ARGONNE ATPESC 10 EXTREME-SCALE COMPUTING

Principles of HPC I/O

ATPESC 2022

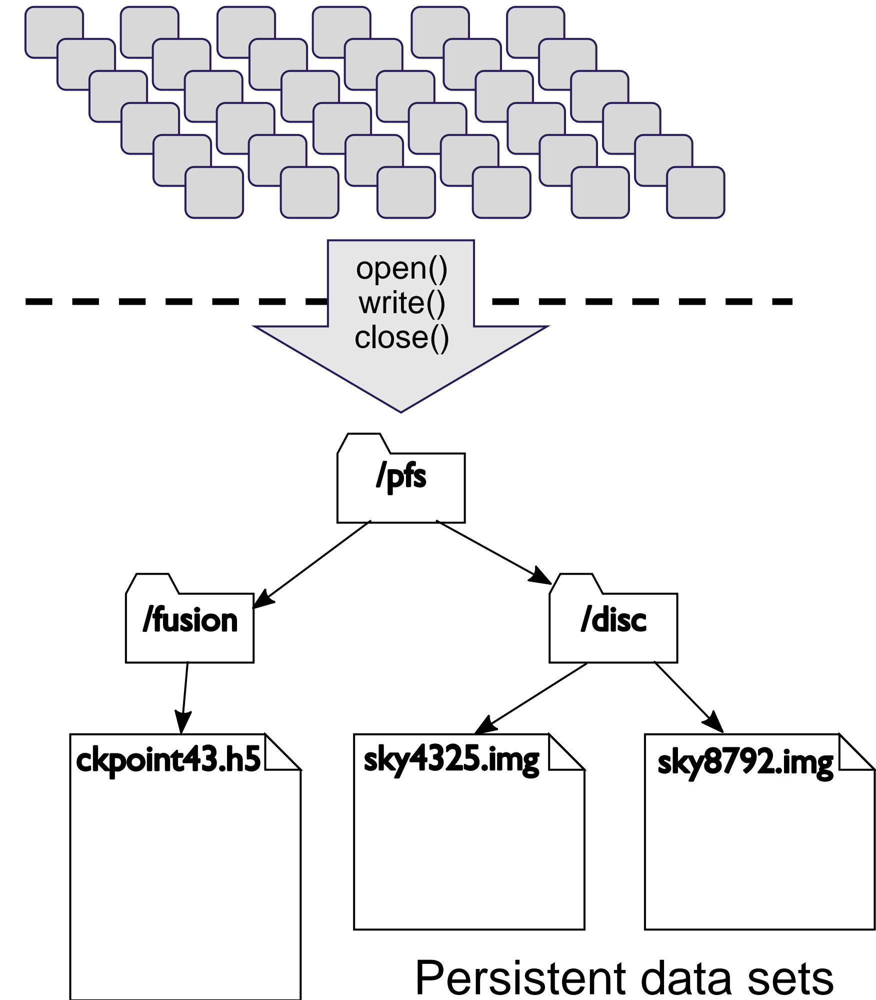
August 5, 2022

Phil Carns
Mathematics and Computer Science Division
Argonne National Laboratory

ECP 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.**
  - Parallel file systems can rapidly store and access enormous volumes of data.
  - They carefully orchestrate data movement between applications, system software, and storage hardware.
  - *This is an important job! Valuable CPU time is wasted if an application spends too long waiting for data.*
- Today's lectures are really all about the proper care and feeding of exotic parallel file systems.

