# Parallel I/O in Practice

**Rob Latham**
**Rob Ross**
Math and Computer Science Division
Argonne National Laboratory
robl@mcs.anl.gov,
rross@mcs.anl.gov

**Brent Welch**
Google
welch.brent@gmail.com

**Katie Antypas**
NERSC
kantypas@lbl.gov

“There is no physics without I/O.”

– Anonymous Physicist
SciDAC Conference
June 17, 2009

(I think he might have been kidding.)

“Very few large scale applications of practical importance are NOT data intensive.”

– AlokChoudhary, IESP, Kobe Japan, April 2012

(I know for sure he was not kidding.)

# About Us

- Rob Latham (robl@mcs.anl.gov)
  - Principle Software Developer, MCS Division, Argonne National Laboratory
  - ROMIO MPI-IO implementation
  - Parallel netCDF high-level I/O library
  - Application outreach
- Rob Ross (rross@mcs.anl.gov)
  - Computer Scientist, MCS Division, Argonne National Laboratory
  - Parallel Virtual File System
  - High End Computing Interagency Working Group (HECIWG) for File Systems and I/O
- Brent Welch (welch.brent@gmail.com)
  - Google, Tech Infrastructure
  - Chief Technology Officer, Panasas
  - Berkeley Sprite OS Distributed Filesystem
  - Panasas ActiveScaleFilesystem
  - IETF pNFS
- Katie Antypas (kantypas@lbl.gov)
  - Department Head for Scientific Computing and Data Services, NERSC
  - Guides application groups towards efficient use of NERSC's Lustre and GPFS file systems.
  - Cori system Project Manager.

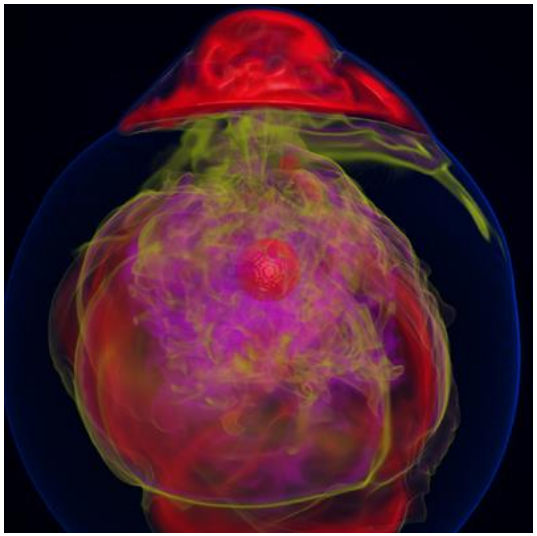
# Outline

- Introduction
- Storage hardware
- RAID's role in storage
- File system overview
- Parallel file system technology
- Lunch discussion
- I/O Forwarding
- In-system storage
- Science Drivers
- I/O libraries
  - POSIX
  - MPI-IO
  - Parallel-NetCDF
  - HDF5
- Characterizing I/O with Darshan
- Wrapping up

# Computational Science

- Use of computer simulation as a tool for greater understanding of the real world
  - Complements experimentation and theory
- Problems are increasingly computationally expensive
  - Large parallel machines needed to perform calculations
  - Critical to leverage parallelism in all phases
- Data access is a huge challenge
  - Using parallelism to obtain performance
  - Finding usable, efficient, and portable interfaces
  - Understanding and tuning I/O


IBM Blue Gene/Q system at Argonne National Laboratory.

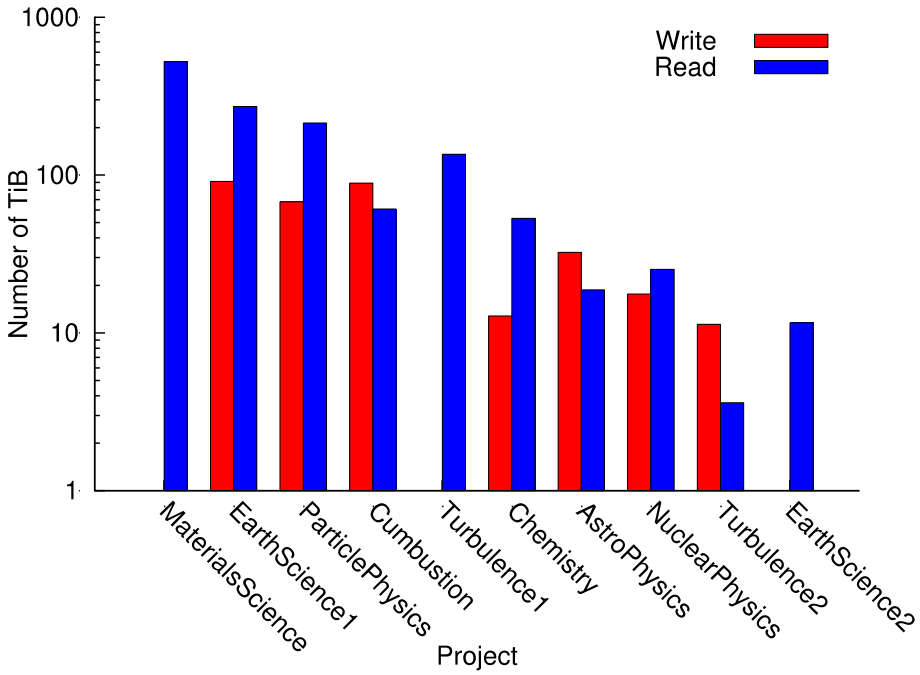


Visualization of entropy in Terascale Supernova Initiative application. Image from Kwan-Liu Ma's visualization team at UC Davis.

# Data Volumes in Computational Science

**Data requirements for select 2014 INCITE applications at ALCF (BG/Q)**

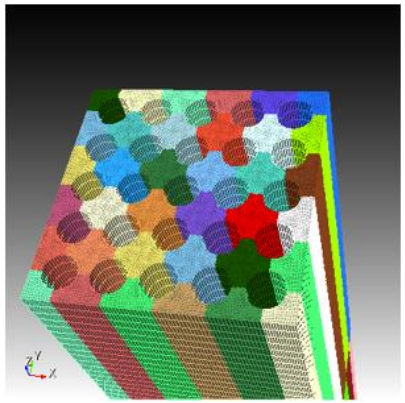
| PI | Project | On-line Data (TBytes) | Off-line Data (TBytes) |
|---|---|---|---|
| Khokhlov | Combustion in Reactive Gases | 100 | 1000 |
| Jordan | Seismic Hazard Analysis | 204 | 125 |
| Washington | Climate Science | 60 | 200 |
| Vary | Nuclear Structure and Reactions | 52 | 27 |
| Fischer | Reactor Thermal Hydraulic Modeling | 100 | 200 |
| Mackenzie | Quantum Chromodynamics | 2000 | 1000 |
| Langer | Plasma Physics | 1333 | 200 |
| Shun | Turbulent Combustion | 600 | 1000 |
| Galli | Physical Chemistry | 512 | 1000 |

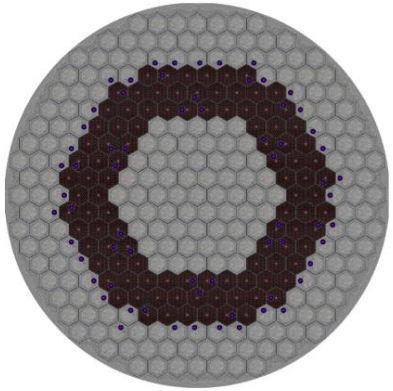



Top 10 data producer/consumers instrumented with Darshan over the month of July, 2011. Surprisingly, three of the top producer/consumers almost exclusively read existing data.

# Application Dataset Complexity vs I/O

- I/O systems have very simple data models
  - Tree-based hierarchy of containers
  - Some containers have streams of bytes (files)
  - Others hold collections of other containers (directories or folders)
- Applications have data models appropriate to domain
  - Multidimensional typed arrays, images composed of scan lines, variable length records
  - Headers, attributes on data
- Someone has to map from one to the other!





**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.

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

# Challenges in Application I/O

- Leveraging aggregate communication and I/O bandwidth of clients
  - …but not overwhelming a resource limited I/O system with uncoordinated accesses!
- Limiting number of files that must be managed
  - Also a performance issue
- Avoiding unnecessary post-processing
- Often application teams spend so much time on this that they never get any further:
  - Interacting with storage through convenient abstractions
  - Storing in portable formats

**Parallel I/O software is available that can address all of these problems, when used appropriately.**

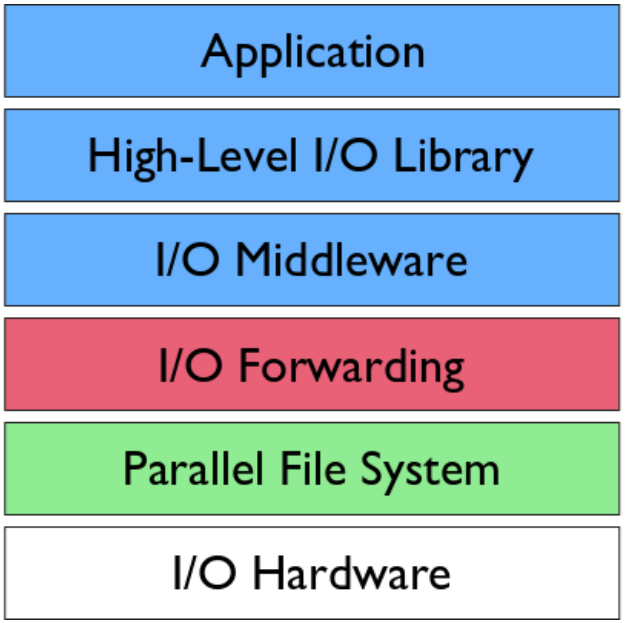
# I/O for Computational Science

**High-Level I/O Library**
maps application abstractions onto storage abstractions and provides data portability.

*HDF5, Parallel netCDF, ADIOS*

**I/O Forwarding**
bridges between app. tasks and storage system and provides aggregation for uncoordinated I/O.

*IBM ciod, IOFSL, Cray DVS*




**I/O Middleware**
organizes accesses from many processes, especially those using collective I/O.

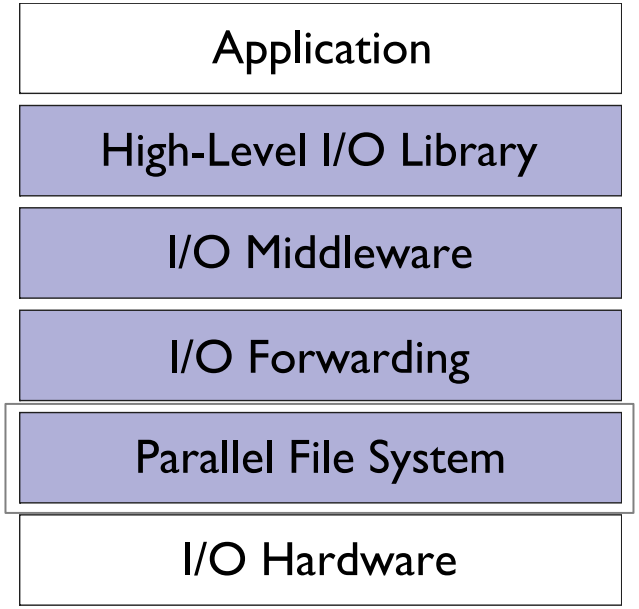
*MPI-IO*

**Parallel File System**
maintains logical space and provides efficient access to data.

*PVFS, PanFS, GPFS, Lustre*

**Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.**

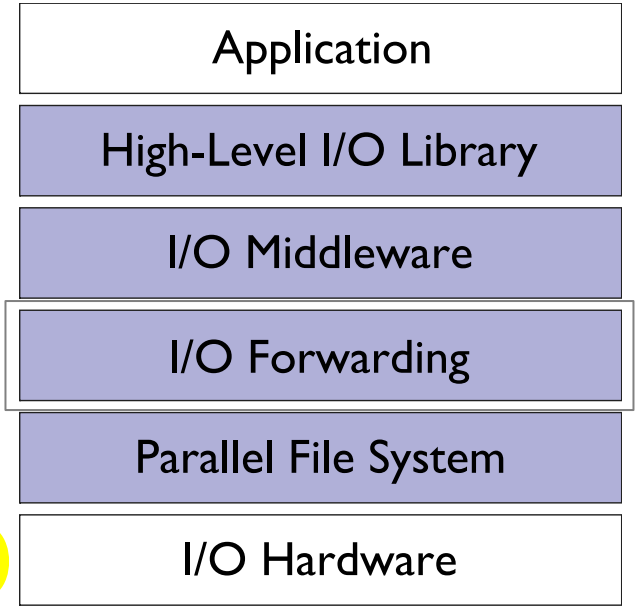
# Parallel File System



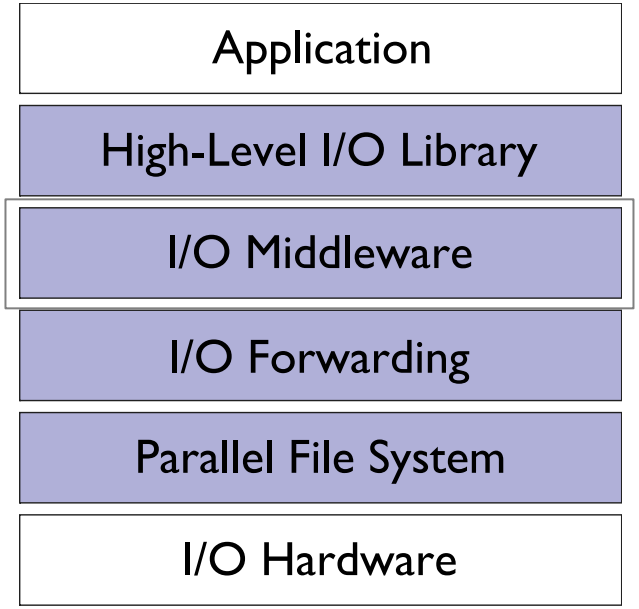

- Manage storage hardware
  - Present single view
  - Stripe files for performance
- In the I/O software stack
  - Focus on concurrent, independent access
  - Publish an interface that middleware can use effectively
    - Rich I/O language
    - Relaxed but sufficient semantics

# I/O Forwarding

- Present in some of the largest systems
  - Provides bridge between system and storage in machines such as the Blue Gene/P
- Allows for a point of aggregation, hiding true number of clients from underlying file system
- Poor implementations can lead to unnecessary serialization, hindering performance

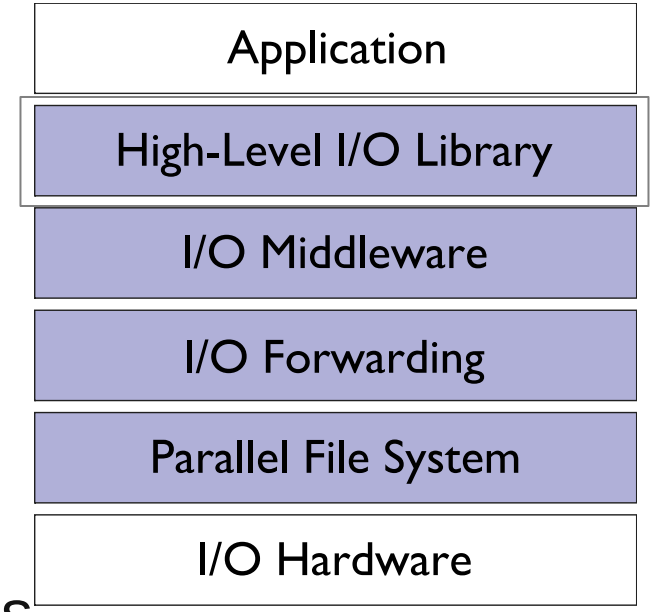


# I/O Middleware




- Match the programming model (e.g. MPI)
- Facilitate concurrent access by groupsof processes
  - Collective I/O
  - Atomicity rules
- Expose a generic interface
  - Good building block for high-level libraries
- Efficiently map middleware operations into PFS ones
  - Leverage any rich PFS access constructs, such as:
    - Scalable file name resolution
    - Rich I/O descriptions

# High Level Libraries



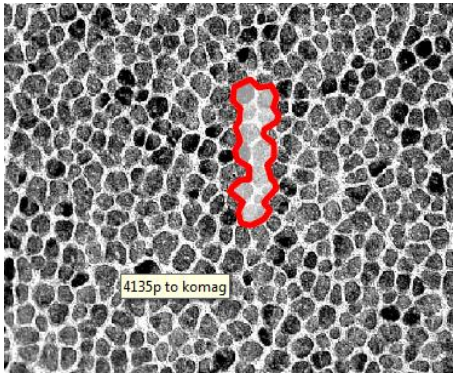

- Match storage abstraction to domain
  - Multidimensional datasets
  - Typed variables
  - Attributes
- Provide self-describing, structured files
- Map to middleware interface
  - Encourage collective I/O
- Implement optimizations that middleware cannot, such as
  - Caching attributes of variables
  - Chunking of datasets

# What we've said so far…

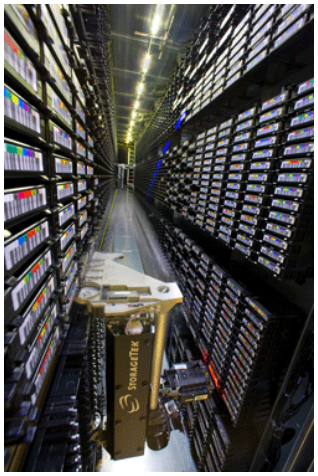
- Application scientists have basic goals for interacting with storage
  - Keep productivity high (meaningful interfaces)
  - Keep efficiency high (extracting high performance from hardware)
- Many solutions have been pursued by application teams, with limited success
  - This is largely due to reliance on file system APIs, which are poorly designed for computational science
- Parallel I/O teams have developed software to address these goals
  - Provide meaningful interfaces with common abstractions
  - Interact with the file system in the most efficient way possible

# Storage Hardware

# Building Storage Systems from Bits



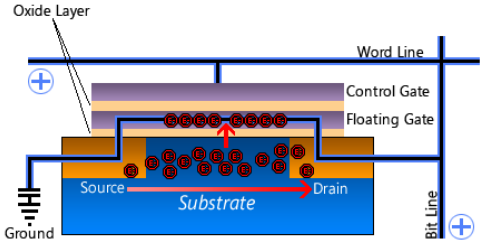

Magnetic or Solid State storage bits



Storage Devices

Software to aggregate many devices for performance

Software to handle device failures (erasure codes on blocks, across devices)





Software to handle software failures (server failover, write-ahead logging)

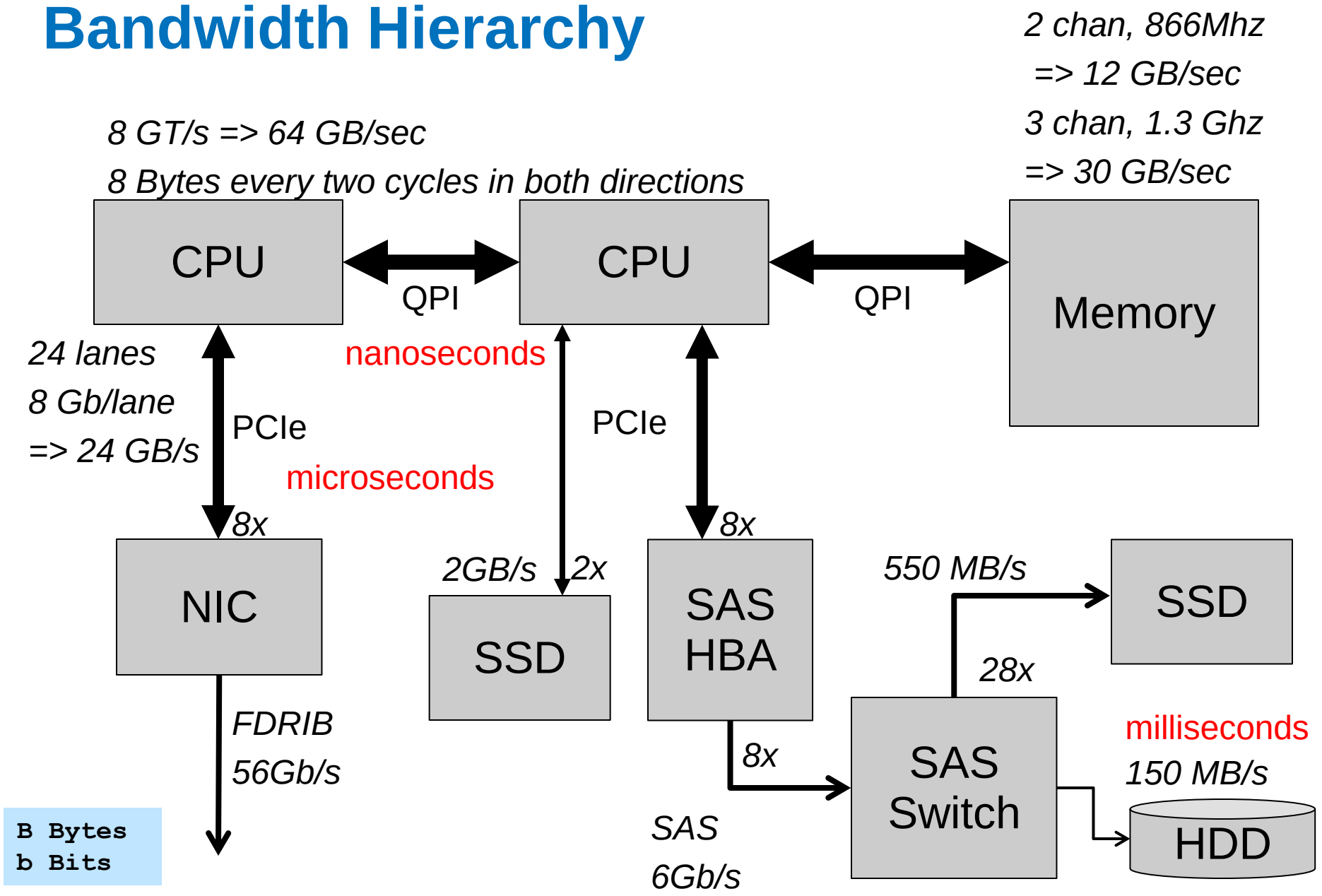
APIs to layer structure over raw storage

# The Storage Latency Hierarchy

| Technology | Latency | Size (e.g.) |
|---|---|---|
| L1 CPU Cache | 4 cycles (~1 nsec) | 32K |
| L2 CPU Cache | 10 cycles | 256K |
| LLC CPU Cache | 40 cycles | 1 MB |
| DRAM | 240 cycles | 16 GB |
| NVRAM | 2400 cycles | 64 GB |
| RDMA Read | 6K cycles (2 usec) | 16 GB |
| FLASH Read | 150K cycles (50 usec) | 128 GB |
| FLASH Write | 1500K cycles (500 usec) | 128 GB |
| HDD Write min | 1500K cycles (500 usec)* | 4 TB |
| HDD Read min | 15000K cycles (5 msec) | 4 TB |
| HDD Read max | 75000K cycles (25 msec) | 4 TB |
| Tape File Access | 150000000K cycles (50 sec) | 6 TB |

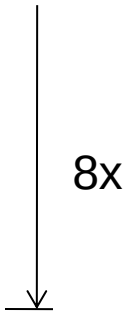
* Write to track cache

# Bandwidth Hierarchy



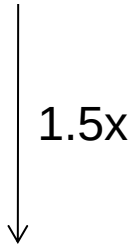
# Capacity vs Bandwidth

- Areal density increases by 40% per year
  - Per drive capacity increases by 50% to 100% per year
  - 2008: **500 GB**
  - 2009: **1 TB**
  - 2010: **2 TB**
  - 2011: **3 TB**
  - 2012: **4 TB**
  - 2014: **6 TB**
  - 2015: **8 TB**



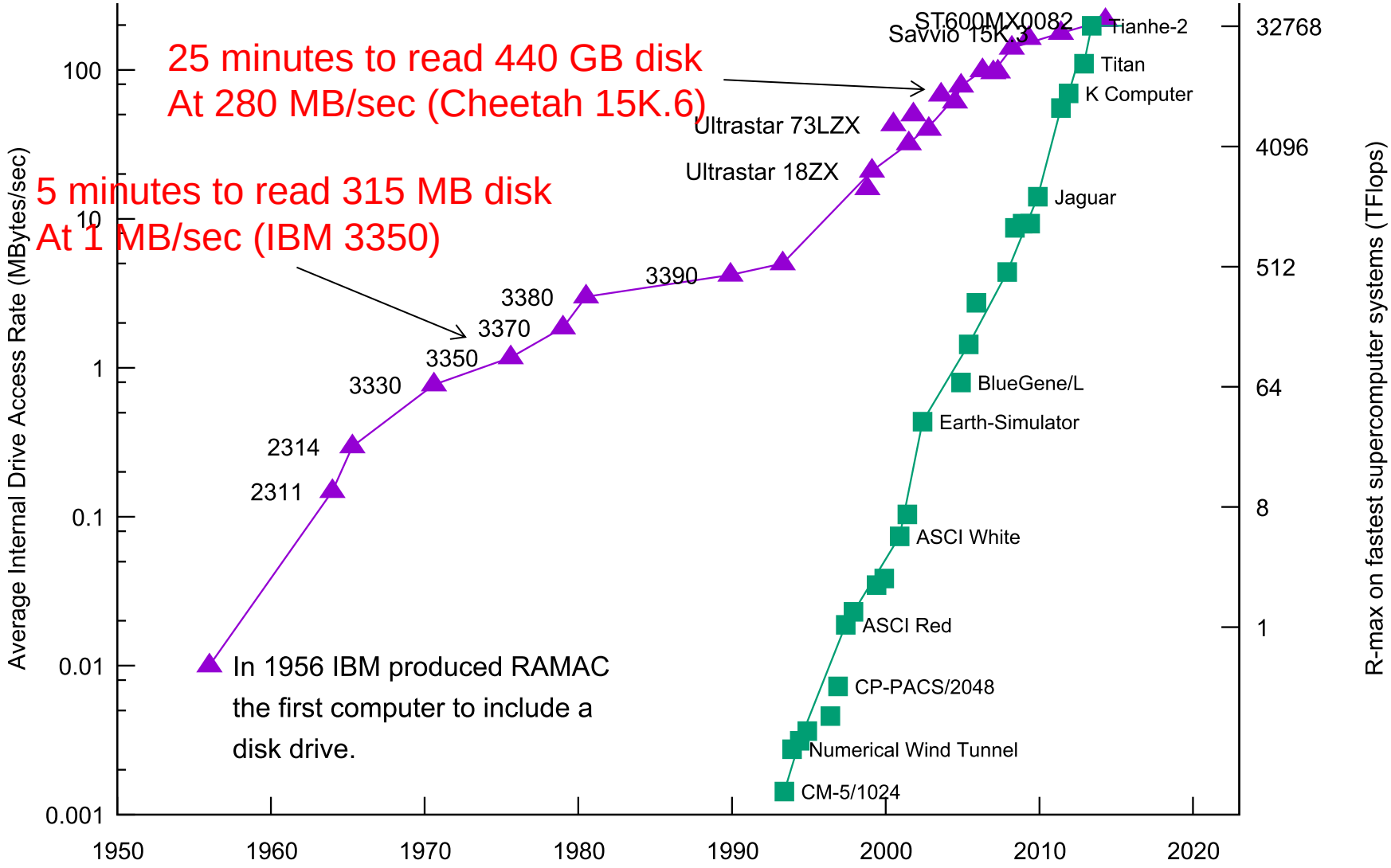

Takes longer and longer to completely read each new generation of drive

- Drive interface speed increases by 15-20% per year
  - 2008: 500 GB disk (WD RE2): **98 MB/sec**
  - 2009: 1 TB disk (WD RE3): **113 MB/sec** (+15%)
  - 2010: 2 TB disk (WD RE4): **138 MB/sec** (+22%)
  - 2013: 4 TB disk (WD SAS): **150 MB/sec** (+ 8%)



# Disk Transfer Rates over Time

7.4 hours to read 4 TB SATA
At 150 MB/sec



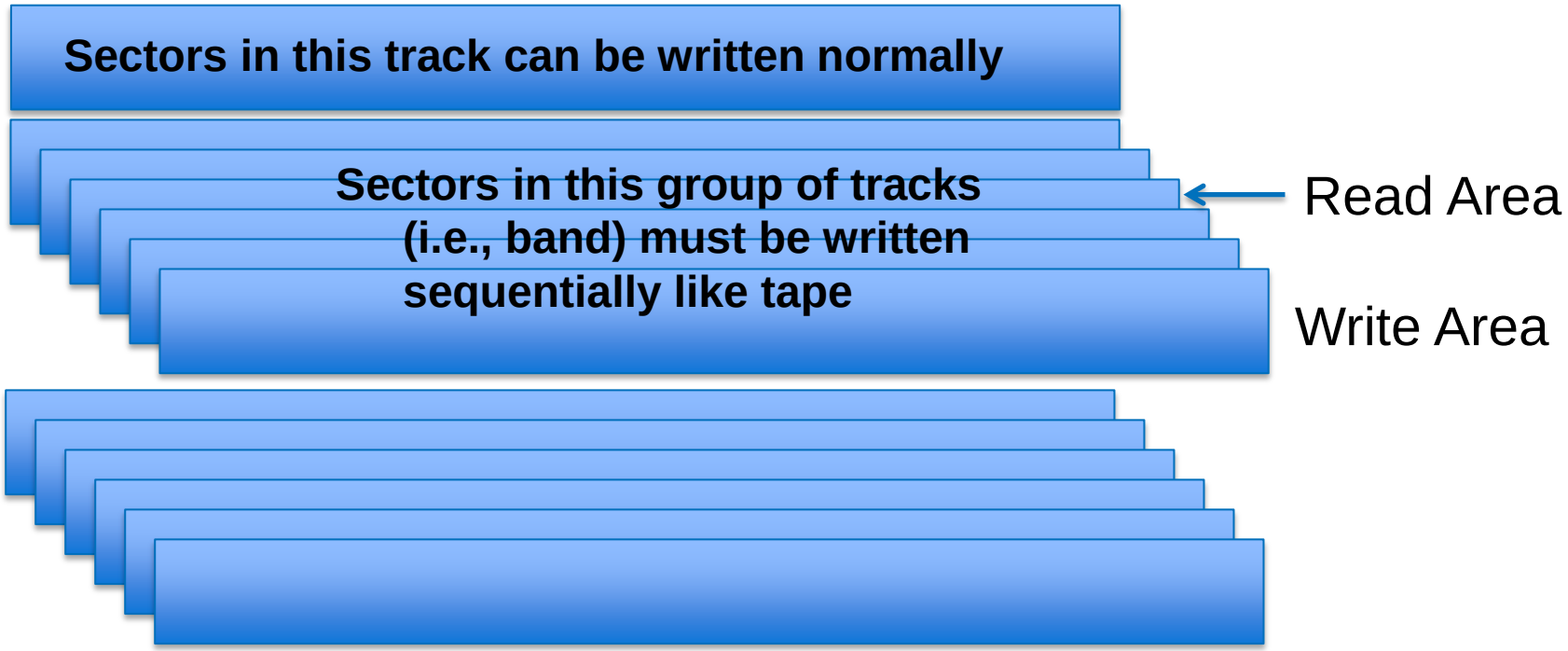

Thanks to R. Freitas of IBM Almaden Research Center for providing much of the data for this graph.

# HDD Technology

- Hard drive manufactures have roadmaps out to 60TB/disk over the next decade
  - Approximately. Might take longer. Probably be higher capacity.
- However, advances in drive interface speeds will continue to lag advances in drive capacity
  - As usual, every new generation of drive capacity will mean it takes longer to read or write the entire device
- New Technology
  - Helium filled drives
  - Shingled Magnetic Recording (SMR)
  - Heat Assisted Magnetic Recording (HAMR)

# Shingled Magnetic Recording (SMR)

- Overlapping recording tracks are like shingles on your roof
- Advanced signal processing extracts the desired bit values from the narrower, non-overlapped region during a Read



# FLASH and SSD

# SSD Components

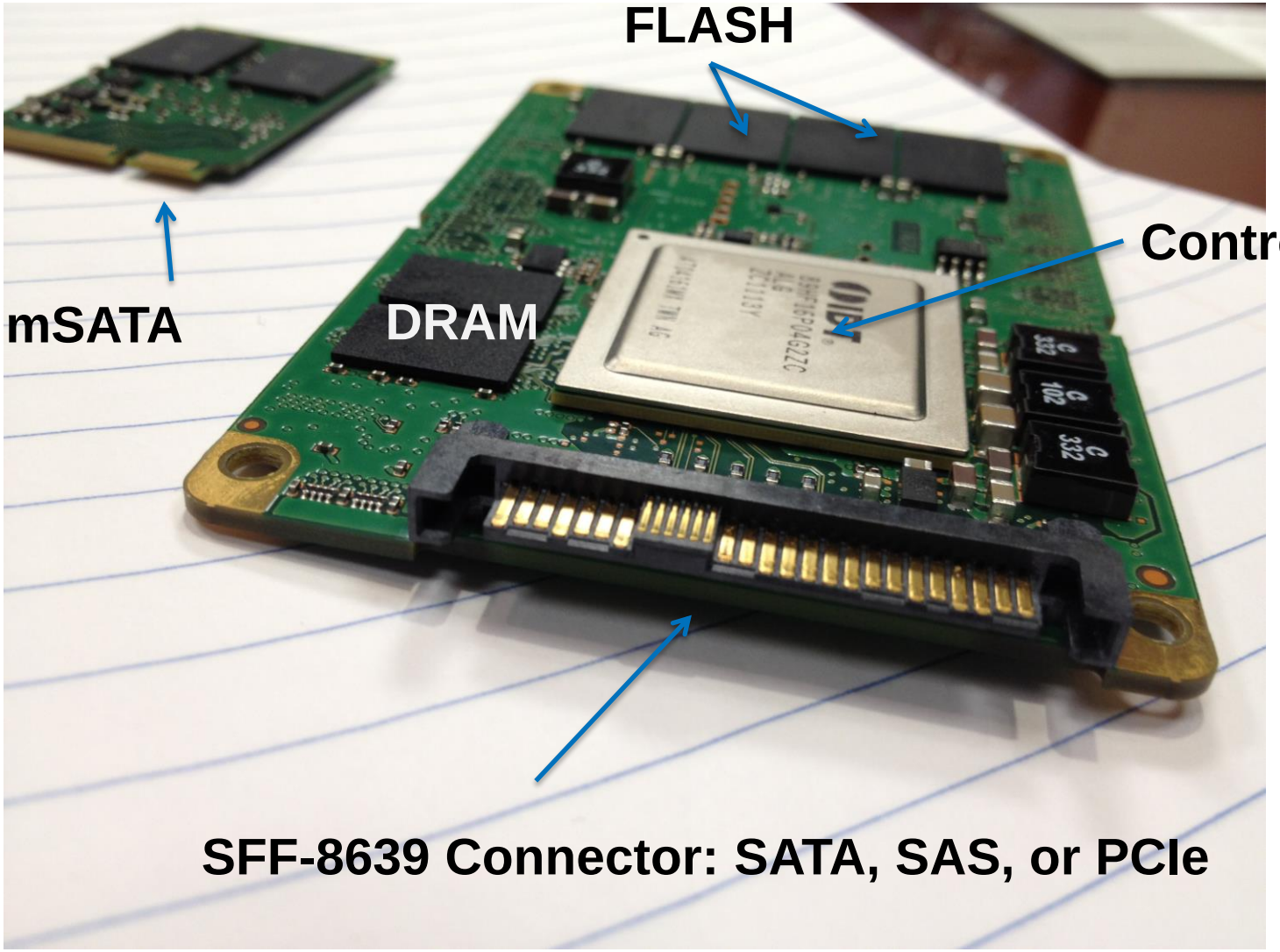
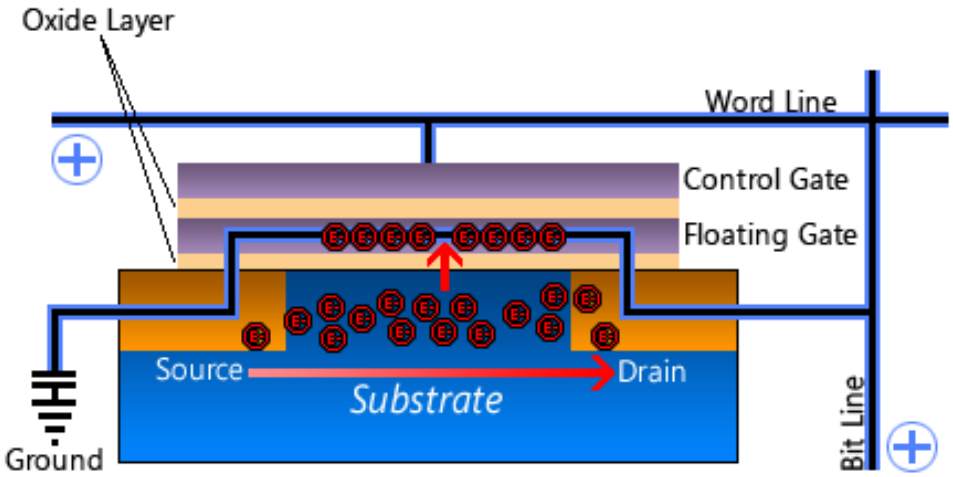



Image courtesy Brent Welch, personal collection

# FLASH Characteristics

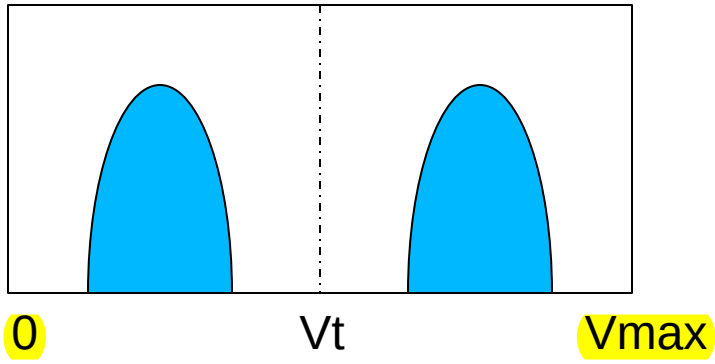
- Non-volatile
  - Each bit is stored in a “floating gate” that holds value without power
  - Electrons can leak, so shelf life and write count is limited
- Page-oriented
  - Smaller (e.g., 4K) read/write block based on addressing logic
  - Larger (e.g., 1MB) erase block to amortize the time it takes to erase
- Flash Translation Layer (FTL)
  - allows wear leveling
  - requires garbage collection
- Performance
  - Fast reads (no seeks)
  - Slower writes
  - Slow erase cycles
  - Background tasks cause interference (1 to 10 msec)

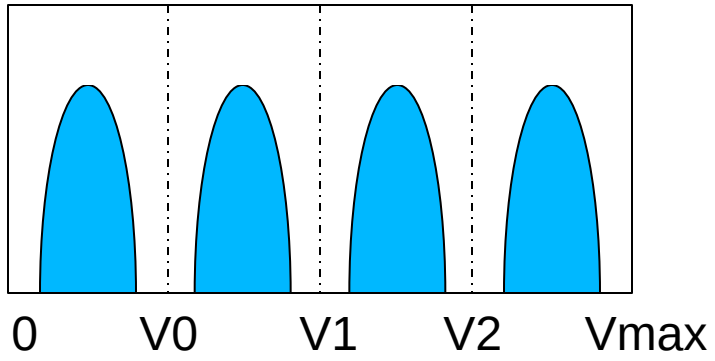



http://icrontic.com/articles/how_ssds_work

# FLASH Reliability

- SLC – Single Level Cell
  - One threshold, one bit
  - $10^5$ to $10^6$ write cycles per page
- MLC – Multi Level Cell
  - Multiple thresholds, multiple bits (2 bits)
  - N bits requires $2^N$Vt levels
  - $10^4$ write cycles per page
  - Denser and cheaper, but slower and less reliable
- TLC – Triple Level Cell
  - Cheapest, slowest writes
  - 500 write cycles per page!

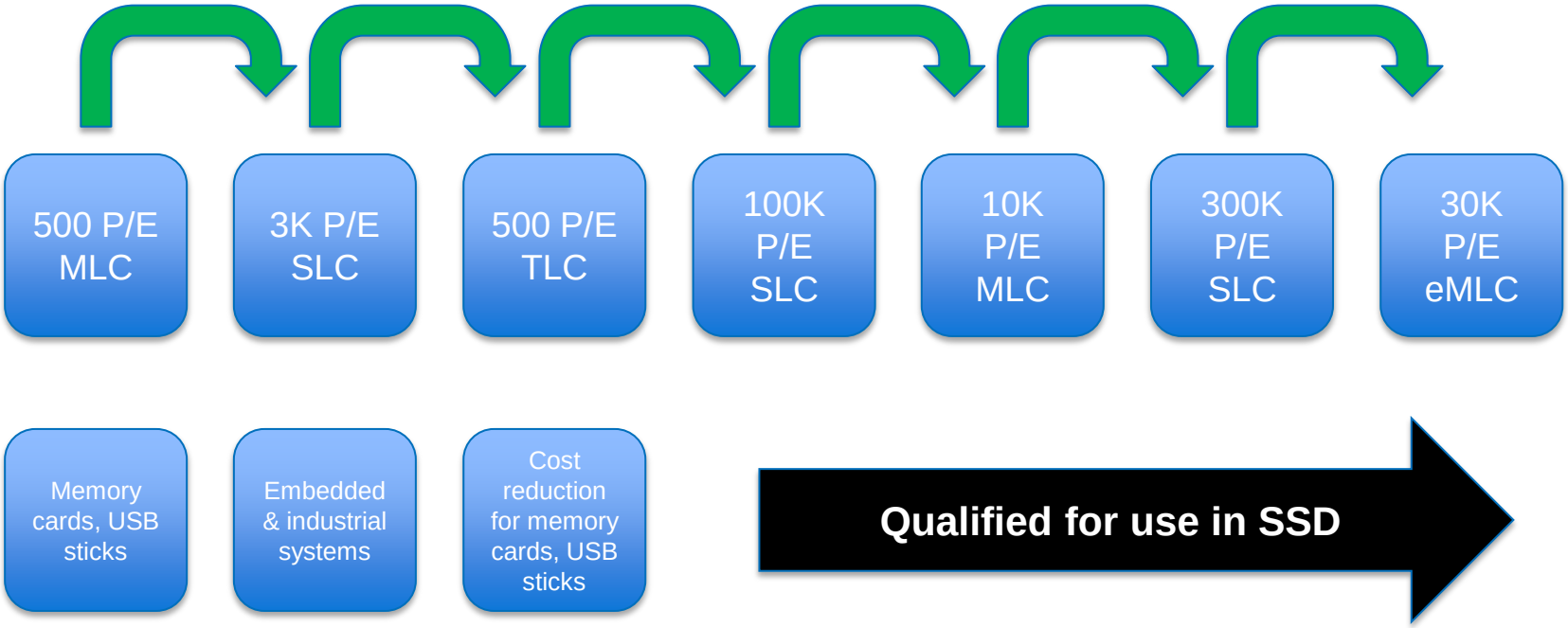



There is a countable number of electrons (e.g, 30) for Vmax

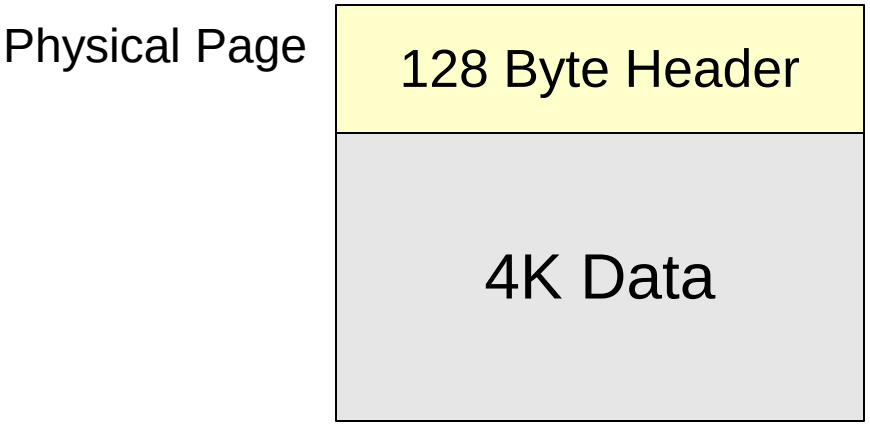
# NAND Process Evolution

Same NAND technology (e.g., 24nm) evolves different bit density and P/E cycles as they gain experience with the device and refine the program/erase process



# FLASH Translation Layer (FTL)

- Level of indirection supports wear leveling
  - Page map indirection allows controller to write to any free page
  - Page write may trigger background copies and erases
- Wear leveling is critical
  - Different pages will wear out at different times depending on how often each page is written
  - Pages in an Erase Block have to be garbage collected together
- Over provisioning
  - 120 GB device is physically 128GB to support wear leveling




Vendors are on 3rd (or 4th) generation algorithms

# SSD Rules of Thumb

- Large I/O Queue Depths exploit device parallelism
  - High IOP/s come from doing many async ops concurrently
- Read-only or Write-only yields best performance
  - Mixed workloads increase device-level conflicts
  - Long tail to operation latencies
- Sequential over-write is friendly to wear leveling
  - Subtlely: previous writes were re-organized by FTL
- Writes < 1 block cause premature wear out
  - 512b sector is emulated by read-modify-write of 8K
- Not all devices are power-fail safe
  - Consumer grade lack protection of DRAM buffers

# 3D X-Point NVRAM

- New Technology from Intel/Micron
  - Similar to Phase Change and Resistive Memories
  - Bit addressable. No Erase requirement
- "~1000 faster" than NAND
  - But still 3x to 5x slower than DRAM
- Process has a trajectory to catch NAND density
  - This is a big deal, first parts are 128 Gb (vs. 256 Gb)
- Much improved durability
  - However, even at 100 million write cycles (vs. 10K), you could wear it out in seconds w/out wear leveling
- Products will be memory (DIMM) or storage devices (PCIe)

# Why can't we junk all the disks

- Storage Hierarchy is DRAM, SCM, FLASH, Disk, Tape
- Cannot manufacture enough bits via Wafers vs. Disks
  - SSD 10x per-bit cost, and the gap isn't closing
  - Cost of semiconductor FAB is >> cost of disk manufacturing facility
  - World-wide manufacturing capacity of semi-conductor bits is perhaps 1% the capacity of making magnetic bits
    - 500 Million disks/year (2012 est) avg 1TB => 500 Exabytes (all manufacturers)
    - 30,000 wafers/month (micron), 4TB/wafer (TLC) => 1.4 Exabytes (micron)
- And Tape doesn't go away, either
  - Still half the per-bit cost, and much less lifetime cost
  - Tape is just different
    no power at rest
    physical mobility
    higher per-device bandwidth (1.5x to 2x)

# Web Pricing, August 31, 2015

| Description | Price | 1yr % chg | $/GB | |
|---|---|---|---|---|
| PNY 8 GB DDR3 1333 MHz | 60 | -20% | $7.50 | |
| Patriot 16 GB 1600 MHz | 80 | -50% | $5.00 | 5.0 |
| Corsair 64 GB 2400 MHz | 842 | | $13.16 | |
| Sony 32 GB microSD UX (70 MB/s) | 15 | -25% | $0.47 | |
| Sony 32 GB microSD UY (95 MB/s) | 21 | | $1.52 | |
| Samsung 120 GB SSD MZ-75E120B/AM | 59 | -33% | $0.50 | |
| Samsung 1 TB SSD MZ-75E1T0B/AM | 337 | -28% | $0.33 | 0.3 |
| Samsung 1 TB SSD MZ-7KE1T0BW | 460 | | $0.45 | |
| WD 2TB USB HDD | 82 | -17% | $0.040 | |
| WD 4TB HDD WD40EZRX Green | 135 | | $0.033 | 0.03 |
| Seagate 8TB 5900 RPM HDD ST8000AS0002 | 284 | | $0.035 | |
| Hitachi 8TB 7200 RPM HDD HE8 0F23668 | 478 | | $0.058 | |

Data from web search, August 31 2015

# Base-2 vs Base-10 measurements

| Unit | Base-10 | Base-2 | % diff |
|---|---|---|---|
| KB / KiB | 10^3 | 2^10 = 1,024 | 2.5% |
| MB / MiB | 10^6 | 2^20 = 1,048,576 | 5.0% |
| GB / GiB | 10^9 | 2^30 = 1,073,741,824 | 7.5% |
| TB / TiB | 10^12 | 2^40 = 1,099,511,627,776 | 10% |
| PB / PiB | 10^15 | 2^50 = 1,125,899,906,842,624 | 12.5% |
| EB / EiB | 10^18 | 2^60 = 1,152,921,504,606,846,976 | 15% |

Storage vendors sell in base-10 units (Megabyte)

Even though a disk sector is an even power of 2

512 bytes or 4096 bytes

Computer scientists often think in base-2 units (Mebibyte)

Even though they use base-10 unit terms

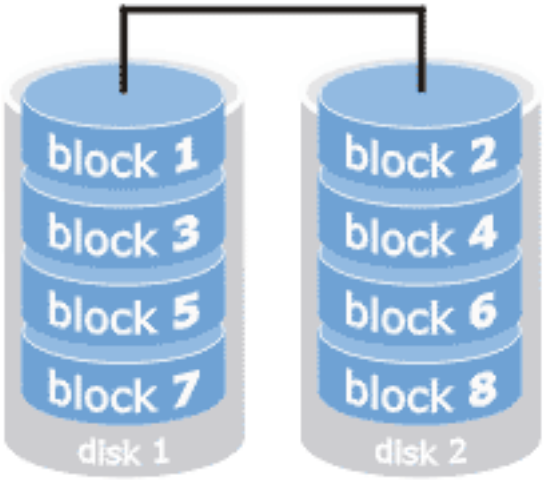
```
GB - Bytes
Gb - Bits
```

# RAID and Erasure Codes

The design and performance of the underlying data protection system has a fundamental effect on the behavior of the storage system

# The Disk Bandwidth/Reliability Problem

- Disks are slow: use lots of them in a parallel file system
- However, disks are unreliable, and lot's of disks are even more unreliable

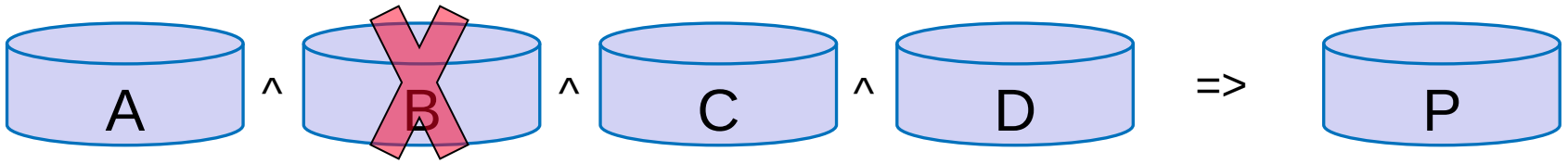



- **This simple two-disk system is twice as fast, but half as reliable, as a single-disk system**

# RAID Overview

- RAID is a way to aggregate multiple physical devices into a larger virtual device
  - Redundant Array of Inexpensive Disks
  - Redundant Array of Independent Devices
- Invented by Patterson, Gibson, Katz, et al
  - http://www.cs.cmu.edu/~garth/RAIDpaper/Patterson88.pdf
- Redundant data is computed and stored so the system can recover from disk failures
  - RAID was invented for bandwidth
  - RAID was successful because of its reliability

# RAID and Data Protection




- RAID equation generates redundant data:
  - P = A xor B xor C xor D  (encoding)
  - B = P xor A xor C xor D  (data recovery)

- **RAID equations are "erasure codes" because you can erase something (i.e., lose a disk) and get it back using the erasure code**

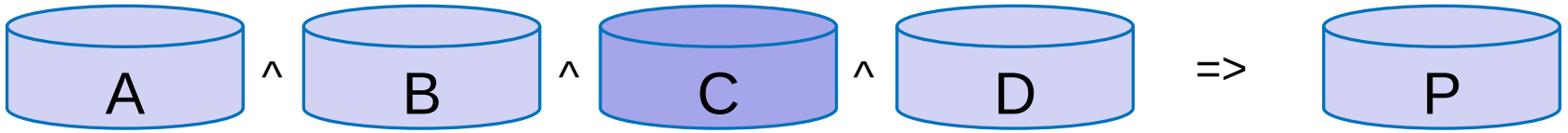
| XOR | 1 | 0 |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |

# Erasure codes / Reed Solomon

- Data stripes organized as D data and N encoding chunks
  - May tolerate N failures
    - RAID-4 and RAID-5 have N=1, with a single encoding chunk from XOR parity (P)
    - RAID-6 is when N=2. Encoding chunks are called P, Q
    - RAID-1 (mirroring) is a degenerate case where encoding chunk equals data chunk
- First redundancy unit is a simple XOR
  - Additional redundancy units require more complex math (Galois Field)
  - Can be done efficiently with combination of XOR and table lookups
- Failures versus corruption
  - Tolerate up to N failures (i.e., erasures)
  - Detect (i.e., locate) and repair up to N-1 corruptions
    - Very important to avoid silent corruptions and error propagation

# The Small Write Problem

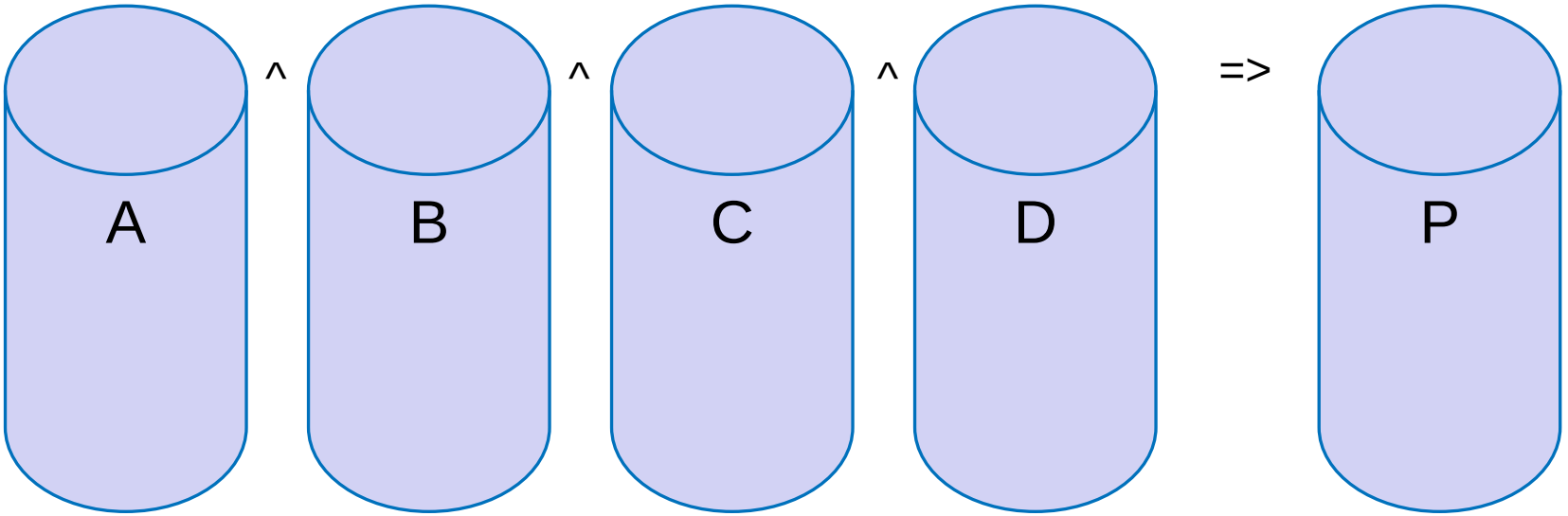
- When you only write part of a stripe, you need to compute parity across data blocks that aren't in-hand
- Two approaches
  - Large write: read the unwritten components
  - Small write: read the written components




- 4-cycle write to update one disk
  - Read old value of C
  - XOR with new value of C and save the result in T
  - Write new value of C
  - Read old value of P
  - XOR T and old P, write the result as the new P

# The problem with RAID

- Traditional block-oriented RAID protects and rebuilds entire drives
  - *Drive capacity increases have outpaced drive bandwidth*
  - It takes longer to rebuild each new generation of drives
  - Media defects on surviving drives interfere with rebuilds
- We need faster rebuilds, and a way to handle media defects



# Blade Capacity and Speed History

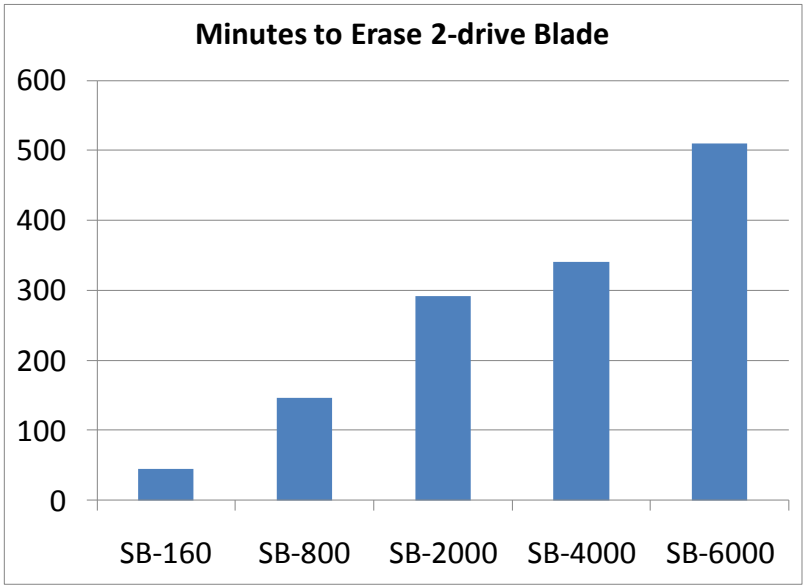
Compare time to write a blade (two disks) from end-to-end over 4* generations of Panasas blades

