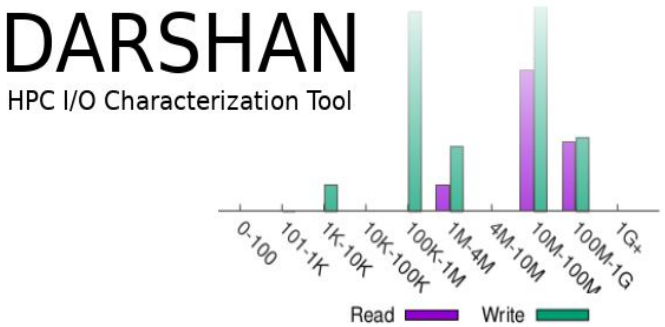
March 28, 2023

Argonne NATIONAL LABORATORY

Understanding and Improving the I/O Behavior of Scientific Computing Applications



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

MCS CS Seminar Series

Understanding and improving HPC I/O

- Characterizing and understanding application I/O workloads is critical to ensuring efficient use of an evolving and increasingly complex HPC I/O stack
  - Deep layers of coordinating I/O libraries and entirely new-to-HPC storage paradigms (e.g., object storage)
  - Emerging storage hardware (e.g., CXL) and storage architectures (e.g., burst buffers)
- I/O analysis tools are invaluable in helping to navigate this complexity and to better understand I/O
  - Characterize I/O behavior of individual jobs to inform tuning decisions
  - Characterize job populations to better understand system-wide I/O stack usage and optimize deployments

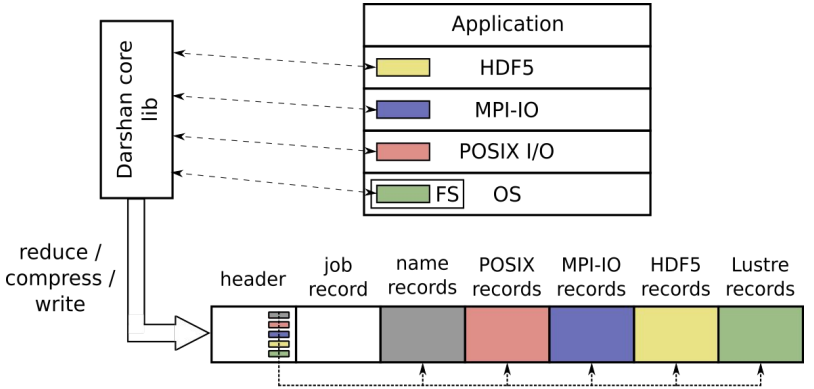
Darshan: An I/O characterization tool for HPC applications

What is Darshan?

- Darshan is a lightweight I/O characterization tool that captures concise views of HPC application I/O behavior
  - Produces a summary of I/O activity for each instrumented job
    - Counters, histograms, timers, & statistics
    - If requested by user, full I/O traces
- Widely available
  - Deployed (and commonly enabled by default) at many HPC facilities around the world
- Easy to use
  - No code changes required to integrate Darshan instrumentation
  - Negligible performance impact; just “leave it on”
- Modular
  - Adding instrumentation for new I/O interfaces or storage components is straightforward

How does Darshan work?

- Darshan records file access statistics independently on each process
- At app shutdown, collect, aggregate, compress, and write log data
- After job completes, analyze Darshan log data
  - `darshan-parser` - provides complete text-format dump of all counters in a log file
  - *PyDarshan* - Python analysis module for Darshan logs, including a summary tool for creating HTML reports



- Originally designed for MPI applications, but in recent Darshan versions (3.2+) any dynamically-linked executable can be instrumented
  - In MPI mode, a log is generated for each *app*
  - In non-MPI mode, a log is generated for each *process*

Using Darshan

Instrumenting apps with Darshan

Traditional usage on HPC platforms

- On many HPC platforms (e.g., ALCF Theta, NERSC Cori & Perlmutter, OLCF Summit), Darshan is already installed and typically enabled by default

```
snyder@thetalogin4:~> module list |& tail -n 5
 20) cray-mpich/7.7.14
 21) nompirun/nompirun
 22) adaptive-routing-a3
 23) darshan/3.3.0
 24) xalt
```

Darshan 3.3.0 is enabled by default on ALCF Theta

```
ssnyder@perlmutter:login37:~> module load darshan
ssnyder@perlmutter:login37:~> module -t list |& tail -n 5
Nsight-Systems/2022.2.1
cudatoolkit/11.7
craype-accel-nvidia80
gpu/1.0
darshan/3.4.0
```

Darshan module can typically be explicitly loaded if not available by default, e.g., Darshan 3.4.0 on NERSC Perlmutter

Instrumenting apps with Darshan

Traditional usage on HPC platforms

- On many HPC platforms (e.g., ALCF Theta, NERSC Cori & Perlmutter, OLCF Summit), Darshan is already installed and typically enabled by default
  - **Just compile and run your apps like normal**

```
ssnyder@perlmutter:login05: cc -o mpi-io-test mpi-io-test.c
ssnyder@perlmutter:login05: ldd mpi-io-test | grep darshan
        libdarshan.so.0 => /global/common/software/nersc/pm
```

```
ssnyder@perlmutter:nid004489: srun -n 32 ./mpi-io-test
# Using mpi-io calls.
nr_procs = 32, nr_iter = 1, blk_sz = 16777216, coll = 0
# total_size = 536870912
# Write: min_t = 0.416507, max_t = 0.456917, mean_t = 0.43
# Read:  min_t = 0.010588, max_t = 0.014461, mean_t = 0.01
Write bandwidth = 1174.985593 Mbytes/sec
Read bandwidth = 37124.695394 Mbytes/sec
```

E.g., compiling and running a simple example on NERSC Perlmutter

Instrumenting apps with Darshan

Traditional usage on HPC platforms

- On many HPC platforms (e.g., ALCF Theta, NERSC Cori & Perlmutter, OLCF Summit), Darshan is already installed and typically enabled by default
  - ➢ **Just compile and run your apps like normal**

Important caveats related to non-MPI usage:

- Requires dynamically-linked executables
- Non-MPI mode must be explicitly enabled via env variable
  - `export DARSHAN_ENABLE_NONMPI=1`
- Some systems may have dated Darshan versions that don't properly support non-MPI mode

Instrumenting apps with Darshan

Traditional usage on HPC platforms

- On many HPC platforms (e.g., ALCF Theta, NERSC Cori & Perlmutter, OLCF Summit), Darshan is already installed and typically enabled by default
  - Just compile and run your apps like normal
  - **Logs are written to a central repository for all users when the app terminates**

```
ssnyder@perlmutter:login05: darshan-config --log-path
/pscratch/darshanlogs
ssnyder@perlmutter:login05: cd /pscratch/darshanlogs/2023/3/14
ssnyder@perlmutter:login05: ls | grep snyder
ssnyder_mpi-io-test_id6058027-191211_3-14-39483-261794756305089
8457_1.darshan
```

‘**`darshan-config --log-path`**’ command can be used to find output log directory. Directory is further organized into year/month/day subdirectories.

Log file name includes username, app name, and job ID for easy identification.

