Parallel I/O in Practice

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

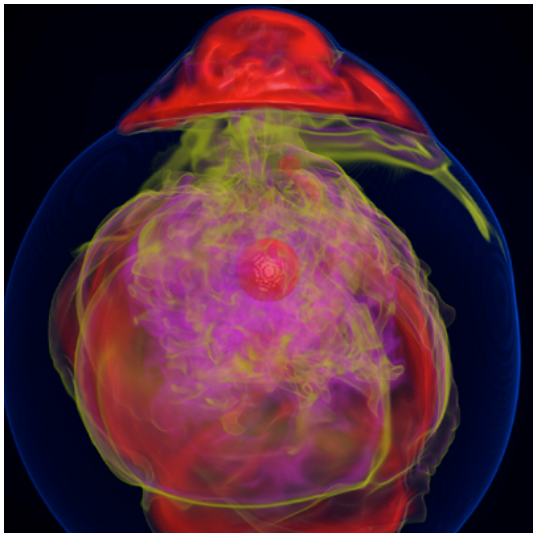
Marc Unangst and Brent Welch
Panasas, Inc.

mju@panasas.com, welch@panasas.com

Computational Science

- Use of computer simulation as a tool for greater understanding of the real world
  - Complements experimentation and theory
- Problems are increasingly computationally challenging
  - 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, 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.

Goals of This Tutorial

- Cover parallel I/O systems from bottom (storage) to top (high-level I/O libraries)
- Provide an understanding of how these pieces fit together to provide a resource for computational science applications
- Introduce the interfaces that one can use to access storage at various levels
- Describe both “entrenched” interfaces and ones that are on the horizon

About Us

- Rob Latham (robl@mcs.anl.gov)
  - Senior Software Developer, MCS Division, Argonne National Laboratory
  - Parallel Virtual File System
  - ROMIO MPI-IO implementation
  - Parallel netCDF high-level I/O library
- Rob Ross (rross@mcs.anl.gov)
  - Computer Scientist, MCS Division, Argonne National Laboratory
  - Parallel Virtual File System
  - SciDAC Scientific Data Management Center
  - High End Computing Interagency Working Group (HECIWG) for File Systems and I/O
- Brent Welch (welch@panasas.com)
  - Director of Architecture, Panasas
  - Berkeley Sprite OS Distributed Filesystem
  - Panasas ActiveScale Filesystem
  - IETF pNFS
- Marc Unangst (mju@panasas.com)
  - Software Architect, Panasas
  - CMU NASD object storage & distributed filesystem
  - Panasas ActiveScale Filesystem
  - SPEC SFS

Outline of the Day

- Introduction
- Storage system models
- File systems (part 1)

Break

- File systems (part 2)
- Benchmarking
- POSIX

Lunch

- MPI-IO
- Parallel netCDF
- HDF5
- New and upcoming user interfaces

Break

- I/O Understanding and tuning
- Future storage technologies
- Closing remarks

"There is no physics without I/O."
– Anonymous Physicist
SciDAC Conference
June 17, 2009

(I think he might have been kidding.)

Large-Scale Data Sets

Application teams are beginning to generate 10s of Tbytes of data in a single simulation. For example, a recent GTC run on 29K processors on the XT4 generated over 54 Tbytes of data in a 24 hour period [1].

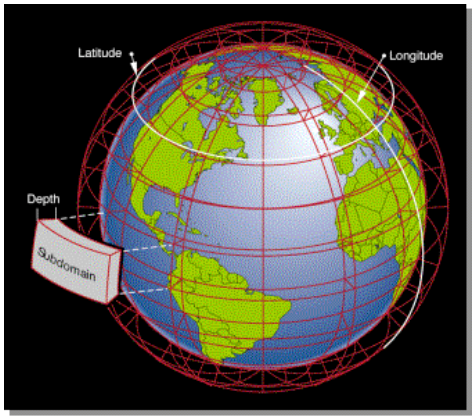
Data requirements for select 2008 INCITE applications at ALCF

PI	Project	On-Line Data	Off-Line Data
Lamb, Don	FLASH: Buoyancy-Driven Turbulent Nuclear Burning	75TB	300TB
Fischer, Paul	Reactor Core Hydrodynamics	2TB	5TB
Dean, David	Computational Nuclear Structure	4TB	40TB
Baker, David	Computational Protein Structure	1TB	2TB
Worley, Patrick H.	Performance Evaluation and Analysis	1TB	1TB
Wolverton, Christopher	Kinetics and Thermodynamics of Metal and Complex Hydride Nanoparticles	5TB	100TB
Washington, Warren	Climate Science	10TB	345TB
Tsigelny, Igor	Parkinson's Disease	2.5TB	50TB
Tang, William	Plasma Microturbulence	2TB	10TB
Sugar, Robert	Lattice QCD	1TB	44TB
Siegel, Andrew	Thermal Striping in Sodium Cooled Reactors	4TB	8TB
Roux, Benoit	Gating Mechanisms of Membrane Proteins	10TB	10TB

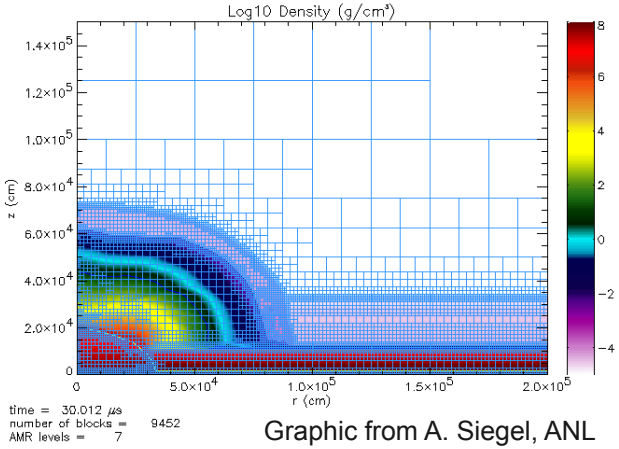
[1] S. Klasky, personal correspondence, June 19, 2008.

Applications, Data Models, and I/O

- Applications have data models appropriate to domain
  - Multidimensional typed arrays, images composed of scan lines, variable length records
  - Headers, attributes on data
- 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)
- Someone has to map from one to the other!



Graphic from J. Tannahill, LLNL



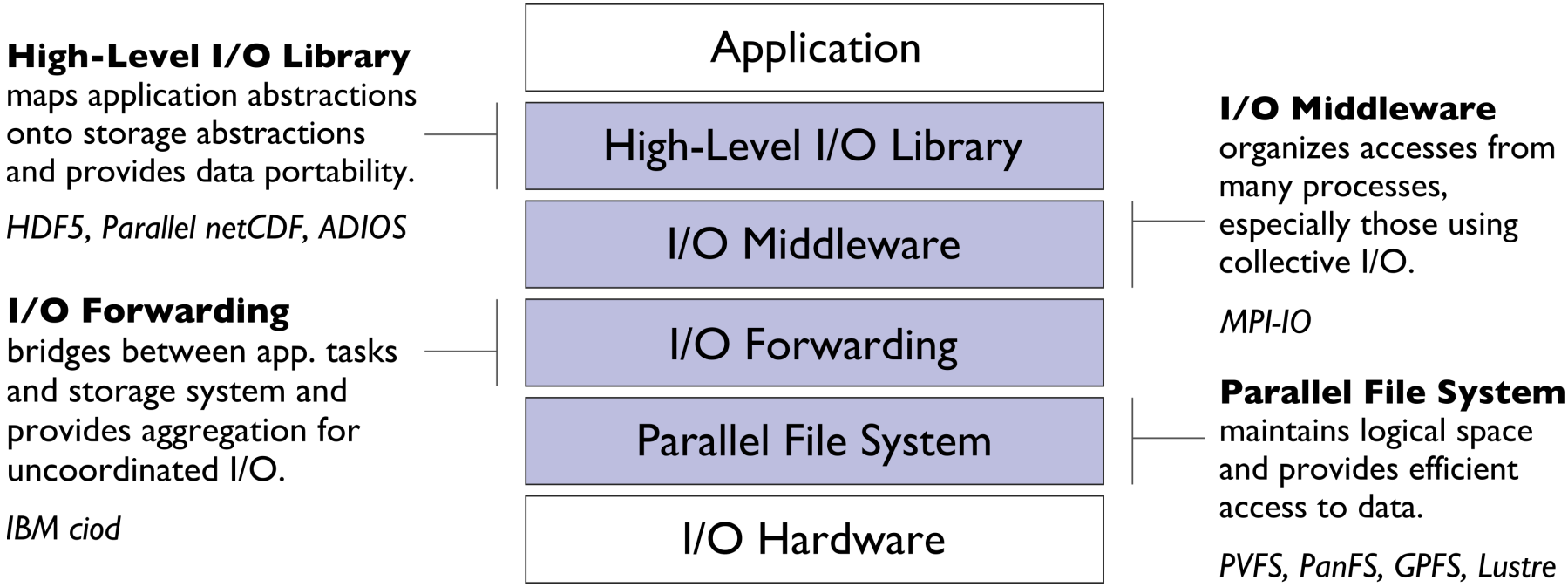
Graphic from A. Siegel, ANL

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

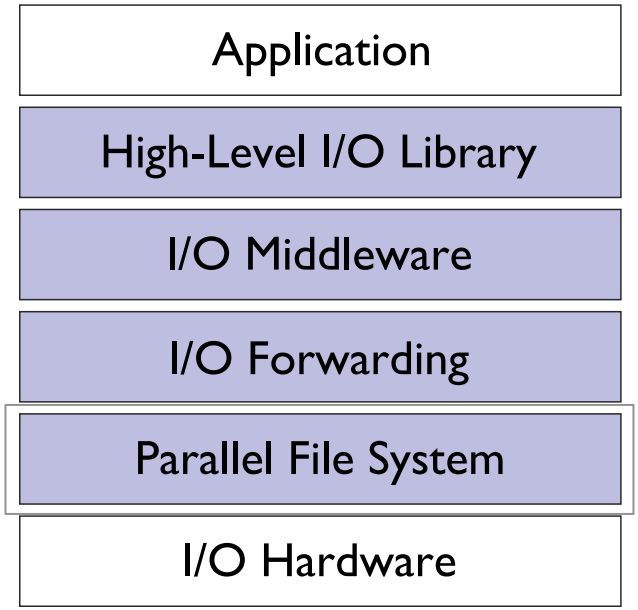
Parallel I/O software is available to address all of these problems, when used appropriately.

I/O for Computational Science



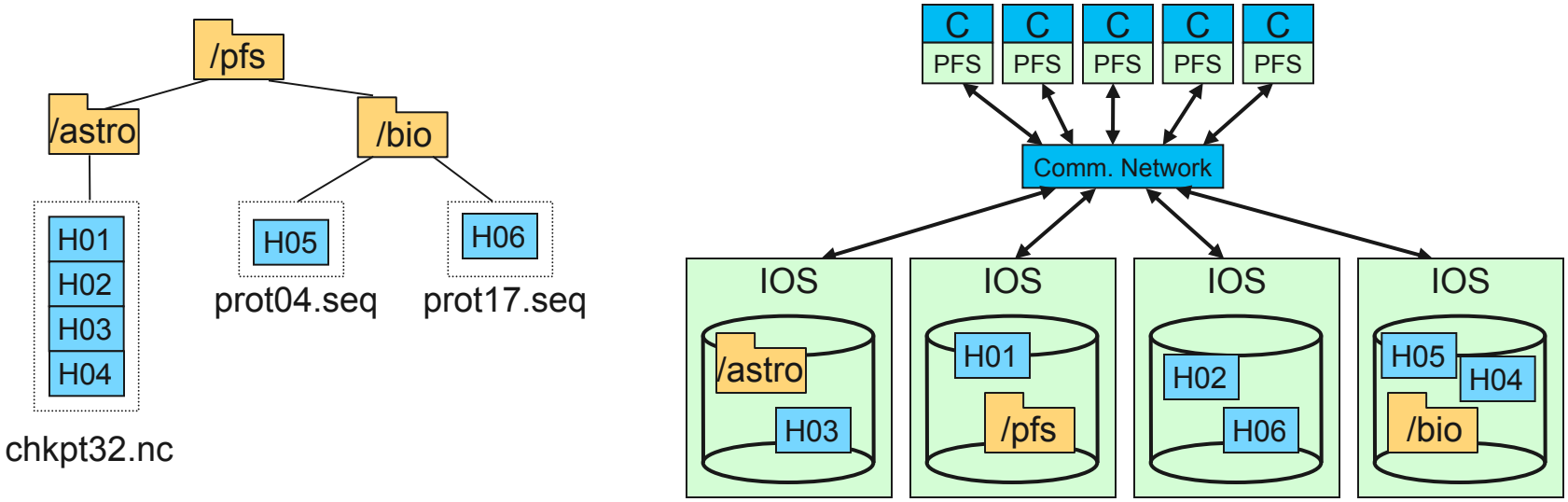
Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.

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

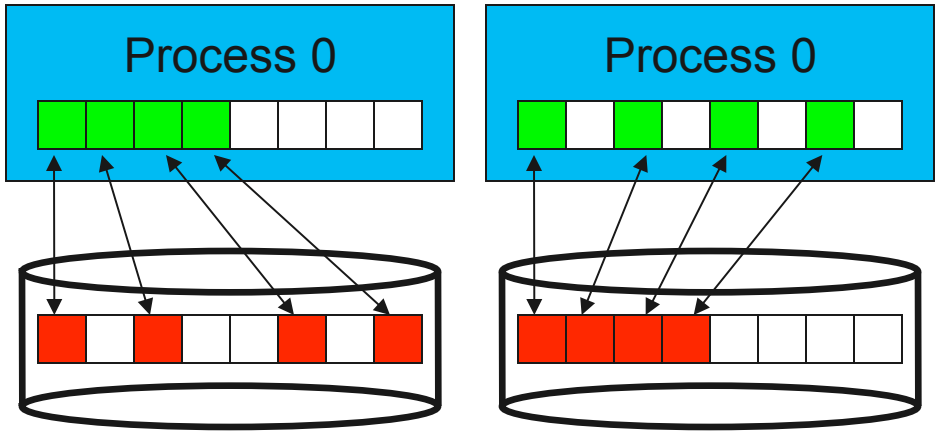
Parallel File Systems



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.

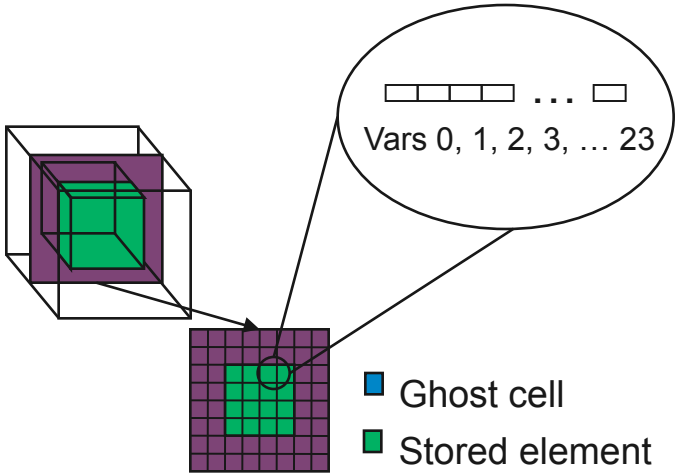
- Building block for HPC I/O systems
  - Present storage as a single, logical storage unit
  - Stripe files across disks and nodes for performance
  - Tolerate failures (in conjunction with other HW/SW)
- User interface is often POSIX file I/O interface, not very good for HPC

Contiguous and Noncontiguous I/O



Noncontiguous in File

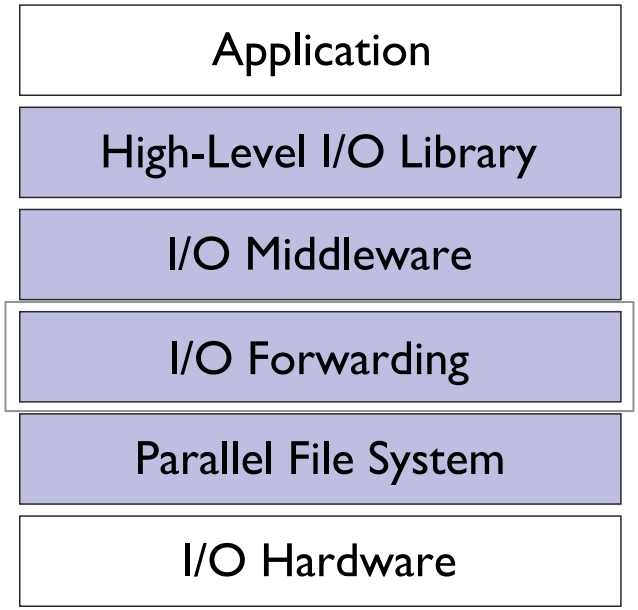
Noncontiguous in Memory



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

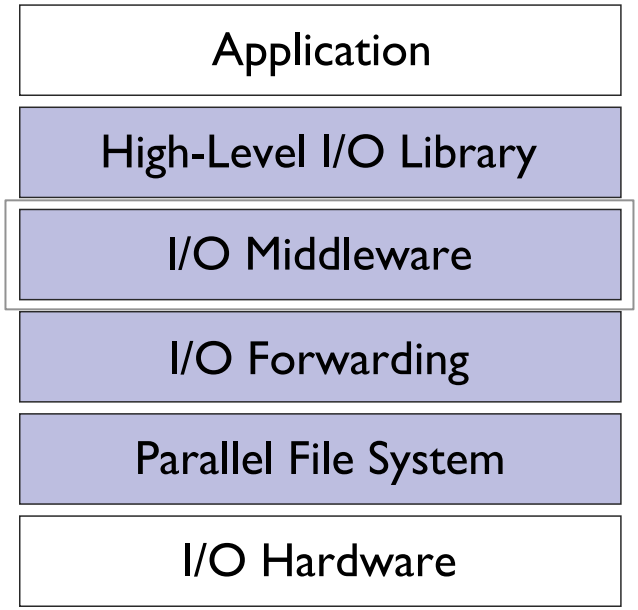
I/O Forwarding



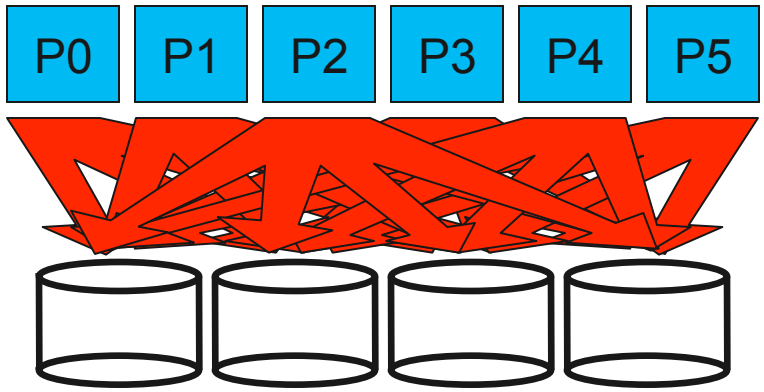
- Newest layer in the stack
  - Present in some of the largest systems
  - Provides bridge between system and storage in machines such as the Blue Gene/P
- Allows for a point of aggregation, hiding true number of clients from underlying file system
- Poor implementations can lead to unnecessary serialization,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

I/O Middleware

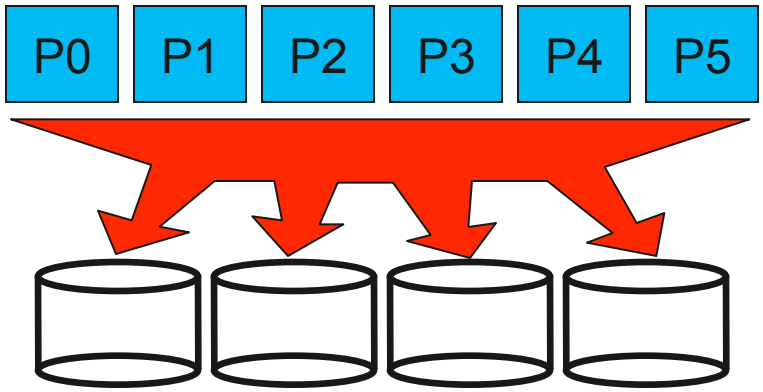
- Match the programming model (e.g. MPI)
- Facilitate concurrent access by groups of processes
  - Collective I/O
  - Atomicity rules
- Expose a generic interface
  - Good building block for high-level libraries
- Efficiently map middleware operations into PFS ones
  - Leverage any rich PFS access constructs,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



Independent and Collective I/O



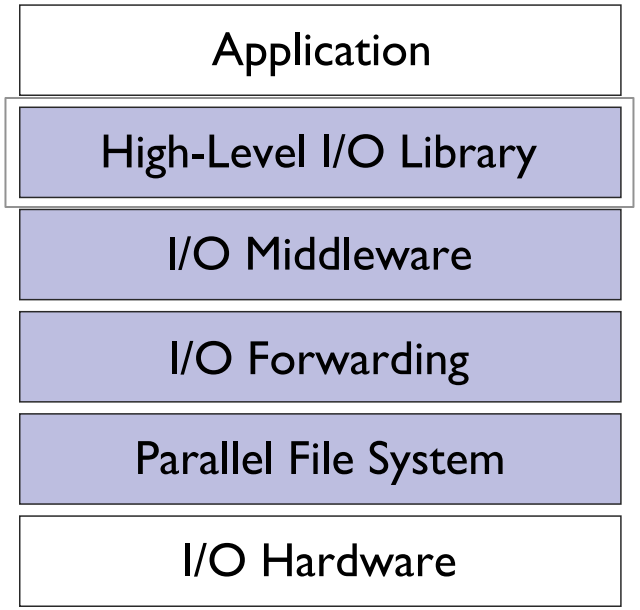
Independent I/O



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

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

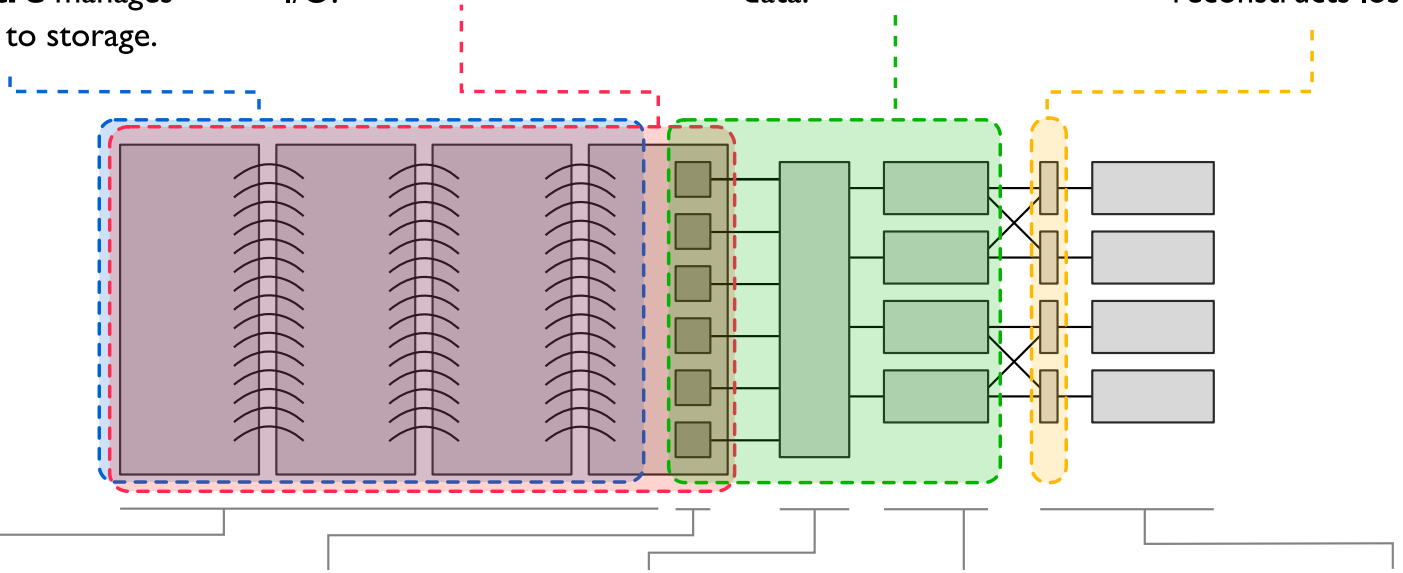
I/O Hardware and Software on Blue Gene/P

High-level I/O libraries execute on compute nodes, mapping application abstractions into flat files, and encoding data in portable formats.
I/O middleware manages collective access to storage.

I/O forwarding software runs on compute and gateway nodes, bridges networks, and provides aggregation of independent I/O.

Parallel file system code runs on gateway and storage nodes, maintains logical storage space and enables efficient access to data.

Drive management software or firmware executes on storage controllers, organizes individual drives, detects drive failures, and reconstructs lost data.



Compute nodes
40,960 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each

Gateway nodes
640 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each

Commodity network
900+ port 10 Gigabit Ethernet Myricom switch complex

Storage nodes
136 two dual core Opteron servers with 8 Gbytes of RAM each

Enterprise storage
17 DataDirect S2A9900 controller pairs with 480 1 Tbyte drives and 8 InfiniBand ports per pair

Architectural diagram of the 557 TFlop IBM Blue Gene/P system at the Argonne Leadership Computing Facility.

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 System Models

File Systems

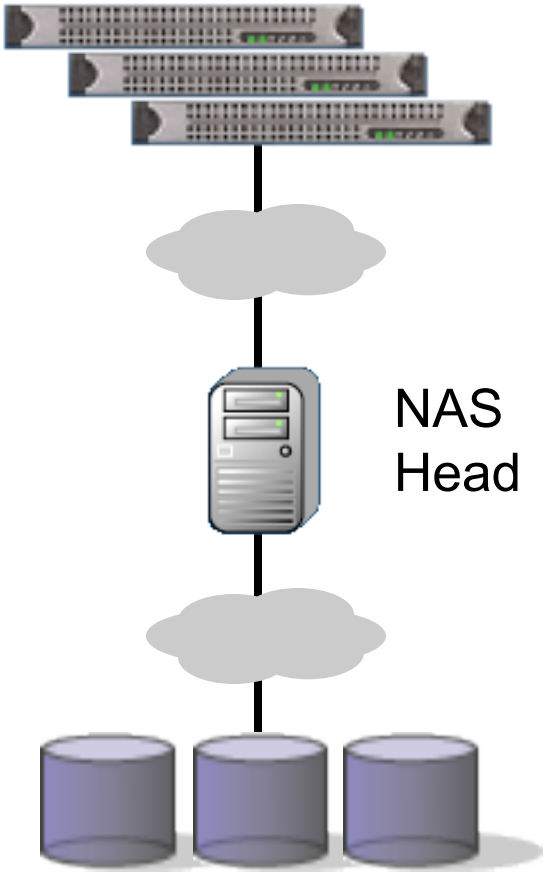
- File systems have two key roles
  - Organizing and maintaining the file name space
  - Storing contents of files
- Local file systems are used by a single operating system instance (client) with direct access to the disk
  - e.g. NTFS or ext3 on your laptop drive
- Networked file systems provide access to one or more clients who might not have direct access to the disk
  - e.g. NFS, AFS, etc.
  - Parallel file systems (PFSes) are a special kind of networked file system written to provide high-performance I/O when multiple clients share file system resources (files)

Parallel File System Design Issues

- Have to solve same problems as local filesystem, at scale
  - Block allocation
  - Metadata management
  - Data reliability and error correction
- Additional requirements
  - Cache coherency
  - High availability
  - Scalable capacity & performance

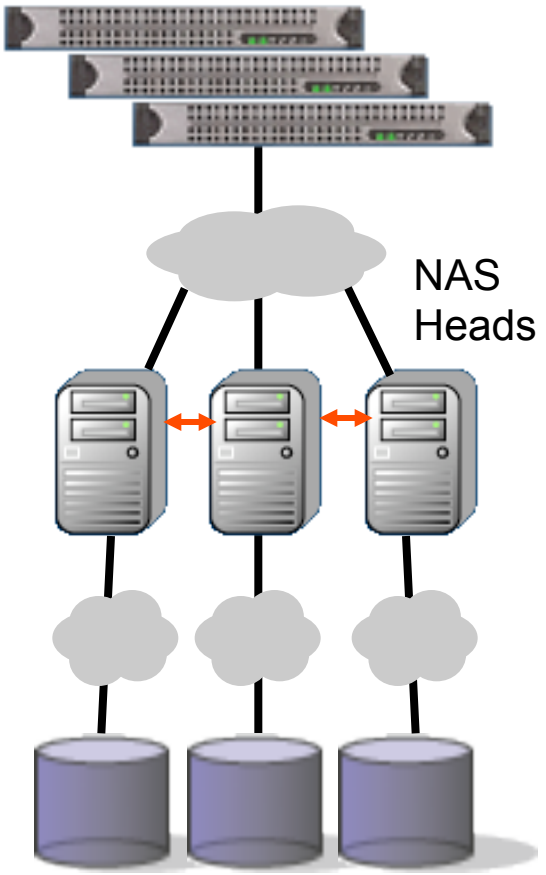
Network Attached Storage (NAS)

- File server exports local filesystem using a file-oriented protocol
  - NFS & CIFS are widely deployed
  - HTTP/WebDAV? FTP?
- Scalability limited by server hardware
  - Uses same building blocks (CPU, RAM, I/O and memory buses) as clients
  - Handles moderate number of clients
  - Handles moderate amount of storage
- A nice model until it runs out of steam
  - “Islands of storage”
  - Bandwidth to a file limited by server bottleneck



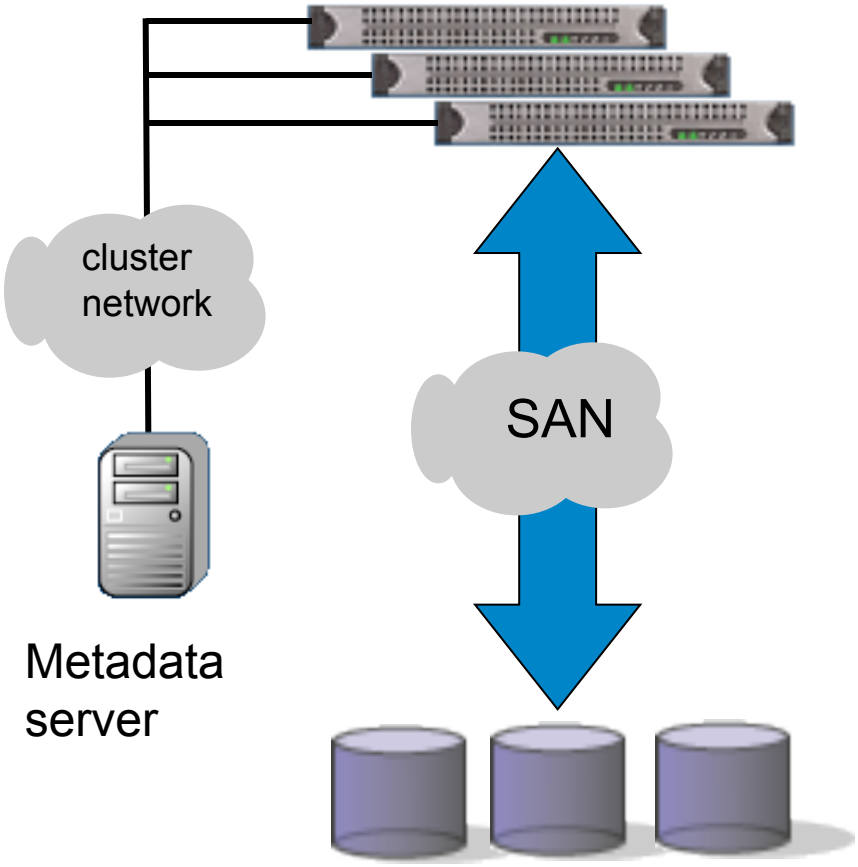
Clustered NAS

- More scalable than single-headed NAS
  - Multiple NAS heads control back-end storage
  - “In-band” NAS head still limits performance and drives up cost
- Two primary architectures
  - Private storage, forward requests to owner (pictured)
  - Re-export SAN file system via NAS protocol
- NFS shortcomings for HPC
  - No good mechanism for dynamic load balancing
  - Poor coherency (or no client caching)
  - No parallel access to data (until pNFS)
- Isilon, NetApp GX, BlueArc, AFS



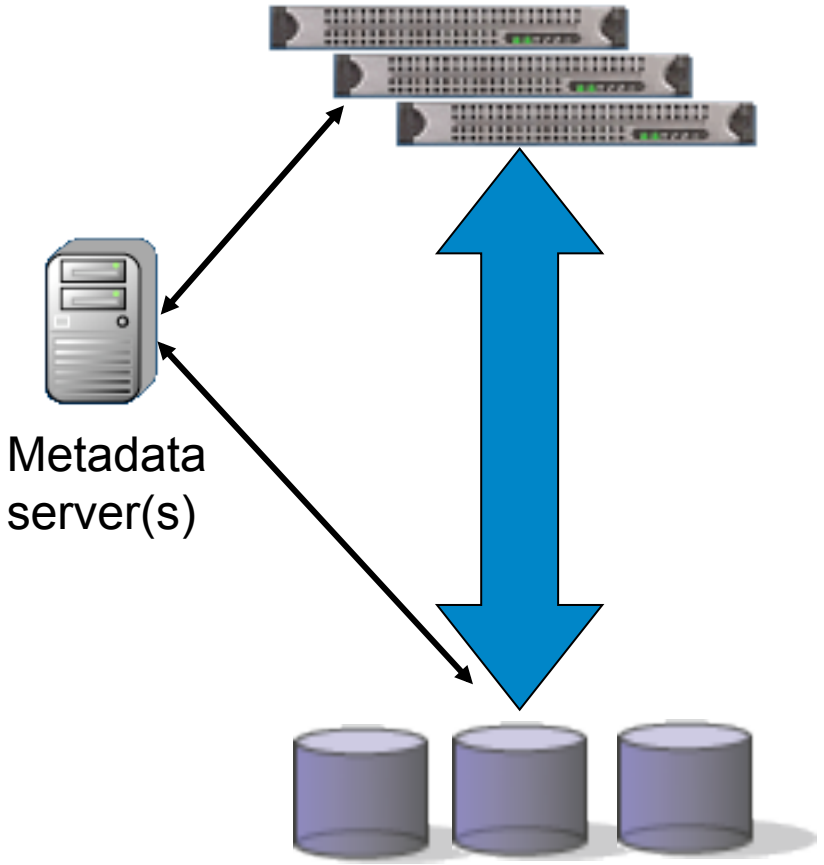
SAN Shared Disk File Systems

- SAN provides common management and provisioning for host storage
  - Block devices accessible via iSCSI/FC
  - Wire-speed performance potential
- Originally for local host FS
- Extended to create shared file system
  - Asymmetric (pictured): separate metadata server manages blocks (and sometimes inode operations)
  - Symmetric: all nodes share metadata & block management
  - Reads & writes go direct to storage via the SAN
- NAS access can be provided by “file head” client node(s) that re-export the SAN file system via NAS protocol
- IBM GPFS, Sun QFS, SGI CXFS



Object-based Storage Clusters

- Object Storage Devices
  - High-level interface (inode/file-like)
  - Block management inside the device
  - Some variants include security
  - OSD standard (SCSI T10)
- File system layered over objects
  - Metadata server manages namespace and external security
  - OSD manages block allocation and internal security
  - Out-of-band data transfer directly between OSDs and clients
- High performance through clustering
  - Scalable to thousands of clients
  - 55+ GB/sec demonstrated to single filesystem



Object Storage Architecture

- Raises storage's level of abstraction
  - From logical blocks to objects (object is a container for data and attributes)
  - Allows storage to understand how different blocks of a object are related
  - Provides storage with necessary info to optimize storage resources
- An evolutionary improvement to standard (SCSI) storage interface

Block Based Disk

Operations:

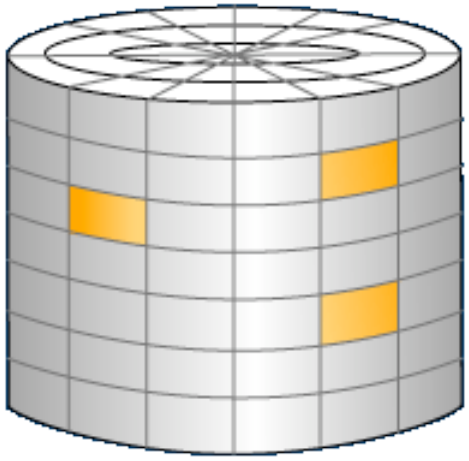
Read block
Write block

Addressing:

Block range

Allocation:

External



Object Based Disk

Operations:

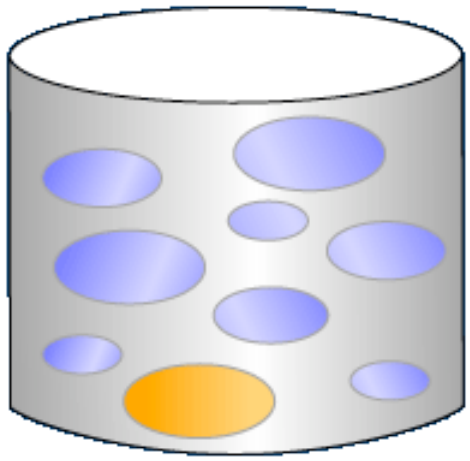
Create object
Delete object
Read object
Write object
Get Attribute
Set Attribute

Addressing:

[object, byte range]

Allocation:

Internal



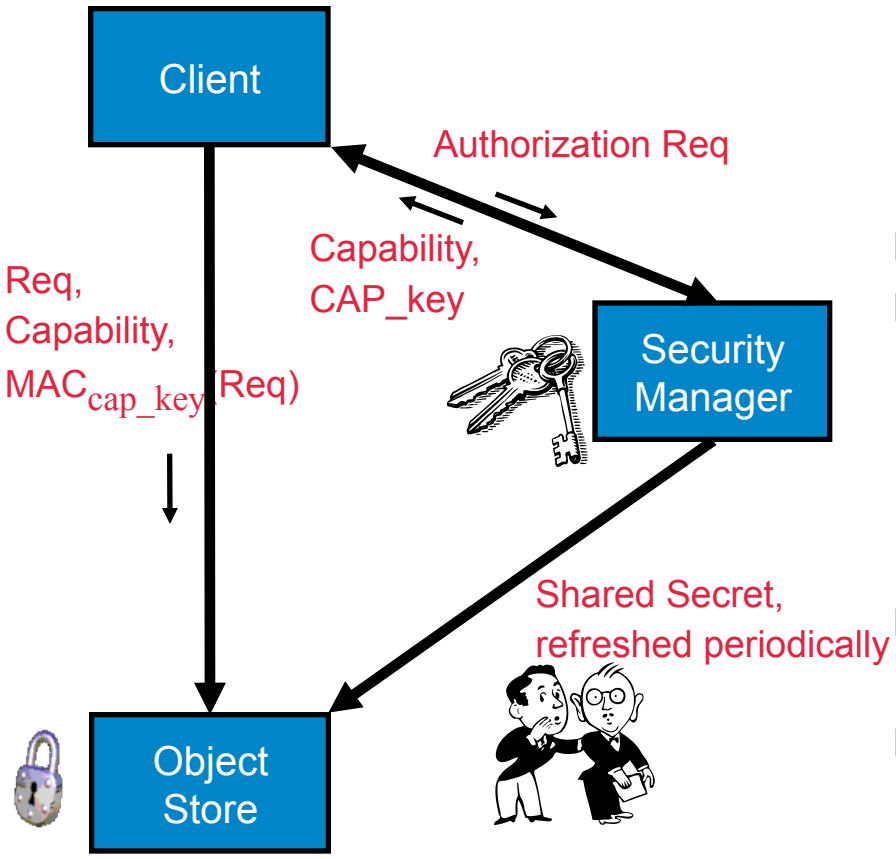
Wide Variety of Object Storage Devices

- Disk array subsystem
- Lustre, PVFS

- Panasas StorageBlade
- 2 SATA disks, CPU and GE NIC

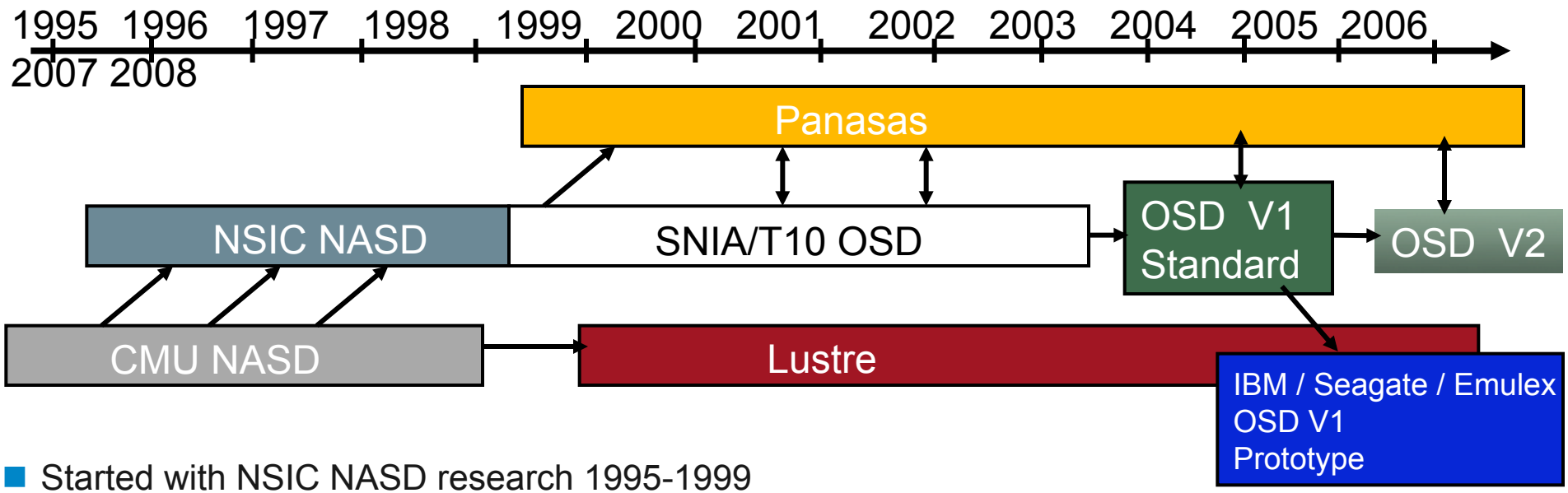
- Prototype Seagate OSD
- Highly integrated, single disk

T10 OSD Security Model



- All operations are secured by a capability
  - Is the command valid?
  - Is the command allowed to access the specified object ?
- Manager and OSD are trusted
- Security achieved by:
  - Manager – authenticates/ authorizes clients and generates credentials.
  - OSD – validates credential that a client presents.
- Credential is signed
  - OSD and Manager share a secret
- POLICY ACCESS TAG attribute allows fine-grained access revocation

Object Storage Standardization



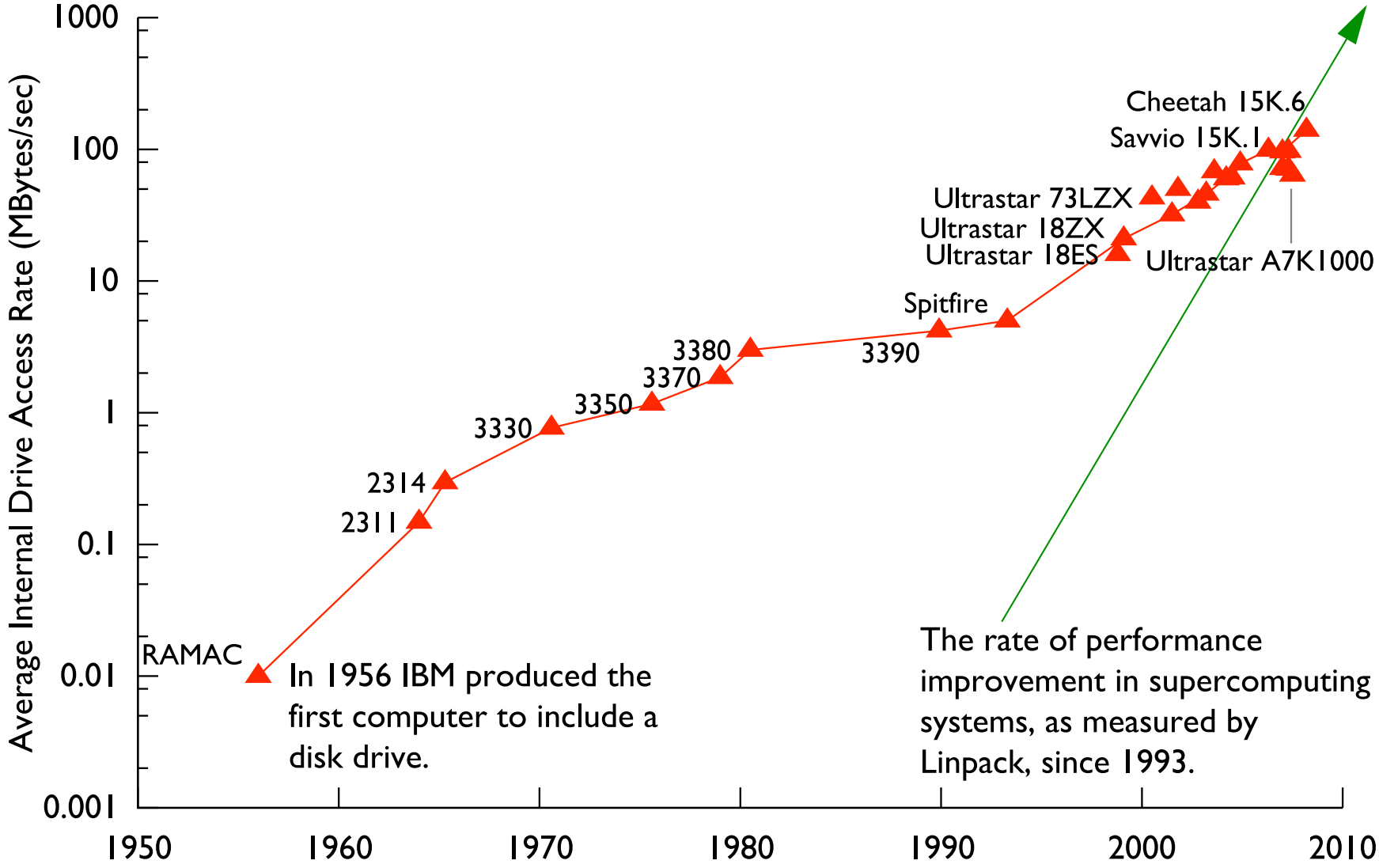
- Started with NSIC NASD research 1995-1999
  - HP, IBM, Quantum, STK, Seagate, and CMU
  - Eventually became SNIA Technology working group in '99
    - 45 participating companies
- 1999 moves to SNIA/T10 working group
- 1/2005: ANSI ratifies V1 T10 OSD standard (ANSI/INCITS 400-2004)
  - SNIA TWG finalizing OSD V2 features (target mid '08)
    - Snapshots, import/export, multi-object capabilities and extended attributes

Strengths of Object Storage

- Object maintains data relationship within OSD
  - Decisions on data layout can be optimized based on object size and usage
  - OSD can be self-organizing, self-optimizing
- Extensible attributes
  - Built-in: size, timestamps, etc.
  - Filesystem defined: owner, ACLs, etc.
  - Application defined: HSM tags, content metadata, etc.
- Access credentials are signed, cached at clients, enforced at device
  - Clients can be untrusted (bugs & attacks expose only authorized object data)
  - Protocol encodes security decisions, not policy
- Command set works with SCSI architecture model (SAM)
  - Encourages cost-effective implementation by storage device vendors
  - Protocol designed with embedded system restrictions in mind

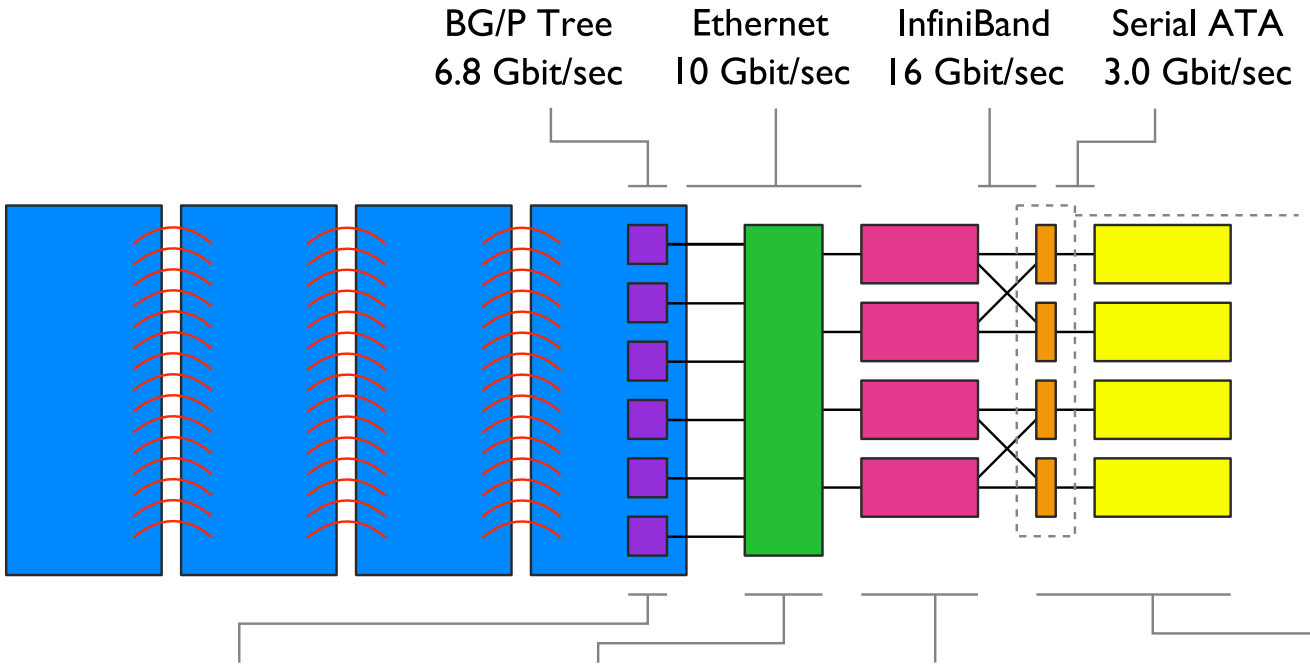
Parallel File Systems

Disk Access Rates over Time



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

Blue Gene/P Parallel Storage System



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

640 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each

Commodity network primarily carries storage traffic.

900+ port 10 Gigabit Ethernet Myricom switch complex

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

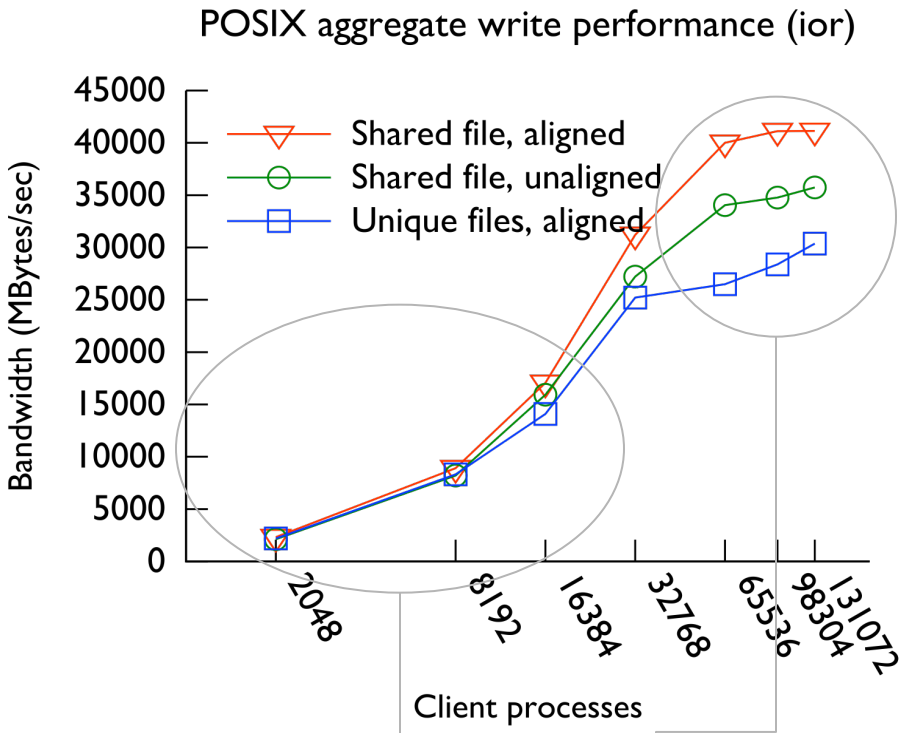
136 two dual core Opteron servers with 8 Gbytes of RAM each

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

17 DataDirect S2A9900 controller pairs with 480 1 Tbyte drives and 8 InfiniBand ports per pair

Architectural diagram of the 557 TFlop IBM Blue Gene/P system at the Argonne Leadership Computing Facility.

Snapshot of Performance on Blue Gene/P

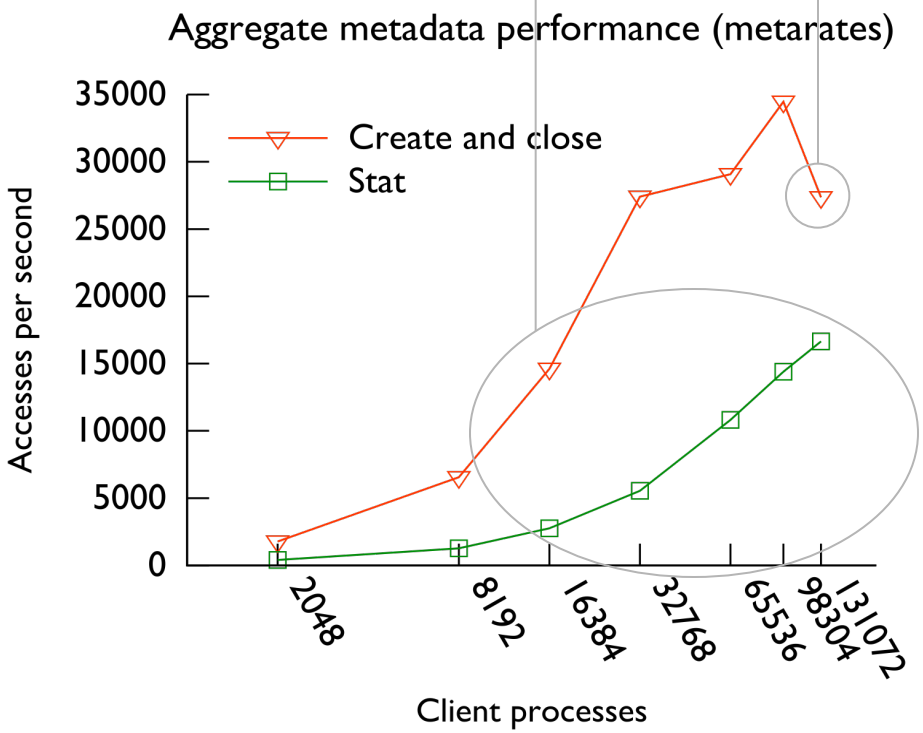


Maximum I/O rate of 300 Mbytes/sec per I/O forwarding node limits performance in this region.

Effective BW out of storage racks limits performance in this region (writing to /dev/null achieves around 65 Gbytes/sec).

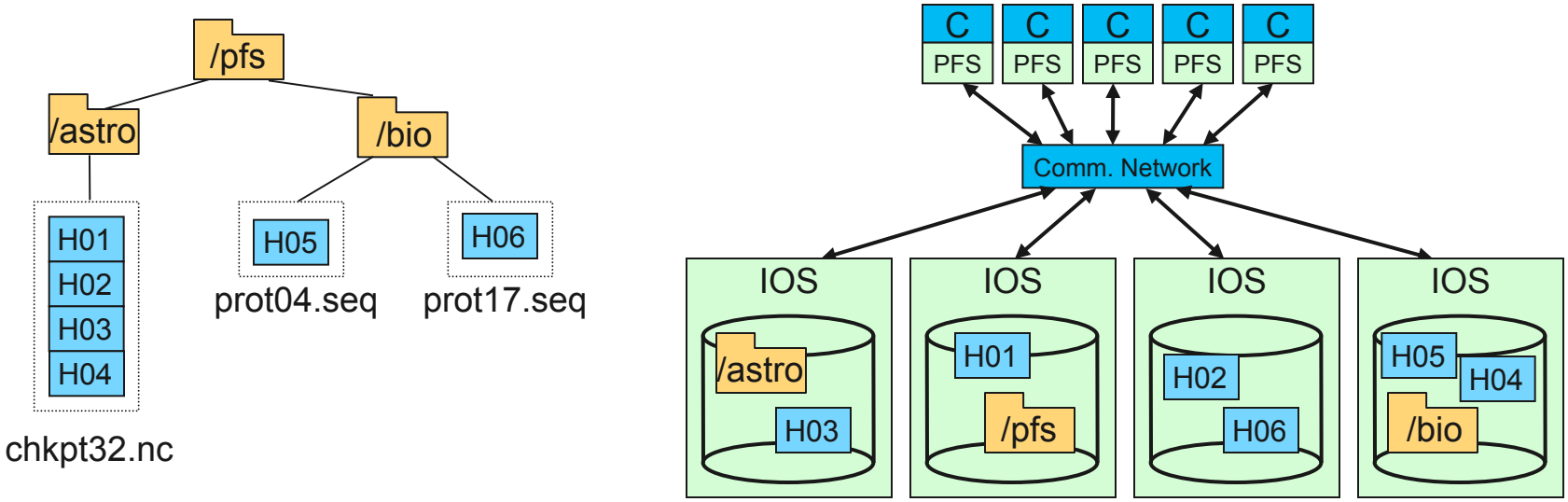
Low stat performance relative to create may be due to poor choice of server-side cache size (256 Kbytes)?

We believe this drop is due to a disk going bad in a storage rack; waiting on repeat testing to confirm.



Lang et. al, "I/O Performance Challenges at Leadership Scale", to appear in SC09, November 2009.

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.

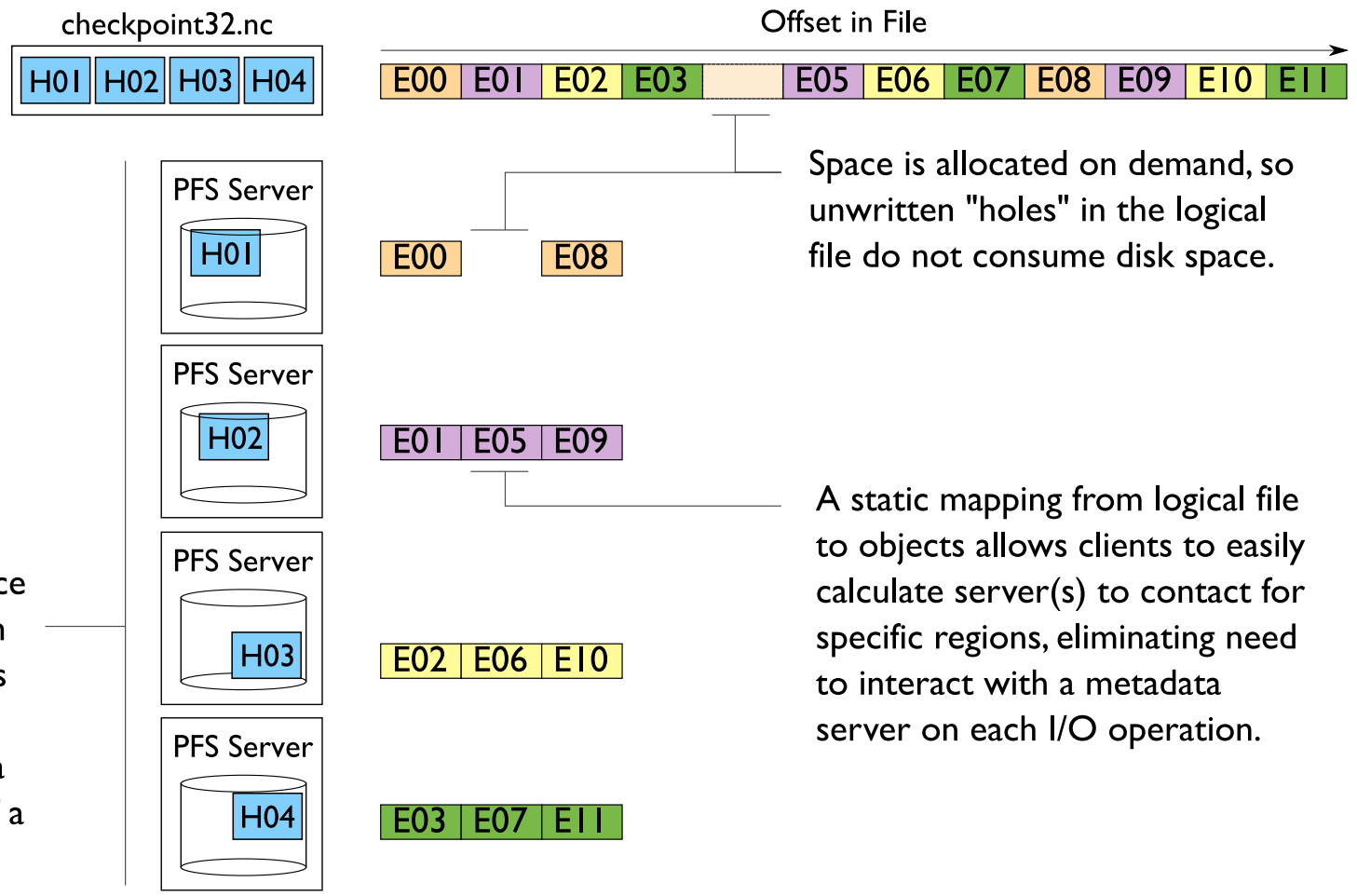
- Building block for HPC I/O systems
  - Present storage as a single, logical storage unit
  - Stripe files across disks and nodes for performance
  - Tolerate failures (in conjunction with other HW/SW)

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.

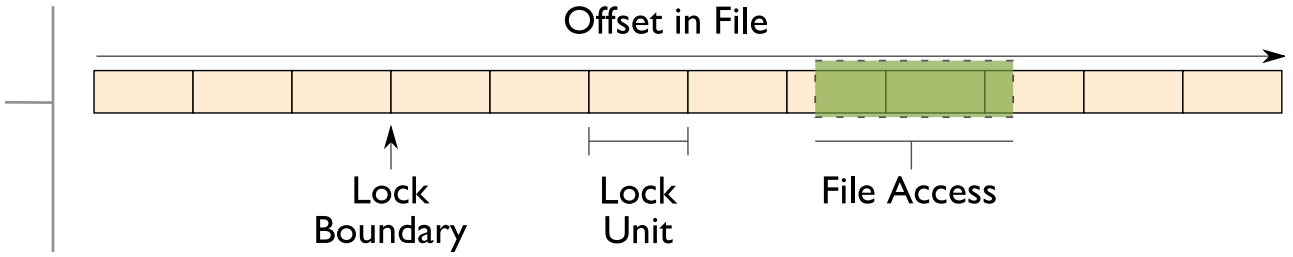
A static mapping from logical file to objects allows clients to easily calculate server(s) to contact for specific regions, eliminating need to interact with a metadata server on each I/O operation.

Locking in Parallel File Systems

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

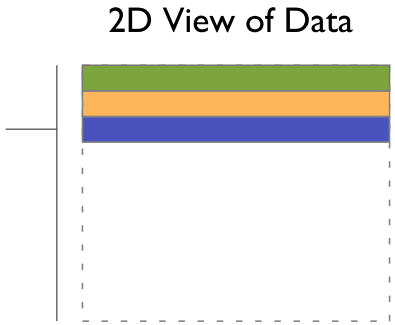
- Files are broken up into lock units
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access

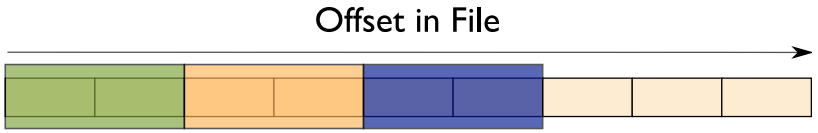
If an access touches any data in a lock unit, the lock for that region must be obtained before access occurs.



Locking and Concurrent Access

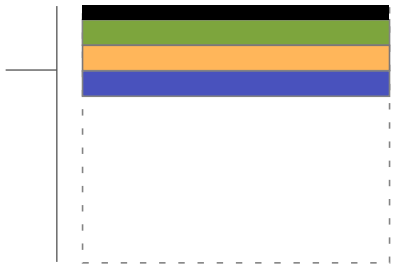
The left diagram shows a row-block distribution of data for three processes. On the right we see how these accesses map onto locking units in the file.

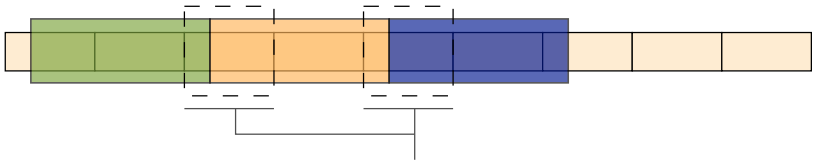




When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.

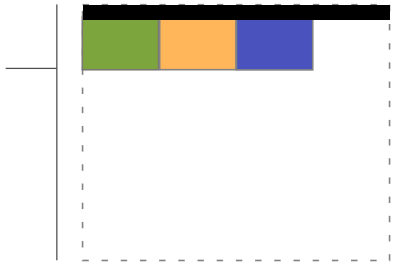
In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.

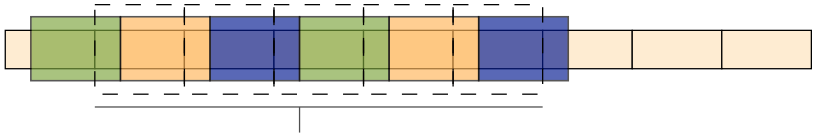




These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

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



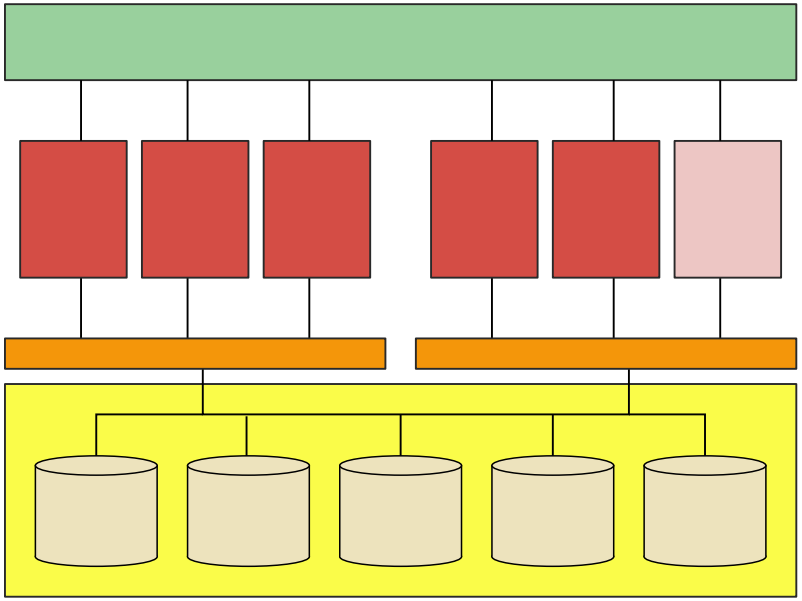


When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

Fault Tolerance and Parallel File Systems

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

- RAID techniques hide disk failures
- Redundant controllers and shared access to storage
- Heartbeat software and quorum directs server failover



System network connects storage to compute resources.

Storage servers manage independent portions of a shared storage resource. In this diagram, five of six servers are active, while one is passive (a backup).

Storage controllers group individual drives into logical units (LUNs) and use RAID techniques to hide drive failures.

LUNs are accessible by all storage servers, but only one server accesses any LUN at one time.

Production Parallel File Systems

- All four systems scale to support the very largest compute clusters
  - LLNL Purple, LANL RoadRunner, Sandia Red Storm, etc.
- All but GPFS delegate block management to "object-like" data servers or OSDs
- Approaches to metadata vary
- Approaches to fault tolerance vary
- Emphasis on features, "turn-key" deployment, vary

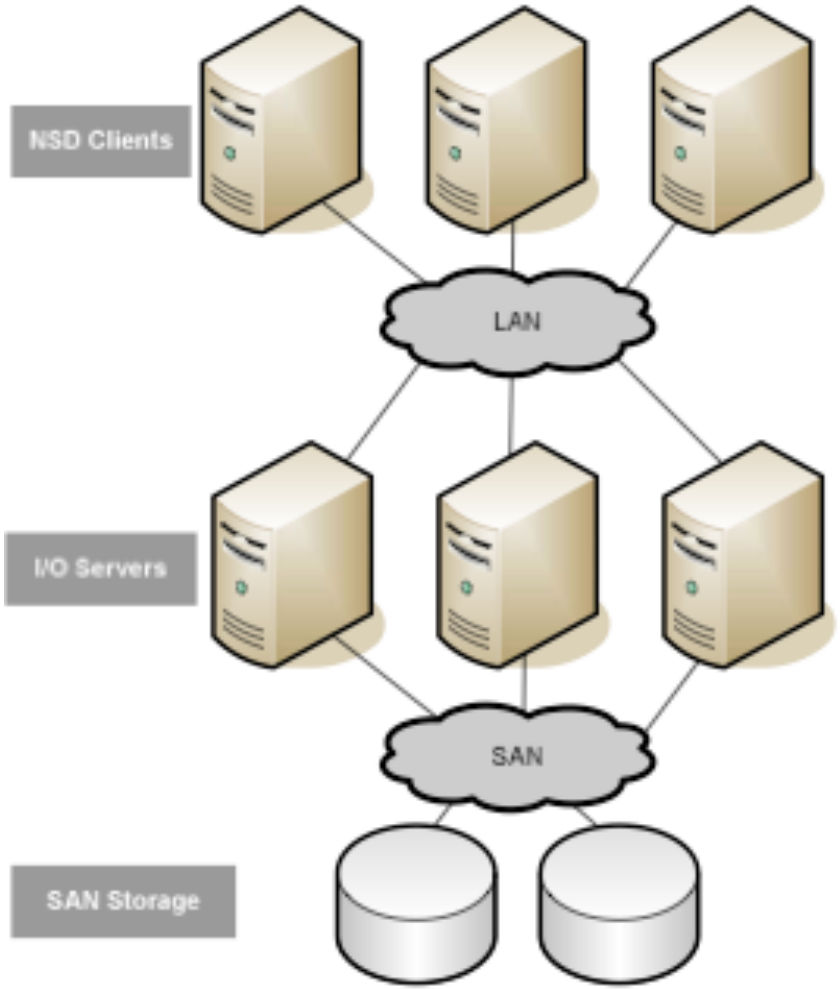
GPFS

panasas®

lustre®

IBM GPFS

- General Parallel File System
- Legacy: IBM Tiger multimedia filesystem
- Commercial product
- 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



GPFS: Block Allocation

- I/O server exports exports local disk via block-oriented NSD protocol
- Block allocation map shared by all nodes
  - Block map split into N regions
  - Each region has 1/Nth of each I/O server's blocks
- Writing node performs block allocation
  - Locks a region of the block map to find free blocks
  - Updates inode & indirect blocks
  - If # regions ~= # client nodes, block map sharing reduced or eliminated
- Stripe each block across multiple I/O servers (RAID-0) for performance
- Large block size (1-4 MB) typically used
  - Increases transfer size per I/O server
  - Minimizes block allocation overhead
  - Not great for small files

GPFS: Metadata Management

- Symmetric model with distributed locking
- Each node acquires locks and updates metadata structures itself
- Global token manager manages locking assignments
  - Client accessing a shared resource contacts token manager
  - Token manager gives token to client, or tells client current holder of token
  - Token owner manages locking, etc. for that resource
  - Client acquires read/write lock from token owner before accessing resource
- inode updates optimized for multiple writers
  - Shared write lock on inode
  - “Metanode token” for file controls which client writes inode to disk
  - Other clients send inode updates to metanode, which merges them

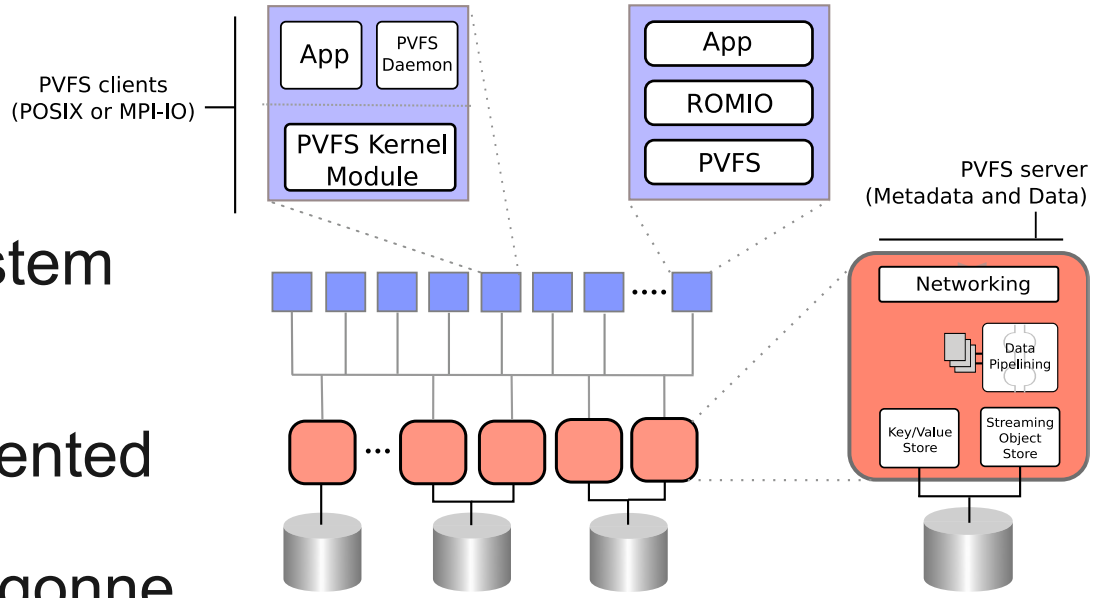
GPFS: Caching

- Clients cache reads and writes
- Strong coherency, based on distributed locking
- Client acquires R/W lock before accessing data
- Optimistic locking algorithm
  - First node accesses 0-1023, locks 0…EOF
  - Second node accesses 1024-2047
    - First node reduces its lock to 0…1023
    - Second node locks 1024…EOF
  - Lock splitting assumes client will continue accessing in current pattern (forward or backward sequential)
- Client cache (“page pool”) pinned and separate from OS page/buffer cache

GPFS: Reliability

- RAID underneath I/O server to handle disk failures & sector errors
- Replication across I/O servers supported, but typically only used for metadata
- I/O server failure handled via dual-attached RAID or SAN
  - Backup I/O server takes over primary's disks if it fails
  - Can designate up to 8 potential owners for a disk (serial failover)
- Nodes journal metadata updates before modifying FS structures
  - Journal is per-node, so no sharing/locking issues
  - Journal kept in shared storage (i.e., on the I/O servers)
  - If node crashes, another node replays its journal to make FS consistent
- Quorum/consensus protocol to determine set of "online" nodes
  - Disk leases or SCSI-3 persistent reservations used for fencing