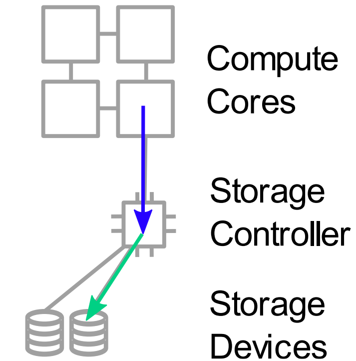


ECP EXASCALE COMPUTING PROJECT Argonne NATIONAL LABORATORY

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

A look under the hood

Workstation (laptop) storage path

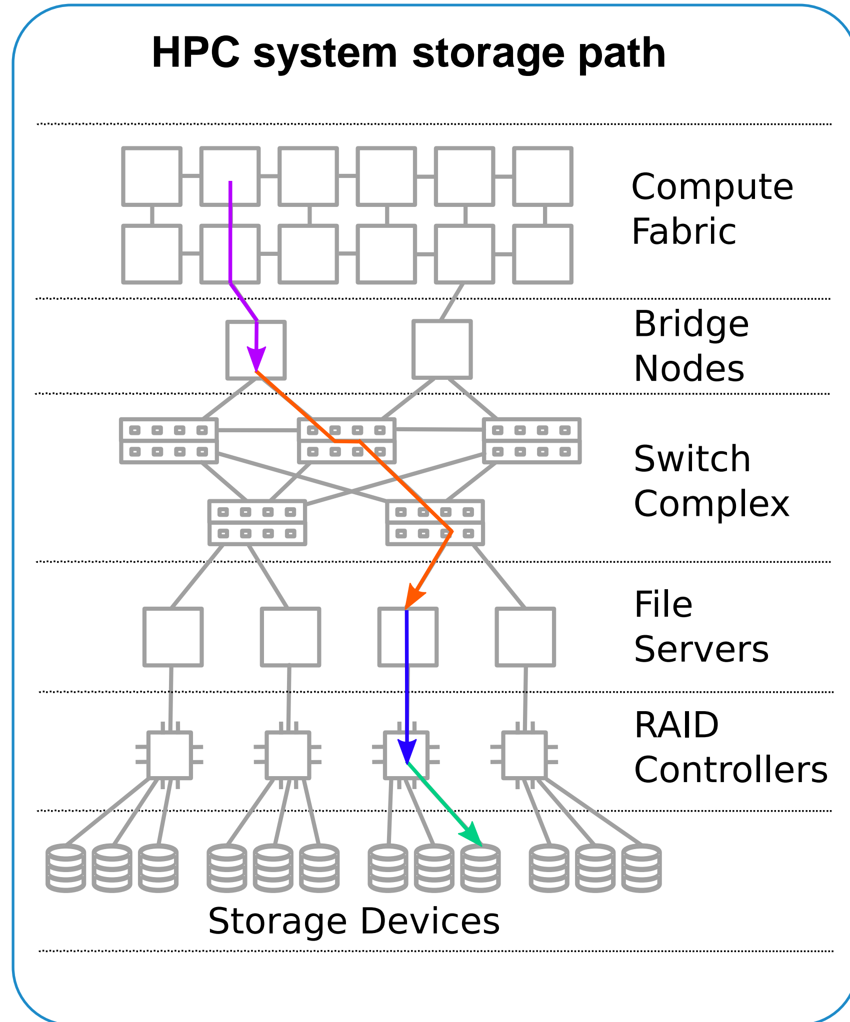


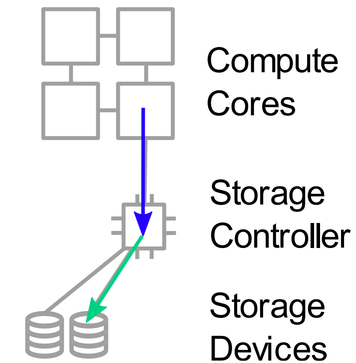
- A typical workstation only has a few storage devices (probably just one).
- The path between applications and storage is short.
- Properties:
  - Low latency
  - Low bandwidth

