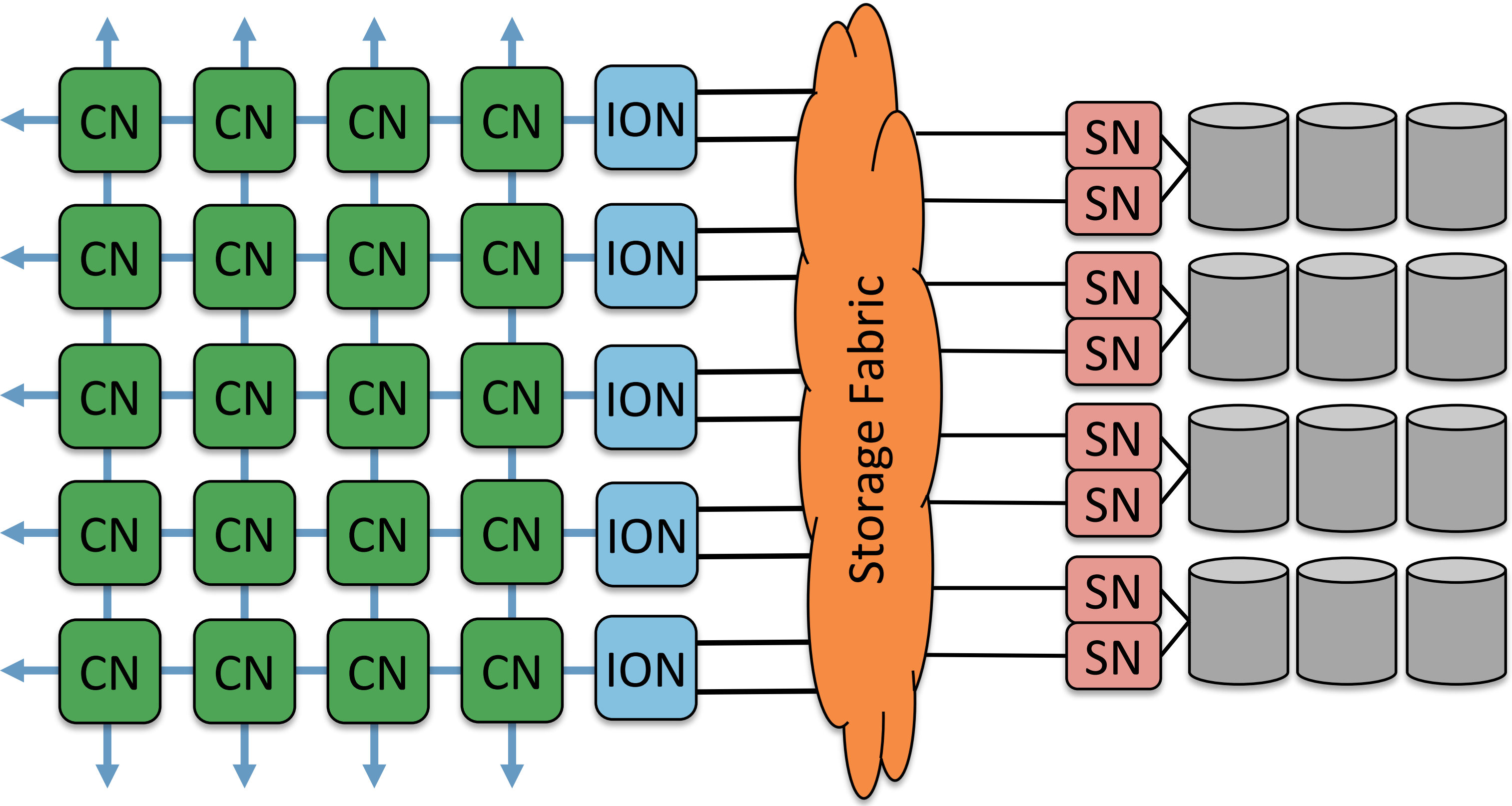
I/O Architectures and Technology

ATPESC 2019

Glenn K. Lockwood
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

Q Center, St. Charles, IL (USA)
July 28 – August 9, 2019

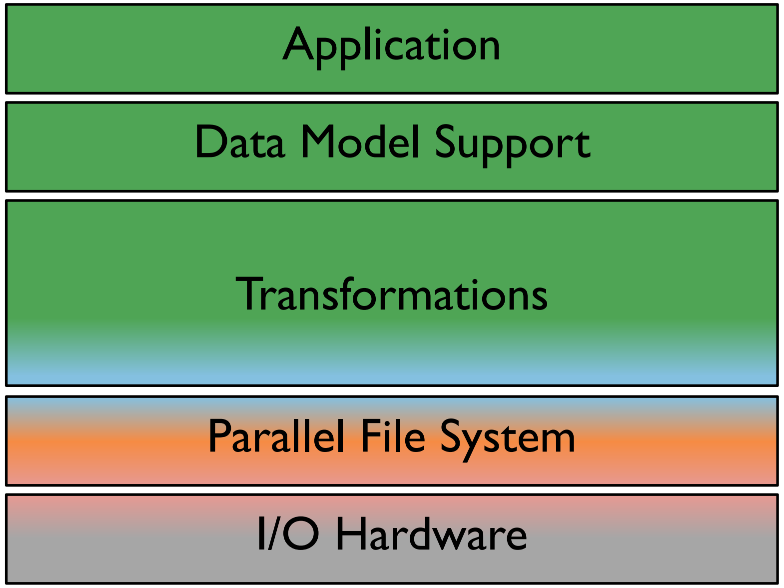
The Archetypal Parallel Storage System

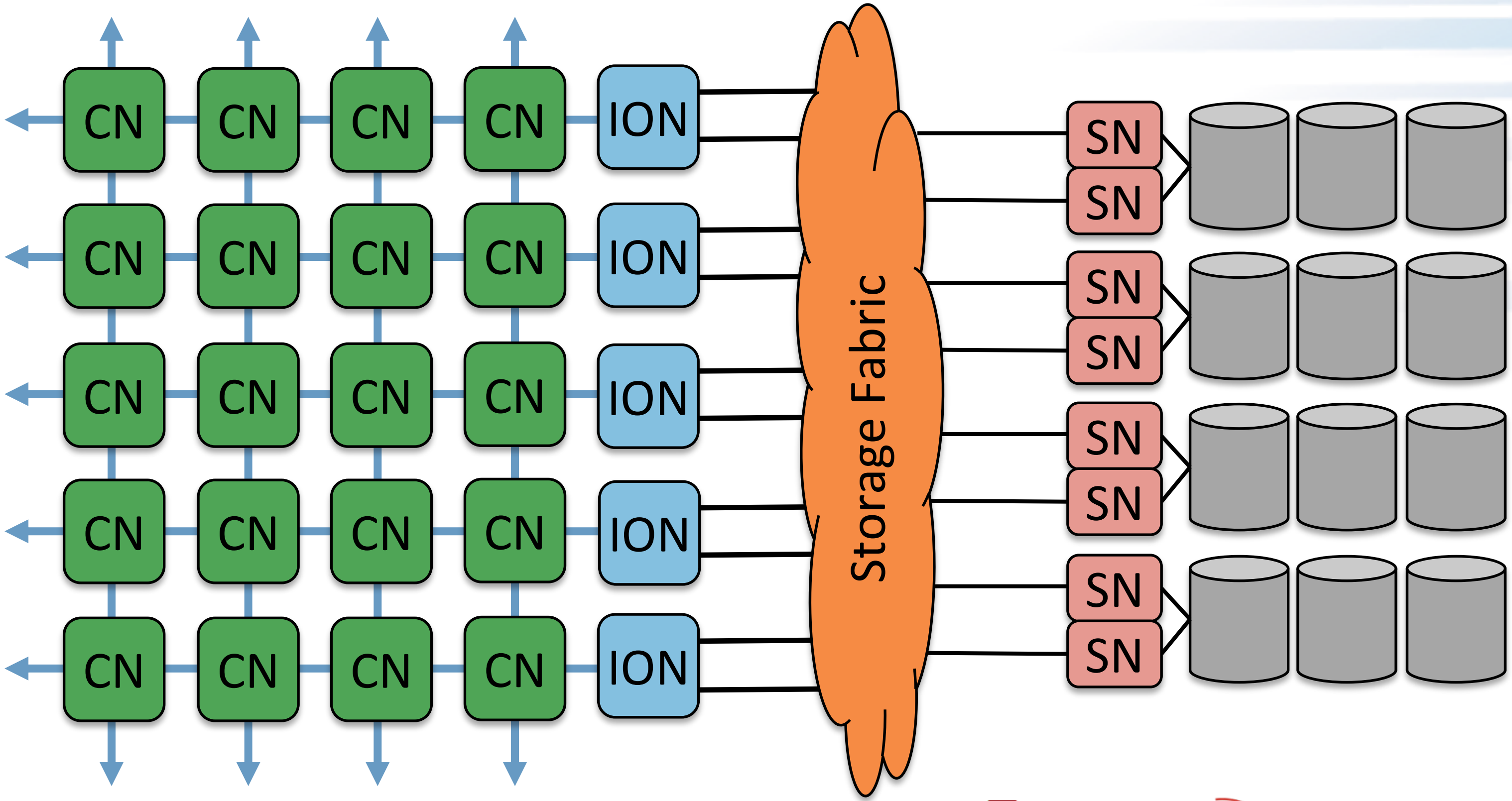


- **I/O Nodes**
  - I/O forwarding (CIOD, DVS, LNet)
  - May provide buffering/caching
- **Storage Fabric**
  - Carries file system or block protocols
  - InfiniBand, Ethernet, Fibre Channel
  - NFS, NSD, LNet; SCSI, NVMe
- **Storage Nodes**
  - Converts file system protocols to block protocols
  - Moderates permissions, file layout
  - Lustre OSSes, GPFS NSD servers
- **Storage Arrays**
  - Adds parity to data (RAID)