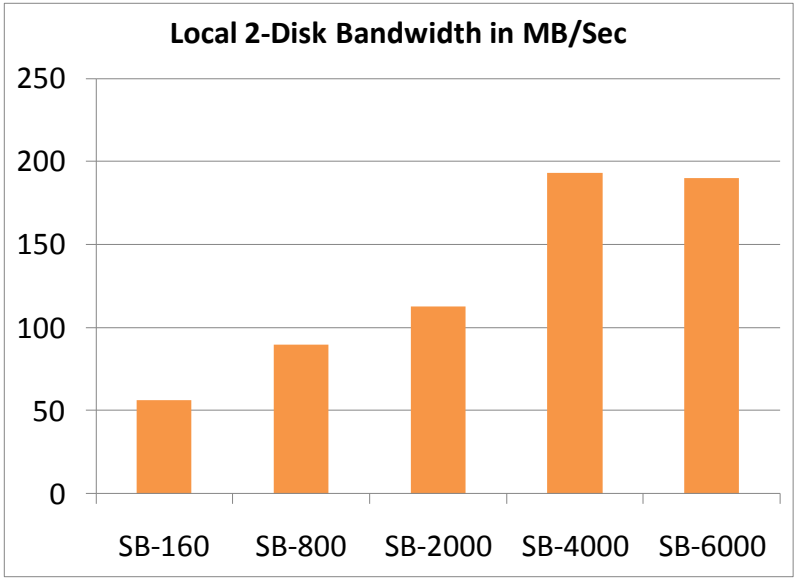
*SB-4000 same family as SB-6000*

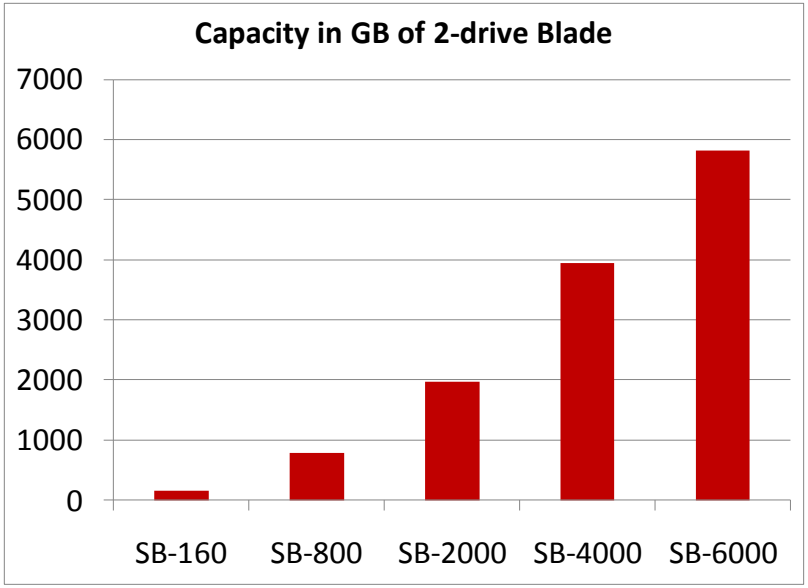
Capacity increased 39x

Bandwidth increased 3.4x (function of CPU, memory, disk)

Time goes from 44 min to > 8 hrs



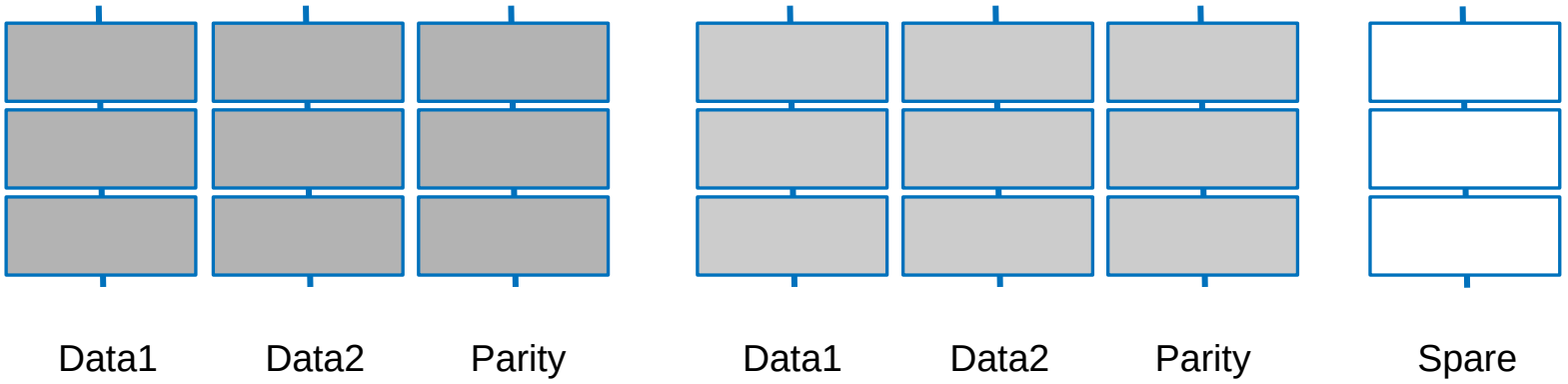



# Improving RAID

- Improving rebuild times
  - Declustered parity groups provide more disk bandwidth
  - Parallel rebuild algorithms provide more XOR and memory bandwidth
  - Declustered rebuilds reduce hot spots
- Improving data robustness
  - Per-file (per-object) RAID equation creates small fault domain
  - “Too many” failures cause loss of one file, not the whole RAID array
  - Cloud storage systems take this approach
- Declustering illustration to show doubling of disk bandwidth and rebuild performance

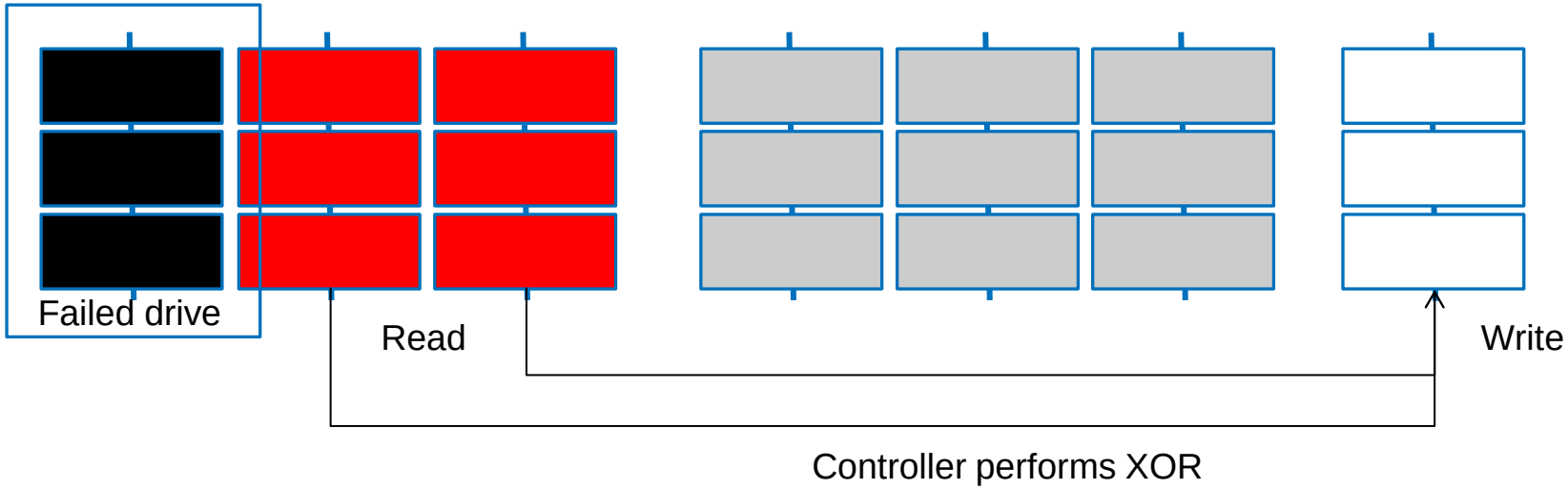
# Traditional RAID Organization

- Multiple RAID Groups
  - 2 Groups, each 2 Data + 1 Parity in this simple 7 disk example
- Global spare disk



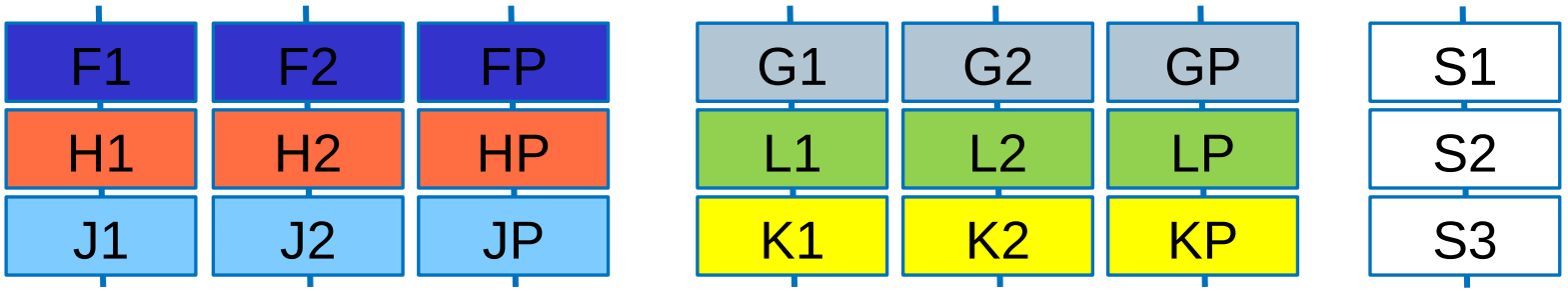
# Traditional RAID Rebuild

- Group with failed drive is busy with reads
- Global spare is busy with writes
- Other RAID groups do not participate
- Uneven utilization slows down parallel I/O using all groups



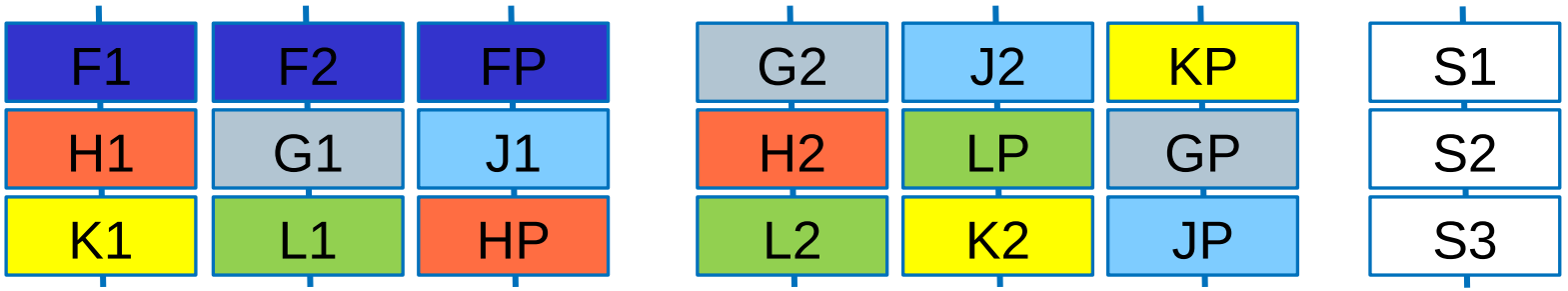
# Declustering Step 1

- Decluster to spread I/O over more devices
  - How do we get all the disks to participate in the rebuild?
- Subdivide devices into multiple partitions
  - F1 xor F2 => FP
  - H1 xor H2 => HP
  - etc



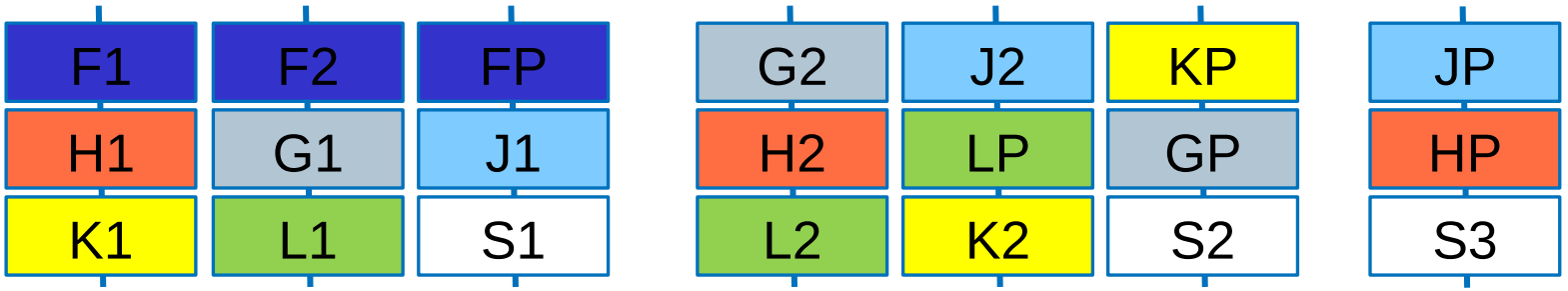
# Declustering Step 2

- Shuffle data and parity blocks
- Each device has at most one piece of a group
  - Must not lose two pieces with one device failure



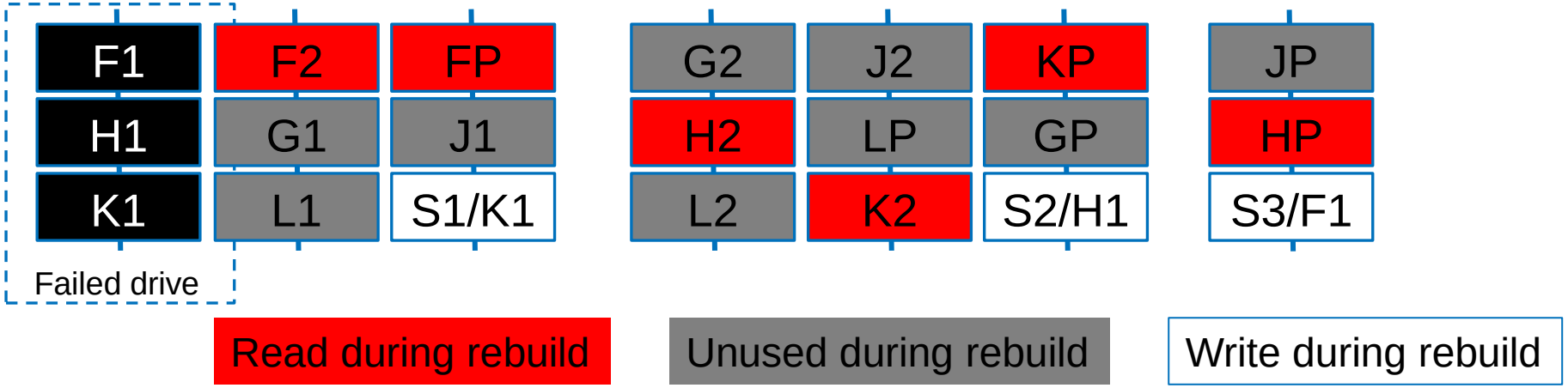
# Declustering Step 3

- Spread out Spare space, too
- Placement constraints on what spare can be used
  - Cannot result in two pieces of a group on one device
  - E.g., cannot reconstruct J2 into S1 because it would be on the same device as J1



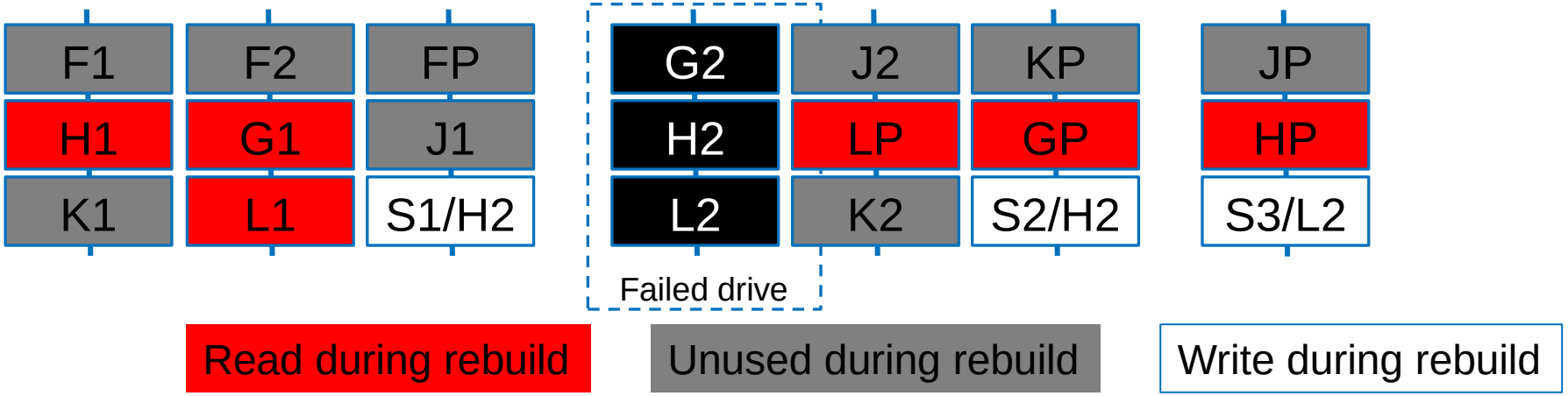
# Declustered RAID Rebuild

- Every surviving drive contributes bandwidth
- Same I/O spread over more spindles
  - Reading 2 drives worth of data
  - Writing 1 drive worth of data
  - 6 spindles active, vs. 3 with traditional rebuild
  - More RAID bandwidth. More uniform performance impact.



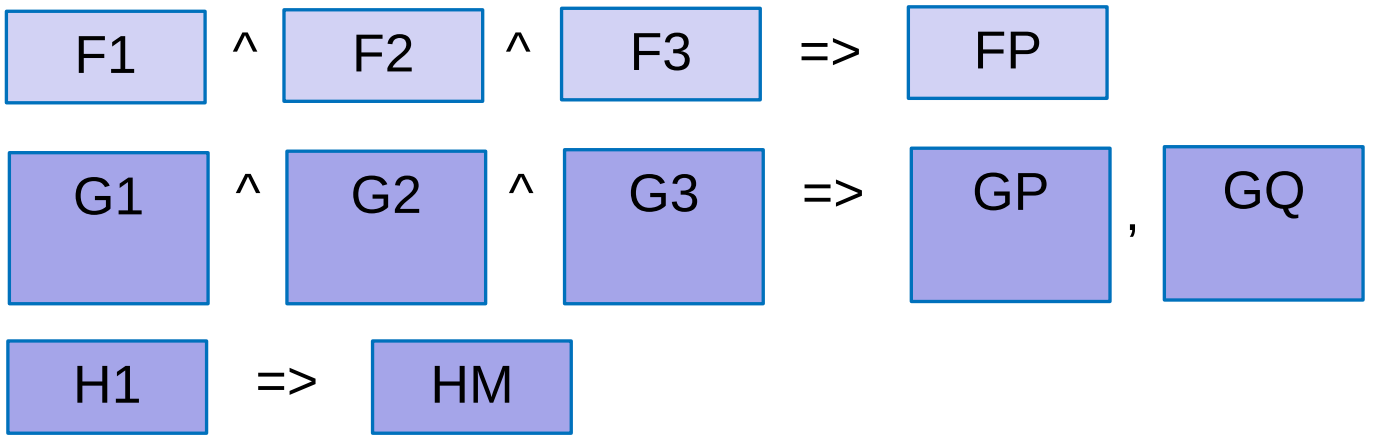
# Declustered RAID Rebuild

- Perfect placement is a hard problem
  - Mark Holland dissertation from 90's



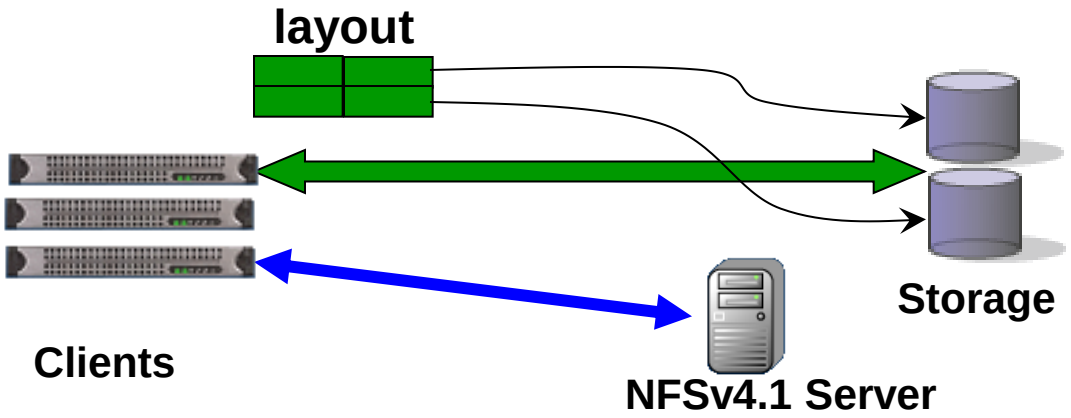
# Object RAID

- Object RAID protects and rebuilds user objects, not disks
  - Failure domain is a user object (e.g., a file), which is typically much, much smaller than the physical storage devices
  - File writer can be responsible for generating redundant data, which avoids central RAID controller bottleneck
  - Different files sharing same devices can have different RAID configurations to vary their level of data protection and performance



# pNFS Layouts

- Client gets a *layout* from the NFS 4.1 Server
- The layout maps the file onto storage devices and addresses
  - Object-based layouts support per-file erasure codes
- The client uses the layout to perform direct I/O to storage
- At any time the server can recall the layout
- Client commits changes and returns the layout when it's done
- pNFS is optional, the client can always use regular NFSv4 I/O

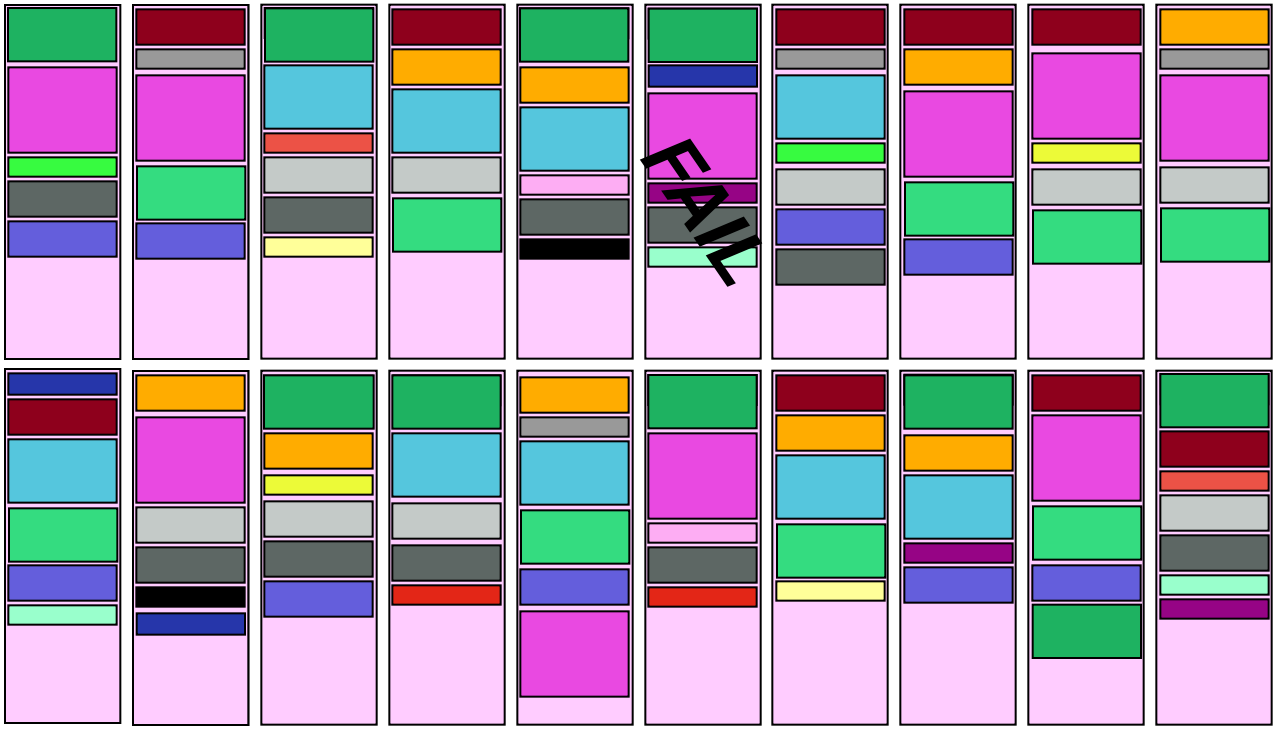


# Parallel Declustered Object RAID

- File attributes replicated on first two component objects
- Components grow & new components created as data written
- Component objects include file data and file parity
- Declustered, randomized placement distributes RAID workload
- Per-file RAID equation creates fine-grain work items for rebuilds

***20 OSD Storage Pool***

***Mirrored or 9-OSD Parity Stripes***




***Read about half of each surviving OSD***

***Write a little to each OSD***

***Scales up in larger Storage Pools***

# DataCenter Scale Erasure Codes

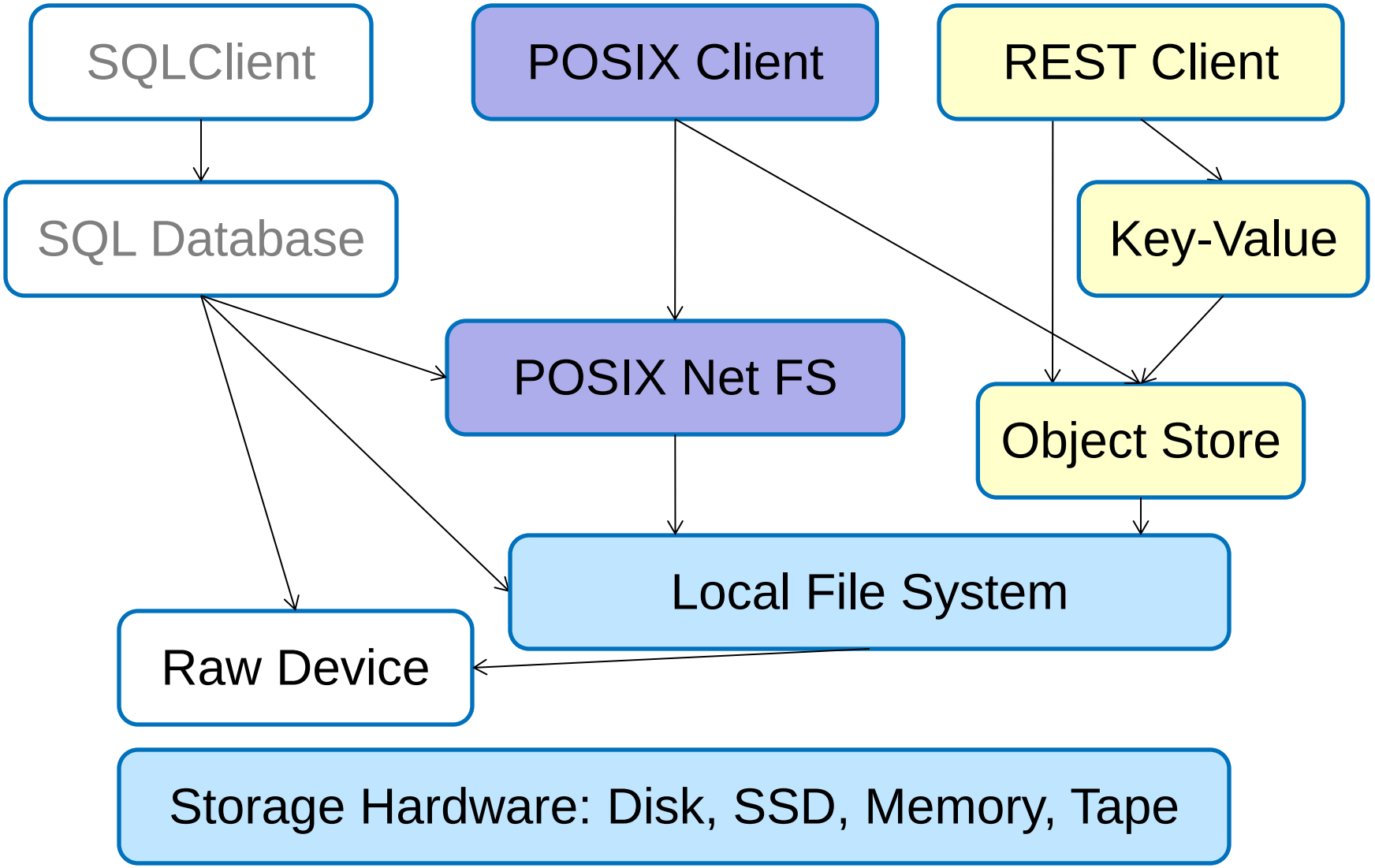
- Erasure codes (8+3, 10+4, etc) not replication
  - 12+6 encoding has 50% overhead and many orders of magnitude more reliability than triplication, which has 200% space overhead.
- Data organized into stripes
  - e.g., 96 MB user data is stripe of 12 Data, 6 Parity chunks
  - 18 chunks dispersed to different servers and racks
- Declustering on a grand scale
  - 18,000 disks on 1000 servers hold 1 billion stripes
  - 1 Failure affects 1/1000 of the stripes
  - 1/1000 of each remaining disk is read during RAID rebuild
  - Recovery of 4TB disk takes e.g., 10 minutes
- Failure Rates
  - 2% Annual Failure Rate (AFR) => 1 disk failure/day
- How to track 1 billion stripes ?
  - (stay tuned)

# RAID Summary

- RAID was invented for performance, but used for protection
- Block RAID is suffering from increased drive sizes
- Object RAID (or triplication) with parallel rebuild provides fast recovery for large scale systems
- Per-file RAID equations allow different performance/protection for different files, and isolate bad failures to individual files
- Declustering spreads RAID workload uniformly over large systems to reduce hot spots in parallel I/O environments
- Erasure codes have less space overhead than replication

# What is a file system and a parallel file system

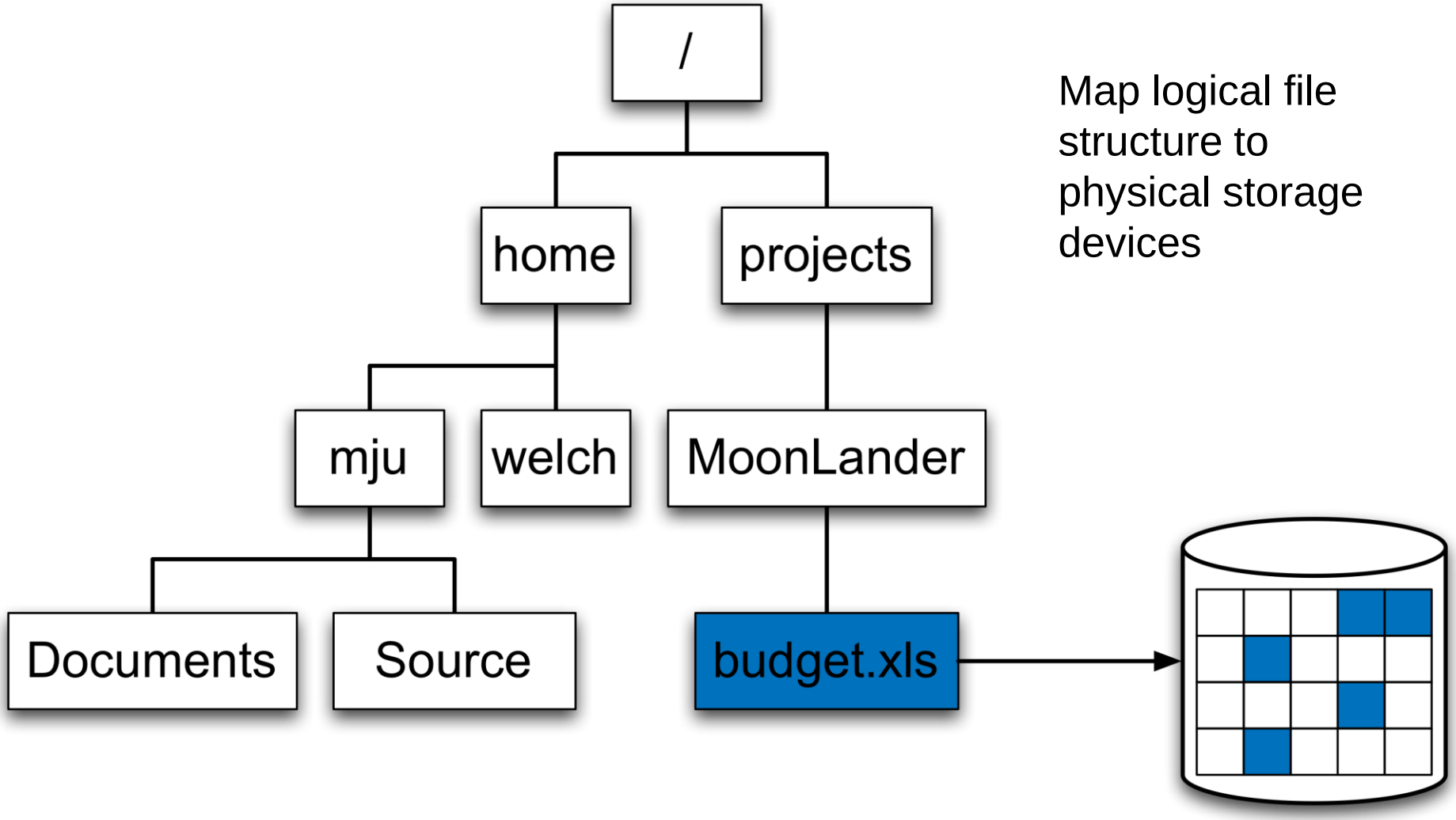
# Storage Stack Overview



# File Systems Part 1

- Local file system structures as a building block
- Network sharing, NAS vs. SAN
- Composing things via kernel VFS layer
- Compare different approaches
  - SAN FS
  - NFS
  - Object FS

# Role of the File System




Map logical file structure to physical storage devices

# File Systems

- File systems have two key roles
  - Organizing and maintaining the file name space
  - Storing contents of files and their attributes

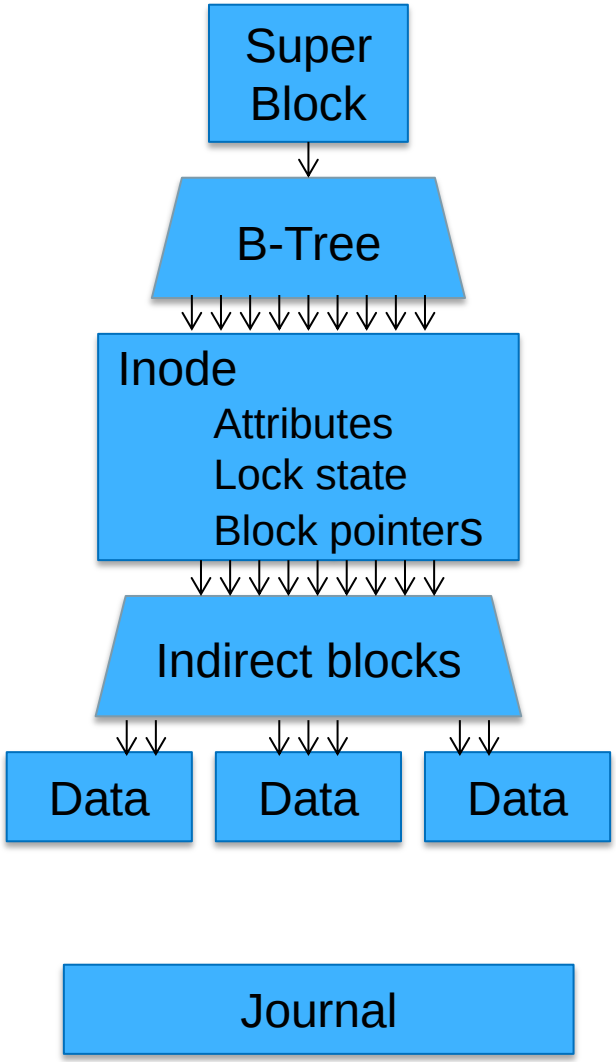

- Networked file systems must solve two new problems
  - File servers coordinate sharing of their data by many clients
  - Scale-out storage systems coordinate actions of many servers
- Parallel file systems (PFS) support parallel applications
  - A special kind of networked file system that provides high-performance I/O when multiple clients share the file system
  - The ability to scale capacity and performance is an important characteristic of a parallel file system implementation

# Local File Systems

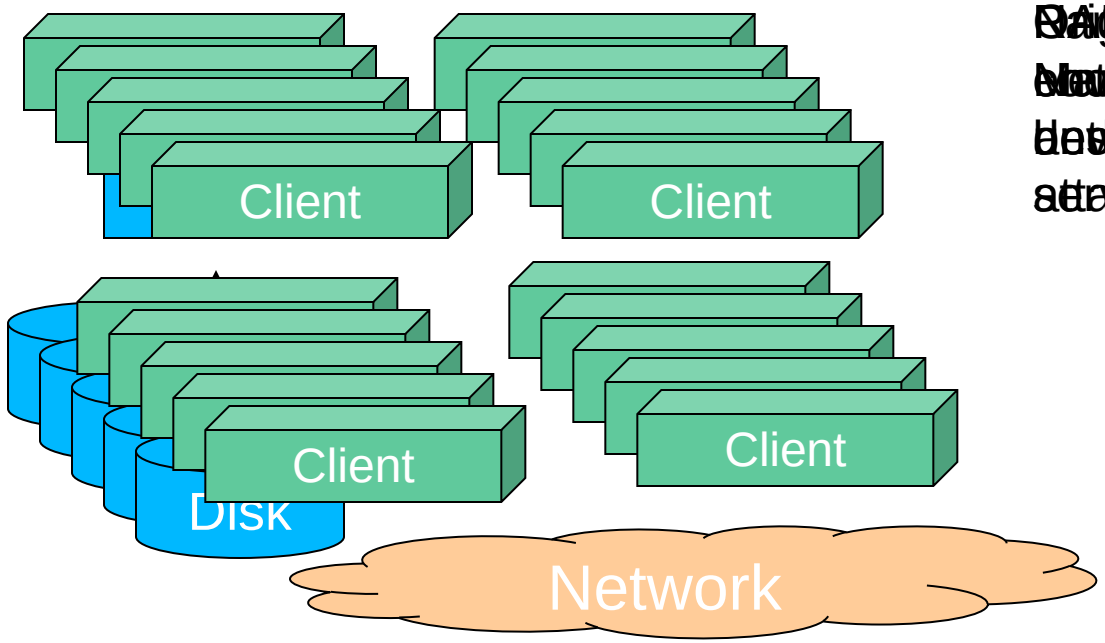
Persistent data structure maps from a user's concept of a file to the data and attributes for that file.

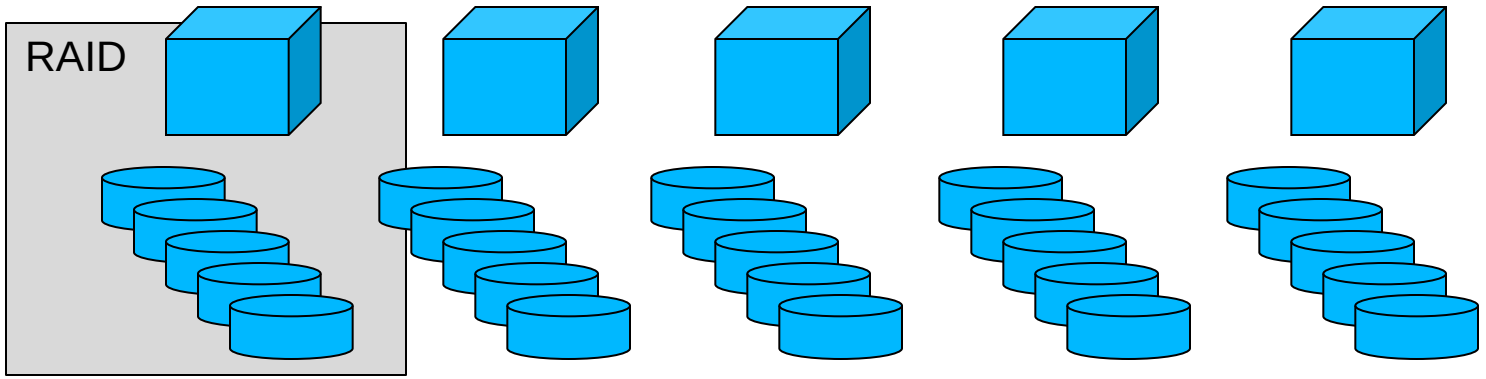
Early research and differentiation was all about optimizing access to a single device

UFS, EXT4, ZFS, NTFS, XFS and BtrFS are local file systems

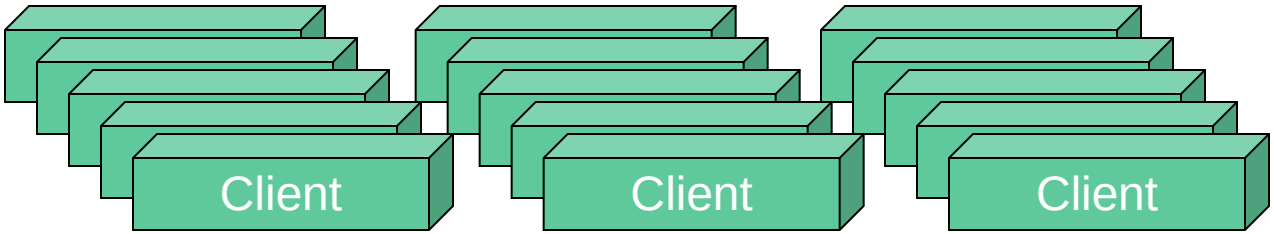


# Scaling the File System



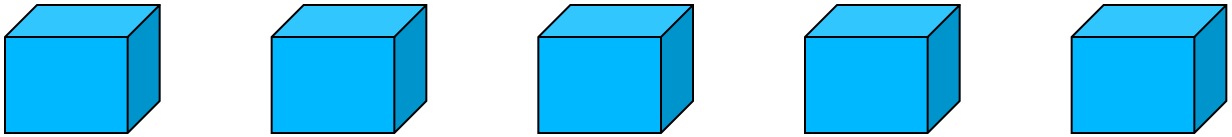



# SAN vs NAS




Ethernet or Infiniband

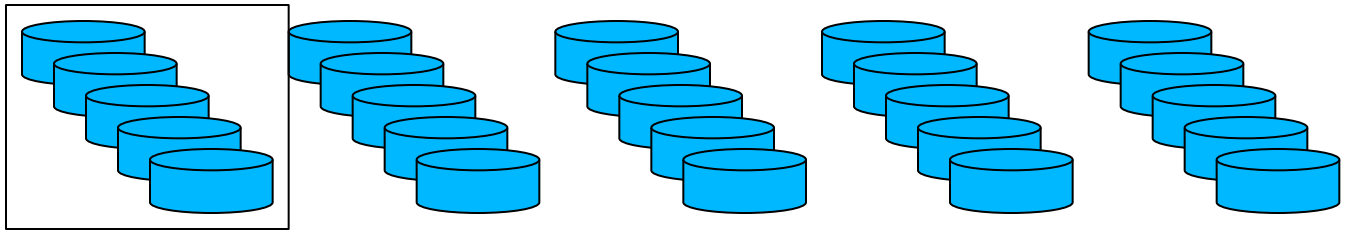
Network Attached Storage (NAS)



Fiber Channel, SAS, Ethernet, Infiniband

Storage Area Network (SAN)

RAID



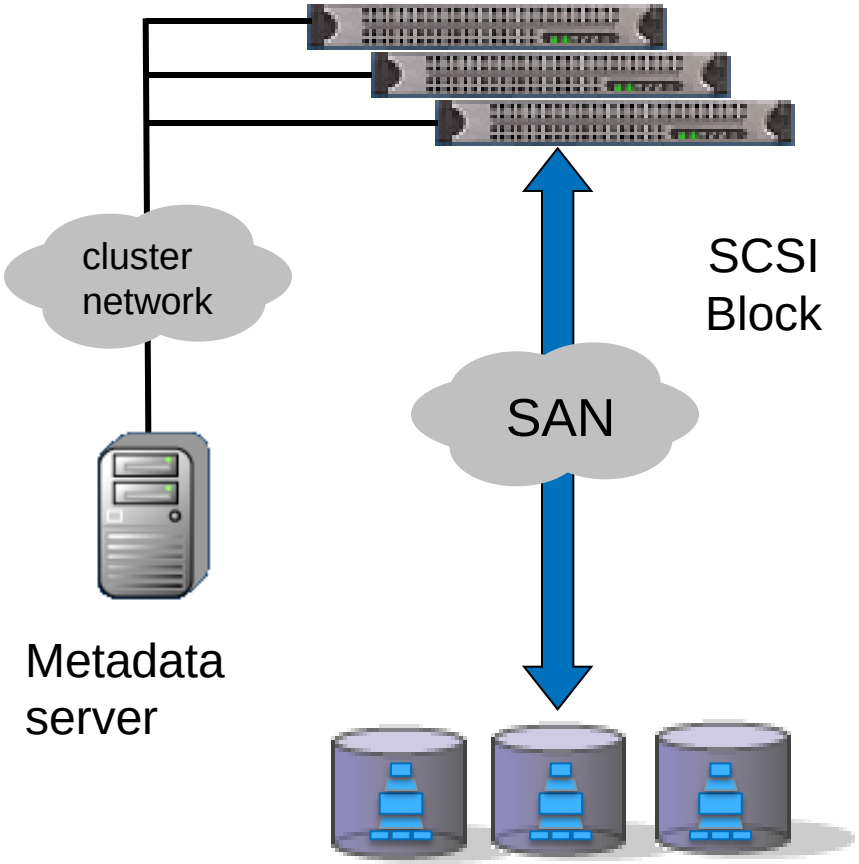
# Distributed File System Functions

- Data virtualization
  - Striping or indirection to spread data among servers
  - Global namespace that spans all servers, visible to all clients
- Coordination (locking and synchronization)
  - Among clients sharing files
  - Among servers sharing physical devices
- Fault tolerance
  - For disk and server hard failures
  - For power failures
  - For software faults
  - For network faults
  - For client failures

# Challenging Scenarios

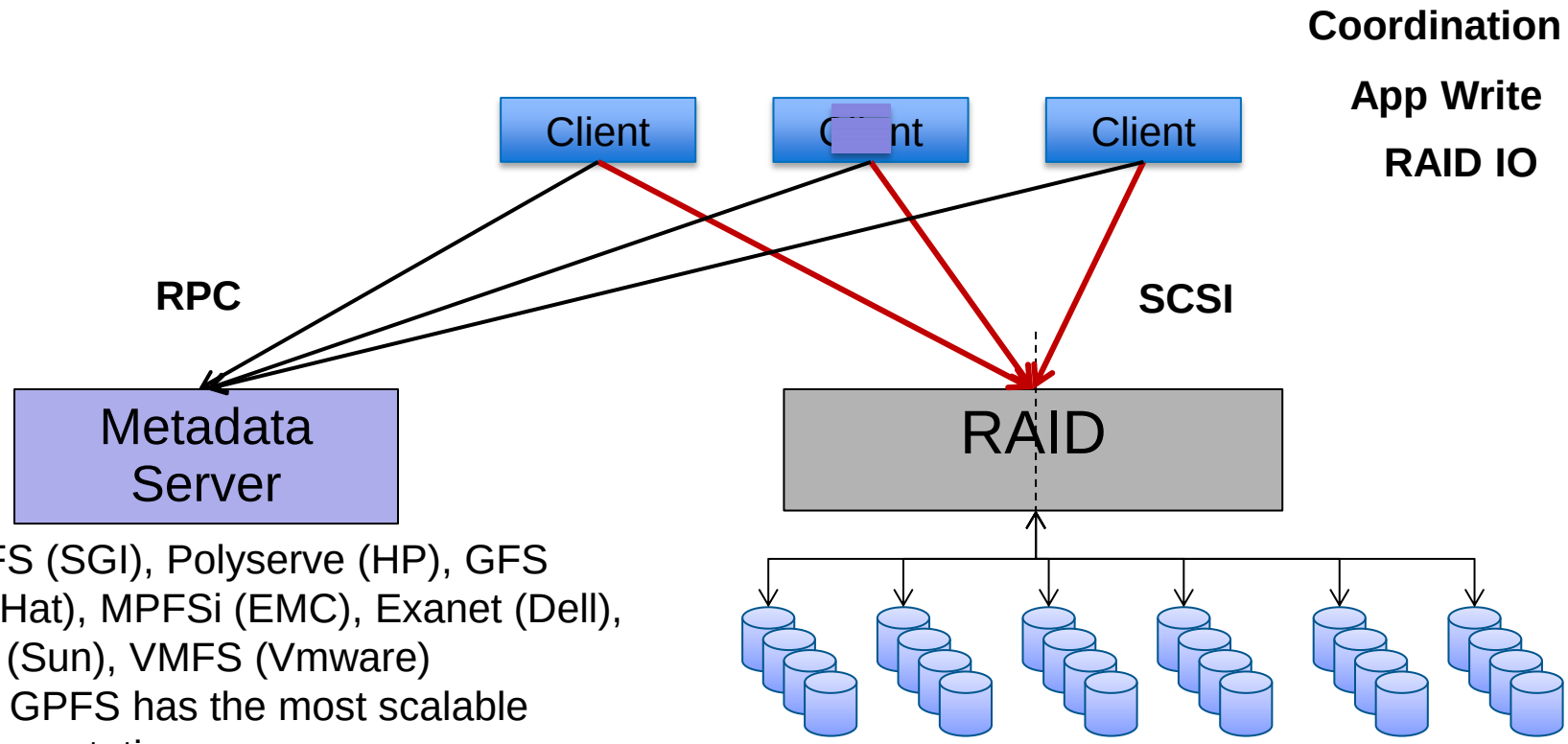
- Concurrent creates/deletes within a shared directory
  - Who owns the lock?
  - Who updates the directory?
  - Who can read the directory?
- `ls -l` in large active directory
  - Who knows how big the files are, and their modify time?
- Concurrent read/writer to a shared file
  - Who knows how big the file is?
  - Is read-ahead or caching feasible?
- Concurrent writers to a shared file
  - Who knows how big the file is?
- It is hard even when nothing goes wrong

# SAN Shared Disk File Systems



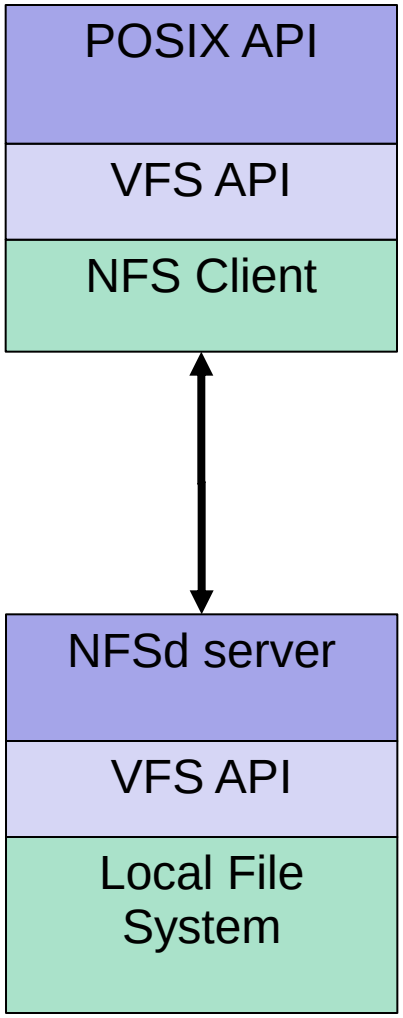
# SAN FS Data Path

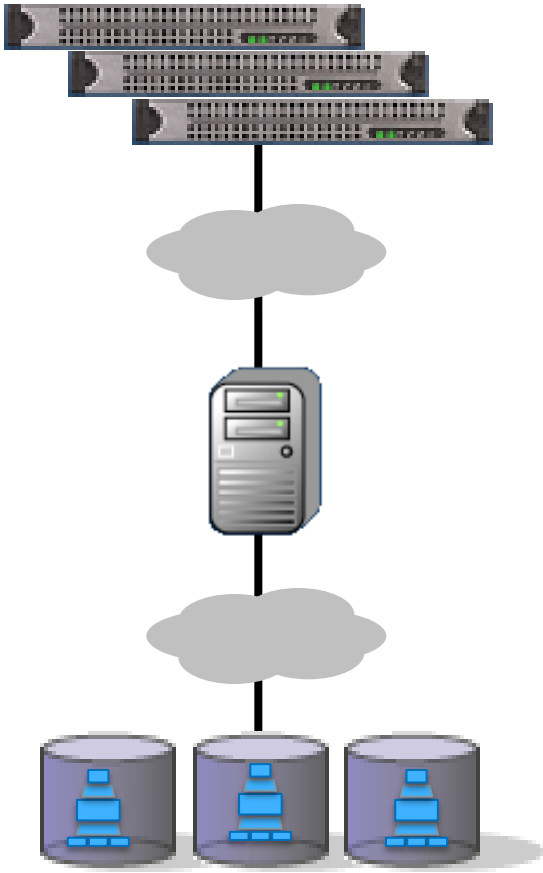
Clients access RAID arrays over the SAN.
Control protocol with metadata server coordinates access to shared disk via locking protocol
Local file system data structures are exposed to the clients




•CXFS (SGI), Polyserve (HP), GFS (RedHat), MPFSi (EMC), Exanet (Dell), QFS (Sun), VMFS (Vmware)
•IBM GPFS has the most scalable implementation

# Network Attached Storage (NAS)

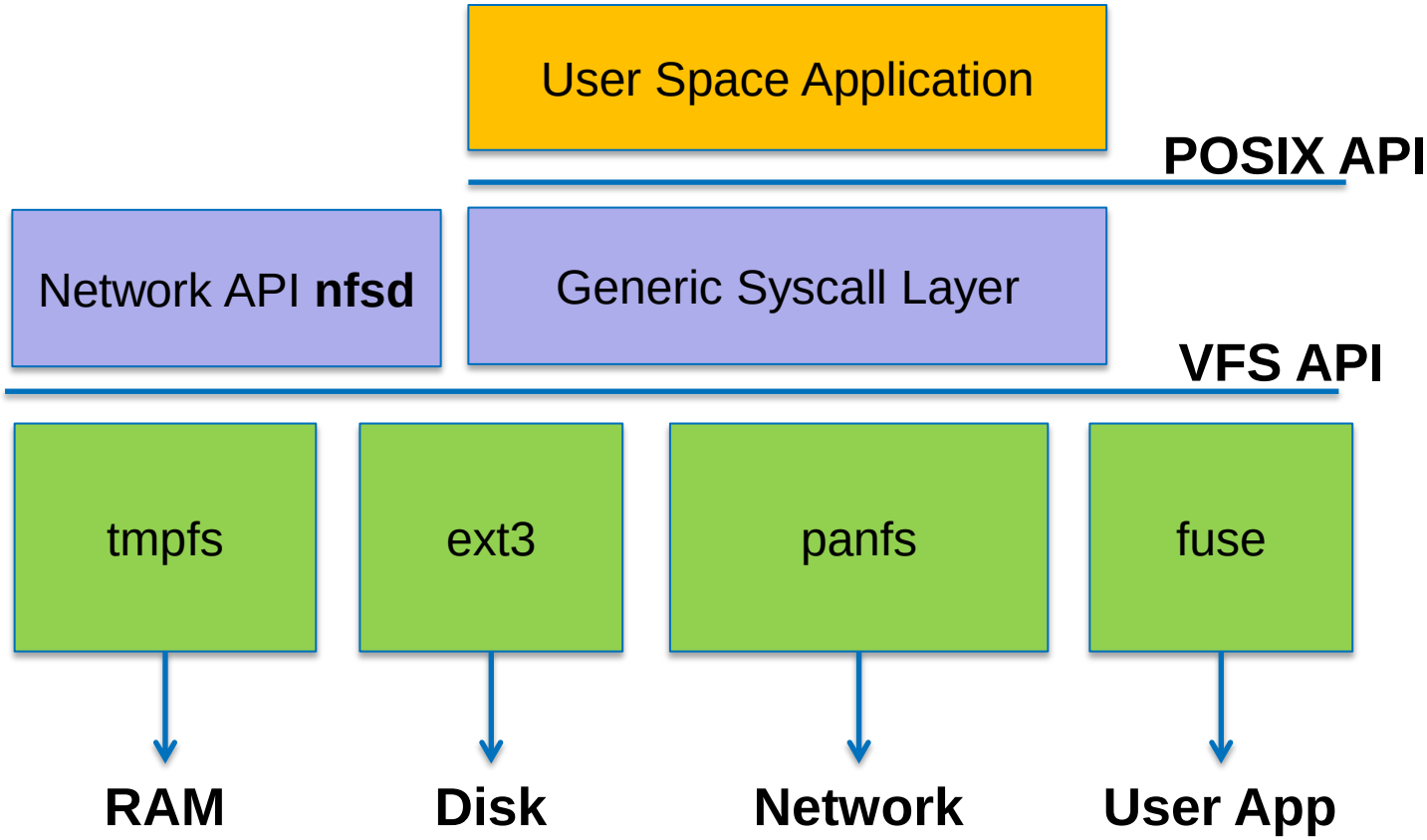


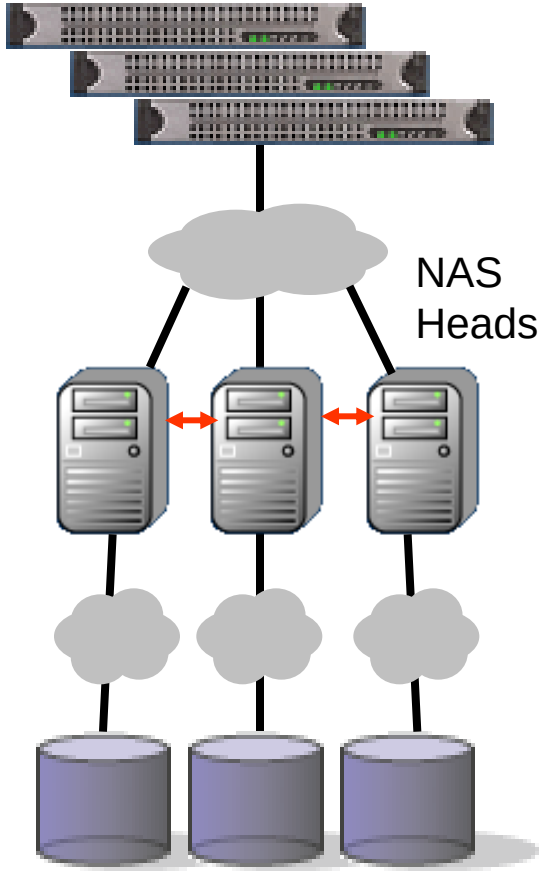
In kernel API layering to support NFS

# Kernel VFS Layer

- Virtual File System kernel API
  - Invented in 1980's when NFS came around
  - Handles multiple local file systems as well

