Argonne NATIONAL LABORATORY

IO SLEUTHING: ADVENTURES IN BANDWIDTH

ROB LATHAM
Research Software Developer
Argonne National Laboratory

https://github.com/radix-io/io-sleuthing

20 March 2023
Argonne National Lab (and virtual)

ABOUT ME

- "It's not Dr"
- Undergraduate at Lehigh University
  - Eternally grateful for the guy down the hall who said "You should try Linux"
- First job at a Linux cluster vendor
  - Big expensive vector supercomputers (e.g. Cray) replaced with cheap commodity racks of Linux nodes.
  - Installed a lot of Argonne-developed software on the machines
- Rob Ross: "hey, we're looking for developers if you know anyone"
  - I'm still here, and still working for Rob
- Storage and I/O with a focus on application experience
  - Lots of work in MPI-IO, Parallel-NetCDF, tutorials
- 2022 BSSW fellow (which supported this material)

THEMES

- “Do I have an I/O problem?”
  - Performance
- “Where is my I/O problem?”
  - Diagnosis
- “How do I fix my I/O problem?”
  - Tuning
- “I want to learn even more about I/O problems!”
  - Attempting a “living presentation”
  - https://github.com/radix-io/io-sleuthing
    - Job scripts, experiments, configurations:
    - experiment yourself: let’s talk about what you find out

“A supercomputer is a device for turning compute-bound problems into I/O-bound problems.”

- Ken Batcher

“There is no physics without I/O.”

– Anonymous Physicist
SciDAC Conference
June 17, 2009

(I think he might have been kidding.)

“Very few large scale applications of practical importance are NOT data intensive.”

– Alok Choudhary, IESP, Kobe Japan, April 2012

(I know for sure he was not kidding.)

TODAY'S OUTLINE

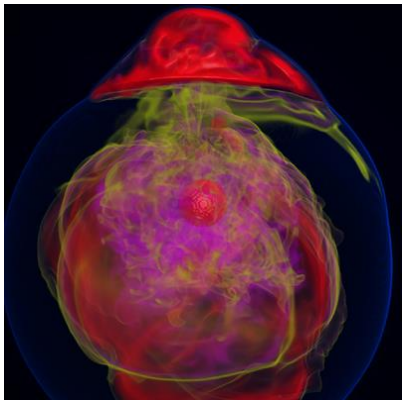
- Part One
  - Challenges
  - File systems
  - Benchmarking
  - Demo: Lustre + striping
  - Darshan
- Part Two
  - MPI-IO
  - High-level I/O libraries
    - Parallel-NetCDF
    - HDF5
  - Machine Learning workloads

STAGE SETTING: CONCEPTS, TERMS, ETC

COMPUTATIONAL SCIENCE

- Computer simulation as a tool promotes greater understanding of the real world
  - Complements experimentation and theory
- Problems are increasingly computationally expensive
  - Large parallel machines are needed to perform calculations
  - Leveraging parallelism in all phases is critical
- Data access is a huge challenge and includes
  - Using parallelism to obtain performance
  - Finding usable, efficient, and portable interfaces
  - Understanding and tuning I/O

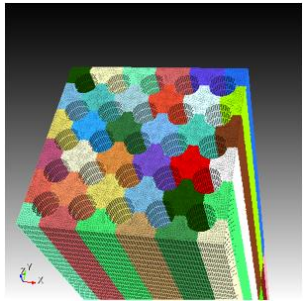
HPE Polaris system at Argonne National Laboratory.

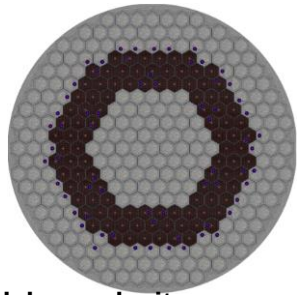


Visualization of entropy in Terascale Supernova Initiative application. Image from Kwan-Liu Ma's visualization team at UC Davis.

APPLICATION DATASET COMPLEXITY VS. I/O

- I/O systems have very simple data models
  - Tree-based hierarchy of containers
  - Some containers have streams of bytes (files)
  - Others hold collections of other containers (directories or folders)
- Applications have data models appropriate to domain
  - Multidimensional typed arrays, images composed of scan lines, records of variable length
  - Headers, attributes on data
- Someone has to map from one to the other!





Model complexity: Spectral element mesh (top) for thermal hydraulics computation coupled with finite element mesh (bottom) for neutronics calculation

Scale complexity: Spatial range from the reactor core, in meters, to fuel pellets, in millimeters

Images from T. Tautges (Argonne) (upper left), M. Smith (Argonne) (lower left), and K. Smith (MIT) (right).

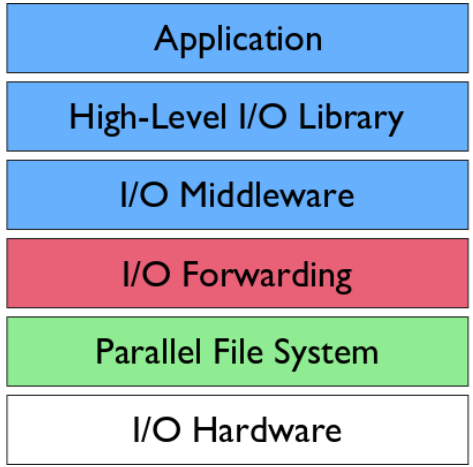
I/O FOR COMPUTATIONAL SCIENCE

High-Level I/O Library
maps application abstractions onto storage abstractions and provides data portability.

HDF5, Parallel netCDF, ADIOS

I/O Forwarding
bridges between app. tasks and storage system and provides aggregation for uncoordinated I/O.

IBM ciod, IOFSL, Cray DVS



I/O Middleware
organizes accesses from many processes, especially those using collective I/O.

MPI-IO

Parallel File System
maintains logical space and provides efficient access to data.

PVFS, PanFS, GPFS, Lustre

Additional I/O software provides improved performance and usability over accessing the parallel file system directly. Reduces or (ideally) eliminates need for optimization in application codes.

STORAGE

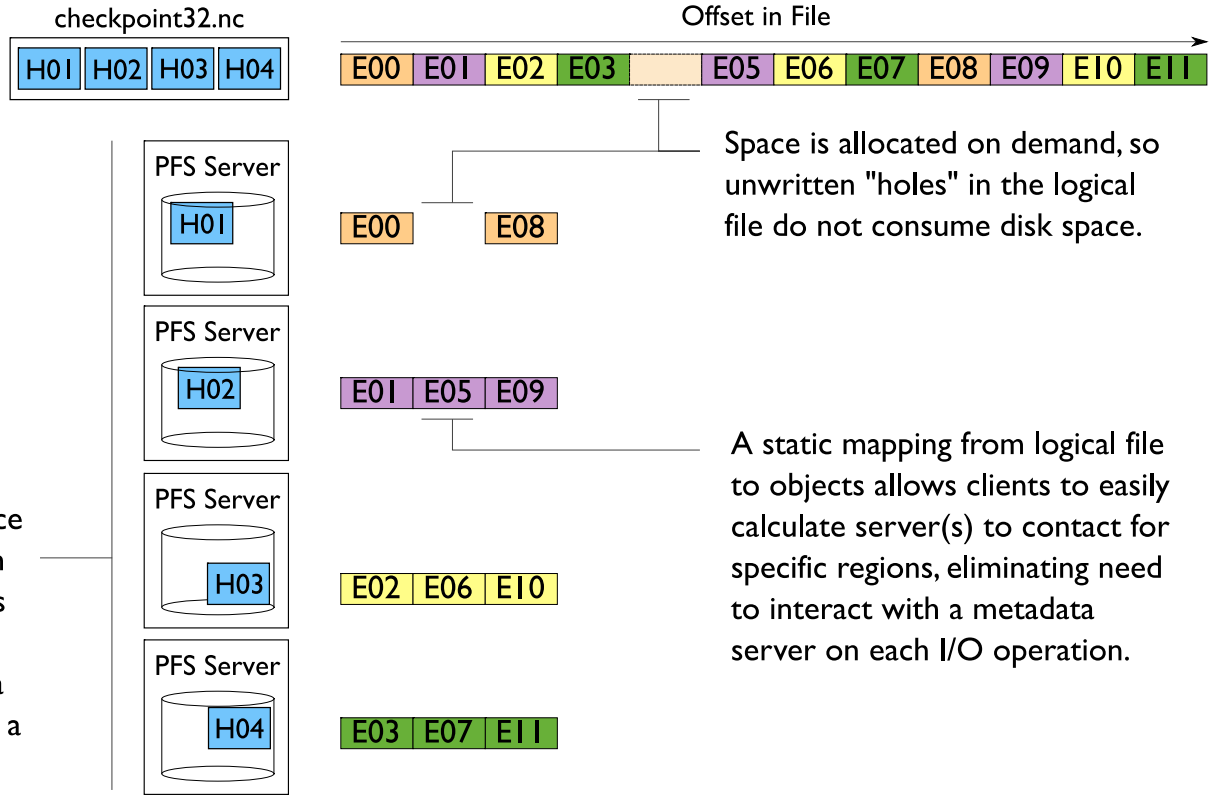
- File systems, blob storage, databases, keyval stores
- Only have a half day, so let's focus on parallel file systems
  - Specifically, Lustre, but you might encounter GPFS or others
- Software abstractions:
  - "performance portability"
  - almost there but not quite

DATA DISTRIBUTION IN PARALLEL FILE SYSTEMS

Logically a file is an extendable sequence of bytes that can be referenced by offset into the sequence.

Metadata associated with the file specifies a mapping of this sequence of bytes into a set of objects on PFS servers.

Extents in the byte sequence are mapped into objects on PFS servers. This mapping is usually determined at file creation time and is often a round-robin distribution of a fixed extent size over the allocated objects.



Space is allocated on demand, so unwritten "holes" in the logical file do not consume disk space.

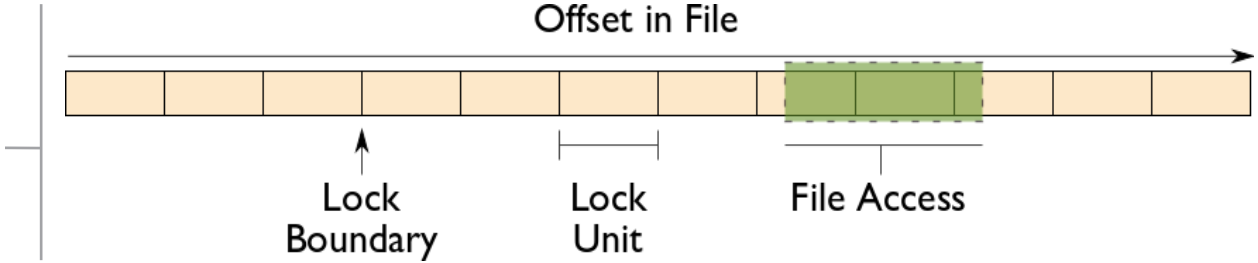
A static mapping from logical file to objects allows clients to easily calculate server(s) to contact for specific regions, eliminating need to interact with a metadata server on each I/O operation.

LOCKING IN PARALLEL FILE SYSTEMS

Most parallel file systems use **locks** to manage concurrent access to files

- Files are divided into lock units aligned with blocks or stripes
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access
- These locks occur behind the scenes: different from locks an application might call explicitly

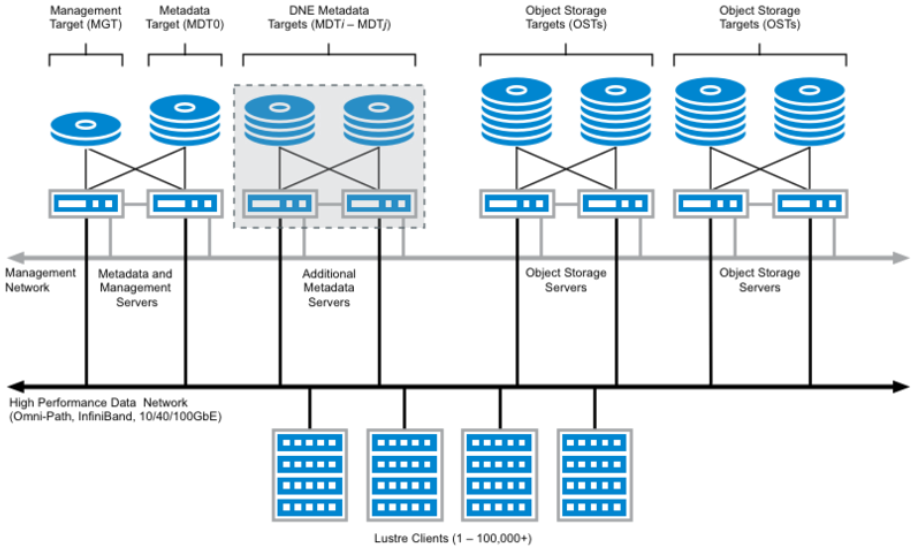
If an access touches any data in a lock unit, the lock for that region must be obtained before access occurs.



FILESYSTEMS: LUSTRE

https://www.lustre.org/

- Metadata servers (MDT)
  - E.g File creation
- Storage servers (OST)
  - Data lives here
  - So does all the parallelism
- Clients
  - POSIX interface (`open, write, read, close`)



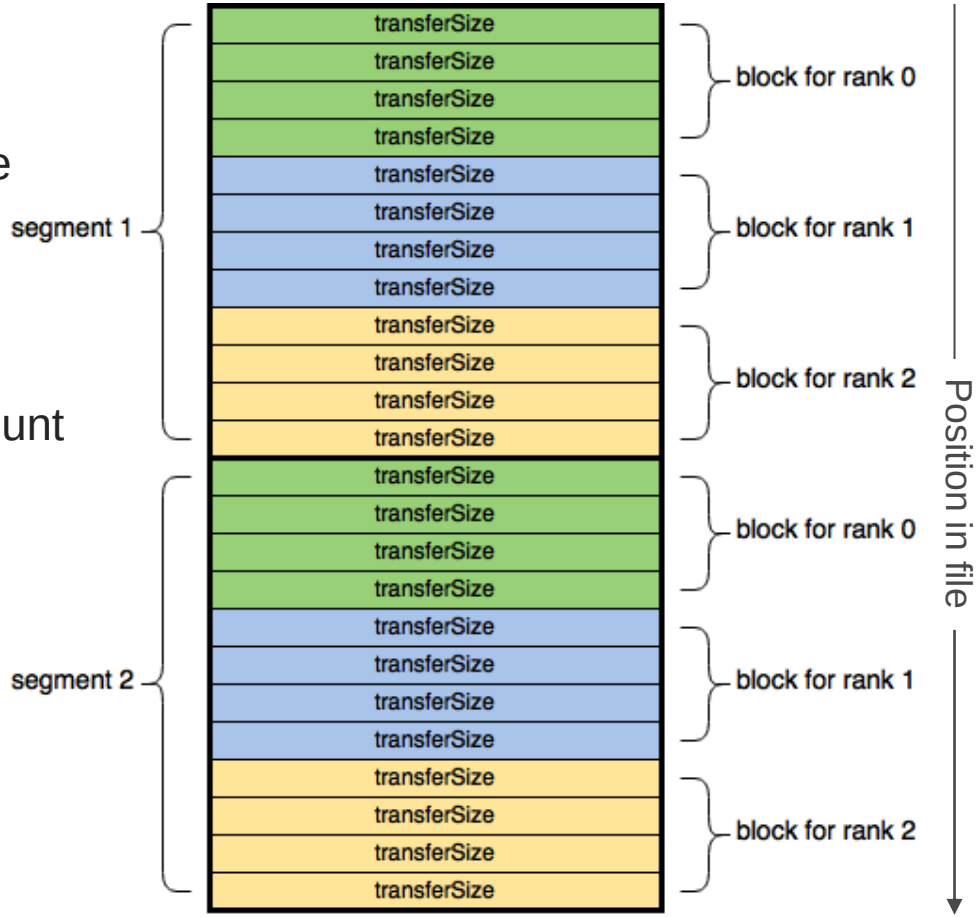
BASIC THROUGHPUT BENCHMARKING WITH IOR

Thanks to Glenn Lockwood for much of this material

THE IOR BENCHMARK

- MPI application benchmark
  - reads and writes data in configurable ways
  - I/O pattern can be interleaved or random
- Input:
  - transfer size, block size, segment count
  - interleaved or random
- Output: Bandwidth and IOPS
- Configurable backends
  - POSIX, STDIO, MPI-IO
  - HDF5, PnetCDF, S3, rados

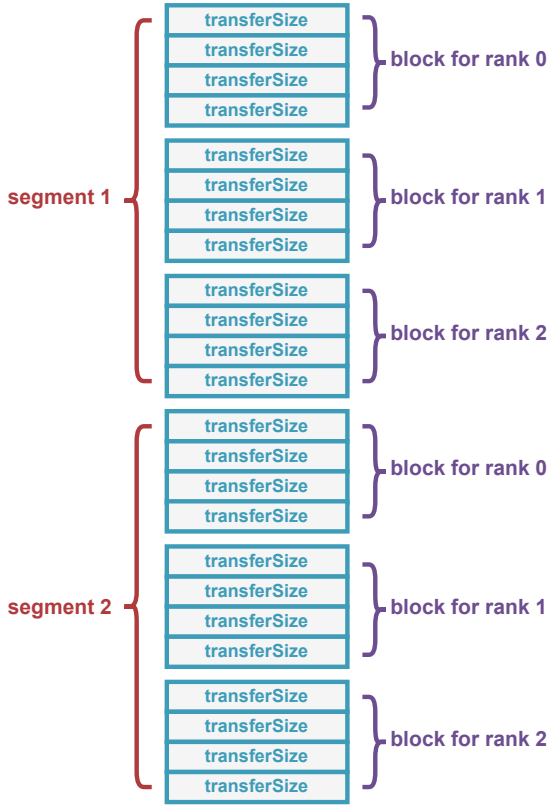
https://github.com/hpc/ior



FIRST ATTEMPT AT BENCHMARKING AN I/O PATTERN

- 120 GB/sec Lustre file system
- 4 compute nodes, 16 ppn, 200 Gb/s NIC
- Performance makes no sense
  - write performance is awful
  - read performance is mind-blowingly good

```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64
...
Operation   Max(MiB)
write         9539.38
read        492123.04
```



