Using the NIH HPC Storage Systems Effectively

Tim Miller

btmiller@helix.nih.gov

https://hpc.nih.gov

Motivation

- Often get e-mail complaining about slow access to directories
- File storage systems are often over-saturated, leading to system problems that take staff time to resolve
- Users’ work is often negatively impacted (sometimes without their knowledge).

Course overview/outline

- Overview of HPC systems storage
  - Different areas (/home, /data, /scratch, object store)
  - Quotas and quota increases
- Understanding input and output (I/O)
  - I/O patterns
  - Introduction to data and metadata
- HPC storage systems - under the hood and beyond the basics
  - Basics of storage system architecture (and what they mean for you as a user)
  - System components – storage servers and disks
  - Caches – local and remote
- Putting it all together – using storage effectively
  - Profiling and benchmarking your application's I/O usage
  - Understanding when you're generating too much I/O on the system

Who are you?

- HPC account?
- Amount of storage used?
- Do you develop your own code or scripts to process data?
- Particular problems/issues?

Some foolish(?) assumptions

- You have basic working familiarity with the Linux command line
- You understand file and directory permissions, at least at a basic level
- You want to understand the HPC storage systems a little better so you can get your science done more efficiently.
- You're not interested in become a storage system engineer or administrator
  - i.e. you understand that some of the material given here is somewhat simplified for easier digestion

Course overview/outline

- Overview of HPC systems storage
  - Different areas (/home, /data, /scratch, /lscratch, object store)
  - Quotas and quota increases
- Understanding input and output (I/O)
  - I/O patterns
  - Introduction to data and metadata
- HPC storage systems - under the hood and beyond the basics
  - Basics of storage system architecture (and what they mean for you as a user)
  - System components – storage servers and disks
  - Caches – local and remote
- Putting it all together – using storage effectively
  - Profiling and benchmarking your application's I/O usage
  - Understanding when you're generating too much I/O on the system

The NIH HPC filesystems

Summary of file storage options

	Location	Creation	Backups	Space	Available from
/home	network (NFS)	with Helix account	yes	8 GB default quota	B,C,H
/lscratch (nodes)	local	created by user job	no	~850 GB shared	C
/scratch	network (NFS)	created by user	no	75 TB shared	B,H,C
/data	network (GPFS/NFS)	with Biowulf account	no	100 GB default quota	B,C,H

H = helix, B = biowulf login node, C = biowulf compute nodes

- /home is small – only non-buyin filesystem backed up to tape – No shared areas!
  - Snapshots and off-site tape back-ups
- /data - in practice you will keep most of your working files here
  - Quota increases available with justified need
  - Shared data directories available on request
  - Snapshots available (less frequent than /home)
- /scratch – shared area for temporary data
- /lscratch – compute node local scratch; allocated with batch jobs

What are local and network file systems?

- Network file systems are accessed by sending data to one or more servers
  - The server controls the disks that store the data
  - Multiple different client computers can access the filesystems simultaneously
- Local file systems do not send data over the network
  - Disks usually directly attached to the computer the accesses the file system
  - Only that single computer accesses the disks directly

General best practices

BAD	GOOD
Submitting a swarm without knowing how much data it will generate	Run a single job, sum up the output and tmp files, and figure out if you have enough space before submitting the swarm
Directory with 1 million files	Directories with < 5,000 files
100 jobs all reading the same 50 GB file over and over from /data/$USER/	Use /lscratch instead, copy the file there, and have each job access the file on local disk
100 jobs all writing and deleting large numbers of small temporary files	Use ~~/scratch~~ /lscratch instead, have all tmp files written to local disk
Each collaborator having a copy of data on Biowulf	Ask for a shared area and keep shared files there to minimize duplication
Use Biowulf storage for archiving	Move unused or old data back to you local system

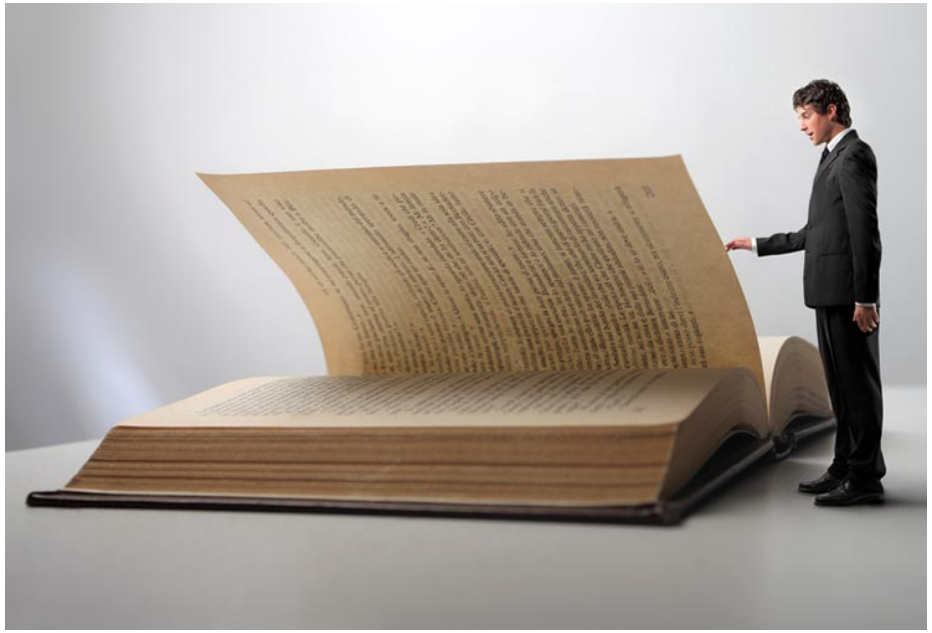
We'll be talking a lot about the "whys" behind some of these rules!

The HPC Object Store

- “Web Scale” storage
  - Highly reliable (dispersed over multiple sites)
  - Easy to expand (just add more disks)
  - Accessed via simple list, put, get, delete semantics (examples forthcoming)
- Different from file based storage systems
  - Objects are accessed by **NAME**, not **PATH**
  - Completely flat name space
  - No concept of directories, but “/” is a valid character in object names
  - Data and metadata are stored together with the object (sometimes true in file storage systems as well)
- More info: https://hpc.nih.gov/storage/object.html
- Unless otherwise noted, the rest of this class deals with **file** (not object) storage.

Use cases for object storage

- Read-intensive workloads
  - Object storage is much more efficient at reading than writing.
  - An **entire** object has to be re-written for each change
    - Computationally expensive to process and disperse the data
    - Lots of over-writing
- Static data
  - Related to the above
  - Data that doesn’t change often, but still used
  - E.g. reference genomics files



Understanding your disk quota

```
[teacher@helix ~]$ checkquota
Mount                 Used      Quota  Percent    Files    Limit
/data:            117.5 GB   500.0 GB   23.50%    16224 31129581
/gs3(cpo):         37.1 GB   100.0 GB   37.14%   118498      n/a
/gs4(wresch):       3.8 GB   100.0 GB    3.78%       46      n/a
/home:              2.3 GB     8.0 GB   28.37%    53598      n/a
mailbox:          274.0 KB     1.5 GB    0.02%
```

- Use the checkquota command
  - Shows all directories <u>you</u> have <u>write</u> access to.
  - Mailbox is only shown on helix
- You can also get this information from your user dashboard
- Some storage systems have limits on the number of files – please keep these in mind (more on this later).

Requesting quota increases

- Default data directory quota: 100 GB
- Quota increases must be justified.
  - Explain number of files, size of files, and scientific use.
  - Don't use "prior experience" or "my lab mate said"
- No quota increases on home directories

HPC Disk Storage Request

This form should be completed by NIH HPC users who require additional storage space in their /data area.

There are no time limits or other restrictions placed on the use of data directory space on the Helix and Biowulf systems, but please use the space responsibly; even hundreds of terabytes won't last forever if files are never deleted. **Disk space on Helix and Biowulf should never be used as archival storage.**

Quick Links
- Storage on Biowulf & Helix
- Sharing data with collaborators
- Storage request form
- Shared data request form

User Information:

Name:
Helx/Biowulf username:
Telephone:
Principal Investigator:
Brief description (one or two lines) of your research project:

Location

Please choose which storage location you wish to increase:

- my personal /data directory
- shared data directory **(only accepted from designated group owner)**

If shared directory is chosen, please specify the directory:

Justification

All items required.

Additional space requested (Example: 240 GB):

How this space requirement was estimated. Example: The input data for a single run of xx program is 100 MB. The output data is about 200 MB, and each run requires 100 MB of temp space. I expect to have about 600 such runs, so will need 400MB*600 = 240 GB.

Using space efficiently

- Instead of running to the quota request form, think about whether you can be using space more efficiently.
  - Move files back to your local computing infrastructure when you're done processing
  - Delete any raw data or intermediate files that you're <u>sure</u> you won't need again (or that are backed up elsewhere)
  - Compress (gzip, bzip2, etc.) files when not in active use (note: not all files compress well).
    - FASTQ files should <u>always</u> be compressed
    - Molecular dynamics binary trajectories generally don't compress much
- Storage space on the NIH HPC systems (including the object store) is NEVER to be used for archiving.

Shared data directories

- Requested from https://hpc.nih.gov/nih/shared_data_request.html
- A new group will be created (group name must begin with a capital letter).
  - Group members must be specified
- Justification for the storage request must be given as with a personal data directory quota increase.
- Don't open your personal data directory to world access!
  - Shared group directories should only be accessible by the group that owns them.
- Requestor becomes "group owner"
  - The only one who can request a quota increase
  - The only one who can request users be added to or removed from the group

HPC Shared Data Request

This form should be completed by Helix/Biowulf users who would like to share data in a secure shared data area. This is done by creating a Unix group that contains two or more Helix/Biowulf users. For more information about Unix sharing data on Helix/Biowulf, see http://hpc.nih.gov/storage/sharing_data.html.

Quick Links
- Storage on Biowulf & Helix
- Sharing data with collaborators
- Storage request form
- Shared data request form

Group Owner: The group owner will have the responsibility of managing the members of the group and the permissions of the files within. **The group owner is the only group member from whom requests will be accepted regarding changes to group membership, permissions and quota.**

Name:

Helix/Biowulf username:

Telephone:

Principal Investigator: The principal investigator will take ownership of the data if the accounts of all members of the group are deleted.

Principal Investigator:

Group Information: Please enter information about the Unix group. The group will be created if it does not yet exist. The group name must begin with an uppercase letter.

Preferred name of the group (must begin with a capital letter and be less than 16 characters):

Members of the group (Helix/Biowulf usernames, space or comma delimited):

Brief description (one or two lines) of your research project:

Justification

Failing to answer all of the items below will result in a delay of your request for a shared data directory.

Amount of space requested (Example: 500 GB):

How this space requirement was estimated. (Example: Each of the six members of our group will be handling about 1000 files of 1GB each, so 6TB in total)

Shared data directories and permissions

```
helix:/spin1/u[illegible]6$ ls -ld /data/SomeLab/
drwxrws--- [illegible] btmiller SomeLab 512 Sep 23 11:43 /data/SomeLab/
```

Group owner – the ONLY user allowed to request quota increases for the group.

- New directories created under the top level will generally **NOT** have the SGID bit set.
- Users can override group ownership and permissions of directories that they create within the shared area.
  - Some applications do this without informing the user.
- Coordination and care among group members is needed.

Shared data directories and permissions

```
helix:/spin1/users 6$ ls -ld /data/SomeLab/
drwxrws--- 3 btmiller SomeLab 512 Sep 23 11:43 /data/SomeLab/
```

Group name – same as the shared data directory name.

- New directories created under the top level will generally **NOT** have the SGID bit set.
- Users can override group ownership and permissions of directories that they create within the shared area.
  - Some applications do this without informing the user.
- Coordination and care among group members is needed.

Shared data directories and permissions

```
helix:/spin1/users 6$ ls -ld /data/SomeLab/
drwxrws--- 3 btmiller SomeLab 512 Sep 23 11:43 /data/SomeLab/
```

Permissions – all group members can write at the top level. The SGID bit ("s" in the group execute permissions) indicates all files created in this directory will have a group ownership of SomeLab.

- New directories created under the top level will generally **NOT** have the SGID bit set.
- Users can override group ownership and permissions of directories that they create within the shared area.
  - Some applications do this without informing the user.
- Coordination and care among group members is needed.
  - Set umask to 007 so that all files/directories created become group writeable
  - Some users request that their primary group be changed to the shared group

A few notes about /scratch

- /scratch is a **network accessed, global** /scratch directory
  - Prior to mid-2015 (before Biowulf 2), /scratch was a **local** disk file system
  - The same scratch directory is available on helix, biowulf, and all of the compute nodes.
  - Files deleted if not accessed for 10 days!
- Good use-cases for /scratch
  - A temporary means of sharing data with a colleague who has a HPC account
  - Storing lightly accessed temporary files that must be accessed from multiple nodes.
- Bad use-cases for /scratch
  - Storing application temporary files that are only accessed from a single host (use /lscratch instead).
  - Storing anything that must be read and written frequently (use /data or /lscratch instead).
  - Storing valuable, hard to reproduce data (no back-ups; use /home or /data as appropriate).

Using /lscratch

- /lscratch = local scratch space on a compute node
- Allocated as a generic resource:
  - sbatch … --gres=lscratch:100 ← allocates 100 GB
  - swarm -f swarmfile --gres=lscratch:100 ...
  - Ibid for sinteractive
  - At /lscratch/$SLURM_JOBID
- Benefits
  - Local to node, no network traffic
  - Fewer users sharing it
  - New nodes have faster solid-state disks

When to use /lscratch (and when not)

- Use /lscratch when…
  - Many jobs will be <u>independently</u> reading in the same data file.
  - Many jobs will be <u>independently</u> writing out output files.
  - Many jobs will be writing out a lot of <u>independent</u> temporary files
  - Jobs will be doing large amounts of random I/O (more on this later)
  - Swarms that read/write a lot of I/O should almost always use lscratch!!!
- Don't use /lscratch (or why bother) when...
  - Your job needs to write out a file visible to multiple nodes (this is rare?)
  - Your job is only reading/writing a few files and not very much data.
- If in doubt, e-mail staff@hpc.nih.gov

Workflow for /lscratch

- Copy high-intensity input data from network directory to lscratch
  - Don’t have 50 independent processes all reading the same input
  - Copy the input into each process’s lscratch directory
  - Multiple copies per host? Can avoid this with clever scripting…
  - Remember to copy results back when done – lscratch goes away after your job finishes!
- Use lscratch for temporary files/scratch space
  - Set environment variables – e.g. $TMPDIR, $SCRATCH
  - Scratch/temp space variables are often program specific – need to know your application
- Use lscratch as a local cache for objects from the object store.