ECP EXASCALE COMPUTING PROJECT Argonne NATIONAL LABORATORY

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

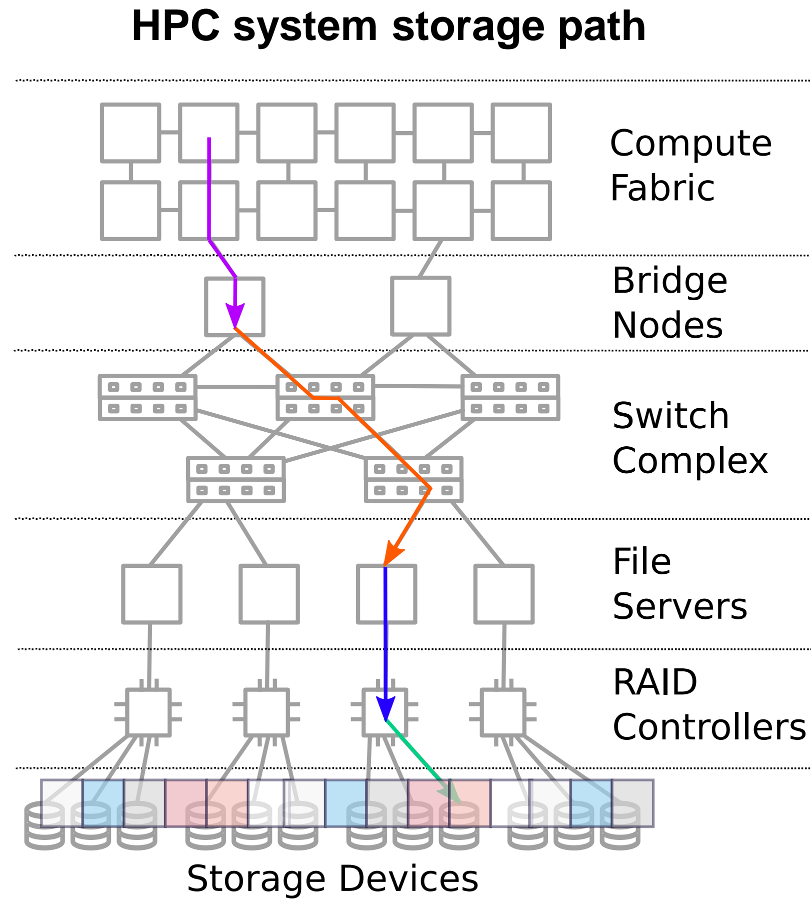
A look under the hood





- An HPC 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

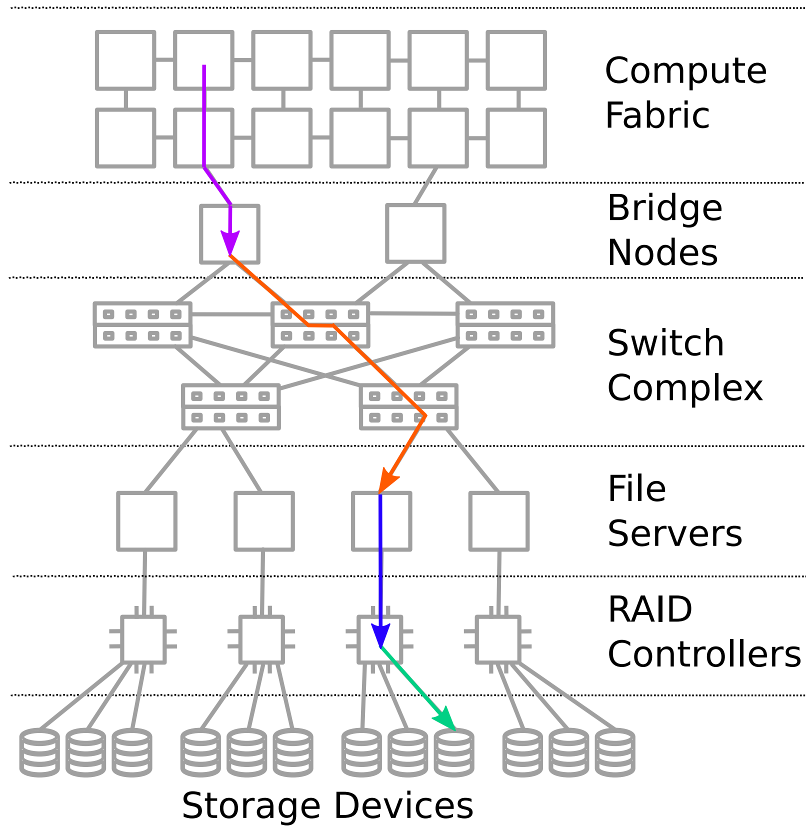


- Large files are not generally stored on a single storage device.
- They are distributed across multiple servers (and then each server further distributes across storage devices).
- This is referred to as **striping**.
- Different file systems use different strategies for striping data.
- Sometimes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

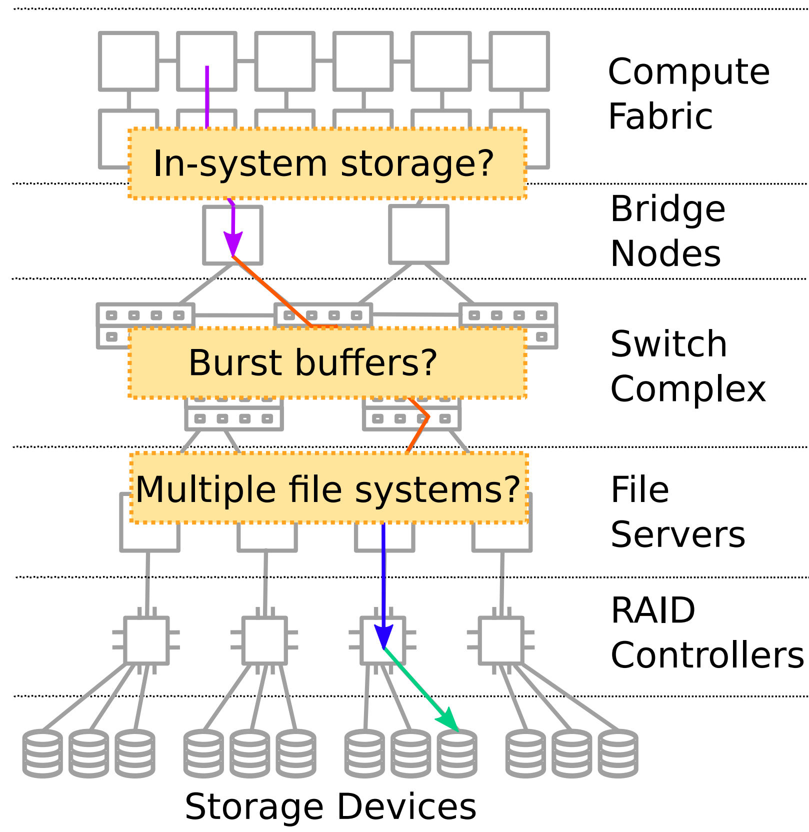
Example of a single logical file striped across all available servers and storage devices

ECP EXASCALE COMPUTING PROJECT Argonne NATIONAL LABORATORY

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. As the day goes on, we will teach tools and techniques to tame this complexity. The key takeaway for now is to understand why HPC systems use specialized storage software.

ECP EXASCALE COMPUTING PROJECT Argonne NATIONAL LABORATORY

Presenting storage to HPC applications

- A parallel file system can be accessed just like any other file system:
  - open() / close() / read() / write()
  - fopen() / fclose() / fprintf() for text data
  - (variations depending on your programming language)
- 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.