TRY BREAKING UP OUTPUT INTO MULTIPLE FILES

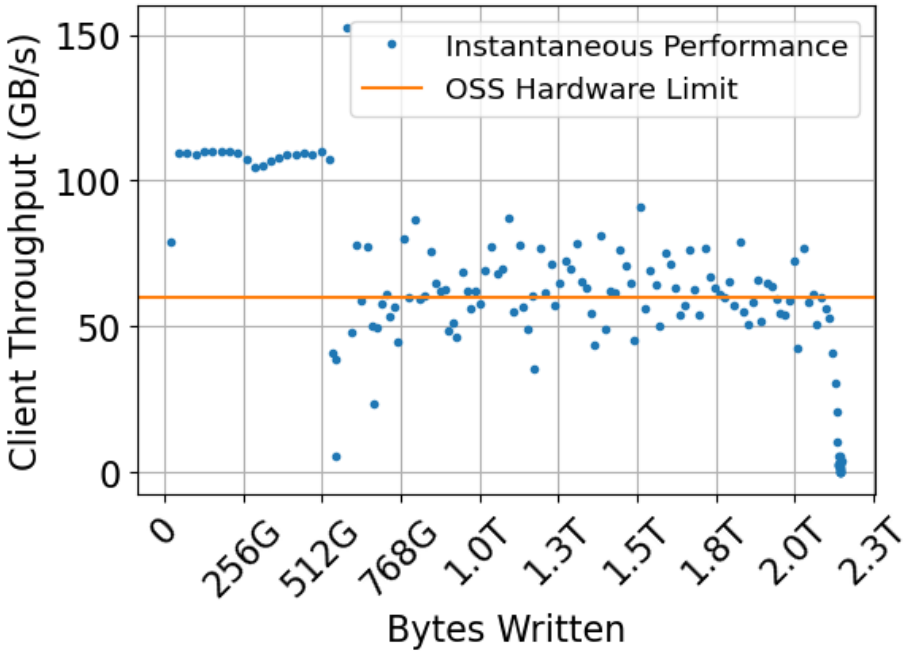
- IOR provides `-F` option to make each rank read/write to its own file instead of default single-shared-file I/O
  - Reduces lock contention within file
  - Can cause metadata load at scale
- Problem: > 400 GB/sec from 4 OSSes is faster than light

```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F
...
Operation   Max(MiB)
write        72852.83
read        481168.60
```

EFFECT OF PAGE CACHE ON MEASURED I/O BANDWIDTH

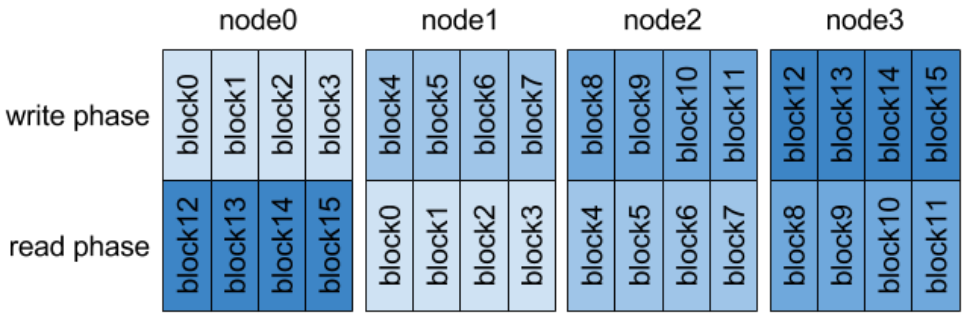
- Unused compute node memory to cache file contents
- Can dramatically affect I/O
  - Writes:
    - only land in local memory at first
    - reordered and sent over network later
    - `max_dirty_mb` and `max_pages_per_rpc` (Lustre)
    - `dirty_background_ratio` and `dirty_ratio` (NFS)
  - Reads:
    - come out of local memory if data is already there
    - read-after-write = it's already there
    - readahead also exists

Time-resolved write bandwidth
4 clients / 256 GiB DDR each
2 OSTs / 60 GB/s write spec



AVOID READING FROM CACHE WITH RANK SHIFTING

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

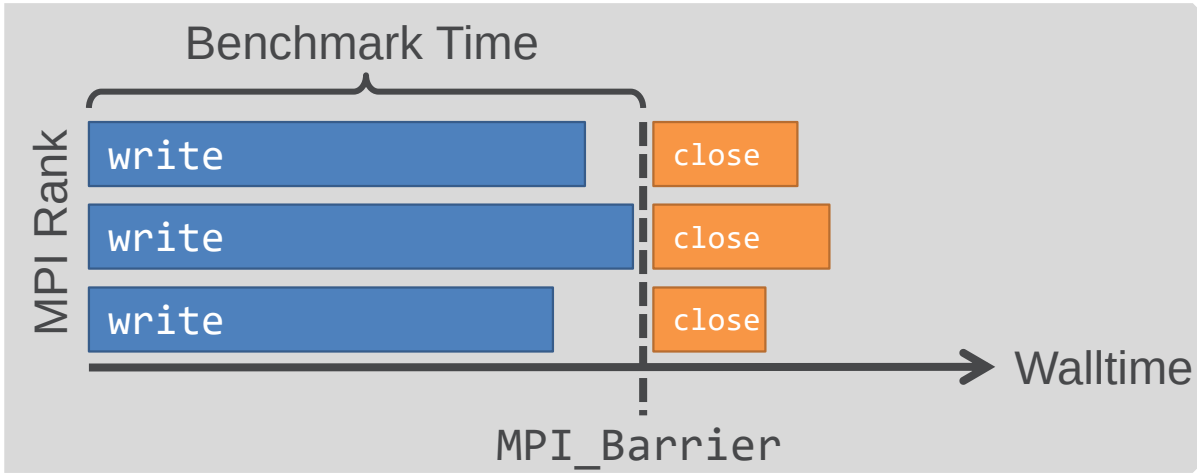



```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F -C
...
Operation   Max(MiB)
write       63692.33
read        28303.09
```

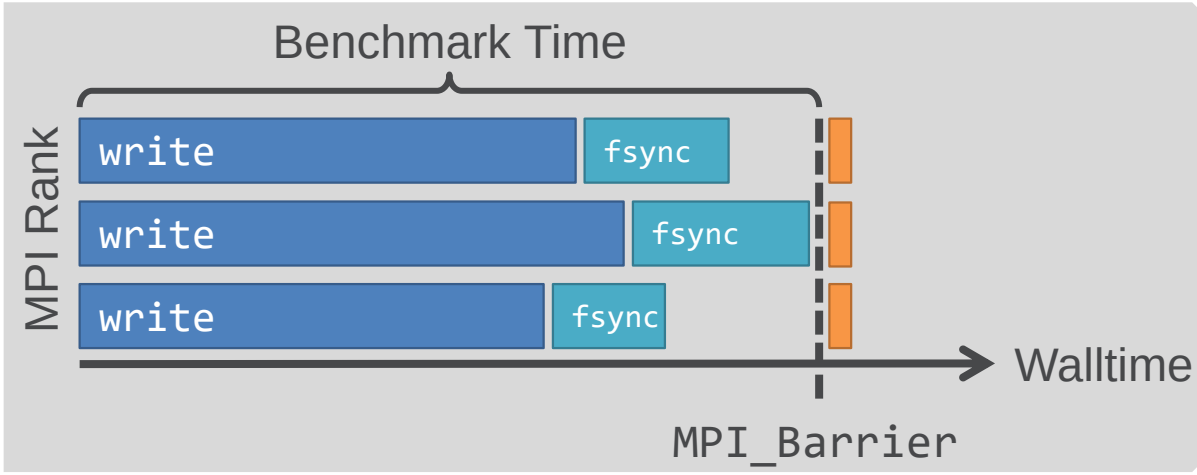
FORCE SYNC TO ACCOUNT FOR WRITE CACHE EFFECTS

- Default: benchmark timer stops when last write completes
- Desired: benchmark timer stops when all data reaches OSSes
  - Use `-e` option to force `fsync(2)` and write back all "dirty" (modified) pages
  - Measures time to write data to durable media—not just page cache
- Without fsync, `close(2)` operation may include hidden sync time

```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F -C -e
...
Operation   Max(MiB)
write       70121.02
read        30847.85
```

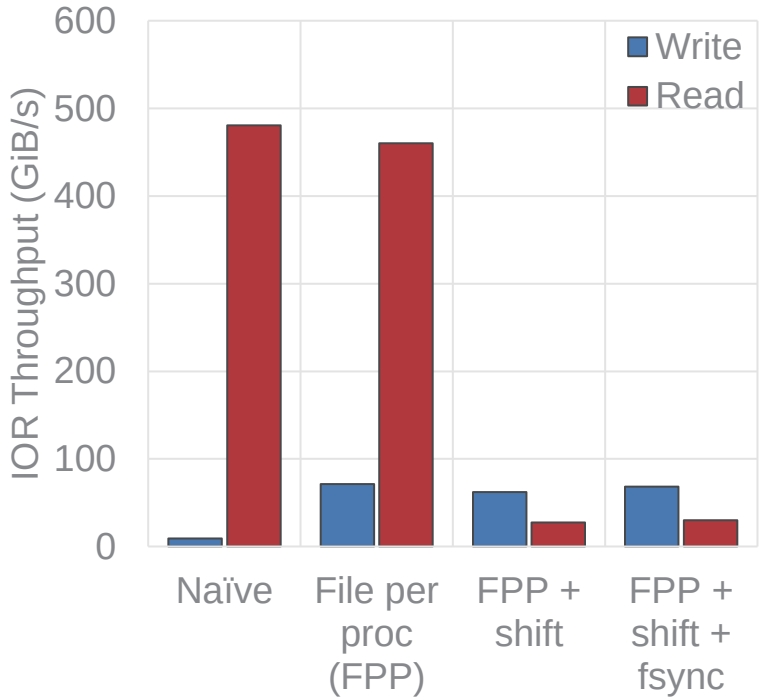


By default, benchmark may appear to hang at the end when files are being closed



With -e / fsync, time to write dirty pages to file system is included

MEASURING BANDWIDTH CAN BE COMPLICATED



- 100x difference from same file system!
  - Client caches and sync
  - File per proc vs. shared file
  - Usual Lustre stuff (e.g., striping)
- For system benchmarking, start with `-F -C -e`
- Page cache is not part of POSIX!
  - Every file system does it differently
  - Understanding its effects requires expert knowledge

IOR ACCEPTANCE TESTS
LUSTRE BANDWIDTH

Only 4 ppn needed

```
$ srun -N 960 -n 3840 ./ior -F -C -e -g -b 4m -t 4m -s 1638 -w -k
$ srun -N 960 -n 3840 ./ior -F -C -e -g -b 4m -t 4m -s 1638 -r
```

Standard args:

- `-F` File-per-process
- `-C` Shift ranks
- `-e` include fsync(2) time

- `-w` Perform write benchmark only
- `-k` Don't delete written files
- `-r` Perform read benchmark only

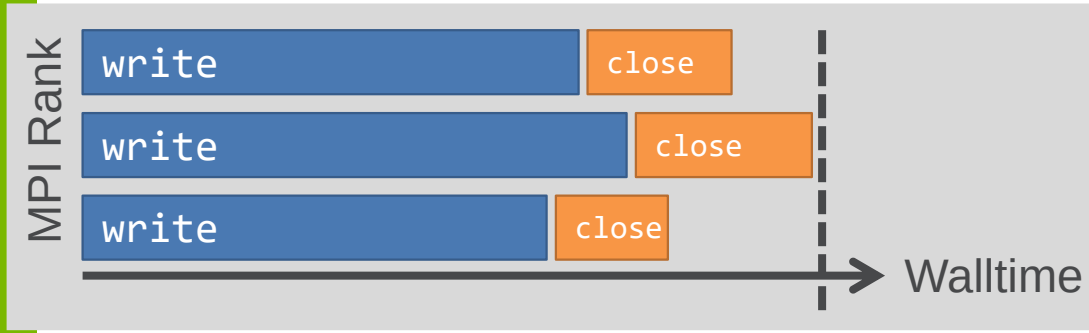
Separate sruns drop client caches

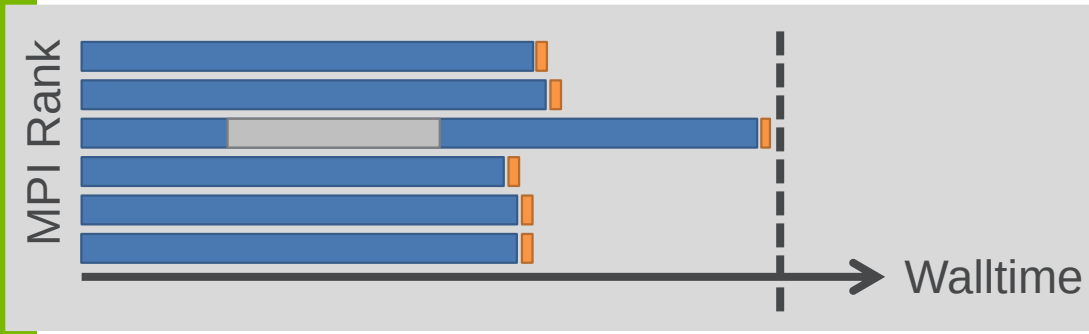
Every rank writes 4 MiB × 1,638
4 MiB at a time
Total ~25 TB

Results:

- 751,709 MB/s write (max)
- 678,256 MB/s read (max)

RUNNING AFOUL OF “WIDE” BENCHMARKING





How much `-b/-s`?

- More is <u>better</u>: overrun cache effects
- More is <u>worse</u>: increase likelihood of hiccup
- Preference: run for 30-60s

What is realistic for you?

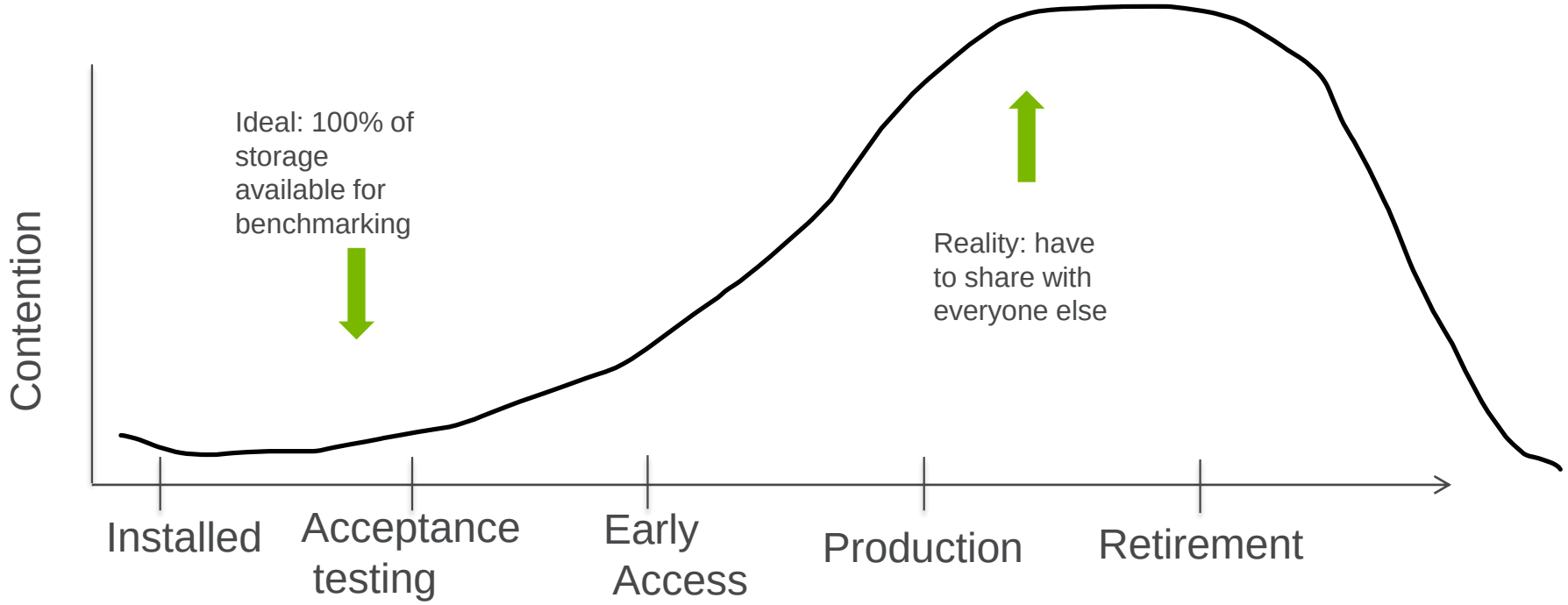
- do you want the big number?
- do you want to emulate user experience?
- small = fast
- big = realistic

EXAMPLE SYSTEM: ALCF POLARIS

- 560 nodes;
  - 2.8 GHz AMD EPYC Milan 7543P 32 core CPU
  - 512 GB of DDR4 RAM
  - 2 local 1.6TB of SSDs in RAID0
- Storage: ALCF production file systems
  - /home: Lustre (8 OST, 4 MDT)
  - /grand: Lustre: (160 OST, 40 MDT, 650 GB/sec)
  - /eagle: Lustre: (160 OST, 40 MDT, 650 GB/sec)
  - /theta-fs0: legacy Lustre
  - /theta-fs1: legacy GPFS
- Network: Slingshot 10 (soon to be slingshot 11)

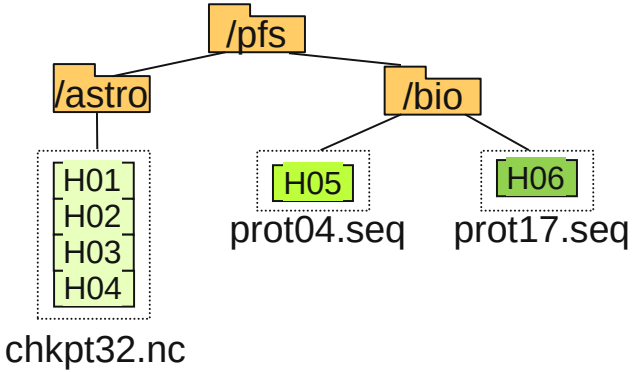
More Polaris information: https://docs.alcf.anl.gov/polaris