  - Makes many small drives (HDDs) look like one big drive (LUN)
  - DDN SFA, NetApp E-series

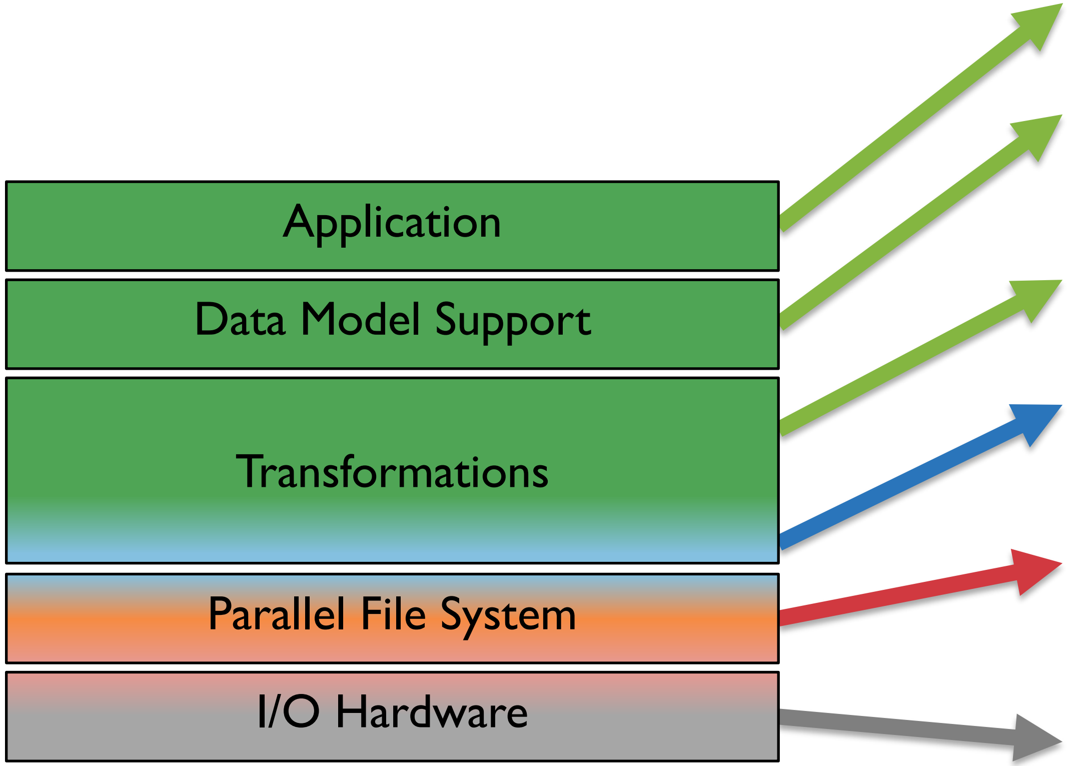
Systems are very different, but the APIs you use shouldn't be

- Understanding performance is easier when you know what's behind the API
- What really happens when you read or write some data?





Systems are very different, but the APIs you use shouldn't be



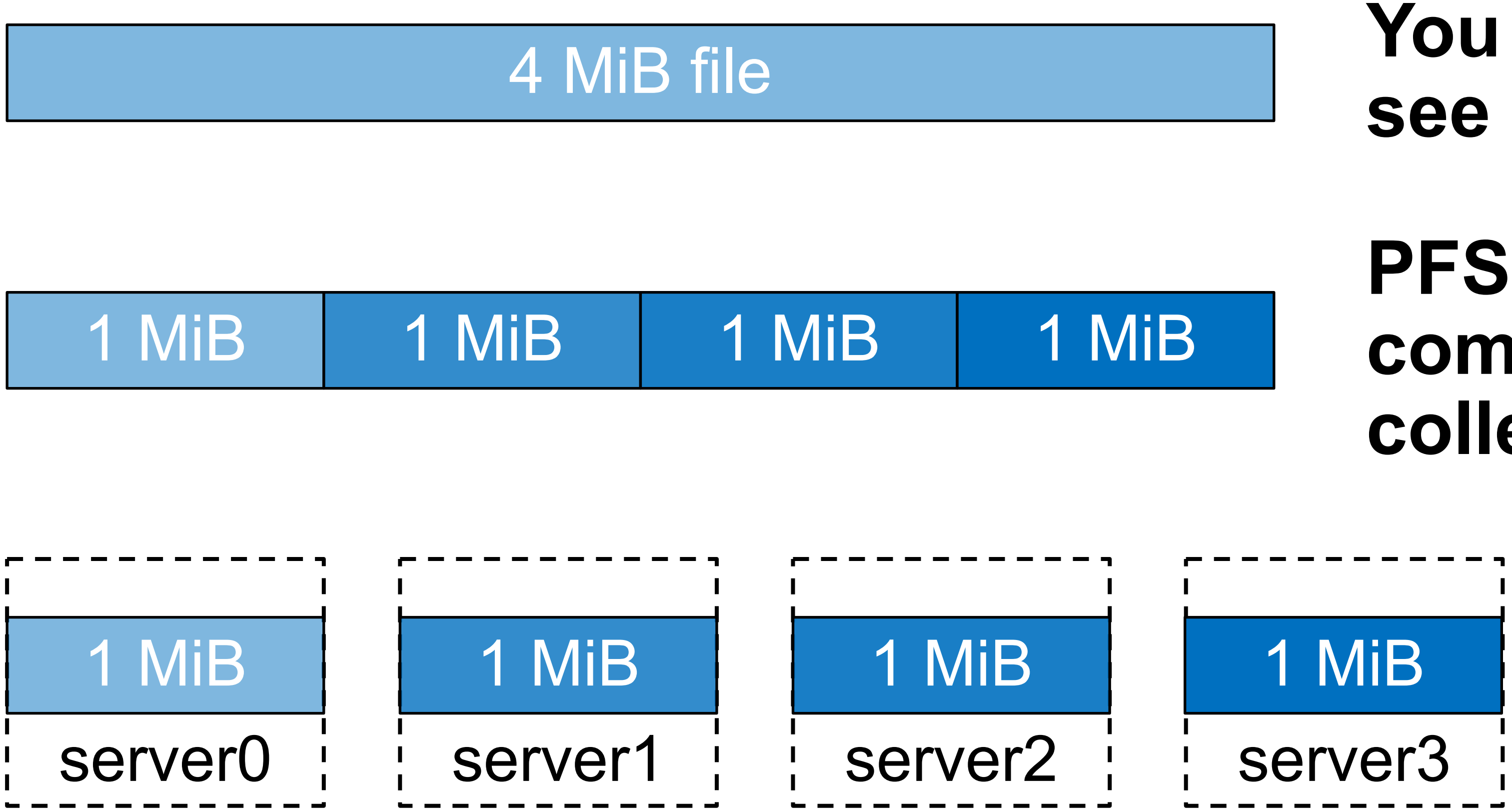
- You are the master of this
- Rob and Quincey presented PnetCDF and HDF5
- Rob presented MPI-IO
- Burst buffers and I/O forwarders: hardware-assisted transformation
- Parallel file systems: architectural performance quirks
- Storage hardware:
  - HDD performance quirks
  - SSD performance quirks
  - Parity's impact on performance

