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
Panasas, Inc.
welch@panasas.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.”

– Alok Choudhary, IESP, Kobe Japan, April 2012

(I know for sure he was not kidding.)

About Us

- Rob Latham (robl@mcs.anl.gov)
  - Senior 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@panasas.com)
  - Chief Technology Officer, Panasas
  - Berkeley Sprite OS Distributed Filesystem
  - Panasas ActiveScale Filesystem
  - IETF pNFS
- Katie Antypas (kantypas@lbl.gov)
  - Group Lead for User Services, NERSC
  - Guides application groups towards efficient use of NERSC's Lustre and GPFS file systems.
  - Collaborates with HDF5 Group and Cray's MPI-IO developers to improve application I/O performance.

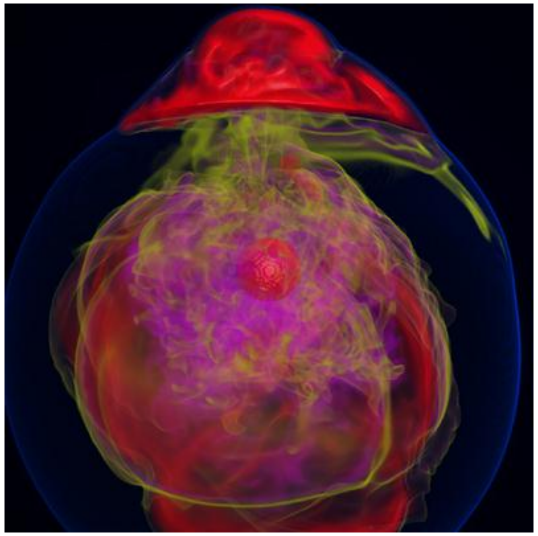
Outline

- Introduction
- Storage hardware
- Flash and future technology
- RAID's role in storage
- File system overview
- Parallel file system technology
- Lunch discussion
- Workloads
- Benchmarking
- POSIX
- MPI-IO
- IO Forwarding
- 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

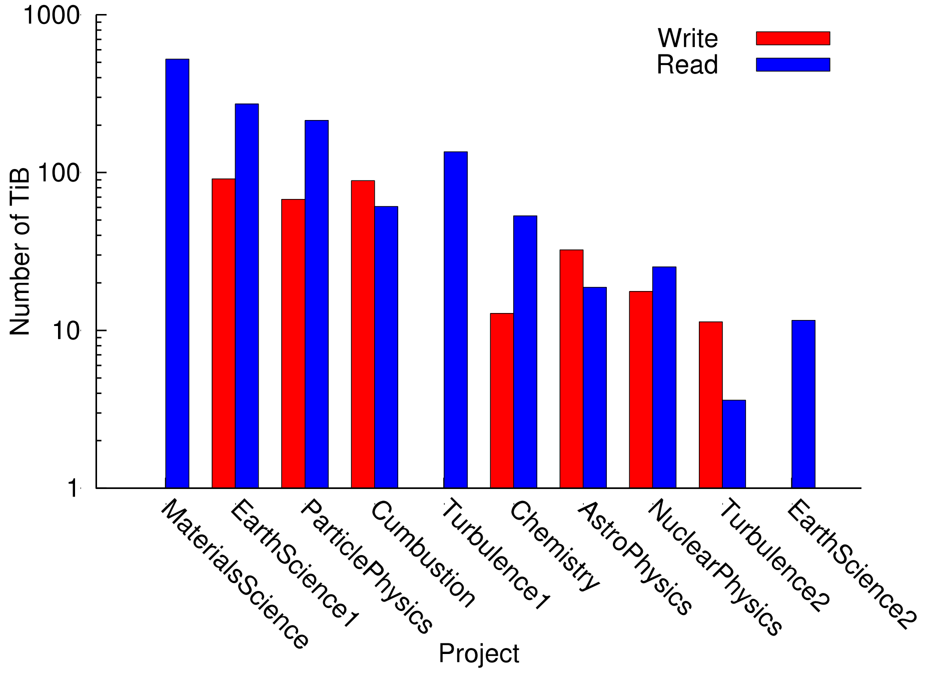
IBM Blue Gene/P system at Argonne National Laboratory.


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 2012 INCITE applications at ALCF (BG/P)

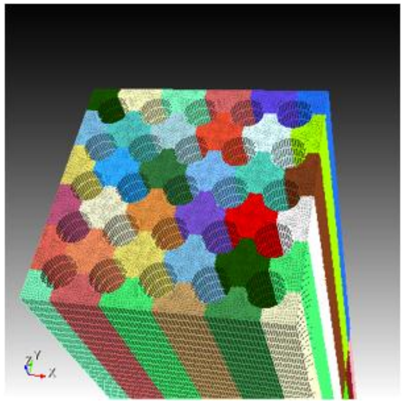
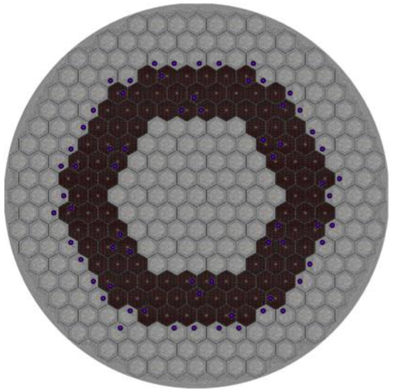
PI	Project	On-line Data (TBytes)	Off-line Data (TBytes)
Lamb	Supernovae Astrophysics	100	400
Khokhlov	Combustion in Reactive Gases	1	17
Lester	CO2 Absorption	5	15
Jordan	Seismic Hazard Analysis	600	100
Washington	Climate Science	200	750
Voth	Energy Storage Materials	10	10
Vashista	Stress Corrosion Cracking	12	72
Vary	Nuclear Structure and Reactions	6	30
Fischer	Reactor Thermal Hydraulic Modeling	100	100
Hinkel	Laser-Plasma Interactions	60	60
Elghobashi	Vaporizing Droplets in a Turbulent Flow	2	4



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.

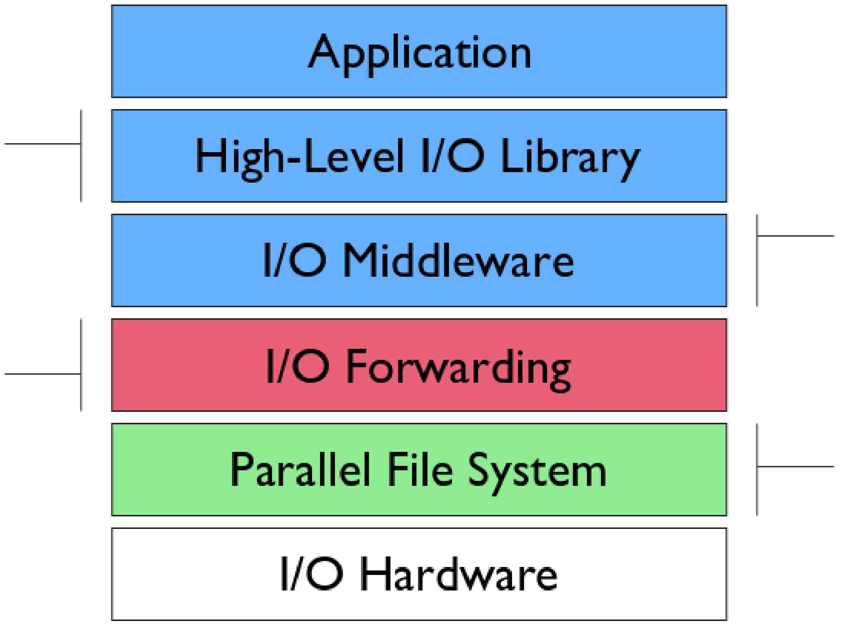
Images from T. Tautges (ANL) (upper left), M. Smith (ANL) (lower left), and K. Smith (MIT) (right).

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.

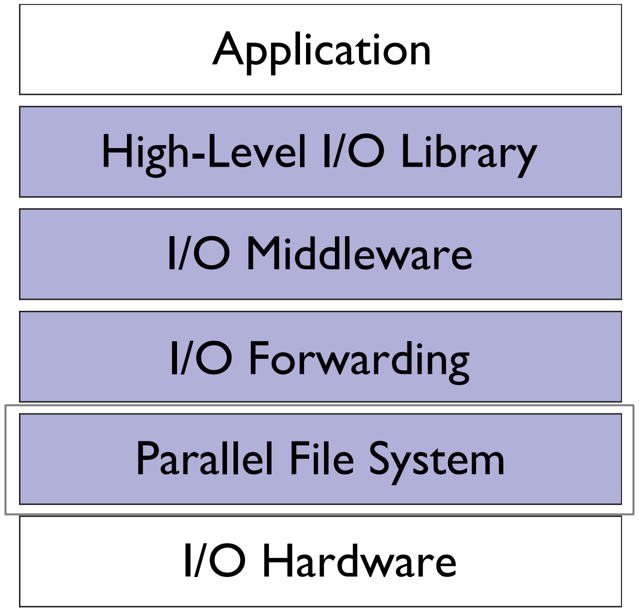
MPI-IO

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.

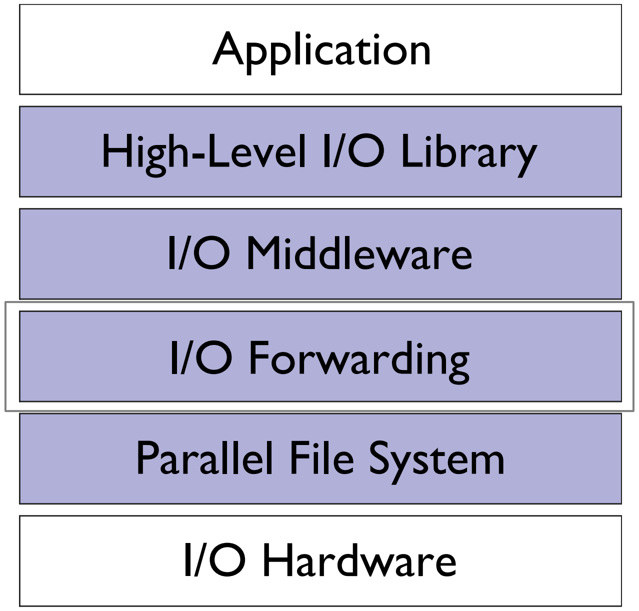
Parallel File System



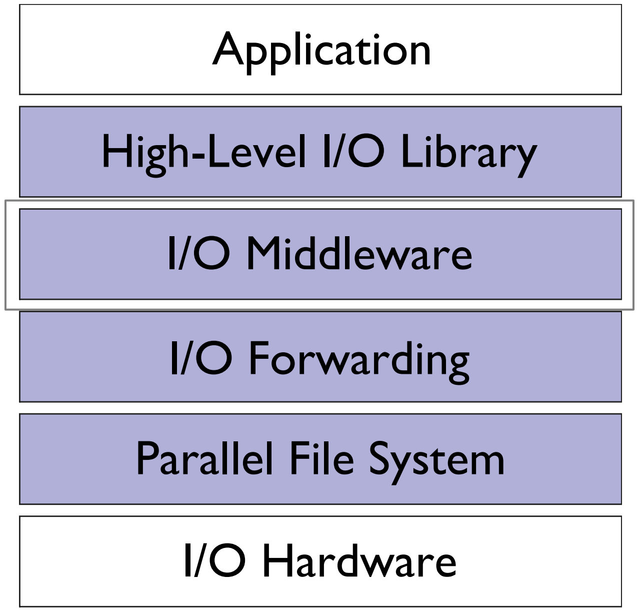
- Manage storage hardware
  - Present single view
  - Stripe files for performance
- In the I/O software stack
  - Focus on concurrent, independent access
  - Publish an interface that middleware can use effectively
    - Rich I/O language
    - Relaxed but sufficient semantics

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

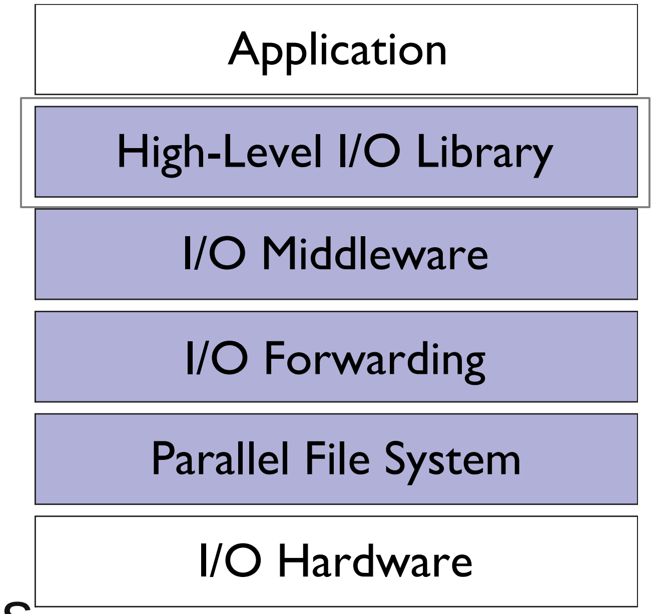


I/O Middleware



- Match the programming model (e.g. MPI)
- Facilitate concurrent access by groups of 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

Storage Hardware

- Bits: painted on metal oxide, or semi-conductor
- Speed hierarchy (CPU, DRAM, Networking, SSD, Disk)
- Storage devices: mechanical (disk, tape) and electronic (SSD)
- For performance, many devices in parallel
- Failures: bit rot, device failure.
- More devices, more failures
- For reliability, add hardware redundancy and lots of software
- Software has bugs, so recovery techniques are necessary

Storage Bits

- Magnetic bits on disk and tape
  - Stable w/out power
  - Encoding techniques use N+M bits to store N user bits
  - Relatively cheap to manufacture
  - Small amounts of electronics for large numbers of bits
- Semi-conductor bits
  - DRAM, constant power draw
  - FLASH, stable w/out power – with caveats
    - Write/Erase cycles wear down the device
    - 10 year storage when new
    - 1 year storage when old
  - At least 10x the cost to manufacture electronic bits compared to magnetic bits

Why can't we junk all the disks

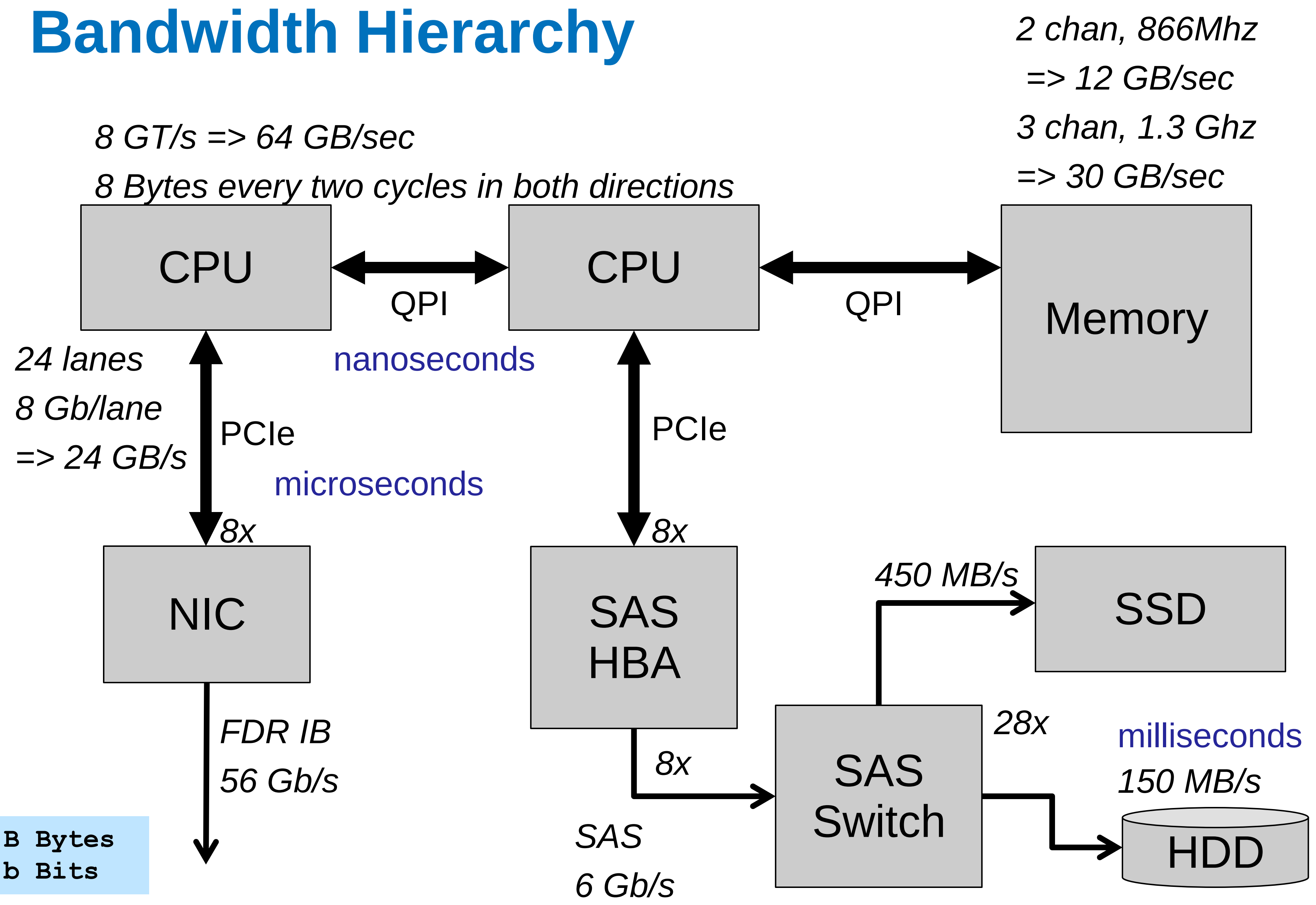
- Storage Hierarchy is DRAM, FLASH, Disk, Tape
- Cannot manufacture enough bits via Wafers vs. Disks
  - SSD 10x per-bit cost, and the gap isn't closing
  - Cost of semiconductor FAB is >> cost of disk manufacturing facility
  - World-wide manufacturing capacity of semi-conductor bits is perhaps 1% the 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)

Bandwidth Hierarchy



Networking Speeds and Feeds

Network	Encoding	Physical	Raw	Effective
FDR IB	66/64	4x 14 Gb/s	56 Gb/s	6+ GB/s
QDR IB	8/10	4x 10 Gb/s	40 Gb/s	4 GB/s
40 GE	10/12.5	4x 10 Gb/s	40 Gb/s	5 GB/s
10 GE	10/12.5	1x 10 Gb/s	10 Gb/s	1.25 GB/s

100 Gb/s (4x 25 Gb/s) projected for 2015

Network adaptor cards and their PCIe interface also limits throughput and affects latency for small packets

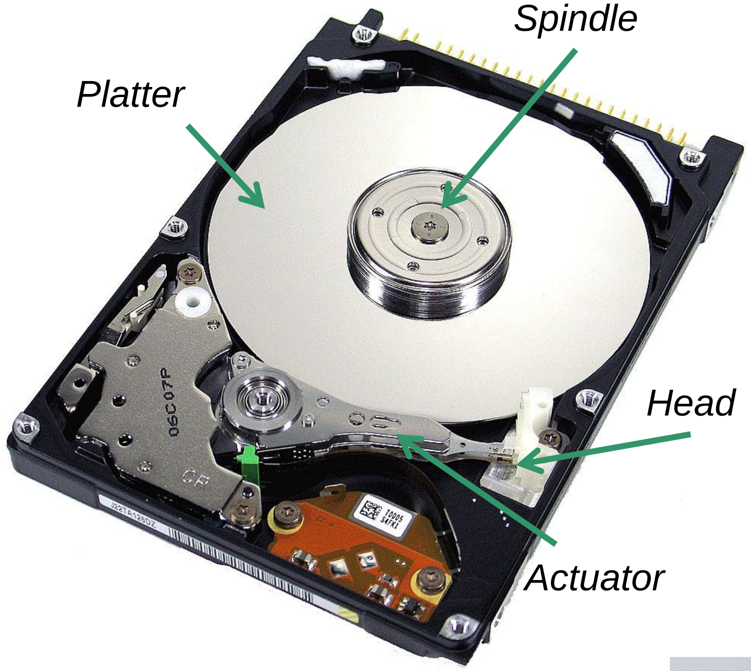
PCIe3, 8 Gb/s per channel and 66/64 encoding
PCIe2, 5 Gb/s per channel and 8/10 encoding

8x PCIe2 is ample for dual 10GE
16x PCIe3 is a match for dual FDR IB

Bandwidth

- CPU sockets have lots of it
  - To memory
  - To PCIe lanes
- High speed networks have a decent amount
  - Affected by protocol (CPU) overhead
- Storage devices are lagging behind
  - Especially hard drives
  - SSD write performance isn’t super great, either

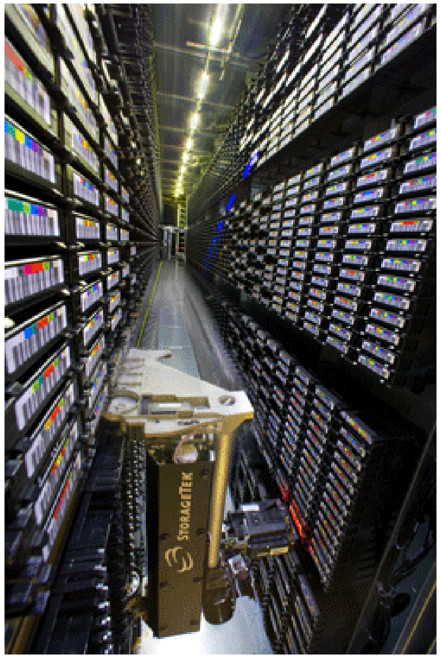
Storage Devices



Magnetic Hard Disk Drives

LTO 6
2.5 TB, 160 MB/sec

From Computer Desktop Encyclopedia
© 1999 The Computer Language Co. Inc.
4mm DAT
Large (330GB)
Medium (150GB)
Small (50GB)



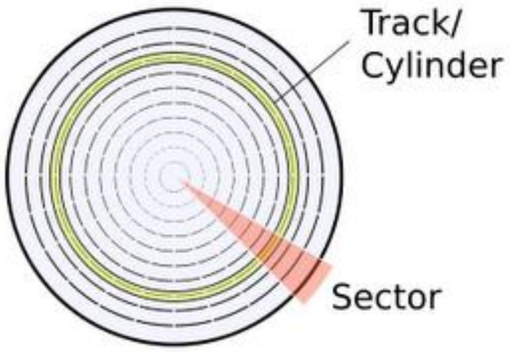
Magnetic Tape

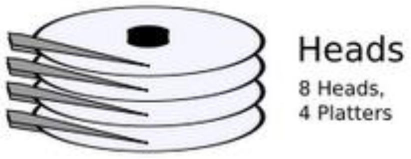
Solid State Storage Devices (flash)

Drive Characteristics

- Capacity (in MB, GB, TB)
  - Function of areal density
  - Areal density = track density * linear density
  - Sector is 512 bytes, moving to 4K
- Transfer Rate (bandwidth) – MB/sec
  - Rate at which a device reads or writes data
  - 1-250 MB/sec depending on seeks
- Access Time (milli-seconds)
  - Delay before the first byte is read
  - Seek time plus (avg) rotational delay
  - 8.33 msec for full rotation at 7200 RPM
  - 1 msec track-to-track seek (or less)
  - 20-30 msec "full stroke" seek (or more)

More sectors per track on outer cylinders





Only one head active at a time, either reading or writing

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

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

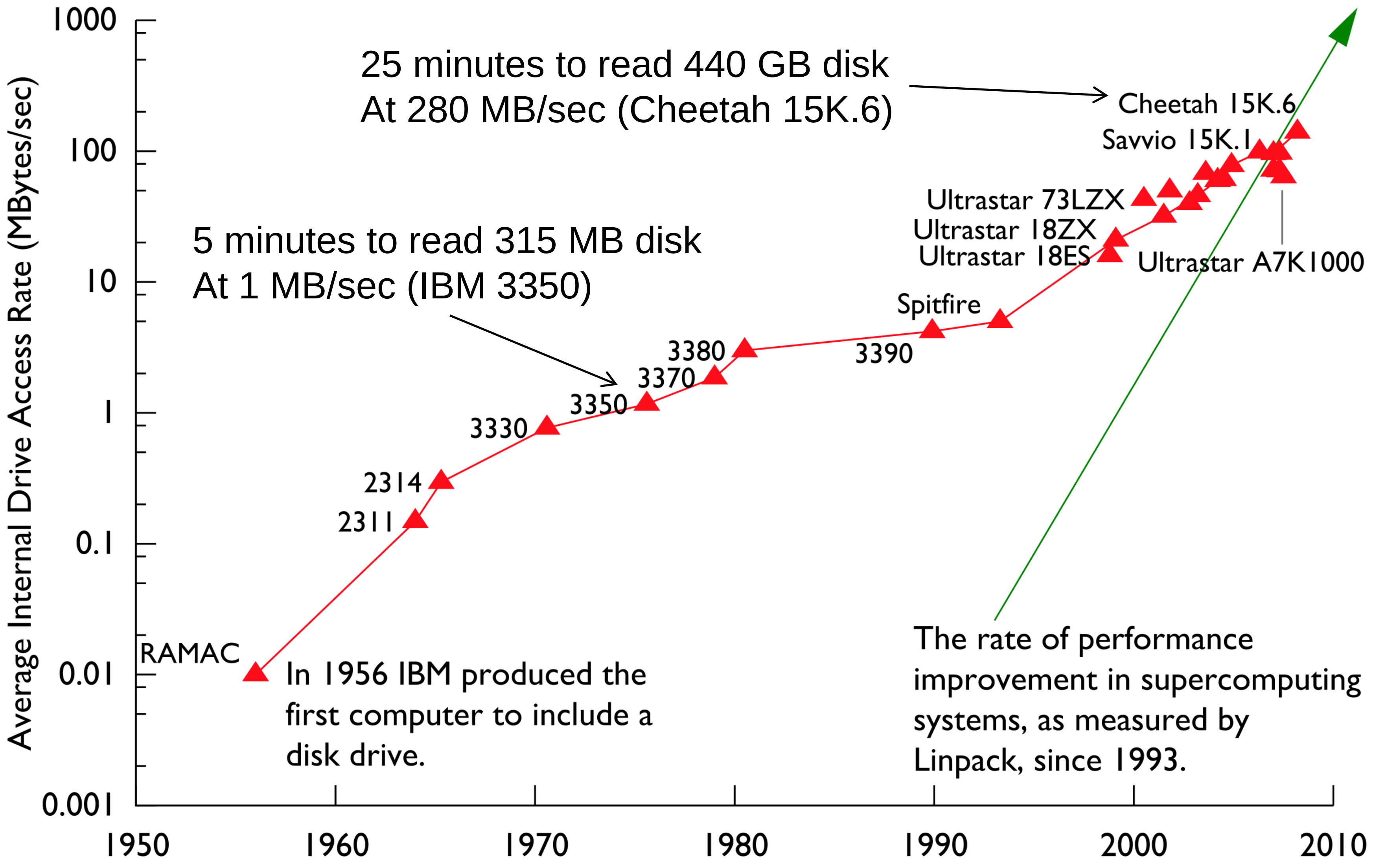
Computer scientists often think in base-2 units (Mebibyte)
- Even though they use base-10 unit terms

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**
- 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%)
- Takes longer and longer to completely read each new generation of drive

Disk Transfer Rates over Time

11 hours to read 4 TB SATA
At 50 MB/sec



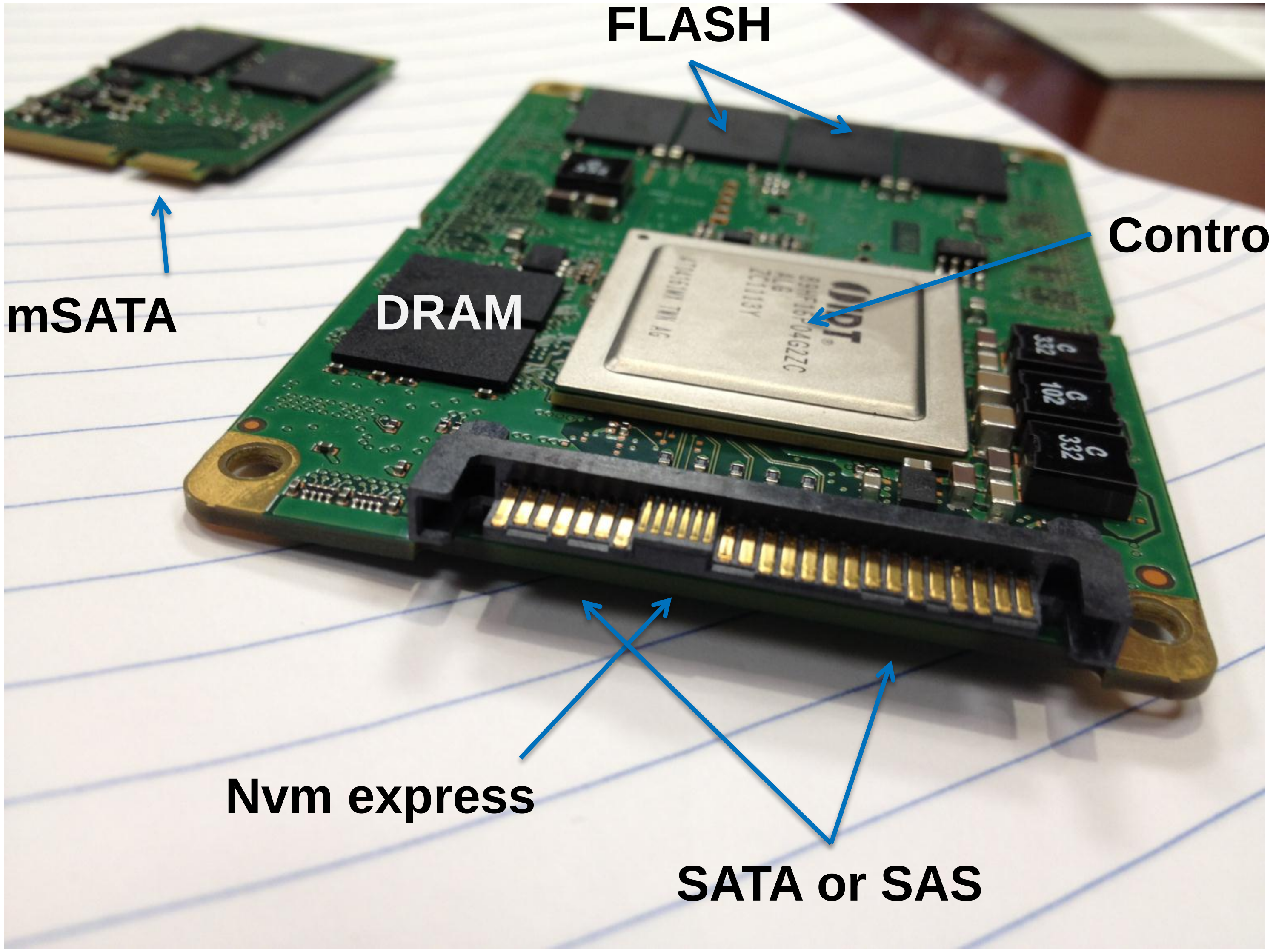
Thanks to R. Freitas of IBM Almaden Research Center for providing much of the data for this graph.

FLASH and SSD

SSD

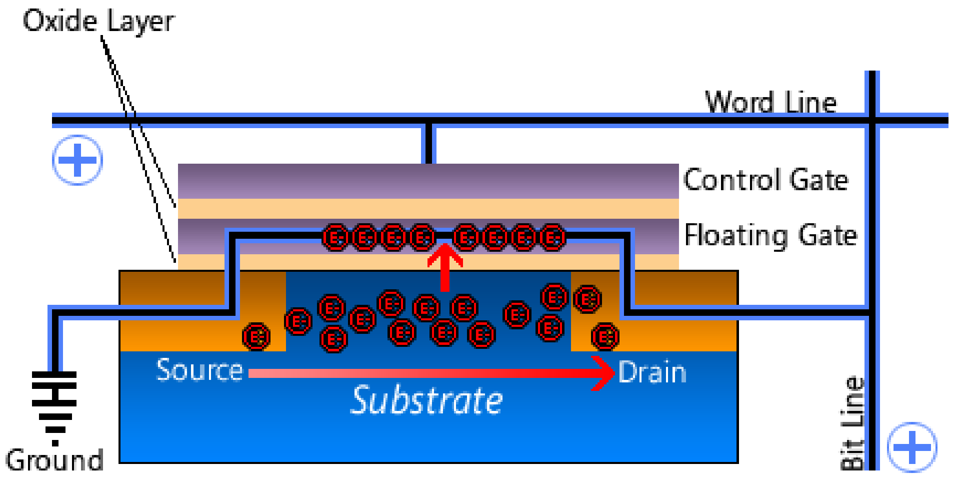
- Interface
  - SATA, SAS, PCIe, NVMexpress
  - Non-disk, PCIe interfaces for low overhead
- Controller
  - Wear leveling, garbage collection, data integrity
- DRAM
  - Fast copy of Flash Translation Layer
  - Write buffer (optional)
- FLASH
  - Many packages to increase concurrency

SSD Components



FLASH Characteristics

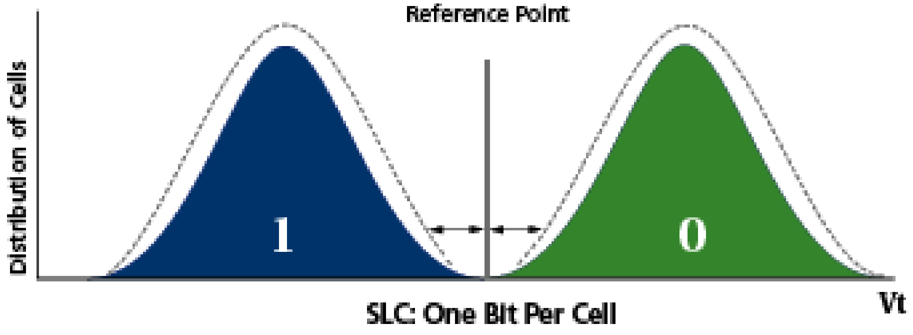
- Non-volatile
  - Each bit is stored in a “floating gate” that holds value without power
  - Electrons can leak, so shelf life and write count is limited
- Page-oriented
  - Smaller (e.g., 8K) read/write block based on addressing logic
  - Larger (e.g., 1MB) erase block to amortize the time it takes 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)

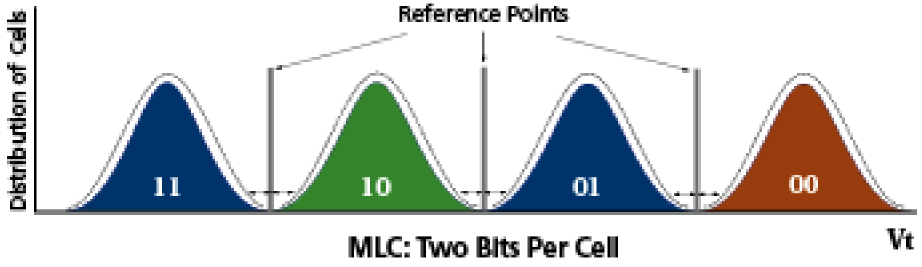


http://icrontic.com/articles/how_ssds_work

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!

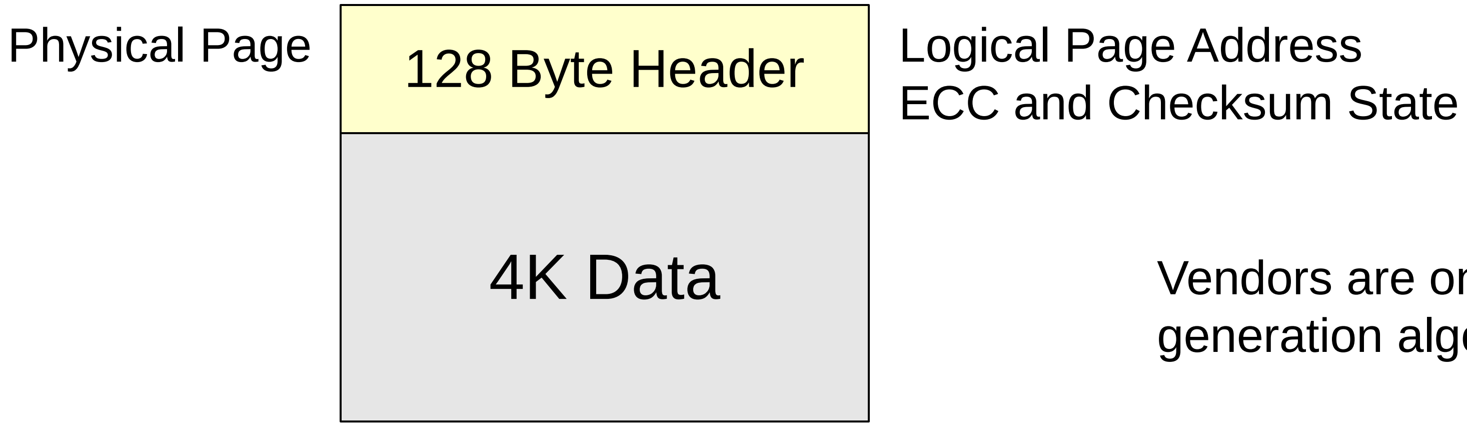




http://www.micron.com/nandcom/

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



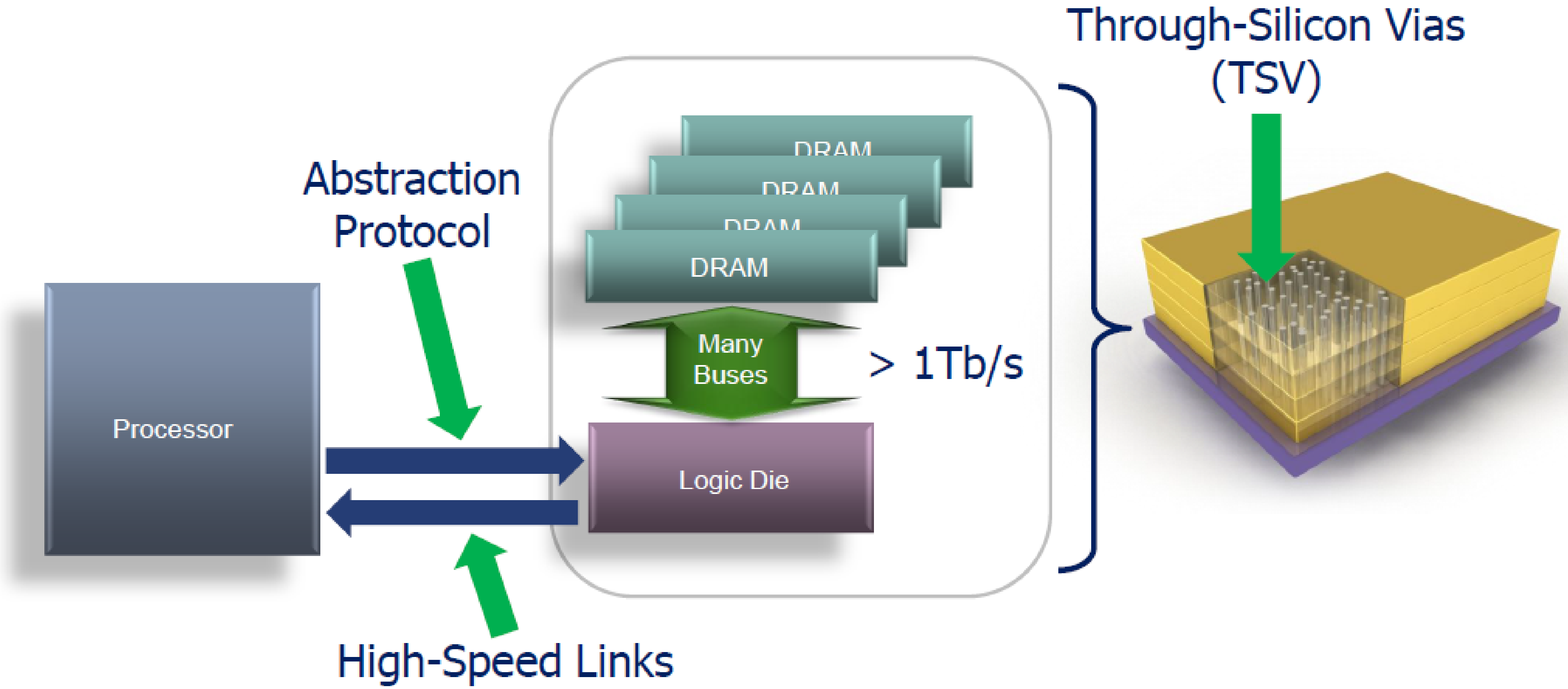
Vendors are on 2nd (or 3rd) generation algorithms

FLASH Trends

- Serial interface speed getting faster
- Write speeds getting slower
- Page size increases from 4K to 8K or 16K
- Erase block increases from 256K to 1M
- Multiple channels per package allow more concurrent operations
- High speed devices use many packages and stripe data to get high bandwidth
- Power failure protection for volatile DRAM inside the device

Future Technologies

Hybrid Memory Cube (HMC)

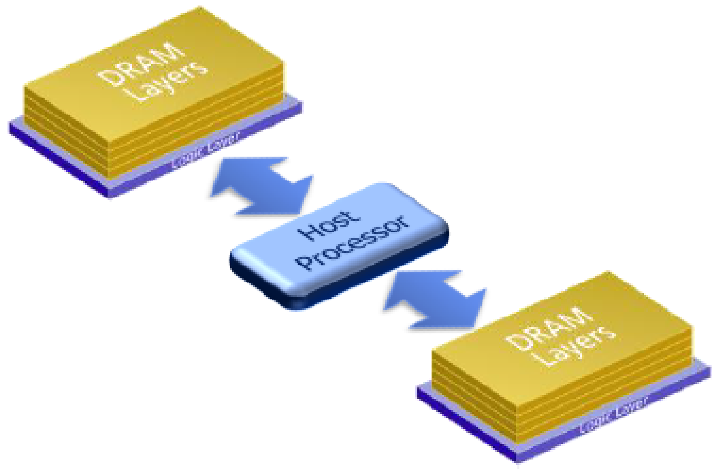


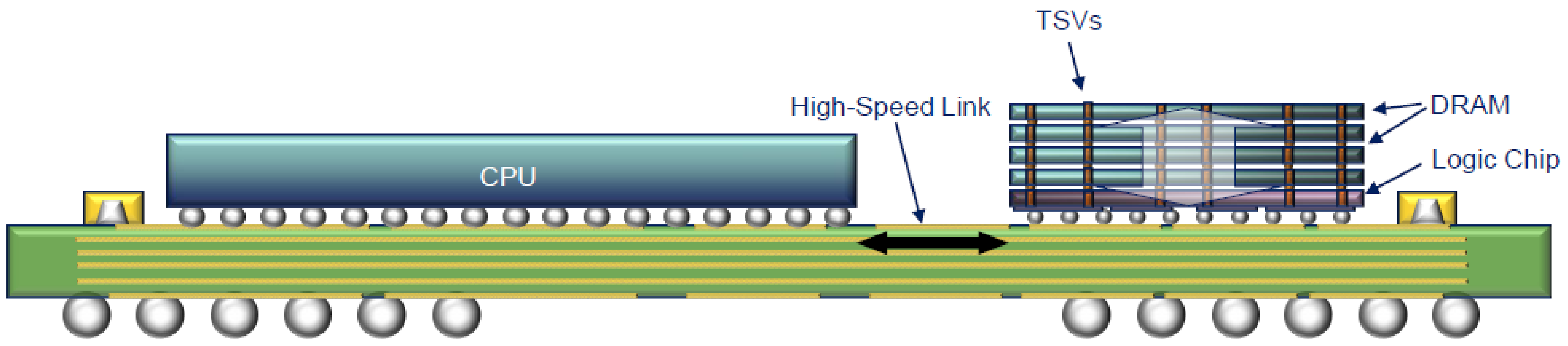
Notes: Tb/s = Terabits / second
HMC height is exaggerated

Figure courtesy Micron

HMC Near Memory – MCM Configuration

- All links are between host CPU and HMC logic layer
- Maximum bandwidth per GB capacity



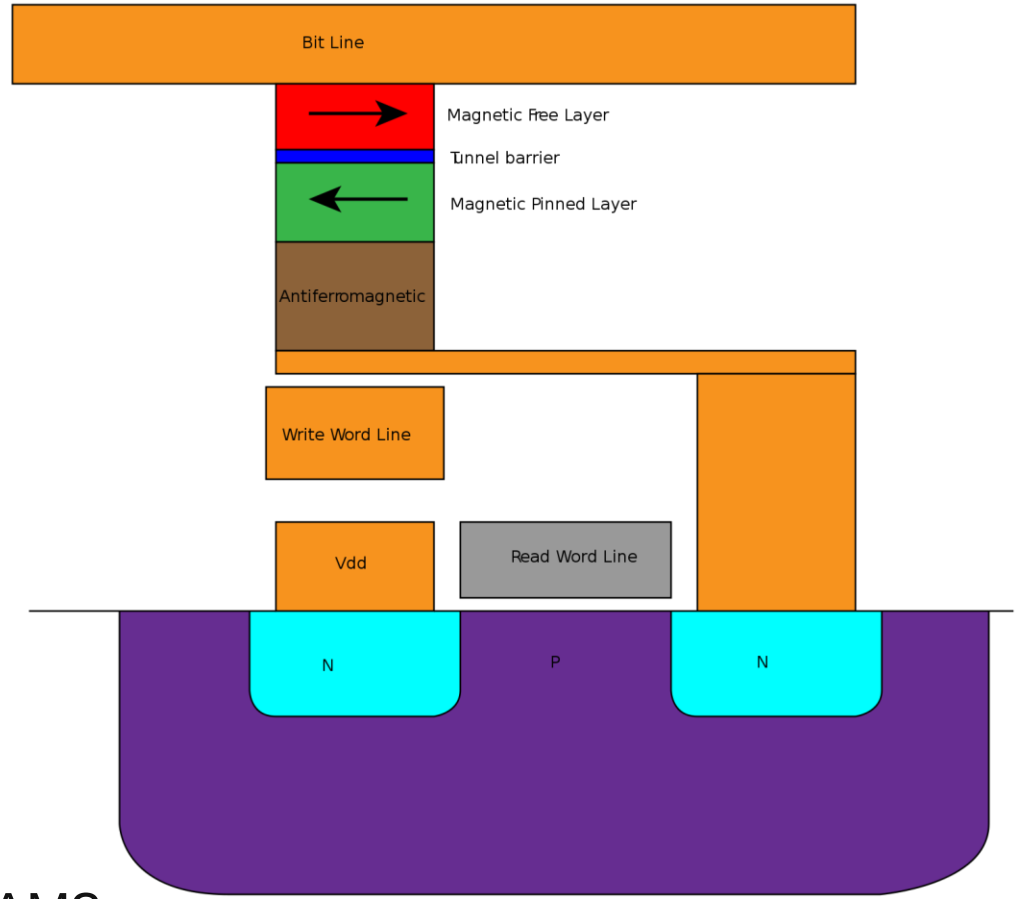


Notes: MCM = multi-chip module
Illustrative purposes only; height is exaggerated

Figure courtesy Micron

STT MRAM

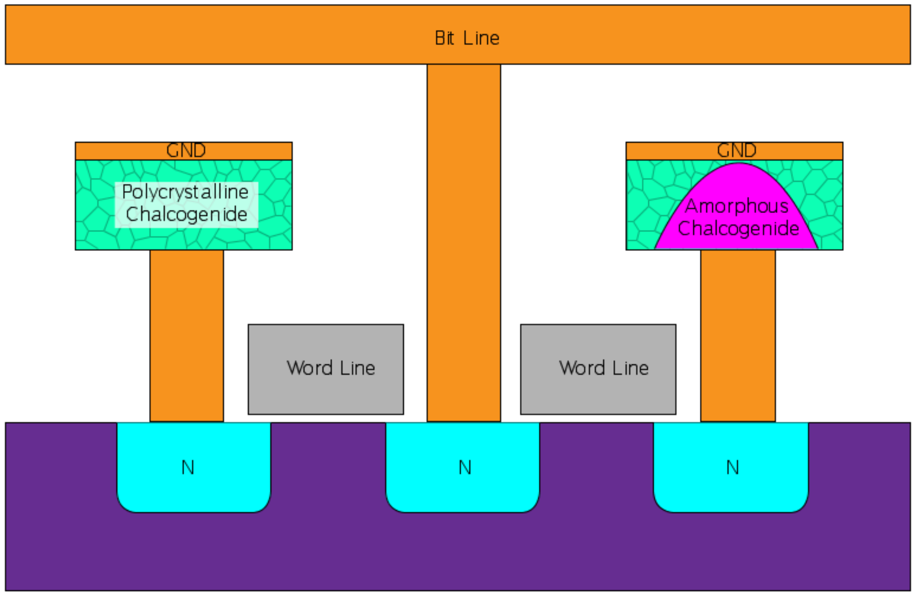
- Magnetoresistive RAM
  - Store bit in magnetic field
  - No power to hold value
  - Low power read/write
- STT
  - Spin Torque Transfer
  - Low power, fast
- Everspin
  - Shipping 4Mb parts
  - Annouced 16Mb part
  - SRAM or Flash interface
- Long term winner
  - Same feature size as DRAM?