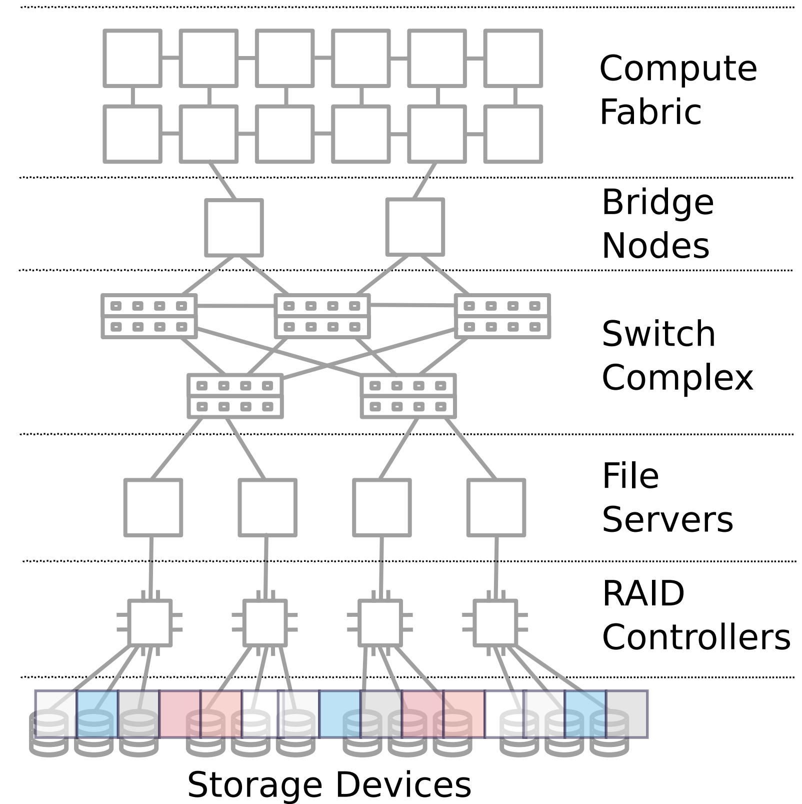
```
fd = open("foo.dat", O_CREAT|O_WRONLY, 0600);
if(fd < 0) {
    perror("open");
    return(-1);
}

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 concurrent access hard?
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 concurrent access hard?
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.
- No device contention. There is no control over path to those devices, though, and at some point in practice you'll also want denser access...

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

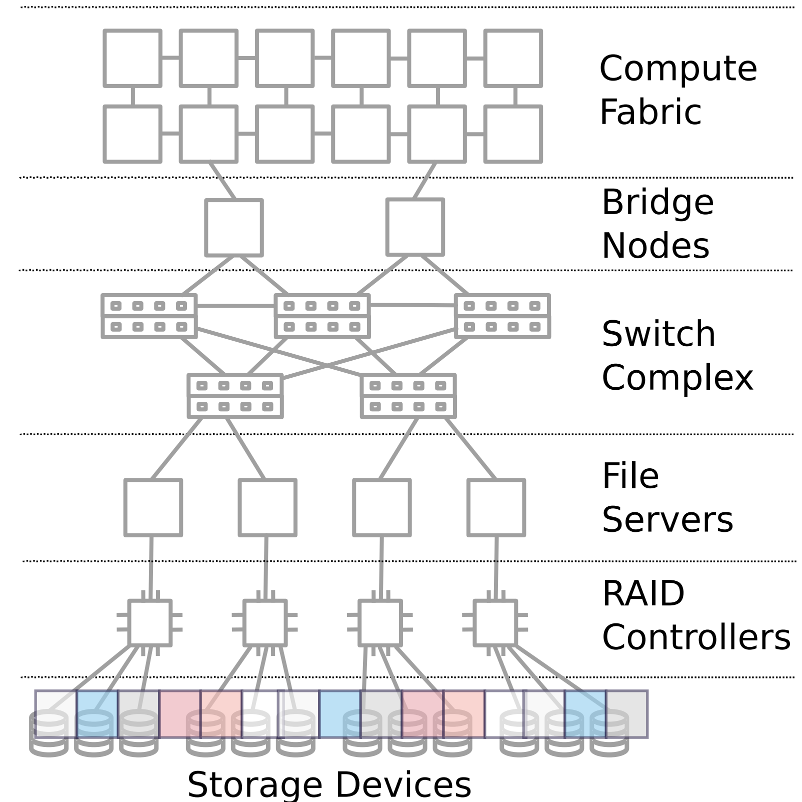
Compute Fabric
Bridge Nodes
Switch Complex
File Servers
RAID Controllers
Storage Devices

ECP EXASCALE COMPUTING PROJECT Argonne NATIONAL LABORATORY

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 the file.
- In this example they still don't overlap, but they write *adjacent* bytes.

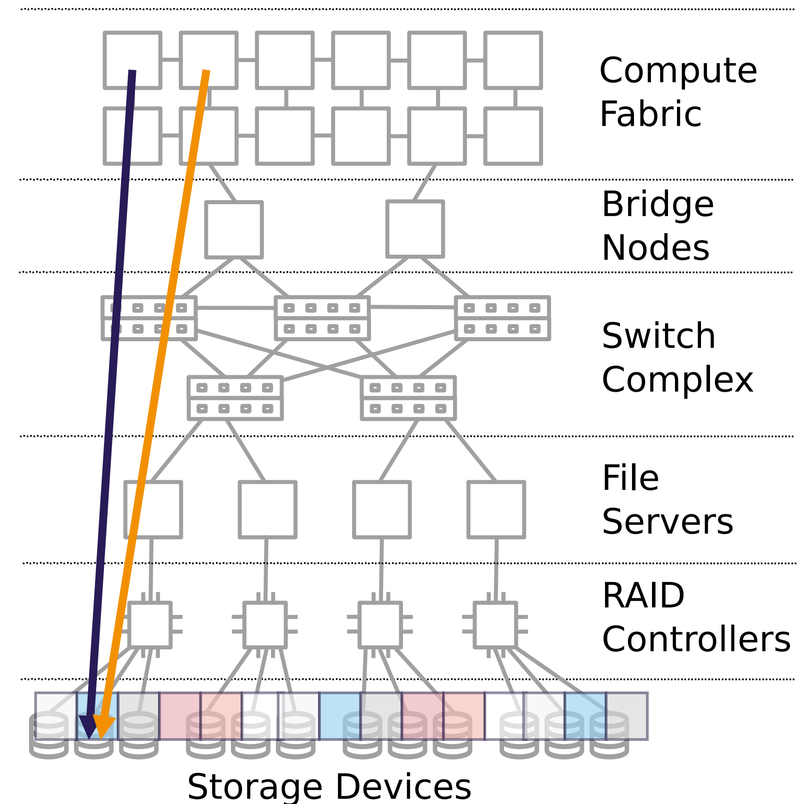
Rank 0: lseek(0); write(256 KiB);
Rank 1: lseek(256 KiB); write(256 KiB);



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 they still don't overlap, but they write adjacent bytes
- The writes might access the same server and storage device.
