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:

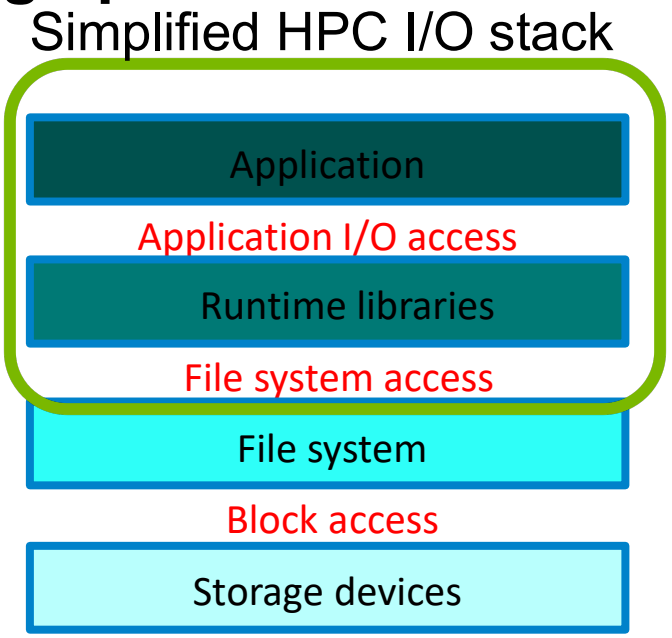
- P. Carns, et al., “Understanding and improving computational science storage access through continuous characterization,” *ACM TOS* 2011.
- P. Carns, et al., “Production I/O characterization on the Cray XE6,” CUG 2013. May 2013.

CHARACTERIZING APPLICATION I/O

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

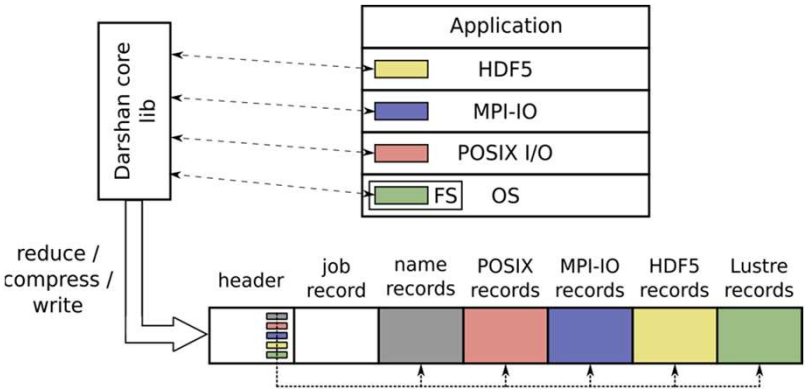
Strategy: observe I/O behavior at the application and library level

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



HOW DOES DARSHAN WORK?

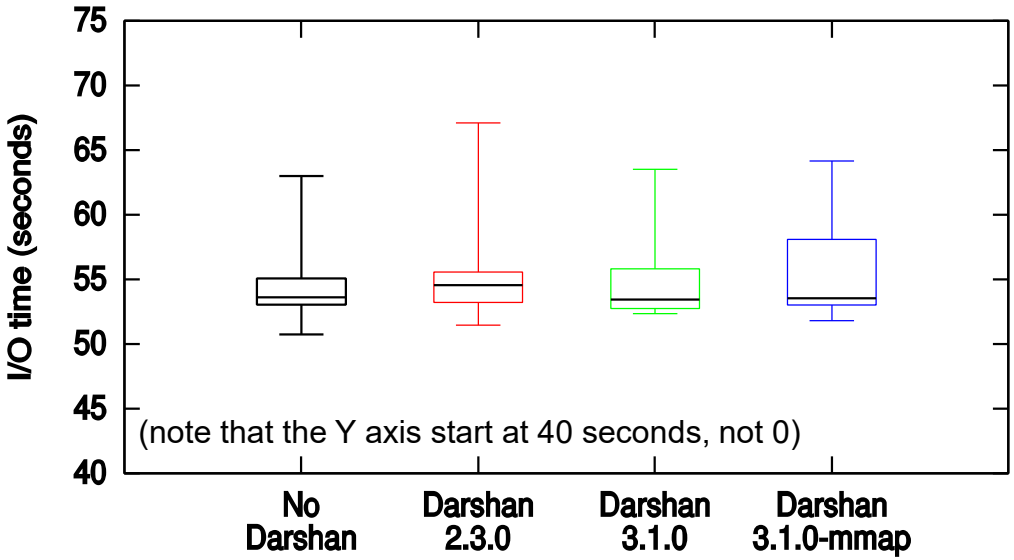
- Darshan records file access statistics independently on each process
- At app shutdown, collect, aggregate, compress, and write log data
- After job completes, analyze Darshan log data
  - `darshan-parser` - provides complete text-format dump of all counters in a log file
  - *PyDarshan* - Python analysis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