A rough guide to choosing a storage system

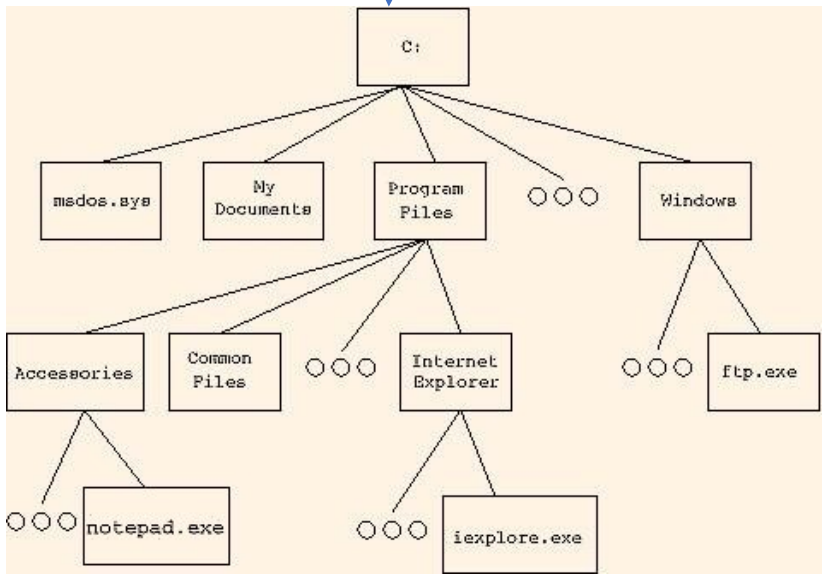
- /home
  - Small files that are very important, low I/O intensity
  - Not for files that need to be shared with other users
- /data
  - Bulk data files and scripts
  - Consider using shared data areas for sharing
  - Shared, parallel filesystems – intensive, single node I/O -> /lscratch.
- /scratch
  - **Low I/O intensity** job data that needs to be accessed from multiple nodes.
  - Good for short-term sharing.
- /lscratch
  - **High I/O intensity** job data that only needs to be accessed from a single node.
- Object
  - Primarily read-only data; low to medium intensity, but large capacity.

Course overview/outline

- Overview of HPC systems storage
  - Different areas (/home, /data, /scratch, object store)
  - Quotas and quota increases
- Understanding input and output (I/O)
  - I/O patterns
  - Introduction to data and metadata
- HPC storage systems - under the hood and beyond the basics
  - Basics of storage system architecture (and what they mean for you as a user)
  - System components – storage servers and disks
  - Caches – local and remote
- Putting it all together – using storage effectively
  - Profiling and benchmarking your application's I/O usage
  - Understanding when you're generating too much I/O on the system

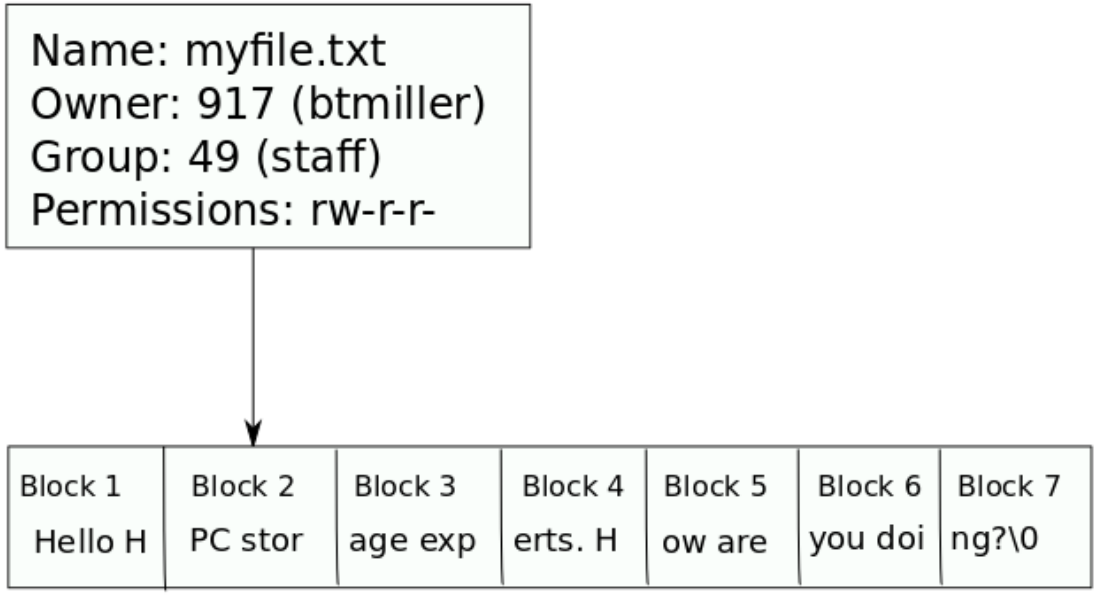
What is a filesystem?

- A filesystem mediates between a physical storage device (e.g. a hard disk or a thumb drive) and a more user-friendly view of files and directories.
- Can span a single computer or a network of computers
- Some popular (?) filesystems:
  - Windows: NTFS, FAT
  - Mac/iOS: HFS+, APFS
  - Linux: ext4, XFS
  - Parallel computing/HPC: NFS, GPFS, Lustre



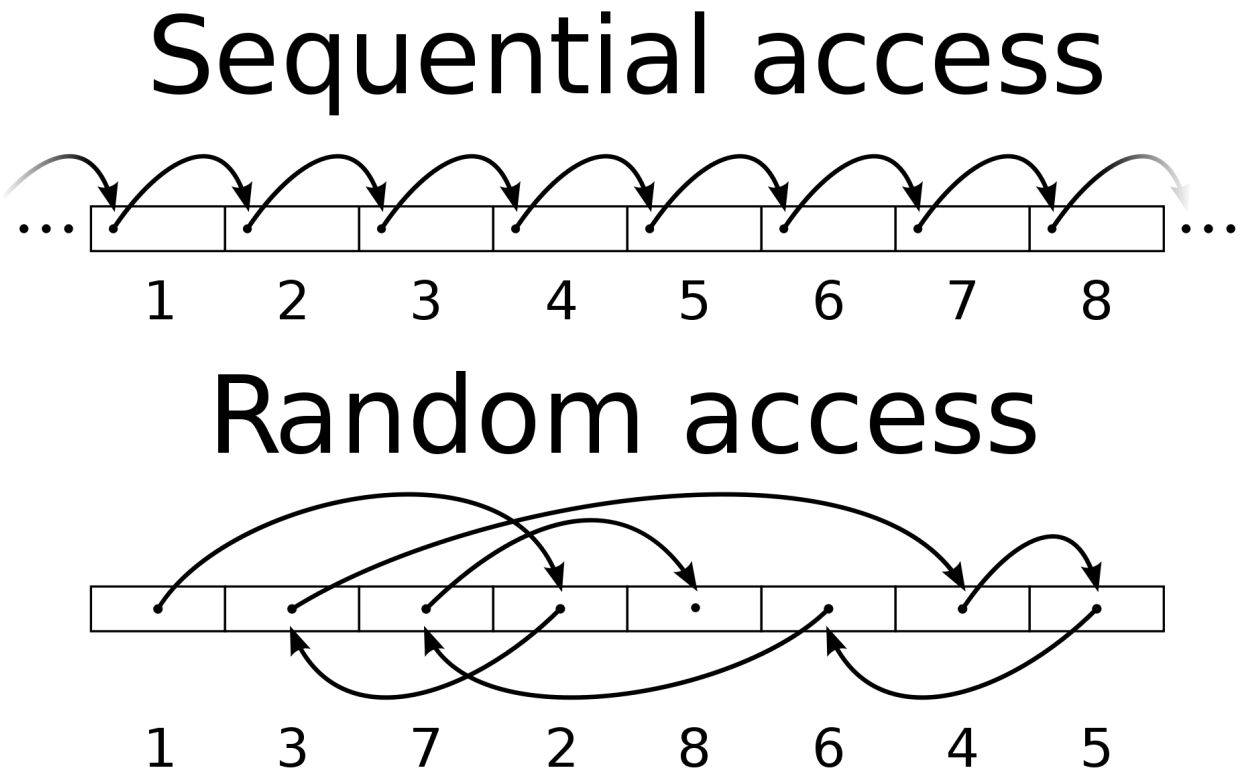
A block about blocks

- A filesystem organizes files and directories into **blocks** of data
  - This is because hard disks can only read and write information in discrete-sized chunks.
- Each non-zero length file you store takes up at least one full block on the filesystem.
  - The exact size of a block differs from filesystem to filesystem and may be chosen by the administrator who creates the filesystem.



Sequential vs. random access

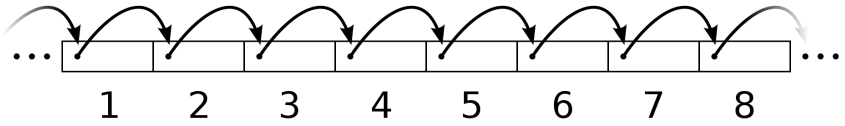
- Sequential access: blocks in a file are read one after the other
  - Example: reading in a series of sequences from a file, one after another.
  - Sometimes referred to as “streaming”.
  - Special cases, e.g. reading a file backwards.
- Random access: blocks in the file are read in a random order.
  - Common in database applications
  - Much harder for I/O systems to optimize



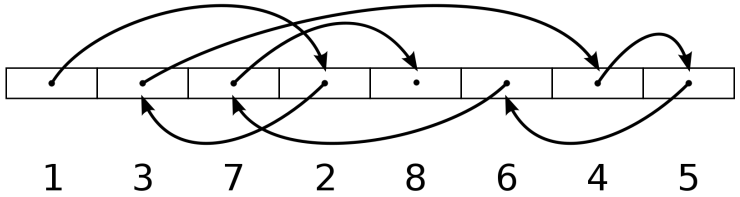
Exercise 1: sequential vs. random?

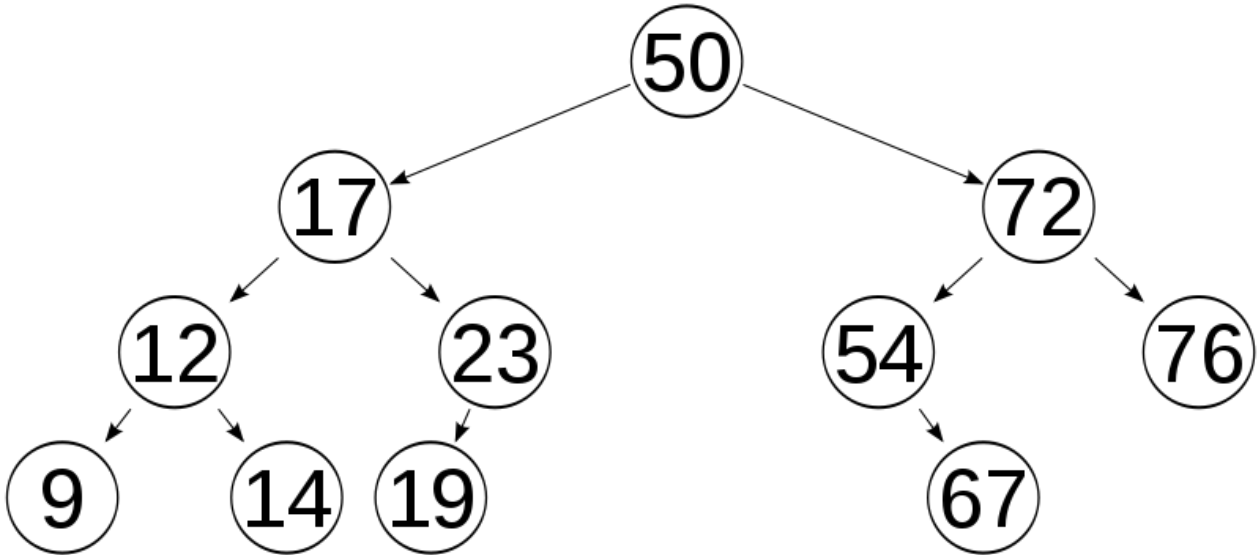
- Are the following procedures likely to have primarily a sequential I/O pattern, or a random one?
  - Reading a fastq file into memory?
  - Extracting random sequences from a BAM file?
  - Searching through a BAM file to find all sequences matching a given pattern?
  - Getting colors of non-contiguous pixels from a PNG file?
  - Traversing a file whose contents are organized as a binary tree?

