Clustered and Parallel Storage System Technologies
FAST09

Brent Welch, Marc Unangst
{welch,mju}@panasas.com
Panasas, Inc.

About Us

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

Thanks to the Robs that did a tutorial with us at SC08
- Rob Latham (robl@mcs.anl.gov)
- Rob Ross (rross@mcs.anl.gov)

Outline of the Day

Part 1

Introduction

Storage System Models

Parallel File Systems

- GPFS
- PVFS
- Panasas
- Lustre

Part 2

Benchmarking

MPI-IO

Future Technologies

Outline of the Day

Part 1

Introduction

Storage System Models

Parallel File Systems

- GPFS
- PVFS
- Panasas
- Lustre

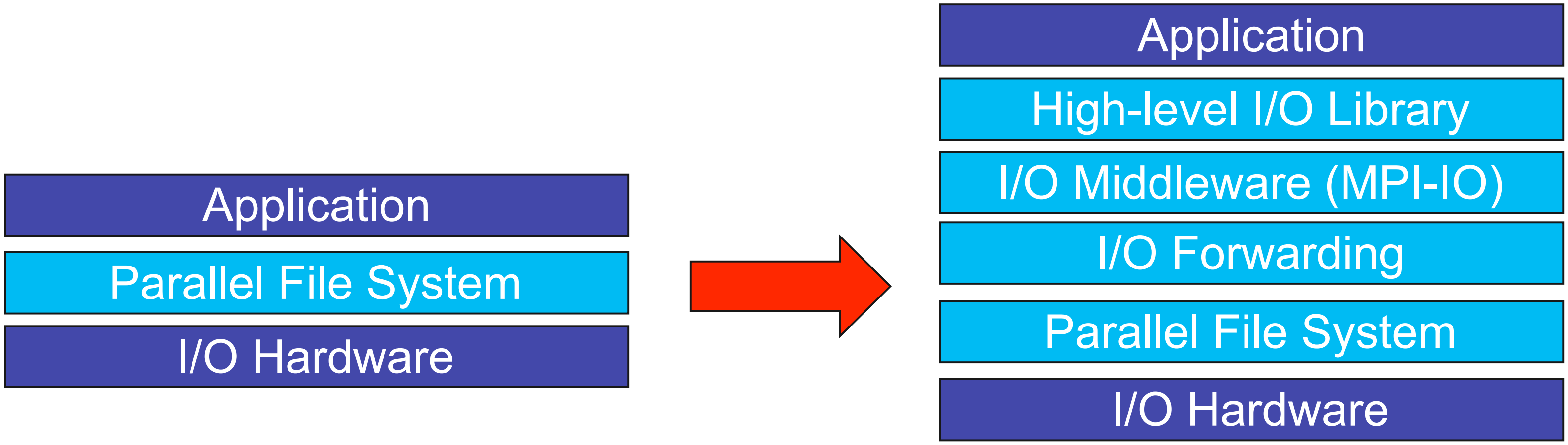
Part 2

Benchmarking

MPI-IO

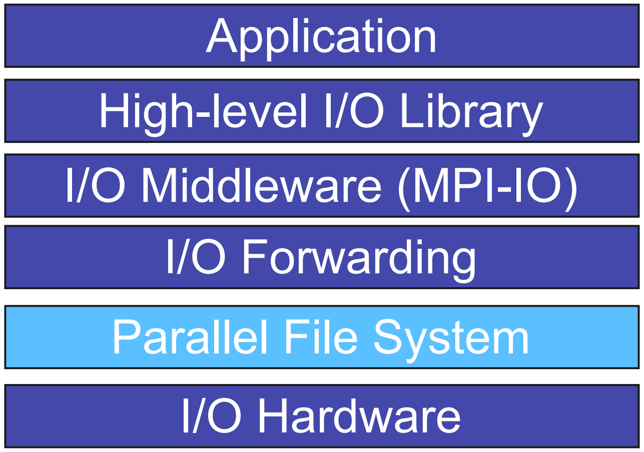
Future Technologies

I/O for Computational Science



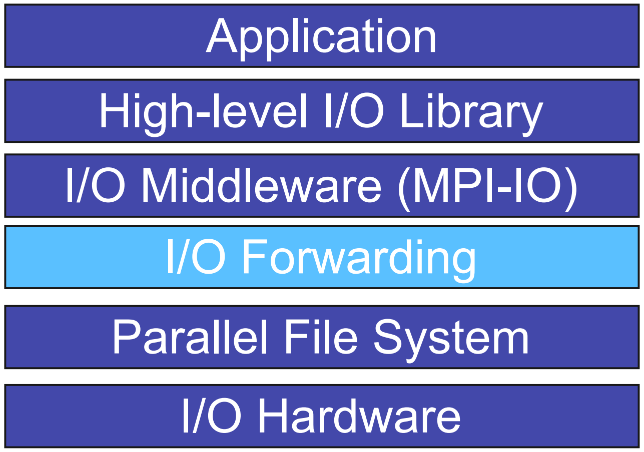
- Parallel file systems support middleware and applications
  - Understanding this context helps motivate some of their features
- Goals of the storage system as a whole:
  - Scalability
  - Parallelism (high bandwidth)
  - Usability

Parallel File System


Application
High-level I/O Library
I/O Middleware (MPI-IO)
I/O Forwarding
Parallel File System
I/O Hardware

- Manage storage hardware
  - Present unified view
  - Stripe files for performance
  - Handle failures
- In the context of the I/O software stack
  - Focus on concurrent, independent access
  - Publish an interface that middleware can use effectively
  - Knowledge of collective I/O usually very limited

I/O Forwarding



Application

High-level I/O Library

I/O Middleware (MPI-IO)

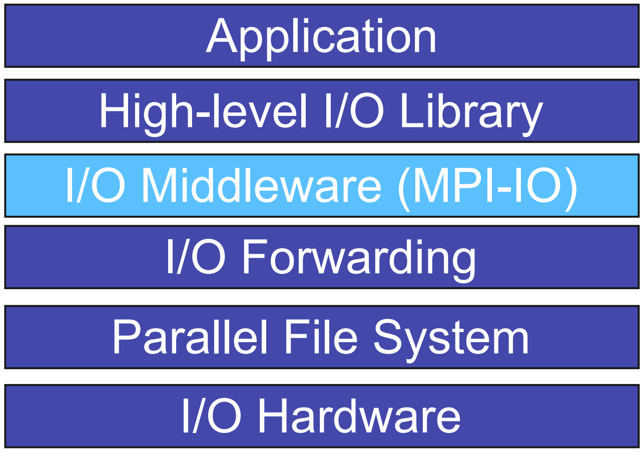
I/O Forwarding

Parallel File System

I/O Hardware

- Present in some of the largest systems
- Newest layer in the stack
- 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



Application

High-level I/O Library

I/O Middleware (MPI-IO)

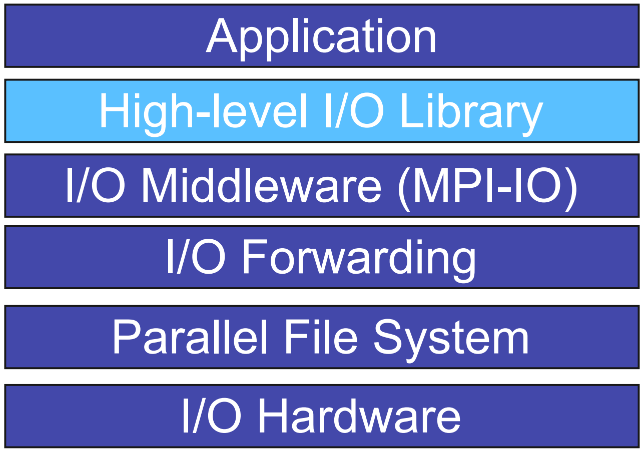
I/O Forwarding

Parallel File System

I/O Hardware

- Match the programming model
- Facilitate concurrent access
  - Collective I/O
  - Atomicity rules
- Expose a generic interface
  - Good building block for high-level libraries
- Efficiently map middleware operations into PFS ones

High Level Libraries


Application
High-level I/O Library
I/O Middleware (MPI-IO)
I/O Forwarding
Parallel File System
I/O Hardware

- Match storage abstraction to domain
  - Multidimensional datasets
  - Typed variables
  - Attributes
- Provide self-describing, structured files
- Map to middleware interface
- Implement higher-level optimizations
  - Caching attributes of variables
  - Chunking of datasets