PVFS



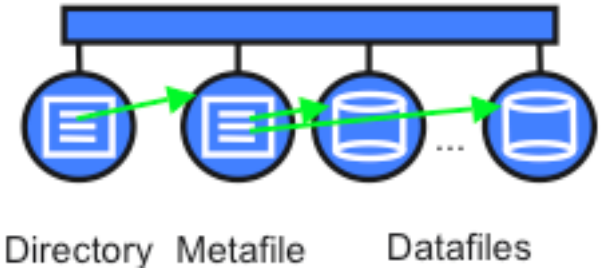
- Parallel Virtual File System
  - Version 2
- Open source, Linux oriented
- Development led by Argonne National Laboratory
  - Supported by many other institutions & companies
- Asymmetric architecture (data servers & clients)
- Data servers use object-like API
- Focused on needs of HPC applications
  - Interface optimized for MPI-IO semantics, not POSIX

PVFS: Block Allocation

- I/O server exports file/object oriented API
  - Storage object (“dataspace”) on an I/O server addressed by numeric handle
  - Dataspace can be stream of bytes or key/value pairs
  - Create dataspace, delete dataspace, read/write
- Files & directories mapped onto dataspaces
  - File may be single dataspace, or chunked/striped over several
- Each I/O server manages block allocation for its local storage
- I/O server uses local filesystem (ext3, XFS, etc.) to store dataspaces
- Key/value dataspace stored using Berkeley DB table

PVFS: Metadata Management

- Directory dataspace contains list of names & metafile handles
- Metafile dataspace contains
  - Attributes (permissions, owner, xattrs)
  - Distribution function parameters
  - Datafile handles
- Datafile(s) store file data
  - Distribution function determines pattern
  - Default is 64 KB chunk size and round-robin placement
- Directory and metadata updates are atomic
  - Eliminates need for locking
  - May require “losing” node in race to do significant cleanup
- System configuration (I/O server list, etc.) stored in static file on all I/O servers



PVFS: Caching

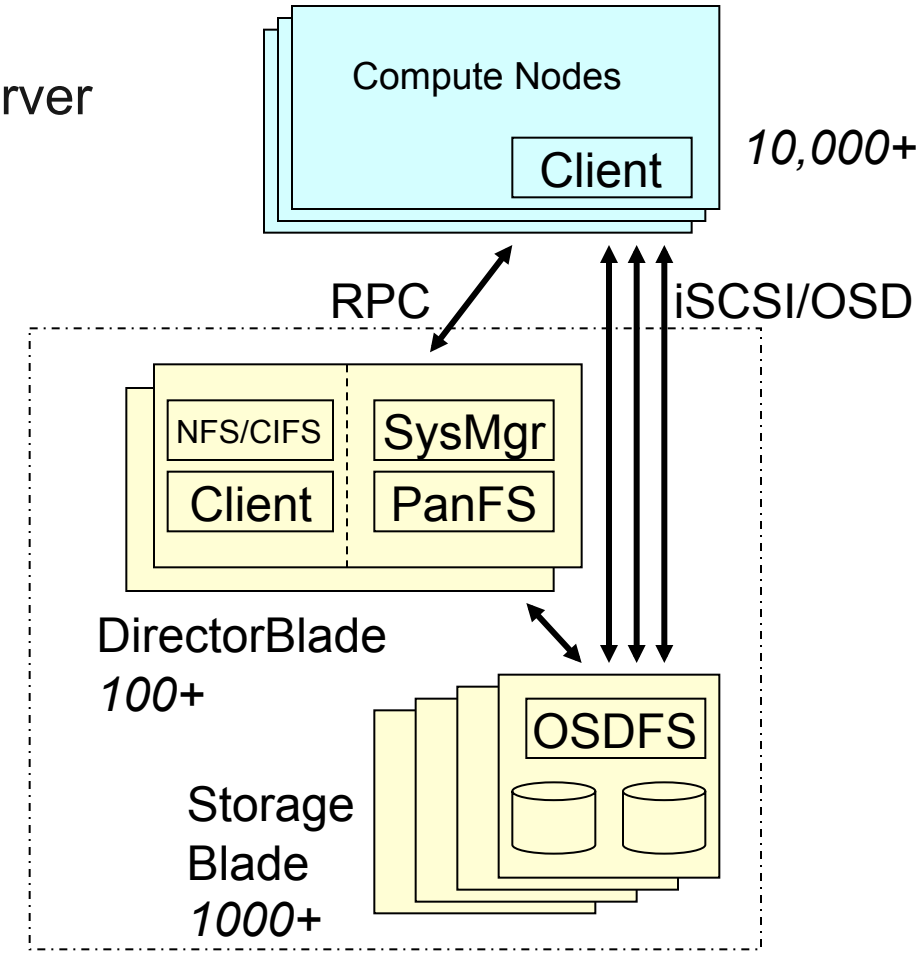
- Client only caches immutable metadata and read-only files
- All other I/O (reads, writes) go through to I/O node
- Strong coherency (writes are immediately visible to other nodes)
- Flows from PVFS2 design choices
  - No locking
  - No cache coherency protocol
- I/O server can cache data & metadata for local dataspaces
- All prefetching must happen on I/O server
- Reads & writes limited by client's interconnect

PVFS: Reliability

- Similar to GPFS
  - RAID underneath I/O server to handle disk failures & sector errors
  - Dual attached RAID to primary/backup I/O server to handle I/O server failures
- Linux HA used for generic failover support
  - Remote control power strip (STONITH) for fencing
- Sequenced operations provide well-defined crash behavior
  - Example: Creating a new file
    - Create datafiles
    - Create metafile that points to datafiles
    - Link metafile into directory (atomic)
  - Crash can result in orphans, but no other inconsistencies

Panasas ActiveScale (PanFS)

- Commercial product based on CMU NASD research
- Complete "appliance" solution (HW + SW), blade server 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: Block Allocation

- OSD exports object-oriented API
  - Objects have a number (object ID), data, and attributes
  - CREATE OBJECT, REMOVE OBJECT, READ, WRITE, GET ATTRIBUTE, SET ATTRIBUTE, etc.
  - Commands address object ID and data range in object
  - Capabilities provide fine-grained revocable access control
- OSD manages private local storage
  - Two SATA drives, 500/750/1000 GB each, 1-2 TB total capacity
- Specialized filesystem (OSDFS) stores objects
  - Delayed floating block allocation
  - Efficient copy-on-write support
- Files and directories stored as “virtual objects”
  - Virtual object striped across multiple container objects on multiple OSDs

PanFS: Metadata Management

- Directory is a list of names & object IDs in a RAID-1 virtual object
- Filesystem metadata stored as object attributes
  - Owner, ACL, timestamps, etc.
  - Layout map describing RAID type & OSDs that hold the file
- Metadata server (DirectorBlade)
  - Checks client permissions & provides map/capabilities
  - Performs namespace updates & directory modifications
  - Performs most metadata updates
- Client modifies some metadata directly (length, timestamps)
- Coarse-grained metadata clustering based on directory hierarchy

PanFS: Caching

- Clients cache reads & writes
- Strong coherency, based on callbacks
  - Client registers callback with metadata server
  - Callback type identifies sharing state (unshared, read-only, read-write)
  - Server notifies client when file or sharing state changes
- Sharing state determines caching allowed
  - Unshared: client can cache reads & writes
  - Read-only shared: client can cache reads
  - Read-write shared: no client caching
  - Specialized “concurrent write” mode for cooperating apps (e.g. MPI-IO)
- Client cache shared with OS page/buffer cache

PanFS: Reliability

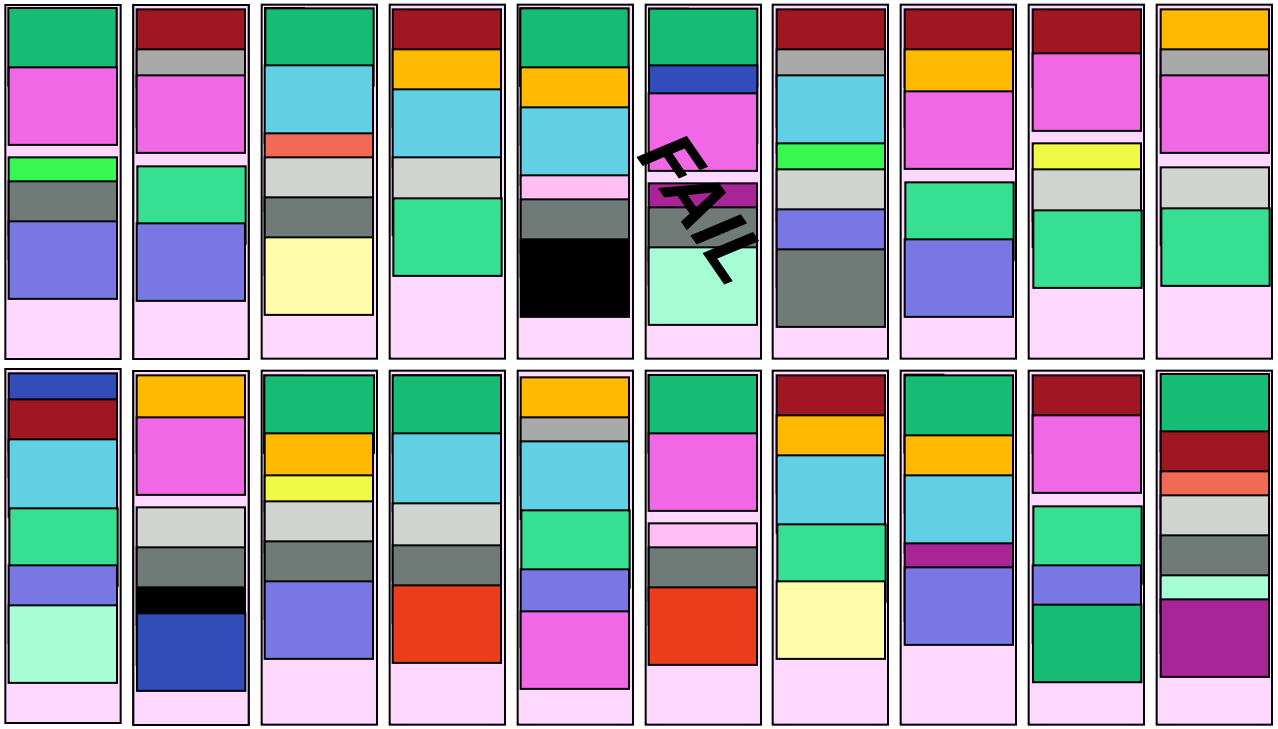
- RAID-1 & RAID-5 across OSDs to handle disk failures
  - Any failure in StorageBlade (disk, RAM, CPU) is handled via rebuild
  - Declustered parity allows scalable rebuild
- "Vertical parity" inside OSD to handle sector errors
- Integrated shelf battery makes all RAM in blades into NVRAM
  - Metadata server journals updates to in-memory log
    - Failover config replicates log to 2nd blade's memory
    - Log contents saved to DirectorBlade's local disk on panic or power failure
  - OSDFS commits updates (data+metadata) to in-memory log
    - Log contents committed to filesystem on panic or power failure
    - Disk writes well ordered to maintain consistency
- System configuration in replicated database on subset of DirectorBlades

PanFS: Declustered RAID

- Each file striped across different combination of StorageBlades
- Component objects include file data and file parity
- File attributes replicated on first two component objects
- Components grow & new components created as data written
- Declustered, randomized placement distributes RAID workload

20 OSD Storage Pool

Mirrored or 9-OSD Parity Stripes



Read about half of each surviving OSD

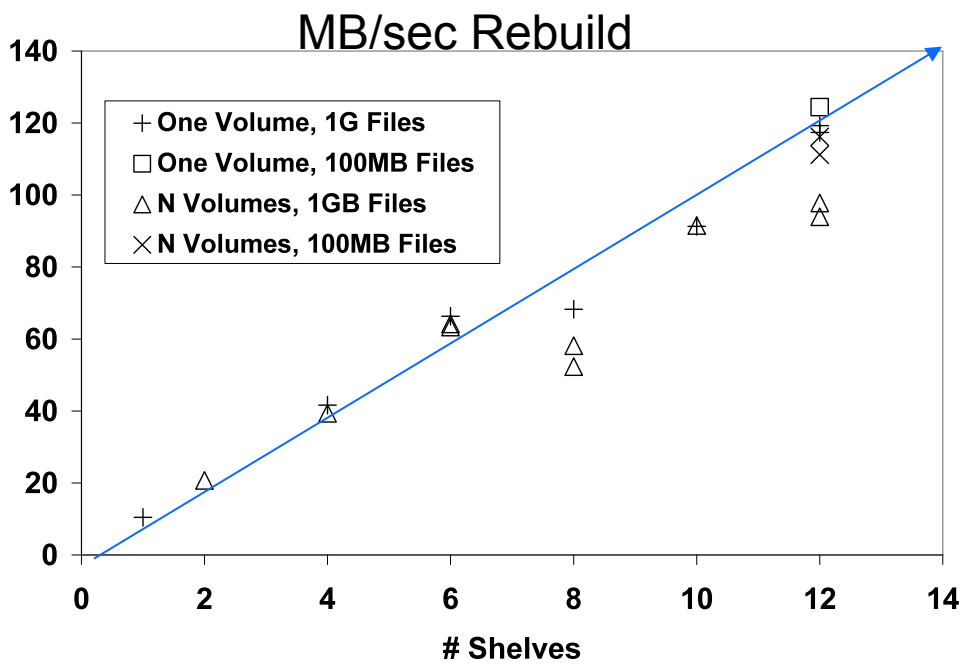
Write a little to each OSD

Scales up in larger Storage Pools

Panasas Scalable Rebuild

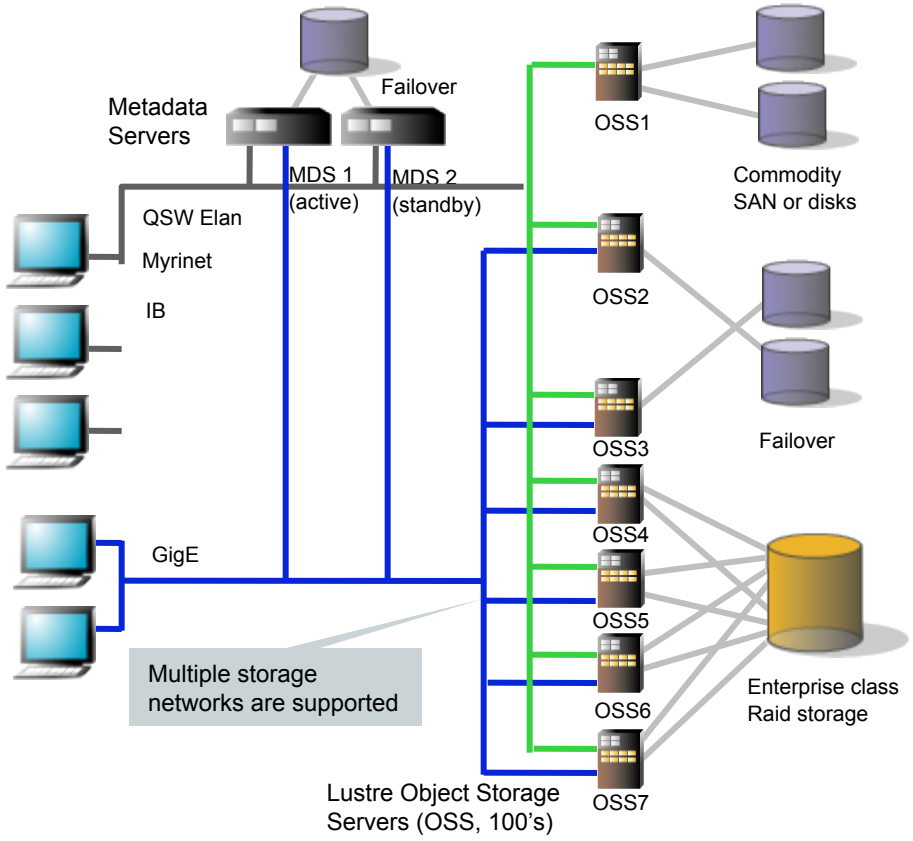
- Two main causes of RAID failures
  - **1) 2nd drive failure in same RAID set during reconstruction of 1st failed drive**
    - Risk of two failures depends on time-to-repair
  - 2) Media failure in same RAID set during reconstruction of 1st failed drive

- Shorter repair time in larger storage pools
  - From 13 hours to 30 minutes
- Four techniques to reduce MTTR
  - Use multiple "RAID engines" (DirectorBlades) in parallel
  - Spread disk I/O over more disk arms (StorageBlades)
  - Reconstruct data blocks only, not unused space
  - Proactively remove failing blades (SMART trips, other heuristics)



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
- Originally Linux-based, Sun now porting 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: Block Allocation

- Each OSS (object storage server) manages one or more OSTs (object storage target)
  - Typically 2-25 OSTs per OSS (max OST size 8 TB)
  - Client communicates with OSS via proprietary RPC protocol
    - RPC built on LNET message-passing facility (based on Sandia Portals)
    - LNET supports RDMA over IB, Myrinet, and Quadrics Elan
- OST stores data in modified ext3 file system
- Currently porting OST to ZFS
  - User-level ZFS via FUSE on Linux
  - In-kernel ZFS on Solaris
- RAID-0 striping across OSTs
  - No dynamic space management among OSTs (i.e., no object migration to balance capacity)
- Snapshots and quota done independently in each OST

Lustre: Metadata

- Metadata server (MDS) hosts metadata target (MDT), which stores namespace tree and file metadata
- MDT uses a modified ext3 filesystem to store Lustre metadata
  - Directory tree of "stub" files that represents Lustre namespace
  - Lustre metadata stored in stub file's extended attributes
    - Regular filesystem attributes (owner, group, permissions, size, etc.)
    - List of object/OST pairs that contain file's data (storage map)
  - Single MDS and single MDT per Lustre filesystem
  - Clustered MDS with multiple MDTs is on roadmap (Lustre 2.0)
- Distributed lock protocol among MDS, OSS, and clients
  - "Intents" convey hints about the high-level file operations so the right locks can be taken and server round-trips avoided
  - If a failure occurs (MDS or OSS), clients do lock recovery after failover

Lustre: Caching

- Clients can cache reads, writes, and some metadata operations
- Locking protocol used to protect cached data and serialize access
  - OSS manages locks for objects on its OSTs
  - MDS manages locks on directories & inodes
  - Client caches locks and can reuse them across multiple I/Os
  - MDS/OSS recalls locks when conflict occurs
  - Lock on logical file range may span several objects/OSTs if file is striped
- Directory locks allow client to do CREATE without round-trip to MDS
  - Only for unshared directory
  - Create not “durable” until file is written & closed
  - Non-POSIX semantic but helpful for many applications
- Client cache shared with OS page/buffer cache

Lustre: Reliability

- Block-based RAID underneath OST/MDT
- Failover managed by external software (Red Hat Cluster Manager, Linux-HA, etc.)
- OSS failover (active/active or clustered)
  - OSTs on dual-ported RAID controller
  - OSTs on SAN with connectivity to all OSS nodes
- MDS failover (active/passive)
  - MDT on dual-ported RAID controller
  - Typically use dedicated RAID for MDT due to different workload
- Crash recovery based on logs and transactions
  - MDS logs operation (e.g., file delete)
  - Later response from OSS cancels log entry
  - Some client crashes cause MDS log rollback
  - MDT & OST use journaling filesystem to avoid fsck
- LNET supports redundant networks and link failover

Design Comparison

	GPFS	PVFS	Panasas	Lustre
Block mgmt	**Shared block map**	Object based	Object based	Object based
Metadata location	With data	With data	With data	**Separate**
Metadata written by	Client	Client	Client, server	Server
Cache coherency & protocol	Coherent; distributed locking	**Cache immutable/ RO data only**	Coherent; callbacks	Coherent; distributed locking
Reliability	Block RAID	Block RAID	**Object RAID**	Block RAID

Other File Systems

- GFS (Google)
  - Single metadata server + 100s of chunk servers
  - Specialized semantics (not POSIX)
    - Relaxed consistency, no concurrent writes
    - Atomic append operation
    - Copy-on-write snapshots
  - Design for failures; all files replicated 3+ times
  - Geared towards colocated processing (MapReduce)
- Ceph (UCSC)
  - OSD-based parallel filesystem
  - Dynamic metadata partitioning between MDSs
  - OSD-directed replication based on CRUSH distribution function (no explicit storage map)
- Clustered NAS
  - NetApp GX, Isilon, BlueArc, etc.

Other Issues

- Reading and writing data is the easy part!
- What about…
  - Monitoring & troubleshooting?
  - Backups?
  - Snapshots?
  - Disaster recovery & replication?
  - Capacity management?
    - Quotas, HSM, ILM
  - System expansion?
  - Retiring old equipment?

Themes

- Scalable clusters require scalable storage
  - Centralized/single anything eventually becomes a bottleneck
- File/object oriented storage API is superior to block oriented
  - Parallel, scalable block allocation
  - Block protocols have poor security & fencing support
  - Block layouts are cumbersome
- Reliability is important
  - Large systems will constantly have something that's broken
  - Tolerating failures is necessary to make forward progress

Benchmarking and Application Performance

Performance Measurement

- Lots of different performance metrics
  - Sequential bandwidth, random I/Os, metadata operations
  - Single-threaded vs. multi-threaded
  - Single-client vs. multi-client
  - N-to-N (file per process) vs. N-to-1 (single shared file)
- Ultimately a method to try to estimate what you really care about
  - "Time to results", aka "How long does my app take?"
- Benchmarks are best if they model your real application
  - Need to know what kind of I/O your app does in order to choose appropriate benchmark
  - Similar to CPU benchmarking – e.g., LINPACK performance may not predict how fast your codes run

What is a benchmark?

- Standardized way to compare performance of different systems
- Properties of a good benchmark
  - Relevant: captures essential attributes of real application workload
  - Simple: Provides an understandable metric
  - Portable & scalable
  - Consistent & repeatable results (on same HW)
  - Accepted by users & vendors
- Types of benchmark
  - Microbenchmark
  - Application-based benchmark
  - Synthetic workload

Microbenchmarks

- Measures one fundamental operation in isolation
  - Read throughput, write throughput, creates/sec, etc.
- Good for:
  - Tuning a specific operation
  - Post-install system validation
  - Publishing a big number in a press release
- Not as good for:
  - Modeling & predicting application performance
  - Measuring broad system performance characteristics
- Examples:
  - IOzone
  - IOR
  - Bonnie++
  - mdtest
  - metarates

Application Benchmarks

- Run real application on real data set, measure time
- Best predictor of application performance on your cluster
- Requires additional resources (compute nodes, etc.)
  - Difficult to acquire when evaluating new gear
  - Vendor may not have same resources as their customers
- Can be hard to isolate I/O vs. other parts of application
  - Performance may depend on compute node speed, memory size, interconnect, etc.
  - Difficult to compare runs on different clusters
- Time consuming – realistic job may run for days, weeks
- May require large or proprietary dataset
  - Hard to standardize and distribute

Synthetic Benchmarks

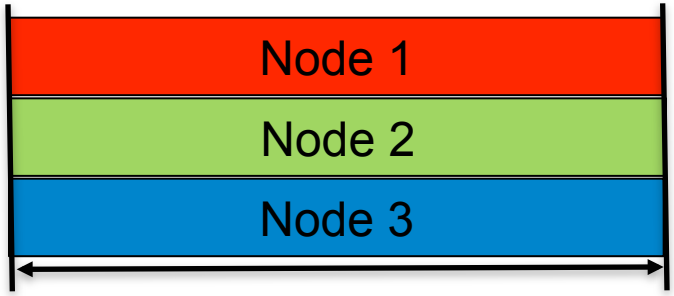
- Selected combination of operations (usually a fractional mix)
  - Operations selected at random or using random model (e.g., Hidden Markov Model)
  - Operations and mix based on traces or sampling real workload
- Can provide better model for application performance
  - However, inherently domain-specific
  - Need different mixes for different applications & workloads
  - The more generic the benchmark, the less useful it is for predicting app performance
  - Difficult to model a combination of applications
- Examples:
  - SPEC SFS
  - TPC-C, TPC-D

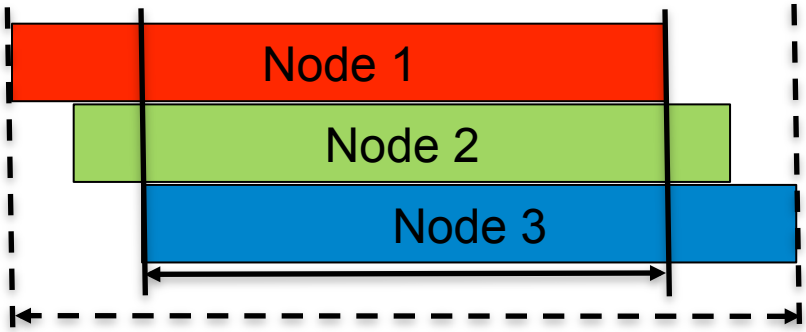
Benchmarks for HPC

- Unfortunately, there are few synthetic HPC benchmarks that stress I/O
- HPC Challenge (http://icl.cs.utk.edu/hpcc/)
  - Seven sub-benchmarks, all "kernel" benchmarks (LINPACK, matrix transpose, FFT, message ping-pong, etc.)
  - Measures compute speed, memory bandwidth, cluster interconnect
  - No I/O measurements
- SPEC HPC2002 (http://www.spec.org/hpc2002/)
  - Three sub-benchmarks (CHEM, ENV, SEIS), all based on real apps
  - Only SEIS has a dataset of any size, and even it is tiny
    - 2 GB for Medium, 93 GB for X-Large
- NAS Parallel Benchmarks (http://www.nas.nasa.gov/Resources/Software/npb.html)
  - Mix of kernel and mini-application benchmarks, all CFD-focused
  - One benchmark (BTIO) does significant I/O (135 GB N-to-1/collective write)
- FLASH I/O Benchmark (http://www-unix.mcs.anl.gov/pio-benchmark/)
  - Simulates I/O performed by FLASH (nuclear/astrophysics application, Net-CDF/HDF5)
- Most HPC I/O benchmarking still done with microbenchmarks
  - IOzone, IOR (LLNL), LANL MPI-IO Test, mdtest, etc.

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 is important
- Figure out the “speed of light” in your system
- Large sequential accesses
  - Readahead can hide latency
  - 7200 RPM SATA 60-100 MB/sec/spindle
  - 15000 RPM FC 100-170 MB/sec/spindle
- 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, 150-200 ops/sec/spindle

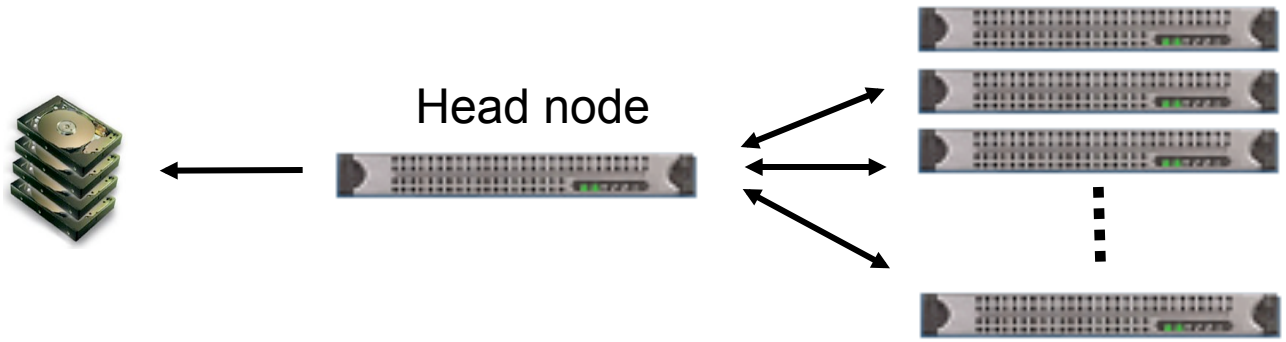
Beware Hidden Bottlenecks

- "I added disks and I/O nodes but my apps aren't running any faster!"

- Common "hidden" bottleneck sources
  - Oversubscribed switch line cards or internal network links
  - NAS head throughput (for in-band filesystems)
  - Cluster head node (for head-node based apps)
  - Cluster CPU or message interconnect (not all jobs are I/O bound)

Head Node I/O

- Traditional storage architecture led to a “head node” app I/O architecture
  - Head node has user logins and access to direct-attached or SAN storage
  - Head node reads data set & transmits it to other nodes
  - Nodes send their results to head node which writes data file to storage
  - Scalability requires an ever faster head node and either a faster local store or a faster (single) file server for the head node’s I/O load
  - Data management can be an issue as datasets are copied to/from the cluster
- Head node processing steps are application bottlenecks

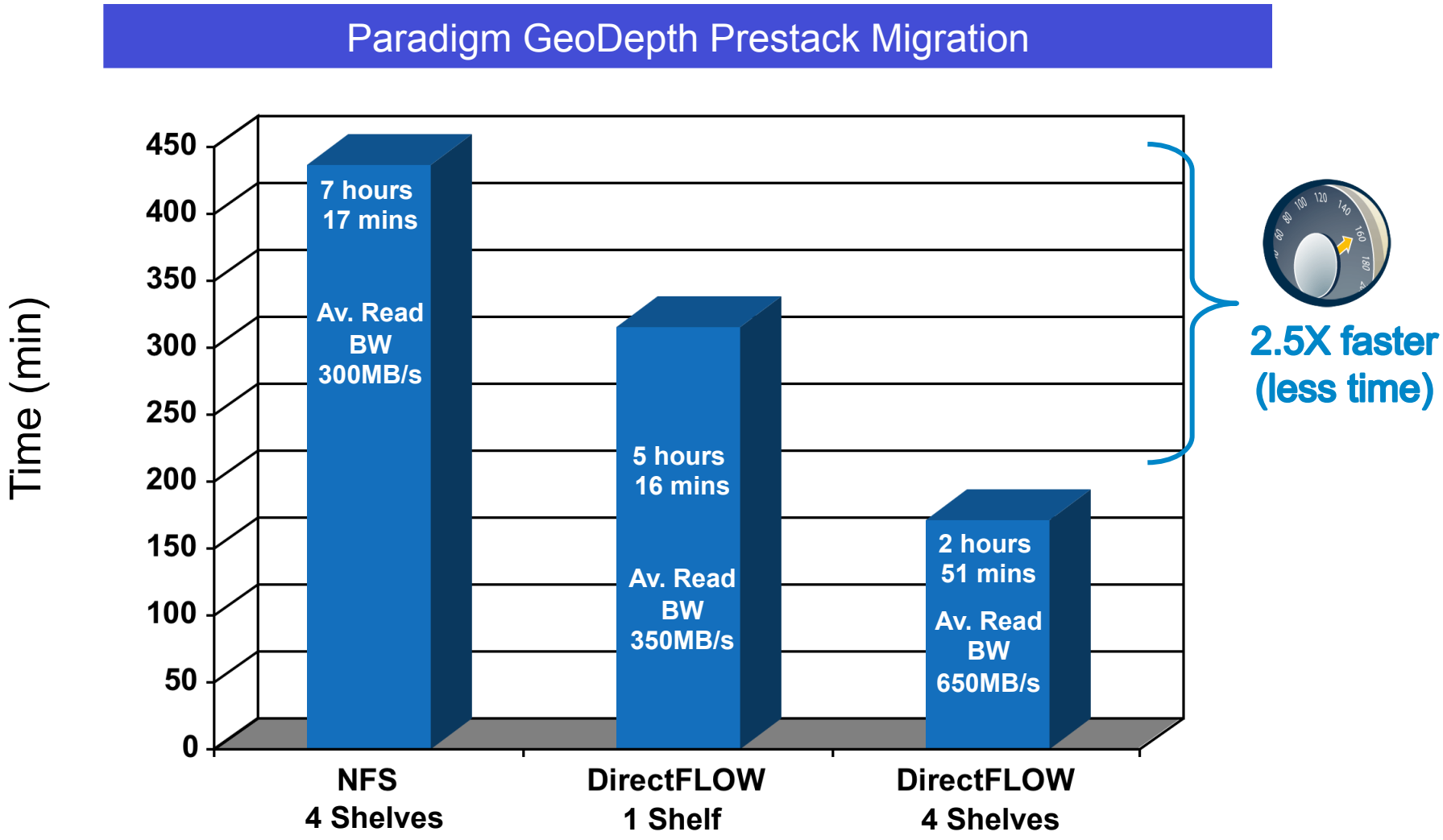


Parallel I/O

- Restructure app to perform I/O in parallel from each compute node
  - Eliminates head node I/O bottleneck
  - Cluster lets you do computation in parallel
  - Without parallel I/O, I/O time dominates as compute time shrinks (bigger/ faster cluster)
- Challenges
  - Requires direct access to storage from all cluster nodes (hard with SAN FS)
  - Easiest way to convert application retains existing file structure (single file for results from all nodes)
  - However, multiple nodes writing to shared file requires coordination
  - Changing application I/O pattern requires source code – hard for COTS apps unless vendor supports both modes
- POSIX API does not provide tools necessary for coordinating I/O
  - Filesystem that provides strict POSIX consistency will sacrifice performance
  - Relaxing semantics can improve performance, but may break applications
  - No standard mechanisms for disclosing caching hints, etc.

Example Parallel I/O Performance Gain

Paradigm™



Source: Paradigm & Panasas, February 2007

Argonne NATIONAL LABORATORY

panasas

PVFS Test Platform: OSC Opteron Cluster

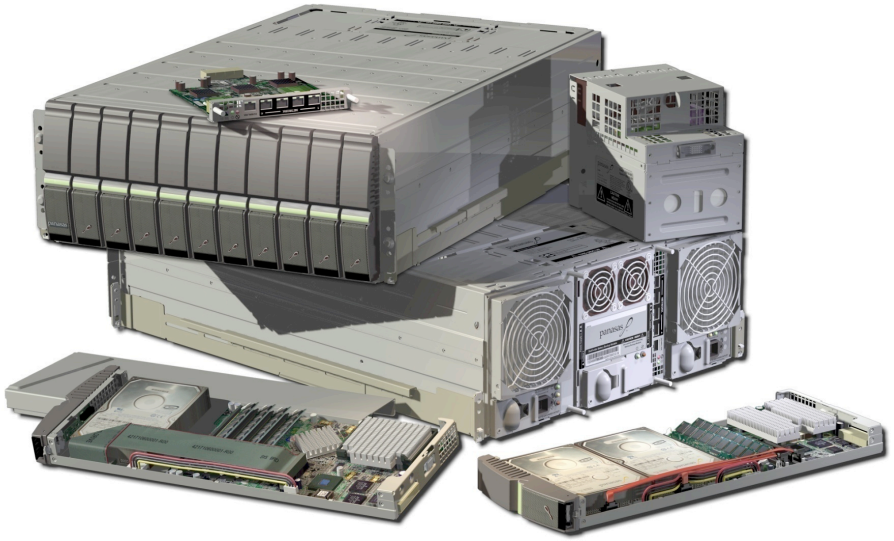
- 338 nodes, each with
  - 4 AMD Opteron CPUs at 2.6 GHz, 8 GB memory
- Gigabit Ethernet network
  - Switch Hierarchy with multiple GBit uplinks
- 16 I/O servers (also serving metadata)
  - 2 2-core Xeon CPU at 2.4 GHz, 3 GB memory
- 120 TB parallel file system
  - Each server has Fibre Channel interconnect to back-end RAID

Ohio Supercomputer Center

Panasas Test Lab

- Hundreds of storage nodes and clients in our lab but of various vintage and for various other tests. We created a small system for these tests:

- 3 Panasas Shelves, each with
  - 10 SB-1000a-XC StorageBlades
    - (1.5GHz Celeron, 2GB, 1TB SATA, 1GE)
  - 1 DB-100a DirectorBlade
    - (1.8GHz 475, 4GB, 1GE)
  - 18-port switch with 10GE uplink
- 48 client nodes
  - 2.8 GHz Xeon, 8GB, 1GE
- GE Backbone
  - 4 GB/sec between clients and shelves



GPFS Test Platform: ASC Purple

- 1536 nodes, each with
  - 8 64-bit Power5 CPUs at 1.9 GHz
  - 32 GB memory
- Federation high-speed interconnect
  - 4Gbyte/sec theoretical bisection bandwidth per adapter
  - ~5.5 Gbyte/sec measured per I/O server w/dual adapters
- 125 I/O servers, 3 metadata servers
  - 8 64-bit Power5 CPUs at 1.9 GHz
  - 32 GB memory
- 300 TB parallel file system
  - HW RAID5 (4+P, 250 GB SATA Drives)
  - 24 RAIDs per I/O server

Lustre Test Platform: LLNL Thunder

- 1024 nodes each with
  - 4 64-bit Itanium2 CPUs at 1.4 GHz
  - 8 GB memory
- Quadrics high-speed interconnect
  - ~900 MB/s of bidirectional bandwidth
  - 16 Gateway nodes with 4 GigE connections to the Lustre network
- 64 object storage servers, 1 metadata server
  - I/O server - dual 2.4 Ghz Xeons, 2GBs ram
  - Metadata Server - dual 3.2 Ghz Xeons, 4 GBs ram
- 170 TB parallel file system
  - HW RAID5 (8+P, 250 GB SATA Drives)
  - 108 RAIDs per rack
  - 8 racks of data disk

Metadata Performance

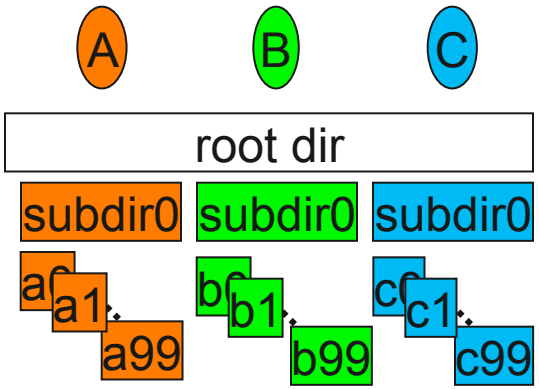
- Storage is more than reading & writing
- Metadata operations change the namespace or file attributes
  - Creating, opening, closing, and removing files
  - Creating, traversing, and removing directories
  - “Stat”ing files (obtaining the attributes of the file, such as permissions and file size)
- Several users 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
- We ran three variations, each with 64 processes:
  - mdtest -d $DIR -n 100 -i 3 -N 1 -v -u
    - Each task creates 100 files in a unique subdirectory
  - mdtest -d $DIR -n 100 -i 3 -N 1 -v -c
    - One task creates 6400 files in one directory
    - Each task opens, removes its own
  - mdtest -d $DIR -n 100 -i 3 -N 1 -v
    - Each task creates 100 files in a single shared directory
- GPFS tests use 16 tasks with 4 tasks on each node
- Panasas tests use 48 tasks on 48 nodes

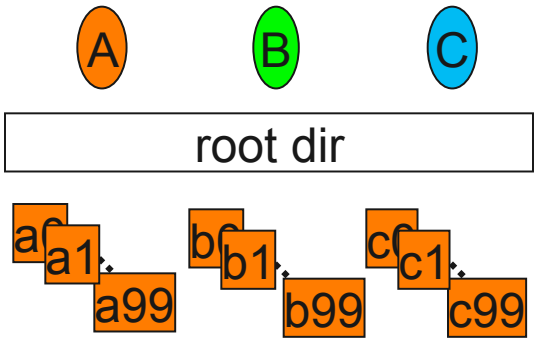
mdtest Variations

Unique Directory



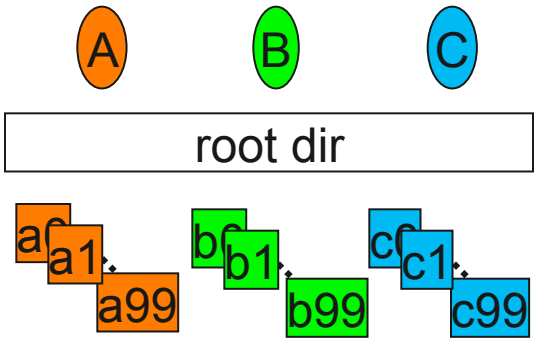
1) Each process (A, B, C) creates own subdir in root directory, then chdirs into it.
2) A, B, and C create, stat, and remove their own files in the unique subdirectories.

