ARGONNE
ATPESC2025
EXTREME-SCALE COMPUTING

Principles of HPC I/O

Phil Carns
carns@mcs.anl.gov
Mathematics and Computer Science Division
Argonne National Laboratory

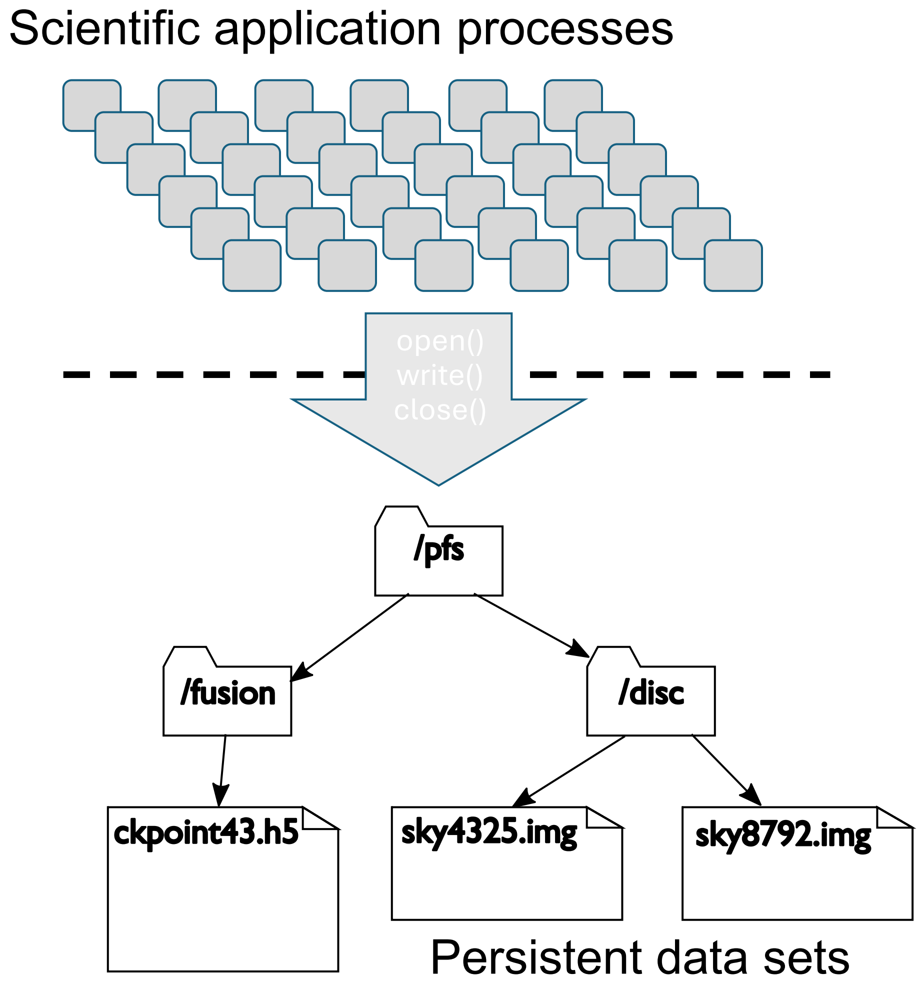
August 7, 2025

What is HPC I/O?

HPC I/O: storing and retrieving persistent scientific data on a high-performance computing platform

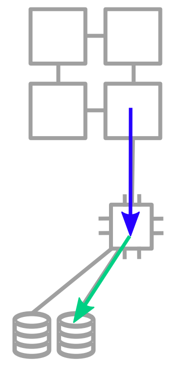
- Data is usually stored on a **parallel file system** that has been optimized to rapidly store and access enormous volumes of data.
- *This is an important job! Valuable CPU time is wasted if applications spend too long waiting for data.*
- It also means that parallel file systems are quite specialized and have some unusual properties.

Today's lectures are really all about the proper care and feeding of exotic parallel file systems.



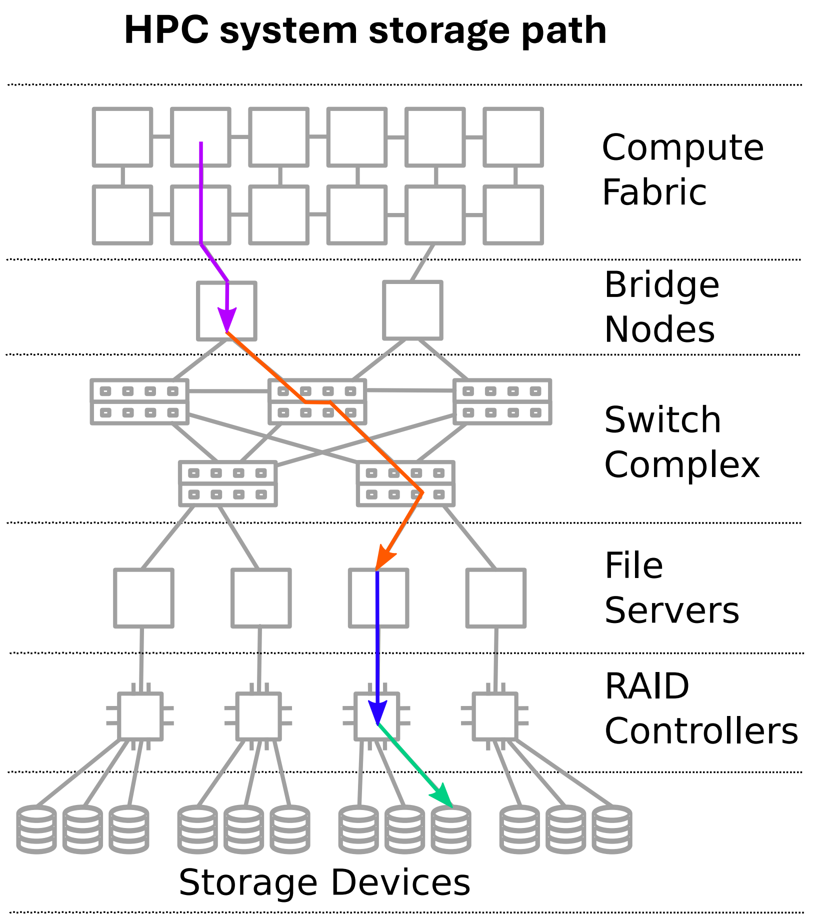
A look under the hood

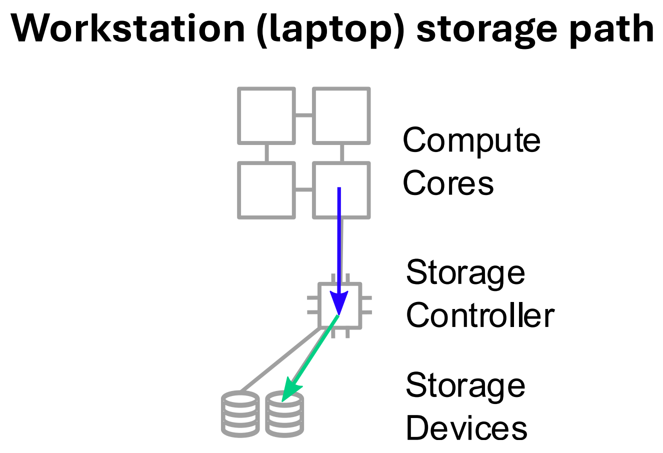
Workstation (laptop) storage path



- A typical workstation/laptop has only one storage device.
- The path between applications and storage is *short*.
- Properties:
  - Low latency
  - Low bandwidth