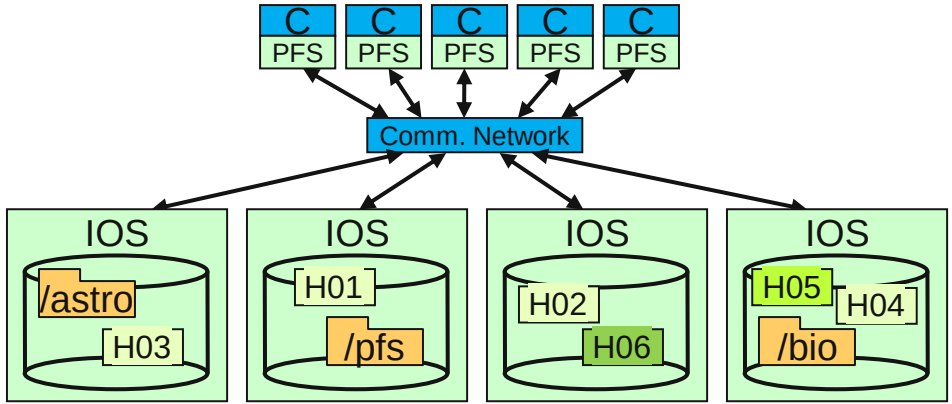
CONTENTION IN BENCHMARKIG



TUNING: STRIPE COUNT

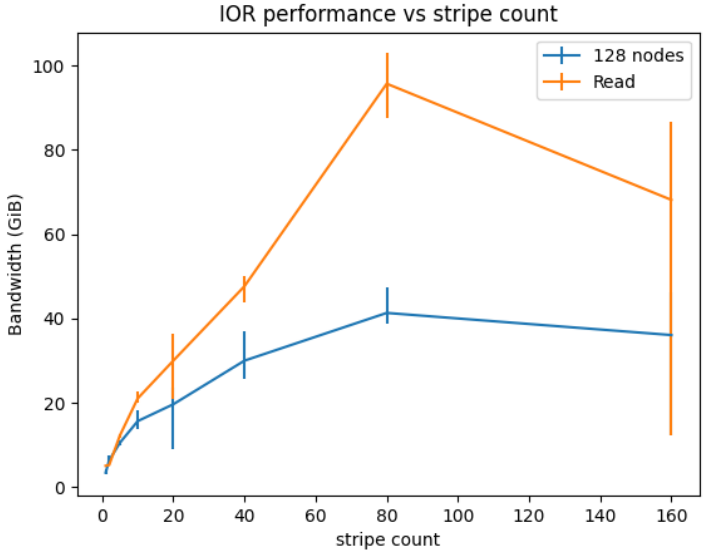
- Parallel file system
  - Parallel in terms of clients
  - Parallel in terms of servers
- File system will hide details, but defaults and details matter for performance





IOR AND STRIPE COUNT

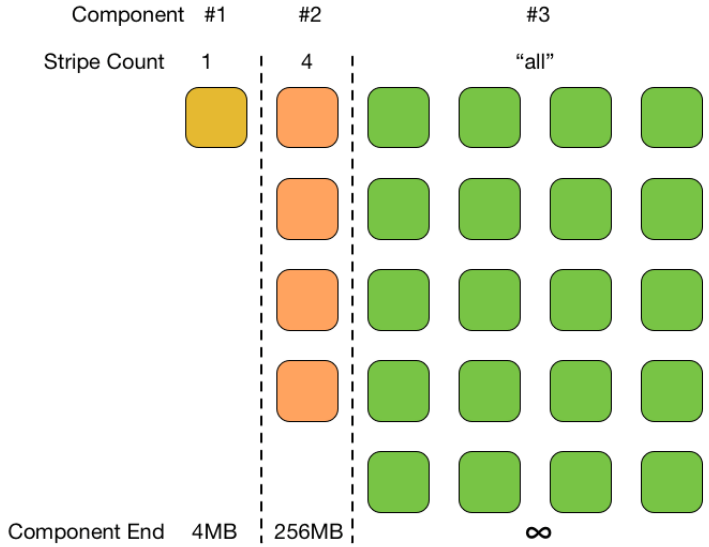
- Default stripe size is 1
  - Why? Most files small: optimizing for common case
- “All the servers” doesn’t seem to hurt performance here
  - lfs setstripe -1 /path/to/file
- Could go further with “overstriping”
  - Didn’t work on Polaris: investigating
- “Where’s my bandwidth?”
  - 128 nodes (network links) here
  - Shared file (so I can experiment with stripe count) means lustre locking overhead/coordination



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

PROGRESSIVE FILE LAYOUT

- Introduced in Lustre-2.10
- Not on by default at ALCF
- Small files land on one server
- Larger files get more parallelism
- Most files small
- Most data lives in big files



USING PFL

- Example:
  - `lfs setstripe -E4M -c1 -E256M -c4 -E-1 -c-1 /grand/radix-io/scratch/robl/pfl/testfile`
- Confirming it worked:
  - `lfs getstripe /grand/radix-io/scratch/robl/pfl/testfile`
- Lots of detail here: focus on "lcme_extent.e_start" and "lcme_extent.e_end"

```
/grand/radix-io/scratch/robl/pfl/testfile
  lcm_layout_gen:    3
  lcm_mirror_count:  1
  lcm_entry_count:   3
    lcme_id:             1
    lcme_mirror_id:      0
    lcme_flags:          init
    lcme_extent.e_start: 0
    lcme_extent.e_end:   4194304
      lmm_stripe_count:  1
      lmm_stripe_size:   1048576
      lmm_pattern:       raid0
      lmm_layout_gen:    0
      lmm_stripe_offset: 119
      lmm_objects:
      - 0: { l_ost_idx: 119, l_fid: [0x100770000:0x12343c1:0x0] }

    lcme_id:             2
    lcme_mirror_id:      0
    lcme_flags:          0
    lcme_extent.e_start: 4194304
    lcme_extent.e_end:   268435456
      lmm_stripe_count:  4
      lmm_stripe_size:   1048576
      lmm_pattern:       raid0
      lmm_layout_gen:    0
      lmm_stripe_offset: -1

    lcme_id:             3
    lcme_mirror_id:      0
    lcme_flags:          0
    lcme_extent.e_start: 268435456
    lcme_extent.e_end:   EOF
      lmm_stripe_count:  -1
      lmm_stripe_size:   1048576
      lmm_pattern:       raid0
      lmm_layout_gen:    0
      lmm_stripe_offset: -1
```

I/O PERFORMANCE FROM LUSTRE

- 1: Please I am begging you stripe your data across more than one server
- 2: There are some other things you could do, too, but please see step 1
  - Lockahead
  - Overstriping
  - File per process (ugh, fine, but only “break glass in case of emergency”)
    - How will you manage thousands of files?

CHARACTERIZING I/O: DARSHAN

CHALLENGES INSTRUMENTING I/O

- How many CPU instructions is “write to this remote file system”?
- Sampling profilers
- Overhead (time, space) of logging every operation?
- Shared resource

DARSHAN : UNDERSTANDING I/O BEHAVIOR AND PERFORMANCE

Thanks to the following for much of this material:

Philip Carns, Shane Snyder, Kevin Harms, Charles Bacon, Sam Lang, Bill Allcock
Math and Computer Science Division and
Argonne Leadership Computing Facility
Argonne National Laboratory

Katie Antypas, Jialin Liu, Quincey Koziol
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

For more information, see:

- P. Carns, et al., “Understanding and improving computational science storage access through continuous characterization,” *ACM TOS* 2011.
- P. Carns, et al., “Production I/O characterization on the Cray XE6,” CUG 2013. May 2013.

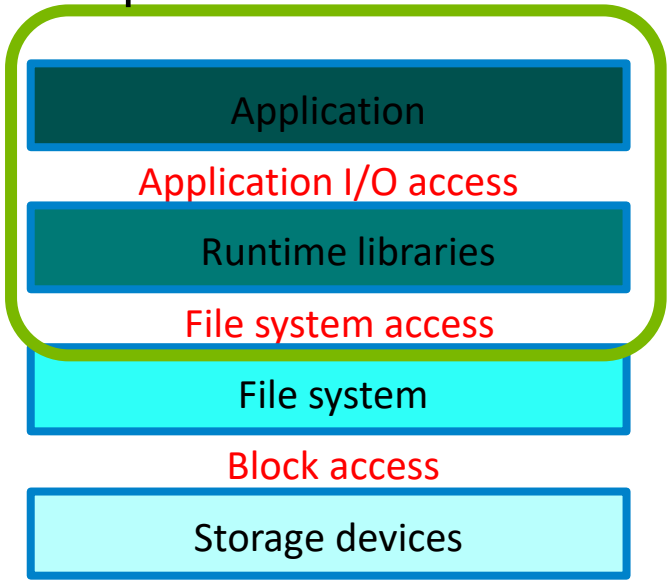
CHARACTERIZING APPLICATION I/O

How is an application using the I/O system?
How successful is it at attaining high performance?

Strategy: observe I/O behavior at the application and library level

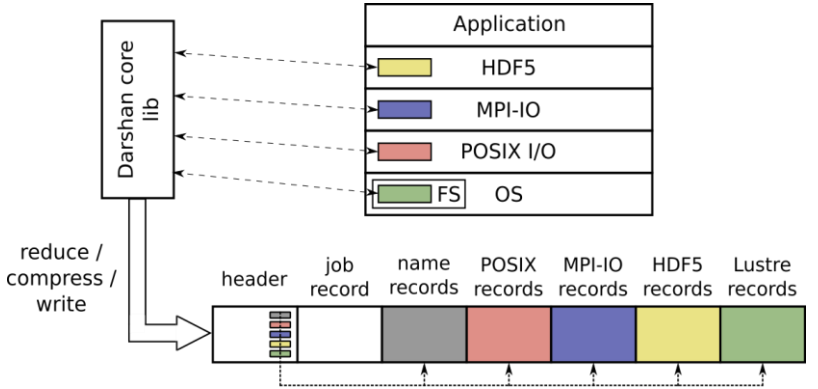
- What did the application intend to do?
- How much time did it take to do it?
- What can be done to tune and improve?

Simplified HPC I/O stack



HOW DOES DARSHAN WORK?

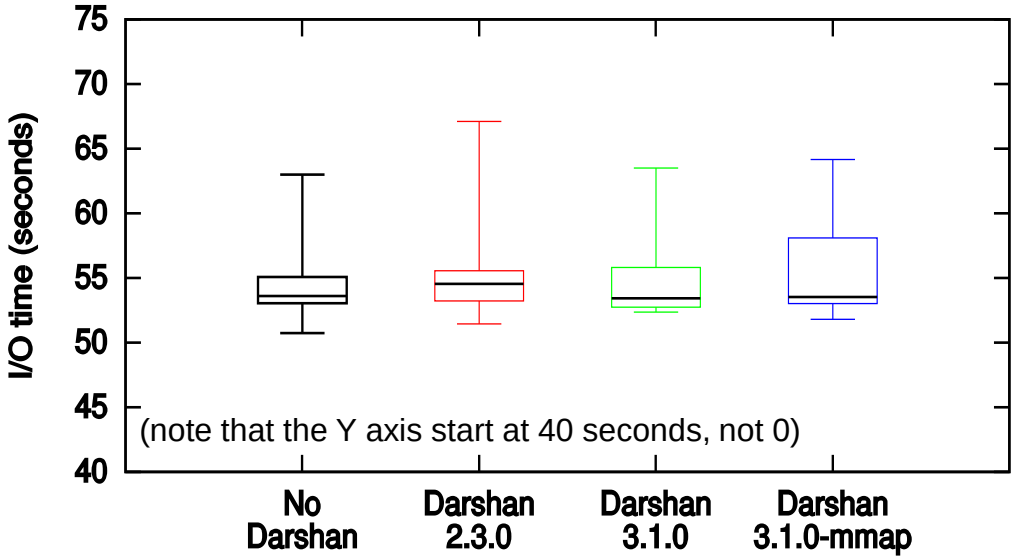
- Darshan records file access statistics independently on each process
- At app shutdown, collect, aggregate, compress, and write log data
- After job completes, analyze Darshan log data
  - `darshan-parser` - provides complete text-format dump of all counters in a log file
  - *PyDarshan* - Python analysis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

WHAT IS THE OVERHEAD OF DARSHAN I/O FUNCTION WRAPPING?

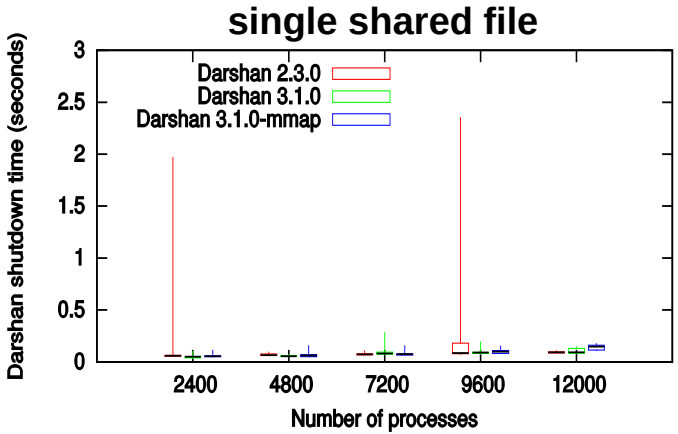
- Compare I/O time of IOR linked against different Darshan versions on *NERSC Edison*
  - File-per-process workload
  - 6,000 MPI processes
  - >12 million instrumented calls
- Note use of box plots
  - Ran each test 15 times
  - I/O variation is a reality
  - Consider I/O performance as a distribution, not a singular value



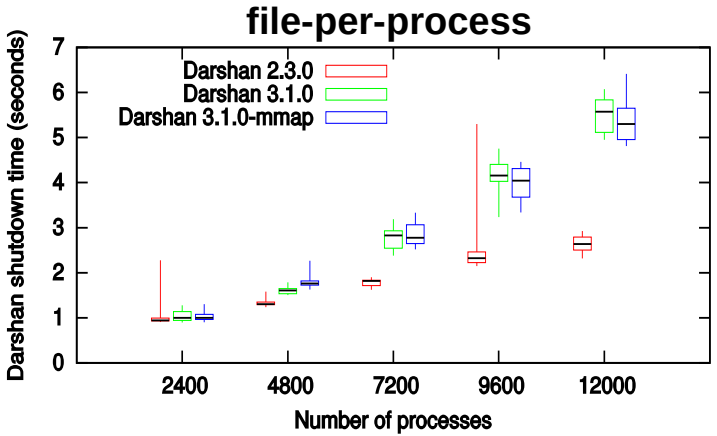
Snyder et al., "Modular HPC I/O Characterization with Darshan," in *Proceedings of 5th Workshop on Extreme-scale Programming Tools (ESPT 2016),* 2016.

DOES SHUTDOWN OVERHEAD IMPACT WALLTIME?

- Darshan aggregates, compresses, and collectively writes I/O data records at MPI_Finalize()
- To test, synthetic workloads are injected into Darshan and resulting shutdown time is measured on *NERSC Edison*



Near constant shutdown time of ~100 ms in all cases



Shutdown time scales linearly with job size: 5–6 sec extra shutdown time with 12K files

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
- Some platforms (ALCF Polaris, NERSC Perlmutter): opt-in via module loading
- Minimal dependencies: not hard to build

```
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
  - ➢ **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

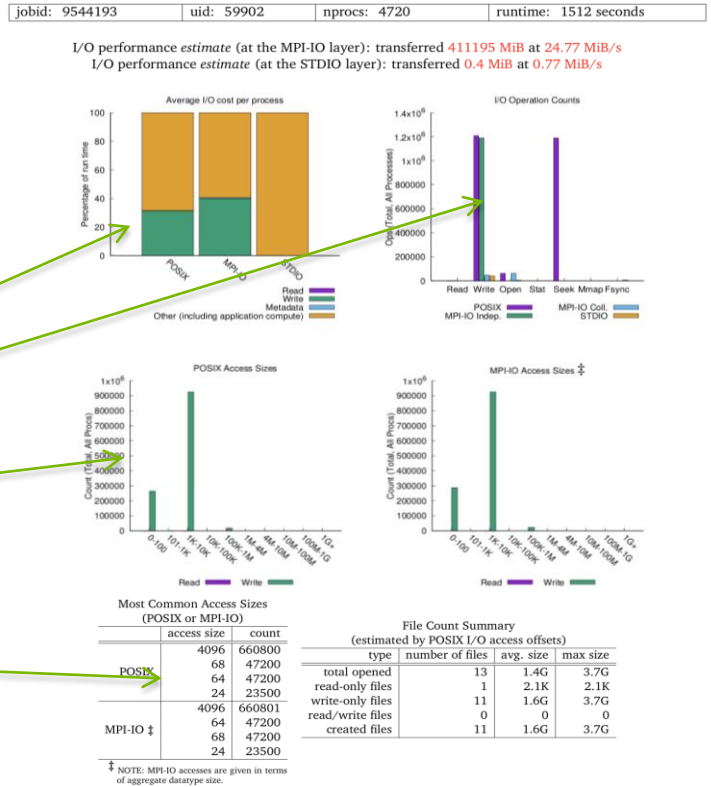
- After locating your log, users can utilize Darshan log analysis tools for gaining insights into application I/O behavior:

```
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 (LEGACY)

- 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



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

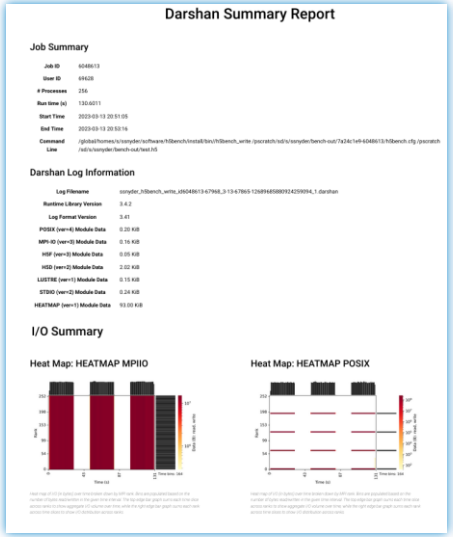
‡ NOTE: MPI-IO accesses are given in terms of aggregate datatype size.