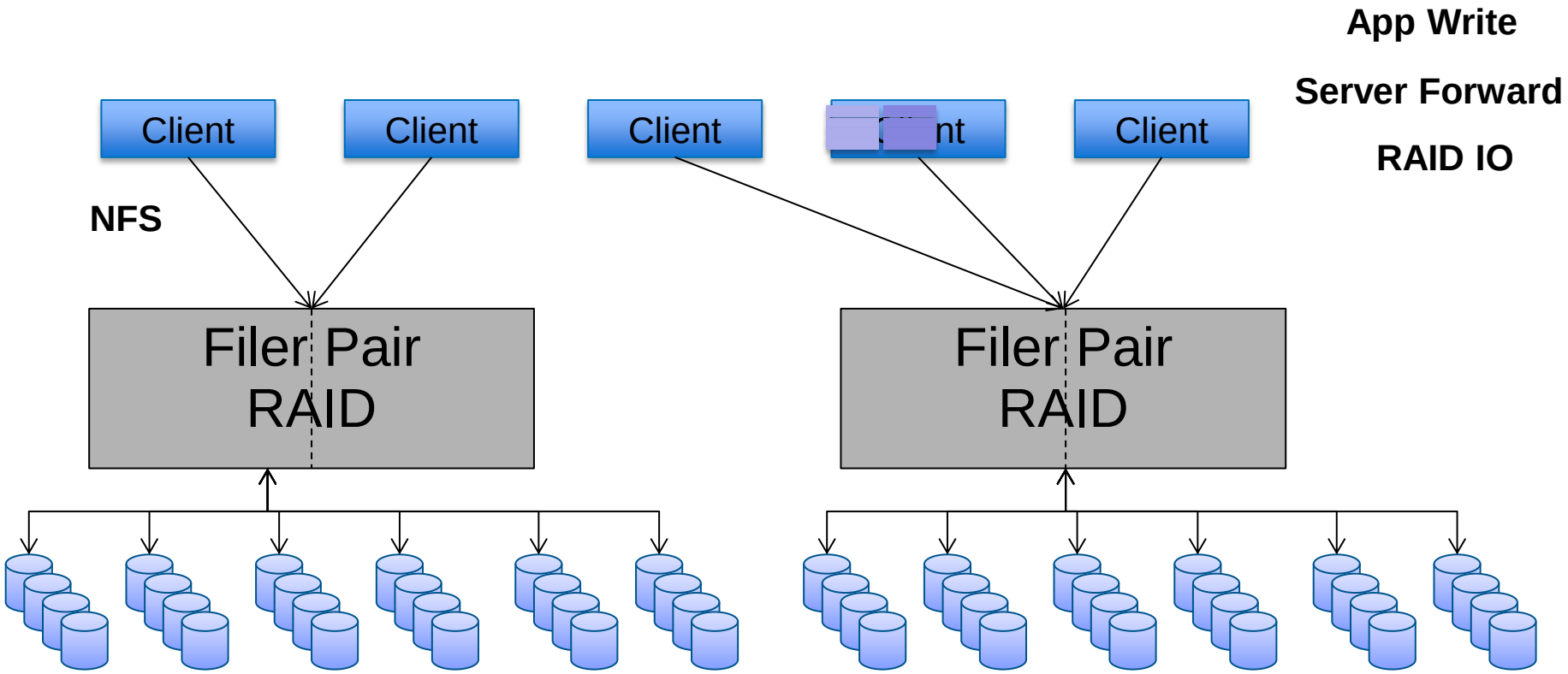


# Clustered NAS



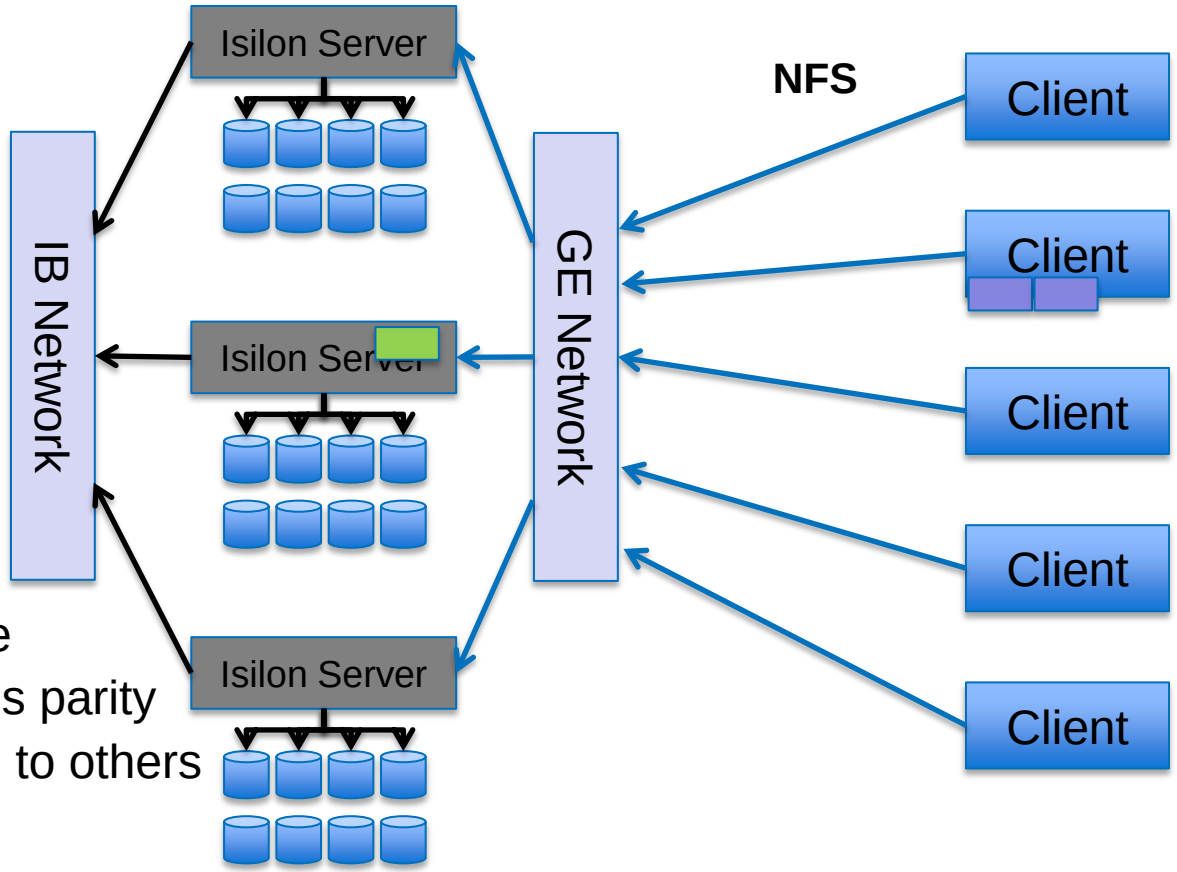
# Clustered NAS Data Path

NFS clients mount a particular Filer. That filer will forward operations to the Filer that owns storage for the file.



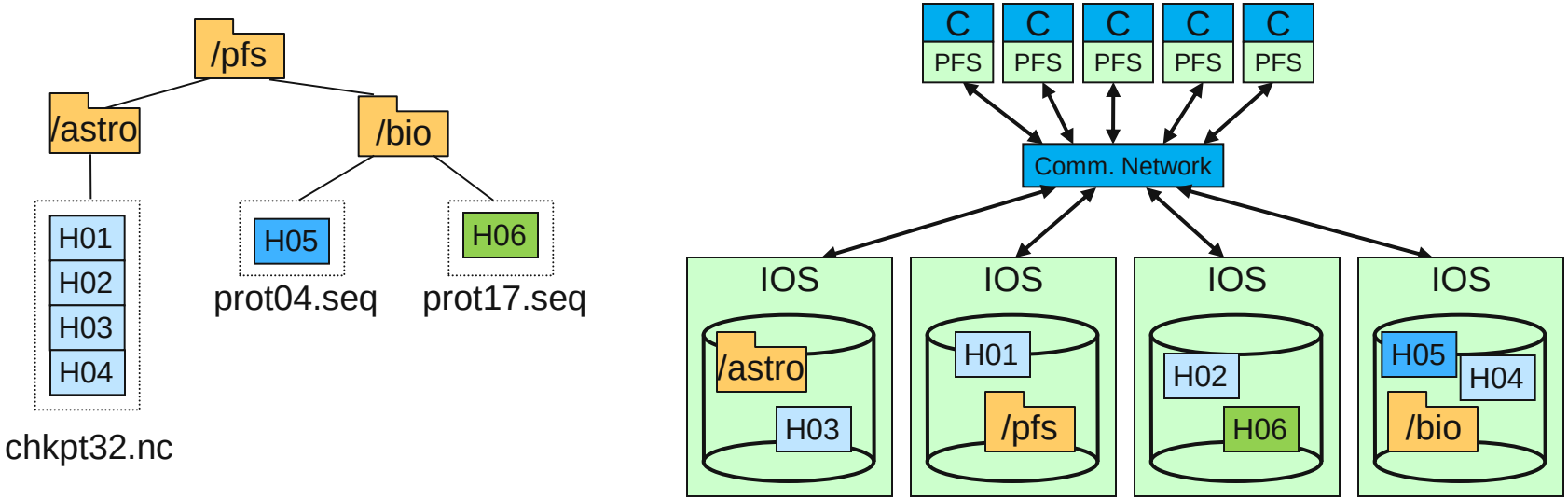
# Isilon Data Path

Isilon nodes compute parity and forward to others
(Ceph, Swift, REST-based systems have a similar strategy)




- Application write
- Server computes parity
- Server forwards to others

# Parallel File Systems




An example parallel file system, with large astrophysics checkpoints distributed across multiple I/O servers (IOS) while small bioinformatics files are each stored on a single IOS

# Data Servers

- All parallel architectures feature some kind of data server, decoupled from the name/metadata server
  - SAN access to a RAID array (SCSI/Block)
  - Network access to proprietary data server
    - Lustre, Ceph, Swift, PVFS, IBRIX, many others
  - iSCSI/OSD access to standardized data server
  - NFS access to NFS server
  - HTTP access to web service
- Data services export some kind of data “object”

# Object Storage Architecture

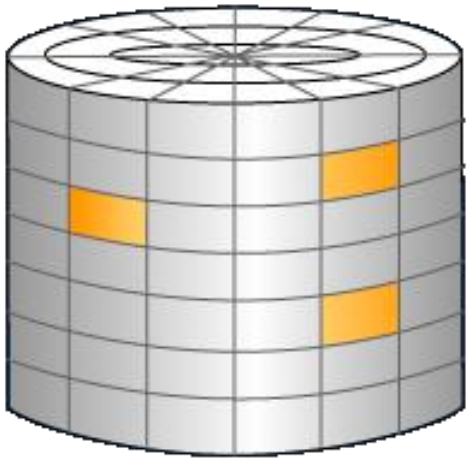
- SAN file systems use Disk interfaces (SCSI)
- NAS systems use File interfaces (VFS)
- Object interface is like a file w/out a name (Inode)
  - iSCSI/OSD standard

## Block Based Device

**Operations**
Read block
Write block

**Addressing**
Block range

**Allocation**
External

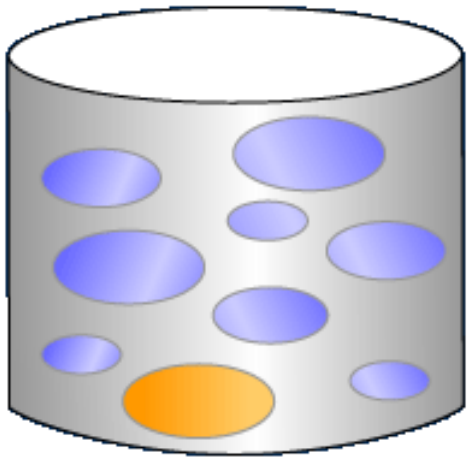


## Object Based Device

**Operations**
Create object
Delete object
Read object
Write object
Get Attribute
Set Attribute

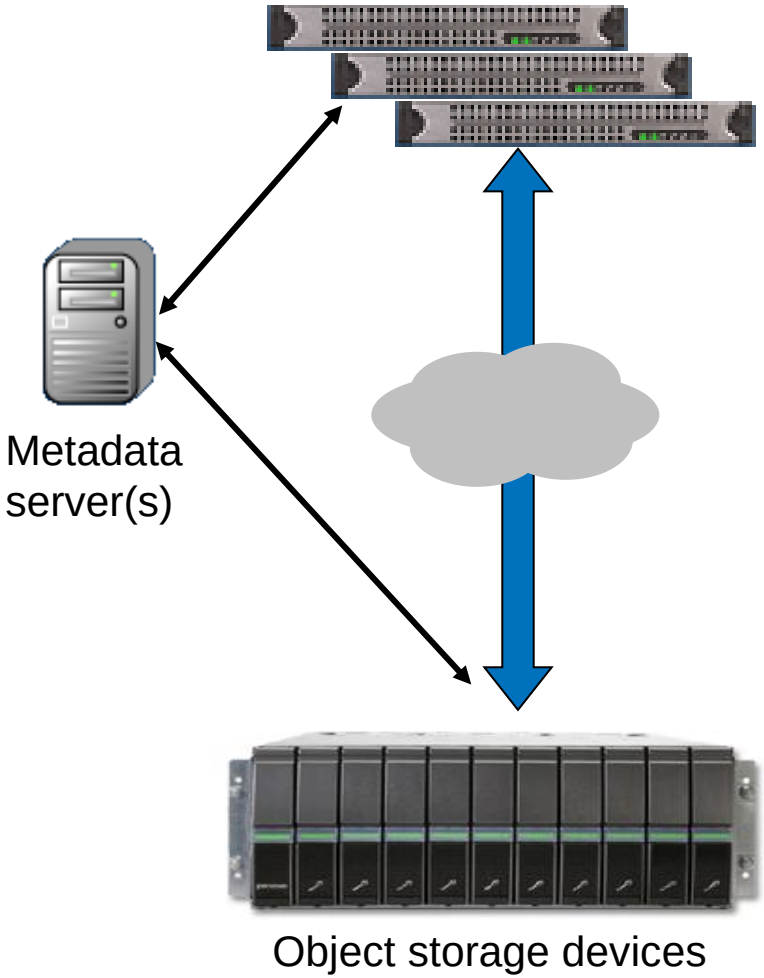
**Addressing**
[object, byte range]

**Allocation**
Internal



# Object-based Storage Clusters

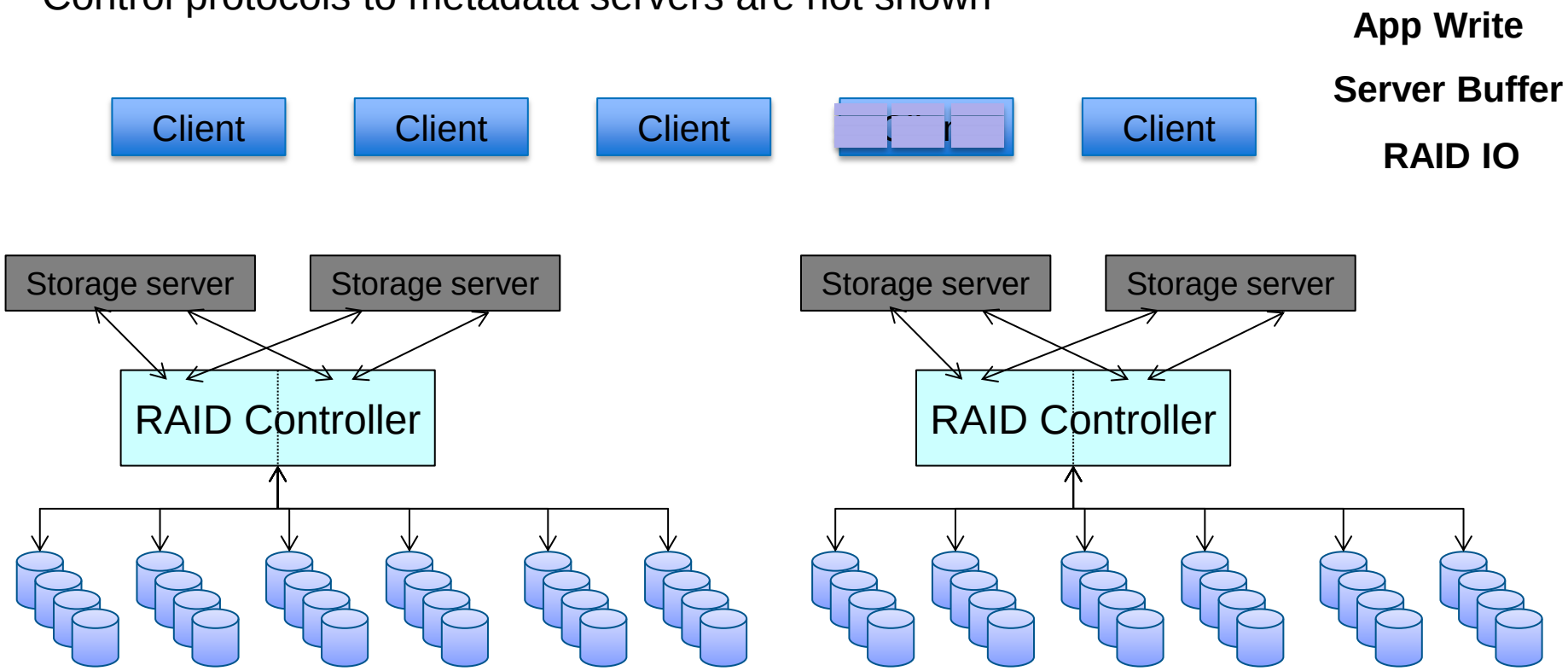
- Lustre, PanFS, Ceph, PVFS
- File system layered over objects
  - Details of block management hidden by the object interface
  - Metadata server manages namespace, access control, and data striping over objects
  - Data transfer directly between OSDs and object-aware clients
- High performance through clustering
  - Scalable to thousands of clients
  - 100+ GB/sec demonstrated to single filesystem



# Lustre and GPFS Data Path

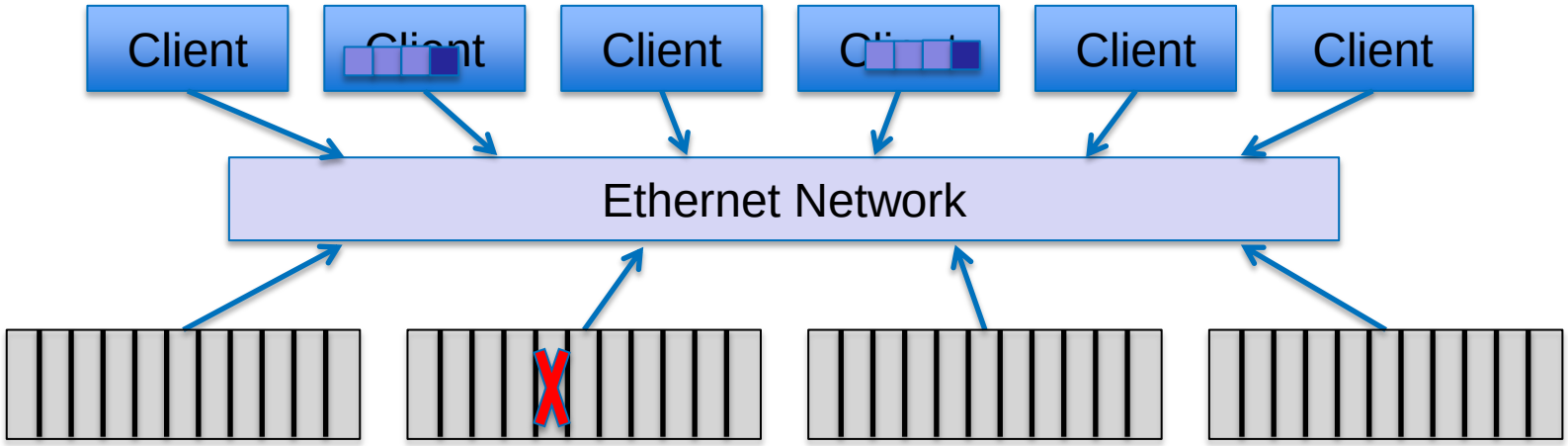
Lustre clients stripe data across Object Storage Servers (OSS), which in turn write data through a RAID controller to Object Storage Targets (OST). OST hides local file system data structures

GPFS has different metadata model but a similar data path
Control protocols to metadata servers are not shown



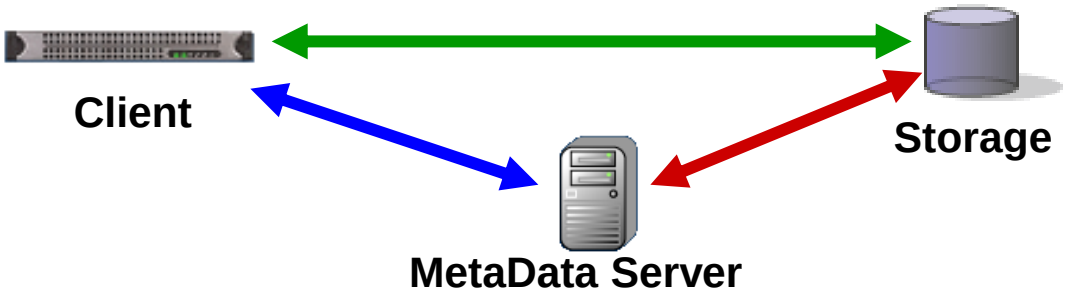
# Panasas Parallel data path

- Data path by-passes RAID controllers and metadata servers
  - Application writes data
  - DirectFlow/pNFS client layer generates redundant data for each stripe
  - Everything is written directly to storage
  - All blades work together on erasure code rebuild



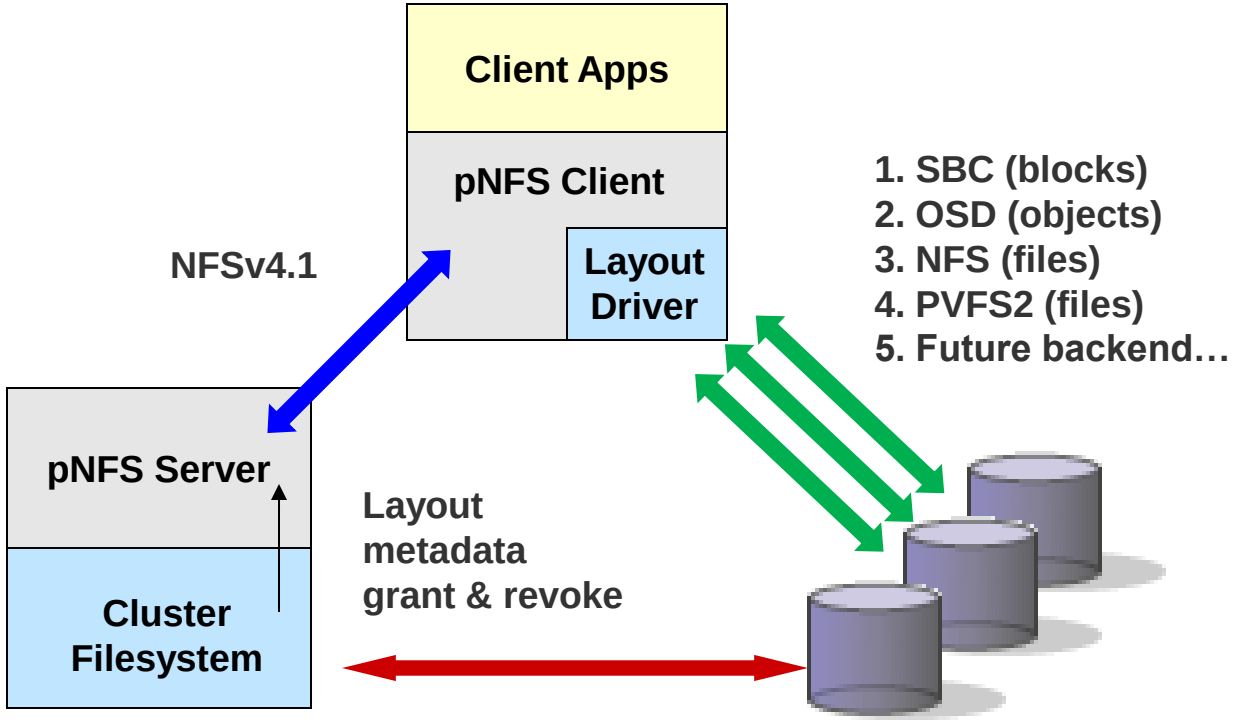
# The pNFS Standard

- The **pNFS** standard defines the NFSv4.1 protocol extensions between the **server** and **client**
- The **I/O** protocol between the **client** and **storage** is specified elsewhere, for example:
  - SCSI **Block** Commands (**SBC**) over Fibre Channel (**FC**)
  - SCSI **Object**-based Storage Device (**OSD**) over iSCSI
  - Network **File** System (**NFS**)
- The **control** protocol between the **server** and **storage** devices is also specified elsewhere, for example:
  - SCSI **Object**-based Storage Device (**OSD**) over iSCSI



# pNFS Client

- Common client for different storage back ends
- Wider availability across operating systems
- Fewer support issues for storage vendors



# Parallel File Systems

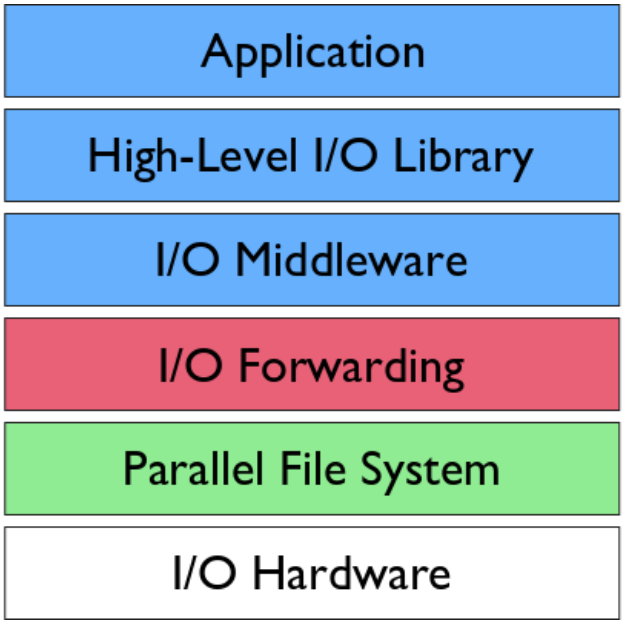
# I/O for Computational Science

**High-Level I/O Library**
maps application abstractions onto storage abstractions and provides data portability.

*HDF5, Parallel netCDF, ADIOS*

**I/O Forwarding**
bridges between app. tasks and storage system and provides aggregation for uncoordinated I/O.

*IBM ciod, IOFSL, Cray DVS*




**I/O Middleware**
organizes accesses from many processes, especially those using collective I/O.

*MPI-IO*

**Parallel File System**
maintains logical space and provides efficient access to data.

*PVFS, PanFS, GPFS, Lustre*

**Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.**

# Goals for this section

- Introduce Lustre, GPFS, Panasas, HDFS
- Compare different approaches to metadata
  - Block Management
  - File Create
- Coordination protocols for correctness
  - Caching
  - Locking
- Fault tolerance protocols for reliability

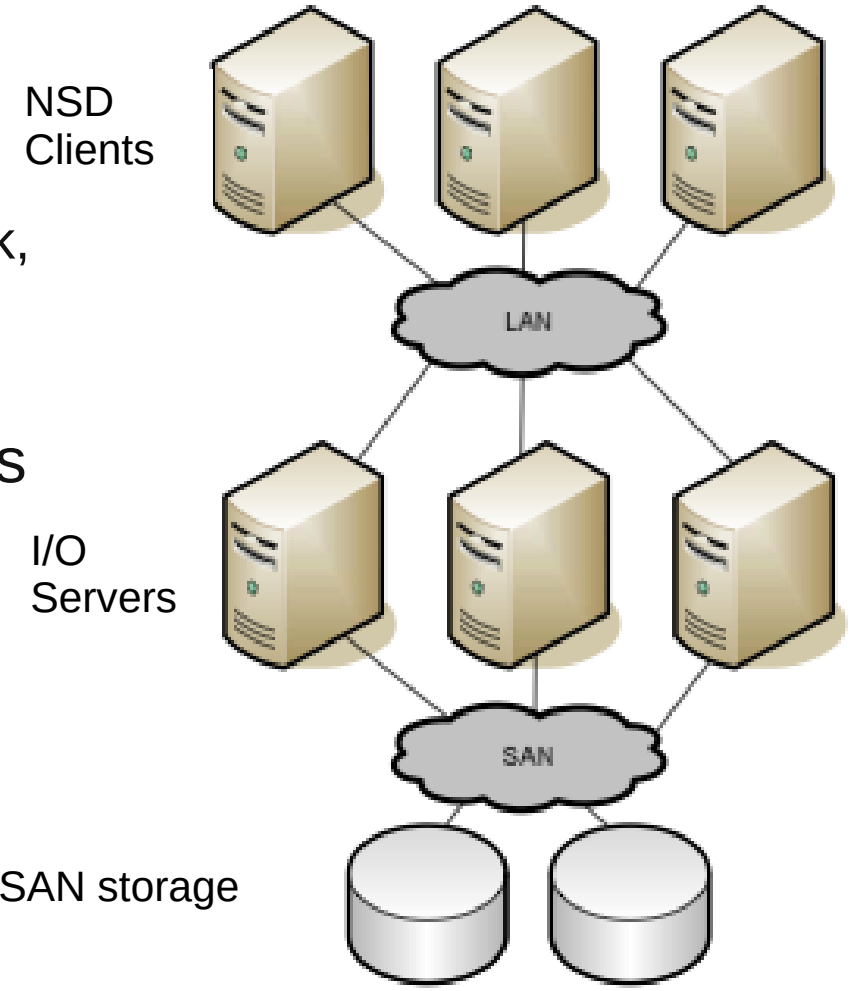
# Production Parallel File Systems

- GPFS, Lustre, Panasas support super computers
  - Cielo, Hopper, MIRA
- HDFS (Google FS) support map reduce (Hadoop)
- Approaches to metadata vary
- Approaches to fault tolerance vary
- Emphasis on features, “turn-key” deployment, vary

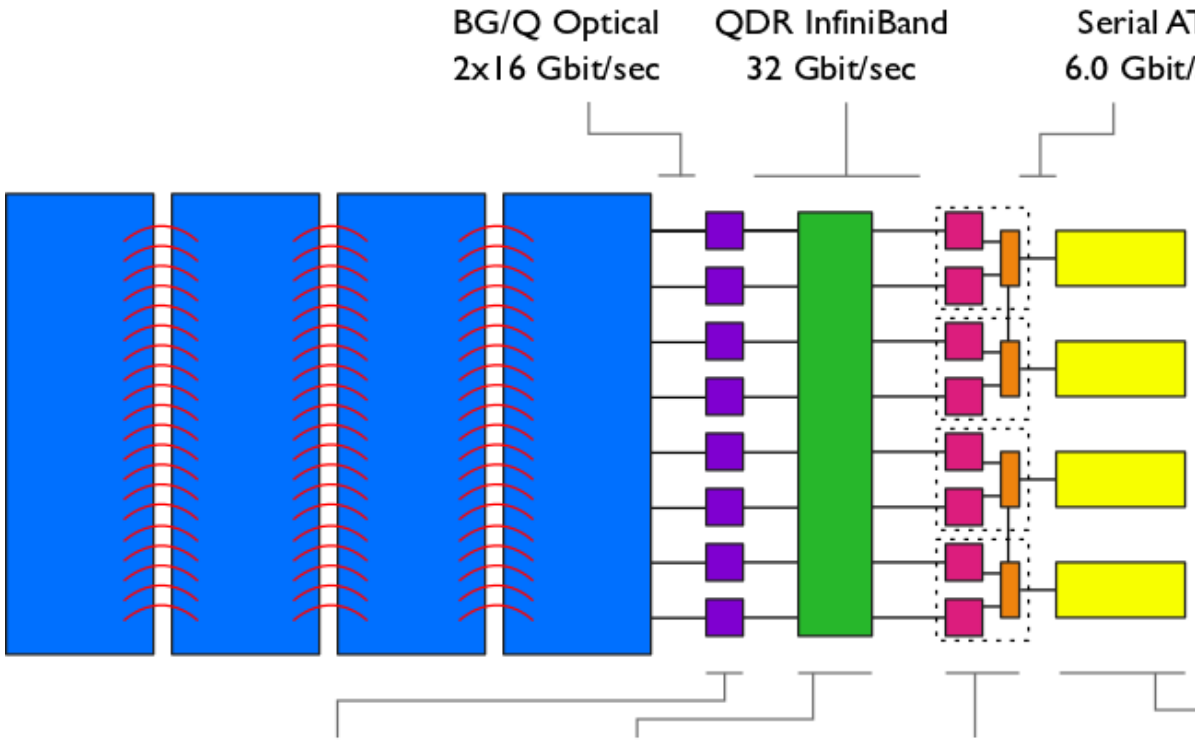

GPFS


# IBM GPFS

- General Parallel File System
- Lots of configuration flexibility
  - AIX, SP3, Linux
  - Direct storage, Virtual Shared Disk, Network Shared Disk
  - Clustered NFS re-export
- Block interface to storage nodes
- Distributed locking
- Blue Gene systems use GPFS



# Blue Gene/Q Parallel Storage System




**Gateway nodes** run parallel file system client software and forward I/O operations from HPC clients.

*384 16-core PowerPC A2 nodes with 16 Gbytes of RAM each*

**Commodity network** primarily carries storage traffic.

*QDR Infiniband Federated Switch*

**Storage nodes** run parallel file system software and manage incoming FS traffic from gateway nodes.

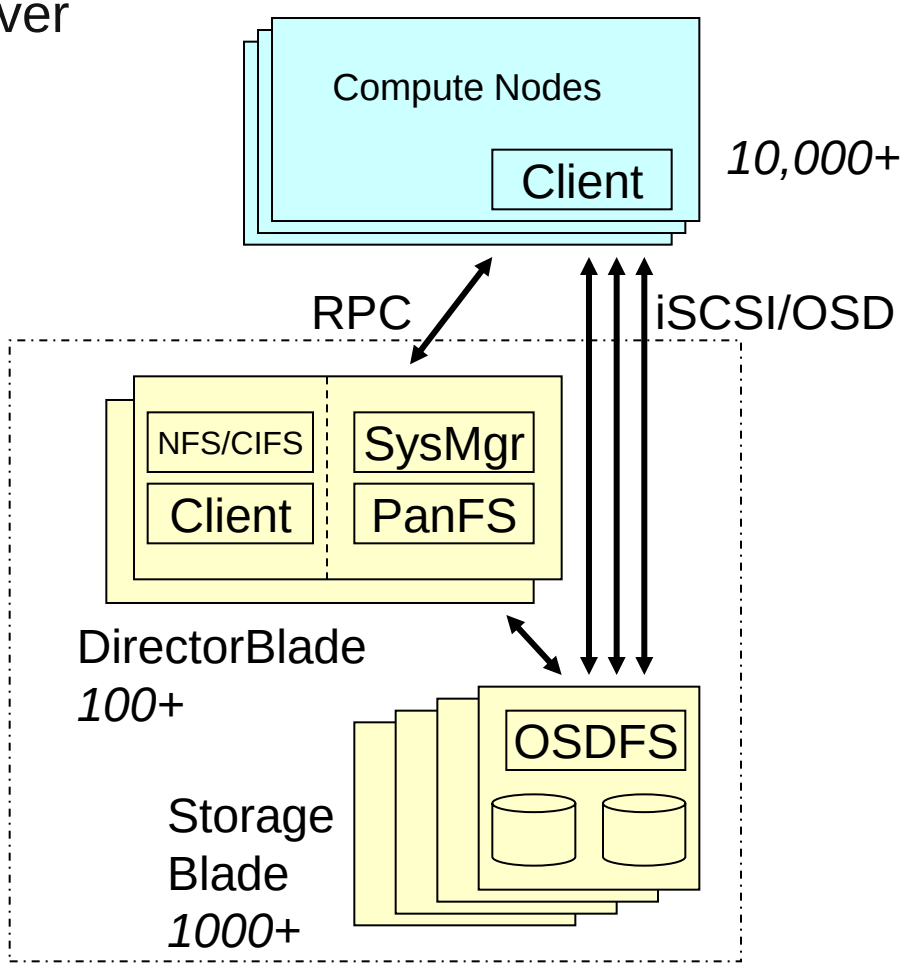
*SFA12KE hosts VM running GPFS servers*

**Enterprise storage** controllers and large racks of disks are connected via InfiniBand.

*16 DataDirect SFA12KE; 560 3 Tbyte drives + 32 200 GB SSD; 16 InfiniBand ports per pair*

# Panasas ActiveScale (PanFS)

- Complete “appliance” solution (HW + SW), blade form factor
  - DirectorBlade = metadata server
  - StorageBlade = OSD
- Coarse grained metadata clustering
- Linux native client for parallel I/O
- NFS & CIFS re-export
- Integrated battery/UPS
- Integrated 10GE switch
- Global namespace



# PanFS at LANL

1 Director, 10 OSD each chassis

104 chassis in largest single system, divided over 12 subnets (lanes)

IO Nodes in each compute cluster route between HSN and 10GE

10 GE

10 GE

...

10 GE

PaScalBB
12 switches

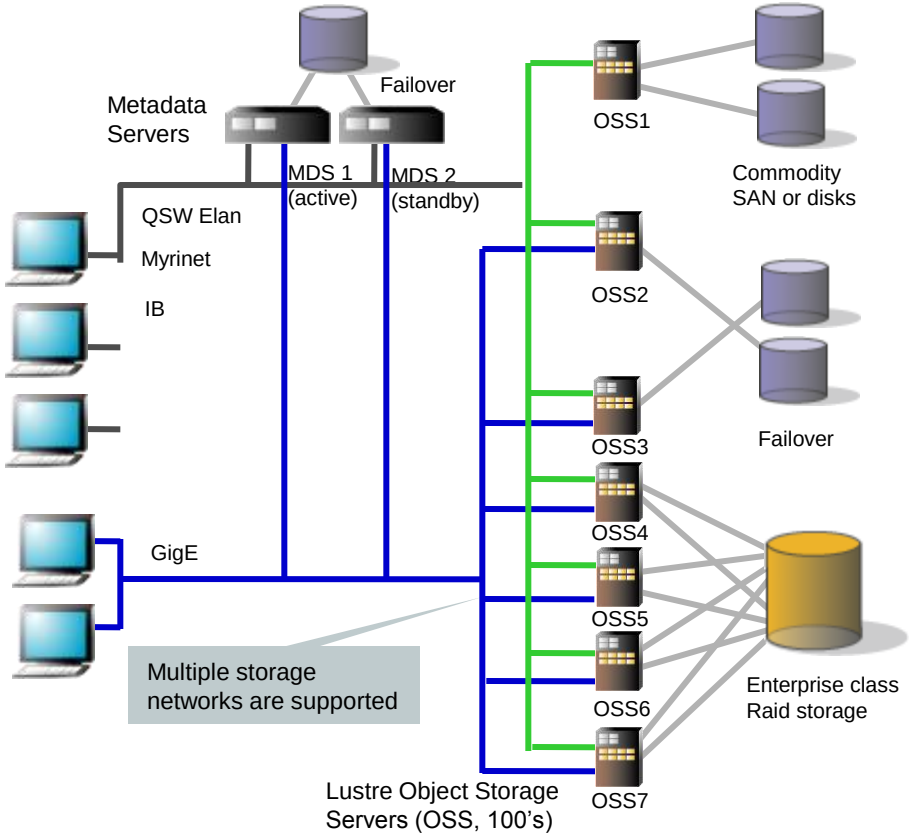
RoadRunner

TLCC

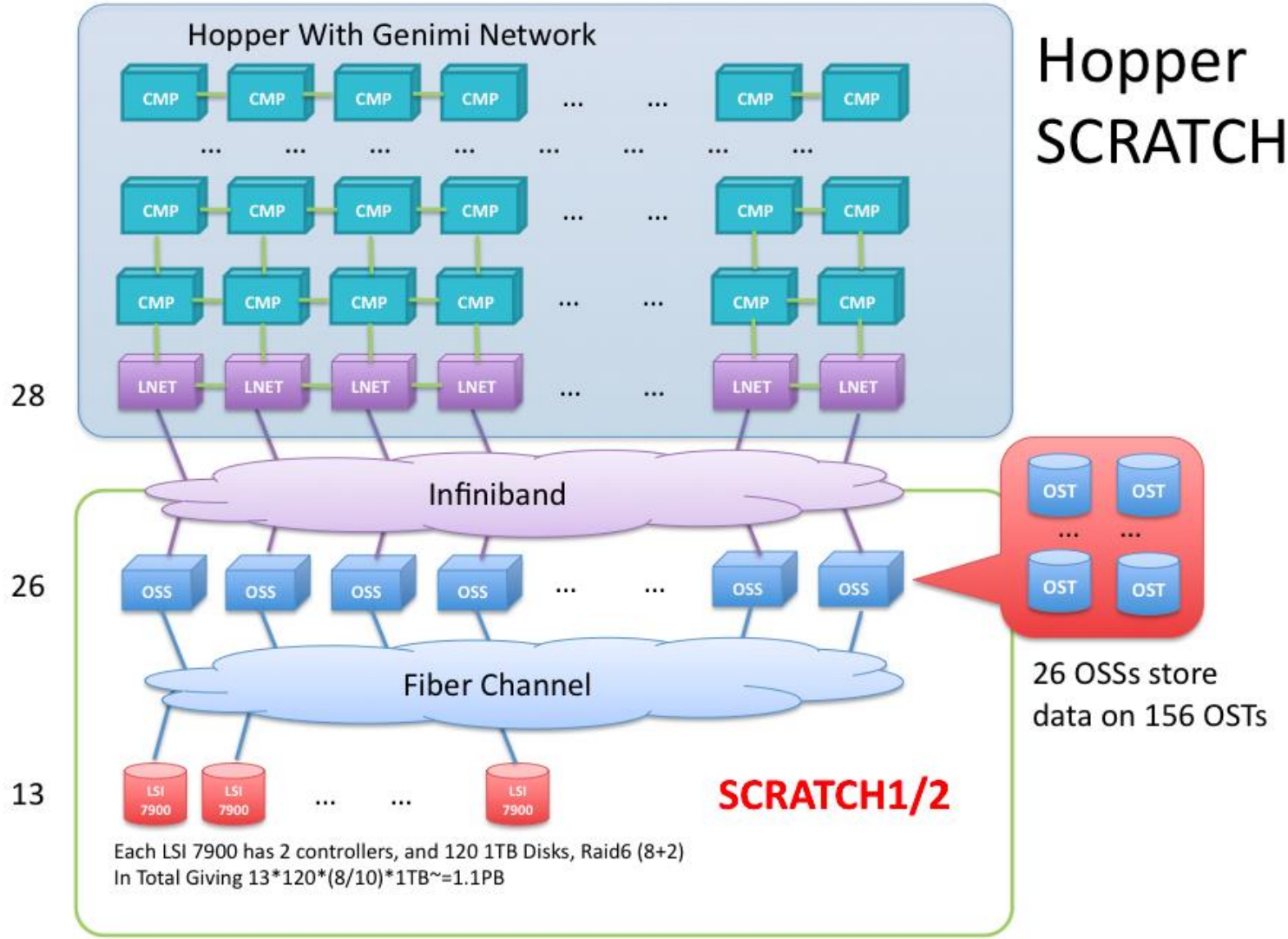
Cielo

# Lustre

- Open source object-based parallel file system
  - Based on CMU NASD architecture
  - Lots of file system ideas from Coda and InterMezzo
  - ClusterFS acquired by Sun, 9/2007
  - Sun acquired by Oracle 4/2009
  - Whamcloud aquired by Intel, 2012
- Originally Linux-based; Sun ported to Solaris
- Asymmetric design with separate metadata server
- Proprietary RPC network protocol between client & MDS/OSS
- Distributed locking with client-driven lock recovery



# Lustre file system on Hopper



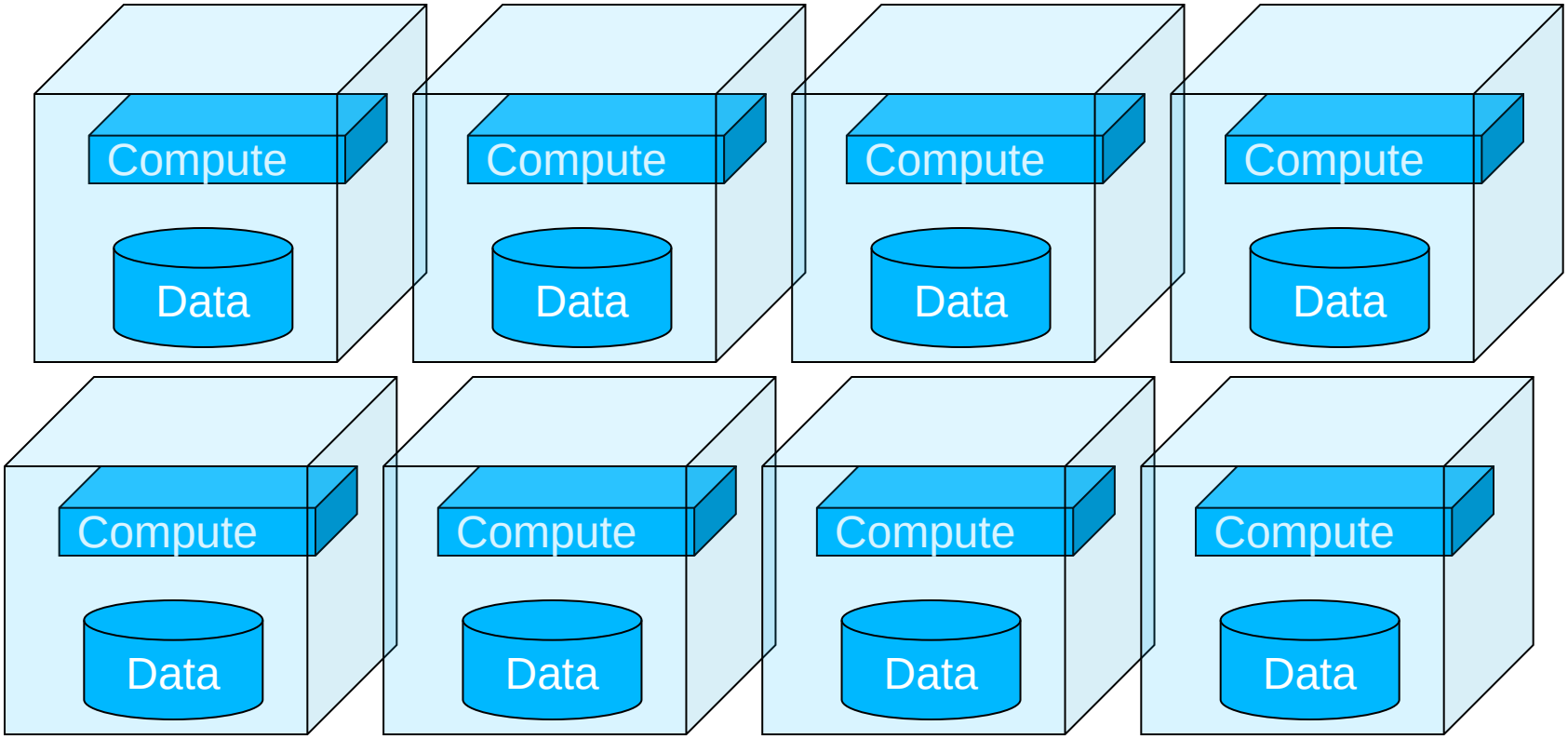

Note: SCRATCH1 and SCRATCH2 have identical configurations.