Sequential access



Random access



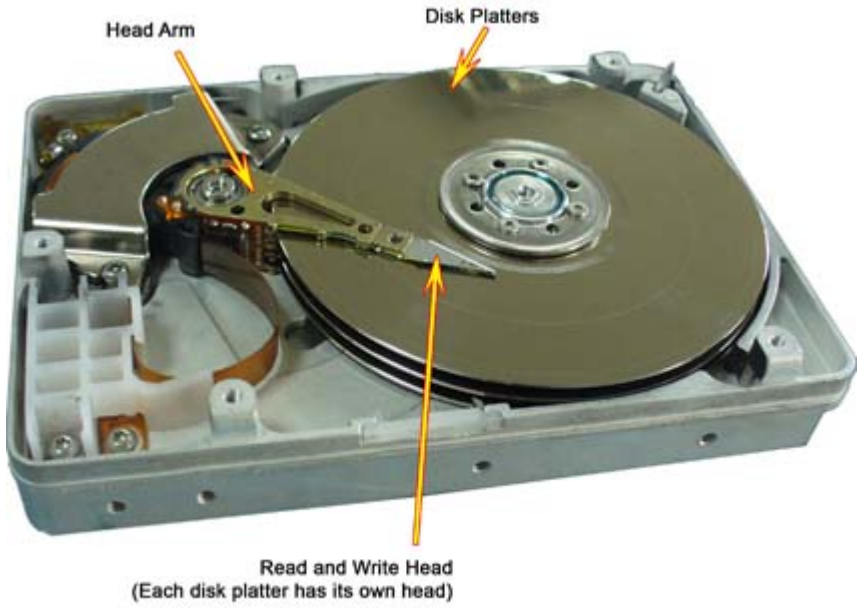


Exercise 2: Your own application

- What kinds of applications do <u>you</u> run on Biowulf?
  - Genomics?
  - Simulation/molecular dynamics?
  - Informatics?
- What kind of workload does it produce?
  - Reading lots of data into memory (sequential)?
  - Un-ordered search through many sequence files (random)?
  - Writing out checkpoints or trajectories for simulation (sequential)?
  - Other/unknown?

Why is random I/O so much slower?

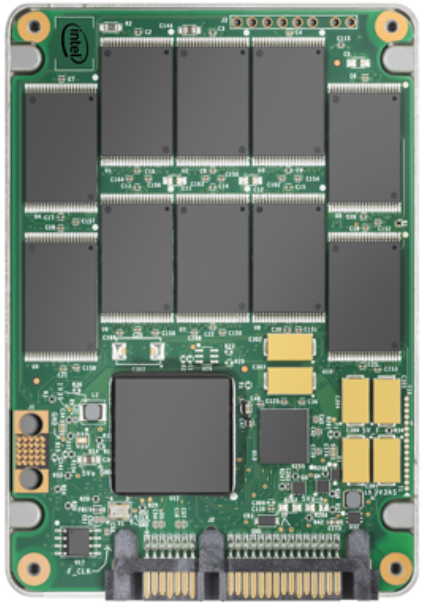
- Until very recently all hard disks were so-called rotational media
  - Data is stored on one or more magnetic platters
  - Data is read and written by heads with small magnets.
- The speed at which the platters and heads could move gives a hard upper limit on performance
  - 7200 RPM drive ~ 0.4 millisecond average latency
  - Compare to nanosecond latency between CPU ops.
- HPC systems use many hard drives working in tandem, but the same basic physics hold.
- Note: this is an oversimplification, but a useful one to keep in mind.



Inside Hard Disk

Why is random I/O so much slower?

- Now we have solid state disks (SSDs)
  - Instead of magnetic platters, data is stored on nonvolatile memory chips.
  - No more having to wait for the platters to physically rotate.
  - Random I/O is much, much better
- Not a panacea
  - Still have to consider reading/writing multiple chips.
  - Semantics of updating flash chips can cause performance degradation.
  - More expensive per GB.
- The HPC storage systems put some metadata and some performance critical data on SSDs, but most data is kept on rotational hard drives!



I/O comes in all sizes

- In addition to the <u>pattern</u> of I/O, we also have to be concerned with the size of I/O.
- Which do you think will be faster?
  - An application that reads 100 MB sequentially by issuing 50 read requests of 2 MB each
  - An application that reads 100 MB sequentially by issuing 5000 read requests of 20 KB each

I/O comes in all sizes

- In addition to the <u>pattern</u> of I/O, we also have to be concerned with the size of I/O.
- Which do you think will be faster?
  - An application that reads 100 MB sequentially by issuing 50 read requests of 2 MB each
  - An application that reads 100 MB sequentially by issuing 5000 read requests of 20 KB each
- The first case is (usually) a lot faster
  - There's a certain amount of overhead in performing requests.
  - Some systems will try to coalesce multiple small requests into bigger ones
  - With random I/O, this can be difficult to impossible.

I/O comes in all sizes

- In addition to the <u>pattern</u> of I/O, we also have to be concerned with the size of I/O
- Which do you think
  - An application [illegible] requests of 2 MB each
  - An application [illegible] requests of 20 KB
- The first case is (usually) a better
  - There's a certain amount of overhead in performing requests.
  - Some systems will try to coalesce multiple small requests into bigger ones
  - With random I/O, this can be difficult to impossible.

Unless you are writing your own application, you don't have any direct control over this. However, it's something to be aware of!

Data vs. metadata

- When we think of a file, we usually think in terms of the **data** it contains.
  - A BAM file contains sequence alignments
  - A molecular dynamics trajectory contains atomic coordinates/velocities
  - An MRI output contains an image of someone's brain.
  - An EM image contains a picture of a cell in the body.
- **Metadata** literally means "data about data"

THINGS GOT TENSE FOR THE TOWNSFOLK WHEN THE THIRD META-ANALYST GANG RODE INTO TOWN.

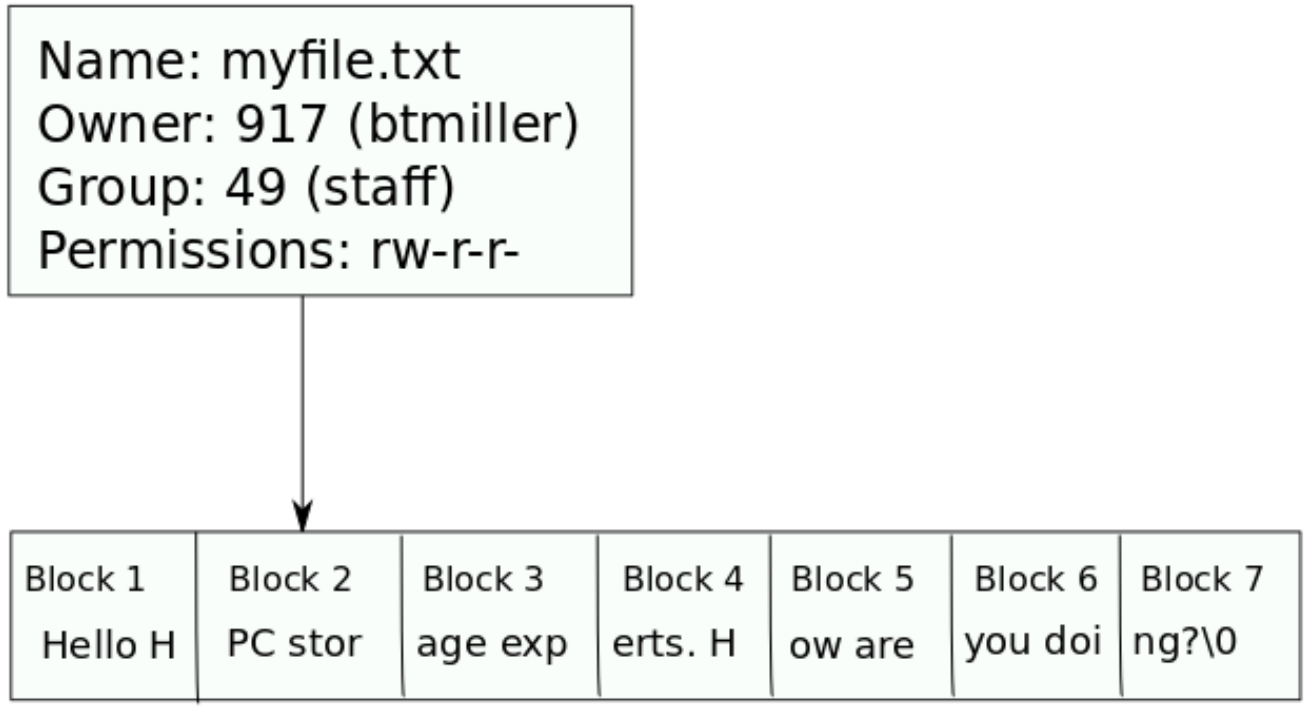
Data vs. metadata

- Examples of metadata
  - When the file was created/accessed
  - The sample ID from which the file was generated
  - The access permissions of the file
  - Where the data is located physically on the underlying disk
  - Can you think of more examples?

THINGS GOT TENSE FOR THE TOWNSFOLK WHEN THE THIRD META-ANALYST GANG RODE INTO TOWN.

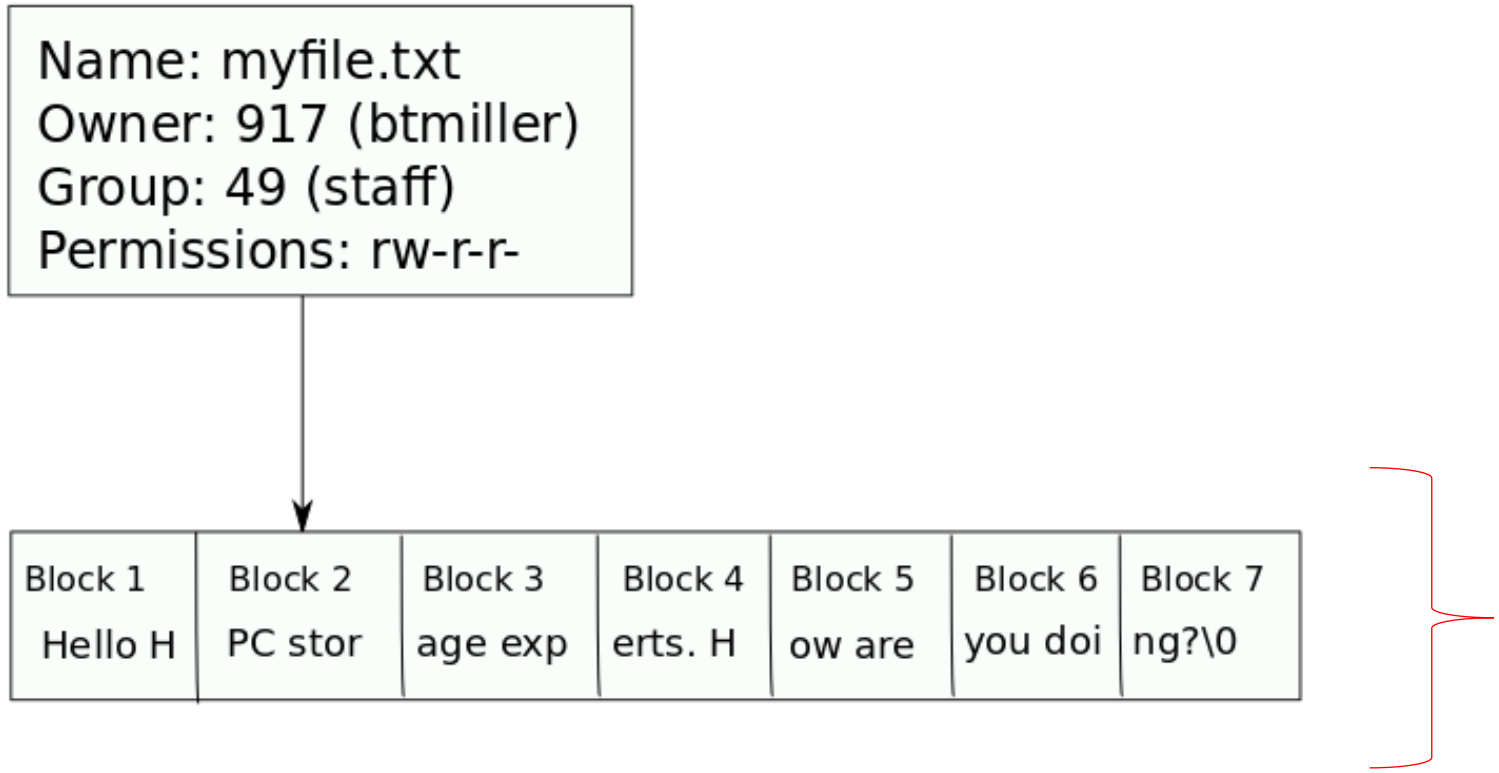
A conceptual diagram

- Remember our simple schematic…



A conceptual diagram

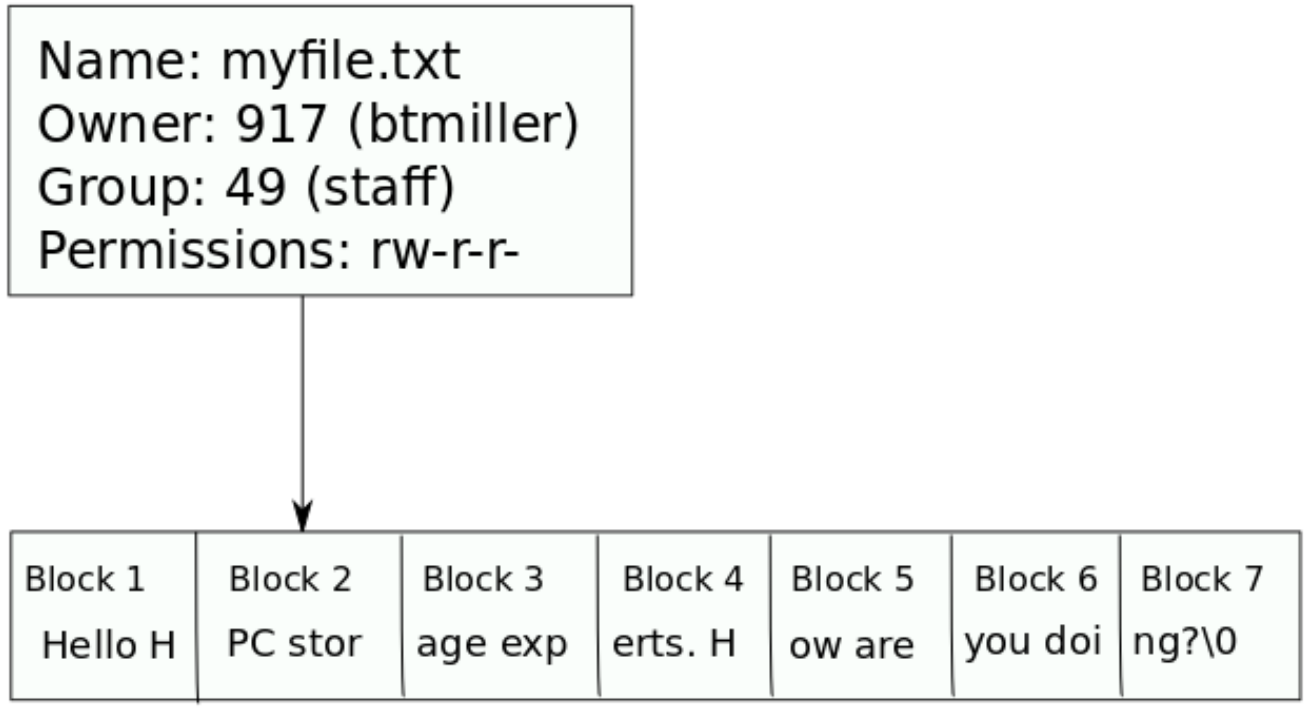
- Remember our simple schematic…



This is the DATA in the file.