Parallel File Systems

Cray Sonexion 2000 (ClusterStor 9000)
248 Lustre OSSes / 10,168 4TB HDDs / 30 PB / 700 GB/sec

Parallel file systems in principle

File system that spreads files across multiple servers (∴ many NICs and drives)

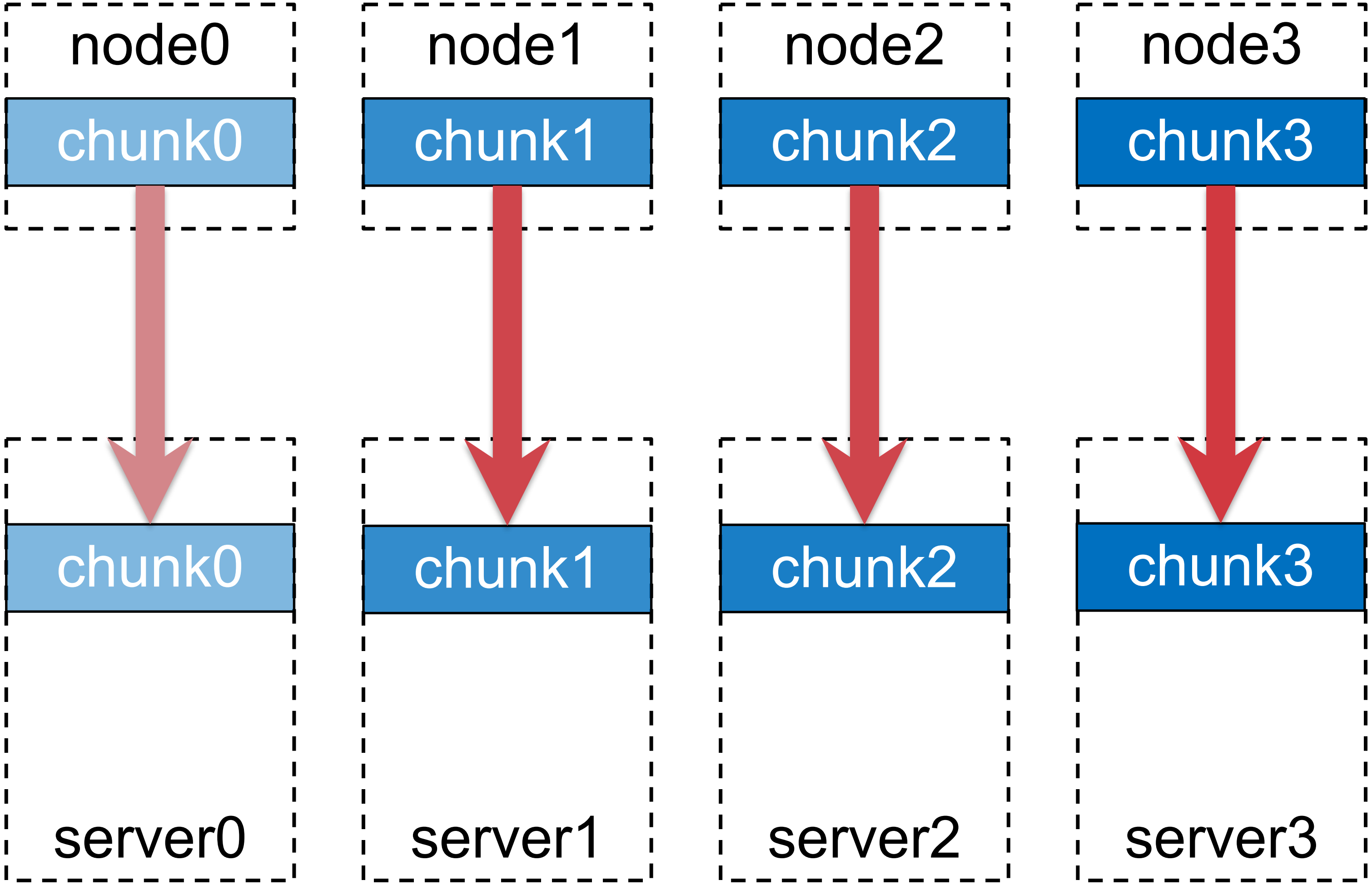


You and your application see one big file

PFS driver on your compute nodes see a collection of chunks

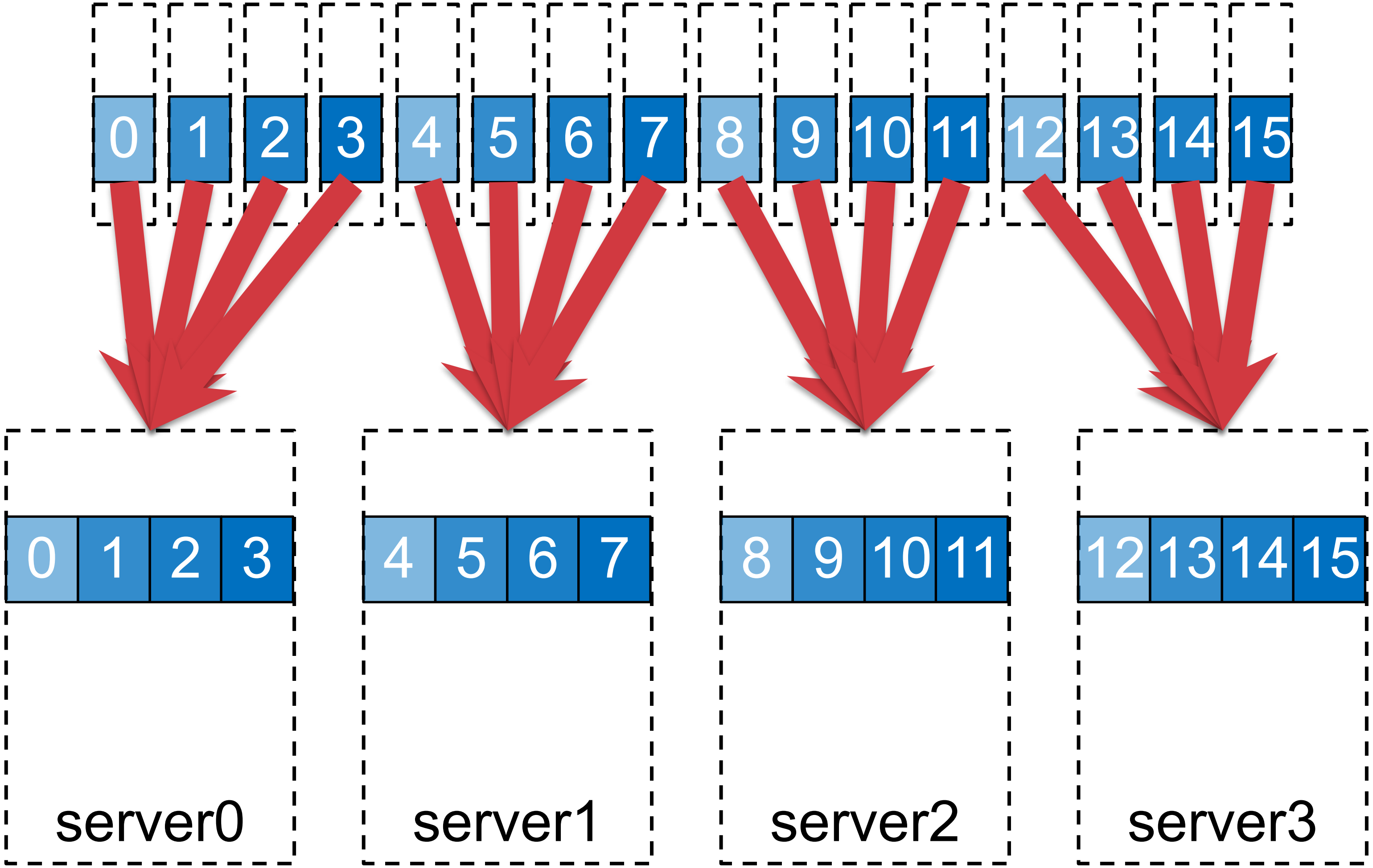
PFS servers see individual chunks

Parallel performance advantages of parallel file systems



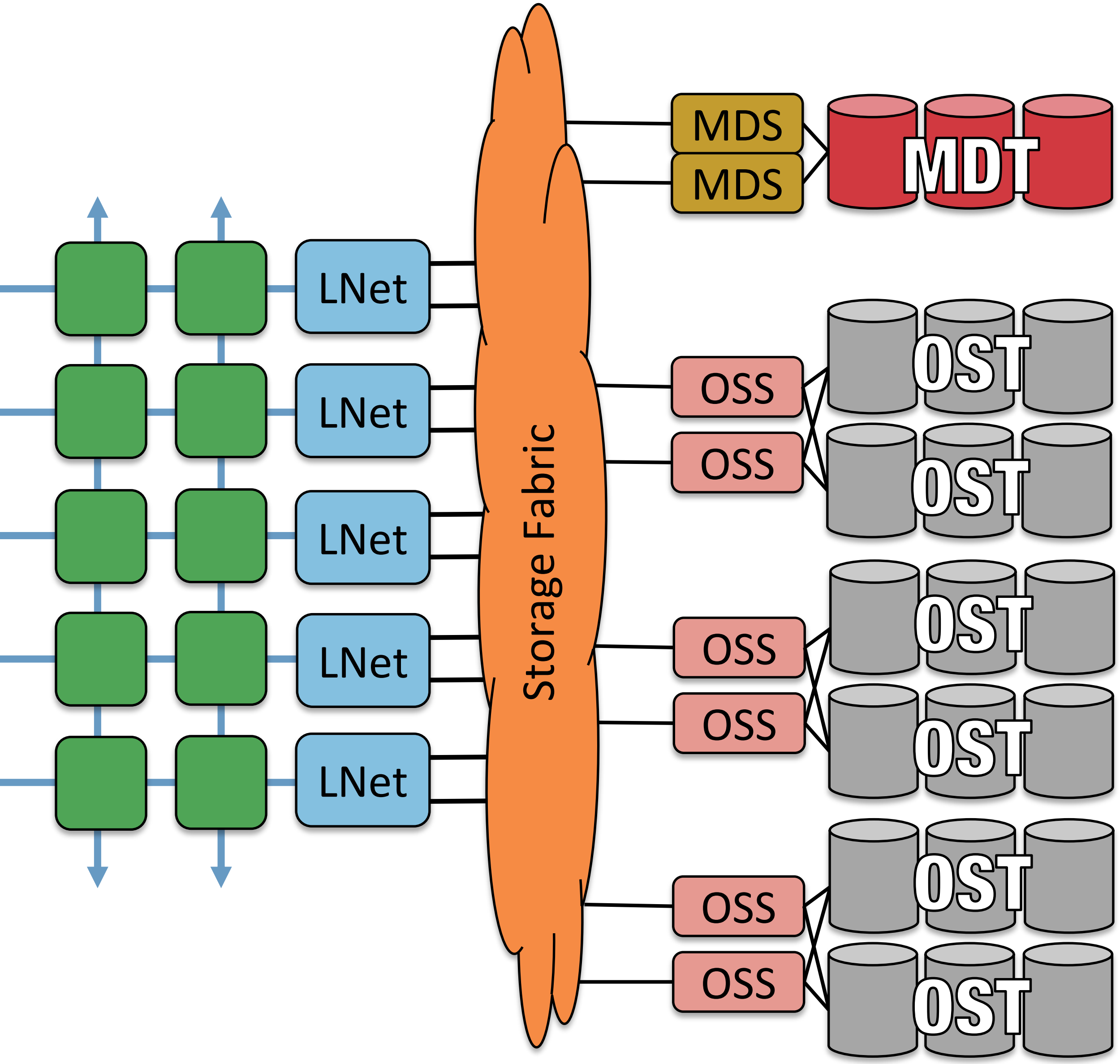
- Nodes and servers can read/write concurrently
- Avoid having to send all data to rank0

Scalability advantages of parallel file systems