Courtesy http://en.wikipedia.org/wiki/User:Cyferz

Phase Change Memory

- Based on state change instead of stored electrons
  - Crystalline vs. amorphous
  - Germanium-Antimony-Tellurium (GST) chalcogenide glass
- Change state by heating to 650ºC and then cooling
  - Cool quickly ⇒ amorphous
  - Cool slowly ⇒ crystalline
- Samsung, Micron shipping devices now (128Mb)
  - 1,000,000 overwrites
- Maybe DRAM replacement in 2015?
  - Byte addressible, but limited writes
  - Much lower power (no refresh)

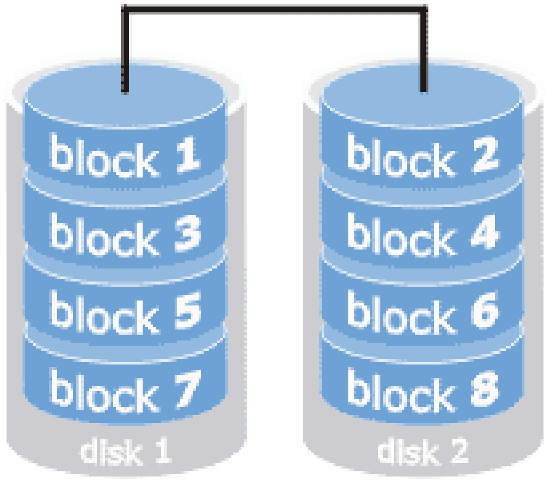


Courtesy http://en.wikipedia.org/wiki/User:Cyferz

RAID and Erasure Codes

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

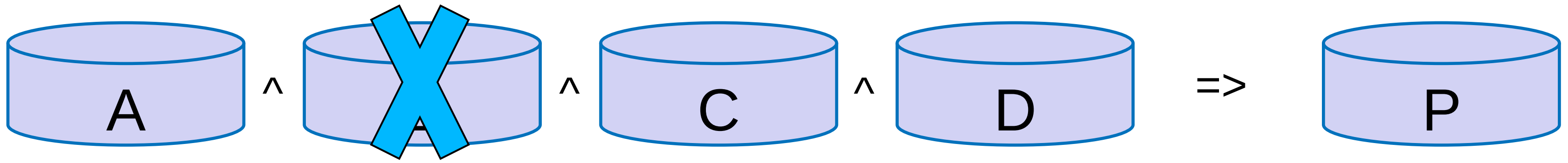


- **This simple two-disk system is twice as fast, but half as reliable, as a single-disk system**

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**

RAID levels

0 – None

- Stripe data over disks, no protection against failures

1 – Mirroring

2- Hamming codes, bit-level parity

3- XOR ECC, arm-locked, byte-level parity

4- XOR ECC, parity stripe unit

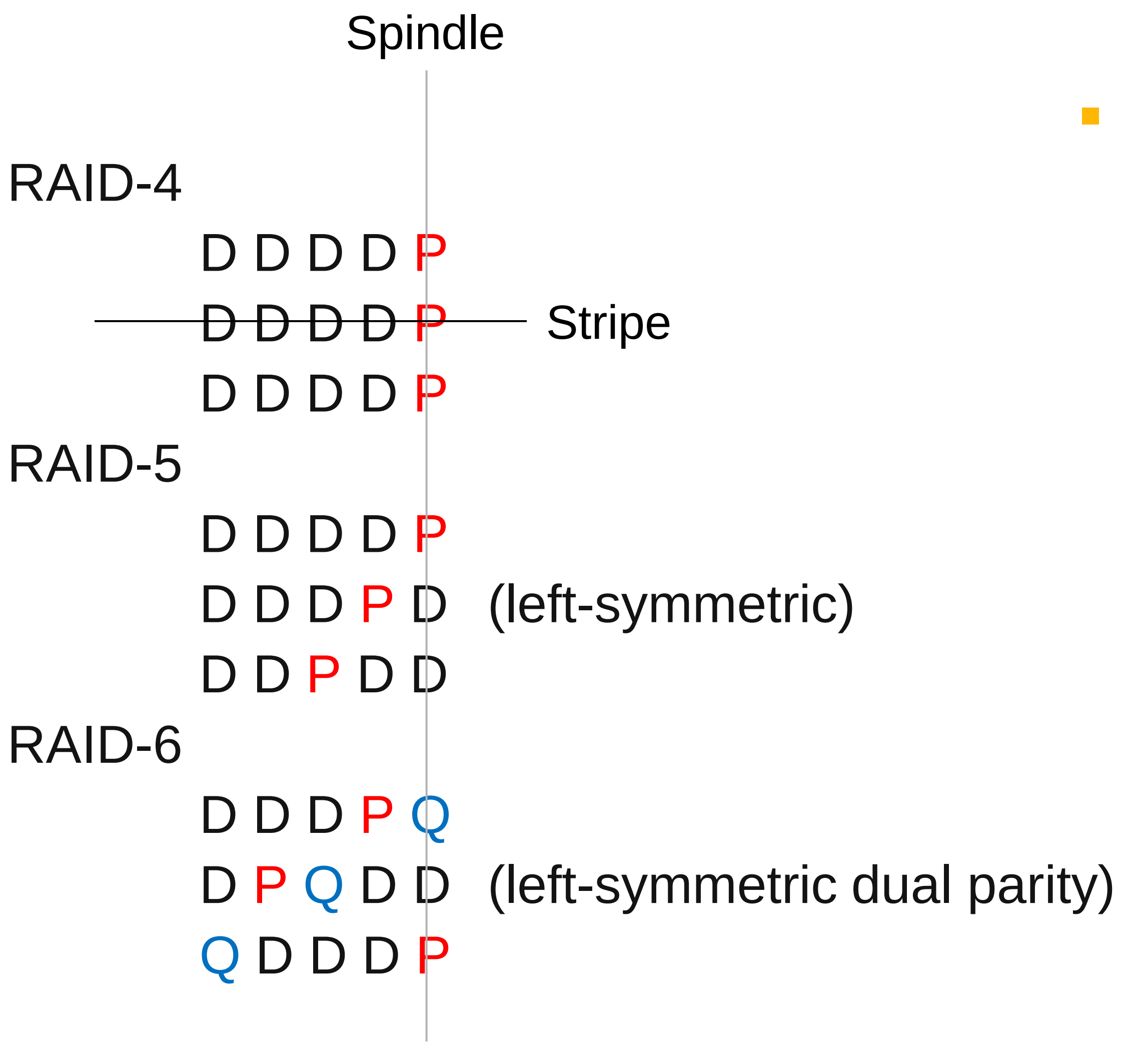
5- XOR ECC, rotated parity stripe unit

6- Multiple failure protection

- Reed-Solomon very popular
- Other erasure codes exist

10 (really 1+0)- both striped and mirrored

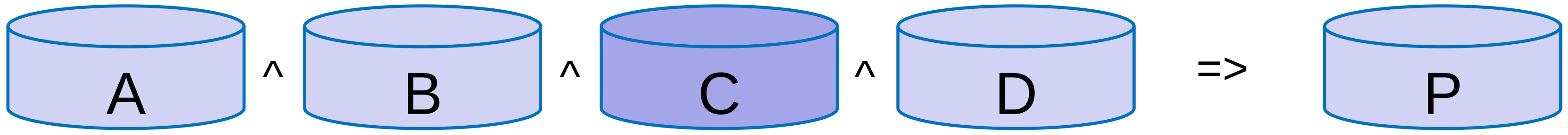
Rotating parity



- **Rotating parity diffuses the load from parity updates across all spindles**

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



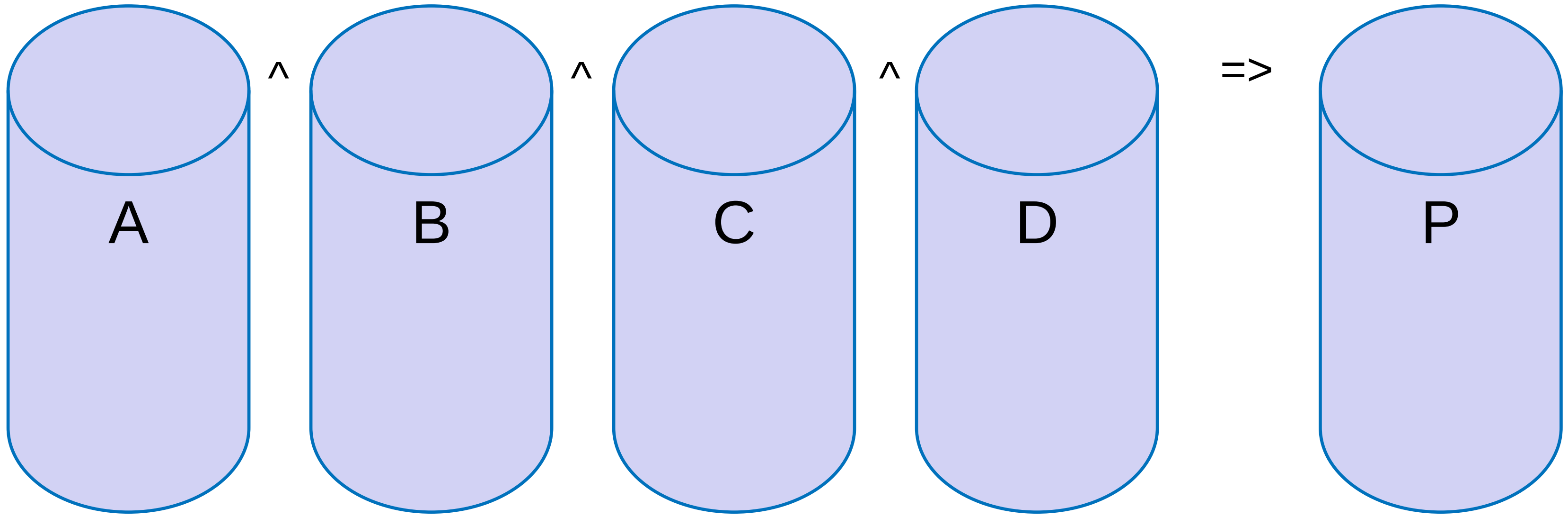
- 4-cycle write to update one disk
  - Read old value of C
  - XOR with new value of C and save the result in T
  - Write new value of C
  - Read old value of P
  - XOR T and old P, write the result as the new P

Erasure codes / Reed Solomon

- Used to provide additional levels of protection
- N is the number of redundancy units
  - May tolerate N failures
    - N=2: P, Q
  - May detect and correct up to N-1 corruptions
- First redundancy unit is a simple XOR
  - Thus, RAID-5 is equivalent to Reed-Solomon with N=1
  - Additional redundancy units require more complex math (Galois Field)
- Failures versus corruption
  - Tolerate up to N failures
  - Detect and repair up to N-1 corruptions

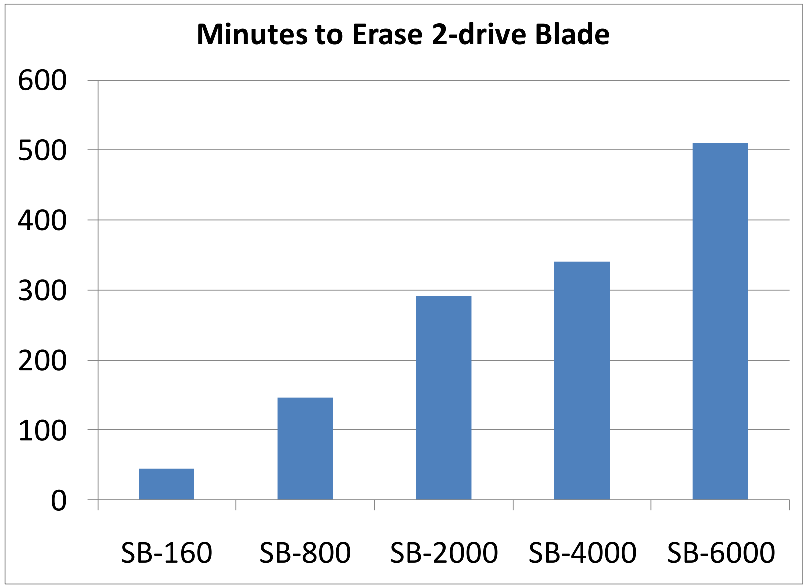
The problem with RAID

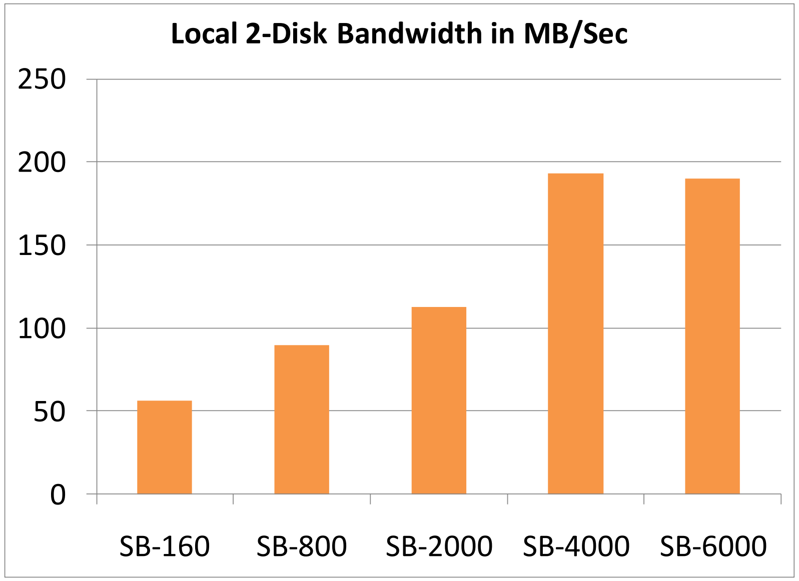
- Traditional block-oriented RAID protects and rebuilds entire drives
  - *Drive capacity increases have outpaced drive bandwidth*
  - It takes longer to rebuild each new generation of drives
  - Media defects on surviving drives interfere with rebuilds
- We need faster rebuilds, and a way to handle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

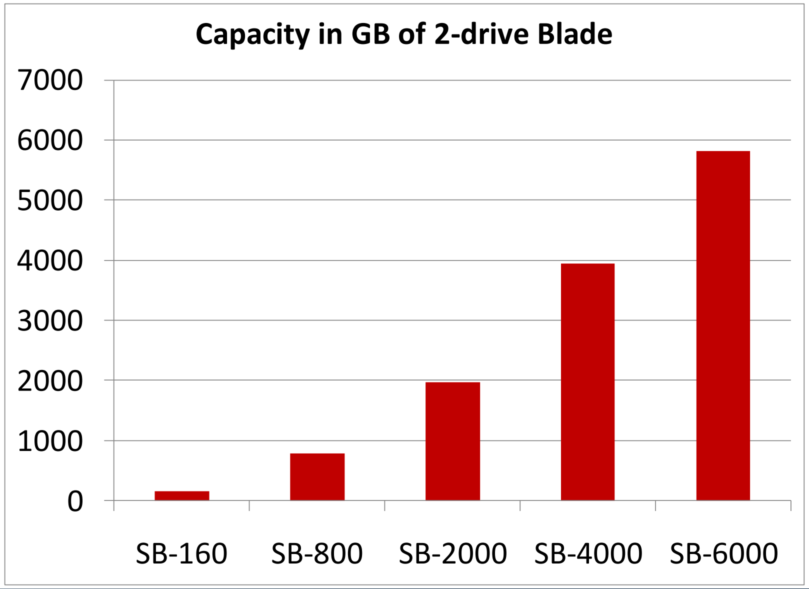


Blade Capacity and Speed History

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





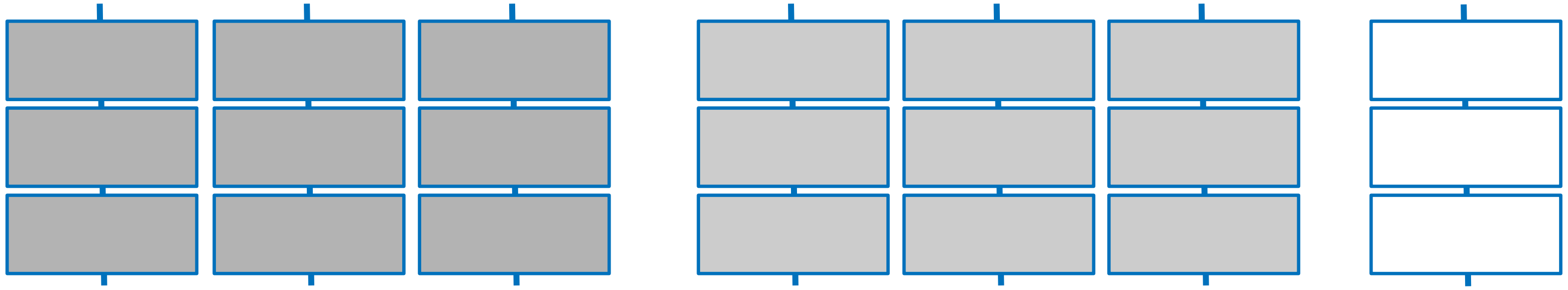


Improving RAID

- 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 resilience to media defects
  - Vertical parity across sectors to fix more media defects
  - Per-file RAID equation creates small fault domain
  - “Too many” cause loss of one file, not the whole RAID array

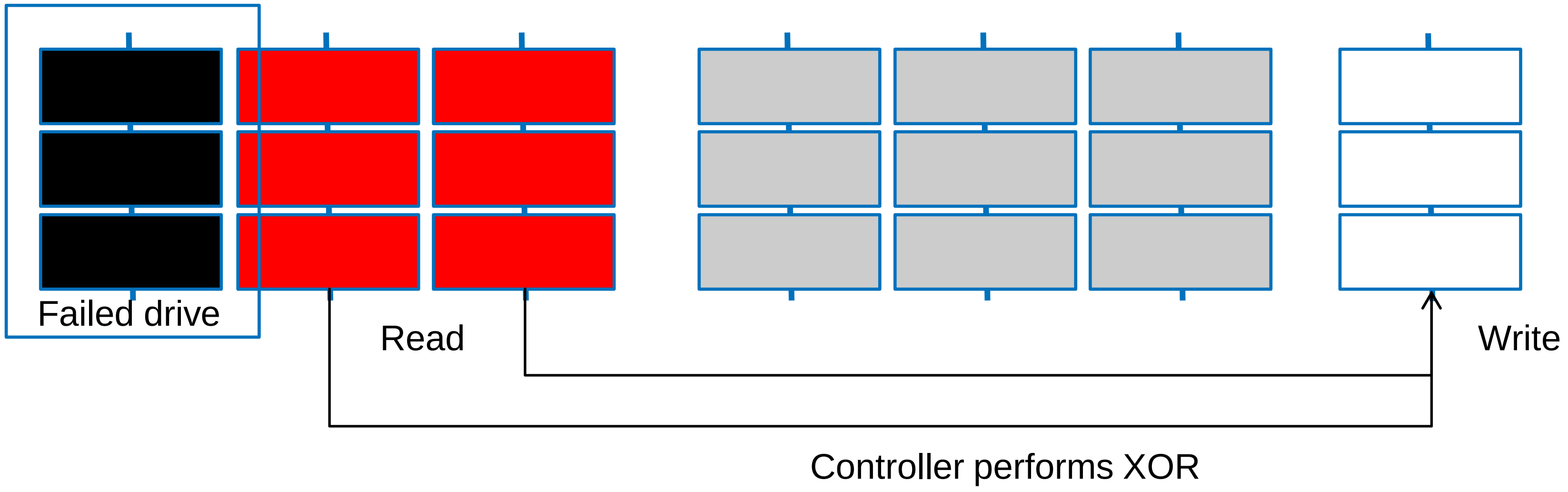
Traditional RAID Organization

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



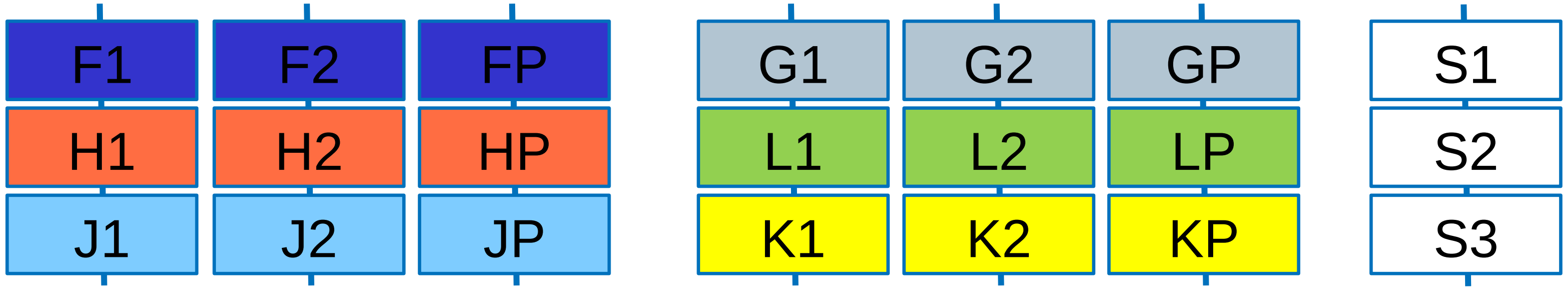
Traditional RAID Rebuild

- 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



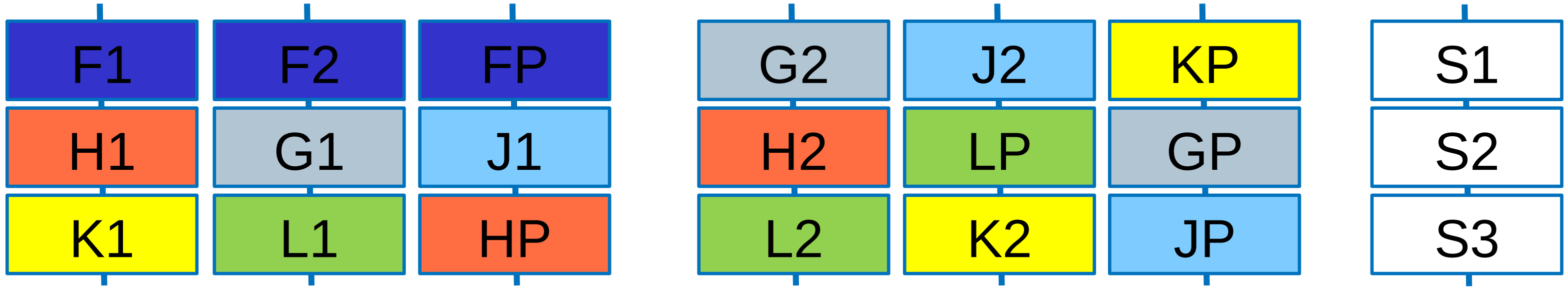
Declustering Step 1

- Subdivide devices into multiple partitions
  - F1 xor F2 => FP
  - H1 xor H2 => HP
  - etc



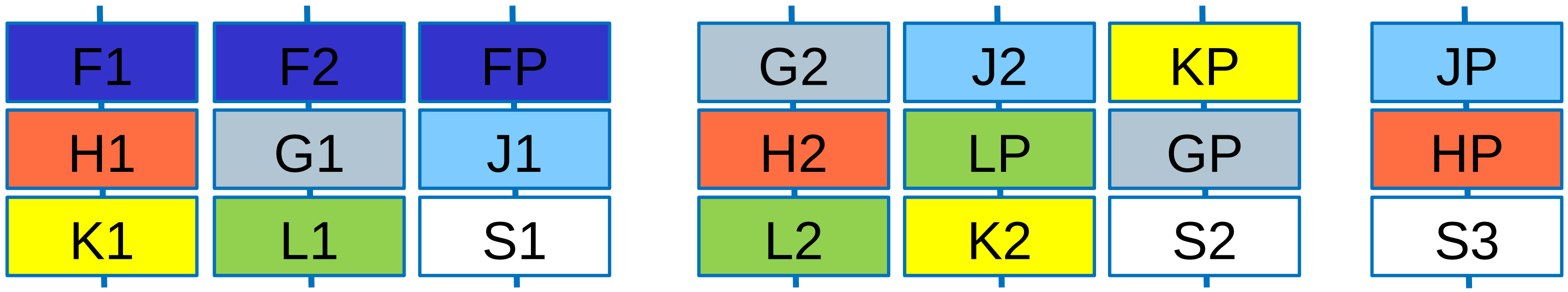
Declustering Step 2

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



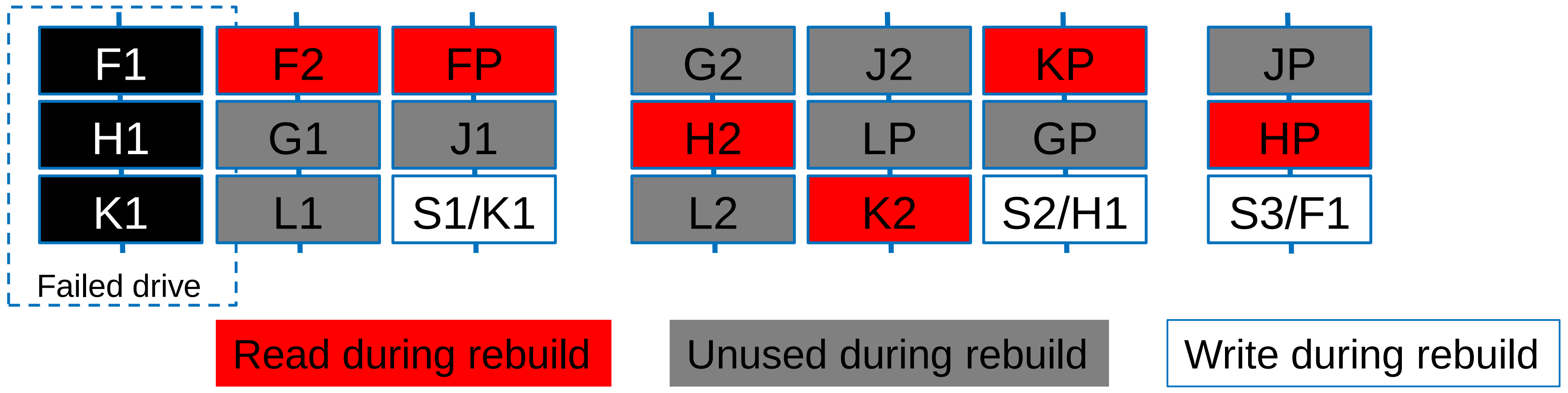
Declustering Step 3

- Spread out Spare space, too
- Placement constraints on what spare can be used
  - Cannot result in two pieces of a group on one device



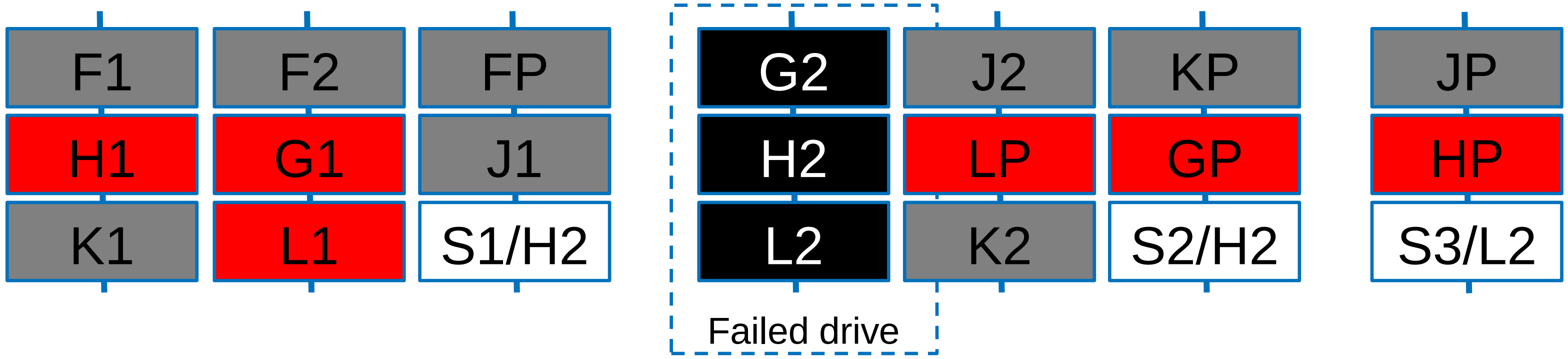
Declustered RAID Rebuild

- 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



Declustered RAID Rebuild

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



🟥 Read during rebuild

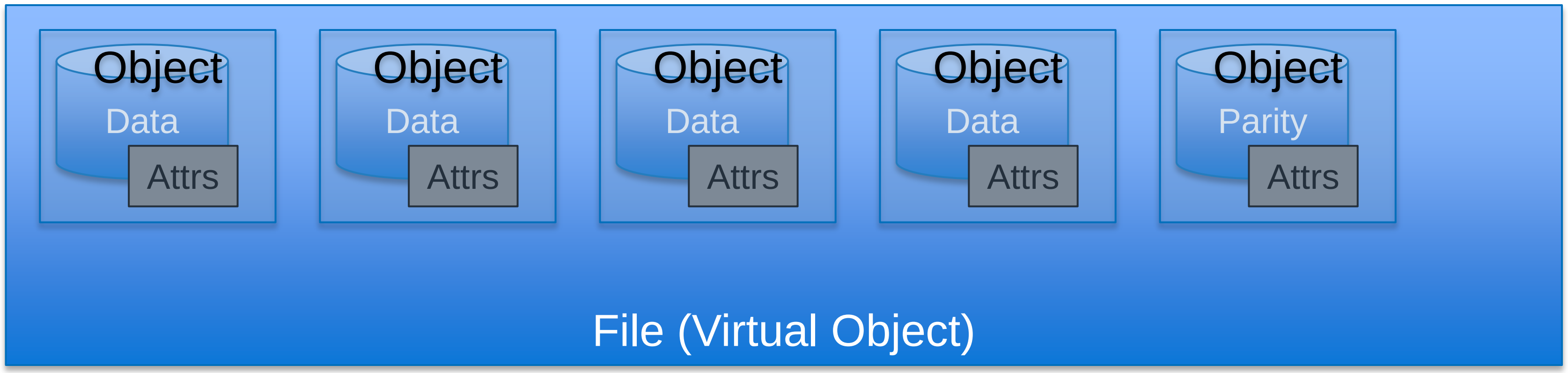
▨ Unused during rebuild

☐ Write during rebuild

Triplication and Google FS (and HDFS)

- Consider data nodes plus their disks as a single failure domain
- Triplicate file (chunks) and spread among data nodes
- Just like declustering, the rebuild workload is diffused among the data nodes
- Data nodes can do their work in parallel

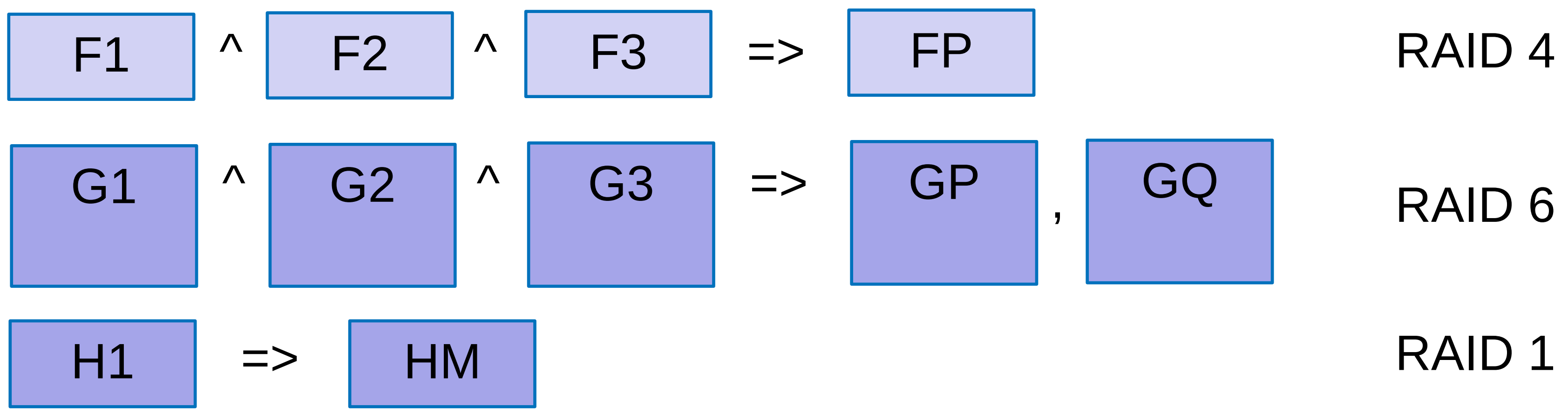
Object RAID



- Per-file data protection
- Small files (<64K) mirrored in two component objects
- Large files use RAID encoding across several component objects
- Parallel file system stores its metadata in object attributes
  - All attributes are mirrored on first two component objects that were created
  - Remaining component objects have just a few attributes
  - Attributes include map, parent, size, date stamps, owner, ACL

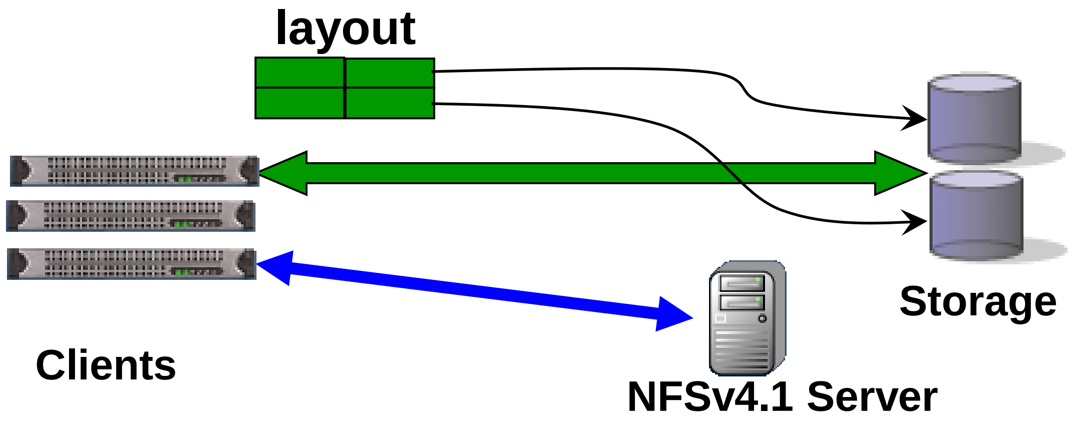
Object RAID

- Object RAID protects and rebuilds files
  - Failure domain is 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 Server
- The layout maps the file onto storage devices and addresses
  - Object-based layouts support per-file RAID
- 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

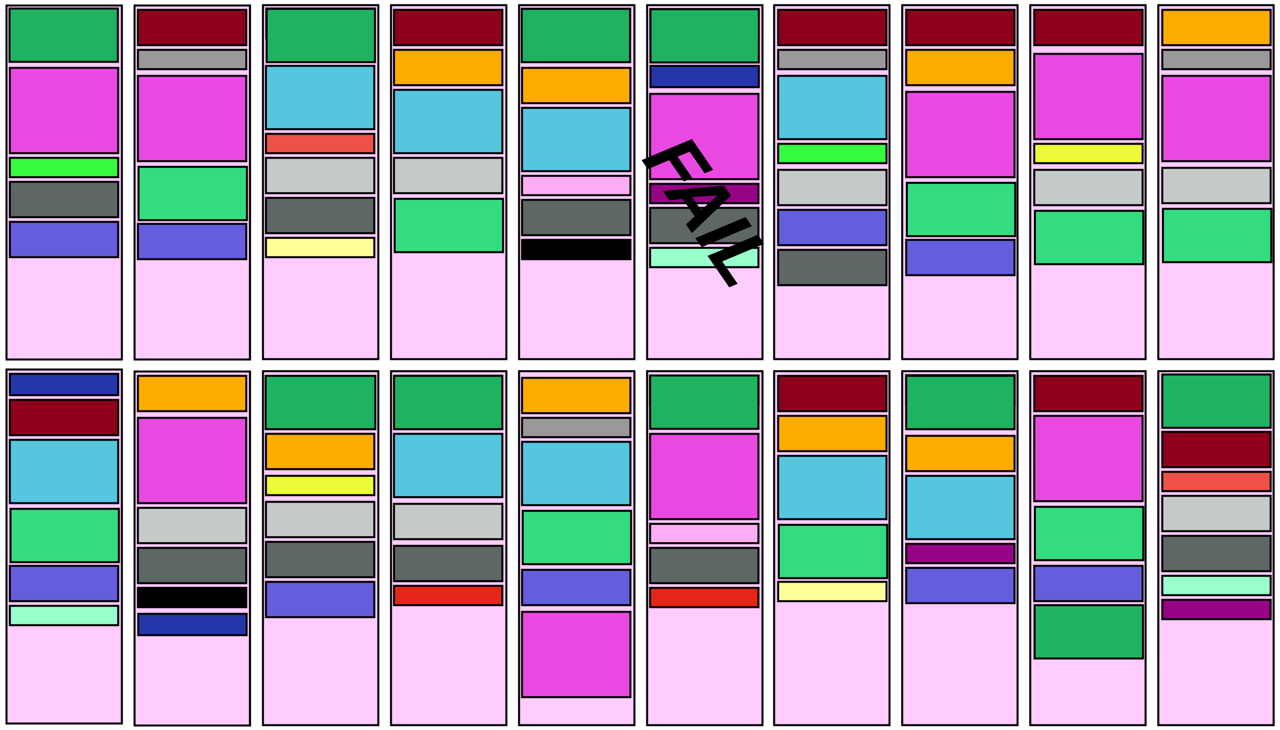


Parallel Declustered Object RAID

- 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



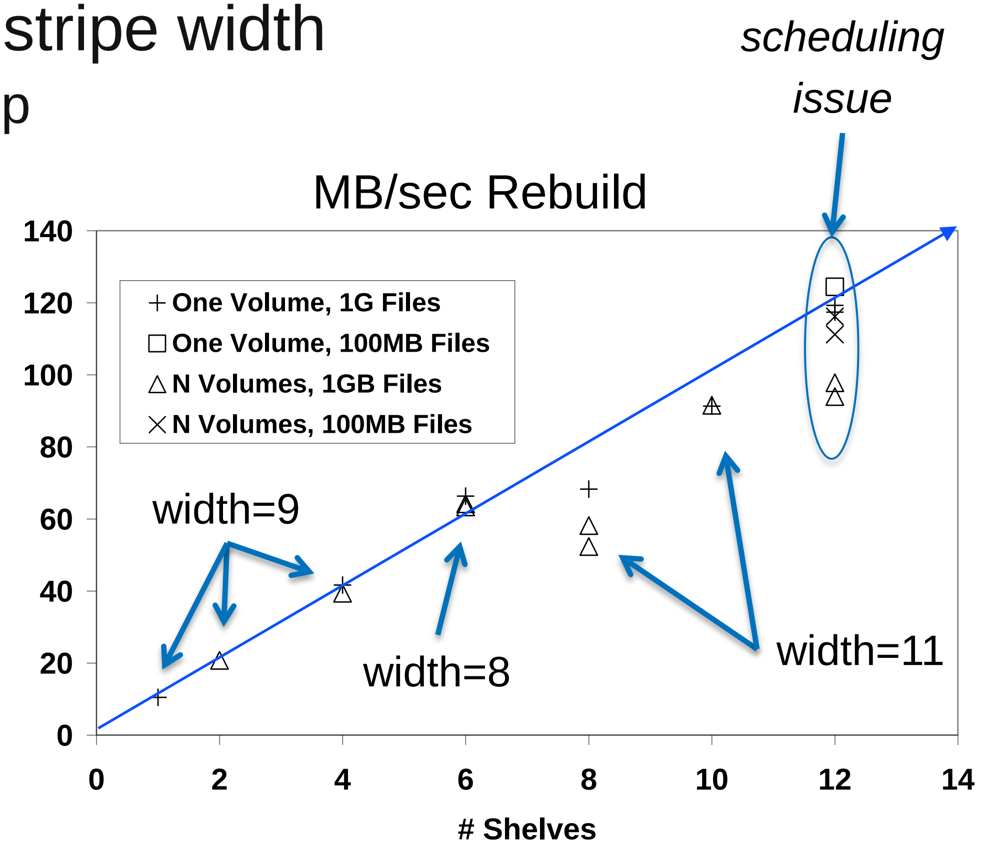
Read about half of each surviving OSD

Write a little to each OSD

Scales up in larger Storage Pools

Panasas Scalable Rebuild

- RAID rebuild rate increases with storage pool size
  - Compare rebuild rates as the system size increases
  - Unit of growth is an 11-blade Panasas “shelf”
    - 4-u blade chassis with networking, dual power, and battery backup
- System automatically picks stripe width
  - 8 to 11 blade wide parity group
    - Wider stripes slower
  - Multiple parity groups
    - Large files
- Per-shelf rate scales
  - 10 MB/s (old hardware)
    - Reading at 70-90 MB/sec
    - Depends on stripe width
  - 30-50 MB/sec (current)
    - Reading at 250-400 MB/sec