A conceptual diagram

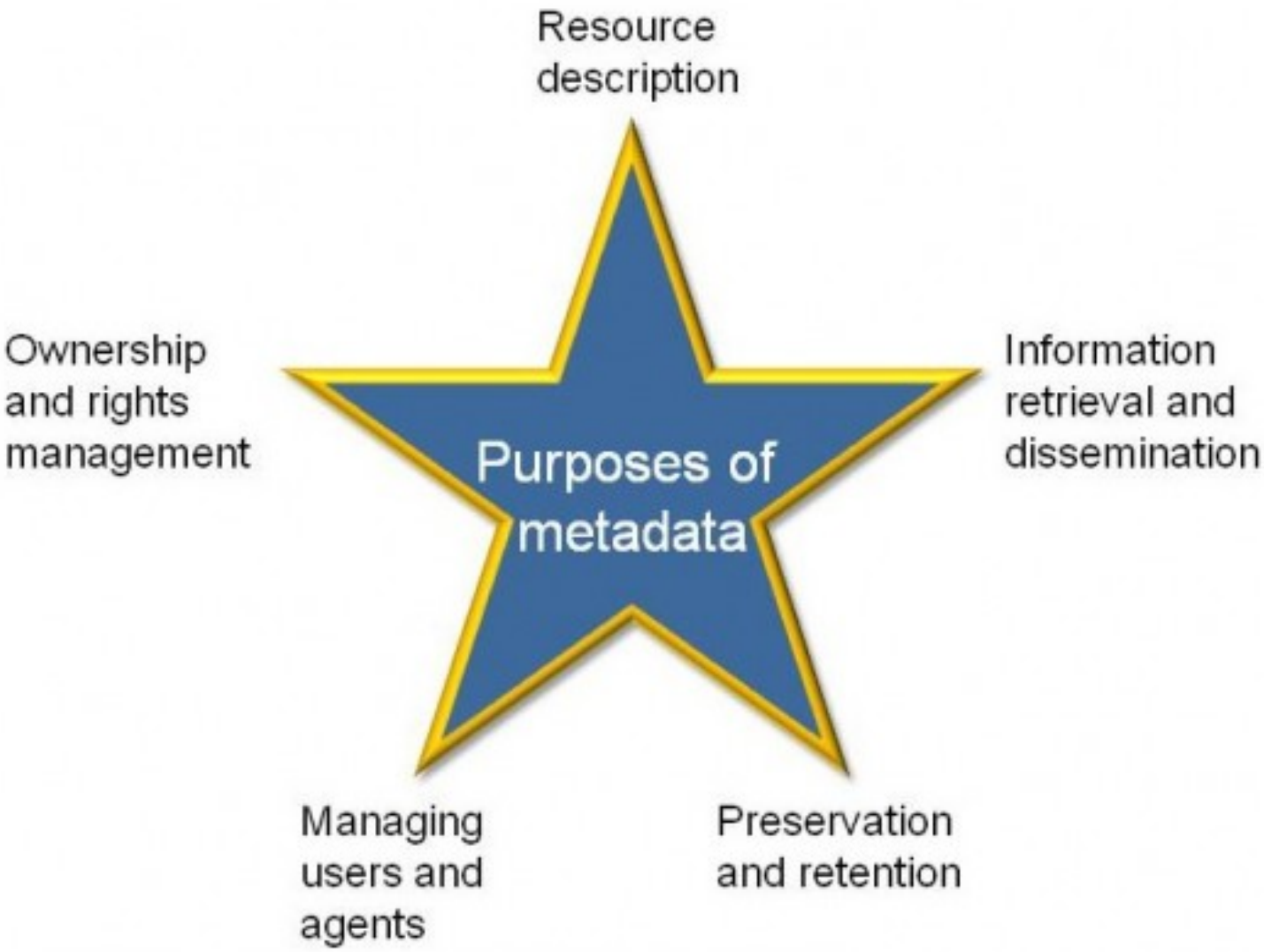
- Remember our simple schematic…



This is (some of) the METADATA

This is the DATA in the file.

Purposes of metadata



How storage systems store metadata

- Most filesystems have the concept of an “i-node”
  - Special disk block on the filesystem that holds information about a file.
  - Limited, pre-defined metadata space
  - Has pointers to the blocks on a disk that hold the data
  - Many recent filesystems allow user-modifiable, arbitrary metadata via setfattr/getfattr (but it’s usually not enabled)
- Some filesystems optimize performance of i-nodes
  - i-nodes can be arranged in clusters so that rotational media can access them more quickly (Berkeley Fast File system)
  - i-nodes are cached in nonvolatile RAM (think of a small, very fast SSD)
  - i-nodes are stored on SSDs (even if the data blocks are on rotational media)

Looking at some metadata with stat

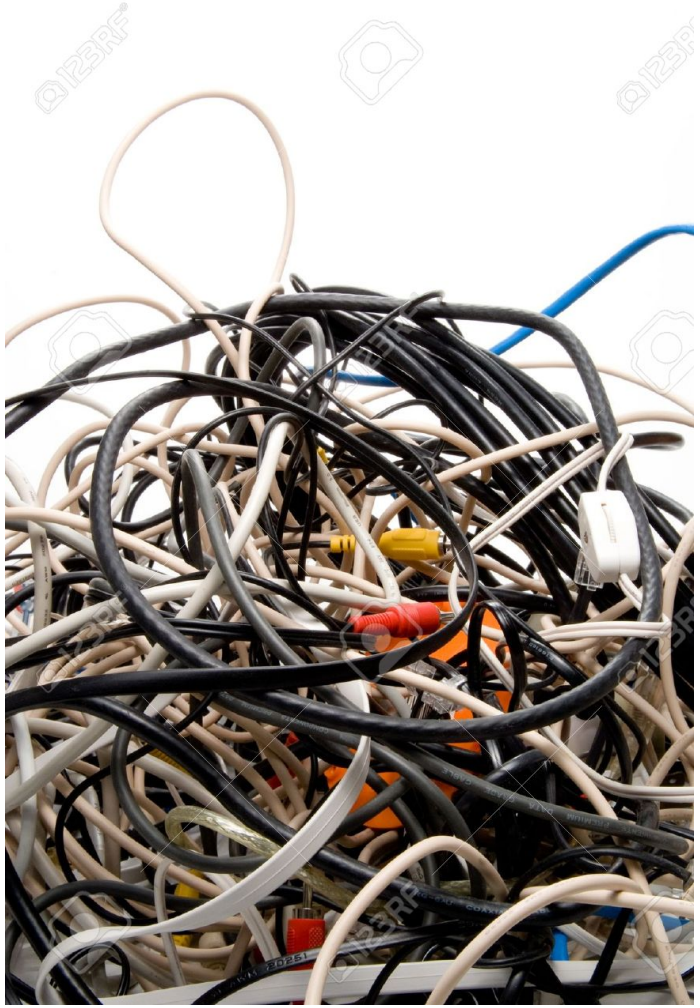
```
biowulf:~ 35$ stat /bin/ls
  File: `/bin/ls'
  Size: 117048          Blocks: 232        IO Block: 4096   regular file
Device: fd01h/64769d    Inode: 1452928     Links: 1
Access: (0755/-rwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2017-02-01 01:36:55.388977943 -0500
Modify: 2016-04-12 04:01:18.000000000 -0400
Change: 2017-02-01 01:36:35.302980076 -0500
```

Some notes on directories

- Directories are special files that hold pointers (links) to other files.
- The more files there are in a directory, the larger amount of space the directory blocks will take up on disk
- Listing directories, resolving path names, moving files, etc. all require operations on the directory blocks.
  - The file system has to iterate through files in the directory individually
  - The bigger the directory is, the longer these operations will take.
  - This is why the HPC staff recommends having < 5000 files per directory
  - Especially true with directories that will have lots of operations happening simultaneously!

File and directory parallel access

- Reading and writing the same file from multiple parallel jobs can cause contention.
  - Especially with writing – due to the need for locks to avoid corruption
- Likewise, lots of different processes listing, creating files, etc. in the same directory is a bottleneck.
- Constant creation and deletion of files can create performance issues, particularly when multiple processes are doing it in the same directory.



Exercise 3: Good practice or bad practice?

- Which of the following are not good practice? Why?
  - Having 1,000,000 data files in a single directory.
  - Having separate runs of a program write output to separate files.
  - Reading a data file in once at program initiation, and then keeping the data cached in memory.
  - Using the name of a file to encode program results.
  - Having a swarm of 1,000 jobs each use the same temporary directory in /scratch.

A few final notes on metadata

- **Every** user on a given filesystem accesses the same pool of metadata.
- Metadata heavy workloads thus radically slow down filesystems for everyone!
- So avoid:
  - Lots of file creation
  - Large number of directory listings.
- Less true on object store…

Course overview/outline

- Overview of HPC systems storage
  - Different areas (/home, /data, /scratch, object store)
  - Quotas and quota increases
- Understanding input and output (I/O)
  - I/O patterns
  - Introduction to data and metadata
- HPC storage systems - under the hood and beyond the basics
  - Basics of storage system architecture (and what they mean for you as a user)
  - System components – storage servers and disks
  - Caches – local and remote
- Putting it all together – using storage effectively
  - Profiling and benchmarking your application's I/O usage
  - Understanding when you're generating too much I/O on the system

Basic storage system architecture

- /home, /scratch, and some /data directories are on a large storage system that uses NFS.

Basic storage system architecture

- Other data directories are on systems running IBM's General Parallel File System (GPFS)

NFS and GPFS

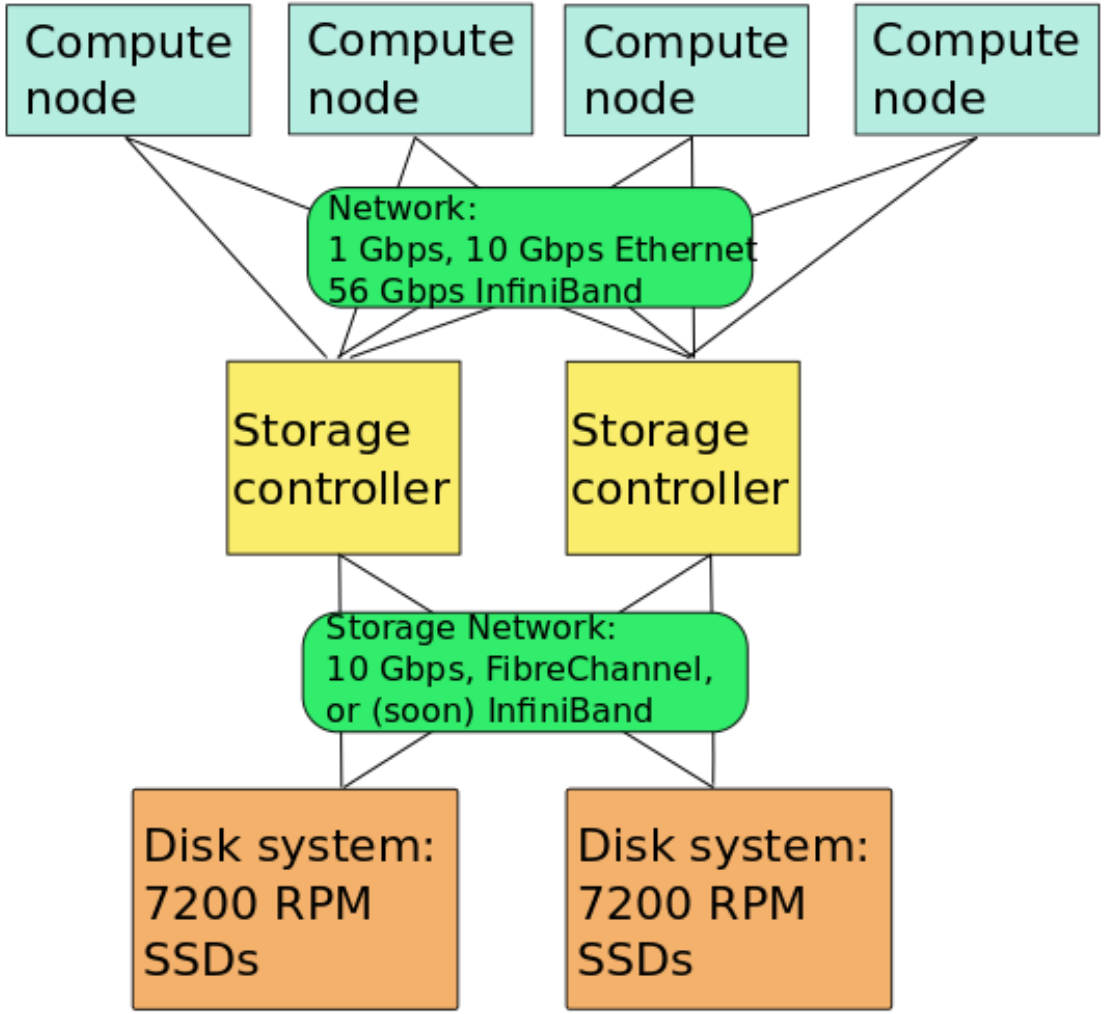
- NFS and GPFS have different back-end implementations, but from a user's perspective, they work the same way.
- The systems perform similarly, though file system performance is variable dependent on how many users are accessing a given filesystem at any one time.
  - There is **no** significant performance advantage to using one system vs. the other.
- Main difference from a feature perspective: GPFS has access control lists (ACLs) whereas the NFS implementation we use does not.
  - ACLs are an advanced way of setting granular file/directory permissions – see https://hpc.nih.gov/storage/acls.html for more details
  - We will not discuss ACLs further in this class (unless someone <u>really</u> wants to).