Single Process



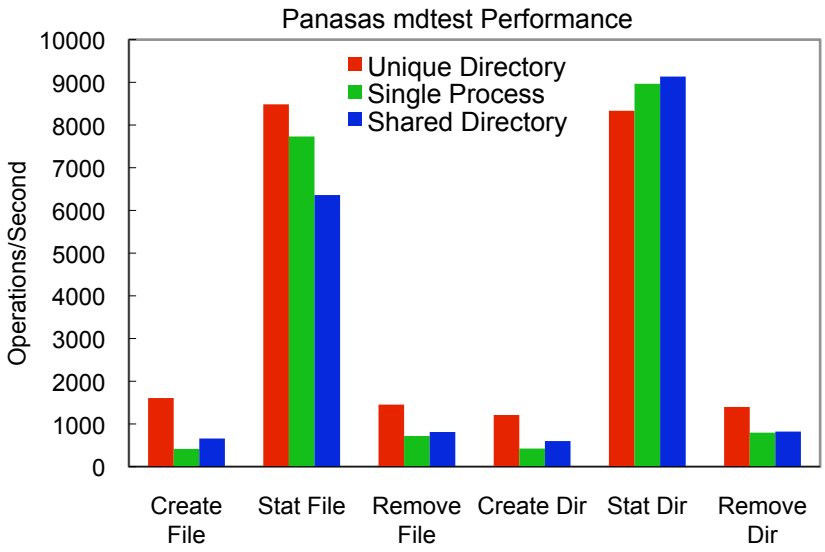
1) Process A creates files for all processes in root directory.
2) Processes A, B, and C open, stat, and close their own files.
3) Process A removes files for all processes.

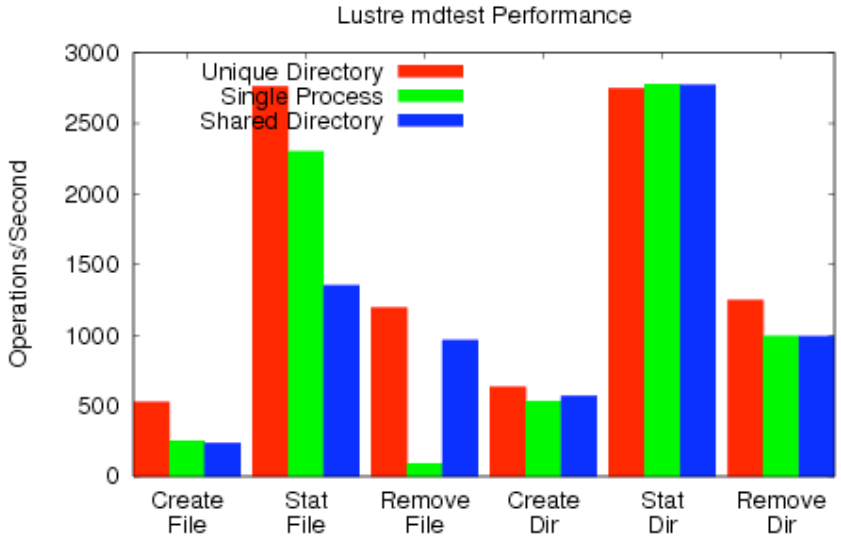
Shared Directory

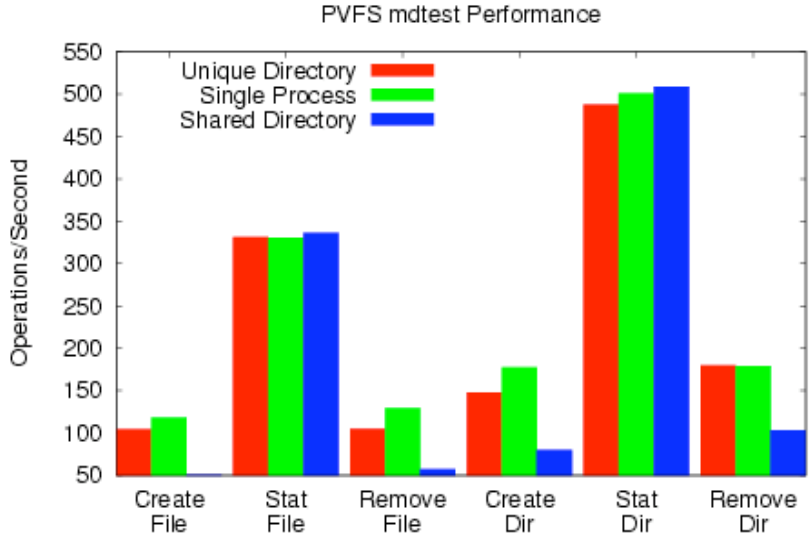


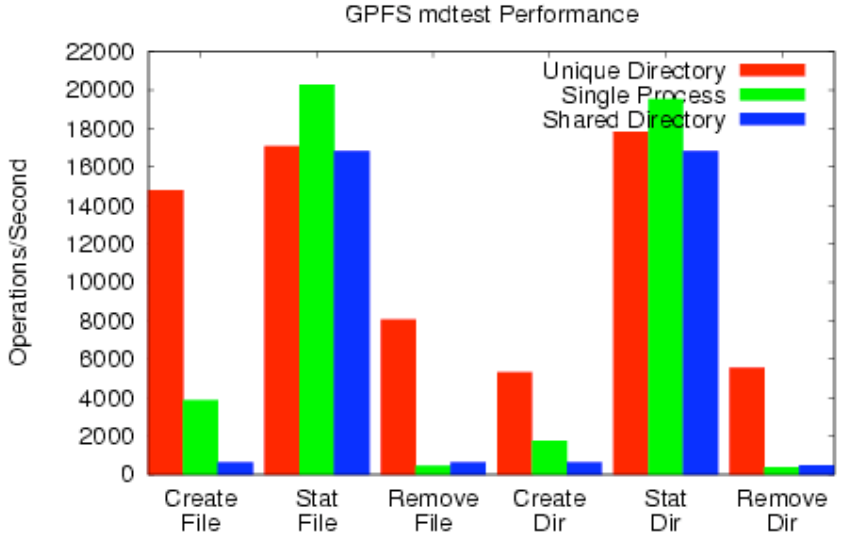
1) Each process (A, B, C) creates, stats, and removes its own files in the root directory.

mdtest Results









mdtest Analysis

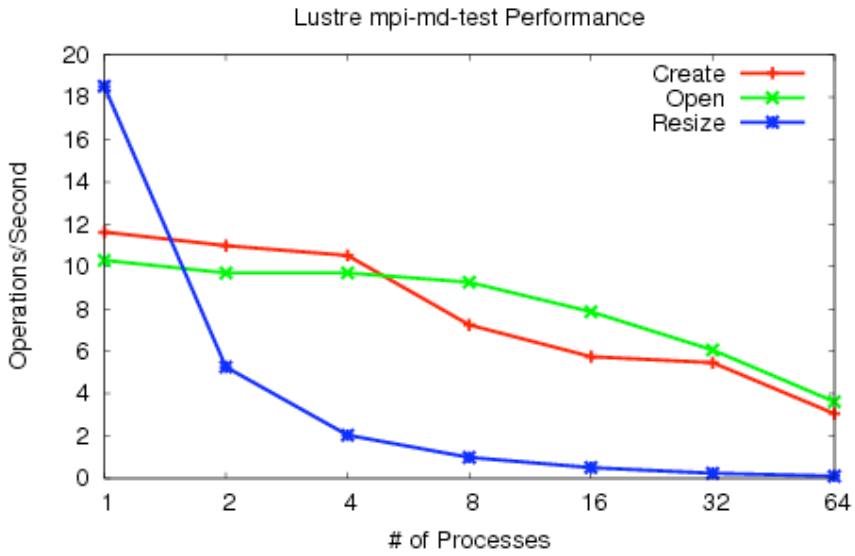
- PVFS
  - No penalty if all processes operate on own files
  - Like fdtree, lack of client caching hurts stat
- GPFS: Very high cost to operating in the same directory
  - Each client must acquire token & modify dir itself
- Lustre has distributed directories, and as a result has a lower penalty for using shared directory
- Panasas coarse-grained metadata clustering not active here, since all procs share common root

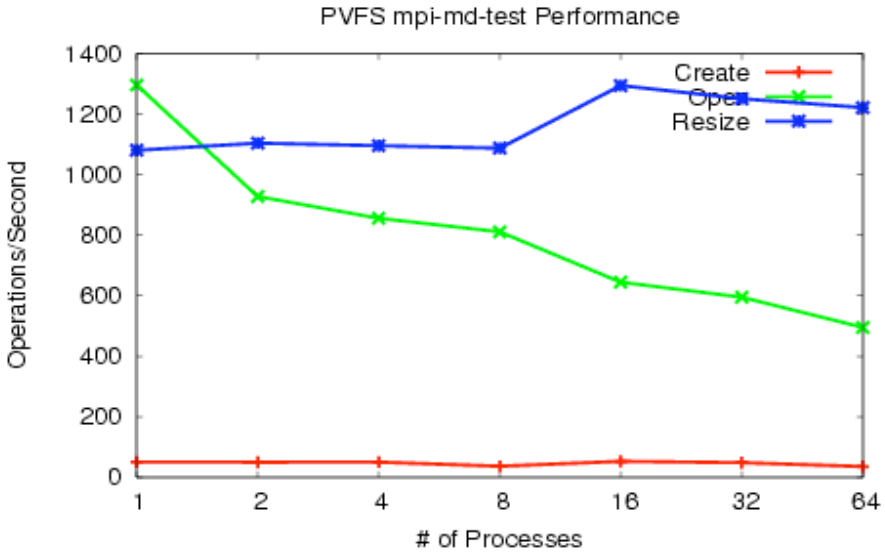
mpi-md-test: MPI-IO Metadata Operations

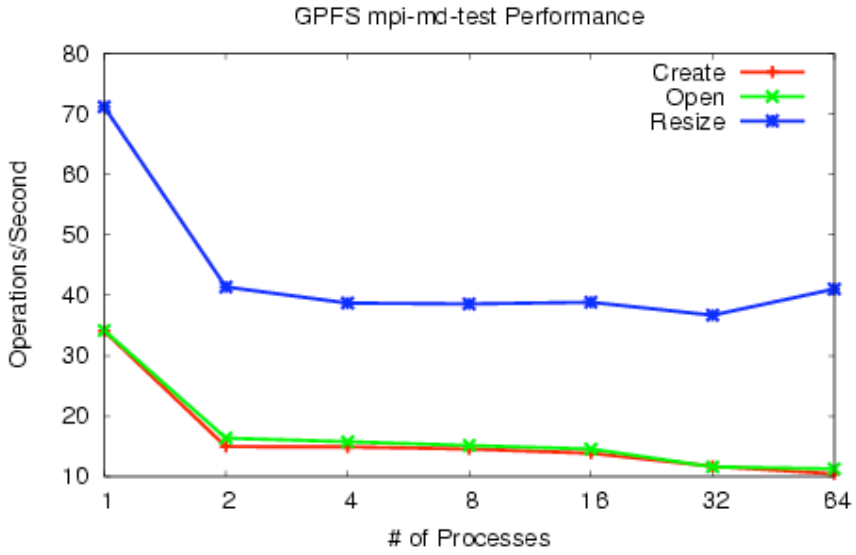
- Written at Argonne National Laboratory
- MPI code that measures performance of several collective MPI-IO metadata routines
  - Create: each process collectively calls `MPI_File_open` to create N files
    - mpi-md-test -O -d ./x -n 1000
  - Open: each process collectively calls `MPI_File_open` on N pre-existing files
    - mpi-md-test -O -d ./x -n 1000 (after prior create run)
  - Resize: each process collectively calls `MPI_File_set_size` on one file
    - mpi-md-test -R -d ./x -n 100
- Collective routines: potential for optimization
  - Perform on one process, broadcast result to others
- Allows us to see performance for coordinated metadata operations
  - How performance scales with number processes
- 64x2, 64x4 for large runs with Lustre, 16x4, 32x4, 32x8 for large GPFS runs

mpi-md-test Results

- Scalable algorithms in PVFS result in performance as good as MPI collectives
- Lustre numbers hampered by MPI resize impl. (resize on all nodes)







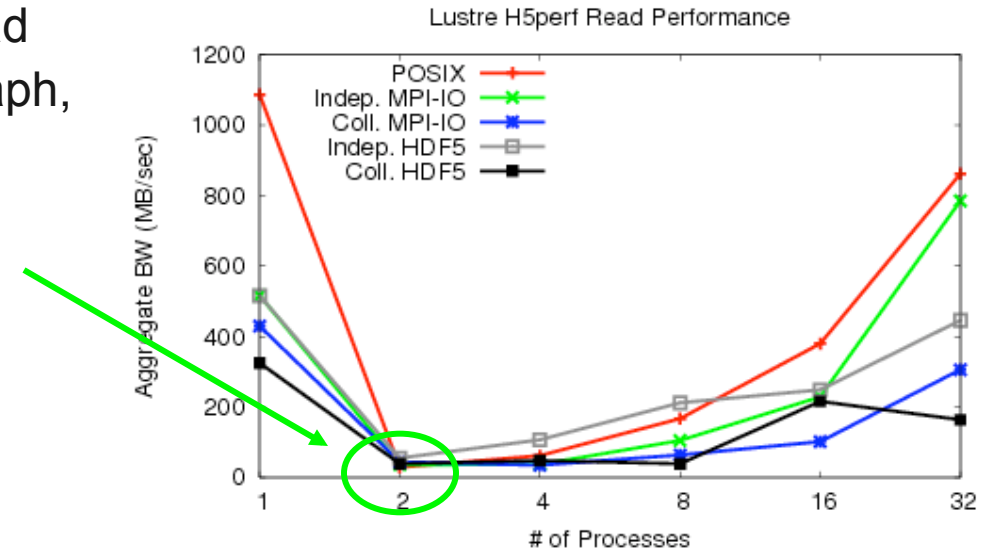
The POSIX I/O Interface

POSIX I/O Introduction

- 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
  - No ability to describe collective I/O accesses
  - It can be very expensive for a file system to guarantee POSIX semantics for heavily shared files (e.g., from clusters)
  - Network file systems like NFS chose not to implement strict POSIX semantics in all cases (e.g., lazy access time propagation)
- Presenting this interface primarily so that we can compare and contrast with other interfaces

Under the Covers of POSIX

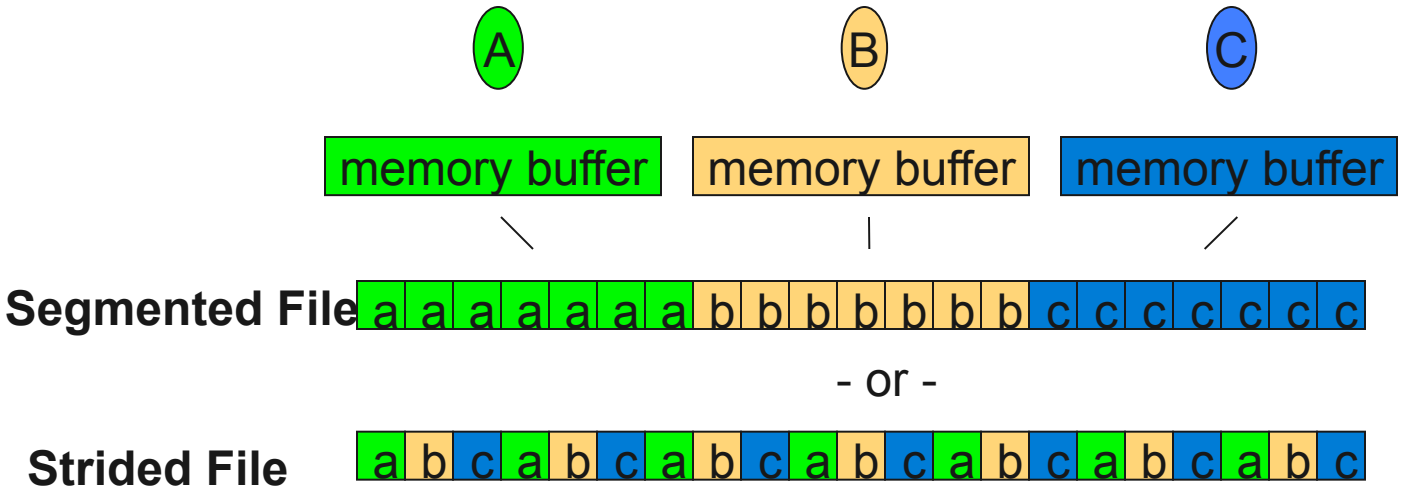
- POSIX API is a bridge between many tools and the file systems below
- Operating system maps these calls directly into file system operations
- File system performs I/O, using block- or region-oriented accesses depending on implementation
- “Compliant” file systems will likely perform locking to guarantee atomicity of operations
  - Can incur substantial overhead
  - Seen in this Lustre H5perf graph, optimizations to speed serial I/O performance can result in substantial overhead when more than one process wants to access the same file



IOR: File System Bandwidth

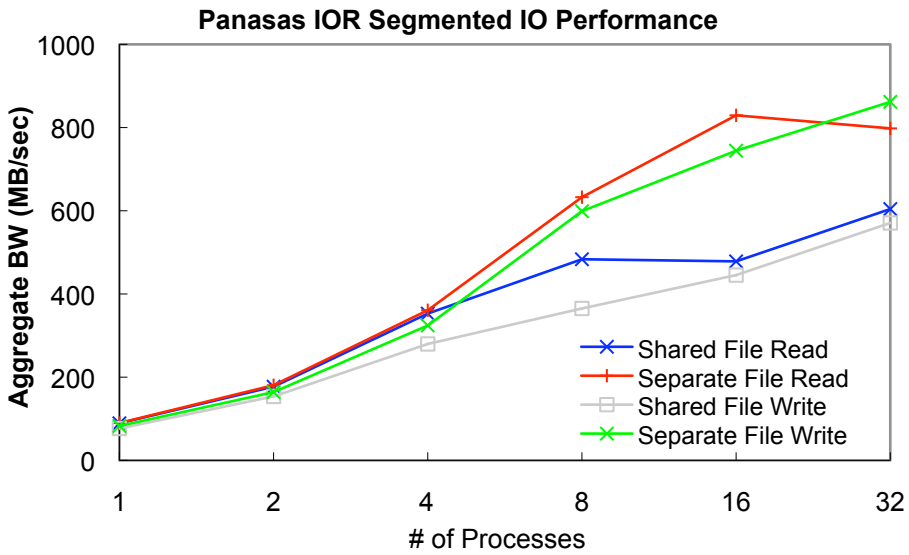
- Written at Lawrence Livermore National Laboratory
- Named for the acronym ‘interleaved or random’
- POSIX, MPI-IO, HDF5, and Parallel-NetCDF APIs
  - Shared or independent file access
  - Collective or independent I/O (when available)
- Employs MPI for process synchronization
- Used here to obtain peak POSIX I/O rates for shared and separate files
  - Running in segmented (contiguous) I/O mode
  - We ran two variations:
    - ./IOR -a POSIX -C -i 3 -t 4M -b 4G -e -v -v -o $FILE
      - Single, shared file
    - ./IOR -a POSIX -C -i 3 -t 4M -b 4G -e -v -v -F -o $FILE
      - One file per process

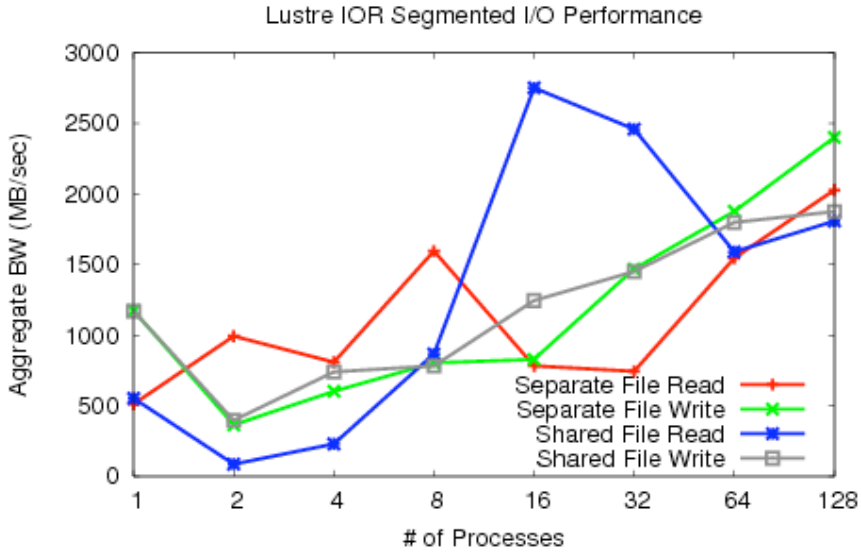
IOR Access Patterns for Shared Files

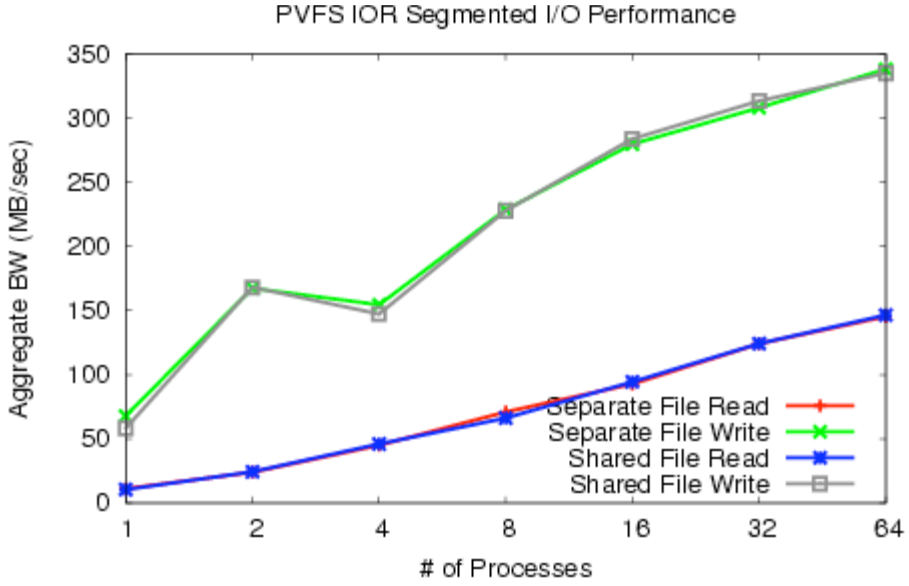


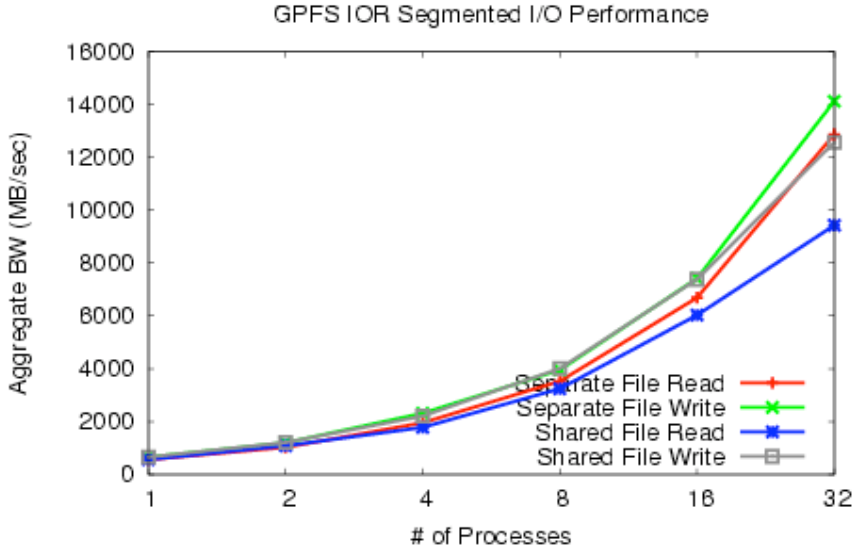
- Primary distinction between the two major shared-file patterns is whether each task's data is contiguous or noncontiguous
- For the segmented pattern, each task stores its blocks of data in a contiguous region in the file
- With the strided access pattern, each task's data blocks are spread out through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

IOR POSIX Segmented Results









IOR POSIX Segmented Analysis

- Aggregate performance increases to a point as more clients are added
  - Striping and multiple network links
- Expect to see a peak and flatten out after that peak
- Sometimes early spikes appear due to cache effects (not seen here)
- Incast hurts PVFS reads
- Panasas shared file 25-40% slower than separate file
  - IOR not using Panasas lazy coherency extensions

POSIX I/O High Performance Computing Extensions

APIs for HPC IO

- POSIX IO APIs (open, close, read, write, stat) have semantics that can make it hard to achieve high performance when large clusters of machines access shared storage.
- A working group (see next slide) of HPC users has drafted API additions for POSIX that will provide standard ways to achieve higher performance.
  - HECEWG: High End Computing Extensions Working Group
- Primary approach is either to relax semantics that can be expensive, or to provide more information to inform the storage system about access patterns.

Contributors

- Lee Ward - Sandia National Lab
- Bill Lowe, Tyce McLarty – Lawrence Livermore National Lab
- Gary Grider, James Nunez – Los Alamos National Lab
- Rob Ross, Rajeev Thakur, William Gropp - Argonne National Lab
- Roger Haskin – IBM
- Brent Welch, Marc Unangst - Panasas
- Garth Gibson- CMU/Panasas
- Alok Choudhary – Northwestern U
- Tom Ruwart- U of Minnesota/IO Performance
- Many Others…
- http://www.opengroup.org/platform/hecewg/

HPC POSIX Enhancement Areas

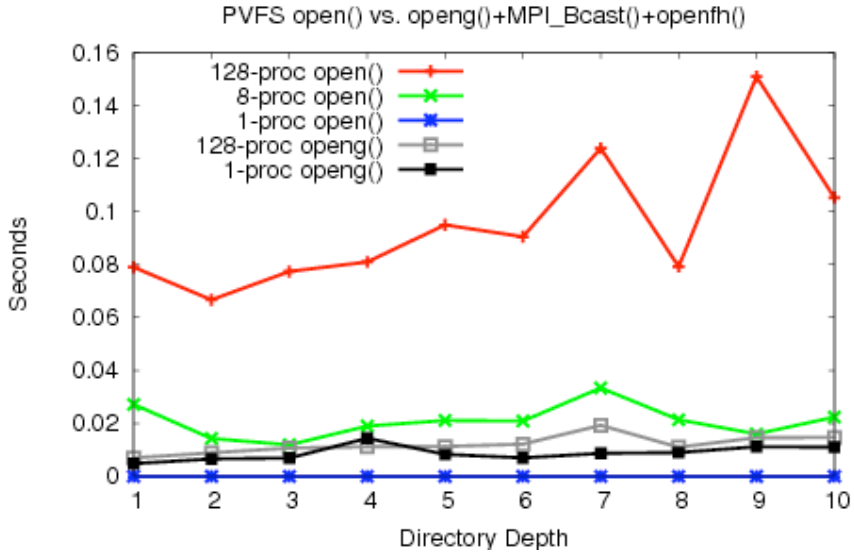
- Metadata
  - optional attributes, bulk attributes
  - statlite(), readdirplus(), readdirlite()
- Coherence
  - last writer wins and other such things can be optional
  - lazyio_propogate(), lazyio_synchronize()
- Shared file descriptors
  - file opens for cooperating groups of processes
  - openg(), openfh()
- Ordering
  - stream of bytes idea needs to move towards distributed vectors of units
  - readx(), writex()

POSIX HPC IO

- **`statlite, fstatlite`**
  - optional attributes
- **`readdirplus, readdirlite`**
  - expose NFS bulk attribute op to applications
- **`lazyio_propogate, lazyio_synchronize, O_LAZY`**
  - Hint to buffer cache management
- **`openg, openfh`**
  - expose file handles to applications
- **`readx, writex`**
  - memory vector to/from file vector
- **http://www.opengroup.org/platform/hecewg/**

POSIX Wrap-Up

- POSIX interface is a useful, ubiquitous interface for basic I/O
- Lacks any constructs useful for parallel I/O
- Should not be used in parallel applications if performance is desired
- However, work is ongoing to improve the POSIX I/O interface!
  - A working group of HEC users is drafting some proposed API additions for POSIX that will provide standard ways to achieve higher performance
  - Two general approaches
    - Relax semantics that can be expensive
    - Better inform the storage system about access patterns
- Providing a substitute for the POSIX open() call allows us to avoid name space operations on many nodes, resulting in much faster open operations when many clients will access the same file.



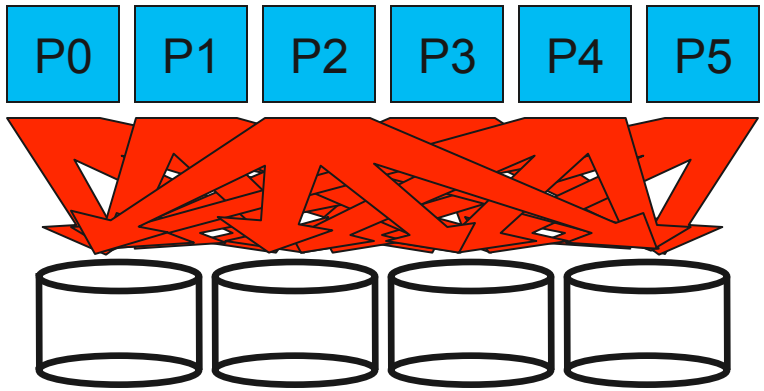
Data from Ruth Klundt (SNL), using Darkstar cluster.

The MPI-IO Interface

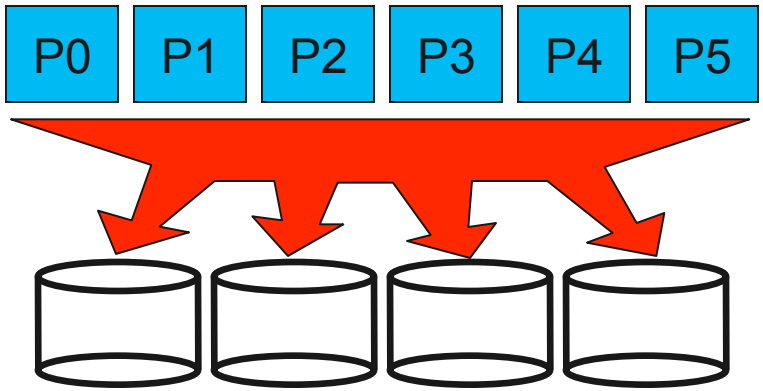
MPI-IO

- 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)

Independent and Collective I/O



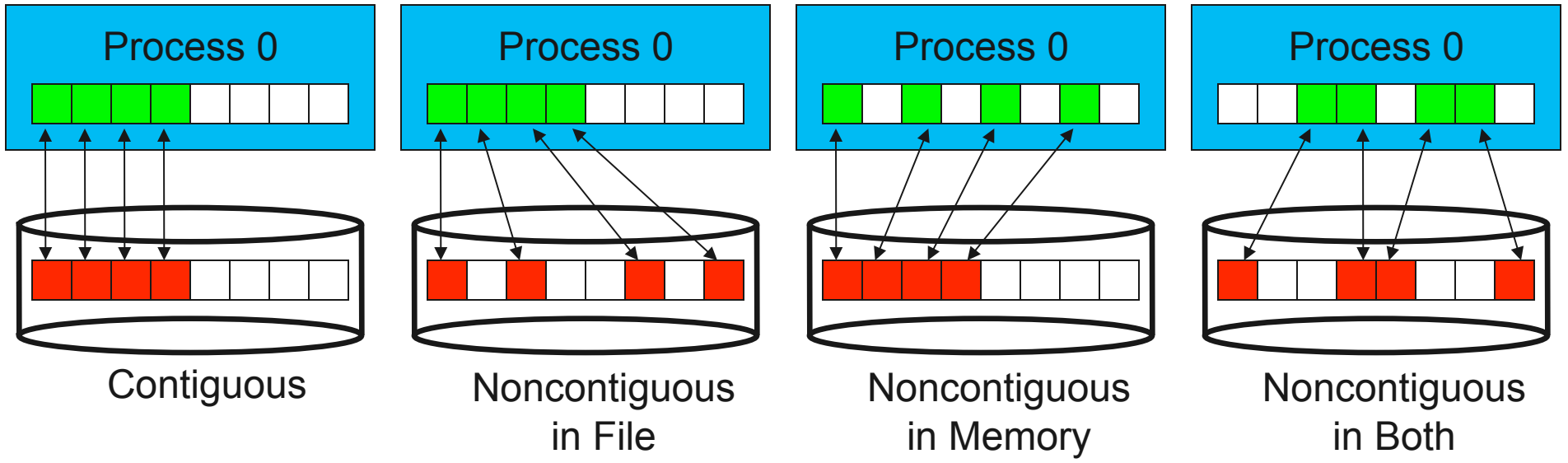
Independent I/O



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

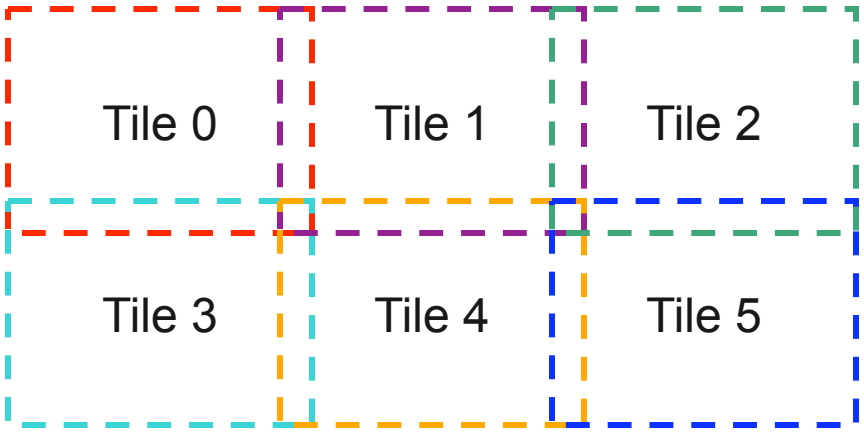


- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous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

Nonblocking and Asynchronous I/O

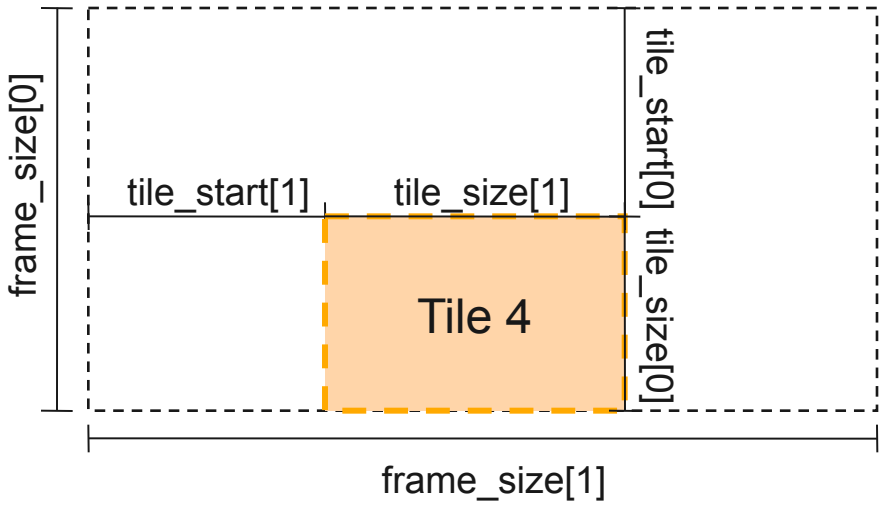
- Blocking, or Synchronous, I/O operations return when buffer may be reused
  - Data in system buffers or on disk
- Some applications like to overlap I/O and computation
  - Hiding writes, prefetching, pipelining
- A nonblocking interface allows for submitting I/O operations and testing for completion later
- If the system also supports asynchronous I/O, progress on operations can occur in the background
  - Depends on implementation
- Otherwise progress is made at start, test, wait calls