A look under the hood

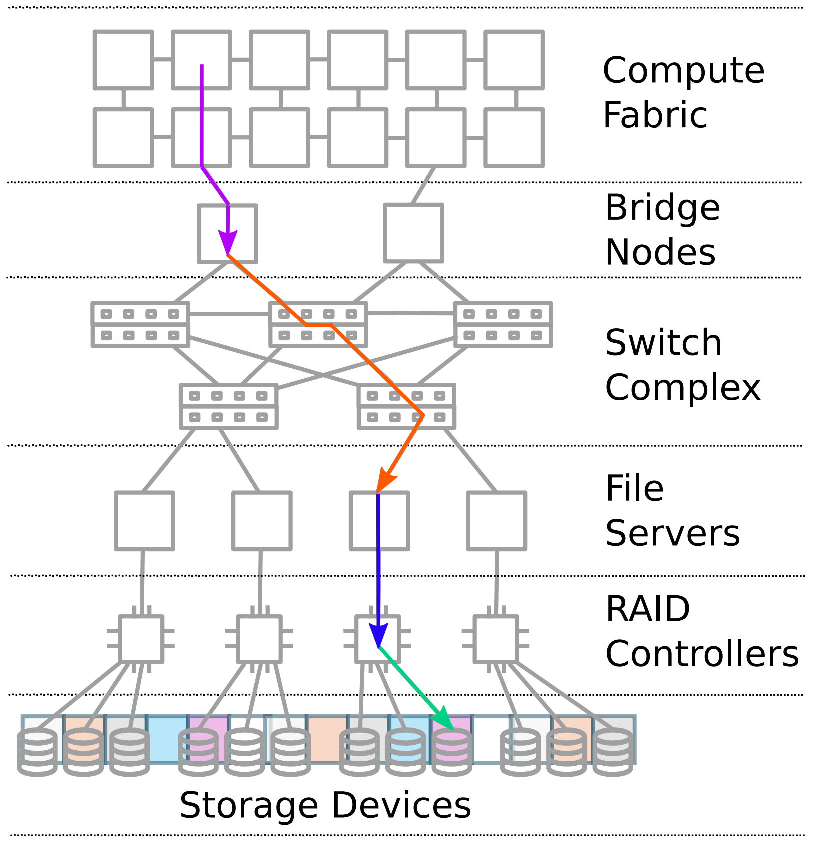




- In contrast, an HPC storage system manages many (e.g., **thousands** of) disaggregated devices.
- Paths between applications and storage devices are quite long, but numerous.
- Properties:
  - High latency
  - High bandwidth

Striping / Layout

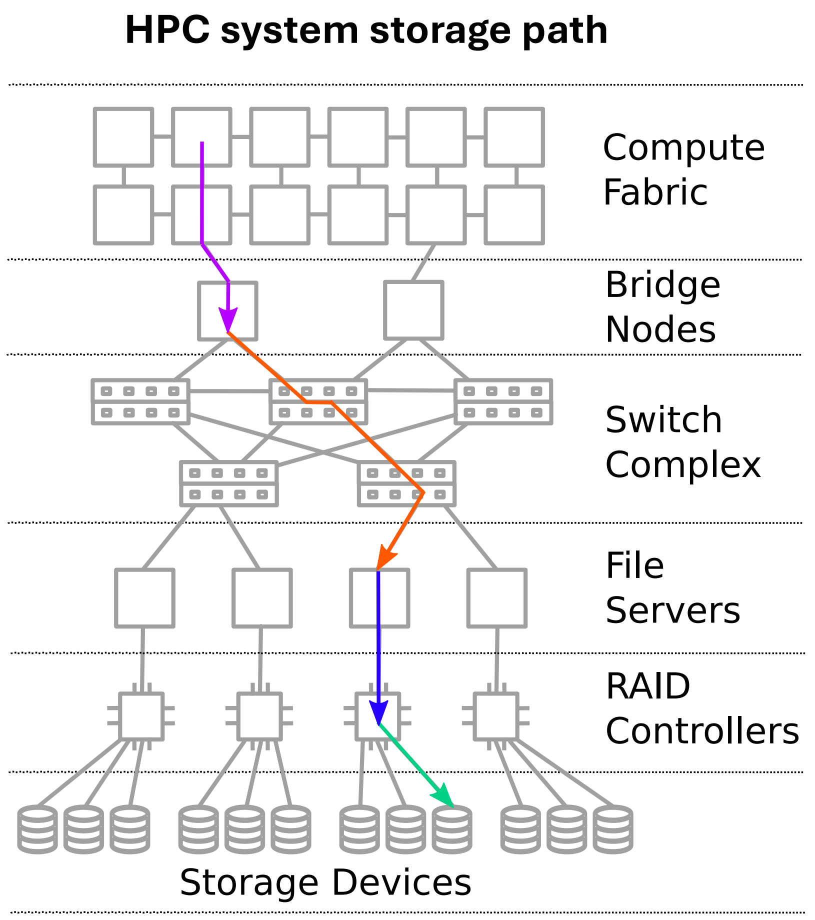
HPC system storage path



- Large files (or sets of files) are not generally stored on a single storage device.
- They are distributed across multiple servers (and then each server further distributes across storage devices).
- This is referred to as **data layout** or **striping**.
- Different file systems use different striping strategies.
- It can usually be tuned to better suit your application.

