School of Parallel Programming & Parallel Architecture for HPC
ICTP
October, 2014

Parallel I/O for HPC

Instructor: Ekpe Okorafor

Topics

- Introduction
- Distributed File Systems (NFS)
- Parallel File Systems: Introduction
- Parallel File Systems (PVFS2)
- Parallel File Systems (Lustre)
- Additional Parallel File Systems (GPFS)
- POSIX I/O API

Topics

- Introduction
- Distributed File Systems (NFS)
- Parallel File Systems: Introduction
- Parallel File Systems (PVFS2)
- Parallel File Systems (Lustre)
- Additional Parallel File Systems (GPFS)
- POSIX I/O API

I/O Needs on Parallel Computers

- High Performance
  - Take advantage of parallel I/O paths (where available)
  - Support application-level data access and throughput needs
  - Scalable with systems size and user number/needs
- Data Integrity
  - Sanely deal with hardware and power failures
- Single Namespace
  - All nodes and users “see” the same file systems
  - Equal access from anywhere on the resource
- Ease of Use
  - Whenever possible, a storage system should be accessible in consistent way, in the same ways as a traditional UNIX-style file systems

Related Topics

- Hardware-based solutions
  - RAID
- File systems commonly used in parallel computing
  - NFS
  - PVFS2
  - Lustre
  - GPFS
- Software I/O libraries
  - POSIX I/O
  - MPI-IO
  - NetCDF
  - HDF5

Topics

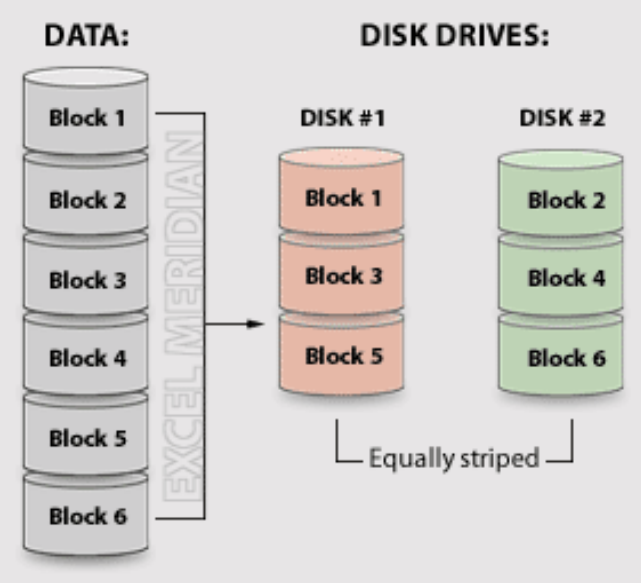
- Introduction
- RAID
- Distributed File Systems (NFS)
- Parallel File Systems: Introduction
- Parallel File Systems (PVFS2)
- Parallel File Systems (Lustre)
- Additional Parallel File Systems (GPFS)
- POSIX I/O API

RAID: Key Concepts

- RAID stands for ***Redundant Array of Inexpensive*** (or: *Independent*) ***Disks*** - term coined by David Patterson, Garth Gibson and Randy Katz
- Aims to achieve greater levels of performance, reliability, and/or larger volume sizes
- Several basic architectures, ranging from RAID 0 through RAID 6
- Groups of two or more disks are treated effectively as single large disks; performance of multiple disks is better than that of individual disks due to bandwidth aggregation and overlap of multiple accesses
- Using multiple disks helps store data in multiple places (redundancy), allowing the system to continue functioning in case of failures
- Both software (OS managed) and hardware (dedicated I/O cards) raid solutions available
  - Hardware solutions are more expensive, but provide better performance without CPU overhead
  - Software solutions provide better flexibility, but have associated computational overhead

RAID 0

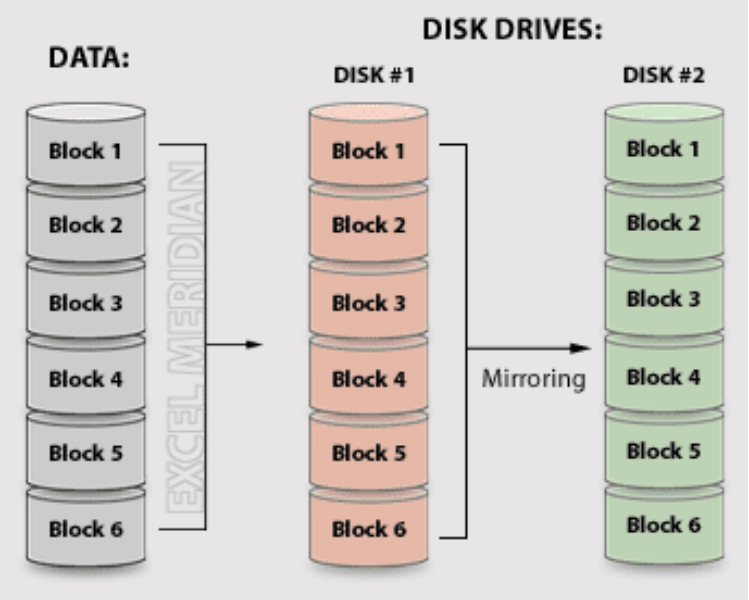
- Simple *disk striping* without fault tolerance (single failure destroys the array)
- Data are striped across multiple disks
- The result of striping is a logical storage device that has the capacity of each disk times the number of disks present in the raid array
- Both read and write performances are accelerated
- Each byte of data can be read from multiple locations, so interleaving reads between disks can help double read performance
- High transfer rates
- High request rates



http://www.drivesolutions.com/datarecovery/raid.shtml

RAID 1

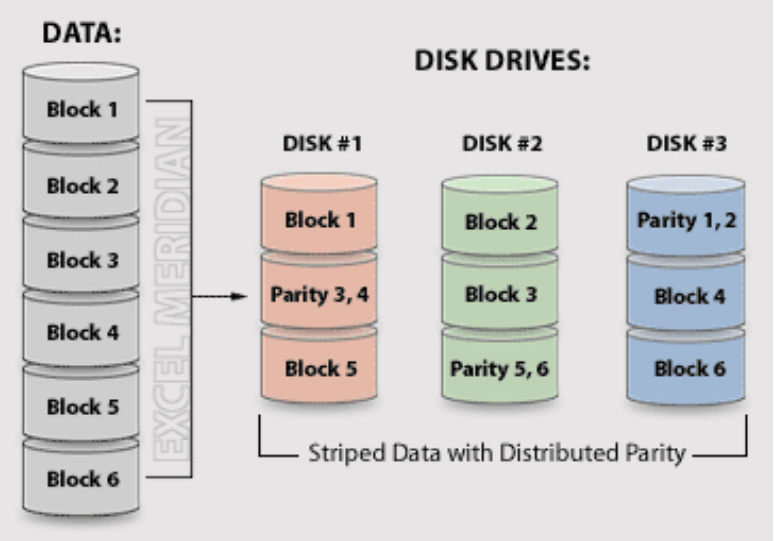
- Called *disk mirroring*
- Frequently implemented with only two drives
- Complete copies of data are stored in multiple locations; no parities computed
- Has the highest spatial overhead for redundant data of all RAID types
- Usable capacity is equivalent to a single component disk capacity
- Read performance is accelerated due to availability of concurrent seek operations
- Writes are somewhat slowed down, as new data have to be written to multiple storage devices (concurrency depends on the controller)
- Continues functioning as long as least one drive is working correctly



http://www.drivesolutions.com/datarecovery/raid.shtml

RAID 5

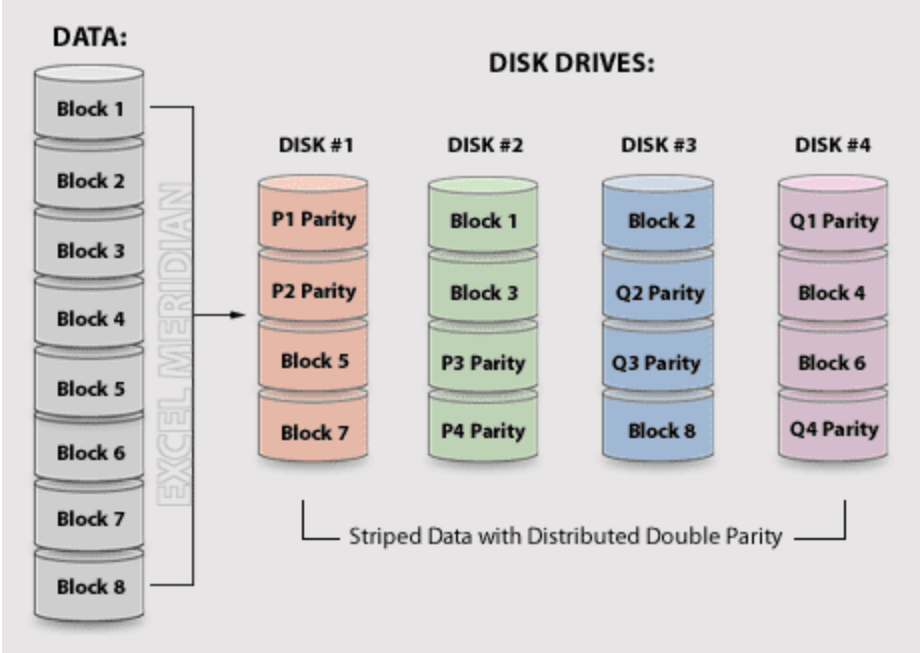
- *Striped* set with *distributed parity*
- Requires at least 3 disks per array
- Similarly to RAID 0, data are distributed across all the disks
- Parity information for each stripe is also distributed across the disks, eliminating the bottleneck of a single parity disk (RAID 4) and equalizing the load across components
- Tolerates single disk failures; the missing data block may be recomputed based on the contents of the parity block and the data blocks of the remaining disks in the same stripe
- Write performance of RAID 5 is reduced due to parity computation for every physical write operation