Example: Visualization Staging



- Often 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

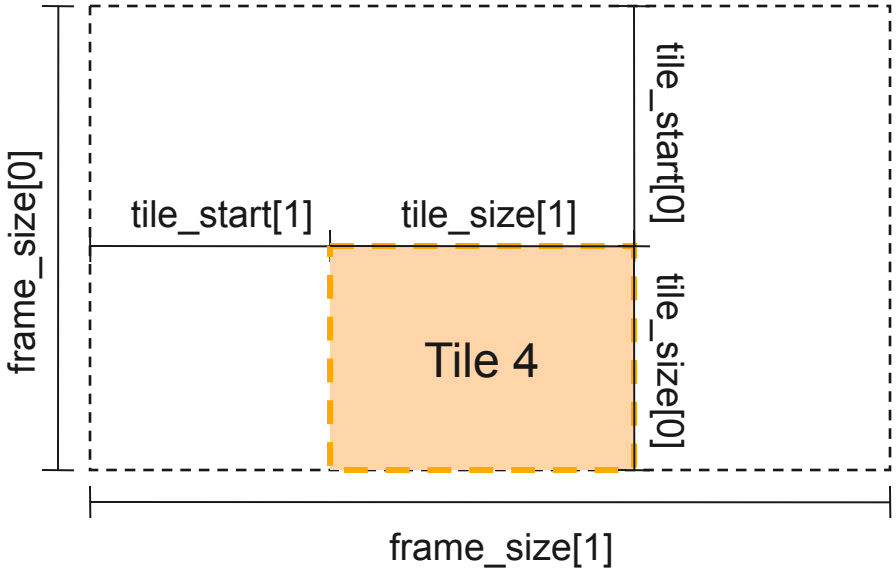
```
MPI_Datatype rgb, filetype;
MPI_File filehandle;
ret = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

/* 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

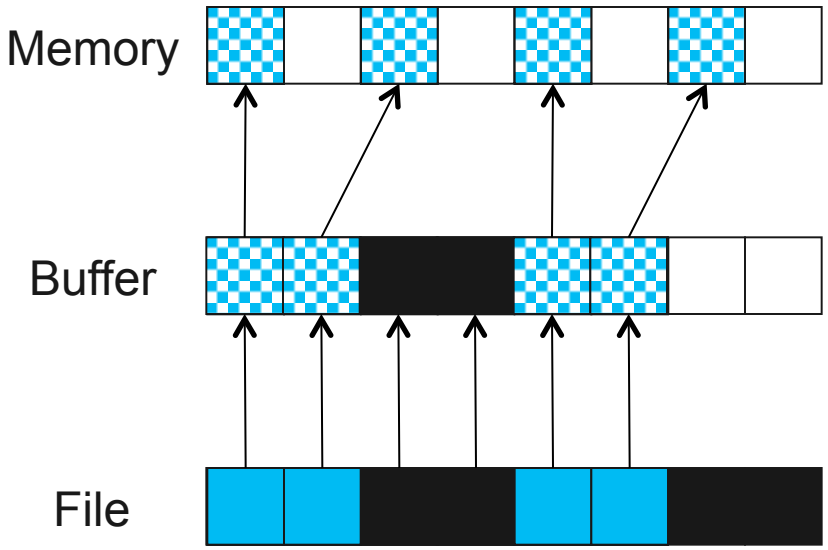
```
/* set file view, skipping header */
ret = MPI_File_set_view(filehandle,
  file_header_size, rgb, filetype, "native",
  MPI_INFO_NULL);
/* collectively read data */
ret = MPI_File_read_all(filehandle, buffer,
  tile_size[0] * tile_size[1], rgb, &status);
ret = MPI_File_close(&filehandle);
```

- 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

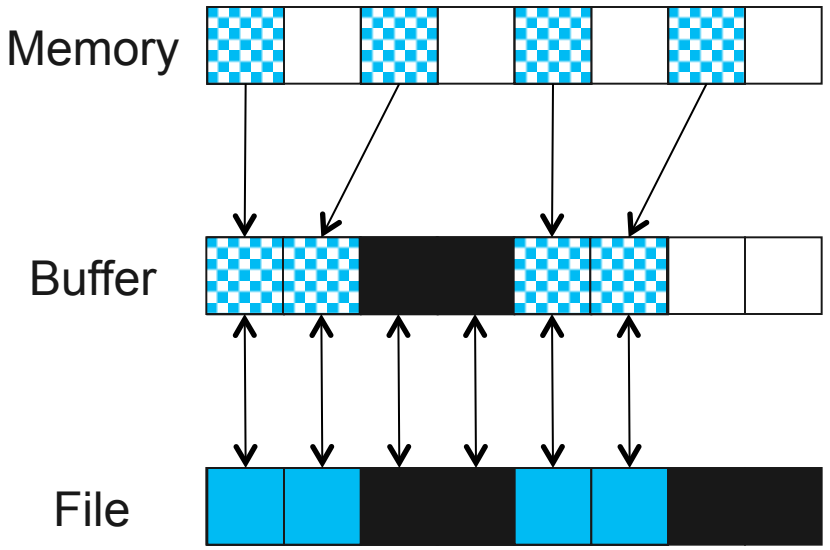
Noncontiguous I/O: Data Sieving



Data Sieving Read Transfers

- Data sieving is used to combine lots of small accesses into a single larger one
  - Remote file systems (parallel or not) tend to have high latencies
  - Reducing # of operations important
- Similar to how a block-based file system interacts with storage
- Generally very effective, but not as good as having a PFS that supports noncontiguous access

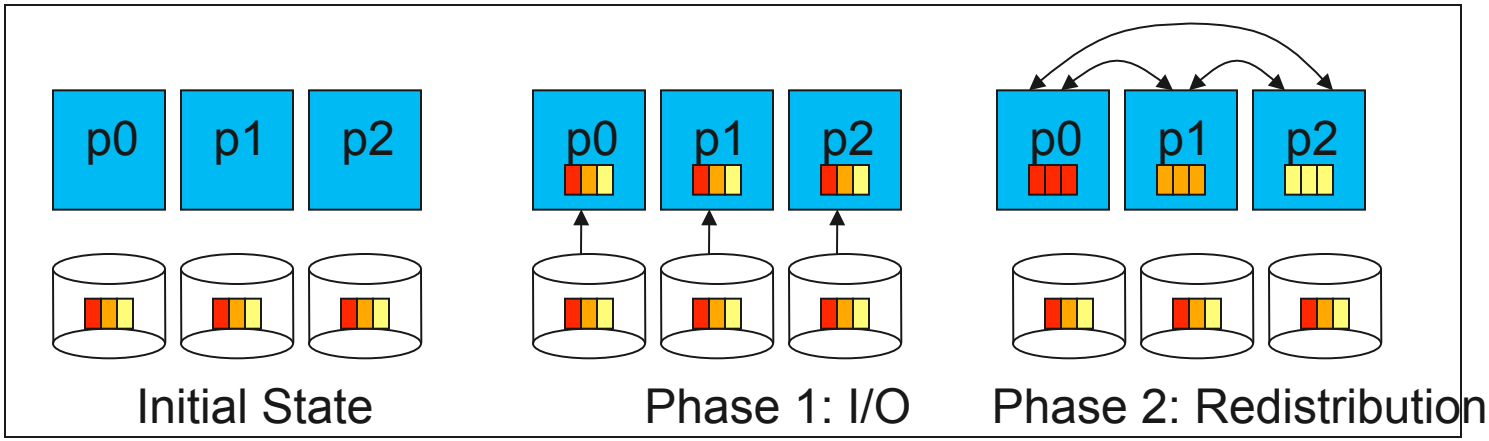
Data Sieving Write Operations



Data Sieving Write Transfers

- Data sieving for writes is more complicated
  - Must read the entire region first
  - Then make changes in buffer
  - Then write the block back
- Requires locking in the file system
  - Can result in false sharing (interleaved access)
- PFS supporting noncontiguous writes is preferred

Collective I/O and Two-Phase I/O

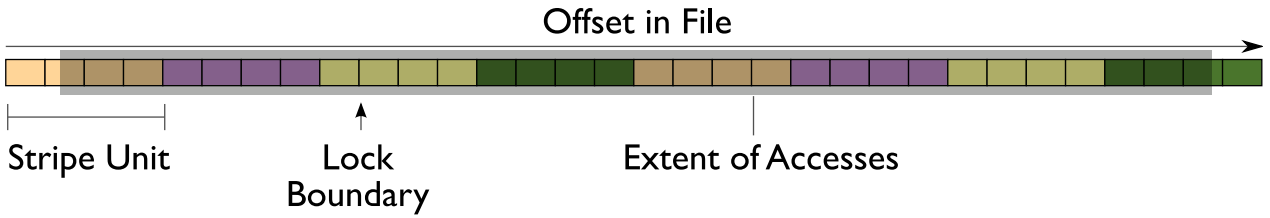


Two-Phase Read Algorithm

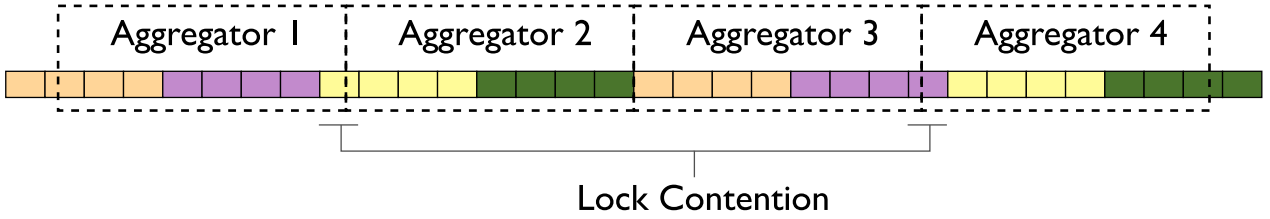
- Problems with independent, noncontiguous access
  - Lots of small accesses
  - Independent data sieving reads lots of extra data, can exhibit false sharing
- Idea: Reorganize access to match layout on disks
  - Single processes use data sieving to get data for many
  - Often reduces total I/O through 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

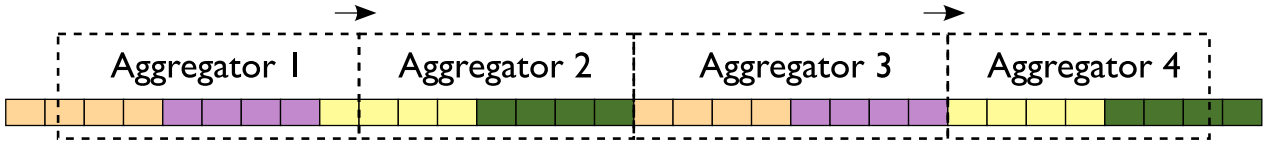
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



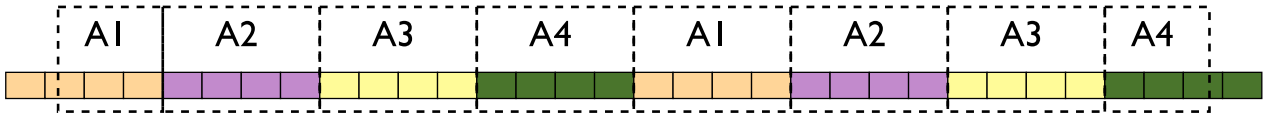
One approach is to evenly divide the region accessed across aggregators.



Aligning regions with lock boundaries eliminates lock contention.



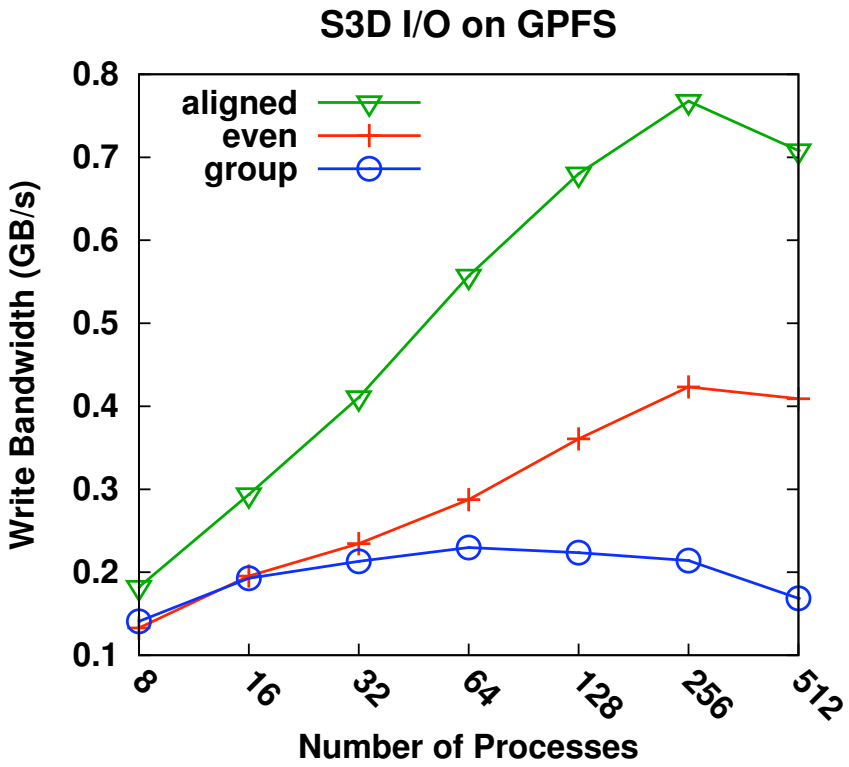
Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).



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.

Impact of Two-Phase I/O Algorithms

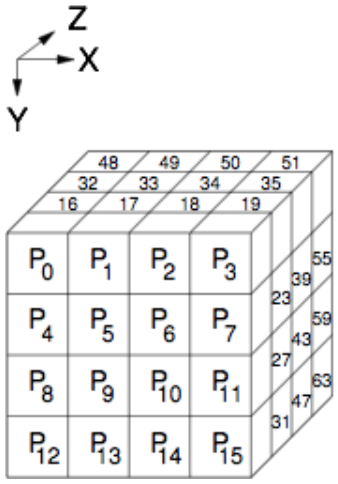
- This graph shows the performance for the S3D combustion code, writing to a single file.
- Aligning with lock boundaries doubles performance over default “even” algorithm.
- “Group” algorithm similar to server-aligned algorithm on last slide.
- Testing on Mercury, an IBM IA64 system at NCSA, with 54 servers and 512KB stripe size.

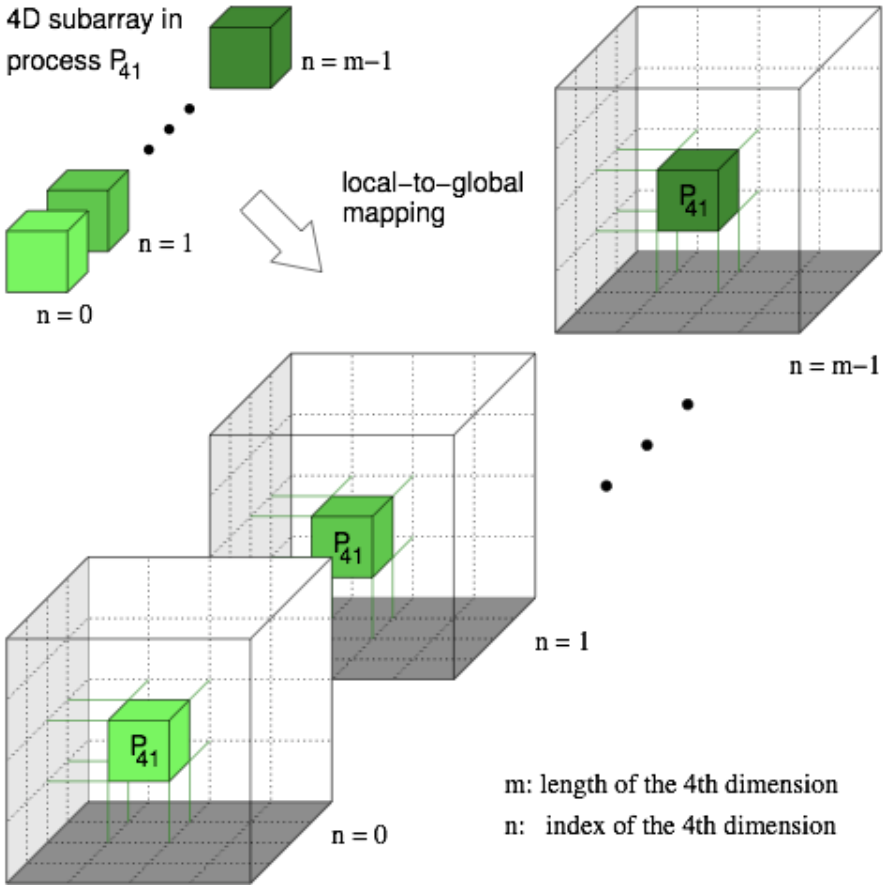


W.K. Liao and A. Choudhary, “Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols,” SC2008, November, 2008.

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





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.

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

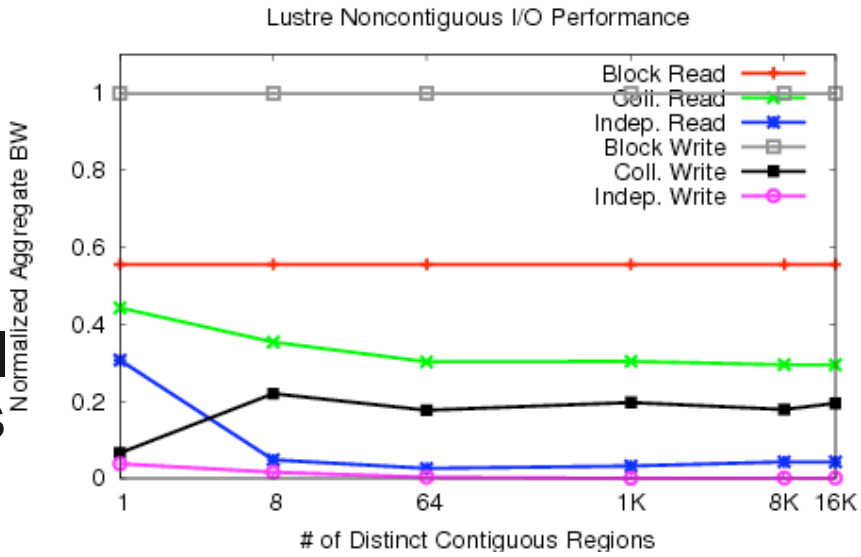
	Coll. Buffering and Data Sieving Disabled	Data Sieving Enabled	Coll. Buffering Enabled (incl. Aggregation)
POSIX writes	102,401	81	**5**
POSIX reads	0	80	0
MPI-IO writes	64	64	64
Unaligned in file	102,399	80	4
Total written (MB)	6.25	**87.11**	6.25
Runtime (sec)	1443	11	6.0
Avg. MPI-IO time per proc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

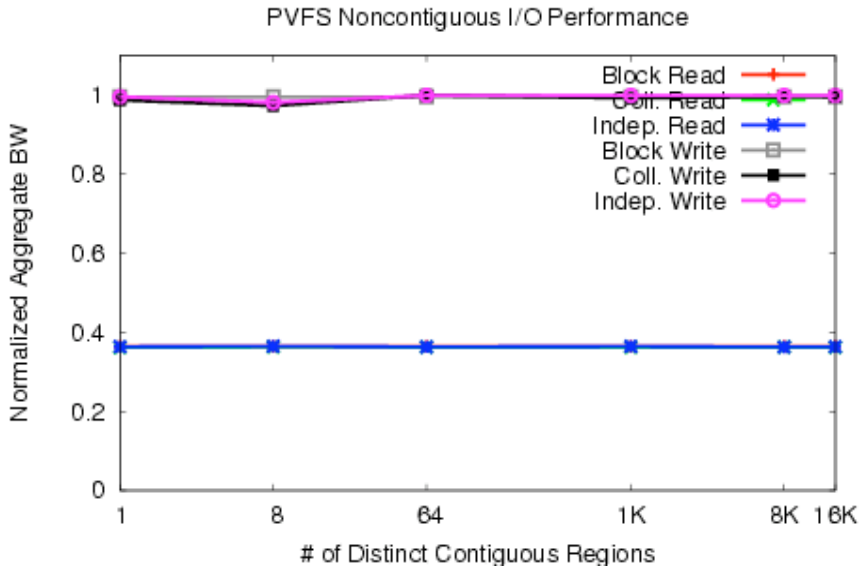
noncontig Benchmark

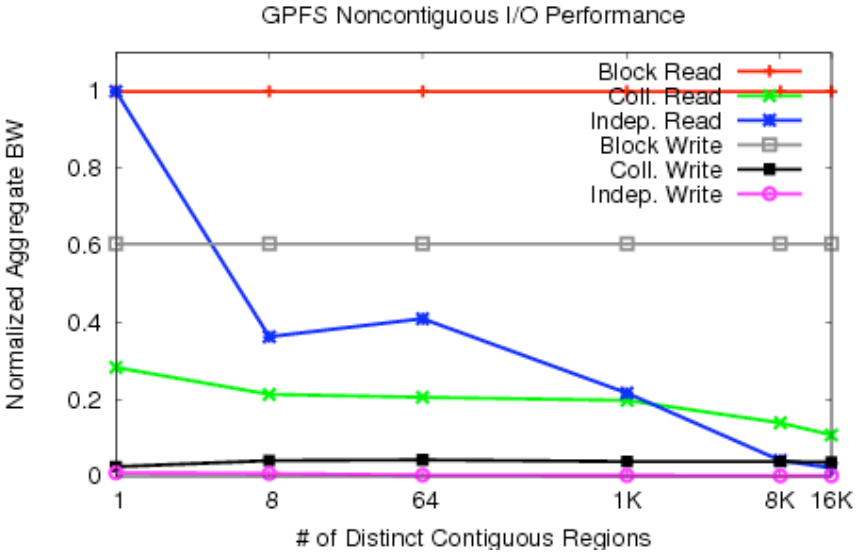
- Contributed by Joachim Worringen (formerly of NEC)
- Constructs a datatype and performs noncontiguous I/O in file
  - Struct of a vector of contigs
  - Option for both independent and collective access
  - Option to vary amount of data, how many pieces
- Far from ideal access pattern for many file systems and MPI-IO implementations
  - Naïve approach: lots and lots of tiny file accesses
  - But lots of room for optimization, esp. in collective case
- Lets us explore how well the file system handles increasingly poor access patterns

noncontig I/O Results

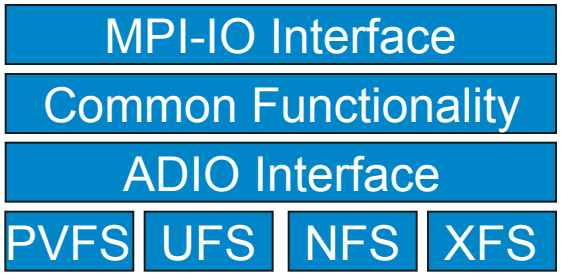
- This is one area where PVFS shines
  - High fraction of block BW for independent, noncontiguous I/O
- All file systems benefit from collective I/O optimizations for all but the most contiguous patterns
  - Collective I/O optimizations can be absolutely critical to performance







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
- Also forms solid basis for high-level I/O libraries
  - But they must take advantage of these features!

The Parallel netCDF Interface and File Format

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

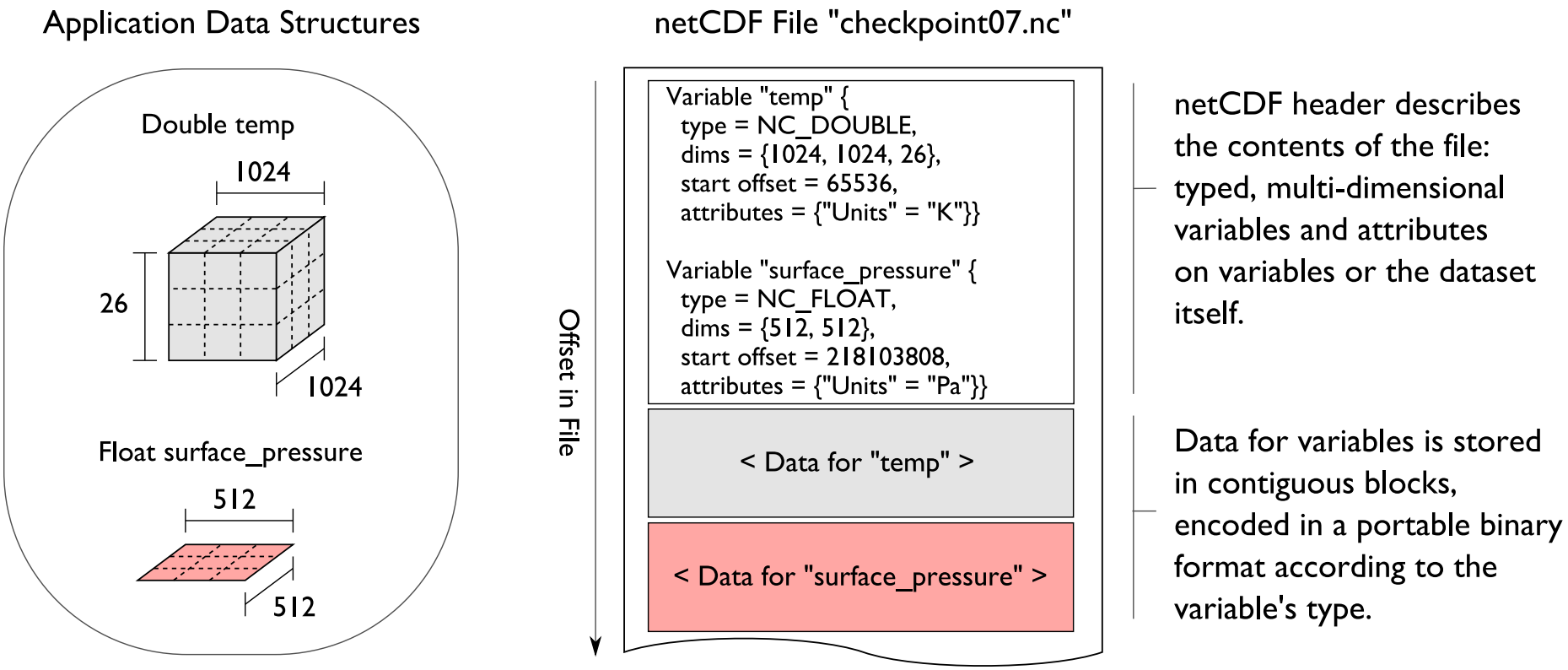
Higher Level I/O Interfaces

- 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)

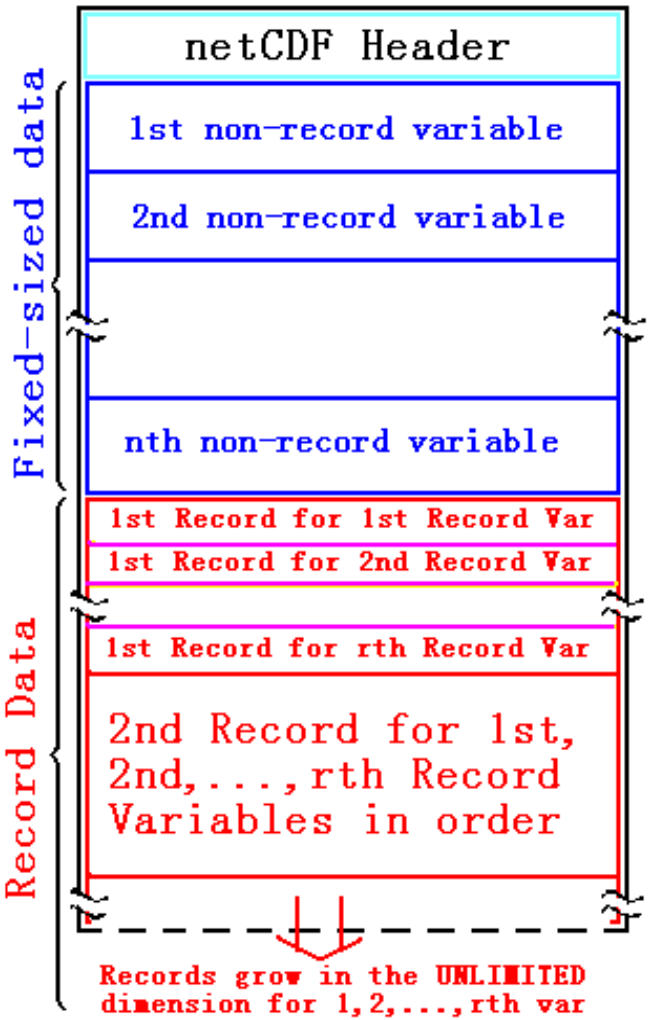
- Based on original "Network Common Data Format" (netCDF) work from Unidata
  - Derived from their source code
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C and Fortran interfaces
  - Portable data format (identical to netCDF)
  - Noncontiguous I/O in memory using MPI datatypes
  - Noncontiguous I/O in file using sub-arrays
  - Collective I/O
- Unrelated to netCDF-4 work (More about netCDF-4 later)

Data Layout in netCDF Files



Record Variables in netCDF

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

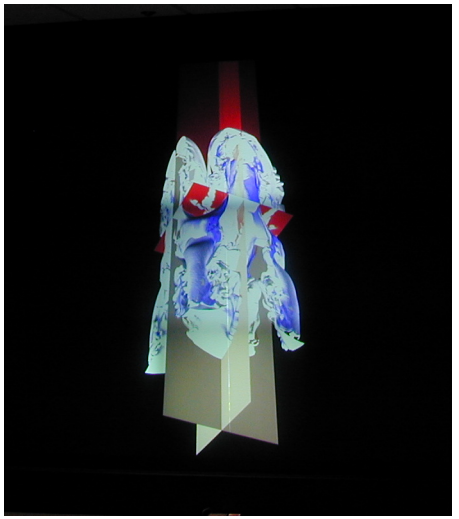


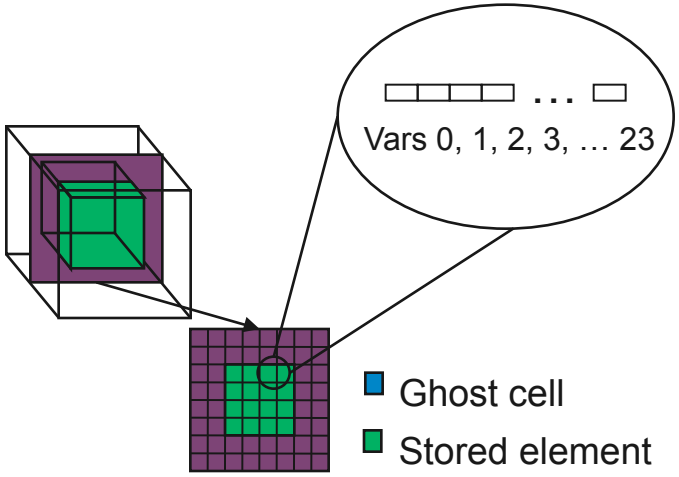
Storing Data in PnetCDF

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

Example: FLASH 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

PnetCDF Wrap-Up

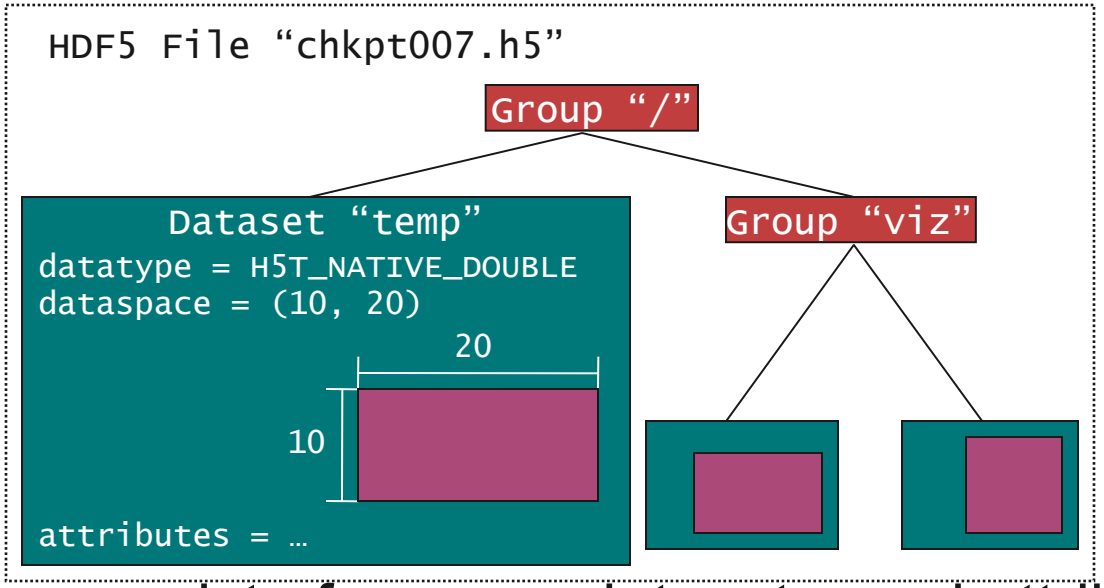
- PnetCDF gives us
  - Simple, portable, self-describing container for data
  - Collective I/O
  - Data structures closely mapping to the variables described
- If PnetCDF meets application needs, it is likely to give good performance
  - Type conversion to portable format does add overhead
- Some limits on (CDF-2) file format:
  - Fixed-size variable: < 4 GiB
  - Per-record size of record variable: < 4 GiB
  - 2^{32} -1 records
  - Work almost complete to relax these limits (CDF-5)

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

HDF5 Files



- HDF5 files consist of groups, datasets, and attributes
  - Groups are like directories, holding other groups and datasets
  - Datasets hold an array of typed data
    - A datatype describes the type (not an MPI datatype)
    - A dataspace gives the dimensions of the array
  - Attributes are small datasets associated with the file, a group, or another dataset
    - Also have a datatype and dataspace
    - May only be accessed as a unit

HDF5 Data Chunking

- Apps often read subsets of arrays (subarrays)
- Performance of subarray access depends in part on how data is laid out in the file
  - e.g. column vs. row major
- Apps also sometimes store sparse data sets
- Chunking describes a reordering of array data
  - Subarray placement in file determined lazily
  - Can reduce worst-case performance for subarray access
  - Can lead to efficient storage of sparse data
- Dynamic placement of chunks in file requires coordination
  - Coordination imposes overhead and can impact performance

Example: FLASH Particle I/O with HDF5

- FLASH "Lagrangian particles" record location, characteristics of reaction
  - Passive particles don't exert forces; pushed along but do not interact
- Particle data included in checkpoints, but not in plotfiles; dump particle data to separate file
- One particle dump file per time step
  - i.e., all processes write to single particle file
- Output includes application info, runtime info in addition to particle data

```
Block=30;
Pos_x=0.65;
Pos_y=0.35;
Pos_z=0.125;
Tag=65;
Vel_x=0.0;
Vel_y=0.0;
vel_z=0.0;
```

Typical particle data

Storing Labels for Particles

Remember:
"S" is for dataspace,
"T" is for datatype,
"D" is for dataset!

