ARGONNE
ATPESC2023
EXTREME-SCALE COMPUTING

Principles of HPC I/O

August 10, 2023

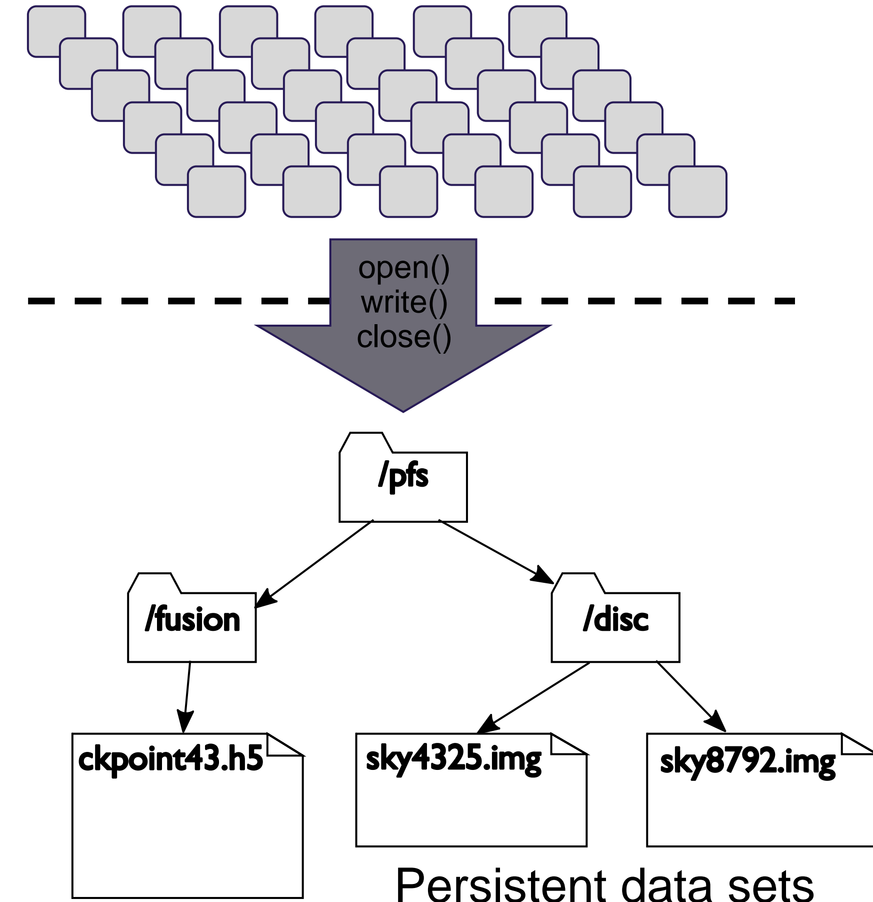
Phil Carns
Mathematics and Computer Science Division
Argonne National Laboratory

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.

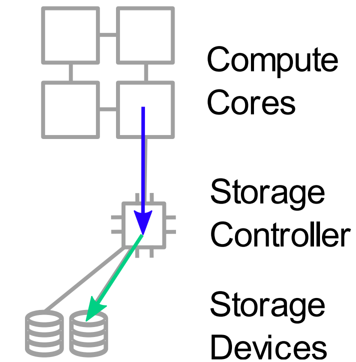
Today's lectures are really all about the proper care and feeding of exotic parallel file systems.