# HDFS and Cloud Storage

- Hardware Environment
  - Large collections of commodity hardware
  - Compute and Disk on each node
- Classes of software
  - Infrastructure
    - Zookeeper configuration manager, Google Chubby
  - Job Scheduler
    - Hadoop Map Reduce, Google Borg
  - Storage Systems
    - Files: HDFS, Google GFS
    - NoSQL Tables: Cassandra, BigTable, Dremel
    - Object Storage: AWS S3, OpenStack Swift
    - many variants, both open source and proprietary
  - Networking

# Cloud Environment

Large pools of uniform hardware: lots of memory, local disks, good network
Compute job is run on node with copy of its data- sometimes
Dedicated boxes host critical infrastructure services:
    Name Node (memory limited), Job Scheduler, Zookeeper
Network infrastructure supports, e.g., 20,000 servers, 10Gb/s each




Low cost hardware, run until failure, offline service

# HDFS and Google FS

- Data Object is a 64 MB chunk of a file
  - Replicated 3 times on different data nodes
  - Erasure codes often used to increase storage efficiency
- Single Name Node keeps all metadata in main memory
  - HDFS, and original GFS
  - Google Colossus improves on this limitation, as have others
- Non-POSIX semantics
  - Write-once objects (no overwrite)
  - Access via programming library, often in Java
- Exposes location information to Map-Reduce applications
  - Map ships function to nodes with data; runs function on local data
  - Reduce collects results of Map phase and generates answer
  - Important to note that a job does not always run on the same node that has a copy of its data in order to efficiently utilize compute nodes
  - Good network infrastructure makes data locality less important

# NoSQL Data Models (Key-Value Stores)

- "No Schema" tables allow addition of columns, and support rows that have different columns populated
- Timestamps on values allow versioning

⇒Sparse 3D data model where the key is <row,col,time>

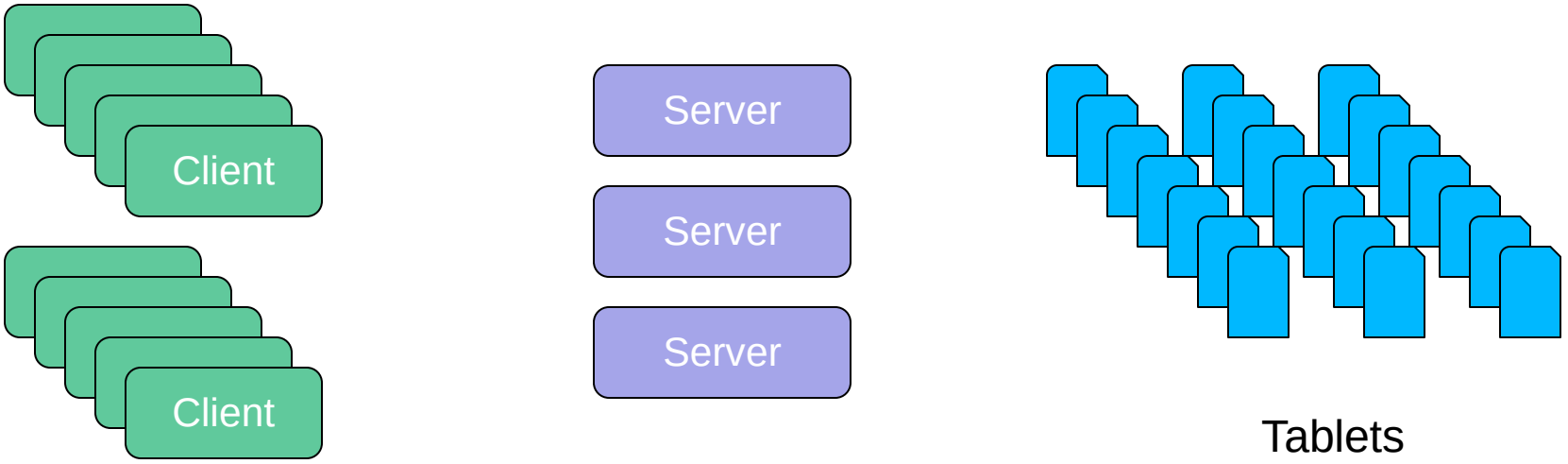
- Google BigTable, Apache Cassandra
- SQL-"like" query languages layered over model
  - Primary limitation is lack of Join (table intersection)
  - Select customerName, orderID
    from Customers and Orders
    where Customer.ID == Orders.customerID
- Scalable implementations
  - Search 1000's of disk drives with a single disk I/O
  - Layered over HDFS or GFS

# Scalable Table Storage

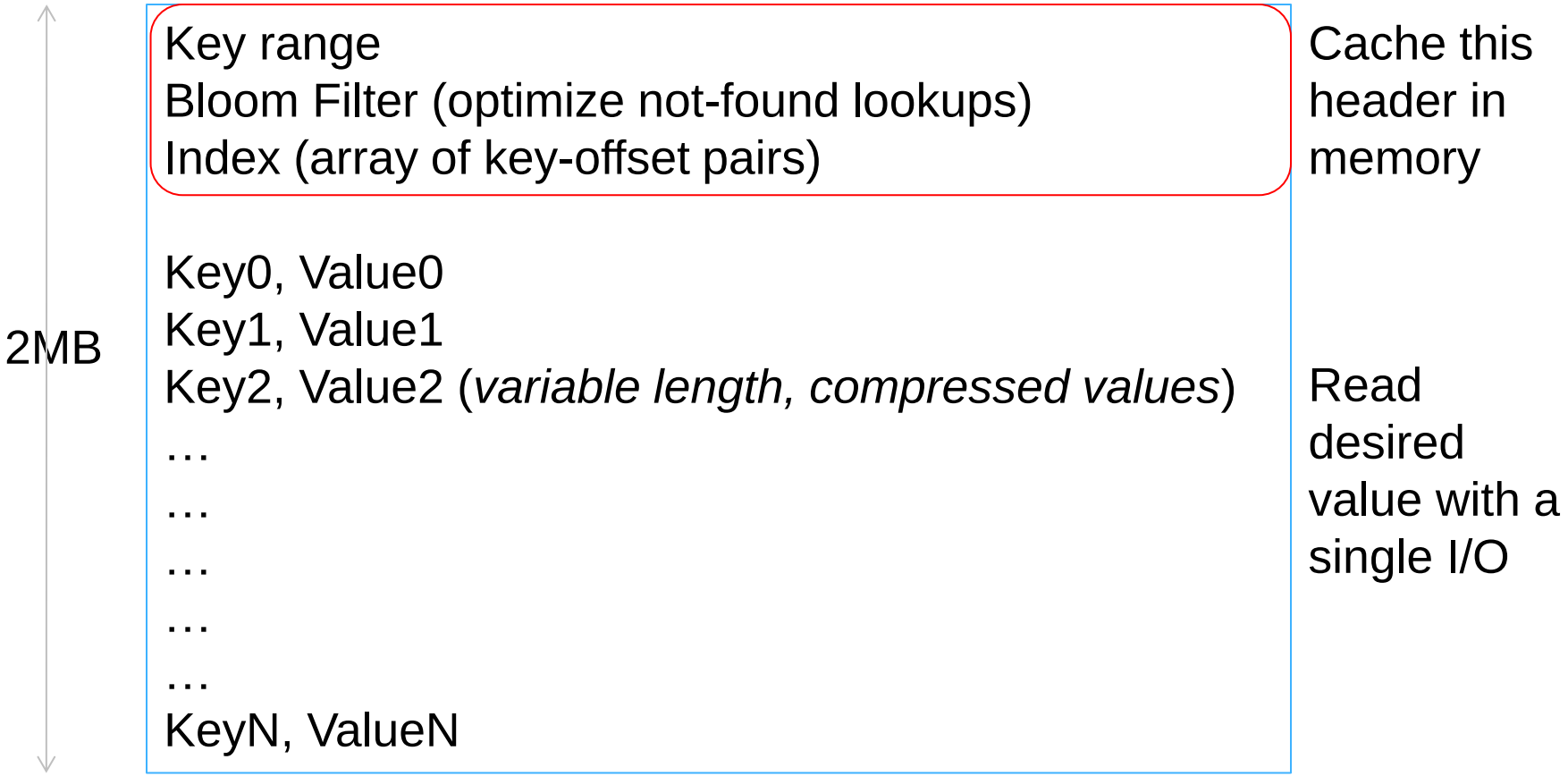
- Log Structured Merge Tree (LSMT)
  - Google SSTable, LevelDB library, many others
- Key-Value
  - Value is a (sparse) row
- Log-structured
  - Updates are logged to append-only chunks
- Merge Tree
  - Chunks (i.e., Tablets) store subset of key-value space
  - Updated tablets get merged and split as needed

# Scaling Horizontally Across Servers

- Partition key space across servers
  - Clients have a ‘shard map’ that defines the partitions
  - Servers own/manage the tablets that store their key range
  - Static or dynamic assignments of servers to keys and tablets

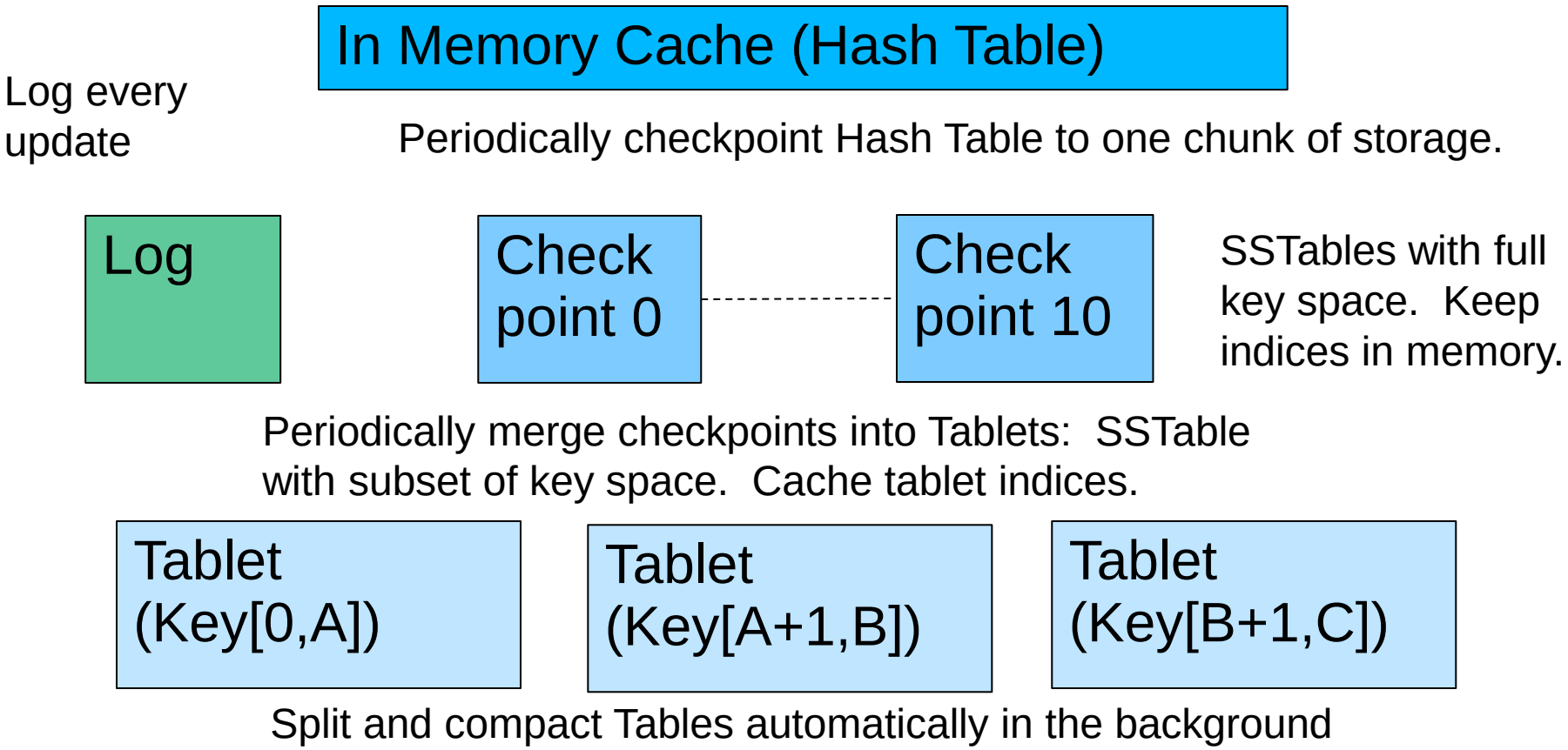


# Sorted String Table (SSTable) Tablet



# Scalable Table Storage

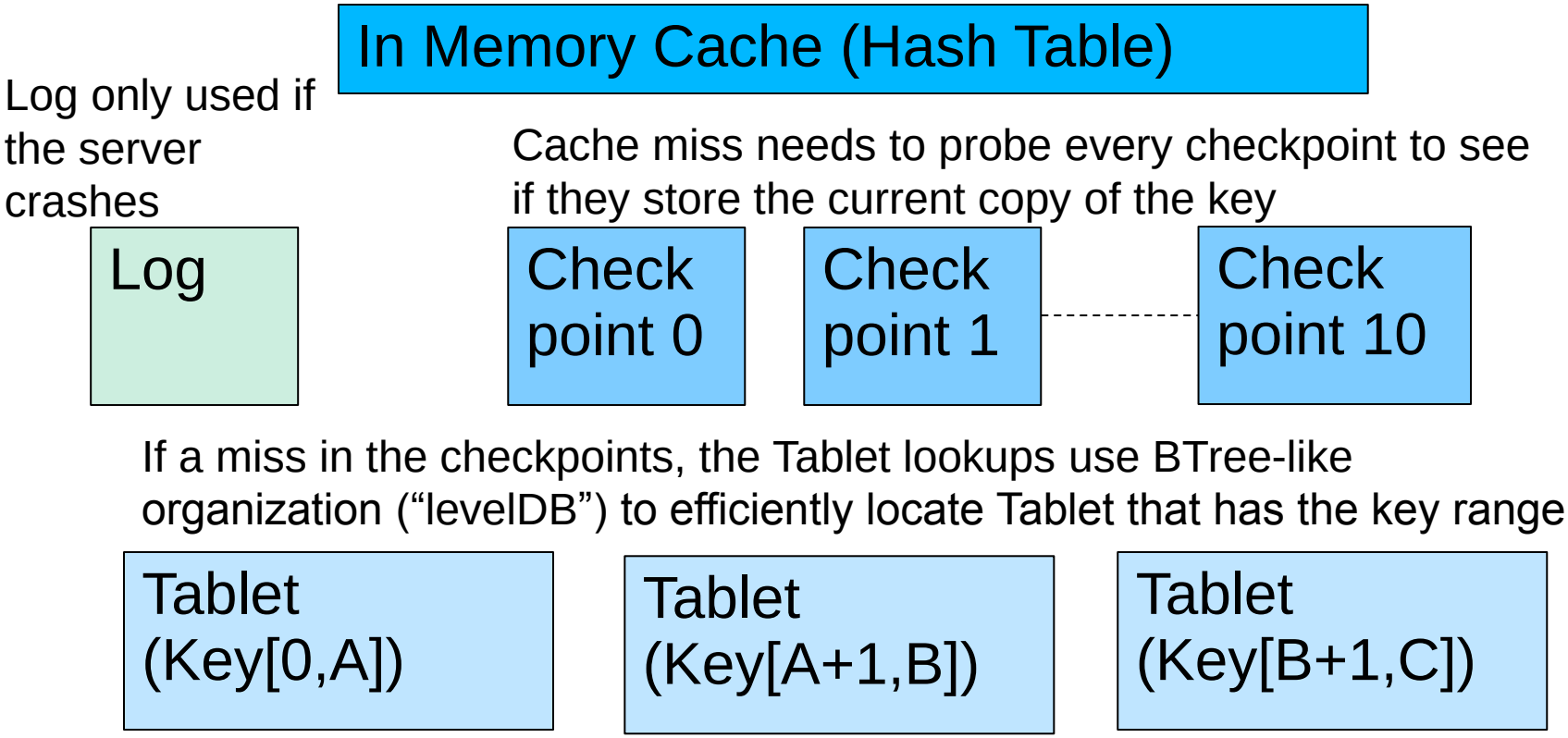
- Log Structured Merge Tree (LSMT)
  - Write path. Each server has cache, log and tablets.



# Scalable Table Storage

- Log Structured Merge Tree
  - Read Path

Hash table supports server's key range. Hot values live here.



# NoSQL Table Summary

- Scalable Key/Value stores
  - Dremel: *Capable of running aggregation queries over trillion-row tables in seconds*
  - http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/36632.pdf
- I/O patterns optimized for re-write of complete chunks in storage
  - Periodic compactions of first level into the Btree
  - Splits as needed to grow Btree
- Ownership of key space is sharded across as many servers as needed (or, available) to get performance
  - In-memory hash tables devoted to a shard of the key space
  - Tablets stored as HDFS file
  - Changing shard ownership is simplified by read-only chunks

# Comparing Parallel File Systems

- Block Management
- How Metadata is stored
- What is cached, and where
- Fault tolerance mechanisms
- Management/Administration

Designer cares about

- Performance
- Reliability
- Manageability
- Cost

Customer cares about

# Block Management

- Delegate block management to data server
  - Panasas, Lustre, PVFS, HDFS
  - Data server uses local file system to store a chunk of a file
    - Panasas OSDFS, Lustre ext4, PVFS (any), HDFS (any)

*OR*

- Distribute block management via locking
  - GPFS, GFS2, Isilon
  - Nodes have their own part of the block map

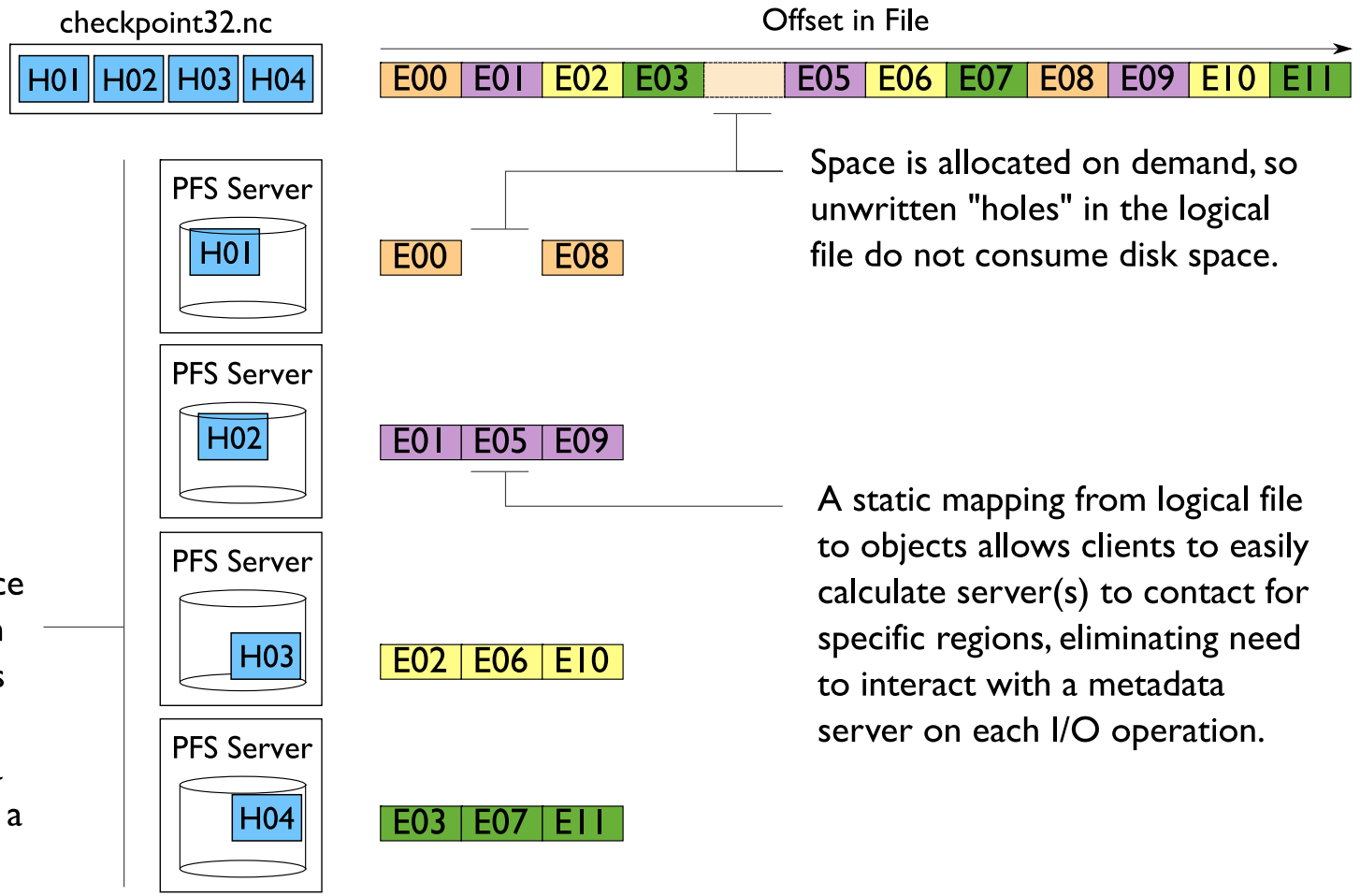
- There are lots of blocks to manage
  - 8 billion 512B sectors on a 4T disk
  - 40 million 4K pages on a 40G SSD
  - Approaches that delegate work to other CPUs are good

# Data Distribution in Parallel File Systems

Logically a file is an extendable sequence of bytes that can be referenced by offset into the sequence.

Metadata associated with the file specifies a mapping of this sequence of bytes into a set of objects on PFS servers.

Extents in the byte sequence are mapped into objects on PFS servers. This mapping is usually determined at file creation time and is often a round-robin distribution of a fixed extent size over the allocated objects.




Space is allocated on demand, so unwritten "holes" in the logical file do not consume disk space.

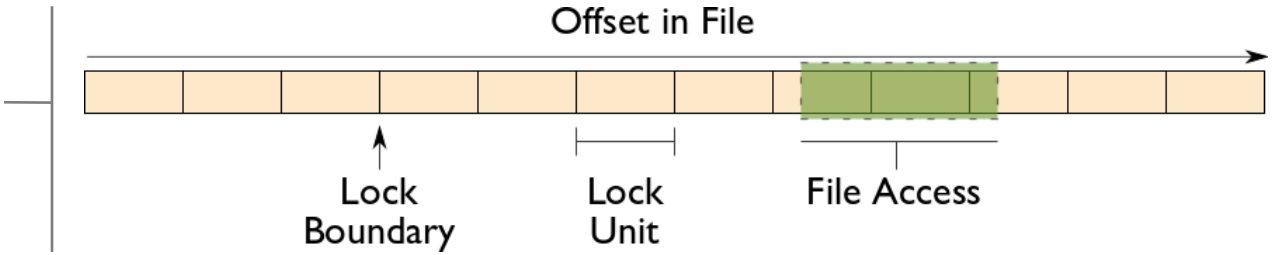
A static mapping from logical file to objects allows clients to easily calculate server(s) to contact for specific regions, eliminating need to interact with a metadata server on each I/O operation.

# Locking in Parallel File Systems

Most parallel file systems use **locks** to manage concurrent access to files

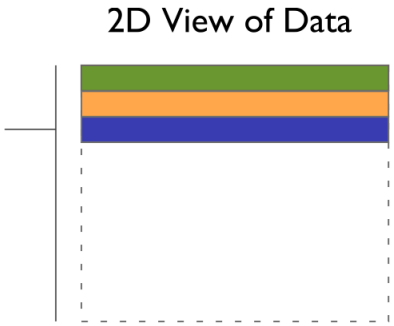
- Files are broken up into lock units
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access
- These locks occur behind scenes: different from locks an application might call explicitly.

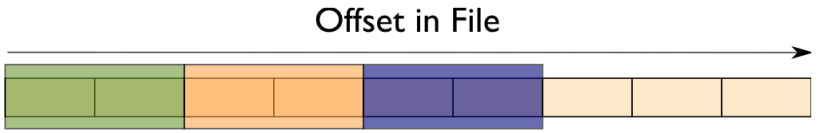
If an access touches any data in a lock unit, the lock for that region must be obtained before access occurs.



# Locking and Concurrent Access

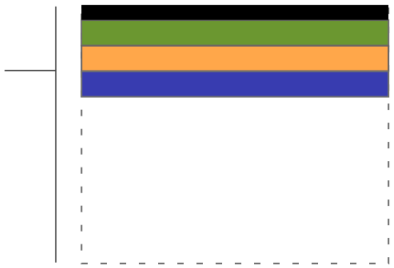
The left diagram shows a row-block distribution of data for three processes. On the right we see how these accesses map onto locking units in the file.

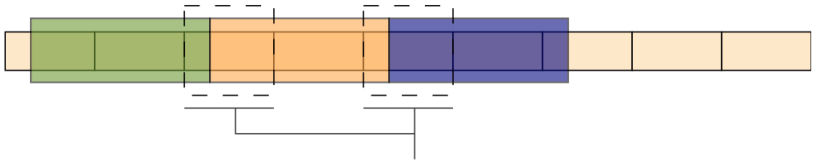



When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.

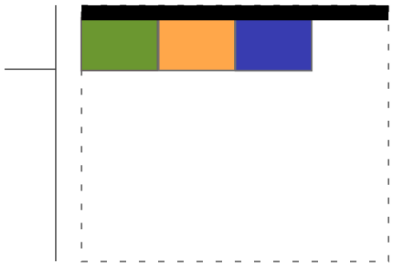
In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.

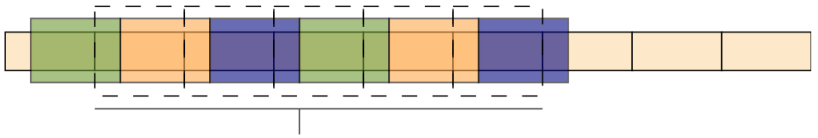




These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

In this example, processes exhibit a block-block access pattern (e.g. accessing a subarray). This results in many interleaved accesses in the file.





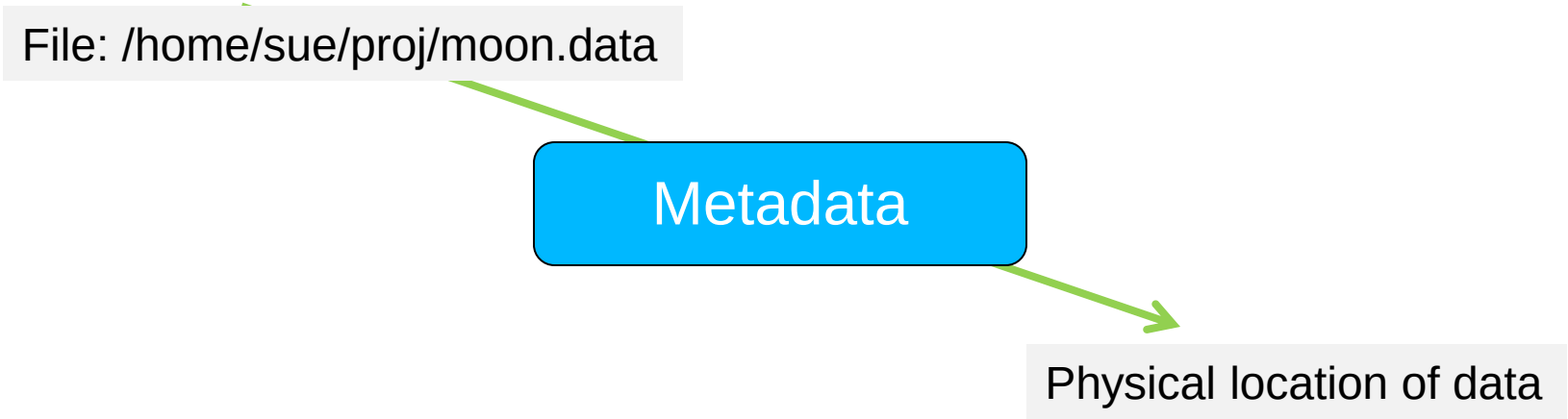
When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

# Delegating locks

- File systems can delegate locking responsibilities to file system clients
  - Even CIFS does it for unshared file access (oplocks)
- Replaces large grain file system lock units (e.g., many blocks) with external (e.g., MPI-based) application synchronization
  - Application agrees not to write the same byte from different processes
  - Network I/O must be byte-aligned, not block-oriented
  - Explicit barriers that flush data to storage and re-sync any caches with storage
- Cassandra tablet server does fine grain locking in memory, only writes private chunks to storage

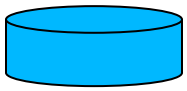
# Meta Data

- Metadata names files and describes where they are located in the distributed system
  - Inodes hold attributes and point to data blocks
  - Directories map names to inodes
- Metadata updates can create performance problems
- Different approaches to metadata are illustrated via the File Create operation

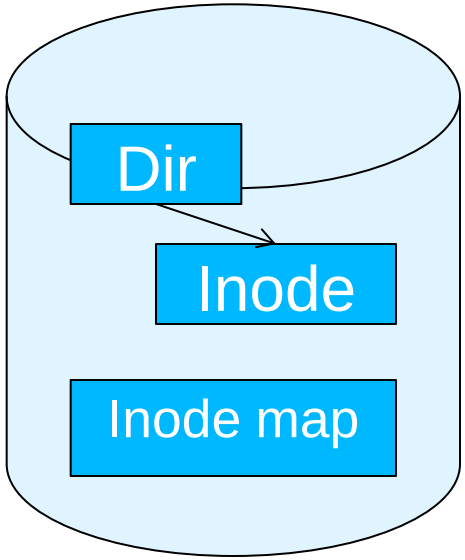


# File Create on Local File System

- 4 logical I/Os
  - Journal update
  - Inode allocation
  - Directory insert
  - Inode update
- Performance determined by journal updates
  - Or lack there of – Journal protects integrity after a crash
  - Details vary among systems



Fast Journal device (SSD)

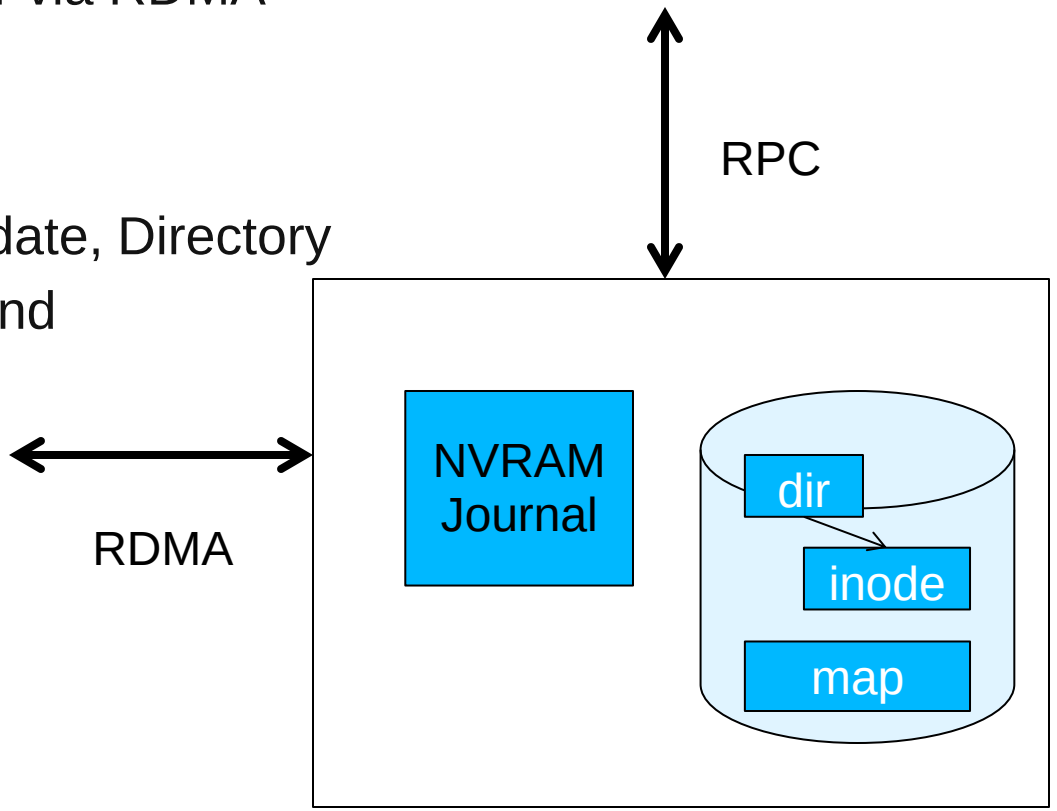



| Inode | Name |
|---|---|
| 0017 | Fred |
| 2981 | Yoshi |
| 7288 | Racheta |

Directory

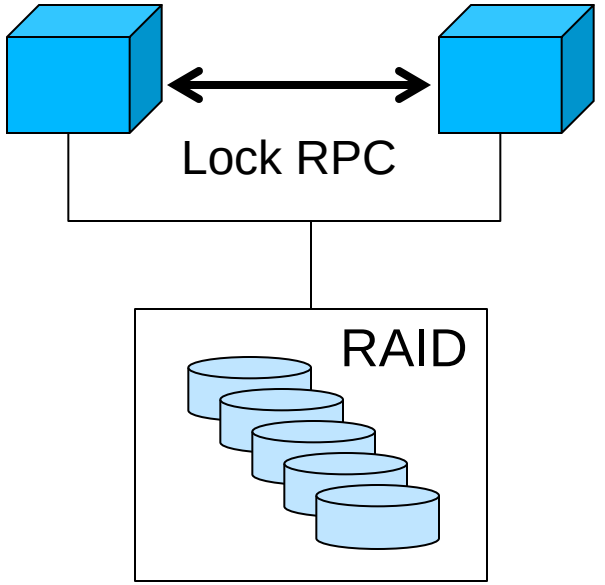
# File Create on NFS Server

- RPC
  - Client to NFS server RPC
- NVRAM update
  - Mirrored copy on peer via RDMA
- Reply to client
- Local I/O
  - Inode alloc, Inode update, Directory
  - Done in the background
- Performance from
  - NVRAM+RDMA



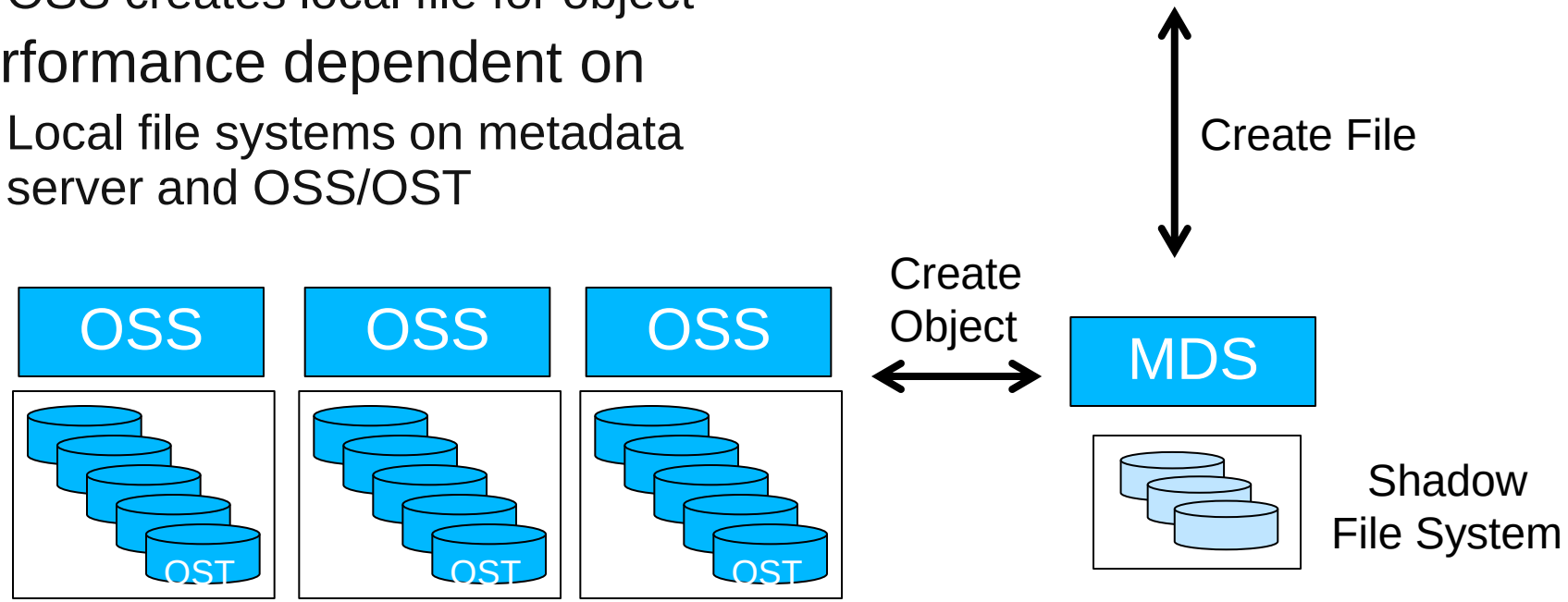
# File Create on SAN FS

- Lock RPC for inode allocation
- Lock RPC for directory insert
- Journal update
- SAN I/O for inode and directory
  - Done in the background
- Performance dependent on
  - Journal updates
  - Lock manager updates
  - GPFS cache of lock ownership
  - SAN I/O



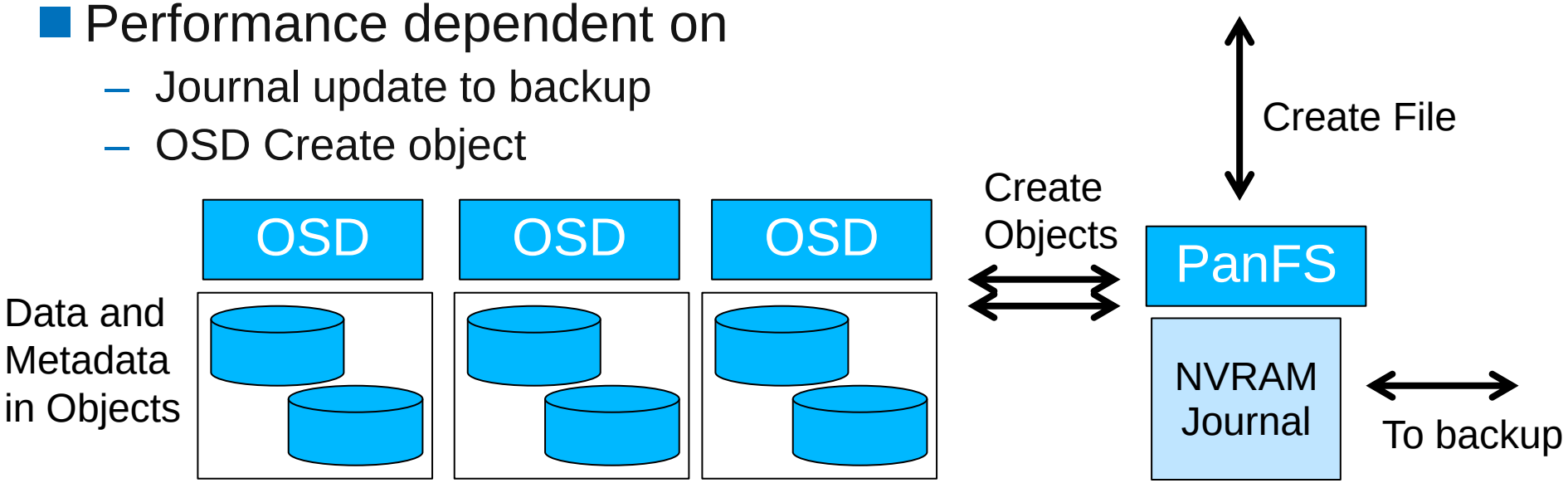
# File Create on Lustre

- Client to Server RPC
- Server creates local file to store metadata
  - Journal update, local disk I/O
- Server creates container object(s)
  - Object create transaction with OSS
  - OSS creates local file for object
- Performance dependent on
  - Local file systems on metadata server and OSS/OST



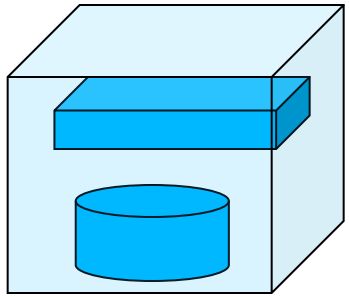
# File Create on PanFS

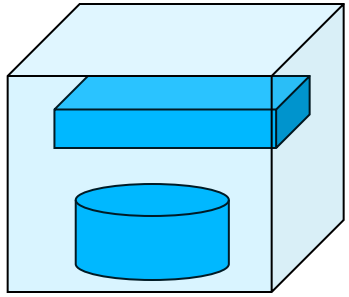
- Client to Server RPC
- MDS updates journal in NVRAM (locally and on backup)
- MDS creates 2 container objects (iSCSI/OSD Create Object)
  - OSDFS journals object create in NVRAM
  - MDS annotates objects with its own metadata (as attributes)
- Reply to client
- Update directory (mirrored OSD write) in background
- Performance dependent on
  - Journal update to backup
  - OSD Create object

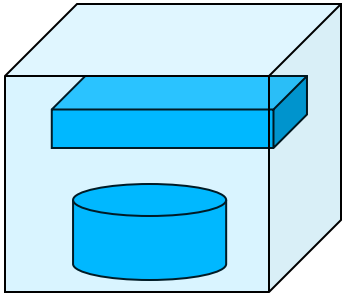


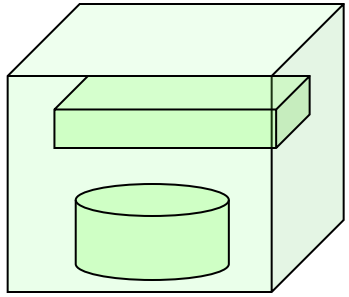
# File Create on HDFS

- RPC to Name Node
- Key-value store update
- Container Create
  - One on the client node
  - One replica “in rack”
  - One more replica “out of rack”
- Performance depends on
  - Metadata memory size
  - Local file system updates on Data Nodes
  - # Copies created before sending reply

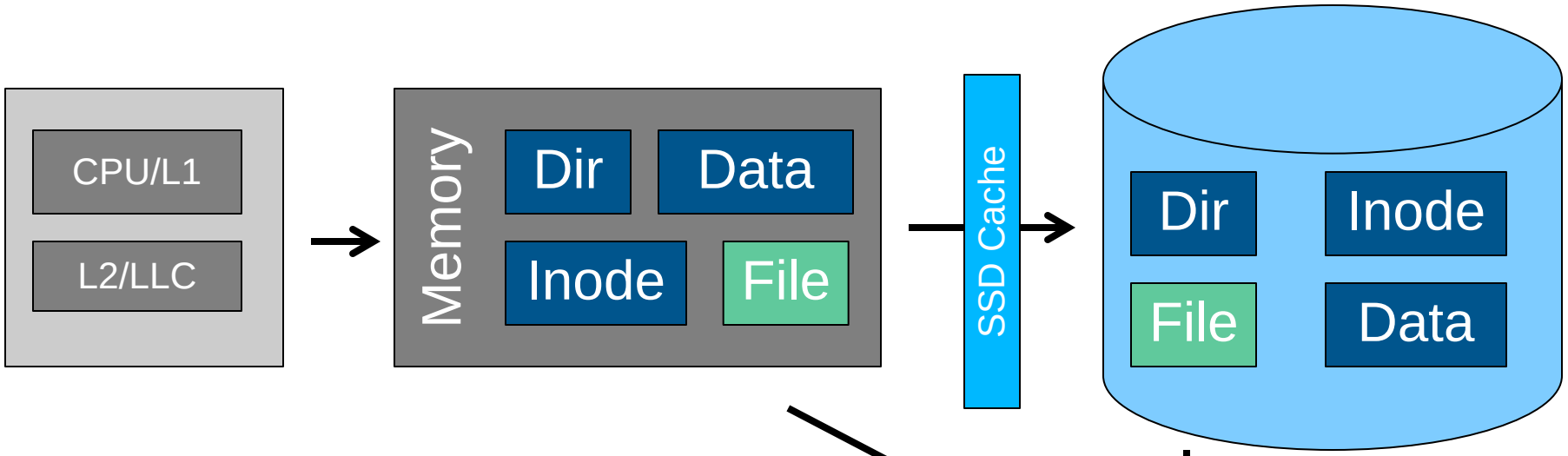


# Caches




DRAM words cached on-chip in L1/L2 cache

Local disk block cached in main memory

Network remote data block cached in memory

Network remote data block cached on local disk

Spinning disk data block cached on SSD

Data object attributes cached by metadata service

File attributes cached by file system client

# Caching

- In data server memory, in front of disk
  - All systems do this for “clean” read data
  - Delayed writes need battery protected memory
    - RAID Controller, with mirroring
    - Panasas StorageBlades, with integrated UPS
- In file system client, in front of network
  - Need *cache consistency* protocol
  - GPFS, DLM lock ownership protocol on blocks
  - Lustre, some caching with DLM protocol
  - Panasas, exclusive, read-only, read-write, concurrent write caching modes with callback protocol
  - PVFS, read-only client caching
  - HDFS, read-only caching of immutable objects

# Fault Tolerance

Combination of hardware and software ensures continued operation in face of failures

- Disk Failures
  - Block RAID
  - Object Erasure Codes
- Service Failure (software crash)
  - Local journal
  - Heartbeat protocols
- Server Failures (hardware crash)
  - Shared disk file system
  - Journal replication to backup buddy
- Client Failures
  - Fencing (SAN zoning, Object Capabilities)
  - GPFS clients members of global quorum

# Journals

- A Journal records what the system is going to do
  - Record is made before file system is modified
  - Protects local disk operations and remote objects operations
- System consults journal after a crash
  - Cleans up the file system w/out expensive sweep
  - Critical for correctness in the face of faults in the system
- Physical device for journal dictates performance
  - No journal: fastest, but you have dirty crashes
  - NVRAM replicated to backup
  - 15K RPM disk
  - RAID controller with battery-backed cache
  - SSD

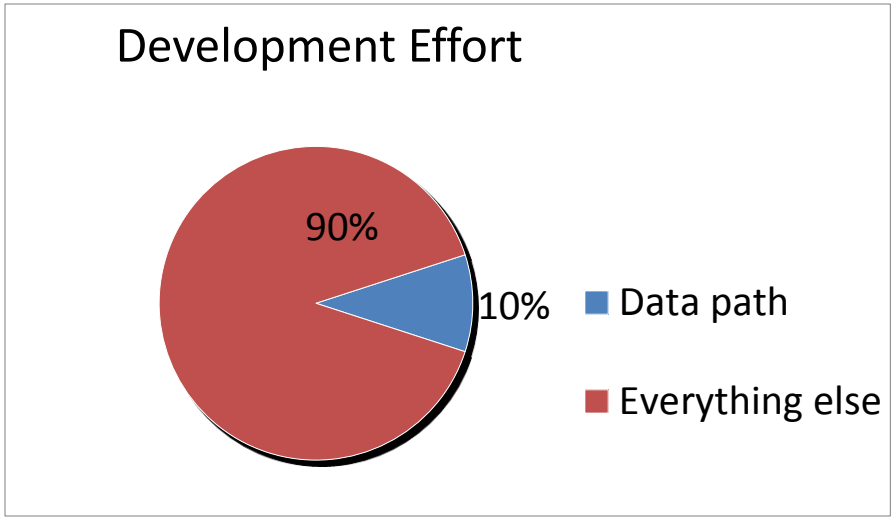
# Design Comparison

| | GPFS | HDFS | Panasas | Lustre |
|---|---|---|---|---|
| **Block mgmt** | **Shared block map** | Object based | Object based | Object based |
| **Metadata location** | **FS Disk Structures** | **Name Node Key-Value** | **Object Attributes** | **Shadow File System** |
| **Metadata written by** | Client | Server | **Client, server** | Server |
| **Cache coherency & protocol** | Coherent; distributed locking | **Cache immutable/RO data only** | Coherent; callbacks | Coherent; distributed locking |
| **Reliability** | Block RAID | **Triplication** | **Object RAID** | Block RAID |

# Other Issues

What about…

- Monitoring & troubleshooting?
- Backups?
- Snapshots?
- Disaster recovery & replication?
- Capacity management?
- System expansion?
- Retiring old equipment?
- Limitations of POSIX?



# Other File Systems

- Ceph (UCSC)
  - OSD-based parallel filesystem
  - Dynamic metadata partitioning between MDSs
  - OSD-directed replication based on CRUSH distribution function (no explicit storage map)
- GlusterFS(Gluster)
  - cloud storage
- Fraunhofer (FhGFS)
  - parallel file system
- VMFS (Vmware)
  - SAN FS optimized for storing VM images
- Clustered NAS
  - NetApp GX, Isilon, BlueArc, etc.
- PVFS – OrangeFS
  - User Space Parallel File System optimized for HPC

# In-System Storage

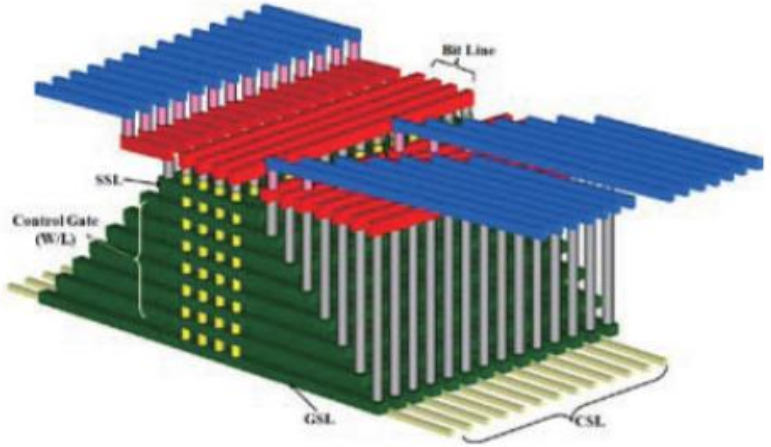
Many thanks to:

**Ning Liu**
Illinois Institute of Technology

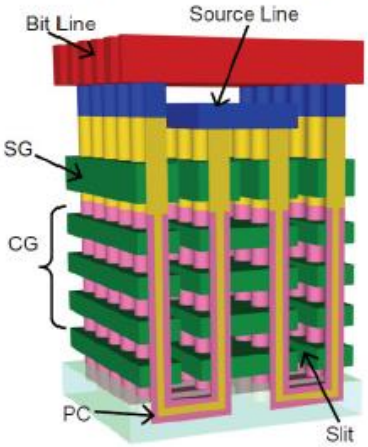
**Jason Cope**
DataDirect Networks

**Chris Carothers**
Rensselear Polytechnic Institute

# What is In-System Storage?

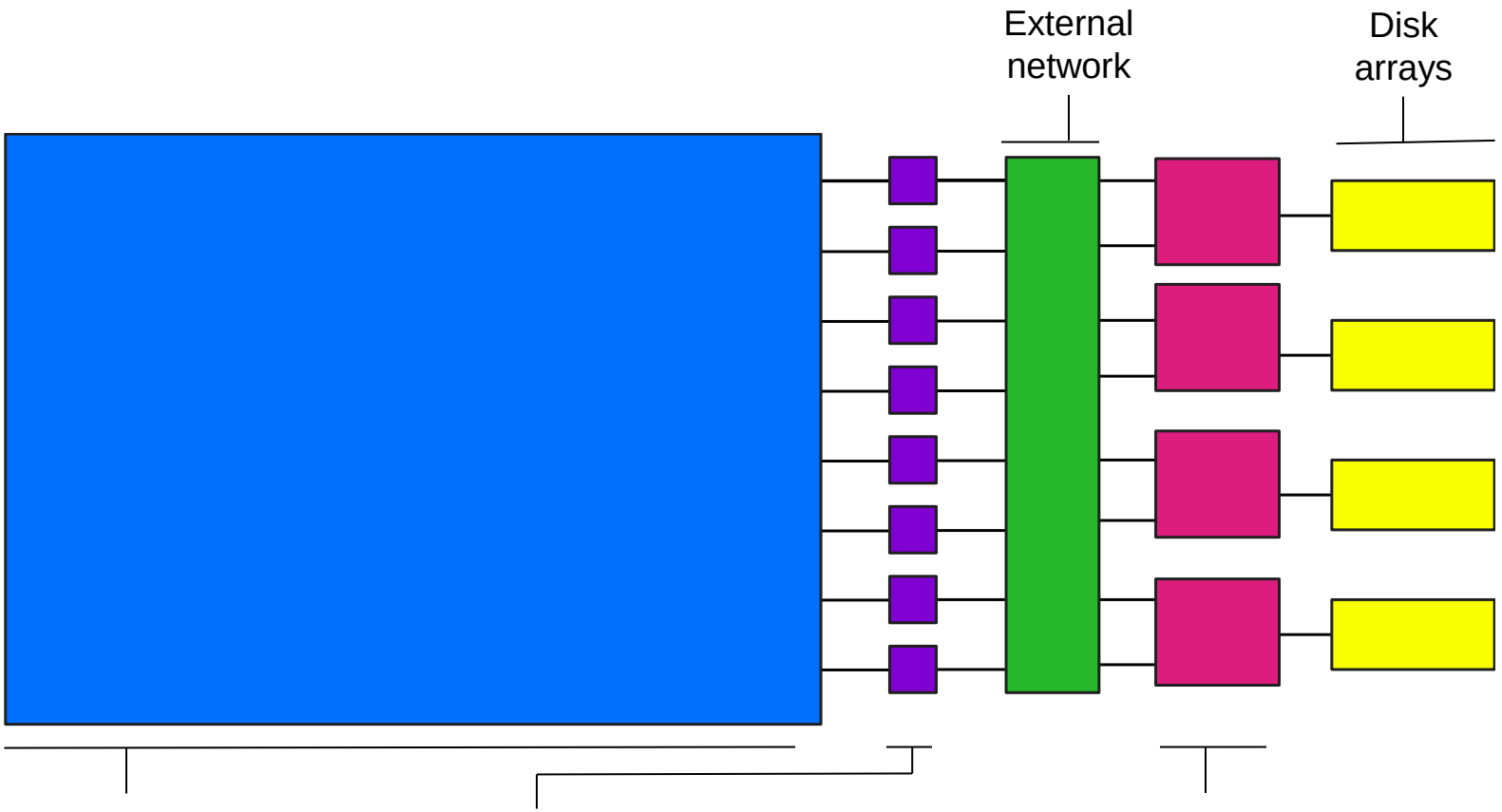



3D NAND by Samsung, VLSIT, 2009




3D NAND by Toshiba , VLSIT, 2009

# Adding In-System Storage: Where?

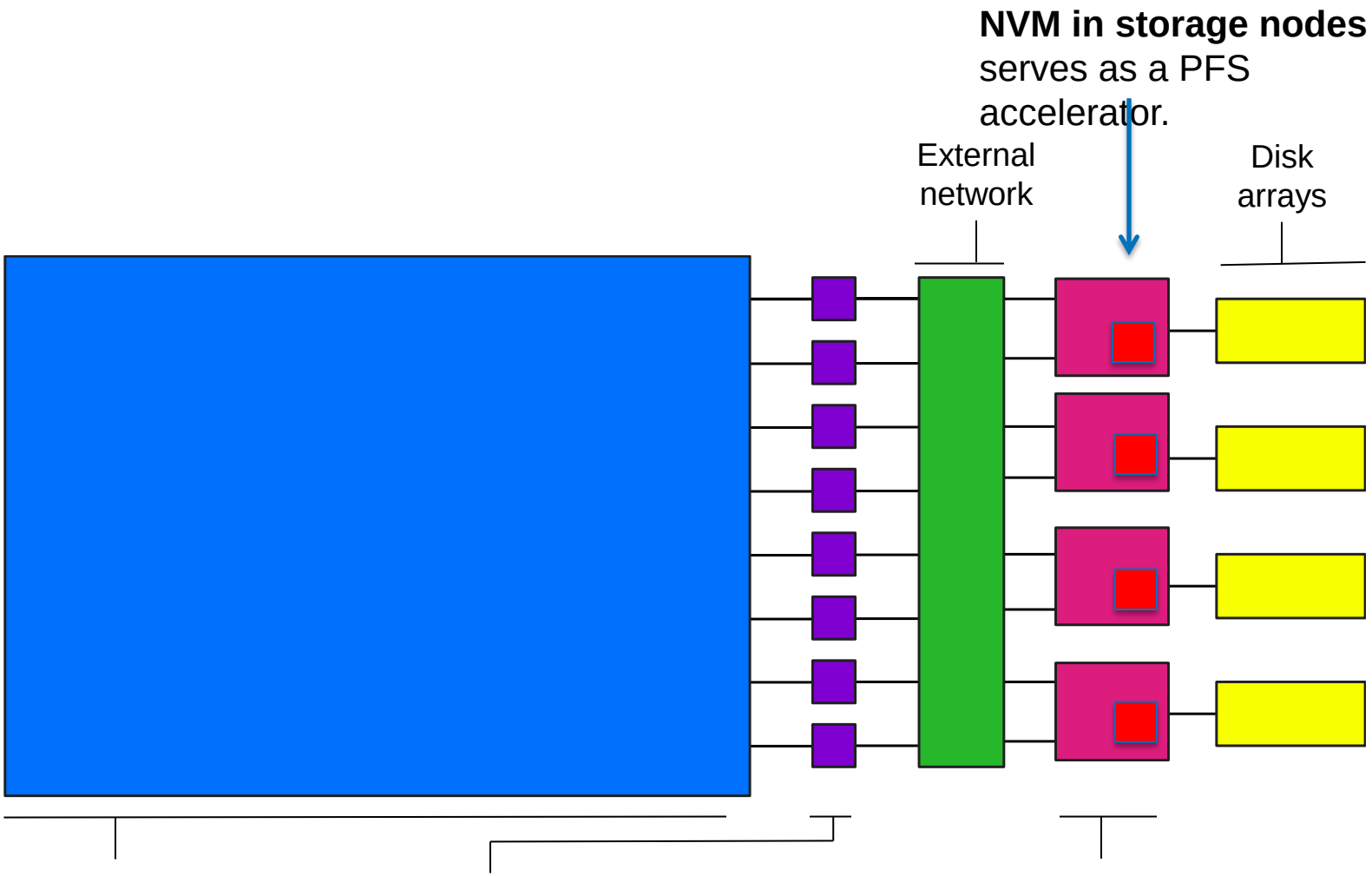



**Compute nodes** run application processes.

**I/O forwarding nodes** (or I/O gateways) shuffle data between compute nodes and external resources, including storage.

**Storage nodes** run the parallel file system.

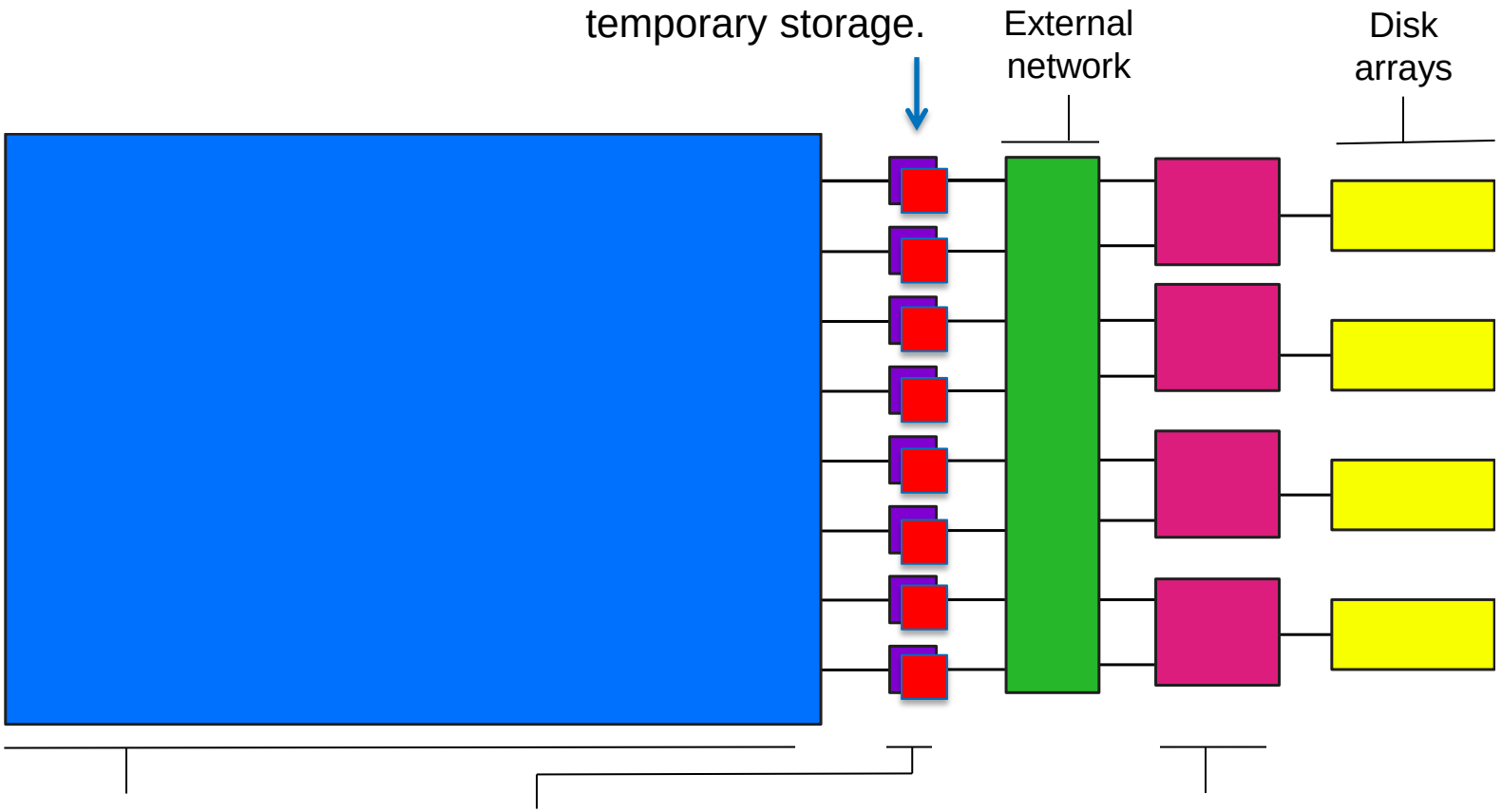
# Adding In-System Storage: Where?



# Adding In-System Storage: Where?

**NVM in I/O nodes** provides a fast staging area and region for temporary storage.

从起源吸收数据，减少I/O数据传送和文件系统调用。




**Compute nodes** run application processes.

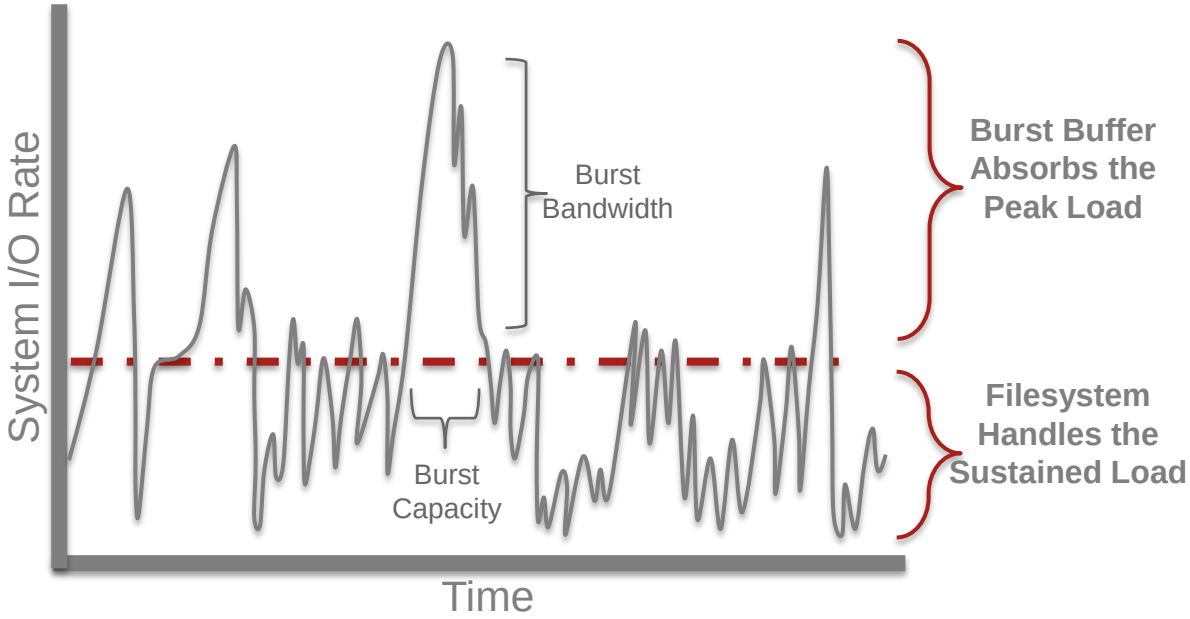
**I/O forwarding nodes** (or I/O gateways) shuffle data between compute nodes and external resources, including storage.