Instrumenting apps with Darshan

Installing and using your own Darshan tools

- In some circumstances, it may be necessary to roll your own install
  - Darshan not installed or lacking necessary features
  - Need to build Darshan in specific software environments (e.g., containers with old compilers)
- Beyond installing from source, Darshan is also available on Spack
  - *darshan-runtime*: runtime instrumentation library linked with application
  - *darshan-util*: log analysis utilities
  - E.g., "**`spack install darshan-runtime`**"
- Once installed, users can `LD_PRELOAD` the darshan-runtime library
  - Output logs are written to directory pointed to by `DARSHAN_LOG_DIR_PATH` environment variable (defaults to `$HOME`)

Analyzing Darshan logs

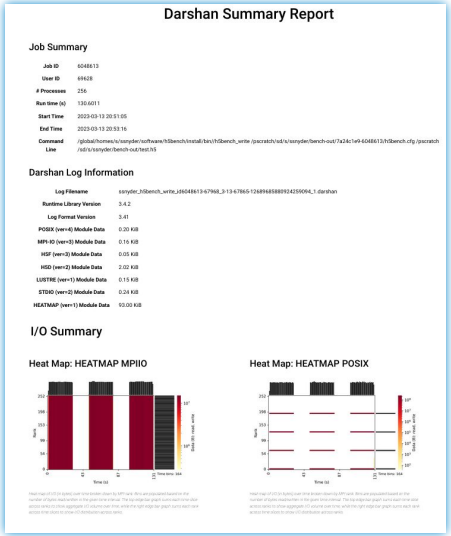
- After locating your log, users can utilize Darshan log analysis tools for gaining insights into application I/O behavior:

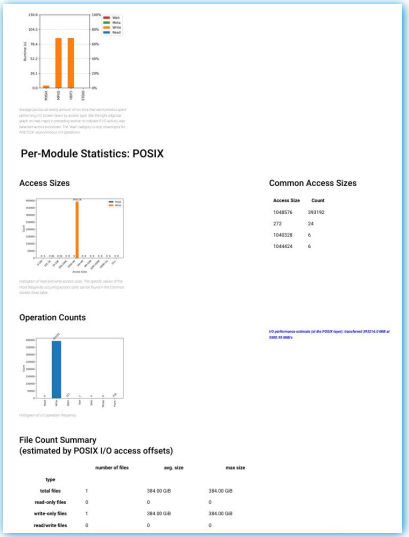
```
shane@shane-x1-carbon: darshan-parser ./log.darshan | grep POSIX_BYTES_WRITTEN | sort -nr -k 5
POSIX   387     6960057185861764086     POSIX_BYTES_WRITTEN     5413869452      /projects/radix
POSIX   452     6960057185861764086     POSIX_BYTES_WRITTEN     5413865644      /projects/radix
POSIX   197     6960057185861764086     POSIX_BYTES_WRITTEN     5413857652      /projects/radix
POSIX   5       6960057185861764086     POSIX_BYTES_WRITTEN     5413852168      /projects/radix
POSIX   451     6960057185861764086     POSIX_BYTES_WRITTEN     5413844532      /projects/radix
POSIX   64      6960057185861764086     POSIX_BYTES_WRITTEN     5413823236      /projects/radix
POSIX   68      6960057185861764086     POSIX_BYTES_WRITTEN     5413788992      /projects/radix
POSIX   195     6960057185861764086     POSIX_BYTES_WRITTEN     5413663132      /projects/radix
POSIX   323     6960057185861764086     POSIX_BYTES_WRITTEN     5413658668      /projects/radix
POSIX   132     6960057185861764086     POSIX_BYTES_WRITTEN     5413648628      /projects/radix
```

If you know what you're looking for, `darshan-parser` can be a quick way to extract important I/O details from a log, e.g., the 10 most heavily written files

Analyzing Darshan logs

- After locating your log, users can utilize Darshan log analysis tools for gaining insights into application I/O behavior:

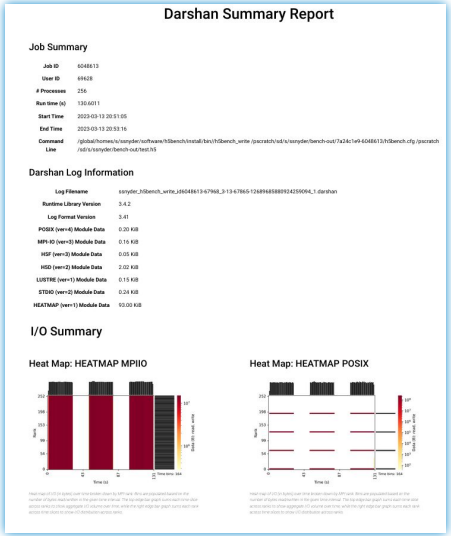


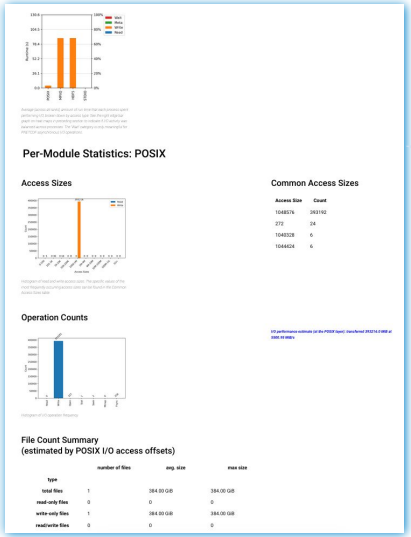


A more user-friendly starting point is the Darshan job summary tool. It can generate a summary report for a log containing useful graphs, tables, and performance estimates describing application I/O behavior

Analyzing Darshan logs

- After locating your log, users can utilize Darshan log analysis tools for gaining insights into application I/O behavior:



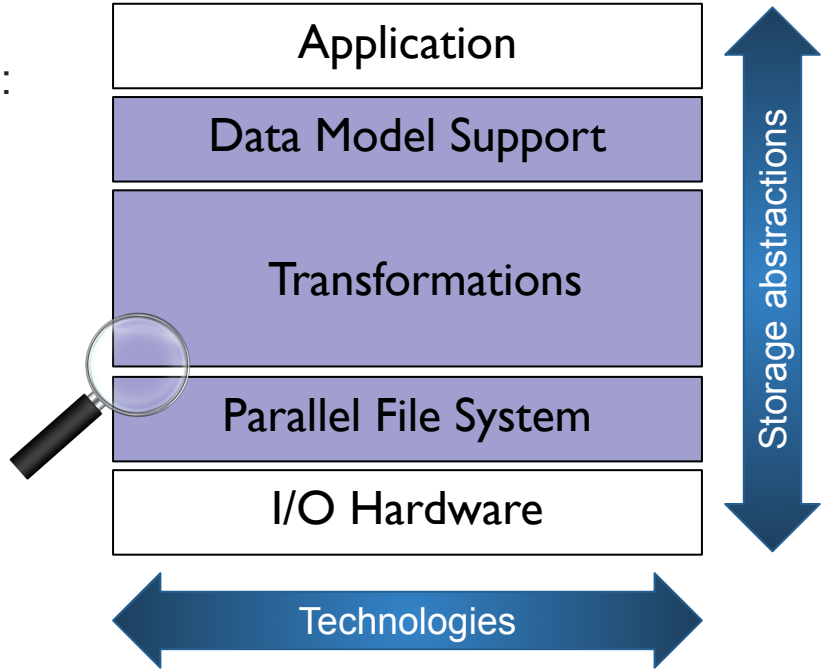


More details on the Darshan job summary tool coming shortly!

Key Darshan instrumentation capabilities

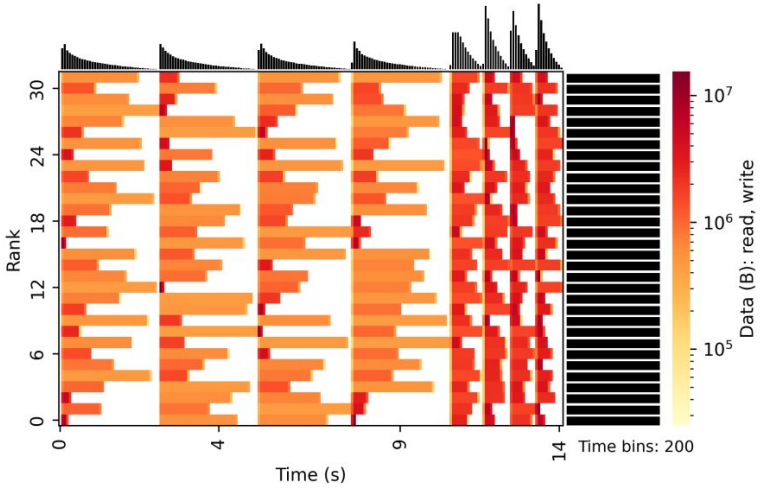
Low-level I/O instrumentation

- Darshan provides in-depth instrumentation of the lower layers of the traditional HPC I/O stack:
  - **MPI-IO** parallel I/O interface
  - **POSIX** file system interface
  - **STDIO** buffered stream I/O interface
  - **Lustre** striping parameters
- Captures fixed set of statistics, properties, and timing info for each file accessed using these interfaces
- Informs on key I/O performance characteristics of foundational components of the HPC I/O stack