RAID Summary

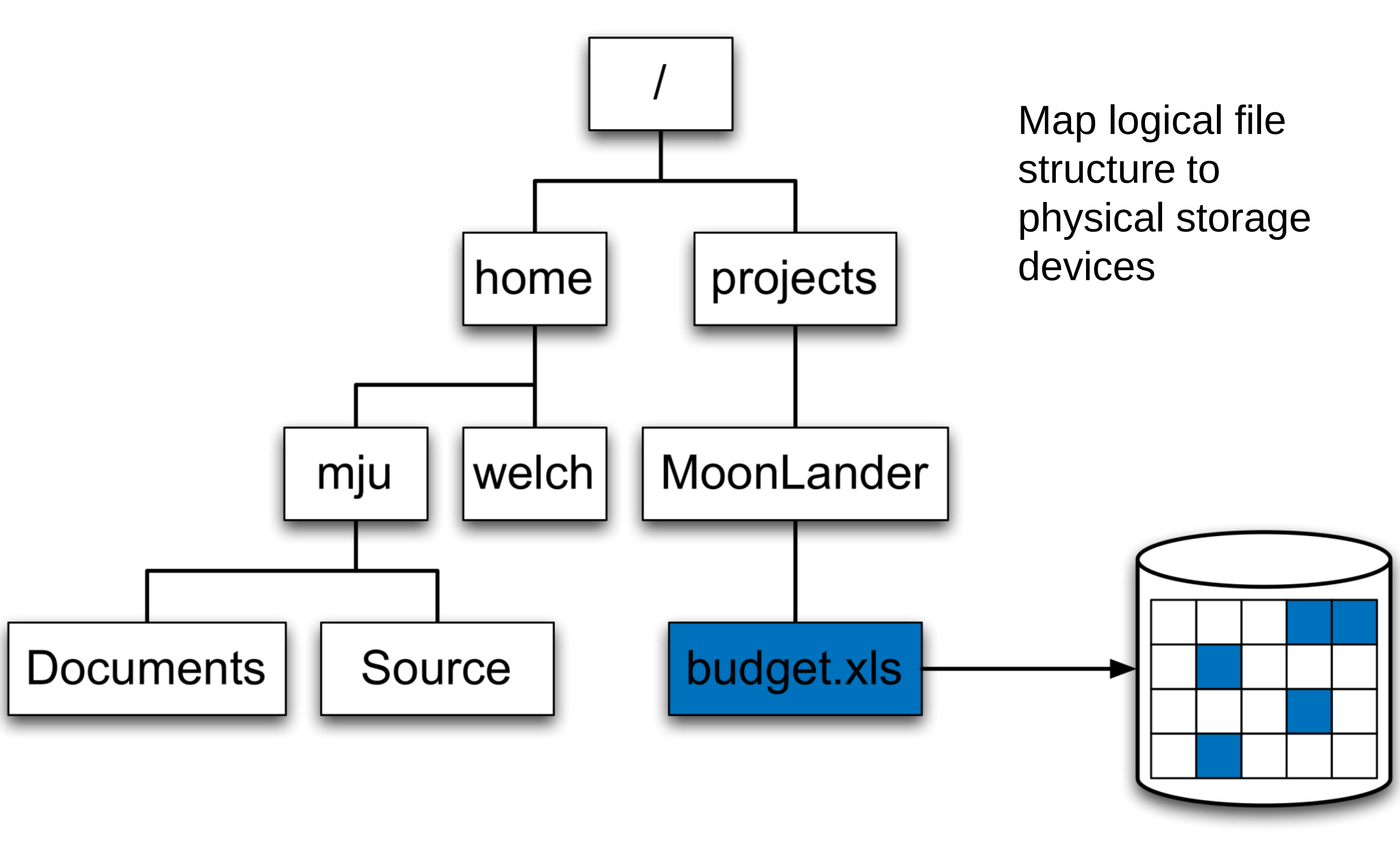
- RAID was invented for performance, but used for protection
- Block RAID is suffering from increased drive sizes
- Object RAID (or triplication) with parallel rebuild provides fast recovery
- Per-file RAID equations allow different performance/protection for different files, and isolate bad failures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

What is a file system and a parallel file system

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

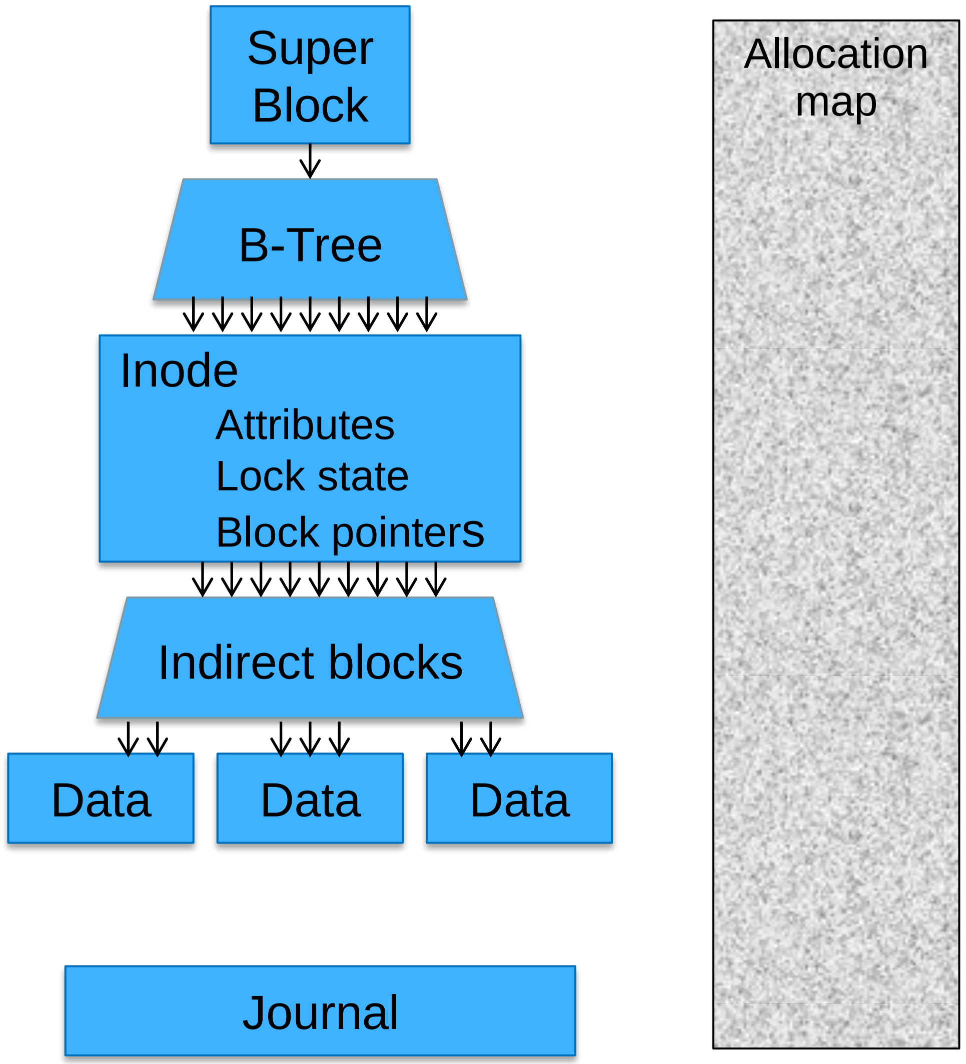
- File systems have two key roles
  - Organizing and maintaining the file name space
  - Storing contents of files and their attributes
- Networked file systems must solve two new problems
  - File servers coordinate sharing of their data by many clients
  - Scale-out storage systems coordinate actions 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

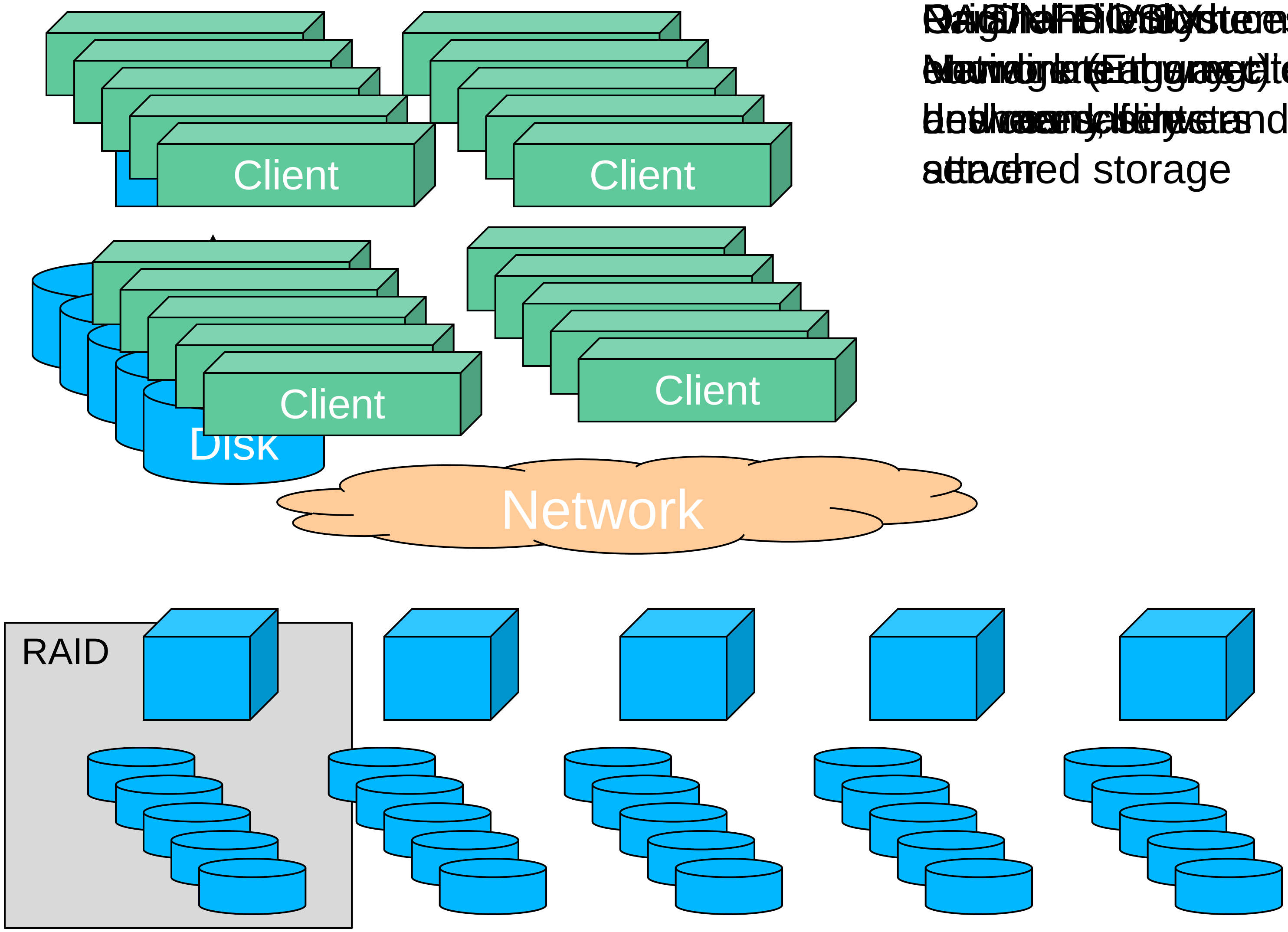
Persistent data structure maps from a user's concept of a file to the data and attributes for that file.

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

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

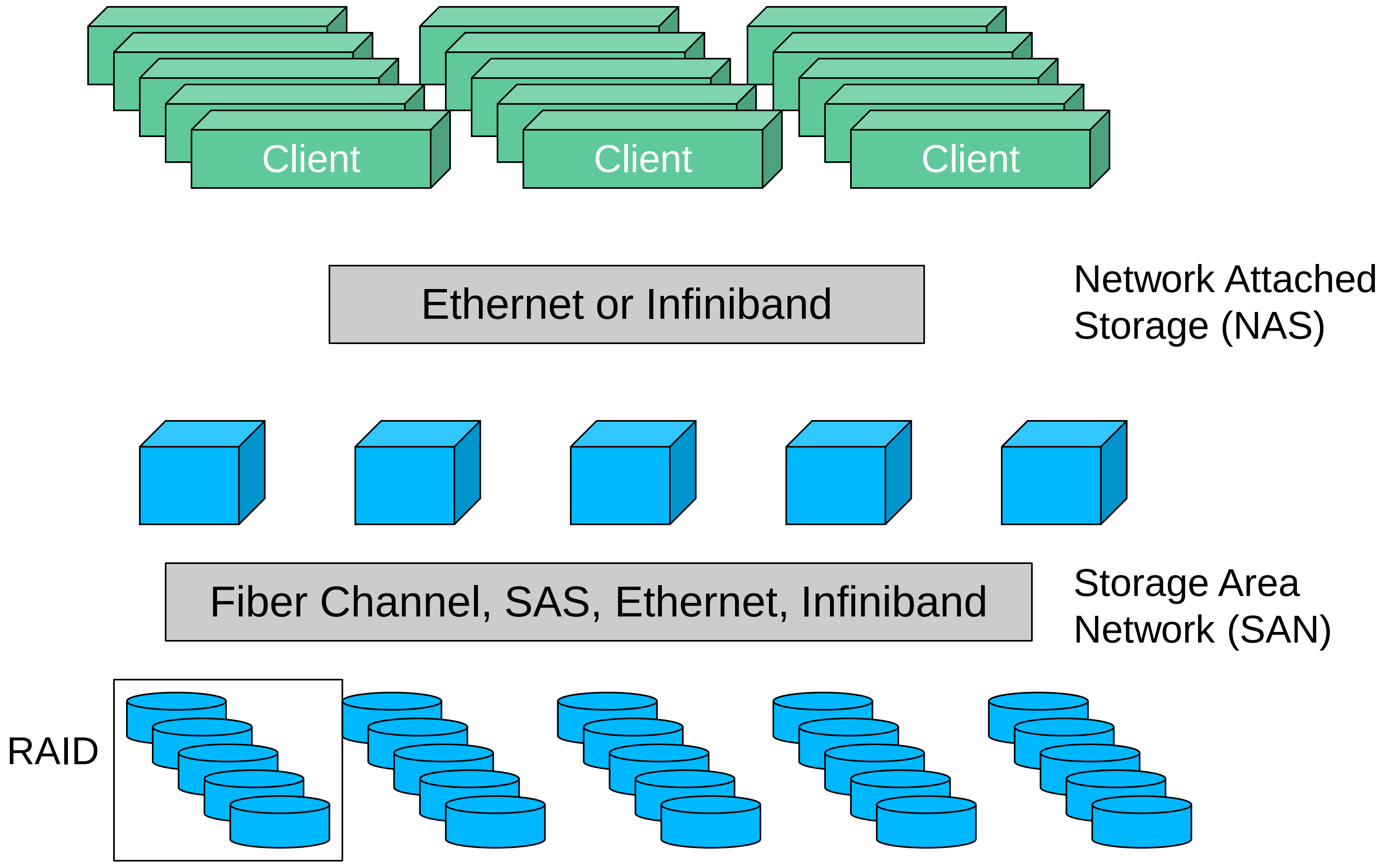


Scaling the File System



[illegible] a
[illegible] clients
[illegible] and
[illegible] attached storage

SAN vs NAS


Client
Client
Client
Ethernet or Infiniband
Network Attached Storage (NAS)
Fiber Channel, SAS, Ethernet, Infiniband
Storage Area Network (SAN)
RAID

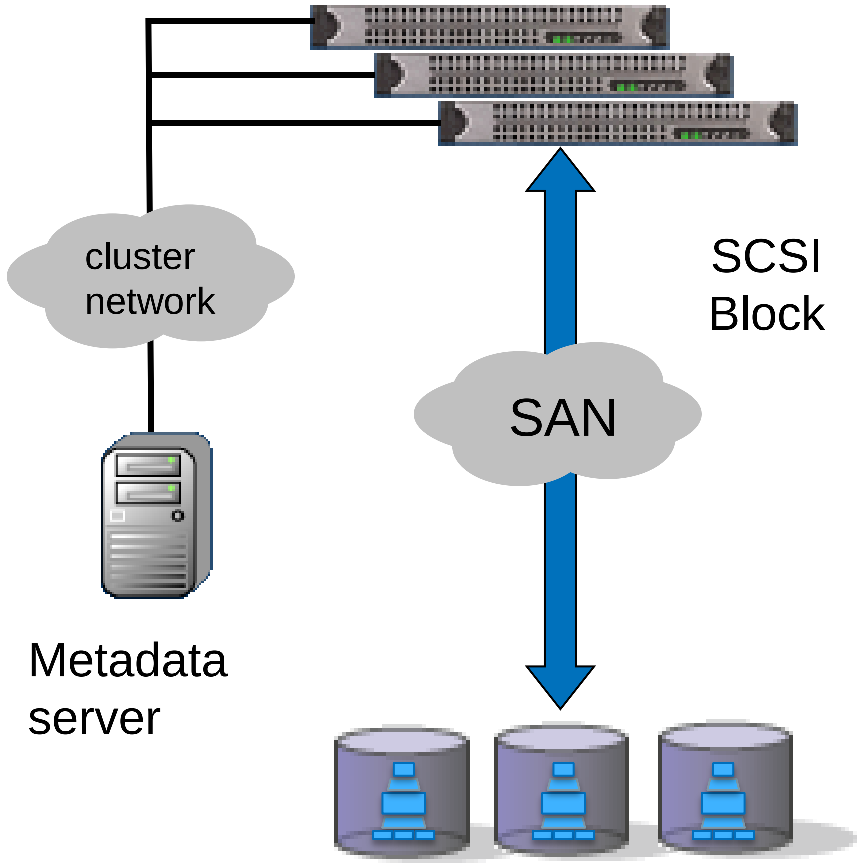
Distributed File System Functions

- 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

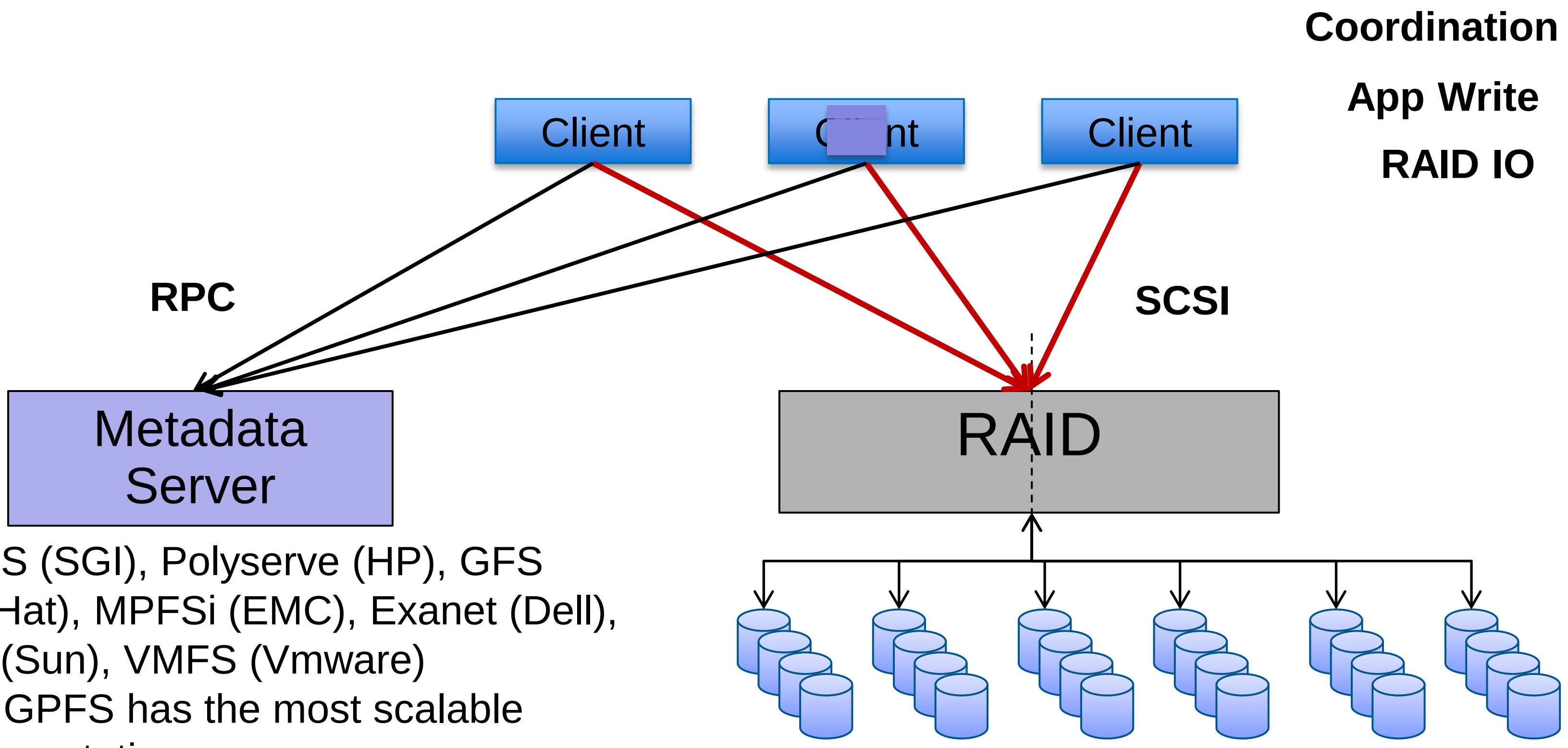
- Concurrent creates/deletes within a shared directory
  - Who owns the lock?
  - Who updates the directory?
  - Who can read the directory?
- **`ls -l`** in large active directory
  - Who knows how big the files are, and their modify time?
- Concurrent read/writer to a shared file
  - Who knows how 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



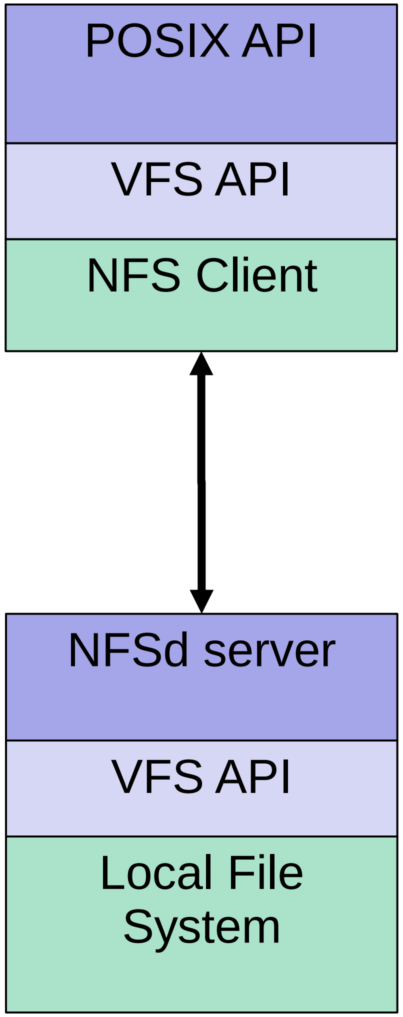
SAN FS Data Path

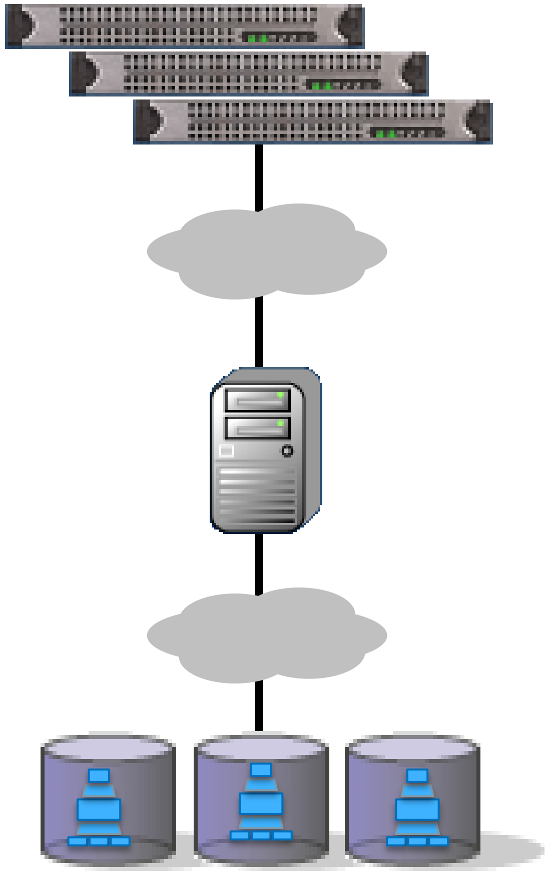
Clients access RAID arrays over the SAN.
Control protocol with metadata server coordinates access to shared disk via locking protocol
Local file system data structures are exposed to the clients



- 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)

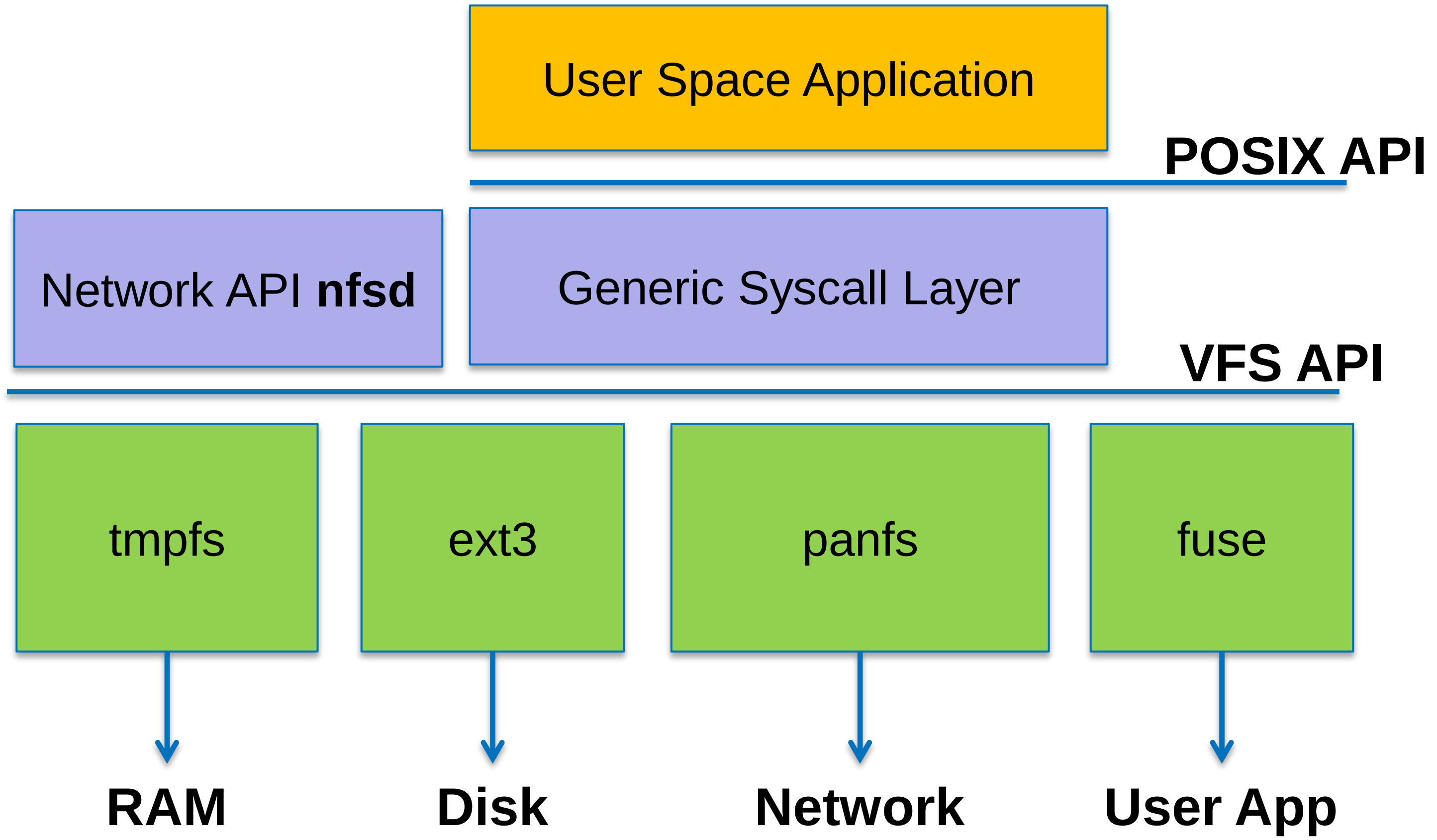




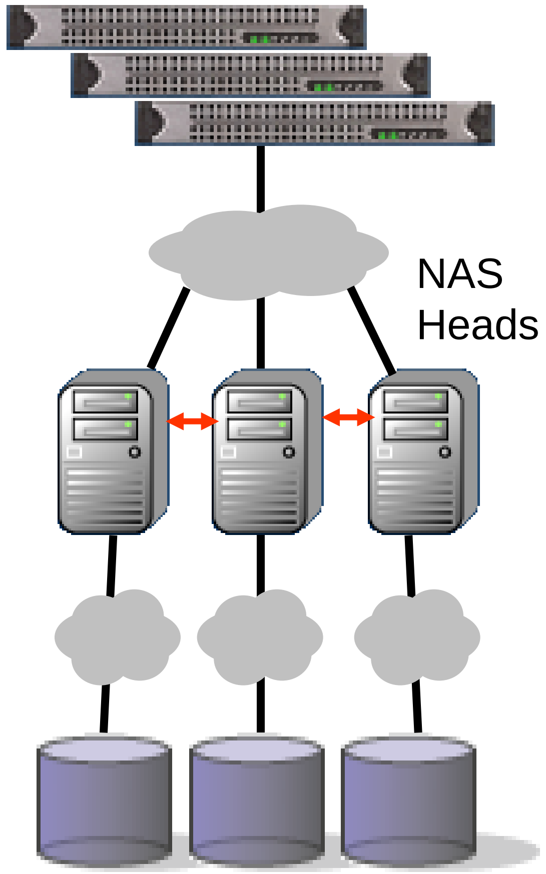
In kernel API layering to support NFS

Kernel VFS Layer

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

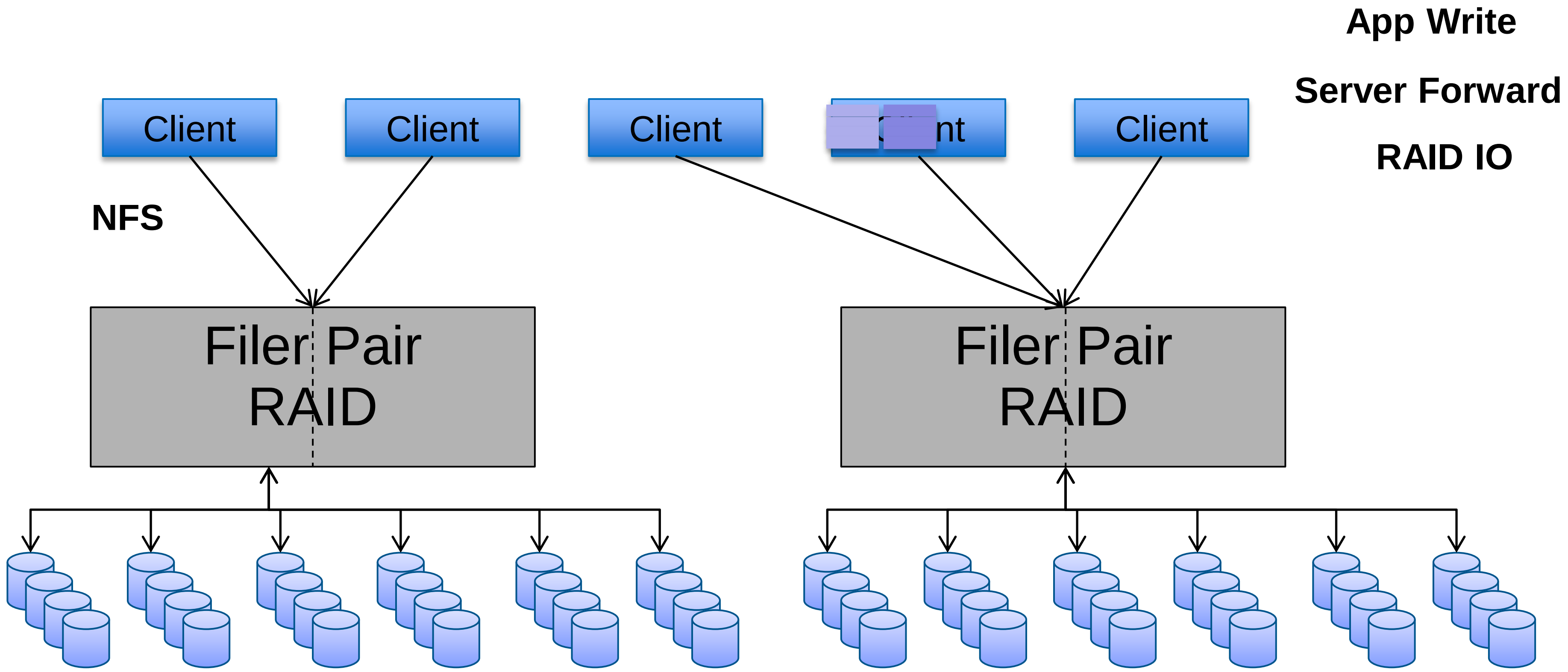


Clustered NAS



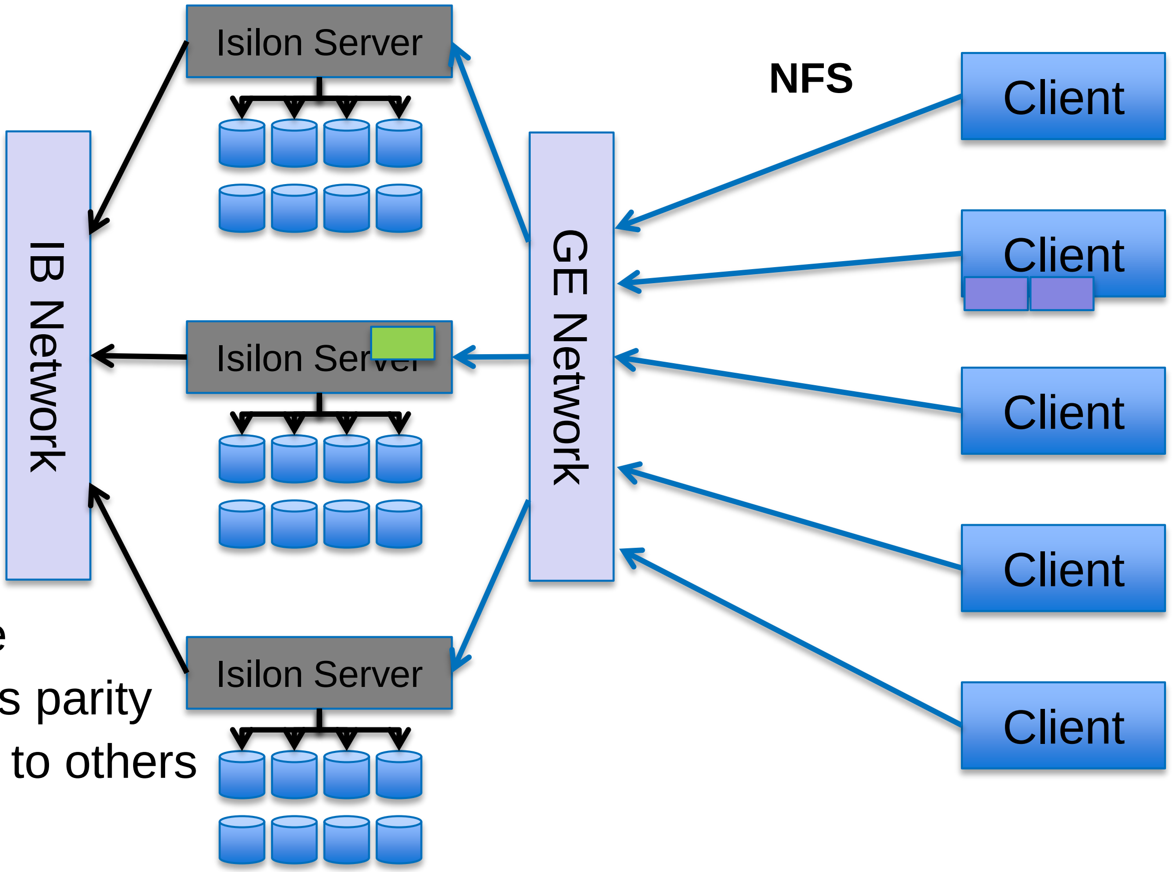
Clustered NAS Data Path

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



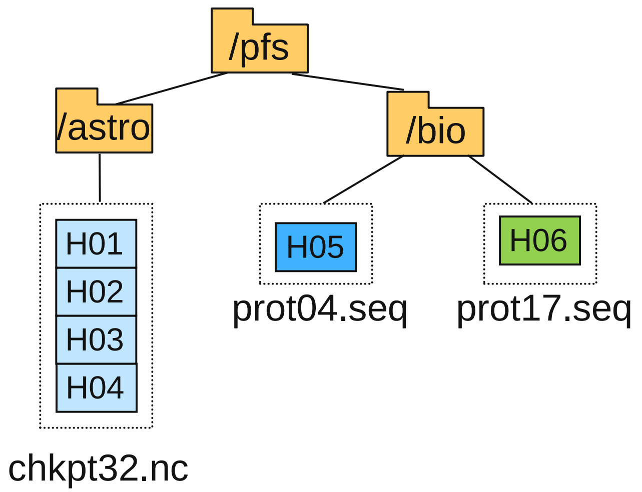
Isilon Data Path

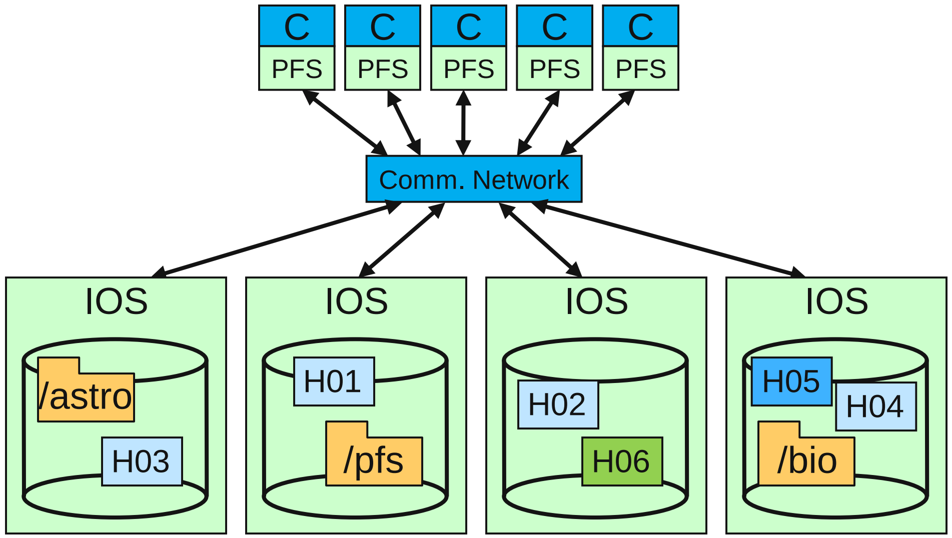
Isilon nodes compute parity and forward to others



- 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

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

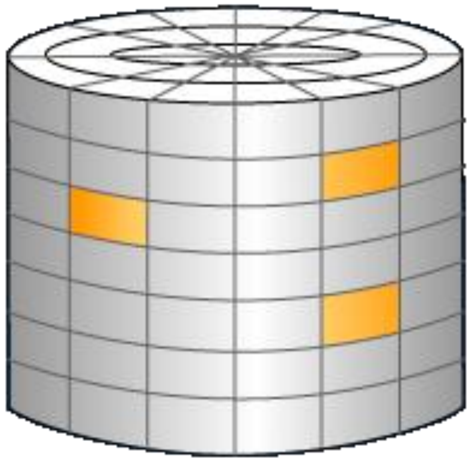
Read block
Write block

Addressing

Block range

Allocation

External



Object Based Device

Operations

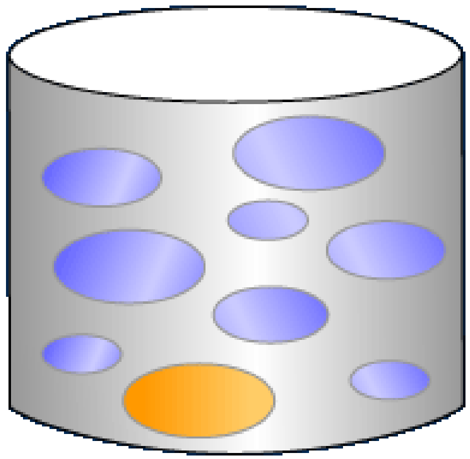
Create object
Delete object
Read object
Write object
Get Attribute
Set Attribute

Addressing

[object, byte range]