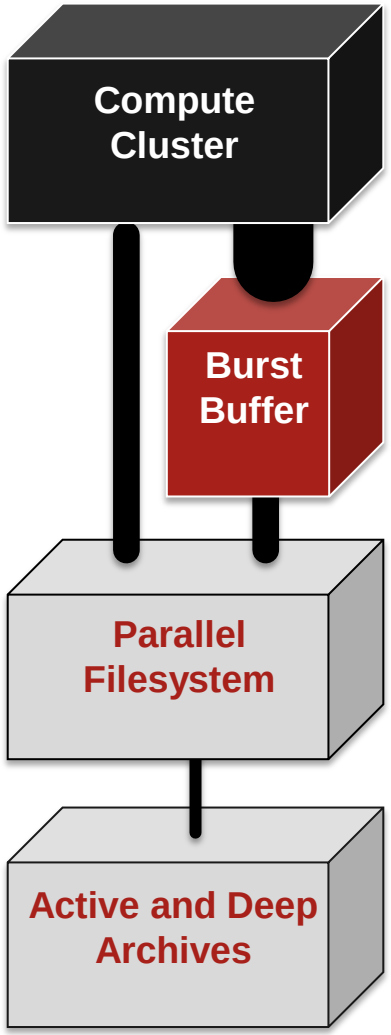
**Storage nodes** run the parallel file system.

# DDN Infinite Memory Engine

*Analysis of a major HPC production storage system*

- 99% of the time, storage BW utilization < 33% of max
- 70% of the time, storage BW utilization < 5% of max

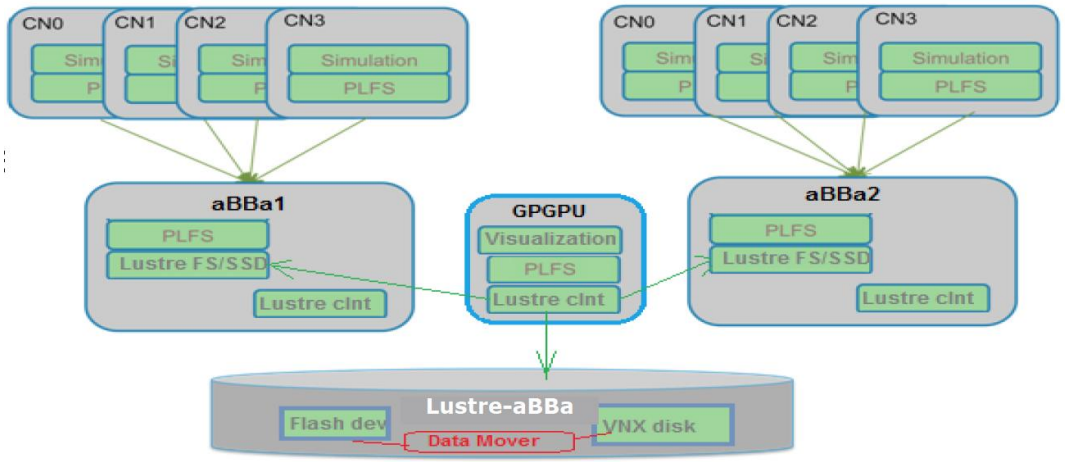



DDN's Infinite Memory Engine™ is a unified, distributed, non-volatile storage pool that is transparently accessible to parallel applications and is tightly integrated with proven high performance parallel filesystems. Configured as a Burst Buffer it changes how we provision I/O performance

Thanks to Jason Cope, Paul Nowoczynski, and Mike Vildibill of DDN for providing this material.

# EMC Active Burst Buffer Appliance (aBBa)



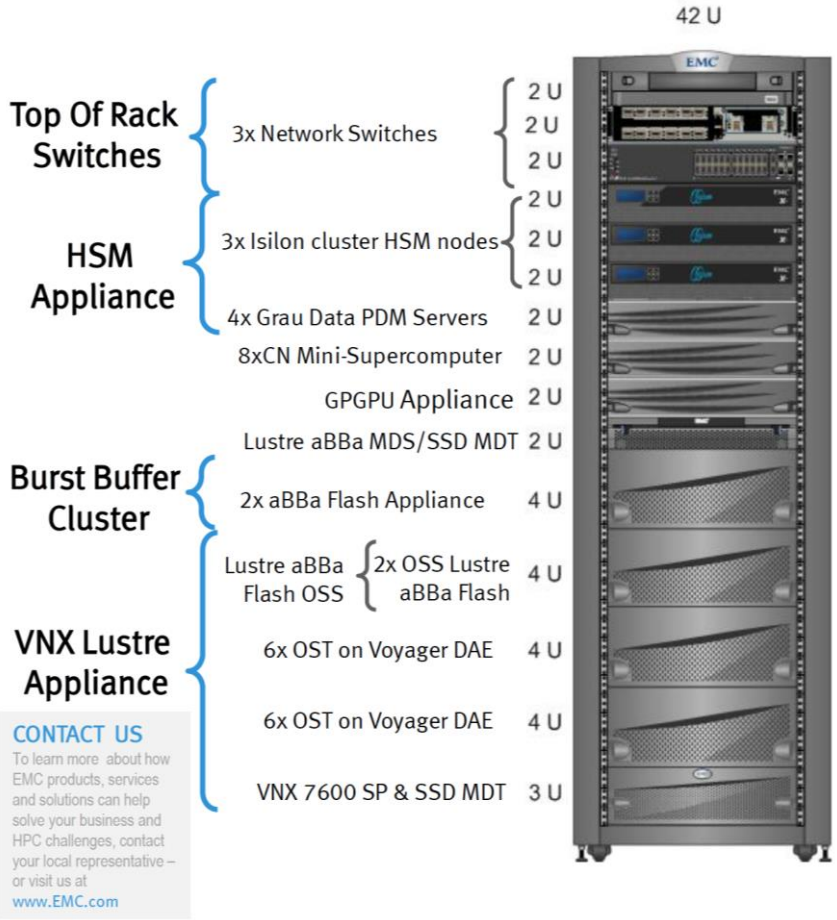

**PLFS** used to move data between CNs and I/O nodes (aBBa nodes), which use Lustre to organize SSD into a temporary storage space.

aBBa nodes are *active*: analysis operations can be run on those nodes.

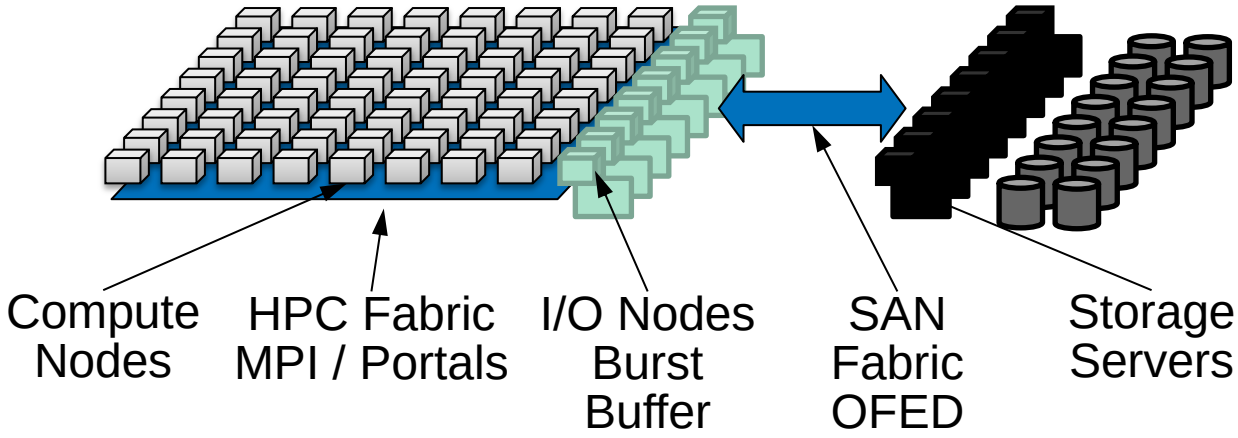
Data is asynchronously moved between aBBa nodes and external storage.

Thanks to John Bent, Richard Grossen, and Sassan Teymouri of EMC for providing this material.

**aBBa** can be deployed as part of a pre-configured Lustre based appliance.



# Burst Buffer Design for the Cori System at NERSC




Create Software to enhance usability and to meet the needs of all NERSC users

- Scheduler enhancements
  - Automatic migration of data to/from flash
  - Dedicated provisioning of flash resources
  - Persistent reservations of flash storage
- Enable In-transit analysis
  - Data processing or filtering on the BB nodes – model for exascale
- Caching mode – data transparently captured by the BB nodes
  - Transparent to user -> no code modifications required

# NERSC Burst Buffer Software Development Timeline

## Phase 3

- BB-node functionality: In Transit, filtering

## Phase 2

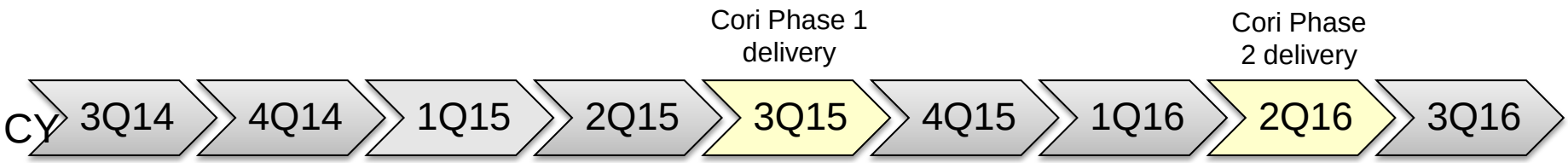
- Usability enhancements: Caching mode

## Phase 1

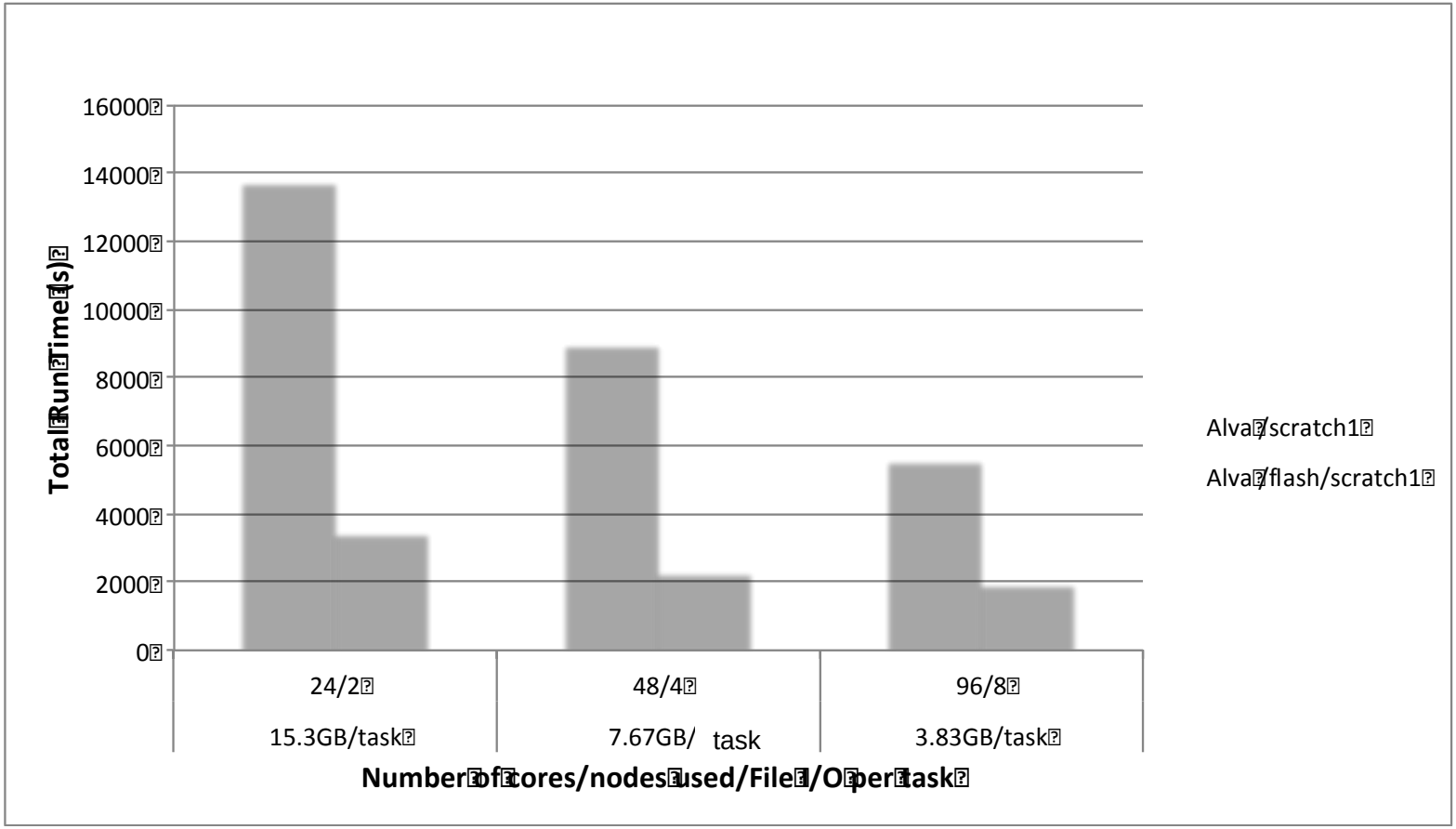
- I/O acceleration: Striping, reserved I/O bandwidth
- Job launch integration: allocation of space – per job or persistently
- Administrative functionality

## Phase 0

- Static mapping of compute to BB node
- User responsible for migration of data



# NWChem Out-of-Core Performance: Flash vs Disk on Burst Buffer testbed

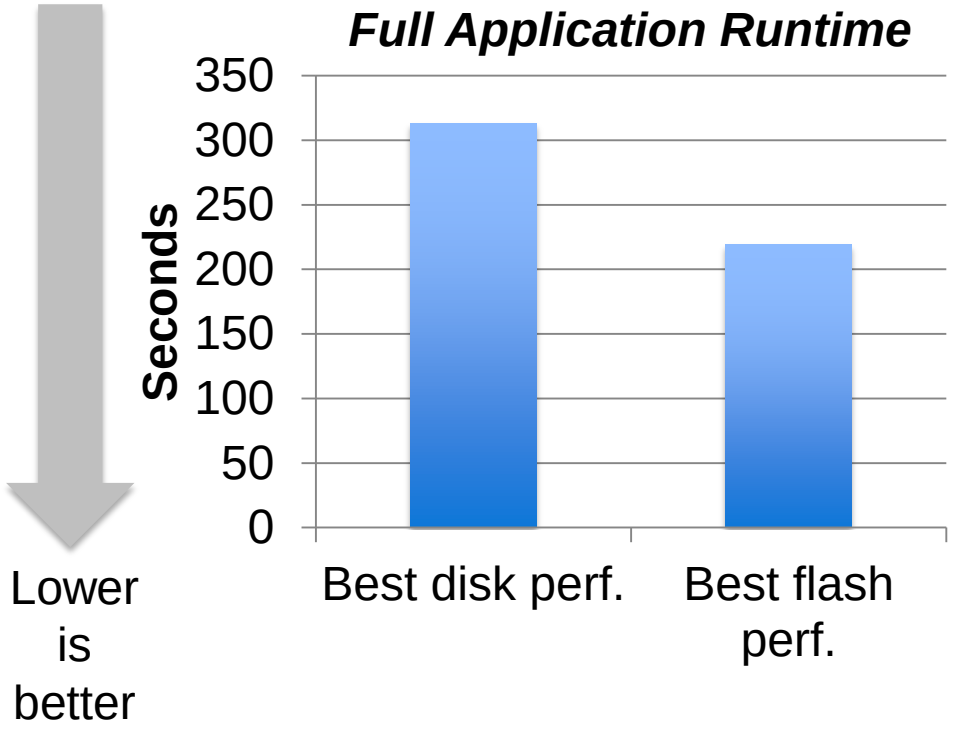


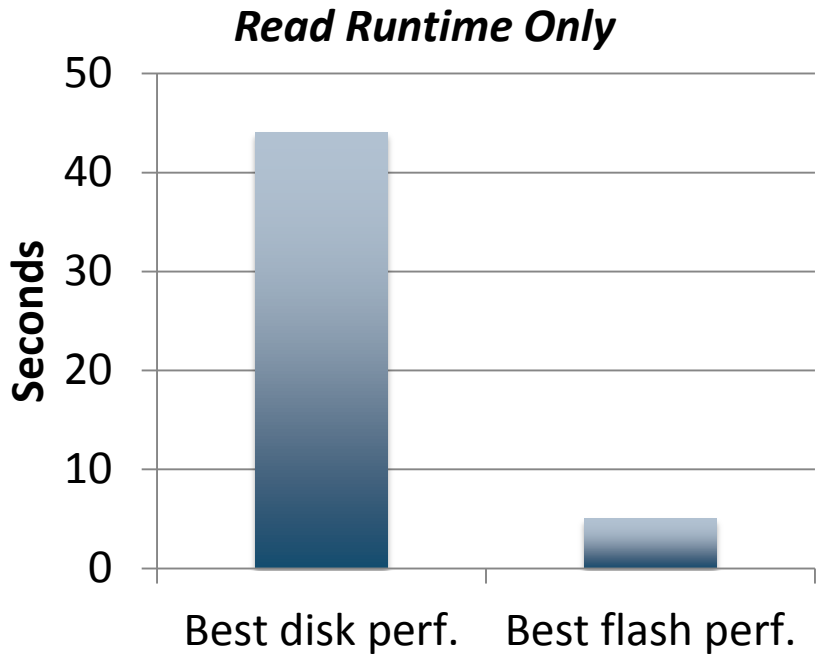

Lower is better

- NWChem MP2 Semi-direct energy computation on 18 water cluster with aug-cc-pvdz basis set
- Geometry (18 water cluster) from A. Lagutschenkov, e.tal, *J. Chem. Phys.* **122**, 194310 (2005).

Work by Zhengji Zhao

# TomoPy performance comparison between flash and disk file systems



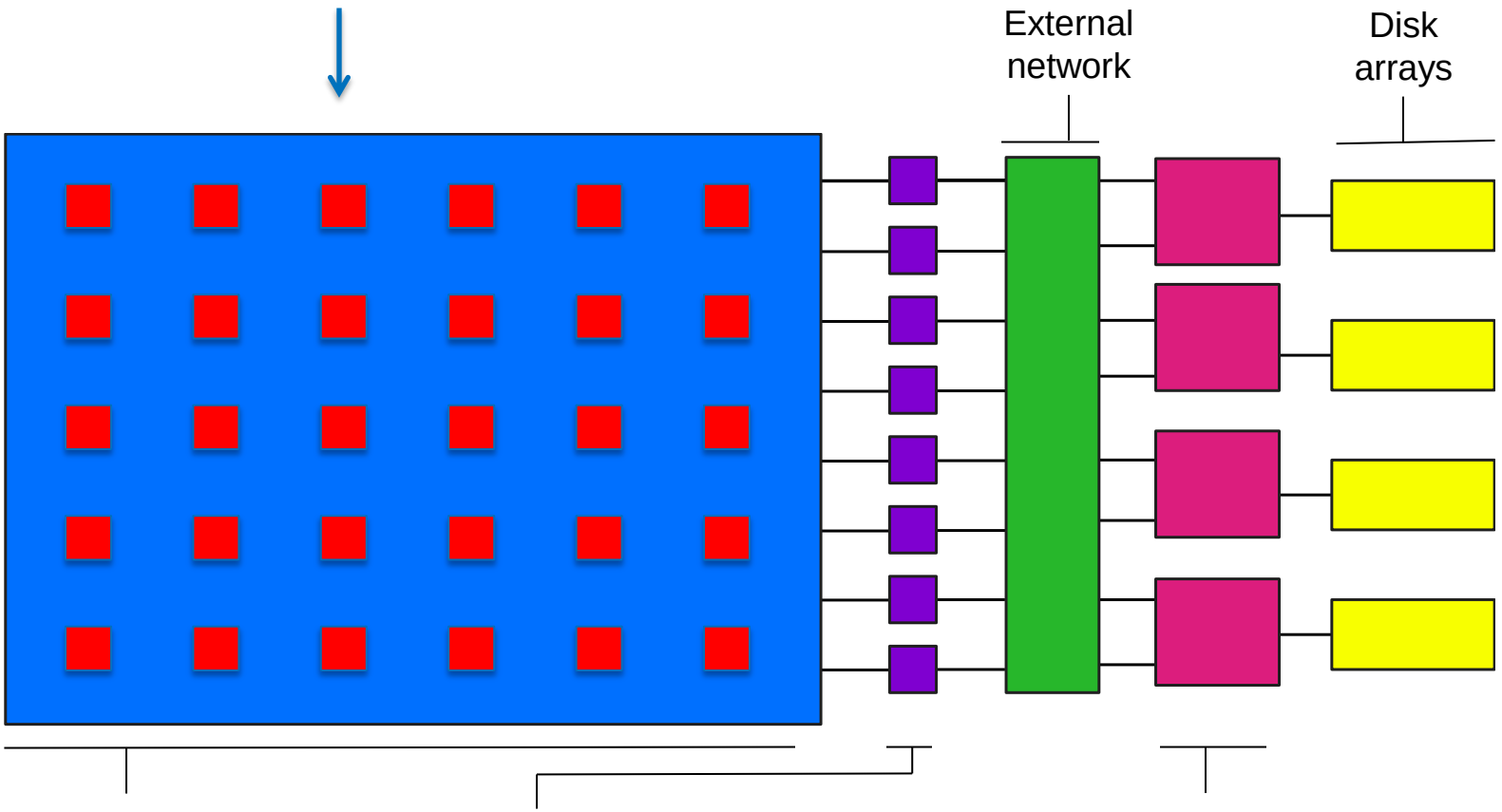




- This I/O intensive application runtime improves by 40% with the only change switching from disk to flash
- Read performance is much better when using Flash: ~8-9x faster than disk
- Disk performance testing showed high variability (3x runtime), whereas the flash runs were very consistent (2% runtime difference)

Work by Chris Daley

# Adding In-System Storage: Where?

**NVM in compute nodes** facilitates certain types of data analytics, alternatives to checkpoint/restart.



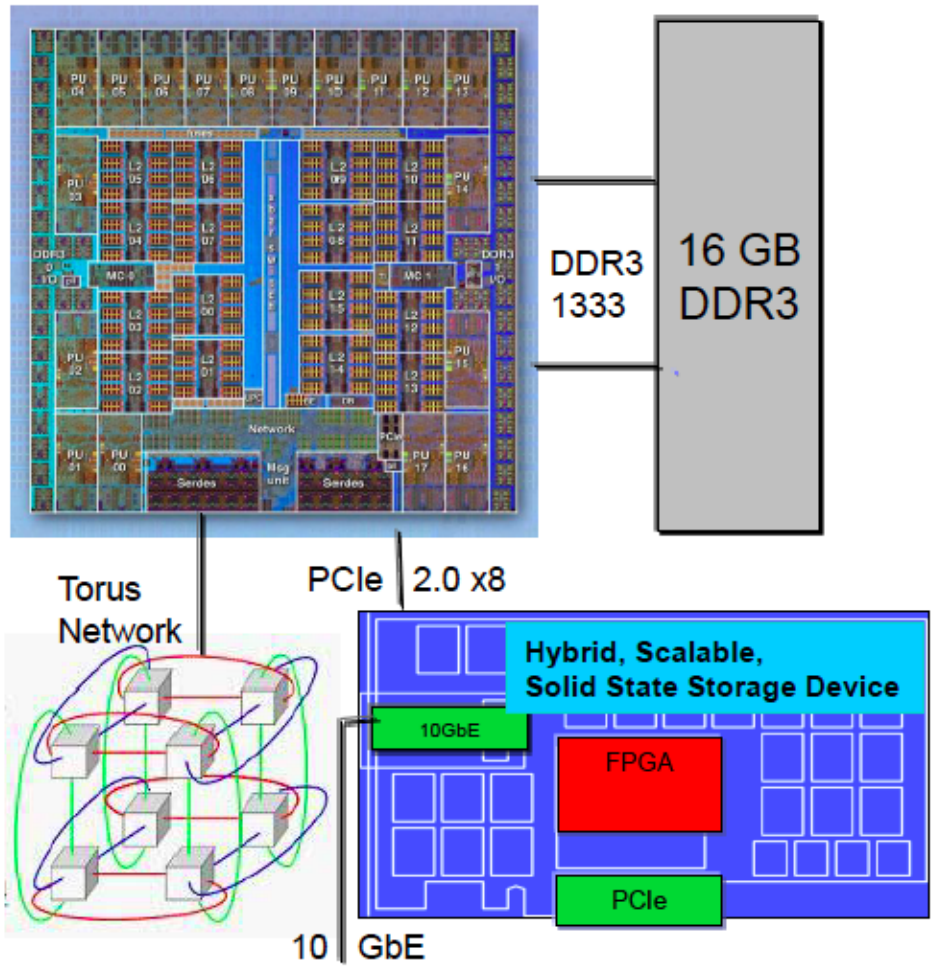

**Compute nodes** run application processes.

**I/O forwarding nodes** (or I/O gateways) shuffle data between compute nodes and external resources, including storage.

**Storage nodes** run the parallel file system.

# Blue Gene Active Storage Node

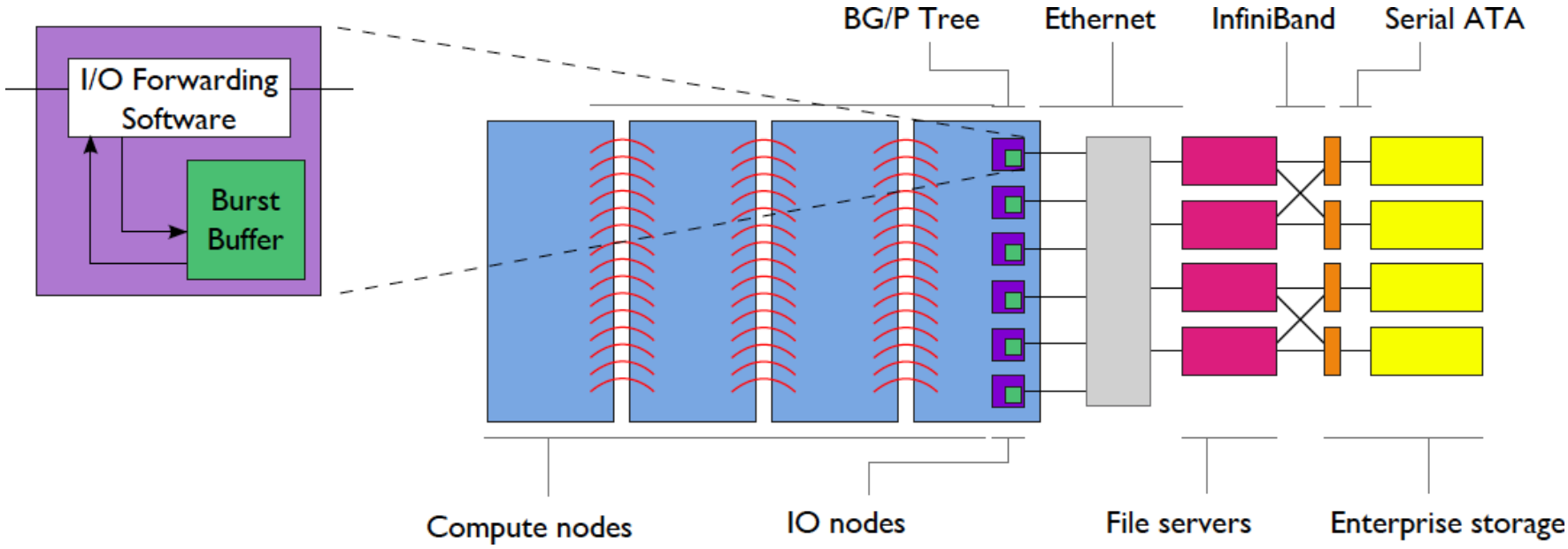
- Blue Gene/Q integrates processors, memory, and networking logic into a single chip
- Each chip supports a PCIe 2.0 x8 interface
- BGAS couples BG/Q node with a hybrid, scalable, solid state storage device
  - Can be integrated with compute-only nodes or used as part of a standalone data-centric computing system
  - Analysis can execute directly on BGAS nodes
- Storage can be managed in a number of ways
  - Combined into an in-system GPFS file system
  - Accessed via HDF5!




Thanks to Blake Fitch of IBM for providing this material. More information at http://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/bgas-BoF/bgas-BoF-fitch.pdf

# Understanding In-System Storage

**"Burst buffer" applications for in-system storage directly address checkpoint cost, and are a compelling use case for this technology. However, we need to understand details better to provision future systems correctly.**

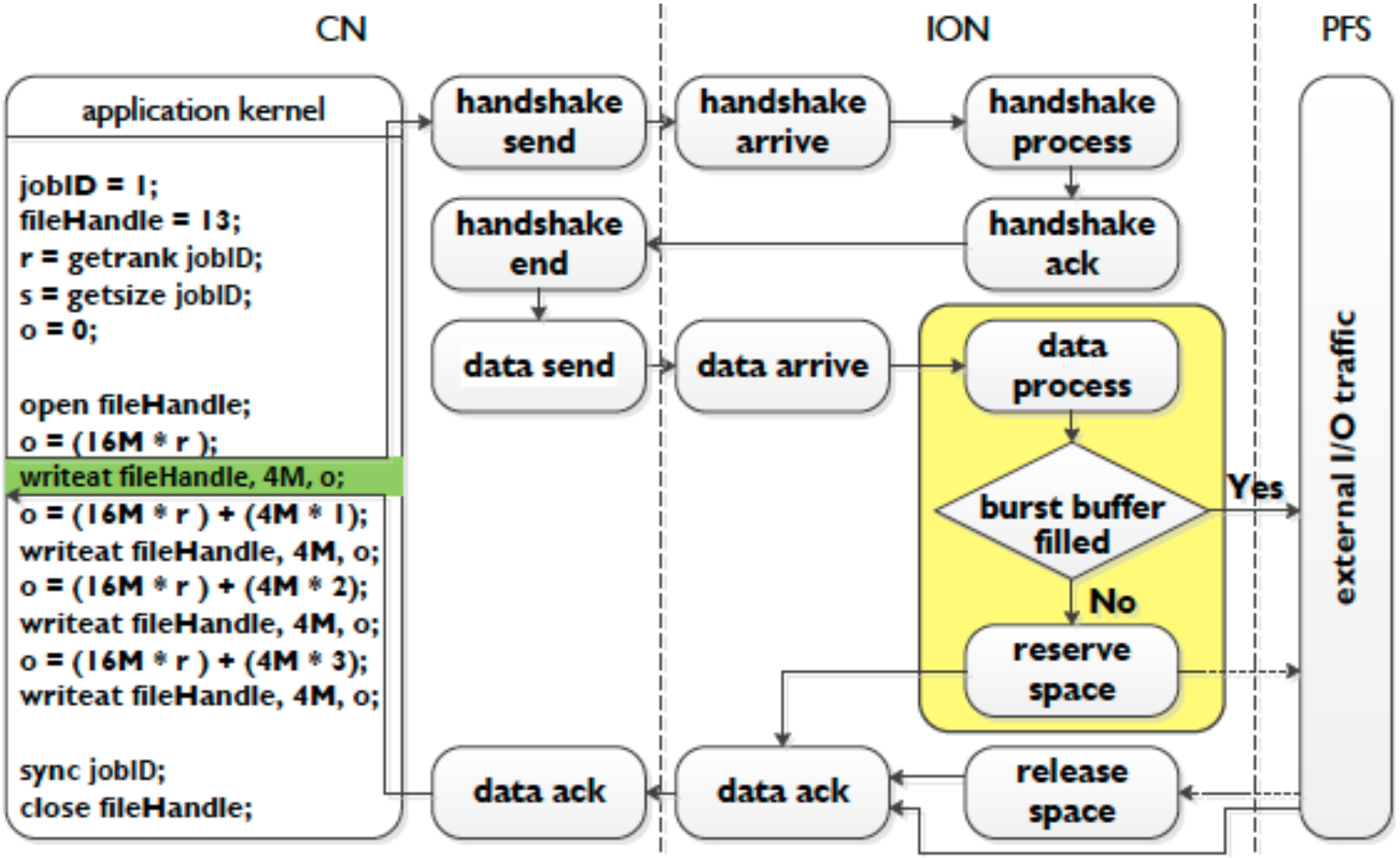


# What's a Burst?

- We quantified the I/O behavior by analyzing one month of production I/O activity on Blue Gene/P from December 2011
  - Application-level access pattern information with per process and per file granularity
  - Adequate to provide estimate of I/O bursts

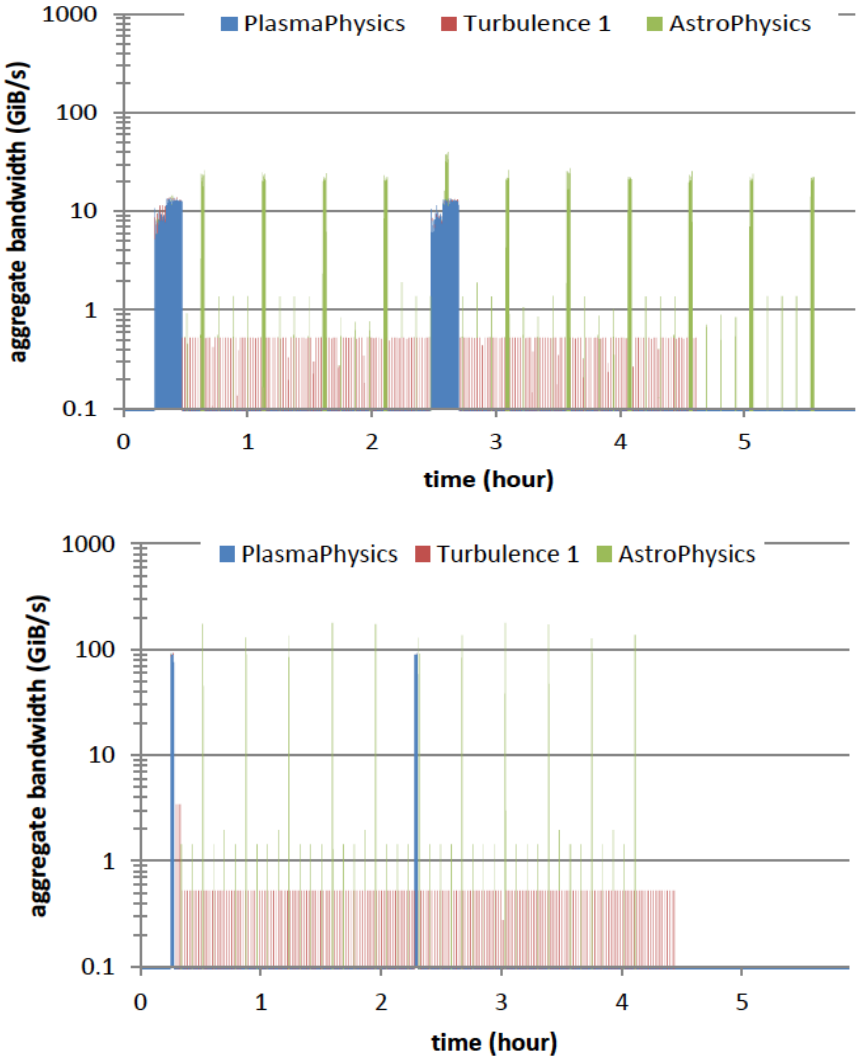
| Project | Procs | Nodes | Total Written | Run Time (hours) | Avg. Size and Subsequent Idle Time for Write Bursts>1 GiB | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | Count | Size | Size/Node | Size/ION | Idle Time (sec) |
| **PlasmaPhysics** | 131,072 | 32,768 | 67.0 TiB | 10.4 | 1 | 33.5 TiB | 1.0 GiB | 67.0 GiB | 7554 |
| | | | | | 1 | 33.5 TiB | 1.0 GiB | 67.0 GiB | end of job |
| **Turbulence1** | 131,072 | 32,768 | 8.9 TiB | 11.5 | 5 | 128.2 GiB | 4.0 MiB | 256.4 MiB | 70 |
| | | | | | 1 | 128.2 GiB | 4.0 MiB | 256.4 MiB | end of job |
| | | | | | 421 | 19.6 GiB | 627.2 KiB | 39.2 MiB | 70 |
| **AstroPhysics** | 32,768 | 8,096 | 8.8 TiB | 17.7 | 1 | 550.9 GiB | 68.9 MiB | 4.3 GiB | end of job |
| | | | | | 8 | 423.4 GiB | 52.9 MiB | 3.3 GiB | 240 |
| | | | | | 37 | 131.5 GiB | 16.4 MiB | 1.0 GiB | 322 |
| | | | | | 140 | 1.6 GiB | 204.8 KiB | 12.8 MiB | 318 |
| **Turbulence2** | 4,096 | 4,096 | 5.1 TiB | 11.6 | 21 | 235.8 GiB | 59.0 MiB | 3.7 GiB | 1.2 |
| | | | | | 1 | 235.8 GiB | 59.0 MiB | 3.7 GiB | end of job |

# Studying Burst Buffers with Parallel Discrete Event Simulation



# Burst Buffers Work for Multi-application Workloads

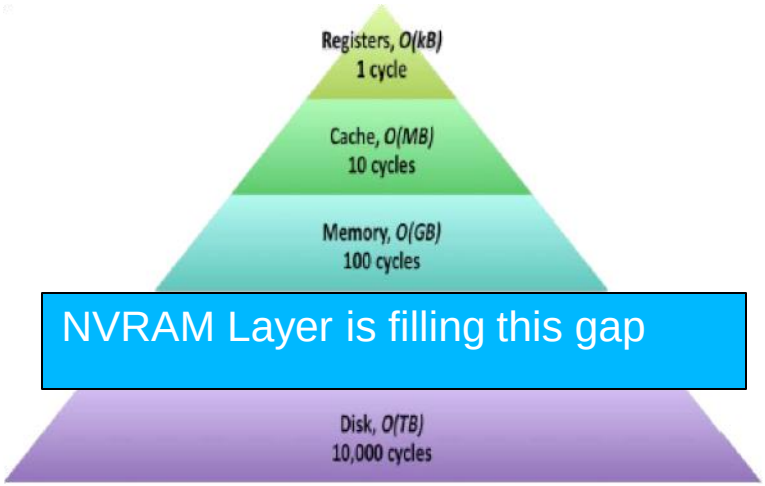
- Burst buffers improve application perceived throughput under mixed I/O workloads.
- Applications' time to solution decrease with burst buffers enabled (from 5.5 to 4.4 hours)
- Peak bandwidth of the external I/O system may be reduced by 50% without a perceived change on the application side
- Tool for co-design




Application perceived I/O rates, with no burst buffer (top), burst buffer (bottom).

# What does the introduction of NVRAM layers into HPC systems mean for the future of parallel I/O systems?

- As NVRAM becomes more cost competitive we could expect NVRAM layers to eat into the parallel-file system layers
- Below the NVRAM layer could be a more storage system more configured for capacity
- In 5 years will our parallel I/O tutorial focus on disk based filesystems?



# Changing Memory and Storage Hierarchy

| Memory/Storage Layer | NERSC-7 (Edison) 2013 | NERSC-8 (Cori) 2016 | NERSC-9 ??? 2020 |
|---|---|---|---|
| **High Bandwidth Memory per node** | None | 16 GB, >400 GB/sec | More high bandwidth memory |
| **DRAM per node** | 64 GB, ~100 GB/sec | 96 GB, 90-100 GB/sec | Maybe/not? |
| **NV-DIMM (byte addressable)** | None | None | Likely |
| **Non-Volatile (file addressable)** | None | 1.5PB, 1.5 TB/sec | 10s PBs, 10s TB/sec |
| **Spinning Disk – /scratch** | 8PB, 130 GB/sec | 28 PB, 700 GB/sec | Collapsed layer 100 PBs ~1 TB/sec |
| **Spinning Disk – longer term (/project)** | ~30 PB, ~70 GB/sec | ~50 PB, ~100 GB/sec | |
| **Tape** | ~40 PB, ~10 GB/sec | ~100PB, ~20 GB/sec | ~100s PB, ~10s GB/sec |

# I/O Forwarding

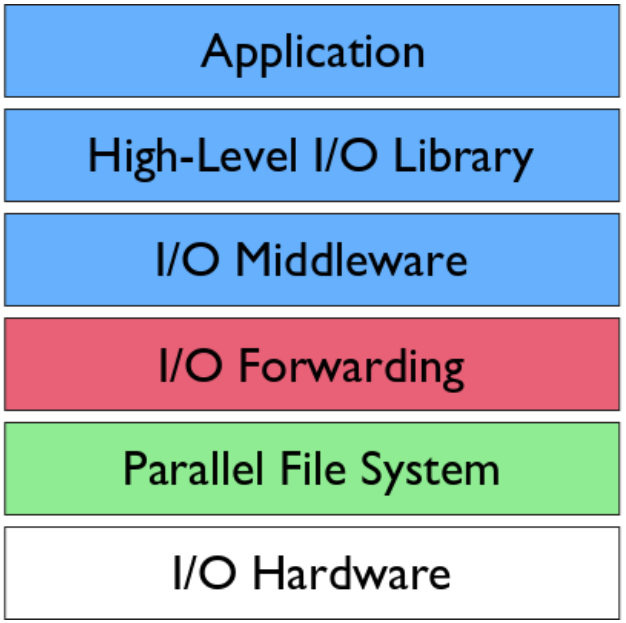
# I/O for Computational Science

**High-Level I/O Library**
maps application abstractions onto storage abstractions and provides data portability.

*HDF5, Parallel netCDF, ADIOS*

**I/O Forwarding**
bridges between app. tasks and storage system and provides aggregation for uncoordinated I/O.

*IBM ciod, IOFSL, Cray DVS*




**I/O Middleware**
organizes accesses from many processes, especially those using collective I/O.

*MPI-IO*

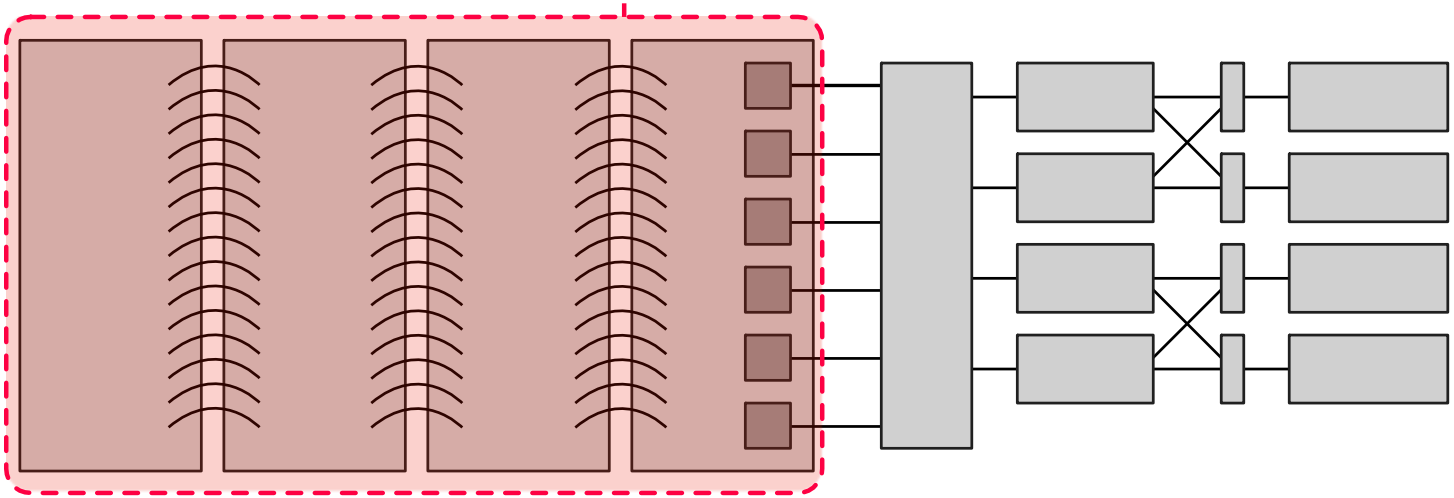
**Parallel File System**
maintains logical space and provides efficient access to data.

*PVFS, PanFS, GPFS, Lustre*

**Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.**

# I/O Forwarding Software

**I/ O forwarding** software runs on compute and gateway nodes and bridges between the compute nodes and external storage.



**Compute nodes**

run I/O forwarding software intercepting I/O calls from application and forwarding to gateway nodes

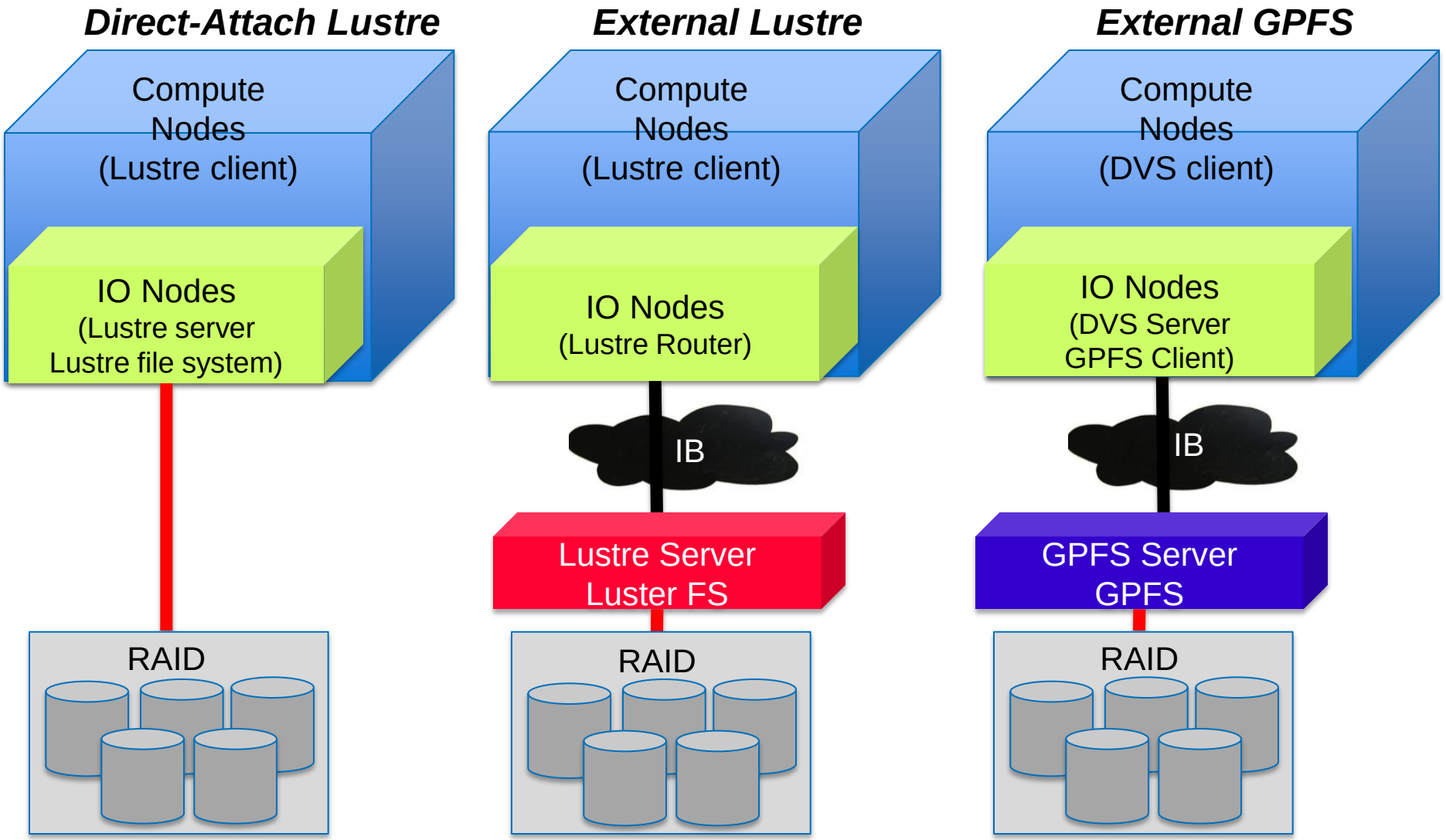
**Gateway nodes**

run I/O forwarding software accepting I/O requests from compute nodes and forward to parallel file system

# Cray's Data Virtualization Service (DVS)

- A distributed network service that allows other file systems besides Lustre (GPFS, Panasas, NFS) to be used on the XT/XE systems
- DVS servers forward data to the underlying file system and forward results back to DVS client
- Light-weight DVS client installed on compute nodes
- Also used to enable shared library applications on Hopper

# Comparison of Direct Attached Lustre, External Lustre, and Alternate External File Systems



# I/O Architectures: Similarities

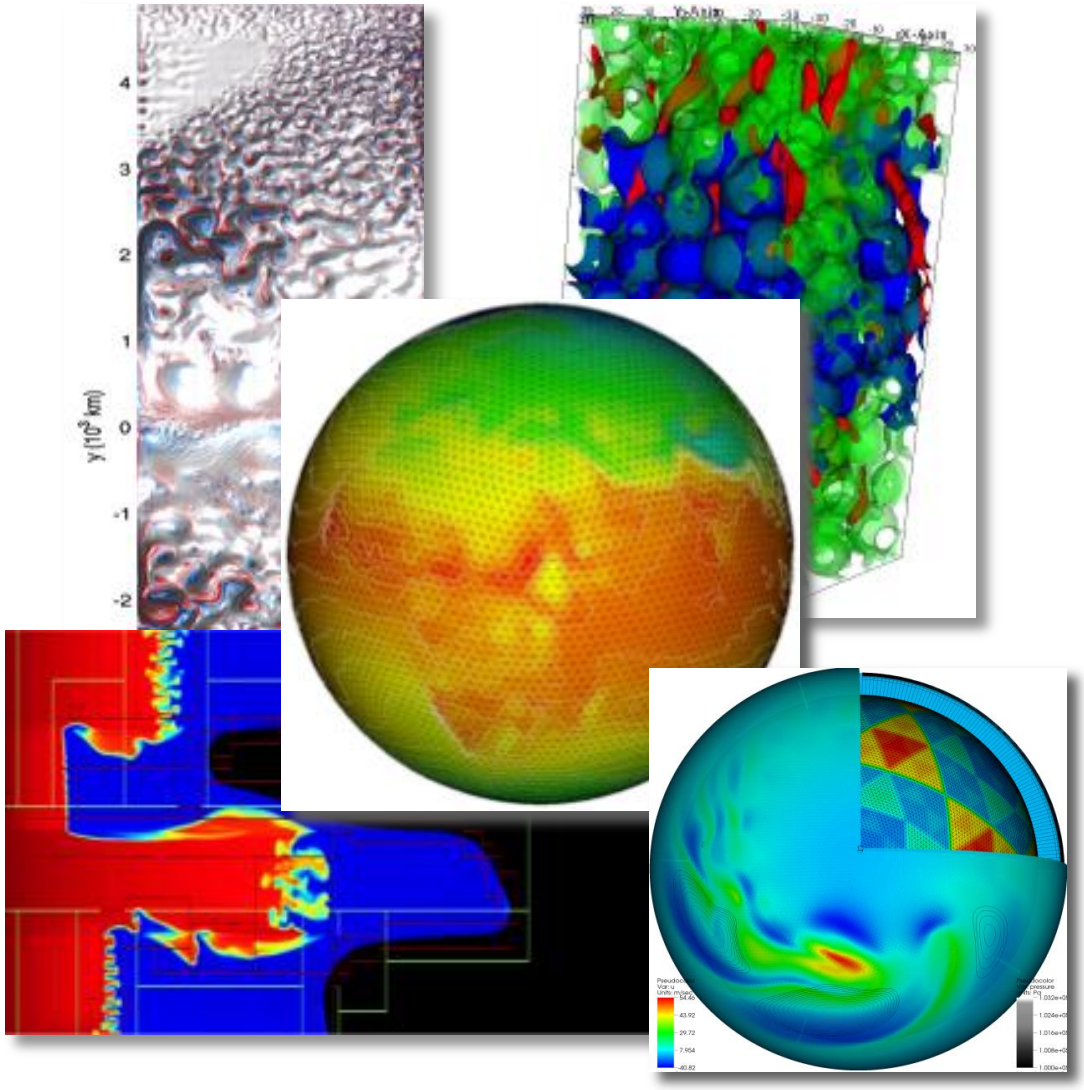
**I/O architectures at large scale have converged to a common model.**

- Compute nodes with indirect access to storage
- I/O nodes that form a bridge to the storage system
  - Lnet, DVS, ciod
- External network fabric connecting HPC system to storage
- Collection of storage servers and enterprise storage hardware providing reliable, persistent storage

# Scientific Applications and I/O

# Scientific I/O: more than hard drives and file systems

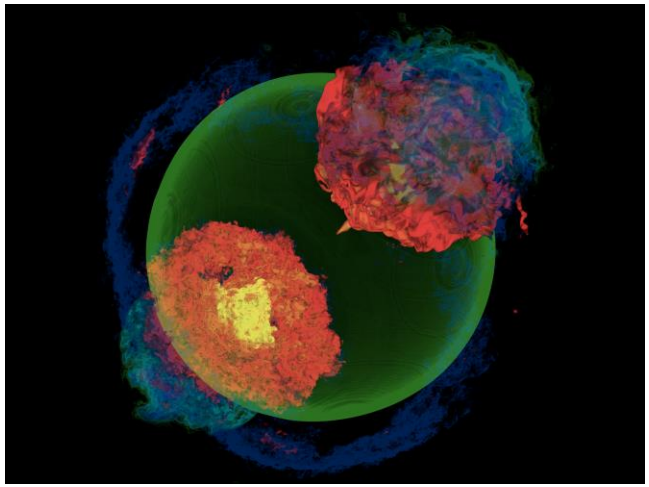
- Scientists think about data in terms of their science problem: molecules, atoms, grid cells, particles
- Ultimately, physical disks store bytes of data
- Layers in between, the application and physical disks are at various levels of sophistication

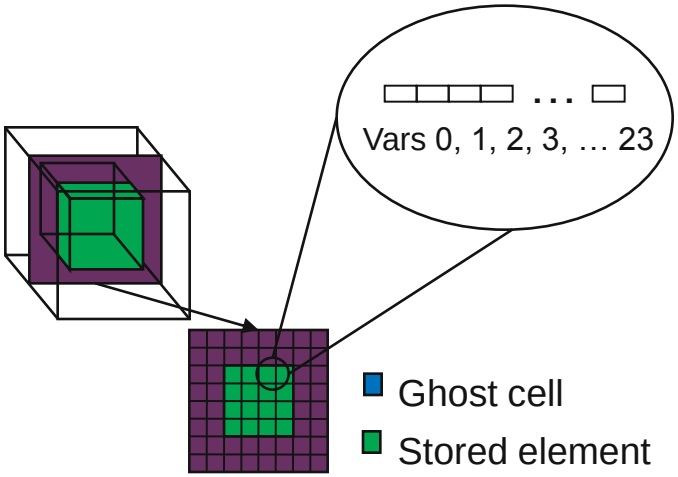



Images from David Randall, Paola Cessi, John Bell, T Scheibe

# Example: FLASH Astrophysics

- FLASH is an astrophysics code for studying events such as supernovae
  - Adaptive-mesh hydrodynamics
  - Scales to 1000s of processors
  - MPI for communication
- Frequently checkpoints:
  - Large blocks of typed variables from all processes
  - Portable format
  - Canonical ordering (different than in memory)
  - Skipping ghost cells





# Example: FLASH's HDF5 requirements

- FLASH AMR structures do not map directly to HDF5 multidimensional arrays
- Must create mapping of the in-memory FLASH data structures into a representation in HDF5 multidimensional arrays
- Chose to
  - Place all checkpoint data in a single file
  - Impose a linear ordering on the AMR blocks
    - Use 4D variables
  - Store each FLASH variable in its own HDF5 variable
    - Skip ghost cells
  - Record attributes describing run time, total blocks, etc.