http://www.drivesolutions.com/datarecovery/raid.shtml

RAID 6

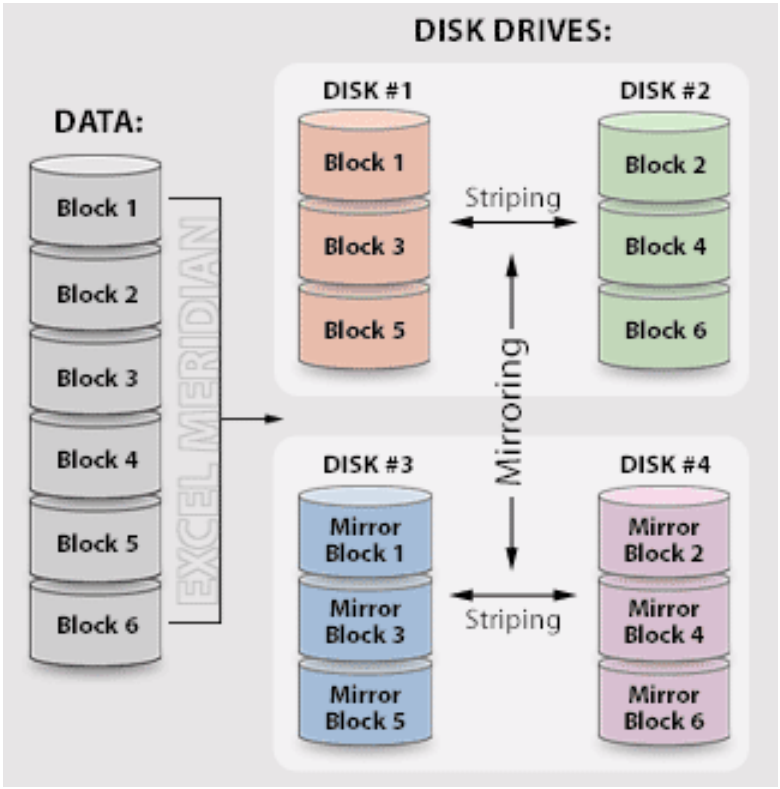
- *Striped* set with *distributed dual parity*
- Requires at least 4 disks per array
- Similar to RAID 5, but with two parity blocks per each stripe computed using different algorithms
- Frequently utilized in large-volume secondary storage, where high availability is of concern
- Can tolerate double simultaneous disk failures; it is practical in large volume sets, in which the rebuild of an array with one failed disk may not complete before the occurrence of a second failure



http://www.drivesolutions.com/datarecovery/raid.shtml

Nested RAID Levels

- Hybrid arrangement of different level RAID arrays in the same storage subsystem
- Combines redundancy schemes of RAID 1, 5 or 6 with the performance of RAID 0
- Typically require an additional increase in minimal number of disks per array over the minimum for any of the component RAID levels
- Simpler hybrids are frequently supported directly by I/O chipsets available on motherboards (RAID 0+1, 10, etc.)
- Can have more than two levels (e.g., RAID 100), with top levels typically implemented in software



Example: RAID 0+1 (mirror of stripes)

Topics

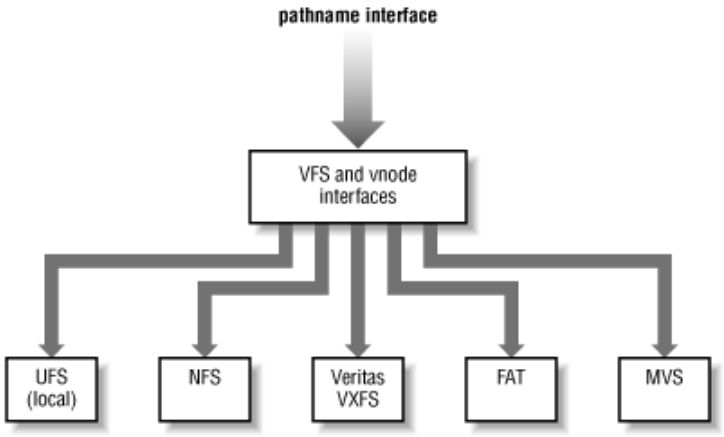
- Introduction
- RAID
- Distributed File Systems (NFS)
- Parallel File Systems: Introduction
- Parallel File Systems (PVFS2)
- Parallel File Systems (Lustre)
- Additional Parallel File Systems (GPFS)
- POSIX I/O API

Distributed File Systems

- A distributed file system is a file system that is stored locally on one system (server) but is accessible by processes on many systems (clients)
- Multiple processes access multiple files simultaneously
- Other attributes of a DFS may include:
  - Access control lists (ACLs)
  - Client-side file replication
  - Server- and client- side caching
- Some examples of DFSes:
  - NFS (Sun)
  - AFS (CMU)
  - DCE/DFS (Transarc / IBM)
  - CIFS (Microsoft)
- Distributed file systems can be used by parallel programs, but they have significant disadvantages:
  - The network bandwidth of the server system is a limiting factor on performance
  - To retain UNIX-style file consistency, the DFS software must implement some form of locking which has significant performance implications

Distributed File System: NFS

- Popular means for accessing remote file systems in a local area network
- Based on the client-server model, the remote file systems may be “mounted” via NFS and accessed through the Linux Virtual File System (VFS) layer
- NFS clients cache file data, periodically checking with the original file for any changes
- The loosely-synchronous model makes for convenient, low-latency access to shared spaces
- NFS avoids the common locking systems used to implement POSIX semantics
- Most client implementations are open-source; many servers remain proprietary



NFS support via VFS layer in Linux

Why NFS is bad for Parallel I/O

- Clients can cache data indiscriminately, and tend to do that at arbitrary block boundaries
- When nearby regions of a file are written by different processes on different clients, the result is undefined due to lack of consistency control
- All file operations are remote operations; extensive file locking is required to implement sequential consistency
- Communication between client and server typically uses relatively slow communication channels, adding to performance degradation
- Inefficient specification (e.g., a read operation involves two RPCs; one for look-up of file handle and second for reading of file data)

Topics

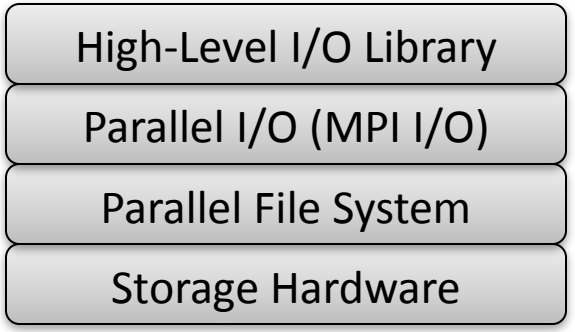
- Introduction
- RAID
- Distributed File Systems (NFS)
- Parallel File Systems: Introduction
- Parallel File Systems (PVFS2)
- Parallel File Systems (Lustre)
- Additional Parallel File Systems (GPFS)
- POSIX I/O API

Parallel File Systems

- Parallel File System is one in which there are multiple servers as well as clients for a given file system, equivalent of RAID across several file systems.
- Multiple processes can access the same file simultaneously
- Parallel File Systems are usually optimized for high performance rather than general purpose use, common optimization criterion being :
  - Large block sizes (≥ 64kB)
  - Relatively slow metadata operations (eg. fstat()) compared to reads and writes
  - Special APIs for direct access and additional optimizations
- Examples of Parallel file systems include :
  - GPFS (IBM)
  - Lustre (Cluster File Systems/Sun)
  - PVFS2 (Clemson/ANL)

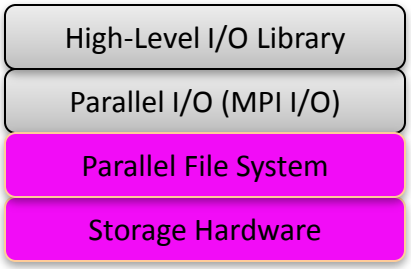
Characteristics of Parallel File Systems

- Three Key Characteristics:
  - Various hardware I/O data storage resources
  - Multiple connections between these hardware devices and compute resources
  - High-performance, concurrent access to these I/O resources
- Multiple physical I/O devices and paths ensure sufficient bandwidth for the high performance desired
- Parallel I/O systems include both the hardware and number of layers of software



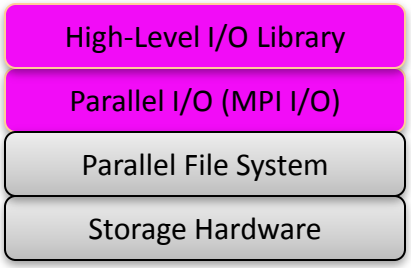
Parallel File Systems: Hardware Layer

- I/O Hardware is usually comprised of disks, controllers, and interconnects for data movement
- Hardware determines the maximum raw bandwidth and the minimum latency of the system
- Bisection bandwidth of the underlying transport determines the aggregate bandwidth of the resulting parallel I/O system
- At the hardware level, data is accessed at the granularity of blocks, either physical disk blocks or logical blocks spread across multiple physical devices such as in a RAID array
- Parallel File Systems :
  - manage data on the storage hardware,
  - present this data as a directory hierarchy,
  - coordinate access to files and directories in a consistent manner
- File systems usually provide a UNIX like interface, allowing users to access contiguous regions of files



Parallel File Systems: Other Layers

- Lower level interfaces may be provided by the file system for higher-performance access
- Above the parallel file systems are the parallel I/O layers provided in the form of libraries such as MPI-IO
- The parallel I/O layer provides a low level interface and operations such as collective I/O
- Scientific applications work with structured data for which a higher level API written on top of MPI-IO such as HDF5 or parallel netCDF are used
- HDF5 and parallel netCDF allow the scientists to represent the data sets in terms closer to those used in their applications, and in a portable manner