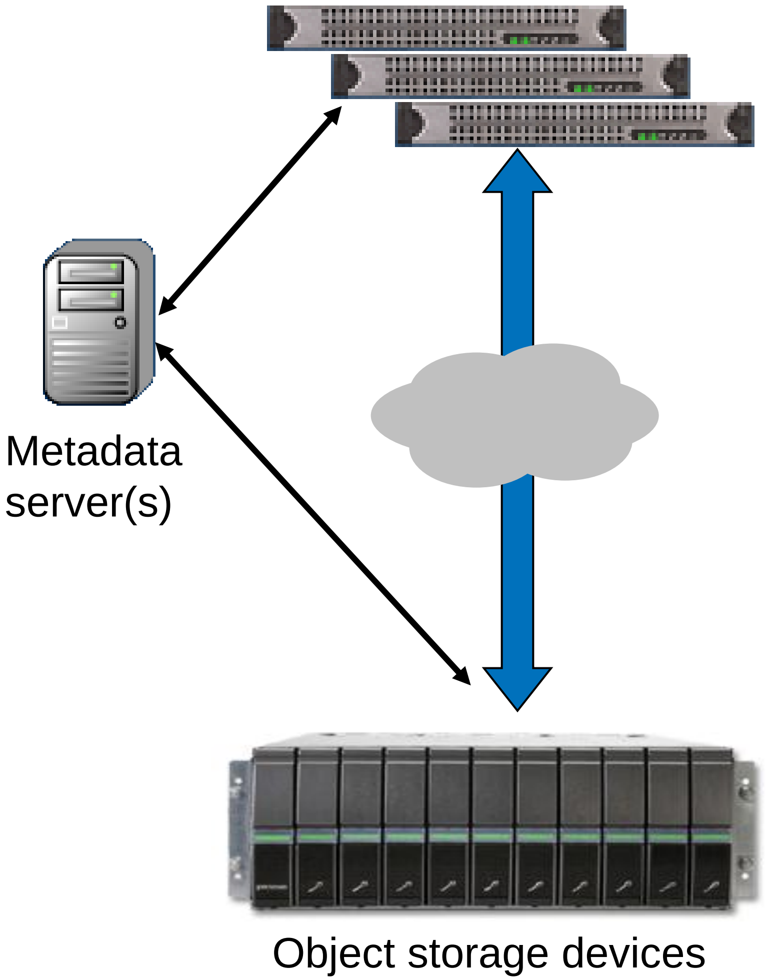
Allocation

Internal



Object-based Storage Clusters

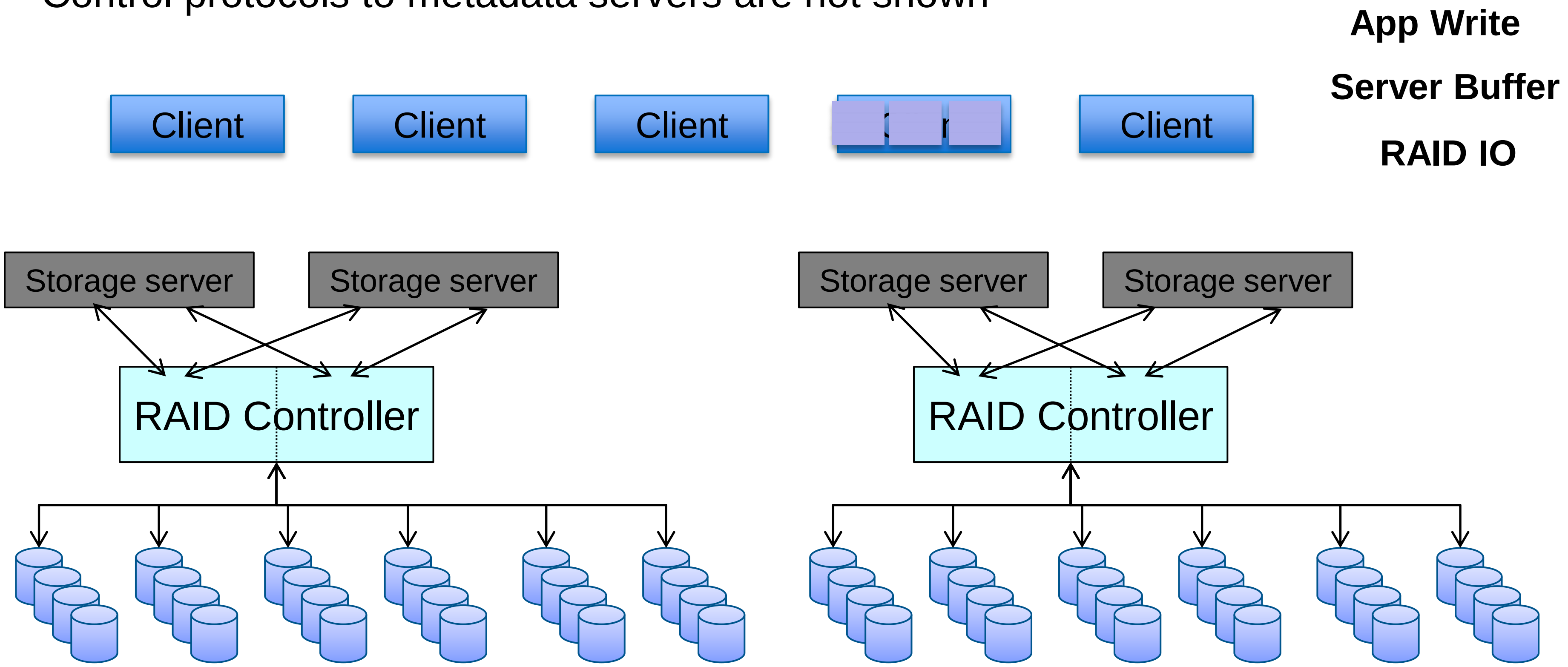
- Lustre, PanFS, Ceph, PVFS
- File system layered over objects
  - Details of block management hidden by the object interface
  - Metadata server manages namespace, access control, and data striping over objects
  - Data transfer directly between OSDs and clients
- High performance through clustering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



Lustre and GPFS Data Path

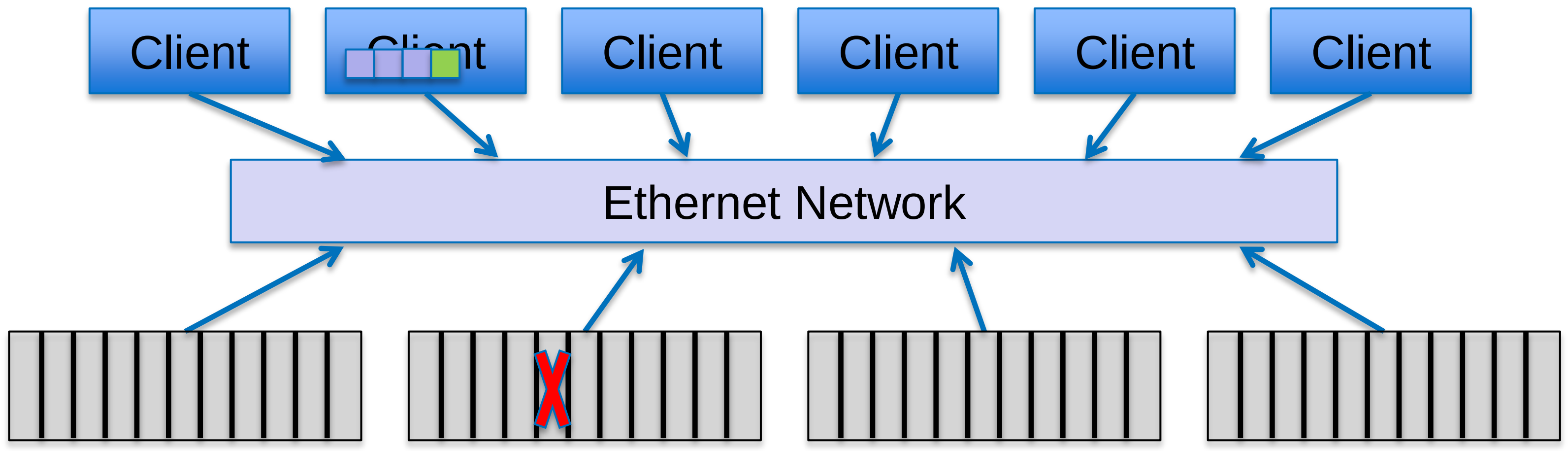
Lustre clients stripe data across Object Storage Servers (OSS), which in turn write data through a RAID controller to Object Storage Targets (OST). OST hides local file system data structures

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



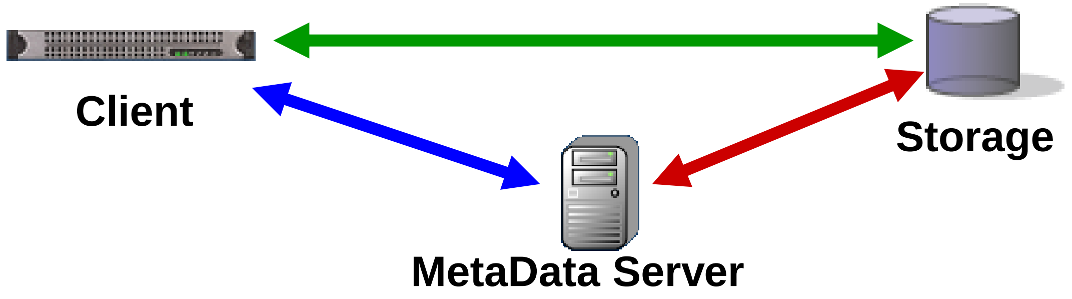
Panasas Parallel Data Path

- Data path by-passes RAID controllers and metadata servers
  - Control (RPC) path to metadata servers not shown here
  - Application writes data
  - DirectFlow/pNFS client layer generates redundant data for each stripe
  - Everything is written directly to storage
  - All blades work together on RAID rebuild



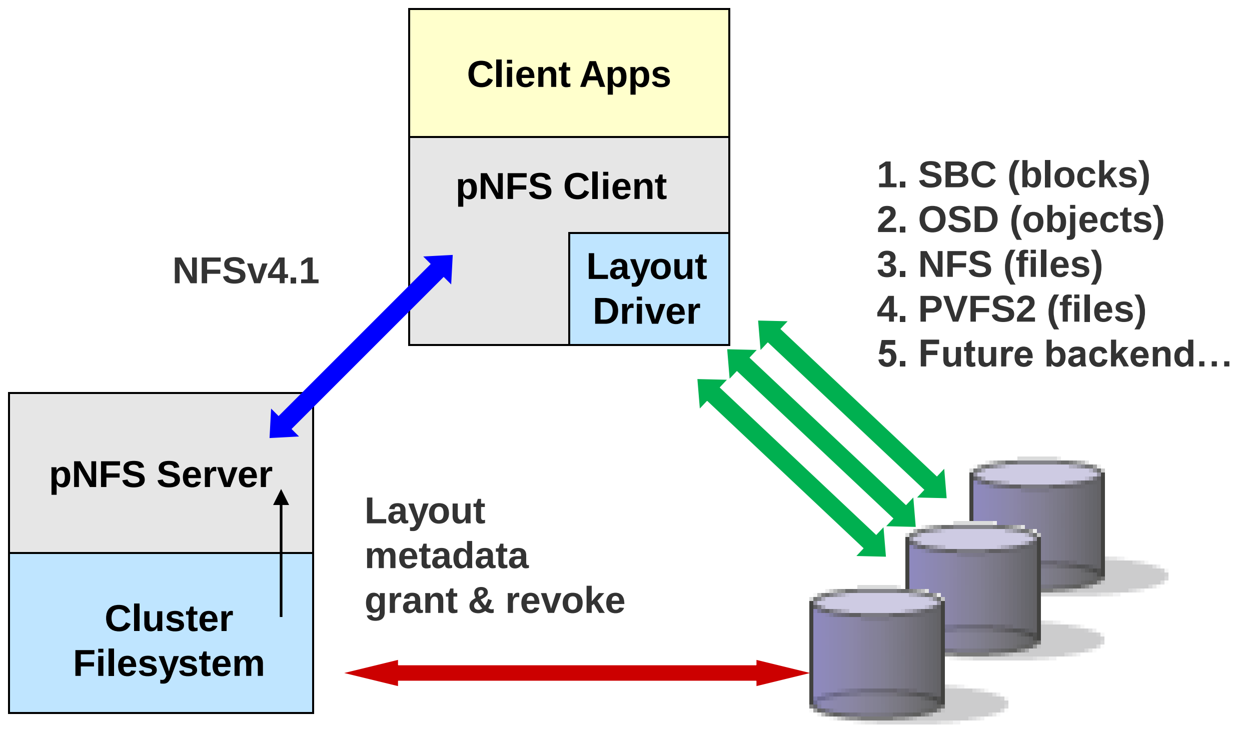
The pNFS Standard

- 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



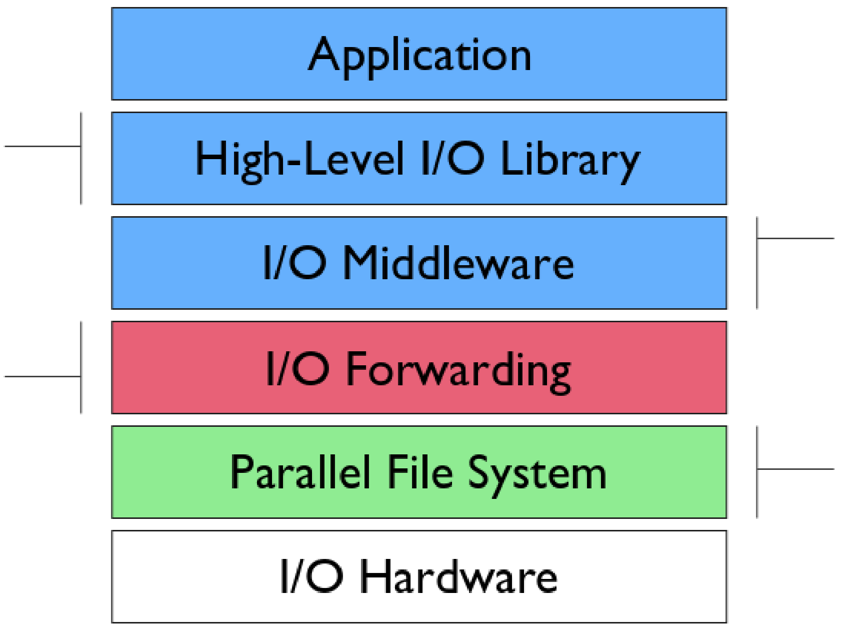
Linux Release Cycle 2011-2012

Kernel	Merge Window	What's New
2.6.38	Jan 2011	More generic pNFS code, still disabled, not fully functional
2.6.39	Apr 2011	Files-based back end, read, write, commit on the client. Linux server is read-only via pNFS.
3.0	Jun 2011	Object-based back end (RAID-1 only)
3.1	Sep 2011	Block-based back end
3.2	Dec 2011	Object RAID Engine adds RAID-5
3.3	Feb 2012	Bug Fixes
3.4	Apr 2012	iSCSI/OSD auto login
3.5	July 2012	Bug Fixes

- RHEL 6 and SLES 11 based on 2.6.32
  – Backporting pNFS is in progress
- RHEL 7 and SLES 12 based on 3.*
  – Integrated pNFS of all flavors – timeline 2013

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

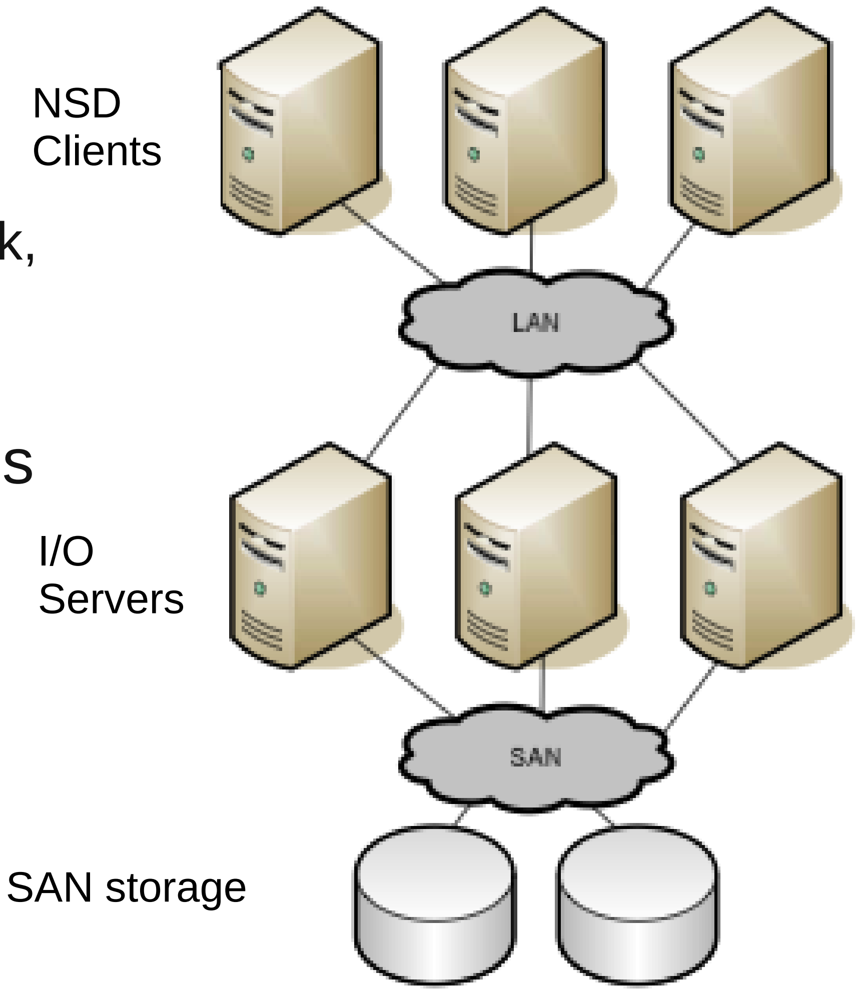
Production Parallel File Systems

- 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

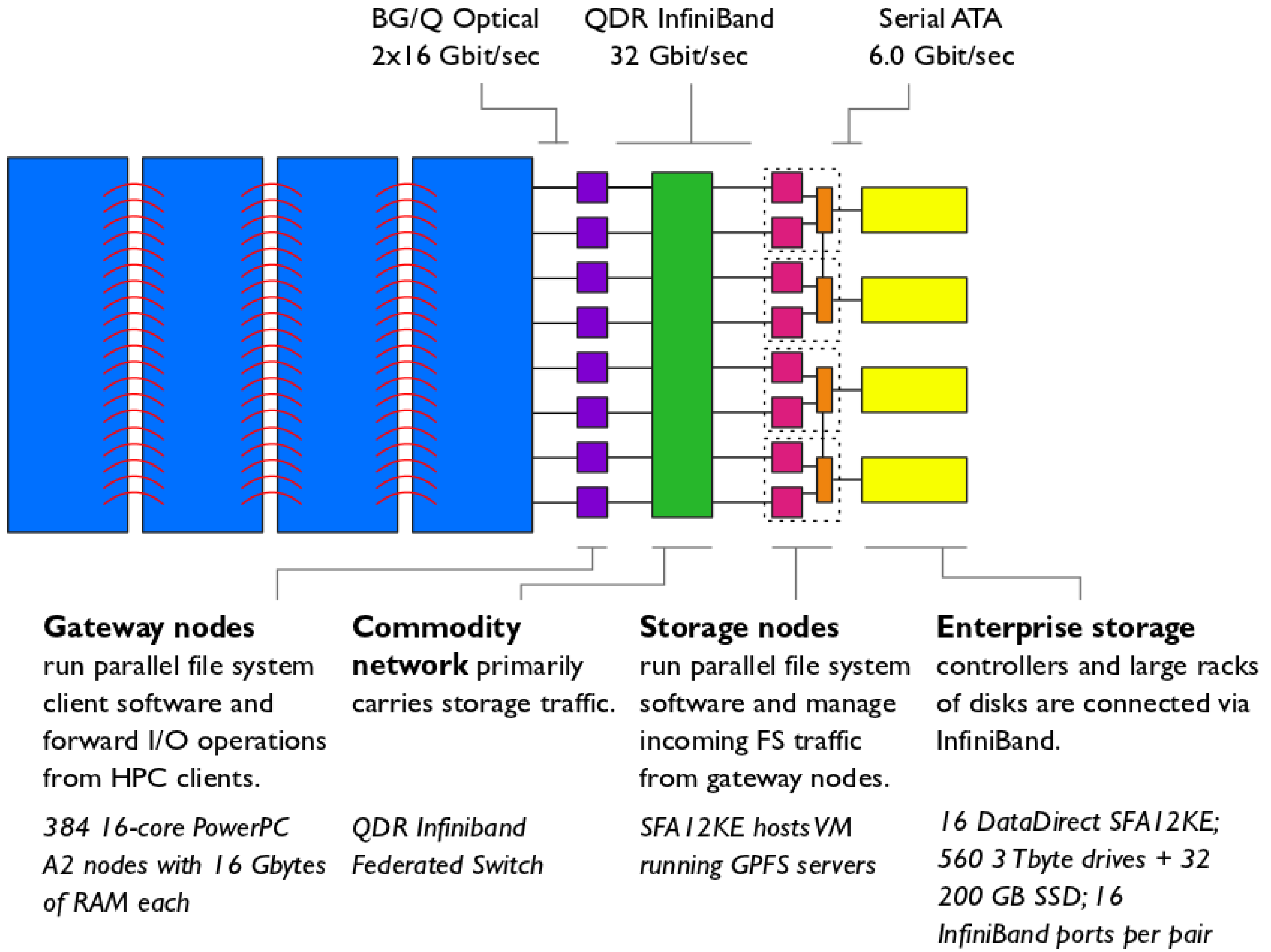
GPFS

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.

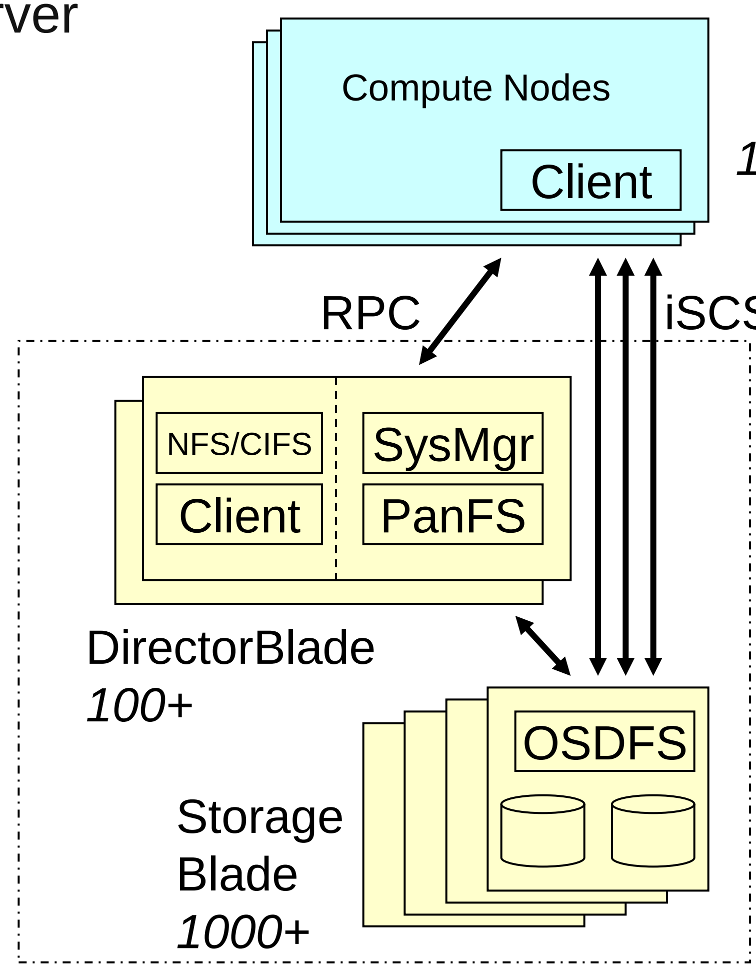
SFA12KE hosts VM running GPFS servers

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

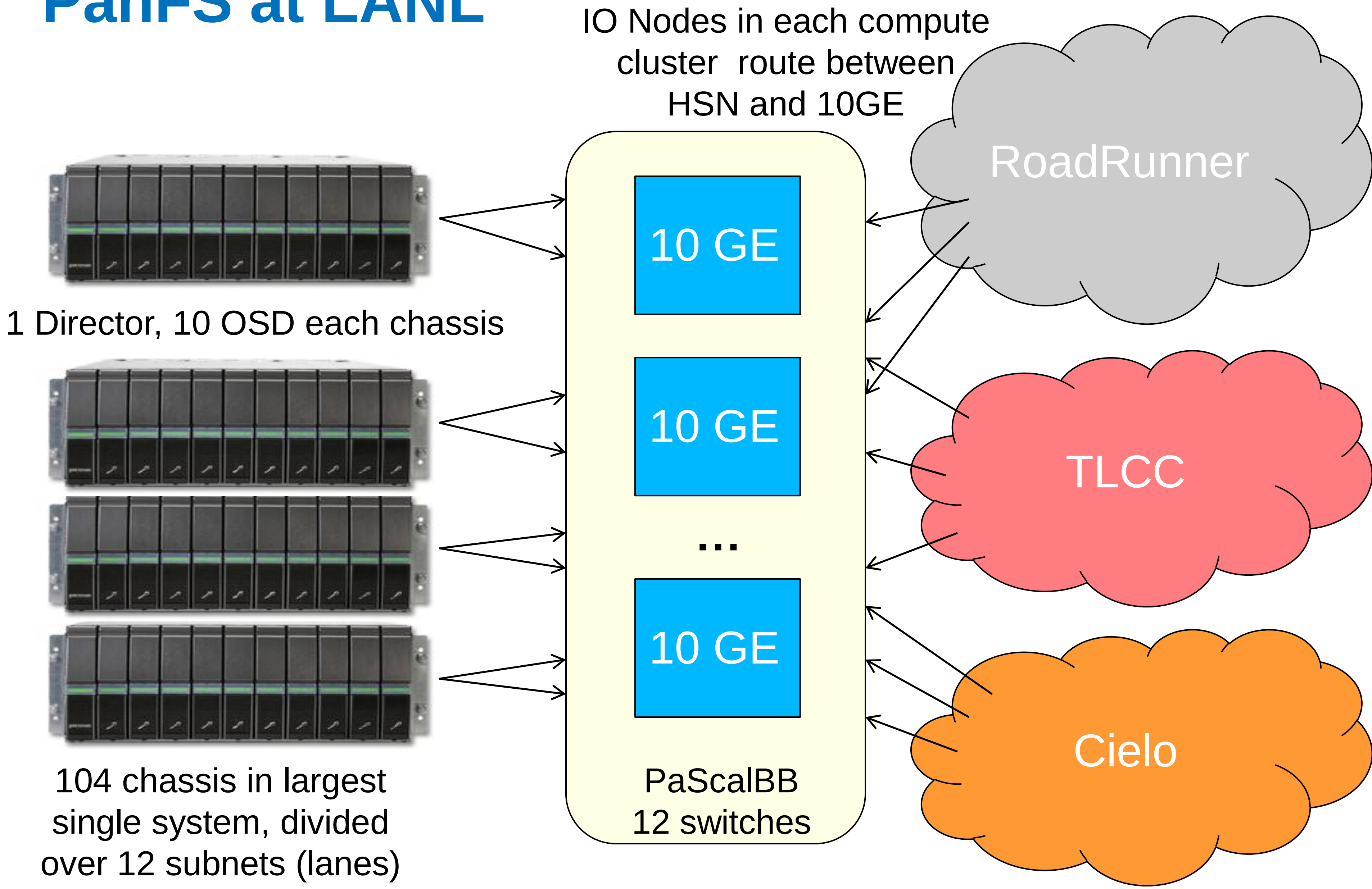
16 DataDirect SFA12KE; 560 3 Tbyte drives + 32 200 GB SSD; 16 InfiniBand ports per pair

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

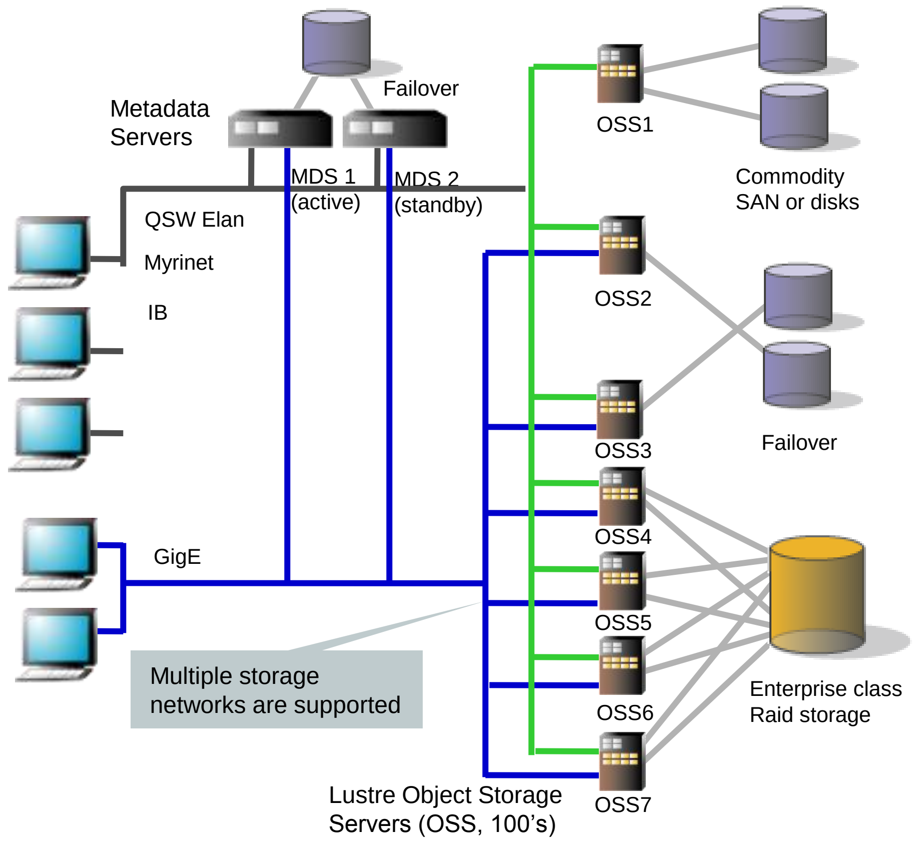


PanFS at LANL


IO Nodes in each compute cluster route between HSN and 10GE
1 Director, 10 OSD each chassis
104 chassis in largest single system, divided over 12 subnets (lanes)
10 GE
10 GE
...
10 GE
PaScalBB
12 switches
RoadRunner
TLCC
Cielo

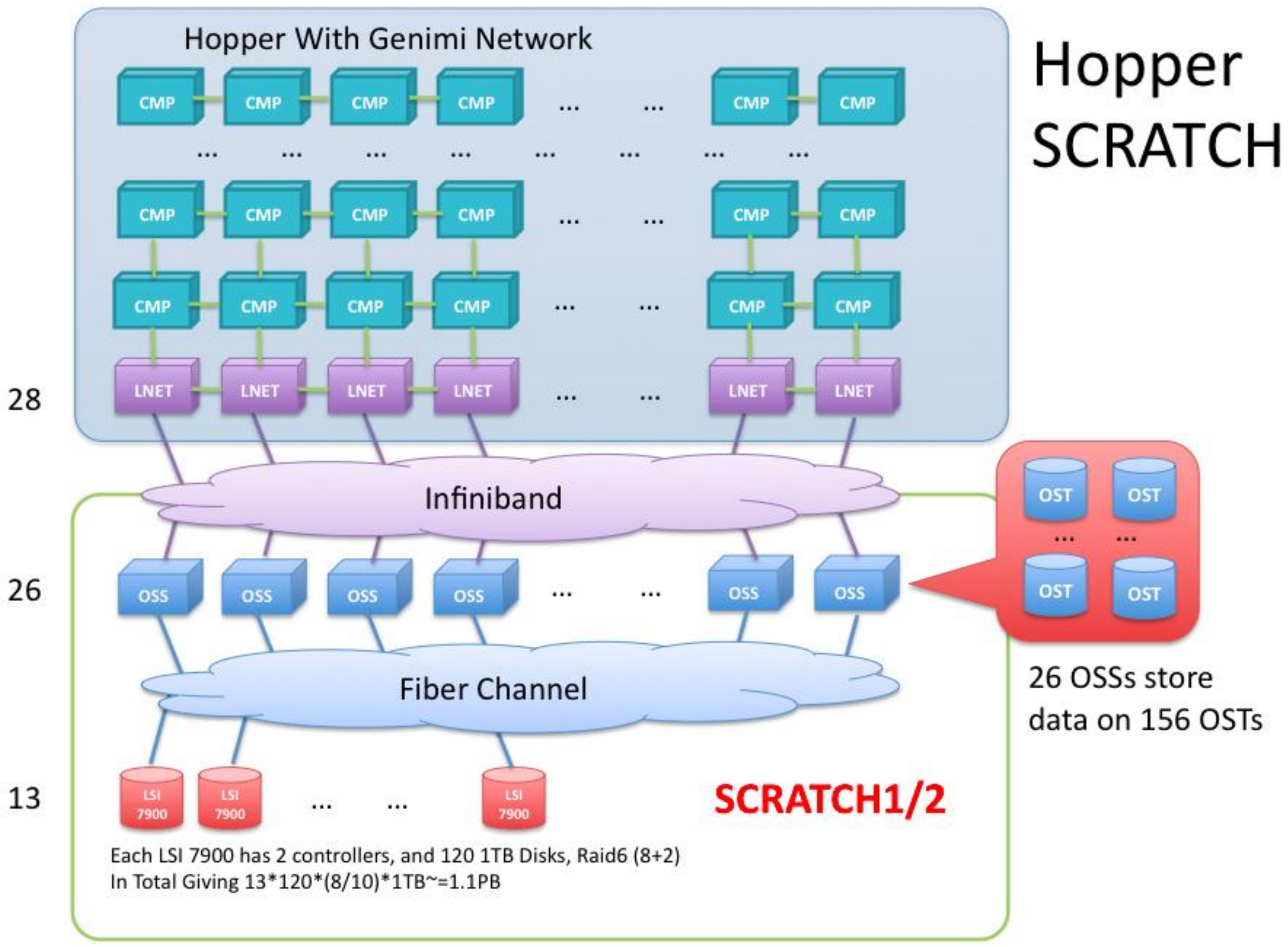
Lustre

- 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 material from www.lustre.org and various talks

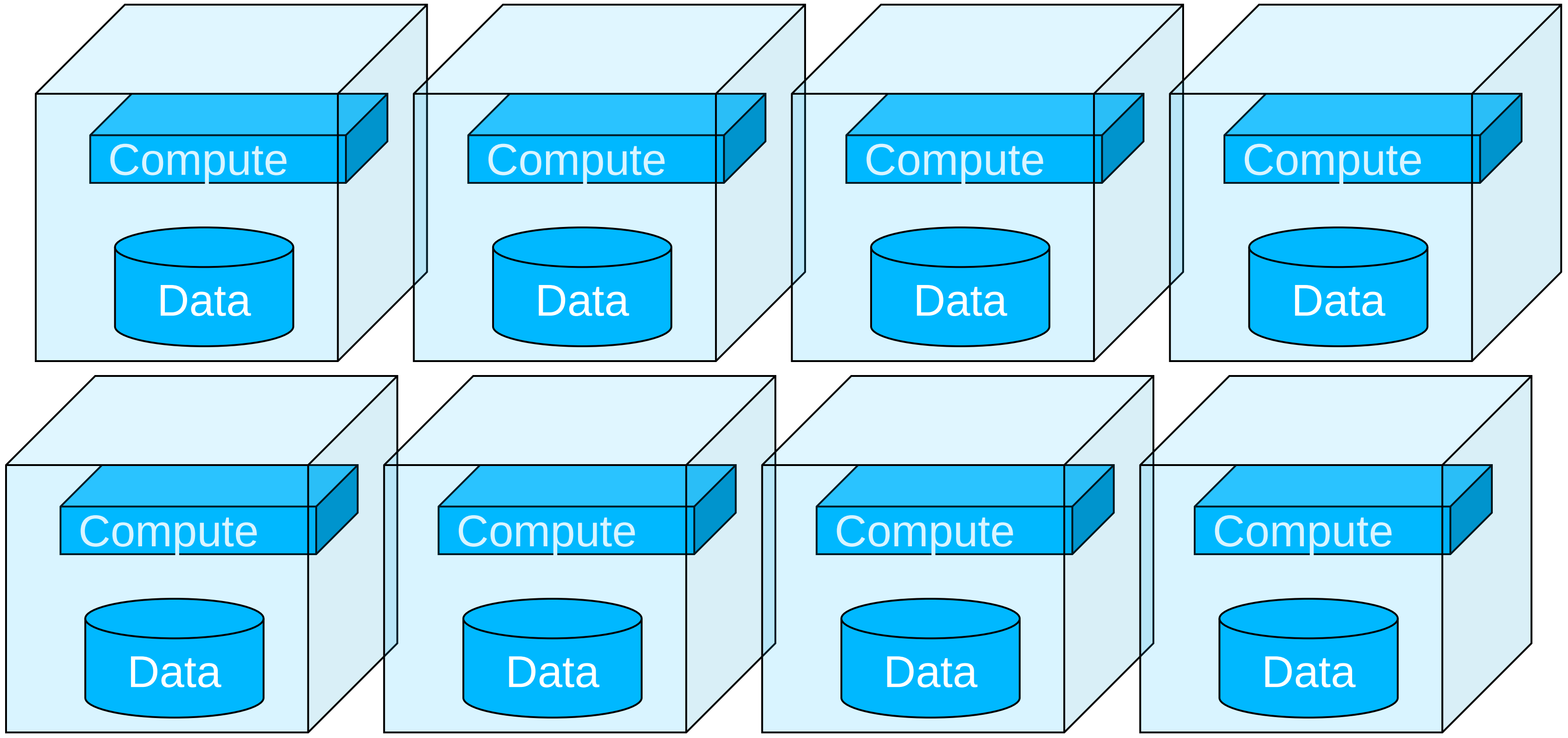
Lustre file system on Hopper


Hopper With Genimi Network
CMP
LNET
Infiniband
OSS
OST
Fiber Channel
LSI 7900
28
26
13
SCRATCH1/2
Hopper SCRATCH
26 OSSs store data on 156 OSTs
Each LSI 7900 has 2 controllers, and 120 1TB Disks, Raid6 (8+2)
In Total Giving 13*120*(8/10)*1TB~=1.1PB

Note: SCRATCH1 and SCRATCH2 have identical configurations.

Hadoop Environment

Data Node, Job Node in one box: lots of memory, local disks, ok network
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 often oversubscribed



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
- Single Name Node keeps all metadata in main memory
- Non-POSIX semantics
  - Access via programming library
- 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
- Hadoop is open source implementation
  - Google has its own proprietary implementations
  - Hadoop job scheduler, HDFS file system, Zookeeper configuration management, Cassandra bigtables, many more

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

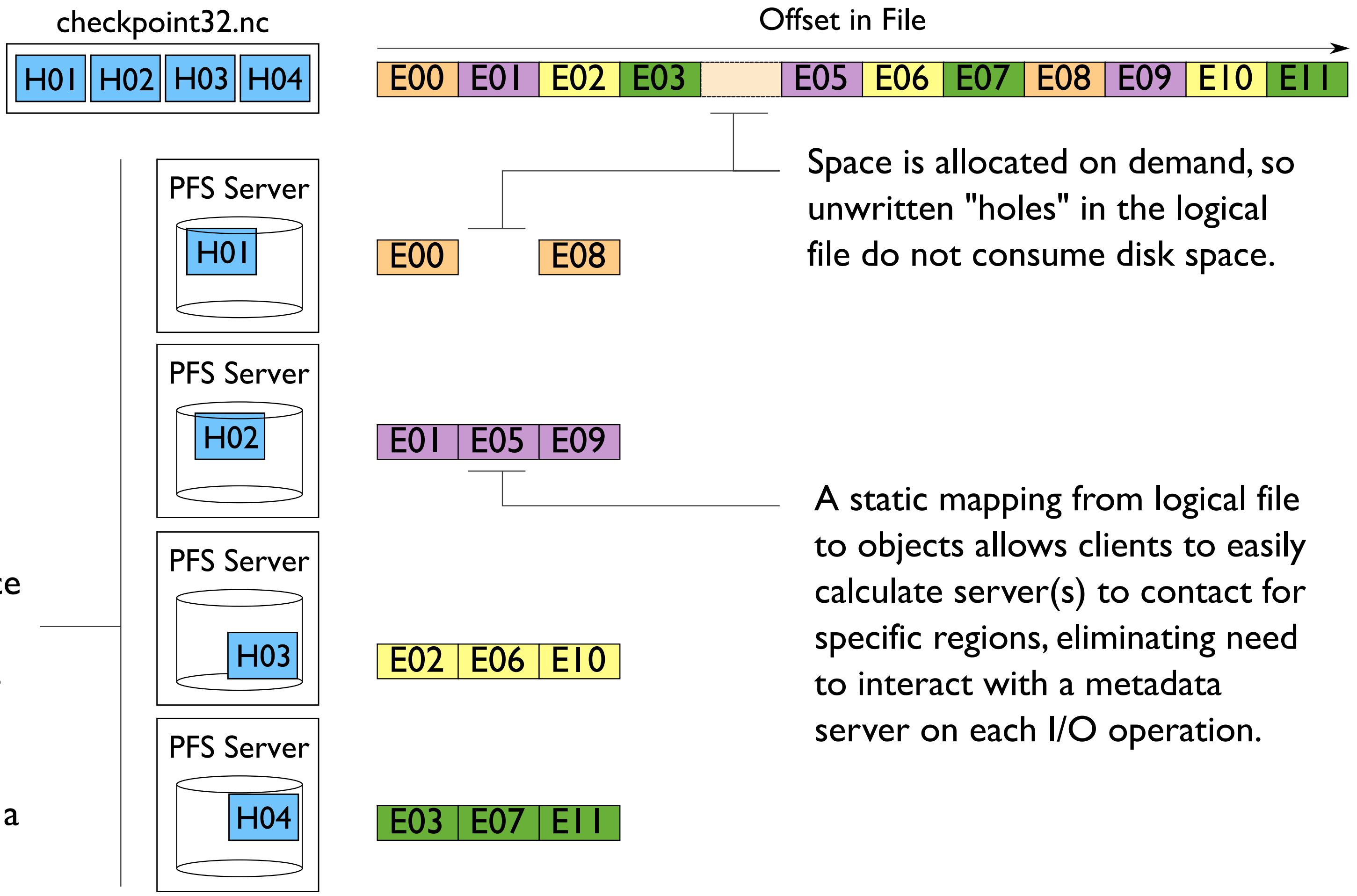
- Delegate block management to “object server”
  - Panasas, Lustre, PVFS, HDFS
  - I/O server uses local file system to store a chunk of a file
    - Panasas OSDFS, Lustre ext4, PVFS (any), HDFS (any)
- Distribute block management via locking
  - GPFS, GFS2
  - Nodes have their own part of the block map
- There are lots of blocks to manage
  - 8 billion 512B sectors on a 4T disk
  - 40 million 4K pages on a 40G SSD

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.

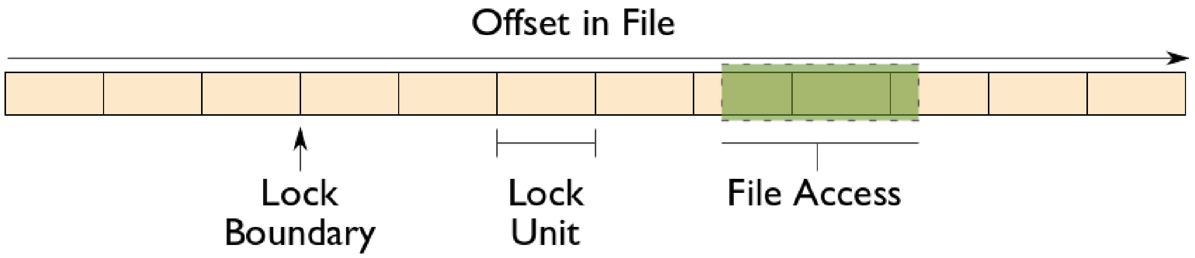
A static mapping from logical file to objects allows clients to easily calculate server(s) to contact for specific regions, eliminating need to interact with a metadata server on each I/O operation.

Locking in Parallel File Systems

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

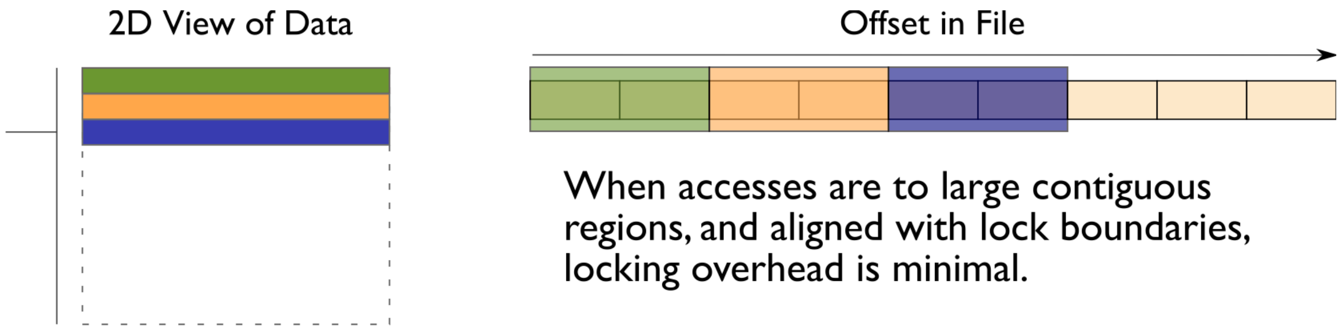
- Files are broken up into lock units
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access

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



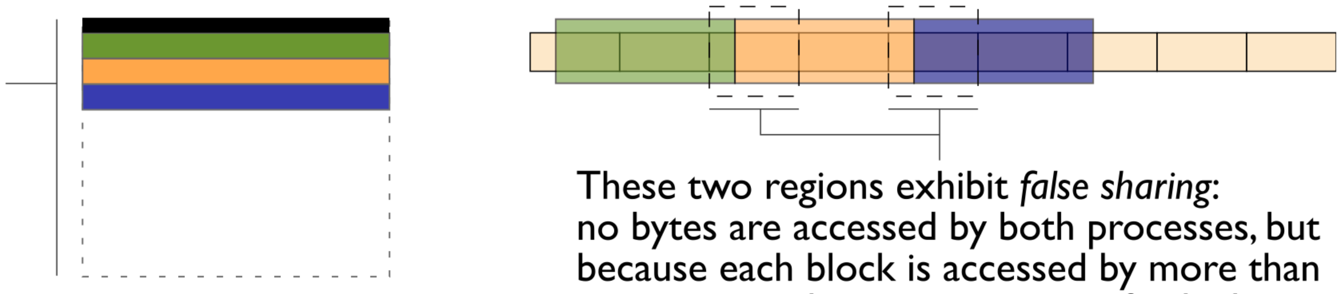
Locking and Concurrent Access

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.