Low-level I/O instrumentation

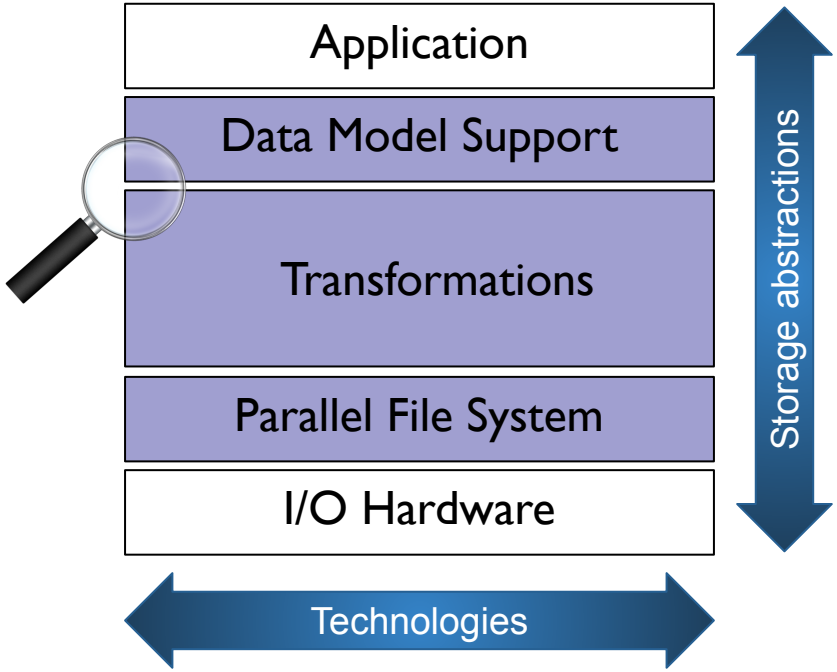
- Beyond its traditional capture mode, Darshan offers key features for obtaining finer-grained details of low-level I/O activity:
  - **Heatmap module**: captures histograms of I/O activity at each process using a fixed size histogram
    - Available for POSIX, MPI-IO, and STDIO interfaces by default in 3.4+ versions of Darshan
  - **DXT modules**: captures full I/O traces at each process using a configurable buffer size
    - Available for POSIX and MPI-IO modules
    - Enabled using `DXT_ENABLE_IO_TRACE` environment variable



Heatmaps showcase application I/O intensity across time, ranks, and interfaces – helpful for identifying hot spots, I/O and compute phases, etc.

High-level I/O library instrumentation

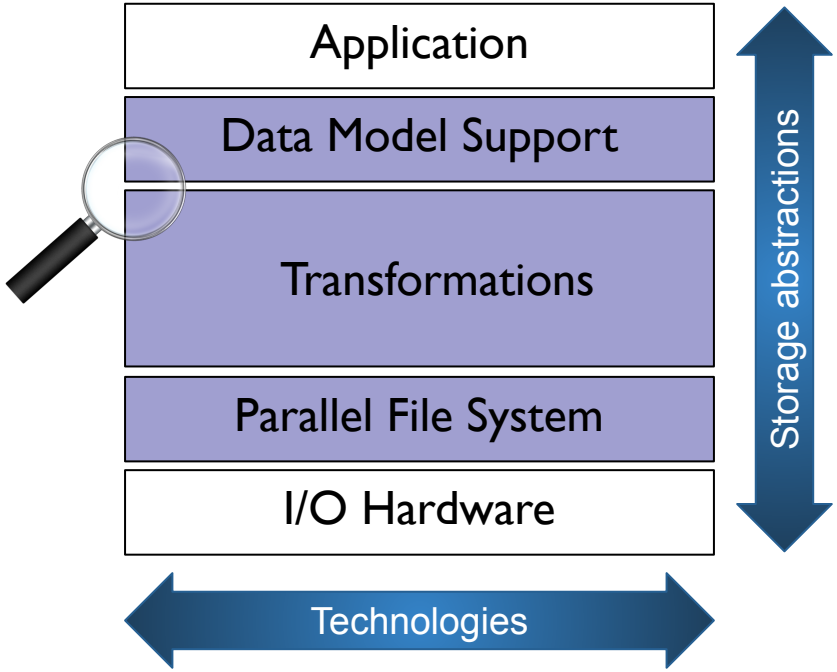
- Darshan similarly provides in-depth instrumentation of popular high-level I/O libraries for HPC
  - **HDF5**: detailed instrumentation of accesses to HDF5 files and datasets available starting in 3.2+ versions
  - **PnetCDF**: detailed instrumentation of accesses to PnetCDF files and variables available starting in 3.4.1+ versions
- Full-stack characterization allows deeper understanding of app usage of I/O libraries, as well as underlying performance characteristics for these usage patterns



High-level I/O library instrumentation

- Darshan similarly provides in-depth instrumentation of popular high-level I/O libraries for HPC
  - **HDF5**: detailed instrumentation of accesses to HDF5 files and datasets available starting in 3.2+ versions
  - **PnetCDF**: detailed instrumentation of accesses to PnetCDF files and variables available starting in 3.4.1+ versions

PnetCDF module contributed by
Wei-Keng Liao (NWU)



PyDarshan log analysis framework

PyDarshan log analysis framework

- Darshan has traditionally offered only the C-based darshan-util library and a handful of corresponding tools to users for log file analysis
  - Complicates development of custom Darshan analysis tools
- PyDarshan developed to simplify the interfacing of analysis tools with log data
  - Use Python CFFI module to define bindings to the native darshan-utils C API
  - Expose Darshan log data as dictionaries, pandas dataframes, and NumPy arrays
- PyDarshan should provide a richer ecosystem for development of Darshan log analysis tools, either by users or by the Darshan team

Available via PyPI or Spack:

- ★ "`pip install darshan`"
- ★ "`spack install py-darshan`"

PyDarshan development led by Jakob Luttgau (UTK), Tyler Reddy and Nik Awtrey (LANL)

PyDarshan job summary tool

- PyDarshan includes a new job summary tool that is replacing the original `darshan-job-summary.pl` script
  - Generates detailed HTML reports summarizing application I/O behavior using different plots, graphs, and statistics
  - Builds off popular Python libraries like `matplotlib` (plotting), `seaborn` (plotting), and `mako` (HTML templating)
- Users can generate summary reports for a given Darshan log file using the following command:
  - '**`python -m darshan summary <path_to_log_file>`**'
  - Generates output HTML report matching input log file name

PyDarshan job summary tool

Detailed job metadata

Job Summary

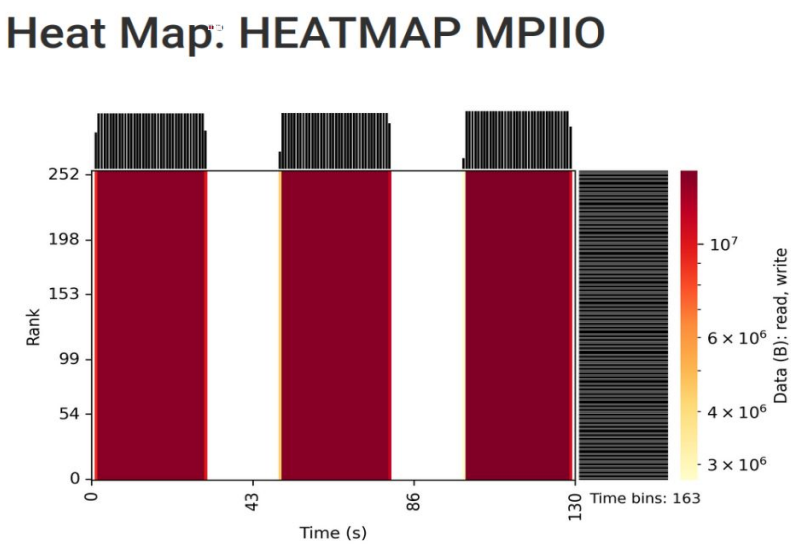
Job ID	6553753
User ID	69628
# Processes	256
Run time (s)	96.3942
Start Time	2023-03-27 11:15:18
End Time	2023-03-27 11:16:55
Command Line	/global/homes/s/ssnyder/software/h5bench/install/bin//h5bench_write /pscratch/sd/s/ssnyder/bench-out2/c1802902-6553753 /h5bench.cfg /pscratch/sd/s/ssnyder/bench-out2/test.h5