Outline of the Day

Part 1

Introduction

Storage System Models

Parallel File Systems

- GPFS
- PVFS
- Panasas
- Lustre

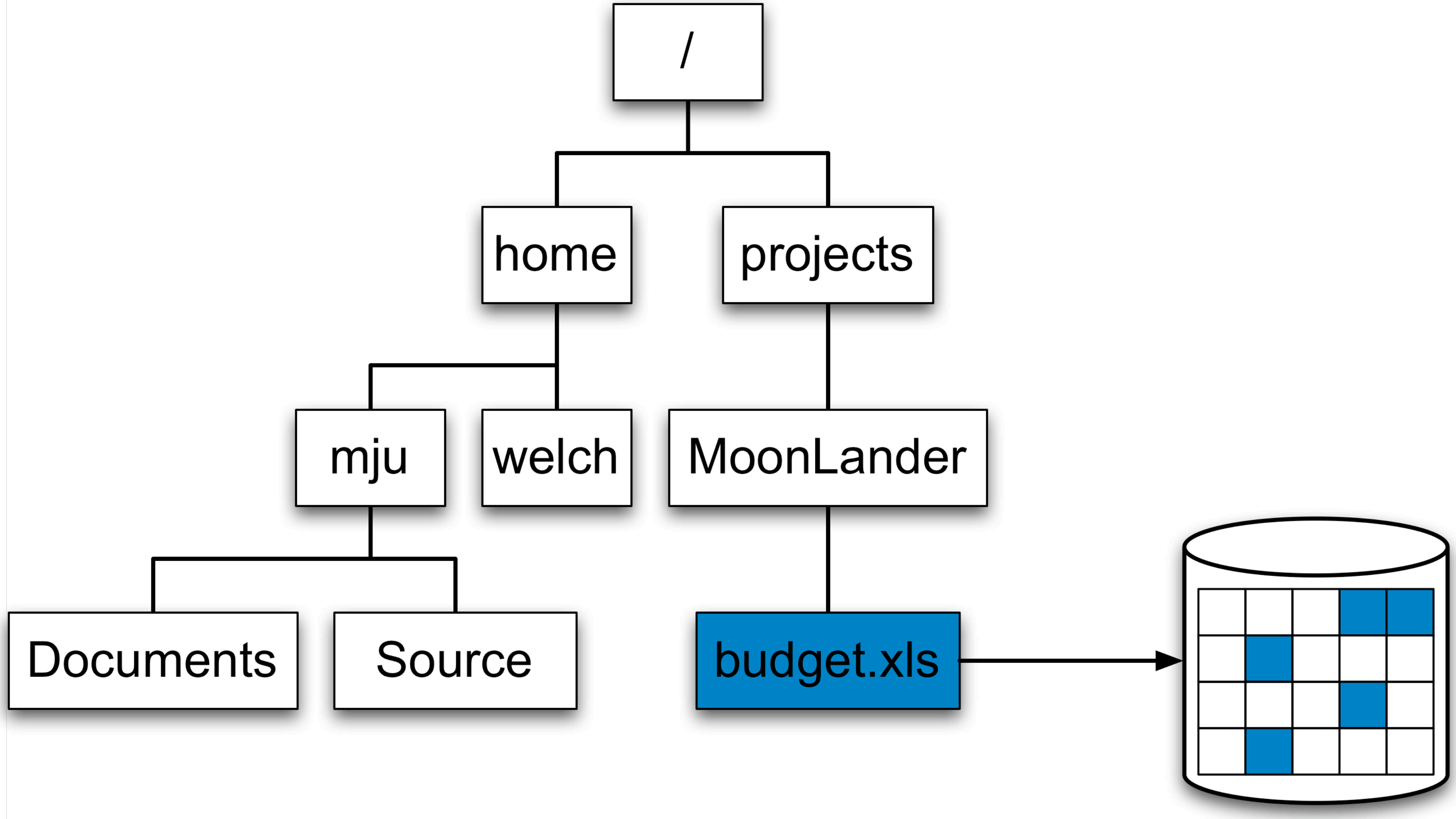
Part 2

Benchmarking

MPI-IO

Future Technologies

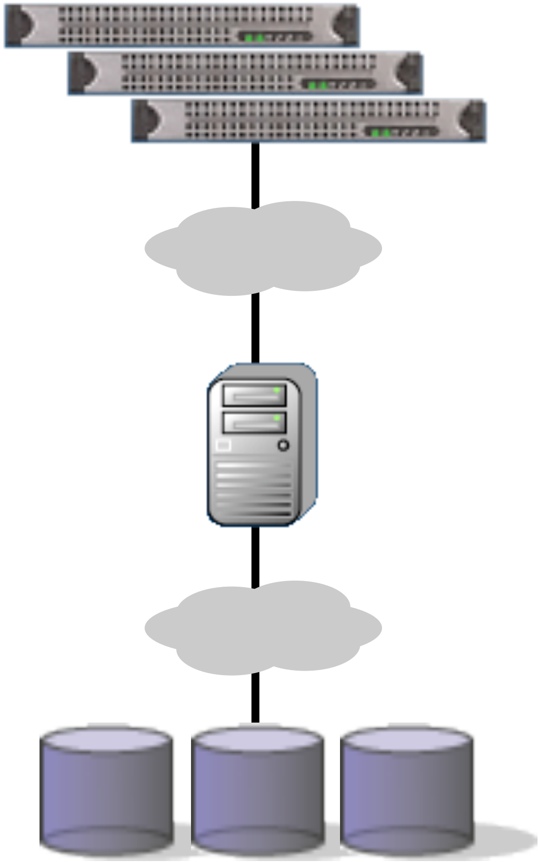
Role of the File System



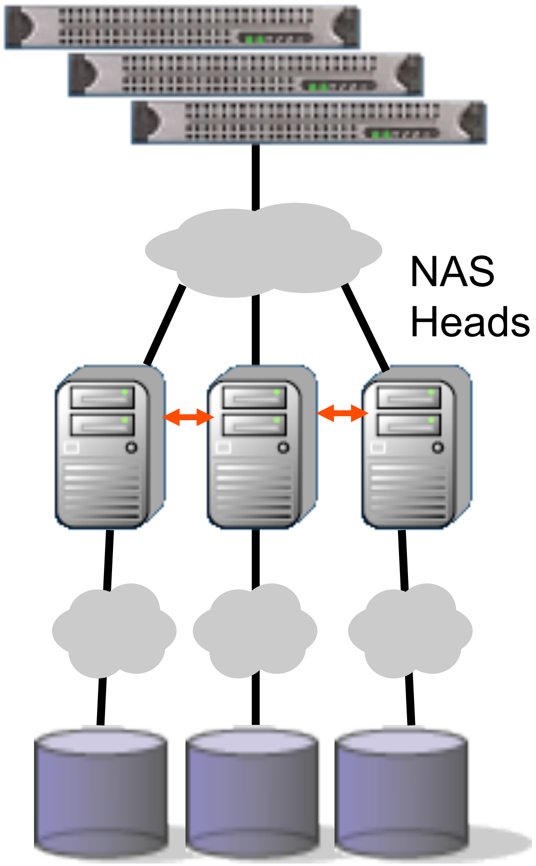
Parallel File System Design Issues

- Same problems as local filesystem
  - Block allocation
  - Metadata management
  - Data reliability and error correction
- Additional requirements
  - Cache coherency
  - High availability
  - Scalable capacity & performance

Network Attached Storage (NAS)

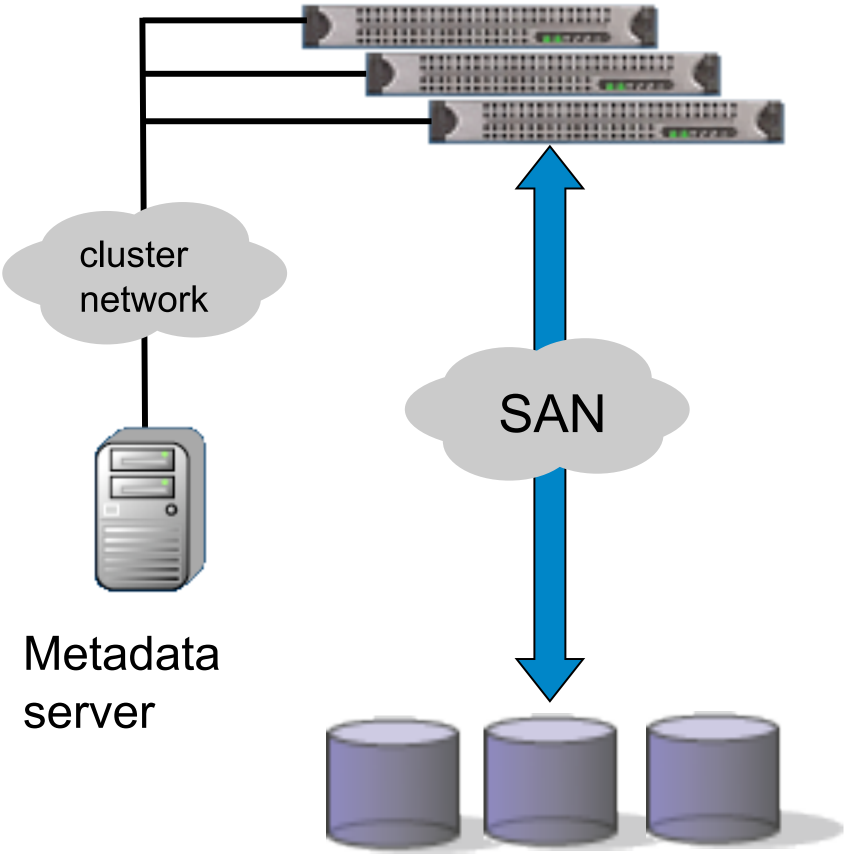


Clustered NAS

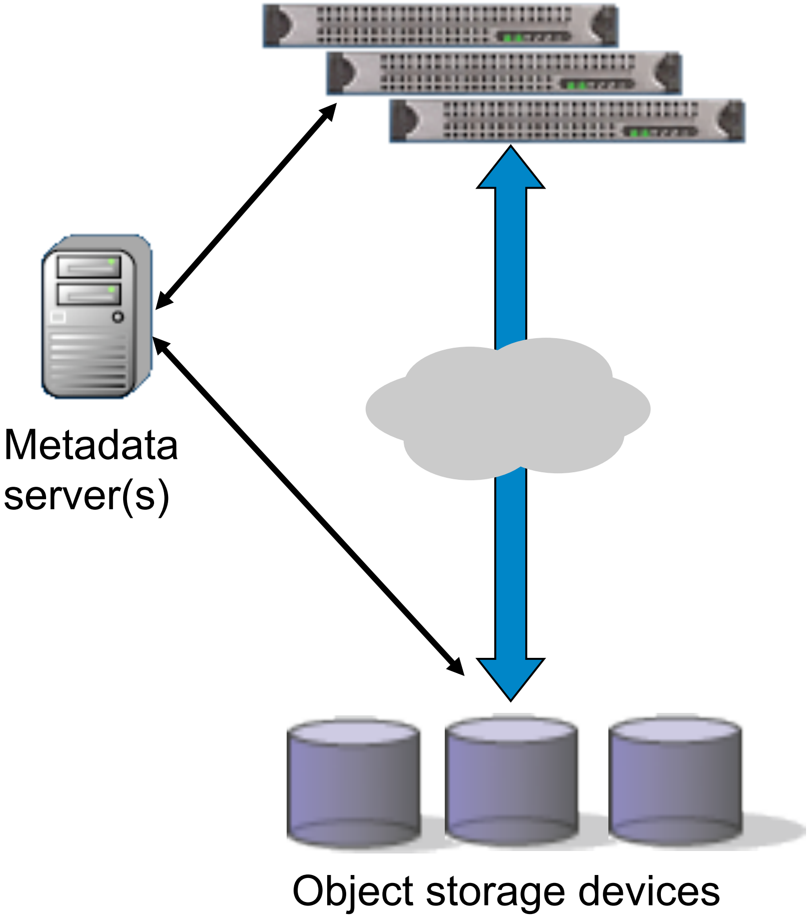


NAS
Heads

SAN Shared Disk File Systems



Object-based Storage Clusters

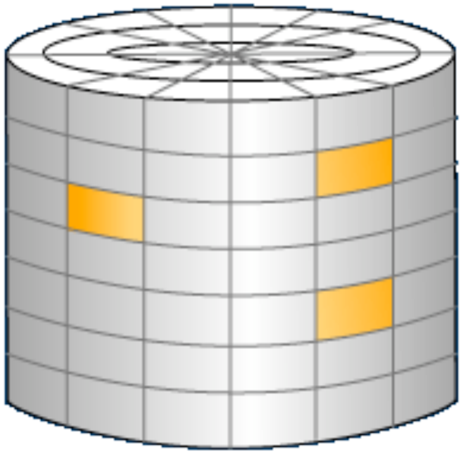


Metadata
server(s)

Object storage devices

Object Storage Architecture

Block Based Disk



Operations

Read block
Write block

Addressing

Block range

Allocation

External

Operations

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

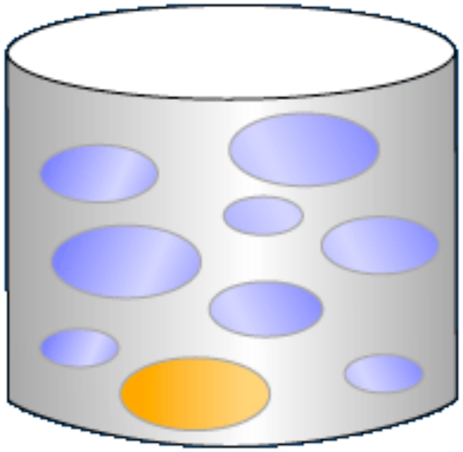
Addressing

[object, byte range]

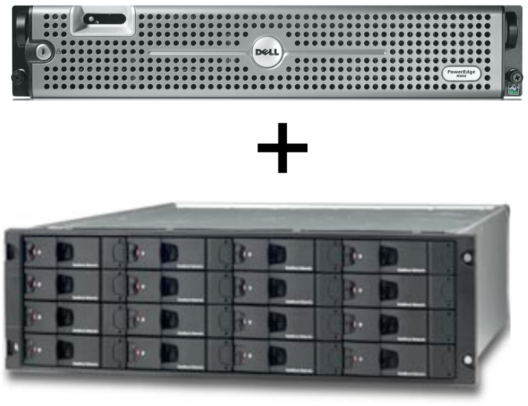
Allocation

Internal

Object Based Disk



What’s in an OSD?

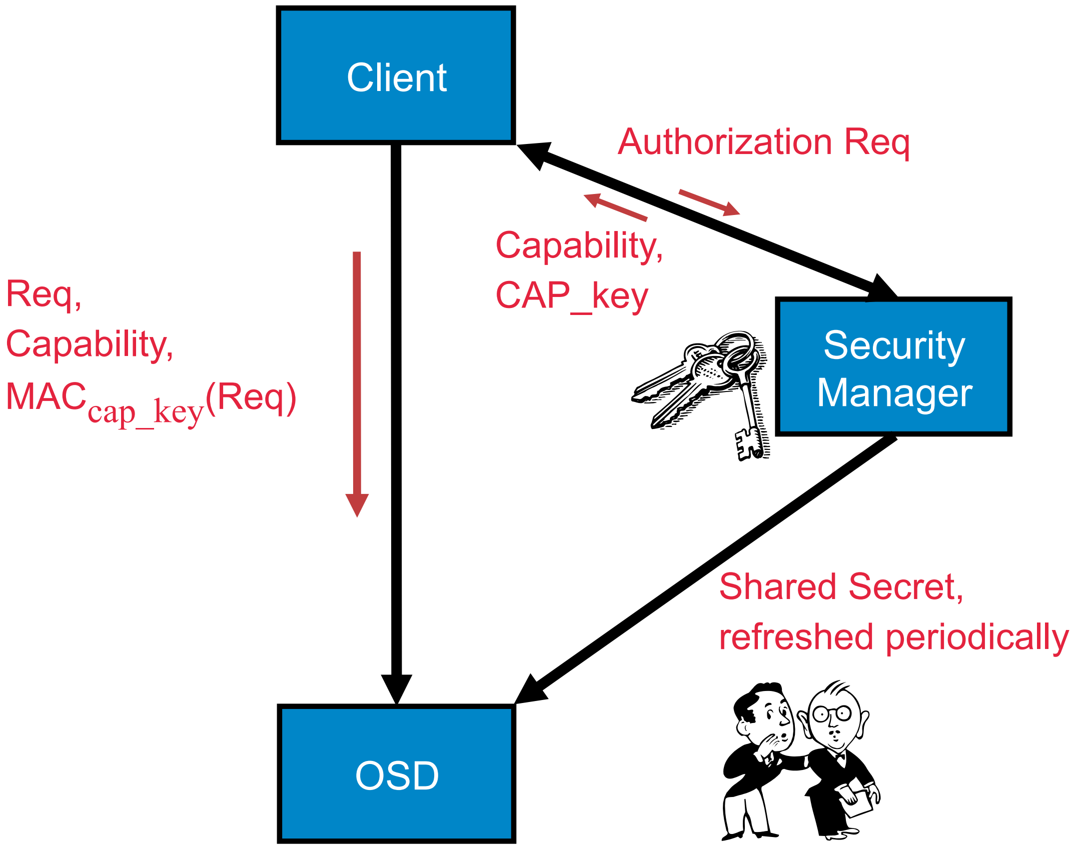


Lustre OSS
PVFS storage node

Panasas
StorageBlade

Seagate
prototype

SCSI T10 OSD Security Model



Client

Authorization Req

Capability,
CAP_key

Security Manager

Req,
Capability,
MAC_{cap_key}(Req)

Shared Secret,
refreshed periodically

OSD

Strengths of Object Storage

- Scalable block allocation
- Data relationships exposed to OSD
- Extensible metadata
- Fine-grained security
- Command set friendly to embedded devices

Outline of the Day

Part 1

- Introduction
- Storage System Models
- **Parallel File Systems**
  - **GPFS**
  - **PVFS**
  - **Panasas**
  - **Lustre**