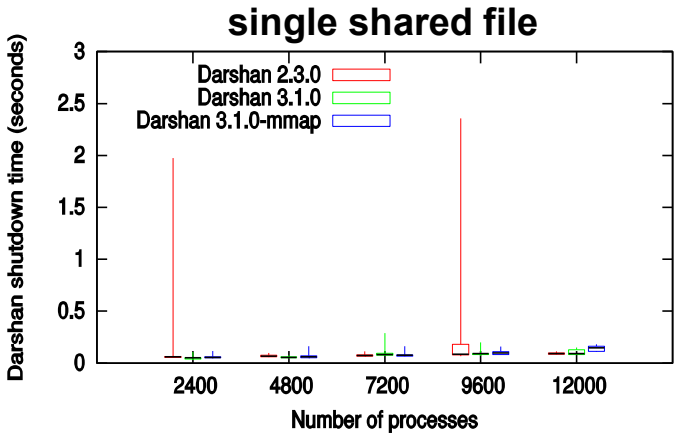
- Compare I/O time of IOR linked against different Darshan versions on *NERSC Edison*
  - File-per-process workload
  - 6,000 MPI processes
  - >12 million instrumented calls
- Note use of box plots
  - Ran each test 15 times
  - I/O variation is a reality
  - Consider I/O performance as a distribution,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



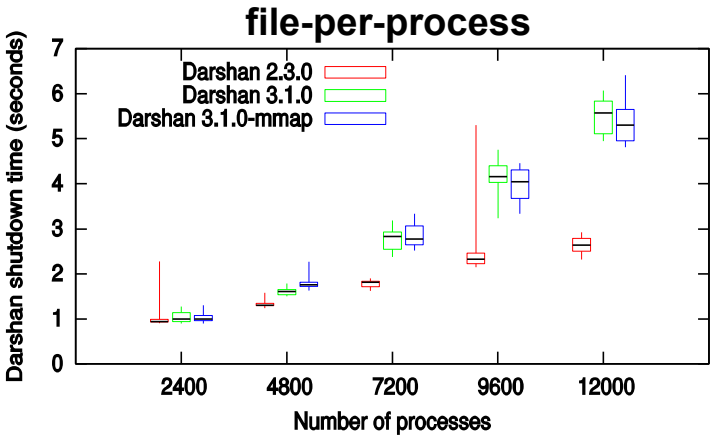
Snyder et al., "Modular HPC I/O Characterization with Darshan," in *Proceedings of 5th Workshop on Extreme-scale Programming Tools (ESPT 2016),* 2016.