Topics

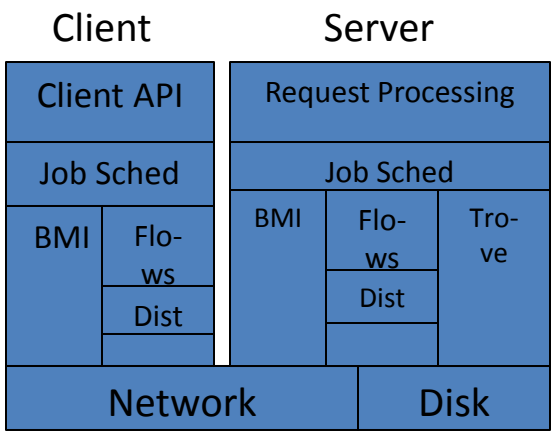
- Introduction
- RAID
- Distributed File Systems (NFS)
- Parallel File Systems: Introduction
- Parallel File Systems (PVFS2)
- Parallel File Systems (Lustre)
- Additional Parallel File Systems (GPFS)
- POSIX I/O API

Parallel File Systems: PVFS2

- PVFS2 designed to provide:
  - modular networking and storage subsystems
  - structured data request format modeled after MPI datatypes
  - flexible and extensible data distribution models
  - distributed metadata
  - tunable consistency semantics
  - support for data redundancy
- Supports variety of network technologies including Myrinet, Quadrics, and Infiniband
- Also supports variety of storage devices including locally attached hardware, SANs and iSCSI
- Key abstractions include:
  - Buffered Message Interface (BMI): non-blocking network interface
  - Trove: non-blocking storage interface
  - Flows: mechanism to specify a flow of data between network and storage

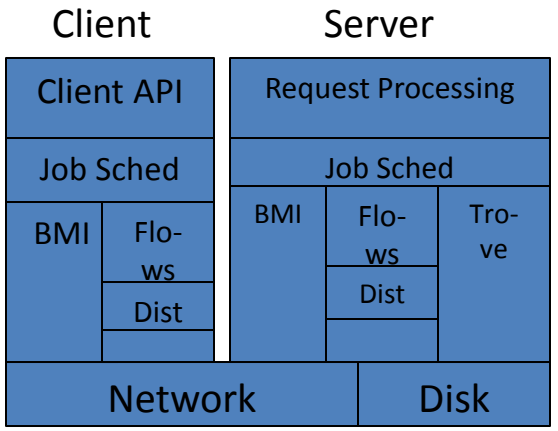
PVFS2 Software Architecture

- Buffered Messaging Interface (BMI)
  - Non blocking interface that can be used with many High performance network fabrics
  - Currently TCP/IP and Myrinet (GM) networks exist
- Trove:
  - Non blocking interface that can be used with a number of underlying storage mechanisms
  - Trove storage objects consist of stream of bytes (data) and keyword/value pair space
  - Keyword/value pairs are convenient for arbitrary metadata storage and directory entries



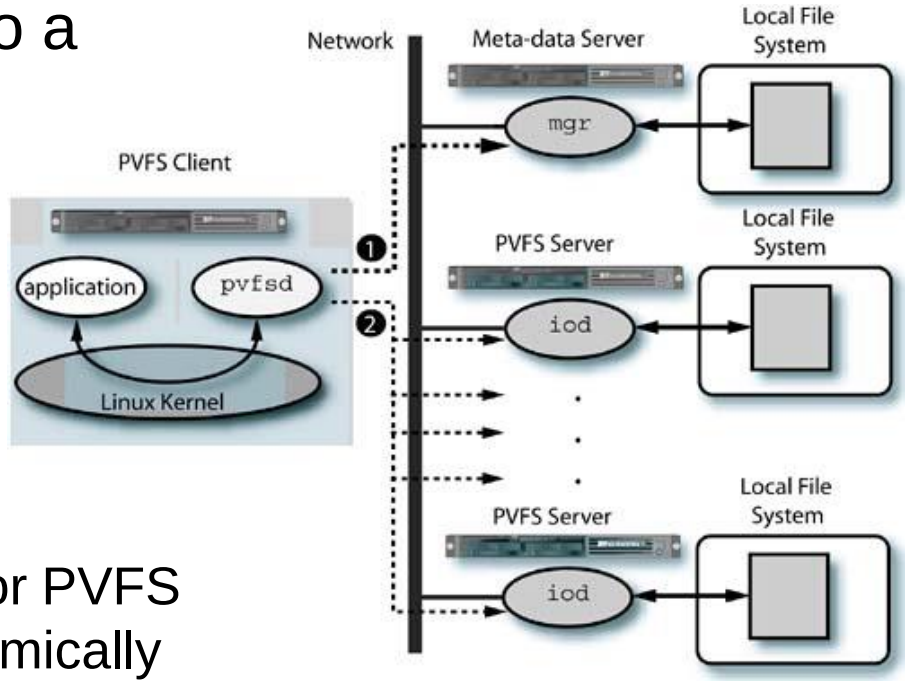
PVFS2 Software Architecture

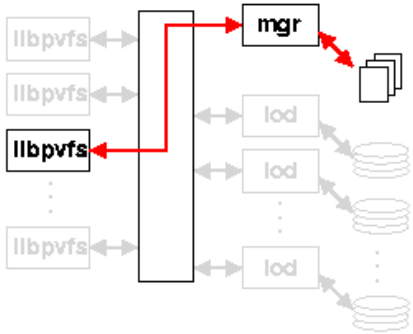
- Flows:
  - Combine network and storage subsystems by providing mechanism to describe flow of data between network and storage
  - Provide a point for optimization to optimize data movement between a particular network and storage pair to exploit fast paths
- The job scheduling layer provides a common interface to interact with BMI, Flows, and Trove and checks on their completion
- The job scheduler is tightly integrated with a state machine that is used to track operations in progress



The PVFS2 Components

- The four major components to a PVFS system are:
  - Metadata Server (mgr)
  - I/O Server (iod)
  - PVFS native API (libpvfs)
  - PVFS Linux kernel support
- Metadata Server (mgr):
  - manages all the file metadata for PVFS files, using a daemon which atomically operates on the file metadata
  - PVFS avoids the pitfalls of many storage area network approaches, which have to implement complex locking schemes to ensure that metadata stays consistent in the face of multiple accesses

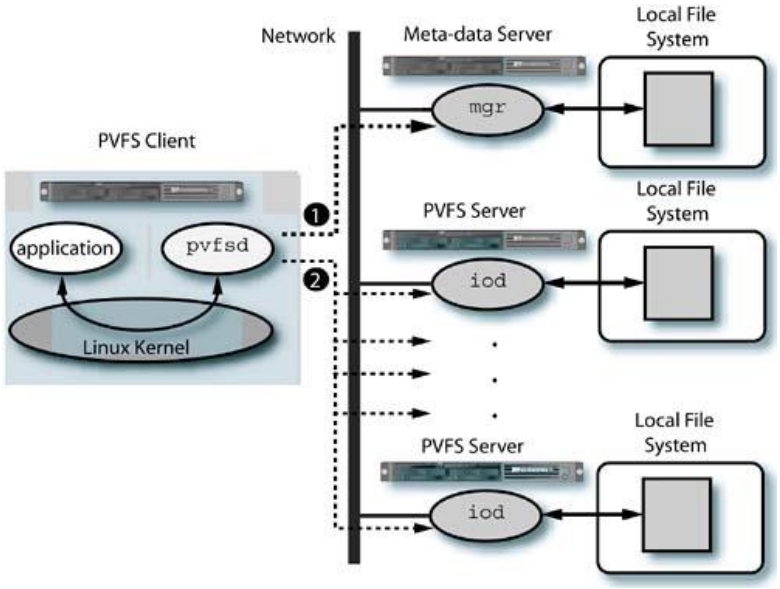


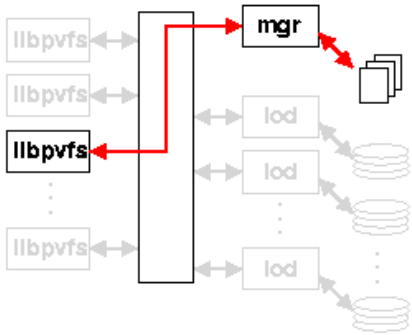


metadata access

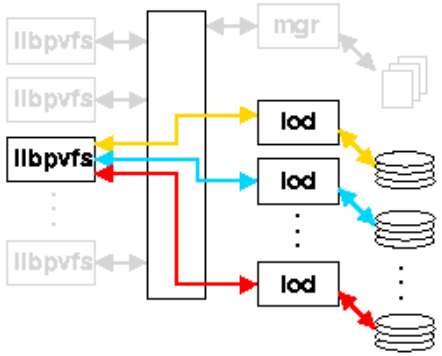
The PVFS2 Components

- I/O daemon:
  - handles storing and retrieving file data stored on local disks connected to a node using traditional read(), write(), etc for access to these files
- PVFS native API provides user-space access to the PVFS servers
- The library handles the operations necessary to move data between user buffers and PVFS servers





metadata access



data access

http://csi.unmsm.edu.pe/paralelo/pvfs/desc.html

Topics

- Introduction
- RAID
- Distributed File Systems (NFS)
- Parallel File Systems: Introduction
- Parallel File Systems (PVFS2)
- Parallel File Systems (Lustre)
- Additional Parallel File Systems (GPFS)
- POSIX I/O API