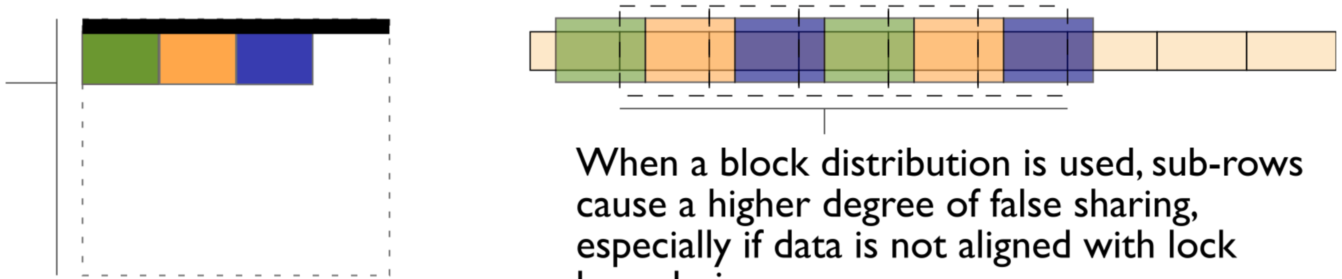
When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.

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.



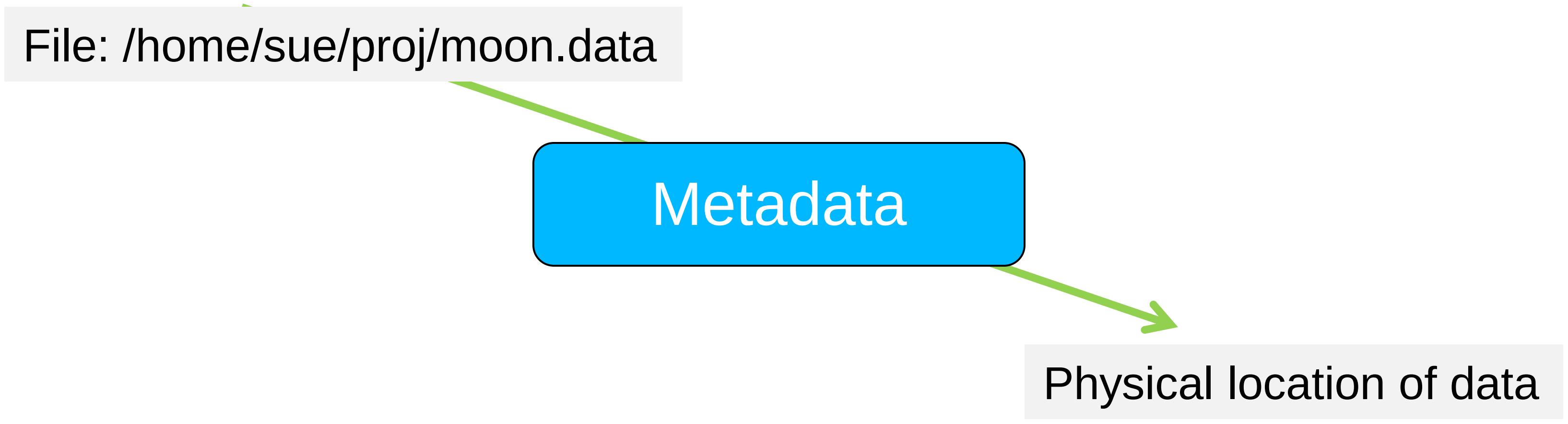
When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

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
  - Explicit barriers that flush data to storage and re-sync any caches with storage

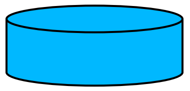
Meta Data

- 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

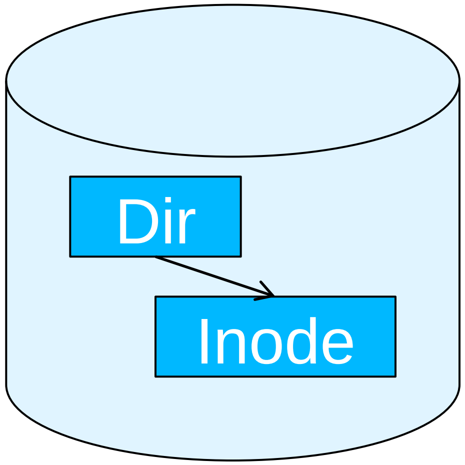


File Create on Local File System

- 3 logical I/Os
  - Journal update
  - Directory insert
  - Inode update
- Performance determined by journal updates
  - Or lack there of
  - Details vary among systems



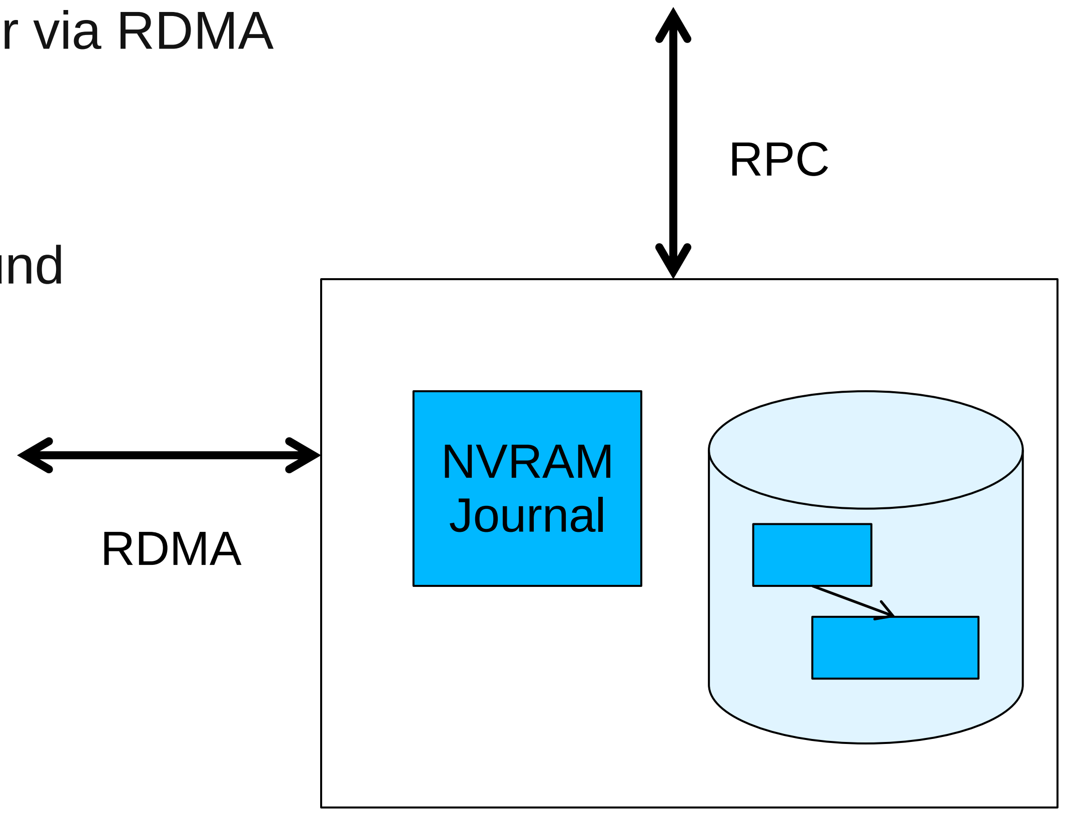
Fast Journal device (SSD)



Inode	Name
0017	Fred
2981	Yoshi
7288	Racheta

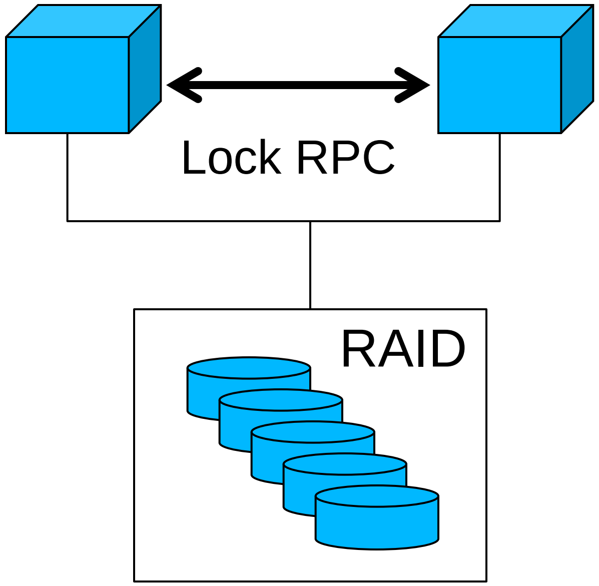
File Create on NFS Server

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



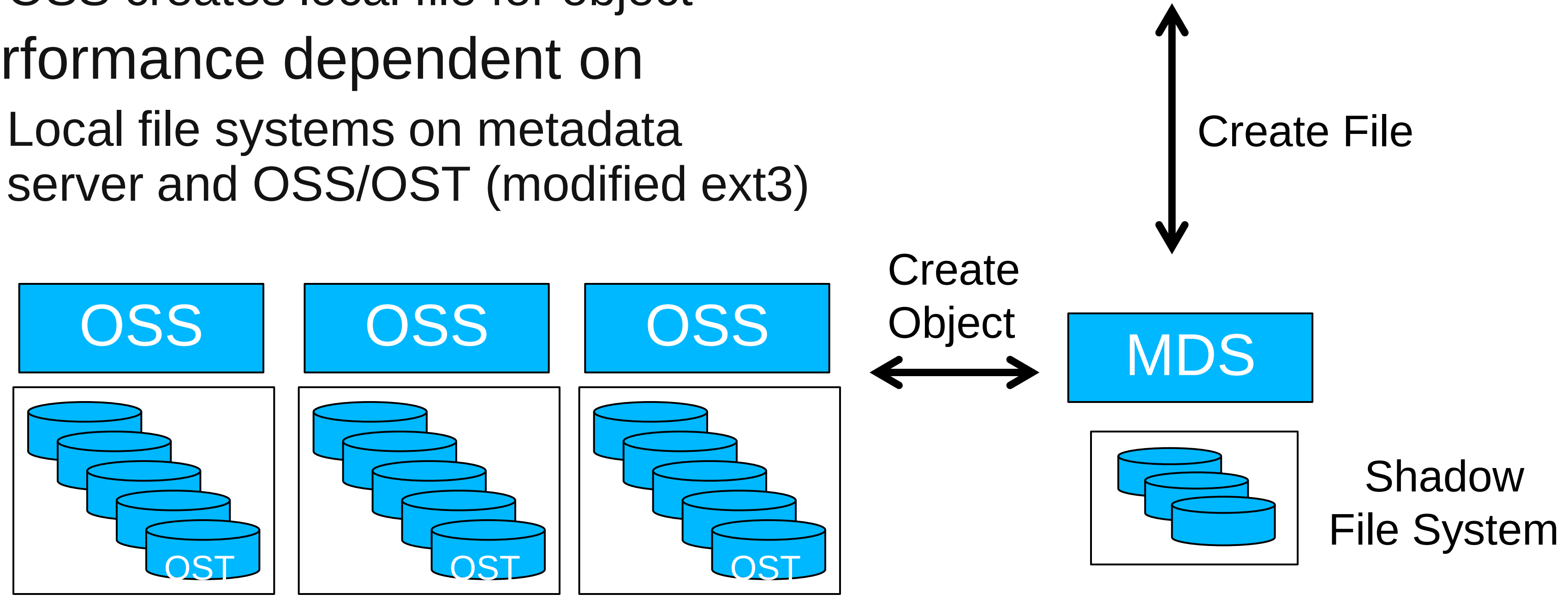
File Create on SAN FS

- 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 caches lock ownership
  - SAN I/O



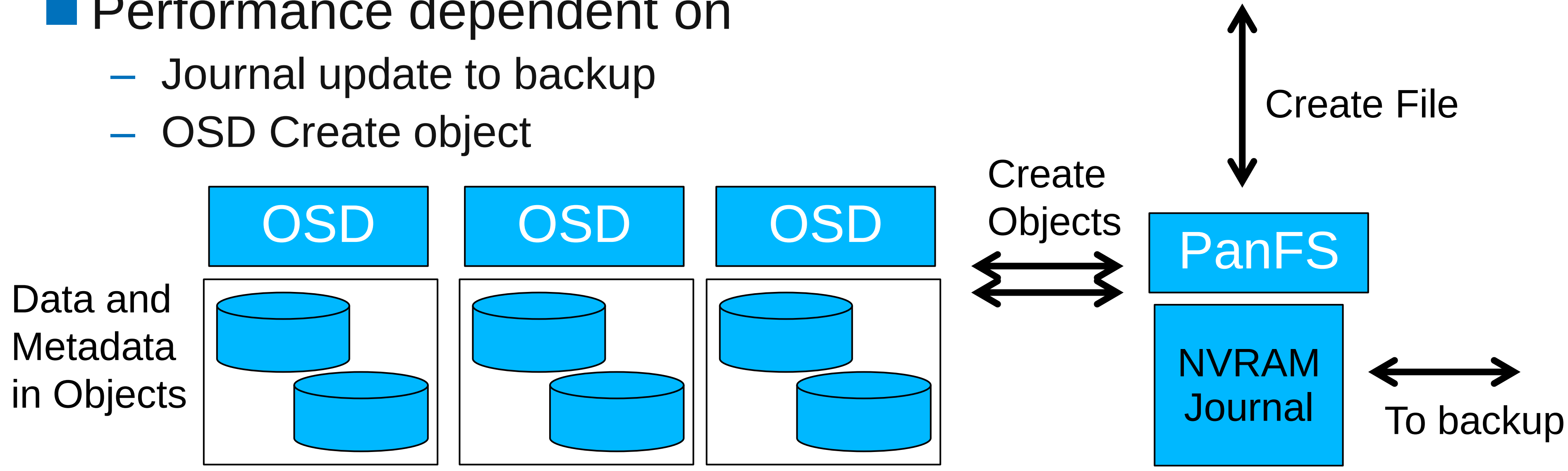
File Create on Lustre

- 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 (modified ext3)



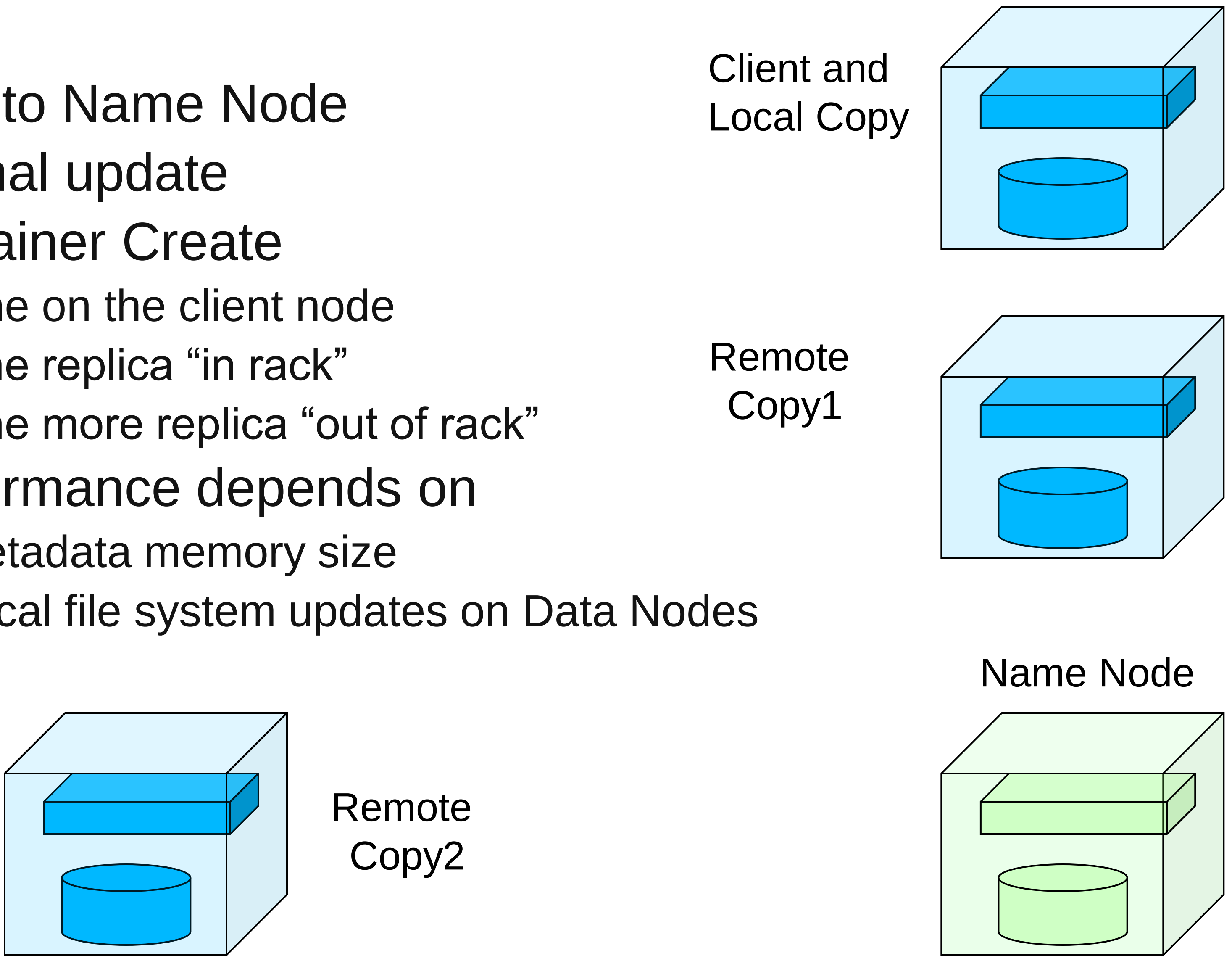
File Create on PanFS

- 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

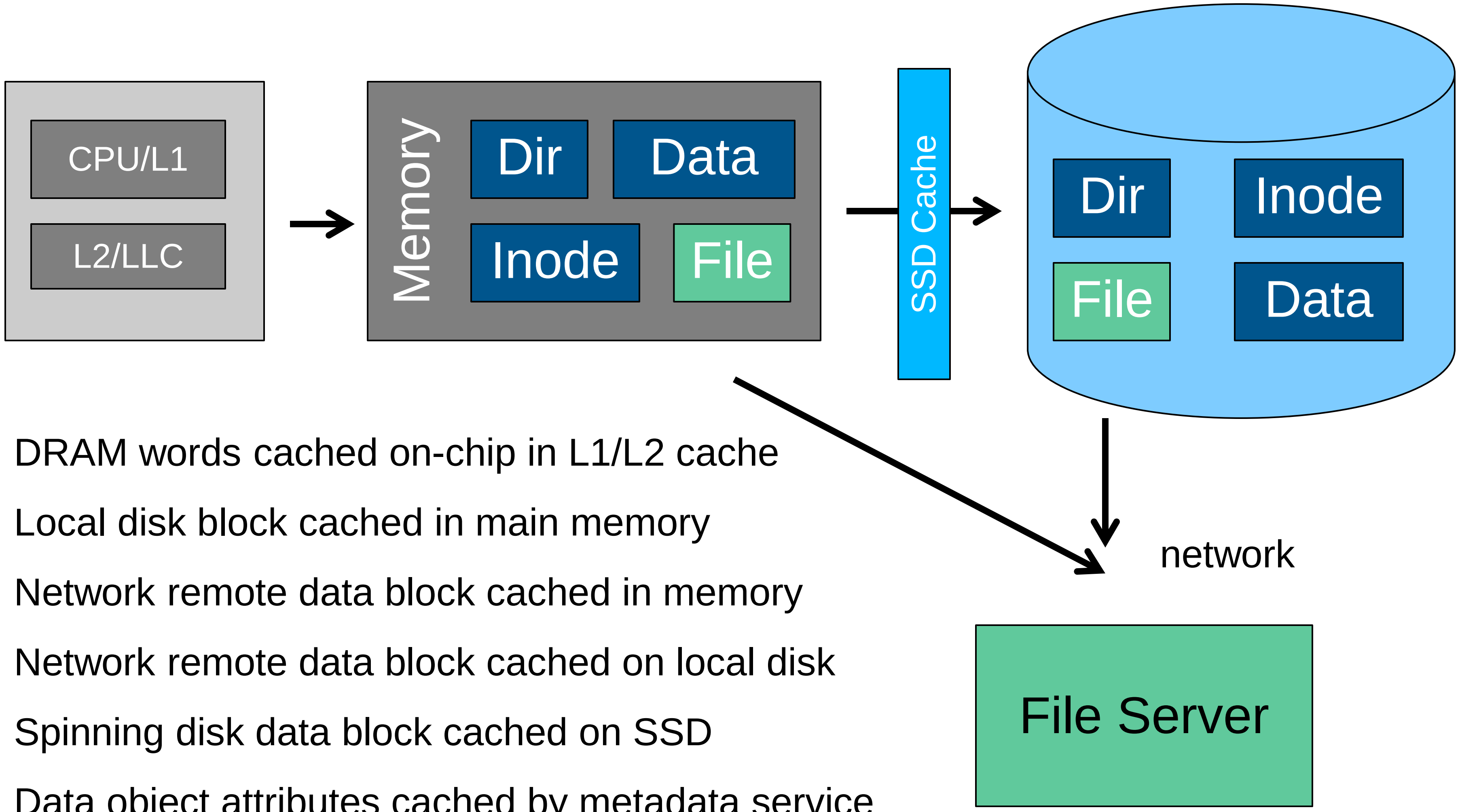


File Create on HDFS

- RPC to Name Node
- Journal 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



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 RAID or Triplication
- 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

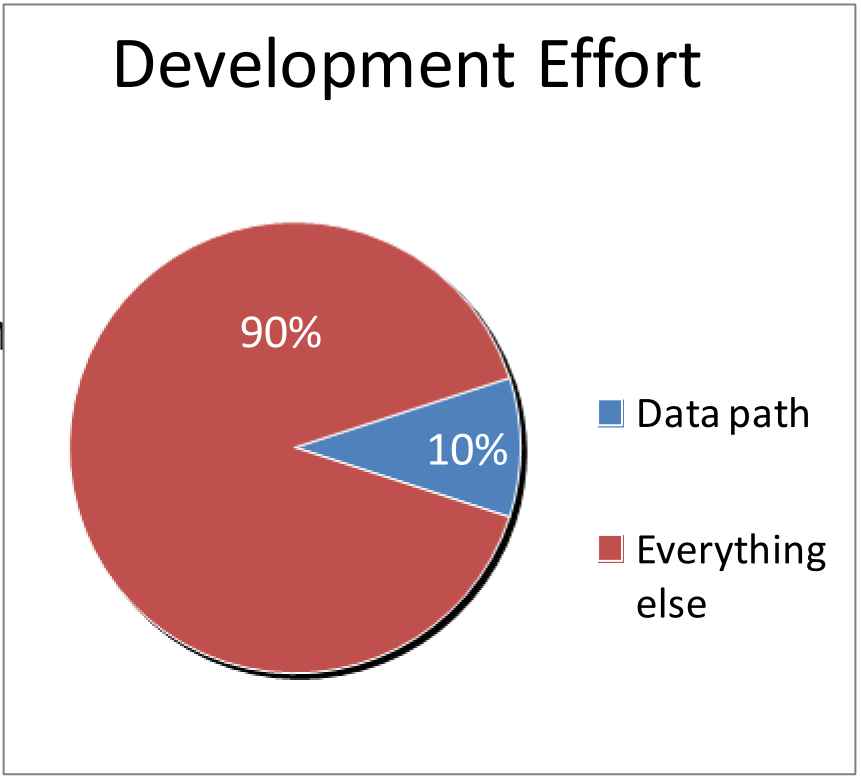
Design Comparison

	GPFS	HDFS	Panasas	Lustre
Block mgmt	**Shared block map**	Object based	Object based	Object based
Metadata location	**FS Disk Structures**	**Name Node**	**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

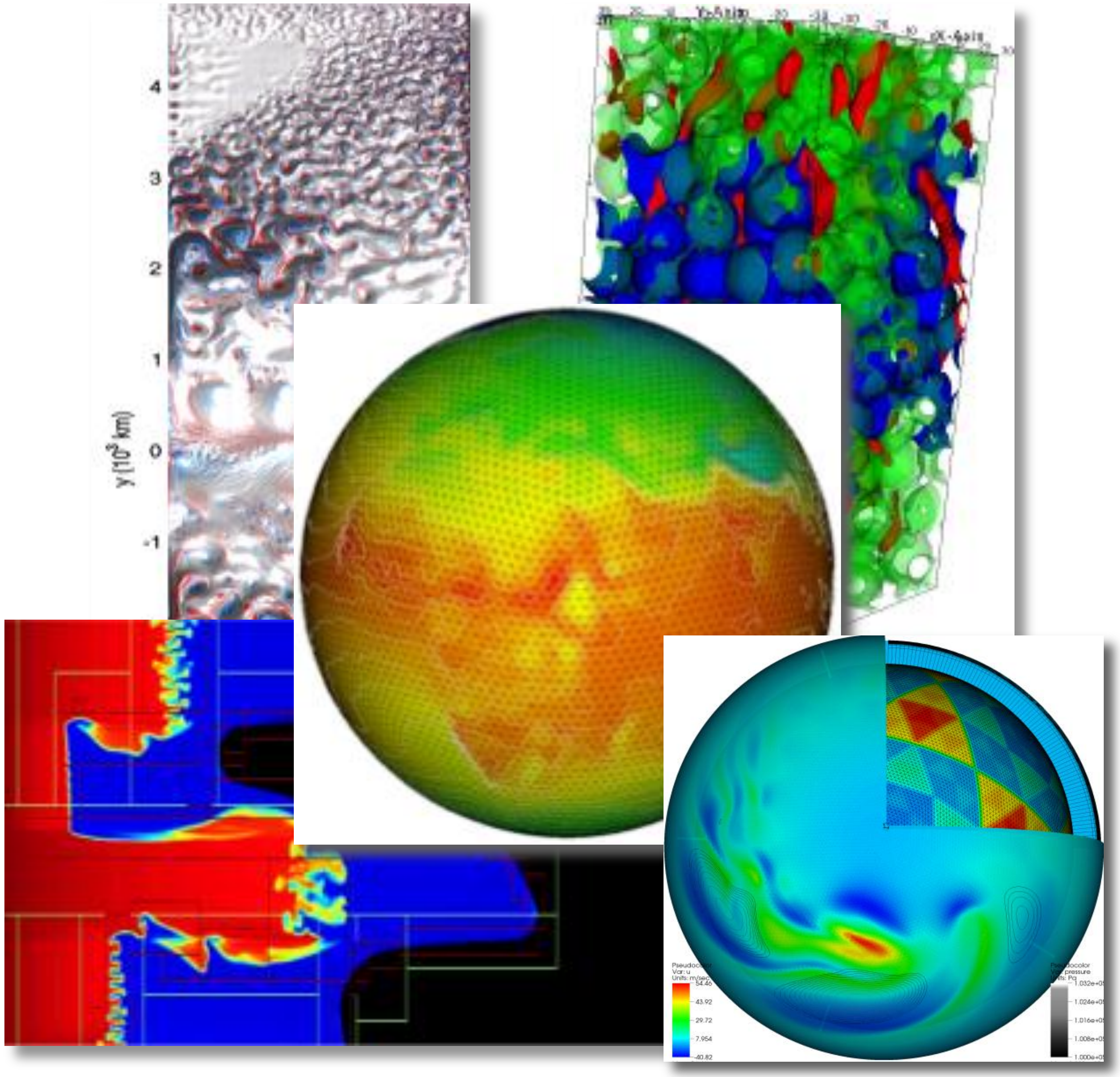
Workloads and User Wish list

Why you might need to do I/O

- Checkpoint/Restart files
  - System or node could fail; protect your application so you don't have to start from the beginning
  - Need to run longer than wall clock time allows
- Analysis files
- Visualization files
- Out-of-core algorithms

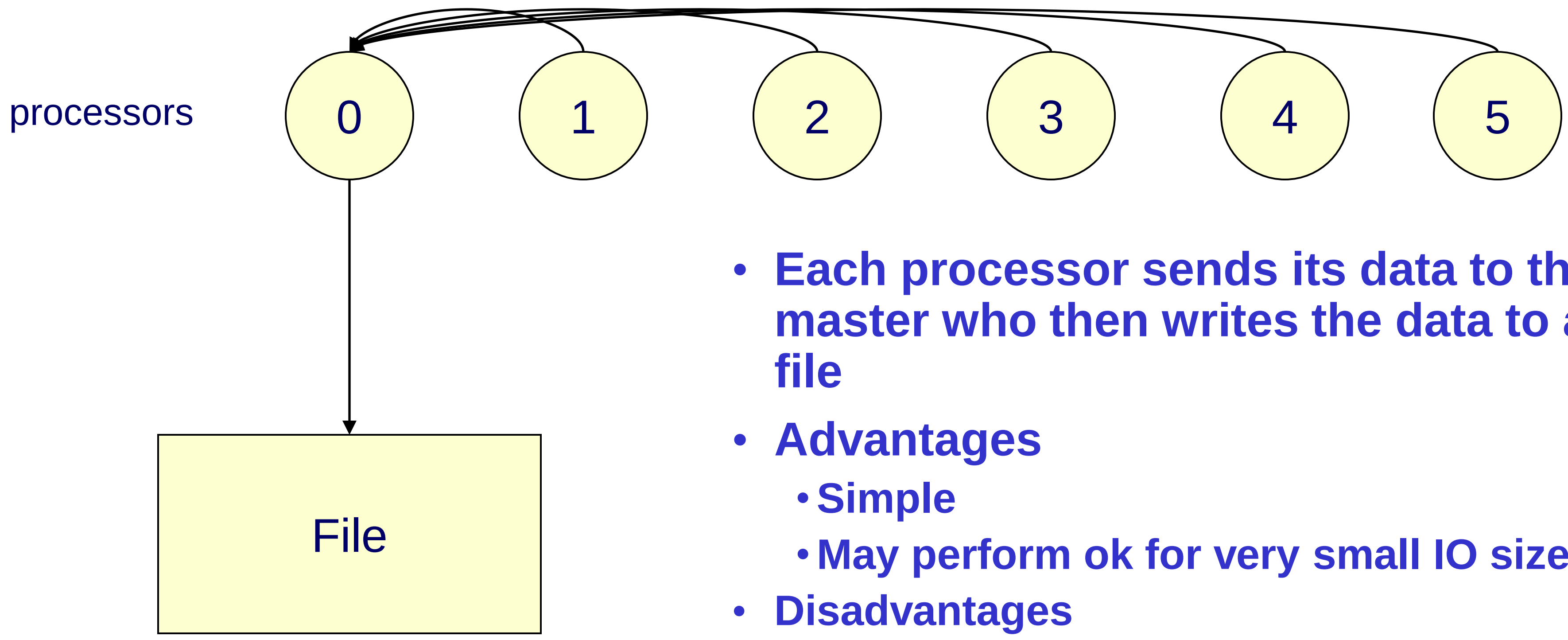
Why is Scientific I/O so difficult?

- 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



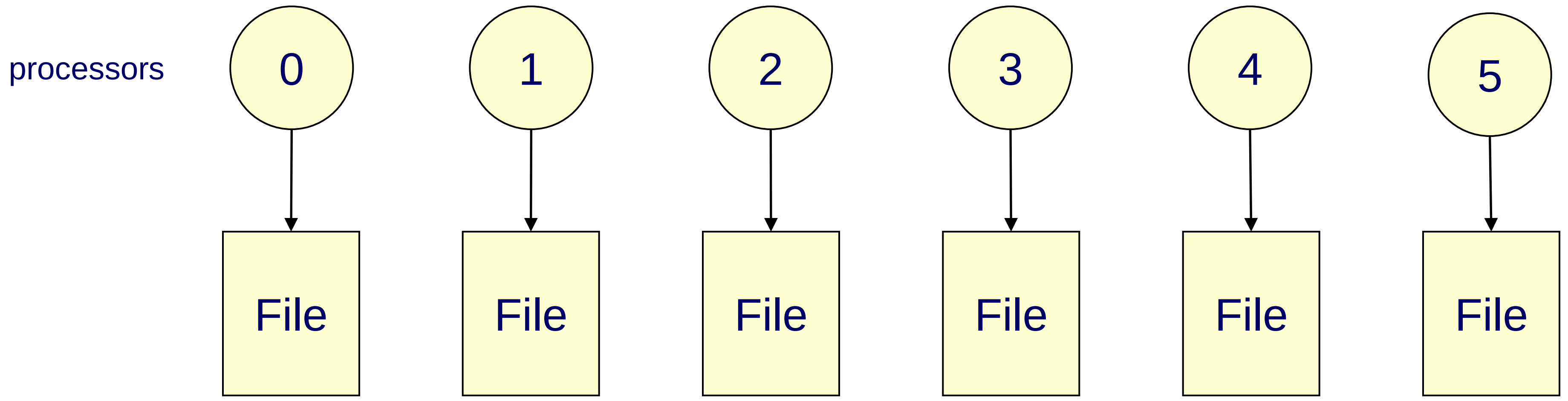
Images from David Randall, Paola Cessi, John Bell, T Scheibe

Serial I/O



- **Each processor sends its data to the master who then writes the data to a file**
- **Advantages**
  - **Simple**
  - **May perform ok for very small IO sizes**
- **Disadvantages**
  - **Not scalable**
  - **Not efficient, slow for any large number of processors or data sizes**
  - **May not be possible if memory constrained**

Parallel I/O Multi-file

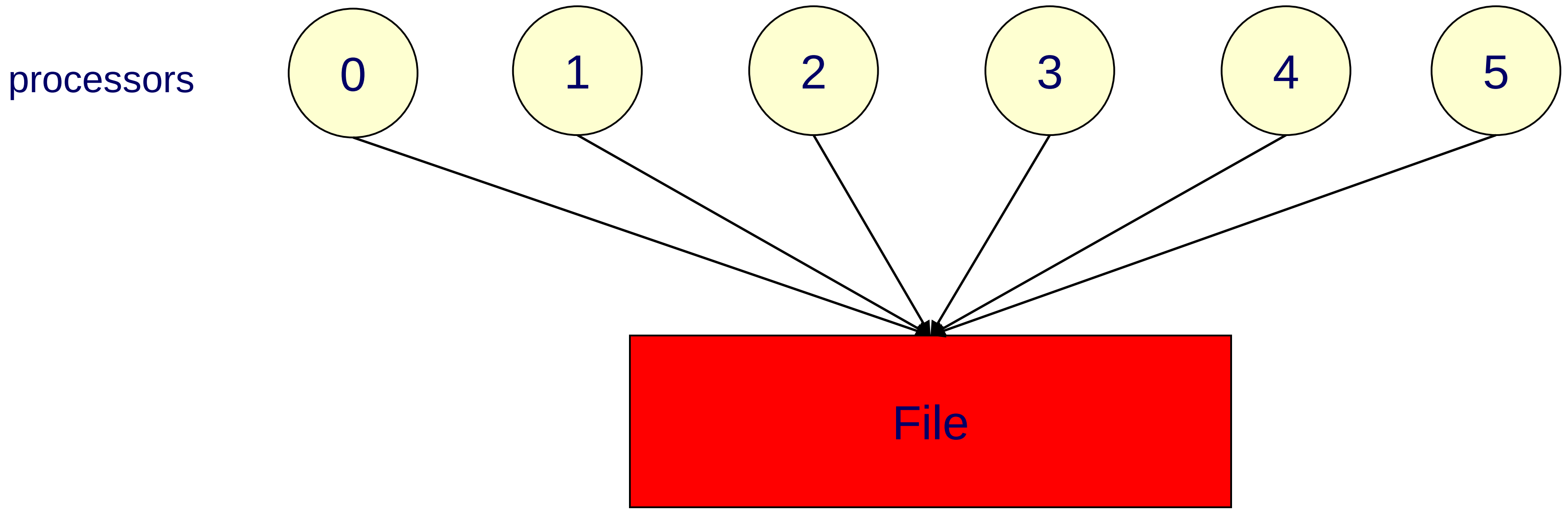


- Each processor writes its own data to a separate file
- Advantages
  - Simple to program
  - Can be fast -- (up to a point)
- Disadvantages
  - Can quickly accumulate many files
  - Hard to manage
  - Requires post processing
  - Difficult for storage systems, HPSS, to handle many small files
  - Can overwhelm the file system with many writers

Flash Center IO Nightmare…

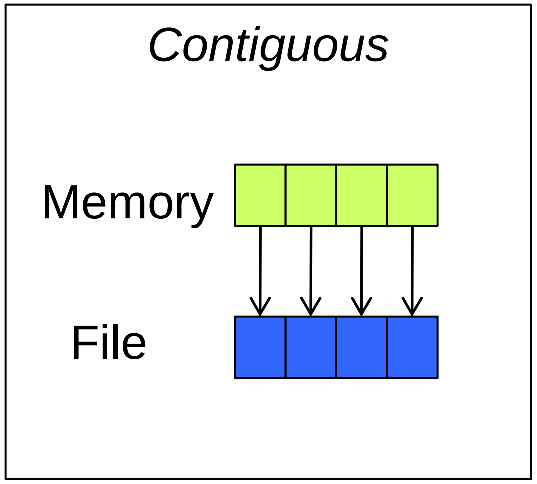
- Large 32,000 processor run on LLNL BG/L
- Parallel IO libraries not yet available
- Intensive I/O application
  - checkpoint files .7 TB, dumped every 4 hours, 200 dumps
    - used for restarting the run
    - full resolution snapshots of entire grid
  - plotfiles - 20GB each, 700 dumps
    - coarsened by a factor of two averaging
    - single precision
    - subset of grid variables
  - particle files 1400 particle files 470MB each
- 154 TB of disk capacity
- 74 million files!
- Unix tool problems
- Took 2 years to sift though data, sew files together

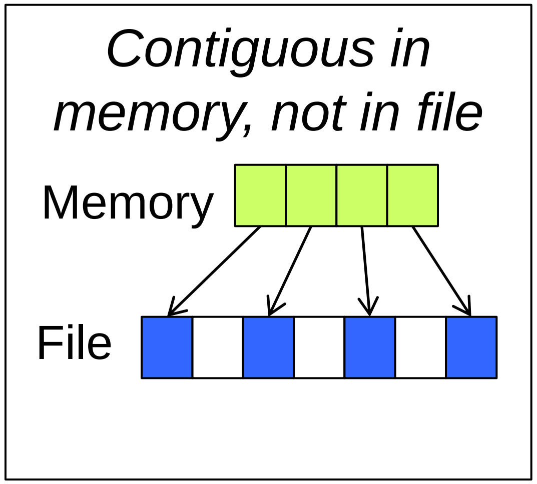
Parallel I/O Single-file

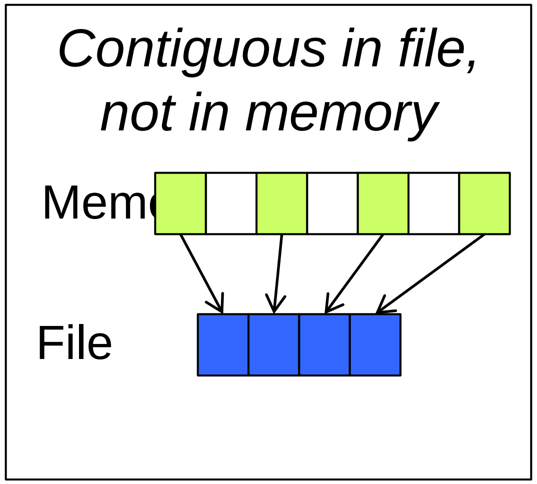


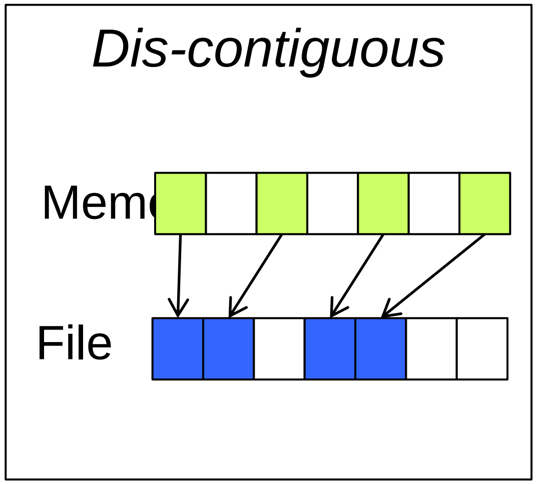
- Each processor writes its own data to the same file using MPI-IO mapping
- Advantages
  - Single file
  - Manageable data
- Disadvantages
  - Shared files may not perform as well as one-file-per-processor models

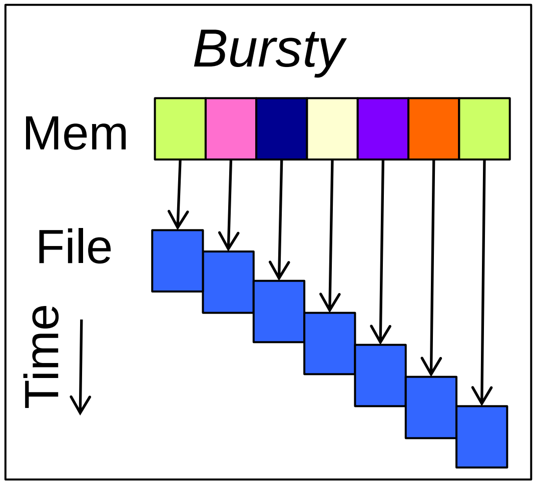
Access Patterns

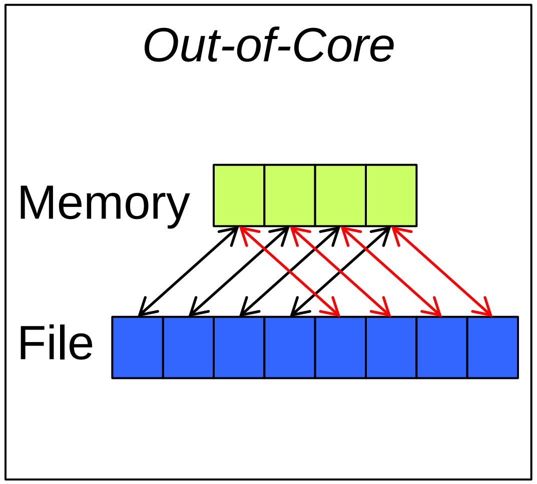












Parallel I/O:
A User Perspective

- Wish List
  - Write data from multiple processors into a single file
  - File can be read in the same manner regardless of the number of CPUs that read from or write to the file. (eg. want to see the logical data layout… not the physical layout)
  - Do so with the same performance as writing one-file-per-processor
  - And make all of the above portable from one machine to the next
- Inconvenient Truth: Scientists need to understand about I/O in order to get good performance

Benchmarking

Goals for this section

- Introduce different kinds of I/O workloads
- Rules of thumb about performance
- Introduction to some standard benchmarks

Performance Basics

- I/O Patterns
  - Streaming (i.e., sequential)
    - Start to finish
  - Strided
    - 4, 8, 12, 16, …
  - Random
    - 97, 32, 5, 1354, 1464, 765, …
- Sharing Patterns
  - File-per-process
  - Shared-file

- Metadata Operations
  - File Create
  - File Delete
  - Set Attributes
  - Get Attributes
  - Directory lookups
  - Directory tree walks
  - File updates (i.e., writes)

- What do you mean by "I/O", by "Meta Data ops"?

Workloads

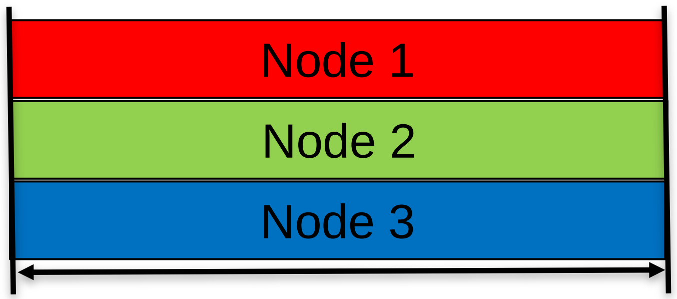
- Streaming I/O
  - Single client, one or more streams per client
  - Scaling clients
  - System throughput, scaling with system size (or not)
- Random I/O
  - Dependent on caching and drive seek performance
- Metadata
  - Create/Delete workloads
  - File tree walk (scans)
- MPI IO
  - Coordinated opens
  - Shared output files
- Interprocess Communication
  - Producer/consumer files
  - Message drop
  - Atomic record updates
- Small I/O
  - Small whole file operations
  - Small read/write operations

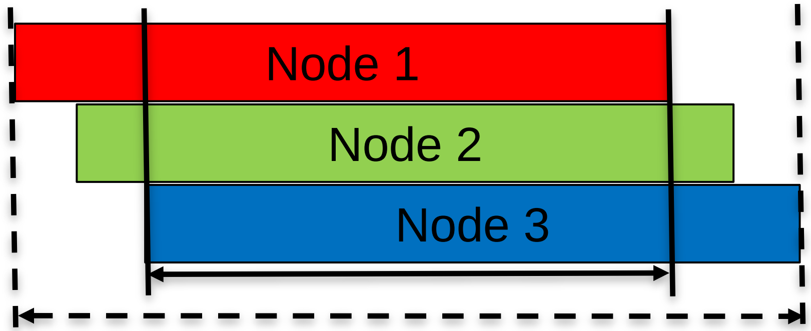
Performance Features

- Streaming I/O
  - Read-ahead
  - Write buffering
    - is there a battery?
- Random I/O
  - Large data cache
  - SSD
- Metadata
  - Asynchronous file delete
  - Stat pre-fetching to optimize tree-walk
  - NVRAM journal updates
- MPI IO
  - FS-specific Hints
- Interprocess Communication
  - Please use MPI
- Small I/O
  - Aggressive metadata cache
  - Large data cache
  - SSD

Benchmarking Pitfalls

- Not measuring what you think you are measuring
  - Most common with microbenchmarks
  - For example, measuring write or read from cache rather than to storage
  - Watch for "faster than the speed of light" results
- Multi-client benchmarks without synchronization across nodes
  - Measure aggregate throughput only when all nodes are transferring data
  - Application with I/O barrier may care more about when last node finishes





- Benchmark that does not model application workload
  - Different I/O size & pattern, different file size, etc.

