com/darshan-hpc/darshan

darshan-io.slack.com



Thank you!