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

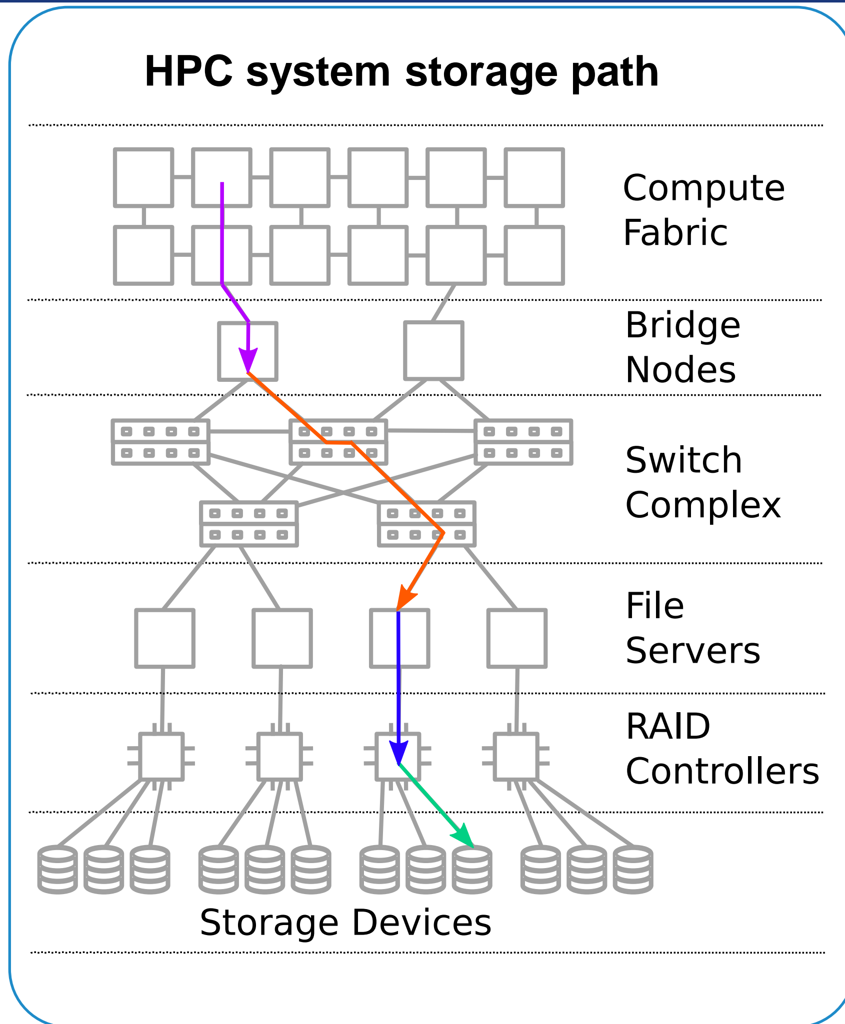
A look under the hood

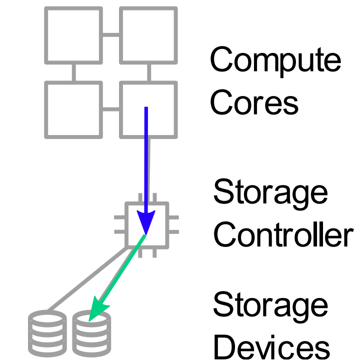
Workstation (laptop) storage path



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

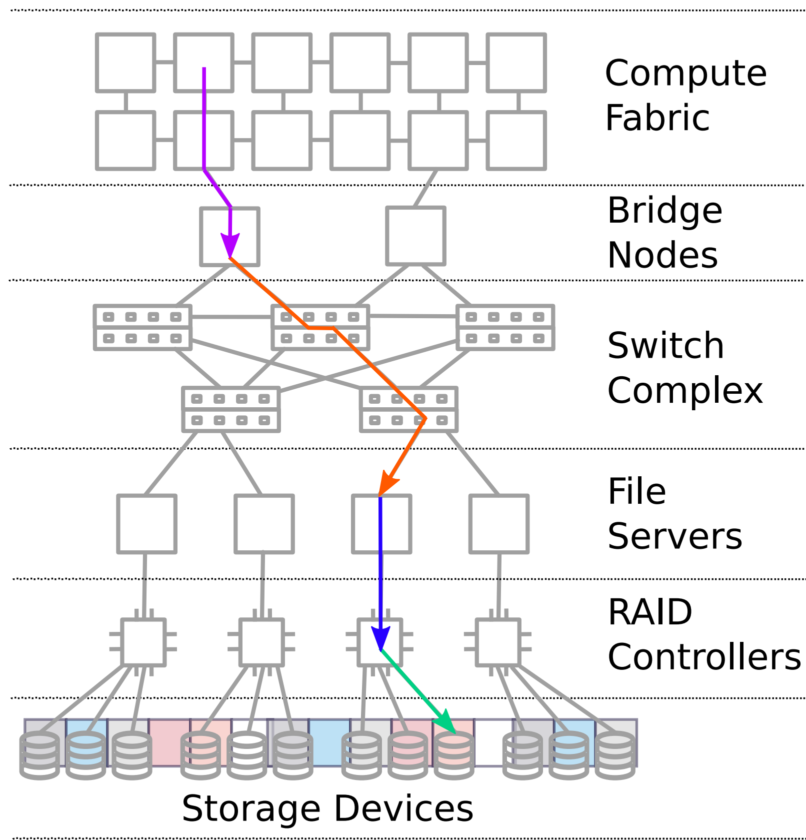
A look under the hood





- 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

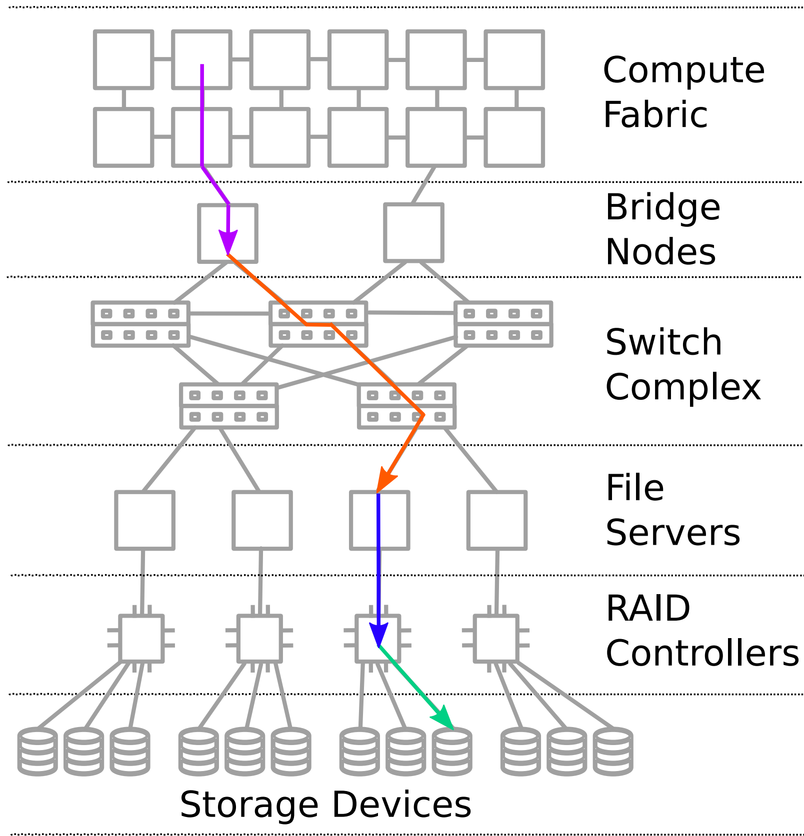
Striping / Layout



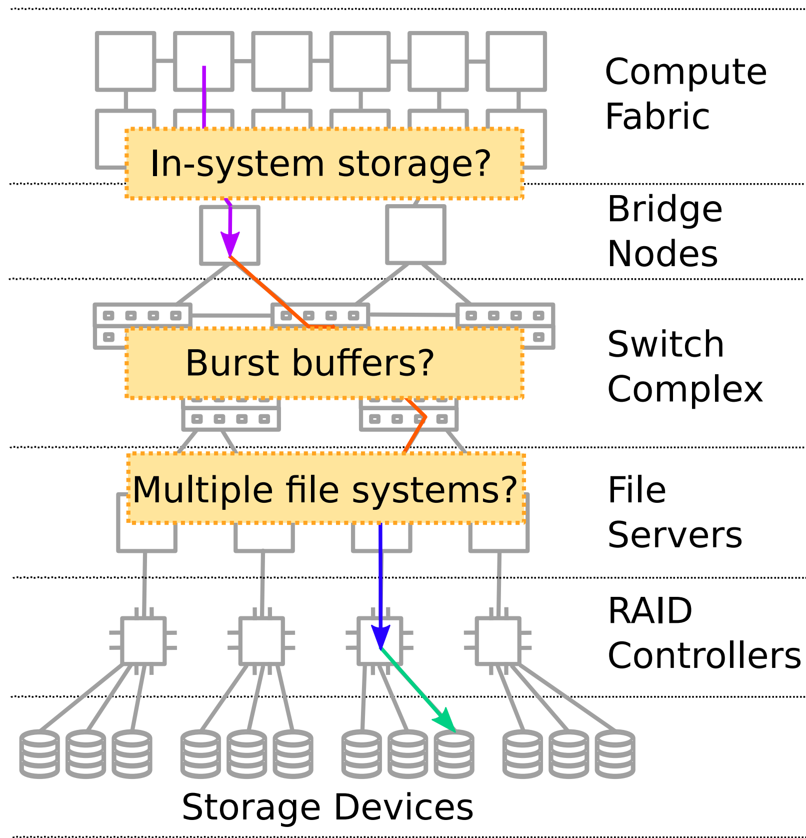
- Large files are not generally stored on a single storage device.
- They are distributed across multiple servers (and then each server further distributes across storage devices).
- This is referred to as **data layout** or **striping**.
- Different file systems use different striping strategies.
- 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?



Is that all?



Each HPC storage system is unique.
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

We'll learn more about some of these options in the next presentation.

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
- Normal file I/O bindings for different languages

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.