- Typically scale compute faster than storage
- Parallel I/O required to scale out to extreme node counts and memory sizes

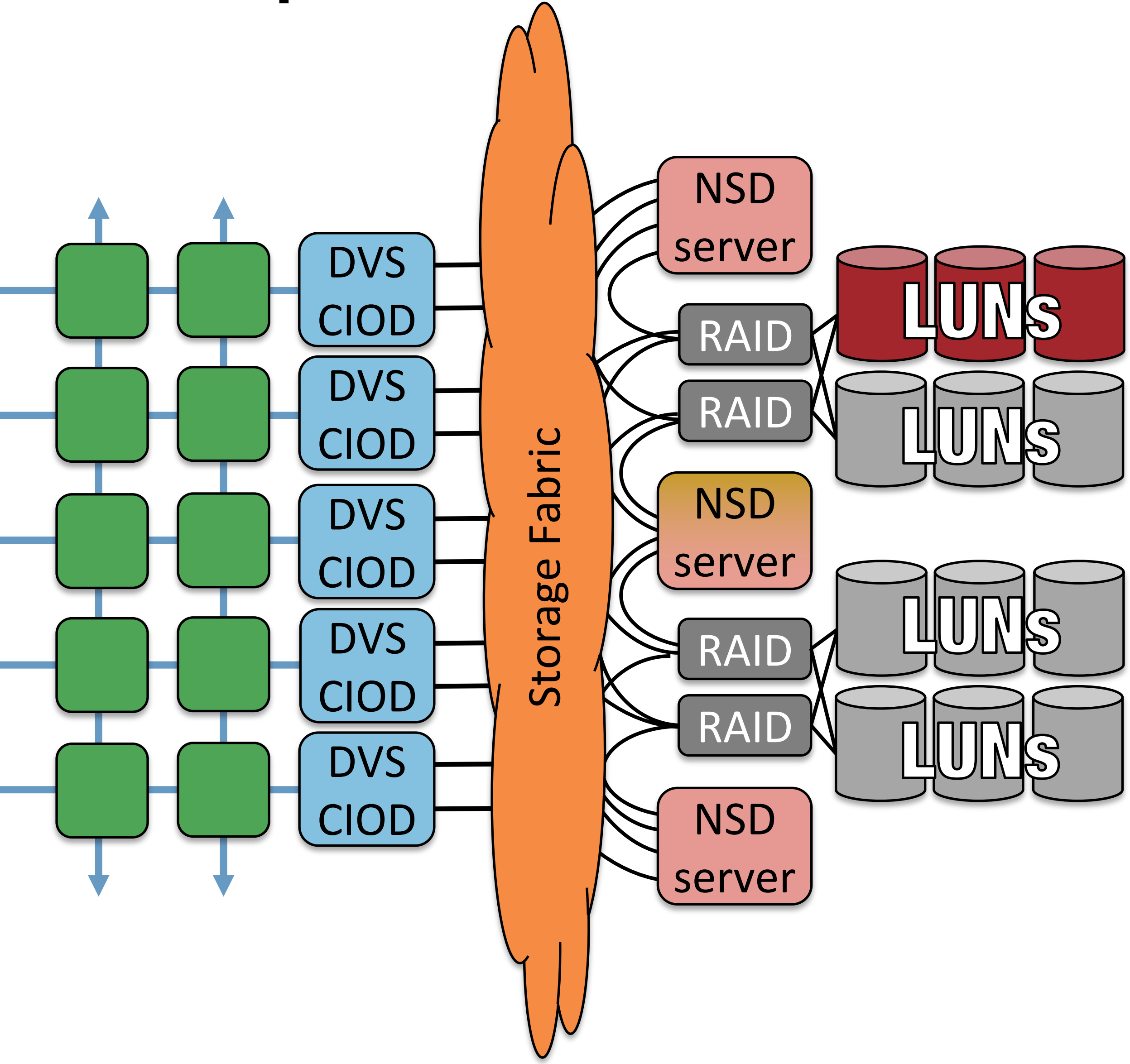
Lustre



Key features

- Metadata and data handled by separate servers ("metadata servers" "object storage servers")
- One file can be striped across many "object storage targets"
  - You choose stripe width(s) and size
  - Striping can vary between files
- Optimized for bandwidth
  - Small, random I/Os do not work well
  - High metadata rates (opens, unlinks) suffer
- 1 MiB is optimal minimum I/O size
  - `lfs getstripe` – interrogate striping of a file
  - `lfs setstripe -c` – set the striping of a file