Example of a single logical file striped across all available servers and storage devices

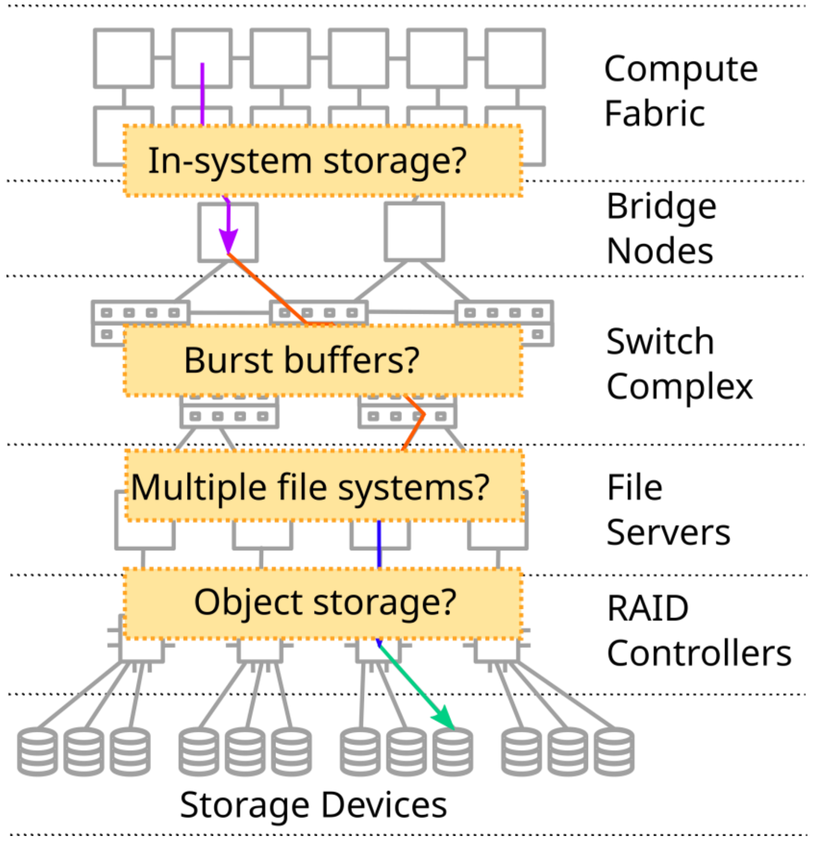
Is that all?


HPC system storage path
Compute Fabric
Bridge Nodes
Switch Complex
File Servers
RAID Controllers
Storage Devices

Is that all?

We'll give some examples near the end of this talk.

HPC system storage path



Each HPC storage system is a special unicorn. Some systems have:

- **In-system storage**: low latency but not shared
- **Burst buffers**: high performance with limited capacity
- **Multiple file systems**: storage systems optimized for different kinds of data
- **Object stores**: alternative methods of organizing data

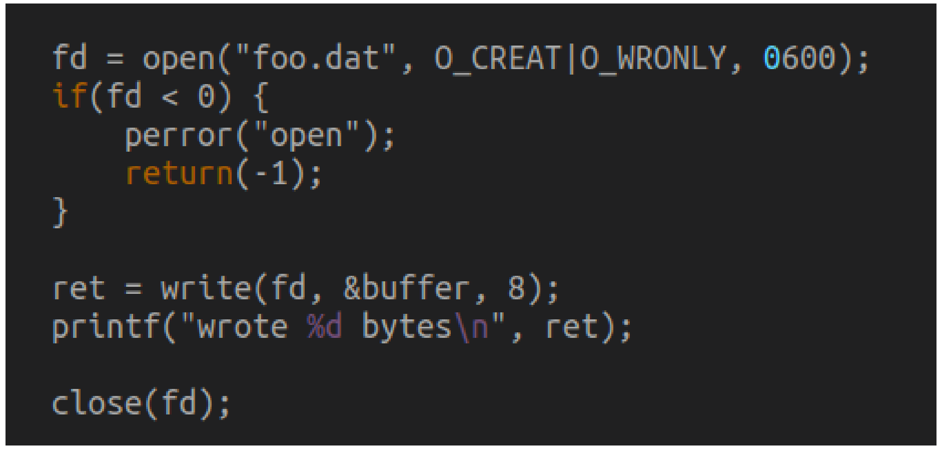
Don't worry. The tools and techniques that we will teach today will help to tame this complexity. The important thing to know for now is *why* HPC storage systems need specialized techniques.

Presenting storage to HPC applications

A parallel file system can be accessed just like any other file system:

- open() / close() / read() / write() for binary data
- fopen() / fclose() / fprintf() for text data
- Various language-specific bindings

- Data is organized in a hierarchy of directories and files.
- We call this API the “POSIX interface”; it is standardized across all UNIX-like systems.
- This API works, and is great for compatibility, but it was created 50 years ago before the rise of parallel computing.



```
fd = open("foo.dat", O_CREAT|O_WRONLY, 0600);
if(fd < 0) {
    perror("open");
    return(-1);
}

ret = write(fd, &buffer, 8);
printf("wrote %d bytes\n", ret);

close(fd);
```

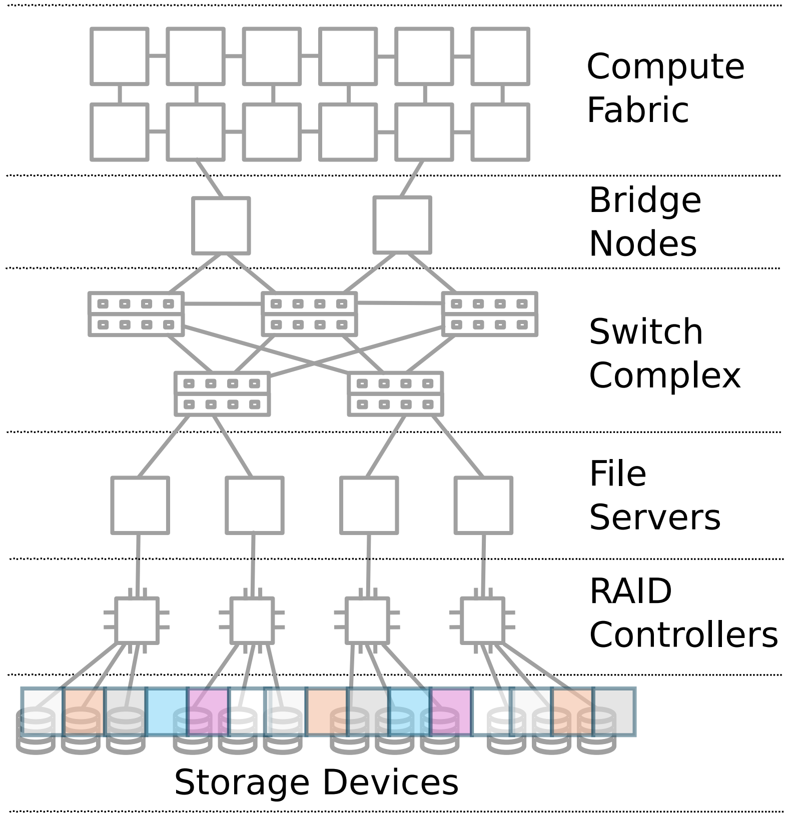
- The API has no concept of parallel access; semantics for that are largely undefined.
- Files are unstructured streams of bytes.
- File descriptors are stateful and unique to each process.