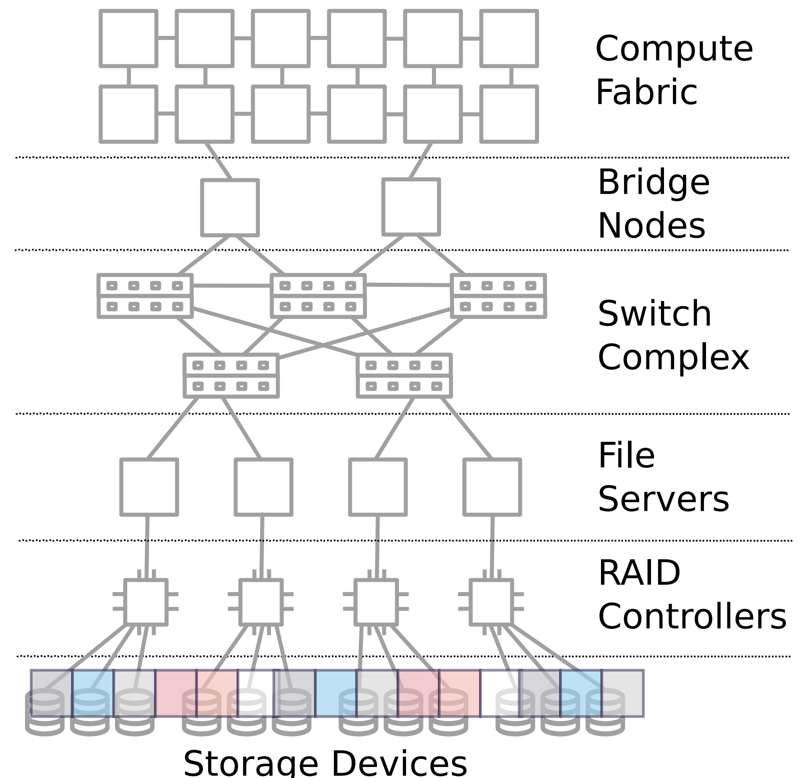
```
fd = open("foo.dat", O_CREAT|O_WRONLY, 0600);
if(fd < 0) {
    perror("open");
    return(-1);
}

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.
- File descriptors are stateful at each process.
- File position is implied.
- Files are unstructured.

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, but to different parts of the file.
- In this example we have a big gap (32 MiB) between them. Assume we are writing reasonably large chunks to optimize bandwidth vs. latency.

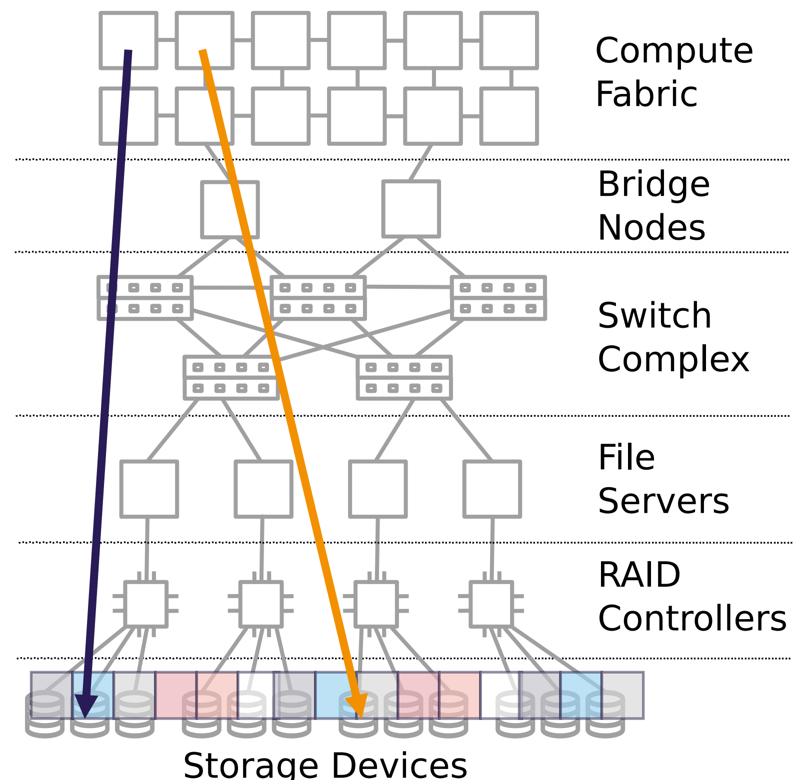
Rank 0: lseek(0); write(256 KiB);
Rank 1: lseek(32 MiB); write(256 KiB);



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, but to different parts of the file.
- In this example we have a big gap (32 MiB) between them. Assume we are writing reasonably large chunks to optimize bandwidth vs. latency.
- 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 more access density...

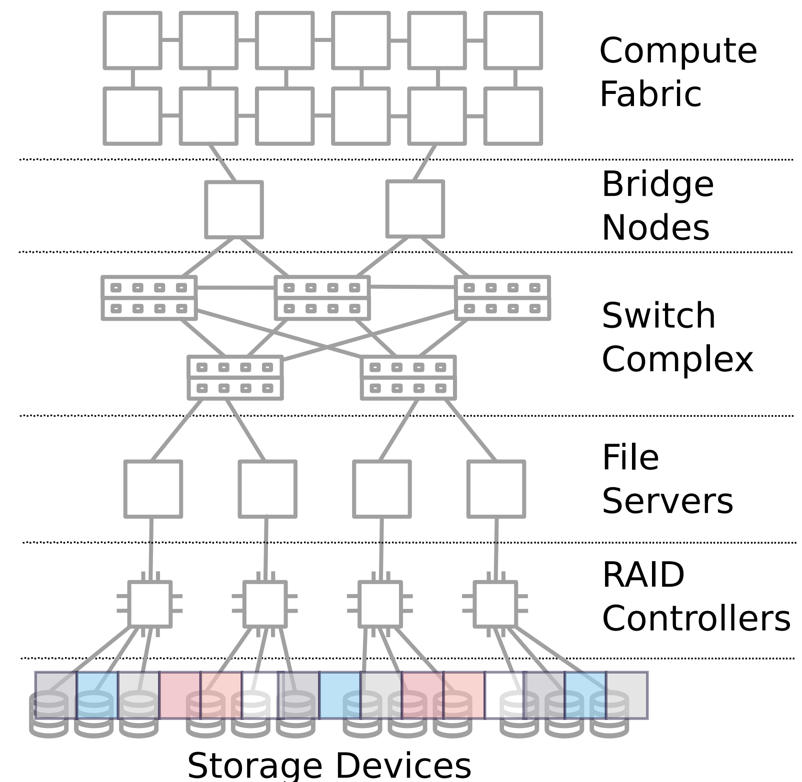
Rank 0: lseek(0); write(256 KiB);
Rank 1: lseek(32 MiB); write(256 KiB);



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

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

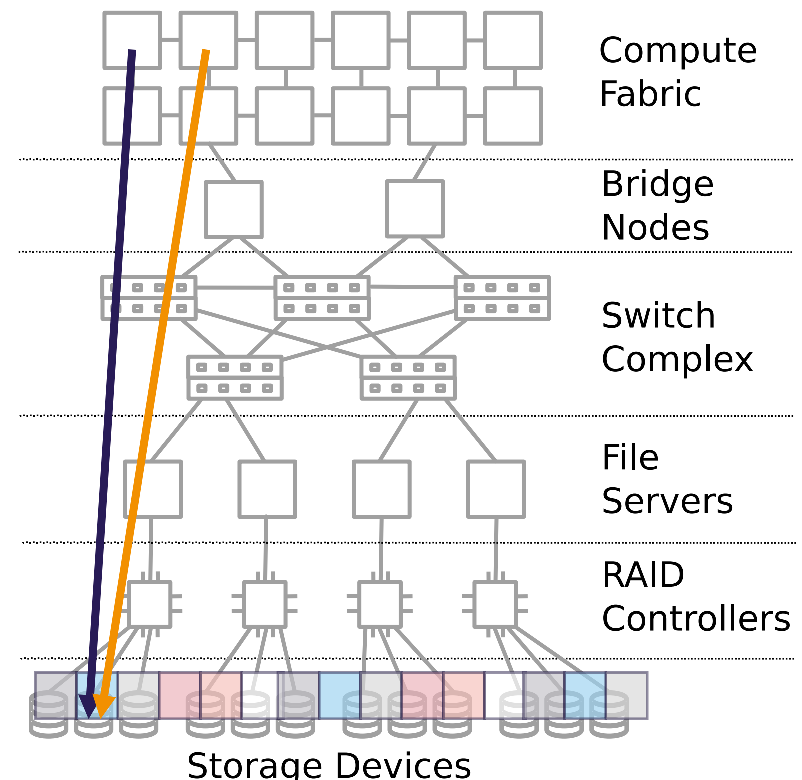