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

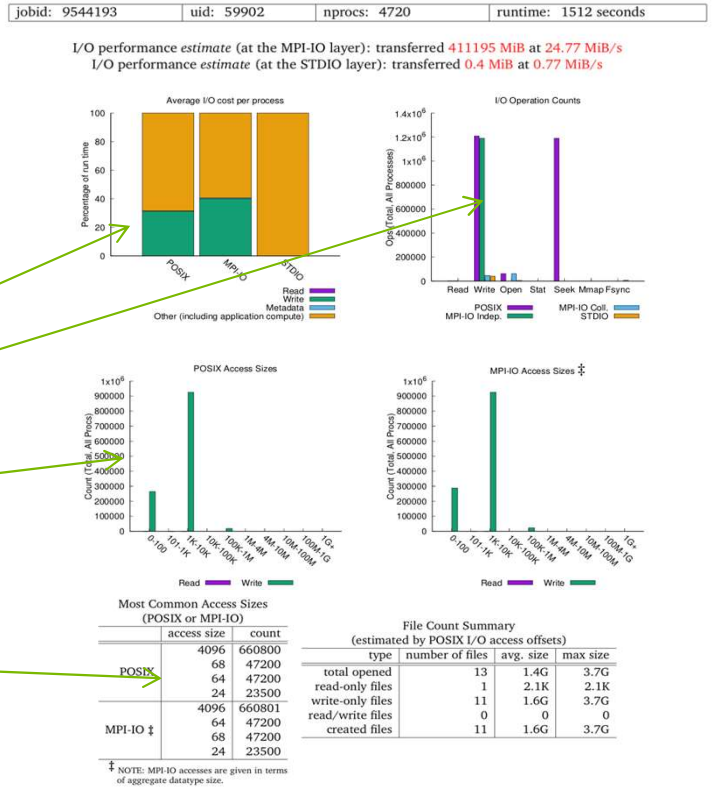
- After locating your log, users can utilize Darshan log analysis tools for gaining insights into application I/O behavior:

```
shane@shane-x1-carbon: darshan-parser ./log.darshan | grep POSIX_BYTES_WRITTEN | sort -nr -k 5
POSIX   387     6966057185861764086     POSIX_BYTES_WRITTEN     5413869452      /projects/radix
POSIX   452     6966057185861764086     POSIX_BYTES_WRITTEN     5413865644      /projects/radix
POSIX   197     6966057185861764086     POSIX_BYTES_WRITTEN     5413857652      /projects/radix
POSIX   5       6966057185861764086     POSIX_BYTES_WRITTEN     5413852168      /projects/radix
POSIX   451     6966057185861764086     POSIX_BYTES_WRITTEN     5413844532      /projects/radix
POSIX   64      6966057185861764086     POSIX_BYTES_WRITTEN     5413823236      /projects/radix