Parallel File Systems: Lustre

- Name loosely derived from “Linux” and “cluster”
- Originally developed by Cluster File Systems Inc., which was acquired by Sun in Oct. 2007
  - Sun’s intention was to bring the benefits of the technology to its native ZFS and Solaris
- Scalable, secure and highly-available file system for clusters
- Targets clusters with 10,000s of nodes, petabytes of storage, and is capable of providing 100’s of GB/s data bandwidth
- Multiplatform: Linux/Solaris clusters (Intel 32/64-bit), BG/L (PowerPC)
- Supports a wide variety of networks (TCP/IP, Quadrics, InfiniBand, Myrinet GM)
  - Remote DMA where available
  - OS bypass for parallel I/O
  - Vector I/O for efficient bulk data movement
- High-performance
  - Separate data manipulation and metadata operations
  - Intelligent serialization
  - Distributed lock management for metadata journaling
  - Intent-based locking (combines lock acquisition with the associated target operation)
- High-reliability
  - No single point of failure (organizes servers in active-active failover pairs)
  - Permits live cluster upgrades
  - “Just-mount” configuration, based on aggregation of server devices
- Supports POSIX semantics
- Open source
- Deployed in systems at LLNL, ORNL, PNNL, LANL, TI Tech (Japan), CEA (Europe), and locally on LONI resources (Queen Bee, Eric, Oliver, Louie)

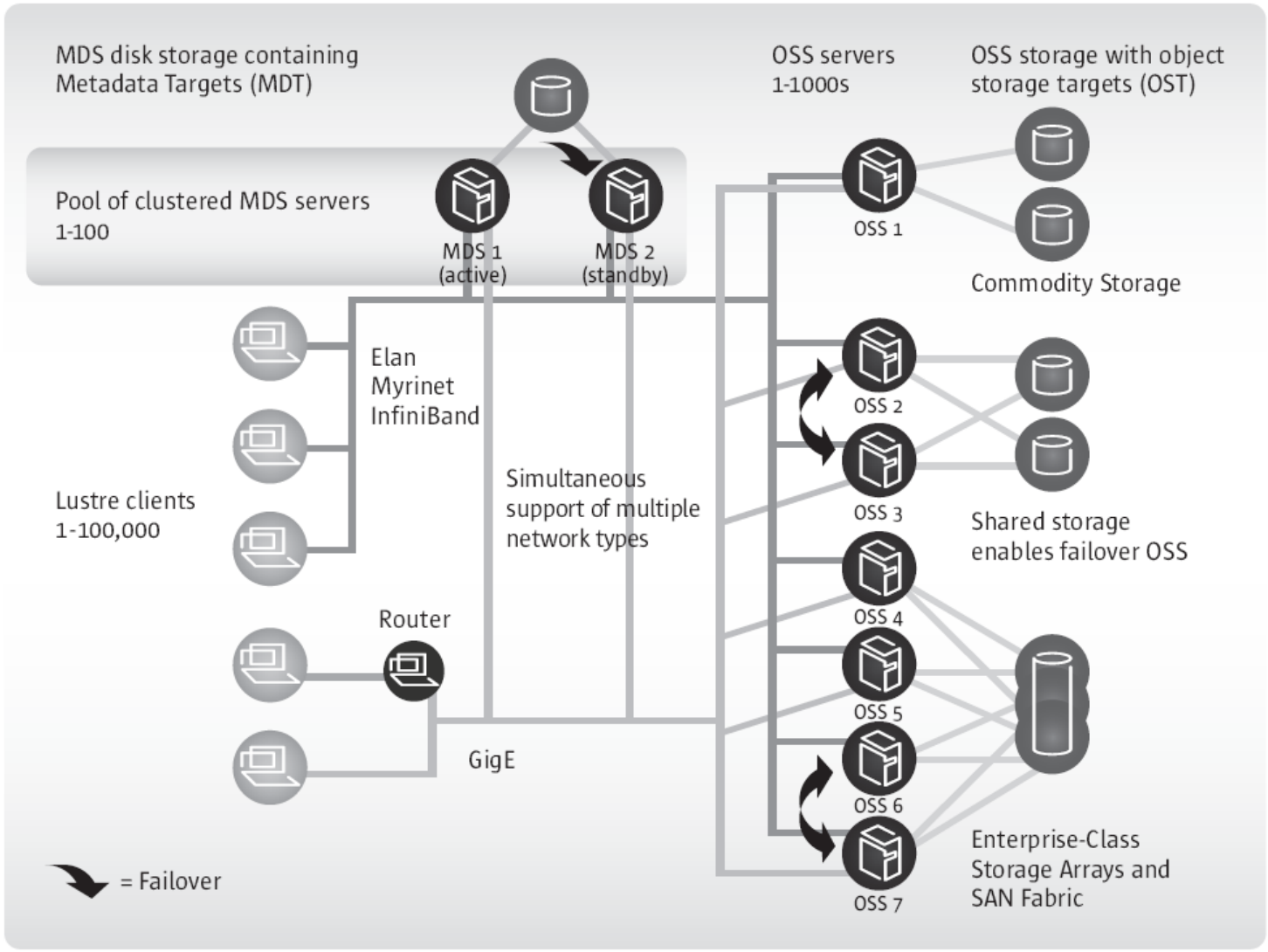
Recent Lustre Datasheet

- Fraction of raw bandwidth utilized by file I/O: >90%
- Achieved single OSS (server) I/O: >2.5 GB/s
- Achieved single client I/O: >2 GB/s
- Single GigE end-to-end throughput: 118 MB/s
- Achieved aggregate I/O: 130 GB/s
- Metadata transaction rate: 15,000 ops/s
- Maximum clients supported: 25,000
- Maximum file size: 1.25 PB
- Maximum file system size: >32 PB

Key Components of Lustre FS

- Clients
  - Generate data I/O and metadata requests
  - No internal storage
- Object Storage Servers (OSS)
  - Uses partitioned storage, with optional LVM (Logical Volume Management)
  - Responsible for reading, writing and modifying data in format imposed by the underlying file system(s)
  - Balance bandwidth between the network and attached storage to prevent bottlenecks
  - May use attached external storage arrays (via Fiber Channel or SAS), including RAID
  - Can handle multiple Object Storage Targets (OSTs), up to 8 TB each; one OST required for each volume
  - Total capacity of Lustre FS is the sum of capacities of all targets
- Metadata Servers (MDS)
  - Similarly to OSSs, responsible for managing the local storage in the native file system format
  - Require low latency access (fast seeks) rather than throughput due to small size of metadata requests (FC and SAS coupled with RAID 0+1 are the recommended storage types for that purpose, and may be different than those used by OSSs)
  - Placing journal on a separate device frequently improves performance (up to 20%)
  - At least four processing cores per MDS recommended

Lustre Architecture

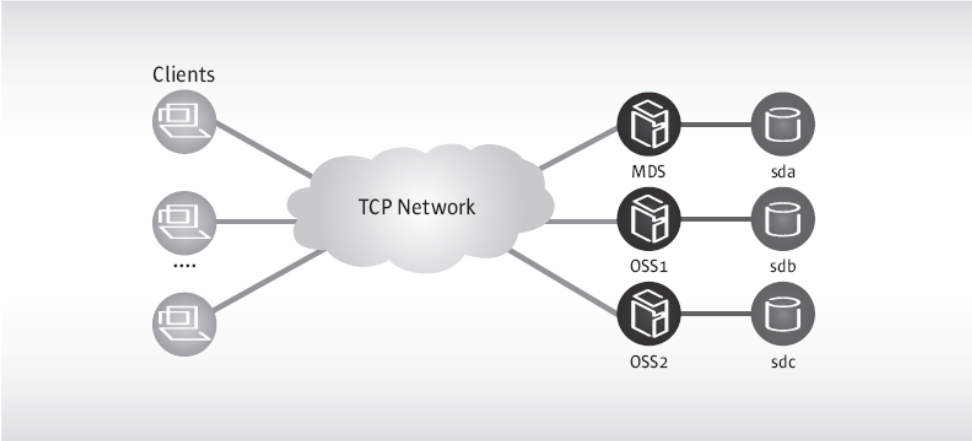


Lustre Component Characteristics

- Clients
  - 1 - 100,000 per system
  - Data bandwidth up to few GB/s; 1000's of metadata ops/s
  - No particular hardware or attached storage characteristics required
- Object Storage Servers (OSS)
  - 1 - 1000 per system
  - 500 MB/s..2.5 GB/s I/O bandwidth
  - Require good network bandwidth and adequate local storage capacity coupled with sufficient number of OSSs in the system
- Metadata Servers (MDS)
  - 2 (in the future: 2 - 100) per system
  - 3,000 - 15,000 metadata ops/s
  - Utilize 1 – 2% of total file system capacity
  - Require memory-rich nodes with lots of CPU processing power

Lustre Configuration

A simple Lustre System



…and its configuration

On the MDS mds.your.org@tcp0:

```
mkfs.lustre --mdt --mgs --fsname=large-fs /dev/sdamount -t
lustre /dev/sda /mnt/mdt
```

On OSS1:

```
mkfs.lustre --ost --fsname=large-fs --mgsnode=mds.your.
org@tcp0 /dev/sdb mount -t lustre /dev/sdb /mnt/ost1
```

On OSS2:

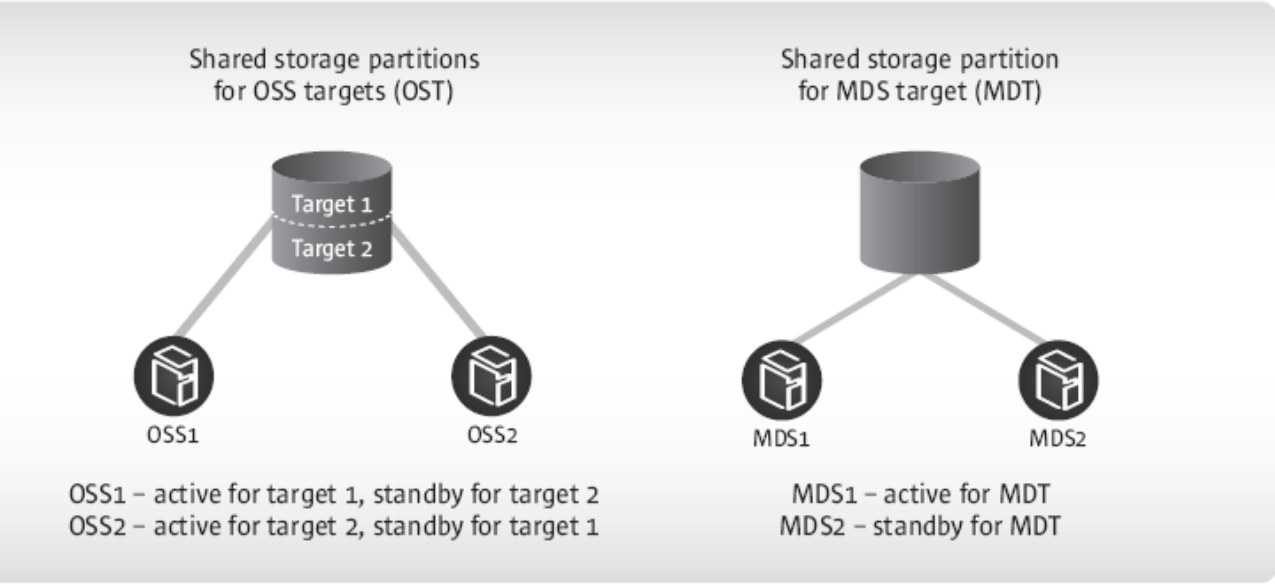
```
mkfs.lustre --ost --fsname=large-fs --mgsnode=mds.your.
org@tcp0 /dev/sdc mount -t lustre /dev/sdc /mnt/ost2
```

On clients:

```
mount -t lustre mds.your.org:/large-fs /mnt/lustre-client
```

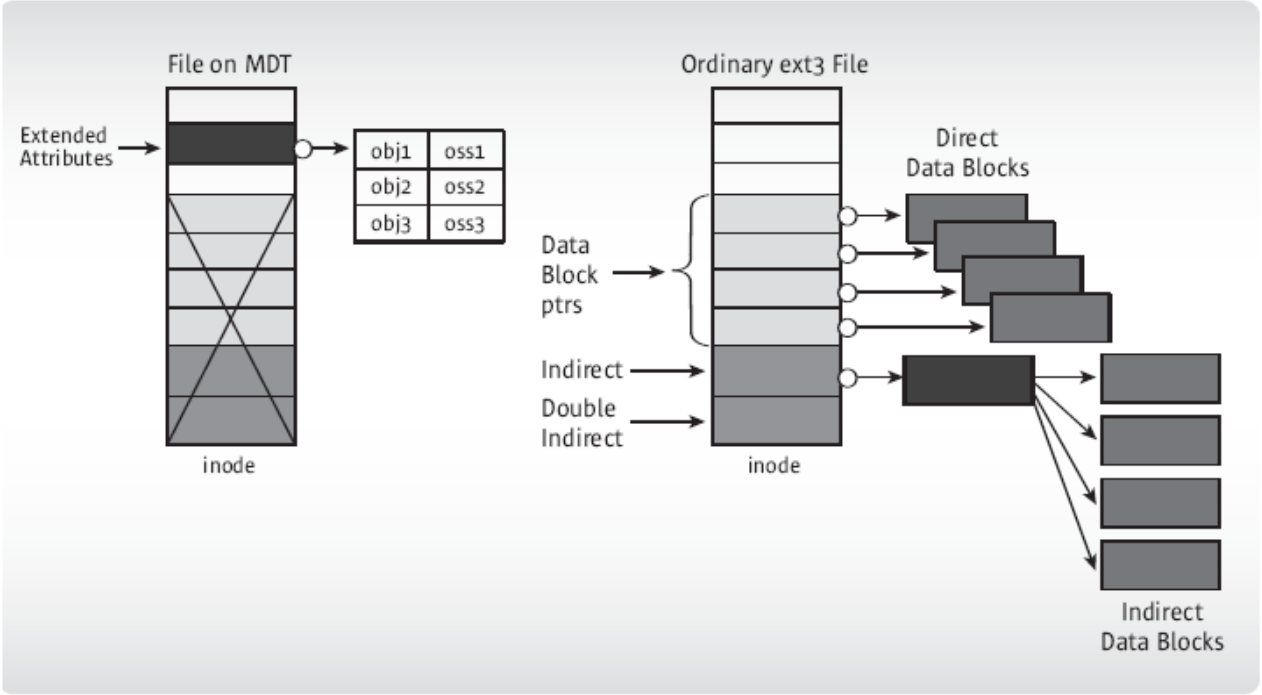
Lustre High-Availability Features

- Transparently handles server reboots and failures
  - The only client-visible effect is increase in request processing delay
- Supports rolling software updates
  - Updated server is taken off-line, upgraded and restarted without taking the system down
  - Alternatively, it may be failed over to the standby server with a new software
- MDSs are configured in active-passive pairs
  - One standby MDS can be used as active MDS for another Lustre FS, minimizing the number of idle nodes in the cluster
- OSSs are configured in active-active pairs to provide redundancy without the extra overhead
- In worst case, a file system checking tool (lfsck) is provided for disaster recovery



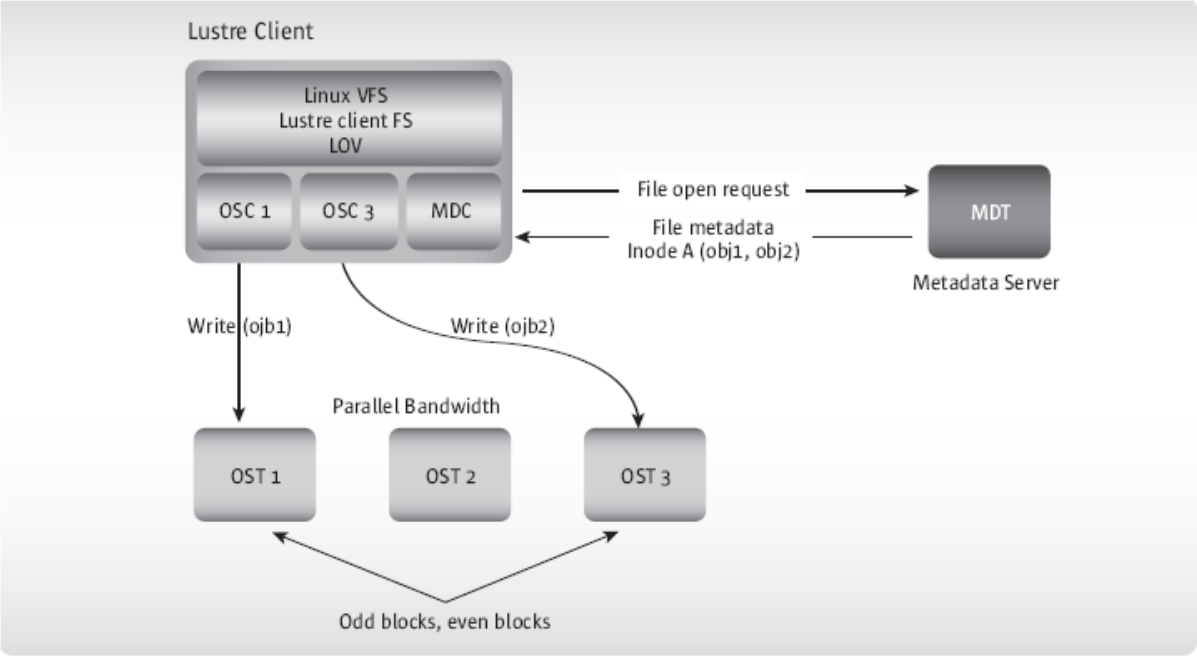
Lustre File Layout

- One *inode* per file
- Instead of pointing directly at data blocks, MDT inodes point to one or more objects associated with files
- Striping is achieved by associating multiple objects with an MDS inode
- The objects are implemented as files on OST file systems and contain actual file data



Lustre File Operations

- File open:
  - Request is send to appropriate MDS
  - Related object pointers are returned to client
- File data access
  - Object pointer is used to access directly the OSS nodes where the file data are stored



Advanced Features of Lustre

- Interoperability
  - Multiple CPU architectures supported within single system (clients and servers are interoperable)
  - Software-level interoperability exists between adjacent releases
- Support for POSIX Access Control Lists (ACL)
- Quotas for both users and groups
- OSS (and OST) addition to increase the total file system capacity may be accomplished without interrupting the operation
- Flexible control of striping parameters (stripe count and stripe size)
  - Default settings at format time
  - Through directory (and subdirectory) attributes
  - Defined by user library calls at file creation time
- Snapshots of all volumes attached to server nodes may be created using LVM utilities, and later grouped in a snapshot file mountable within the Lustre file system
- Backup tools:
  - Fast file scanner, detecting files modified since given time stamp; its output (list of files) can be used directly and in parallel by standard backup clients (e.g. *rsync*)
  - Modified version of *star* utility to backup and restore Lustre stripe information

Parallel File Systems Comparison

	PVFS	GPFS	Lustre
Storage	Storage target	Storage target	OBD
Block device	Only file system device	Any block device	Block device with proprietary driver layer
Nodes directly connected to storage	IOD	NSD, every node in SAN mode	OST
Storage type	IDE, SCSI, RAID	SAN, RAID	RAID, SCSI, limited SAN
Daemon communication	TCP/IP	TCP/IP	Portal
Metadata server	MGR	NSD, every node in SAN mode	MDS