Rank 0: lseek(0); write(256 KiB);
Rank 1: lseek(256 KiB); write(256 KiB);



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

- Consider a case in which two ranks write data simultaneously, but to different parts of the file.
- In this example 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 layout locality.
- Counterintuitively, this **almost certainly causes conflicting access**. The file system's caching and locking granularity is independent of access size.
- Uncoordinated adjacent access can cause "false sharing" and serialize I/O operations that should have proceeded in parallel → poor performance.

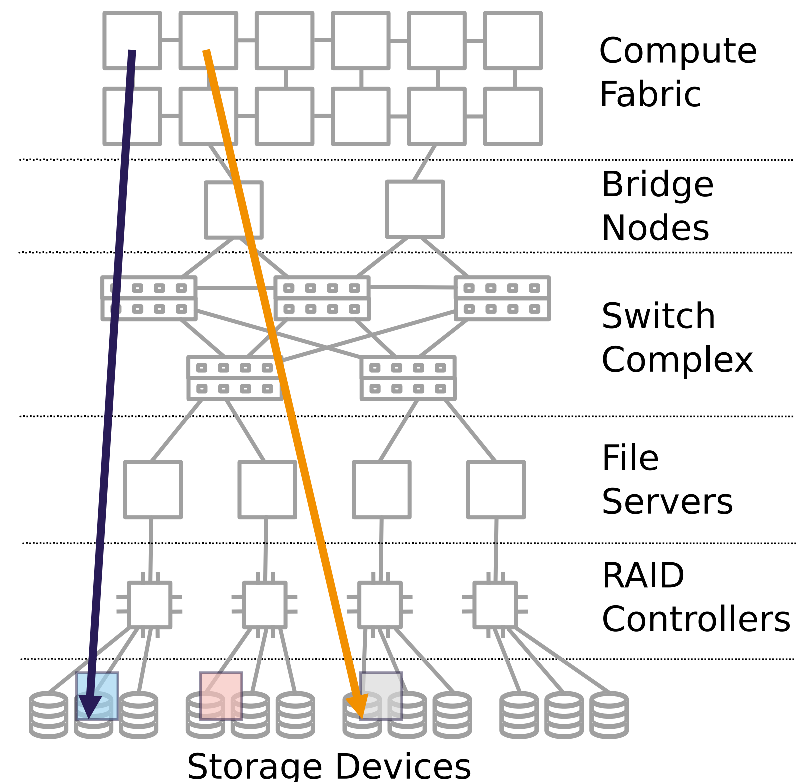
Rank 0: lseek(0); write(256 KiB);
Rank 1: lseek(256 KiB); write(256 KiB);



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?

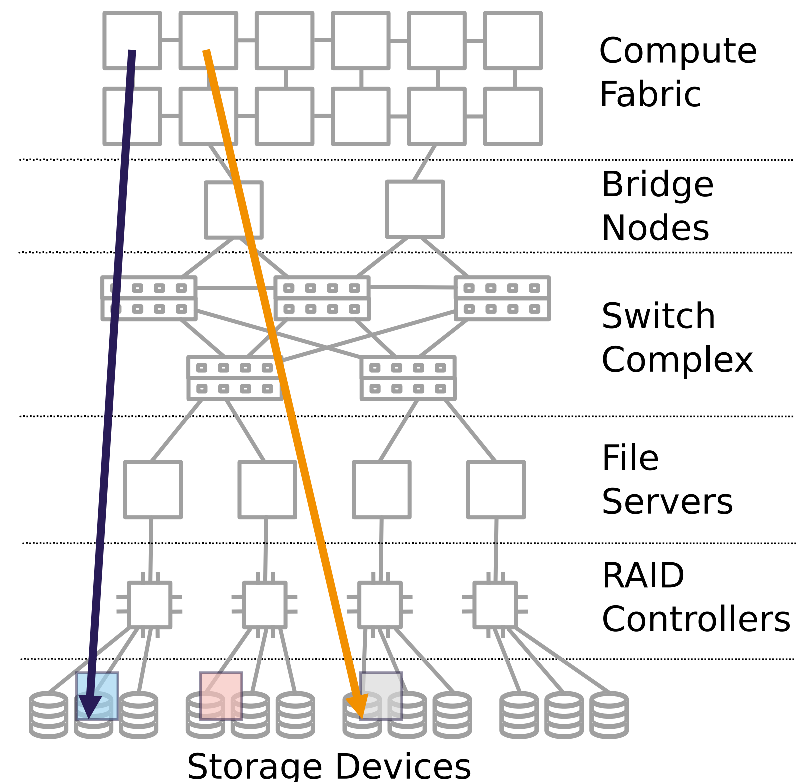
Rank 0: open(“a”); write(256 KiB);
Rank 1: open(“b”); write(256 KiB);



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.

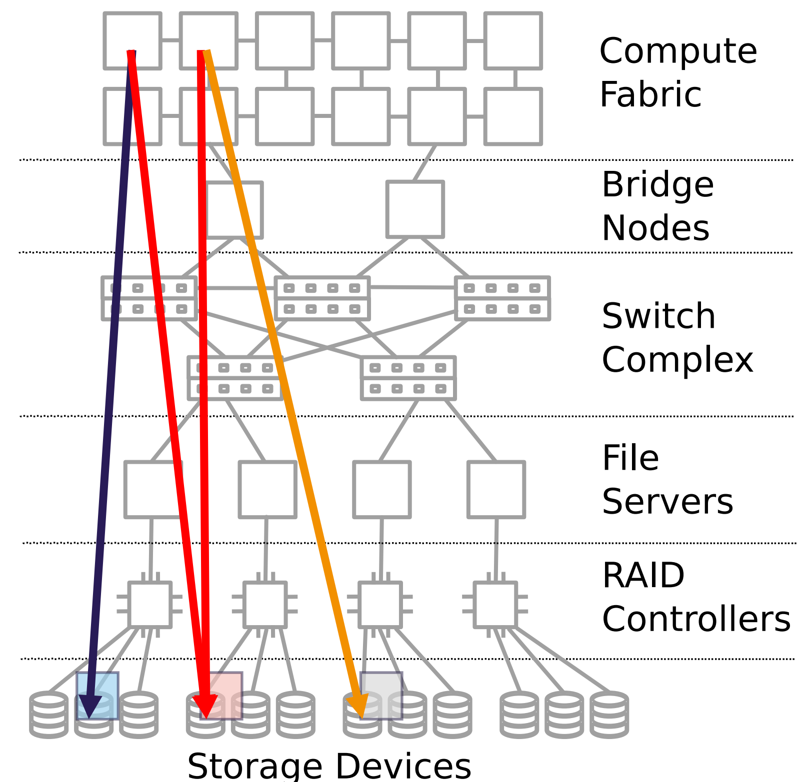
Rank 0: open(“a”); write(256 KiB);
Rank 1: open(“b”); write(256 KiB);



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

- Our speakers this afternoon will teach you about a variety of APIs designed specifically to facilitate parallel access to scientific data.