Why is it difficult to access files concurrently with POSIX?
Example 1: writing different parts of the same file

- Consider a case in which two ranks write data simultaneously to different parts of a file.
- In this example, we have a big gap (32 MiB) between them. Assume we are writing reasonably large chunks to optimize bandwidth.

Rank 0: lseek(0); write(256 KiB);

Rank 1: lseek(32 MiB); write(256 KiB);



ATPESC2025

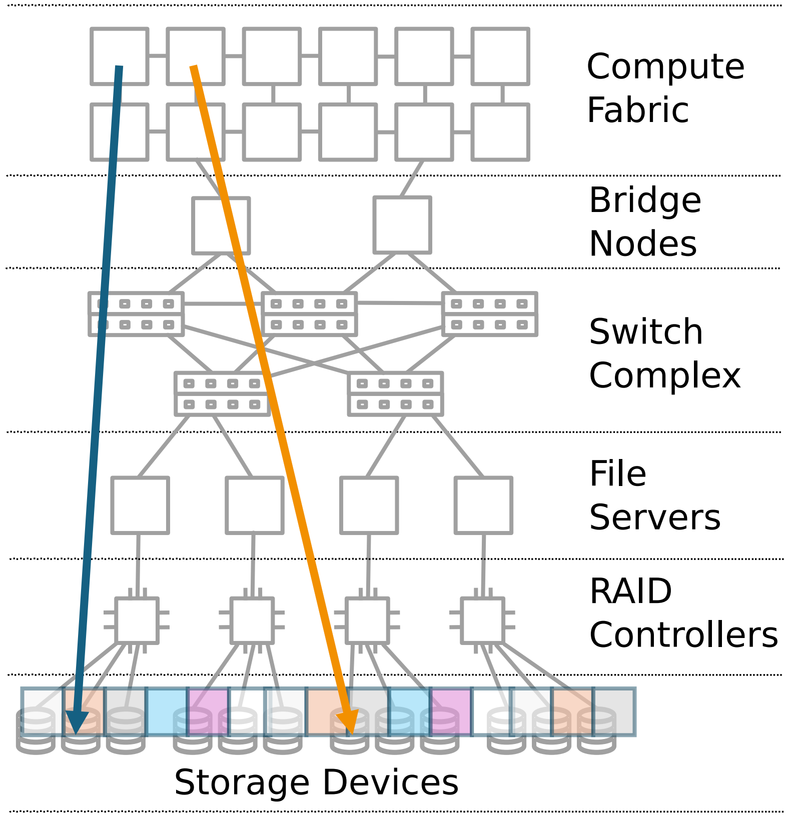
Hands on exercises: https://github.com/radix-io/hands-on

Argonne NATIONAL LABORATORY

Why is it difficult to access files concurrently with POSIX?
Example 1: writing different parts of the same file

- Consider a case in which two ranks write data simultaneously to different parts of a file.
- In this example, we have a big gap (32 MiB) between them. Assume we are writing reasonably large chunks to optimize bandwidth.
- ✓ The writes probably map to different servers and devices.
- ✓ There is no device contention, and both I/O operations can be executed at the same time.
- ? There is also **no coordination** of which path each write will take, though, and eventually you will want access adjacent data…

Rank 0: lseek(0); write(256 KiB);
Rank 1: lseek(32 MiB); write(256 KiB);

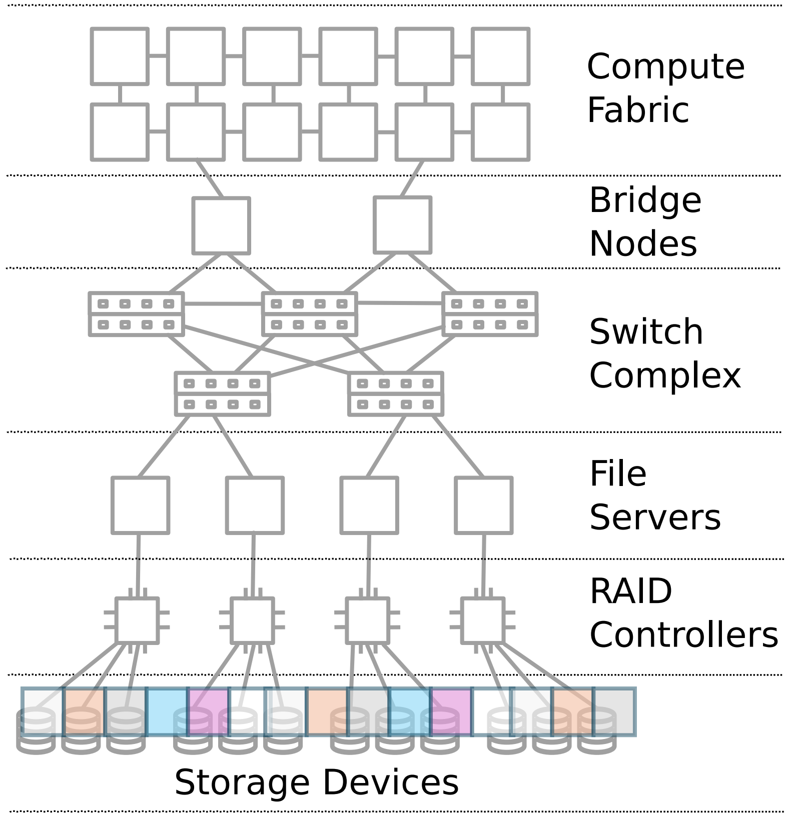


Why is concurrent access hard?
Example 2: writing adjacent parts of the same file

- Consider a case in which two ranks write data simultaneously to different parts of a file.
- In this case the writes still don't overlap, but they access adjacent bytes.