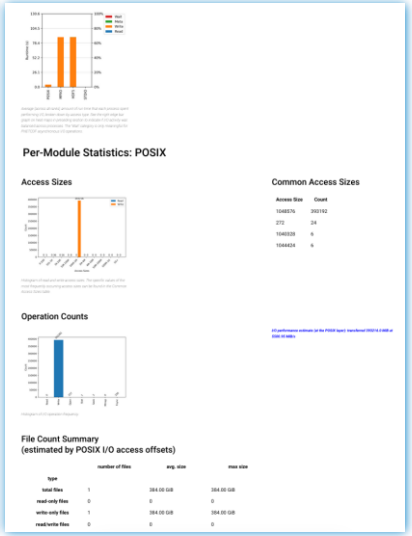
File Count Summary (estimated by POSIX I/O access offsets)

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

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

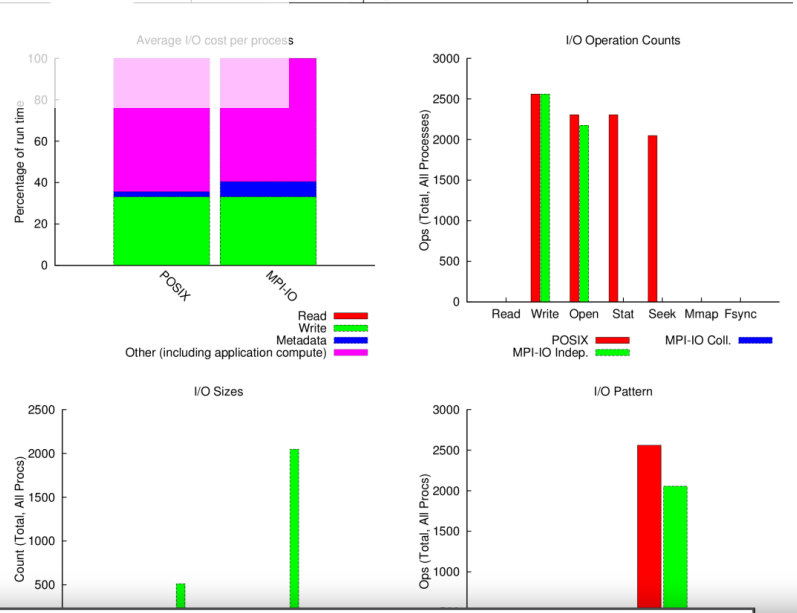
ANALYZING I/O PERFORMANCE PROBLEMS WITH DARSHAN

Lightweight nature of Darshan means it can be always on and help both **users** and HPC **operators**

- Users
  - Many I/O problems can be observed from these logs
  - Study applications on demand to debug specific jobs
- Operators
  - Mine logs to catch problems proactively
  - Analyze user behavior, misbehavior, and knowledge gaps

EXAMPLE: CHECKING USER EXPECTATIONS

- 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

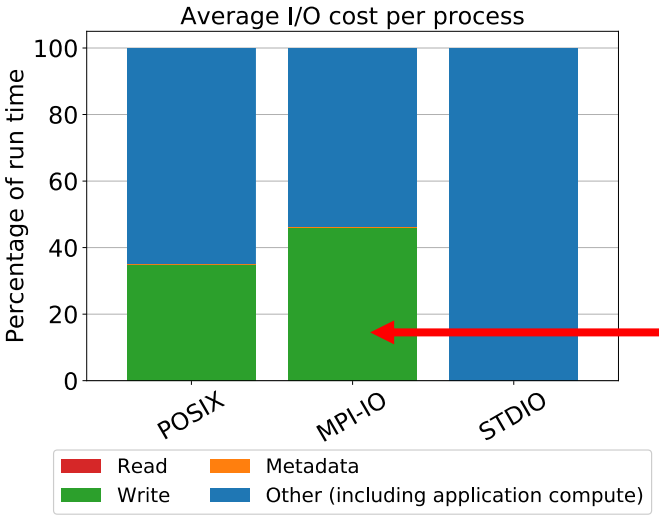
EXAMPLE: WHEN DOING THE RIGHT THING GOES WRONG

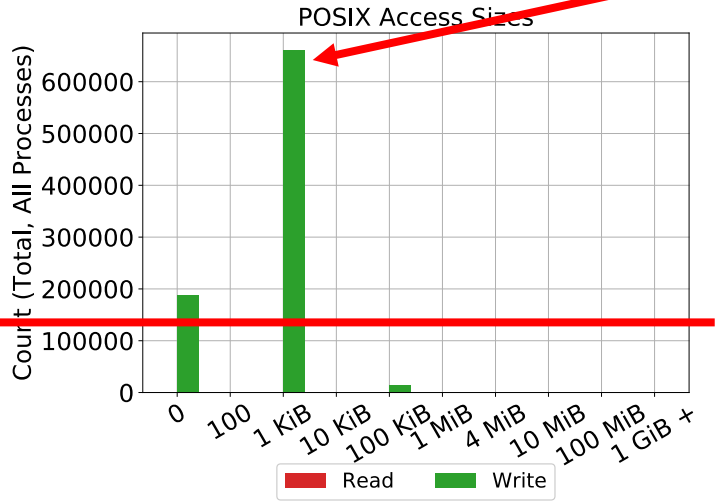
- Large-scale astrophysics application using MPI-IO
- Used 94,000,000 CPU hours at NERSC since 2015

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)

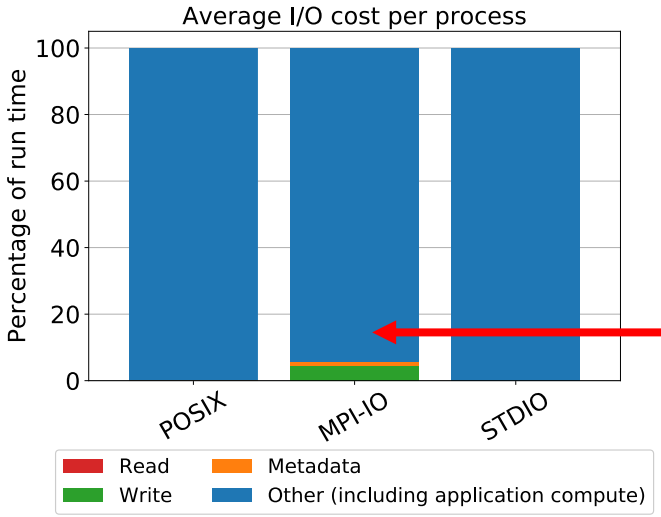
DOING THE RIGHT THING GOES WRONG

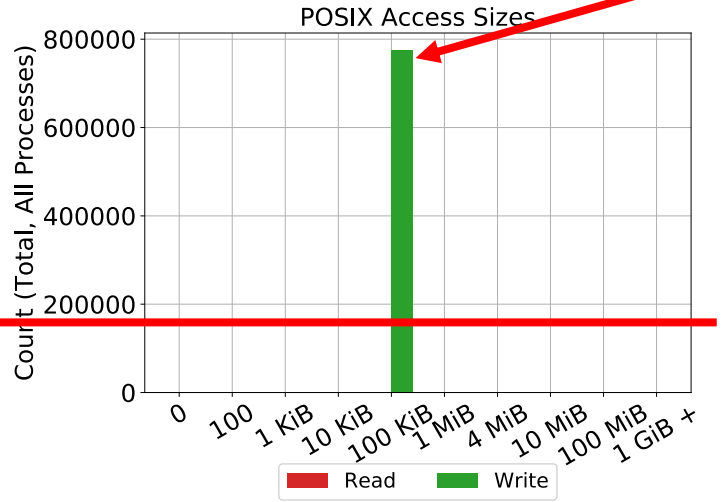
- Collective I/O was being disabled by middleware due to type mismatch in app code
- After type mismatch bug fixed, collective I/O gave 40× speedup

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

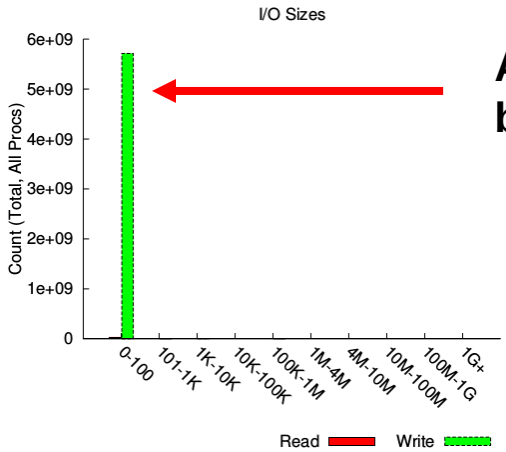
Data Transfer Per Filesystem

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



Almost 6 billion I/O!

Most Common Access Sizes

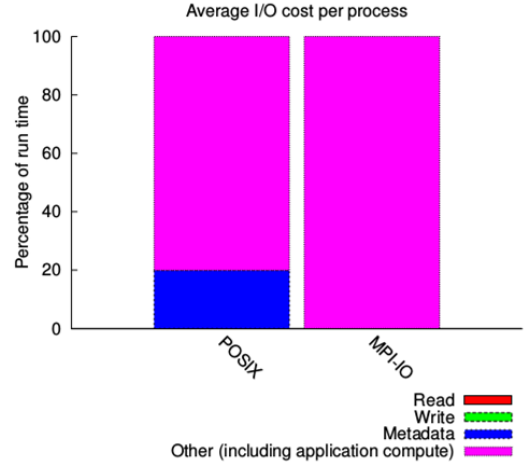
access size	count
1	3418409696
15	2275400442
24	42289948
12	14725053

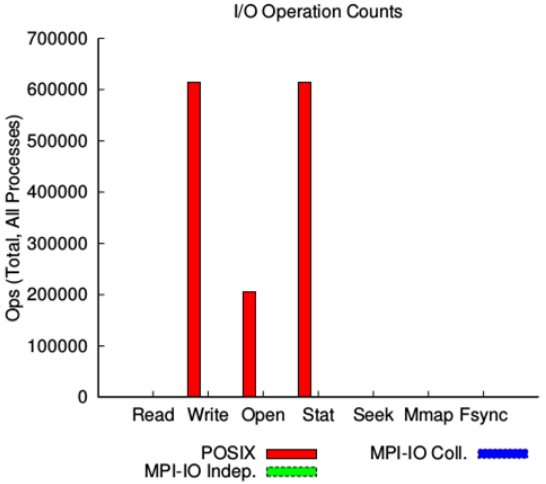
1-byte accesses!

EXAMPLE: EXCESSIVE METADATA

- Scenario: Very high percentage of I/O time spent performing metadata operations (open, close, stat, seek)
  - Close() cost can be misleading due to write-behind cache flushing!
  - Candidates for coalescing files and eliminating 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





PROTIP: STAT IS NOT CHEAP ON PARALLEL FILE SYSTEMS

- Stat() requires a consistent size calculation for the file
  - Store a pre-calculated size on the metadata server(s)
    - IBM Spectrum Scale (GPFS)
    - Lustre with Lazy Size-on-MDT feature
  - Calculate size on demand, triggering broadcast/incasts between metadata server and data servers
    - Lustre default behavior

- No present-day PFS implementations respond well when thousands of processes stat() the same file at once

SYSTEM-LEVEL PERFORMANCE ANALYSIS

- "Always-on" nature of Darshan enables system-wide I/O analysis
- Daily Top 10 I/O Users list at NERSC to identify users…
  - Running jobs in their home directory
  - Who might benefit from the burst buffer
- Can develop heuristics to detect anomalous I/O behavior
  - Highlight jobs spending a lot of time in metadata
  - Automated triggering/alerting

#	Users	Read(GiB)	Write(GiB)	# Jobs
1.	john	9727.3	10192.7	16432
2.	mary	3672.1	3662.1	701
3.	jane	6777.8	155.6	2

#	File Systems	Read(GiB)	Write(GiB)	# Jobs
1.	cscratch	18978.4	16940.1	4026
2.	homes	10122.0	10692.7	16565
3.	bb-shared	233.9	8.0	1

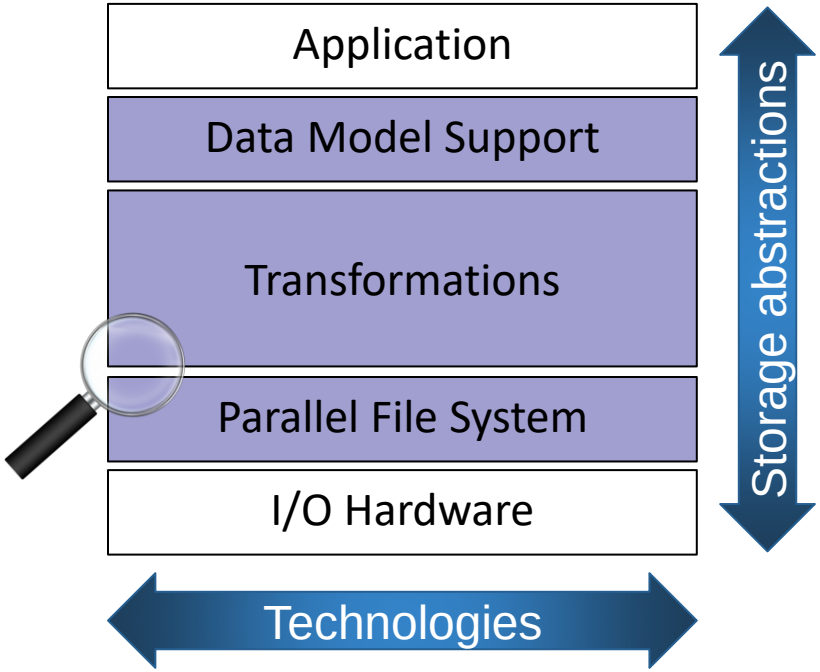
#	Applications	Read(GiB)	Write(GiB)	# Jobs
1.	vasp_std	10078.2	10528.6	16844
2.	pw.x	3672.1	3662.1	701
3.	lmp_cori	6699.4	0.0	1

#	User/App/FS	Read(GiB)	Write(GiB)	# Jobs
1.	john/vasp_std/homes	9727.3	10192.7	16432
2.	mary/pw.x/cscratch	3672.1	3662.1	701
3.	jane/lmp_cori/cscratch	6699.4	0.0	1

Carns et al., "Production I/O Characterization on the Cray XE6," in *Proceedings of the Cray User Group* (CUG'13), 2013.

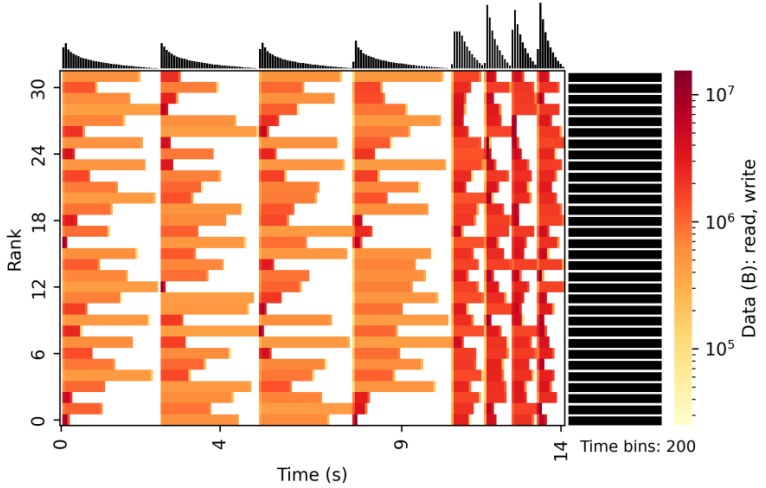
LOW-LEVEL I/O INSTRUMENTATION

- Darshan provides in-depth instrumentation of the lower layers of 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

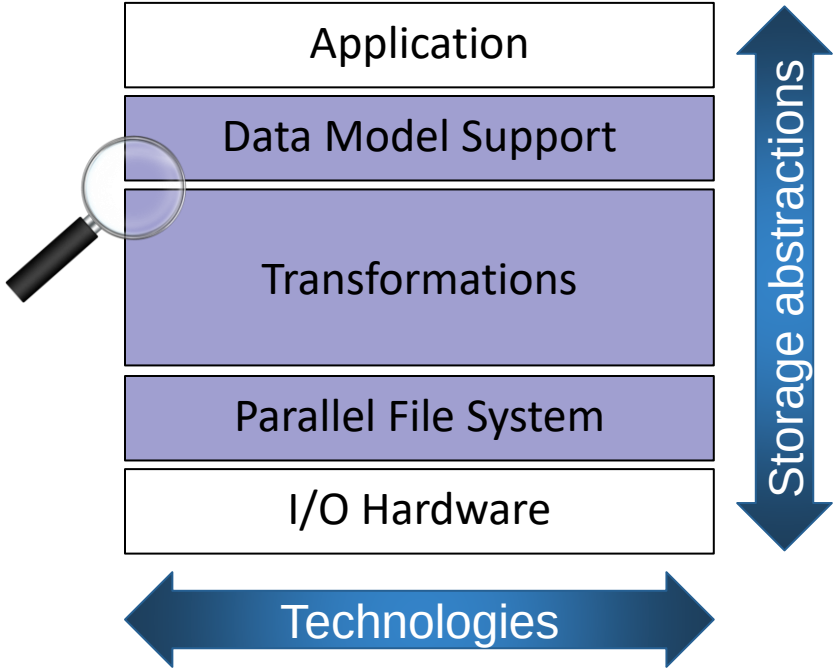
- Beyond its traditional capture mode, Darshan offers key features for obtaining finer-grained details of low-level I/O activity:
  - ➢ **Heatmap module**: captures histograms of I/O activity at each process using a fixed size histogram
    - Available for POSIX, MPI-IO, and STDIO interfaces by default 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

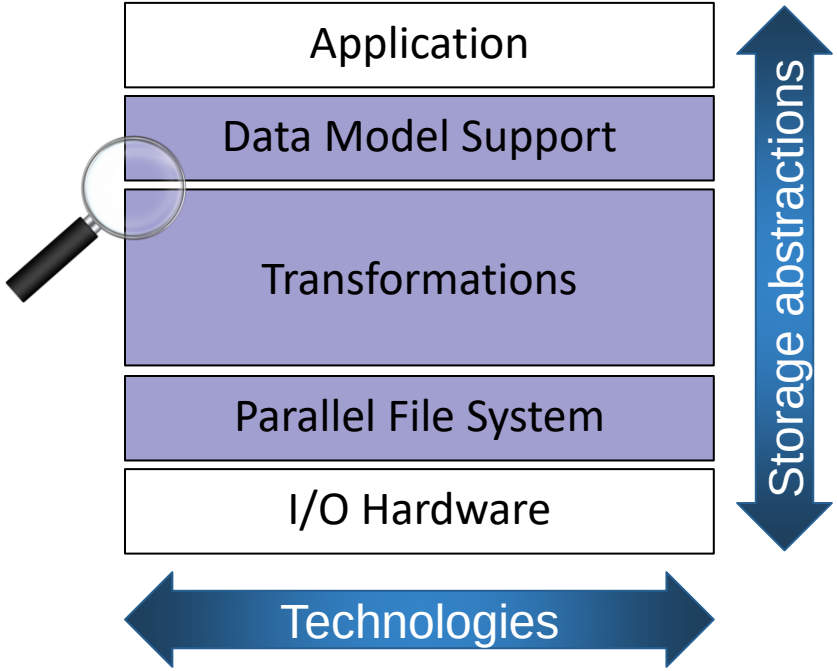
- Darshan similarly provides in-depth instrumentation of popular high-level I/O libraries for HPC
  - **HDF5**: detailed instrumentation of accesses to HDF5 files and datasets available starting in 3.2+ versions
  - **PnetCDF**: detailed instrumentation of accesses to PnetCDF files and variables available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