Analyzing Results

- Sanity-checking results: what is the "speed of light"
- Large sequential accesses
  - Readahead can hide latency
  - 7200 RPM SATA 60-100 MB/sec/spindle
  - 15000 RPM FC 100-170 MB/sec/spindle
  - SAS SSD 100-400 MB/sec/device
  - SSD (PCIe) 1+ GB/sec/slot
- Small random access
  - Seek + rotate limited
  - Readahead rarely helps (and sometimes hurts)
  - 7200 RPM SATA avg access 15 ms, 75-100 ops/sec/spindle
  - 15000 RPM FC avg access 6 ms, 150-300 ops/sec/spindle
  - SSD (Sata) avg access <.1ms, 20K-50K ops/sec/device
  - SSD (PCIe) avg access X us, 785K ops/sec

Rule of Thumb 1

- Bigger is better
  - Large files, large transfers, large numbers of clients generally result in larger aggregate performance
- But bigger isn’t always necessary
  - 64K may be just as good as 4MB if read ahead is working
  - Write buffering for less-than full stripe writes may or may not hurt depending on the quality of the RAID controller

Rule of Thumb 2

- Alignment of data access can be critical
  - Sub-block, non-aligned accesses require pre-fetching and buffering, at the minimum
  - Additional locking overhead between threads can add further overhead
- Middleware can help
  - MPI mechanisms shuffle data among data collectors and go to the file system in large, aligned chunks

Rule of Thumb 3

- Sharing has a cost
  - Sharing between clients requires coordination by the file system, and therefore has a cost
  - E.g., 1024 different clients creating a file in the same directory
- Sharing is important
  - Some systems work hard to make sharing “perfect” so that clients anywhere in the network have an up-to-date view of files and their data
  - NFS caching semantics cause tiny delays in visibility of new files in a directory
- Corollary
  - A dedicated “one host, one wire, one disk” local file system can be optimized in ways that are too expensive for shared network storage systems to match

Rule of Thumb 4

- Use MPI for Message Passing
  - Some applications use the file system as a convenient interprocess communication mechanism
    - *They should use a real message passing infrastructure instead*
  - This works OK at small scale with a single NFS filer, but has horrible scaling properties because it involves the metadata server in every message exchange
- MPI IO has plenty of legitimate uses
  - Coordinated I/O by many processes in one application

Rule of Thumb 5

- Measure the Application
  - Measuring your own application performance is the best test
  - Different systems have different optimizations/bottlenecks
- Corollary
  - Parallel systems are excellent at finding serialization
- Examples
  - Single inode lock on directory limits create rate
  - Small I/O may create multi-cycle RAID I/O

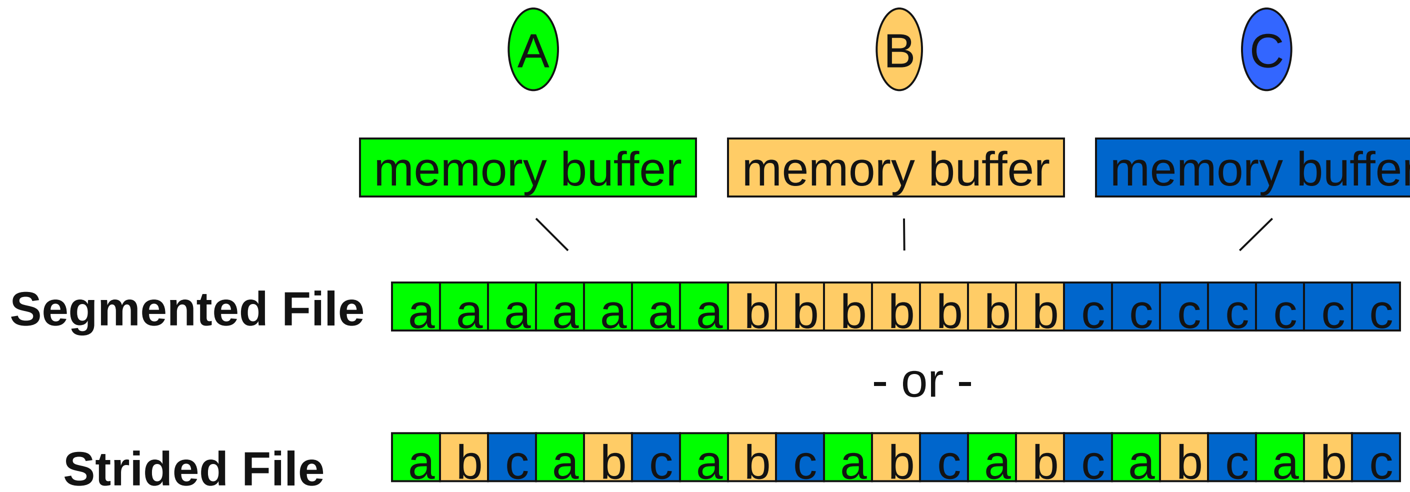
IOR: File System Bandwidth

- 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
    ```

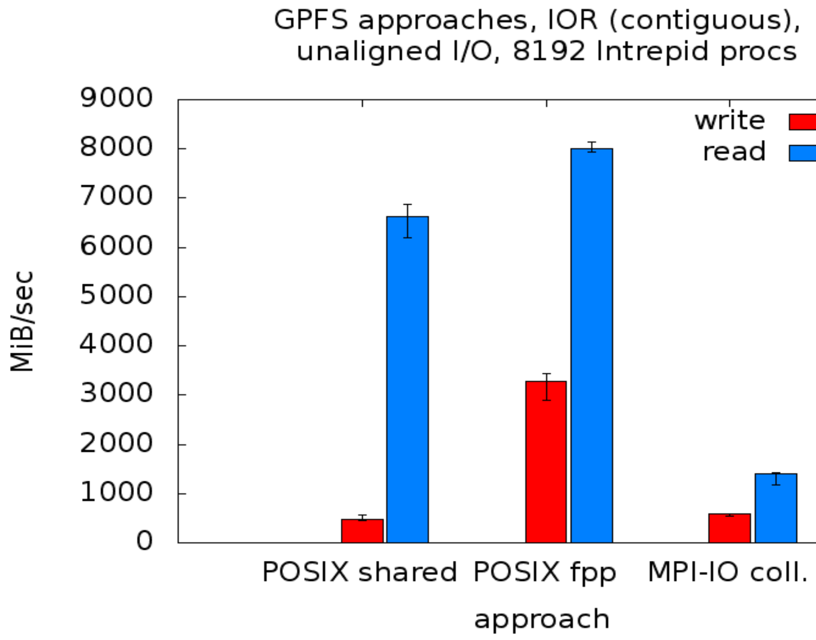
IOR Access Patterns for Shared Files


A
B
C
memory buffer
memory buffer
memory buffer
Segmented File
a a a a a a a b b b b b b b c c c c c c c
- or -
Strided File
a b c a b c a b c a b c a b c a b c a b c

- 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 in this un-aligned workload hits POSIX
  - Communication (two-phase i/o optimization) hits MPI-IO
- Despite data management costs, file per process (fpp) extremely seductive
- IOR the beginning, not end, of journey towards understanding performance



Metadata Performance

- Storage is more than reading & writing
- Metadata operations change the namespace or file attributes
  - Creating, opening, closing, and removing files
  - Creating, traversing, and removing directories
  - “Stat”ing files (obtaining the attributes of the file, such as permissions and file size)
- Several use cases exercise metadata subsystems:
  - Interactive use (e.g. “ls -l”)
  - File-per-process POSIX workloads
  - Collectively accessing files through MPI-IO (directly or indirectly)

mdtest: Parallel Metadata Performance

- Measures performance of multiple tasks creating, stating, and deleting both files and directories in either a shared directory or unique (per task) directories
- Demonstrates potential serialization of multiple, uncoordinated processes for directory access
- Written at Lawrence Livermore National Laboratory
- MPI code, processes synchronize for timing purposes
- Three variations:
  - Each task creates 100 files in a unique subdirectory (-u option)

```
mdtest -d $DIR -n 100 -i 3 -N 1 -v -u
```

  - One task creates 6400 files in one directory (-c option)
  - Each task opens, removes its own

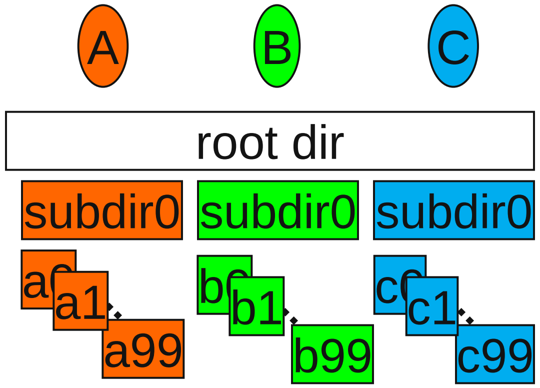
```
mdtest -d $DIR -n 100 -i 3 -N 1 -v -c
```

  - Each task creates 100 files in a single shared directory

```
mdtest -d $DIR -n 100 -i 3 -N 1 -v
```

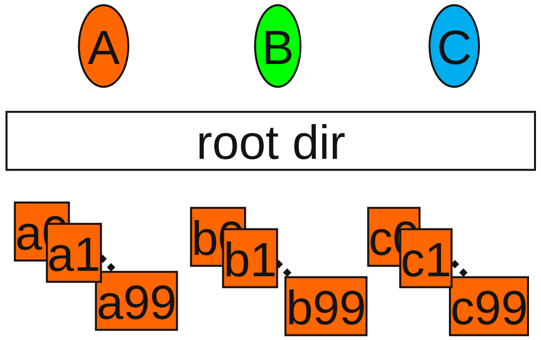
mdtest Variations

Unique Directory (-u)



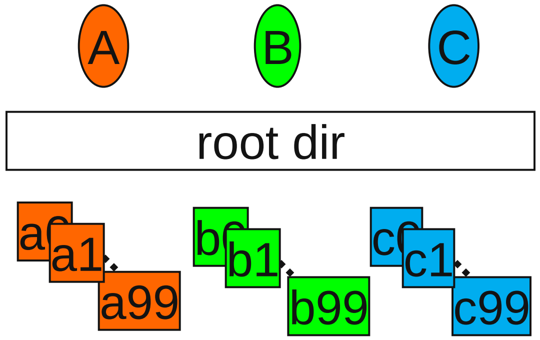
1) Each process (A, B, C) creates own subdir in root directory, then chdirs into it.
2) A, B, and C create, stat, and remove their own files in the unique subdirectories.

Single Process (-c)



1) Process A creates files for all processes in root directory.
2) Processes A, B, and C open, stat, and close their own files.
3) Process A removes files for all processes.

Shared Directory



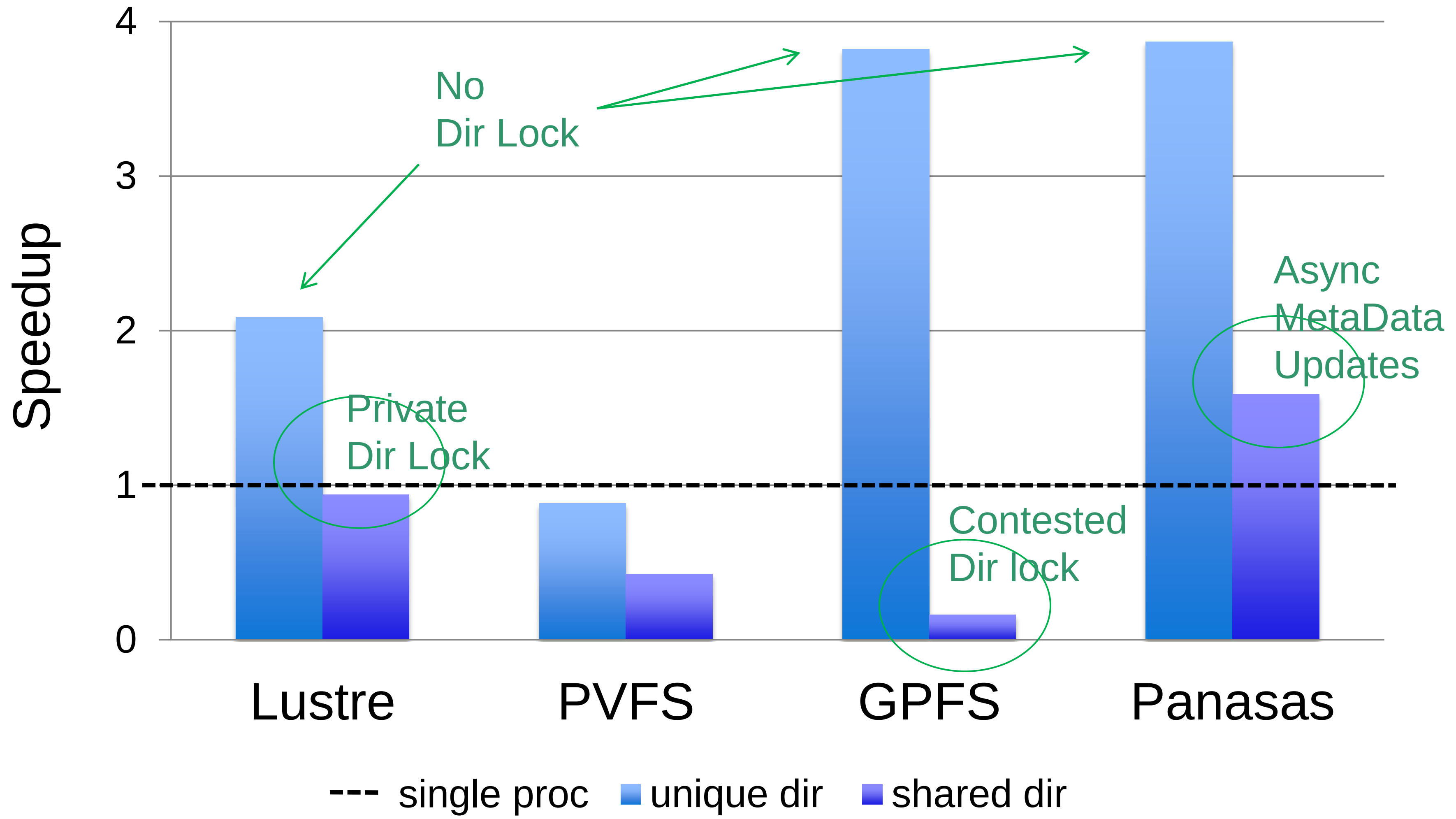
1) Each process (A, B, C) creates, stats, and removes its own files in the root directory.

Performance Disclaimer

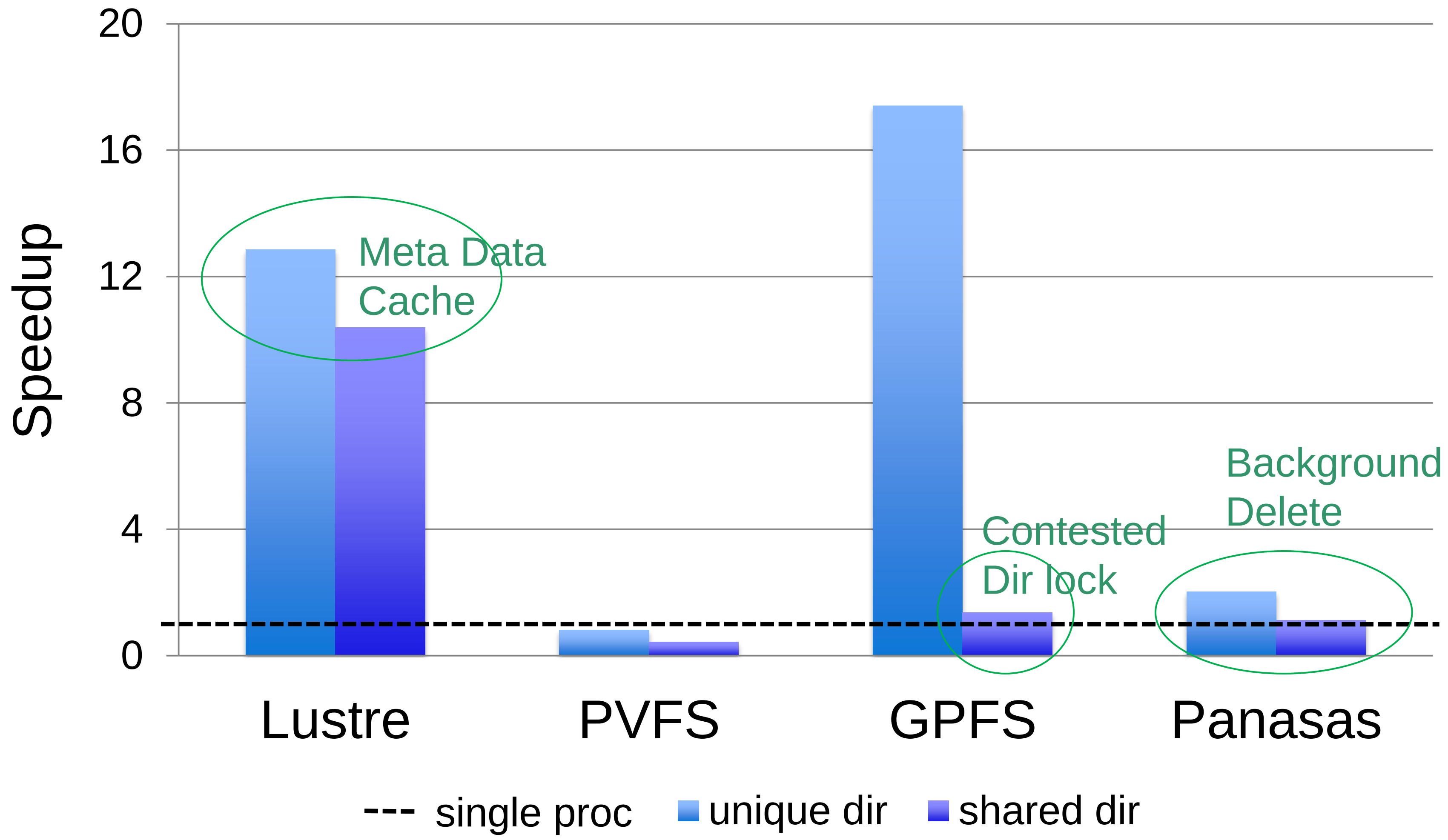
- The following relative results are shown to illustrate that different systems have different performance trade-offs
- The data is a few years old from a set of very different systems, so only self-relative results are shown
- Software updates and differences in hardware configurations will potentially make big differences

=> You should run tests on your own platform

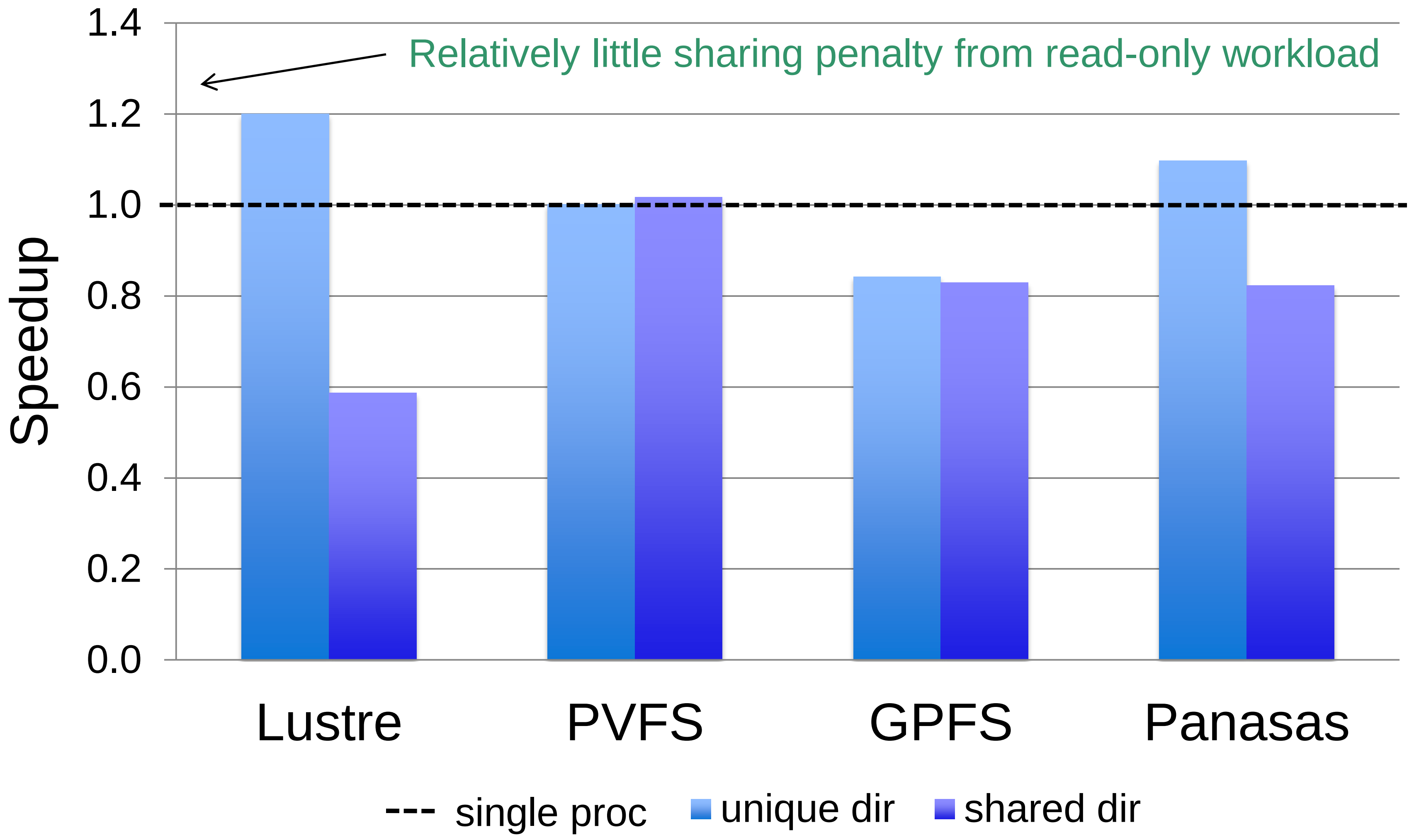
mdtest: Create File



mdtest: Remove File



mdtest: Stat File



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);
```

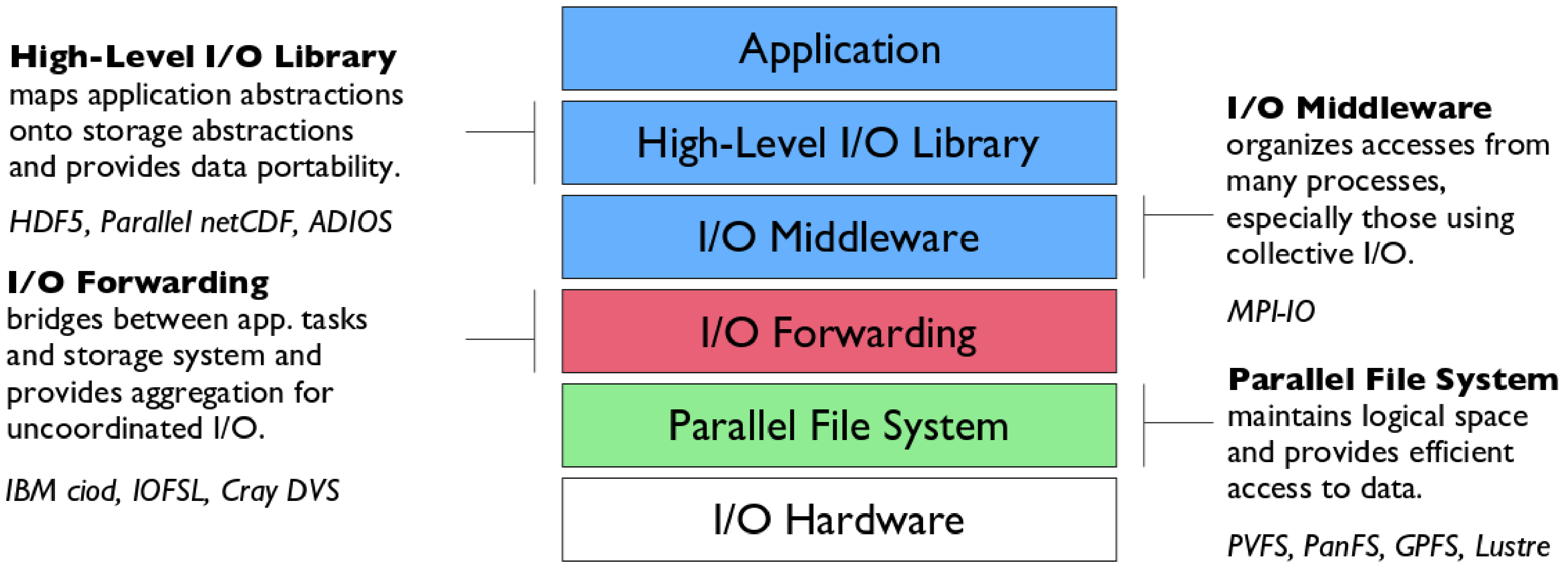
FORTRAN:

```
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

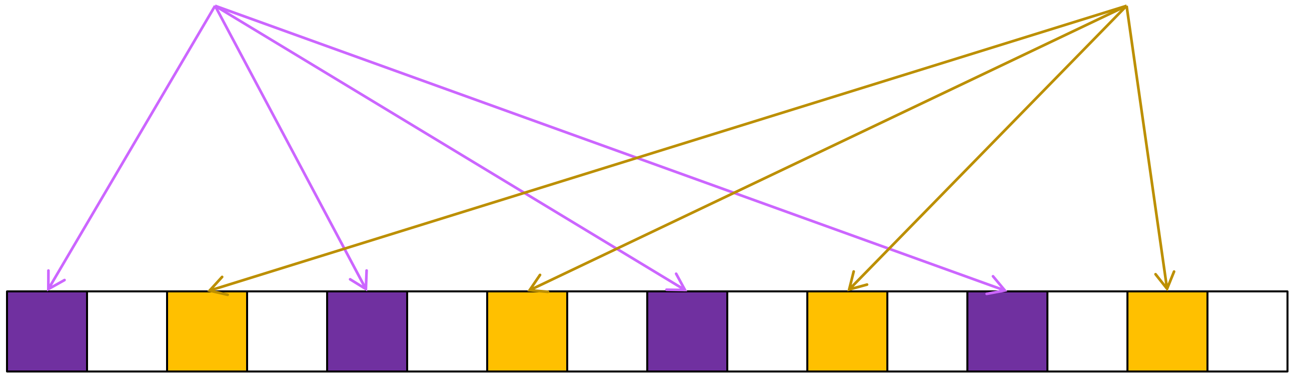
- I/O interface specification for use in MPI apps
- Data model is same as POSIX
  - Stream of bytes in a file
- Features:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - Fortran bindings (and additional languages)
  - System for encoding files in a portable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

Simple MPI-IO

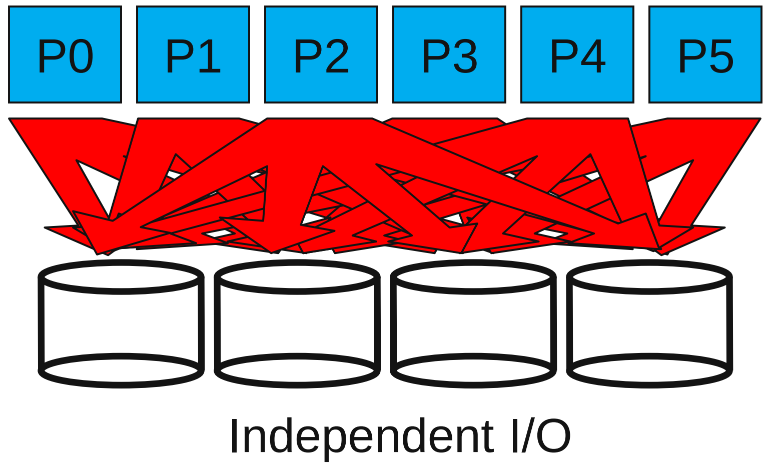
- Collective open: all processes in communicator
- File-side data layout with *file views*
- Memory-side data layout with *MPI datatype* passed to write

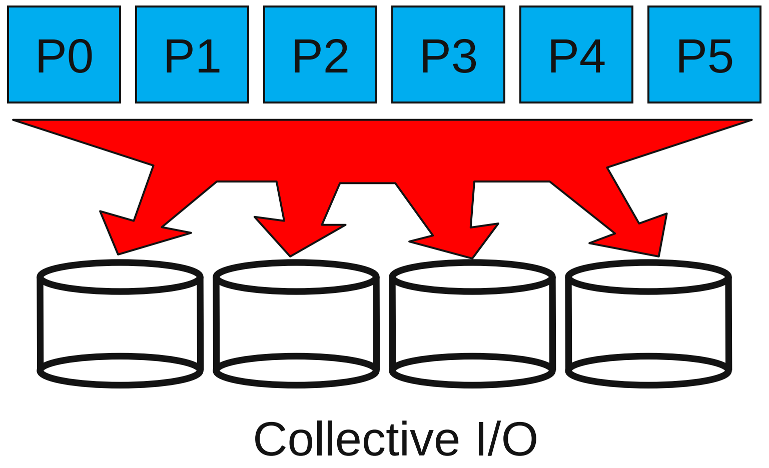
```
MPI_File_open(COMM, name, mode,
    info, fh);
MPI_File_set_view(fh, disp, etype,
    filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
    datatype, status);
```

```
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);
```



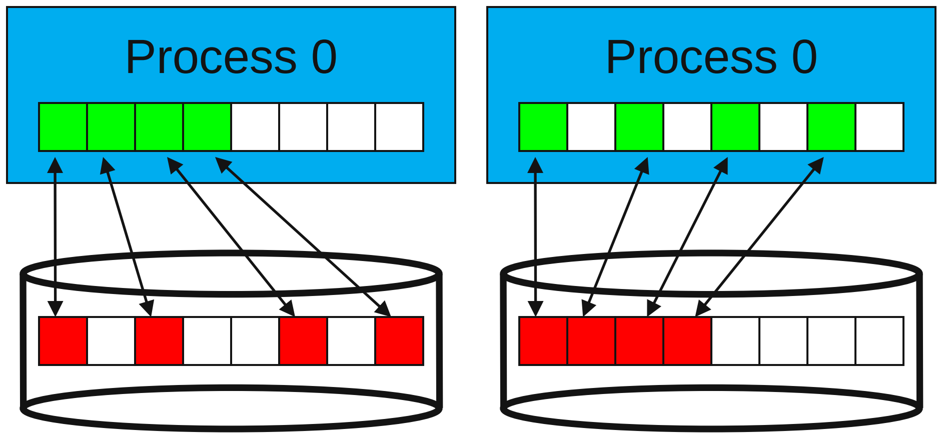
Independent and Collective I/O





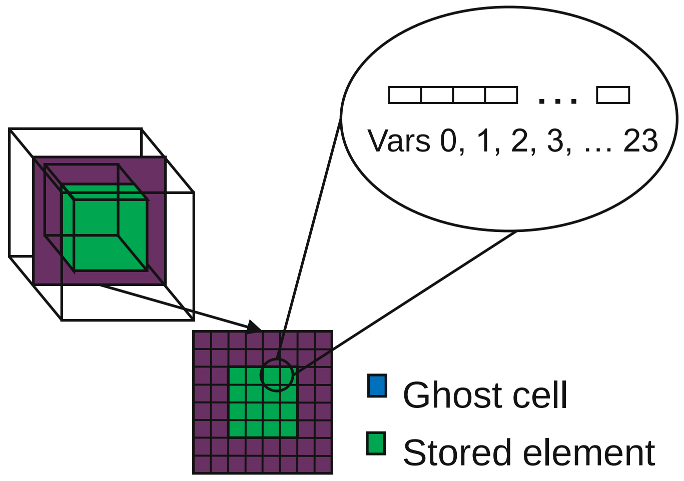
- Independent I/O operations specify only what a single process will do
  - Independent I/O calls do not pass on relationships between I/O on other processes
- Many applications have phases of computation and I/O
  - During I/O phases, all processes read/write data
  - We can say they are collectively 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

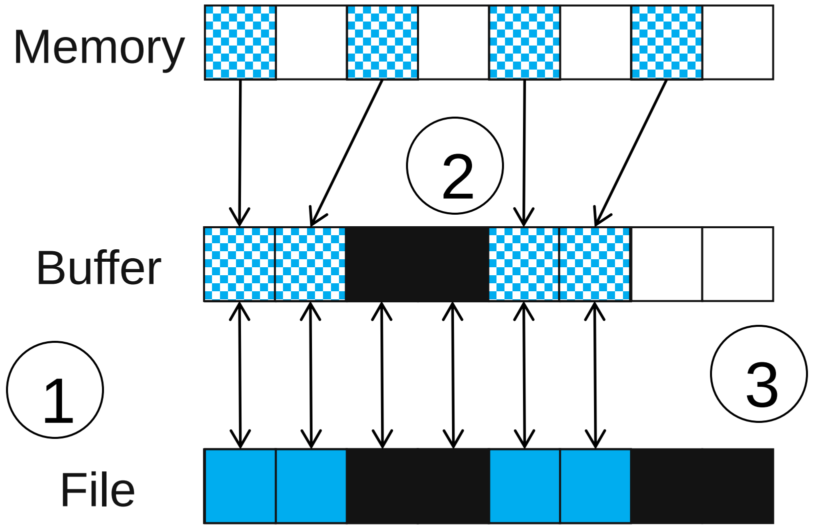


Extracting variables from a block and skipping ghost cells will result in noncontiguous I/O.

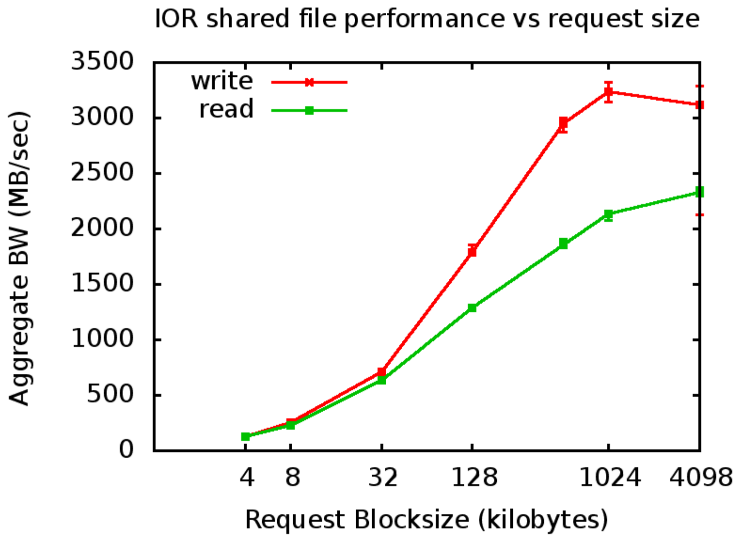
- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous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

Data Sieving Optimization

- One large request better than several smaller operations
- Data sieving for writes is more complicated than for reads
  - Must read the entire region first (1)
  - Then make changes in buffer (2)
  - Then write the block back (3)
- Requires locking in the file system
  - Can result in false sharing (interleaved access)
- PFS supporting noncontiguous writes is preferred

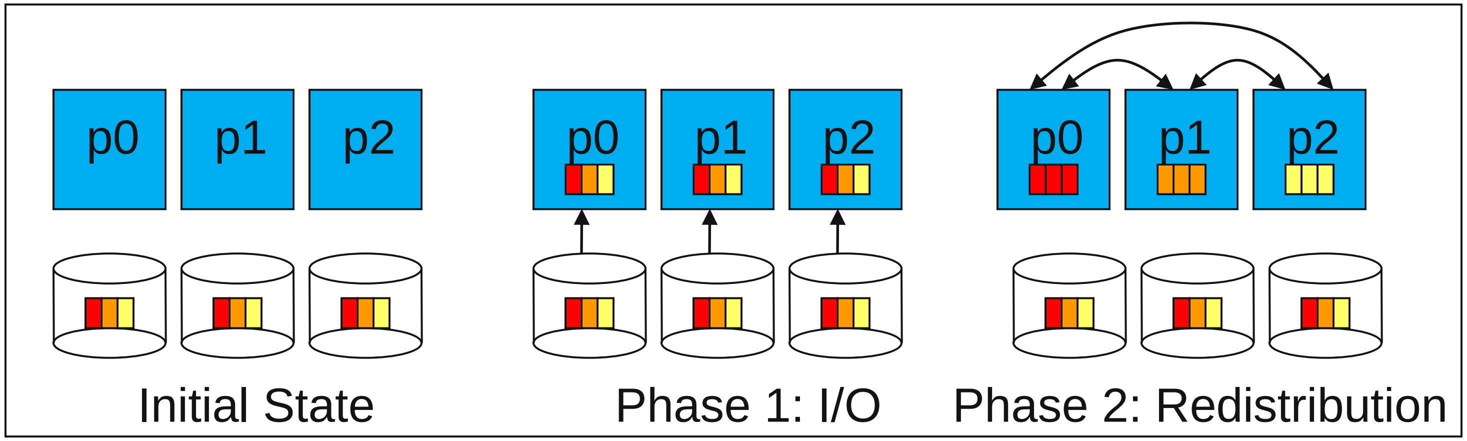


Data Sieving Write Transfers



8192 process IOR: 25x difference between small and large I/O req

Collective I/O and Two-Phase I/O

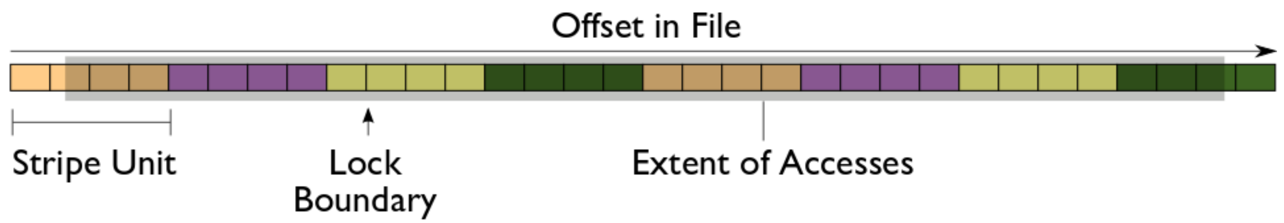


Two-Phase Read Algorithm

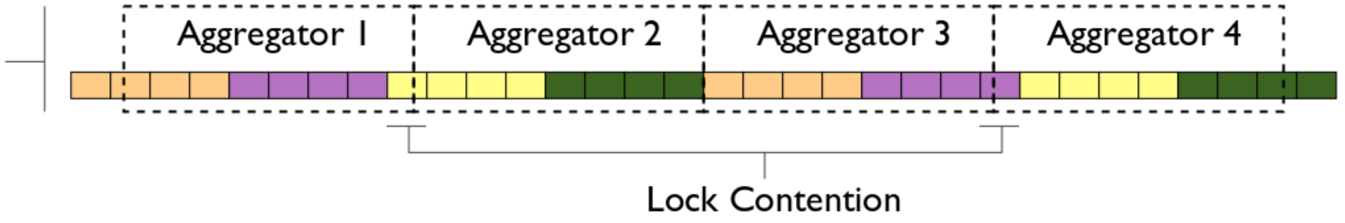
- Problems with independent, noncontiguous access
  - Lots of small accesses
  - Independent data sieving reads lots of extra data, can exhibit false sharing
- Idea: Reorganize access to match layout on disks
  - Single processes use data sieving to get data for many
  - Often reduces total I/O through sharing of common blocks
- Second "phase" redistributes data to final destinations
- Two-phase writes operate in reverse (redistribute then I/O)
  - Typically read/modify/write (like data sieving)
  - Overhead is lower than independent access because there is little or no false sharing
- Note that two-phase is usually applied to file regions, not to actual blocks

Two-Phase I/O Algorithms

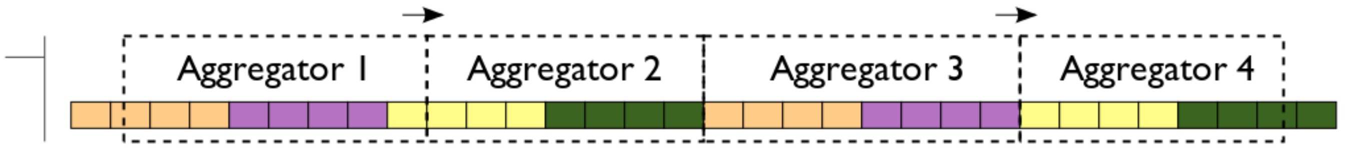
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



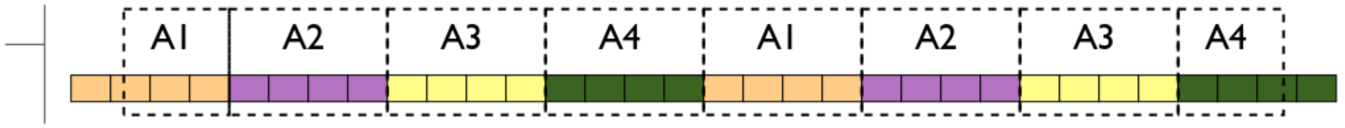
One approach is to evenly divide the region accessed across aggregators.



Aligning regions with lock boundaries eliminates lock contention.



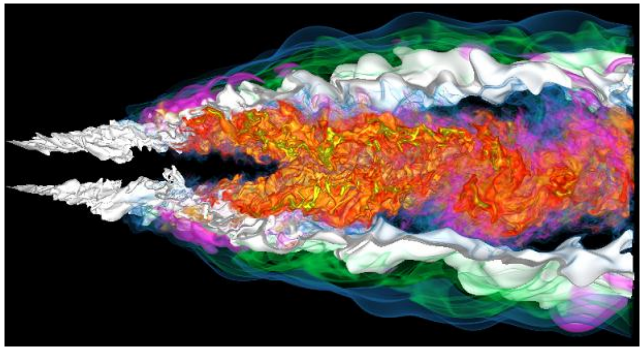
Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).

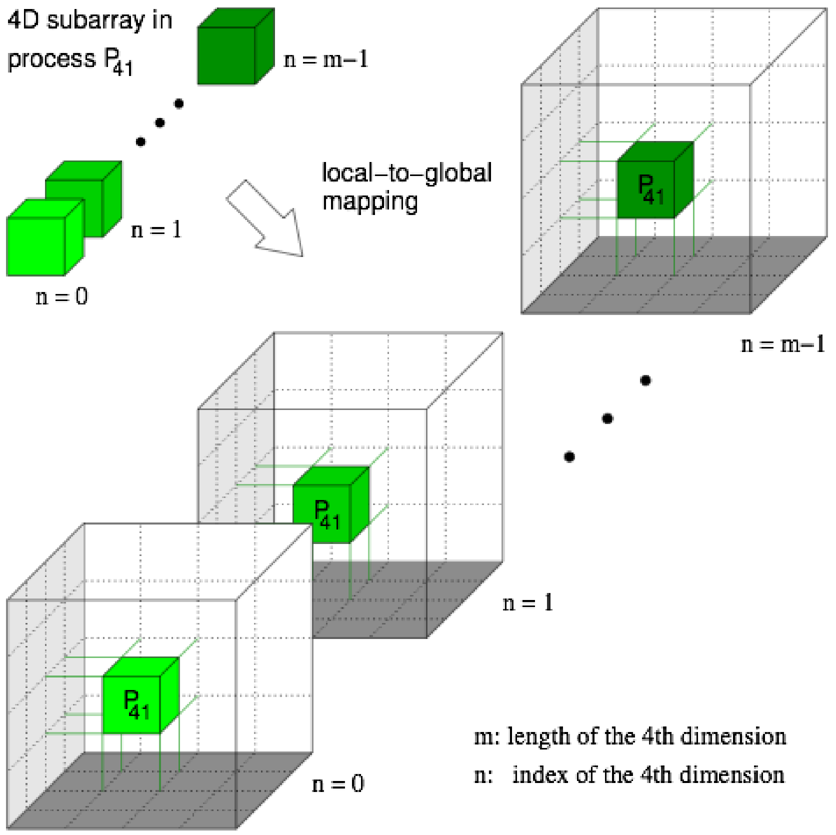


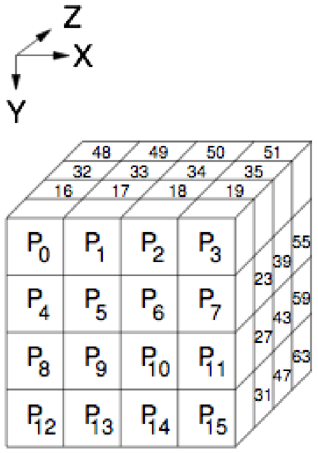
For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November, 2008.