- All of them implement ***portable*** optimizations that shape traffic for parallel file systems.
- If you have no choice but to use POSIX APIs: don't worry, we will also share techniques to help you extract performance there.
- A big feature of today's material is also how to measure, characterize, and understand I/O behavior so that you can continually improve.

High-level I/O libraries: an early sales pitch

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

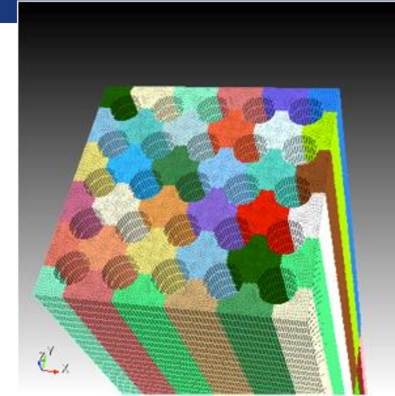
Applications use advanced data models according to their scientific objectives:

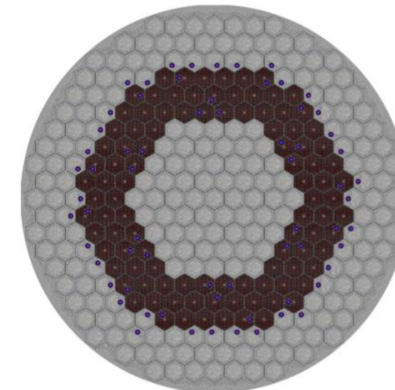
- The data itself: Multidimensional typed arrays, images composed of scan lines, etc.
- Descriptions of data (metadata): Headers, attributes, time stamps, etc.

In contrast, parallel file systems present a very simple data model:

- Tree-based hierarchy of containers
- Containers with streams of bytes (files)
- Containers listing other containers (directories)
- *As we saw in previous slides*: quirky performance properties

You could map between these two models yourself: "The frequency attribute is an 8-byte float in GHz, stored at offset 4096."