HDF5 APPLICATION INSTRUMENTATION EXAMPLE

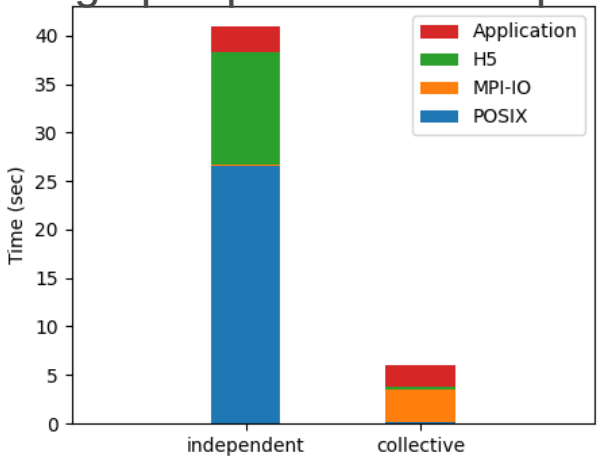
- The MACSio[1] benchmark evaluates behavior of multi-physics I/O workloads using different I/O backends, including HDF5
  - ➢ We instrumented using Darshan's HDF5 module to see what insights we could gain into performance characteristics of independent and collective I/O configurations

b/w: **~30 MB/sec**

POSIX I/O dominates, **H5** incurs non-negligible overhead forming this workload

Negligible time spent in **MPI-IO**

Average per-process time spent in I/O



b/w: **~290 MB/sec**

H5 and **POSIX** incur minimal overhead for this workload

MPI-IO collective I/O algorithm dominates

1. https://github.com/LLNL/MACSio

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 Python 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 end 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 an output HTML report describing job's I/O behavior

PYDARSHAN JOB SUMMARY TOOL

Detailed job metadata

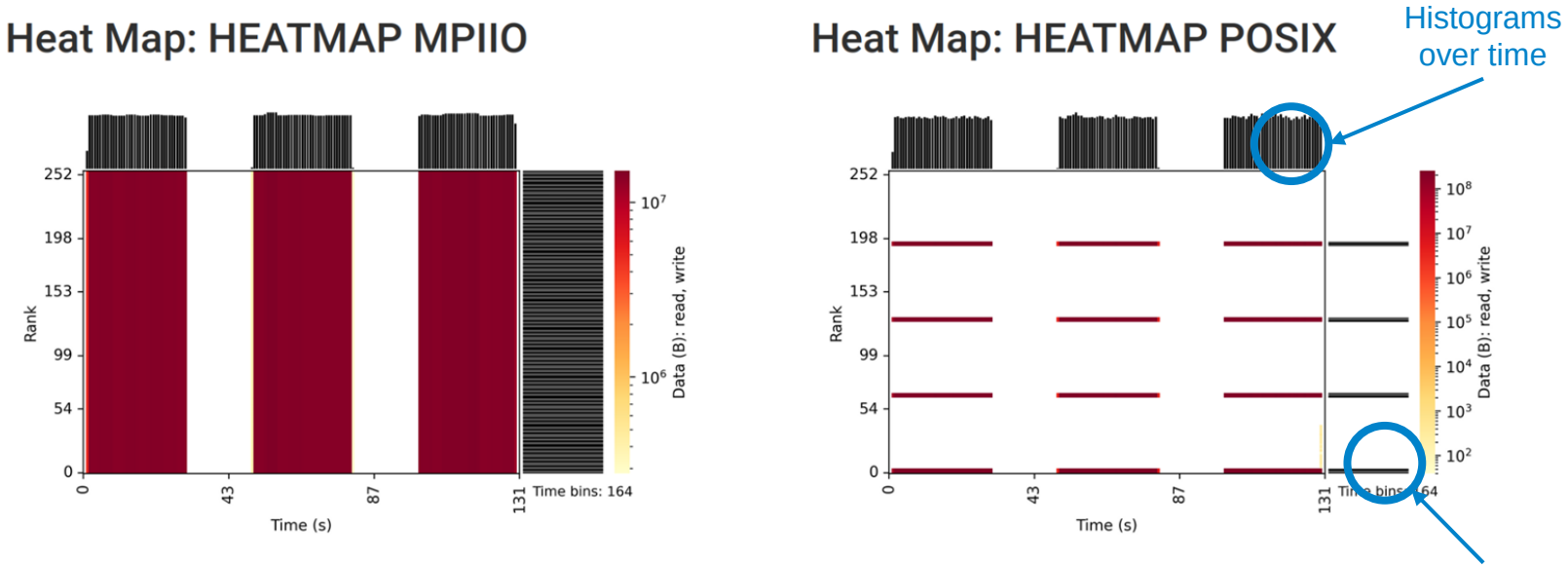
Job Summary

Job ID	6048613
User ID	69628
# Processes	256
Run time (s)	130.6011
Start Time	2023-03-13 20:51:05
End Time	2023-03-13 20:53:16
Command Line	/global/homes/s/ssnyder/software/h5bench/install/bin//h5bench_write /pscratch/sd/s/ssnyder/bench-out/7a24c1e9-6048613 /h5bench.cfg /pscratch/sd/s/ssnyder/bench-out/test.h5

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

PYDARSHAN JOB SUMMARY TOOL

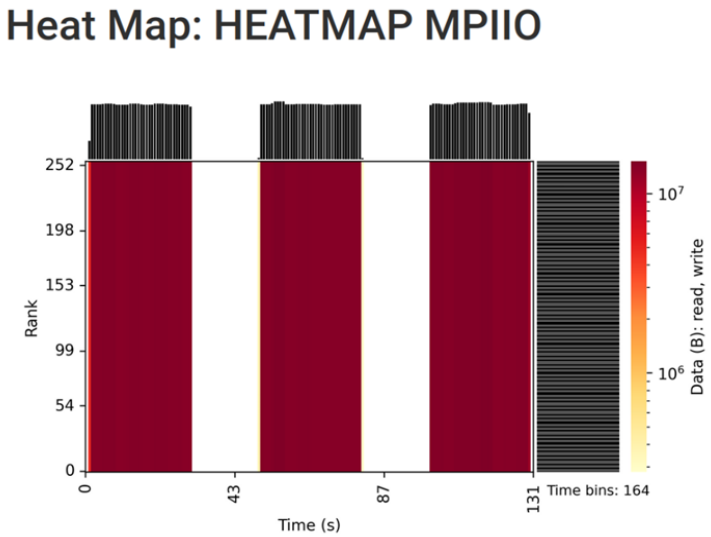
Heatmaps for visualizing I/O activity

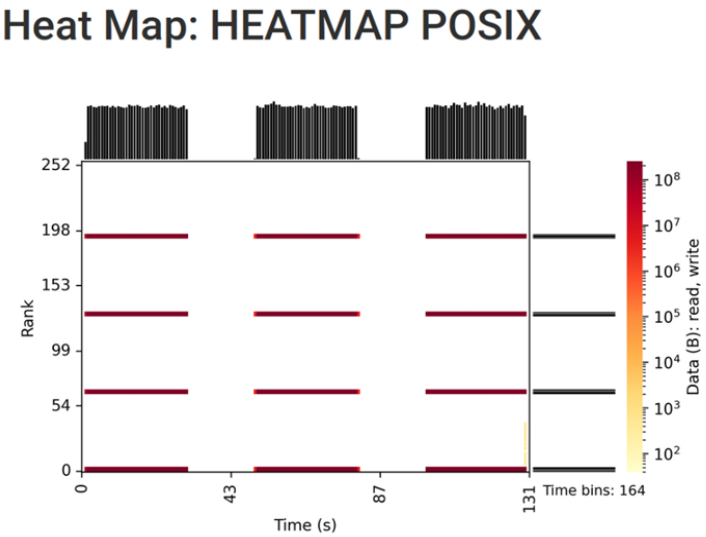


Analyzing I/O behavior over time, ranks, and interfaces can offer key insights into application I/O behavior.

PYDARSHAN JOB SUMMARY TOOL

Heatmaps for visualizing I/O activity

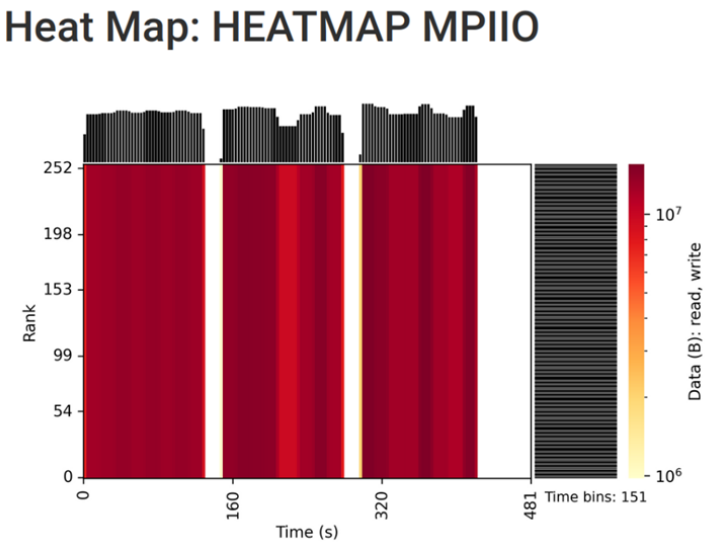


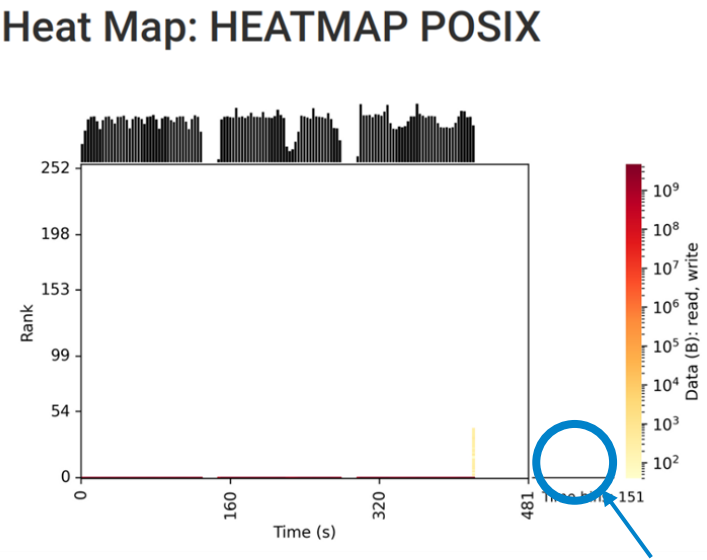


This example heatmap illustrates a typical MPI-IO collective I/O pattern. All MPI ranks perform MPI-IO operations (left), but only a subset of “aggregators” access the file via POSIX operations (right).

PYDARSHAN JOB SUMMARY TOOL

Heatmaps for visualizing I/O activity





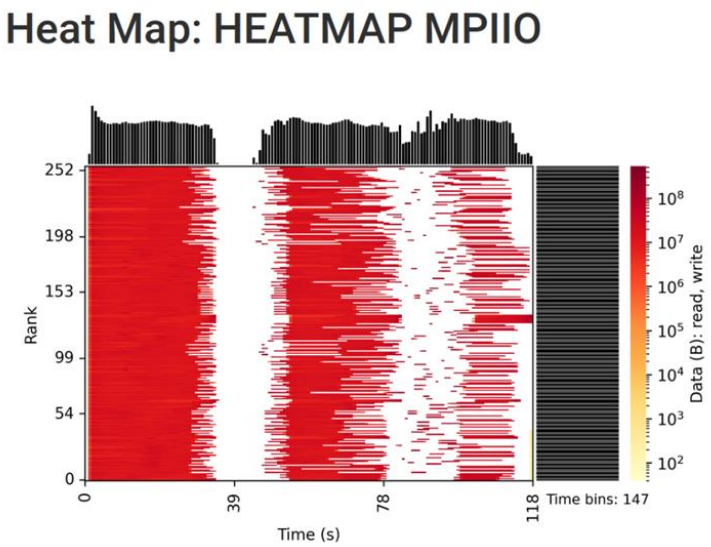
Heatmaps can help quickly detect common pitfalls.

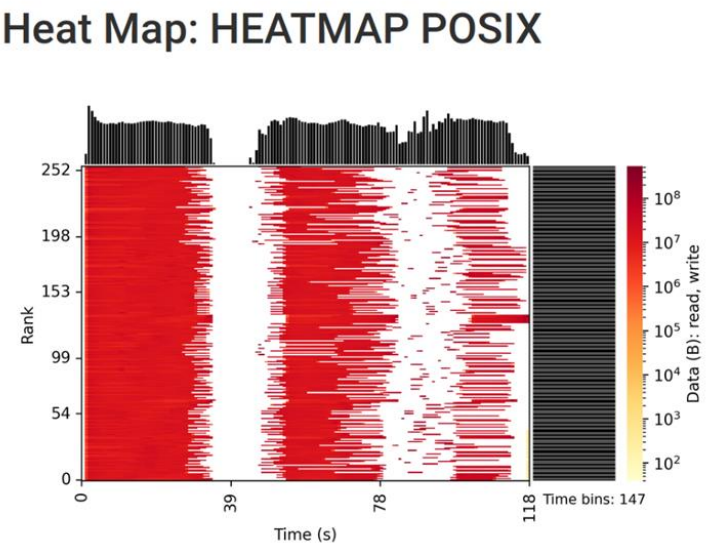
All I/O funneled through rank 0

Oops, I forgot to tell Lustre to use more than one stripe.

PYDARSHAN JOB SUMMARY TOOL

Heatmaps for visualizing I/O activity



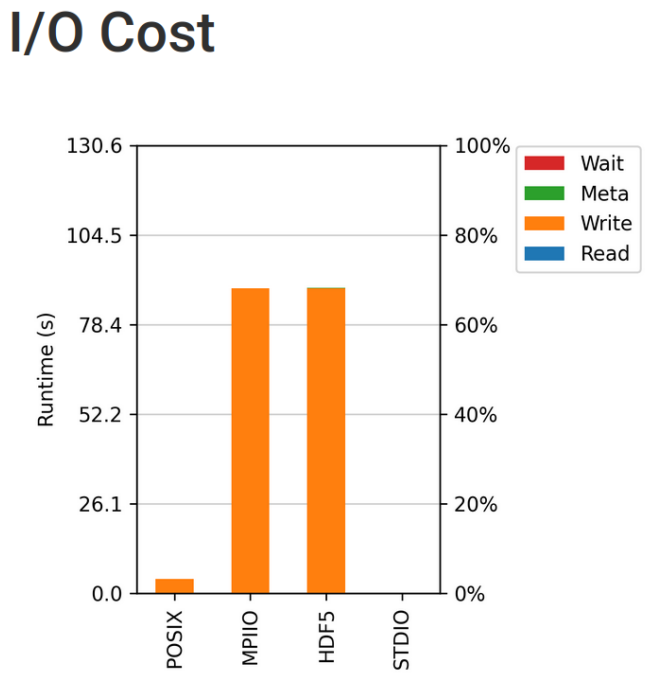


Heatmaps can help quickly detect common pitfalls.

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

PYDARSHAN JOB SUMMARY TOOL

Cross-module I/O comparisons

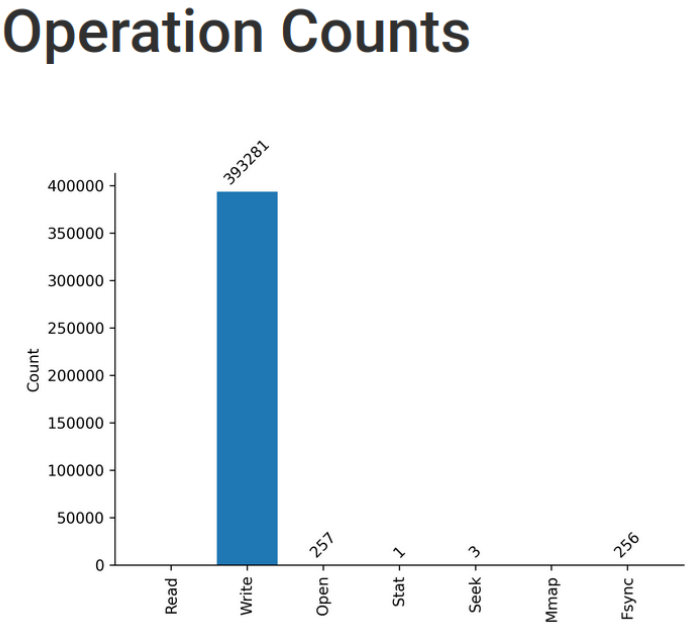


On average, how much time was spent reading, writing, and doing metadata across main I/O interfaces?

If mostly non-I/O (i.e., compute), limited opportunities for I/O tuning.

PYDARSHAN JOB SUMMARY TOOL

Per-module I/O statistics



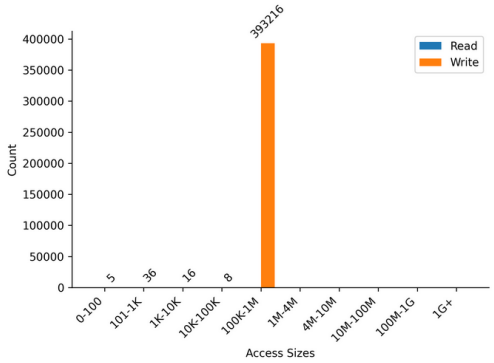
What were the relative totals of different I/O operations across key interfaces?

Lots of metadata operations (open, stat, seek, etc.) could be a sign of poorly performing I/O.

PYDARSHAN JOB SUMMARY TOOL

Per-module I/O statistics

Access Sizes



Common Access Sizes

Access Size	Count
1048576	393192
272	24
1040328	6
1044424	6

Access size distributions and common access sizes are provided to better understand general file access patterns.