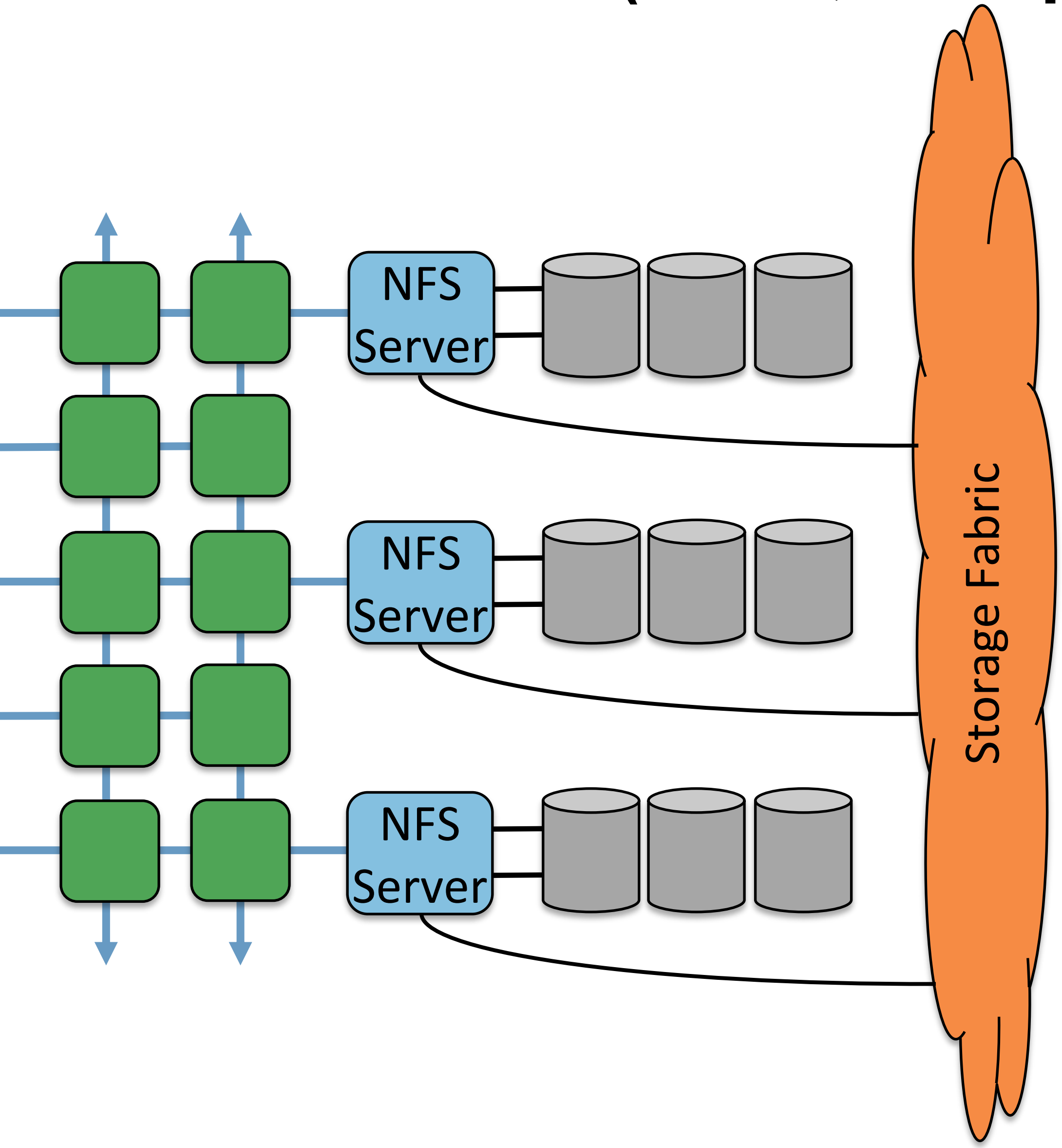
IBM Spectrum Scale



Key features

- Data and metadata can be combined
  - LUNs can store data or metadata
  - NSD servers can serve data LUNs and/or metadata LUNs
- One file's blocks are striped across many data LUNs
  - You cannot choose block size
  - You cannot choose where blocks land
- Fully distributed architecture
  - Many design options; few generic tips
  - Avoid using many files in a single directory
- 4 MiB often optimal minimum I/O size

Clustered NFS (Isilon, NetApp, etc)



Key features

- Highly localized: each server manages its own data and metadata
- File access is serial
  - One file = one server = one data path
  - Accessing file from a server that doesn't "own" that file triggers a back-end data transfer
- Optimized for convenience
  - NFS protocol is ubiquitous
  - Can corrupt data on parallel file access!
- Some design tricks can make this perform very fast

I/O Hardware

Seagate Exos E 4U106
106x14 TB SAS JBOD

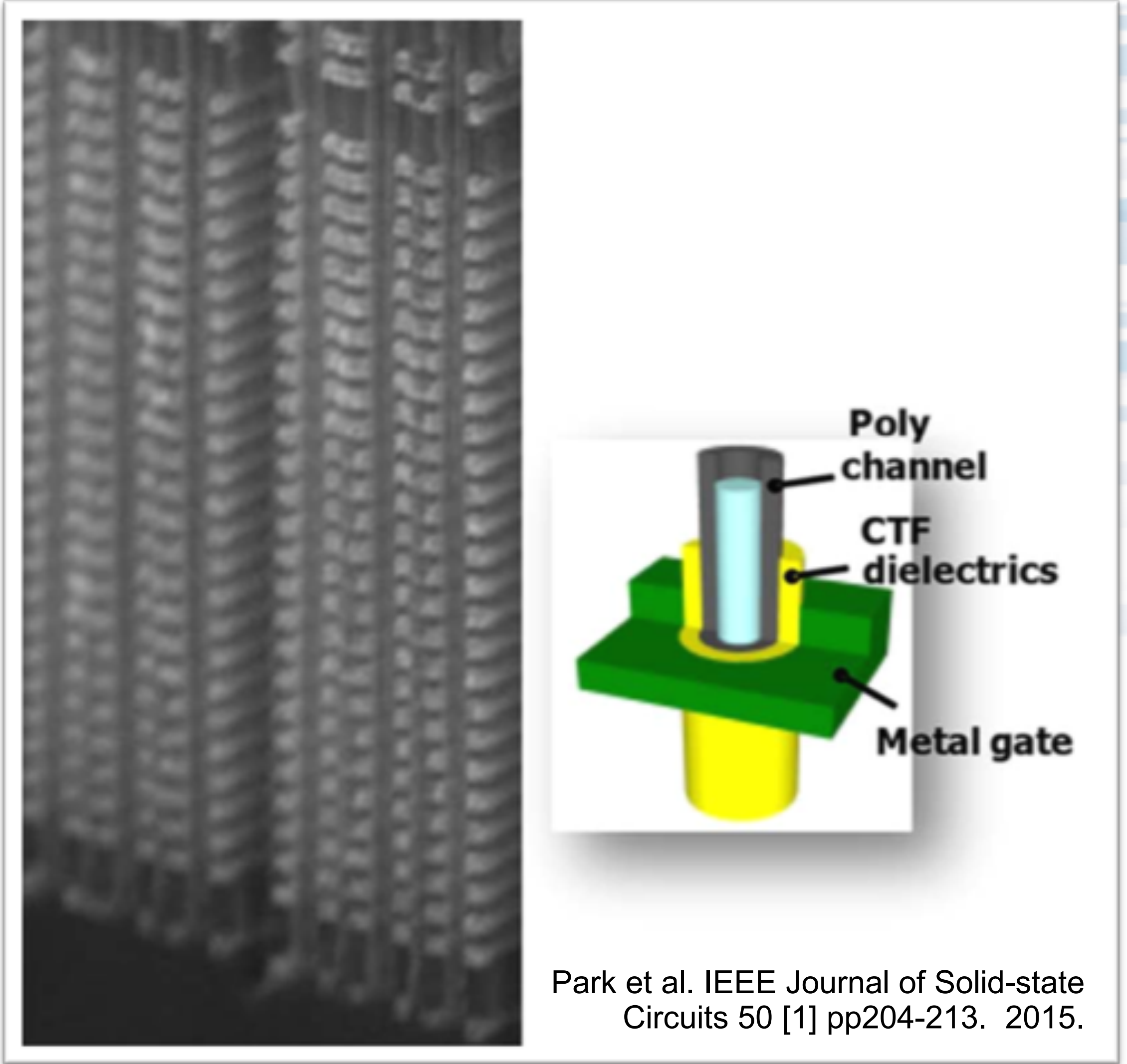
Mellanox SX6536
648-port FDR InfiniBand Switch