```
int string_size = OUTPUT_PROP_LENGTH;
hsize_t dims_2d[2] = {npart_props, string_size};
hid_t dataspace, dataset, file_id, string_type;

/* store string creation time attribute */
string_type = H5Tcopy(H5T_C_S1);
H5Tset_size(string_type, string_size);
dataspace = H5Screate_simple(2, dims_2d, NULL);
dataset   = H5Dcreate(file_id, "particle names",
   string_type, dataspace, H5P_DEFAULT);
if (myrank == 0) {
   status = H5Dwrite(dataset, string_type, H5S_ALL,
   H5S_ALL,  H5P_DEFAULT, particle_labels);
}
```

get a copy of the string type and resize it

Write out all 8 labels in one call

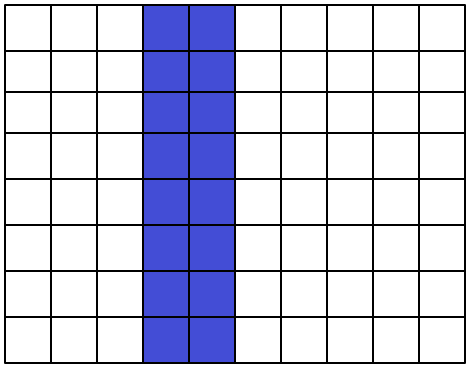
Storing Particle Data with Hyperslabs (1 of 2)

```
hsize_t dims_2d[2];

/* Step 1: set up dataspace –
           describe global layout */

dims_2d[0] = total_particles;
dims_2d[1] = npart_props;

dspace = H5Screate_simple(2, dims_2d, NULL);
dset = H5Dcreate(file_id, "tracer particles",
  H5T_NATIVE_DOUBLE, dspace, H5P_DEFAULT);
```



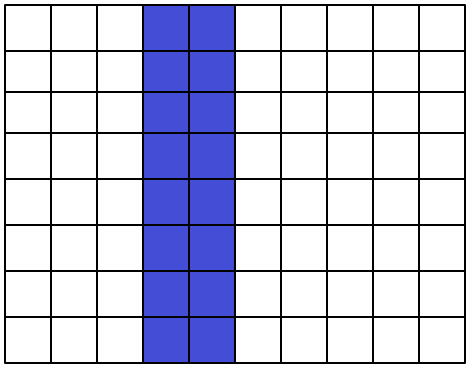
```
local_np = 2,
part_offset = 3,
total_particles = 10,
Npart_props = 8
```

Remember:
"S" is for dataspace,
"T" is for datatype,
"D" is for dataset!

Storing Particle Data with Hyperslabs (2 of 2)

```
hsize_t start_2d[2] = {0, 0},
        stride_2d[1] = {1, 1};
hsize_t count_2d[2] = {local_np,
                       npart_props};

/* Step 2: setup hyperslab for
   dataset in file */
```



```
local_np = 2,
part_offset = 3,
total_particles = 10,
Npart_props = 8
```

```
start_2d[0]  = part_offset; /* different for each process */
status = H5Sselect_hyperslab(dspace,
                    H5S_SELECT_SET,
                    start_2d, stride_2d, count_2d, NULL);
```

dataspace from last slide

- Hyperslab selection similar to MPI-IO file view
- Selections don't overlap in this example (would be bad if writing!)
- `H5SSelect_none()` if no work for this process

Collectively Writing Particle Data

```
/* Step 1: specify collective I/O */
dxfer_template = H5Pcreate(H5P_DATASET_XFER);
ierr = H5Pset_dxpl_mpio(dxfer_template,
  H5FD_MPIO_COLLECTIVE);

/* Step 2: perform collective write */
status = H5Dwrite(dataset,
                  H5T_NATIVE_DOUBLE,
                  memspace,
                  dspace,
                  dxfer_template,
                  particles);
```

"P" is for property list; tuning parameters

dataspace describing memory, could also use a hyperslab

dataspace describing region in file, with hyperslab from previous two slides

Remember:
"S" is for dataspace,
"T" is for datatype,
"D" is for dataset!

Inside HDF5

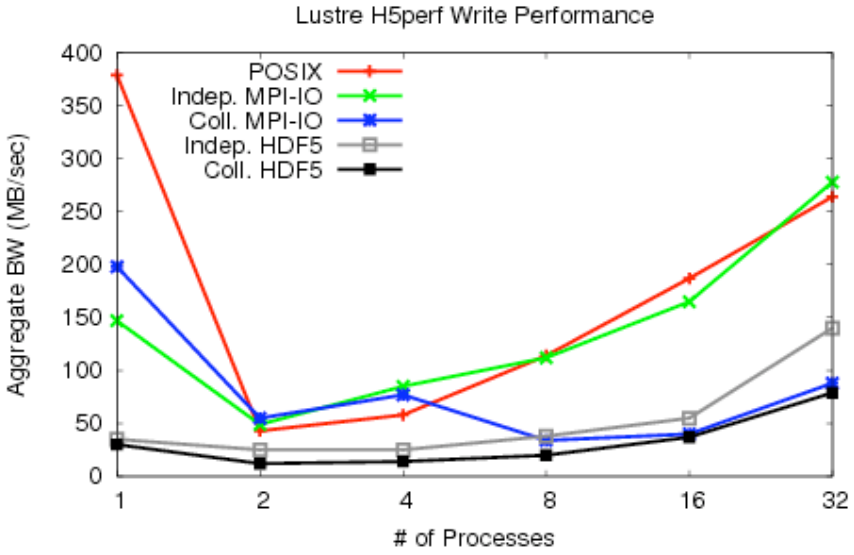
- `MPI_File_open` used to open file
- Because there is no "define" mode, file layout is determined at write time
- In `H5Dwrite`:
  - Processes communicate to determine file layout
    - Process 0 performs metadata updates
  - Call `MPI_File_set_view`
  - Call `MPI_File_write_all` to collectively write
    - Only if this was turned on (more later)
- Memory hyperslab could have been used to define noncontiguous region in memory
- In FLASH application, data is kept in native format and converted at read time (defers overhead)
  - Could store in some other format if desired
- At the MPI-IO layer:
  - Metadata updates at every write are a bit of a bottleneck
    - MPI-IO from process 0 introduces some skew

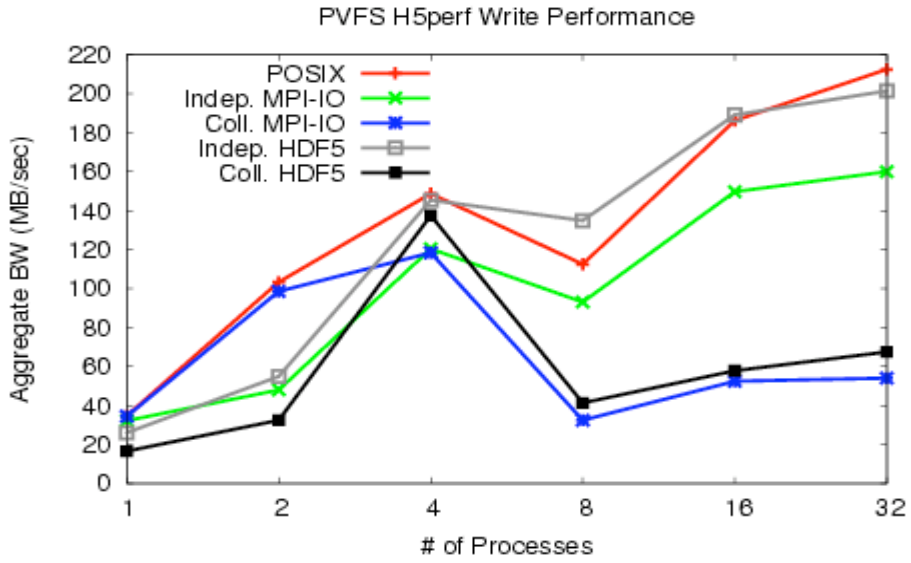
h5perf: HDF5 Benchmark

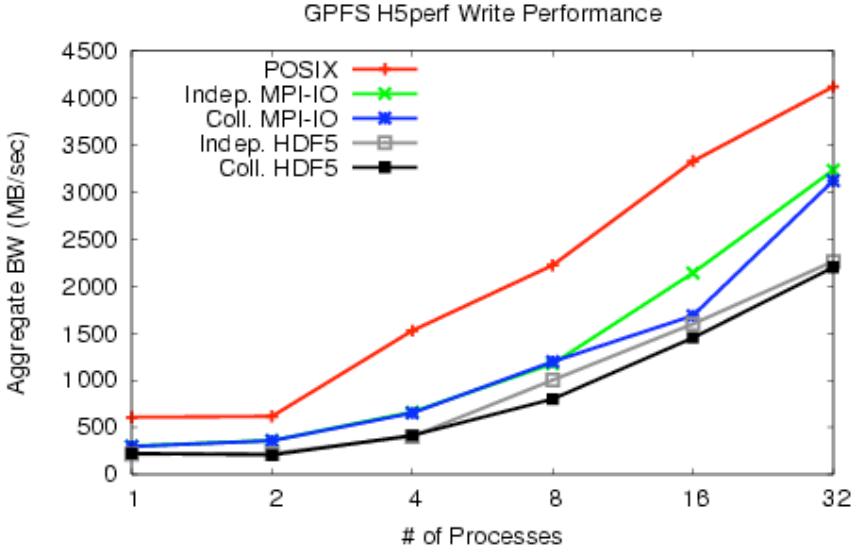
- Written by HDF5 team
- Provides a comparison of peak performance through the various interfaces
  - A little artificial; the interfaces are really used for different purposes
- Similar to IOR, in that it offers APIs for parallel HDF5, MPI-IO, and POSIX
  - Can vary block size, transfer size, number of data sets per file, and size of each data set
  - Optional dataset chunking (not default)
  - Collective and independent I/O options
- Contiguous I/O
- 1-32 clients (open-close time included)

H5perf Write Results

- On Lustre:
  - POSIX and independent MPI-IO have similar performance (expected)
  - Collective MPI-IO and HDF5 lose significant performance
    - Big, aligned blocks don't benefit from collective I/O optimizations
- On GPFS:
  - POSIX significantly faster than MPI-IO (?)
  - All other results are tightly grouped

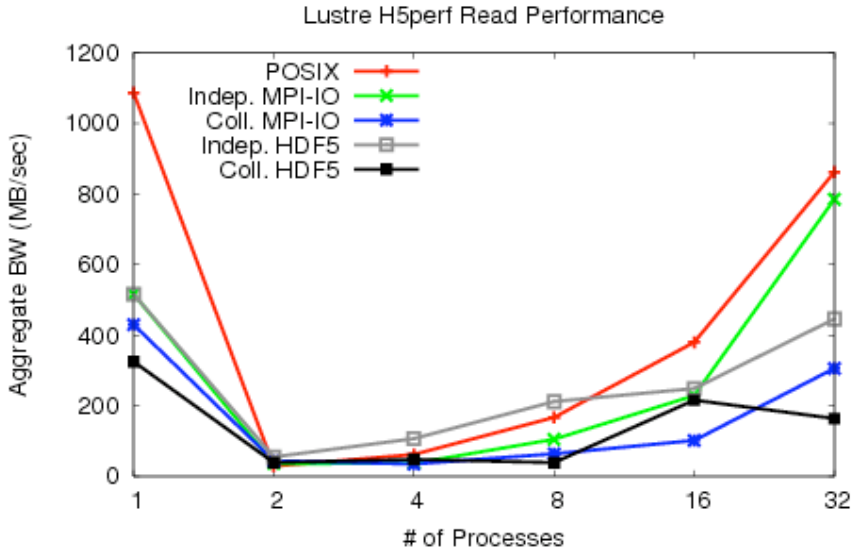


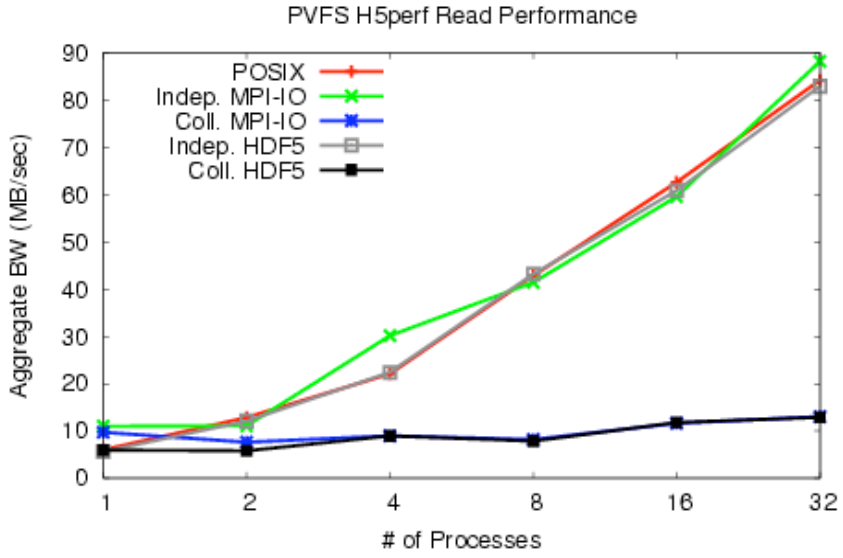


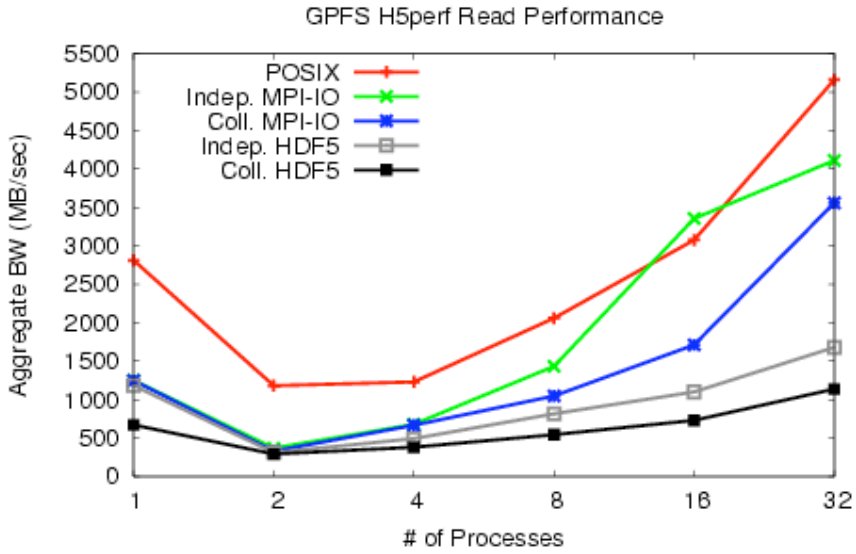


H5perf Read Results

- Locking and HEC don't play well
- Much larger spread between interfaces than in write cases
- Collective I/O isn't a win when you're doing big block I/O at these scales
  - Might help at very large scale to better coordinate access

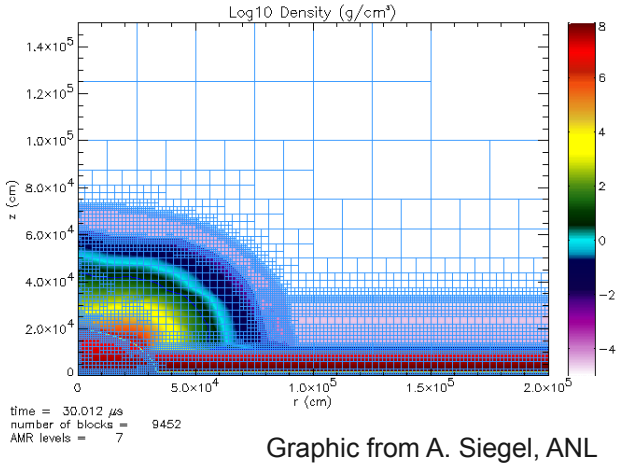






FLASH Astrophysics I/O Kernel

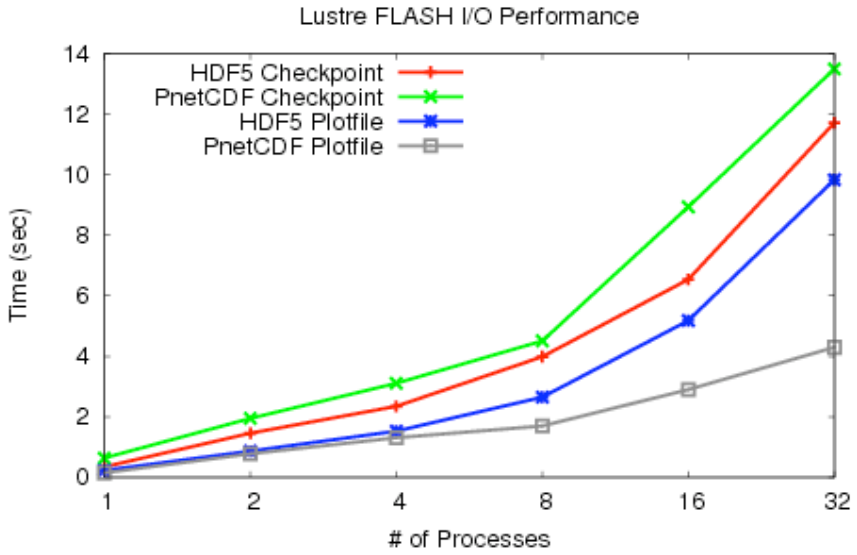
- Written by FLASH team
- Simulation of the I/O performed by the FLASH application
- We'll show both "checkpoint" and "plotfile with corners" results
  - Checkpoints are full dumps necessary for restart
  - Plotfiles are smaller files used for visualization
- Fixed number of blocks per process
- Looking at relative performance of HDF5 and PnetCDF
  - Also absolute time to perform operations on these systems

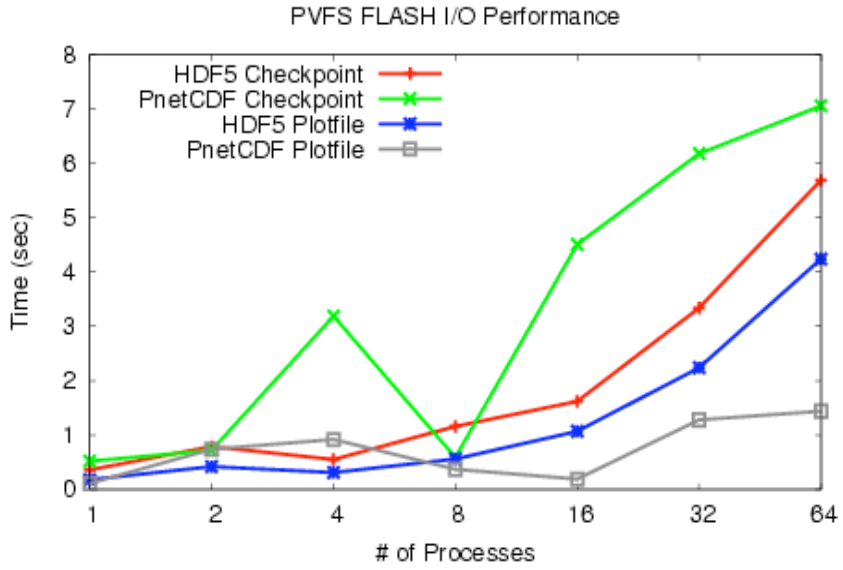


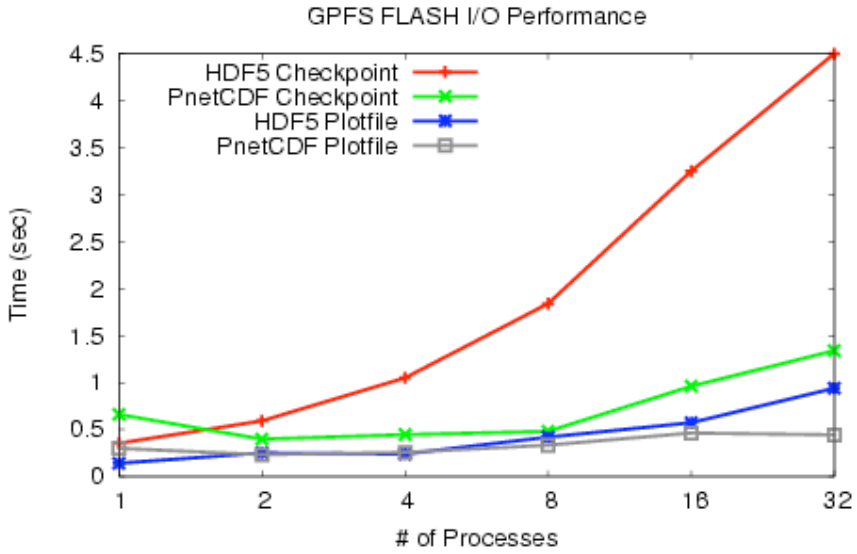
Graphic from A. Siegel, ANL

FLASH I/O Benchmark Results

- Your mileage my vary!
- PnetCDF slower for checkpoints on Lustre, PVFS
  - PnetCDF uses collective MPI-IO calls by default
- PnetCDF considerably faster on GPFS
  - Collective I/O not penalized







The netCDF-4 Effort

Thanks to Quincey Koziol (HDF group), Russ Rew (UCAR), and Ed Hartnett (UCAR) for helping ensure the accuracy of this material.

netCDF-4

- Joint effort between Unidata (netCDF) and NCSA (HDF5)
  - Initial effort NASA funded.
  - Ongoing development Unidata/UCAR funded.
- Combine netCDF and HDF5 aspects
  - HDF5 file format (still portable, self-describing)
  - netCDF API
- Features
  - Parallel I/O
  - C, Fortran, and Fortran 90 language bindings (C++ in development)
  - per-variable compression
  - multiple unlimited dimensions
  - higher limits for file and variable sizes
  - backwards compatible with “classic” datasets
  - Groups
  - Compound types
  - Variable length arrays
  - Data chunking and compression (parallel reads only – serial writes)

NetCDF 4 API Summary

- `nc_create_par("demo", NC_MPIIO|NC_NETCDF4, MPI_COMM_WORLD, MPI_INFO_NULL, &ncfile);`
  - New flag 'NC_NETCDF4'; MPI Communicator, Info
- `nc_open_par("demo", NC_MPIIO, MPI_COMM_WORLD, MPI_INFO_NULL, &ncfile);`
  - Can select POSIX (NC_MPIPOSIX) or MPI-IO (NC_MPIIO)
- `nc_var_par_access(ncfile, varid, NC_COLLECTIVE);`
  - Enable/disable collective I/O access (enabled by default).
- No longer need "send-to-master" model. (very pnetcdf-like)

Comparing PnetCDF and netCDF-4

- netCDF-4: parallel access through new function calls (_par)
  - Open, create take MPI hints (like PnetCDF)
  - Collective I/O by default (like PnetCDF)
  - Same routine can be either independent or collective depending on mode (like HDF5)
  - HDF5 tools understand netCDF-4 datasets

Parallel netCDF	netCDF-4
ncmpi_open	nc_open_par
ncmpi_create	nc_create_par
ncmpi_enddef	nc_enddef
ncmpi_def_dim	nc_def_dim
ncmpi_put_vara_float_all	nc_put_vara_float
ncmpi_begin_indep_data	nc_var_par_access

netCDF-4 wrapup

- Released in June 2008
- Similarities to both HDF5 and Parallel netCDF
  - HDF5: additional routine to toggle collective vs. independent
  - PnetCDF: takes MPI_Comm and MPI_Info as part of open/create calls
  - HDF5 tools understand netCDF-4 datasets
- More information:
  - http://www.unidata.ucar.edu/software/netcdf/netcdf-4/
  - Muqun Yang, “Performance Study of Parallel NetCDF4 in ROMS”, NCSA HDF group, June 30th, 2006

The ADaptable IO System (ADIOS)

Thanks to Scott Klasky (ORNL) for providing background material on ADIOS.

ADaptable IO System (ADIOS)

The goal of ADIOS is to create an easy and efficient I/O interface that hides the details of I/O from computational science applications:

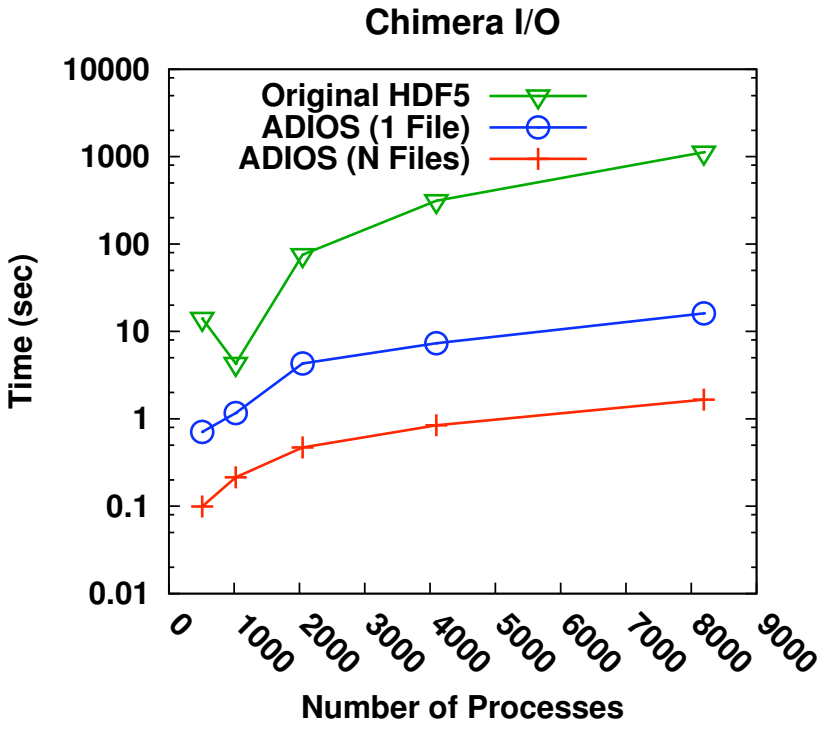
- Operate across multiple HPC architectures and parallel file systems
  - Blue Gene, Cray, IB-based clusters
  - Lustre, PVFS2, GPFS, Panasas, PNFS
- Support many underlying file formats and interfaces
  - MPI-IO, POSIX, HDF5, netCDF
  - Facilitates switching underlying file formats to reach performance goals
- Cater to common I/O patterns
  - Restarts, analysis, diagnostics
  - Different combinations provide different levels of IO performance
- Compensate for inefficiencies in the current I/O infrastructures

ADIOS Binary Packed (BP) File Format

Defers translation into portable format to attain high performance at runtime.

Accelerates writing from large numbers of processes through a log-like storage format:

- Each process writes independently
- Coordinate only twice
  - Once at start to determine writing locations
  - Once at end for metadata collection
- Move the "header" to the end to aid in alignment



I/O times for Chimera astrophysics application on Cray XT at ORNL. "1 File" results may benefit from Lustre optimizations that were not in place at time of testing.

Lightweight Application Characterization with Darshan

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

Darshan Goals

- Capture application-level behavior
  - Both POSIX and MPI-IO
  - Portable across file systems and hardware
- Transparent to users
  - Negligible performance impact
  - No source code changes
- Leadership-class scalability
  - 100,000+ processes

- Scalability tactics:
  - Bounded memory footprint
  - Minimize redundant information
  - Avoid shared resources at run time
  - Scalable algorithms to aggregate information

The Darshan Approach

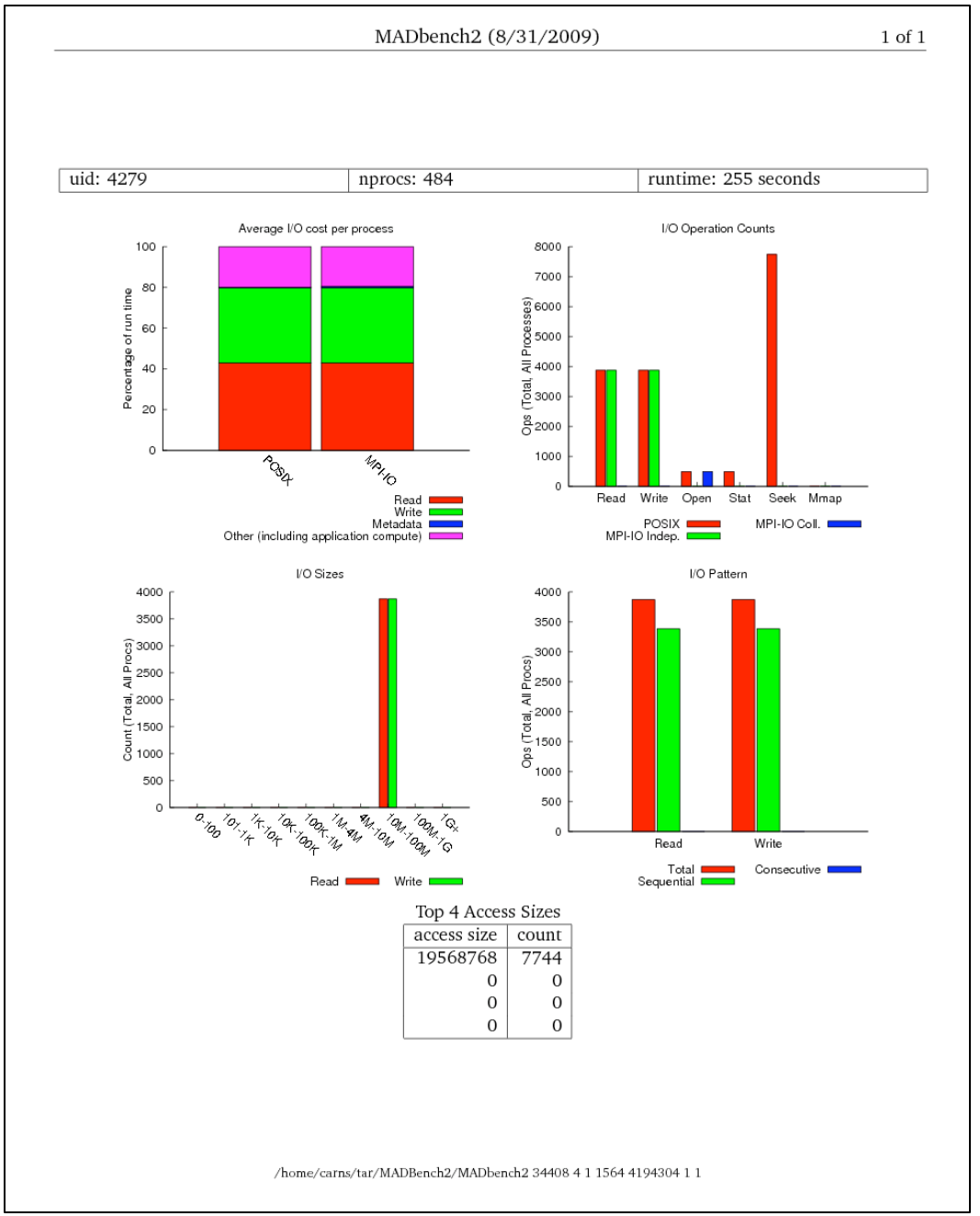
- Use PMPI and ld wrappers to intercept I/O functions
  - Requires re-linking, but no code modification
  - Can be transparently included in mpicc
  - Compatible with a variety of compilers
- Record statistics independently at each process
  - Compact summary rather than verbatim record
  - Independent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

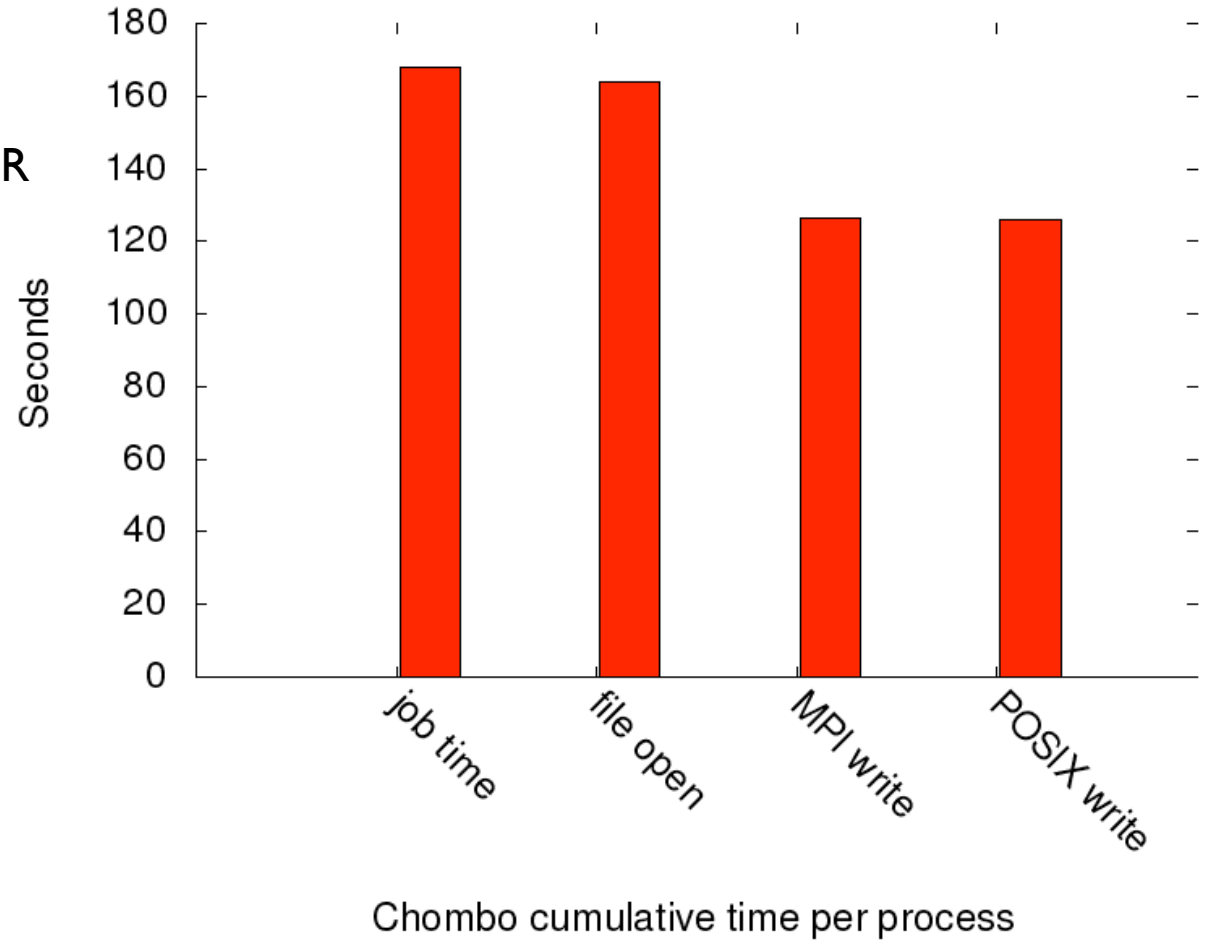
Job Summary

- Job summary tool shows characteristics "at a glance"
- MADBench2 example
- 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
- FIXME – the charts are pretty hard to read



Chombo I/O Benchmark

- Checkpoint writes from AMR framework
- Uses HDF5 for I/O
- Code base is complex
- 512 processes
- 18.24 GB output file

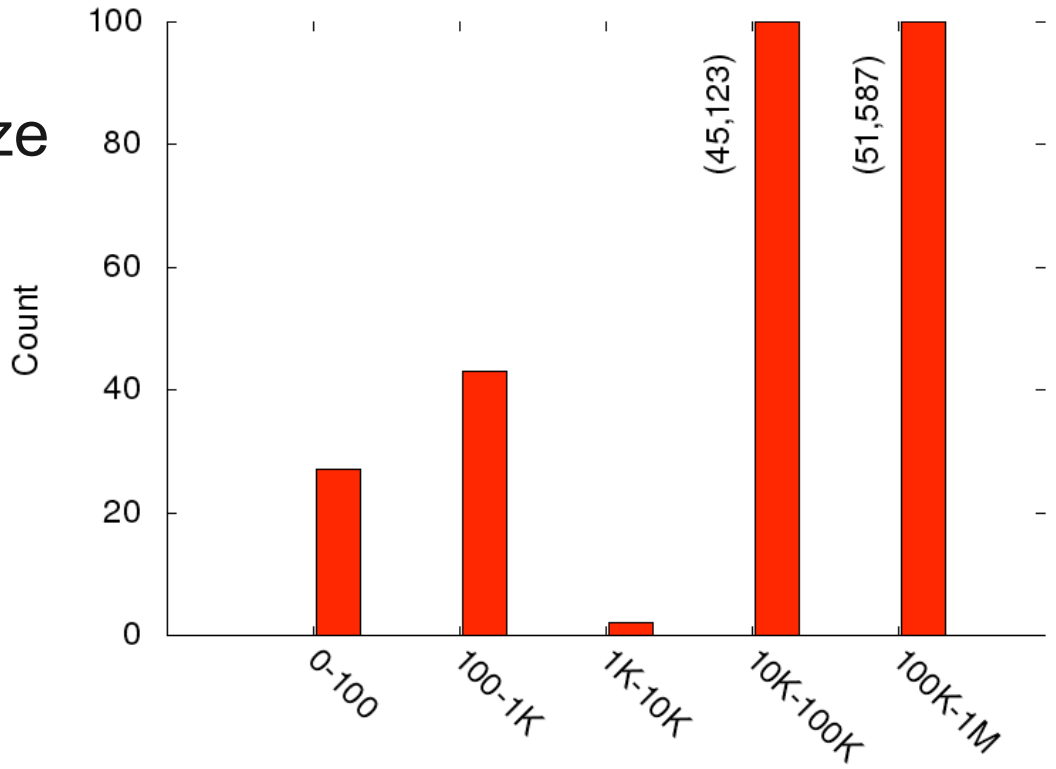


Chombo cumulative time per process

- Why does the I/O take so long in this case?
- Why isn't it busy writing data the whole time?

Chombo I/O Benchmark

- Many write operations, with none over 1 MB in size
- Most common access size is 28,800 (occurs 15622 times)
- No MPI datatypes or collectives
- All processes frequently seek forward between writes
  - Consecutive: 49.25%
  - Sequential: 99.98%
  - Unaligned in file: 99.99%
  - Several recurring regular stride patterns



Chombo write size histogram, 512 procs

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:

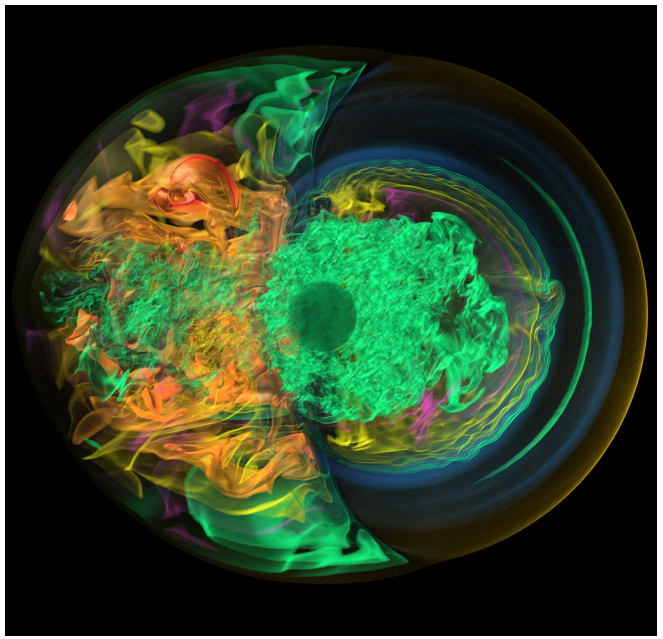
  http://www.mcs.anl.gov/research/projects/darshan

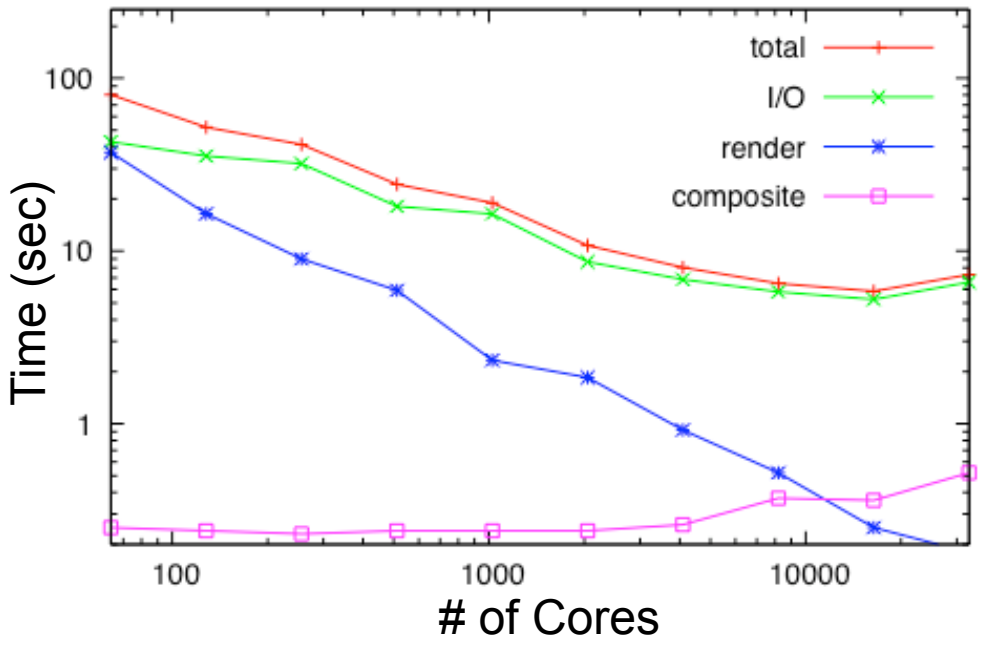
I/O in Parallel Volume Rendering

Thanks to Tom Peterka (ANL) and Hongfeng Yu and Kwan-Liu Ma (UC Davis) for providing the code on which this material is based.

Parallel Volume Rendering

- Supernova model with focus on core collapse
- Parallel rendering techniques scale to 16k cores on Argonne Blue Gene/P
- Produce a series of time steps
- 1120^3 elements (~1.4 billion)
- Structured grid
- Simulated and rendered on multiple platforms, sites
- I/O time now largest component of runtime





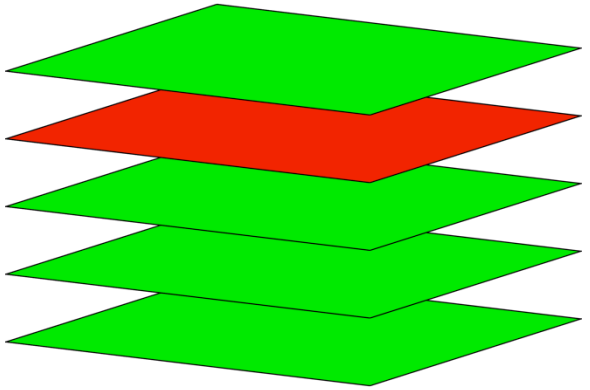
The I/O Code (essentially):