Topics

- Introduction
- RAID
- Distributed File Systems (NFS)
- Parallel File Systems: Introduction
- Parallel File Systems (PVFS2)
- Parallel File Systems (Lustre)
- Additional Parallel File Systems (GPFS)
- POSIX I/O API

General Parallel File System (GPFS)

- Brief history:
  - Based on the Tiger Shark parallel file system developed at the IBM Almaden Research Center in 1993 for AIX
    - Originally targeted at dedicated video servers
    - The multimedia orientation influenced GPFS command names: they all contain “mm”
  - First commercial release was GPFS V1.1 in 1998
  - Linux port released in 2001; Linux-AIX interoperability supported since V2.2 in 2004
- Highly scalable
  - Distributed metadata management
  - Permits incremental scaling
- High-performance
  - Large block size with wide striping
  - Parallel access to files from multiple nodes
  - Deep prefetching
  - Adaptable mechanism for recognizing access patterns
  - Multithreaded daemon
- Highly available and fault tolerant
  - Data protection through journaling, replication, mirroring and shadowing
  - Ability to recover from multiple disk, node and connectivity failures (heartbeat mechanism)
  - Recovery mechanism implemented in all layers

GPFS Features (I)

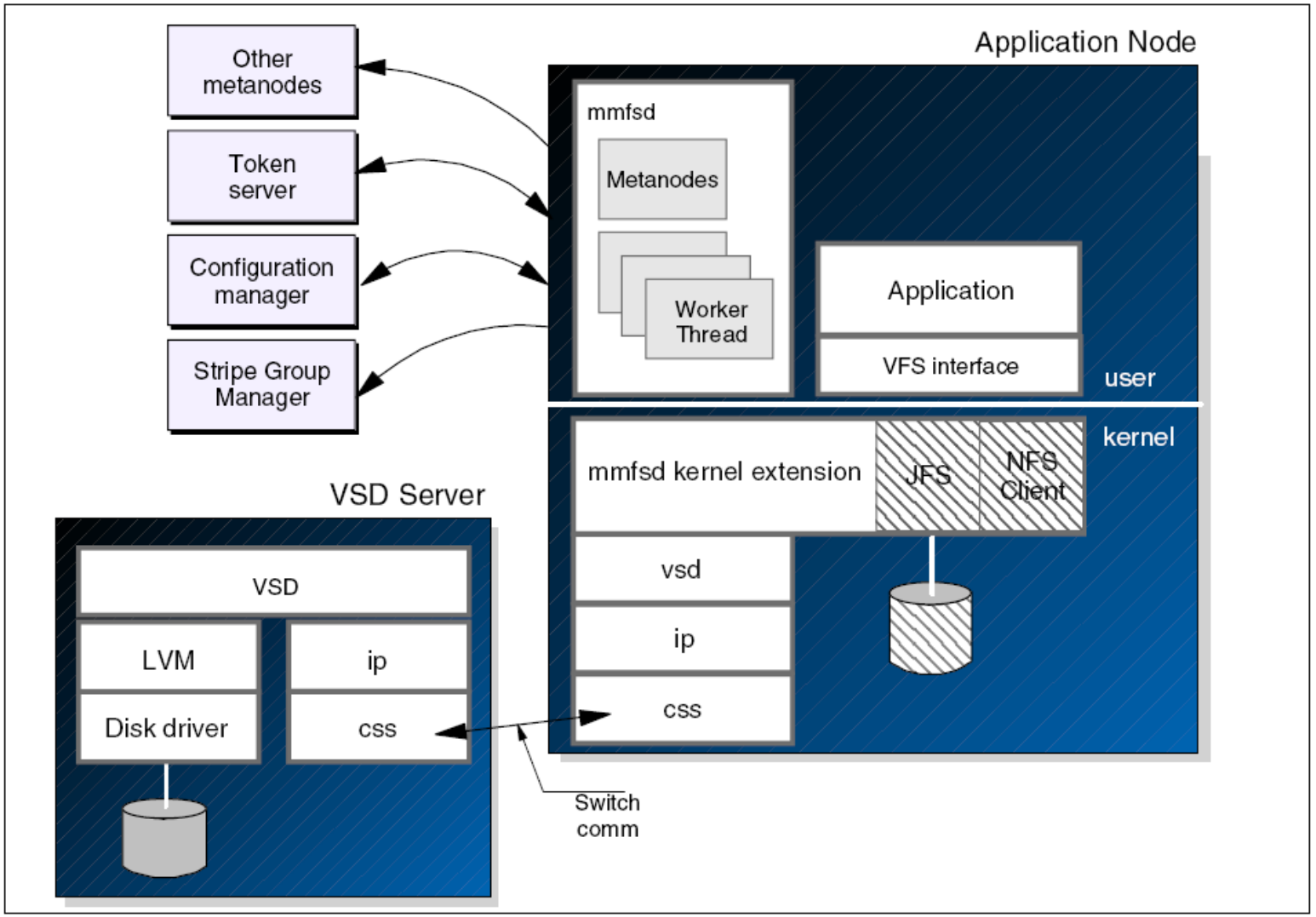
Feature	Benefit
Highly Scalable File System Designed for a cluster environment, GPFS can support hundreds of nodes and over 1000 disks comprising hundreds of terabytes of storage	Although relatively small environments can benefit from GPFS, it allows expansion of file system throughput and capacity as service demands and data volumes increase
Parallel File System (Data Striping) Divides individual files into multiple blocks and stores these blocks across multiple storage nodes in parallel	Higher performance and reliability by eliminating the bottlenecks that typically arise when an entire file resides on a single node
New! Storage Pools User-defined partitioning/grouping of storage	Storage can be partitioned or tiered based on factors such as location, performance and reliability
New! Filesets More granular partitioning of the file system	Fine-grained control of specific subsets of the file system. For example, all files for single project can be managed as a group or limited to a specific total size
New! Policy-Driven Automation Automatically place files in a specific storage pool based on attributes of the file. Files can also be moved between pools or deleted automatically	Simplifies data management and allows matching the cost of storage to the value/importance of the data: critical data can be stored on the fastest storage etc.
Automatic Journaling (Logging) of Metadata Logs all file system metadata (file system transaction records) to shared disks and can replay this data on demand	Enables rapid file system recovery to a consistent state in the event of a node failure

Source: http://www-03.ibm.com/systems/clusters/software/gpfs.pdf

GPFS Features (II)

Concurrent File Access and Block-Level Locking Multiple clients (applications or uses) can access (different parts of) a single file simultaneously; sophisticated lock management prevents collisions/conflicts and ensures data consistency	Enables multiple users, programs or nodes to access a single file simultaneously – accelerates processing for parallel applications that share data and eliminates the need for multiple copies of data, reducing data storage and management costs
High-Availability Automatically recovers from events that would normally interrupt data availability	Applications and users can continue without interruption
Client-Side Data Caching Keeps recently written or read data blocks in client memory	Improves performance by eliminating repetitive processing of requests for frequently accessed files; hides write latency by using "write behind"
Large-Blocksize Options Data blocks of 16K, 64K, 256K, 512K or 1,024K can be used for different applications	Large blocks allow more data to be written or read in a single I/O operation. Enables more efficient use of disk bandwidth and space, particularly important for RAID devices
Access Pattern Recognition and Deep Pre-Fetch Detects access patterns as sequential, random, fuzzy sequential or strided, pre-fetches striped data in parallel and buffers it at the application node	Maximizes bandwidth by enabling disks to read and write simultaneously
Data Protection Both user data and metadata (file system transaction data) can be replicated in the GPFS installation	Ensures reliable access to data and metadata