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




4D subarray in process P_{41}
n = m−1
n = 1
n = 0
local−to−global mapping
P_{41}
n = m−1
n = 1
n = 0
m: length of the 4th dimension
n: index of the 4th dimension



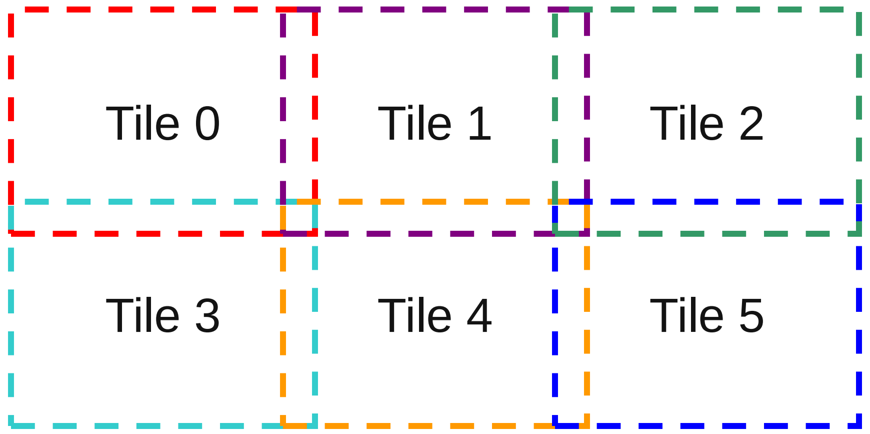
Thanks to Jackie Chen (SNL), Ray Grout (SNL), and Wei-Keng Liao (NWU) for providing the S3D I/O benchmark, Wei-Keng Liao for providing this diagram, C. Wang, H.Yu, and K.-L. Ma of UC Davis for image.

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

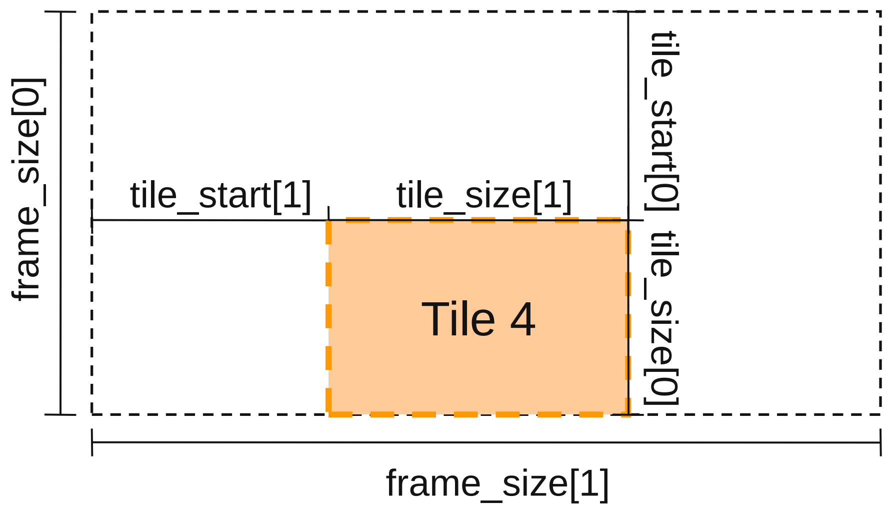
	Coll. Buffering and Data Sieving Disabled	**Data Sieving Enabled**	**Coll. Buffering Enabled (incl. Aggregation)**
POSIX writes	102,401	81	**5**
POSIX reads	0	80	0
MPI-IO writes	64	64	64
Unaligned in file	102,399	80	4
Total written (MB)	6.25	**87.11**	6.25
Runtime (sec)	1443	11	6.0
Avg. MPI-IO time per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

Example: Visualization Staging



- Large frames must be preprocessed before display on a tiled display
- First step in process is extracting “tiles” that will go to each projector
  - Perform scaling, etc.
- Parallel I/O can be used to speed up reading of tiles
  - One process reads each tile
- We’re assuming a raw RGB format with a fixed-length header

MPI Subarray Datatype



- MPI_Type_create_subarray can describe any N-dimensional subarray of an N-dimensional array
- In this case we use it to pull out a 2-D tile
- Tiles can overlap if we need them to
- Separate MPI_File_set_view call uses this type to select the file region

Opening the File, Defining RGB Type

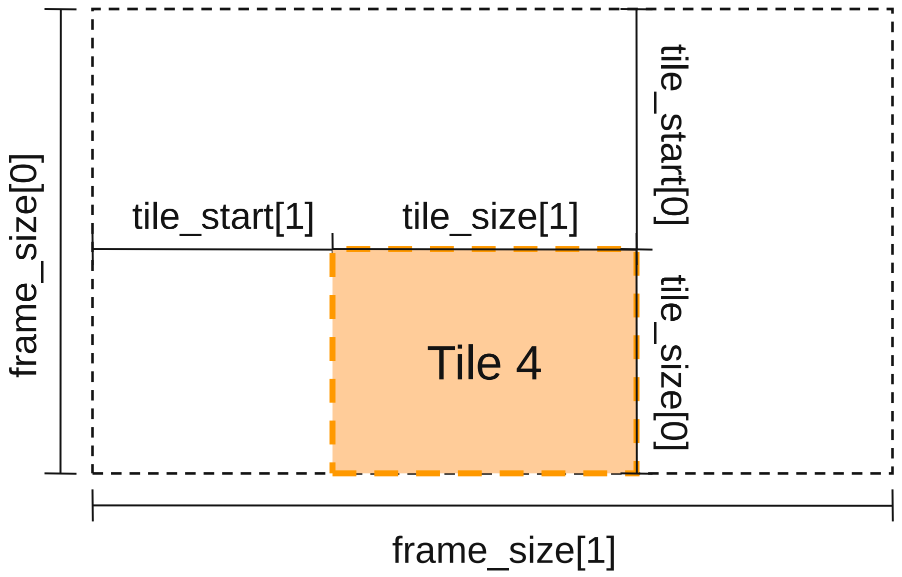
```
MPI_Datatype rgb, filetype;
MPI_File filehandle;
ret = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

/* collectively open frame file */
ret = MPI_File_open(MPI_COMM_WORLD, filename,
   MPI_MODE_RDONLY, MPI_INFO_NULL, &filehandle);

/* first define a simple, three-byte RGB type */
ret = MPI_Type_contiguous(3, MPI_BYTE, &rgb);
ret = MPI_Type_commit(&rgb);
/* continued on next slide */
```

Defining Tile Type Using Subarray

```
/* in C order, last array
 * value (X) changes most
 * quickly
 */
frame_size[1] = 3*1024;
frame_size[0] = 2*768;
tile_size[1] = 1024;
tile_size[0] = 768;
tile_start[1] = 1024 * (myrank % 3);
tile_start[0] = (myrank < 3) ? 0 : 768;
ret = MPI_Type_create_subarray(2, frame_size,
   tile_size, tile_start, MPI_ORDER_C, rgb,
   &filetype);
ret = MPI_Type_commit(&filetype);
```



Reading Noncontiguous Data

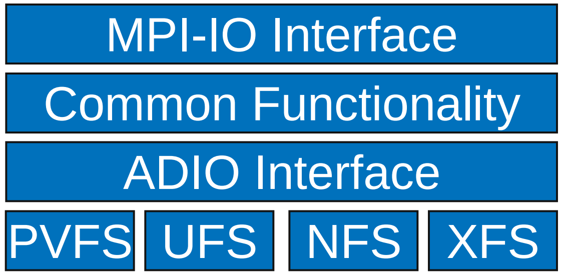
```
/* set file view, skipping header */
ret = MPI_File_set_view(filehandle,
  file_header_size, rgb, filetype, "native",
  MPI_INFO_NULL);
/* collectively read data */
ret = MPI_File_read_all(filehandle, buffer,
  tile_size[0] * tile_size[1], rgb, &status);
ret = MPI_File_close(&filehandle);
```

- MPI_File_set_view is the MPI-IO mechanism for describing noncontiguous regions in a file
  - In this case we use it to skip a header and read a subarray
- Using file views, rather than reading each individual piece, gives the implementation more information to work with (more later)
- Likewise, using a collective I/O call (MPI_File_read_all) provides additional information for optimization purposes (more later)

Under the Covers of MPI-IO

- MPI-IO implementation given a lot of information in this example:
  - Collection of processes reading data
  - Structured description of the regions
- Implementation has some options for how to perform the data reads
  - Noncontiguous data access optimizations
  - Collective I/O optimizations

MPI-IO Implementations



ROMIO's layered architecture.

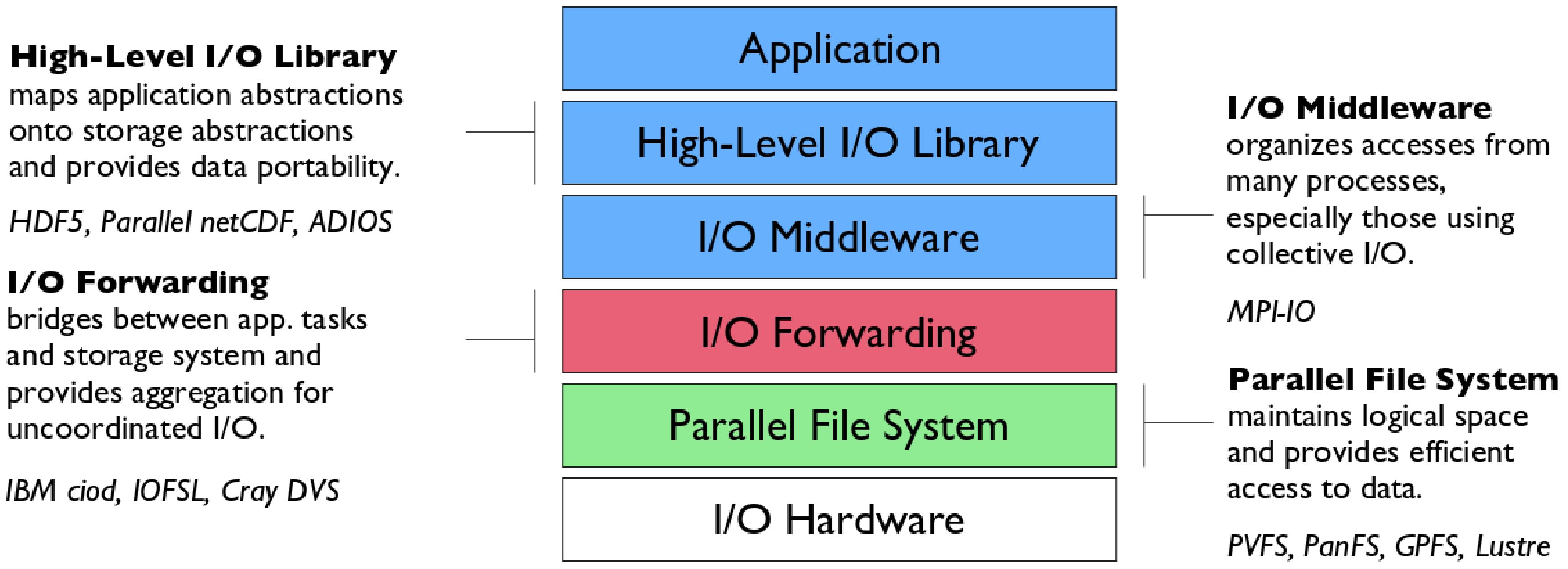
- Different MPI-IO implementations exist
- Three better-known ones are:
  - ROMIO from Argonne National Laboratory
    - Leverages MPI-1 communication
    - Supports local file systems, network file systems, parallel file systems
      - UFS module works GPFS, Lustre, and others
    - Includes data sieving 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

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!

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

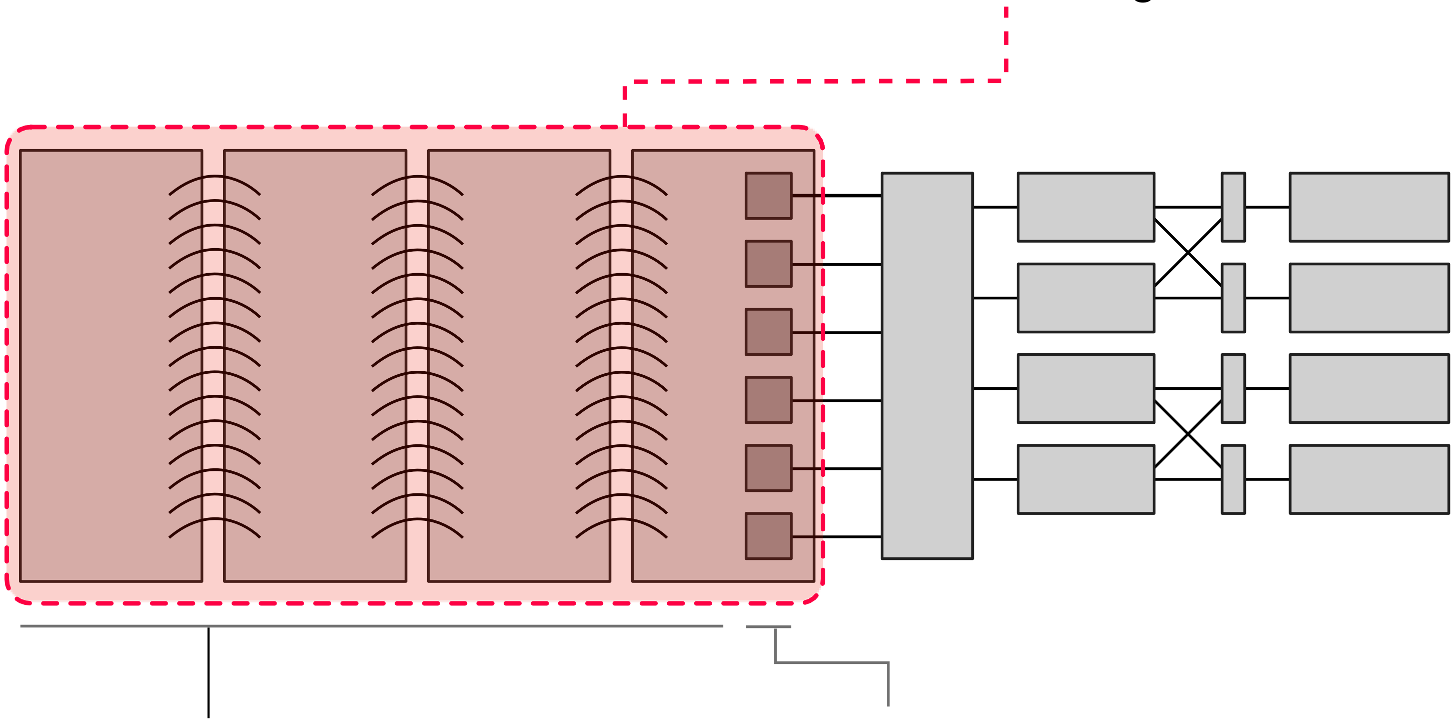
Parallel File System maintains logical space and provides efficient access to data.

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

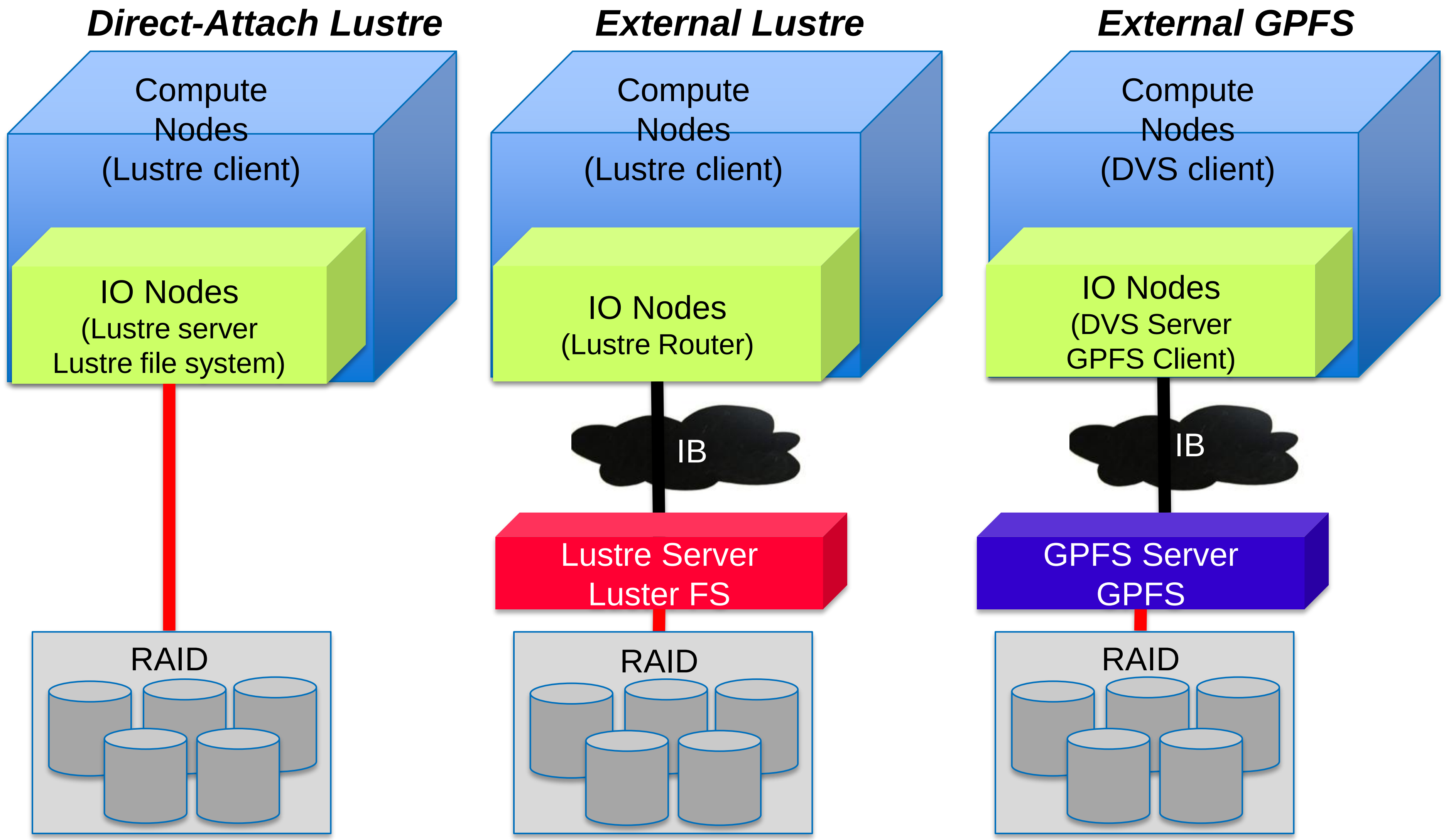
Gateway nodes

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

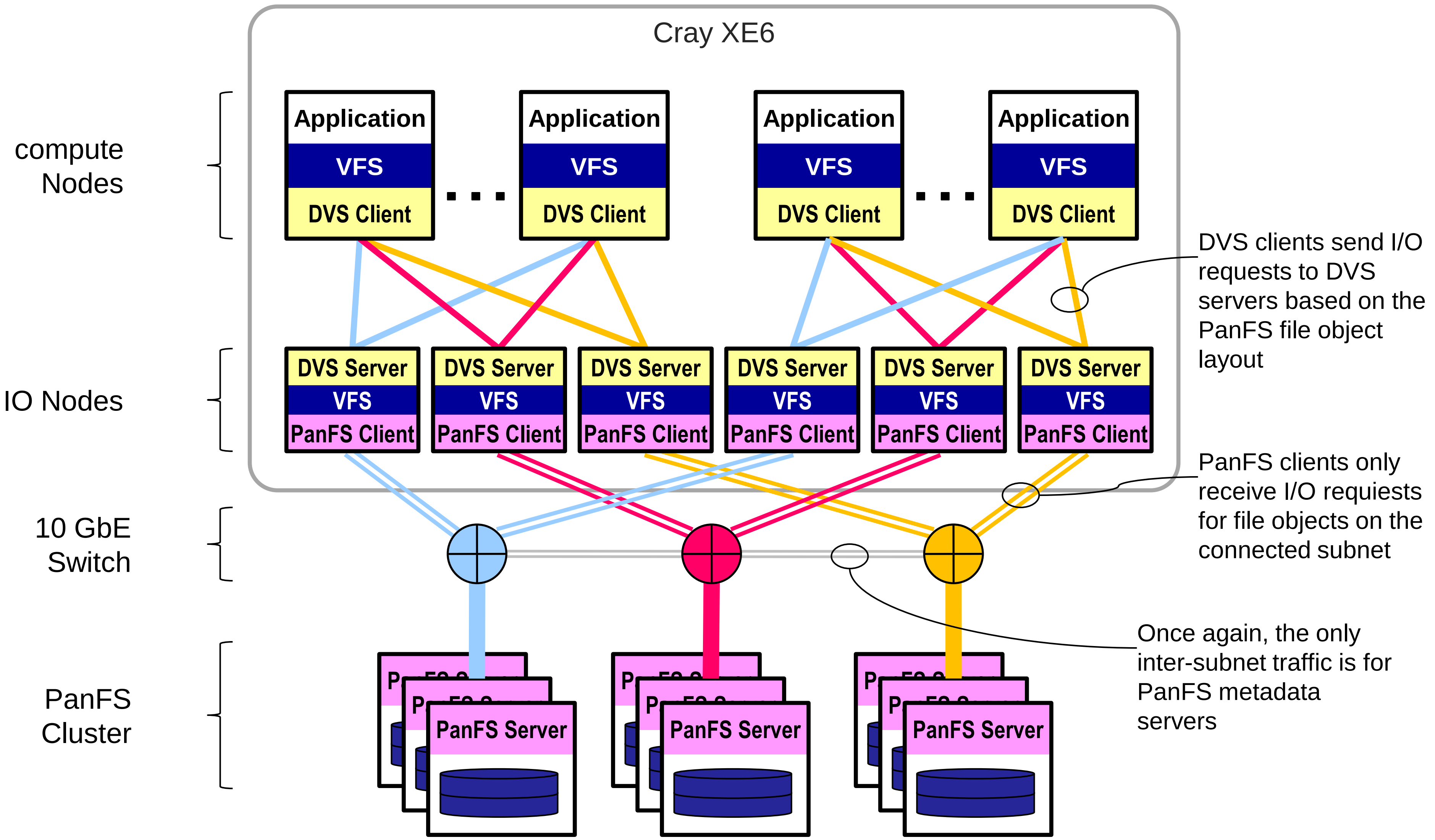
Cray's Data Virtualization Service (DVS)

- 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


Direct-Attach Lustre
External Lustre
External GPFS
Compute Nodes (Lustre client)
IO Nodes (Lustre server Lustre file system)
RAID
Compute Nodes (Lustre client)
IO Nodes (Lustre Router)
IB
Lustre Server Luster FS
RAID
Compute Nodes (DVS client)
IO Nodes (DVS Server GPFS Client)
IB
GPFS Server GPFS
RAID

Panasas and DVS at LANL



DVS clients send I/O requests to DVS servers based on the PanFS file object layout

PanFS clients only receive I/O requiests for file objects on the connected subnet

Once again, the only inter-subnet traffic is for PanFS metadata servers

I/O Architectures: Similarities

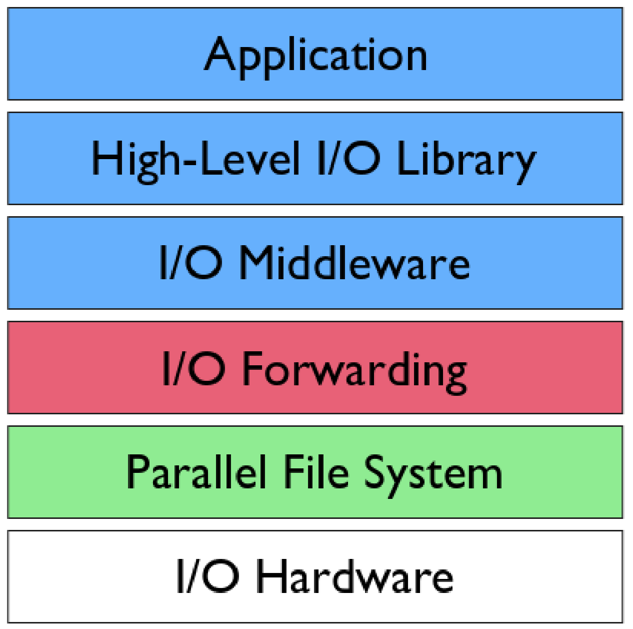
I/O architectures at large scale have converged to a common model.

- 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

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.

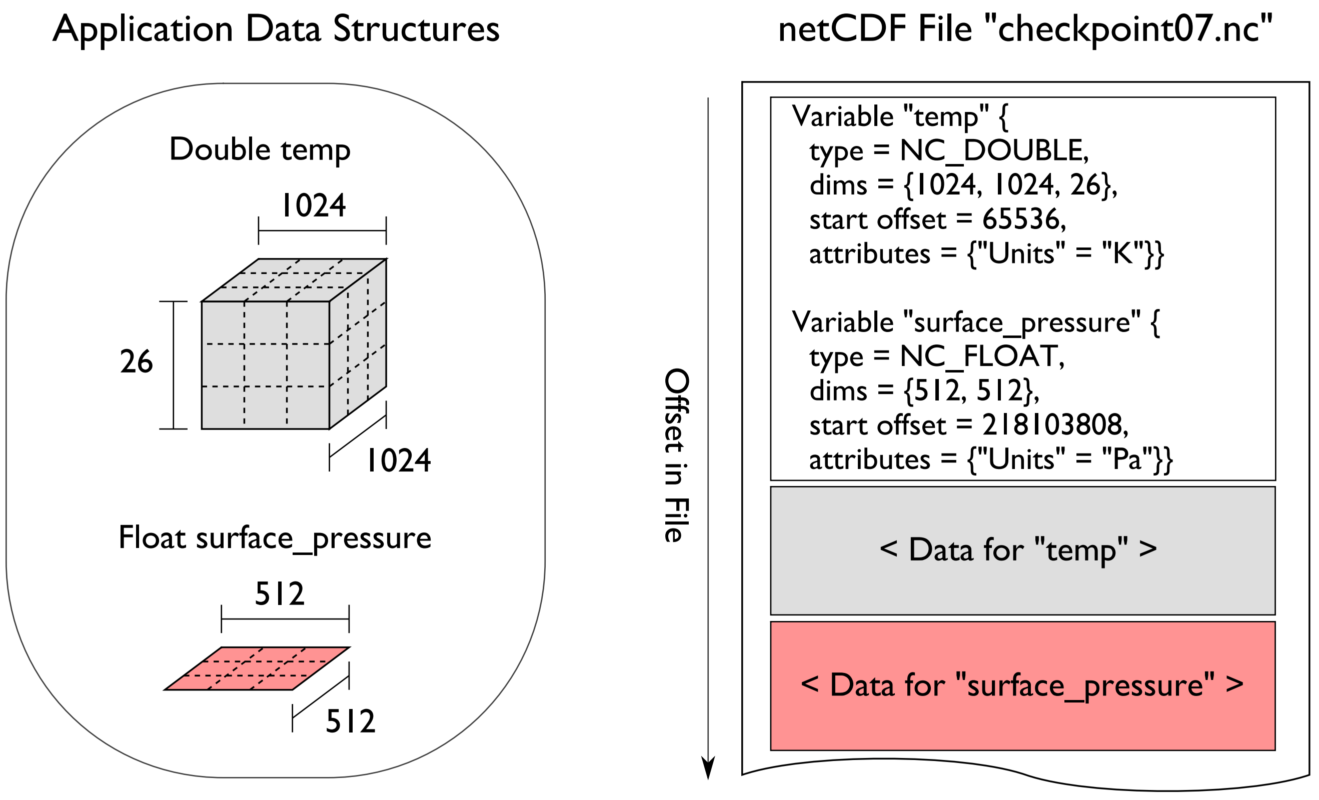
Higher Level I/O Interfaces

- 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



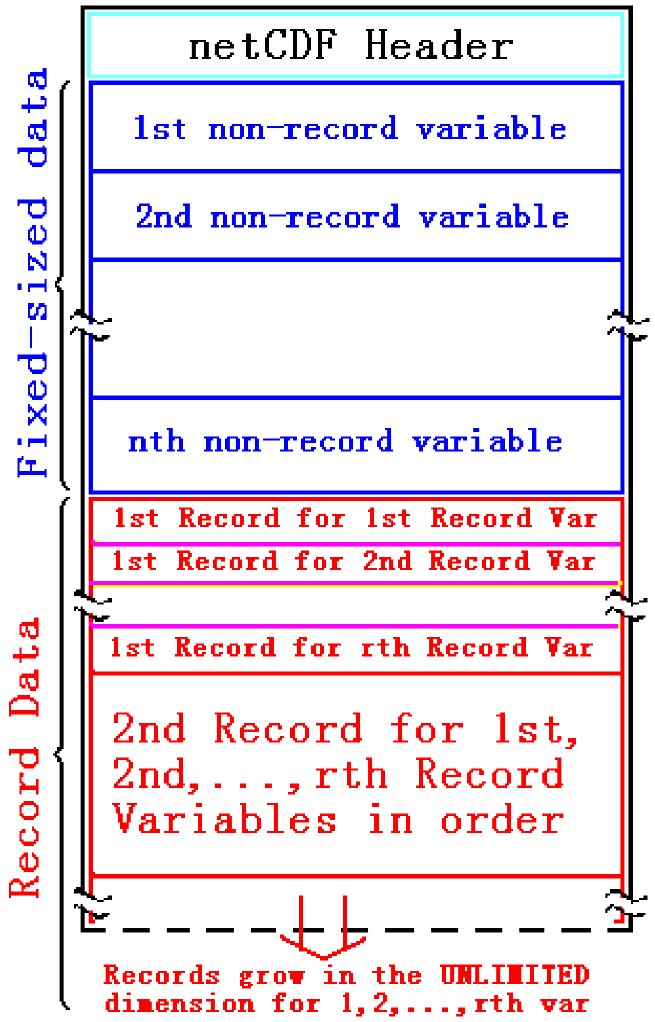
Application Data Structures
Double temp
1024
26
1024
Float surface_pressure
512
512
netCDF File "checkpoint07.nc"
Offset in File
Variable "temp" {
type = NC_DOUBLE,
dims = {1024, 1024, 26},
start offset = 65536,
attributes = {"Units" = "K"}}
Variable "surface_pressure" {
type = NC_FLOAT,
dims = {512, 512},
start offset = 218103808,
attributes = {"Units" = "Pa"}}
< Data for "temp" >
< Data for "surface_pressure" >

netCDF header describes the contents of the file: typed, multi-dimensional variables and attributes on variables or the dataset itself.

Data for variables is stored in contiguous blocks, encoded in a portable binary format according to the variable's type.

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

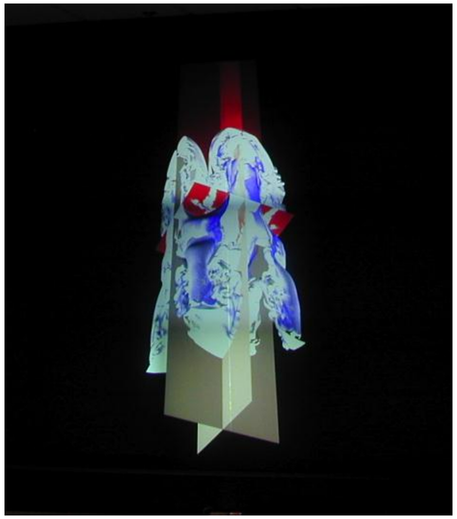

netCDF Header
Fixed-sized data
1st non-record variable
2nd non-record variable
nth non-record variable
Record Data
1st Record for 1st Record Var
1st Record for 2nd Record Var
1st Record for rth Record Var
2nd Record for 1st, 2nd,...,rth Record Variables in order
Records grow in the UNLIMITED dimension for 1,2,...,rth var

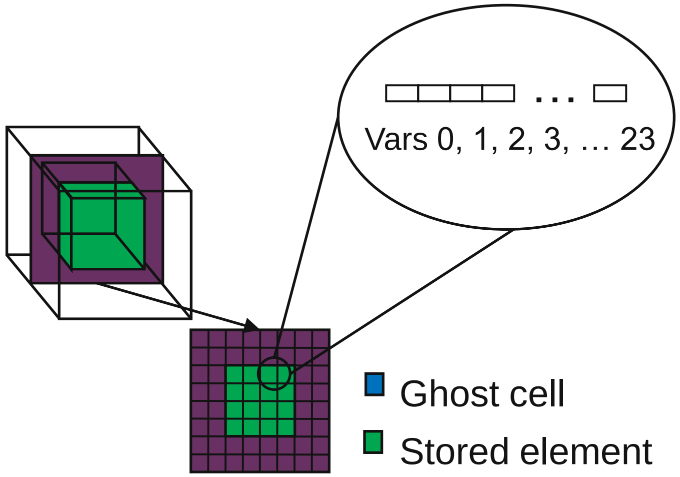
Storing Data in PnetCDF

- 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 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 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];
int varids[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

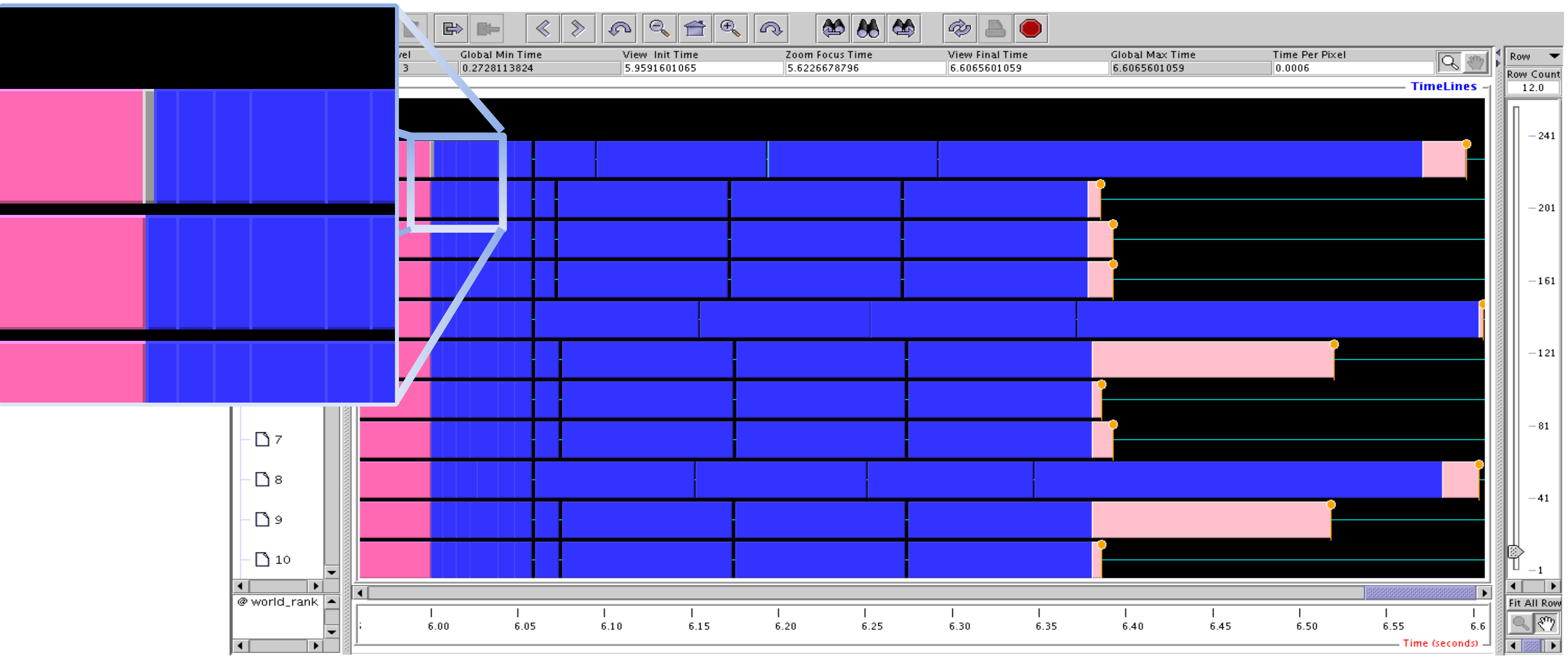
- Inside `ncmpi_put_vara_all` (once per variable)
  - Each process performs data conversion into internal buffer
  - Uses `MPI_File_set_view` to define file region
    - Contiguous region for each process in FLASH case
  - `MPI_File_write_all` collectively writes data
- At ncmpi_close
  - `MPI_File_close`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

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)

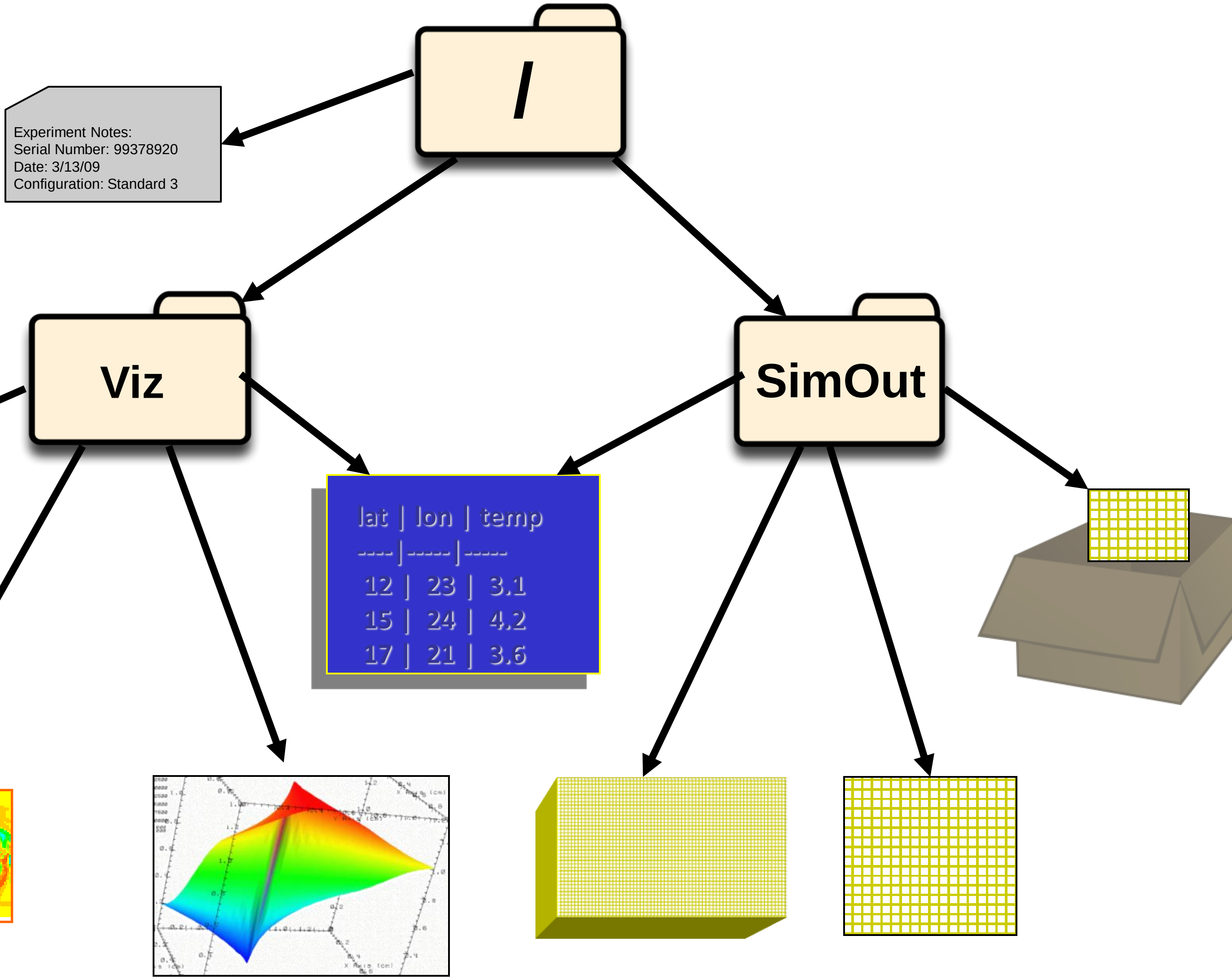
The HDF5 Interface and File Format

HDF5

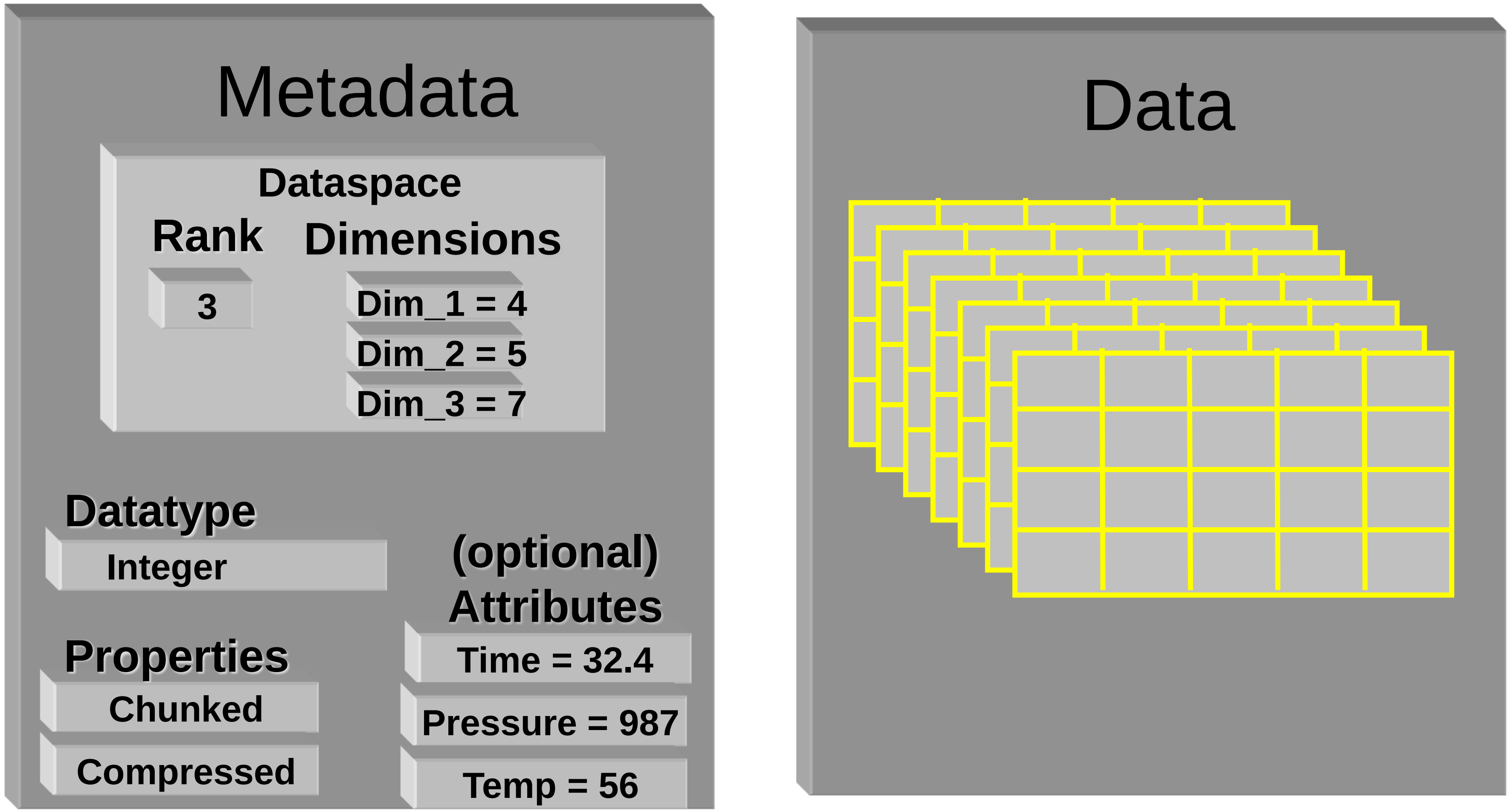
- Hierarchical Data Format, from the HDF Group (formerly of NCSA)
- Data Model:
  - Hierarchical data organization in single file
  - Typed, multidimensional array storage
  - Attributes on dataset, data
- Features:
  - C, C++, and Fortran interfaces
  - Portable data format
  - Optional compression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

HDF5 Groups and Links

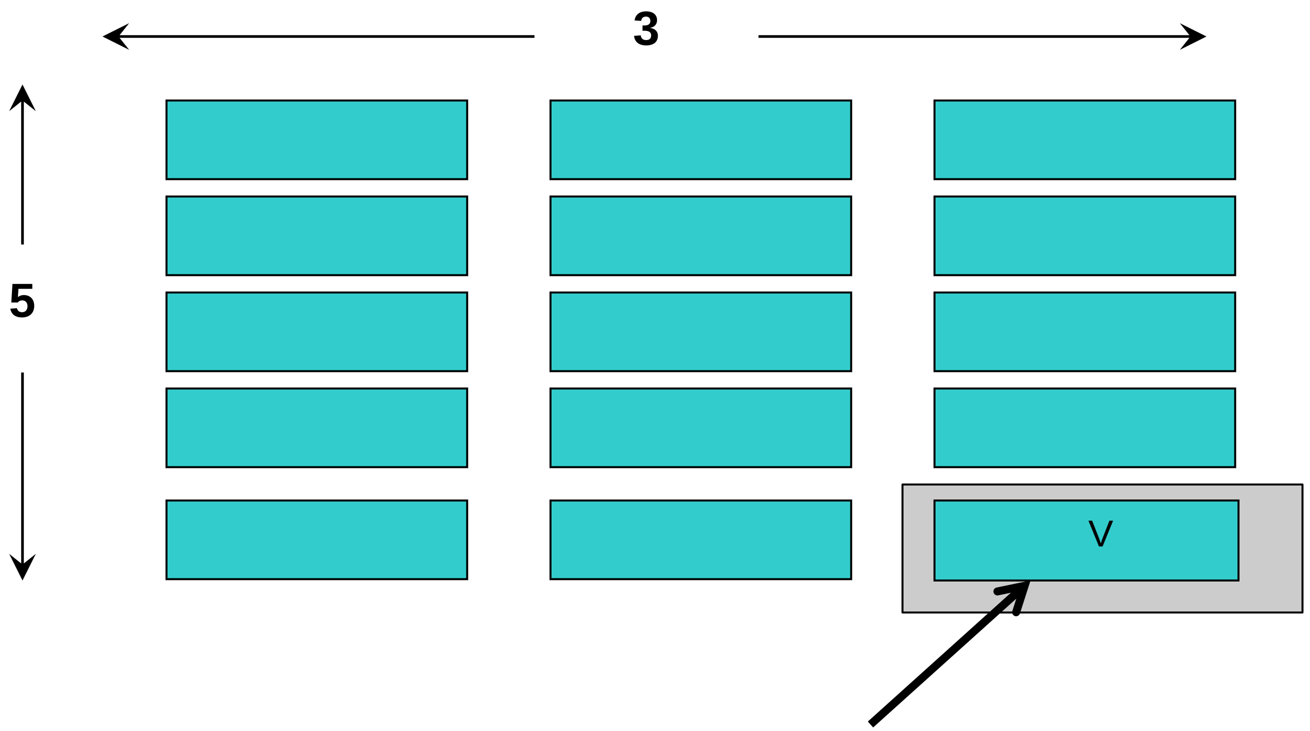
HDF5 groups and links **organize** data objects.