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.

High-level I/O libraries: an early sales pitch

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

Data libraries (like HDF5, PnetCDF, and ADIOS) help to bridge this gap between application data models and file system interfaces.

Why use a high-level data library?

Stay tuned for more information in the following sessions:

2:00 Parallel-NetCDF
2:45 HDF5

More expressive interfaces for scientific data

- e.g., multidimensional variables and their descriptions

Interoperability

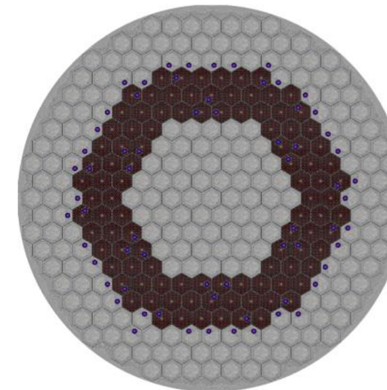
- e.g., enables collaborators to share data in self-describing, well-documented formats

Performance

- e.g., high level libraries implement platform-specific optimizations so that you don't have to

Future proofing

- e.g., interfaces and data formats that outlive specific storage technologies



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.

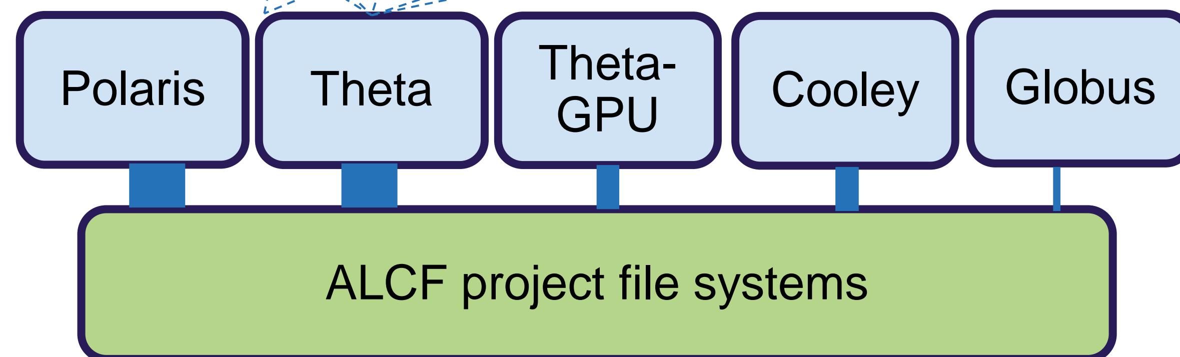
And finally, even if you do everything right … performance can still be surprising

- Thousands of hard drives 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 other data management tasks.

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.