Part 2

- Benchmarking
- MPI-IO
- Future Technologies

Production Parallel File Systems

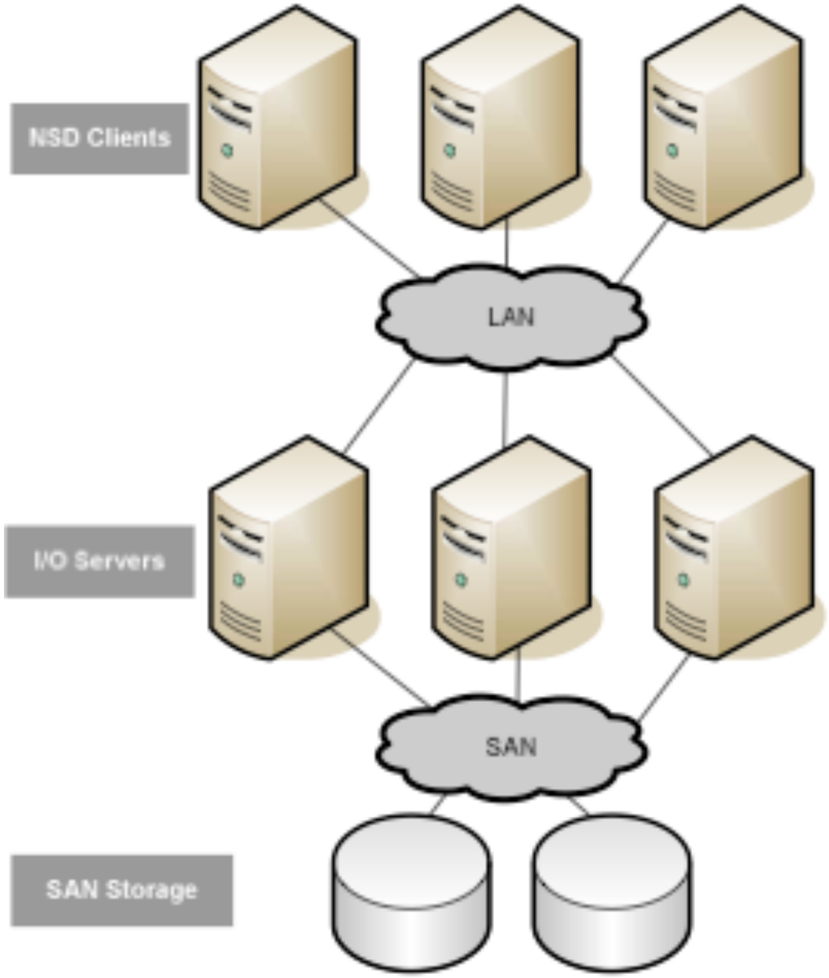
- All four systems scale to support the very largest compute clusters
  - LLNL Purple, LANL RoadRunner, Sandia Red Storm, etc.
- All but GPFS delegate block management to “object-like” data servers or OSDs
- Approaches to metadata vary
- Approaches to fault tolerance vary
- Emphasis on features & “turn-key”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

GPFS PVFS

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 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 file across multiple I/O servers (RAID-0)
- Large block size (1-4 MB) typically used
  - Increases transfer size per I/O server
  - Match block size to RAID stripe width
  - 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 updates inode
  - Other clients send inode updates to metanode

GPFS: Caching

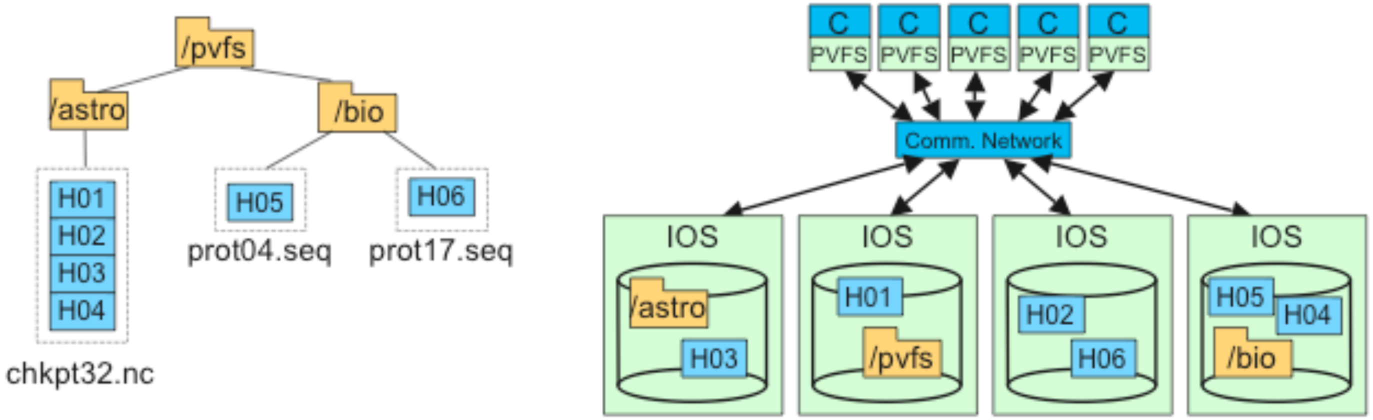
- Clients cache reads and writes
- Strong coherency, based on distributed locking
- Client acquires R/W lock before accessing data
- Optimistic locking algorithm
  - First node accesses 0-1M, locks 0…EOF
  - Second node accesses 8M-9M
    - First node reduces its lock to 0…8191K
    - Second node locks 8192K…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
- 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

PVFS

- Parallel Virtual Filesystem
- Open source
- Linux based
- Community development
  - Led by Argonne National Lab
- Asymmetric architecure (data servers & clients)
- Data servers use object-like API
- Focus on needs of HPC applications
  - Interface optimized for MPI-IO semantics, not POSIX

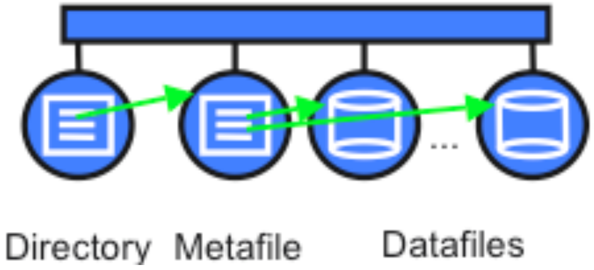


PVFS: Block Allocation

- 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 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

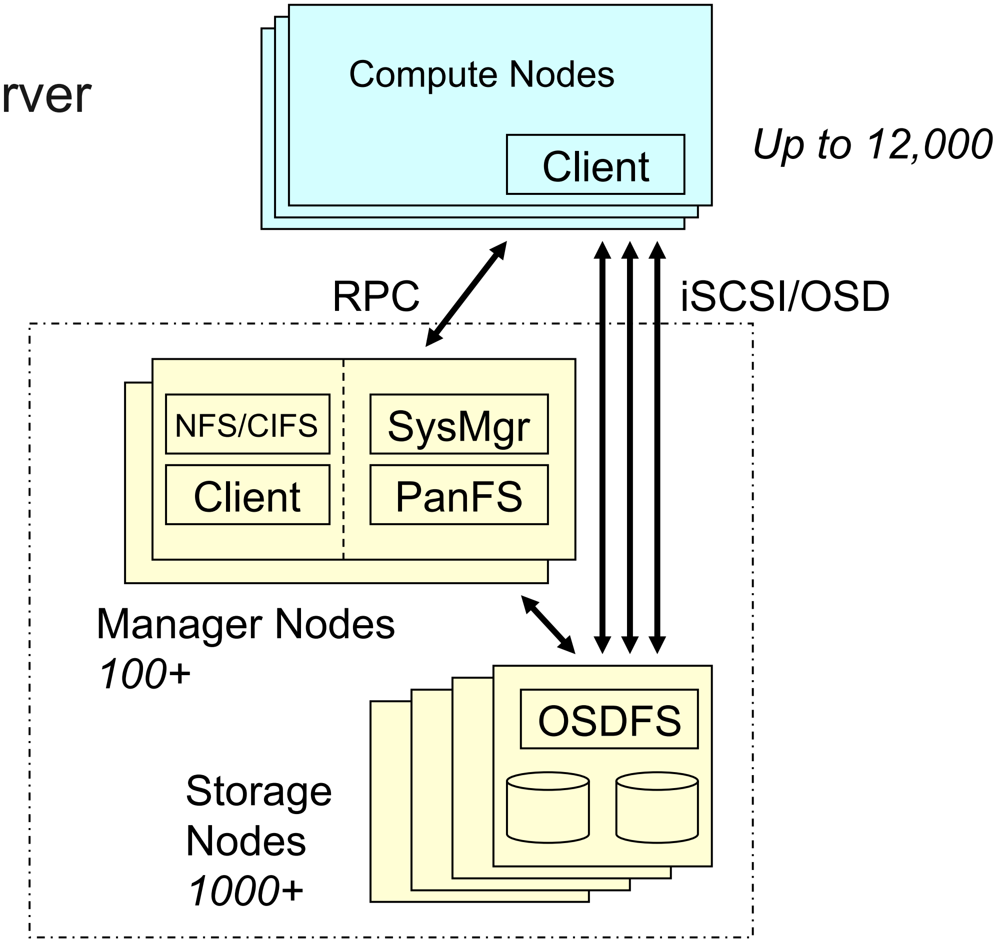
- Client only caches immutable metadata and read-only files
- All other I/O (reads, writes) go through to I/O node
- Strong coherency (writes are immediately visible to other nodes)
- Flows from PVFS2 design choices
  - No locking
  - No cache coherency protocol
- I/O server can cache data & metadata 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
- 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 based on T10 OSD
  - 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

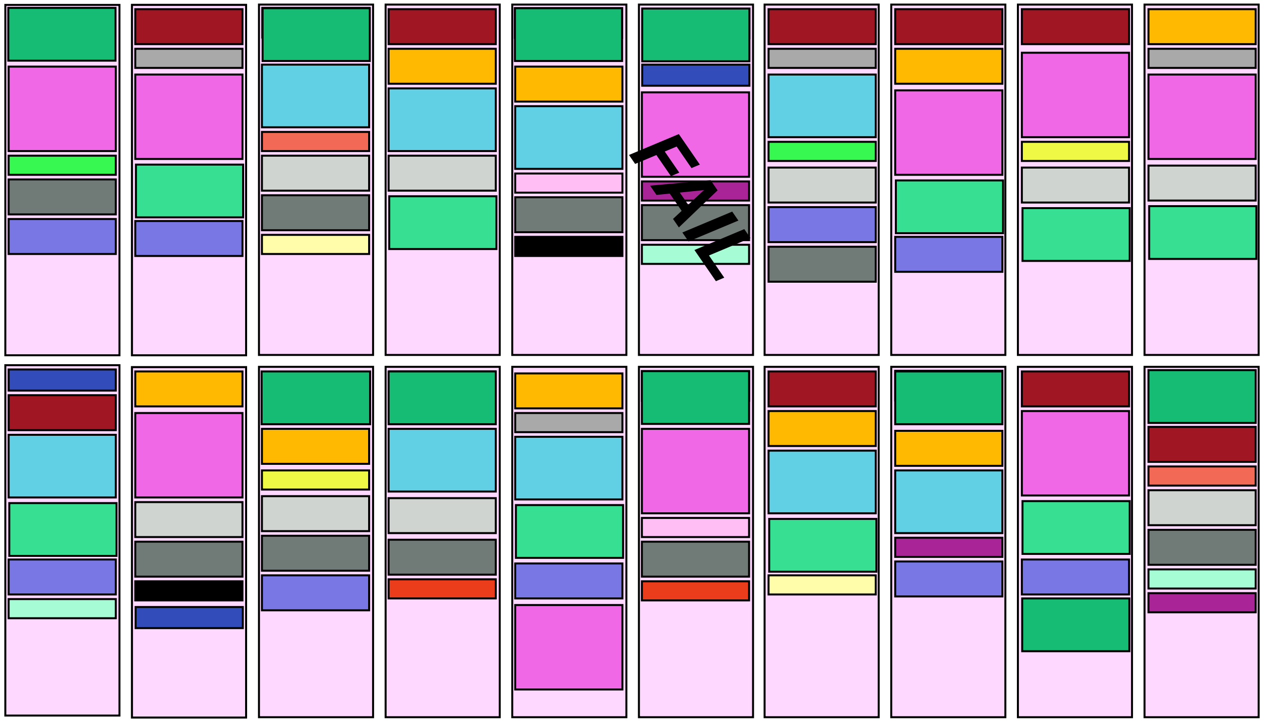
- RAID-1 & RAID-5 across OSDs to handle disk failures
  - Any failure in StorageBlade is handled via rebuild
  - Declustered parity allows scalable rebuild
- “Vertical parity” inside OSD to handle sector errors
- Integrated shelf battery makes all RAM in blades into NVRAM
  - Metadata server journals 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