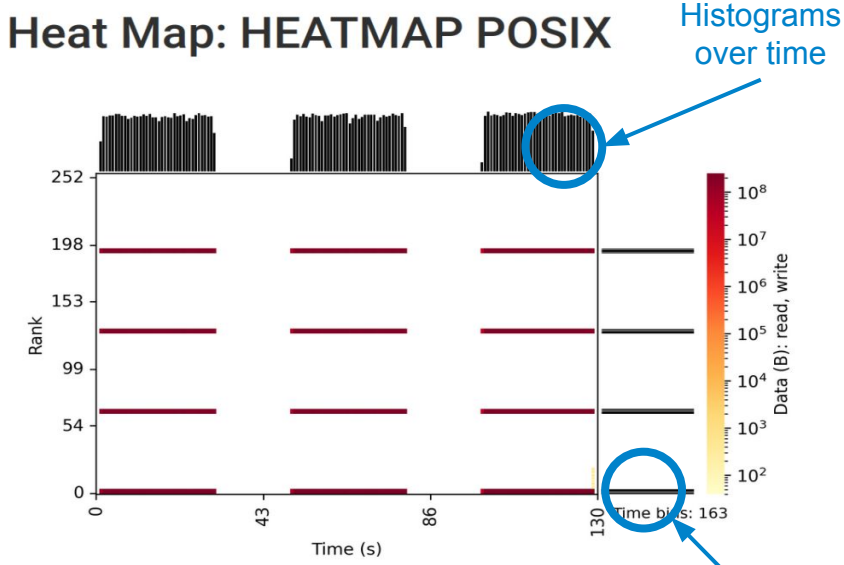
256 process (4 node) h5bench[1] runs on NERSC Perlmutter. h5bench contains lots of parameters (e.g., contiguous vs interleaved accesses, independent vs collective I/O, synchronous vs asynchronous I/O, etc.) for controlling characteristics of generated HDF5 workloads.

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

PyDarshan job summary tool

Heatmaps for visualizing I/O activity

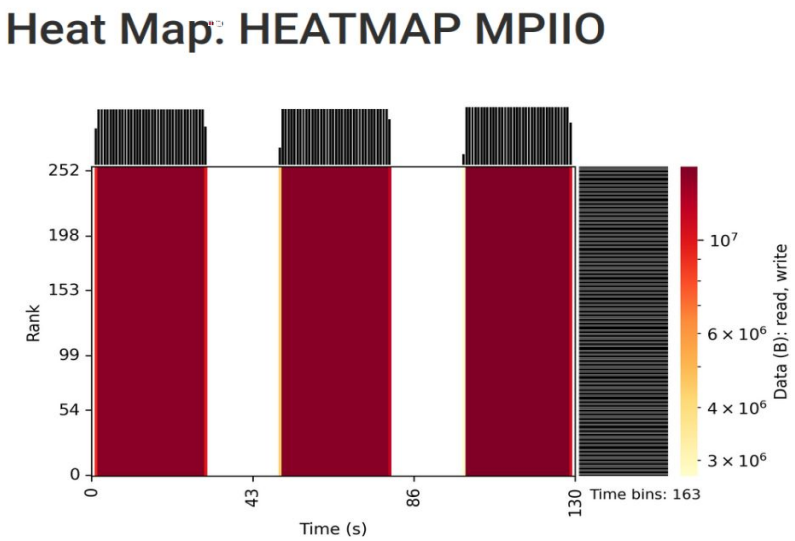


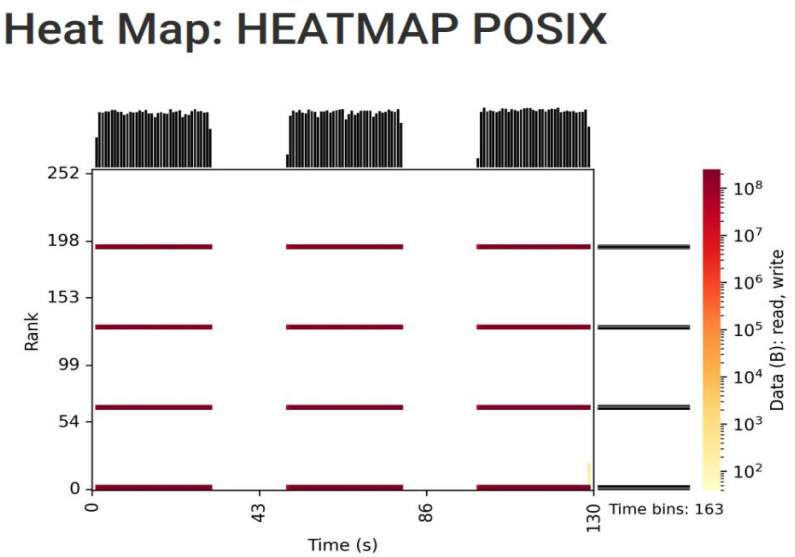


Analyzing I/O behavior over time, ranks, and interfaces can offer key insights into application I/O behavior.

PyDarshan job summary tool

Heatmaps for visualizing I/O activity

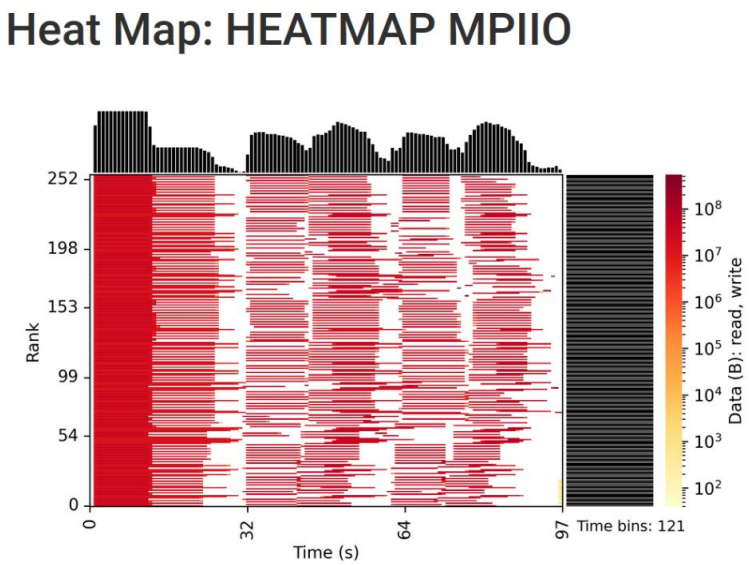


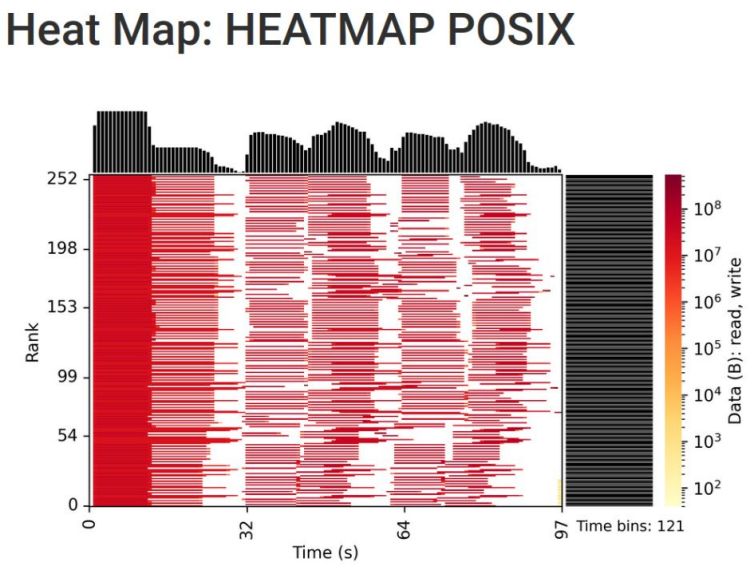


This example heatmap illustrates a typical MPI-IO collective I/O pattern. All MPI ranks perform MPI-IO operations (left), but only a subset of "aggregators" access the file via POSIX operations (right).