- But is there a conflict? Counterintuitively, probably so: the caching and locking granularity is independent of access size Uncoordinated adjacent access can cause "false sharing" and serialization.

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

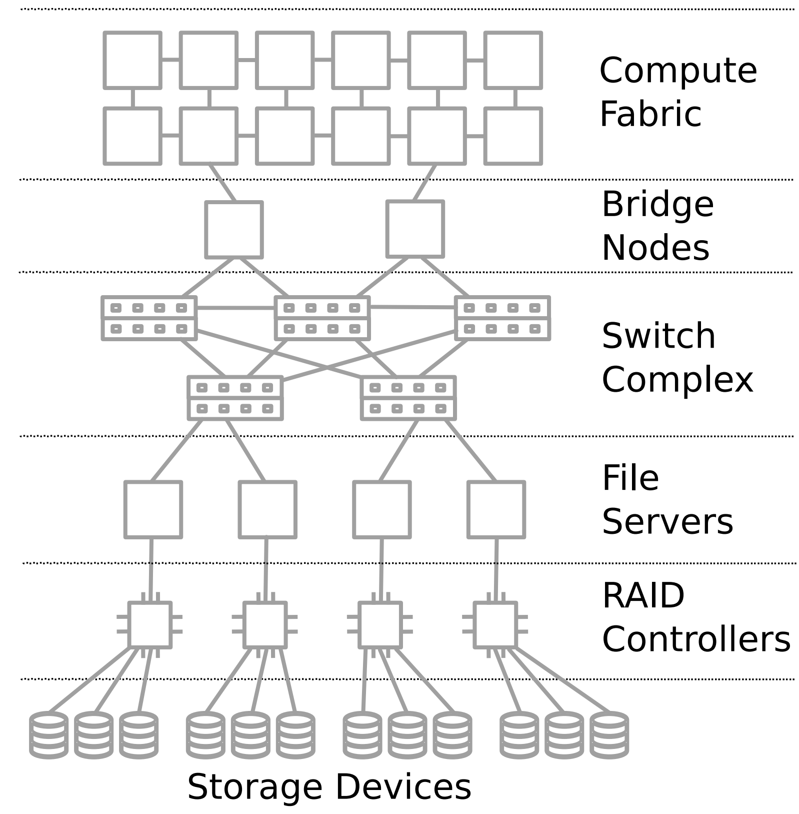


ECP EXASCALE COMPUTING PROJECT
Argonne NATIONAL LABORATORY

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



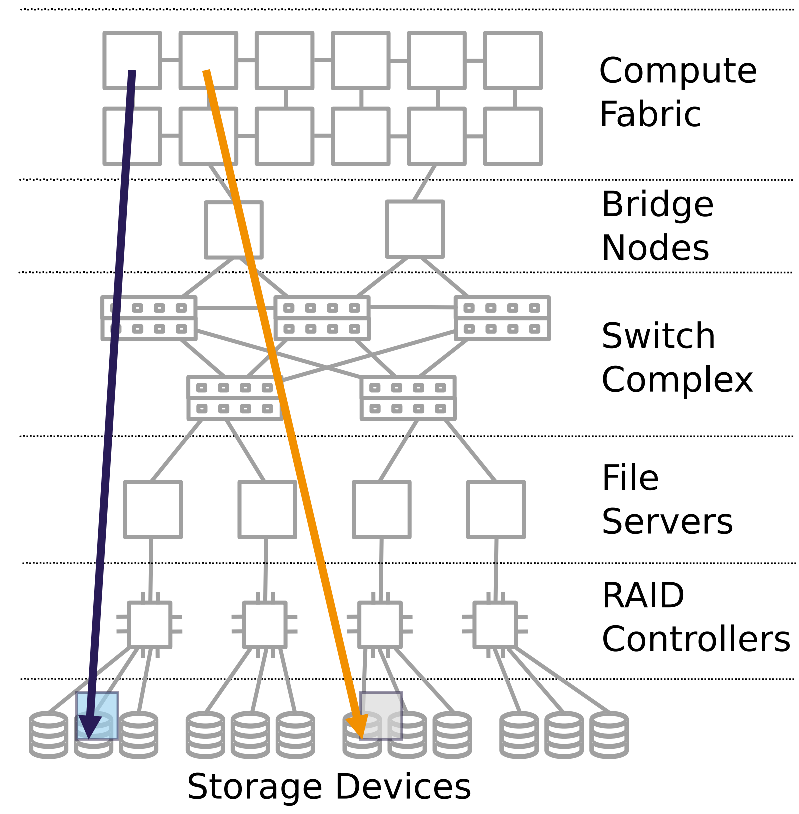
ECP EXASCALE COMPUTING PROJECT
Argonne NATIONAL LABORATORY

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

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 / storage devices.

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

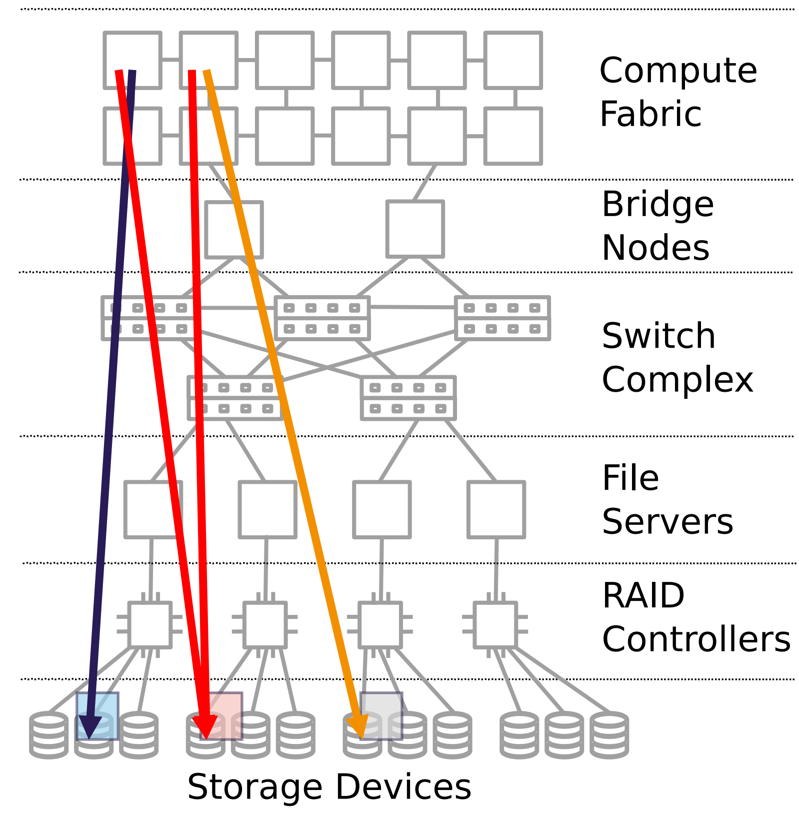


ECP EXASCALE COMPUTING PROJECT
Argonne NATIONAL LABORATORY

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 / storage devices.
- Directories are hierarchical, though, so processes will encounter contention at open() time to coordinate file names.
- Poor ratio of work (metadata) per transfer.
- (Eventually) more burden on user to manage files.

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