HDF5 Dataset



HDF5 Dataset



Datatype: 16-byte integer

Dataspace: Rank = 2

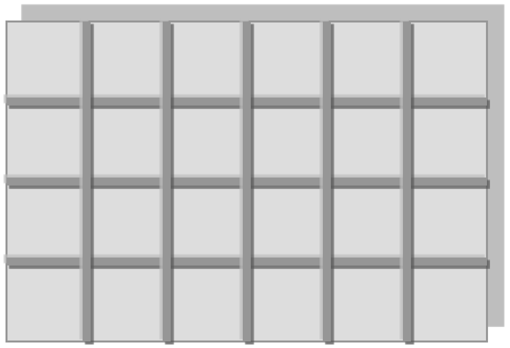
Dimensions = 5 x 3

HDF5 Dataspaces

Two roles:

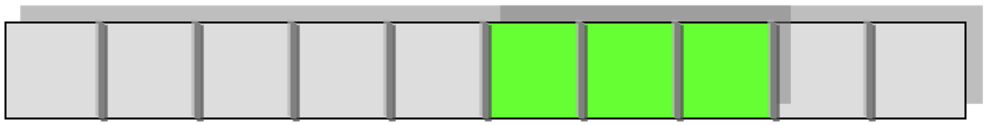
Dataspace contains spatial information (logical layout) about a dataset stored in a file

- 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

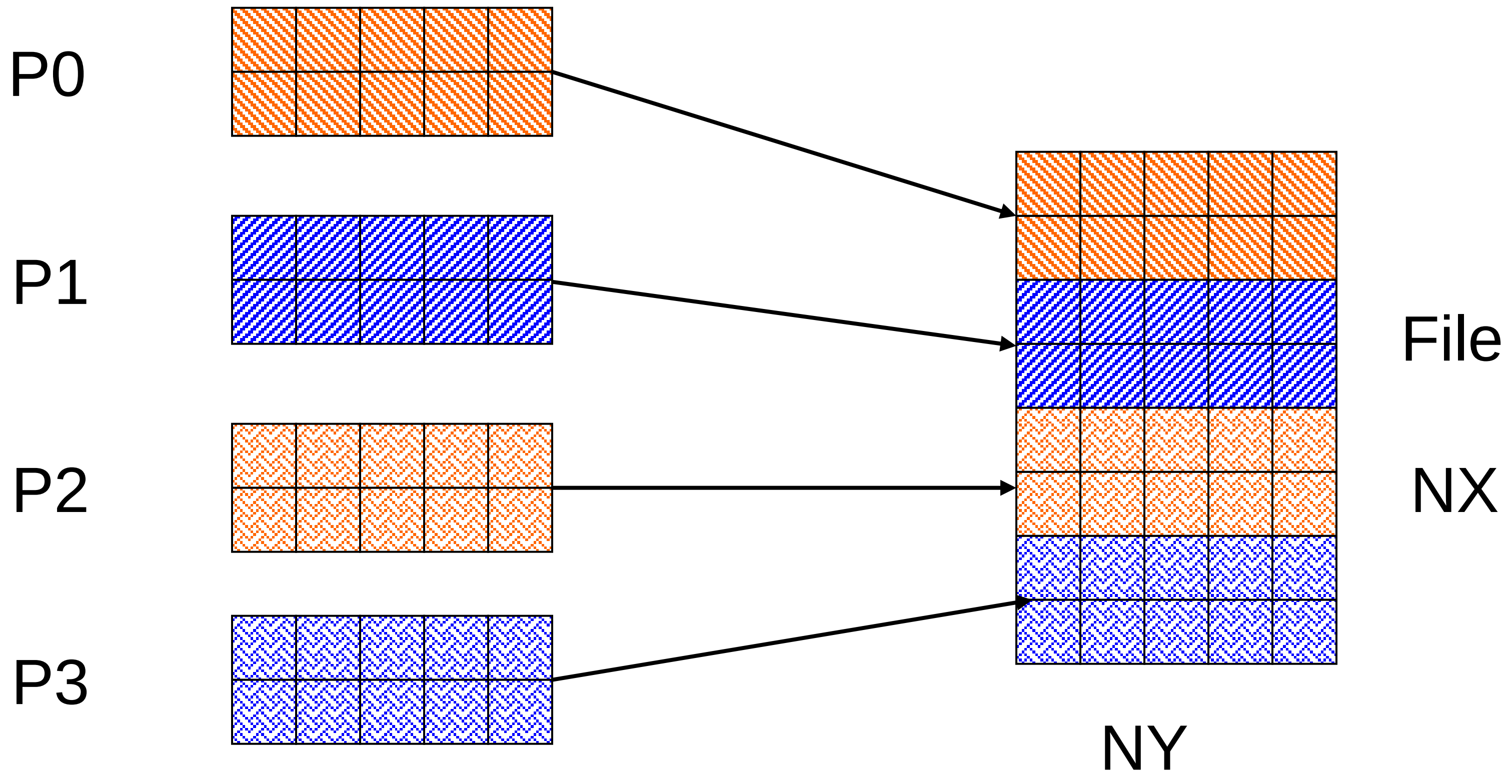
Dimension = 10

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

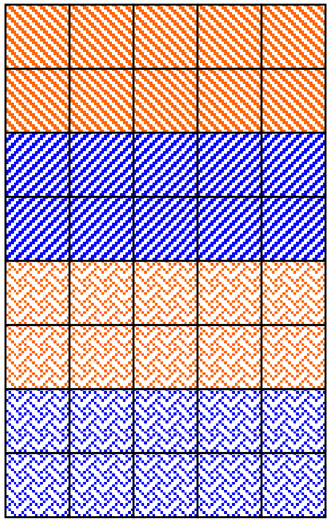
```
/* initialize local grid data */

/* 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);
```



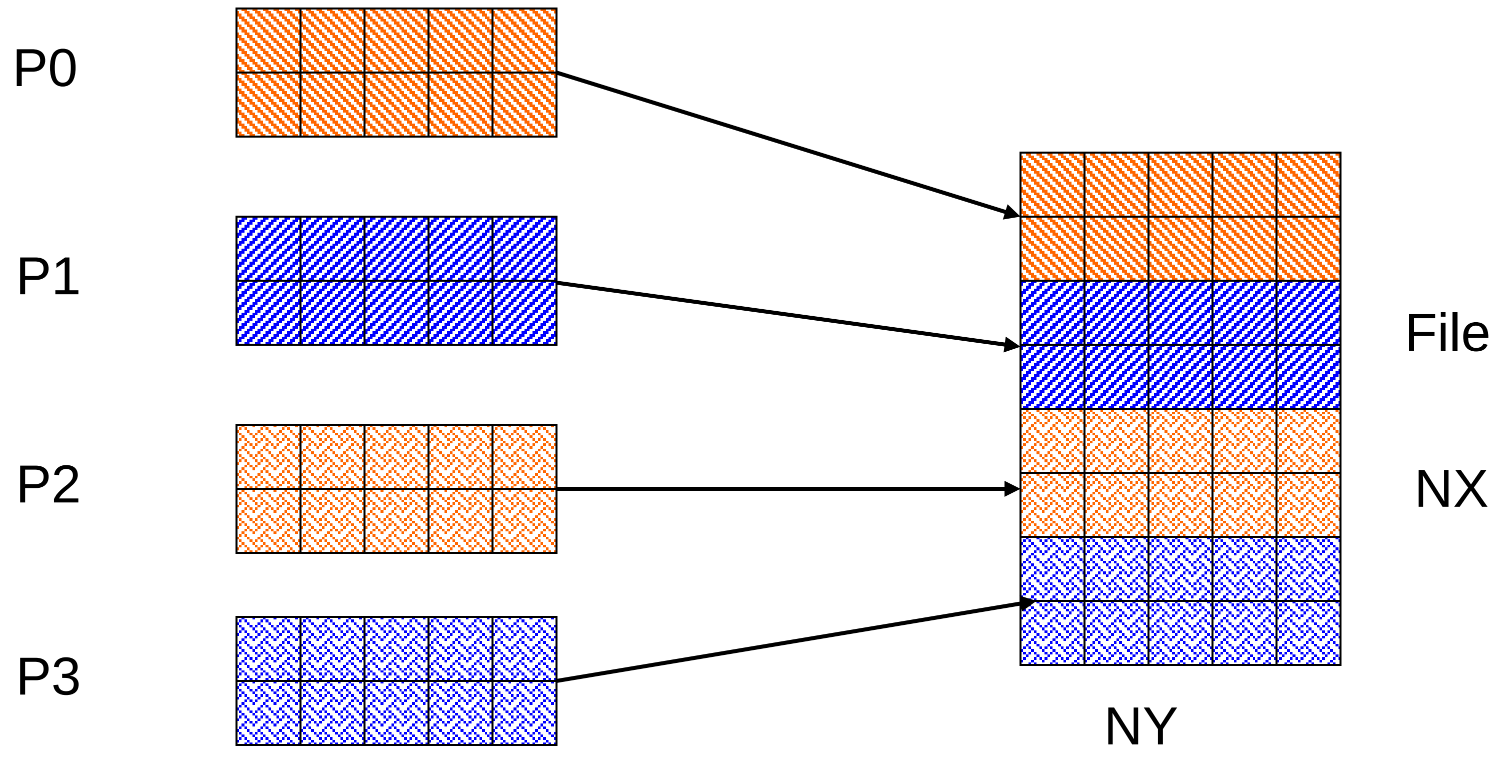
Create Property List

```
/* 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);
```

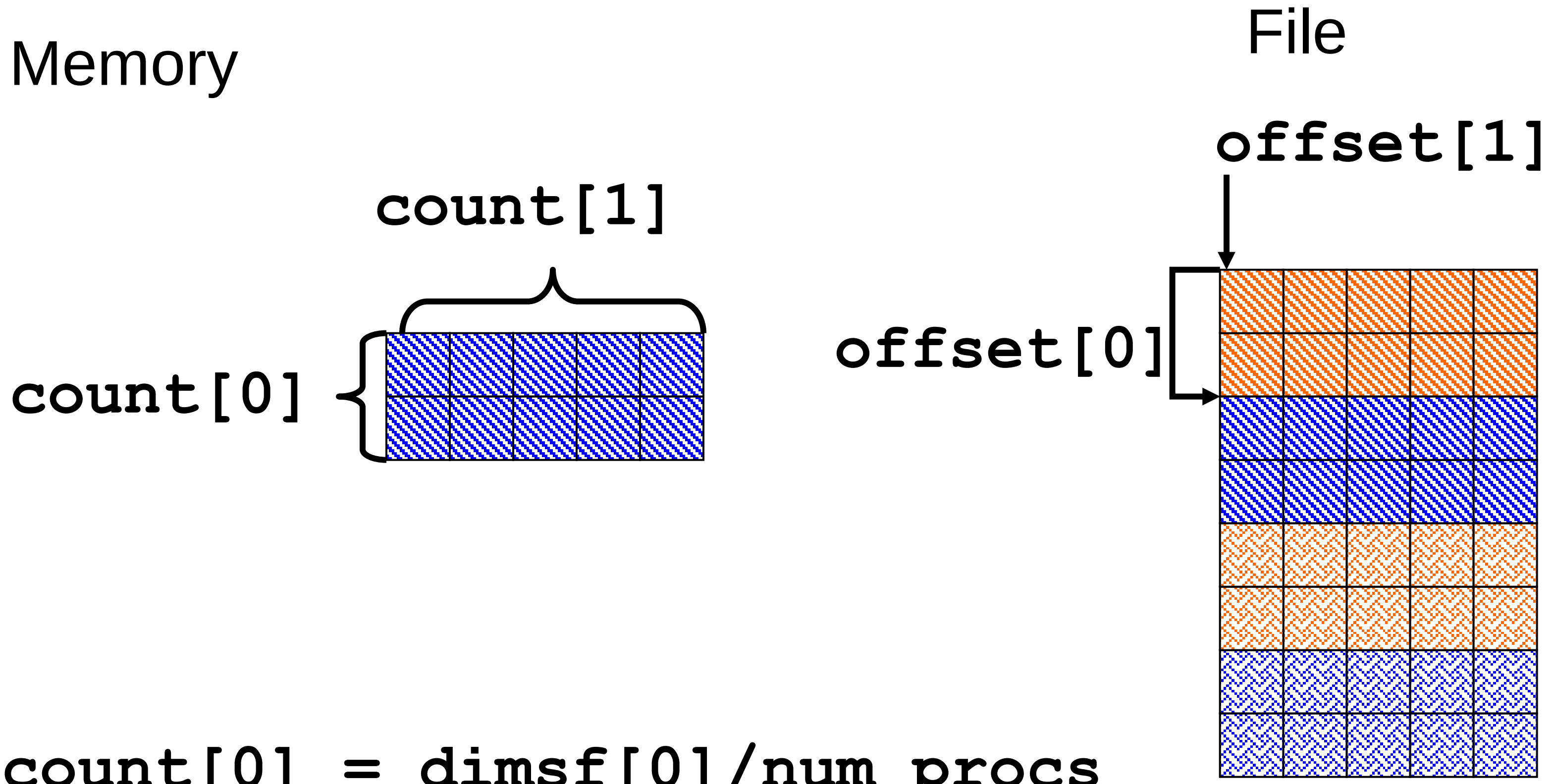
Calculate Offsets



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

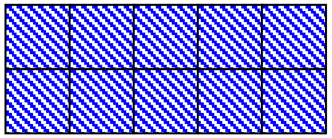


```
count[0] = dimsf[0]/num_procs
count[1] = dimsf[1];
offset[0] = my_proc * count[0];  /* = 2 */
offset[1] = 0;
```

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

Identifier for dataset "dataset1"

Datatype

```
status = H5Dwrite(dset_id, H5T_NATIVE_DOUBLE,
memspace, filespace, plist_id, grid_data);
```

Access Properties:
We choose collective.
This is where other optimizations could be added.

Data buffer

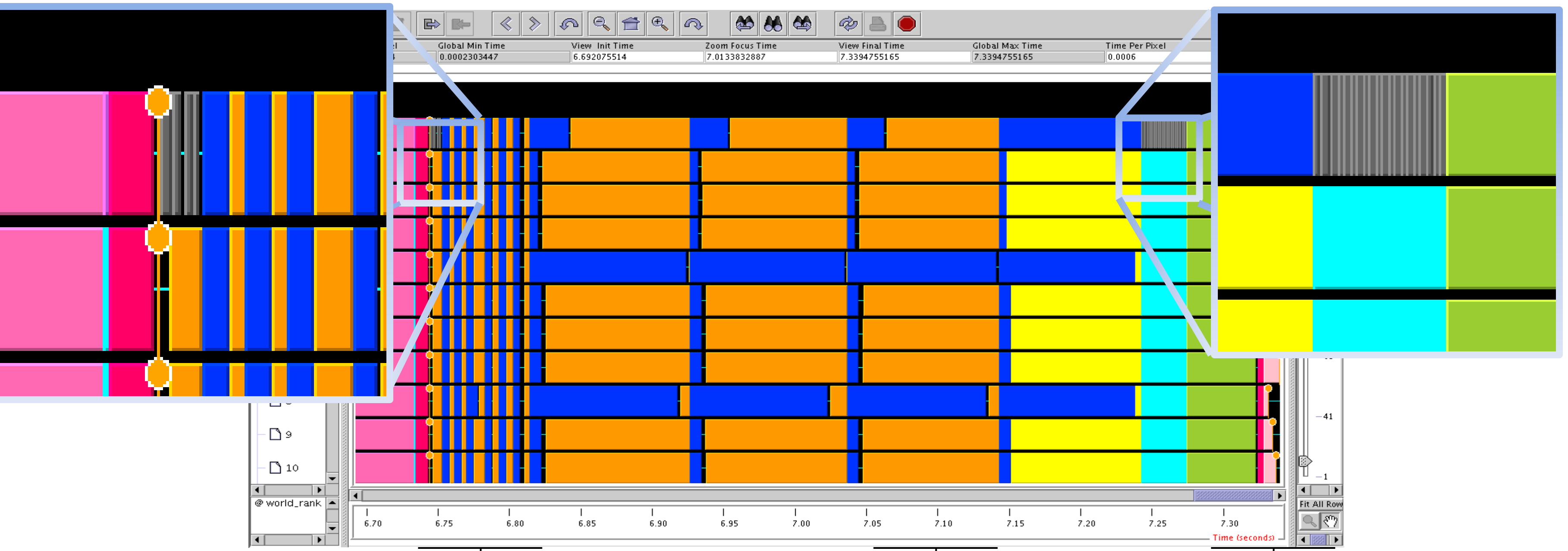
Then close every dataspace and file space that was opened

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



Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0.0002303447	6.692075514	7.0133832887	7.3394755165	7.3394755165	0.0006

2: Collectively write grid, provenance data

4: Collectively write variable (blue)

6: Close file

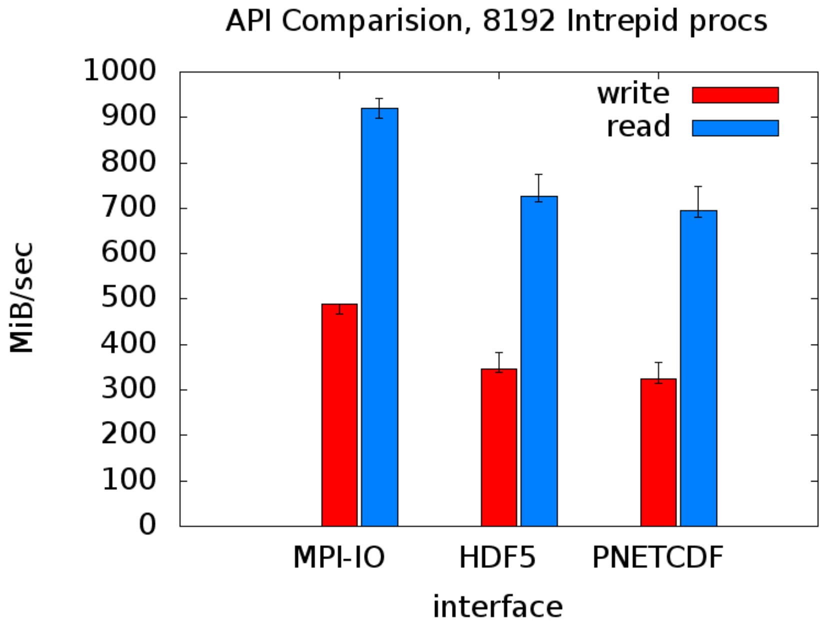
🟪 File open ⬛ Indep. write 🟦 Collective write 🟧 MPI_Allreduce 🟨 File close

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

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 (4 MiB)
- Some performance overhead in using HLL
  - Honestly more than expected



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: it's ok to make your own.

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.

Lightweight Application Characterization with Darshan

Thanks to Phil Carns (carns@mcs.anl.gov) for providing background material on Darshan.

Characterizing Application I/O

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

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

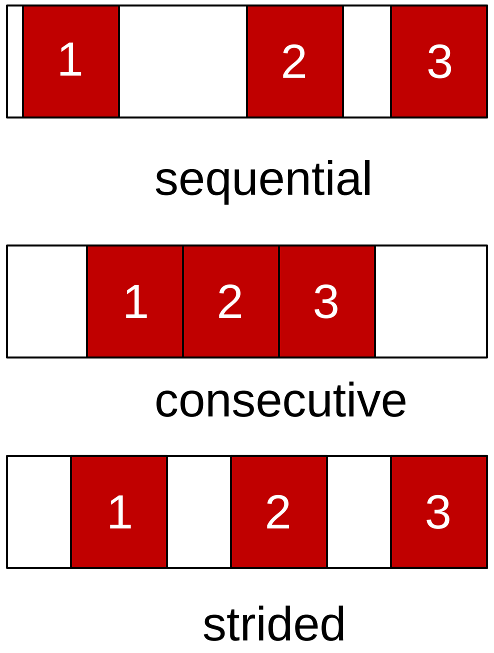
- No code changes, small and tunable memory footprint (~2MB default)
- Characterization data aggregated and compressed prior to writing
- Captures:
  - Counters for POSIX and MPI-IO operations
  - Counters for unaligned, sequential, consecutive, and strided access
  - Timing of opens, closes, first and last reads and writes
  - Cumulative data read and written
  - Histograms of access, stride, datatype, and extent sizes

http://www.mcs.anl.gov/darshan/
P. Carns et al, “24/7 Characterization of Petascale I/O Workloads,” IASDS Workshop, held in conjunction with IEEE Cluster 2009, September 2009.

The Darshan Approach

- Use PMPI and ld wrappers to intercept I/O functions
  - Requires re-linking, but no code modification
  - Can be transparently included in mpicc
  - Compatible with a variety of compilers
- Record statistics independently at each process
  - Compact summary rather than verbatim record
  - Independent data for each file
- 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

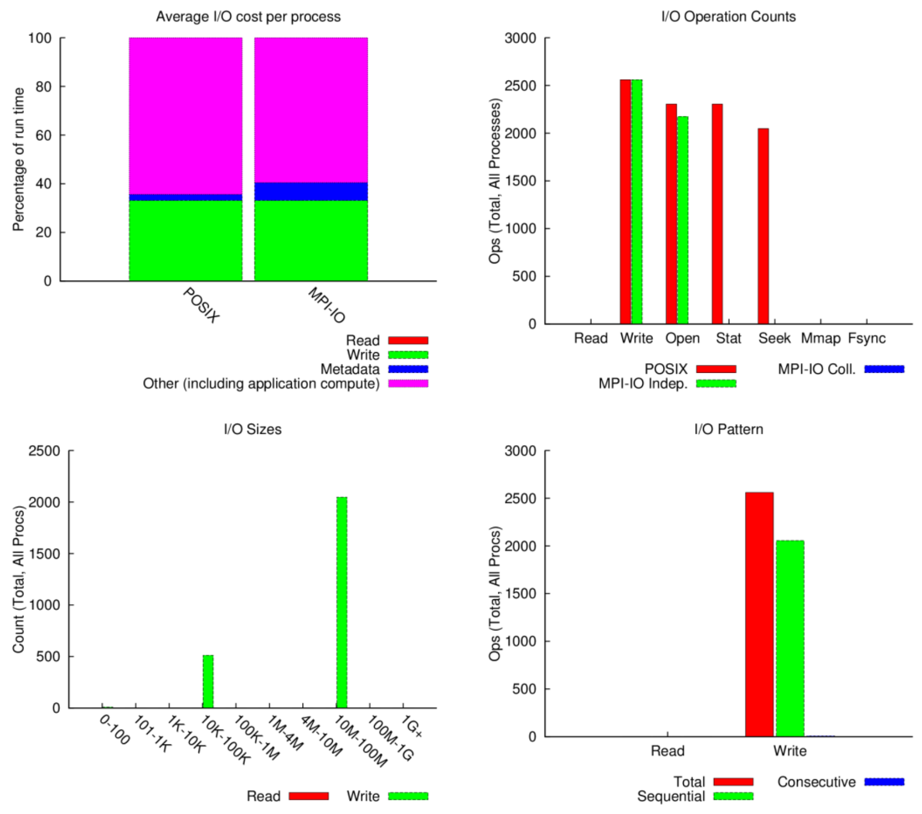
Example Statistics (per file)



- Counters:
  - POSIX open, read, write, seek, stat, etc.
  - MPI-IO nonblocking, collective, indep., etc.
  - Unaligned, sequential, consecutive, strided access
  - MPI-IO datatypes and hints
- Histograms:
  - access, stride, datatype, and extent sizes
- Timestamps:
  - open, close, first I/O, last I/O
- Cumulative bytes read and written
- Cumulative time spent in I/O and metadata operations
- Most frequent access sizes and strides
- Darshan records 150 integer or floating point parameters per file, plus job level information such as command line, execution time, and number of processes.

Darshan Job Summary

jobid: 2[redacted]	uid: [redacted]	nprocs: 4096	runtime: 175 seconds



Most Common Access Sizes

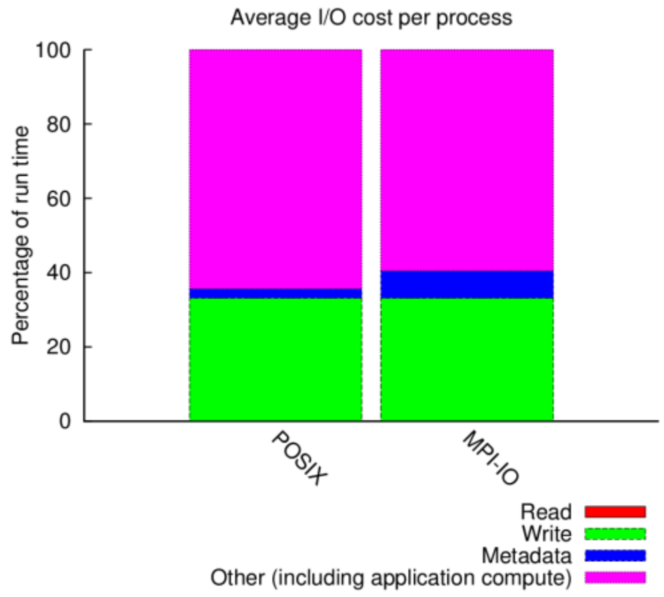
access size	count
67108864	2048
41120	512
8	4
4	3

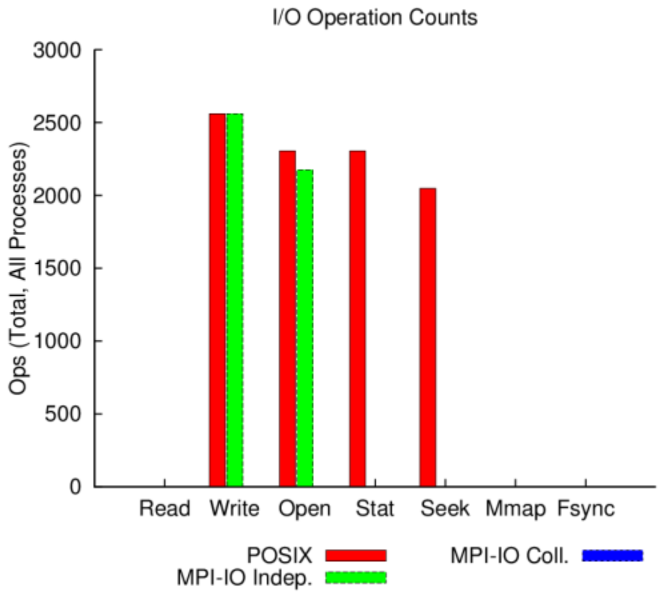
File Count Summary

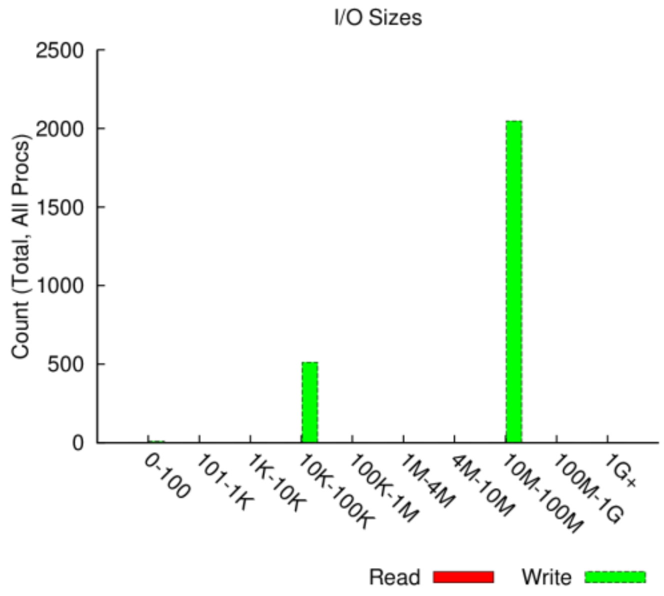
type	number of files	avg. size	max size
total opened	129	1017M	1.1G
read-only files	0	0	0
write-only files	129	1017M	1.1G
read/write files	0	0	0
created files	129	1017M	1.1G

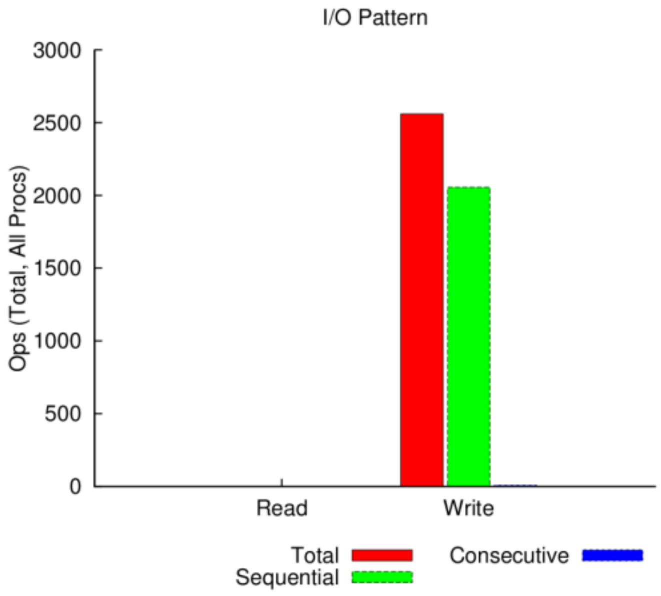
- Job summary tool shows characteristics “at a glance”; available to all users
- Shows time spent in read, write, and metadata
- Operation counts, access size histogram, and access pattern
- Early indication of I/O behavior and where to explore in further
- Example: Mismatch between number of files (R) vs. number of header writes (L)
- The same header is being overwritten 4 times in each data file

jobid: [redacted]	uid: [redacted]	nprocs: 4096	runtime: 175 seconds

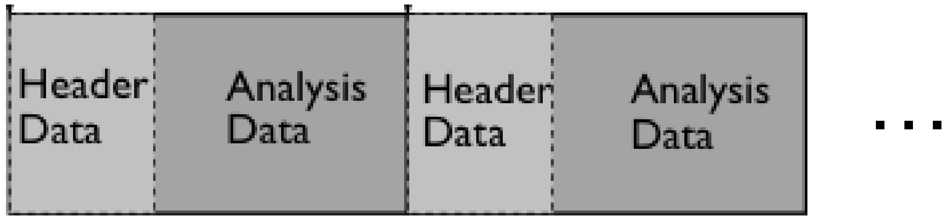








A Data Analysis I/O Example



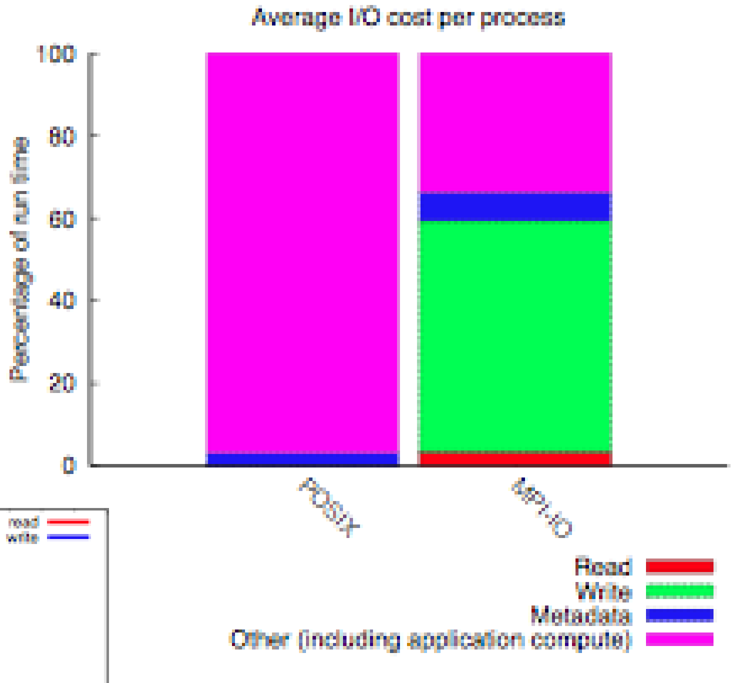
- Variable size analysis data requires headers to contain size information
- Original idea: all processes collectively write headers, followed by all processes collectively write analysis data
- Use MPI-IO, collective I/O, all optimizations
- 4 GB output file (not very large)

Process es	I/O Time (s)	Total Time (s)
8,192	8	60
16,384	16	47
32,768	32	57

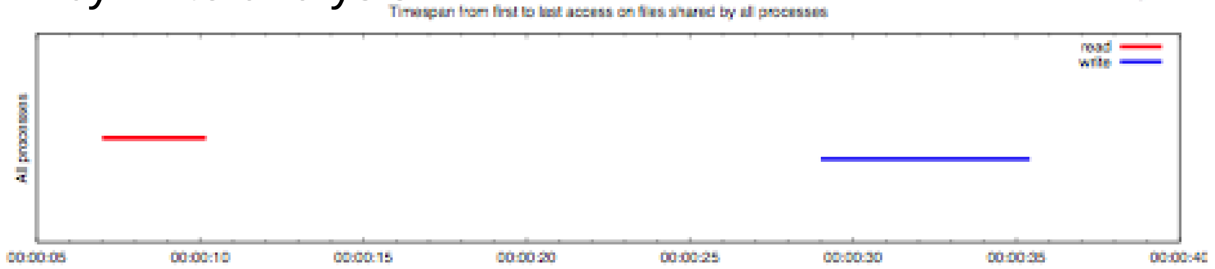
- Why does the I/O take so long in this case?

A Data Analysis I/O Example (continued)

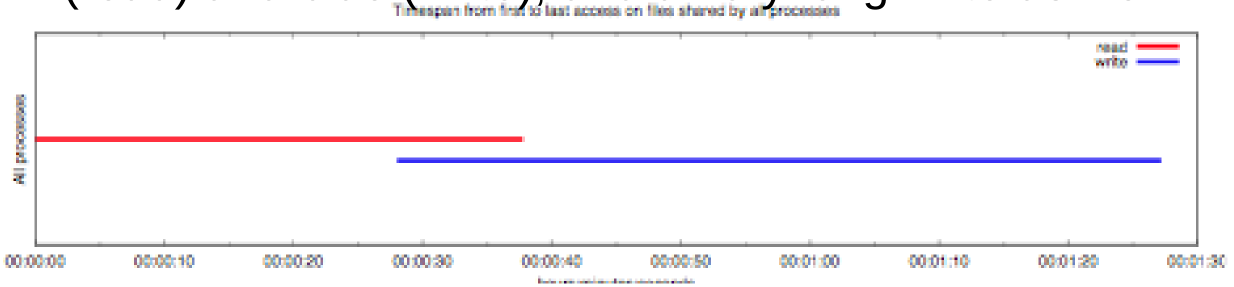
- Problem: More than 50% of time spent writing output at 32K processes. Cause: Unexpected RMW pattern, difficult to see at the application code level, was identified from Darshan summaries.



- What we expected to see, read data followed by write analysis:

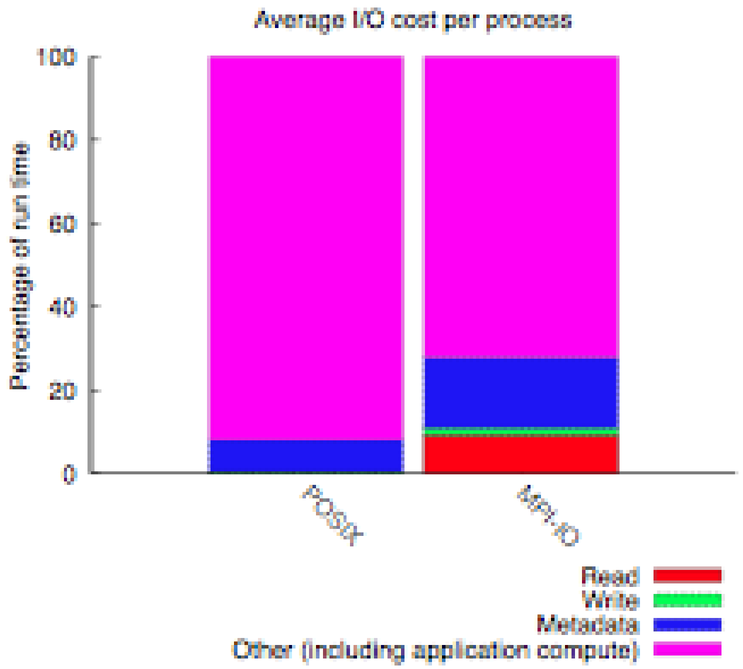


- What we saw instead: RMW during the writing shown by overlapping red (read) and blue (write), and a very long write as well.



A Data Analysis I/O Example (continued)

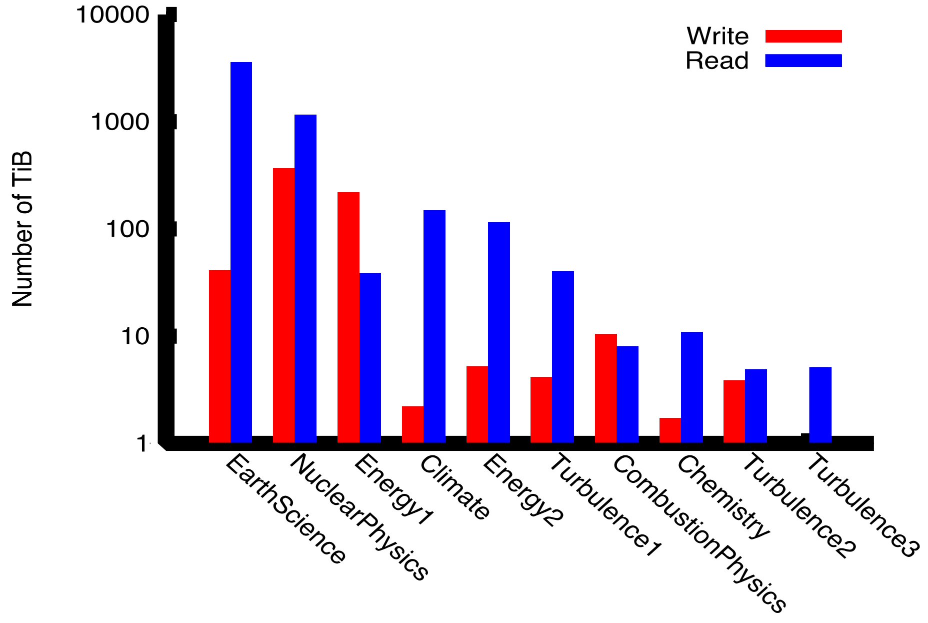
- Solution: Reorder operations to combine writing block headers with block payloads, so that "holes" are not written into the file during the writing of block headers, to be filled when writing block payloads. Also fix miscellaneous I/O bugs; both problems were identified using Darshan.
- Result: Less 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. Also used similar output strategy for cosmology checkpointing, further leveraging the lessons learned.



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

Two Months of Application I/O on ALCF Blue Gene/P

- After additional testing and hardening, Darshan installed on Intrepid
- By default, all applications compiling with MPI compilers are instrumented
- Data captured from late January through late March of 2010
- Darshan captured data on 6,480 jobs (27%) from 39 projects (59%)
- Simultaneously captured data on servers related to storage utilization



Top 10 data producers and/or consumers shown. Surprisingly, most "big I/O" users read more data during simulations than they wrote.

P. Carns et al, "Storage Access Characteristics of Computational Science Applications," forthcoming.

Application I/O on ALCF Blue Gene/P

Application	Mbytes/ sec/CN*	Cum. MD	Files/P roc	Creates/ Proc	Seq. I/O	Mbytes/P roc
EarthScience	0.69	95%	140.67	98.87	65%	1779.48
NuclearPhysics	1.53	55%	1.72	0.63	100%	234.57
Energy1	0.77	31%	0.26	0.16	87%	66.35
Climate	0.31	82%	3.17	2.44	97%	1034.92
Energy2	0.44	3%	0.02	0.01	86%	24.49
Turbulence1	0.54	64%	0.26	0.13	77%	117.92
CombustionPhysics	1.34	67%	6.74	2.73	100%	657.37
Chemistry	0.86	21%	0.20	0.18	42%	321.36
Turbulence2	1.16	81%	0.53	0.03	67%	37.36
Turbulence3	0.58	1%	0.03	0.01	100%	40.40

* Synthetic I/O benchmarks (e.g., IOR) attain 3.93 - 5.75 Mbytes/sec/CN for modest job sizes, down to approximately 1.59 Mbytes/sec/CN for full-scale runs.

P. Carns et al, “Storage Access Characteristics of Computational Science Applications,” forthcoming.

Darshan Summary

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

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

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?
- 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)

- LLNL I/O tests (IOR, fdtree, mdtest)
  - http://www.llnl.gov/icc/lc/siop/downloads/download.html
- 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
- pNFS Problem 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 (Panasas) for their help in creating this material and presenting this tutorial in prior years.