POSIX   68      6966057185861764086     POSIX_BYTES_WRITTEN     5413788992      /projects/radix
POSIX   195     6966057185861764086     POSIX_BYTES_WRITTEN     5413663132      /projects/radix
POSIX   323     6966057185861764086     POSIX_BYTES_WRITTEN     5413658668      /projects/radix
POSIX   132     6966057185861764086     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

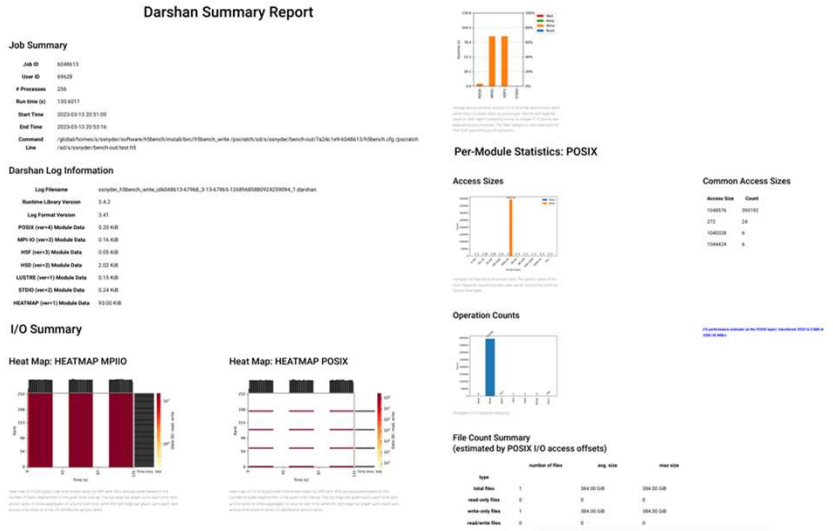
‡ NOTE: MPI-IO accesses are given in terms of aggregate datatype size.

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

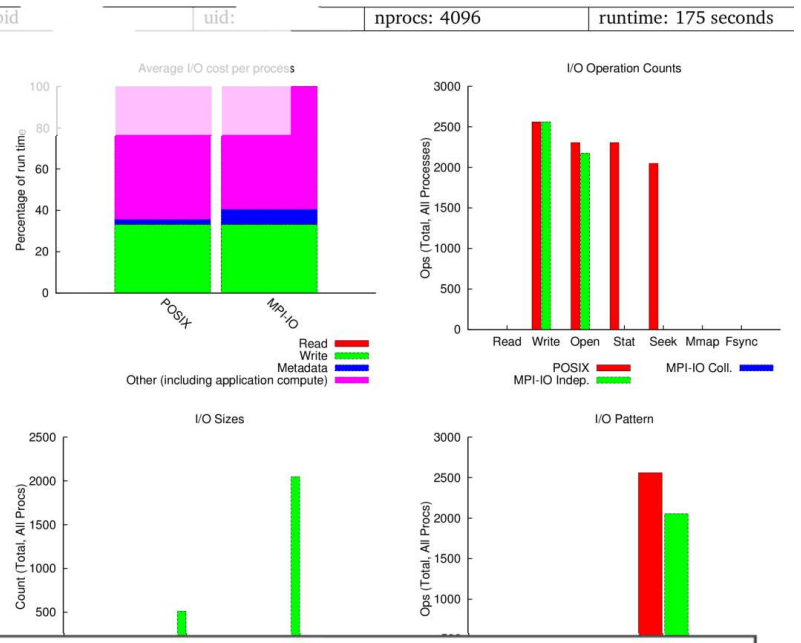
ANALYZING I/O PERFORMANCE PROBLEMS WITH DARSHAN

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

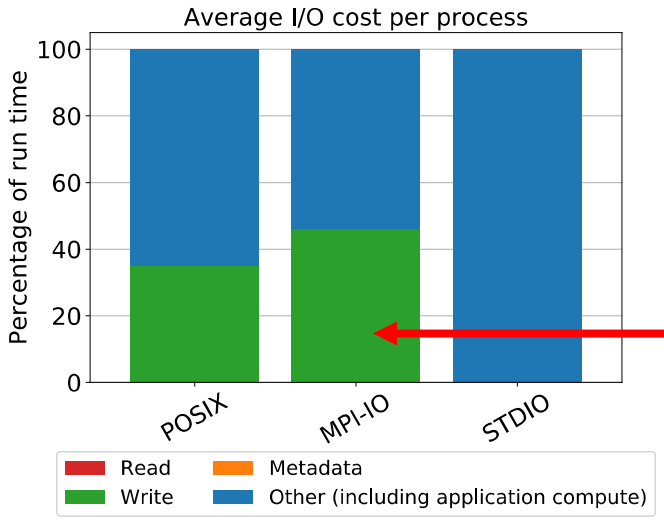
EXAMPLE: WHEN DOING THE RIGHT THING GOES WRONG

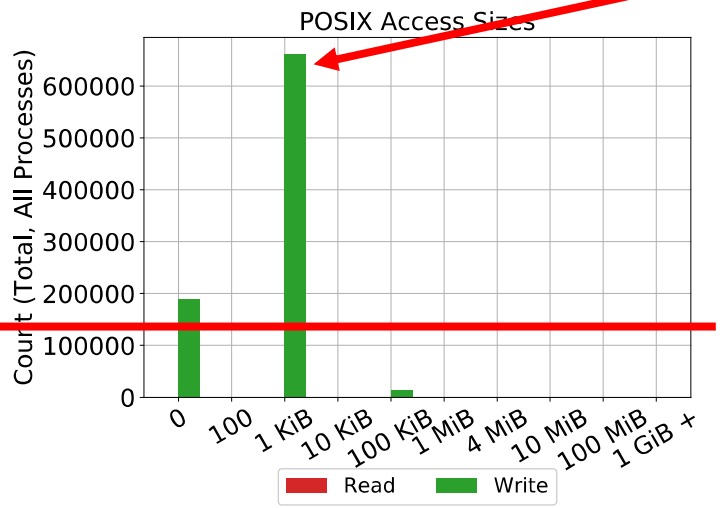
- Large-scale astrophysics application using MPI-IO