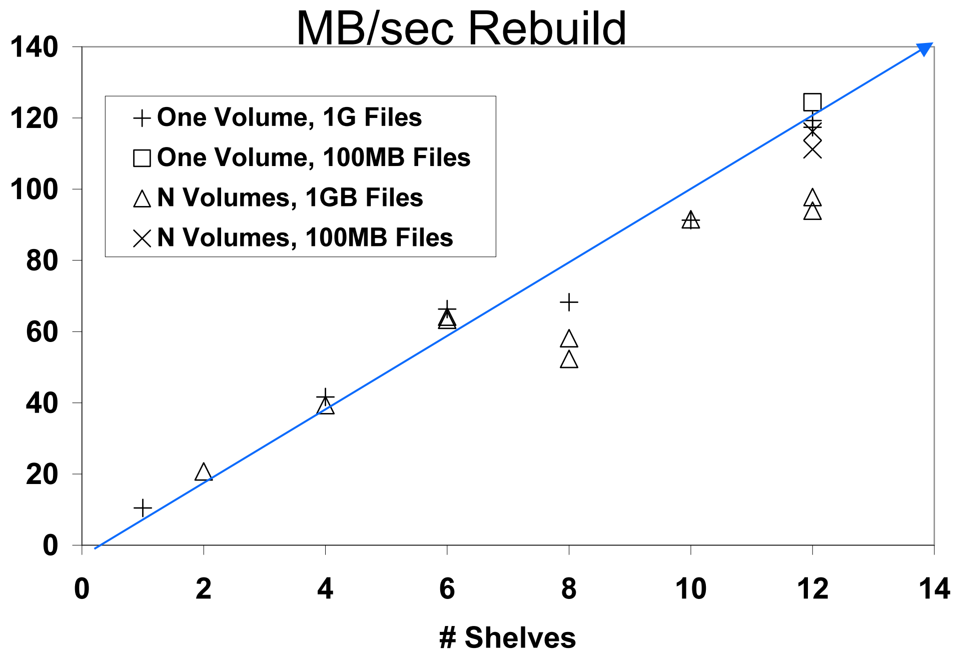
Write a little to each OSD

Scales up in larger Storage Pools

Panasas Scalable Rebuild

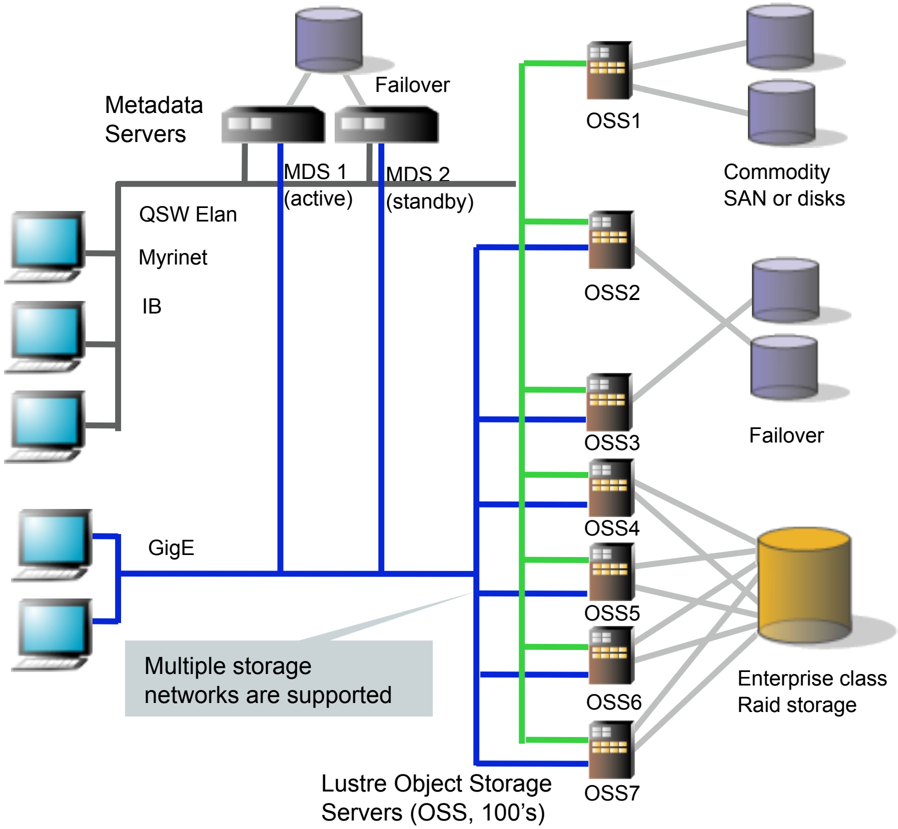
- Two main causes of RAID failures
  1) **2nd drive failure in same RAID set during reconstruction of 1st failed drive**
     - Risk of two failures depends on time-to-repair
  2) Media failure in same RAID set during reconstruction of 1st failed drive

- 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



Multiple storage networks are supported

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
- 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 (Linux-HA)
- 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

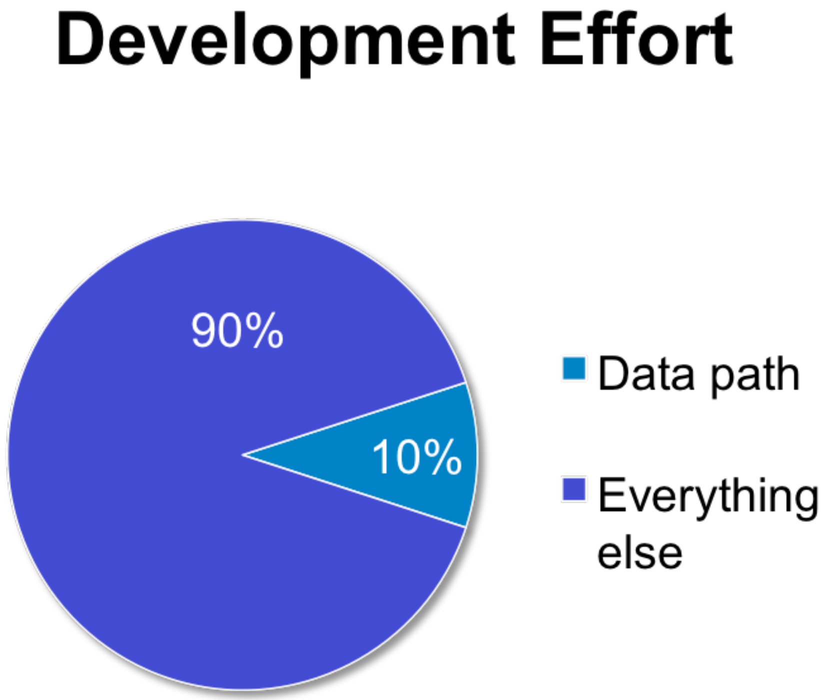
Other File Systems

- GFS (Google)
  - Single metadata server + 100s of chunk servers
  - Specialized semantics (not POSIX)
  - 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

What about…

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



Themes

"A supercomputer is a device for turning compute-bound problems into I/O-bound problems."

- Ken Batcher

- Scalable clusters need scalable storage
- Avoid centralized/single anything
- File/object storage API superior to blocks
- Reliability is important

Outline of the Day

Part 1

Introduction

Storage System Models

Parallel File Systems

- GPFS
- PVFS
- Panasas
- Lustre

Part 2

Benchmarking

MPI-IO

Future Technologies

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

Workloads

- Streaming I/O
  - Single client, one or more streams per client
  - Many clients, file-per-process or shared-file
  - Scaling clients
  - Server throughput, scaling with number of servers
- Random I/O
  - Dependent on caching and drive seek performance
- Metadata
  - Create/Delete workloads
  - File tree walk (scans)
- MPI IO
  - Coordinated opens
  - Shared output files
- Interprocess Communication
  - Producer/consumer files
  - Message drop
  - Atomic record updates
- Small I/O
  - Small whole file operations
  - Small read/write operations

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

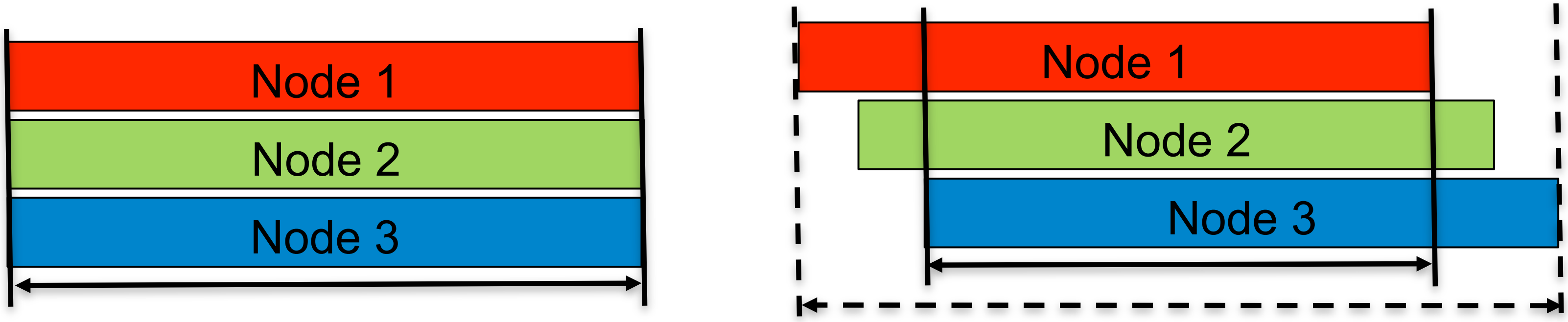
- Selected combination of operations (fractional mix)
  - Operations selected at random or using random model (e.g., Hidden Markov Model)
  - Operations and mix based on traces or sampling real workload
- Can provide better model for application performance
  - However, inherently domain-specific
  - 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
  - FLASH I/O

Benchmarks for HPC

- Unfortunately, there are few synthetic HPC benchmarks that stress I/O
- HPC Challenge
  - 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
  - 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
  - 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
  - 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
  - Sometimes the bottleneck isn’t where you think it is
- 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

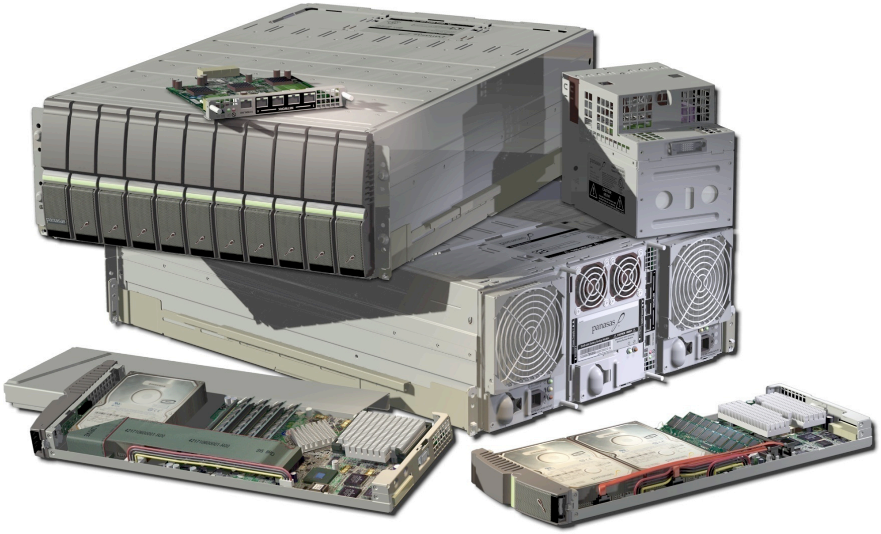
PVFS Test Platform: OSC Opteron Cluster

- 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 Platform: Pittsburgh Lab

- Small test system from our Pittsburgh development lab
- 3 Panasas Shelves, each with
  - 10 SB-1000a-XC StorageBlades
    - (1.5GHz Celeron, 2GB RAM, 2x500GB SATA, 1GE)
  - 1 DB-100a DirectorBlade
    - (1.8GHz 475, 4GB RAM, 1GE)
  - 18-port switch with 10GE uplink
- 48 client nodes
  - 2.8 GHz Xeon, 8GB, 1GE
- GE Backbone
  - 40 GB/s 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

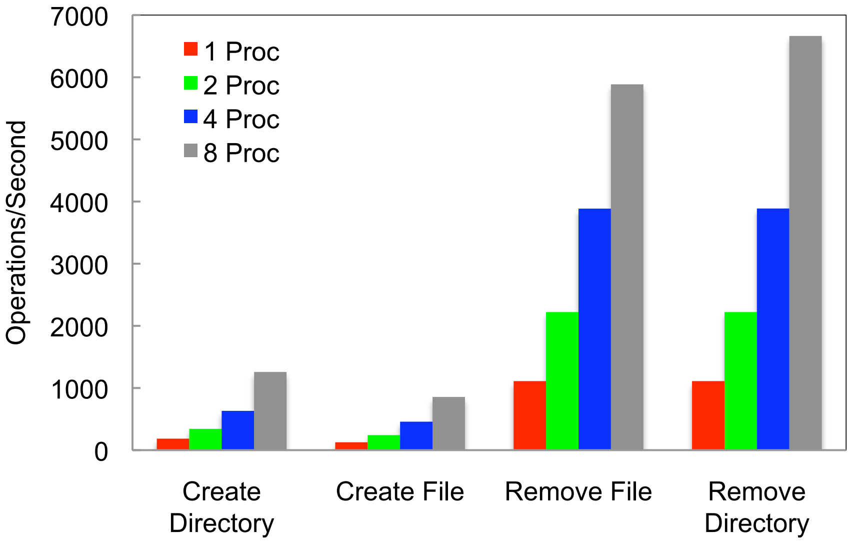
Metadata Performance

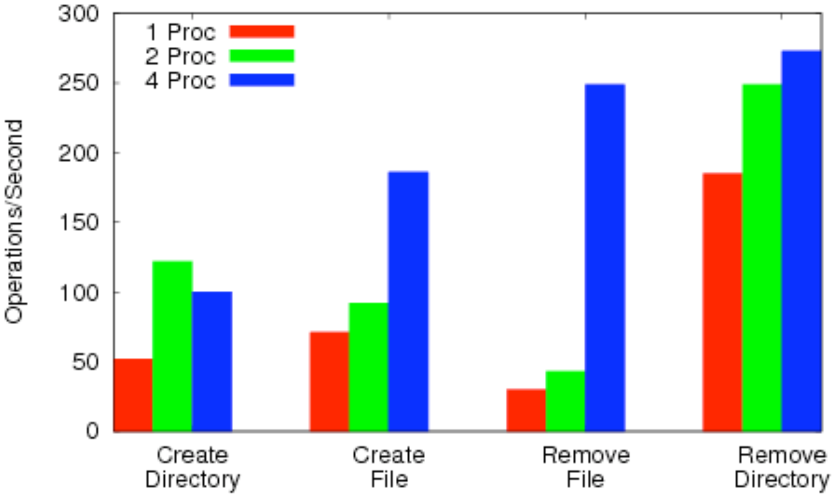
- Storage is more than reading & writing
- Metadata operations change the namespace or file attributes
  - Creating, opening, closing, and removing files
  - Creating, traversing, and removing directories
  - "Stat"ing files (obtaining the attributes of the file, such as permissions and file size)
- 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)