Figuring out where your data is stored

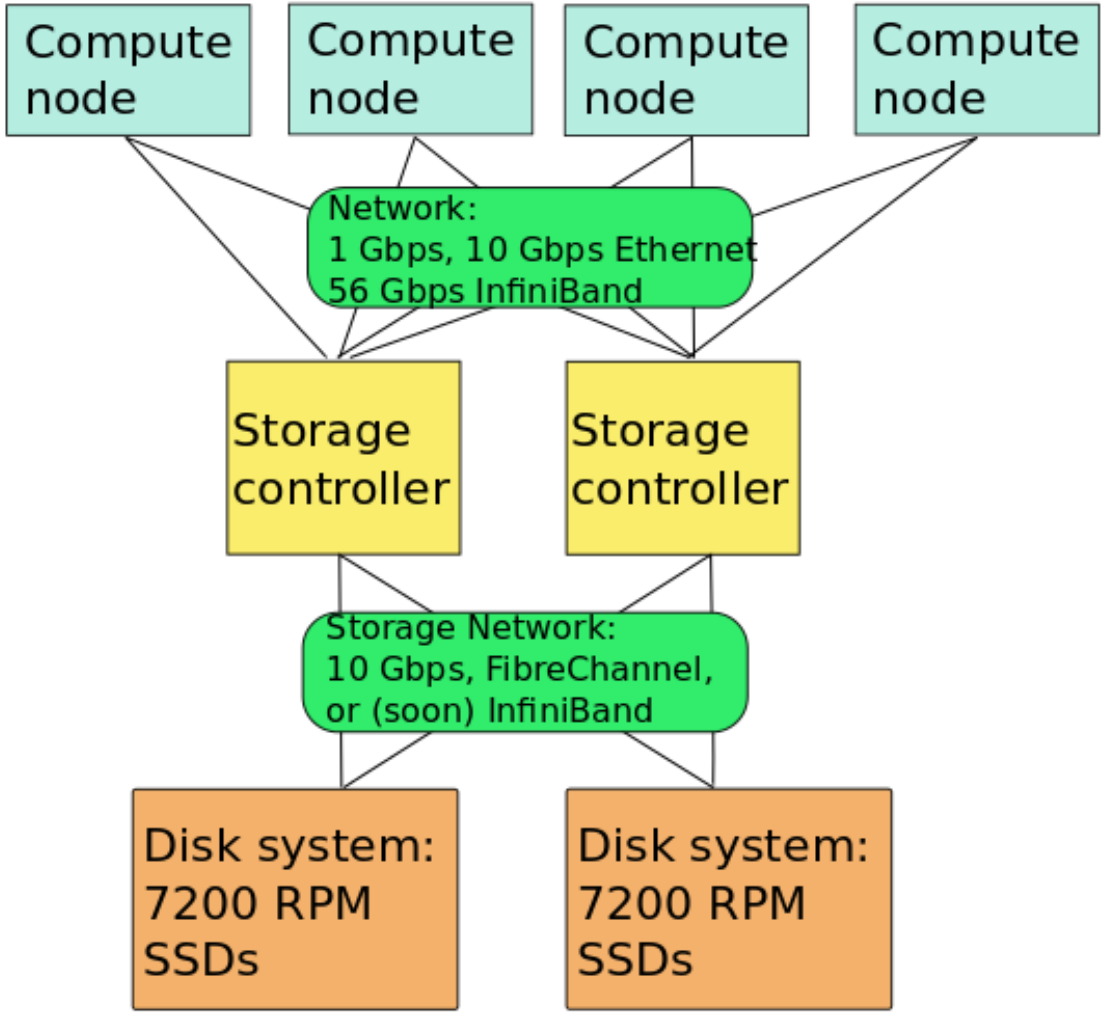
- You can use the “-a” flag on checkquota to see what filesystems the directories you have access to are on.
- spin1 = NFS, /gs[2-6] = GPFS
- **NEVER** refer to any data directory by its absolute path (i.e. use /data/username **NOT** /spin1/users/username
  - The storage admins move directories for a variety of reasons, so the absolute paths can and do change.

```
[teacher@biowulf ~]$ checkquota -a
Mount                        Used       Quota  Percent    Files    Limit
/gs3/users/cpo:           37.1 GB   100.0 GB   37.14%   118498      n/a
/home/teacher:             3.2 GB     8.0 GB   40.12%    52786      n/a
/spin1/users/teacher:    118.8 GB   500.0 GB   23.76%    16226 31129581
```

Storage system architecture schematic



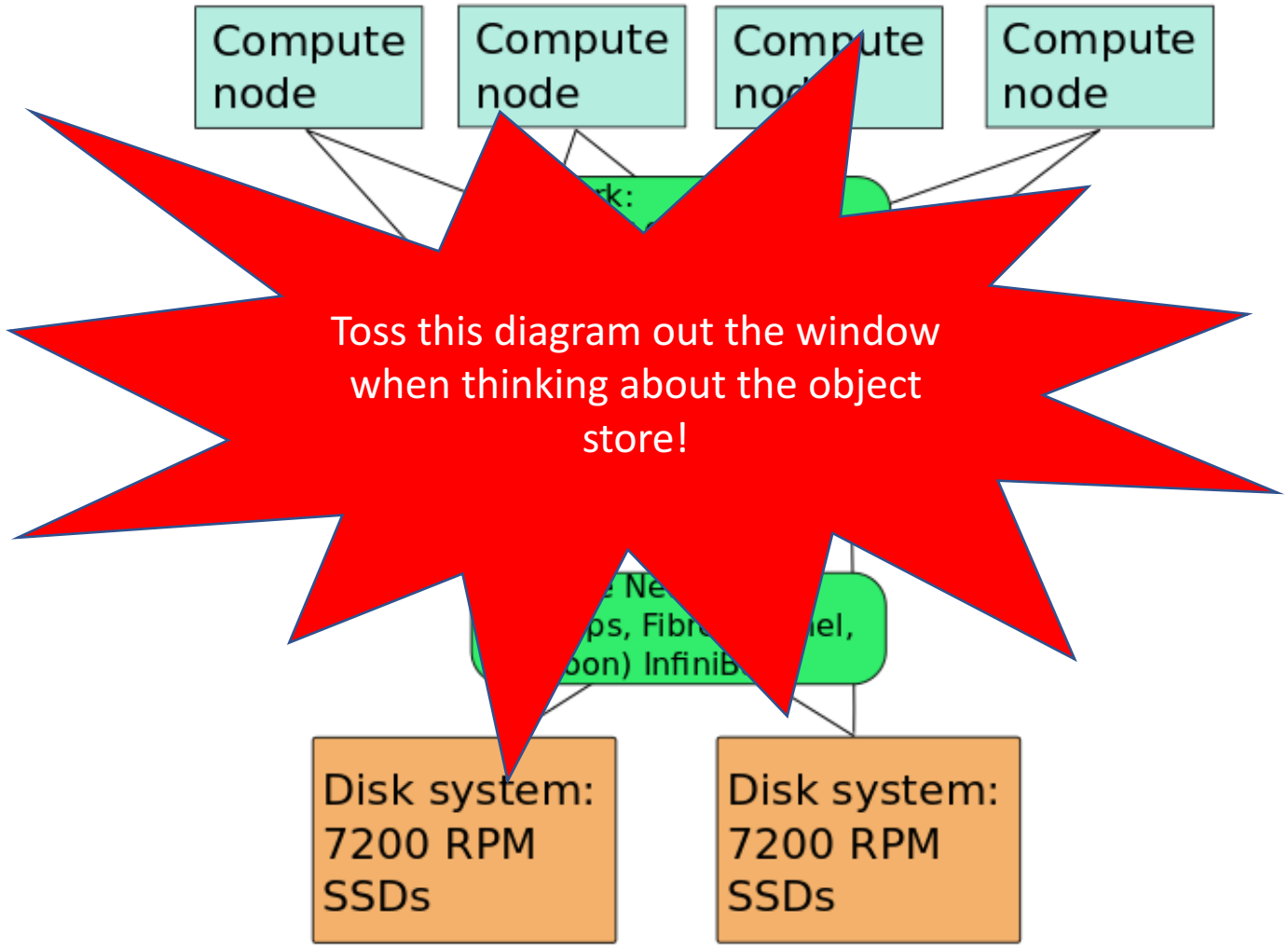
Storage system architecture schematic



Note: this is intended to be a generic diagram, applicable to both NFS (NetApp) and GPFS storage.

Implementation details vary from system to system.

Storage system architecture schematic

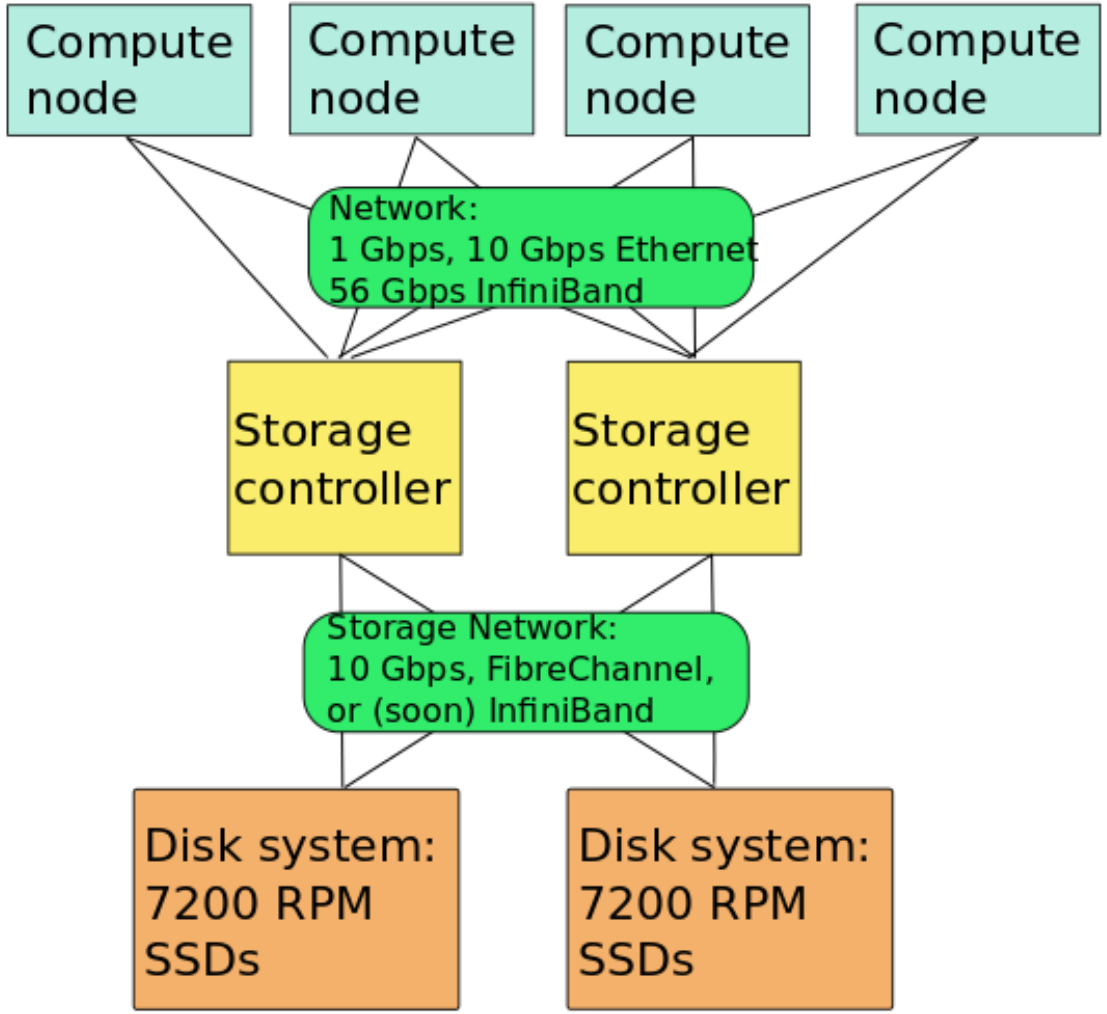


Note: this is intended to be a generic diagram, applicable to both NFS (NetApp) and GPFS storage.

Implementation details vary from system to system.

Consequences of the storage architecture

- Highly parallel system
  - Multiple storage controllers
  - Many operations will be routed through a single controller, though!
  - Redundant access to underlying disks
  - Designed for many users to access simultaneously
- Many places for bottlenecks!
  - “Front end” network from the node to the controller
  - CPU, memory, I/O on the controller
  - “Back end” network from the controller to the disk.
  - On the disks themselves.



One key point to remember…

- In general, the network has far more bandwidth than the back-end disk media.
  - This is just a function of the speed of rotational disks!
  - Just because a node has a 10 Gbps network connection, don't assume you'll get that speed to the storage!
  - Obviously not true for the oldest nodes with only gigabit Ethernet.

>

What you can do to avoid bottlenecks!

- Use /lscratch <u>whenever possible</u>!!!
  - Since lscratch is local to the node – avoid all network operations
  - Also, lscratch on newer nodes is provisioned with SSDs!
- Avoid many parallel I/O operations.
  - They tend to oversaturate the disks
- Try to do I/O on large chunks
  - i.e. read and write large amounts of data
  - Less network overhead, and easier for the disk systems to optimize
  - If you’re using someone else’s code, this is difficult/impossible
- Avoid excessive metadata operations
  - Tend to be filtered to a small amount of disks/controllers.
  - This includes directory operations!