Hard Drives

- **Mechanics**
  - Platters spin at 7.2K or 10K RPM
  - One spindle, one actuator
  - Polarity of magnetic grains + run-length limited coding to encode bits
  - Magnetic read/write heads fly ~3 nm above platter surface
- **Performance**
  - Repositioning (random I/O) takes a "long" time (vs. sequential I/O)
  - Sequential bandwidth ∝ √areal density
    - Bit density not increasing quickly anymore
    - add platters instead
  - IOPS not going up at all
    - short stroke
    - 2nd actuator

Solid-state drives

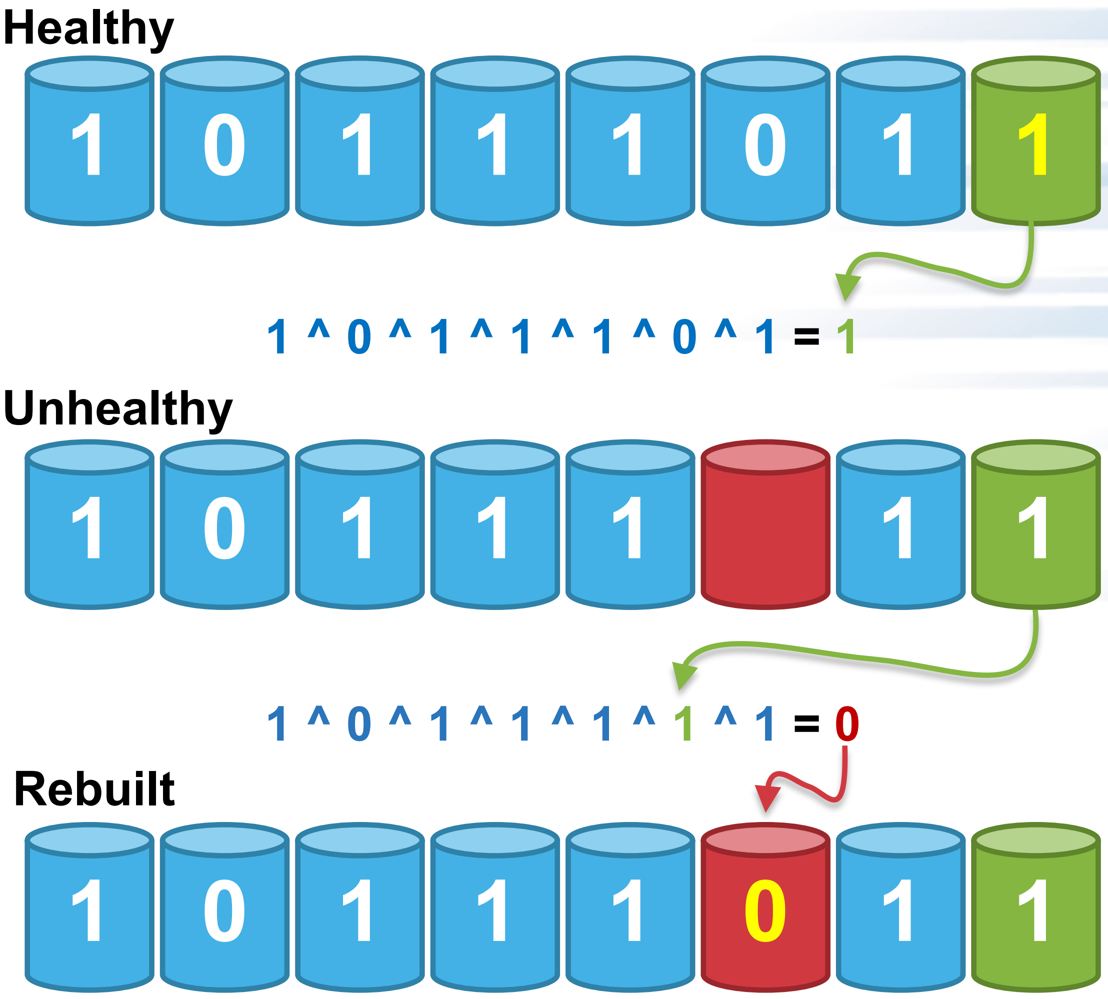
- **Mechanics**
  - Trap electrons inside a cell surrounded by insulator
  - SSD ∋ chips ∋ dies ∋ planes ∋ blocks ∋ pages – highly parallel internals
  - Programs in pages (2K-8K) but erase in blocks (128K – 2M)
  - FTL constantly repacks/recycles blocks
- **Performance**
  - Reduce GC for best performance
    - Align or buffer small I/Os
    - Big I/Os are still better than small
    - Write cliff and jitter are inevitable
  - Deep queues required to fill all parallel channels
    - issue I/O from multiple threads
    - more CPU often needed to drive I/O



Park et al. IEEE Journal of Solid-state Circuits 50 [1] pp204-213. 2015.

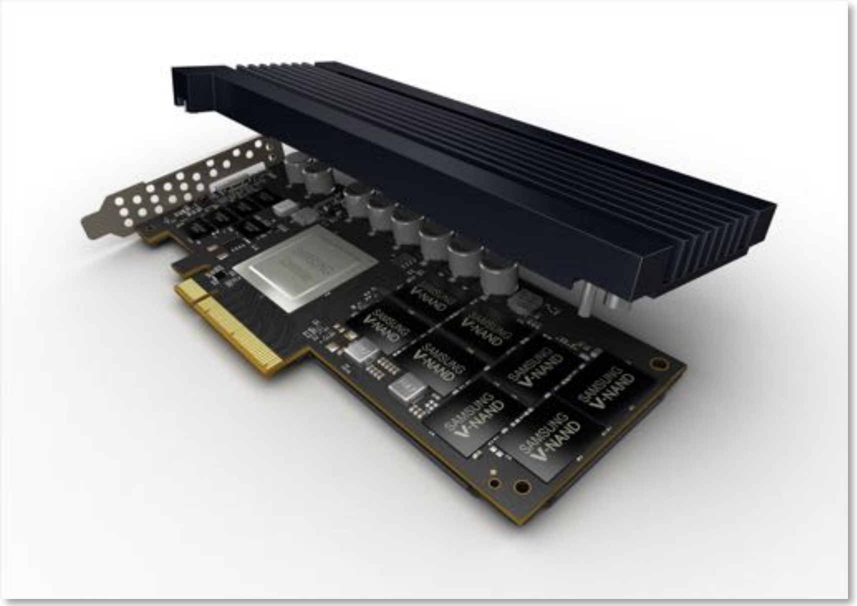
Redundant Array of Independent Disks (RAID)

- **Mechanics**
  - Split data into a stripe composed of N blocks
  - XOR each block and store result on N+1 parity block
  - If a block is lost, XOR remaining blocks and parity to recover lost block
- **Performance**
  - Aligning writes to stripes is critical – otherwise, a partial-stripe write causes
    - a read (whole stripe*)
    - a modify (update stripe and calculate new parity)
    - a write (new data + new parity)
  - Replication used when IOPS are critical
  - Involved in many perf issues in practice
    - Rebuilding a failed disk slows down parallel I/O
    - Parity checks on read slow down all I/O



* Not true since XOR is associative + commutative; can do (old block ^ new block ^ old parity) (thanks Phil!)
https://github.com/glennklockwood/io-algorithms/blob/master/raid.py

Hardware-assisted transformation: Burst buffer architectures



Samsung PM1725a NVMe SSD
Source: Samsung
https://news.samsung.com/medialibrary/global/photo/12105?album=27

NERSC Cori / Cray XC-30

Motivation for Burst Buffers

	Tape	Hard disk drive	Solid-state drive
Sequential reads and writes	360 MB/sec	250 MB/sec	3,000 MB/sec
Random reads and writes	$O(10^{-3})$ ops/sec	$O(10^{2})$ ops/sec	$O(10^{6})$ ops/sec
Internal concurrency	$O(1)$	$O(10)$	$O(100)$
Cost (2019)	$O(\$10/\mathrm{TB})$	$O(\$30/\mathrm{TB})$	$O(\$100/\mathrm{TB})$