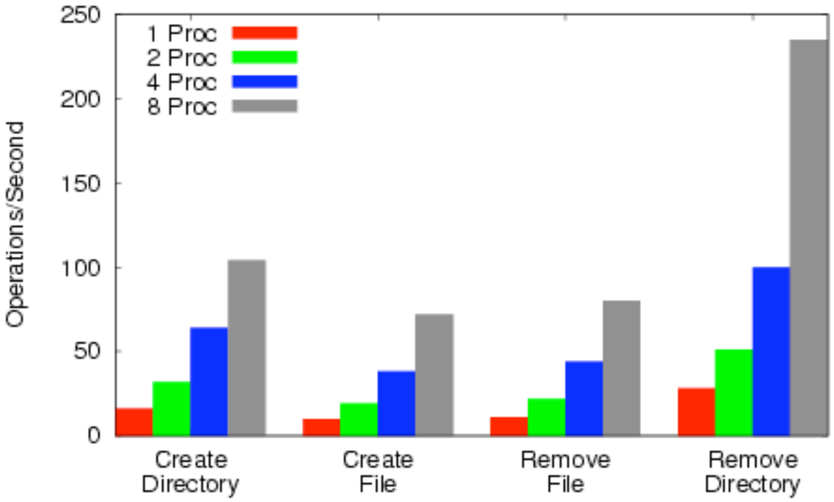
fdtree: Serial Metadata Performance

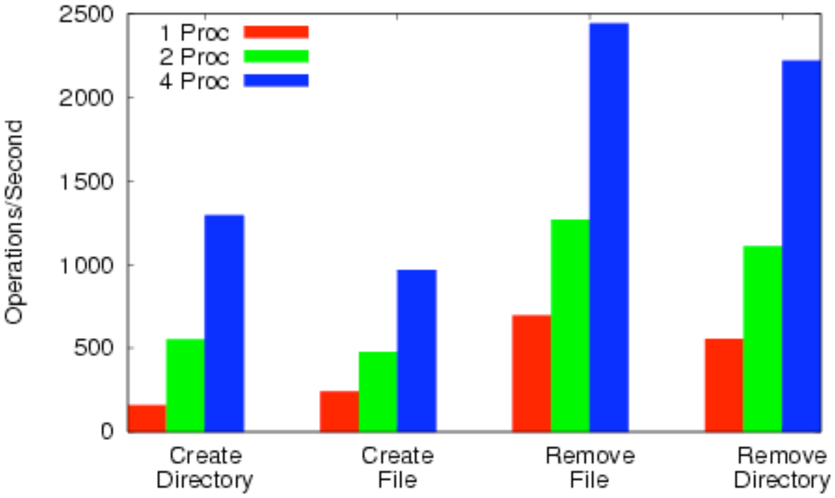
- Written at Lawrence Livermore National Laboratory
- Creates directories and small files in a hierarchical directory structure and then removes them
  - Processes operate independently
- Written as a bash script
  - Uses POSIX interface
  - Similar to an untar operation
- Provides insight into responsiveness to user interaction
- We ran with "-l 3 -d 10 -f 10 -s 10 -o $DIR"
  - Spawned on multiple nodes with LoadLeveler or mpiexec
  - Timing is somewhat coarse grained (processes loosely synced, time measured in whole seconds)

fdtree Results


Panasas fdtree Performance


Lustre fdtree Performance


PVFS fdtree Performance


GPFS fdtree Performance

fdtree Analysis

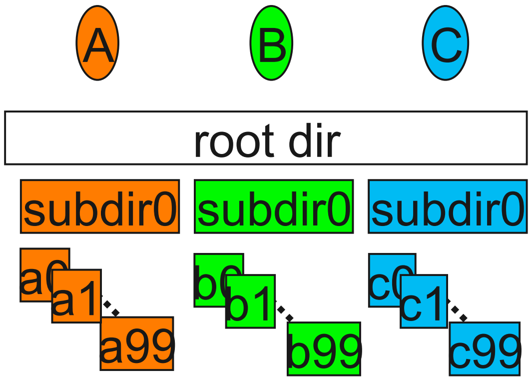
- Lack of caching on clients in PVFS results in slowest performance
- GPFS and Panasas are the fastest of the four and show scalability at these proc counts
  - GPFS faster for creates
  - Panasas faster for deletes
  - GPFS 4-proc directory remove case was probably just out of sync
  - Panasas does deletes in the background
- Question: How many ops/sec do you need on a parallel file system?

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

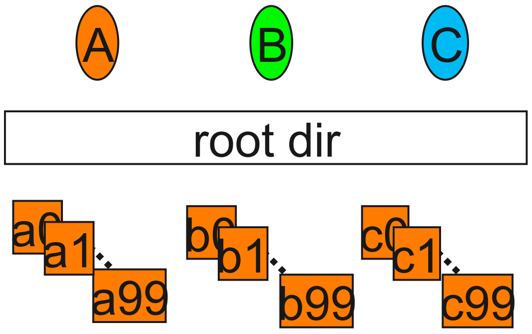
mdtest Variations

Unique Directory



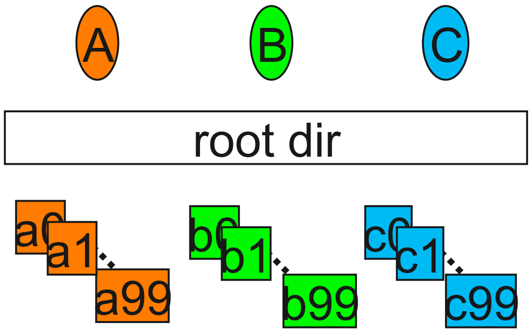
1) Each process (A, B, C) creates own subdir in root directory, then chdirs into it.
2) A, B, and C create, stat, and remove their own files in the unique subdirectories.

Single Process



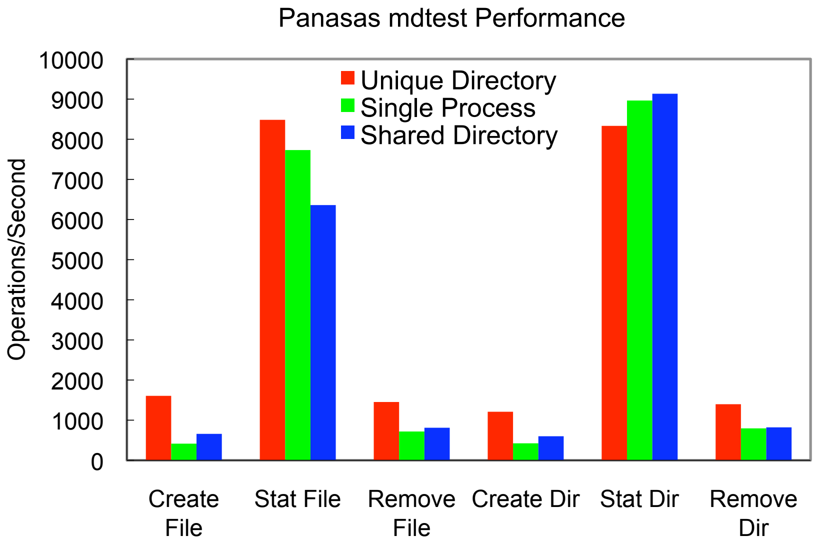
1) Process A creates files for all processes in root directory.
2) Processes A, B, and C open, stat, and close their own files.
3) Process A removes files for all processes.

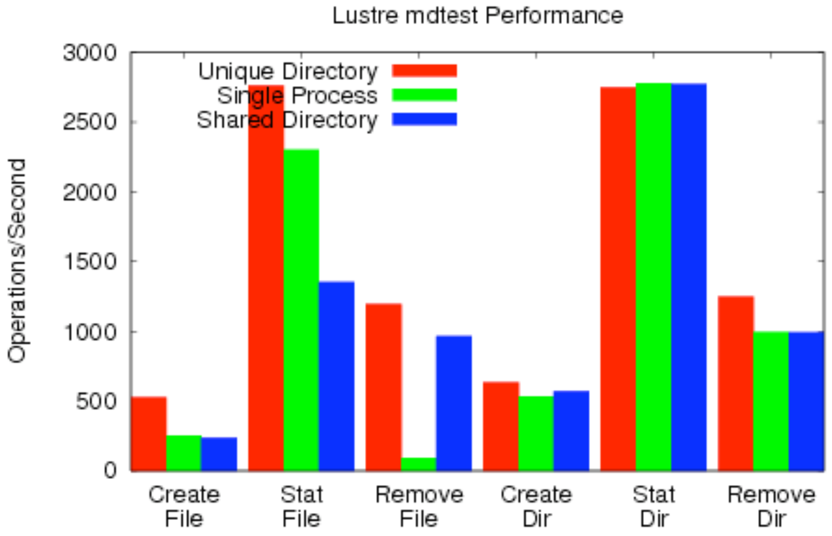
Shared Directory

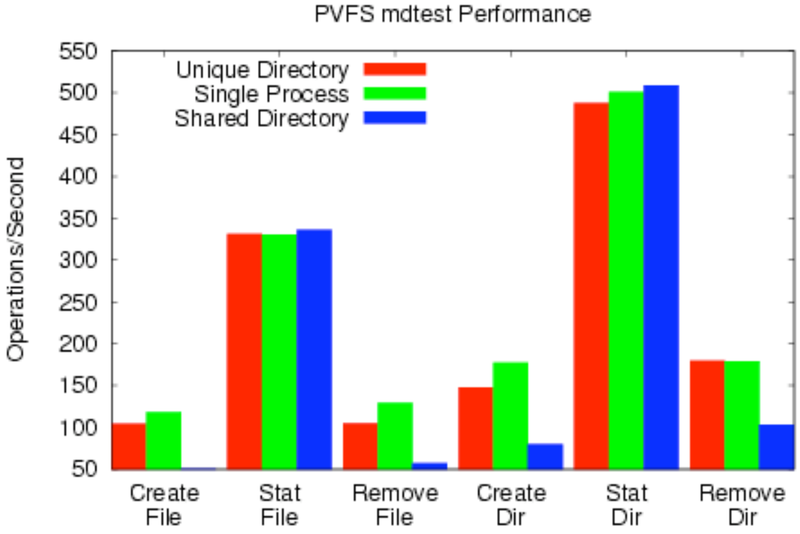


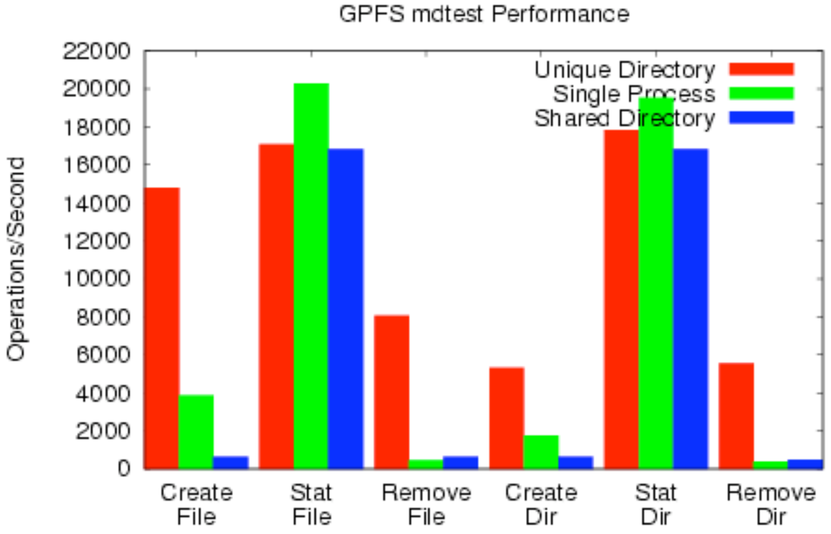
1) Each process (A, B, C) creates, stats, and removes its own files in the root directory.