Exercise 4 – where is the bottleneck?

- For the ”bad practices” we identified earlier (marked in red), what bottlenecks are likely to be relevant?
  - Having 1,000,000 data files in a single directory.
  - Having separate runs of a program write output to separate files.
  - Reading a data file in once at program initiation, and then keeping the data cached in memory.
  - Using the name of a file to encode program results.
  - Having a swarm of 1,000 jobs each use the same temporary directory in /scratch.

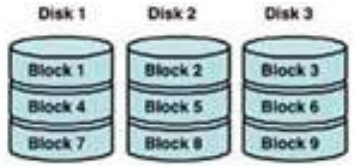
Interlude: A 30,000 foot look at RAID

- Most people are familiar with storing data on a single hard drive.
- But there are some problems…
  - Hard drives (particularly mechanical ones) are likely to fail
  - They also have limited performance
- Many years ago, some very smart people invented RAID (Redundant Array of Inexpensive Disks) to solve these problems.
- You as a user don’t need to understand anything about how it works – I am just mentioning the term since it’s useful for later.

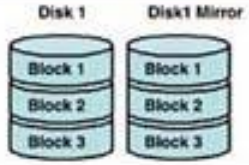
Different RAID levels

- Again – not necessary to understand these – for flavor
- Other levels (6, 10) relevant in many use-cases
- Allows combining many disks to create a much larger storage device

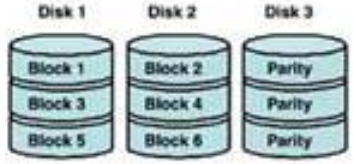
RAID 0 – Striping
- Data blocks written sequentially
- Not really RAID – no redundancy
- Higher performance than single disk access



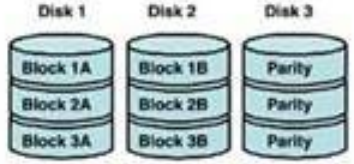
RAID 1 – Mirroring
- Data blocks written to both disks at once
- 100% redundancy with 100% additional capacity required
- Reads can be distributed across both disks to increase performance



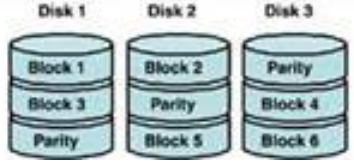
RAID 3 – Striping with Byte Parity
- Adds parity information to rebuild data in the event of disk failure
- High transfer rate and availability with lower capacity required than RAID 1
- Transaction performance low because all disks operate in lockstep



RAID 4 – Striping with Block Parity – Independently accessible disks
- Data blocks written sequentially to each disk in the array
- Parity still protects data from disk failure
- Dedicated parity disk is write bottleneck and leads to poor performance



RAID 5 – Striping with Rotational Parity
- Parity blocks written per row and distributed across all disks
- Parity distribution eliminates single write bottleneck
- Overhead for parity calculation on writes supplemented with parallel microprocessors or caching

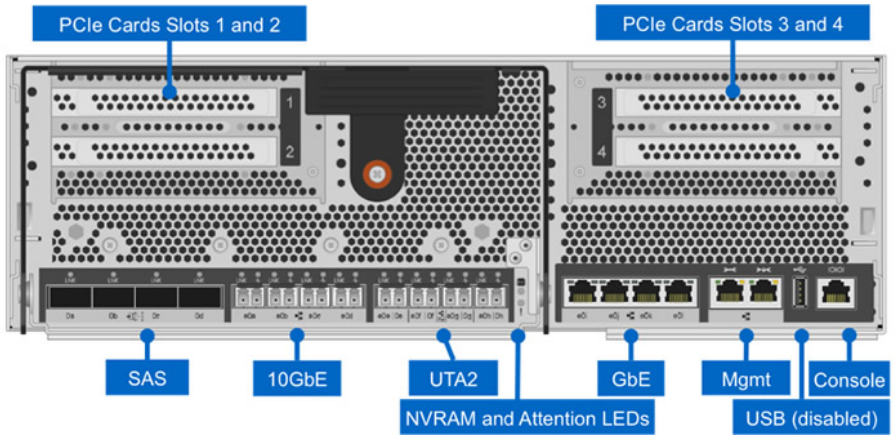


A deeper dive into storage components

- Storage controller
  - Specialized piece of hardware that routes requests between clients and the back-end disk system.
  - Has to understand two kinds of languages
    - The language of users: files, directories, abstract reads and writes
    - The language of hardware: disks, RAID/stripe groups, reading/writing individual blocks
    - Often includes special chips to speed up parity calculations needed for writes.
  - Often a bottleneck if many users hit a particular storage system
- Disk system
  - Back end that takes orders from the storage controller
  - Focuses on holding many individual hard drives in a compact amount of space

A look at a storage controller

- Contains fast network and disk connections
- Also a significant amount of CPU and memory power.
  - CPU is used to process storage requests.
  - Memory is used for caching various data and metadata.
  - Also contains application specific chips used for RAID parity calculations
- Exact nature of these controllers depends heavily on the design of the storage system.
- Some systems integrate the controllers with the disks – others (including the ones used on the HPC systems) separate them.
  - Both approaches are valid.



A look at a disk shelf

- Contains one or more I/O modules that provide access to individual disks.
- Usually all components for disk access are redundant.
- Mostly just provides a convenient (and plug-n-play) mechanism for packaging disks.
- Some systems have controller logic built-into the disk shelves.

Caches

- Definition: a cache is a region of higher-performance storage that sits in front of lower-performing, but larger storage.
  - Example: a small pool of SSDs sitting in front of a large rotational storage system.
- Often implemented in the memory of a computer system.
- Caches exist in many different places in the storage system – so it's helpful to understand how they behave.

Different kinds of caches

- Read
  - Data that is read is kept in the cache, in case it needs to be read again
  - Variation: read-ahead – system tries to predict future reads and put them into cache before they are actually requested
- Write-back
  - Data that is written is kept in the cache. The write is acknowledged as soon as the data is secure in the cache.
  - The system tries to write the data back to the underlying storage at some convenient time.
- Write-through
  - Data that is written is kept in the cache. However, the write is not acknowledged until the data is secure on the underlying (slow) storage.

A look at node-level caching

On this node, 47 GB is cached.

```
cn0012:~ 1$ free -m
             total       used       free     shared    buffers     cached
Mem:        129022      76686      52336          0        166      47260
-/+ buffers/cache:      29259      99763
Swap:         2047          7       2040
```

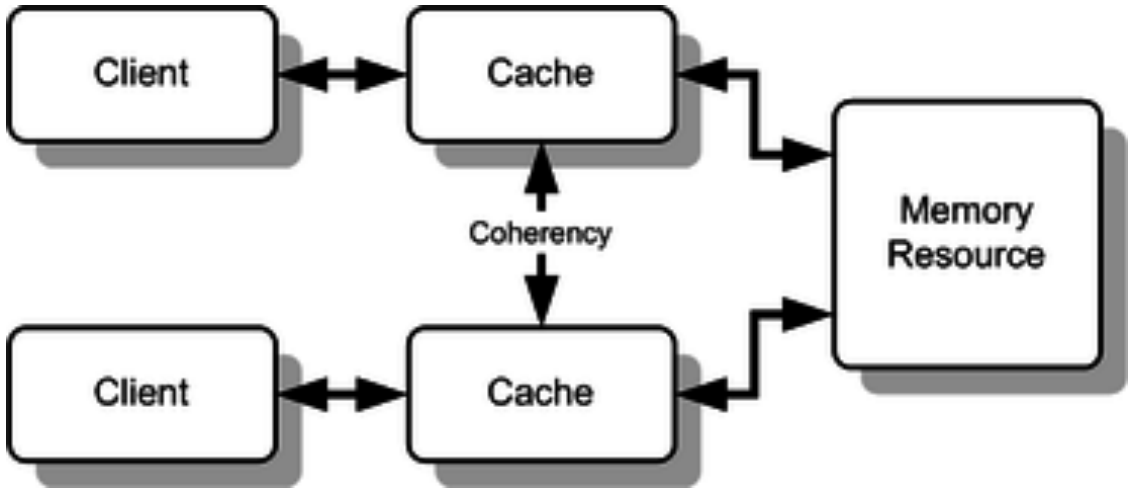
- The Linux kernel likes to cache data in memory
  - Any memory not being used by an application tends to get used for cache.
  - However, if the system is under memory pressure, the caches will get dropped and I/O performance may suffer.
- GPFS does not use the Linux kernel's cache
  - Uses a (potentially) smaller area (~ 2 GB) called the page pool
  - Unlike the kernel cache, the page pool is dedicated (but lots of I/O will exhaust it).

Where caches live in the storage hierarchy

- At all levels! (almost)
  - Nodes, storage controllers, disks
  - But NOT in the network
- Node level caches are (arguably) the most useful
  - Network storage is often a bottleneck
- Controller-level caches also very important
  - Prevents the controller from having to go back to the disks
  - Leads to faster acknowledgements of I/O operations

One pit-fall with caches

- Data consistency
  - If nodes cn0001 and cn0002 both have a copy of some data in cache, how are these kept in sync?
  - Problem also exists with metadata (and can be a lot worse – permissions changes not instantaneous!)
- Different storage systems solve this in different ways
  - But usually if lots of nodes “hit” the same data, there’s overhead to keep things in sync!



Points to remember regarding caches

- You don’t have any control over which data gets cached and which doesn’t.
- But if you’re doing lots of operations on a small amount of data, there’s a good chance that it will wind up in cache somewhere.
- Caches have potential pit-falls, especially in regards to seeing a consistent view of data between nodes.
  - Linux is very good at handling this
  - However, don’t expect content written on one node to be immediately visible from another
  - Having multiple processes write to the same file is <u>always</u> a bad idea (without some sophisticated coding or using locking [way beyond the scope of this class]).

Course overview/outline

- Overview of HPC systems storage
  - Different areas (/home, /data, /scratch, object store)
  - Quotas and quota increases
- Understanding input and output (I/O)
  - I/O patterns
  - Introduction to data and metadata
- HPC storage systems - under the hood and beyond the basics
  - Basics of storage system architecture (and what they mean for you as a user)
  - System components – storage servers and disks
  - Caches – local and remote
- Putting it all together – using storage effectively
  - Profiling and benchmarking your application's I/O usage
  - Understanding when you're generating too much I/O on the system

How do you know if you're abusing the storage systems?

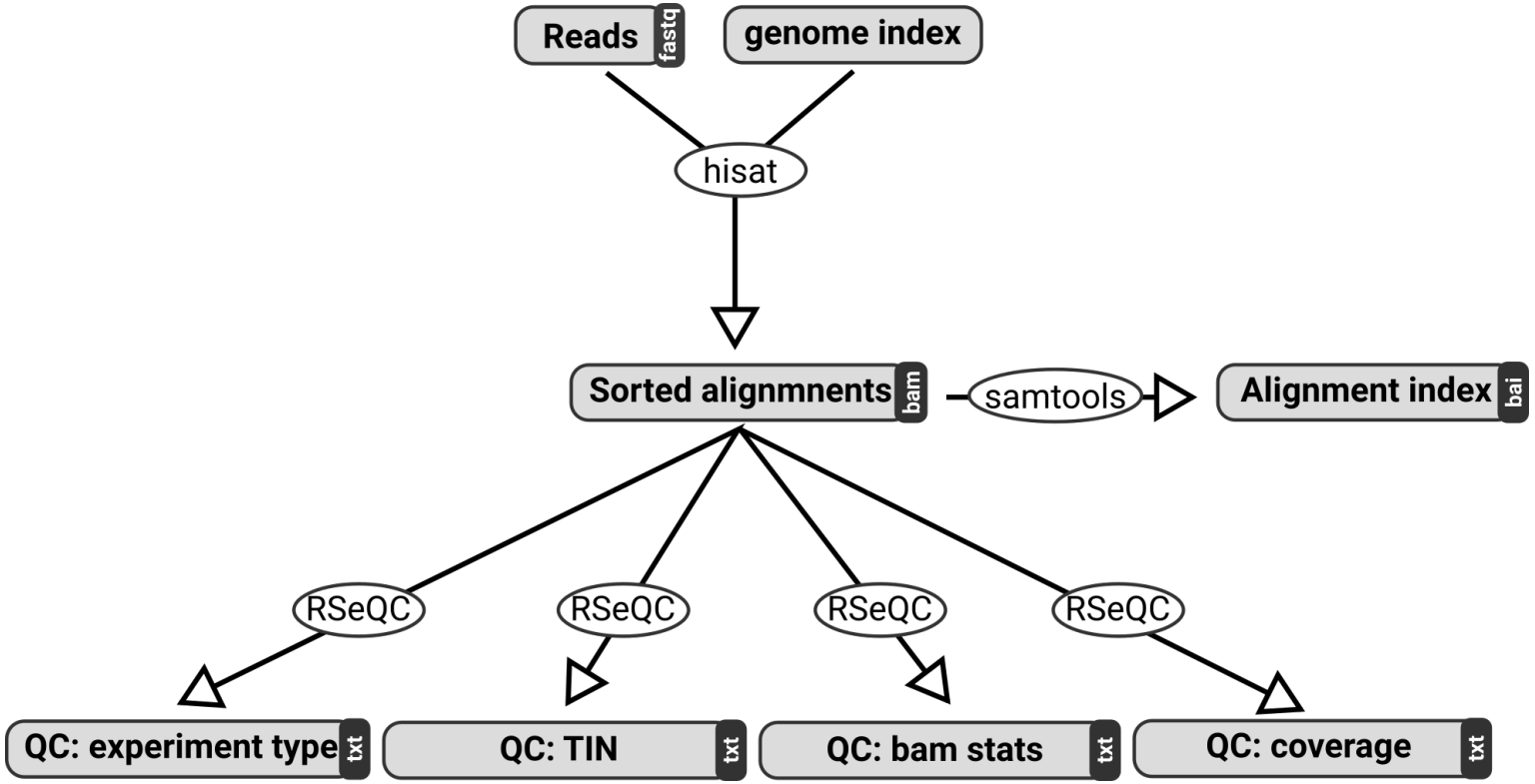
- This can be difficult to know, but there are a few clues
  - Very slow access to working directories involved with the job. (ls etc. take a long time to return)
  - For swarms, job completion speed was acceptable for small numbers of jobs, but gets dramatically slower as the size or number of jobs increases.
    - You did test with small-scale jobs first, right?!
  - The problem may be with another user's jobs, but if you started seeing problems right after you started a bunch of jobs, you are the prime suspect.
- HPC staff will notify you if we notice your jobs having an impact
  - However, please be proactive and don't wait for us to notice the problem
  - If we send you mail, it means you're having a significant negative impact on system performance.