PyDarshan job summary tool

Heatmaps for visualizing I/O activity



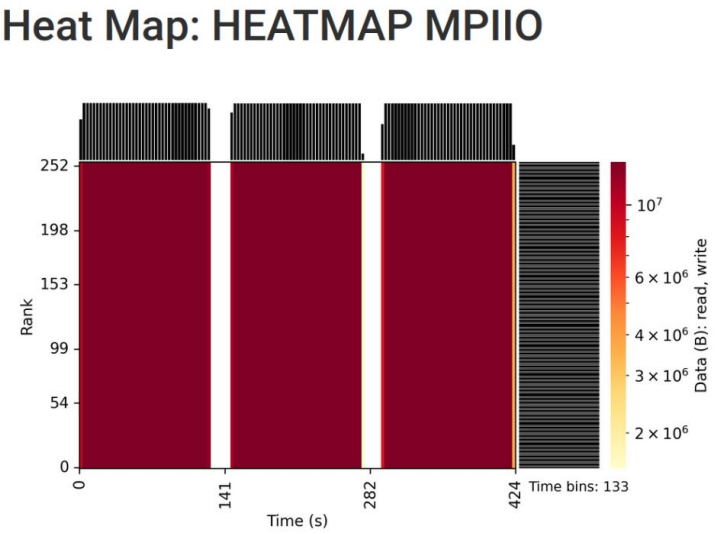


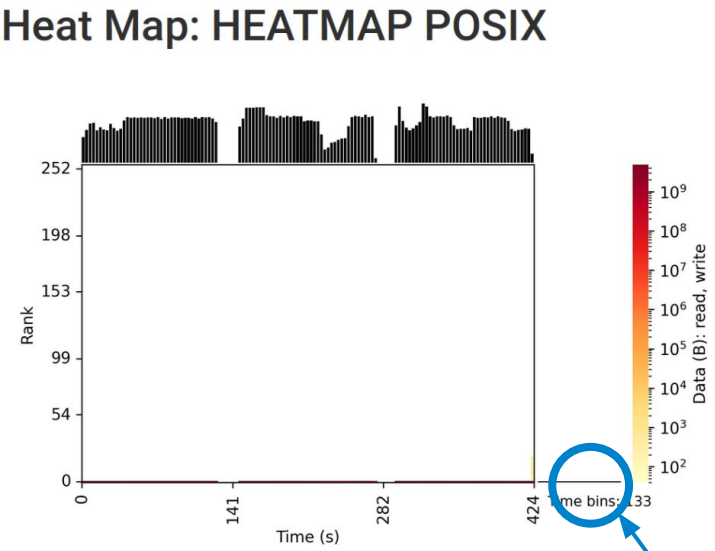
Heatmaps can help quickly detect common I/O pitfalls.

I could have sworn I enabled collective I/O in HDF5.

PyDarshan job summary tool

Heatmaps for visualizing I/O activity





Heatmaps can help quickly detect common I/O pitfalls.

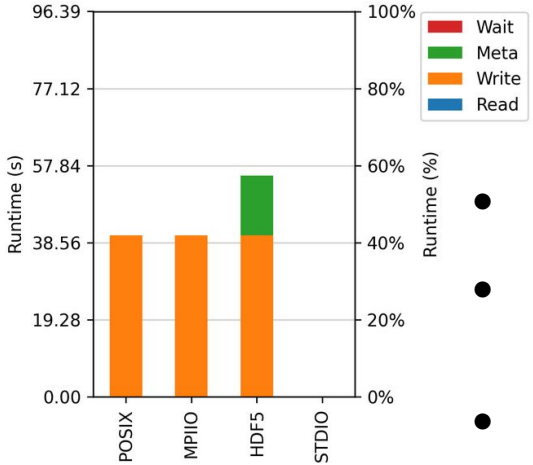
Oops, I enabled collective I/O, but forgot to tell Lustre to use more than one stripe.

All I/O funneled through rank 0

PyDarshan job summary tool

Average I/O cost across APIs

independent

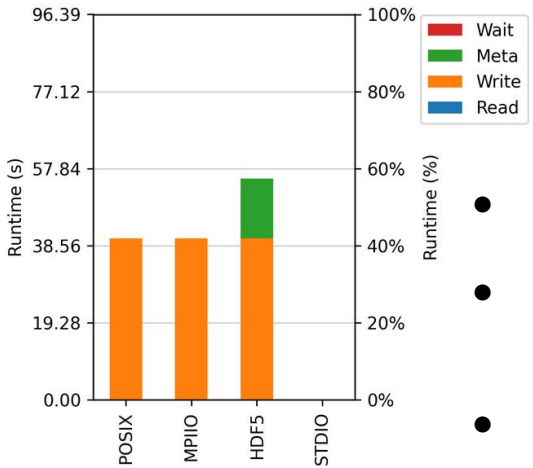


- POSIX write dominates
- MPI-IO/HDF5 negligible write overheads
- HDF5 incurs additional metadata overhead (flushes?)

PyDarshan job summary tool

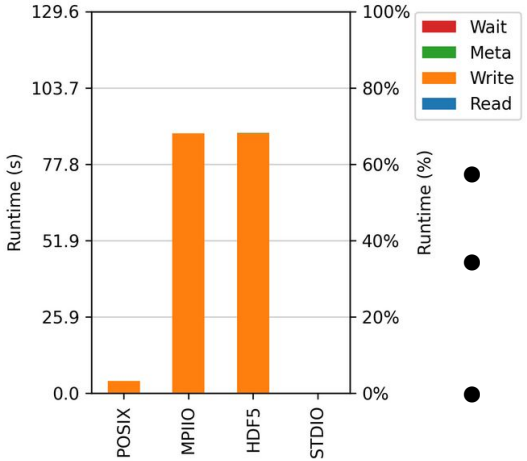
Average I/O cost across APIs

independent



- POSIX write dominates
- MPI-IO/HDF5 negligible write overheads
- HDF5 incurs additional metadata overhead (flushes?)

collective

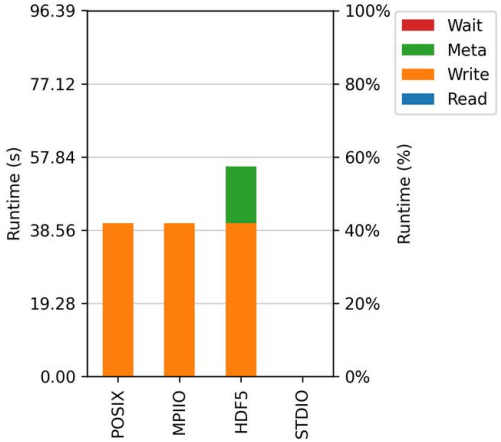


- POSIX write nearly negligible
- MPI-IO collective algorithm cost dominates
- No comparable HDF5 metadata overhead