mdtest Results









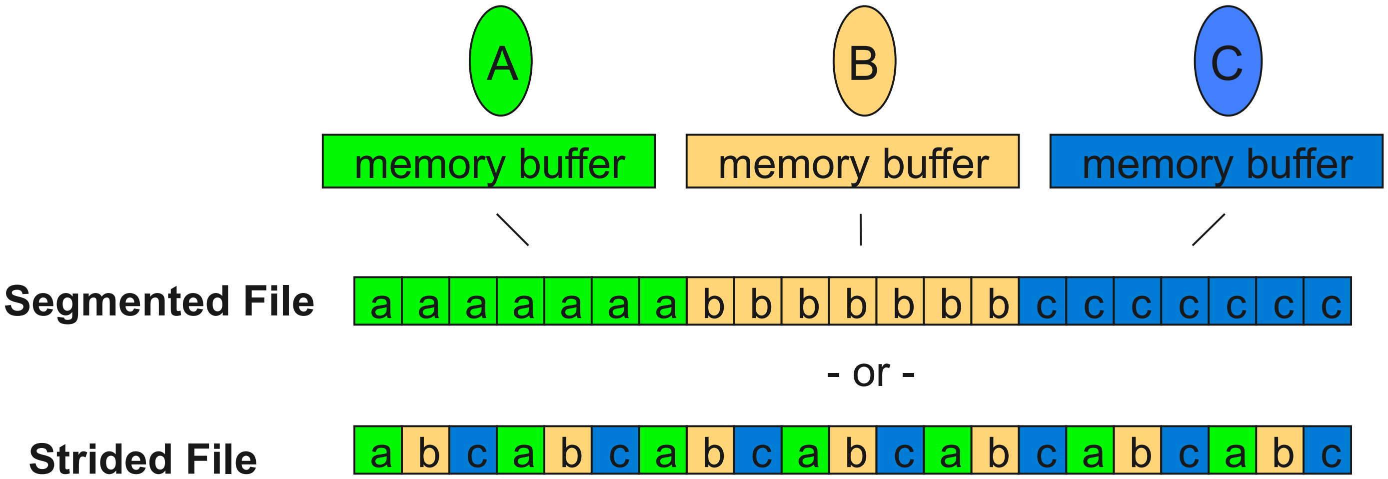
mdtest Analysis

- PVFS
  - No penalty for stat in shared dir
  - Lack of client caching hurts stat throughput
- GPFS
  - Very high cost to operating in the same directory
  - Each client must acquire token & modify dir itself
- Lustre
  - Single MDS and directory lock limit shared dir case
- Panasas
  - Coarse-grained metadata clustering not active, since all procs share common root
  - Directory lock on metadata server limits parallelism

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



Segmented File

a a a a a a a b b b b b b b c c c c c c c

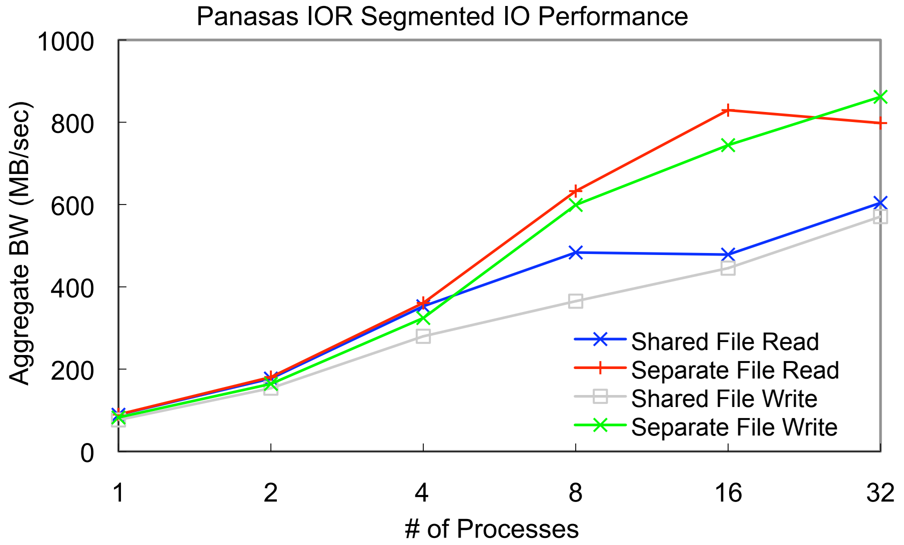
- or -

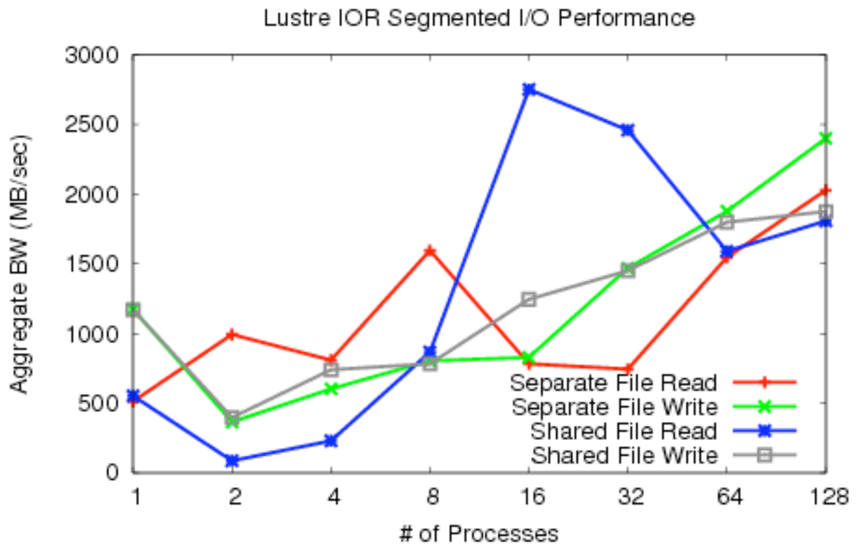
Strided File

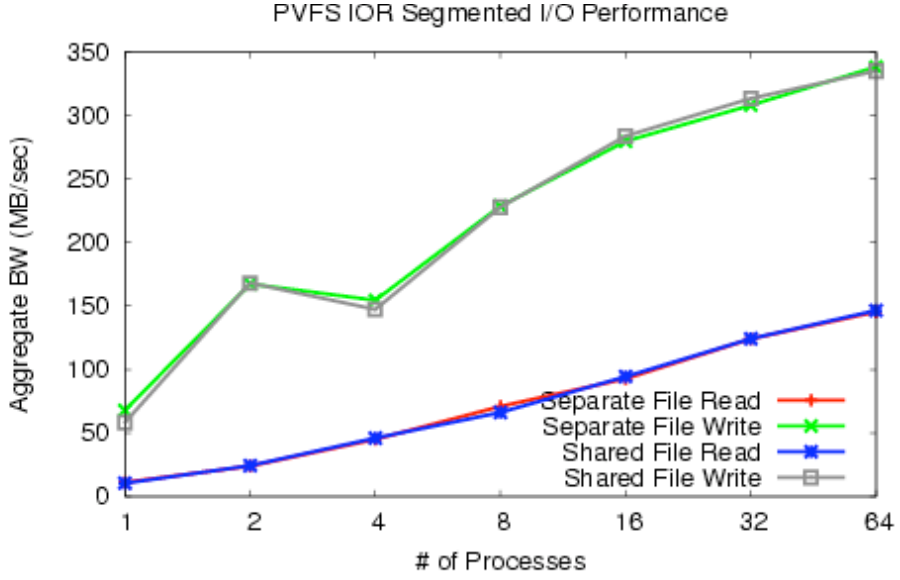
a b c a b c a b c a b c a b c a b c a b c

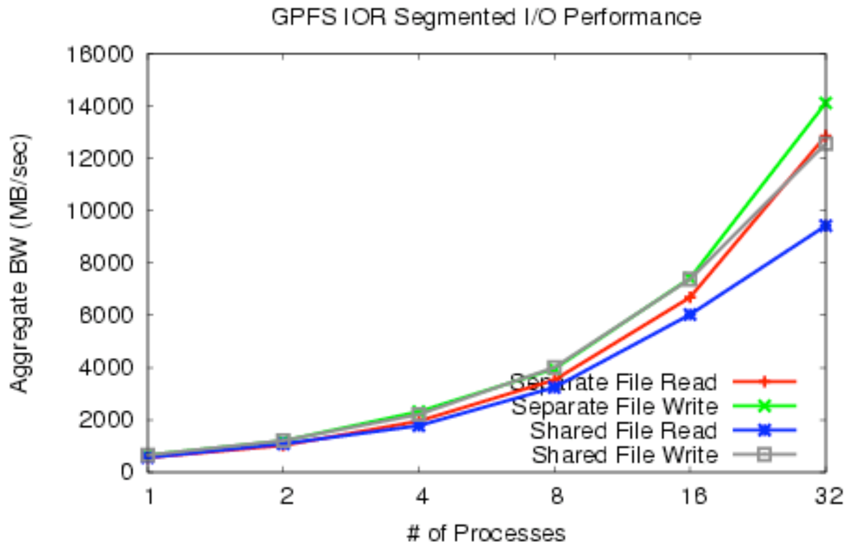
- Primary distinction between the two major shared-file patterns is whether each task’s data is contiguous or noncontiguous
- For the segmented pattern, each task stores its blocks of data in a contiguous region in the file
- With the strided access pattern, each task’s data blocks are spread out through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

IOR POSIX Segmented Results


Panasas IOR Segmented IO Performance


Lustre IOR Segmented I/O Performance


PVFS IOR Segmented I/O Performance


GPFS IOR Segmented I/O Performance

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

Outline of the Day

Part 1

Introduction

Storage System Models

Parallel File Systems

- GPFS
- PVFS
- Panasas
- Lustre

Part 2

Benchmarking

MPI-IO

Future Technologies

What’s wrong with POSIX?

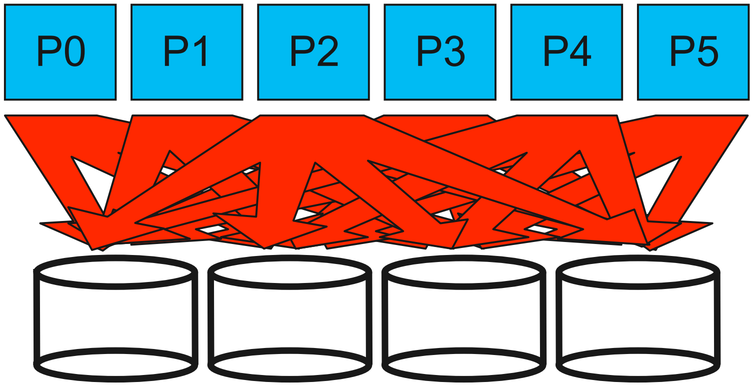
- It’s a useful, ubiquitous interface for basic I/O
- It lacks constructs useful for parallel I/O
  - Cluster application is really one program running on N nodes, but looks like N programs to the filesystem
  - No support for noncontiguous I/O
  - No hinting/prefetching
- Its rules hurt performance for parallel apps
  - Atomic writes, read-after-write consistency
  - Attribute freshness

- POSIX should not be used (directly) in parallel applications that want good performance
  - But developers use it anyway