GPFS Architecture

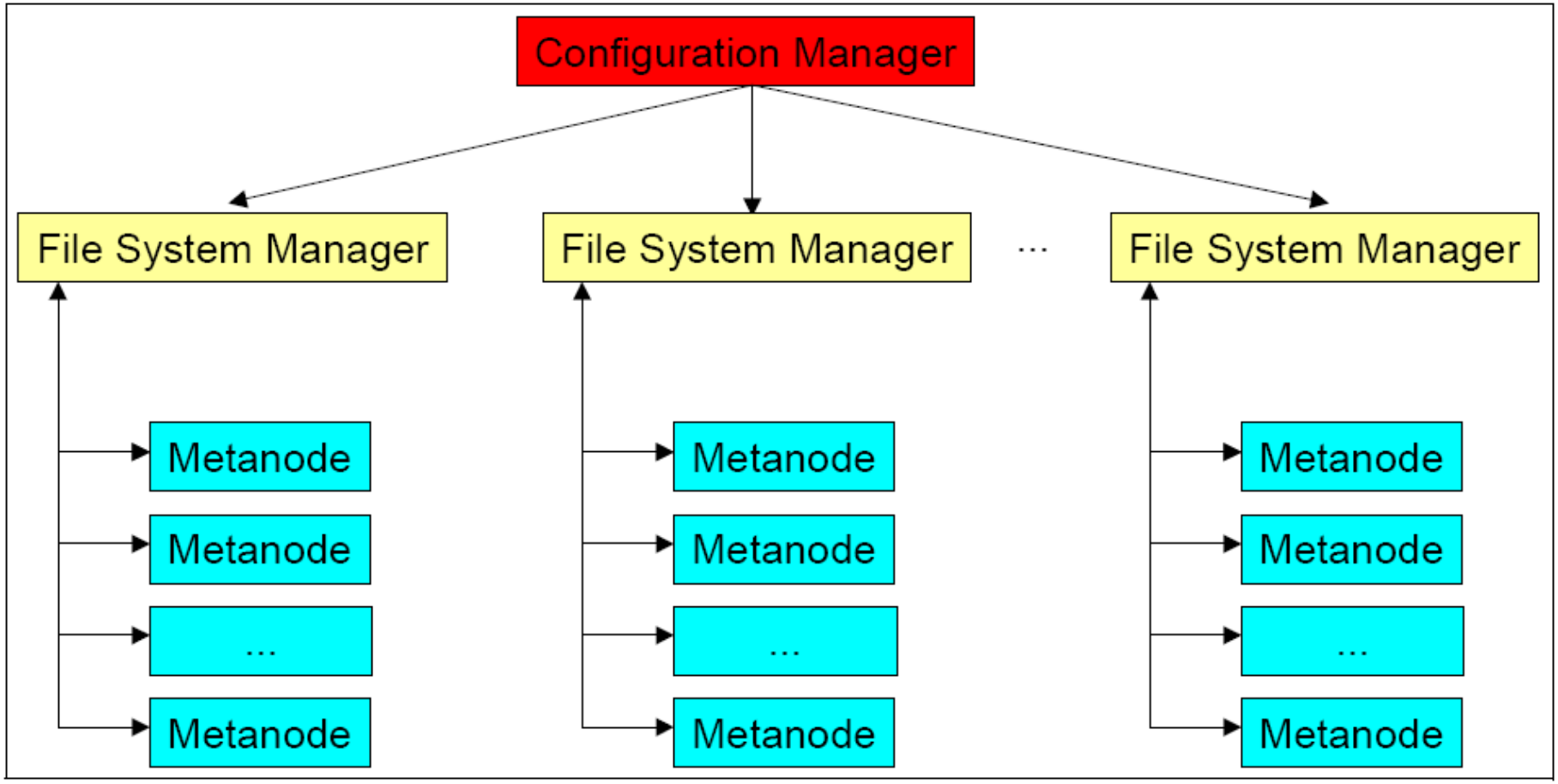


Source: http://www.redbooks.ibm.com/redbooks/pdfs/sg245610.pdf

Components Internal to GPFS Daemon

- Configuration Manager (CfgMgr)
  - Selects the node acting as Stripe Group Manager for each file system
  - Checks for the quorum of nodes required for the file system usage to continue
  - Appoints successor node in case of failure
  - Initiates and controls recovery procedure
- Stripe Group Manager (FSMgr, aka File System Manager)
  - Strictly one per each GPFS file system
  - Maintains availability information of disks comprising the file system (physical storage)
  - Processes modifications (disk removals and additions)
  - Repairs file system and coordinates data migration when required
- Metanode
  - Manages metadata (directory block updates)
  - Its location may change (e.g. a node obtaining access to the file may become the metanode)
- Token Manager Server
  - Synchronizes concurrent access to files and ensures consistency among caches
  - Manages *tokens*, or per-object locks
    - Mediates token migration when another node requests token conflicting with the existing token (token stealing)
  - Always located on the same node as Stripe Group Manager

GPFS Management Functions & Their Dependencies

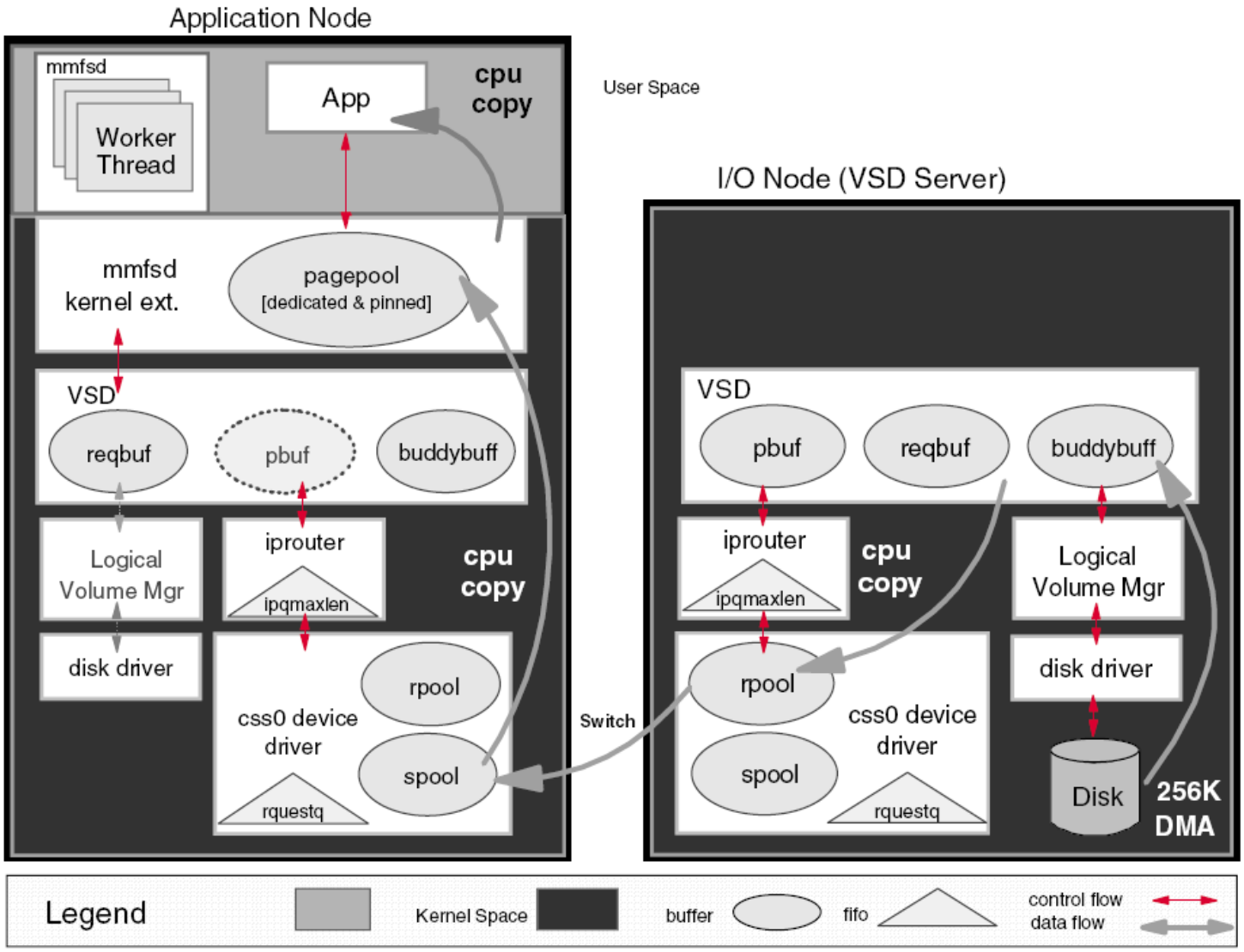


Source: http://www.redbooks.ibm.com/redbooks/pdfs/sg246700.pdf

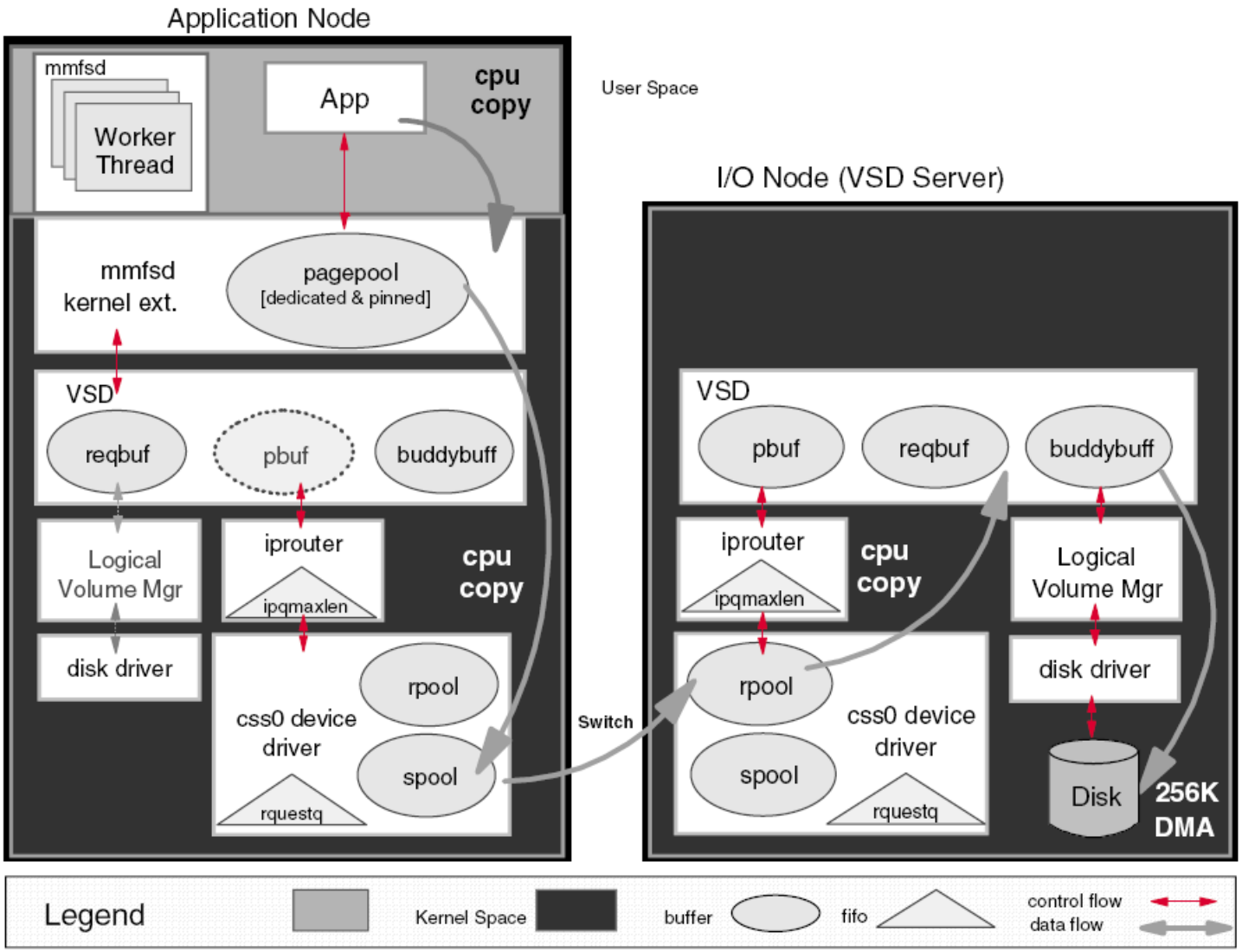
Components External to GPFS Daemon

- Virtual Shared Disk (VSD, aka *logical volume*)
  - Enables nodes in one SP system partition to share disks with the other nodes in the same system partition
  - VSD node can be a client, a server (owning a number of VSDs, and performing data reads and writes requested by client nodes), or both at the same time
- Recoverable Virtual Shared Disk (RVSD)
  - Used together with VSD to provide high availability against node failures reported by Group Services
  - Runs recovery scripts and notifies client applications
- Switch (interconnect) Subsystem
  - Starts switch daemon, responsible for initializing and monitoring the switch
  - Discovers and reacts to topology changes; reports and services status/error packets
- Group Services
  - Fault-tolerant, highly available and partition-sensitive service monitoring and coordinating changes related to another subsystem operating in the partition
  - Operates on each node within the partition, plus the control workstation for the partition
- System Data Repository (RSD)
  - Location where the configuration data are stored