In general, larger access sizes perform better with most storage systems.

AVAILABLE DARSHAN ANALYSIS TOOLS

- Documentation: 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
    - For example, darshan-parser user_app_numbers.darshan | grep write
    - Useful for building your own analysis
- Third-party tools (incomplete list!)
  - **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/
  - DXT Explorer: visualize detailed “extended tracing” data https://github.com/hpc-io/dxt-explorer
  - Drishti: a “darshan coach”: https://github.com/hpc-io/drishti-io

MPI-IO

MPI-IO

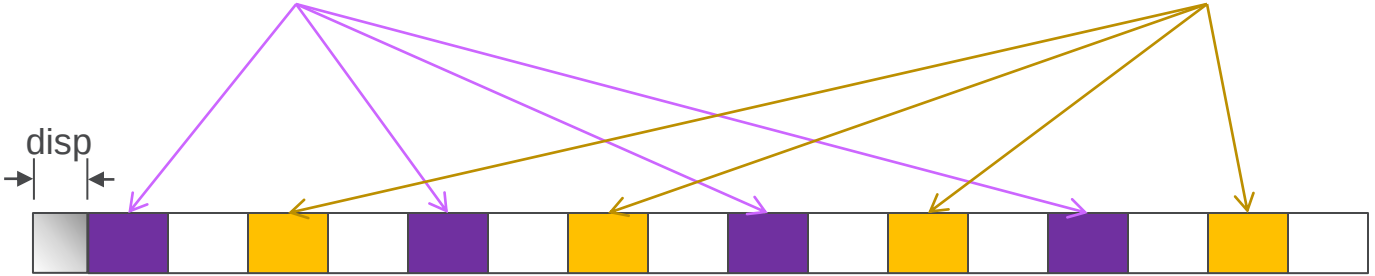
- I/O interface specification for use in MPI apps
- Data model is same as POSIX: stream of bytes in a file
- Features many improvements over POSIX:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - Fortran bindings (and additional languages)
  - System for encoding files in a portable format (external32)
    - Not self-describing – just a well-defined encoding of types
- Implementations available on most platforms (more later)

SIMPLE MPI-IO

- Collective open: all processes in communicator
- File-side data layout with *file views*
- Memory-side data layout with *MPI datatype* passed to write

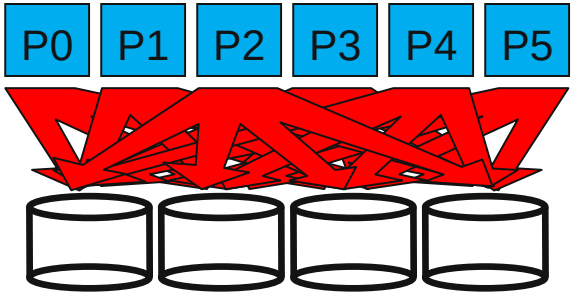
```
MPI_File_open(COMM, name, mode,
          info, fh);
MPI_File_set_view(fh, disp, etype,
          filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
          datatype, status);
```

```
MPI_File_open(COMM, name, mode,
          info, fh);
MPI_File_set_view(fh, disp, etype,
          filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
          datatype, status);
```

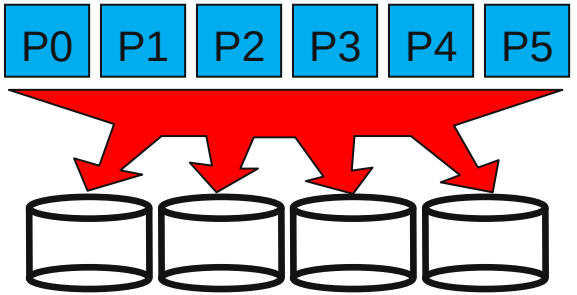


INDEPENDENT AND COLLECTIVE I/O

- Independent I/O operations
  - Specify only what a single process will do
  - Do not pass on relationships between I/O on other processes
- Many applications have alternating phases of computation and I/O
  - During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions are called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole, more opportunities for optimization in lower software layers, better performance



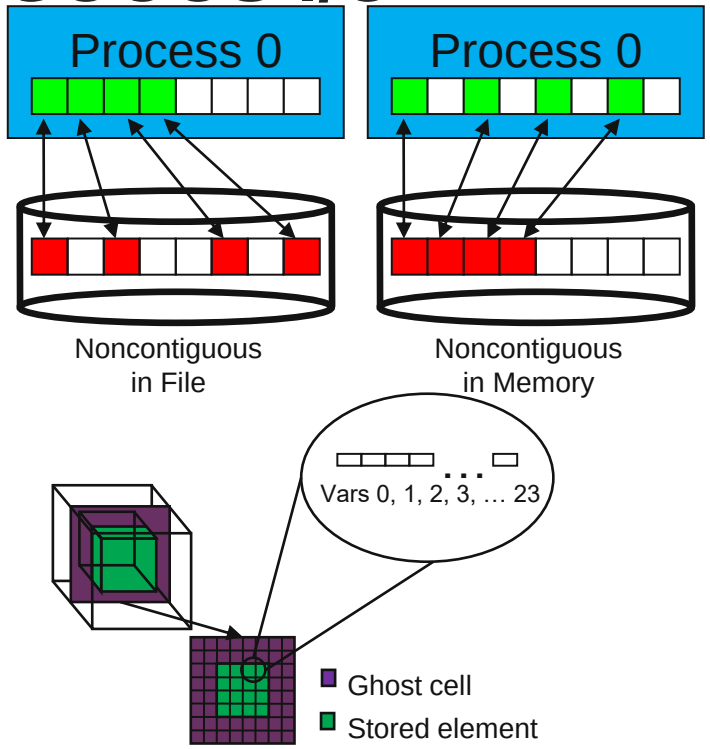
Independent I/O



Collective I/O

CONTIGUOUS AND NONCONTIGUOUS I/O

- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory
  - Noncontiguous in file
  - Noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g., block decomposition)
- Describing noncontiguous accesses with a single operation passes more knowledge to I/O system

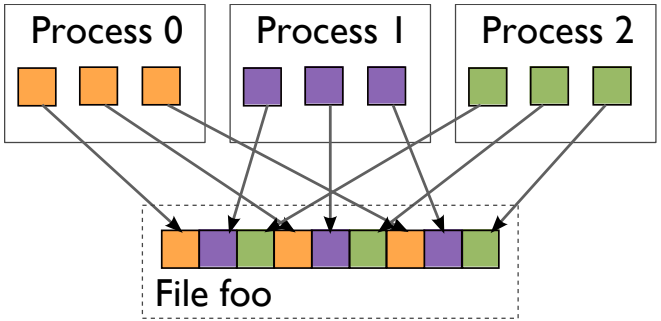


Extracting variables from a block and skipping ghost cells will result in noncontiguous I/O

I/O TRANSFORMATIONS

Software between the application and the PFS performs transformations, primarily to improve performance

- Goals of transformations:
  - Reduce number of I/O operations to PFS (avoid latency, improve bandwidth)
  - Avoid lock contention (eliminate serialization)
  - Hide huge number of clients from PFS servers
- "Transparent" transformations don't change the final file layout
  - File system is still aware of the actual data organization
  - File can be later manipulated using serial POSIX I/O



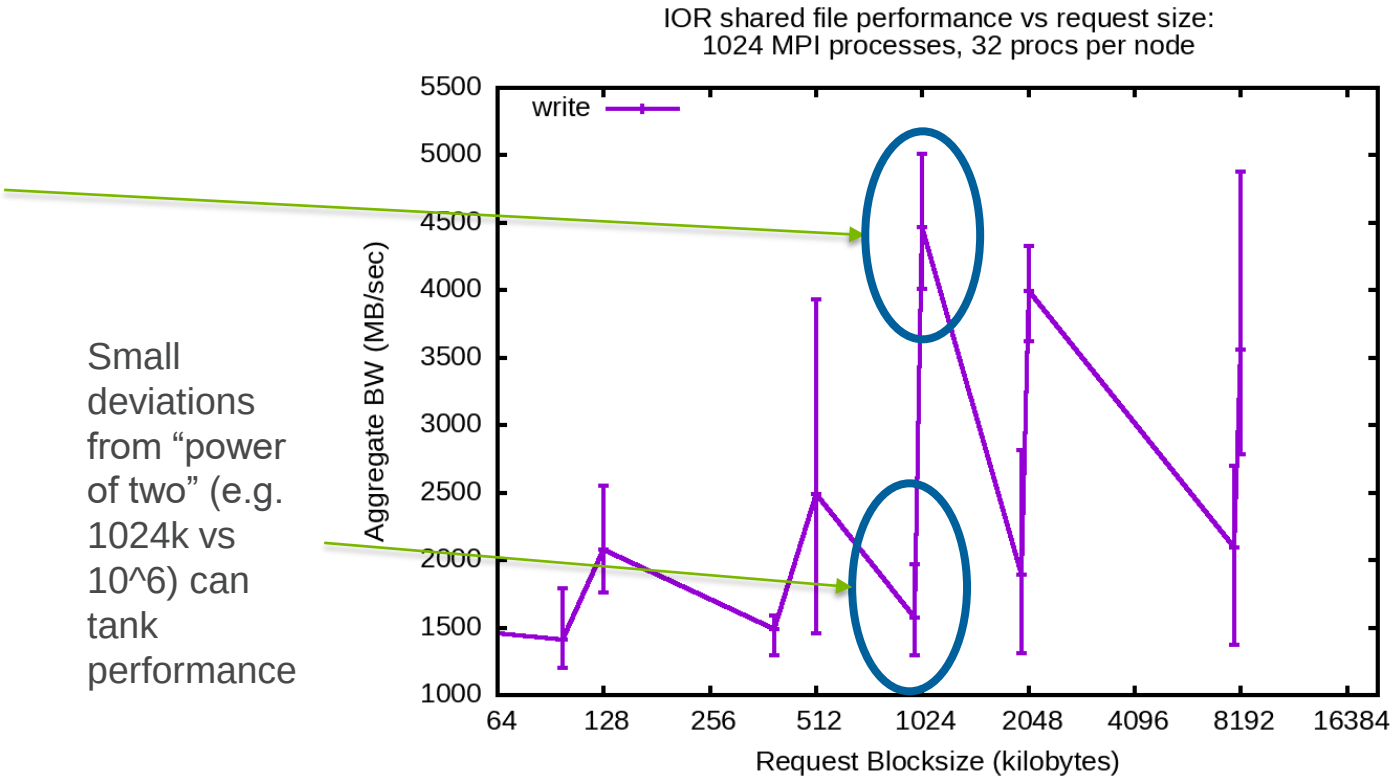
When we think about I/O transformations, we consider the mapping of data between application processes and locations in file

REQUEST SIZE AND I/O RATE

Request matches Lustre “stripe size”: good performance with low variability

Small deviations from “power of two” (e.g. 1024k vs 10^6) can tank performance

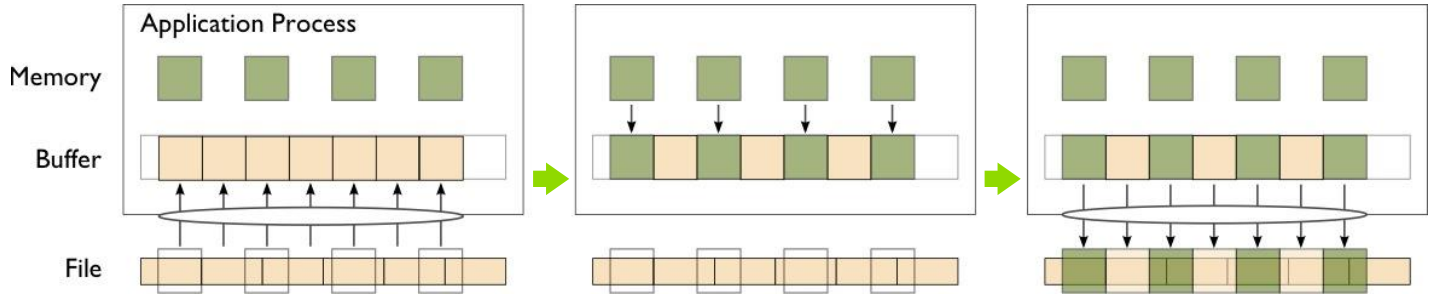
In general, larger requests better.



Tests run on 1K processes of HPE/Cray Theta at Argonne

REDUCING NUMBER, INCREASING SIZE OF OPERATIONS

- **Because most operations go over the network, I/O to a PFS incurs more latency than with a local FS**
- *Data sieving* is a technique to address I/O latency by combining operations:
  - When reading, application process reads a large region holding all needed data and pulls out what is needed
  - When writing, three steps required (below)



Step 1: Data in region to be modified are read into intermediate buffer (1 read).

Step 2: Elements to be written to file are replaced in intermediate buffer.

Step 3: Entire region is written back to storage with a single write operation.

NONCONTIG WITH IOR

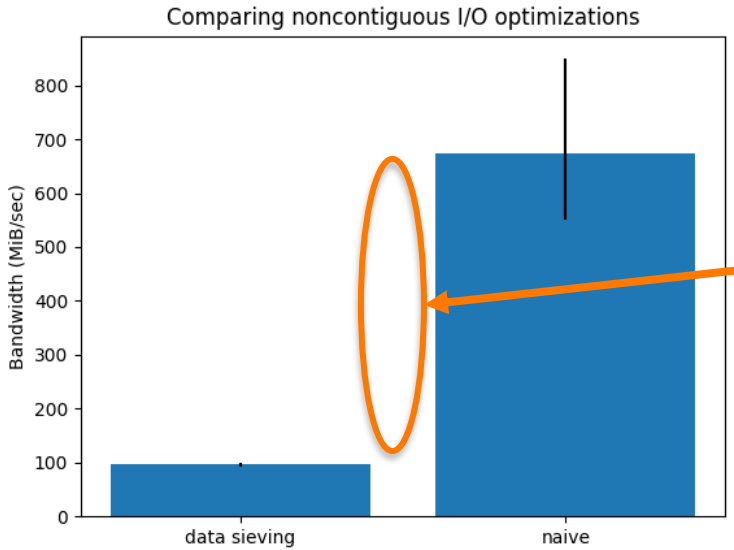
- IOR can describe access with an MPI datatype
  - `--mpiio.useStridedDatatype -b … -s …`

- (buggy in recent versions: use 4.0rc1 or newer)

DATA SIEVING IN PRACTICE

Not always a win, particularly for writing:

- Enabling data sieving instead made writes slower: why?
  - Locking to prevent false sharing (not needed for reads)
  - Multiple processes per node writing simultaneously
  - Internal ROMIO buffer too small, resulting in write amplification [1]



	Naiive	Data Sieving
MPI-IO writes	192	192
MPI-IO Reads	0	0
Posix Writes	192000	192000
Posix Reads	0	192015
MPI-IO bytes written	1 920 000 000	1 920 000 000
MPI-IO bytes read	0	0
Posix bytes read	0	100 039 006 128
Posix bytes written	1 920 000 000	100 564 552 704

[1] Selected Darshan statistics

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

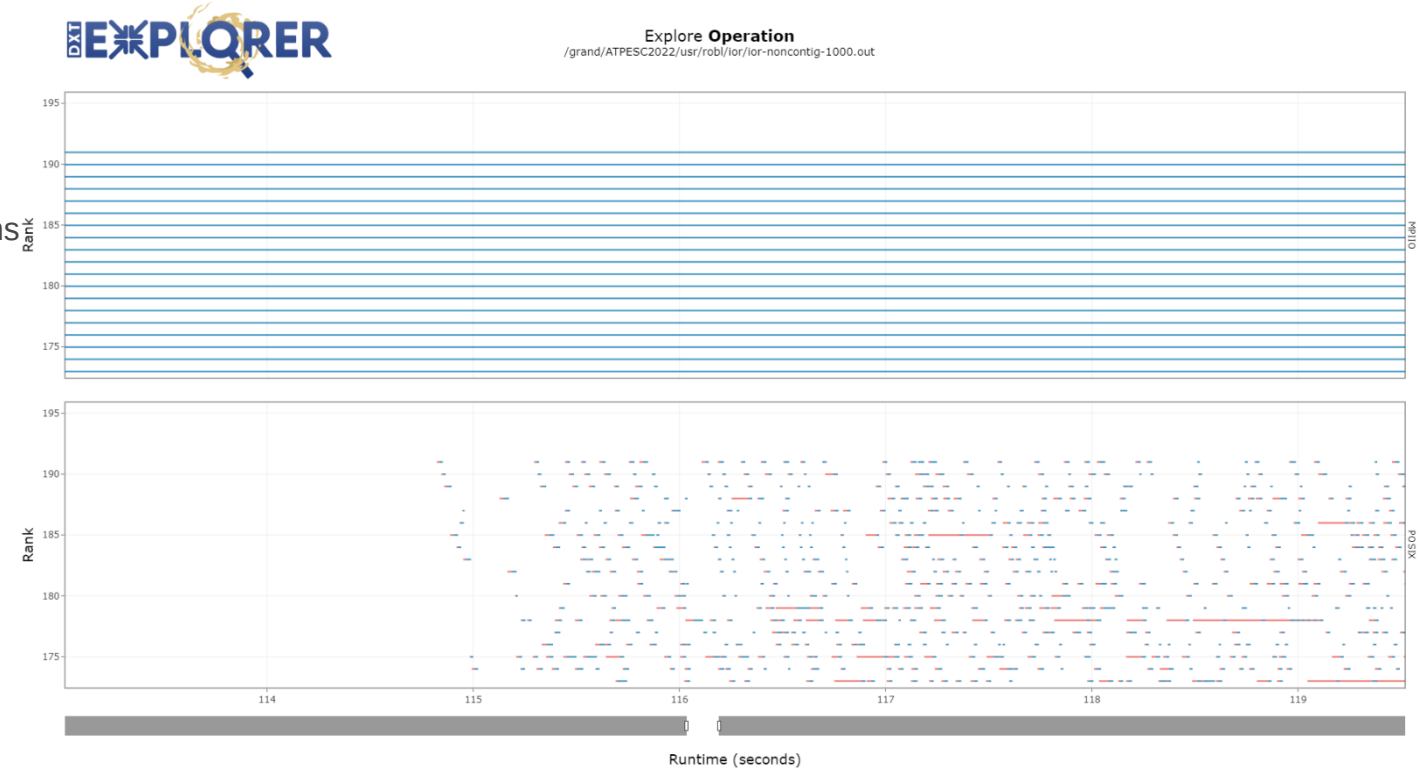
DATA SIEVING: TIME LINE

Top: MPI I/O call describing noncontiguous regions

One MPI I/O call (top) turns into many POSIX operations (below)

Independent: no coordination possible. Each process does its own data sieving.