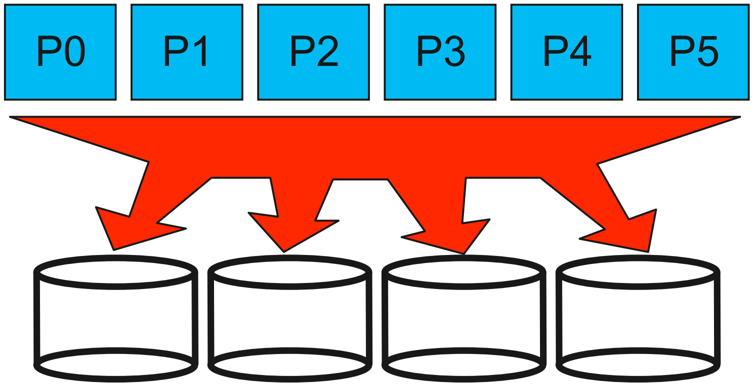
MPI-IO

- 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



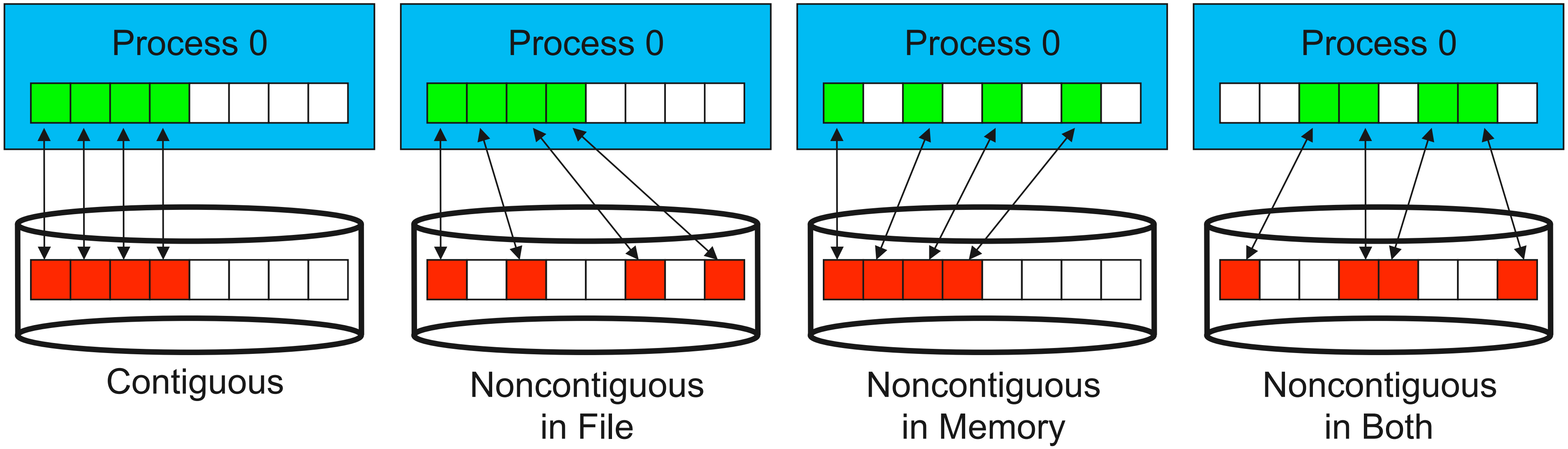
Independent I/O



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

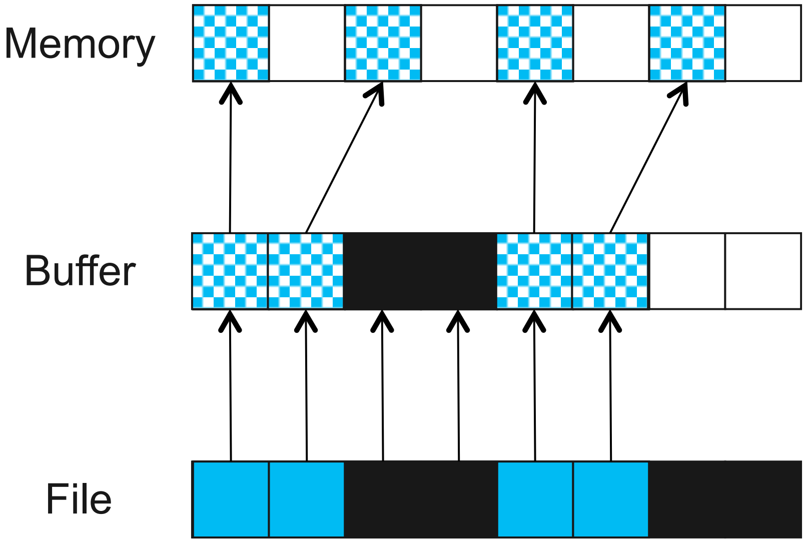
Nonblocking and Asynchronous I/O

- Blocking/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

Under the Covers of MPI-IO

- MPI-IO implementation gets a lot of information
  - 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

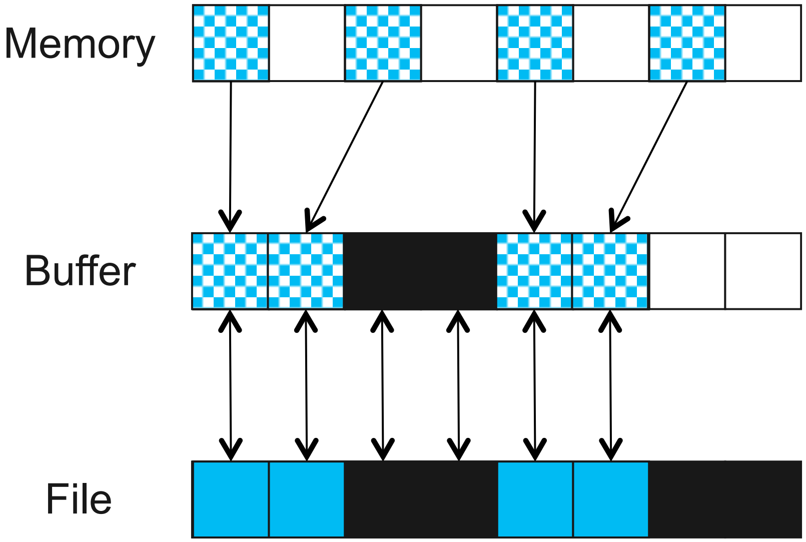
Noncontiguous I/O: Data Sieving



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

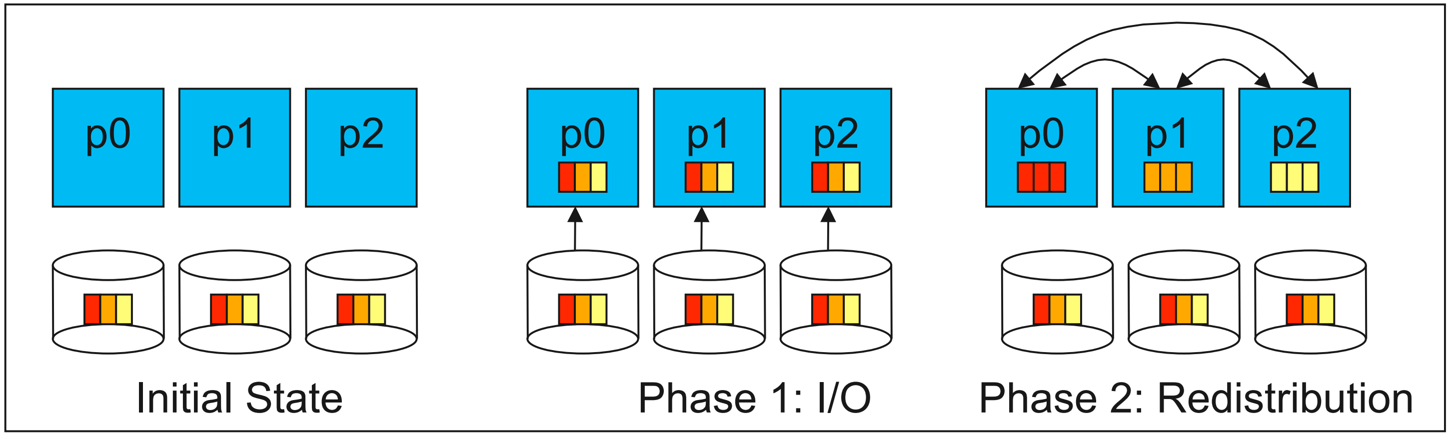
Data Sieving Write Operations



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

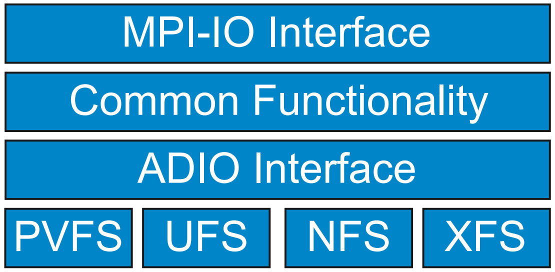
Collective I/O and Two-Phase I/O



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

MPI-IO Implementations


MPI-IO Interface

Common Functionality

ADIO Interface

PVFS | UFS | NFS | XFS

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!

Outline of the Day

Part 1

- Introduction
- Storage System Models
- Parallel File Systems
  - GPFS
  - PVFS
  - Panasas
  - Lustre

Part 2

- Benchmarking
- MPI-IO
- **Future 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

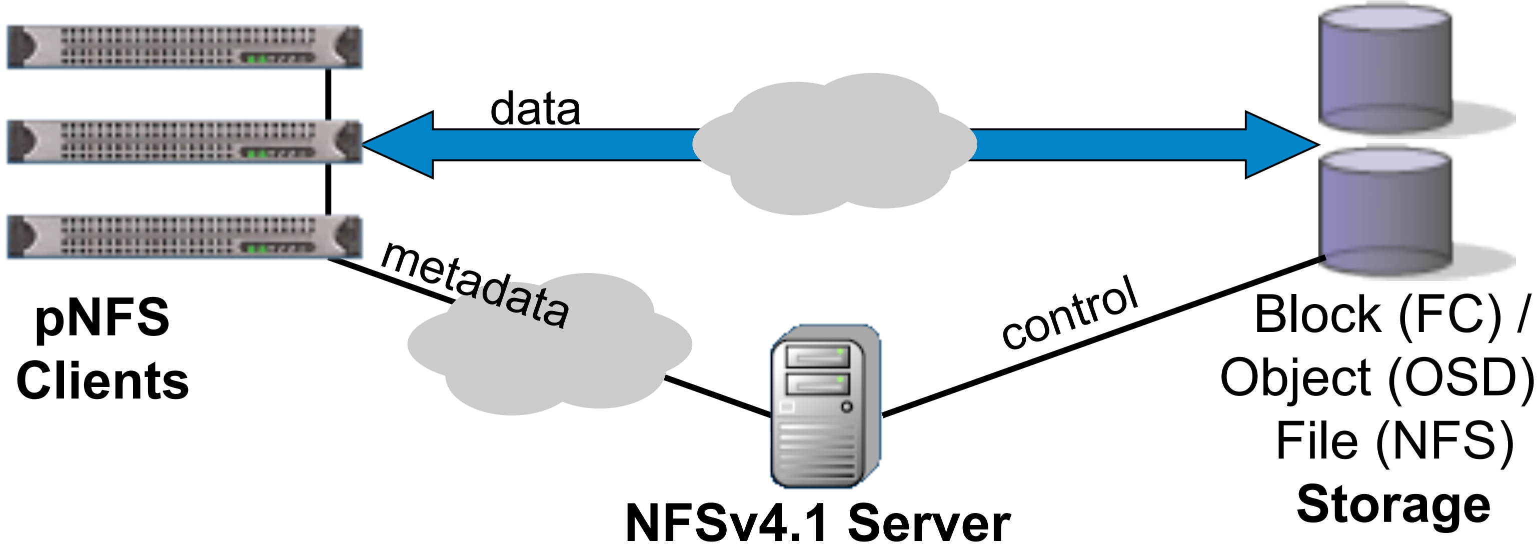
- NFS created in ’80s to share data among engineering workstations
- NFSv3 widely deployed
- NFSv4 several years in the making, lots of new stuff
  - Integrated Kerberos (or PKI) user authentication
  - Integrated File Locking and Open Delegations (stateful server!)
  - ACLs (hybrid of Windows and POSIX 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), …
  - 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
- *Many others*

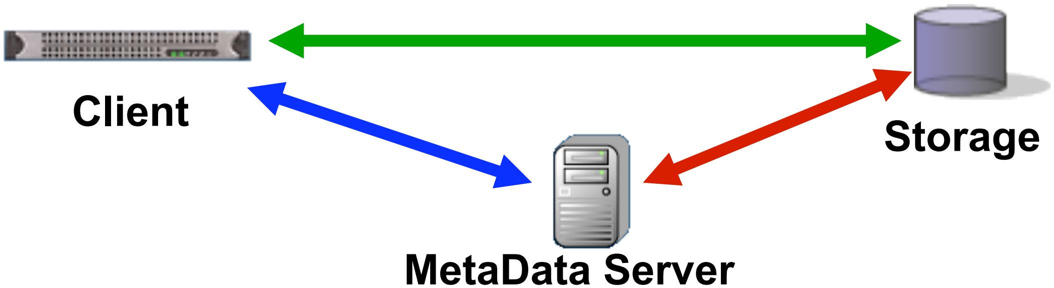
pNFS: Standard Storage Clusters

- 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



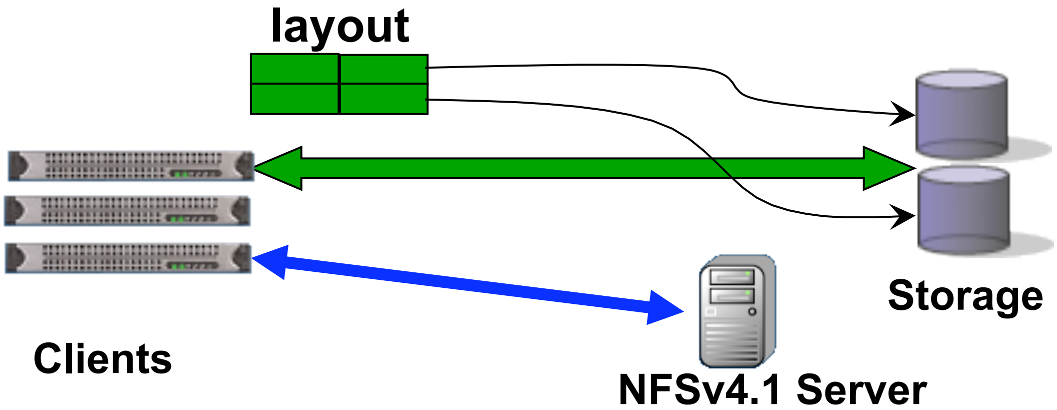
The pNFS Standard

- 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



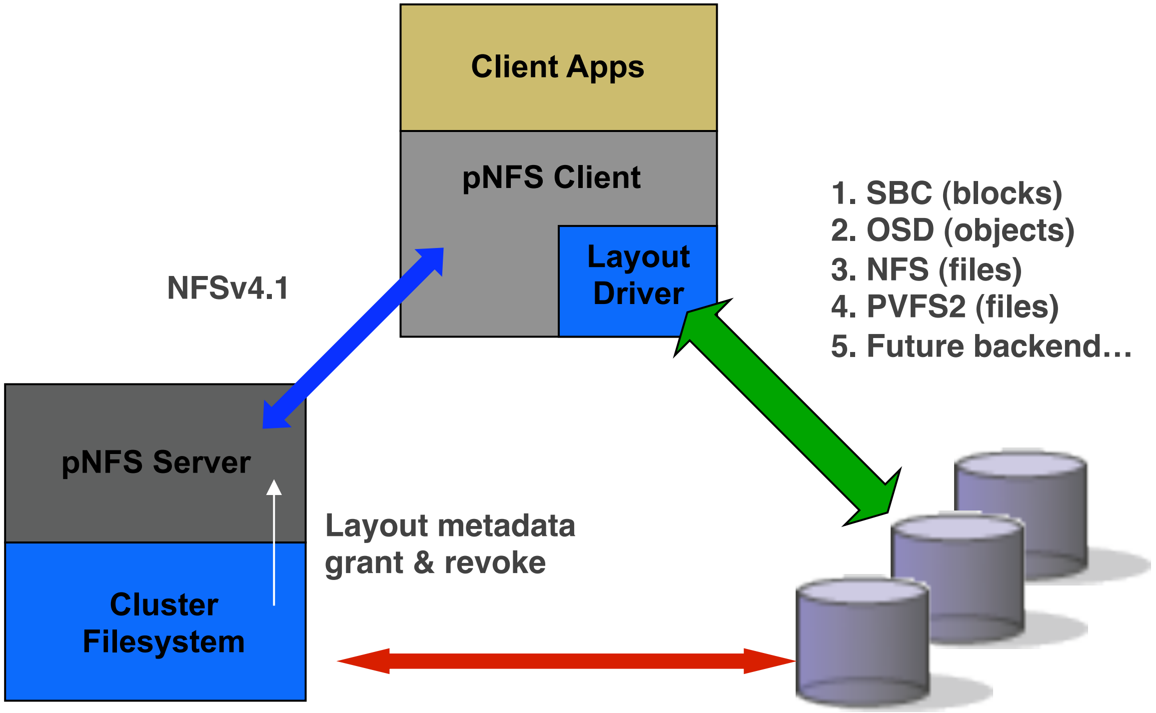
pNFS Layouts

- 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



1. SBC (blocks)
2. OSD (objects)
3. NFS (files)
4. PVFS2 (files)
5. Future backend…

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 bottleneck
  - 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
  - PVFS
  - Your project… (e.g., dcache.org)