Rank 0: lseek(0); write(256 KiB);

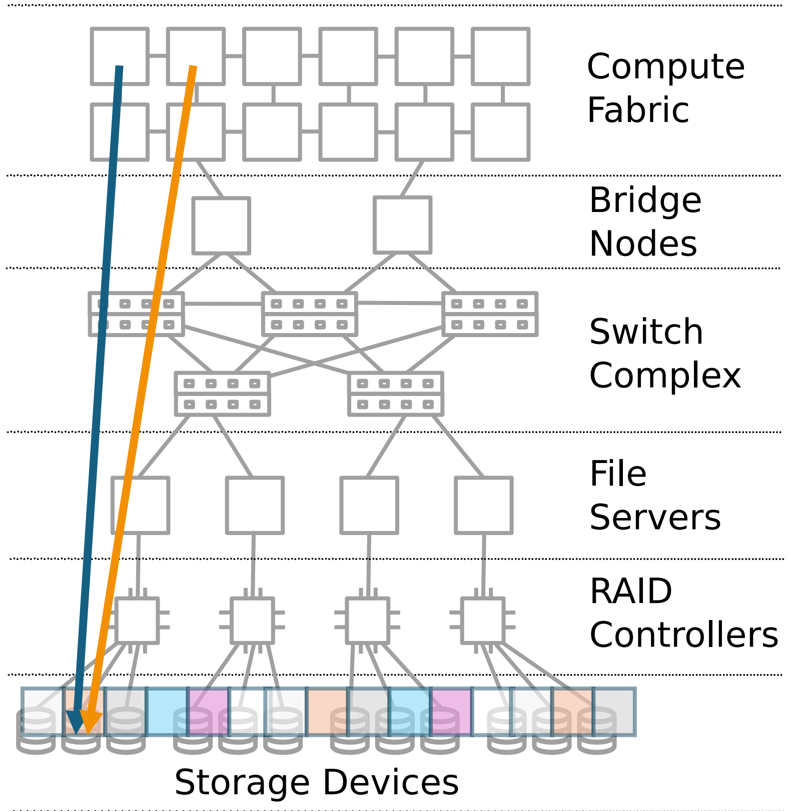
Rank 1: lseek(256 KiB); write(256 KiB);



Why is concurrent access hard?
Example 2: writing adjacent parts of the same file

- Consider a case in which two ranks write data simultaneously to different parts of a file.
- In this case the writes still don’t overlap, but they access adjacent bytes.
- ? The writes are likely to access the same server and storage device because of locality.
- ☒ Counterintuitively, this **almost certainly causes conflicting access**. The granularity of caching and locking has no relationship to the access size.
- ☒ Uncoordinated adjacent access can cause “false sharing” and serialize I/O operations that should have proceeded in parallel.

Rank 0: lseek(0); write(256 KiB);
Rank 1: lseek(256 KiB); write(256 KiB);

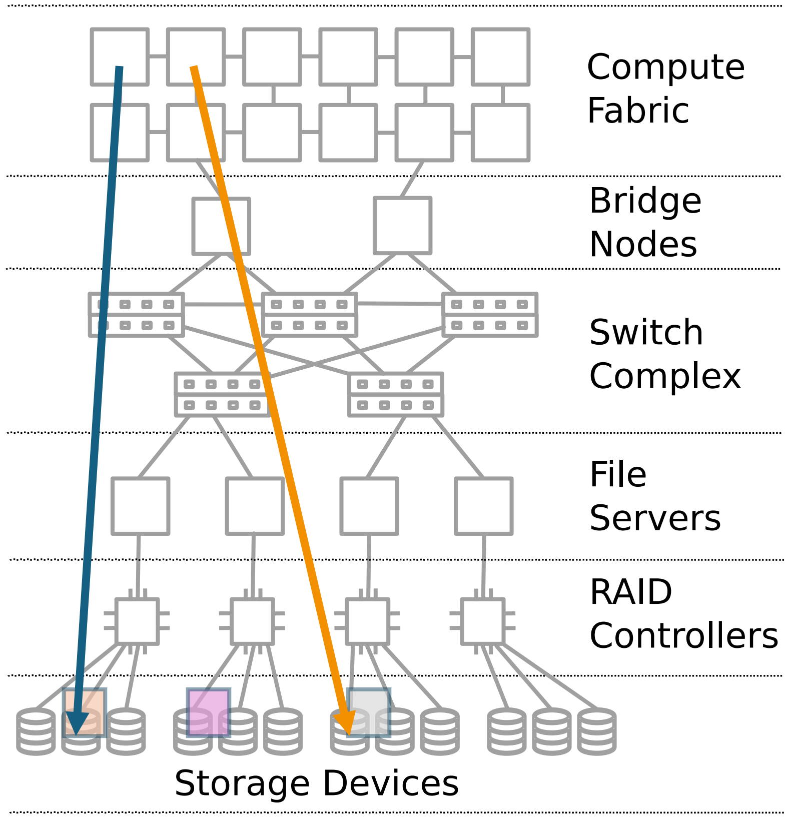


Why is concurrent access hard?
Example 3: writing separate files

- Consider a case in which two ranks write data simultaneously to different files.
- There is no possibility of I/O conflict. That should be good, right?

Rank 0: open("a"); write(256 KiB);

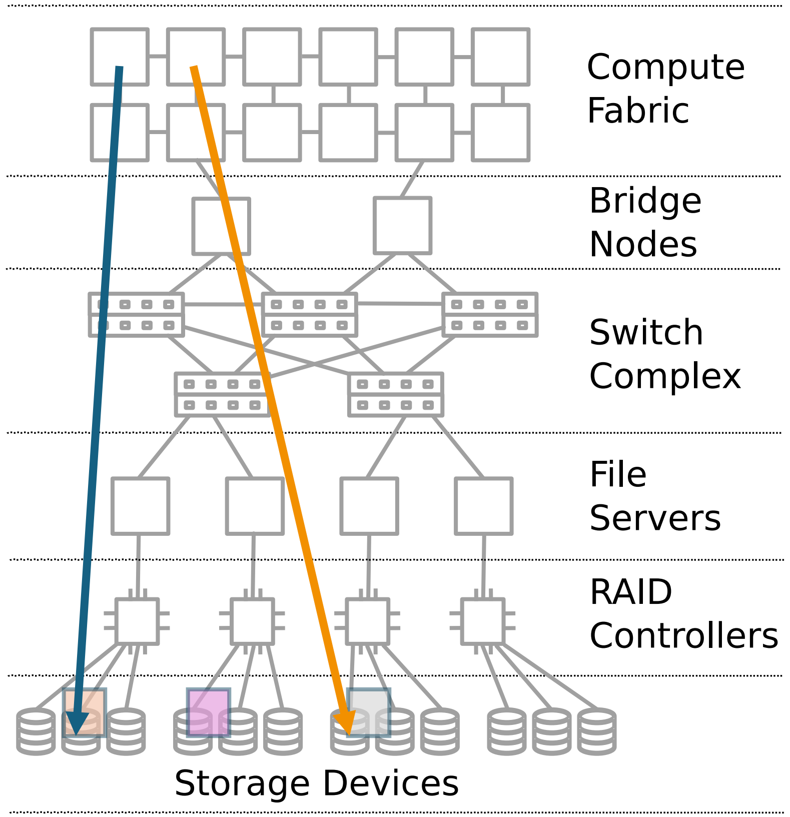
Rank 1: open("b"); write(256 KiB);



Why is concurrent access hard?
Example 3: writing separate files

- Consider a case in which two ranks write data simultaneously to different files.
- There is no possibility of I/O conflict. That should be good, right?
- ✓ The writes are indeed issued to independent servers and storage devices. This probably works well at small scale.