ECP EXASCALE COMPUTING PROJECT
Argonne NATIONAL LABORATORY

Why is concurrent access hard?
The common theme

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

The sequential POSIX API does not provide enough "big picture" information to the storage system. This makes it difficult to apply aggregate optimizations that would organize storage traffic.

Because each process acts independently using a POSIX API, the storage system has no choice but to service each I/O operation in isolation (even if there are thousands or even millions in flight). There is minimal opportunity to aggregate or structure the flow of data.

ECP EXASCALE COMPUTING PROJECT Argonne NATIONAL LABORATORY

Why is concurrent access hard?
Well, what does that leave?

- **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.
  - "***high-level I/O libraries***"
- All of them implement ***portable*** optimizations that shape traffic for parallel file systems.
- If you must 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.

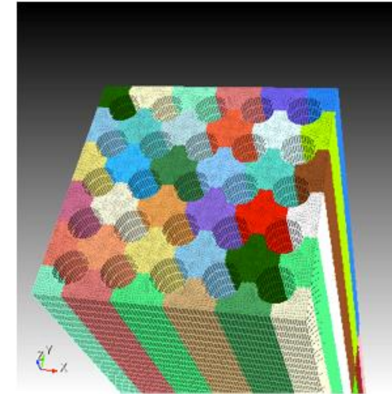
Hands on exercises: https://github.com/radix-io/hands-on

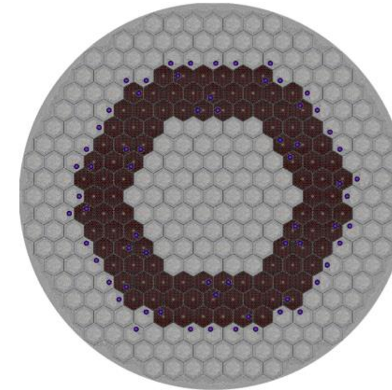
High-level I/O libraries: an early sales pitch