Key pNFS Participants

panasas

IBM

EMC²
where information lives

PARALLEL DATA LABORATORY
CARNEGIE MELLON UNIVERSITY

NetApp

Sun
microsystems

center for
information
technology
integration

- Univ. of Michigan/CITI (Files over PVFS and NFSv4)
- NetApp (Files over NFSv4)
- IBM (Files, based GPFS)
- EMC (Blocks, based on MPFS/HighRoad)
- Sun (Files over NFSv4, Objects based on OSDv1)
- Panasas (Objects based on Panasas OSDs)
- Carnegie Mellon (performance and correctness testing)

Current Status

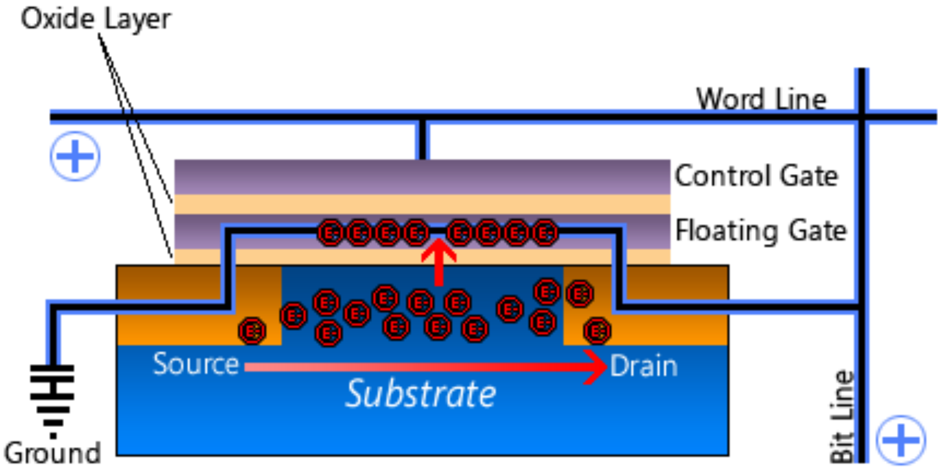
- IETF NFSv4.1 I-D accepted as Proposed Standard by IESG (yay!)
  - Expect RFC number “any day”
- Reference open source client done by CITI
  - CITI owns NFSv4 Linux client and server
- Development progress since FAST08
  - Forward port to closely track HOL Linux kernel tree
  - Patch set preparation for review by Linux maintainers
  - Lots of stabilization
- Prototype interoperability began in 2006
  - San Jose Connect-a-thon Spring ’06, ’07, ’08, ‘09
  - Ann Arbor NFS Bake-a-thon September ’06
  - Austin NFS Bake-a-thon June ’07, October ‘08
- Availability
  - kernel.org adoption by the end of 2009
  - Production releases 2010

The problem with rotating media

- Areal density increases by 40% per year
  - Per drive capacity increases by 70% to 100% per year
  - 2008: **1 TB**
  - 2009: **2 TB** (enterprise SATA available 2nd half of 2009)
  - Drive vendors prepared to continue like this for years to come
- Drive interface speed increases by 10-15% per year
  - 2008: 500 GB disk (WD RE2): **98 MB/sec**
  - 2009: 1 TB disk (WD RE3): **113 MB/sec**
- 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…

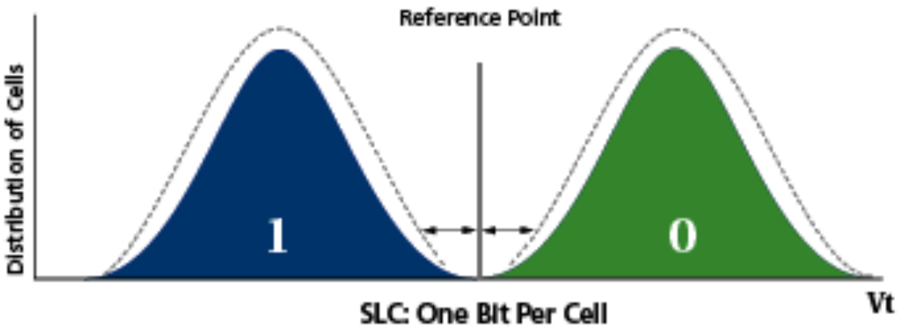
- Non-volatile
  - Each bit is stored in a “floating gate” that holds value without power
  - Electrons can leak, so shelf life and write count is limited
- Page-oriented
  - Read, write, and erase operations apply to large chunks
  - Smaller (e.g., 4K) read/write block based on addressing logic
  - Larger (e.g., 256K) erase block to amortize the time it takes to erase
- Medium speed
  - Slower than DRAM
  - Faster than disks (especially for read, not always for write)
  - Write speed heavily dependent on workload
- Relatively cheap

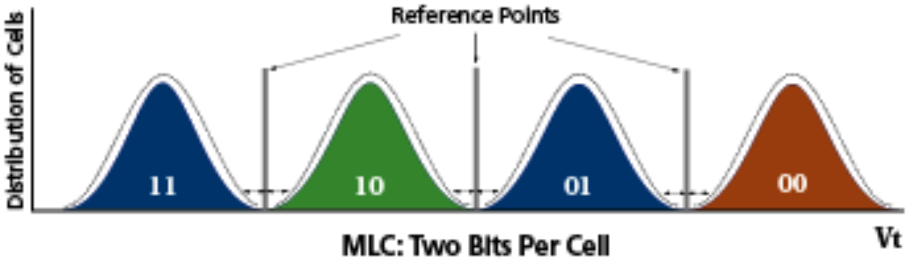


http://icrontic.com/articles/how_ssds_work

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

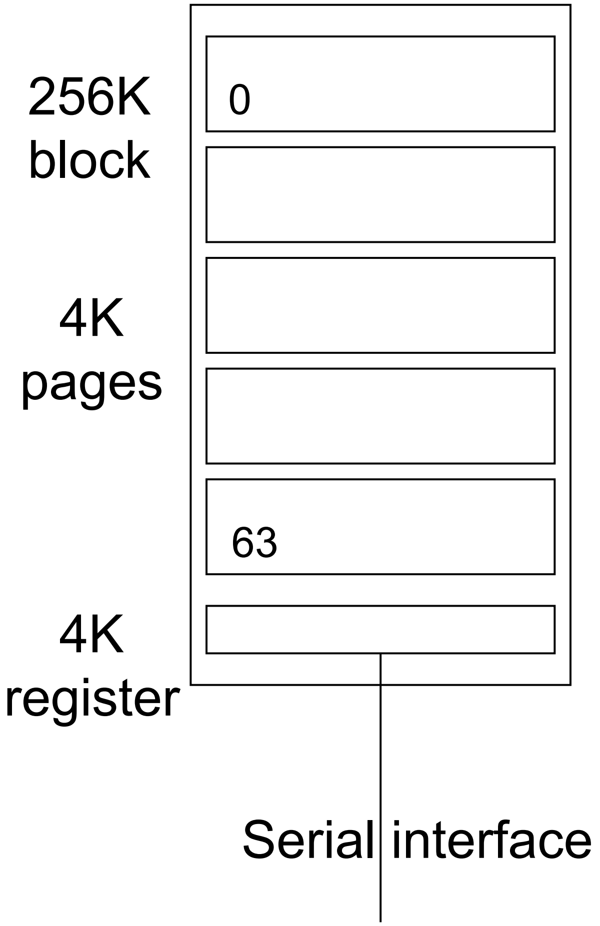




http://www.micron.com/nandcom/

FLASH Speeds

Samsung 4 GB Device



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

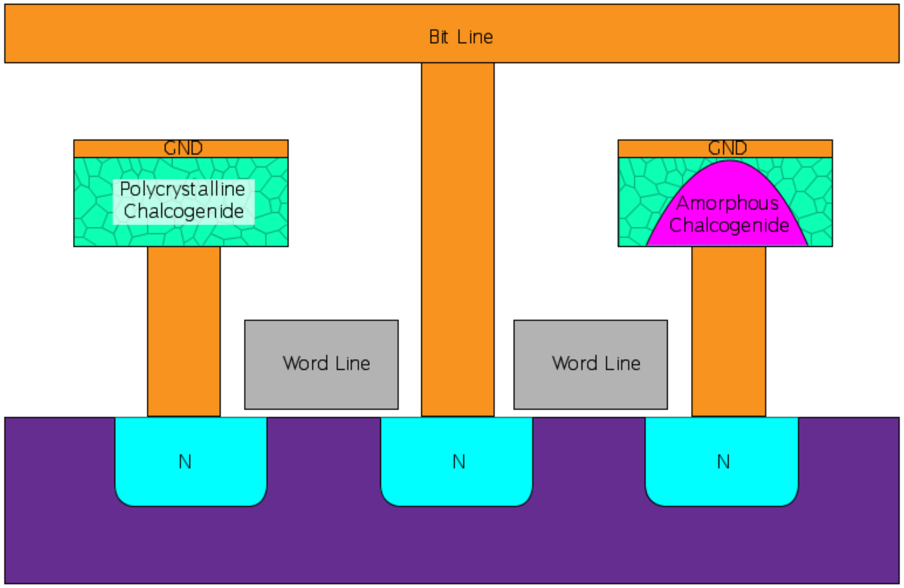
- Write performance heavily dependent on workload and wear leveling algorithms
- Writes are slower with less free space

FLASH in the Storage Hierarchy

- On the compute nodes
  - High reliability local storage for OS partition
  - Local cache for memory checkpoints?
    - Device write speeds vary widely
      - 4 MB/sec for a cheap USB
      - 80 or 100 MB/sec for MTron or Zeus
      - 600 MB/sec for Fusion-io ioDrive
  - One Fusion-io board could double cost of 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

FLASH Summary

- FLASH is a midpoint between DRAM and HDDs
  - Attractive because of cost and non-volatile
  - Performance and reliability characteristics make the system design non-trivial
- Phase-change memories are a newer technology that may replace FLASH in 2-5 years
  - Material that changes magnetic polarity when voltage applied
    - Like old core memory but at the VLSI scale instead of wires and magnets
  - More like DRAM in access characteristics (e.g., no block erase required)
  - 100+ million erase cycles
  - Sounds promising…



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

Wrapping Up

- We've covered a lot of ground in a short time
  - Disk drives & filesystems
  - Benchmarking
  - Programming middleware
  - pNFS and FLASH
- There is no magic in high performance I/O
  - Under the covers it looks a lot like shared memory or message passing
  - Knowing how things work will lead you to better performance
- Things will continue to get more complicated, but hopefully easier too!
  - Remote access to data
  - More layers to I/O stack
  - Domain-specific application interfaces

Thank you!

Brent Welch, Marc Unangst
{welch,mju}@panasas.com
Panasas, Inc.

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
- 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

Online References: Filesystems

- ROMIO MPI-IO
  - http://www.mcs.anl.gov/romio/
- POSIX I/O Extensions
  - http://www.opengroup.org/platform/hecewg/

- 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/

Online References: Benchmarks

- 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
- HPC Challenge
  - http://icl.cs.utk.edu/hpcc/)
- SPEC HPC2002
  - http://www.spec.org/hpc2002/
- NAS Parallel Benchmarks
  - http://www.nas.nasa.gov/Resources/Software/npb.html

Online References: pNFS

- NFS Version 4.1
  - draft-ietf-nfsv4-minorversion1-29.txt
  - draft-ietf-nfsv4-pnfs-obj-09.txt
  - draft-ietf-nfsv4-pnfs-block-09.txt
  - http://tools.ietf.org/wg/nfsv4/
- 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