# FLASH HDF5 Usage

- Annotations describing data, experiment

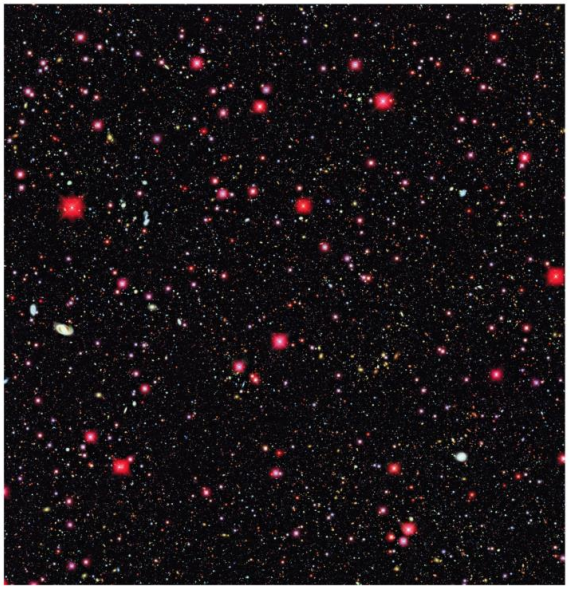
```
attribute_id = H5Acreate(group_id,
  “iteration“,H5T_NATIVE_INT, dataspace_id, H5P_DEFAULT);
status = H5Awrite(attribute_id, H5T_NATIVE_INT, temp);
```

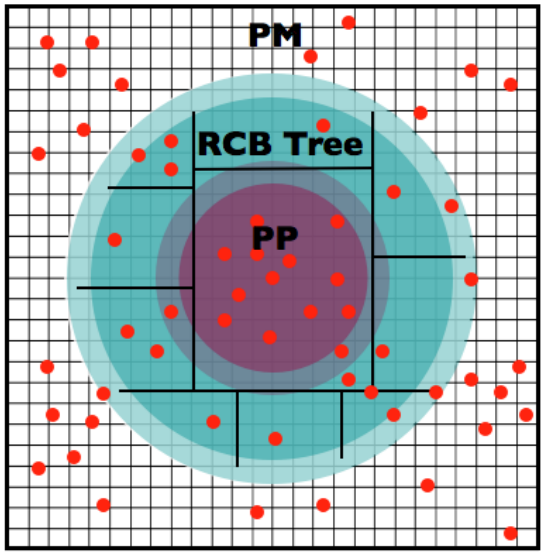
- HDF5 variables for each FLASH variable

```
ierr = H5Sselect_hyperslab(dataspace, H5S_SELECT_SET,
  start_4d, stride_4d, count_4d, NULL);
status = H5Dwrite(dataset, H5T_NATIVE_DOUBLE, memspace,
  dataspace, dxfer_template, unknowns);
```

# HACC: understanding cosmos via simulation

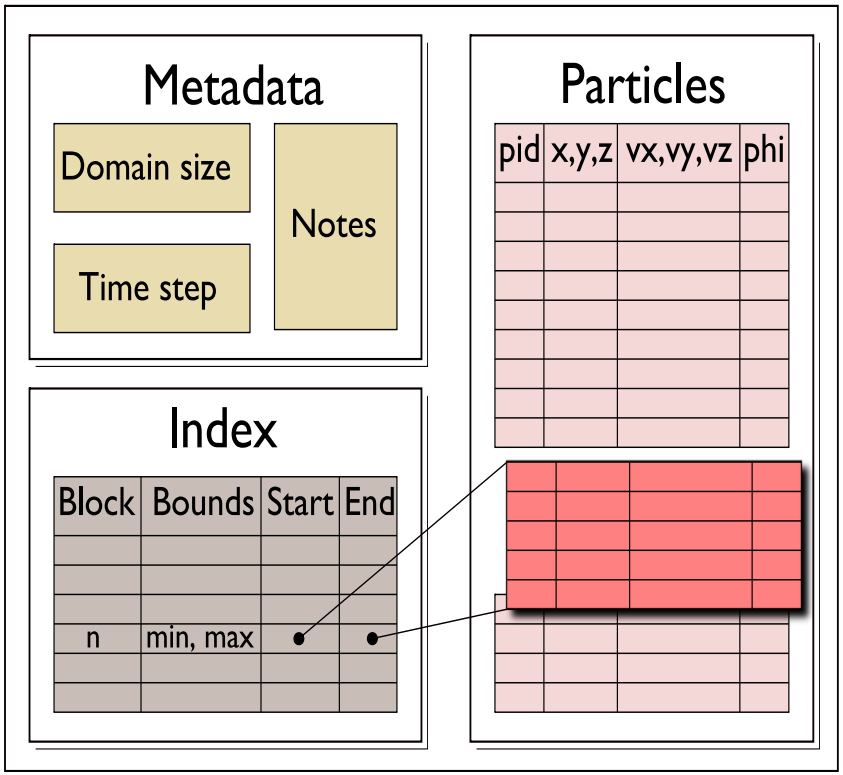
- "Cosmology = Physics + Simulation" (Salman Habib)
- Sky surveys collecting massive amounts of data
  - (~100 PB)
- Understanding of these massive datasets rests on modeling distribution of cosmic entities
- Seed simulations with initial conditions
- Run for 13 billion (simulated) years
- Comparison with observed data validates physics model.
- I/O challenges:
  - Checkpointing
  - analysis





# Parallel NetCDF Particle Output
## Exploratory Collaboration with Northwestern and Argonne

- Metadata, index, and particle data
- Self-describing portable format
- Can be read with different number of processes than written
- Can be queried for particles within spatial bounds




File schema for analysis output enables spatial queries of particle data in a high-level self-describing format.

# HACC and Parallel-NetCDF

- Attributes describing the dataset

```
ncmpi_put_att_text(ncfile, NC_GLOBAL, "notes",
strlen(notes), notes);
```

- Description of data separate from use of data (bi-modal interface)

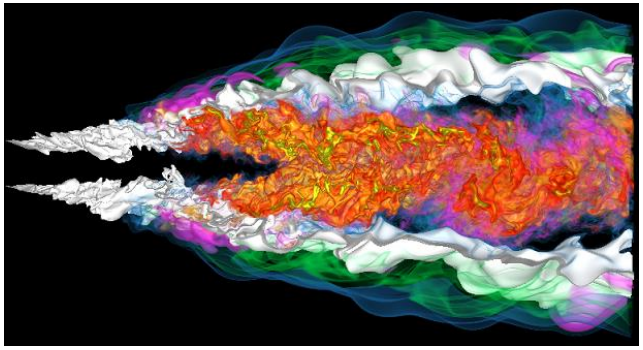
```
ncmpi_def_var(ncfile, “Velocity”, NC_FLOAT,
ndims, dimpids, &var_velid);
```

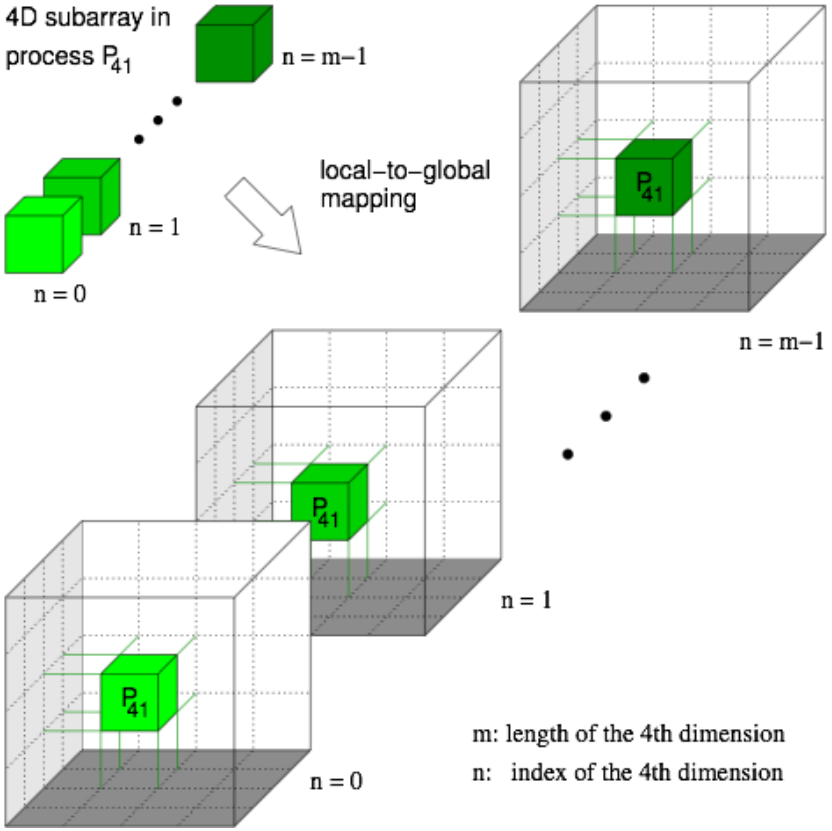
- Store data into variable

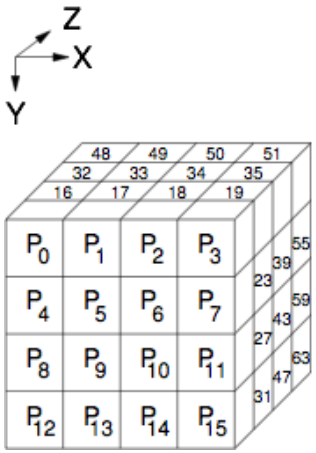
```
ncmpi_put_vara_float_all(ncfile, var_velid, start, count,
  &data_vel[0][0]);
```

# S3D Turbulent Combustion Code

- S3D is a turbulent combustion application using a direct numerical simulation solver from Sandia National Laboratory
- Checkpoints consist of four global arrays
  - 2 3-dimensional
  - 2 4-dimensional
  - 50x50x50 fixed subarrays






Thanks to Jackie Chen (SNL), Ray Grout (SNL), and Wei-Keng Liao (NWU) for providing the S3D I/O benchmark, Wei-Keng Liao for providing this diagram, C. Wang, H.Yu, and K.-L. Ma of UC Davis for image.

# S3D's demands on MPI-IO

- Describes 3-d and 4-d arrays in file with mpidatatypes and `MPI_File_set_view`
- File format neither portable nor self-describing when using MPI-IO
- S3D-IO

# S3D and MPI-IO code

- Describe layout on disk with MPI datatypes

```
integer       g_sizes(3), subsizes(3), starts(3)
! ... set up global and local sizes, starting offsets ...
call MPI_Type_create_subarray(3, g_sizes, subsizes, &
                  starts, MPI_ORDER_FORTRAN, MPI_REAL8, &
threeD_ftype, ierr)
call MPI_Type_commit(threeD_ftype, ierr)
call MPI_File_open(gcomm, trim(filename), open_mode,
&file_info, fp, ierr)
call MPI_File_set_view(fp, iOffset, MPI_REAL8, threeD_ftype,&
     'native', MPI_INFO_NULL, ierr)
```

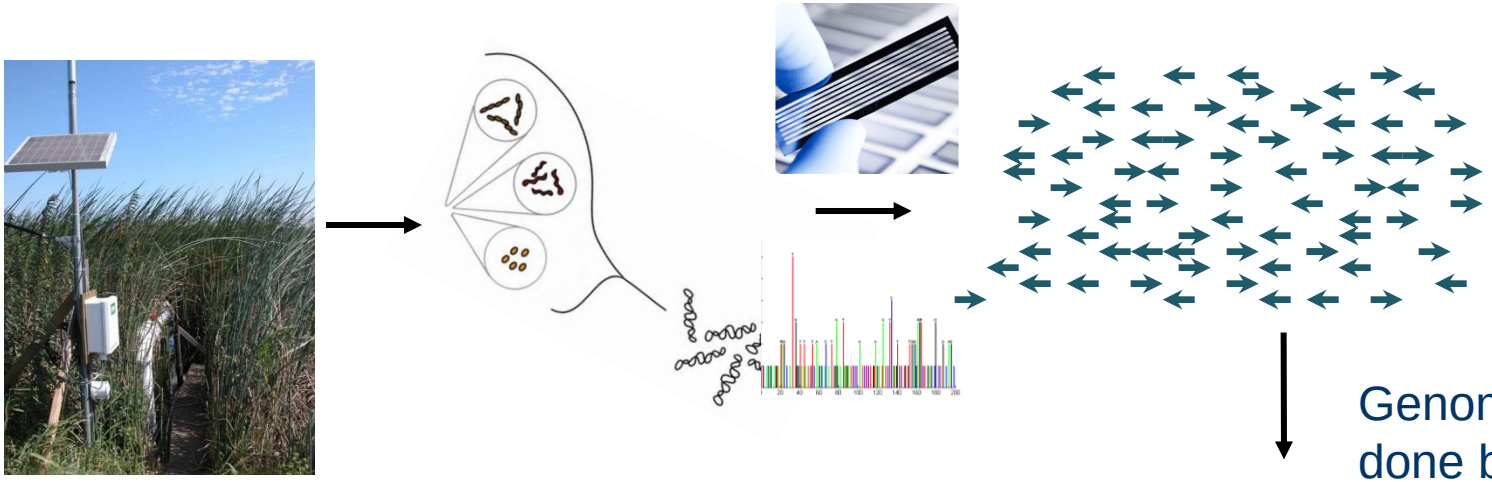
- Write each variable to file.  S3D does this once per variable

```
call MPI_File_write_all(fp, temp, nx_ny_nz, MPI_REAL8, &
mstatus, ierr)
call MPI_File_write_all(fp, u, nx_ny_nz*3, MPI_REAL8,
&mstatus, ierr)
```

# AllPaths-LG de novo Assembler

Microbial DNA extracted and sequenced

Reads (short segments of DNA) generated by sequencer



Samples taken from the wild

Genome Assembly done by AllPaths-LG

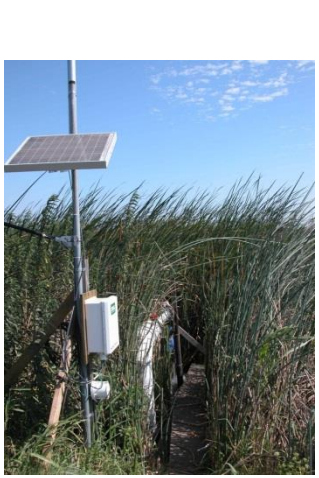
Complete Genome

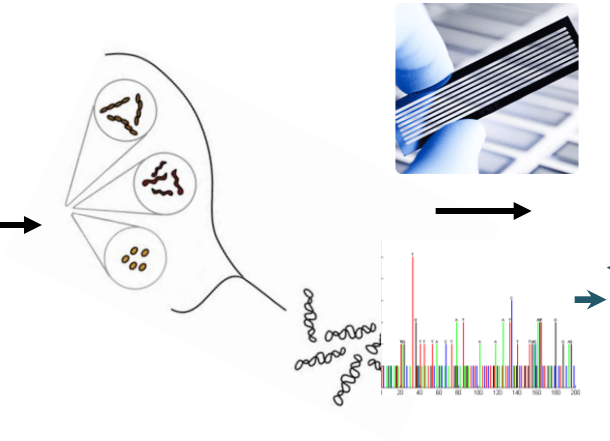
Kirsten Fagnan, LBL

# AllPaths-LG de novo Assembler

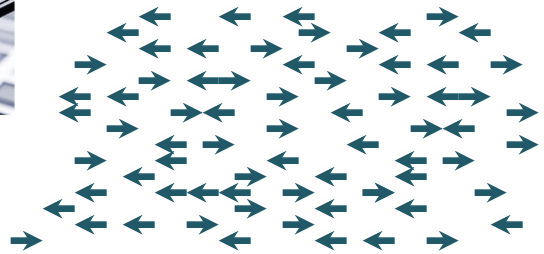
Microbial DNA extracted and sequenced

Reads (short segments of DNA) generated by sequencer



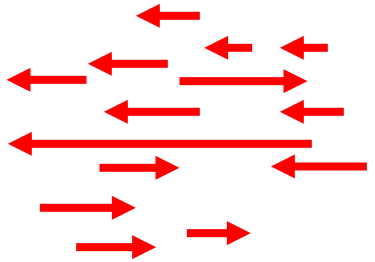






Genome Assembly done by AllPaths-LG

Samples taken from the wild

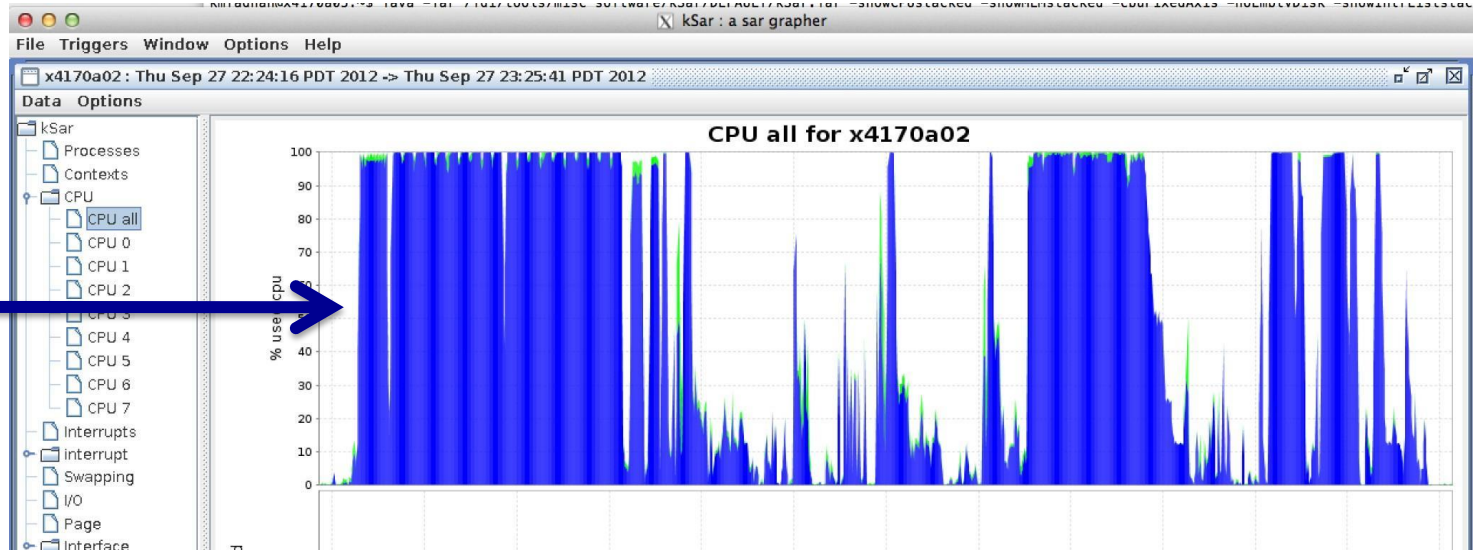


Contigs

Kirsten Fagnan, LBL

# AllPaths-LG de novo Assembler – CPU and Memory Usage

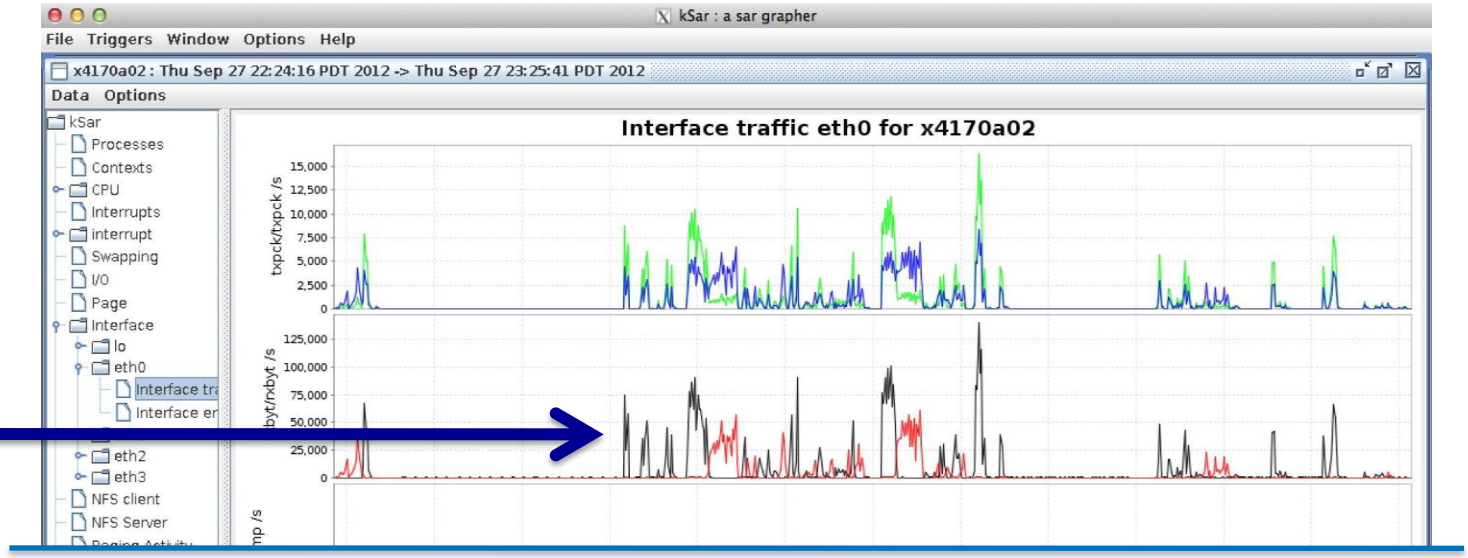
CPU Usage gets to 100% on 8 cores during many phases of the calculation



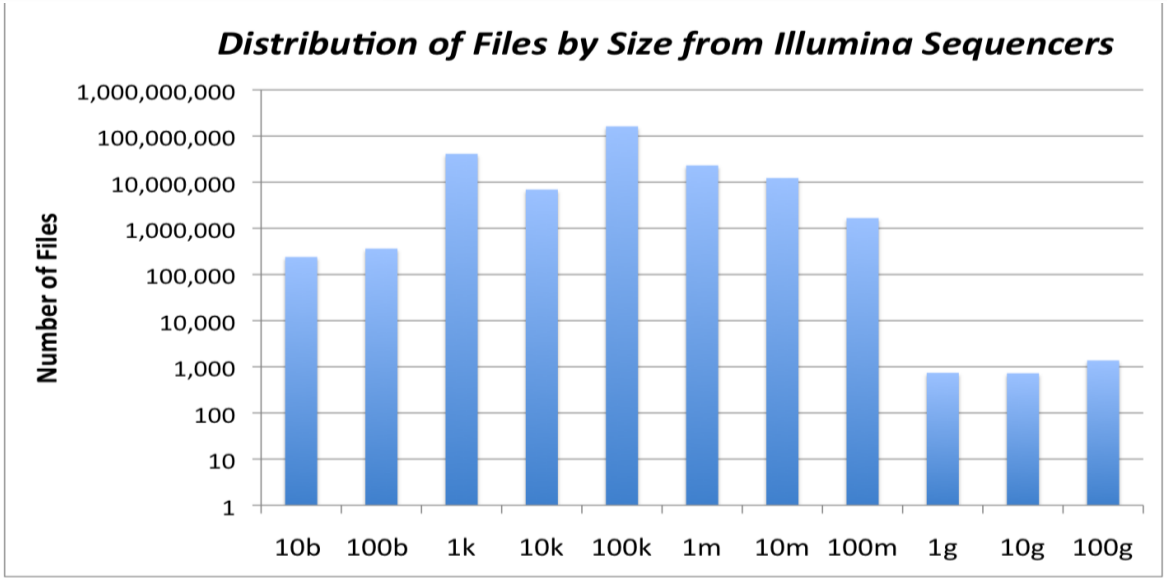

Memory Usage is high for assemblies

# AllPaths-LG de novo Assembler – File System I/O

Periods of relatively high I/O during extension and error correction phases writes(black), reads(red)




85% of files are less than 100k in size




Kirsten Fagnan, LBL

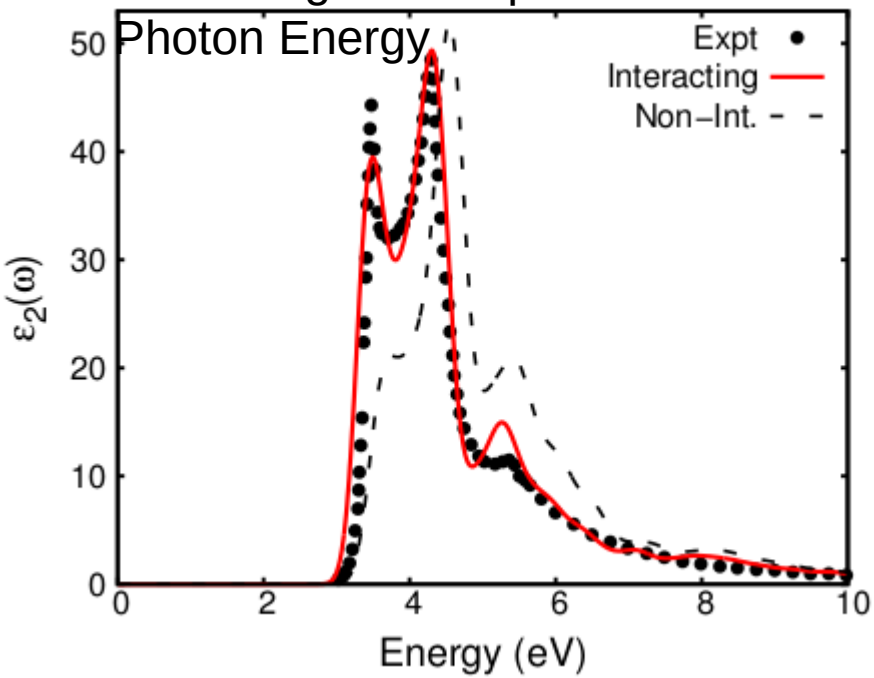
# Berkeley GW

A material science code that sits on top of DFT codes like
- Quantum ESPRESSO /
- PARATEC /
- PARSEC / SIESTA

FFTs, Dense Linear Algebra and Large Reduction Loops.

BerkeleyGW

Silicon Light Absorption vs. Photon Energy




Jack Deslippe, LBL

# Memory and I/O Layout of BerkeleyGW before parallel I/O implementation

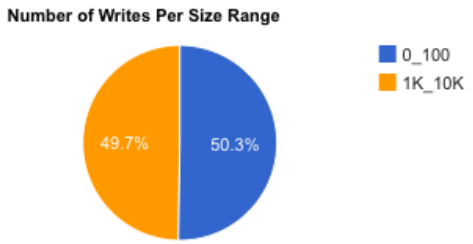
Six dimensional data structure of k points

(ik, iv, ic, ikp, ivp, icp)

- Loop over last two array indices
- Each processor sends data to processor 0
- Processor 0 writes out 4 dimensional array data

IO Summary from Darshan

| Start | End | Wallclock (secs) | MB Read | MB Written | Estimated I/O Rate (MB/sec) | Estimated Percent Time Spent in I/O |
|---|---|---|---|---|---|---|
| 08-09 12:58:44 | 08-09 13:40:47 | 2,523 | 40.6 | 76,298.5 | 91.91 | 32.92% |




Jack Deslippe, LBL

# Memory and I/O Layout of BerkeleyGW after parallel I/O implementation

Six dimensional data structure of k points

A (ik, iv, ic, ikp, ivp, icp)

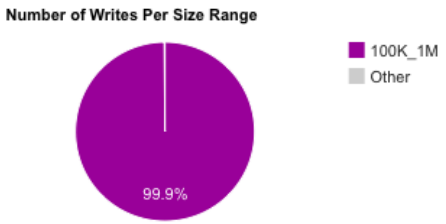
- Use HDF5 Library to parallelize I/O
- Create a 4 dimensional hyperslab
- Loop over last two array indices
- Use Collective I/O
- Writer nodes matches underlying Lustre striping parameters

File layout

A(:,:,:,:,1,1)
A(:,:,:,:,2,1)
A(:,:,:,:,3,1)
A(:,:,:,:,n,1)
A(:,:,:,:,1,2)
A(:,:,:,:,2,2)
A(:,:,:,:,3,1)
A(:,:,:,:,n,n)

IO Summary from Darshan

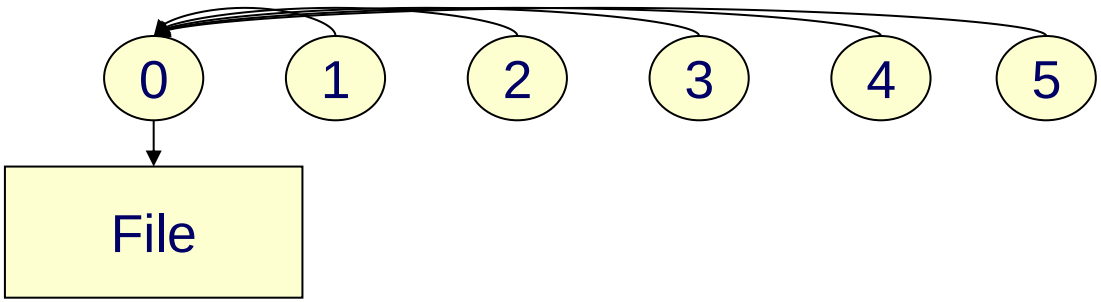
| Start | End | Wallclock (secs) | MB Read | MB Written | Estimated I/O Rate (MB/sec) | Estimated Percent Time Spent in I/O |
|---|---|---|---|---|---|---|
| 08-08 17:17:10 | 08-08 17:22:42 | 332 | 150.8 | 76,294.0 | 1,963.63 | 11.73% |



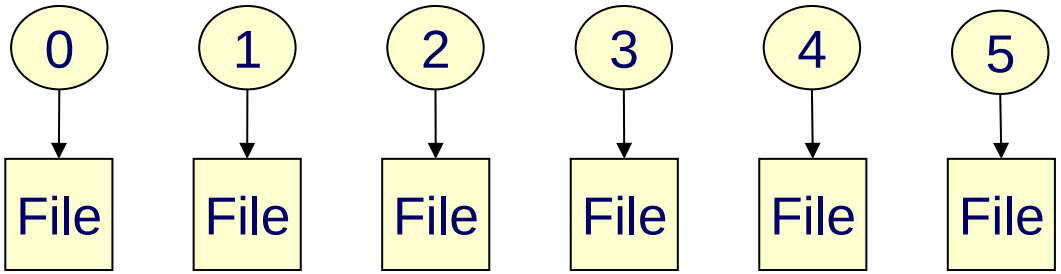

Jack Deslippe, LBL

# Serial, multi-file parallel and shared file parallel I/O

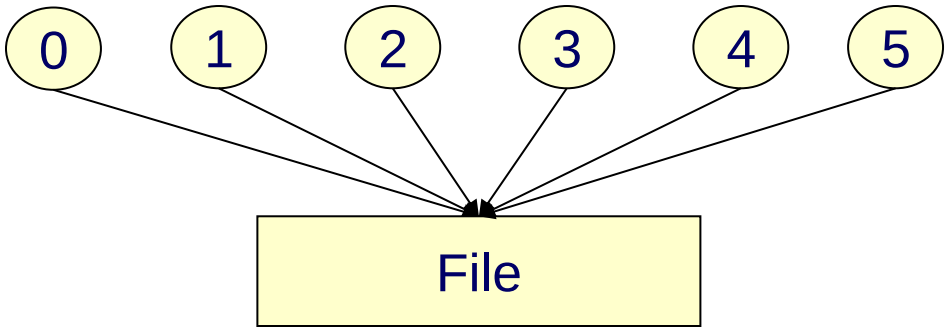
**Serial I/O**



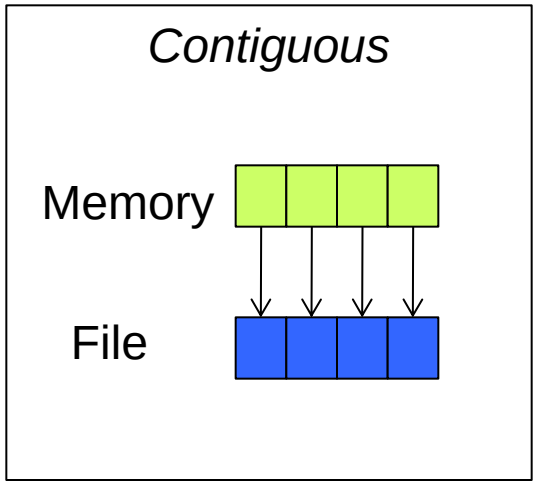

**Parallel Multi-file I/O**

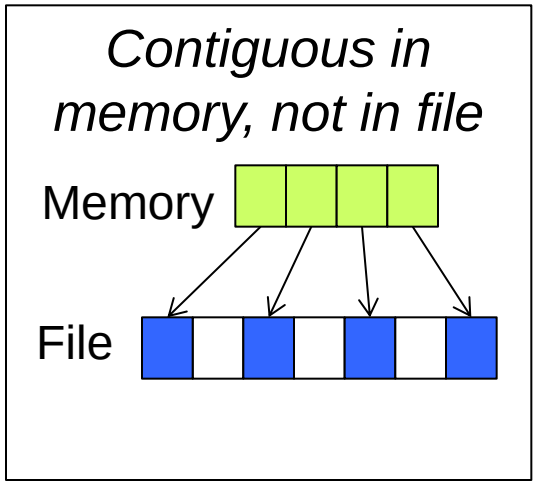


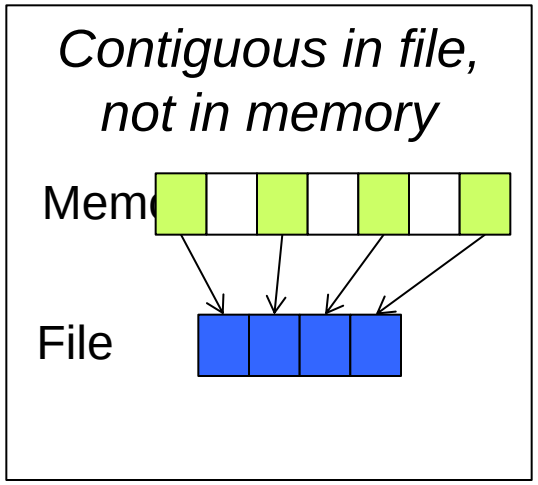

**Parallel Shared-file I/O**

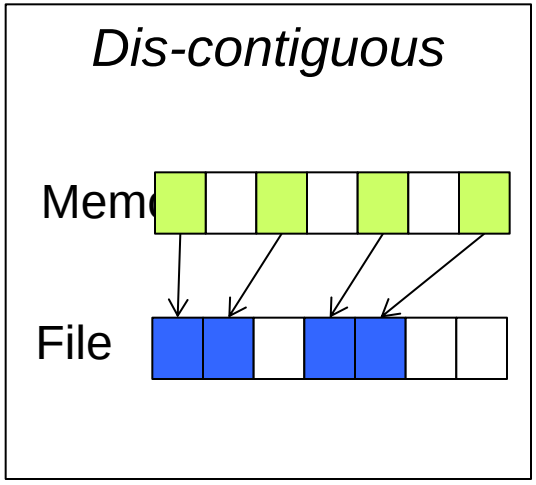


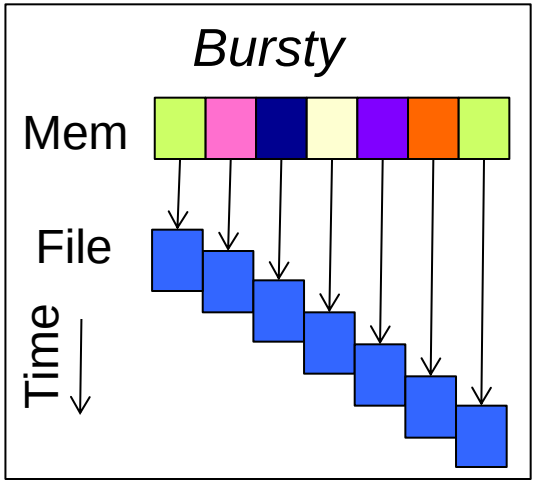
# Access Patterns

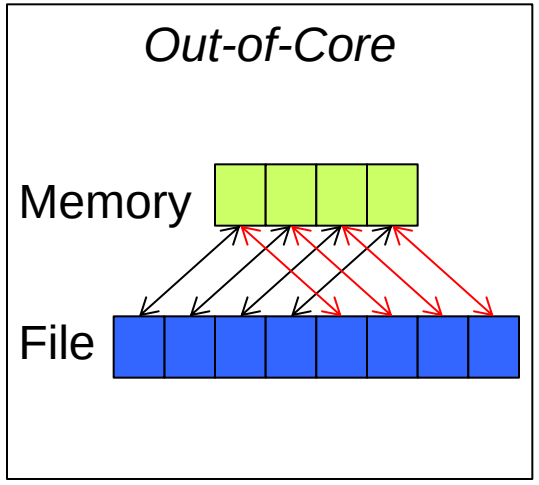


# Case study take-aways

- Scientists have a data model in mind when they think about storing their data for later use
- Goal is to try to preserve as much of that model as possible while moving data to/from storage effiicently
- Different tools available, none are perfect (as we have seen)
- Using the tools can help with productivity, allows for optimizations that can help with performance (and performance portability) as well

# POSIX I/O

*(It stinks but everybody uses it)*

# POSIX I/O

- POSIX is the IEEE Portable Operating System Interface for Computing Environments
- “POSIX defines a standard way for an application program to obtain basic services from the operating system”
  - Mechanism almost all serial applications use to perform I/O
- POSIX was created when a single computer owned its own file system

# What's wrong with POSIX?

- It's a useful, ubiquitous interface for basic I/O
- It lacks constructs useful for parallel I/O
  - Cluster application is really one program running on N nodes, but looks like N programs to the filesystem
  - No support for noncontiguous I/O
  - No hinting/prefetching
- Its rules hurt performance for parallel apps
  - Atomic writes, read-after-write consistency
  - Attribute freshness
- POSIX should not have to be used (directly) in parallel applications that want good performance
  - But developers use it anyway

# Deficiencies in serial interfaces

POSIX:

```
fd = open("some_file", O_WRONLY|O_CREAT,
 S_IRUSR|S_IWUSR);
ret = write(fd, w_data, nbytes);
ret = lseek(fd, 0, SEEK_SET);
ret = read(fd, r_data, nbytes);
ret = close(fd);
```

FORTRAN:

```
OPEN(10, FILE='some_file',  &
    STATUS="replace", &
   ACCESS="direct", RECL=16);
WRITE(10, REC=2) 15324
CLOSE(10);
```

- Typical (serial) I/O calls seen in applications
- No notion of other processors
- Primitive (if any) data description methods
- Tuning limited to open flags
- No mechanism for data portability
  - Fortran not even portable between compilers

# The MPI-IO Interface

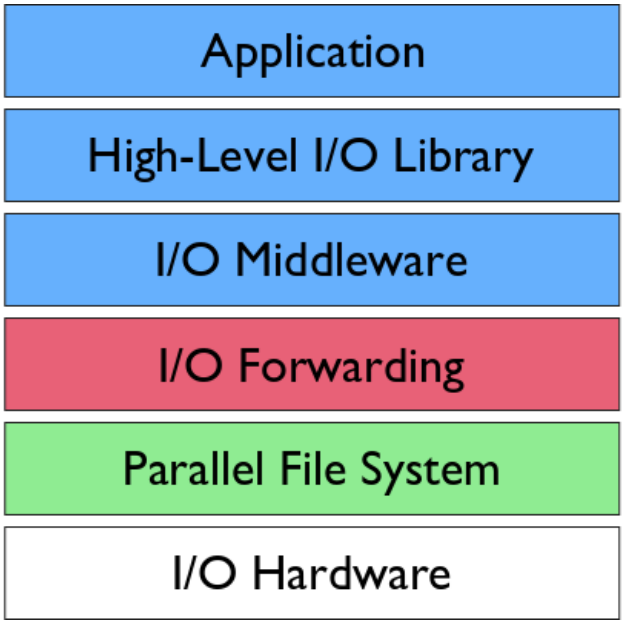
# I/O for Computational Science

**High-Level I/O Library**
maps application abstractions onto storage abstractions and provides data portability.

*HDF5, Parallel netCDF, ADIOS*

**I/O Forwarding**
bridges between app. tasks and storage system and provides aggregation for uncoordinated I/O.

*IBM ciod, IOFSL, Cray DVS*




**I/O Middleware**
organizes accesses from many processes, especially those using collective I/O.

*MPI-IO*

**Parallel File System**
maintains logical space and provides efficient access to data.

*PVFS, PanFS, GPFS, Lustre*

**Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.**

# MPI-IO

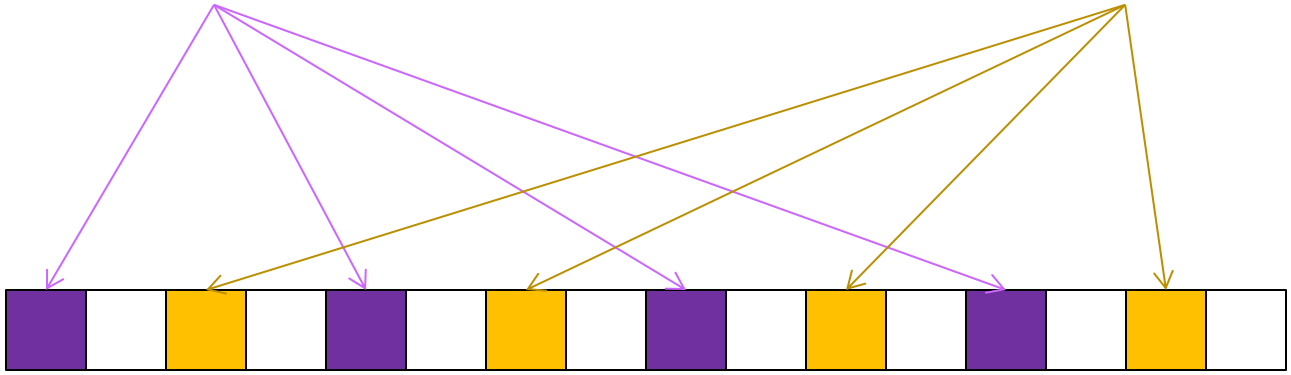
- I/O interface specification for use in MPI apps
- Data model is same as POSIX
  - Stream of bytes in a file
- Features:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - Fortran bindings (and additional languages)
  - System for encoding files in a portable format (external32)
    - Not self-describing - just a well-defined encoding of types
- Implementations available on most platforms (more later)

# Simple MPI-IO

- Collective open: all processes in communicator
- File-side data layout with *file views*
- Memory-side data layout with *MPI datatype* passed to write

```
MPI_File_open(COMM, name, mode,
info, fh);
MPI_File_set_view(fh, disp, etype,
filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
datatype, status);
```

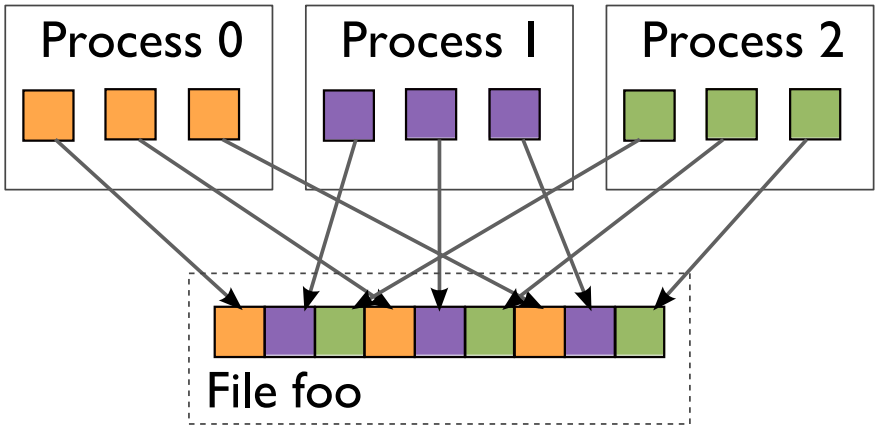
```
MPI_File_open(COMM, name, mode,
    info, fh);
MPI_File_set_view(fh, disp, etype,
filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
datatype, status);
```



# I/O Transformations

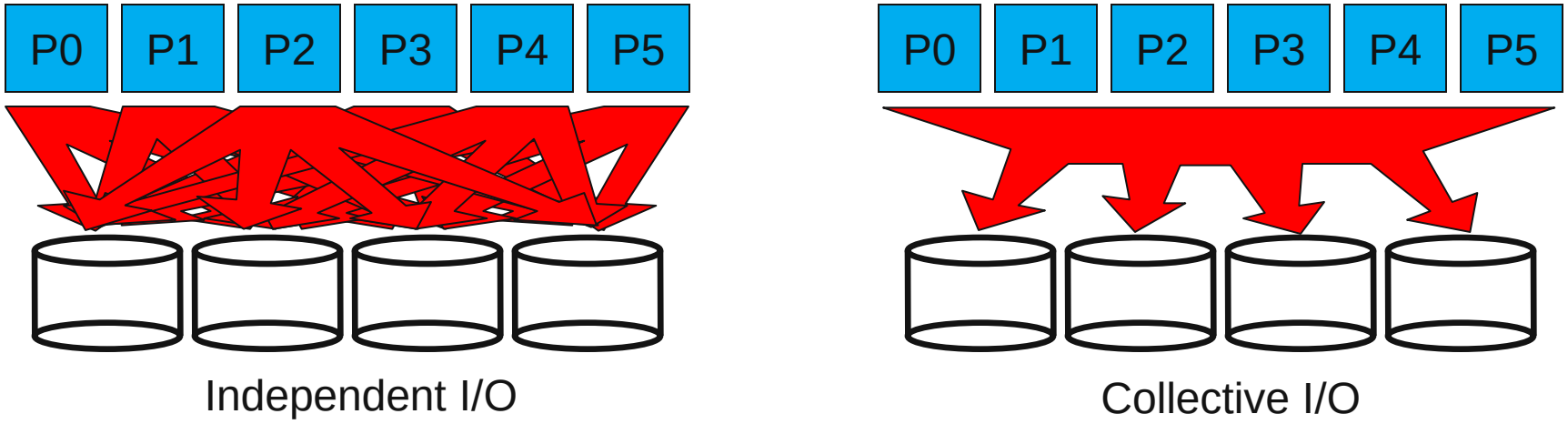
**Software between the application and the PFS performs transformations, primarily to improve performance.**

- Goals of transformations:
  - Reduce number of operations to PFS (avoiding latency)
  - Avoid lock contention (increasing level of concurrency)
  - Hide number of clients (more on this later)
- With "transparent" transformations, data ends up in the same locations in the file
  - i.e., the file system is still aware of the actual data organization



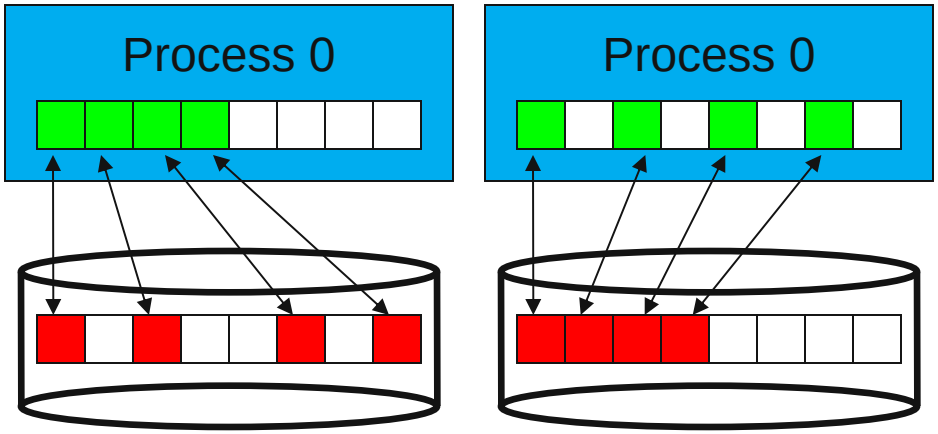

When we think about I/O transformations, we consider the mapping of data between application processes and locations in file.

# Independent and Collective I/O



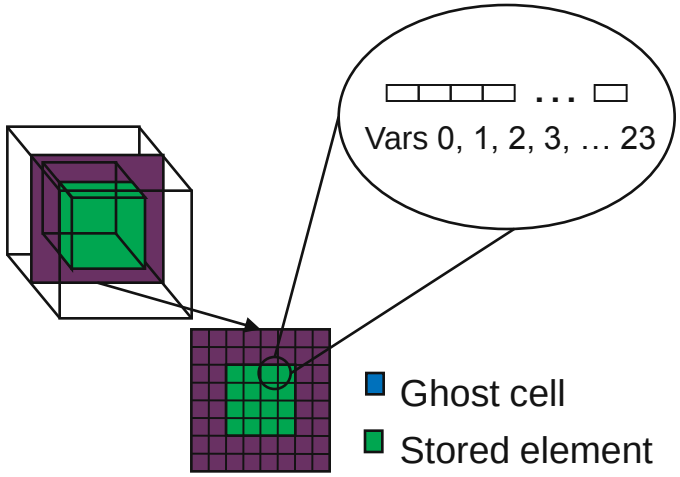

- Independent I/O operations specify only what a single process will do
  - Independent I/O calls do not pass on relationships between I/O on other processes
- Many applications have phases of computation and I/O
  - During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions are called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole, more opportunities for optimization in lower software layers, better performance

# Contiguous and Noncontiguous I/O




Noncontiguous in File

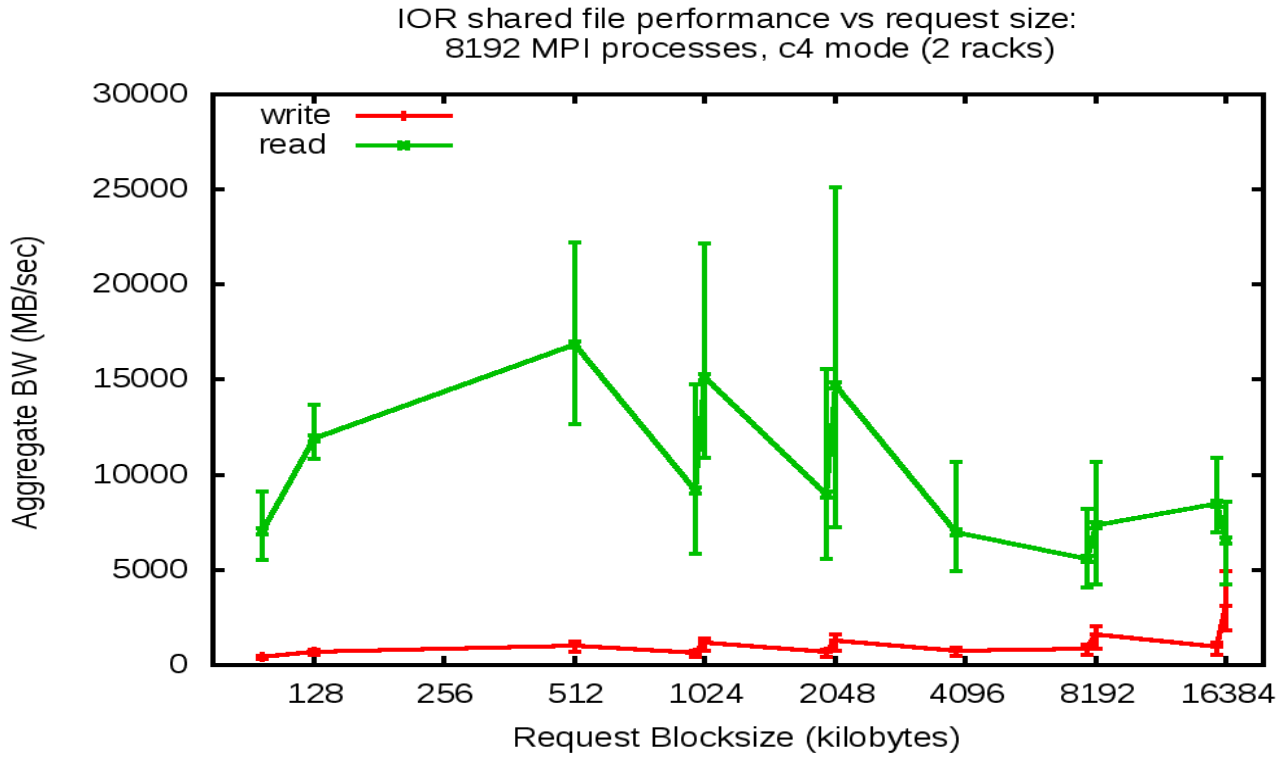
Noncontiguous in Memory




Extracting variables from a block and skipping ghost cells will result in noncontiguous I/O.

- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous accesses with a single operation passes more knowledge to I/O system

# Request Size and I/O Rate



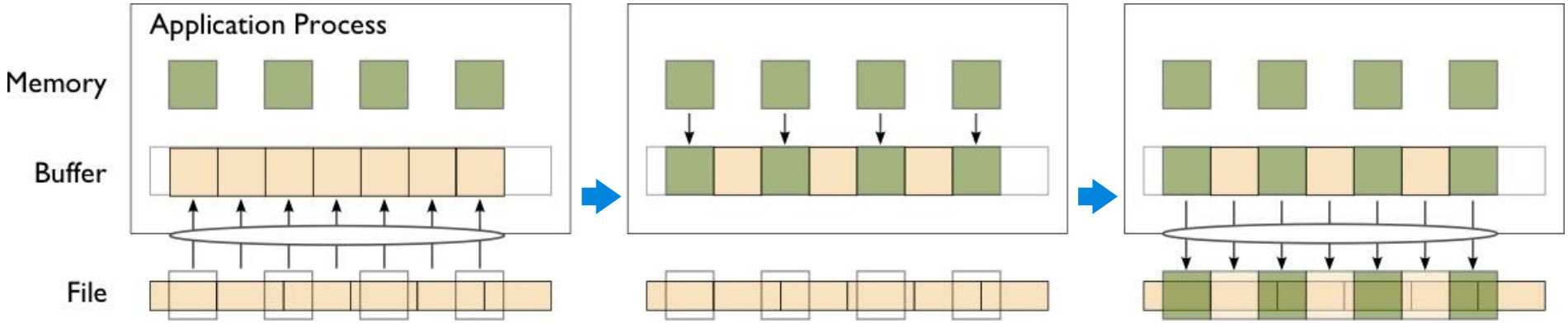

**Interconnect latency has a significant impact on effective rate of I/O. Typically I/O should be in the O(Mbytes) range (at least 16 MiB on this GPFS config).**

Tests run on 8K processes of IBM Blue Gene/Q at ANL.

# Reducing Number of Operations

**Since most operations go over the network, I/O to a PFS incurs more latency than with a local FS.** *Data sieving* is a technique to address I/O latency by combining operations:

- When reading, application process reads a large region holding all needed data and pulls out what is needed
- When writing, three steps required (below)




**Step 1**: Data in region to be modified are read into intermediate buffer (1 read).

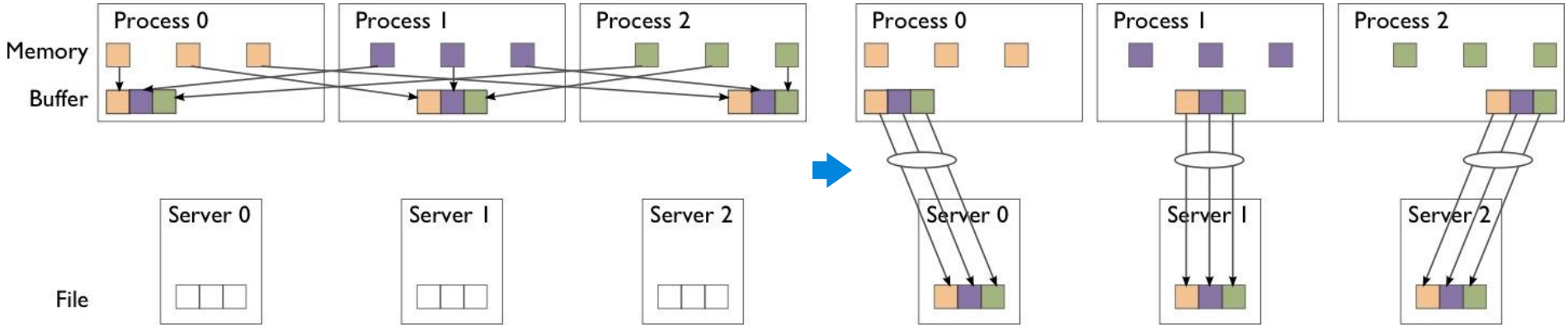
**Step 2**: Elements to be written to file are replaced in intermediate buffer.

**Step 3**: Entire region is written back to storage with a single write operation.

# Avoiding Lock Contention

**To avoid lock contention when writing to a shared file, we can reorganize data between processes.** *Two-phase I/O* splits I/O into a data reorganization phase and an interaction with the storage system (two-phase write depicted):

- Data exchanged between processes to match file layout

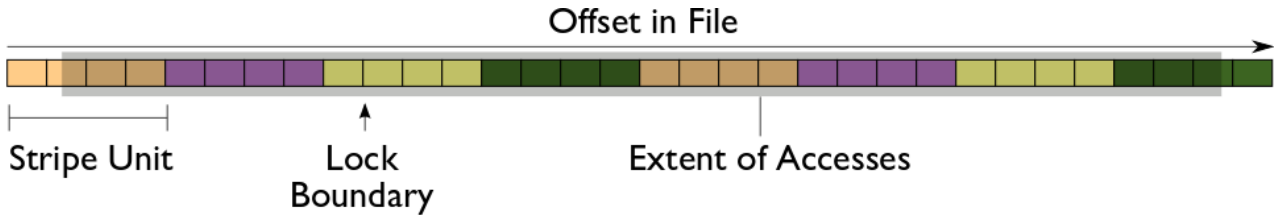



**Phase 1**: Data are exchanged between processes based on organization of data in file.

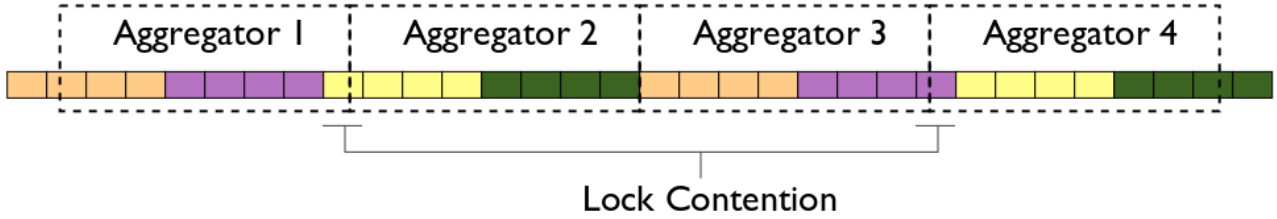
**Phase 2**: Data are written to file (storage servers) with large writes, no contention.