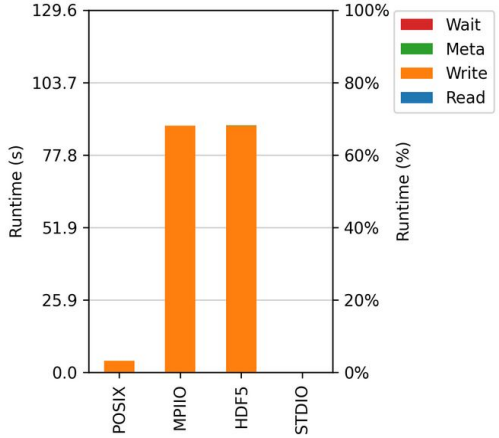
PyDarshan job summary tool

Average I/O cost across APIs

independent



collective



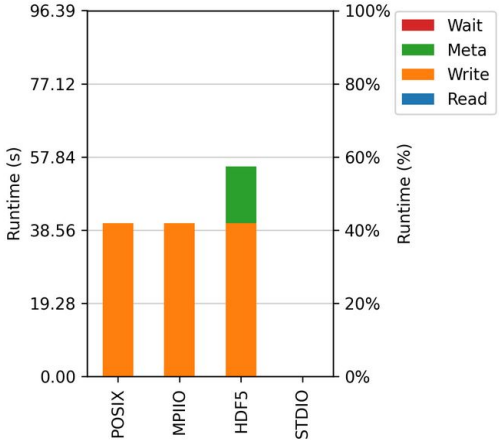
In this head-to-head comparison, independent mode (~55 seconds avg. I/O time) actually performs better than collective (~85 seconds avg. I/O time).

Collective I/O behavior affected by many factors (access patterns, FS parameters, job scale, MPI-IO parameters, dynamic system state, etc.).

PyDarshan job summary tool

I/O performance estimates

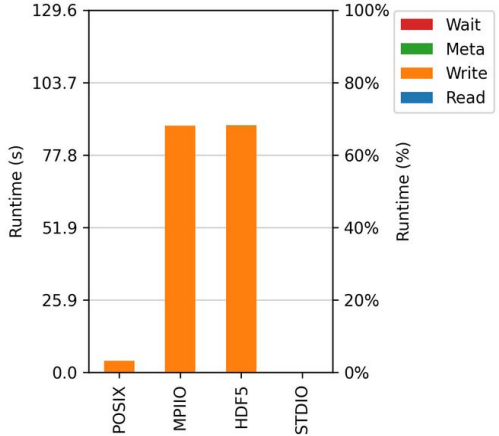
independent



I/O performance estimate (at the MPI-IO layer): transferred 393216.0 MiB at 7108.62 MiB/s

I/O performance estimate (at the POSIX layer): transferred 393216.0 MiB at 7112.29 MiB/s

collective



I/O performance estimate (at the MPI-IO layer): transferred 393216.0 MiB at 4451.17 MiB/s

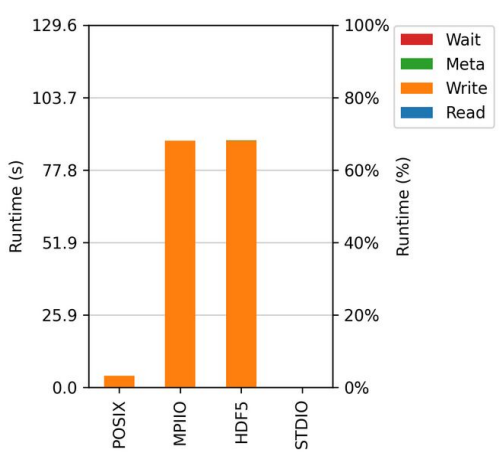
I/O performance estimate (at the POSIX layer): transferred 393216.0 MiB at 5571.27 MiB/s

PyDarshan job summary tool

I/O performance estimates

While I/O cost plots are based on averages across all processes, performance estimates are based on the slowest observed process.

Average cost metrics aren't the greatest at quantifying collective I/O – we are working to integrate more effective metrics into Darshan (e.g. cost by slowest process).



I/O performance estimate (at the MPI-IO layer): transferred 393216.0 MiB at 4451.17 MiB/s

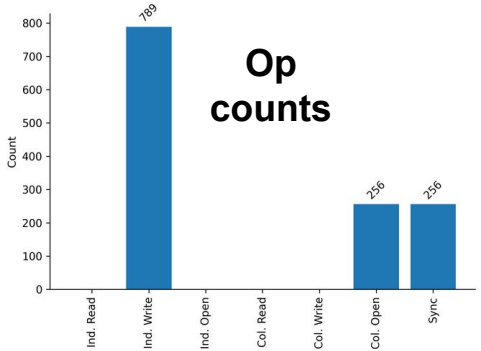
I/O performance estimate (at the POSIX layer): transferred 393216.0 MiB at 5571.27 MiB/s

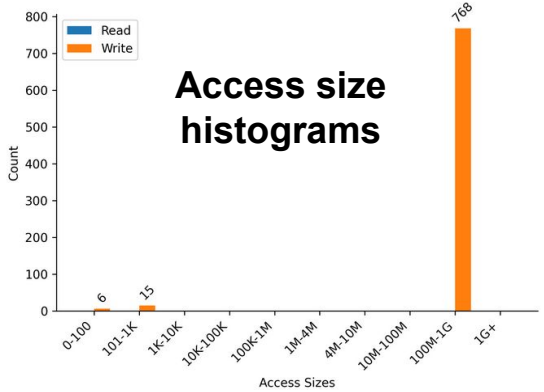
PyDarshan job summary tool

More per-API I/O stats

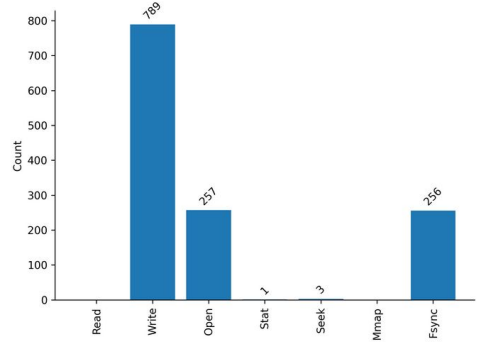
independent mode

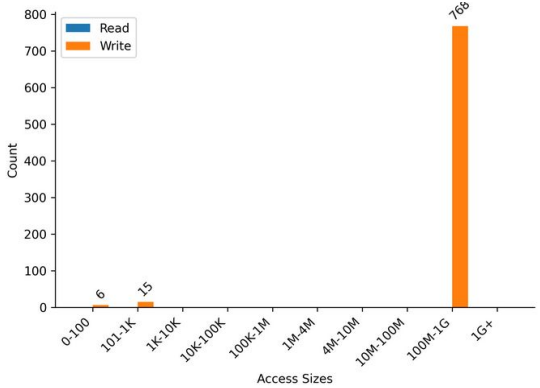
MPI-IO





POSIX





MPI-IO independent operations mostly map 1-to-1 with POSIX file operations.

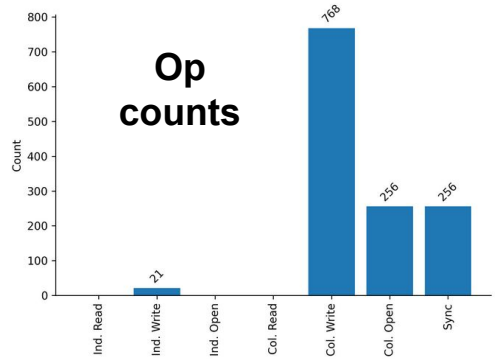
Access sizes mirror the 512 MiB accesses being made at the HDF5 layer.