- 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

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

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





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.

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

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

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

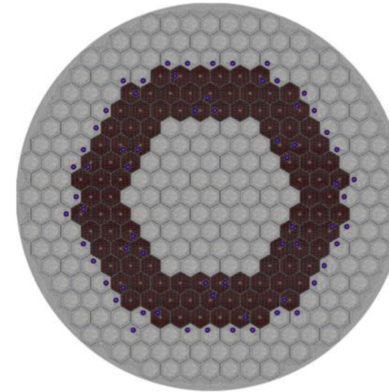
Why use a high-level data library?

- 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 hide the details of platform-specific optimizations
- Future proofing
  - e.g., interfaces and data formats that outlive specific storage technologies

Stay tuned for more information in the following sessions:

2:00 Parallel-NetCDF
2:45 HDF5

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



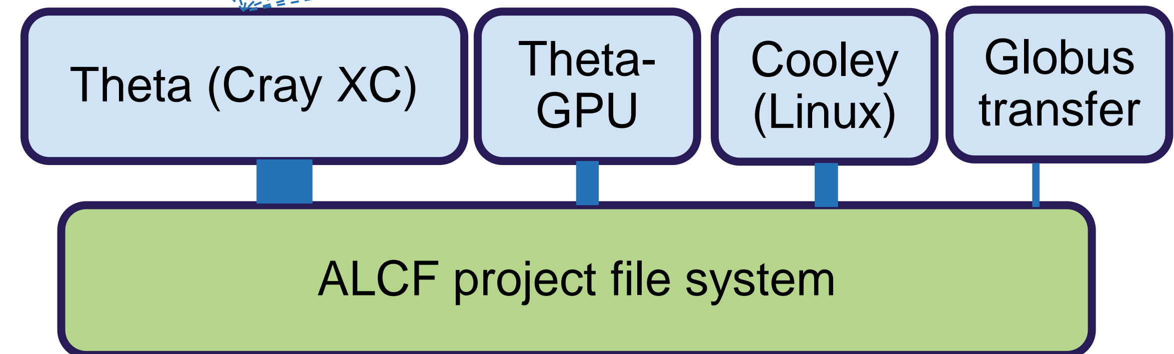
Model complexity:
Spectral element mesh (top) for thermal hydraulics computation coupled with finite element mesh (bottom) for neutronics calculation.

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

- Why:
  - 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.***