# Two-Phase I/O Algorithms

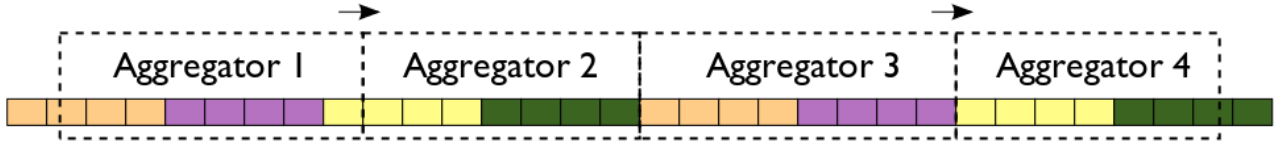
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



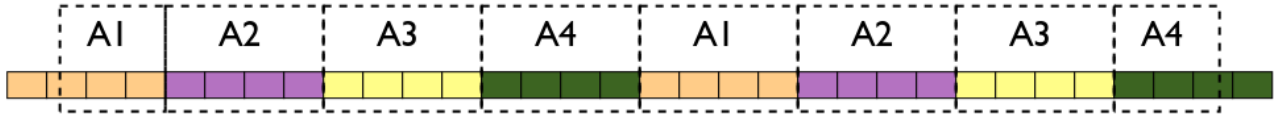

One approach is to evenly divide the region accessed across aggregators.




Aligning regions with lock boundaries eliminates lock contention.




Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).



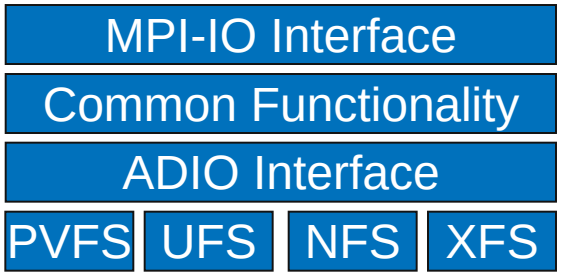

For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November, 2008.

# Impact of Optimizations on S3D I/O

- Testing with PnetCDF output to single file, three configurations, 16 processes
  - All MPI-IO optimizations (collective buffering and data sieving) disabled
  - Independent I/O optimization (data sieving) enabled
  - Collective I/O optimization (collective buffering, a.k.a. two-phase I/O) enabled

| | Coll. Buffering and Data Sieving Disabled | Data Sieving Enabled | Coll. Buffering Enabled (incl. Aggregation) |
|---|---|---|---|
| POSIX writes | 102,401 | 81 | **5** |
| POSIX reads | 0 | 80 | 0 |
| MPI-IO writes | 64 | 64 | 64 |
| Unaligned in file | 102,399 | 80 | 4 |
| Total written (MB) | 6.25 | **87.11** | 6.25 |
| Runtime (sec) | 1443 | 11 | 6.0 |
| Avg. MPI-IO time per proc (sec) | **1426.47** | 4.82 | 0.60 |

# MPI-IO Implementations



ROMIO's layered architecture.

- Different MPI-IO implementations exist
- Three better-known ones are:
  - ROMIO from Argonne National Laboratory
    - Leverages MPI-1 communication
    - Supports local file systems, network file systems, parallel file systems
      - UFS module works GPFS, Lustre, and others
    - Includes data sieving and two-phase optimizations
  - MPI-IO/GPFS from IBM (for AIX only)
    - Includes two special optimizations
      - Data shipping -- mechanism for coordinating access to a file to alleviate lock contention (type of aggregation)
      - Controlled prefetching -- using MPI file views and access patterns to predict regions to be accessed in future
  - MPI from NEC
    - For NEC SX platform and PC clusters with Myrinet, Quadrics, IB, or TCP/IP
    - Includes listless I/O optimization -- fast handling of noncontiguous I/O accesses in MPI layer

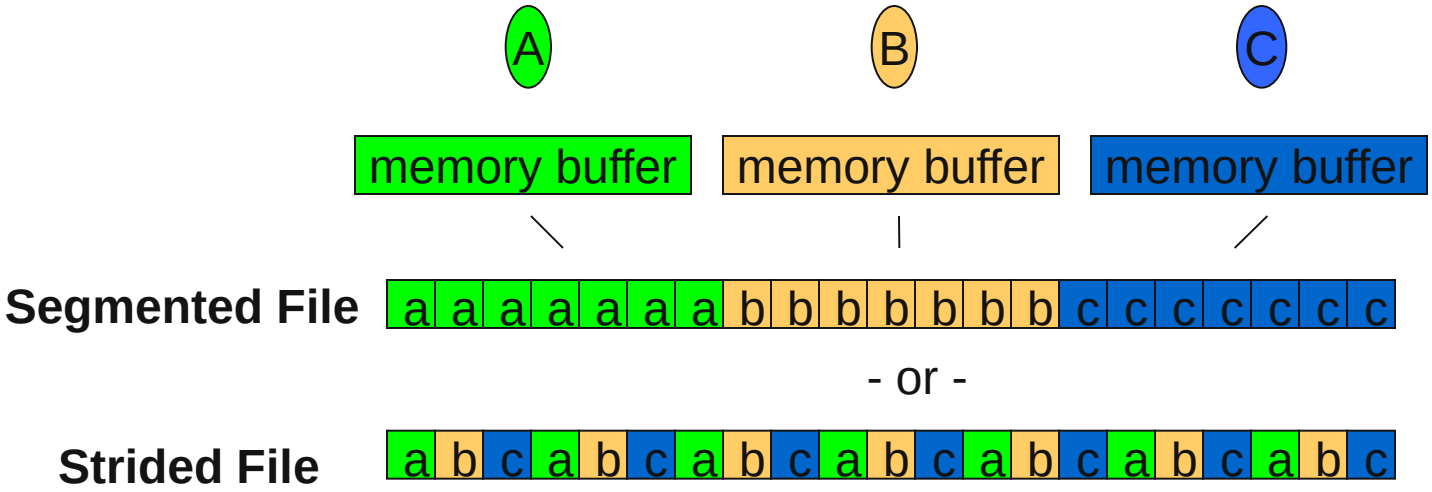
# IOR: File System Bandwidth

- Written at Lawrence Livermore National Laboratory
- Named for the acronym ‘interleaved or random’
- POSIX, MPI-IO, HDF5, and Parallel-NetCDF APIs
  - Shared or independent file access
  - Collective or independent I/O (when available)
- Employs MPI for process synchronization
- Used to obtain peak POSIX I/O rates for shared and separate files
  - Single Shared Output File:

    ./IOR -a POSIX -C -i 3 -t 4M -b 4G -e -v -v -o $FILE
  - One File per Process (-F option)

    ./IOR -a POSIX -C -i 3 -t 4M -b 4G -e -v -v -F -o $FILE

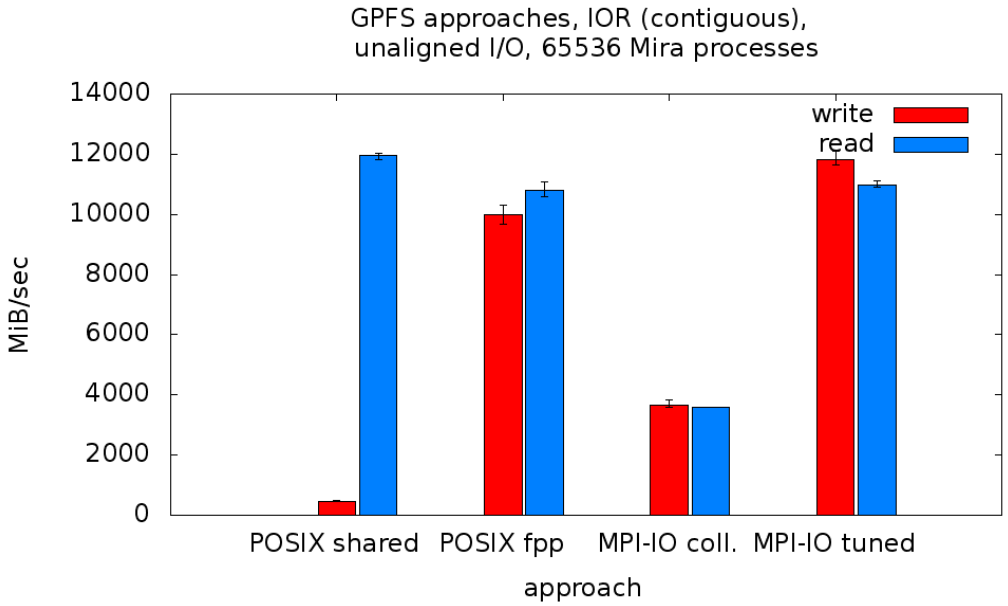
# IOR Access Patterns for Shared Files




- Primary distinction between the two major shared-file patterns is whether each task's data is contiguous or noncontiguous
- For the segmented pattern, each task stores its blocks of data in a contiguous region in the file
- With the strided access pattern, each task's data blocks are spread out through a file and are noncontiguous

# GPFS Access three ways

- POSIX shared vs MPI-IO collective
  - Locking overhead for unaligned writes hits POSIX hard
- Default MPI-IO parameters not ideal
  - Reported to IBM; simple tuning brings MPI-IO back to parity
  - “Vendor Defaults” might give you bad first impression
- File per process (fpp) extremely seductive, but entirely untenable on current generation.



# MPI-IO Wrap-Up

- MPI-IO provides a rich interface allowing us to describe
  - Noncontiguous accesses in memory, file, or both
  - Collective I/O
- This allows implementations to perform many transformations that result in better I/O performance
- Ideal location in software stack for file system specific quirks or optimizations
- Also forms solid basis for high-level I/O libraries
  - But they must take advantage of these features!

# The Parallel netCDF Interface and File Format

Thanks to Wei-Keng Liao, Alok Choudhary, and Kui Gao(NWU) for their help in the development of PnetCDF.

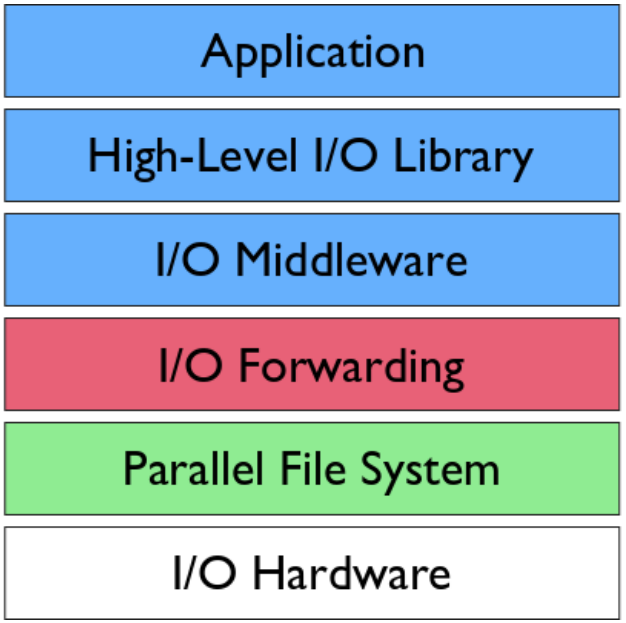
# I/O for Computational Science

**High-Level I/O Library**
maps application abstractions onto storage abstractions and provides data portability.

*HDF5, Parallel netCDF, ADIOS*

**I/O Forwarding**
bridges between app. tasks and storage system and provides aggregation for uncoordinated I/O.

*IBM ciod, IOFSL, Cray DVS*




**I/O Middleware**
organizes accesses from many processes, especially those using collective I/O.

*MPI-IO*

**Parallel File System**
maintains logical space and provides efficient access to data.

*PVFS, PanFS, GPFS, Lustre*

**Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.**

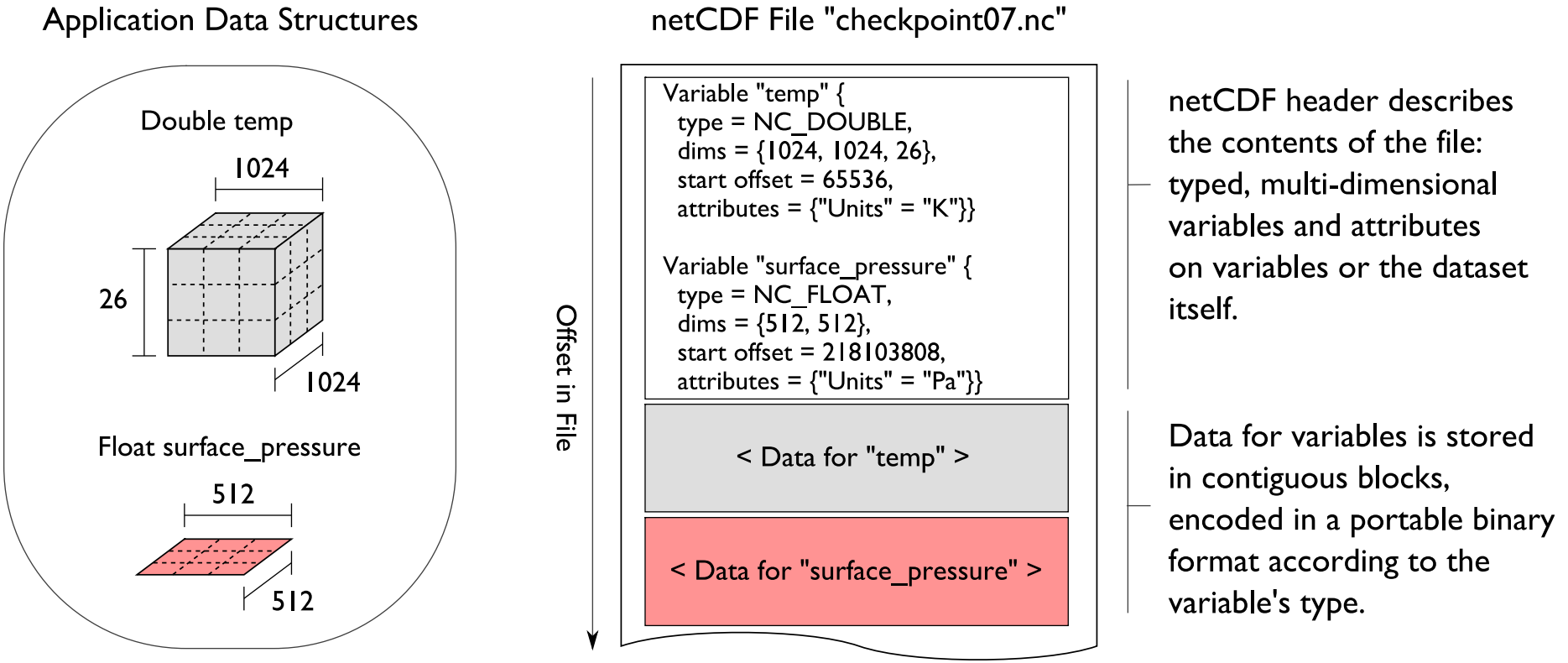
# Higher Level I/O Interfaces

- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data in file
  - Interfaces for discovering contents
- Present APIs more appropriate for computational science
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays and I/O on subsets of these arrays
- Both of our example interfaces are implemented on top of MPI-IO

# Parallel NetCDF (PnetCDF)

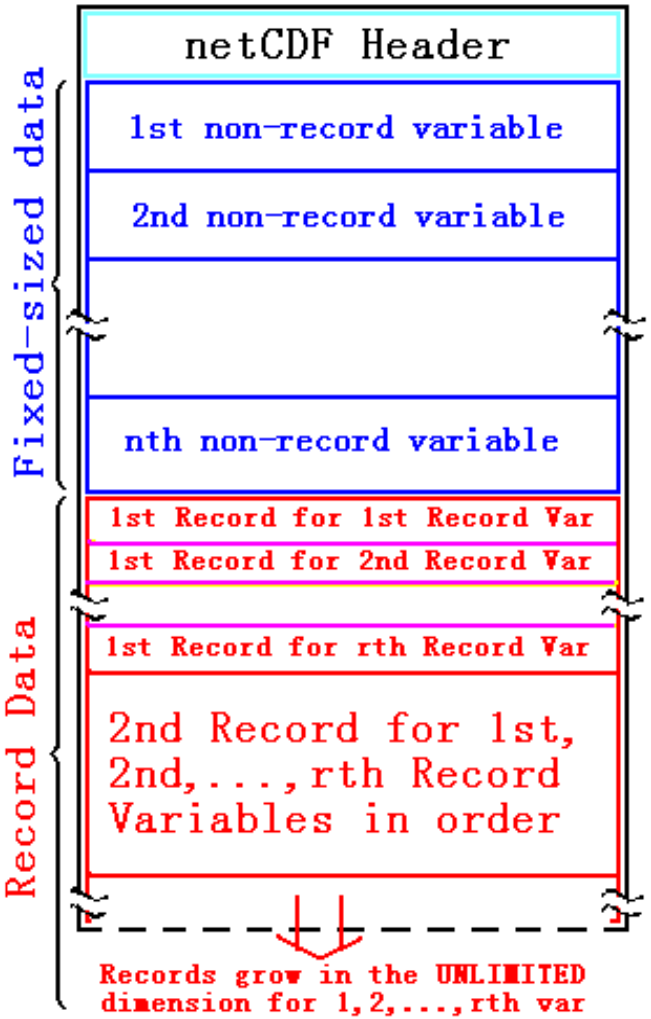
- Based on original “Network Common Data Format” (netCDF) work from Unidata
  - Derived from their source code
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C, Fortran, and F90 interfaces
  - Portable data format (identical to netCDF)
  - Noncontiguous I/O in memory using MPI datatypes
  - Noncontiguous I/O in file using sub-arrays
  - Collective I/O
  - Non-blocking I/O
- Unrelated to netCDF-4 work
- Parallel-NetCDF tutorial:
  - http://trac.mcs.anl.gov/projects/parallel-netcdf/wiki/QuickTutorial

# Data Layout in netCDF Files



# Record Variables in netCDF

- Record variables are defined to have a single “unlimited” dimension
  - Convenient when a dimension size is unknown at time of variable creation
- Record variables are stored after all the other variables in an interleaved format
  - Using more than one in a file is likely to result in poor performance due to number of noncontiguous accesses



# Storing Data in PnetCDF

- Create a dataset (file)
  - Puts dataset in define mode
  - Allows us to describe the contents
    - Define dimensions for variables
    - Define variables using dimensions
    - Store attributes if desired (for variable or dataset)
- Switch from define mode to data mode to write variables
- Store variable data
- Close the dataset

# Example: FLASH with PnetCDF

- FLASH AMR structures do not map directly to netCDF multidimensional arrays
- Must create mapping of the in-memory FLASH data structures into a representation in netCDF multidimensional arrays
- Chose to
  - Place all checkpoint data in a single file
  - Impose a linear ordering on the AMR blocks
    - Use 4D variables
  - Store each FLASH variable in its own netCDF variable
    - Skip ghost cells
  - Record attributes describing run time, total blocks, etc.

# Defining Dimensions

```
int status, ncid, dim_tot_blks, dim_nxb,
  dim_nyb, dim_nzb;
MPI_Info hints;
/* create dataset (file) */
status = ncmpi_create(MPI_COMM_WORLD, filename,
  NC_CLOBBER, hints, &file_id);
/* define dimensions */
status = ncmpi_def_dim(ncid, "dim_tot_blks",
  tot_blks, &dim_tot_blks);
status = ncmpi_def_dim(ncid, "dim_nxb",
  nzones_block[0], &dim_nxb);
status = ncmpi_def_dim(ncid, "dim_nyb",
  nzones_block[1], &dim_nyb);
status = ncmpi_def_dim(ncid, "dim_nzb",
  nzones_block[2], &dim_nzb);
```

Each dimension gets a unique reference

# Creating Variables

```
int dims = 4, dimids[4];
intvarids[NVARS];
/* define variables (X changes most quickly) */
dimids[0] = dim_tot_blks;
dimids[1] = dim_nzb;
dimids[2] = dim_nyb;
dimids[3] = dim_nxb;
for (i=0; i< NVARS; i++) {
   status = ncmpi_def_var(ncid, unk_label[i],
     NC_DOUBLE, dims, dimids, &varids[i]);
}
```

Same dimensions used for all variables

# Storing Attributes

```
/* store attributes of checkpoint */
status = ncmpi_put_att_text(ncid, NC_GLOBAL,
   "file_creation_time", string_size,
   file_creation_time);
status = ncmpi_put_att_int(ncid, NC_GLOBAL,
   "total_blocks", NC_INT, 1, tot_blks);
status = ncmpi_enddef(file_id);

/* now in data mode … */
```

# Writing Variables

```
double *unknowns; /* unknowns[blk][nzb][nyb][nxb]
   */
size_t start_4d[4], count_4d[4];
start_4d[0] = global_offset; /* different for each
   process */
start_4d[1] = start_4d[2] = start_4d[3] = 0;
count_4d[0] = local_blocks;
count_4d[1] = nzb;  count_4d[2] = nyb;
   count_4d[3] = nxb;
for (i=0; i< NVARS; i++) {
    /* ... build datatype "mpi_type" describing
       values of a single variable ... */
    /* collectively write out all values of a
       single variable */
    ncmpi_put_vara_all(ncid, varids[i], start_4d,
       count_4d, unknowns, 1, mpi_type);
}
status = ncmpi_close(file_id);
```

Typical MPI buffer-count-type tuple

# Inside PnetCDF Define Mode

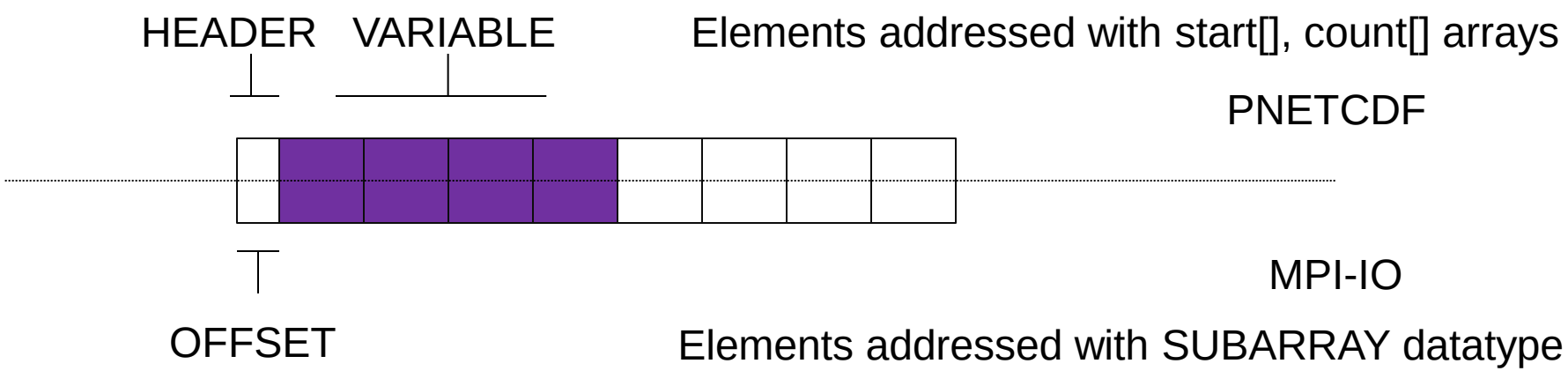
- In define mode (collective)
  - Use `MPI_File_open` to create file at create time
  - Set hints as appropriate (more later)
  - Locally cache header information in memory
    - All changes are made to local copies at each process
- At ncmpi_enddef
  - Process 0 writes header with `MPI_File_write_at`
  - `MPI_Bcast` result to others
  - Everyone has header data in memory, understands placement of all variables
    - No need for any additional header I/O during data mode!

# Inside PnetCDF Data Mode

- Inside `ncmpi_put_vara_all` (once per variable)
  - Each process performs data conversion into internal buffer
  - Uses `MPI_File_set_view` to define file region
    - Contiguous region for each process in FLASH case
  - `MPI_File_write_all` collectively writes data
- At ncmpi_close
  - `MPI_File_close` ensures data is written to storage

- MPI-IO performs optimizations
  - Two-phase possibly applied when writing variables
- MPI-IO makes PFS calls
  - PFS client code communicates with servers and stores data

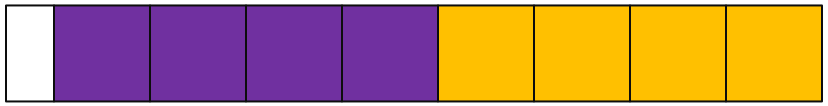
# Parallel-NetCDF and MPI-IO



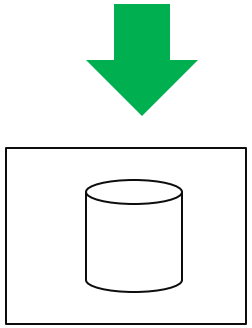

- `ncmpi_put_vara_all` describes access in terms of arrays, elements of arrays
  - E.g. Give me a 3x3 subcube of this larger 1024x1024 array
- Library translates into MPI-IO calls
  - `MPI_Type_create_subarray`
  - `MPI_File_set_view`
  - `MPI_File_write_all`

# Parallel-NetCDF write-combining optimization

```
ncmpi_iput_vara(ncfile, varid1,
    &start, &count, &data,
    count, MPI_INT, &requests[0]);
ncmpi_iput_vara(ncfile, varid2,
    &start,&count, &data,
    count, MPI_INT, &requests[1]);
ncmpi_wait_all(ncfile, 2, requests, statuses);
```



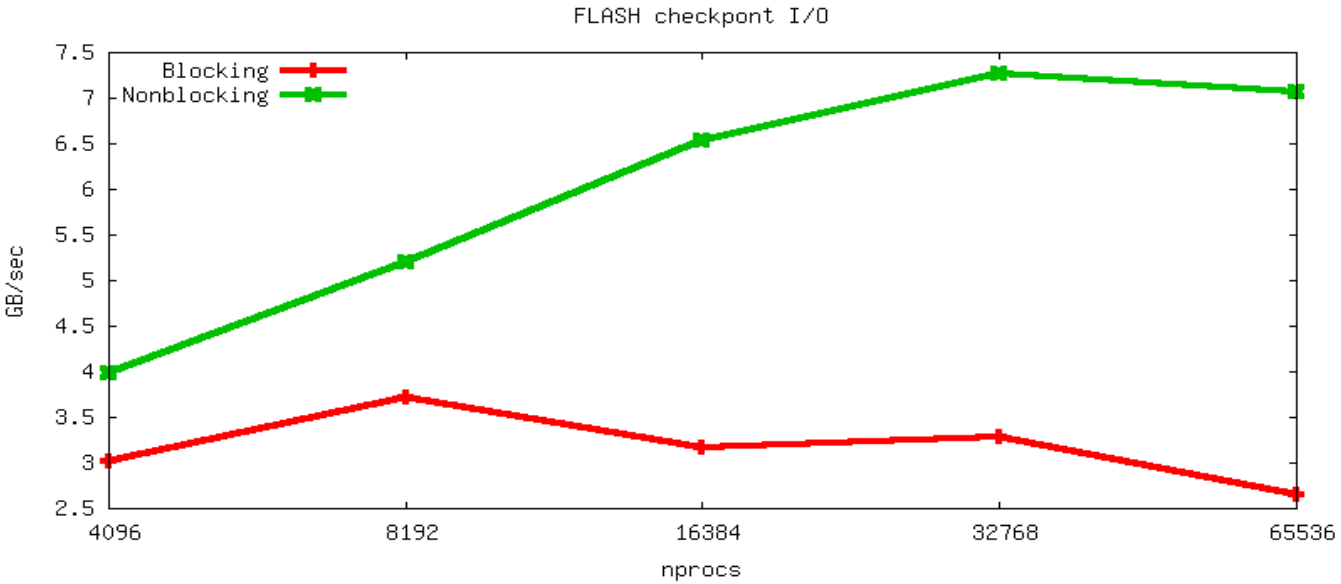
HEADER VAR1 VAR2



- netCDF variables laid out contiguously
- Applications typically store data in separate variables
  - temperature(lat, long, elevation)
  - Velocity_x(x, y, z, timestep)
- Operations posted independently, completed collectively
  - Defer, coalesce synchronization
  - Increase average request size

# FLASH Astrophysics and the write-combining optimization

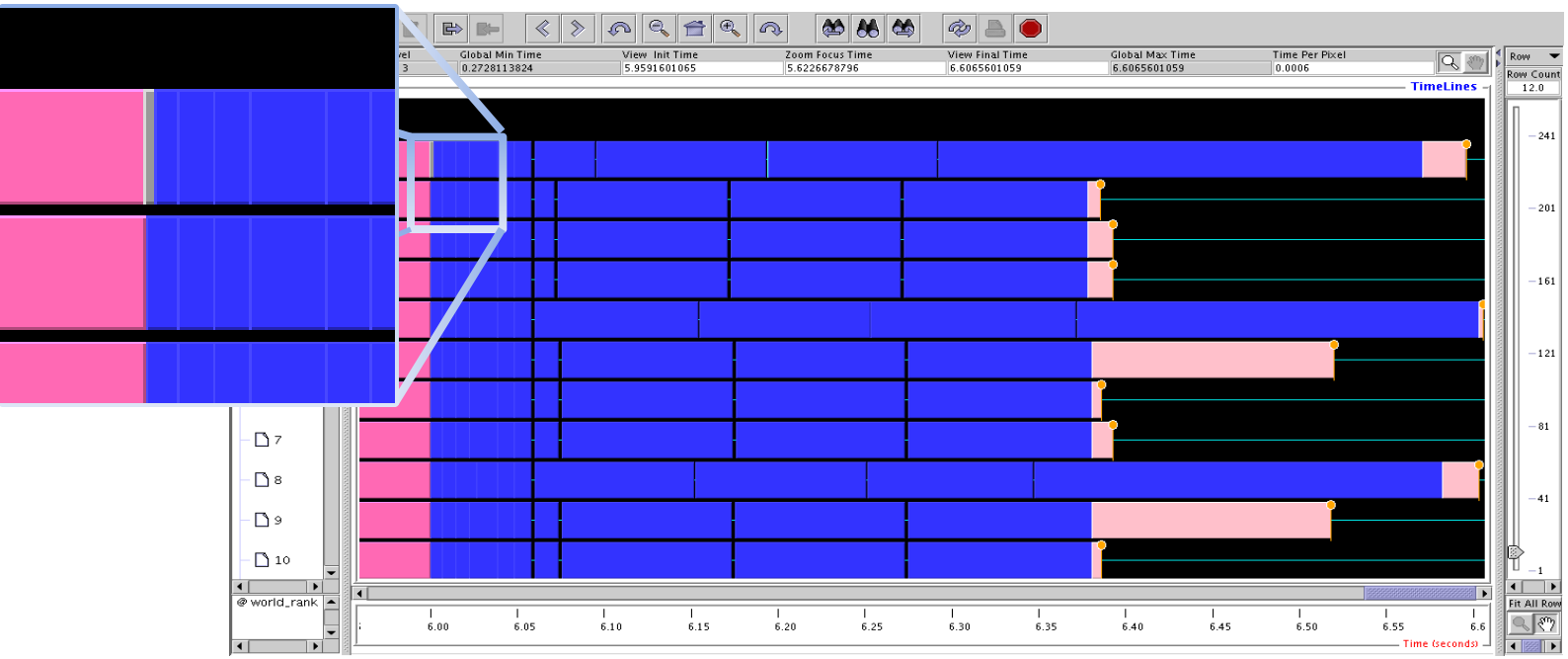
- FLASH writes one variable at a time
- Could combine all  4D variables (temperature, pressure, etc) into one 5D variable
  - Altered file format (conventions) requires updating entire analysis toolchain
- Write-combining provides improved performance with same file conventions
  - Larger requests, less synchronization.
  - Convinced HDF to develop similar interface



# Inside Parallel netCDF: Jumpshot view

1: Rank 0 write header (independent I/O)

3: Collectively write 4 variables




I/O Aggregator

2: Collectively write app grid, AMR data

4: Close file

File open | Indep. write | Collective write | File close

# PnetCDF Wrap-Up

- PnetCDF gives us
  - Simple, portable, self-describing container for data
  - Collective I/O
  - Data structures closely mapping to the variables described
- If PnetCDF meets application needs, it is likely to give good performance
  - Type conversion to portable format does add overhead
- Some limits on (old, common CDF-2) file format:
  - Fixed-size variable:  < 4 GiB
  - Per-record size of record variable: < 4 GiB
  - $2^{32}$ -1 records
  - New extended file format to relax these limits (CDF-5, released in pnetcdf-1.1.0; Unidata NetCDF-4.4 will support it as well)

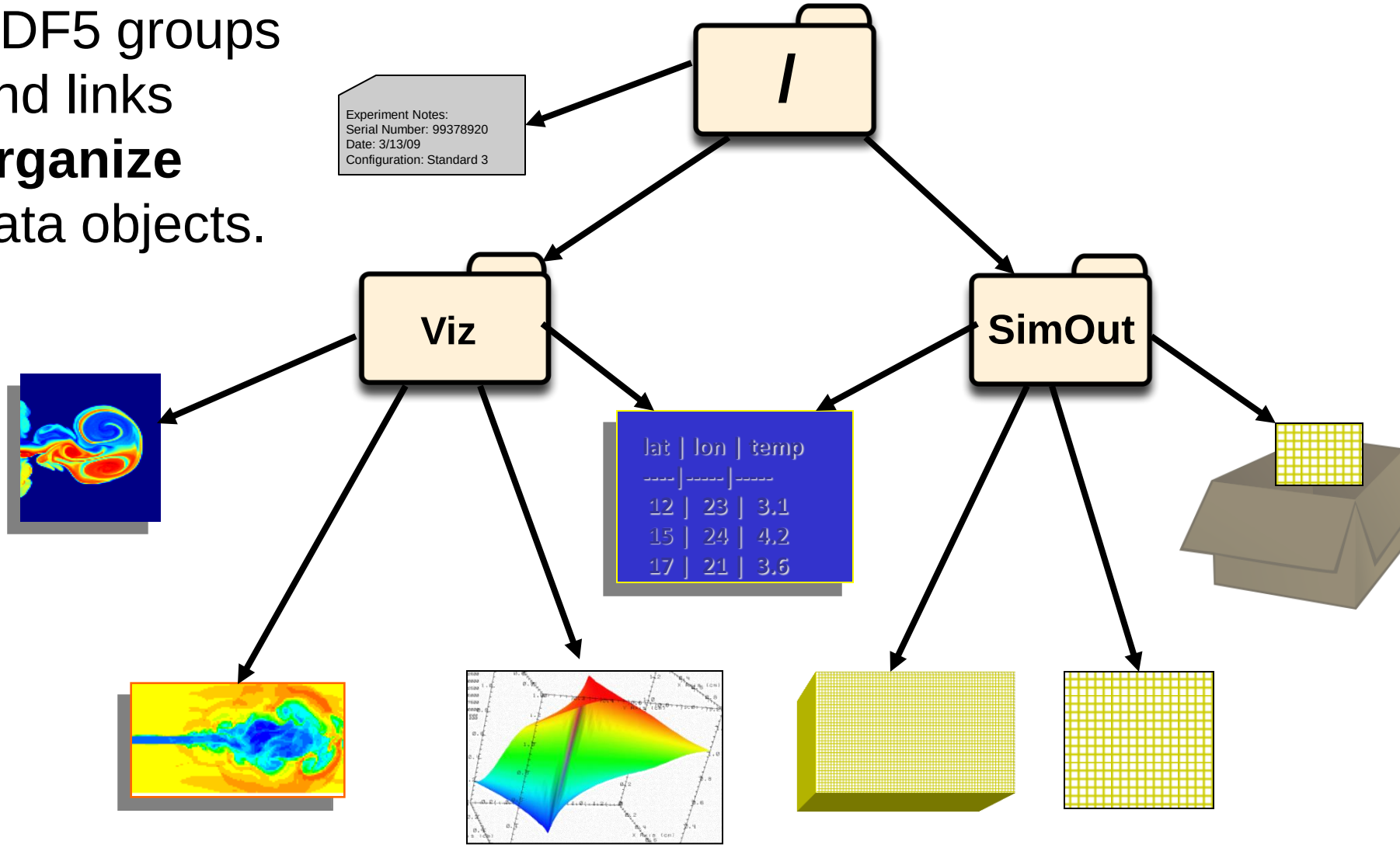
# The HDF5 Interface and File Format

# HDF5

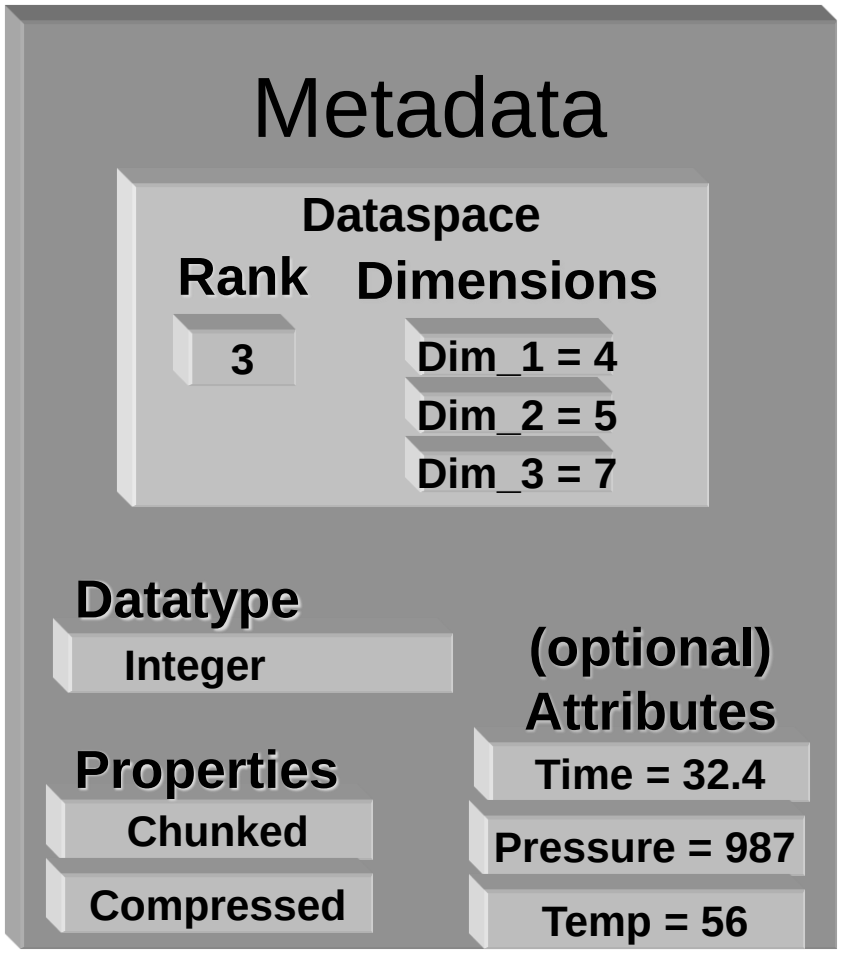
- Hierarchical Data Format, from the HDF Group (formerly of NCSA)
- Data Model:
  - Hierarchical data organization in single file
  - Typed, multidimensional array storage
  - Attributes on dataset, data
- Features:
  - C, C++, and Fortran interfaces
  - Portable data format
  - Optional compression (not in parallel I/O mode)
  - Data reordering (chunking)
  - NoncontiguousI/O (memory and file) with hyperslabs
- Parallel HDF5 tutorial:
  - http://www.hdfgroup.org/HDF5/Tutor/parallel.html

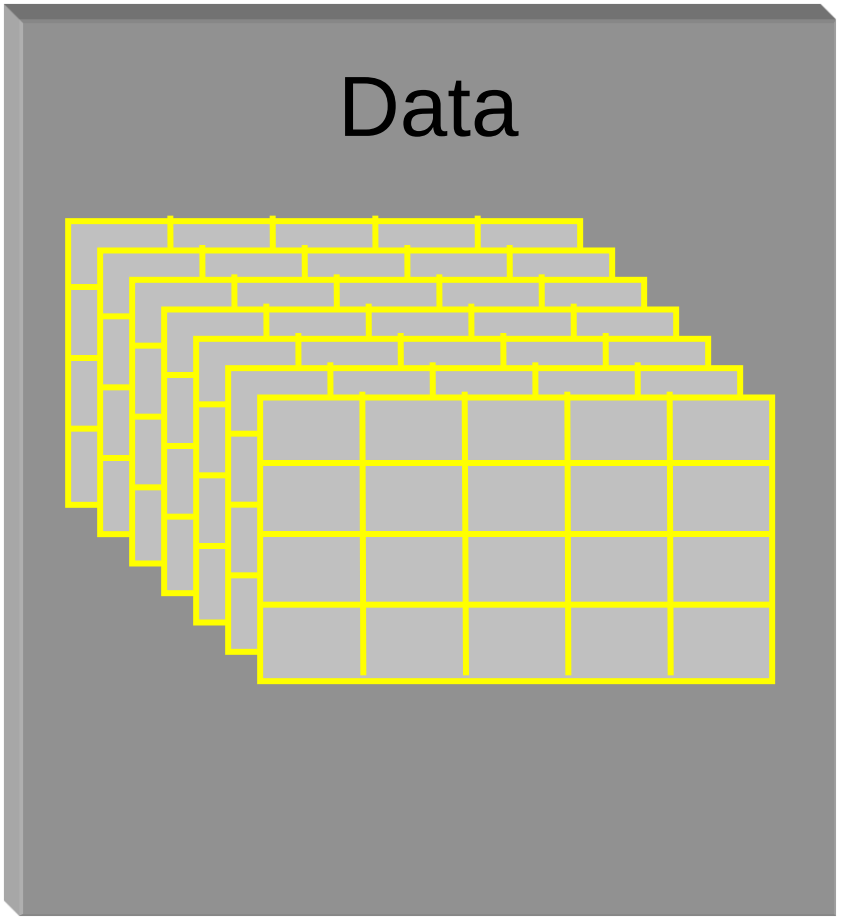
# HDF5 Groups and Links

HDF5 groups and links **organize** data objects.

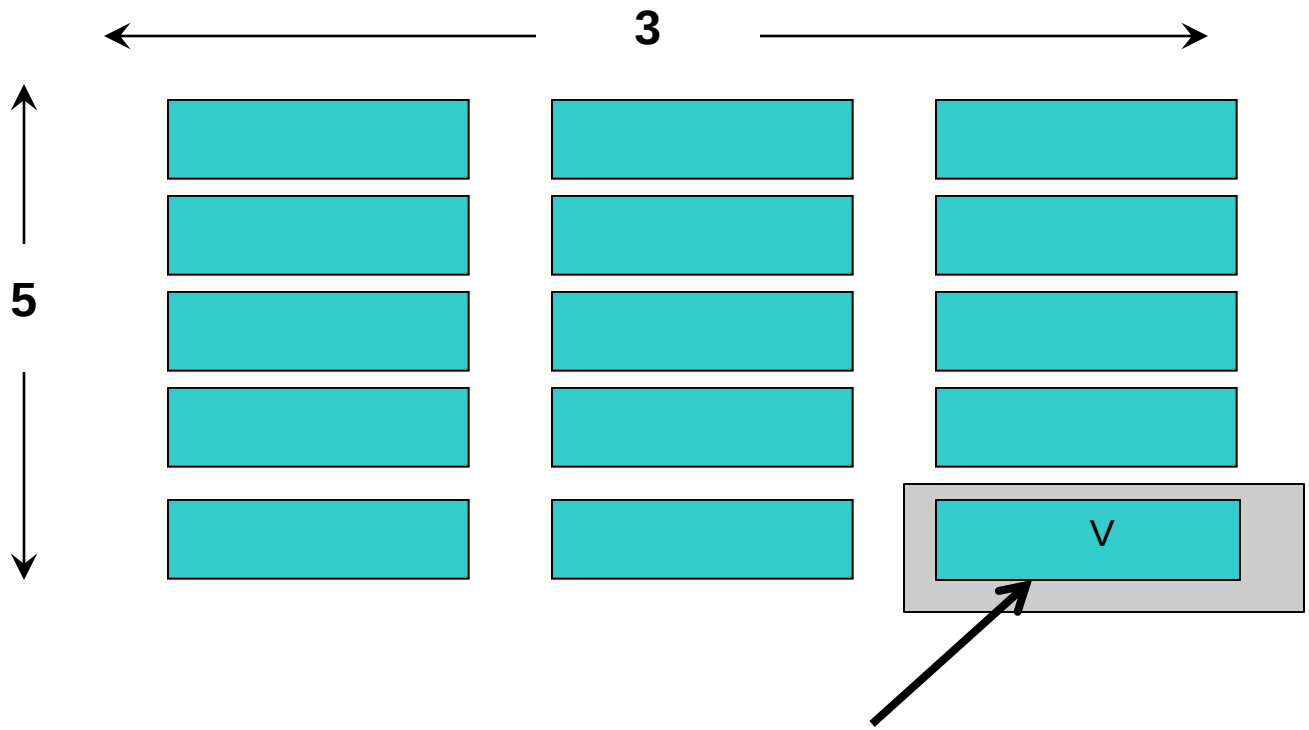


# HDF5 Dataset



# HDF5 Dataset




**Datatype:** **16-byte integer**

**Dataspace:** **Rank = 2**

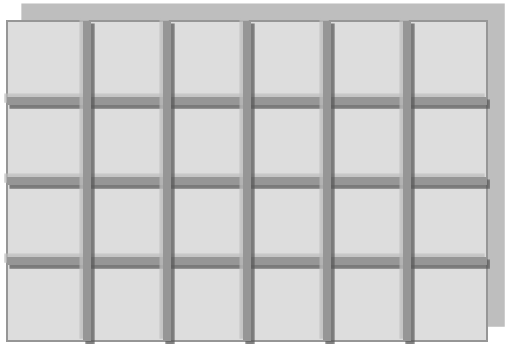
**Dimensions = 5 x 3**

# HDF5 Dataspaces

Two roles:

Dataspace contains spatial information (logical layout) about a dataset stored in a file

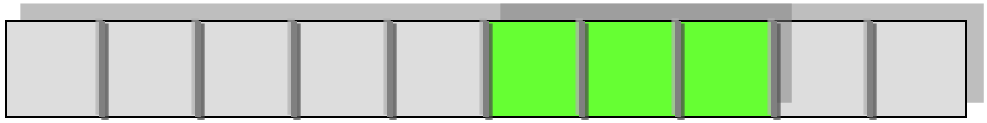
- Rank and dimensions
- Permanent part of dataset definition



Rank = 2

Dimensions = 4x6

Subsets: Dataspace describes application's data buffer and data elements participating in I/O



Rank = 1

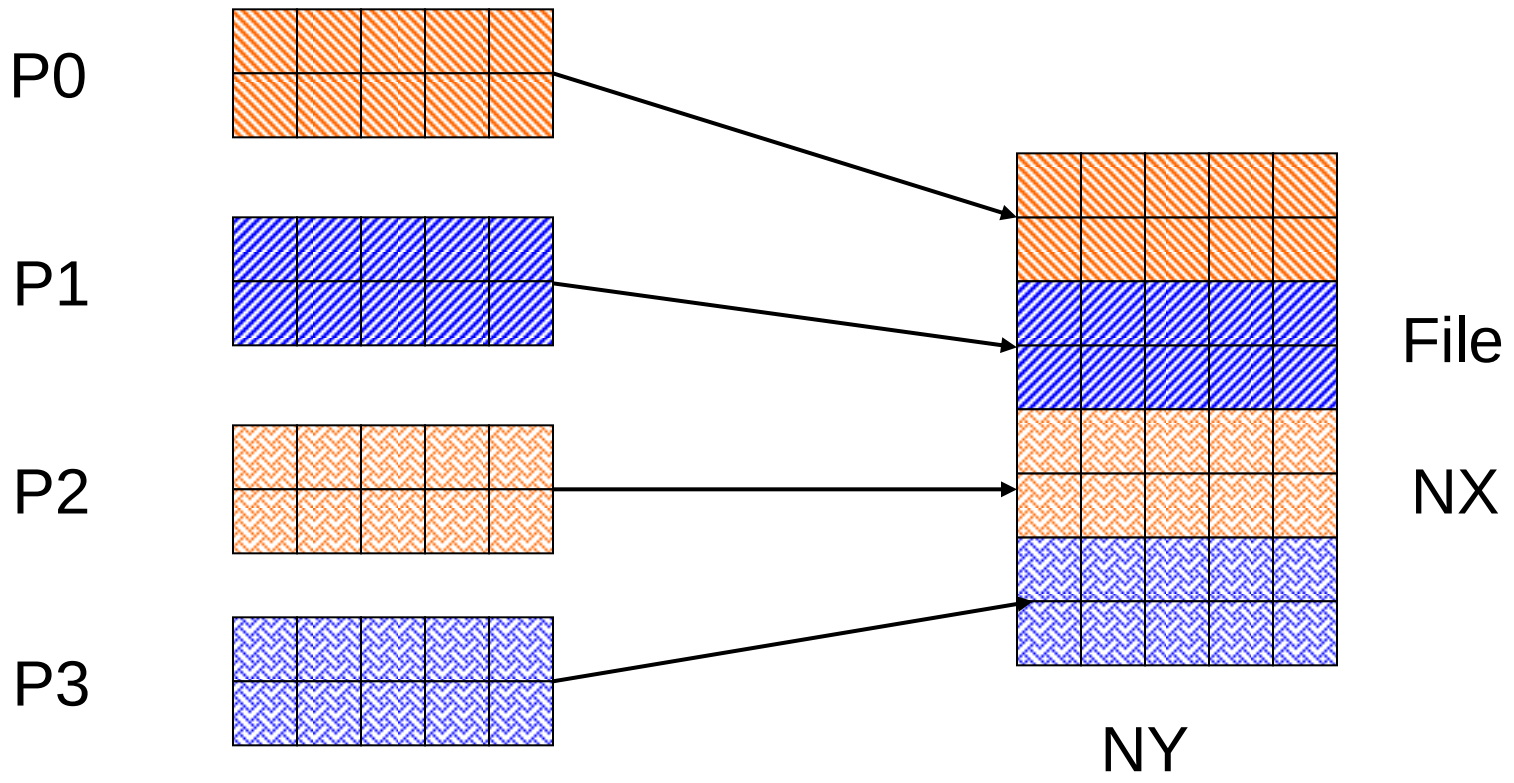
Dimension = 10

# Basic Functions

| | |
|---|---|
| H5**F**create (H5**F**open) | *create (open) File* |
| H5**S**create_simple/H5**S**create | *create dataSpace* |
| H5**D**create (H5**D**open) | *create (open) Dataset* |
| H5**S**select_hyperslab | *select subsections of data* |
| H5**D**read, H5**D**write | *access Dataset* |
| H5**D**close | *close Dataset* |
| H5**S**close | *close dataSpace* |
| H5**F**close | *close File* |

*NOTE: Order not strictly specified.*

# Example: Writing dataset by rows



# Writing by rows: Output of h5dump

```
HDF5 "grid_rows.h5" {
GROUP "/" {
 DATASET "dataset1" {
       DATATYPE  H5T_IEEE_F64LE
       DATASPACE  SIMPLE { ( 8, 5 ) / ( 8, 5 ) }
       DATA {
          18, 18, 18, 18, 18,
          18, 18, 18, 18, 18,
          19, 19, 19, 19, 19,
          19, 19, 19, 19, 19,
          20, 20, 20, 20, 20,
          20, 20, 20, 20, 20,
          21, 21, 21, 21, 21,
          21, 21, 21, 21, 21
       }
    }
}
```

# Initialize the file for parallel access

```
/* first initialize MPI */

/* create access property list */
plist_id = H5Pcreate(H5P_FILE_ACCESS);

/* necessary for parallel access */
status = H5Pset_fapl_mpio(plist_id,
MPI_COMM_WORLD, MPI_INFO_NULL);

/* Create an hdf5 file */
file_id = H5Fcreate(FILENAME, H5F_ACC_TRUNC,
H5P_DEFAULT, plist_id);

status = H5Pclose(plist_id);
```

# Create file dataspace and dataset

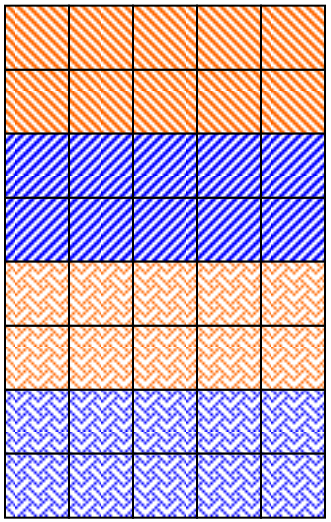
```
/* initialize local grid data */

/* Create the dataspace */


dimsf[0] = NX;
dimsf[1] = NY;


filespace = H5Screate_simple(RANK, dimsf,NULL);

/* create a dataset */
dset_id = H5Dcreate(file_id, "dataset1”,
H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT, H5P_DEFAULT,
H5P_DEFAULT);
```



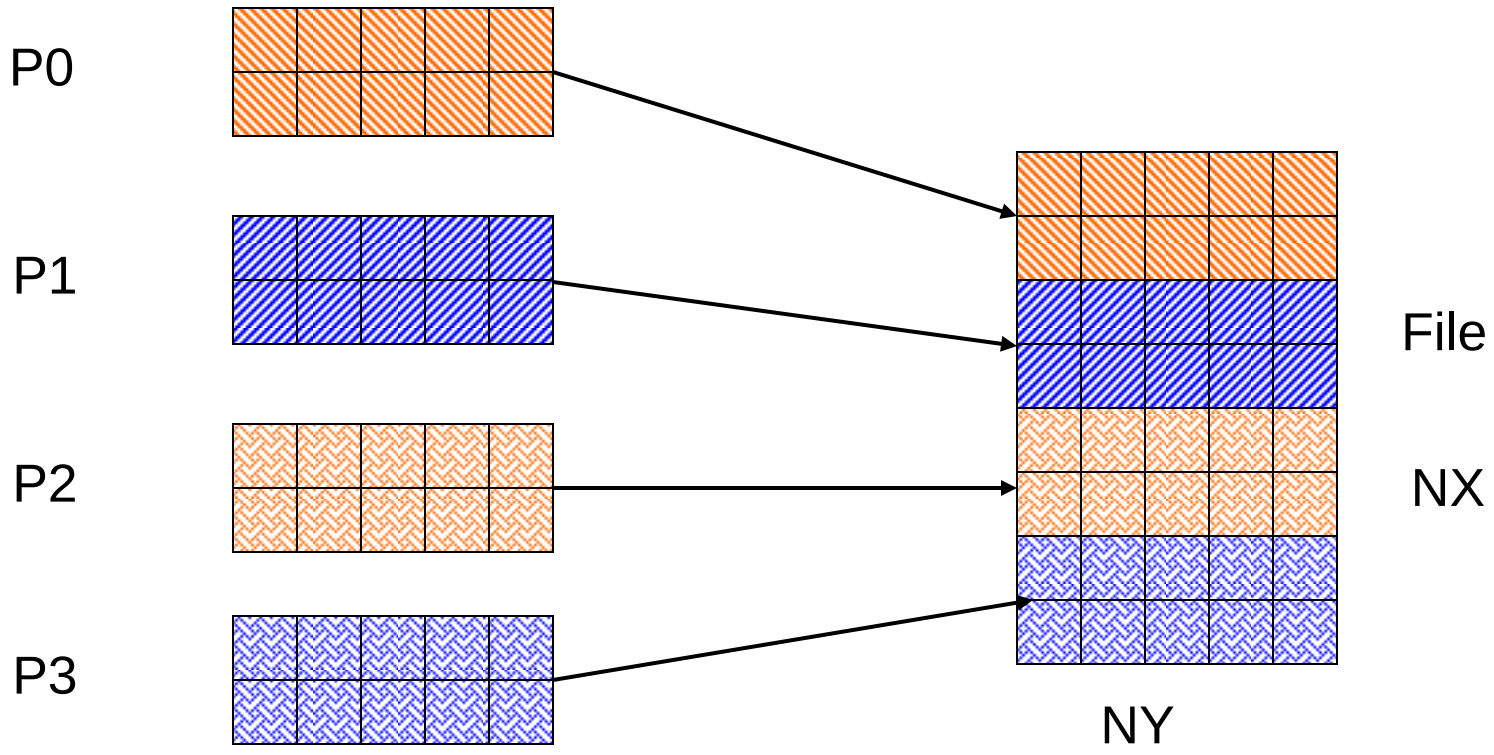
# Create Property List

```
/* Create property list for collective dataset
write. */

plist_id = H5Pcreate(H5P_DATASET_XFER);

/* The other option is HDFD_MPIO_INDEPENDENT */
H5Pset_dxpl_mpio(plist_id,H5FD_MPIO_COLLECTIVE);
```