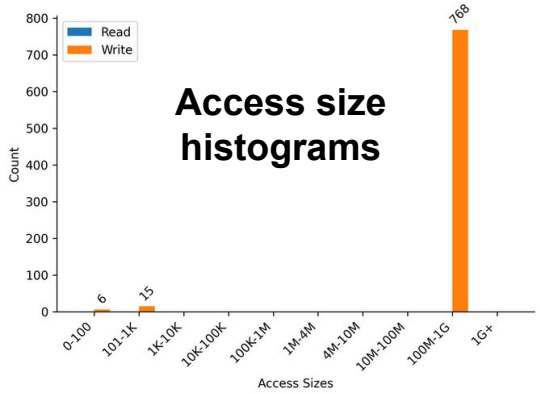
PyDarshan job summary tool

More per-API I/O stats

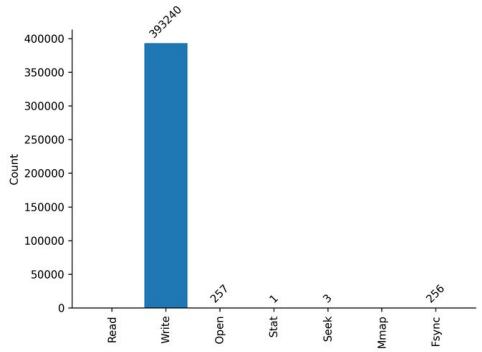
collective mode

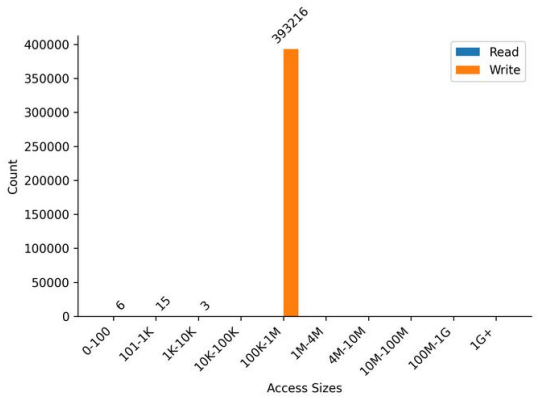
MPI-IO





POSIX





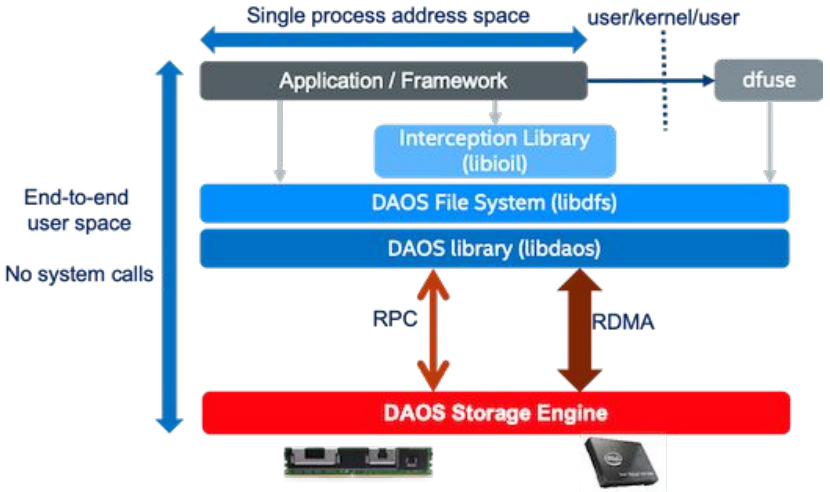
768 MPI-IO collective operations transform into nearly 400K POSIX file operations.

POSIX access sizes now match the collective I/O algorithm buffer size (which equals the Lustre stripe width, 1 MiB).

What’s next for Darshan?

DAOS instrumentation

- 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
- Darshan instrumentation will provide valuable insights into various ways apps and I/O middleware can utilize DAOS

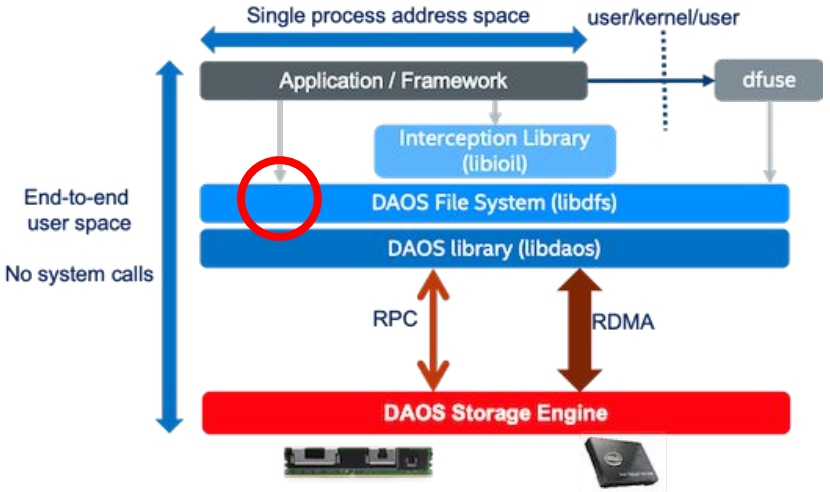


Various access methods for DAOS users.

Figure courtesy of Intel

DAOS instrumentation

- 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
- Darshan instrumentation will provide valuable insights into various ways apps and I/O middleware can utilize DAOS
  - **Direct usage of POSIX-like DAOS file system (libdfs) interface**

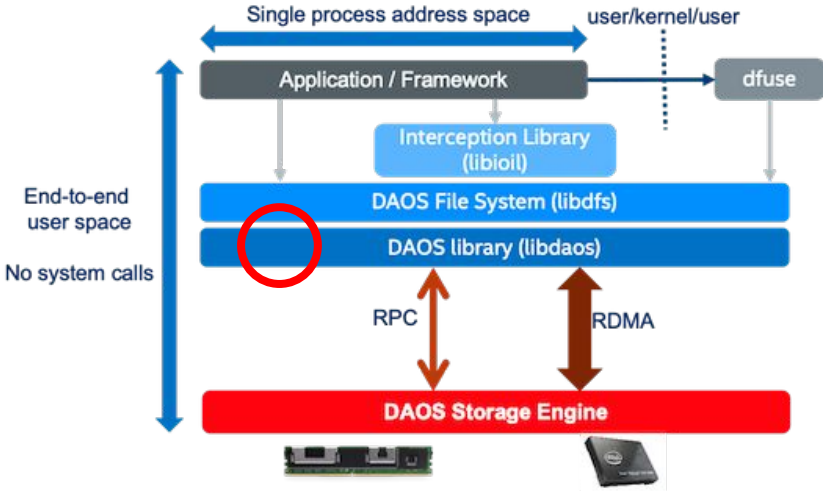


Various access methods for DAOS users.

Figure courtesy of Intel

DAOS instrumentation

- 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
- Darshan instrumentation will provide valuable insights into various ways apps and I/O middleware can utilize DAOS
  - Direct usage of POSIX-like DAOS file system (libdfs) interface
  - **Direct usage of native DAOS object (libdaos) interface**

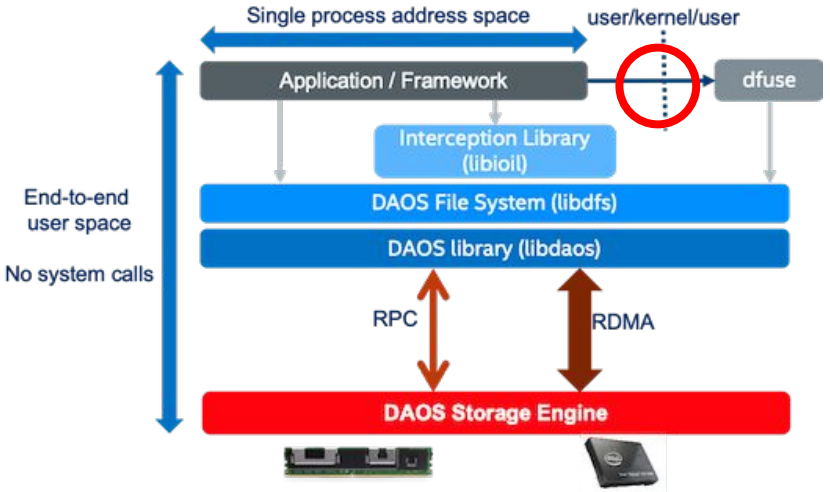


Various access methods for DAOS users.