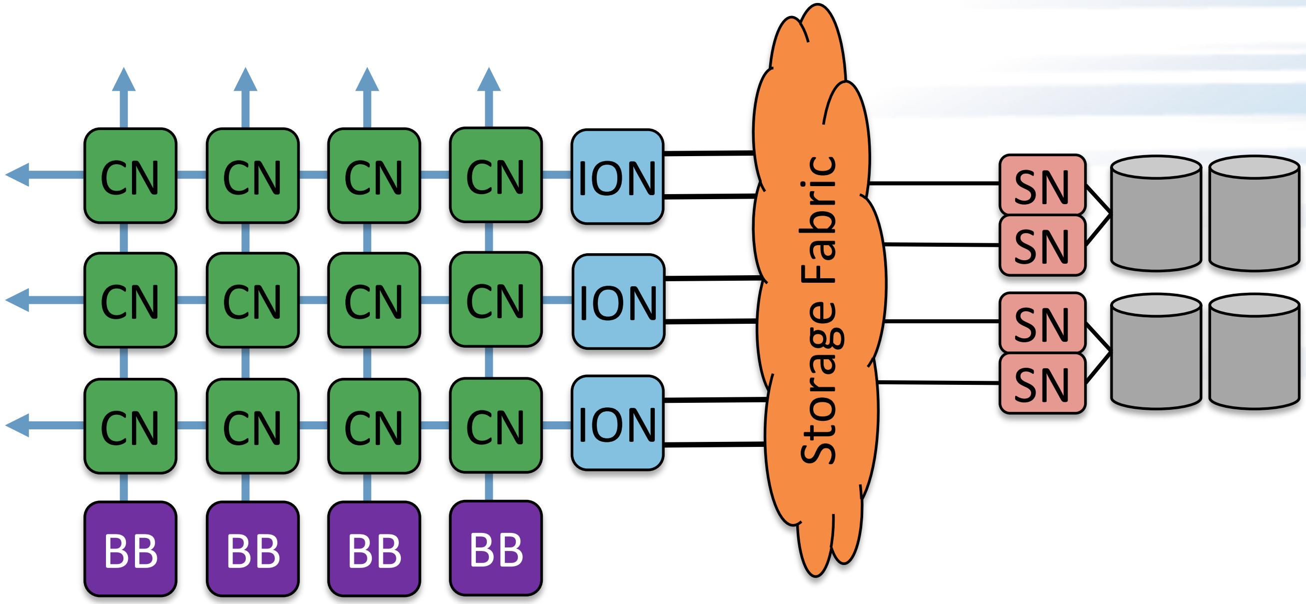
- SSDs are better for performance
- HDDs are better for capacity
- Use a little flash and a lot of disk to get the best of both worlds

Performance sources:

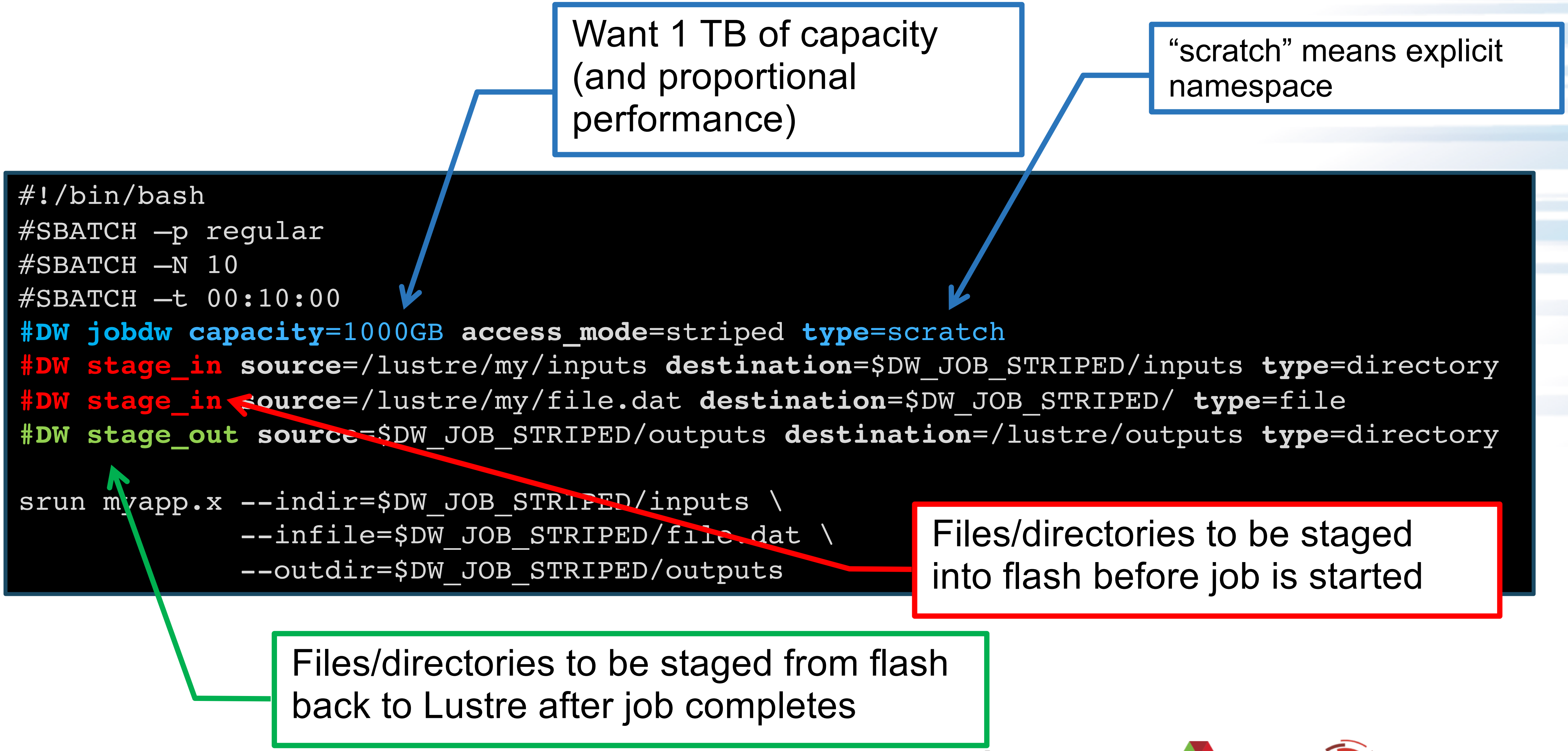
- IBM TS1155 data sheet (https://www.ibm.com/downloads/cas/AZGD8GMB)
- Seagate ST14000NM0048 data sheet (https://www.seagate.com/www-content/datasheets/pdfs/exos-x-14-channel-DS1974-4-1812US-en_US.pdf)
- Samsung 983DCT data sheet (https://www.samsung.com/semiconductor/global.semi.static/Data_Center_SSD_983_DCT.Product_Brief.pdf)

Burst buffers in practice

- **Burst buffers come in two use modes**
  1. **explicit** – separate namespace
  2. **transparent** – looks like the regular parallel file system but performs like all-flash
- **Burst buffer resources are scheduled**
  - request burst buffer in job script
  - data does not always remain after job completes
  - provide explicit, non-standard controls for staging data



Explicit burst buffers in practice: Slurm and DataWarp example



Want 1 TB of capacity (and proportional performance)

“scratch” means explicit namespace

```
#!/bin/bash
#SBATCH –p regular
#SBATCH –N 10
#SBATCH –t 00:10:00
#DW jobdw capacity=1000GB access_mode=striped type=scratch
#DW stage_in source=/lustre/my/inputs destination=$DW_JOB_STRIPED/inputs type=directory
#DW stage_in source=/lustre/my/file.dat destination=$DW_JOB_STRIPED/ type=file
#DW stage_out source=$DW_JOB_STRIPED/outputs destination=/lustre/outputs type=directory

srun myapp.x --indir=$DW_JOB_STRIPED/inputs \
             --infile=$DW_JOB_STRIPED/file.dat \
             --outdir=$DW_JOB_STRIPED/outputs
```