Rank 0: open("a"); write(256 KiB);
Rank 1: open("b"); write(256 KiB);

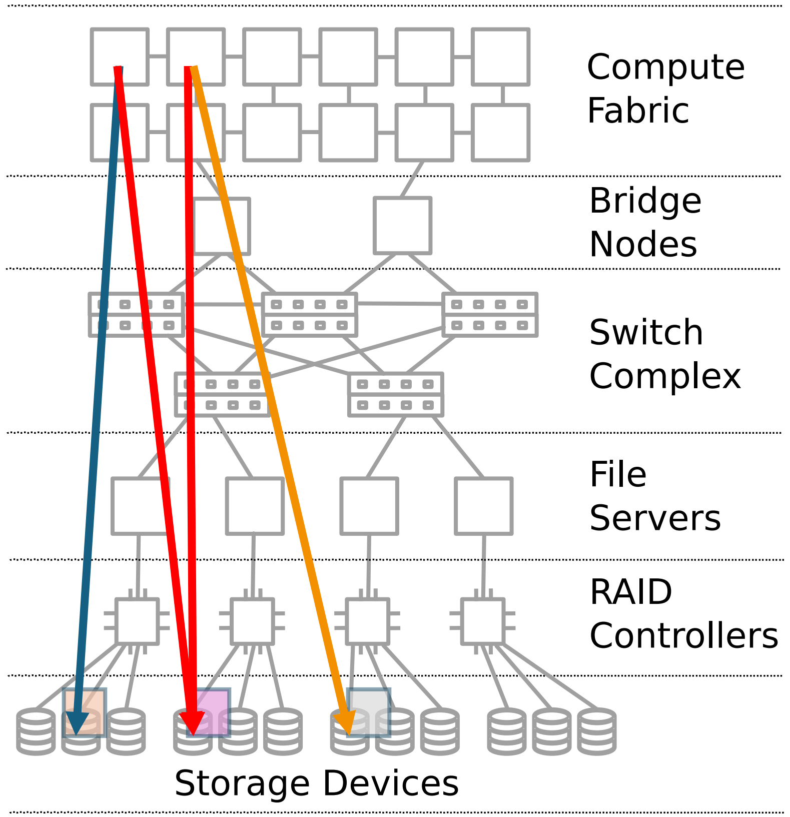


Why is concurrent access hard?
Example 3: writing separate files

- Consider a case in which two ranks write data simultaneously to different files.
- There is no possibility of I/O conflict. That should be good, right?
- ✓ The writes are indeed issued to independent servers and storage devices. This probably works well at small scale.
- ✗ Directories are hierarchical, though, so processes will conflict at open() time to coordinate access to the parent directory. This problem gets progressively worse at scale.
- ✗ It also makes the file system spend more time managing metadata (names, permissions, attributes) than performing productive data transfer.
- ✗ It also (eventually) places more burden on the user to manage files.

Rank 0: open("a"); write(256 KiB);

Rank 1: open("b"); write(256 KiB);



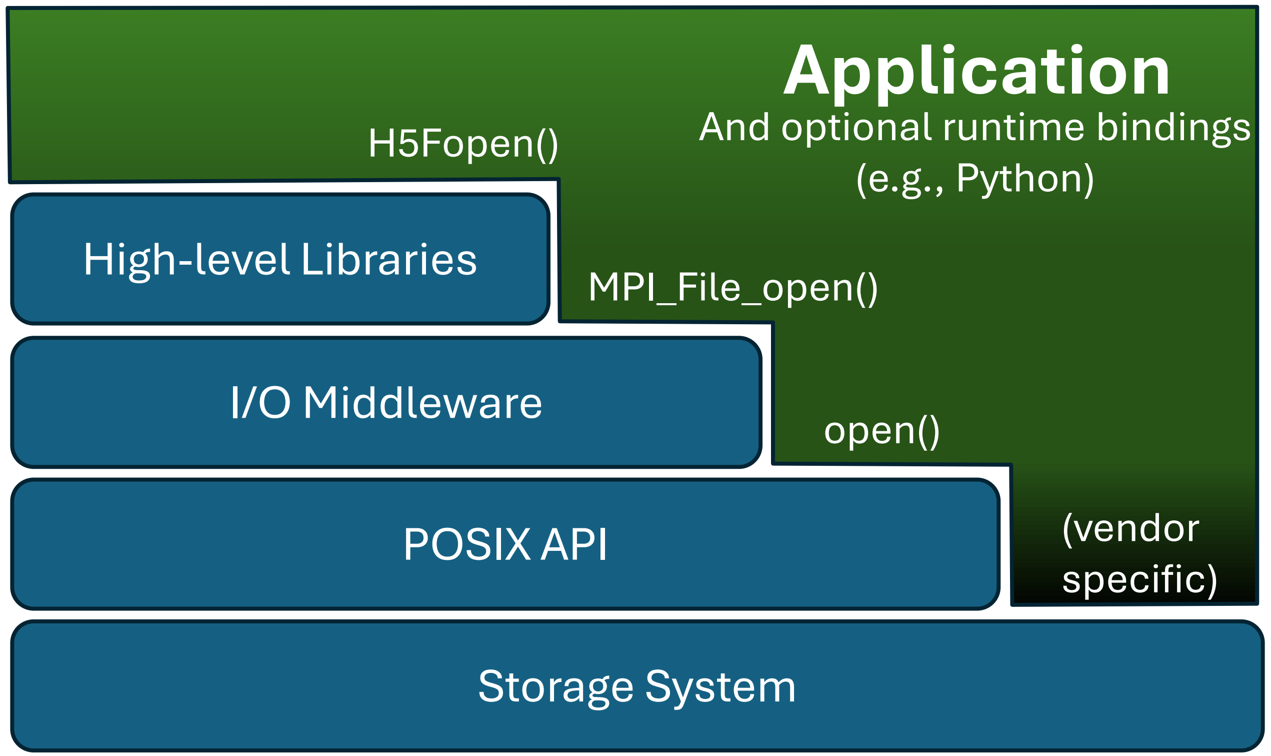
Why is concurrent access hard?
The common theme

There is a common underlying problem in each of the preceding examples:

Fundamentally, the sequential POSIX API cannot describe a complete, coordinated parallel access pattern to the file system.