```
MPI_Init(&argc, &argv);
ncmpi_open(MPI_COMM_WORLD, argv[1], NC_NOWRITE,
   info, &ncid));
ncmpi_inq_varid(ncid, argv[2], &varid);
buffer =calloc(sizes[0]*sizes[1]*sizes[2],sizeof(float));
for (i=0; i<blocks; i++) {
   decompose(rank, nprocs, ndims, dims, starts, sizes);
   ncmpi_get_vara_float_all(ncid, varid,
        starts, sizes, buffer);
}
ncmpi_close(ncid));
MPI_Finalize();
```

- Read-only workload: no switch between define/data mode
- Omits error checking, full use of inquire (ncmpi_inq_*) routines
- Collective I/O of noncontiguous (in file) data
- “black box” decompose function:
  - divide 1120^3 elements into roughly equal mini-cubes
  - “face-wise” decomposition ideal for I/O access, but poor fit for volume rendering algorithms

Volume Rendering and pNetCDF

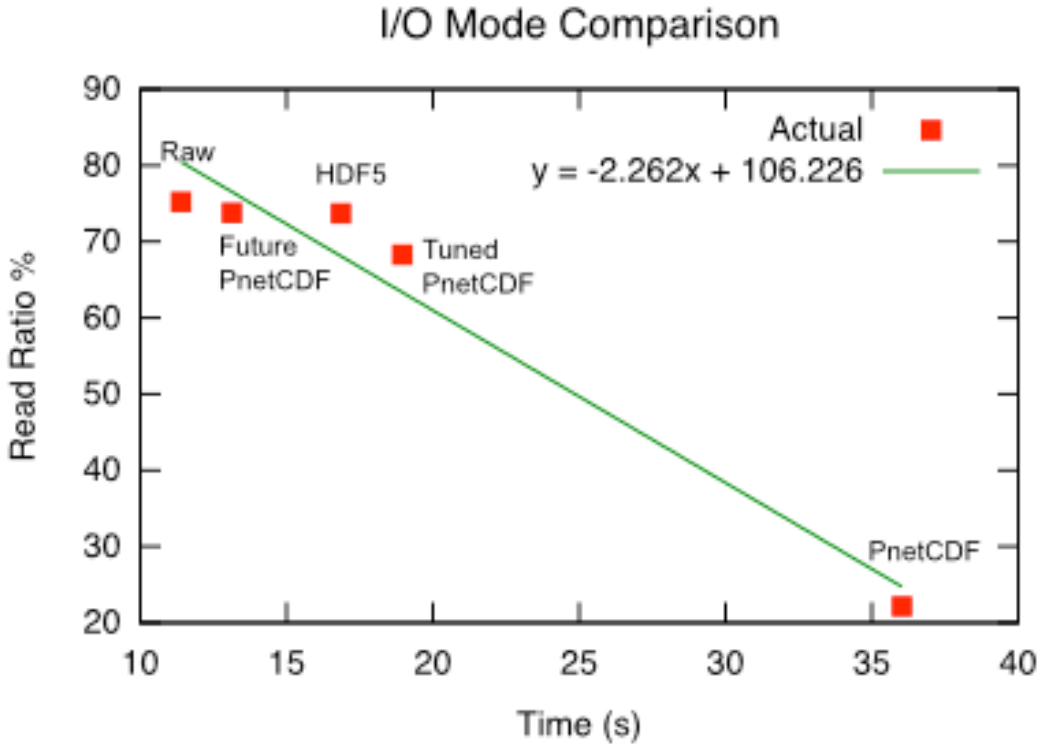
- Original data: netCDF formatted
- Two approaches for I/O
  - Pre-processing: extract each variable to separate file
    - Lengthy, duplicates data
  - Native: read data in parallel, on-demand from dataset
    - Skip preprocessing step but slower than raw
- Why so slow?
  - 5 large “record” variables in a single netcdf file
    - Interleaved on per-record basis
  - Bad interaction with default MPI-IO parameters



Record variable interleaving is performed in N-1 dimension slices, where N is the number of dimensions in the variable.

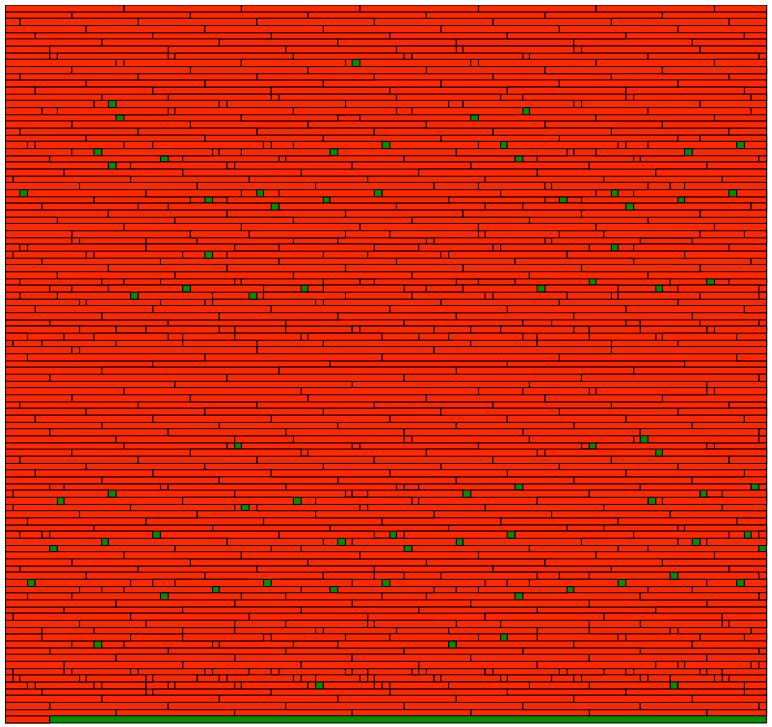
Access Method Comparison

- MPI-IO hints matter
- HDF5: many small metadata reads
- Interleaved record format: bad news

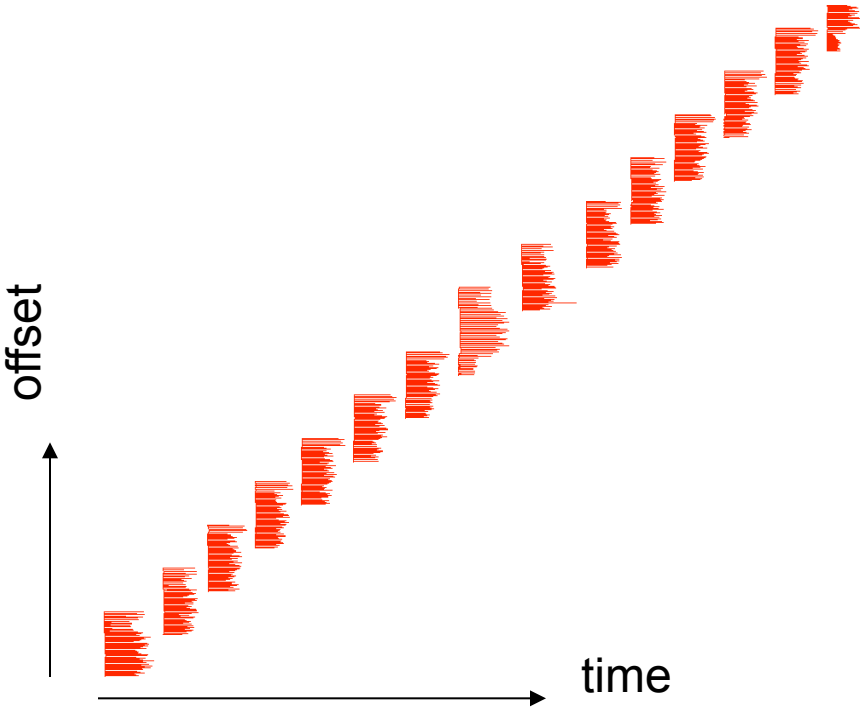


API	time (s)	accesses	read data (MB)	efficency
MPI (raw data)	11.388	960	7126	75.20%
PnetCDF (no hints)	36.030	1863	24200	22.15%
PnetCDF (hints)	18.946	2178	7848	68.29%
HDF5	16.862	23450	7270	73.72%
PnetCDF (beta)	13.128	923	7262	73.79%

Analysis: MPI-IO to extracted data

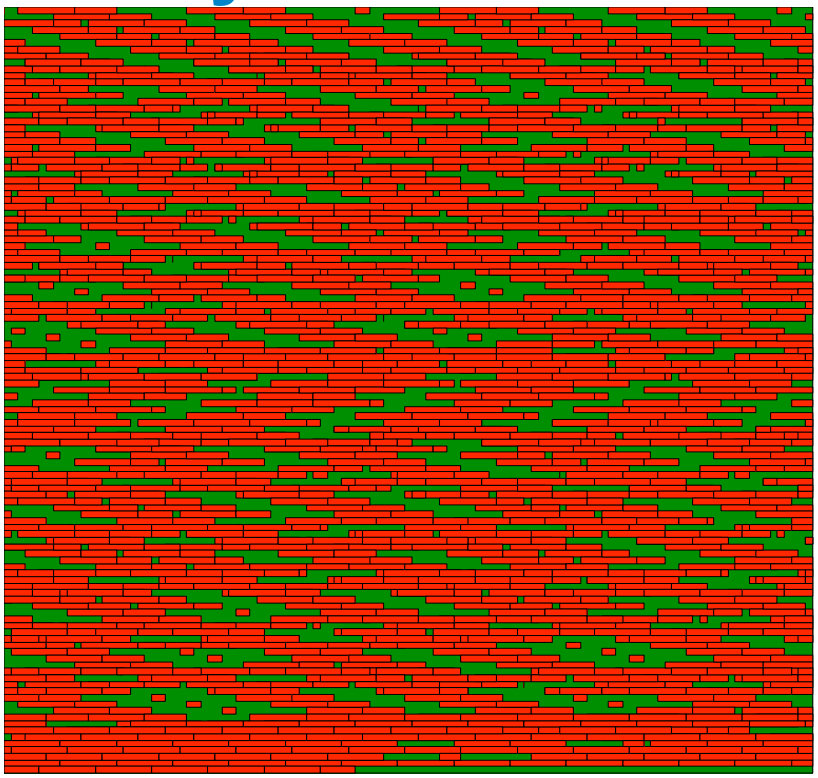


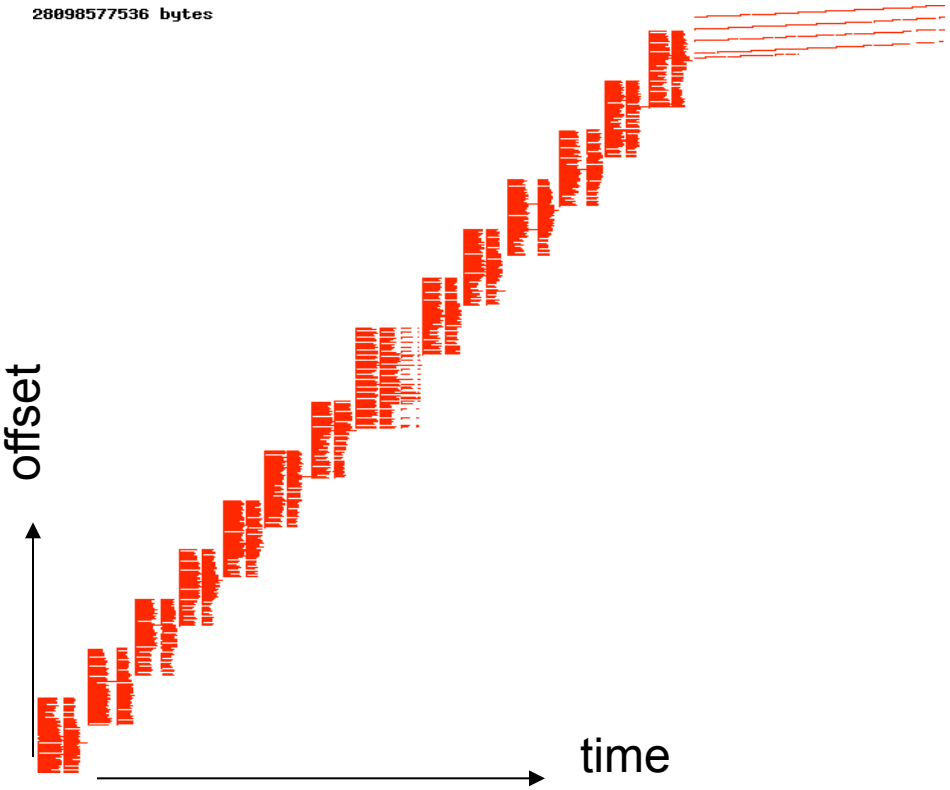
- 2D depiction of file accesses
- Pre-processing extracted variable
- 5GB file



- 15 rounds of I/O
- Round i+1 overlaps slightly with round i (75% efficiency)

Analysis: Parallel netCDF, no hints

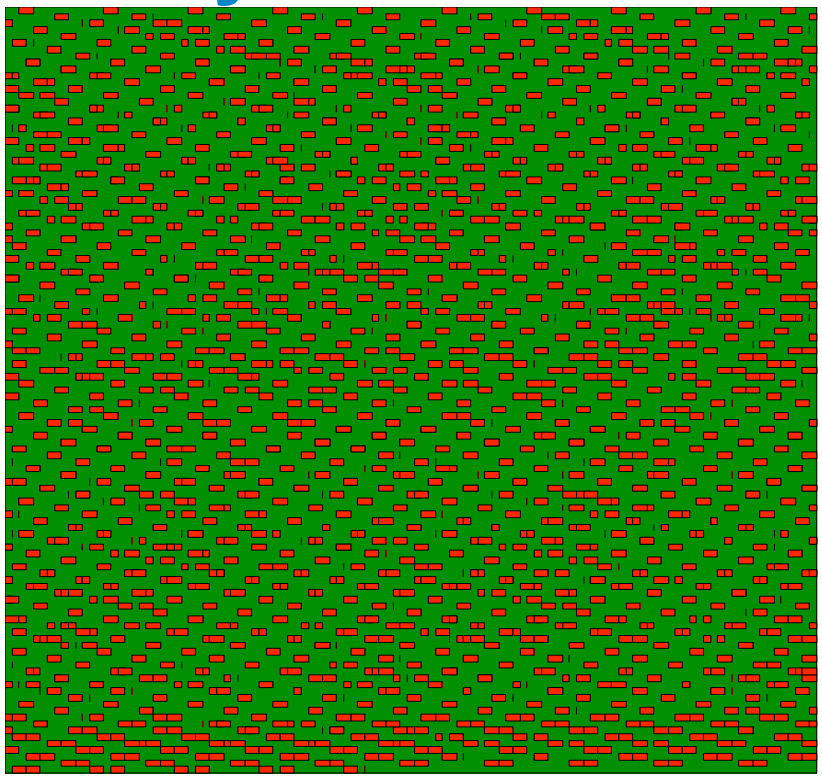


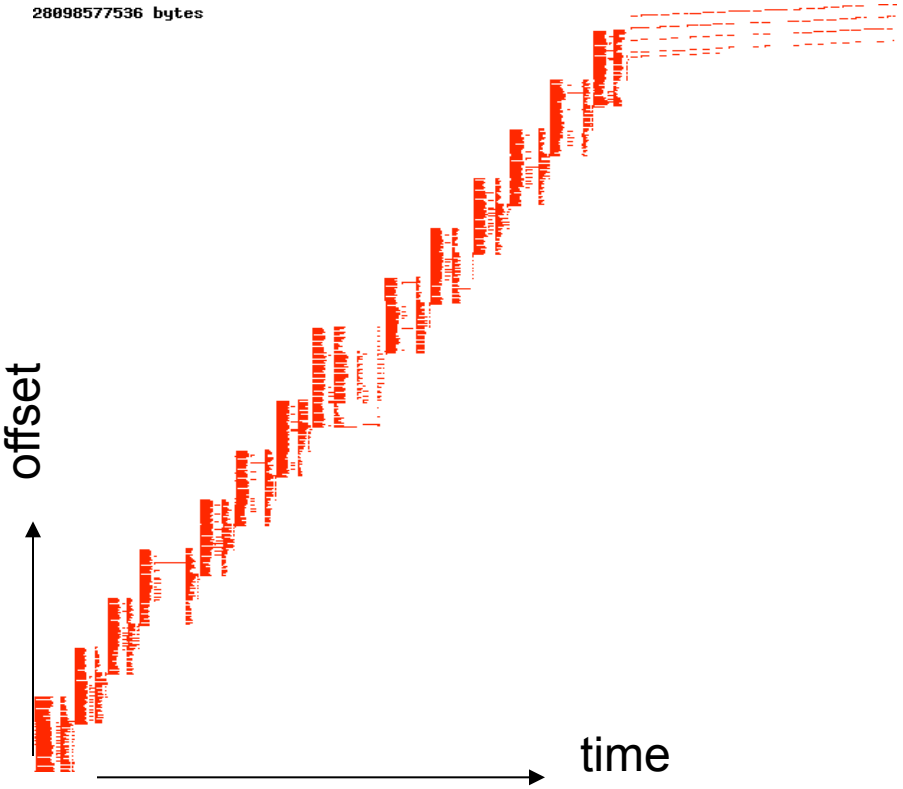


- Block depiction of 28 GB file
- Record variable scattered
- Reading in way too much data!

- Y axis larger here
- Default “cb_buffer_size” hint not good for interleaved netCDF record variables

Analysis: Parallel netCDF, hints

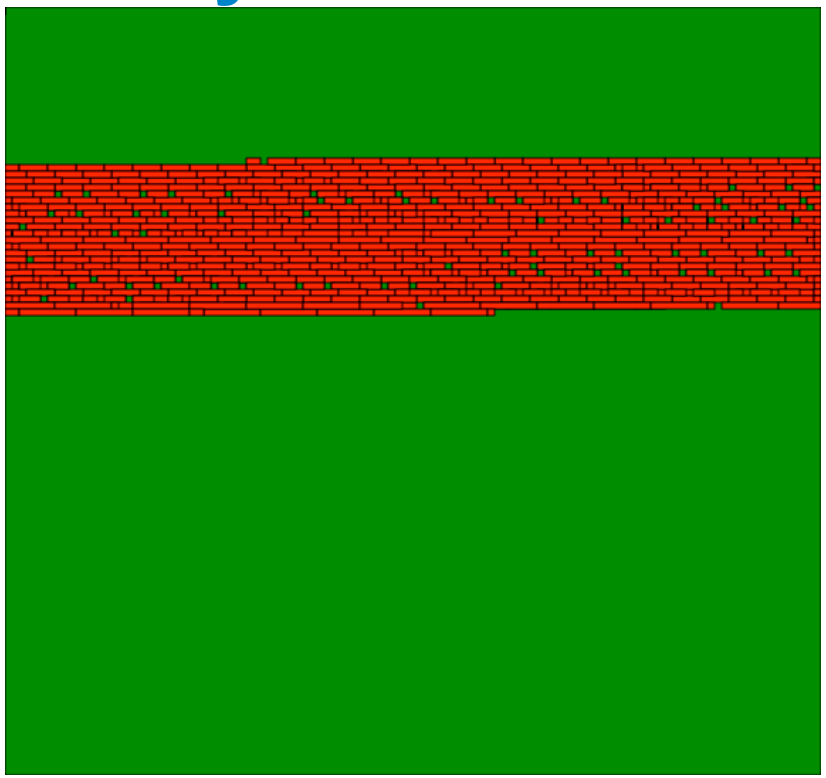


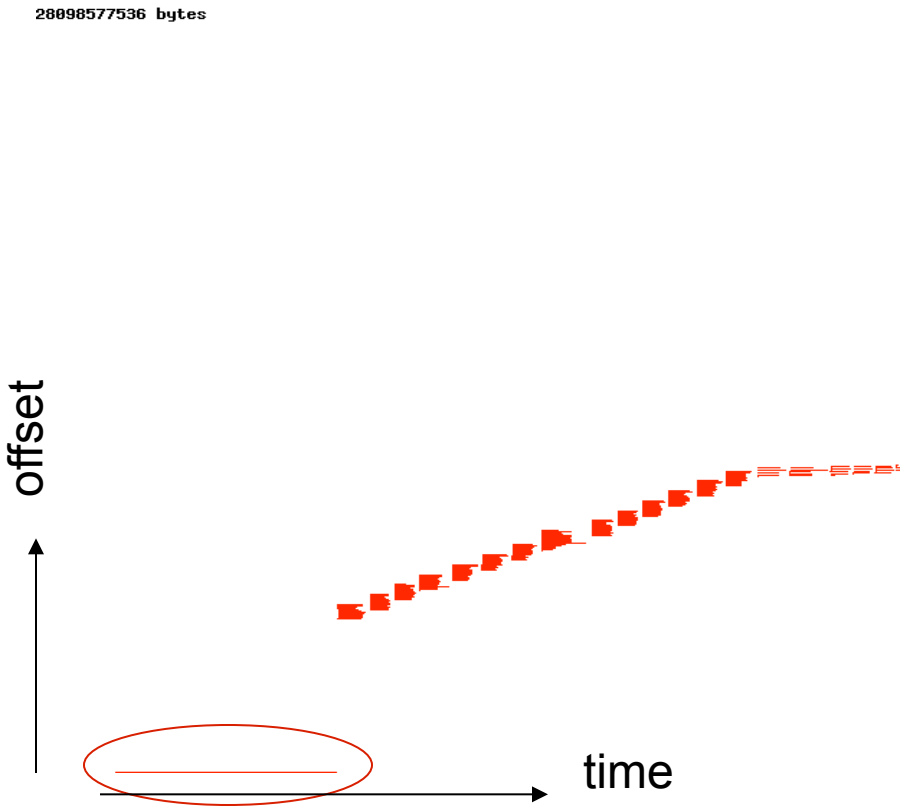


- With tuning, much less reading
- Better efficiency, but still short of MPI-IO

- Still some overlap
- “cb_buffer_size” now size of one netCDF record
- Better efficiency, at slight perf cost

Analysis: Parallel HDF5

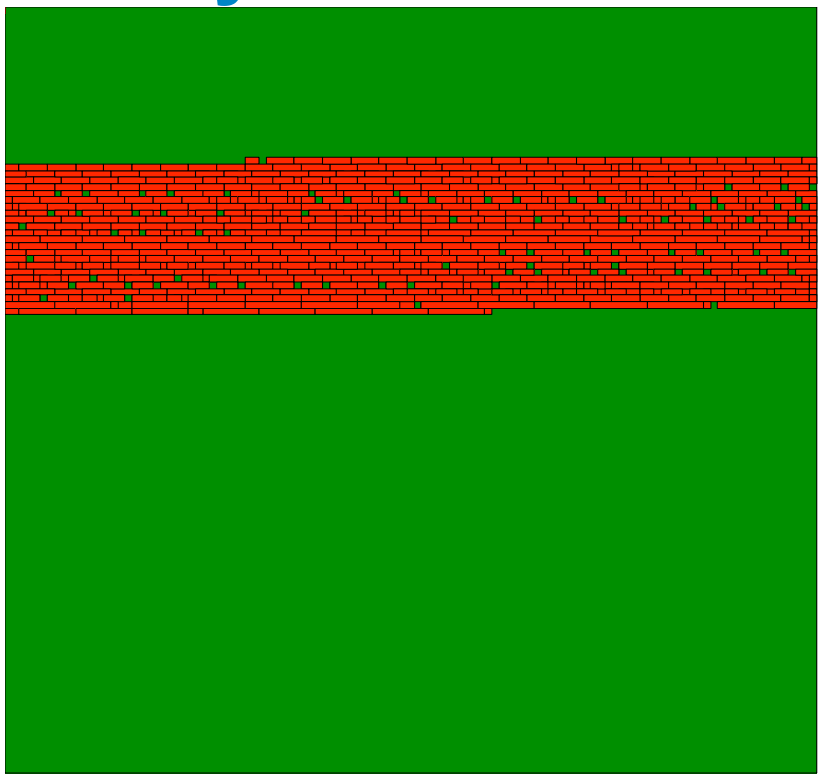


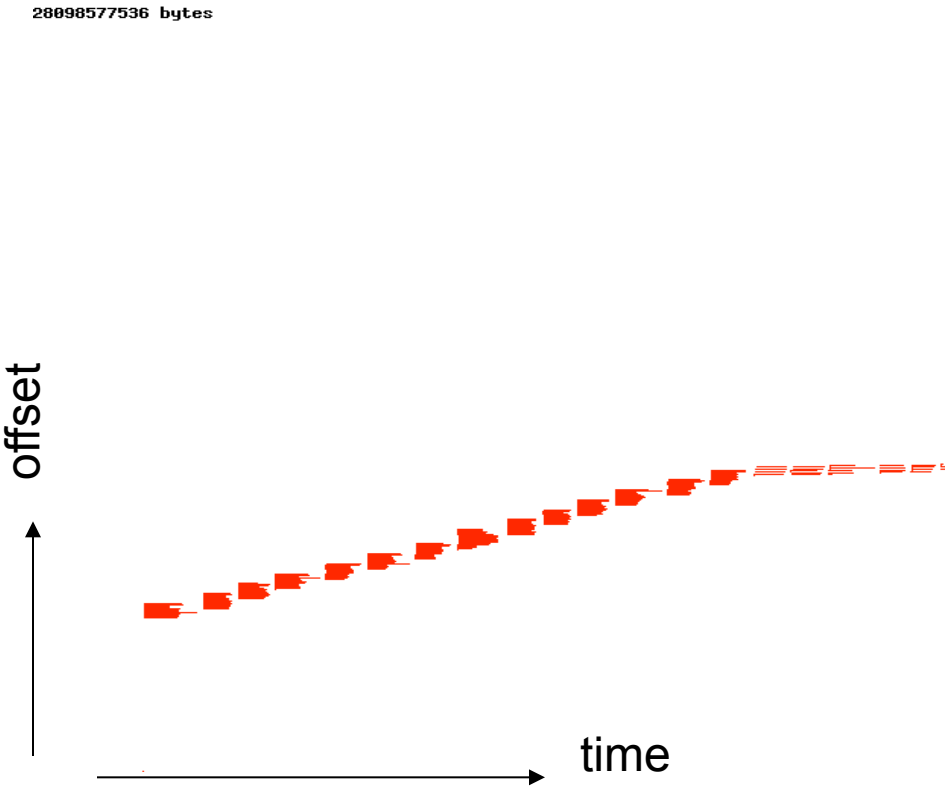


- Different file format, different characteristics
- Data exhibits spatial locality

- Thousands of metadata reads
  - All clients read MD from file
- Reads could be batched. Not sure why not (implementation detail).

Analysis: new Parallel netCDF





- Development effort to relax netCDF file format limits
- No need for record variables
- Data nice and compact like MPI-IO and HDF5

- Rank 0 reads header, broadcasts to others
  - Much more scalable approach
- Approaching MPI-IO efficiency
- Maintains netCDF benefits
  - Portable, self-describing, etc.

Future Storage Technologies

Storage Futures

- pNFS
  - An extension to the NFSv4 file system protocol standard that allows direct, parallel I/O between clients and storage devices
  - Eliminates the scaling bottleneck found in today's NAS systems
  - Supports multiple types of back-end storage systems, including traditional block storage, other file servers, and object storage systems
- FLASH and other non-volatile devices
  - New level in storage hierarchy

Why a Standard for Parallel I/O?

- NFS is the only network file system standard
  - Proprietary file systems have unique advantages, but can cause lock-in
- NFS widens the playing field
  - Panasas, IBM, EMC want to bring their experience in large scale, high-performance file systems into the NFS community. Sun and NetApp want a standard HPC solution.
  - Broader market benefits vendors
  - More competition benefits customers
- What about open source
  - NFSv4 Linux client is very important for NFSv4 adoption, and therefore pNFS
  - Still need vendors that are willing to do the heavy lifting required in quality assurance for mission critical storage

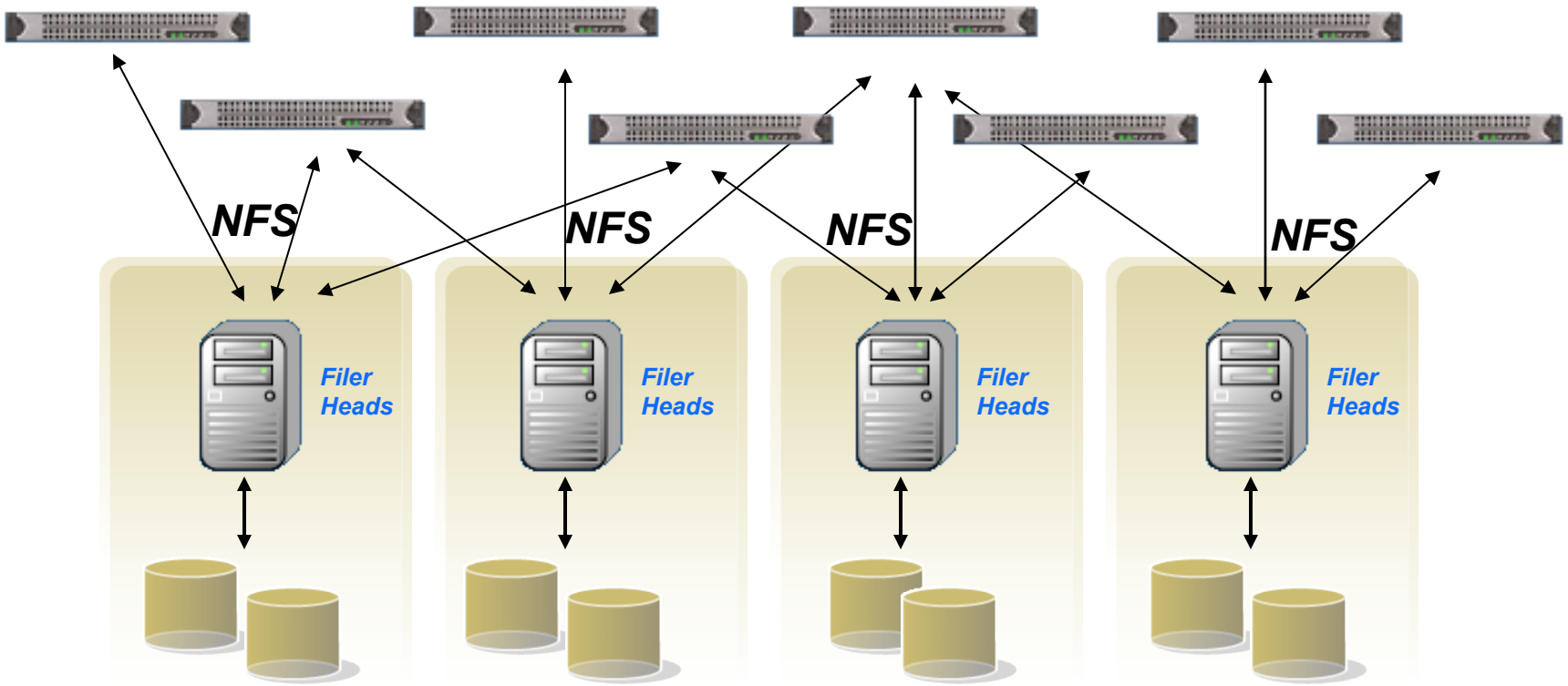
NFSv4 and pNFS

- NFS created in ’80s to share data among engineering workstations
- NFSv3 widely deployed
- NFSv4 several years in the making, lots of new stuff
  - Integrated Kerberos (or PKI) user authentication
  - Integrated File Locking and Open Delegations (stateful server!)
  - ACLs (hybrid of Windows and POSIX models)
  - Official path to add (optional) extensions
- NFSv4.1 adds even more
  - pNFS for parallel IO
  - Directory Delegations for efficiency
  - RPC Sessions for robustness, better RDMA support

Whence pNFS

- Gary Grider (LANL) and Lee Ward (Sandia)
  - Spoke with Garth Gibson about the idea of parallel IO for NFS in 2003
- Garth Gibson (Panasas/CMU) and Peter Honeyman (UMich/CITI)
  - Hosted pNFS workshop at Ann Arbor in December 2003
- Garth Gibson, Peter Corbett (NetApp), Brent Welch
  - Wrote initial pNFS IETF drafts, presented to IETF in July and November 2004
- Andy Adamson (CITI), David Black (EMC), Garth Goodson (NetApp), Tom Pisek (Sun), Benny Halevy (Panasas), Dave Noveck (NetApp), Spenser Shepler (Sun), Brian Pawlowski (NetApp), Marc Eshel (IBM), (*Many Others* …)
  - Dean Hildebrand (CITI) did pNFS prototype based on PVFS
  - NFSv4 working group commented on drafts in 2005, folded pNFS into the 4.1 minorversion draft in 2006
- pNFS approved by IETF December 2008
  - expect RFC in 2009

Traditional NAS

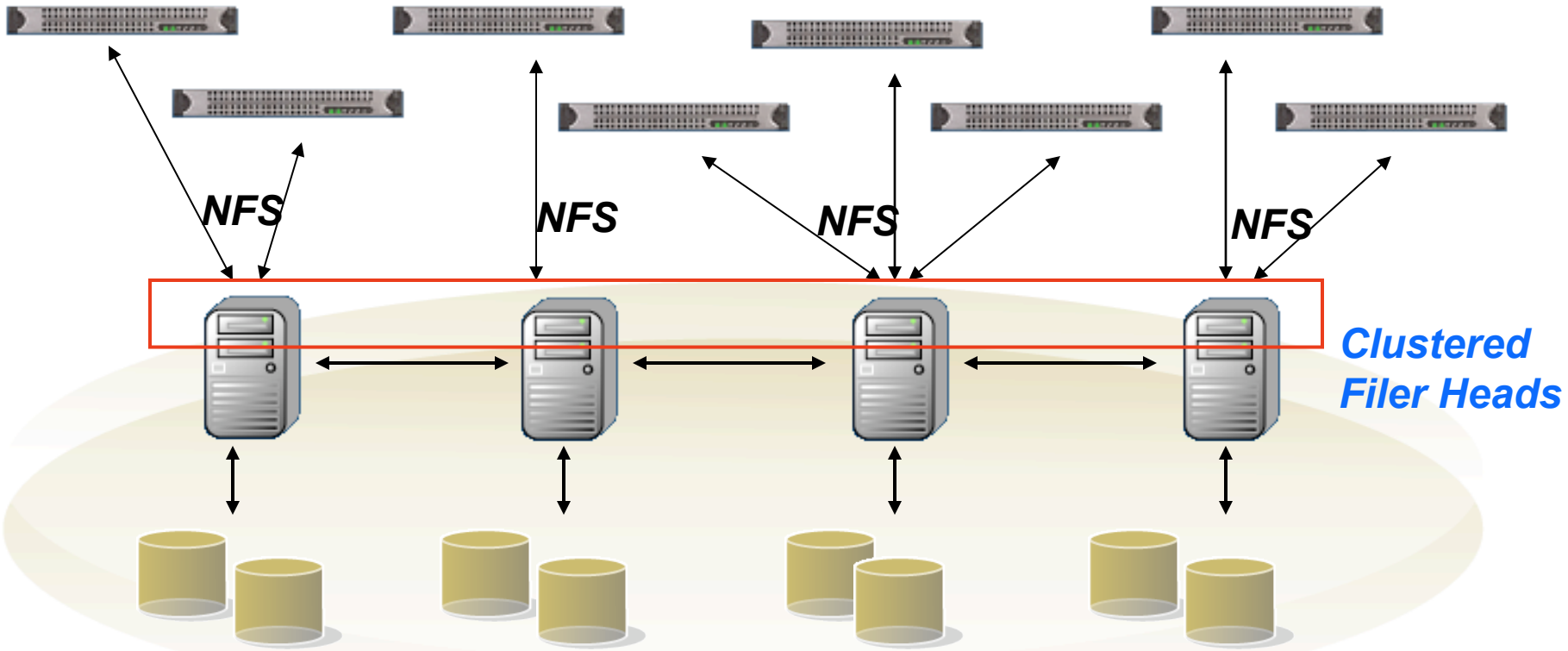


"Islands of Storage"

Filer Heads create I/O performance bottlenecks

Multiple instances create management challenges

Clustered NAS

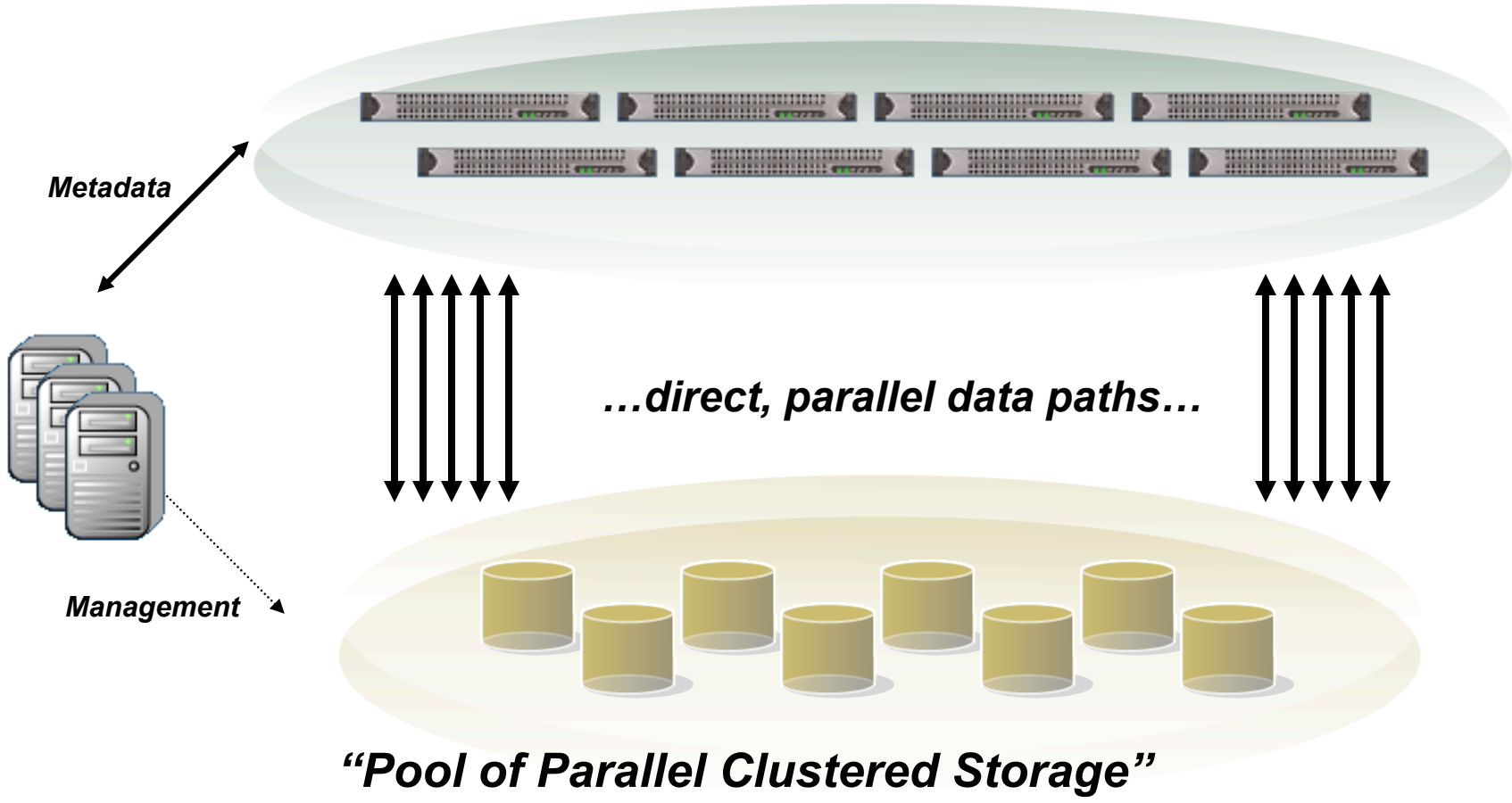


"Bridged Islands of Storage"

"In-band" Filer Head protocol creates I/O performance bottlenecks

Load balancing becomes management & performance issue

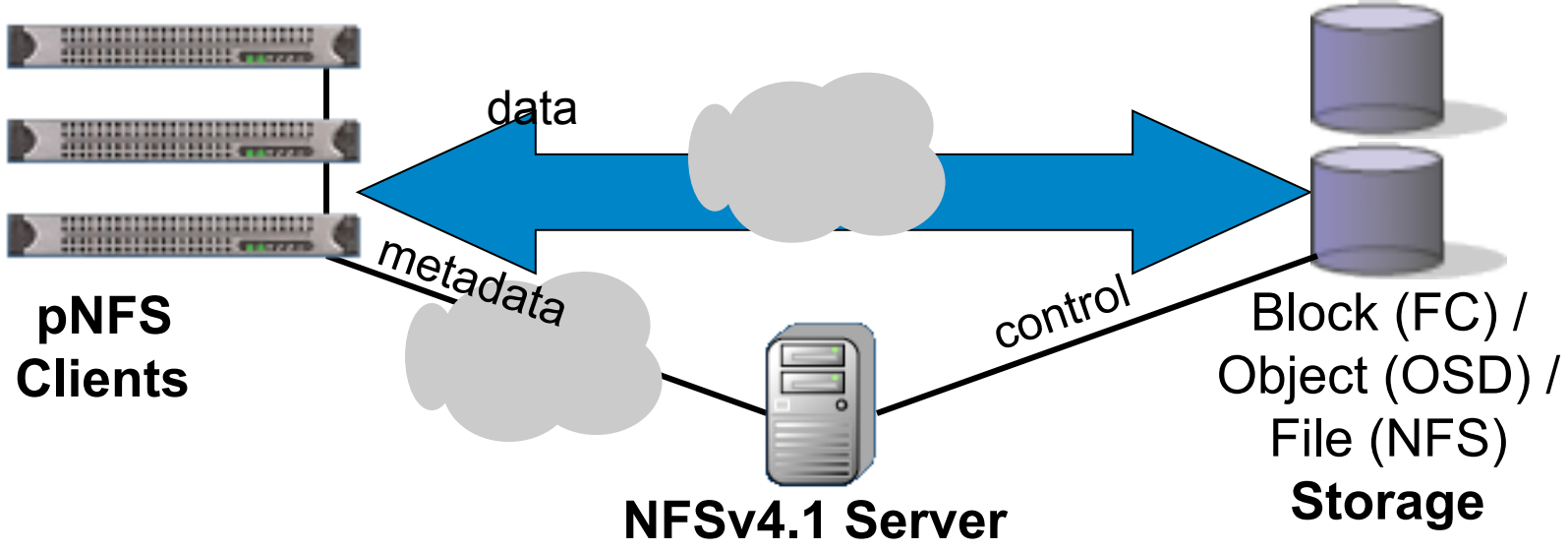
Parallel IO



I/O Performance Bottlenecks and Management Challenges Solved as Filers Removed from Data Path

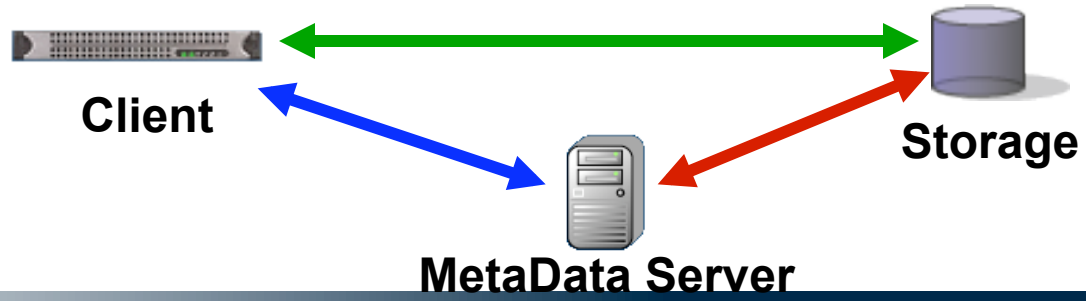
pNFS: Standard Storage Clusters

- pNFS is an extension to the Network File System v4 protocol standard
- Allows for parallel and direct access
  - From Parallel Network File System clients
  - To Storage Devices over multiple storage protocols
  - Moves the Network File System server out of the data path



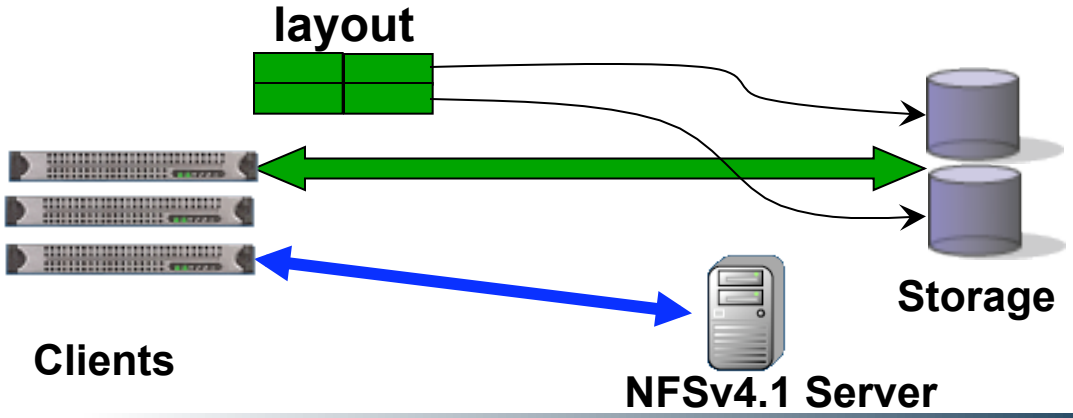
The pNFS Standard

- 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



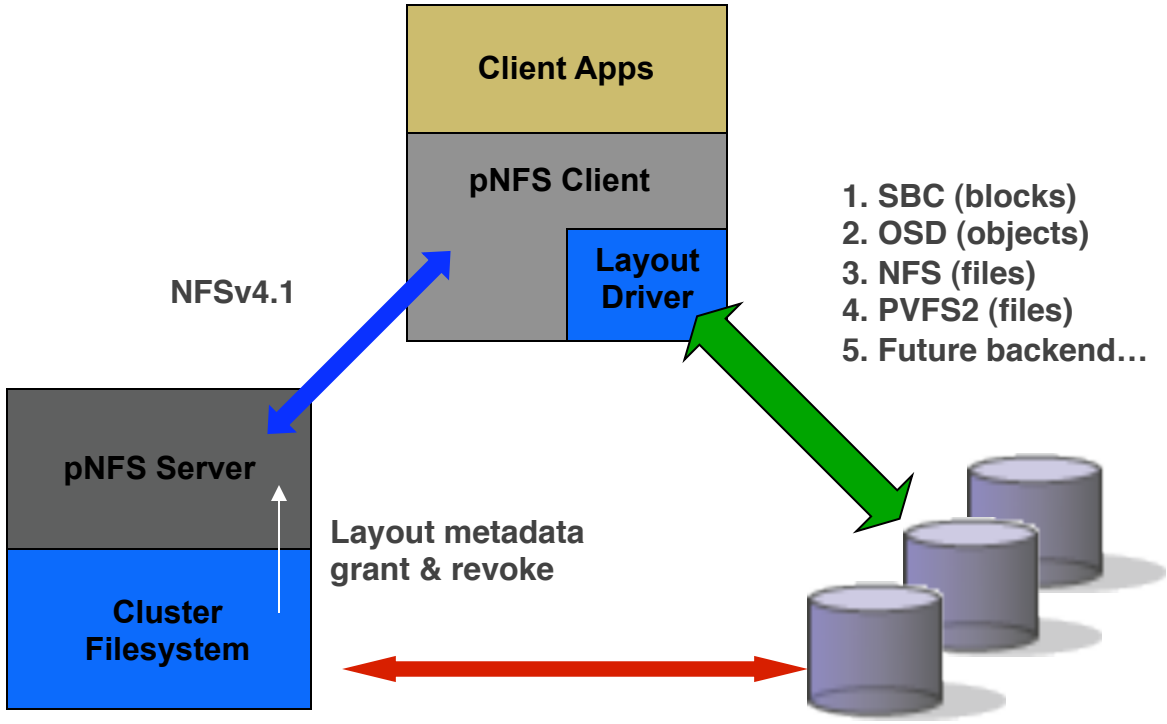
pNFS Layouts

- Client gets a *layout* from the NFS Server
- The layout maps the file onto storage devices and addresses