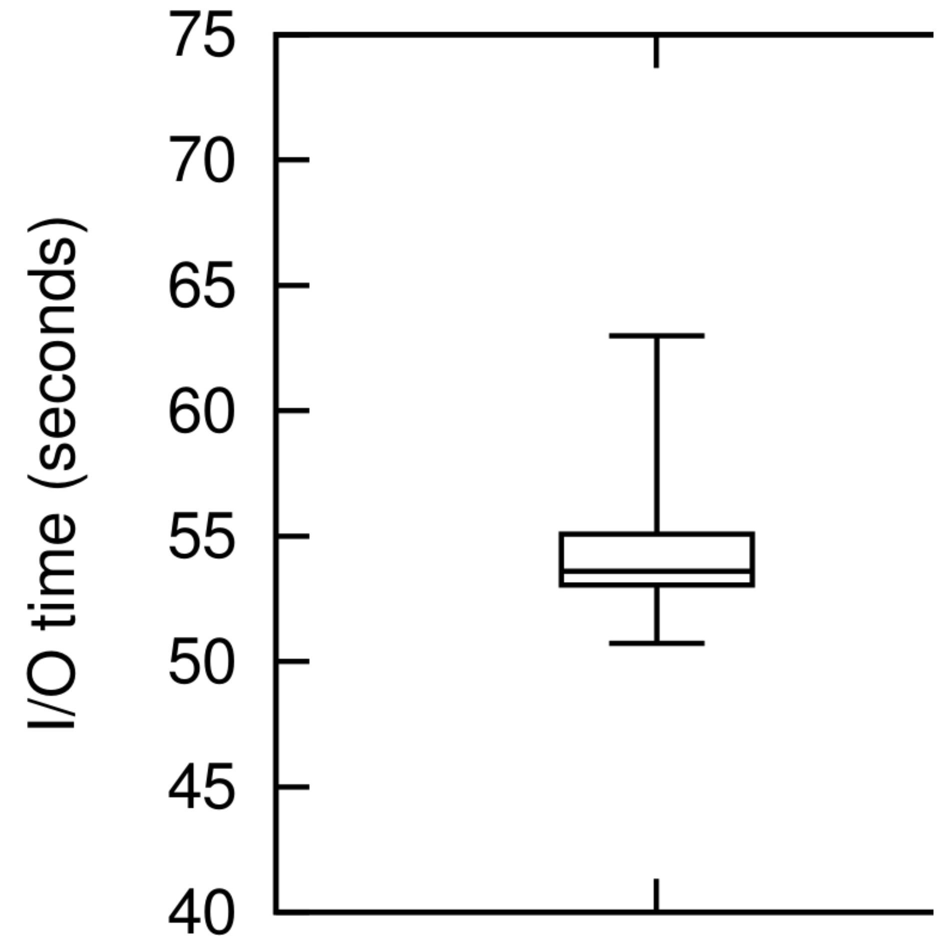
```
[          ~]$ qstat |grep running
1139867        24:00:00  8192   running    MIR-48000-7BFF1-8192
1139871        24:00:00  8192   running    MIR-00000-33FF1-8192
1143326        12:00:00  2048   running    MIR-40C00-73FF1-2048
1151809        12:00:00  4096   running    MIR-40000-737F1-4096
1153083        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
```



ATPESC 10
ECP EXASCALE COMPUTING PROJECT
Argonne NATIONAL LABORATORY

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.

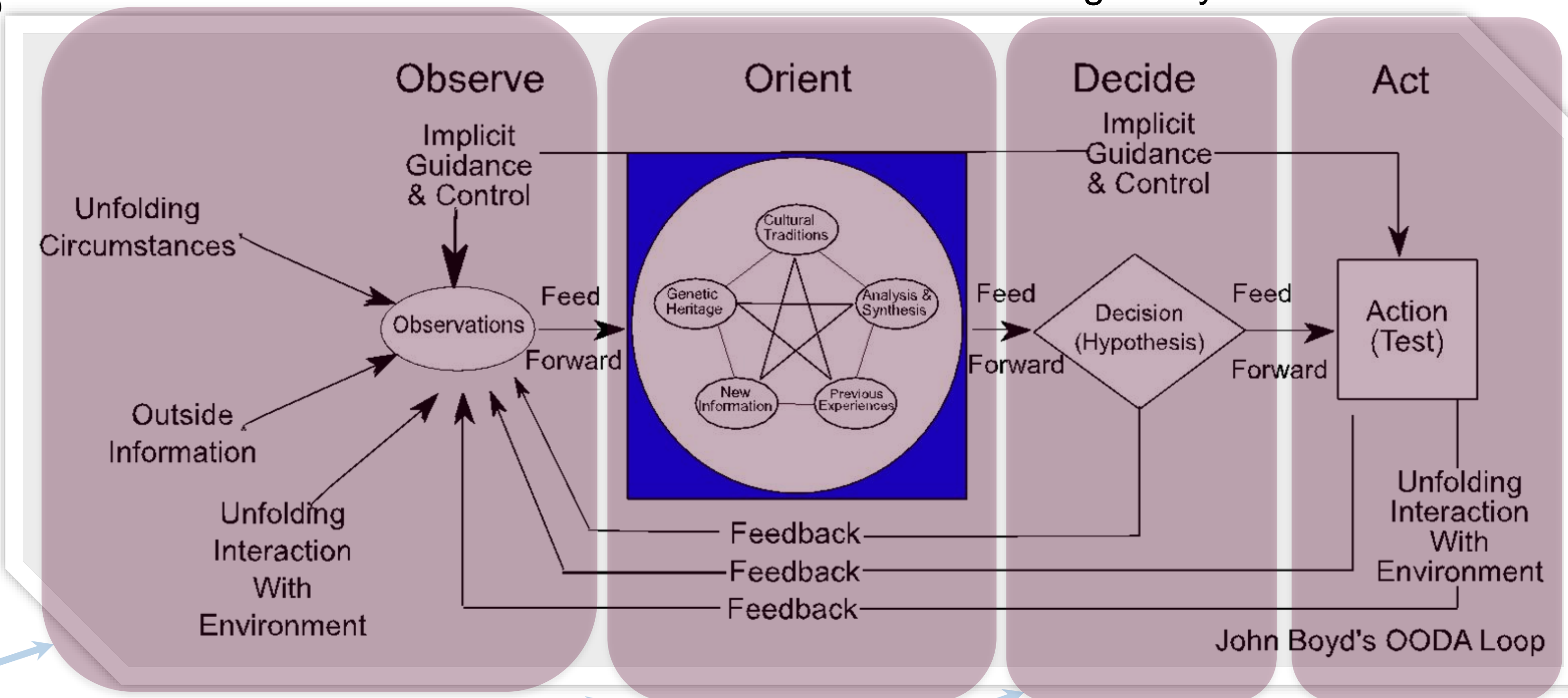


Parting thoughts: I/O performance optimization is an ongoing process

Figure by Patrick Edwin Moran

Applications are updated, systems change, and new allocations are granted.

We want to “teach a man to fish” by equipping you with the tools you need to monitor and improve your I/O performance.



Performance characterization tools (e.g., Darshan)

Background knowledge about how storage systems work (e.g., this presentation)

Facility resources (e.g., ALCF, OLCF, and NERSC staff and documentation)

Optimization techniques, tools, and libraries (e.g., later presentations today)

Thank you!