Because each process issues I/O operations independently, the storage system must service each one in isolation (even if there are thousands or even millions in flight). There isn't much opportunity to aggregate or structure the flow of data.

Help is on the way: Application-level I/O interface layers



- High level interfaces **translate between scientific data concepts (hierarchical, multi-dimensional, multi-variable, with rich metadata) and file system constructs (directories and opaque files)** so that you don't have to. Benefits include:
  - Performance portability
  - Expressive interfaces
  - Future proofing
  - Interoperability
- Don't worry if you have to use POSIX; we can help with that too!
- We will discuss all of these layers, and how to use them effectively, in depth today.

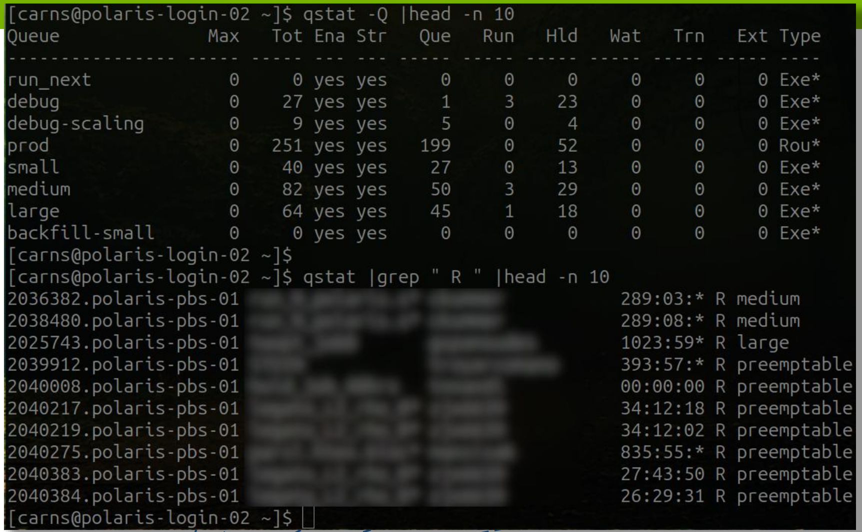
Hands on exercises:
https://github.com/radix-io/hands-on

One final warning: even if you do everything right, … performance can still be surprising

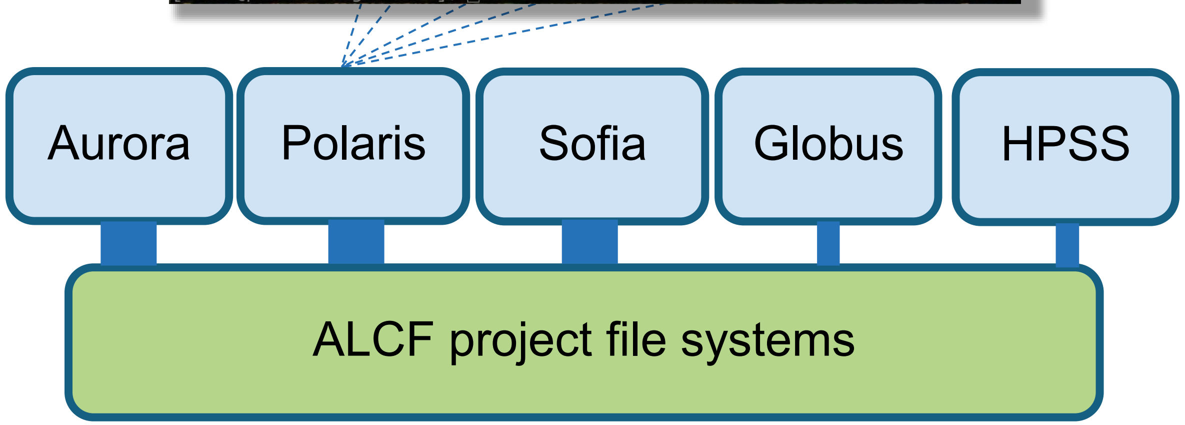
- Thousands of storage devices will *never* perform perfectly at the same time.
- You are sharing storage with many other users across multiple HPC systems.
- You are also sharing storage with remote transfers, tape archives, and more.

Compute nodes belong exclusively to you during a job allocation, but the storage system does not. ***Storage performance varies in ways that are fundamentally different from compute performance.***