# Calculate Offsets

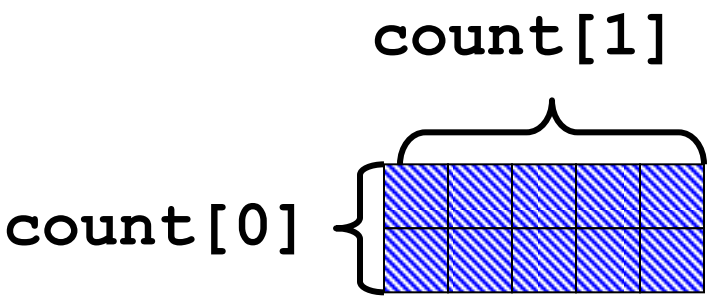



Every processor has a 2d array, which holds the number of blocks to write and the starting offset

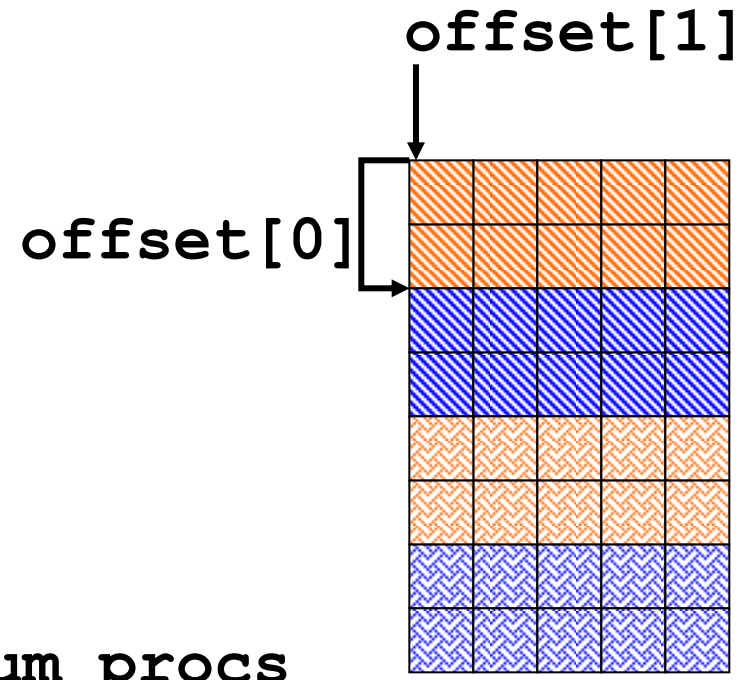
# Example: Writing dataset by rows

Process 1

Memory




File

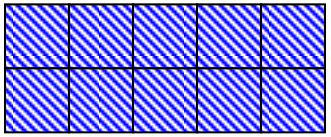



```
count[0] = dimsf[0]/num_procs
count[1] = dimsf[1];
offset[0] = my_proc * count[0];   /* = 2 */
offset[1] = 0;
```

# Writing and Reading Hyperslabs

- Distributed memory model: data is split among processes
- PHDF5 uses HDF5 hyperslab model
- Each process defines memory and file hyperslabs
- Each process executes partial write/read call
  - Collective calls
  - Independent calls

# Create a Memory Space select hyperslab



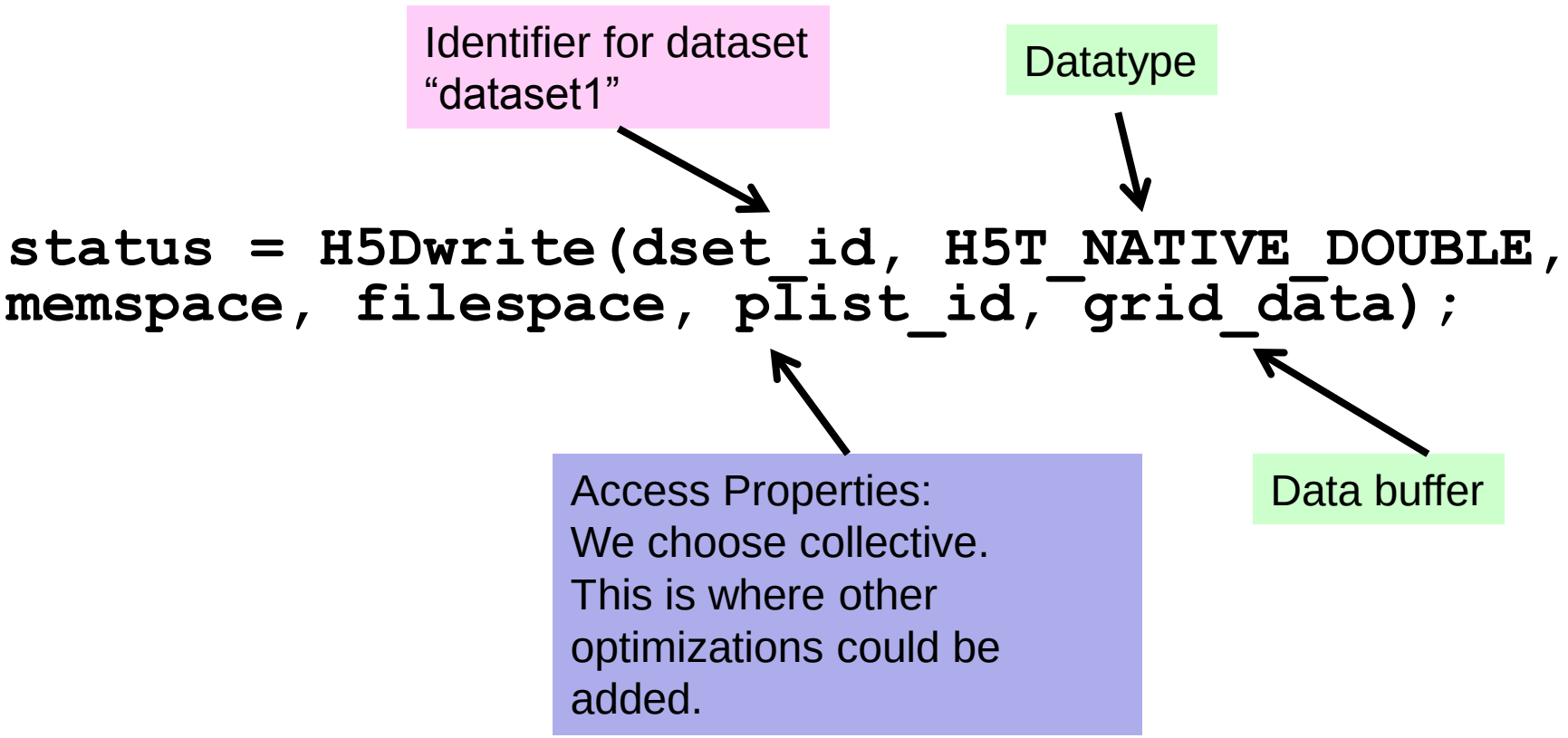
```
/* Create the local memory space */
memspace = H5Screate_simple(RANK, count, NULL);


filespace = H5Dget_space (dset_id);

/* Create the hyperslab -- says how you want to
lay out data */

status = H5Sselect_hyperslab(filespace,
H5S_SELECT_SET, offset, NULL, count, NULL);
```

# Write Data



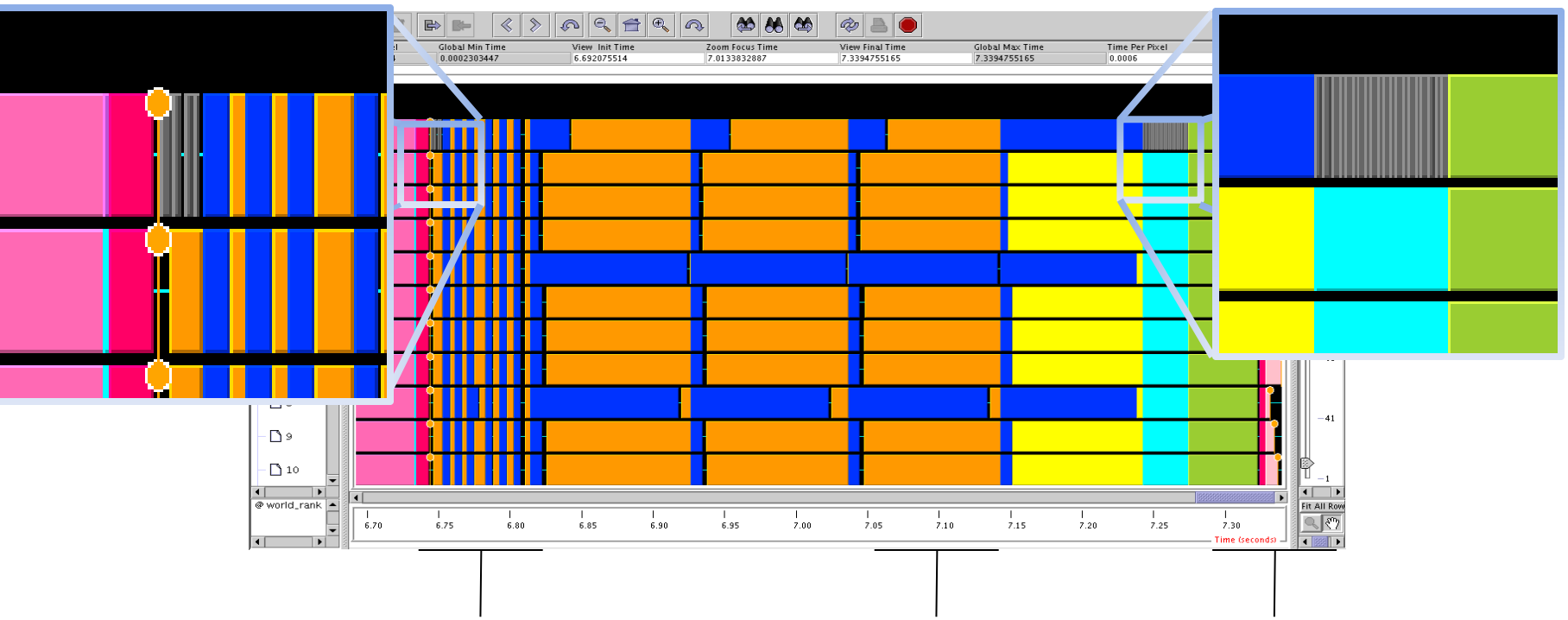

Then close every dataspace and file space that was opened

# Inside HDF5: Jumpshot view

1: Rank 0 writes initial structure (multiple independent I/O)

3: Determine location For variable (orange)

5: Rank 0 writes final md




2: Collectively write grid, provenance data

4: Collectively write variable (blue)

6: Close file

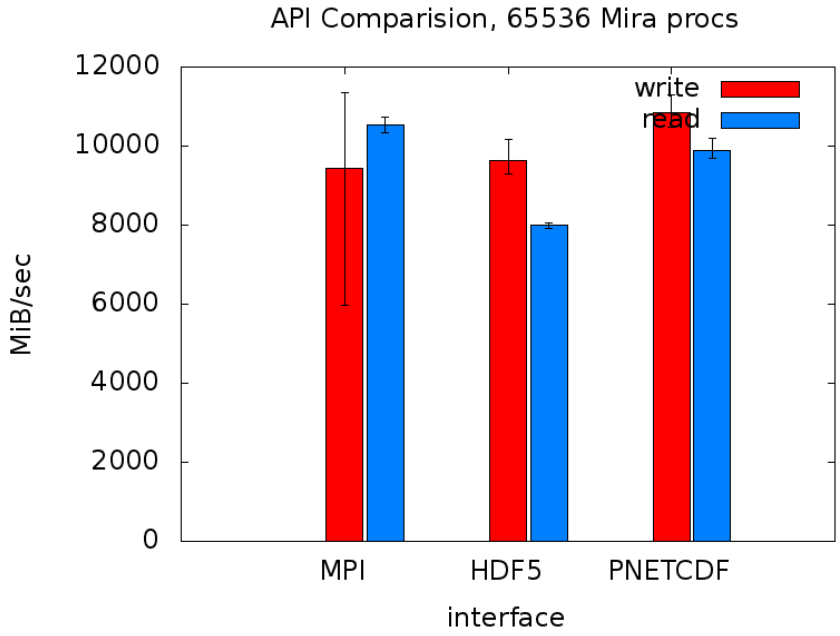
File open | Indep. write | Collective write | MPI_Allreduce | File close

# HDF5 Wrap-up

- Tremendous flexibility: 300+ routines
- H5Lite high level routines for common cases
- Tuning via property lists
  - “use MPI-IO to access this file”
  - “read this data collectively”
- Extensive on-line documentation, tutorials (see “On Line Resources” slide)
- New efforts:
  - Journaling: make datasets more robust in face of crashes (Sandia)
  - Fast appends (finance motivated)
  - Single-writer, Multiple-reader semantics
  - Aligning data structures to underlying file system
  - Multiple-dataset I/O: similar to Parallel-NetCDF operation-combining optimization

# Comparing I/O libraries

- IOR to evaluate HDF5, pnetcdf somewhat artificial
  - HLL typically hold structured data
- HDF5, pnetcdf demonstrate performance parity for these access sizes (6 MiB on Mira)
- I/O libraries deliver benefits with slight (if any) cost to performance



# Other High-Level I/O libraries

- NetCDF-4: http://www.unidata.ucar.edu/software/netcdf/netcdf-4/
  - netCDF API with HDF5 back-end
- ADIOS: http://adiosapi.org
  - Configurable (xml) I/O approaches
- SILO: https://wci.llnl.gov/codes/silo/
  - A mesh and field library on top of HDF5 (and others)
- H5part: http://vis.lbl.gov/Research/AcceleratorSAPP/
  - simplified HDF5 API for particle simulations
- GIO: https://svn.pnl.gov/gcrm
  - Targeting geodesic grids as part of GCRM
- PIO:
  - climate-oriented I/O library; supports raw binary, parallel-netcdf, or serial-netcdf (from master)
- … Many more: my point: likely one already exists for your domain

# Parallel I/O Wrap-up

- Assess the cost benefit of using shared file parallel-IO for the lifetime of your project
  - How much overhead can you afford?
  - Slower runtime, could save years of post-processing, visualization and analysis time later
- Use high level parallel I/O libraries over MPI-IO.
  - They don't cost you performance (sometimes improve it)
  - Gain: portability, longevity, programmability
- MPI-IO is the layer where most optimizations are implemented – tune these parameters carefully
- Watch out for the key parallel-I/O pitfalls – unaligned block sizes and small writes
  - MPI-IO layer can often solve these pitfalls on your behalf.

# Understanding I/O Behavior and Performance

Thanks to the following for much of this material:

**Kevin Harms, Charles Bacon, Sam Lang, Bill Allcock**
Math and Computer Science Division and Argonne Leadership Computing Facility
Argonne National Laboratory

**Yushu Yao and Katie Antypas**
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

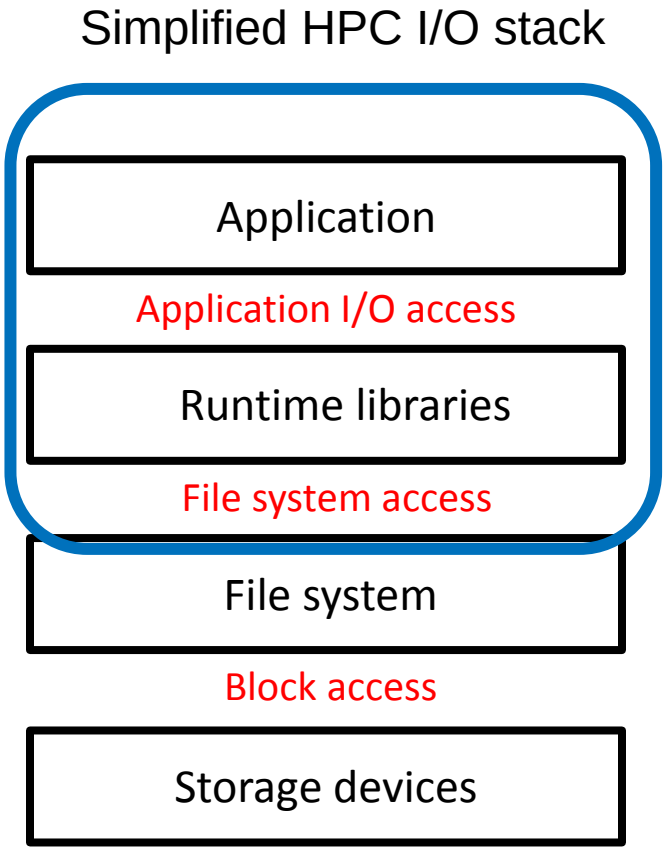
For more information, see:

- P. Carns et al. Understanding and improving computational science storage access through continuous characterization. ACM TOS. 2011.
- P. Carns et al. Production I/O characterization on the Cray XE6. CUG 2013. May, 2013.

# Characterizing Application I/O

**How are applications using the I/O system, and how successful are they at attaining high performance?**

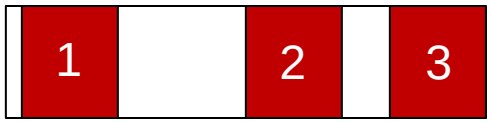
- The best way to answer these questions is by observing behavior at the application and library level
- What did the application intend to do, and how much time did it take to do it?
- In this portion of the training course we will focus on ***Darshan***, a scalable tool for characterizing application I/O activity.



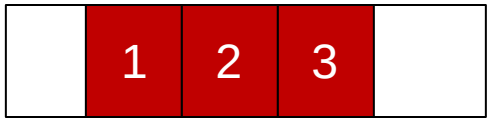
# What does Darshan do

**Darshan** (Sanskrit for “sight”) is a tool we developed for I/O characterization at extreme scale:

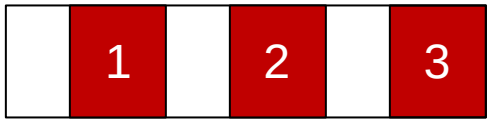
- No code changes, easy to enable
  - Enabled by default at ALCF and NERSC, optionally available at OLCF
- **Negligible performance impact**: just “leave it on”
- Produces a summary of I/O activity for each job
- Captures:
  - Counters for file I/O and MPI-IO calls, some PnetCDF and HDF5 calls
  - Counters for unaligned, sequential, consecutive, and strided access
  - Timing of opens, closes, first and last reads and writes
  - Cumulative data read and written
  - Histograms of access, stride, datatype, and extent sizes




sequential




consecutive




strided

# The technology behind Darshan

- Intercepts I/O functions using link-time wrappers
  - No code modification
  - Can be transparently enabled in MPI compiler scripts
  - Compatible with all major C, C++, and Fortran compilers
- Record statistics independently at each process, for each file
  - Bounded memory consumption
  - Compact summary rather than verbatim record
- Collect, compress, and store results at shutdown time
  - Aggregate shared file data using custom MPI reduction operator
  - Compress remaining data in parallel with zlib
  - Write results with collective MPI-IO
  - Result is a single gzip-compatible file containing characterization information
- Works for Linux clusters, Blue Gene, and Cray systems

# How to use Darshan

- Compile a C, C++, or FORTRAN program that uses MPI
- Run the application

- Look for the Darshan log file
- This will be in a particular directory (depending on your system's configuration)
  - <dir>/<year>/<month>/<day>/<username>_<appname>*.darshan.gz
  - Mira: see /projects/logs/darshan/
  - Edison: see /scratch1/scratchdirs/darshanlogs/
- Application must run to completion and call MPI_Finalize() to generate a log file

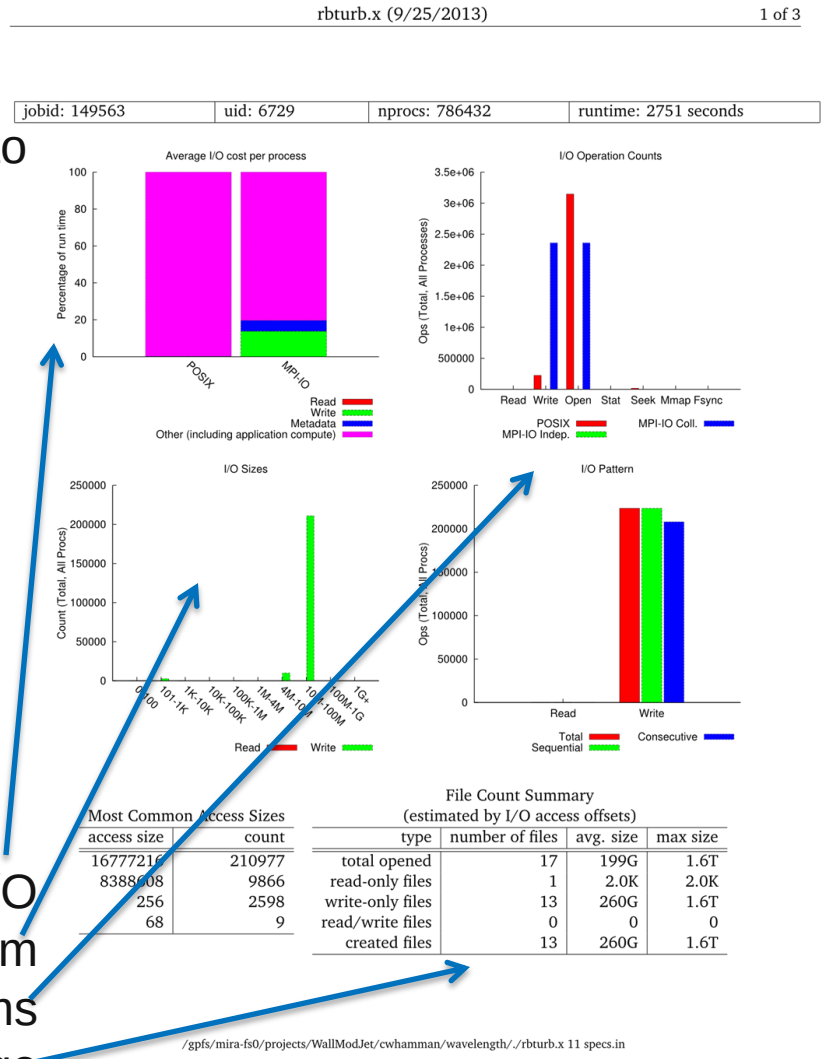
- Use Darshan command line tools to analyze the log file

- Warning/disclaimer: Darshan does not currently work for *F90* programs on Mira

# Darshan analysis example

- Each job instrumented with Darshan produces a single characterization log file
- Darshan command line utilities are used to analyze these log files
- Example: Darshan-job-summary.pl produces a 3-page PDF file summarizing various aspects of I/O performance

- This figure shows the I/O behavior of a 786,432 process turbulence simulation (production run) on the Mira system at ANL
- Application is write intensive and benefits greatly from collective buffering



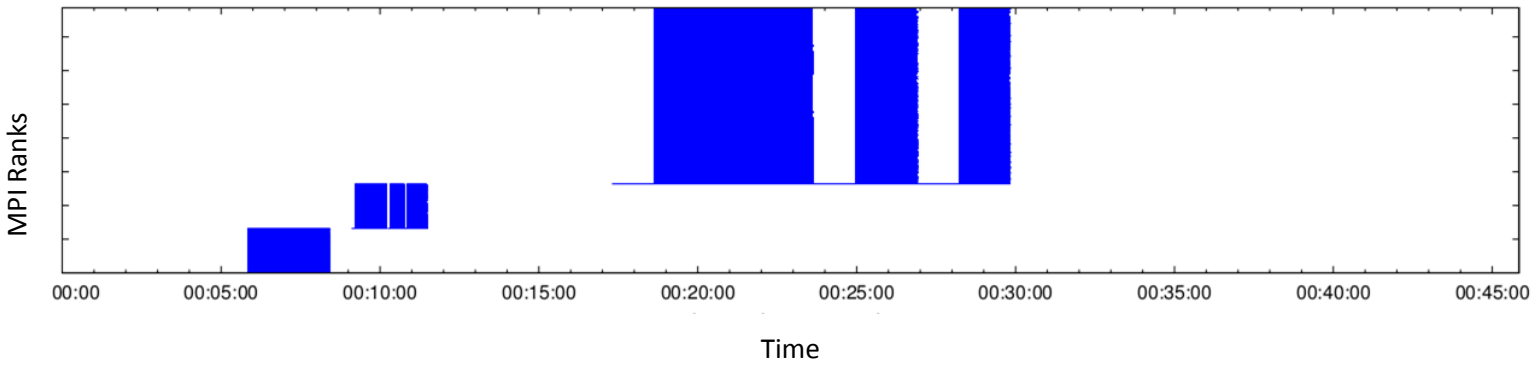

Percentage of runtime in I/O

Access size histogram

Access type histograms

File usage

# Darshan analysis example (page 2)




This graph (and others like it) are on the second page of the darshan-job-summary.pl output. This example shows intervals of I/O activity from each MPI process.

# Available Darshan analysis tools

- http://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html
- Key tools:
  - **Darshan-job-summary.pl**: creates pdf with graphs for initial analysis
  - **Darshan-summary-per-file.sh**: similar to above, but produces a separate pdf summary for every file opened by application
  - **Darshan-parser**: dumps all information into text format

Darshan-parser example (see all counters related to write operations):

```
darshan-parser user_app_numbers.darshan.gz |grep WRITE
```

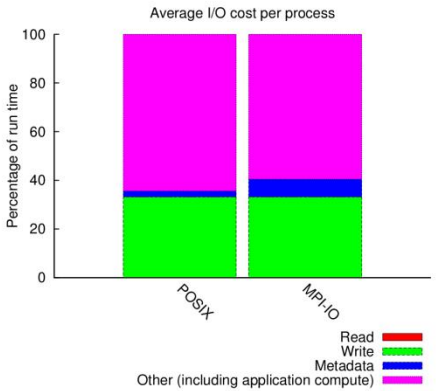
See documentation above for definition of output fields

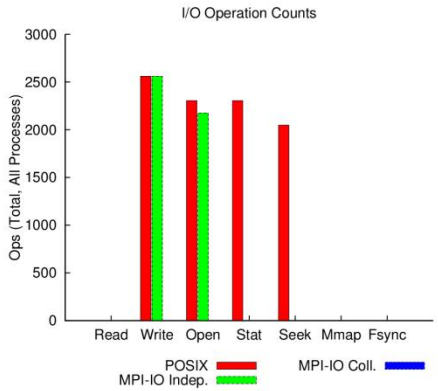
# Looking for I/O Performance Problems

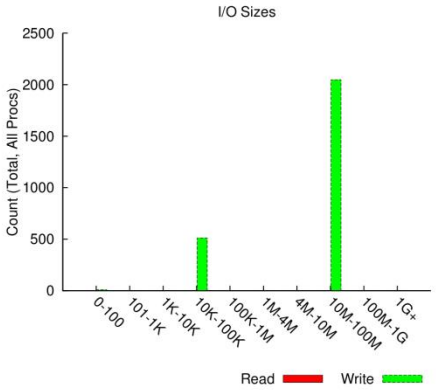
- Lightweight nature of Darshan means “always on”
- Many I/O problems can be seen from these logs
- We can study applications on-demand, or mine logs to catch problems pro-actively

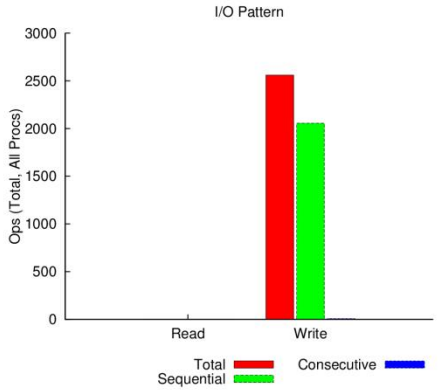
# Example: checking user expectations

jobid: | uid: | nprocs: 4096 | runtime: 175 seconds




Most Common Access Sizes

| access size | count |
|---|---|
| 67108864 | 2048 |
| 41120 | 512 |
| 8 | 4 |
| 4 | 3 |

File Count Summary

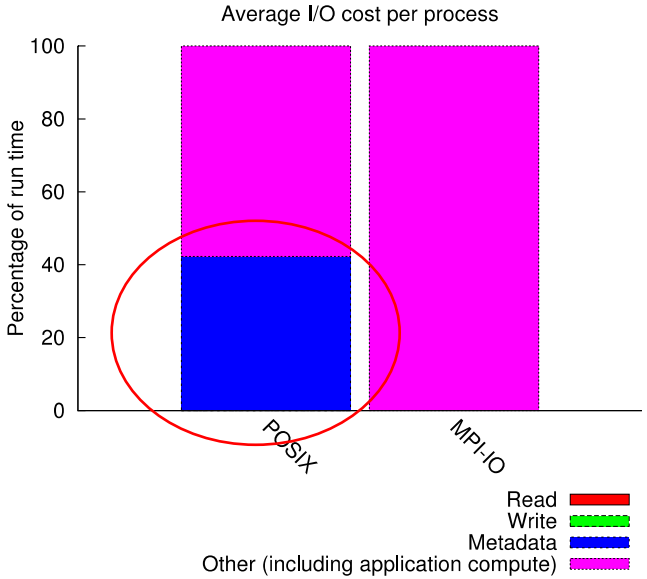
| type | number of files | avg. size | max size |
|---|---|---|---|
| total opened | 129 | 1017M | 1.1G |
| read-only files | 0 | 0 | 0 |
| write-only files | 129 | 1017M | 1.1G |
| read/write files | 0 | 0 | 0 |
| created files | 129 | 1017M | 1.1G |

- User opened 129 files (one “control” file, and 128 data files)
- Should be one header, about 40 KiB, per data file
- This example shows 512 headers being written
  - Code bug: header was written 4x per file

# Performance Debugging: Simulation Output

- HSCD combustion physics application
  - HSCD was writing 2-3 files per process with up to 32,768 cores
  - Darshan attributed 99% of the I/O time to metadata (on Intrepid BG/P)

| jobid: 0 | uid: 1817 | nprocs: 8192 | runtime: 863 seconds |
|---|---|---|---|



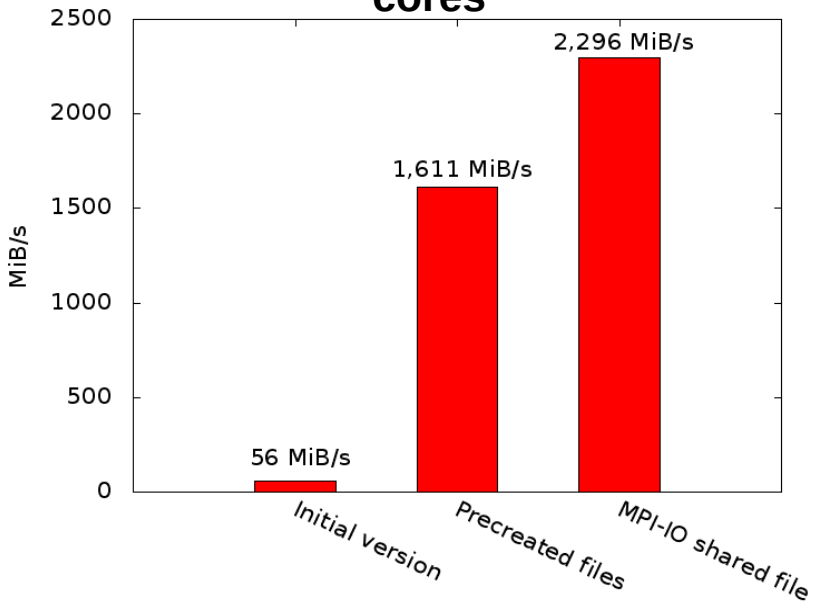

File Count Summary

| type | number of files | avg. size | max size |
|---|---|---|---|
| total opened | 16388 | 2.5M | 8.1M |
| read-only files | 0 | 0 | 0 |
| write-only files | 16388 | 2.5M | 8.1M |
| read/write files | 0 | 0 | 0 |
| created files | 16388 | 2.5M | 8.1M |

# Simulation Output (continued)

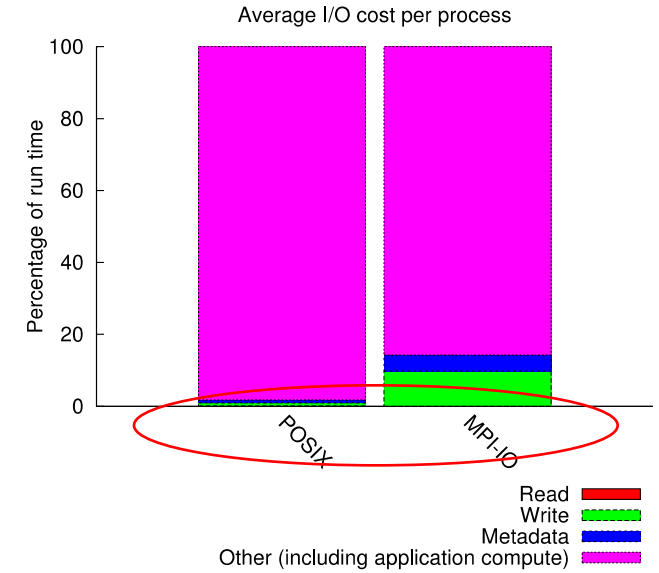
- With help from ALCF catalysts and Darshan instrumentation, we developed an I/O strategy that used MPI-IO collectives and a new file layout to reduce metadata overhead
- **Impact: 41X improvement in I/O throughput for production application**

**HSCD I/O performance with 32,768 cores**



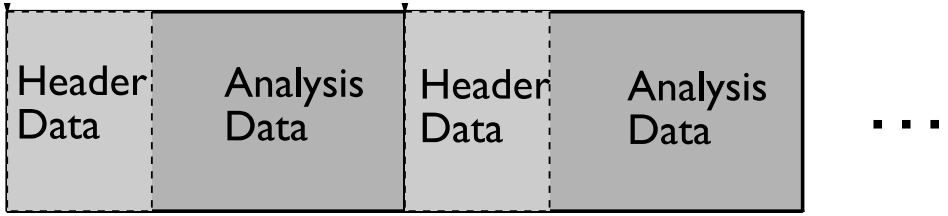

File Count Summary

| type | number of files | avg. size | max size |
|---|---|---|---|
| total opened | 8 | 515M | 2.0G |
| read-only files | 2 | 2.2K | 3.7K |
| write-only files | 6 | 686M | 2.0G |
| read/write files | 0 | 0 | 0 |
| created files | 6 | 686M | 2.0G |



# Performance Debugging: An Analysis I/O Example

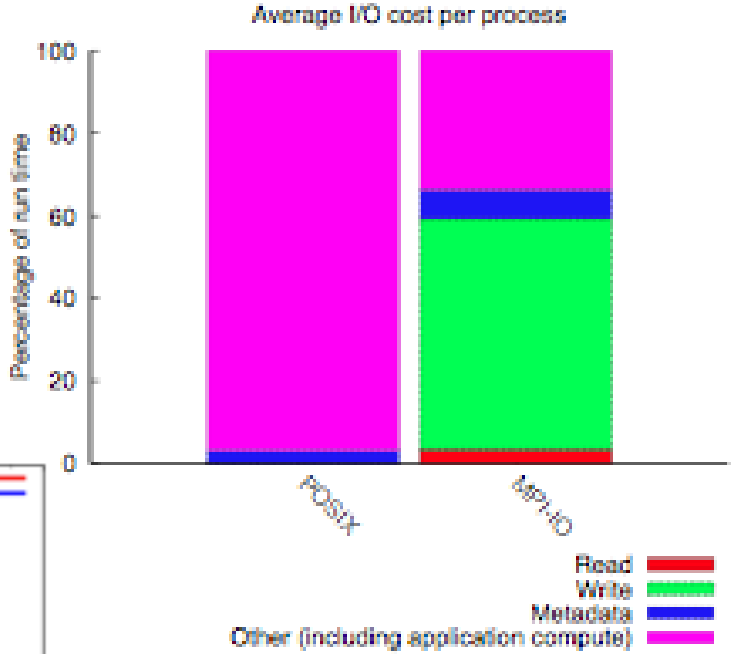
- Variable-size analysis data requires headers to contain size information
- Original idea: all processes collectively write headers, followed by all processes collectively write analysis data
- Use MPI-IO, collective I/O, all optimizations
- 4 GB output file (not very large)
- Why does the I/O take so long in this case?

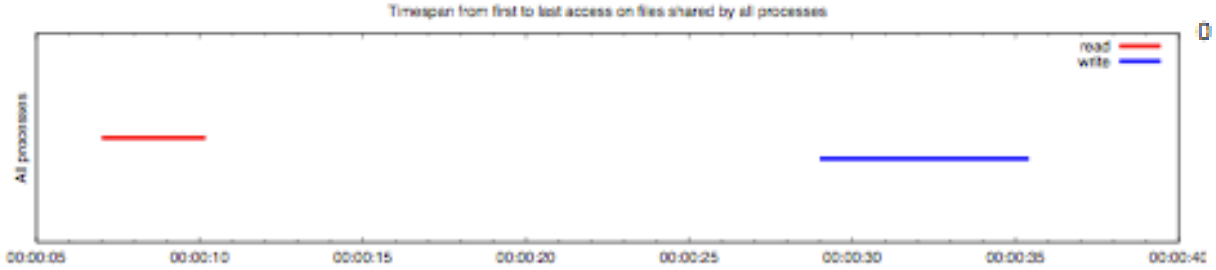



| Processes | I/O Time (s) | Total Time (s) |
|---|---|---|
| 8,192 | 8 | 60 |
| 16,384 | 16 | 47 |
| 32,768 | 32 | 57 |

# An Analysis I/O Example (continued)

- **Problem**: More than 50% of time spent writing output at 32K processes. Cause: Unexpected RMW pattern, difficult to see at the application code level, was identified from Darshan summaries.
- What we expected to see, read data followed by write analysis:

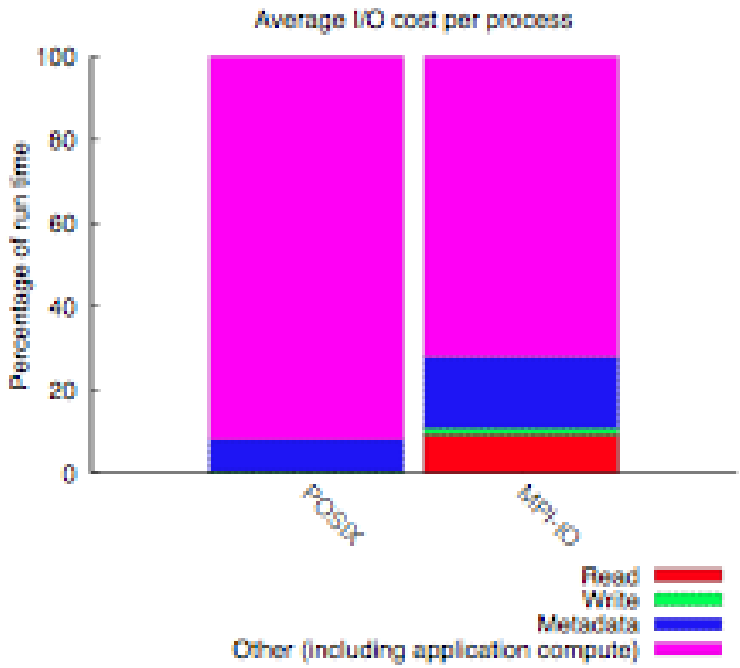



- What we saw instead: RMW during the writing shown by overlapping red (read) and blue (write), and a very long write as well.

# An Analysis I/O Example (continued)

- **Solution**: Reorder operations to combine writing block headers with block payloads, so that "holes" are not written into the file during the writing of block headers, to be filled when writing block payloads
- **Result**: Less than 25% of time spent writing output, output time 4X shorter, overall run time 1.7X shorter
- **Impact**: Enabled parallel Morse-Smale computation to scale to 32K processes on Rayleigh-Taylor instability data




| Processes | I/O Time (s) | Total Time (s) |
|---|---|---|
| 8,192 | 7 | 60 |
| 16,384 | 6 | 40 |
| 32,768 | 7 | 33 |

# Example: redundant read traffic

- Scenario: Applications that read more bytes of data from the file system than were present in the file
  - Even with caching effects, this type of job can cause disruptive I/O network traffic
  - Candidates for aggregation or collective I/O

- Example:
  - Scale: 6,138 processes
  - Run time: 6.5 hours
  - Avg. I/O time per proce 27 minutes

File Count Summary
(estimated by I/O access offsets)

| type | number of files | avg. size | max size |
|---|---|---|---|
| total opened | 1299 | 1.1G | 8.0G |
| read-only files | 1187 | 1.1G | 8.0G |
| write-only files | 112 | 418M | 2.6G |
| read/write files | 0 | 0 | 0 |
| created files | 112 | 418M | 2.6G |

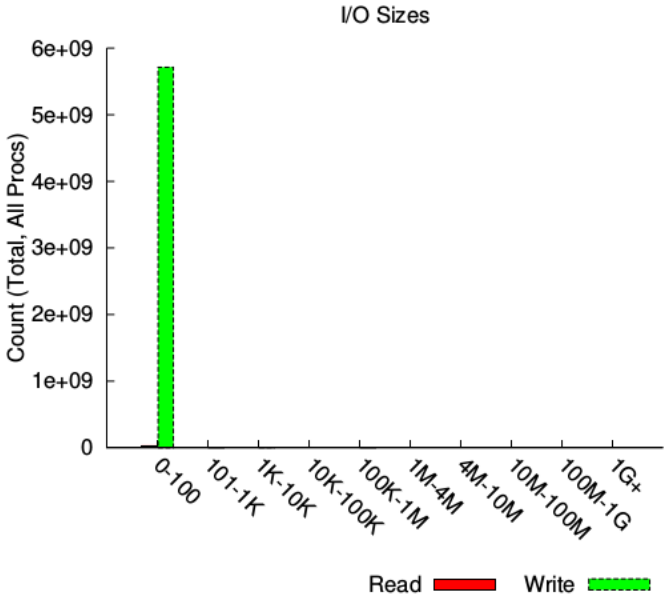
- 1.3 TiB of file data
- 500+ TiB read!

Data Transfer Per Filesystem

| File System | Write | | Read | |
|---|---|---|---|---|
| | MiB | Ratio | MiB | Ratio |
| / | 47161.47354 | 1.00000 | 575224145.24837 | 1.00000 |

# Example: small writes to shared files

- Scenario: Small writes can contribute to poor performance
  - Particularly when writing to shared files
  - Candidates for collective I/O or batching/buffering of write operatior



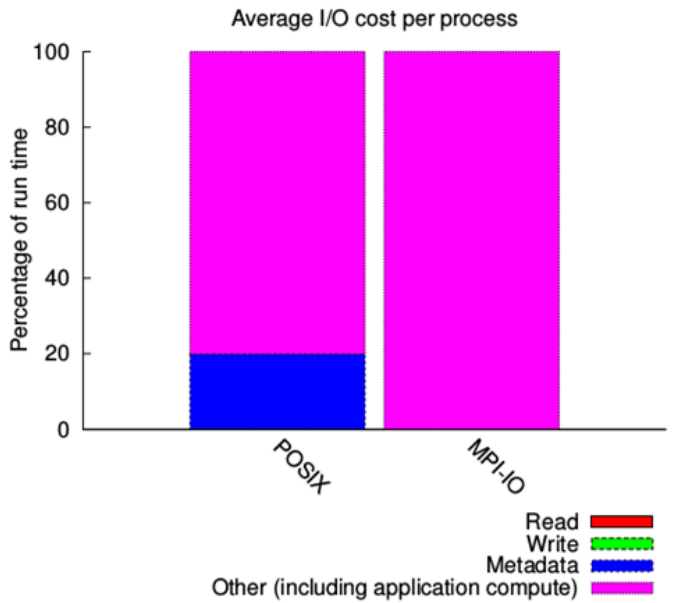

- Example:
  - Issued 5.7 billion writes to shared files, each less than 100 bytes in size
  - Averaged just over 1 MiB/s per process during shared write phase

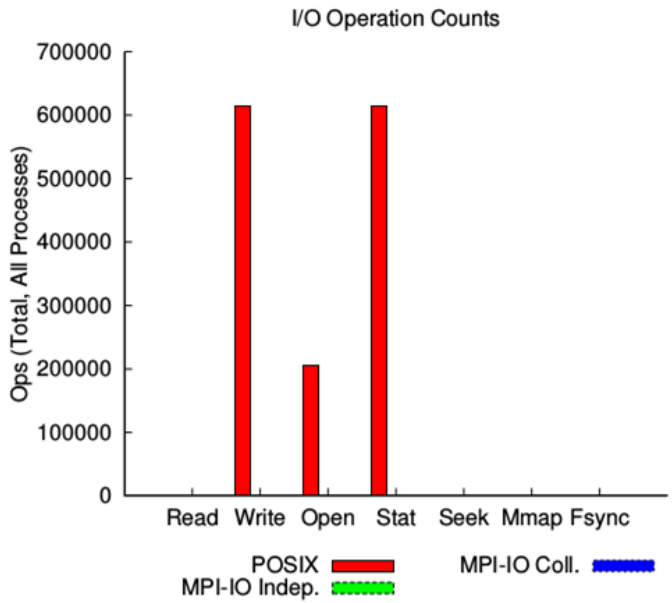
Most Common Access Sizes

| access size | count |
|---|---|
| 1 | 3418409696 |
| 15 | 2275400442 |
| 24 | 42289948 |
| 12 | 14725053 |

# Example: excessive metdata overhead

- Scenario: Very high percentage of I/O time spent performing metadata operations such as open(), close(), stat(), and seek()
  - Close() cost can be misleading due to write-behind cache flushing
  - Candidates for coalescing files and eliminating extra metadata calls
- Example:
  - Scale: 40,960 processes for 229 seconds, 103 seconds of I/O
  - 99% of I/O time in metadata operations
  - Generated 200,000+ files with 600,000+ write() and 600,000+ stat() calls



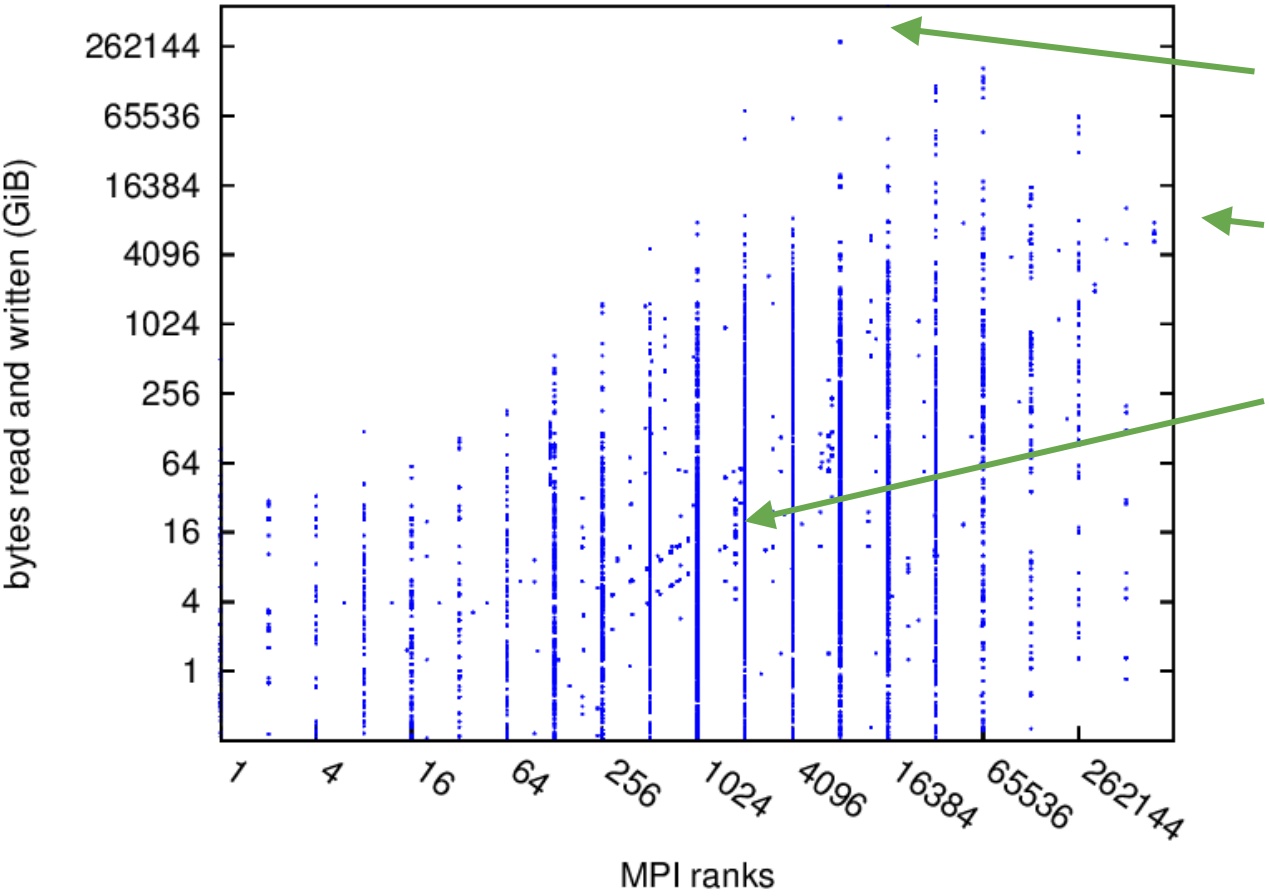
# Metadata side topic: what's so bad about stat()?

- stat() is actually quite cheap on most file systems
- But not a large-scale HPC I/O system!

- The usual problem is that stat() requires a consistent size calculation for the file
- To do this, a PFS has two options:
  - Store a precalculated size on the metadata server, which becomes a source of contention
  - Calculate size on demand, which might cause a storm of requests to *all* servers
- No present-day PFS deployments respond very well when thousands of processes stat() the same file at once

# Example: system-wide analysis

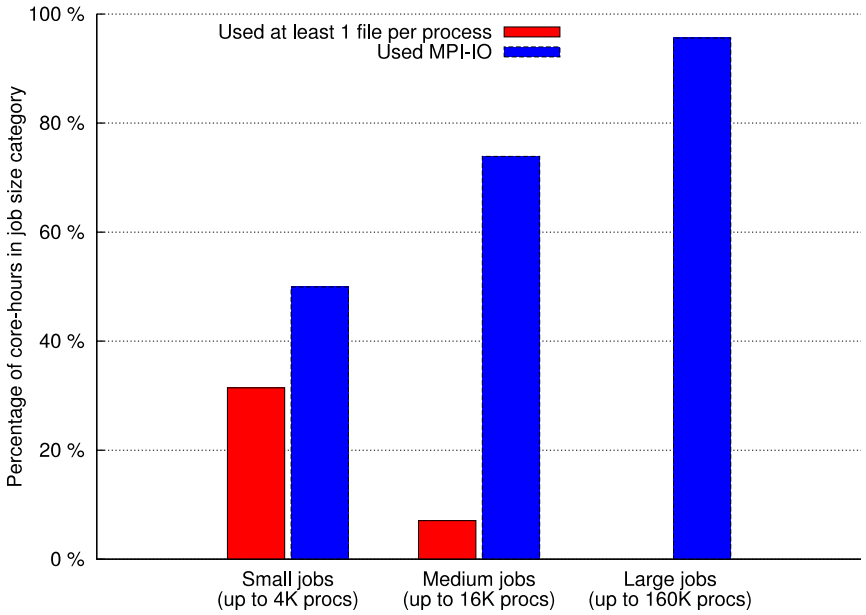
- Job size vs. data volume for Mira BG/Q system in 2014
  (~128,000 logs as of October, ~8 PiB of traffic)



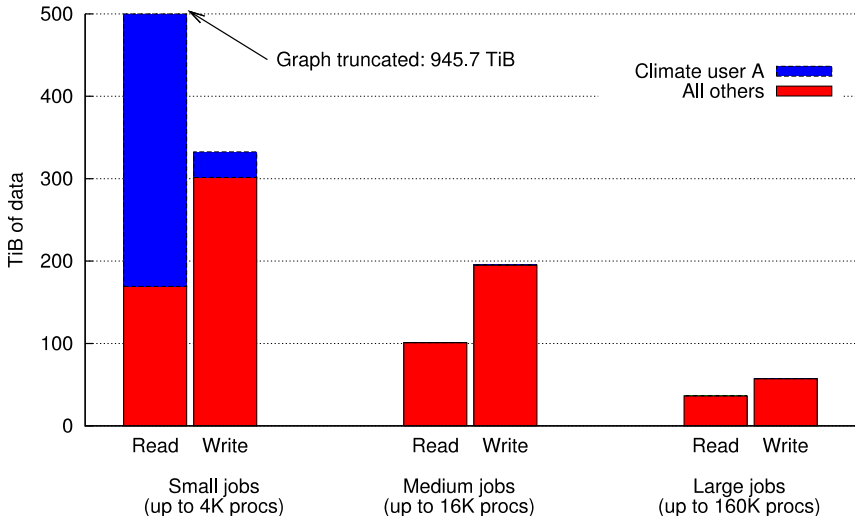

- Biggest by volume: ~300 TiB
- Biggest by scale: 768K processes
- Probably some scaling experiments?
- Most jobs use power of 2 numbers of processes on Mira

# Understanding system usage

- Aggregate data from Darshan can provide a broad view of system usage and application trends
- Examples from 4 months of data collected on Intrepid BG/P system in 2013:




MPI-IO usage becomes more prevalent at scale, while file-per-process access patterns become less prevalent.




This graph shows I/O volume at various scales. Write-intensive behavior is evident in each category. This data also illustrates how a single user on the system can dominate I/O activity in some cases.

# I/O Understanding Takeaway

- Scalable tools like Darshan can yield useful insight
  - Identify characteristics that make applications successful

    …and those that cause problems.
  - Identify problems to address through I/O research
- Petascale performance tools require special considerations
  - Target the problem domain carefully to minimize amount of data
  - Avoid shared resources
  - Use collectives where possible

- For more information, see:

  http://www.mcs.anl.gov/research/projects/darshan

# I/O Performance Tuning "Rules of thumb"

- Use collectives when possible
- Use high-level libraries (e.g. HDF5 or PnetCDF) when possible
- A few large I/O operations are better than many small I/O operations
- Avoid unnecessary metadata operations, especially *stat()*
- Avoid writing to shared files with POSIX
- Avoid leaving gaps/holes in files to be written later
- Use tools like Darshan to check assumptions about behavior

# Wrapping Up

- We've covered a lot of ground in a short time
  - Very low-level, serial interfaces
  - High-level, hierarchical file formats
- Storage is a complex hardware/software system
- There is no magic in high performance I/O
  - Lots of software is available to support computational science workloads at scale
  - Knowing how things work will lead you to better performance
- Using this software (correctly) can dramatically improve performance (execution time) and productivity (development time)

# Printed References

- John May, Parallel I/O for High Performance Computing, Morgan Kaufmann, October 9, 2000.
  - Good coverage of basic concepts, some MPI-IO, HDF5, and serial netCDF
  - Out of print?
- Quincey Koziol, Prabhat, editors, High Performance Parallel I/O, Chapman and Hall/ CRC, October 2014
  - Survey of nearly every tool, library, file system available
- William Gropp, Ewing Lusk, and Rajeev Thakur, Using MPI-2: Advanced Features of the Message Passing Interface, MIT Press, November 26, 1999.
  - In-depth coverage of MPI-IO API, including a very detailed description of the MPI-IO consistency semantics

# On-Line References (1 of 4)

- netCDF and netCDF-4
  - http://www.unidata.ucar.edu/packages/netcdf/
- PnetCDF
  - http://www.mcs.anl.gov/parallel-netcdf/
- ROMIO MPI-IO
  - http://www.mcs.anl.gov/romio/
- HDF5 and HDF5 Tutorial
  - http://www.hdfgroup.org/
  - http://www.hdfgroup.org/HDF5/
  - http://www.hdfgroup.org/HDF5/Tutor
- POSIX I/O Extensions
  - http://www.opengroup.org/platform/hecewg/
- Darshan I/O Characterization Tool
  - http://www.mcs.anl.gov/research/projects/darshan

# On-Line References (2 of 4)

- PVFS

  http://www.pvfs.org
- Panasas

  http://www.panasas.com
- Lustre

  http://www.lustre.org
- GPFS

  http://www.almaden.ibm.com/storagesystems/file_systems/GPFS/

# On-Line References (3 of 4)

- IOR benchmark
  - https://github.com/chaos/ior
- Parallel I/O Benchmarking Consortium (noncontig, mpi-tile-io, mpi-md-test)
  - http://www.mcs.anl.gov/pio-benchmark/
- FLASH I/O benchmark
  - http://www.mcs.anl.gov/pio-benchmark/
  - http://flash.uchicago.edu/~jbgallag/io_bench/ (original version)
- b_eff_io test
  - http://www.hlrs.de/organization/par/services/models/mpi/b_eff_io/
- mpiBLAST
  - http://www.mpiblast.org

# On Line References (4 of 4)

- NFS Version 4.1
  - 5661: NFSv4.1 protocol
  - 5662:  NFSv4.1 XDR Representation
  - 5663: pNFS Block/Volume Layout
  - 5664: pNFS Objects Operation
- pNFSProblem Statement
  - Garth Gibson (Panasas), Peter Corbett (Netapp), Internet-draft, July 2004
  - http://www.pdl.cmu.edu/pNFS/archive/gibson-pnfs-problem-statement.html
- Linux pNFS Kernel Development
  - http://www.citi.umich.edu/projects/asci/pnfs/linux

# Acknowledgements

This work is supported in part by U.S. Department of Energy Grant DE-FC02-01ER25506, by National Science Foundation Grants EIA-9986052, CCR-0204429, and CCR-0311542, and by the U.S. Department of Energy under Contract DE-AC02-06CH11357.

Thanks to Rajeev Thakur (ANL), Bill Loewe (Panasas), and Marc Unangst (Google) for their help in creating this material and presenting this tutorial in prior years.