- 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



pNFS Client

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



pNFS is not…

- Improved cache consistency
  - NFS has open-to-close consistency enforced by client polling of attributes
  - NFSv4.1 directory delegations can reduce polling overhead
- Perfect POSIX semantics in a distributed file system
  - NFS semantics are good enough (or, all we'll give you)
  - But note also the POSIX High End Computing Extensions Working Group
    - http://www.opengroup.org/platform/hecewg/
- Clustered metadata
  - Not a server-to-server protocol for scaling metadata
  - But, it doesn't preclude such a mechanism

Is pNFS Enough?

- Standard for out-of-band metadata
  - Great start to avoid classic server bottle neck
  - NFS has already relaxed some semantics to favor performance
  - But there are certainly some workloads that will still hurt
- Standard framework for clients of different storage backends
  - Files
  - Objects
  - Blocks
  - PVFS2
  - Your project… (e.g., dcache.org)

pNFS Status

- Implementation interoperability continues
  - San Jose Connect-a-thon March ’06, February ’07, May ’08, June ‘09
  - Ann Arbor NFS Bake-a-thon September ’06, October ’07
  - Dallas pNFS inter-op, June ’07, Austin February ’08, Sept ’08, **October ‘09**
- Server vendors waiting for Linux client
  - Sun, NetApp, EMC, IBM, Panasas, …
  - 2.6.30
    - exofs object storage file system (local) and iSCSI/OSDv2
  - 2.6.31
    - most of nfsv4.1: sessions, 4.1 as an option, no pnfs yet
    - server back channel is absent
  - 2.6.32
    - Finish nfsv4.1 including server callbacks
  - 2.6.33
    - Merge window opens around the end of the year
  - Goal to complete patch adoption by Q3 2010

How to use pNFS today

- Up-to-date GIT tree from Linux developers
  - bhalevy@panasas.com manages the source trees
- RedHat fedora RPMs that include pNFS
  - steved@redhat.com builds experimental packages
- pNFS mailing list, pnfs@linux-nfs.org
- http://open-osd.org
  - Useful to get to OSD target, the user level program
  - Exofs uses kernel initiator, need the target

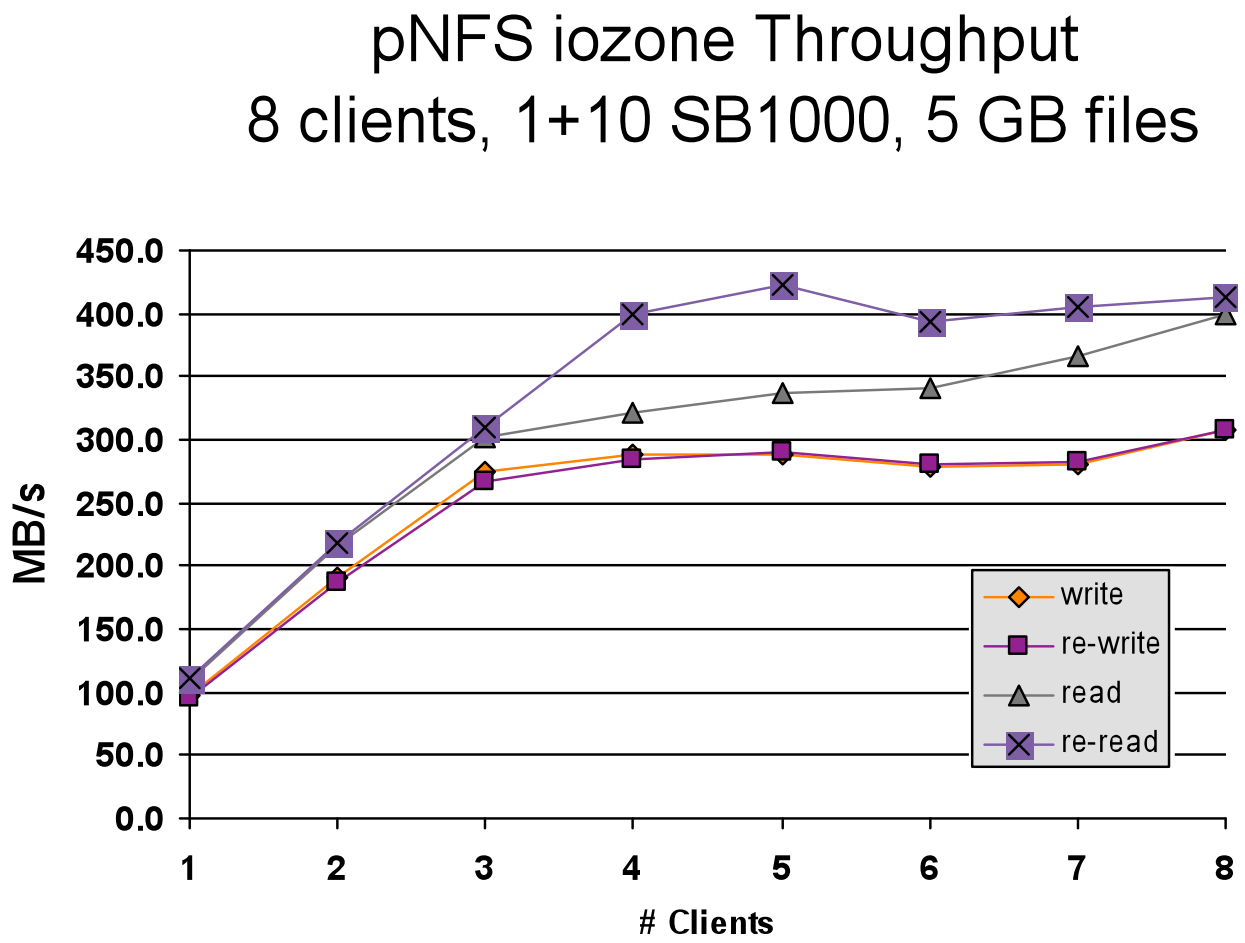
How to use pNFS today

- Benny's git tree:
  git://linux-nfs.org/~bhalevy/linux-pnfs.git
- The the kernel rpms can be found at:
  http://fedorapeople.org/~steved/repos/pnfs/i686
  http://fedorapeople.org/~steved/repos/pnfs/x86_64
- The source rpm can be found at:
  http://fedorapeople.org/~steved/repos/pnfs/source/
- Bug database
  https://bugzilla.linux-nfs.org/index.cgi
- OSD target
  http://open-osd.org/

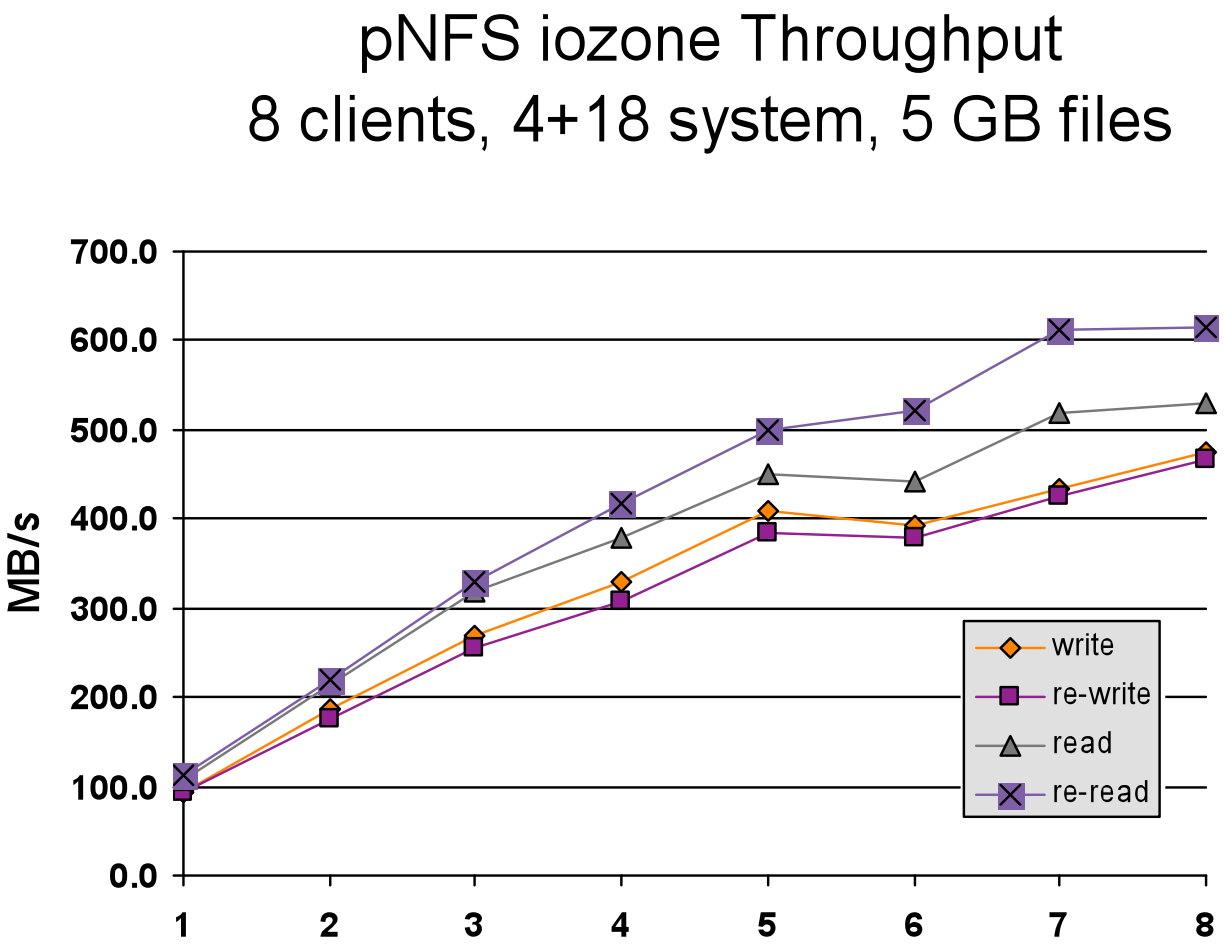
Key pNFS Participants

- Panasas (Objects)
- ORNL and ESSC/DoD funding Linux pNFS development
- Network Appliance (Files over NFSv4)
- IBM (Files, based on GPFS)
- EMC (Blocks, HighRoad MPFSi)
- Sun (Files over NFSv4)
- U of Michigan/CITI (Linux maintainers, EMC and Microsoft contracts)
- LSI – open source block-based server
- DESY – Java-based implementation

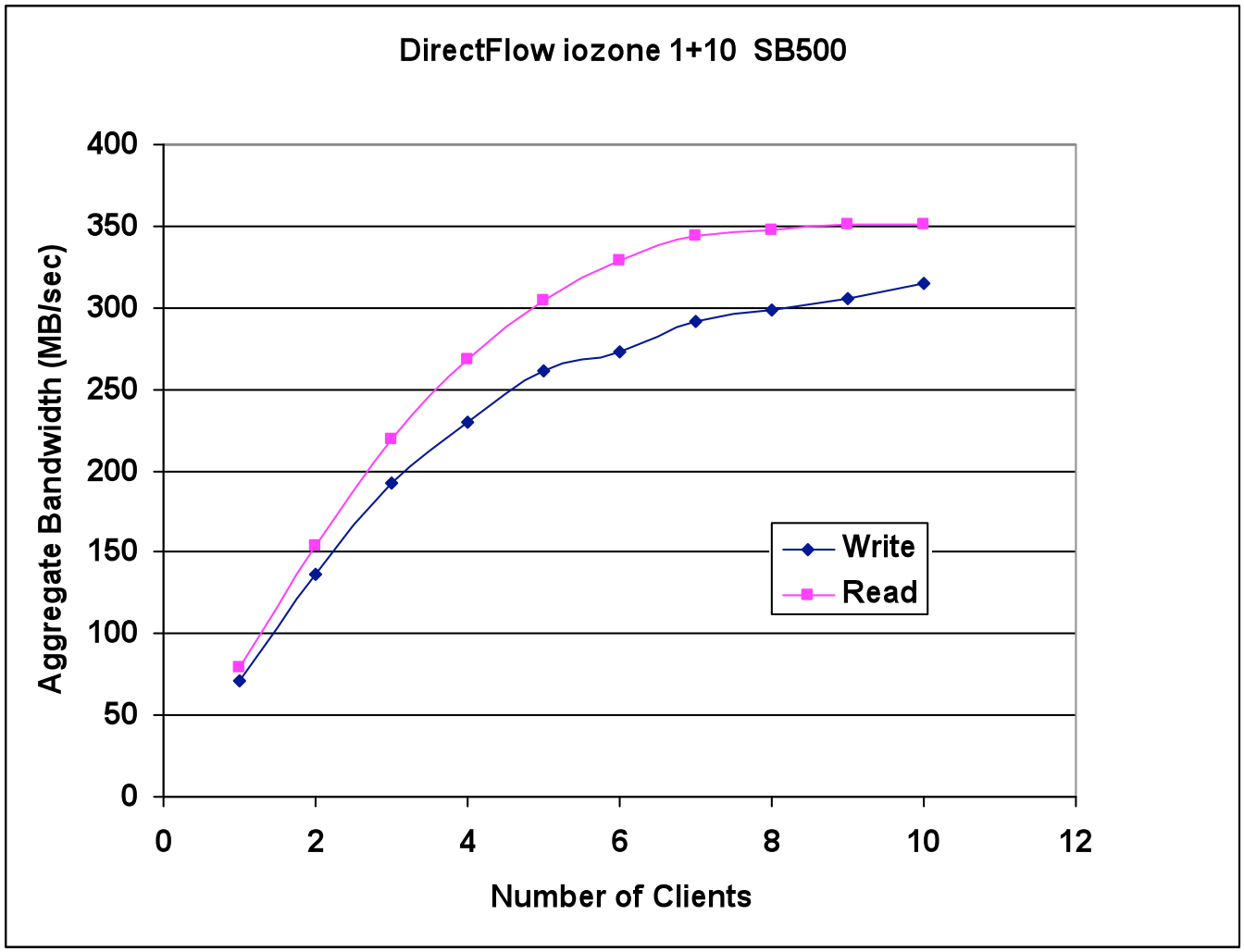
Prototype PNFS Performance



Prototype PNFS Performance



Panasas DirectFlow Performance



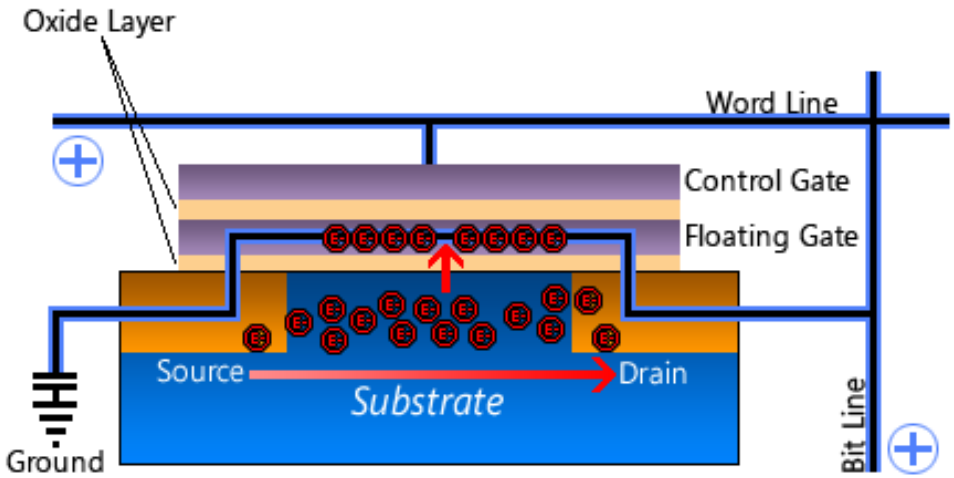
FLASH and Nonvolatile Storage

The problem with rotating media

- Areal density increases by 40%/year
  - Per drive capacity increases by 70% to 100% per year
  - 2TB “enterprise SATA” drives available in 2009
  - 3TB desktop drives available first half of 2010
  - Drive vendors prepared to continue like this for years to come
- Drive interface speed increases by 10% per year
  - Takes longer and longer to completely read each new generation of drive
- Seek times and rotational speeds not increasing all that much
  - 15,000 RPM and 2.5 ms/sec still the norm for high end
  - Significant power problems with higher RPM and faster seeks
    - Aerodynamic drag and friction loads go as the square of speed

FLASH is…

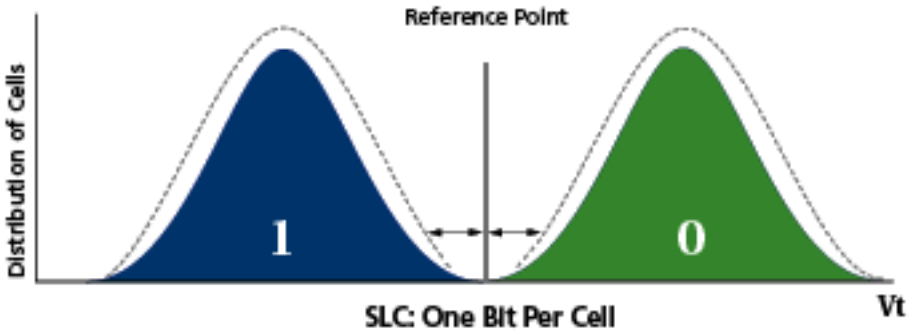
- Non-volatile
  - Each bit is stored in a “floating gate” that holds value without power
  - Electrons can leak, so shelf life and write count is limited
- Page-oriented
  - Read, write, and erase operations apply to large chunks
  - Smaller (e.g., 4K) read/write block based on addressing logic
  - Larger (e.g., 256K) erase block to amortize the time it takes to erase
- Medium speed
  - Slower than DRAM
  - Faster than disks for reading
  - Write speed dependent on workl
- Relatively cheap

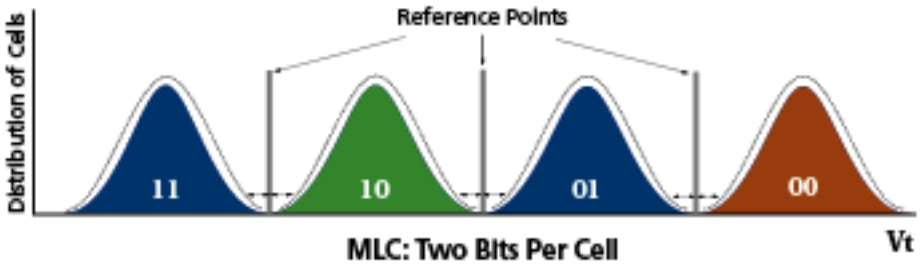


http://icrontic.com/articles/how_ssds_work

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 now, 3 & 4 soon)
  - N bits requires 2^N Vt levels
  - 10^4 write cycles per page
  - Denser and cheaper, but slower and less reliable
- Wear leveling is critical
  - Pre-erase blocks before writing is required
  - Page map indirection allows shuffling of pages to do wear leveling

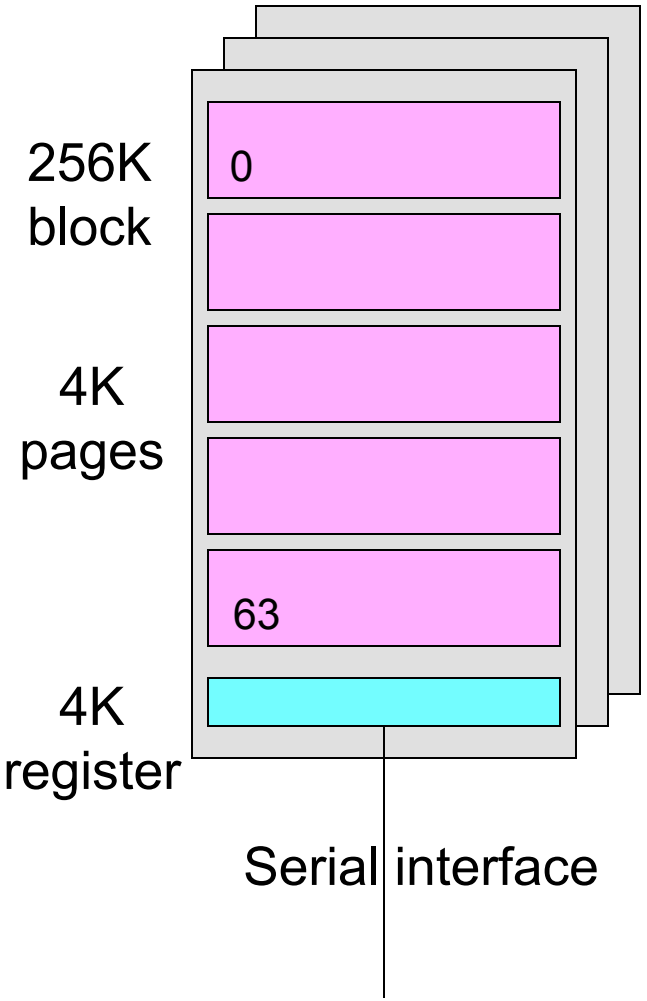




http://www.micron.com/nandcom/

FLASH Speeds

- Samsung 4GB Device
  - 16K erase blocks



100 usec	Transfer 4K over serial interface	40 MB/sec
25 usec	Load 4K register from Flash	160 MB/ sec
125 usec	Read latency	32 MB/sec
200 usec	Store 4K register to FLASH	20 MB/sec
225 usec	Write latency	16 MB/sec
1.5 msec	Erase 256K block	170 MB/ sec
1.725 msec	Worse case write	2.3 MB/sec

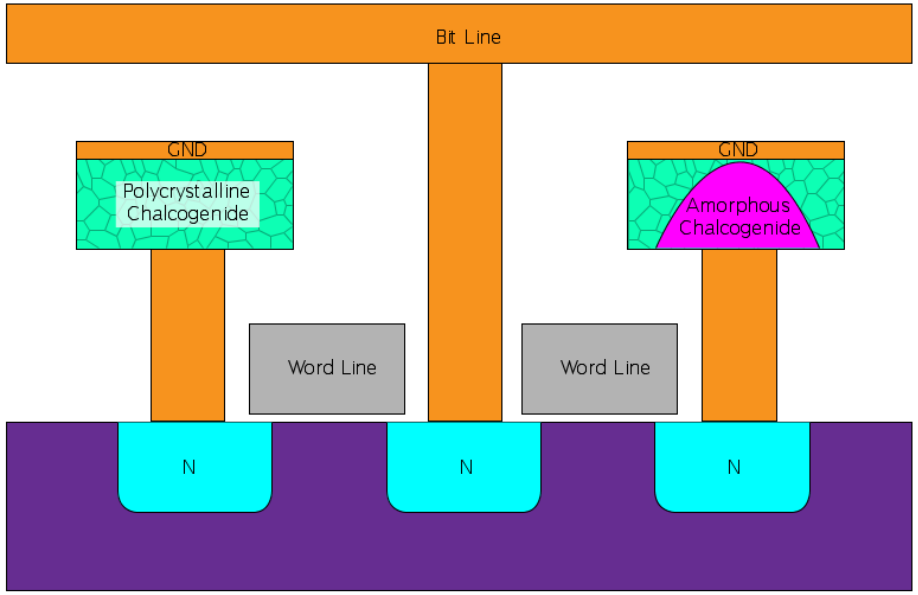
- Write performance heavily dependent on workload and wear leveling algorithms
- Writes are slower with less free space

FLASH in the Storage Hierarchy

- On the compute nodes
  - High reliability local storage for OS partition
  - Local cache for memory checkpoints ?
    - Device write speeds range from 4 MB/sec for a cheap USB, to 80 or 100 MB/sec for MTron or Zeus, up to 600 MB/sec for Fusion IO
  - One Fusion IO SSD (Solid State Disk) could double cost of compute node
- On the storage server
  - Metadata storage
  - Low latency log device
  - Replacement for NVRAM ? Probably not enough write bandwidth to absorb all the write data

Phase Change Memory

- GST: Germanium-Antimony-Tellurium Chalcogenide glass
  - Crystalline vs. Amorphous structure has different resistance
  - Write a bit by heating to two different temperatures
- DRAM-like device – no block erase required
- Slow (like FLASH) writes, but permanent
  - No leakage, but wear leveling still an issue
- Read speed 2x slower than DRAM today, and improving
- Manufacturability advantage over DRAM and FLASH when feature size gets small (20nm and below)
- Storage Devices in 2010
- Main memory by 2015 ?



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

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://hdf.ncsa.uiuc.edu/HDF5/
  - http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor/index.html
- 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
  - draft-ietf-nfsv4-minorversion1-26.txt
  - draft-ietf-nfsv4-pnfs-obj-09.txt
  - draft-ietf-nfsv4-pnfs-block-09.txt
- 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) and Bill Loewe (Panasas) for their help in creating this material and presenting this tutorial in prior years.