Figure courtesy of Intel

DAOS instrumentation

- 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
- Darshan instrumentation will provide valuable insights into various ways apps and I/O middleware can 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**

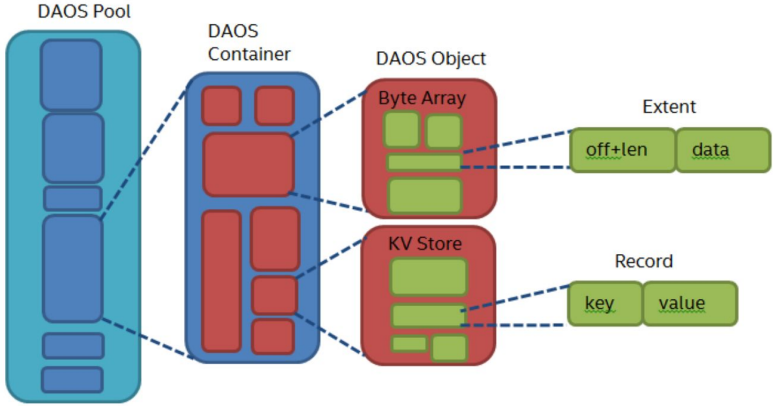


Various access methods for DAOS users.

Figure courtesy of Intel

DAOS instrumentation

- The libdfs file interface is a natural fit for Darshan's traditional "record per-file" instrumentation strategy
- But, instrumenting the native libdaos object interface is a bit more complicated, as DAOS objects can take multiple forms
  - Array objects
    - Extent-based access, similar to files
  - Key-val objects
    - Data accessed using arbitrary keys



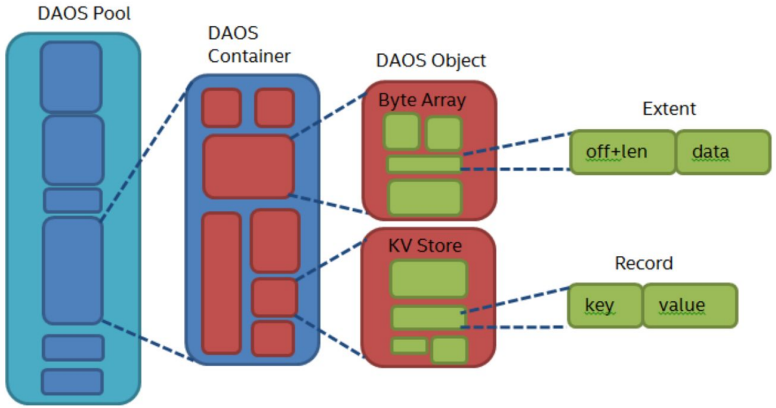
DAOS storage model. DAOS objects can be accessed using either key-val or array interfaces.

Figure courtesy of Intel

DAOS instrumentation

- The libdfs file interface is a natural fit for Darshan's traditional "record per-file" instrumentation strategy
- But, instrumenting the native libdaos object interface is a bit more complicated, as DAOS objects can take multiple forms
  - Array objects
    - Extent-based access, similar to files
  - Key-val objects
    - Data accessed using arbitrary keys

What helpful information could Darshan provide regarding key access distributions of DAOS key-val objects?



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

Figure courtesy of Intel

PyDarshan analysis enhancements

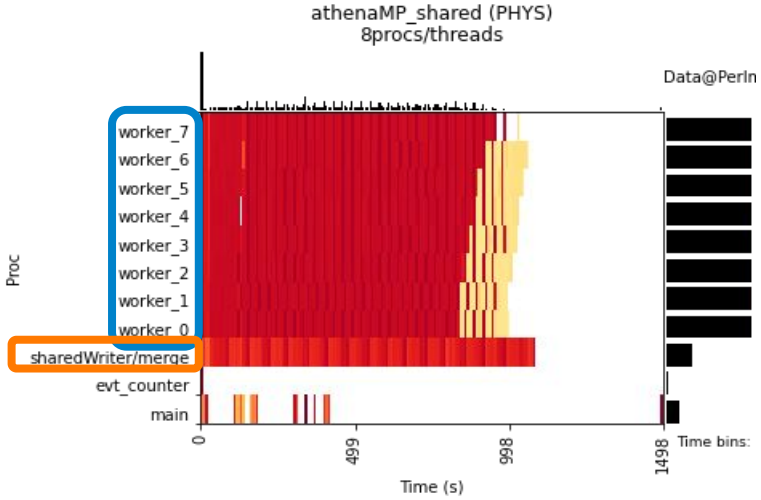
Enabling multi-log analysis

- Darshan analysis tools have traditionally operated on a single input log, but analysis across multiple logs is useful in different contexts
  - Analysis of workflows
  - Analysis of arbitrary log collections
- We would like to support new PyDarshan analysis capabilities enabling reporting on I/O behavior beyond the context of a single “job”

PyDarshan analysis enhancements

Enabling multi-log analysis

- Darshan analysis tools have traditionally operated on a single input log, but analysis across multiple logs is useful in different contexts
  - **Analysis of workflows**
  - Analysis of arbitrary log collections
- We would like to support new PyDarshan analysis capabilities enabling reporting on I/O behavior beyond the context of a single “job”



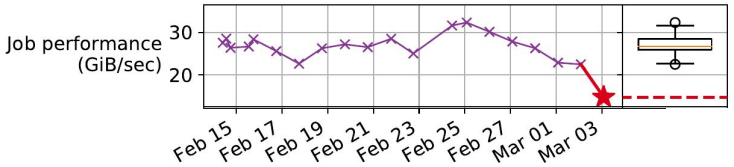
Heatmap visualization of an HEP multiprocess analysis workflow (AthenaMP). 8 **workers** read input data, while a **shared writer** process writes all worker output data from shared memory.

Athena analysis contributed by Rui Wang (ANL).

PyDarshan analysis enhancements

Enabling multi-log analysis

- Darshan analysis tools have traditionally operated on a single input log, but analysis across multiple logs is useful in different contexts
  - Analysis of workflows
  - **Analysis of arbitrary log collections**
- We would like to support new PyDarshan analysis capabilities enabling reporting on I/O behavior beyond the context of a single “job”



Visualizing overall app I/O performance over time to determine changes in I/O behavior. Borrowed from an I/O analysis tool from the TOKIO project that predated PyDarshan.

https://www.anl.gov/mcs/tokio-total-knowledge-of-io

G.K. Lockwood et al. "UMAMI: a recipe for generating meaningful metrics through holistic I/O performance analysis." PDSW’17.

PyDarshan analysis enhancements

From write-optimized to analysis-friendly formats

- Darshan's native log format is optimized for efficient writing by apps
  - Minimizes instrumentation overheads, but *creates problems for log analysis tools*
- We want to explore popular industry solutions like Apache Parquet/Arrow to help transform Darshan data into more analysis-friendly formats
  - Columnar format can save storage/memory and speed up analysis tasks, particularly when analyzing lots of Darshan data (e.g., all logs collected at a facility)
  - Integrations with popular data analysis frameworks like pandas and Dask

Wrapping up

- Darshan is an invaluable tool for HPC application scientists, facilities, and I/O researchers for better understanding application I/O behavior
  - Detailed instrumentation of application access to multiple layers of the HPC I/O stack
    - High-level I/O library usage
    - MPI-IO transformations
    - File system access (i.e., POSIX)
  - Helpful tools for extracting salient data from Darshan logs and summarizing for users

- Please reach out with any questions, comments, or feedback!

- Darshan website, docs: https://www.mcs.anl.gov/research/projects/darshan/
- Source code, issue tracking: https://github.com/darshan-hpc/darshan
- Darshan-users mailing list: darshan-users@lists.mcs.anl.gov

Acknowledgement

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research also used resources and data generated from resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.