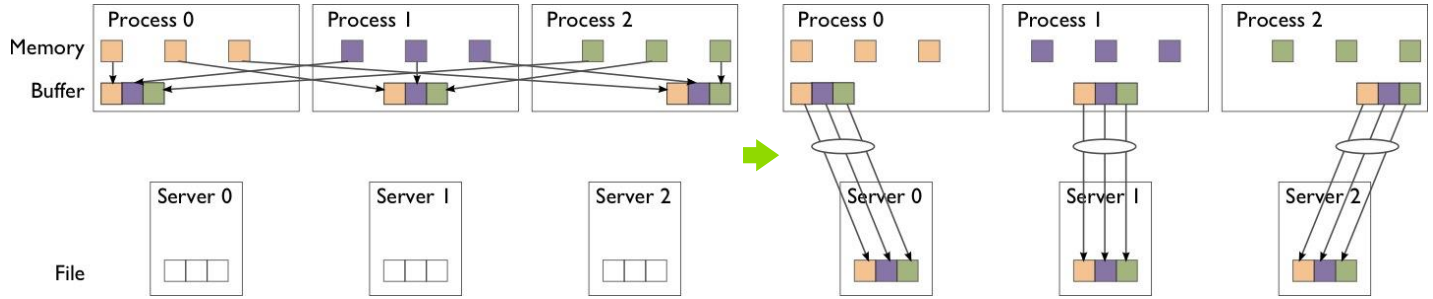
Gaps between operations show lock acquisition.



https://github.com/hpc-io/dxt-explorer Interactive log analysis tool by Jean Luca Bez

AVOIDING LOCK CONTENTION

- **To avoid lock contention when writing to a shared file, we can reorganize data between processes**
- *Two-phase I/O* splits I/O into a data reorganization phase and an interaction with the storage system (two-phase write depicted):
  - Data exchanged between processes to match file layout
  - 0th phase determines exchange schedule (not shown)

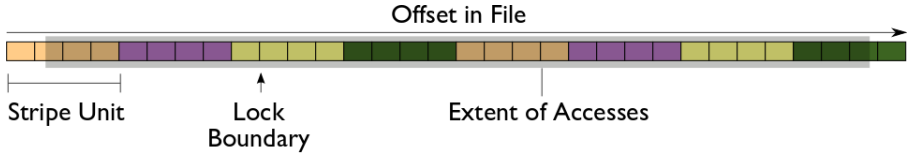


Phase 1: Data are exchanged between processes based on organization of data in file.

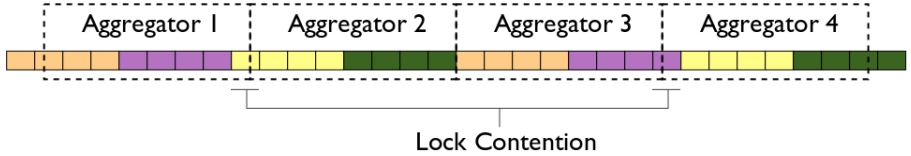
Phase 2: Data are written to file (storage servers) with large writes, no contention.

TWO-PHASE I/O ALGORITHMS

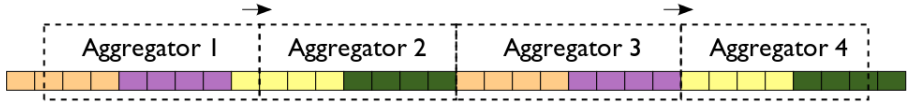
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



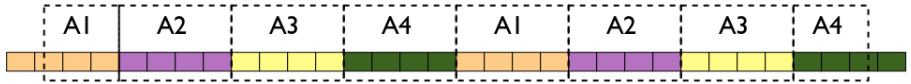
One approach is to evenly divide the region accessed across aggregators.



Aligning regions with lock boundaries eliminates lock contention.



Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).

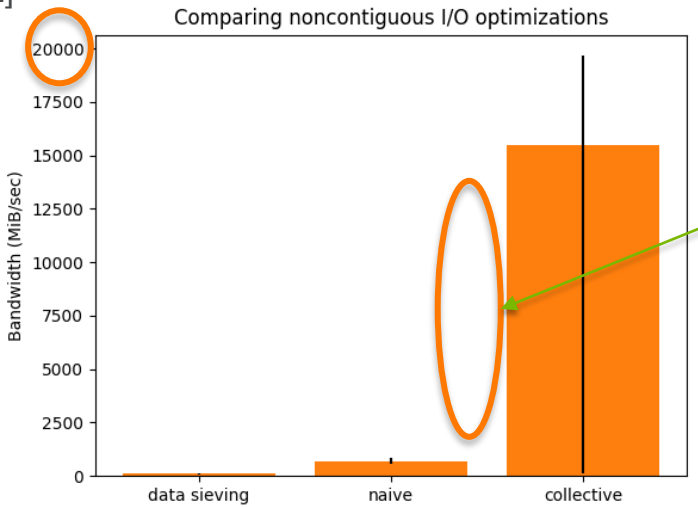


For more information, see W.K. Liao and A. Choudhary, “Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols,” SC2008, November 2008.

TWO-PHASE I/O IN PRACTICE

- Consistent performance independent of access pattern
  - Note re-scaled y axis [1]
- No write amplification, no read-modify-write
- Some network communication but networks are fast
- Requires “temporal locality” -- not great if writes “skewed”, imbalanced, or some process enter collective late.
- (Yes, those are some “impressive” error bars…)

[1]



[2]

	Naiive	Data Sieving	Two-phase
MPI-IO writes	192	192	**192**
MPI-IO Reads	0	0	**0**
Posix Writes	192000	192000	**1832**
Posix Reads	0	192015	**0**
MPI-IO bytes written	1 920 000 000	1 920 000 000	**1 920 000 000**
MPI-IO bytes read	0	0	**0**
Posix bytes read	0	100 039 006 128	**0**
Posix bytes written	1 920 000 000	100 564 552 704	**1 920 000 000**

Selected Darshan statistics

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

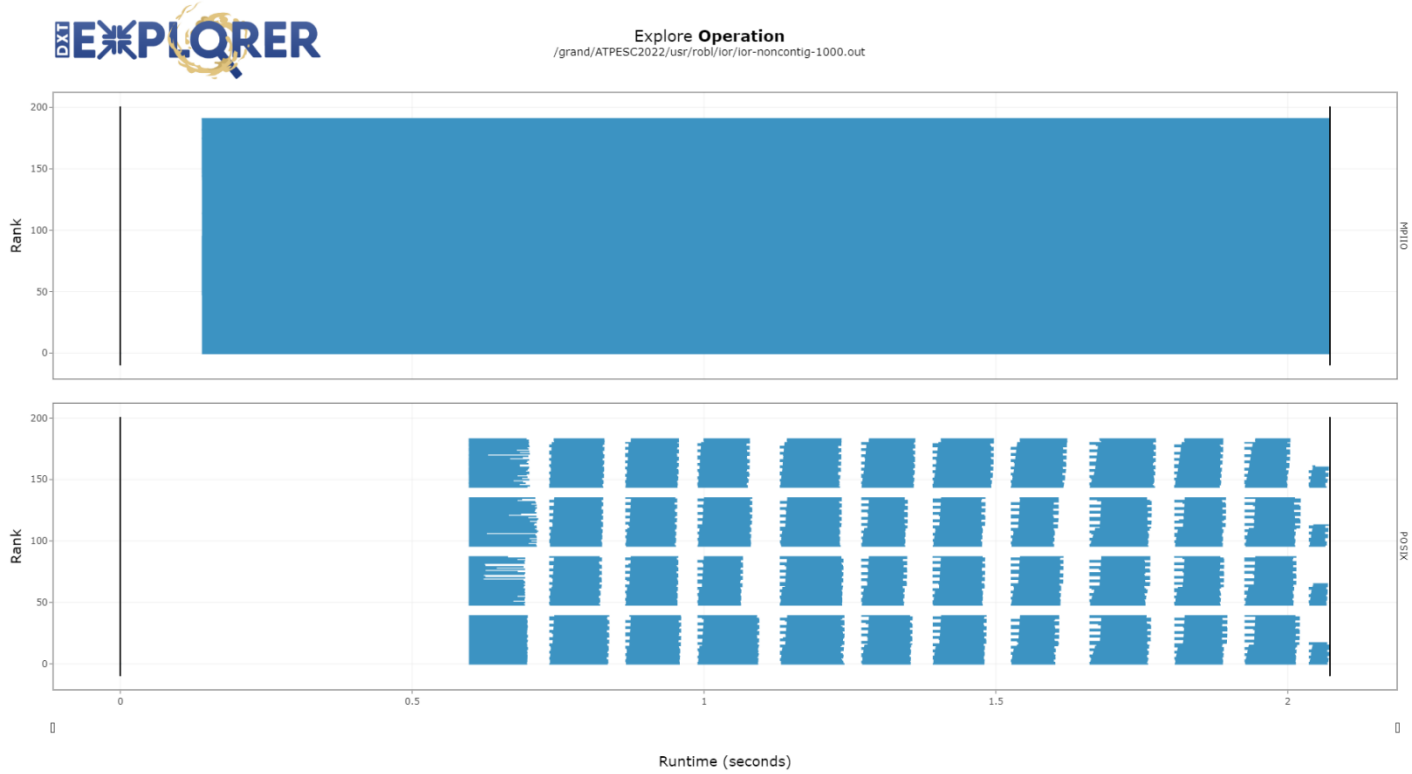
TWO-PHASE I/O: TIME LINE

Top: collective MPI I/O call describing noncontiguous regions

One collective MPI I/O call per process: library transforms request.

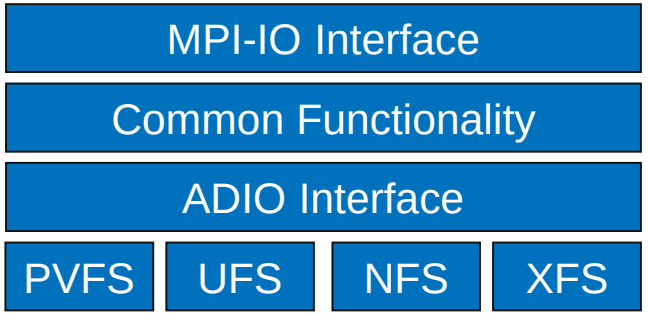
Lustre-specific optimization: select processes and request sizes based on file stripe size, stripe count.

Gaps between operations show data exchange over network



MPI-IO IMPLEMENTATIONS

- Different MPI-IO implementations exist
- Today two better-known ones are:
  - ROMIO from Argonne National Laboratory
    - Leverages MPI-1 communication
    - Supports local file systems, network file systems, parallel file systems
      - UFS module works on GPFS, Lustre, and others
    - Includes data sieving and two-phase optimizations
    - Basis for many vendor MPI implementations
    - https://wordpress.cels.anl.gov/romio/
  - OMPIO from OpenMPI
    - Emphasis on modularity and tighter integration into MPI implementation
    - https://docs.open-mpi.org/en/v5.0.x/faq/ompio.html



ROMIO's layered architecture.

HIGH LEVEL I/O LIBRARIES: HDF5 AND OTHERS

HIGHER LEVEL I/O INTERFACES

- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data in file
  - Interfaces for discovering contents

- Present APIs are more appropriate for computational science
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays and I/O on subsets of these arrays

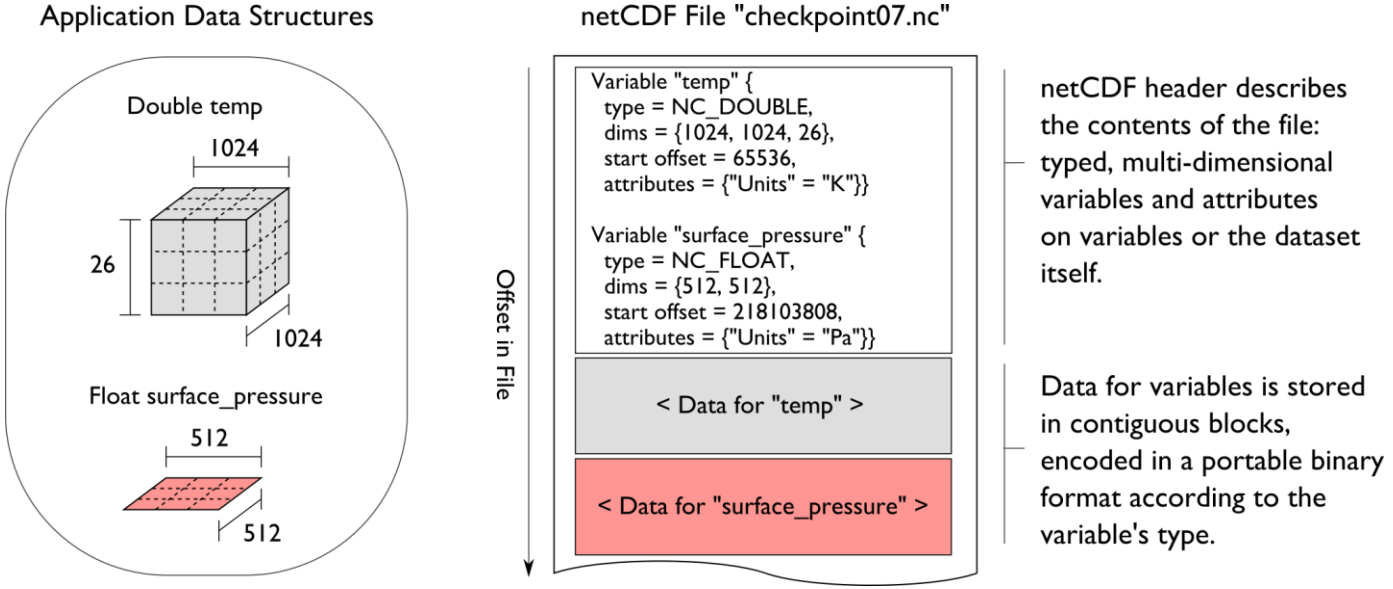
- Both of our example interfaces are implemented on top of MPI-IO

PARALLEL NETCDF (PNETCDF)

- Based on original “Network Common Data Format” (netCDF) work from Unidata
  - Derived from their source code
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C, Fortran, and F90 interfaces
  - Portable data format (identical to netCDF)
  - Noncontiguous I/O in memory using MPI datatypes
  - Noncontiguous I/O in file using sub-arrays
  - Collective I/O
  - Non-blocking I/O
- Unrelated to netCDF-4 work
- Parallel-NetCDF tutorial:
  - https://parallel-netcdf.github.io/wiki/QuickTutorial.html

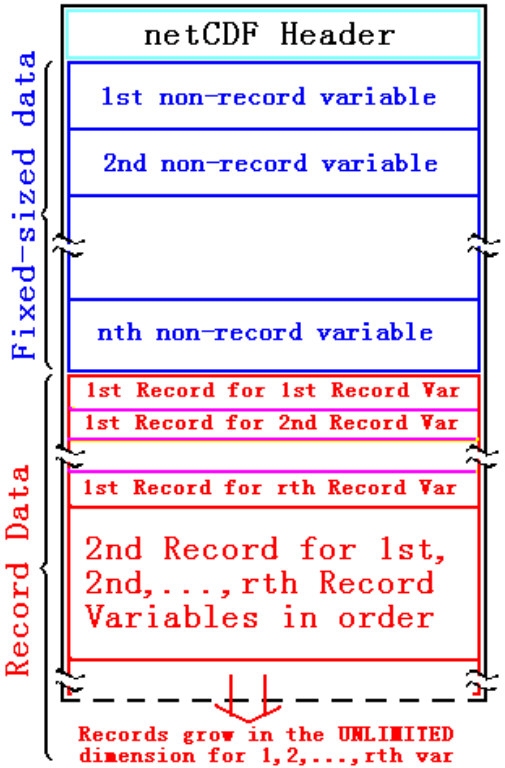
NETCDF DATA MODEL

The netCDF model provides a means for storing multiple, multi-dimensional arrays in a single file.