Refactoring a workload – general principles

- Look for places where lots of parallel processes are doing I/O
  - Think about if only one process could do I/O and communicate with other processes (probably not possible with swarm).
  - Can some or all of that I/O use /lscratch instead of /scratch or /data?
- Think about bottlenecks in the workflow
  - E.g. the whole workload has to wait until one file is updated
  - Does the usage on a shared filesystem cause delays in this process?
- Does the workflow behavior change over time?
  - Do jobs have different I/O patterns in the beginning, middle, or end of their runs.
  - Would staggering this I/O be possible?

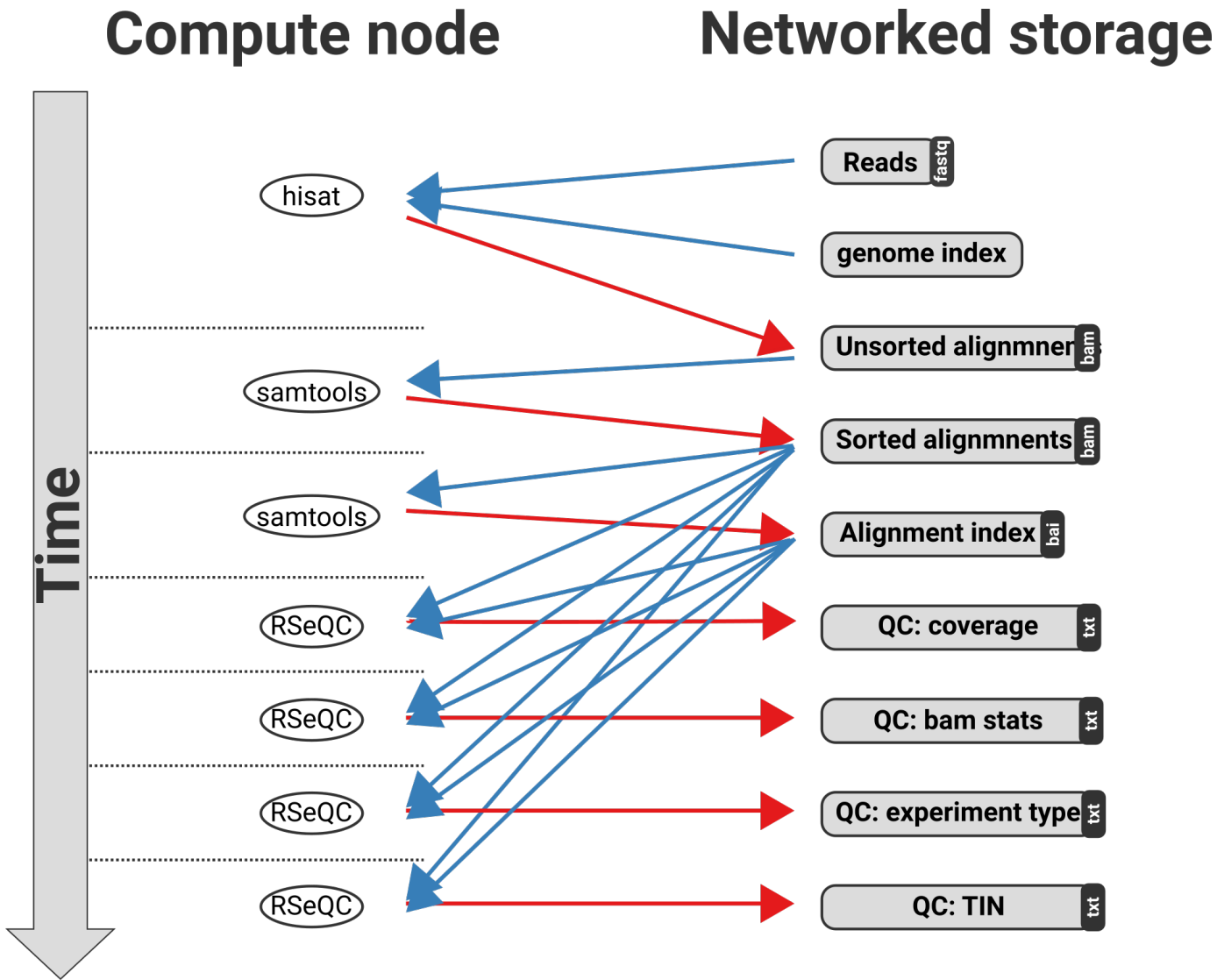
Workload refactoring: an example

- Start of a real-world genomics pipeline
  - Aligns sequences, creates index
  - RSeQC performs QC steps before continuing the pipeline
- Pipeline ran much more quickly after workload was refactored to use I/O more efficiently!



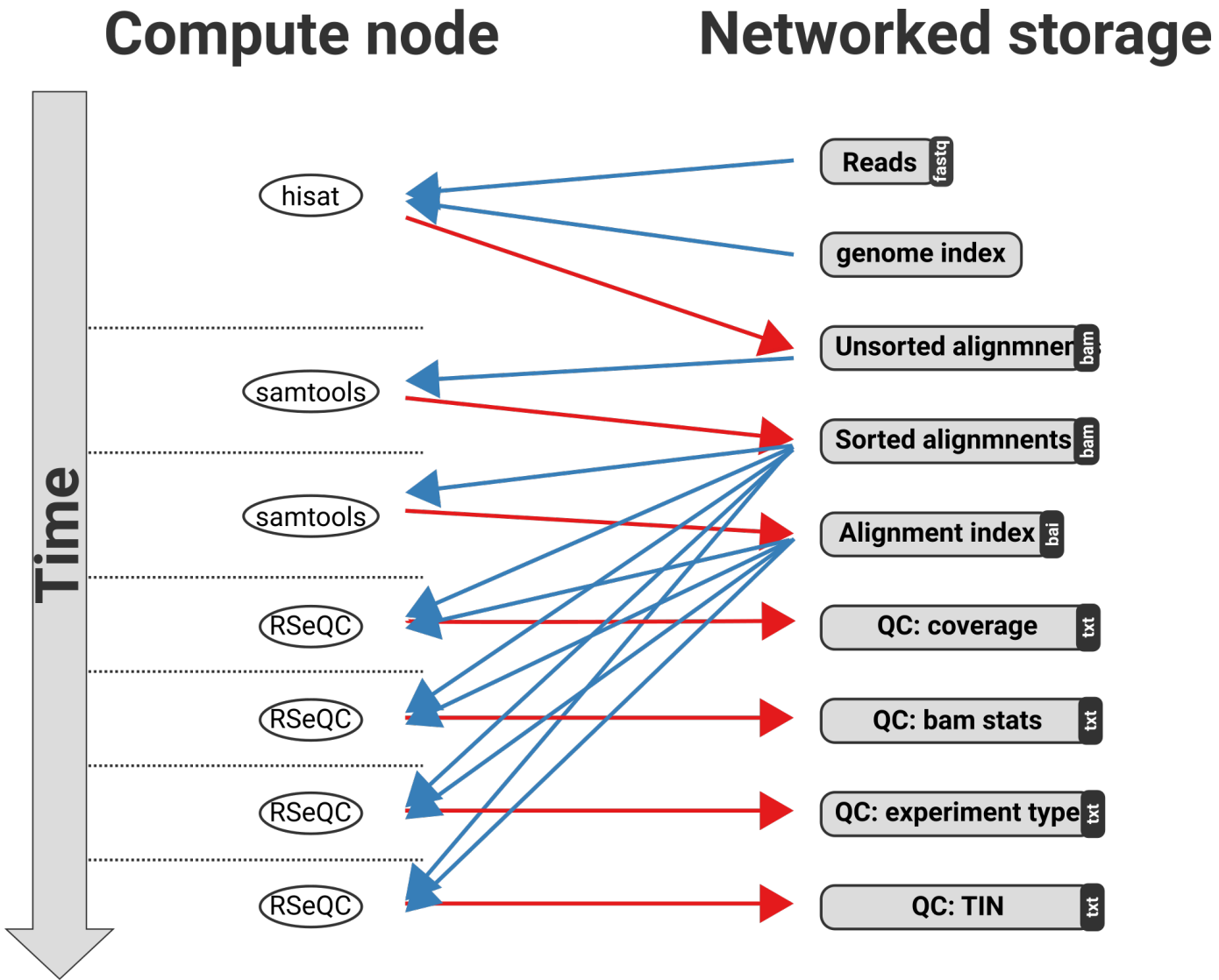
Original Pipeline

- What is wrong with this picture?



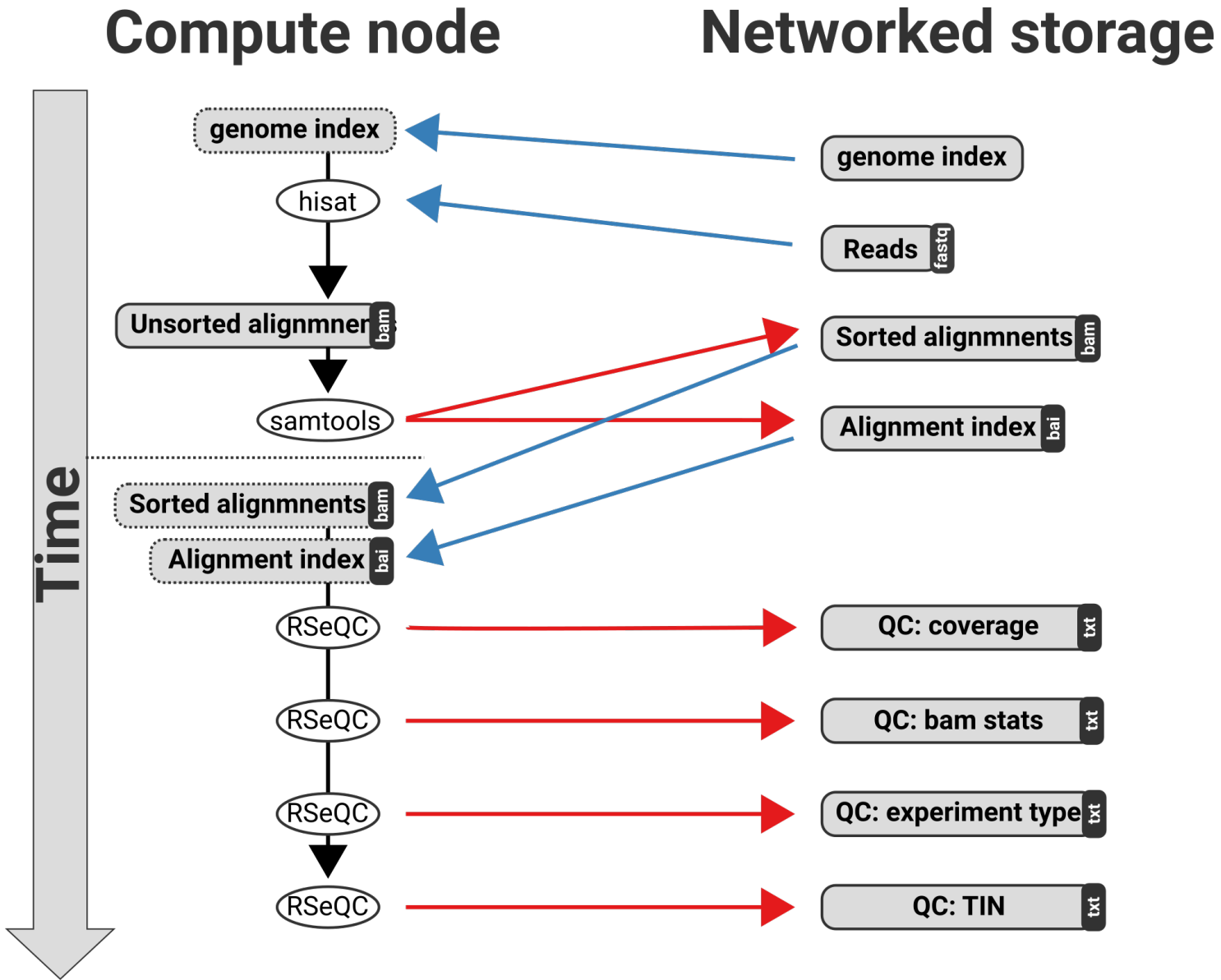
Original Pipeline

- Lots of read and writes back to network storage.
- Many parallel processes reading the same data from the storage system.
- Depending on the exact analysis done by RSeQC, random I/O is heavy.
  - Remember, lots of RSeQC processes in parallel.
- How would you fix this?