Files/directories to be staged into flash before job is started

Files/directories to be staged from flash back to Lustre after job completes

Caching burst buffers in practice: Slurm and DataWarp example

Want 1 TB of capacity (and proportional performance)

cf. "scratch" in previous example

```
#!/bin/bash
#SBATCH –p regular
#SBATCH –N 10
#SBATCH –t 00:10:00
#DW jobdw capacity=1000GB access_mode=striped type=cache pfs=/lustre/my

srun myapp.x --indir=$DW_JOB_STRIPED_CACHE/inputs \
             --infile=$DW_JOB_STRIPED_CACHE/file.dat \
             --outdir=$DW_JOB_STRIPED_CACHE/outputs
```

Directory to be mirrored into burst buffer

Inputs are read into flash on demand; outputs are flushed to Lustre on demand

Staging data in and out

Explicit Mode

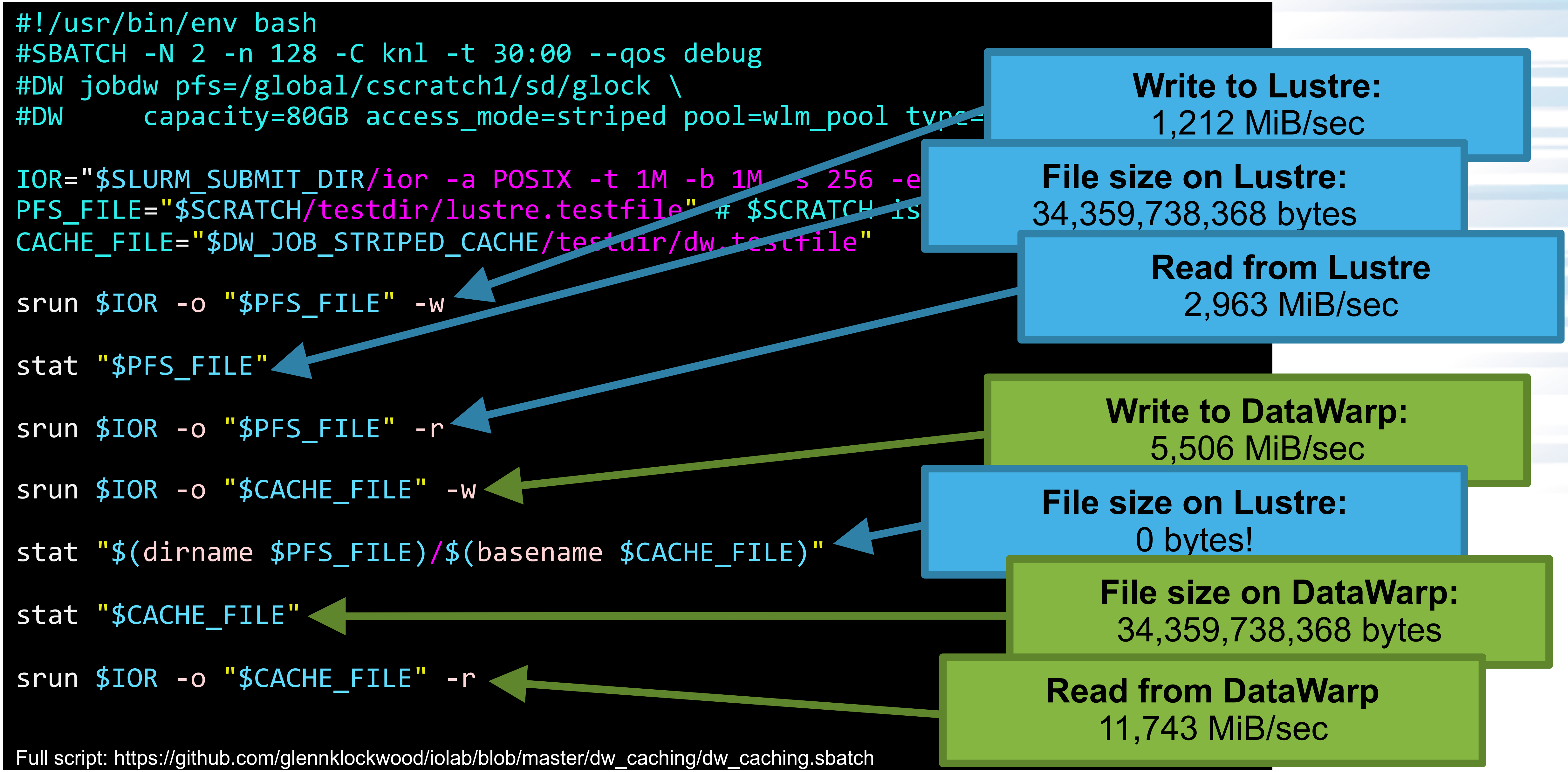
- Get your own private namespace
- Exceeding capacity request causes ENOSPC
- Explicitly define "hot" data to be available on flash before job starts
- Explicitly define data worth staging back to PFS after job completion
- If you don't mind managing your own staging for best performance

Caching Mode

- Looks like the regular PFS
- Exceeding capacity request causes stage out
- First read always comes from PFS
- All undeleted data is automatically staged out after job completion
- If you want better performance with minimal effort

Expert users can explicitly manage data staging in both cases

Both modes change data consistency behavior!



```
#!/usr/bin/env bash
#SBATCH -N 2 -n 128 -C knl -t 30:00 --qos debug
#DW jobdw pfs=/global/cscratch1/sd/glock \
#DW     capacity=80GB access_mode=striped pool=wlm_pool type=

IOR="$SLURM_SUBMIT_DIR/ior -a POSIX -t 1M -b 1M -s 256 -e
PFS_FILE="$SCRATCH/testdir/lustre.testfile" # $SCRATCH is
CACHE_FILE="$DW_JOB_STRIPED_CACHE/testdir/dw.testfile"

srun $IOR -o "$PFS_FILE" -w

stat "$PFS_FILE"

srun $IOR -o "$PFS_FILE" -r

srun $IOR -o "$CACHE_FILE" -w

stat "$(dirname $PFS_FILE)/$(basename $CACHE_FILE)"

stat "$CACHE_FILE"

srun $IOR -o "$CACHE_FILE" -r
```

Full script: https://github.com/glennklockwood/iolab/blob/master/dw_caching/dw_caching.sbatch

Write to Lustre:
1,212 MiB/sec

File size on Lustre:
34,359,738,368 bytes

Read from Lustre
2,963 MiB/sec

Write to DataWarp:
5,506 MiB/sec

File size on Lustre:
0 bytes!

File size on DataWarp:
34,359,738,368 bytes

Read from DataWarp
11,743 MiB/sec

```
#!/usr/bin/env bash
#SBATCH -N 2 -n 128 -C knl -t 30:00 --qos debug
#DW jobdw pfs=/global/cscratch1/sd/glock \
#DW     capacity=80GB access_mode=striped pool=wl

IOR="$SLURM_SUBMIT_DIR/ior -a POSIX -t 1M -b 1M -
PFS_FILE="$SCRATCH/testdir/lustre.testfile" # $SC
CACHE_FILE="$DW_JOB_STRIPED_CACHE/testdir/dw.test

srun $IOR -o "$PFS_FILE" -w

stat "$PFS_FILE"

srun $IOR -o "$PFS_FILE" -r

srun $IOR -o "$CACHE_FILE" -w

stat "$(dirname $PFS_FILE)/$(basename $CACHE_FILE

stat "$CACHE_FILE"

srun $IOR -o "$CACHE_FILE" -r
```

Full script: https://github.com/glennklockwood/iolab/blob/master/dw_caching/dw_caching.sbatch

Burst buffer take-aways

- Performance is typically better
- Your data is not necessarily "just there"
    - Be mindful of transparent caching (*implicit* data management)
    - *Explicit* data management adds some complexity

Architecture and performance take-aways

- **Systems are very different, but the APIs you use shouldn't be**
- **For POSIX I/O, underlying storage system architecture affects performance**
- **Big I/Os are generally better than small I/Os**
  - Full stripe (e.g., 1 MiB – 8 MiB) avoids read-modify-write due to parity
  - Bigger can trigger more parallelism under the hood (good) or memory pressure (bad)
- **Aligned I/Os are better than misaligned I/Os**
  - Avoid read-modify-write due to false sharing
  - Avoid lock contention on parallel file systems
  - Avoid excessive garbage collection in SSDs
- **Use I/O middleware when possible**
  - MPI-IO understands stripe geometry and parallelism
  - PnetCDF and HDF5 understand alignment

Thank you!