```
[          ~]$ qstat |grep running
1139867          24:00:00  8192   running     MIR-48000-7BFF1-8192
1139871          24:00:00  8192   running     MIR-00000-33FF1-8192
1143326          12:00:00  2048   running     MIR-40C00-73FF1-2048
1151809          12:00:00  4096   running     MIR-40000-737F1-4096
1153083          24:00:00  16384  running     MIR-04000-77FF1-16384
1178836          12:00:00  512    running     MIR-408C0-73BF1-512
1178840          12:00:00  512    running     MIR-40880-73BB1-512
1179437          12:00:00  512    running     MIR-40840-73B71-512
1179755          02:00:00  4096   running     MIR-08000-3B7F1-4096
1179810          05:45:00  2048   running     MIR-08C00-3BFF1-2048
```



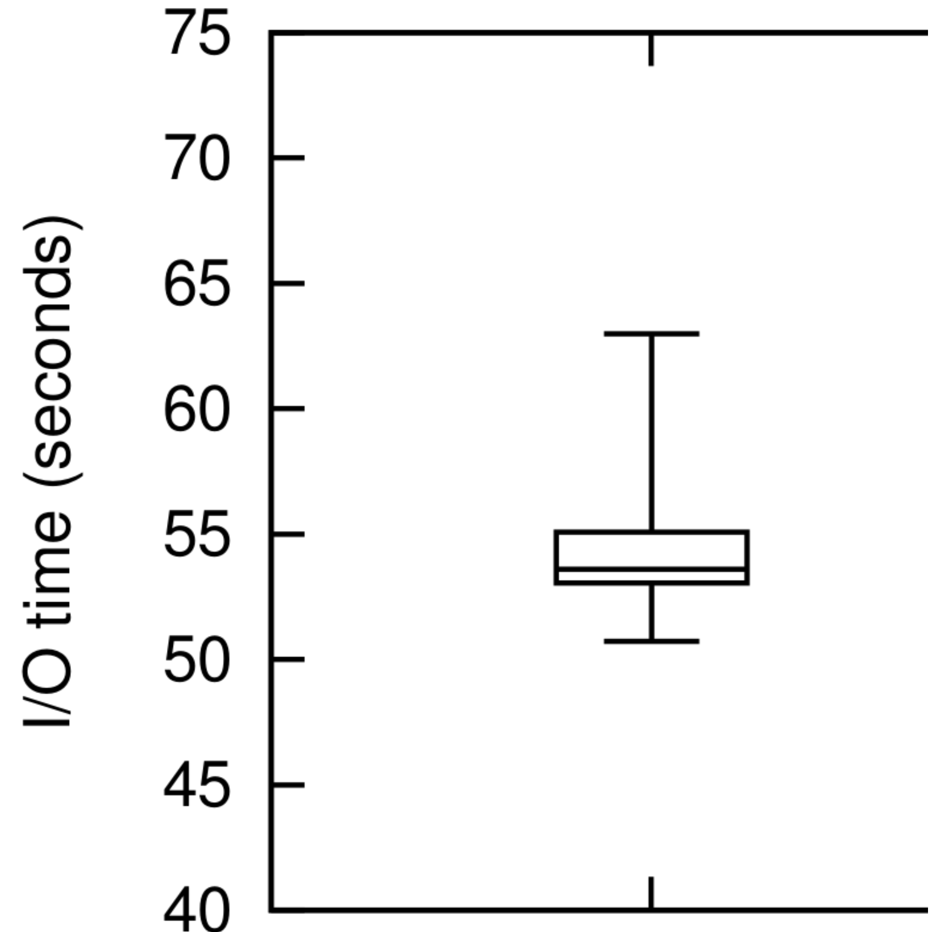
How to account for variability

Take multiple samples when measuring I/O performance.

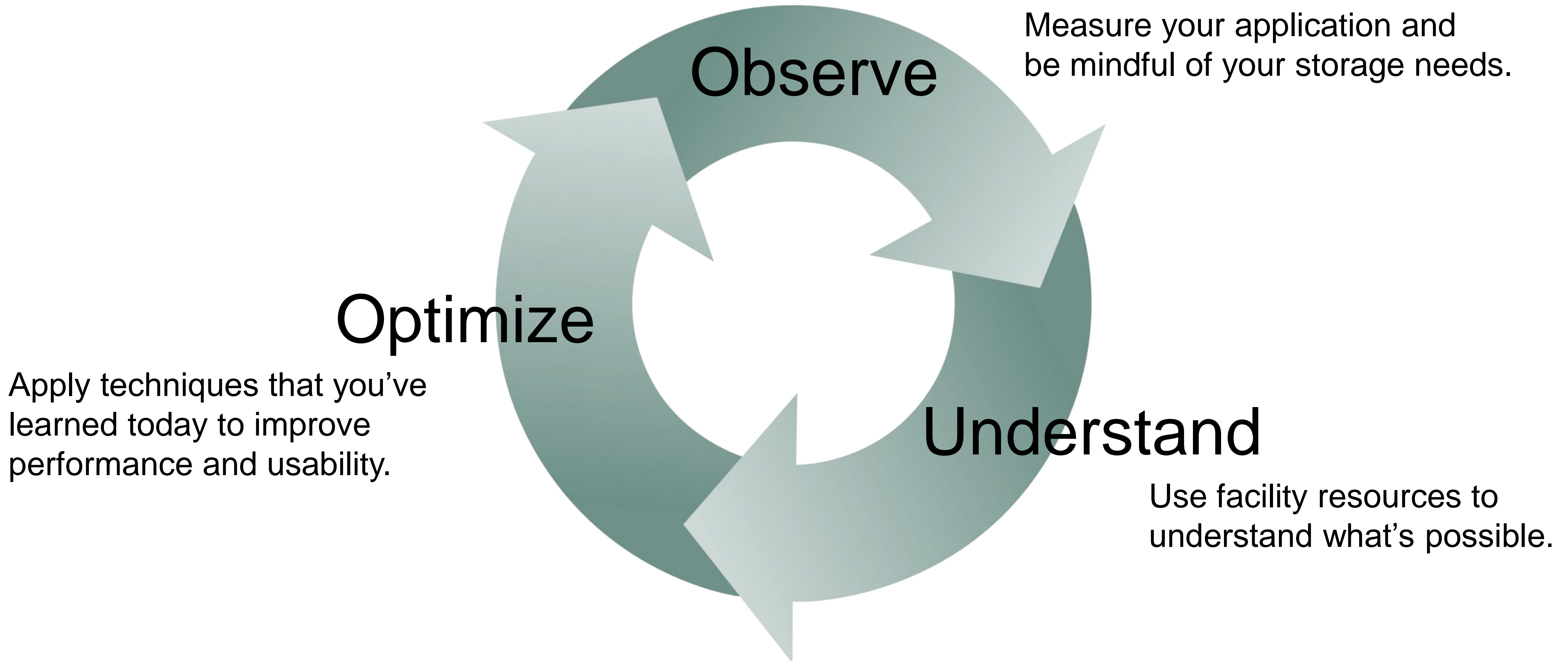
This figure shows 15 samples of I/O time from a 6,000 process benchmark on the (now retired) Edison system.

How do you assess if a change in your application helped or hurt performance under these conditions?

We will have a hands-on exercise later in the day that you can use to investigate this phenomenon first hand.



Parting message: I/O optimization is an ongoing process



Thank you!