- Used 94,000,000 CPU hours at NERSC since 2015

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 ~~1675.25 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)

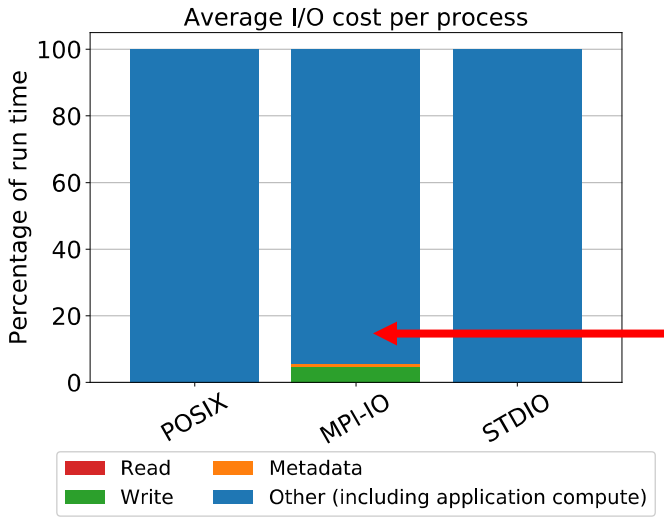
DOING THE RIGHT THING GOES WRONG

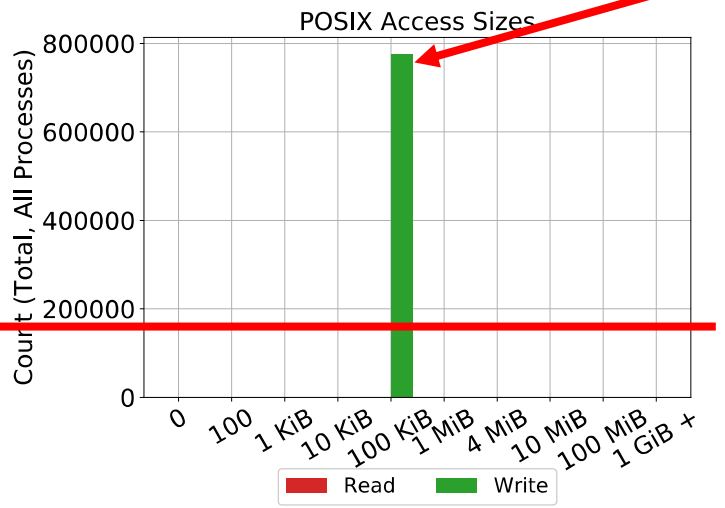
- Collective I/O was being disabled by middleware due to type mismatch in app code
- After type mismatch bug fixed, collective I/O gave 40× speedup

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

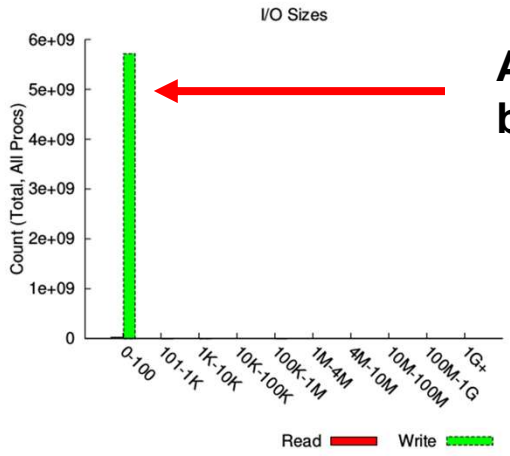
Data Transfer Per Filesystem

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

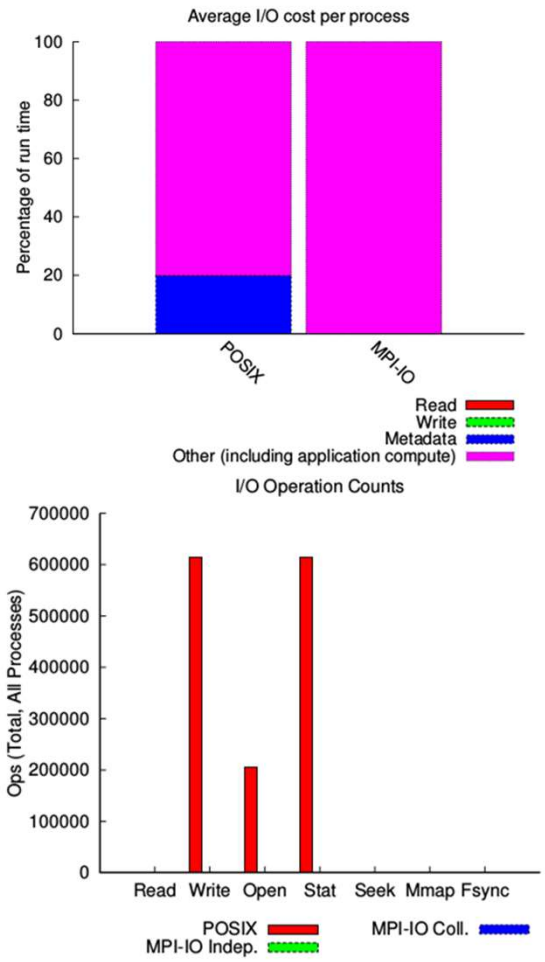
access size	count
1	3418409696
15	2275400442
24	42289948
12	14725053

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.

Slide 54

CMS1 I added a date to this reference, but I just guessed based
Steele, Carolyn M., 11/5/2018