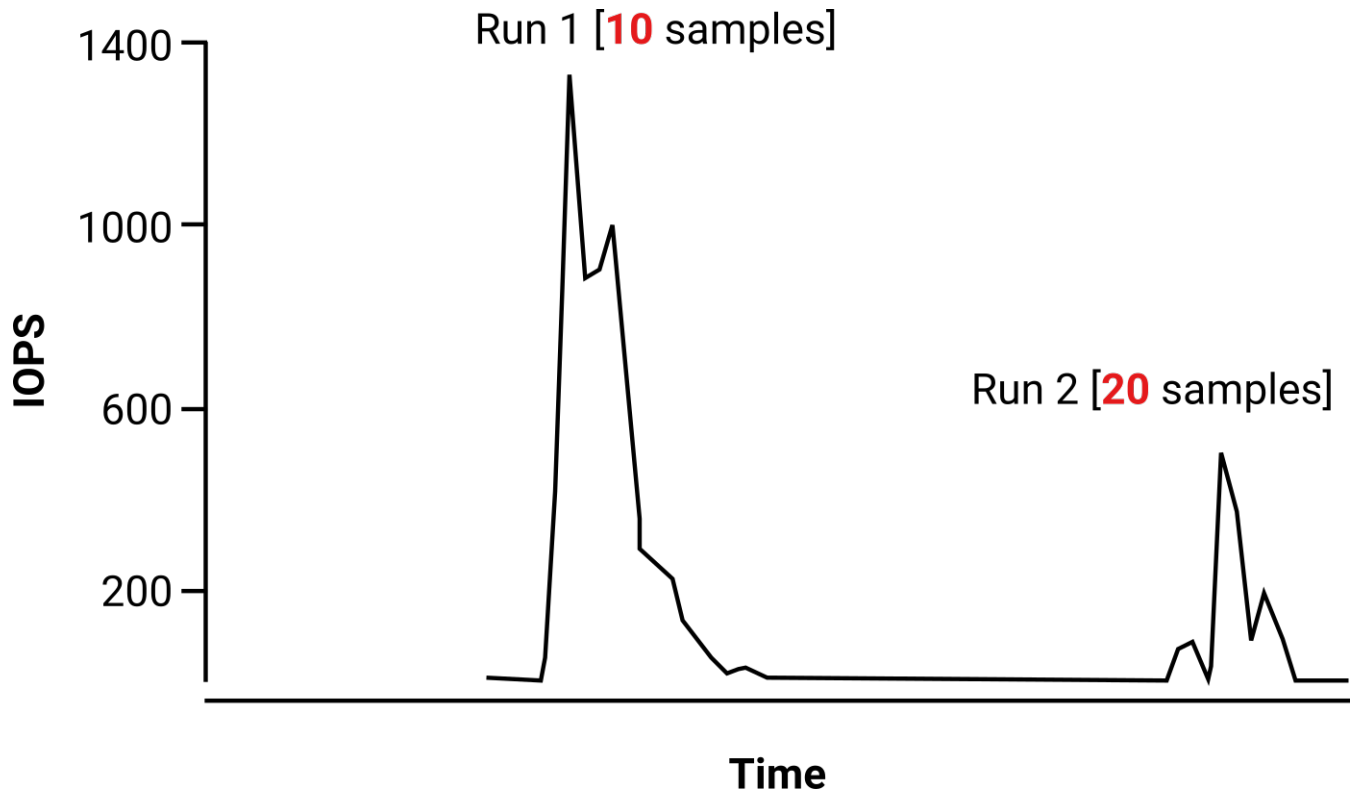
Refactored pipeline

- Only writes to networked storage when needed.
  - E.g. not after the initial read, only when index is built.
- Instead of each RSeQC reading the data from network storage, use local scratch
- Only write out final result files.



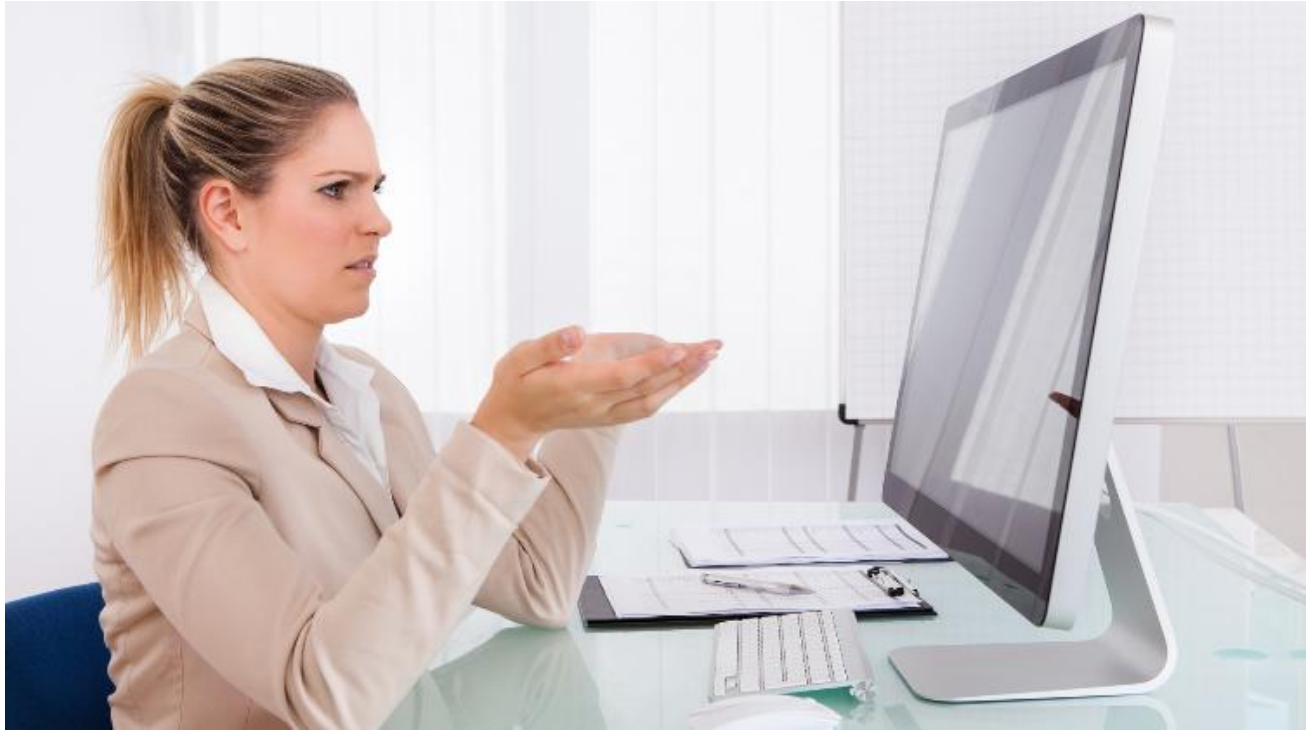
IOPS comparison

- Bottom line 1: User was able to do more science in less time.
- Bottom line 2: HPC storage admins did not have to troubleshoot performance problems.



Do you need to benchmark I/O usage?

- In most cases probably not
  - …unless your code is running more slowly than you like (or the staff have informed you of a problem)
  - ...and you think I/O might be a factor in that slowness
  - ...and you've already tried refactoring your workload to avoid bottlenecks.
- In this case it is **STRONGLY RECOMMENDED** that you discuss a benchmarking plan of action with the HPC staff!



Benchmarking I/O usage – the naïve way

- Note – Naïve does not mean ineffective!
- Sum up all files read and written by the application, e.g.:
  - Read 20 x 50 GB input files (1 TB total)
  - Write 1 x 100 GB output file (100 GB total)
  - Write and read 200 x 10 GB temp. files (2 TB x 2 = 4 TB total)
  - Total I/O requirement – 5.1 TB
- Divide by the run-time of your application to see throughput per second
  - If the job ran for 5 hours (18,000 seconds) the average I/O will be ~ 283 MB/sec.
- For swarms, remember to multiply by the number of sub-jobs in the swarm!

Exercise 5: try estimating I/O on your own

- Use the HPC user dashboard (https://hpc.nih.gov/dashboard.html) to find a recent job that you ran.
  - If you don't have a recent job, I can assign you one from one of the Biowulf class examples.
- Use standard *nix utilities (ls, du) to figure out how much I/O the job had generated.
  - Note, you might need to know something about how the application works to understand how it uses temporary files.
  - The point here is not to be totally accurate – it's to get a general feel for the quantity of I/O.
- Divide by the job's runtime to get average I/O usage.
- Think about some potential problems with this naïve estimation method.

Potential pitfalls with this approach

- Estimating can be extremely difficult, especially with temporary files
  - Hard to tell exactly how many temporary files are generated
  - Temporary files might get read (and re-written) multiple times!
- I/O usage is not (usually) consistent throughout the life-time of a job
  - Many jobs are heavy on I/O during start-up and shut-down phases
  - Temporary file usage might also not be continuous throughout the job
- Need to know something (or a lot) about how your application behaves in order to get the full picture.

Approach 2: Profiling I/O

- Only for very advanced users and developers
  - May require knowledge of how to compile code.
  - Interpreting results can be challenging
- TAU performance profiler: http://www.cs.uoregon.edu/research/tau/home.php
  - Can use library interposition (intercepting I/O calls)
  - Can also recompile code to get more detailed results
- PAPI: http://icl.utk.edu/papi/
  - Requires specially written code
  - Not installed on the HPC systems, but possibly could be.
  - Better suited for testing on local development workstations

Approach 2a: A much easier (but much more limited) profiling approach

- "pidstat" command in Linux will show you the I/O to local disk generated by your application
  - Local disk does not include GPFS or NFS filesystems
  - Only useful for profiling lscratch use (which is usually not a bottleneck)
  - Use an interactive node with screen or running in the background to collect the instrumentation yourself
- Example:

```
cn0123$ pidstat -d 10 120
```

Prints out KB/sec read and written every 10 seconds for 2 minutes.

TAU example

- “fastbayes” example from the Biowulf class.
- Let’s see how much I/O it generates!
- Have to call with tau_exec (load the “tau” module)
- Will produce a file - profile.0.0.0 (more files for parallel programs)

```
cn2686:/data/btmiller/biowulf-class/freebayes 19$ tau_exec -io -T serial freebayes --fasta-reference PyYM_genome.fasta N67.bam
```

Measure I/O

This is a serial (non-parallel) program

Output from tau example

- Process profile files with pprof (output mildly redacted)

```
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        3,689        3,784           1        3813    3784652 .TAU application
  2.4           89           89        3208           0         28 read()
  0.1            2            2         583           0          5 fwrite()
  0.1            2            2           4           0        581 fopen64()
  0.0        0.523        0.523           4           0        131 fclose()
  0.0        0.089        0.089          14           0          6 write()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
      3207  2.683E+06          1  9.775E+04  4.152E+05  Bytes Read
      3207  2.683E+06          1  9.775E+04  4.152E+05  Bytes Read : .TAU application => read()
        10  2.268E+04         10       4633       8941  Bytes Read <file=N67.bam>
      3197  2.683E+06          1  9.804E+04  4.158E+05  Bytes Read <file=PyYM_genome.fasta>
       597       7563          1      25.61        309  Bytes Written
       583       7563          1      25.45      312.7  Bytes Written : .TAU application => fwrite()
        14         34         27       32.5      1.763  Bytes Written : .TAU application => write()
       582        197          1       12.5      13.07  Bytes Written <file=stderr>
         1       7563       7563       7563          0  Bytes Written <file=stdout>
      2758  1.134E+04      0.383       6288       2264  Read Bandwidth (MB/s)
         5  1.134E+04      0.383       3451       4480  Read Bandwidth (MB/s) <file=N67.bam>
      2753       8191          1       6293       2255  Read Bandwidth (MB/s) <file=PyYM_genome.fasta>
       439      65.67          1      8.883      7.735  Write Bandwidth (MB/s)
        14         34      1.929      29.23       9.59  Write Bandwidth (MB/s) <file=PyYM_genome.fasta.fai>
         3       8191       2048       4778       2554  [GROUP=MAX_MARKER] Read Bandwidth (MB/s)
         2  1.134E+04        312       5826       5514  [GROUP=MAX_MARKER] Read Bandwidth (MB/s) <file=N67.bam>
         2  1.134E+04        312       5826       5514  [GROUP=MAX_MARKER] Read Bandwidth (MB/s) <file=N67.bam> : .TAU application => read()
         4         58       5.25      26.94      20.43  [GROUP=MAX_MARKER] Write Bandwidth (MB/s)
         2         32      10.33      21.17      10.83  [GROUP=MAX_MARKER] Write Bandwidth (MB/s) <file=PyYM_genome.fasta.fai>
---------------------------------------------------------------------------------------
```

Number of times invoked and execution time of I/O calls.

Breakdown of bandwidth, amount of data read or written per file and I/O call.

Final Notes on TAU

- Sampling profiler
  - Uses its own library routines to intercept and instrument I/O calls
  - Induces some overhead.
  - Does not sample **every** call; need to make sure you have enough samples for meaningful statistics.
- Can do <u>much</u> more than sampling I/O
  - But it's mostly of interest to software developers
- https://hpc.nih.gov/apps/tau.html

Approach 3: Collecting kernel level statistics

- HPC staff is developing tools that can intercept I/O calls (reads, writes, etc.) automatically without user intervention.
  - Has some overhead – more depending on how many calls are being made
  - We are thinking about the best way to deploy this.
  - This is done in the kernel (which is the core of the Linux operating system).
- Can provide more detailed information than user-space profilers
  - Aggregation of statistics over multiple, unrelated processes.
  - Gathering information on sequential vs. random I/O.
  - Gathering information on the sizes and latencies of requests.
- If you have a very difficult issue with your job's I/O, contact staff@hpc.nih.gov - we can consider this approach.

staff@hpc.nih.gov

Steve Bailey

Steven Fellini, Ph.D.

Susan Chacko, Ph.D.

David Godlove, Ph.D.

David Hoover, Ph.D.

Picture unavailable

Patsy Jones

Picture unavailable

Charles Lehr

Jean Mao, Ph.D.

Tim Miller

Sandy Orlow, Ph.D.

Charlene Osborn

Mark Patkus

Dan Reisman

Wolfgang Resch, Ph.D.

Rick Troxel

Sylvia Wilkerson

Wrap-up; Q&A

- Thank you for coming!
  - We hope you are able to apply the lessons learned to your own particular storage issues.
  - PLEASE reach out to staff@hpc.nih.gov for assistance; we'd love to work with you **proactively** instead of **reactively**.
- Any further questions? Discussion of particular problems?
- Please provide feedback on this course!
  - E-mail Tim btmiller@helix.nih.gov
  - E-mail Mark patkus@helix.nih.gov
  - General questions staff@hpc.nih.gov