RECORD VARIABLES IN NETCDF

- Record variables are defined to have a single "unlimited" dimension
  - Convenient when a dimension size is unknown at time of variable creation
- Record variables are stored after all the other variables in an interleaved format
  - Using more than one in a file is likely to result in poor performance due to number of noncontiguous accesses



TUNING FOR PARALLEL-NETCDF

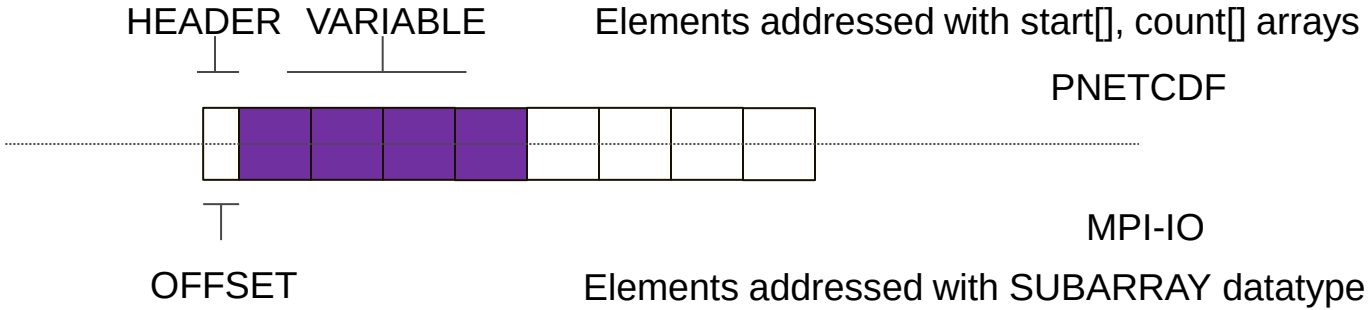
- Not a lot of tuning parameters in pnetcdf itself
  - All the MPI-IO tuning parameters apply
  - All the file system tuning parameters apply
- API choices can have large impact
- Record variables require careful consideration

PNETCDF TUNING: INFO OBJECT

- Create an Info object as you would for MPI codes
- Add key/value strings to info object
- Pass that to create or open routine
- A few hints are pnetcdf specific
  - Alignment of header, variables
  - Gating experimental features

```
MPI_Info_create(&info);
MPI_Info_set(info, "striping_factor", "-1");
MPI_Info_set(info, "romio_ds_write", "disable");
ncmpi_create(comm, filename, NC_CLOBBER,
      info, &ncfile)
```

PARALLEL-NETCDF AND MPI-IO

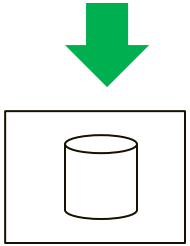


- `ncmpi_put_vara_all` describes access in terms of arrays, elements of arrays
  - For example, “Give me a 3x3 subcube of this larger 1024x1024 array”
- Library translates into MPI-IO calls
  - `MPI_Type_create_subarray`
  - `MPI_File_set_view`
  - `MPI_File_write_all`

PARALLEL-NETCDF WRITE-COMBINING OPTIMIZATION

```
ncmpi_iput_vara(ncfile, varid1,
     &start, &count, &buffer1,
     count, MPI_INT, &requests[0]);
ncmpi_iput_vara(ncfile, varid2,
     &start,&count, &buffer2,
     count, MPI_INT, &requests[1]);
ncmpi_wait_all(ncfile, 2, requests, statuses);
```

HEADER VAR1 VAR2



- netCDF variables laid out contiguously
- Applications typically store data in separate variables
  - temperature(lat, long, elevation)
  - Velocity_x(x, y, z, timestep)
- Operations posted independently, completed collectively
  - Defer, coalesce synchronization
  - Increase average request size

	Separate	Combined
POSIX writes	161	2
POSIX reads	0	1
MPI-IO indep writes	1	1
MPI-IO coll. writes	160	16

Selected Darshan stats from 16 processes each writing 10 variables to a dataset:
Note effects of data sieving and deciding against collective I/O
https://xgitlab.cels.anl.gov/ATPESC-IO/hands-on/blob/master/array/solutions/10-array-pnetcdf-op-combine-compare.c

PNETCDF EXAMPLE

- Write five variables to a file
- Do some “work”

Case 1: one collective at a time

```
srand(10+rank);
for (int j=0; j< NVARS; j++)  {
    /* mimic doing some computation
     * additionally, pretend the computation is unevenly distributed across
     * processes */
    usleep(rand()%5000000);

    ret = ncmpi_put_vara_all(ncfile, varid[j], start, count,
            write_buf, count[0], MPI_INT);
    if (ret != NC_NOERR) handle_error(ret, __LINE__);
}
```

Case 2: non-blocking operations

```
srand(10+rank);
for (int j=0; j< NVARS; j++)  {
    /* mimic doing some computation
     * additionally, pretend the computation is unevenly distributed across
     * processes */
    usleep(rand()%5000000);
    /* the non-blockig operations don't actually do any i/o, so we can issue
     * them non-collectively */
    ret = ncmpi_iput_vara(ncfile, varid[j], start, count,
            write_buf, count[0], MPI_INT,
            &requests[j]);
}
/* in the non-blocking case, this collective wait routine is where all the
 * work happens: there is no background i/o thread  */
ret = ncmpi_wait_all(ncfile, NVARS, requests, statuses);
if (ret != NC_NOERR) handle_error(ret, __LINE__);

/* check status of each nonblocking call */
for (int j=0; j< NVARS; j++)
    if (statuses[j] != NC_NOERR) handle_error(statuses[j], __LINE__);
```

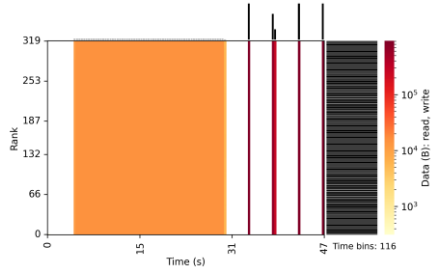
https://github.com/radix-io/io-sleuthing/tree/main/examples/pnetcdf

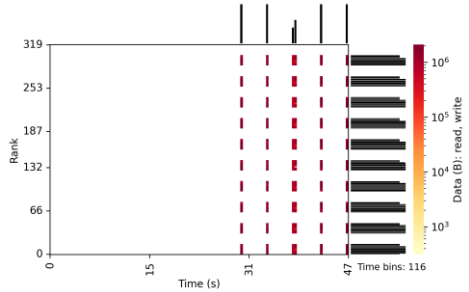
COMPARING APPROACHES WITH DARSHAN

MPI-IO

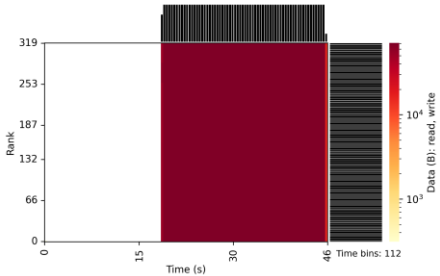
POSIX

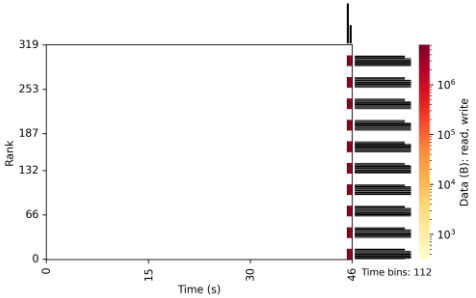
blocking





Non-blocking





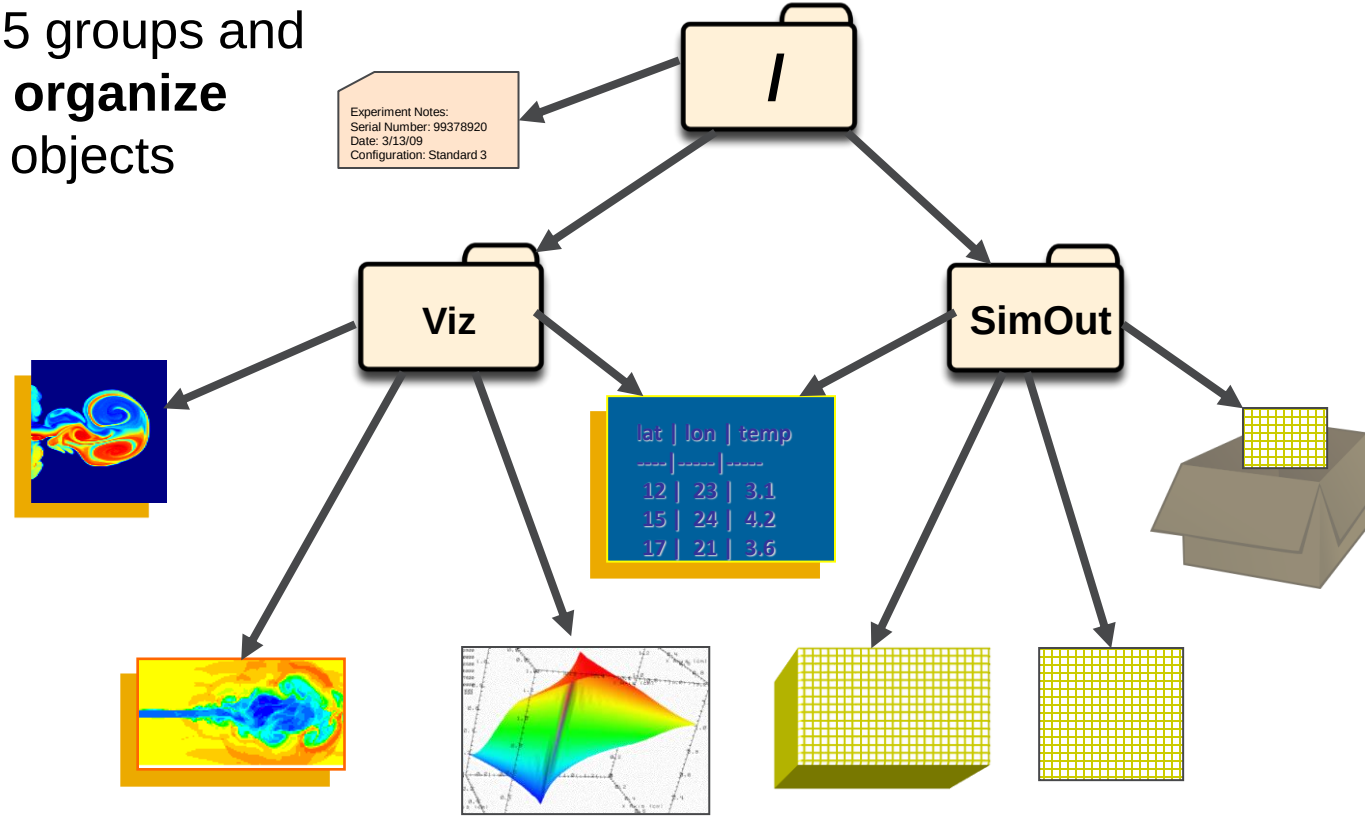
The HDF5 Interface and File Format

HDF5

- Hierarchical Data Format, from The HDF Group (formerly of NCSA)
  - https://www.hdfgroup.org/
- Data Model:
  - Hierarchical data organization in single file
  - Typed, multidimensional array storage
  - Attributes on any HDF5 "object" (dataset, data, groups)
- Features:
  - C, C++, Fortran, Java (JNI) interfaces
    - Community-supported Python, Lua, R
  - Portable data format
  - Optional compression (even in parallel I/O mode)
  - Chunking: efficient row or column oriented access
  - Noncontiguous I/O (memory and file) with hyperslabs
- Parallel HDF5 tutorial:
  - https://portal.hdfgroup.org/display/HDF5/Introduction+to+Parallel+HDF5

HDF5 GROUPS AND LINKS

HDF5 groups and links **organize** data objects



INITIALIZE THE FILE FOR PARALLEL ACCESS

```
/* first initialize MPI */

/* create access property list */
plist_id = H5Pcreate(H5P_FILE_ACCESS);

/* necessary for parallel access */
status = H5Pset_fapl_mpio(plist_id,
MPI_COMM_WORLD, MPI_INFO_NULL);

/* Create an hdf5 file */
file_id = H5Fcreate(FILENAME,
H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);

status = H5Pclose(plist_id);
```

CREATE PROPERTY LIST

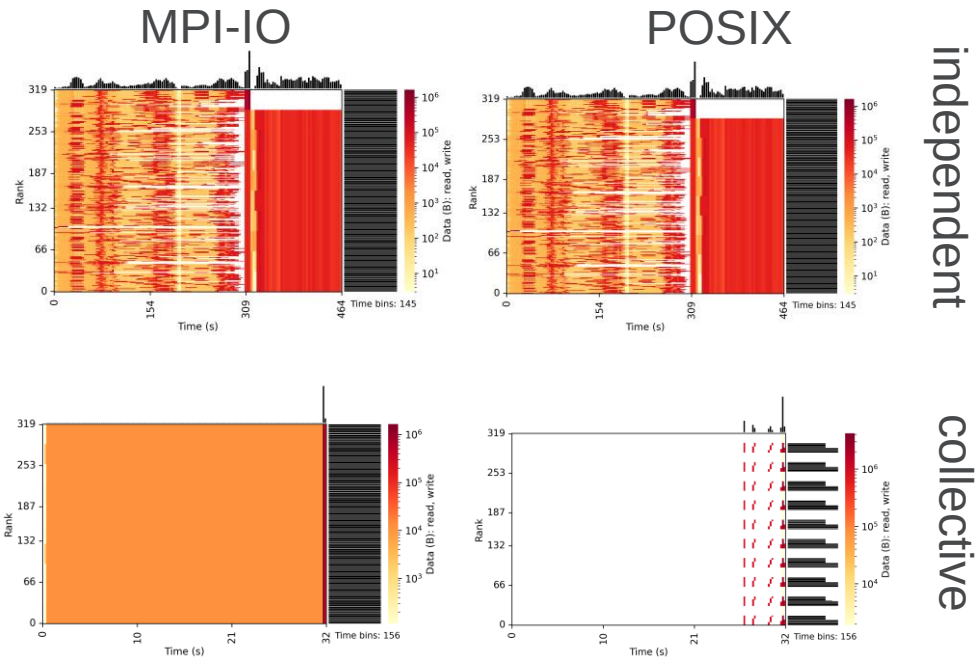
```
/* Create property list for collective dataset
write. */

plist_id = H5Pcreate(H5P_DATASET_XFER);

/* The other option is HDFD_MPIO_INDEPENDENT */
H5Pset_dxpl_mpio(plist_id,H5FD_MPIO_COLLECTIVE);
```

EFFECT OF HDF5 TUNING

- HDF5 property lists can have big impact on internal operations

- Collective I/O vs. Independent I/O
  - Huge reduction in operation count
  - Implies all processes hit I/O at same time

- Collective metadata (new in 1.10.2)
  - Further reduction in op count, especially reads (reading HDF5 internal layout information)
  - Big implications for performance at scale



https://github.com/radix-io/io-sleuthing/tree/main/examples/hdf5

NEW HDF5 FEATURES:

- New in HDF5-1.14.0
- Async operations
  - Potential for background progress
- Multi-dataset I/O
  - Similar to pnetcdf “operation combining”

DATA MODEL I/O LIBRARIES

- Parallel-NetCDF: http://www.mcs.anl.gov/pnetcdf
- HDF5: http://www.hdfgroup.org/HDF5/
- NetCDF-4: http://www.unidata.ucar.edu/software/netcdf/netcdf-4/
  - netCDF API with HDF5 back-end
- ADIOS: http://adiosapi.org
  - Configurable (xml) I/O approaches
- SILO: https://wci.llnl.gov/codes/silo/
  - A mesh and field library on top of HDF5 (and others)
- H5part: http://vis.lbl.gov/Research/AcceleratorSAPP/
  - simplified HDF5 API for particle simulations
- GIO: https://svn.pnl.gov/gcrm
  - Targeting geodesic grids as part of GCRM
- PIO:
  - climate-oriented I/O library; supports raw binary, parallel-netcdf, or serial-netcdf (from master)
- … Many more: consider existing libs before deciding to make your own.
- Note absence of a “machine learning” library – research opportunity for someone!

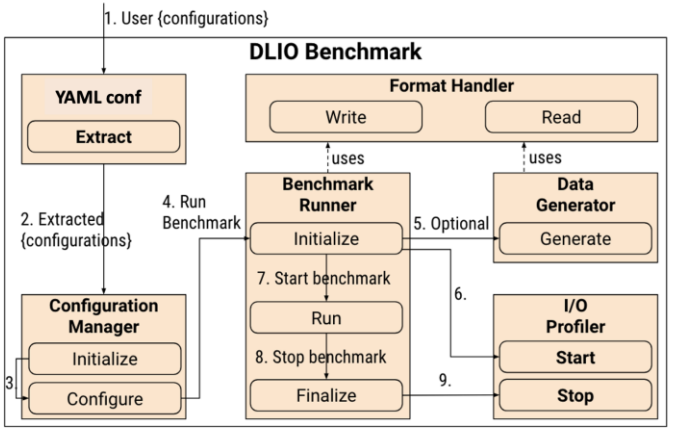
ML WORKLOADS

SIMULATION I/O VS AI I/O

- Simulation:
  - Collective reads/writes
  - Could be irregular, small, but never random
  - Standardization: MPI and libraries
- AI
  - Repeated reads of training model
  - No coordination among processes
  - No “middleware for AI” (yet)
  - Sometimes part of a workflow, not a single application

DLIO: A DEEP LEARNING BENCHMARK

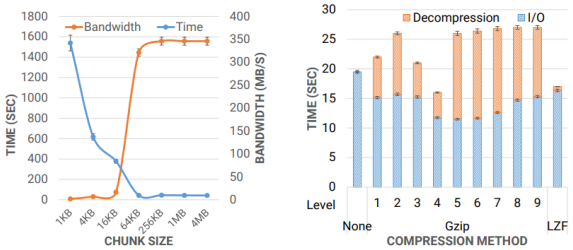
- Framework for evaluating deep learning I/O performance
- Replaces computationally intensive phases with sleep
- But performs I/O the way deep learning framework would
  - Calls “torch DataLoader” or “tensorflow.data” loaders with synthetic data
- More information:
  - https://argonne-lcf.github.io/dlio_benchmark a
  - https://github.com/argonne-lcf/dlio_benchmark



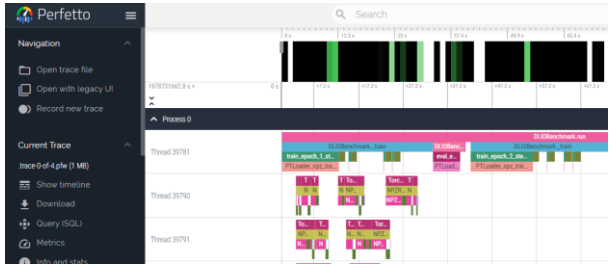
DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications, Harihan Devarajan et al, CCGrid21

DLIO TRAINING: RAPID EXPERIMENTATION

- Able to sweep across a wide range of parameters quickly
- Lessons learned in DLIO simulations apply directly to actual ML framework/workflow
- Helps improve Darshan
- Future work: integrating DLIO instrumentation with Darshan's py-darshan interface for improved reporting



Experimenting with access ("chunk") size and compression; from *DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications*, Harihan Devarajan et al, CCGrid21



DLIO tracing of workflow

ADDITIONAL TOPICS

- Helpful tools:
  - Ltrace and strace
    - Confirming behavior of I/O libraries
  - Gdb
    - “why is everyone stalled in this collective?”
- Technologies
  - GPU programming?
    - NVIDIA’s ‘gpu direct storage’: https://developer.nvidia.com/gpudirect-storage
  - DAOS:
    - Novel storage architecture, showing up on Aurora

BIG PICTURE SUMMARY

- I/O subsystems complex with lots of layers
- Initial experiences not likely to be ideal
- Use libraries and frameworks (where available)
  - Portability across file systems, machines, storage technologies
- Darshan helps Scientists and I/O folks meet on common ground
- Consultants at your site (e.g. ALCF, OLCF, NERSC) love solving problems

ACKNOWLEDGEMENTS

- Some material drawn from *Parallel I/O In Practice,* our full day SC tutorial. Thanks to Brent Welch, Rob Ross, Glenn Lockwood, Katie Antypas, Marc Unangst, Rajeev Thakur, and Bill Loewe.
- Some material drawn from ATPESC “Data and I/O Day”. Thanks to Phil Carns and Shane Snyder
- This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.
- I carried out this work under a 2022 Better Scientific Software (BSSW) fellowship. Thanks to BSSW and ECP IDEAS for their support.