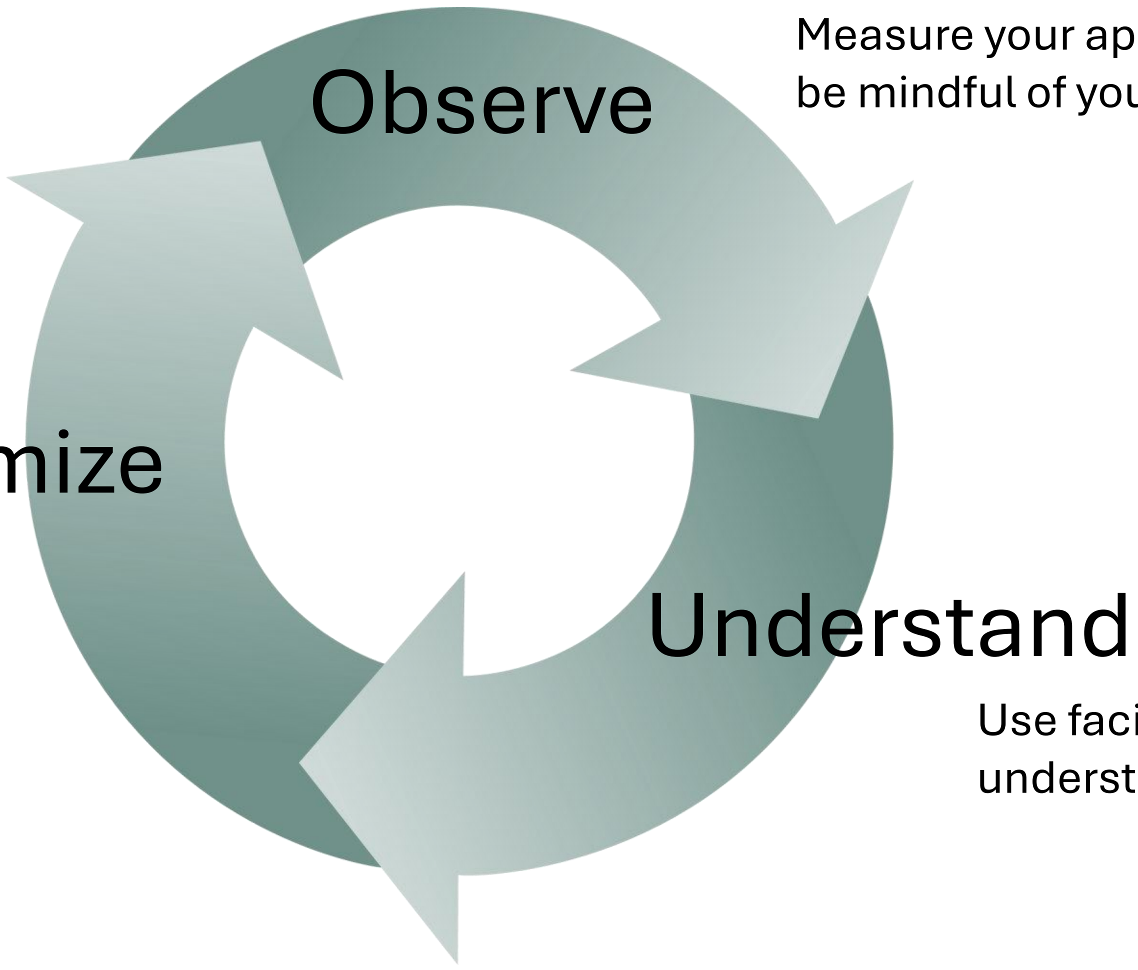
Best practice: take multiple samples of I/O performance to fully understand subtle changes. Want a demonstration of what variance you can expect? See the hands-on/variance exercise.



```
[carns@polaris-login-02 ~]$ qstat -Q |head -n 10
Queue              Max   Tot Ena Str   Que   Run   Hld   Wat   Trn   Ext Type
---------------- ----- ----- --- --- ----- ----- ----- ----- ----- ----- ----
run_next             0     0 yes yes     0     0     0     0     0     0 Exe*
debug                0    27 yes yes     1     3    23     0     0     0 Exe*
debug-scaling        0     9 yes yes     5     0     4     0     0     0 Exe*
prod                 0   251 yes yes   199     0    52     0     0     0 Rou*
small                0    40 yes yes    27     0    13     0     0     0 Exe*
medium               0    82 yes yes    50     3    29     0     0     0 Exe*
large                0    64 yes yes    45     1    18     0     0     0 Exe*
backfill-small       0     0 yes yes     0     0     0     0     0     0 Exe*
[carns@polaris-login-02 ~]$
[carns@polaris-login-02 ~]$ qstat |grep " R " |head -n 10
2036382.polaris-pbs-01 [redacted]             289:03:* R medium
2038480.polaris-pbs-01 [redacted]             289:08:* R medium
2025743.polaris-pbs-01 [redacted]             1023:59* R large
2039912.polaris-pbs-01 [redacted]             393:57:* R preemptable
2040008.polaris-pbs-01 [redacted]             00:00:00 R preemptable
2040217.polaris-pbs-01 [redacted]             34:12:18 R preemptable
2040219.polaris-pbs-01 [redacted]             34:12:02 R preemptable
2040275.polaris-pbs-01 [redacted]             835:55:* R preemptable
2040383.polaris-pbs-01 [redacted]             27:43:50 R preemptable
2040384.polaris-pbs-01 [redacted]             26:29:31 R preemptable
[carns@polaris-login-02 ~]$
```



Parting thoughts on the Principles of HPC I/O: HPC I/O optimization is an ongoing process



Measure your application and be mindful of your storage needs.

Use facility resources to understand what's possible.

Apply techniques that you've learned today to improve performance and usability.

HPC storage systems

We mentioned earlier that each HPC system has its own unique storage architecture.

What kinds of high-performance storage systems can you find at the Office of Science compute facilities?

Let's take a quick tour!

https://docs.olcf.ornl.gov/data/index.html#orion-lustre-hpe-clusterstor-filesystem
https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#nvme

HPC storage systems

Oak Ridge (OLCF) Lawrence Berkeley (NERSC) Argonne (ALCF)

Facility documentation is your friend! Some of the systems that we discuss will change over time.

- Orion parallel file system (Lustre)
  - Multiple storage tiers are transparently combined:
    - Metadata tier: ~10 PiB capacity
    - Performance tier: ~11 PiB capacity, ~10 TiB/s performance
    - Capacity tier: ~680 PiB capacity, ~5 TiB/s performance
  - File layouts: Orion uses a sophisticated progressive file layout. Unlike some Lustre file systems, it is **not** recommended for you to manually change file stripe settings on Orion!

- Node-local SSDs on Frontier
  - NVMe drives provide 5 GiB/s read and 2 GiB/s write per node.
  - Excellent for latency, but they are not shared between nodes or accessible outside of your job.
  - You must explicitly request them in your job script.
  - You must explicitly stage data on and off of them.

https://docs.nersc.gov/filesystems/
https://docs.nersc.gov/filesystems/perlmutter-scratch/

HPC storage systems

Oak Ridge (OLCF) **Lawrence Berkeley (NERSC)** Argonne (ALCF)

- Perlmutter Scratch parallel file system (Lustre)
  - Unlike the Orion file system at OLCF, the Perlmutter scratch file system consists of a single all-flash (high-performance) tier of storage.
  - 35 PiB capacity, 5 TiB/s throughput
  - File layouts: 1 server per file by default
    - This layout is great for small files accessed by single processes, but you'll want to increase the striping if you need to access large files in parallel.

https://docs.alcf.anl.gov/data-management/filesystem-and-storage/
https://docs.alcf.anl.gov/aurora/data-management/daos/daos-overview/

HPC storage systems

Oak Ridge (OLCF) Lawrence Berkeley (NERSC) **Argonne (ALCF)**

- Flare and Eagle parallel file systems (Lustre)
  - Similar caveats as Polaris Scratch: you may need to tune the Lustre settings for your use case; the default settings are only optimal for small, non-shared files.
- Polaris also has local NVMe drives
  - Similar caveats as Frontier local disks: you need to think about how to stage data.
- Aurora also has a DAOS storage system
  - 230 PiB capacity, 30 TiB/s throughput.
  - Under the covers it is a distributed object store.
    - You can access it like a conventional file system or use libraries like HDF5 or PyTorch that have been adapted to access DAOS natively.
  - We'll learn more about how to use it this afternoon! DAOS is remarkably fast but has a little bit more of a learning curve.

Thank you!

Any questions before we move on to the next presentation?

ARGONNE
ATPESC2025
EXTREME-SCALE COMPUTING

ARGONNE TRAINING PROGRAM ON EXTREME-SCALE COMPUTING

Produced by Argonne National Laboratory, a U.S. Department of Energy Laboratory managed by UChicagoArgonne, LLC under contract DE-AC02-06CH11357.

Special thanks to the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF) for the use of their resources during the training event.