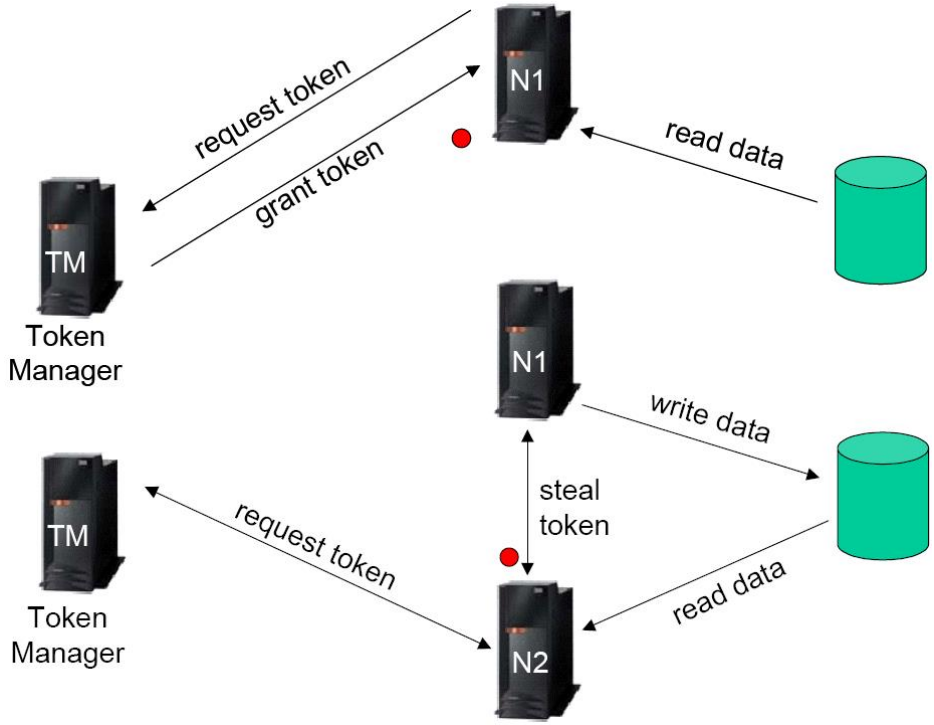
Read Operation Flow in GPFS



Write Operation Flow in GPFS

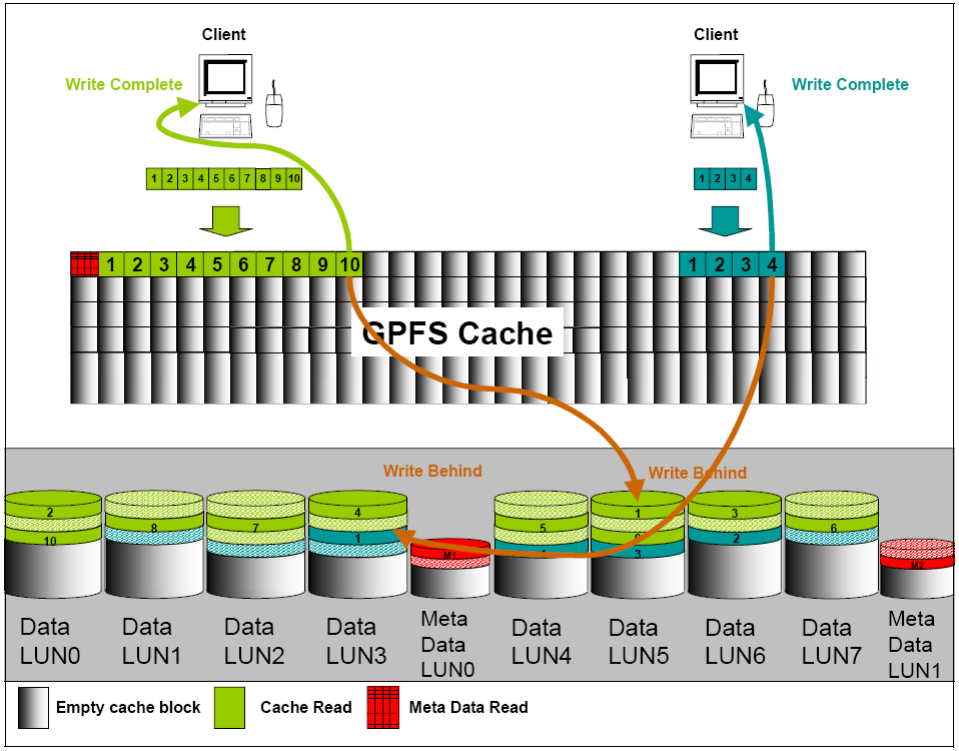


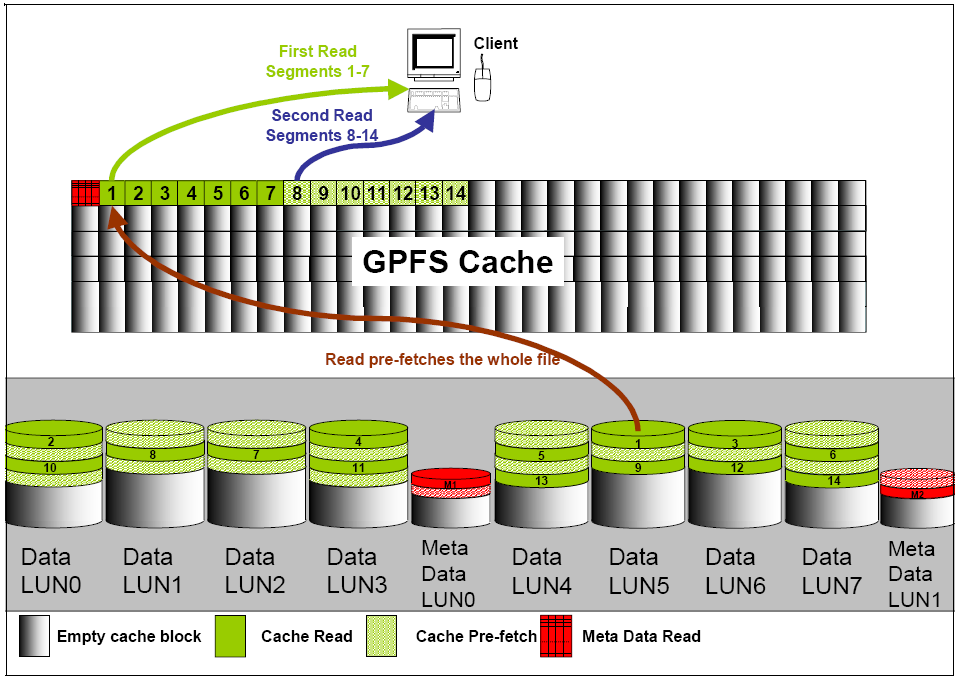
Token Management in GPFS



- First lock request for an object requires a message from node N1 to the token manager
- Token server grants token to N1 (subsequent lock requests can be granted locally)
- Node N2 requests token for the same file (lock conflict)
- Token server detects conflicting lock request and revokes token from N1
- If N1 was writing to file, the data is flushed to disk before the revocation is complete
- Node N2 gets the token from N1

GPFS Write-behind and Prefetch

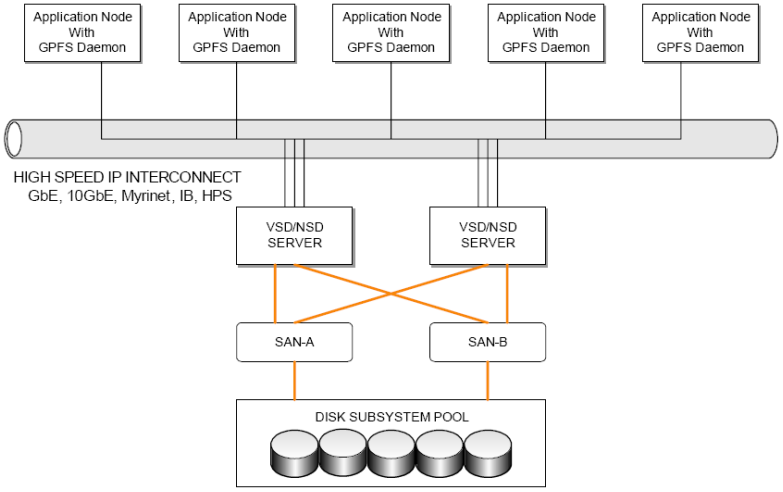




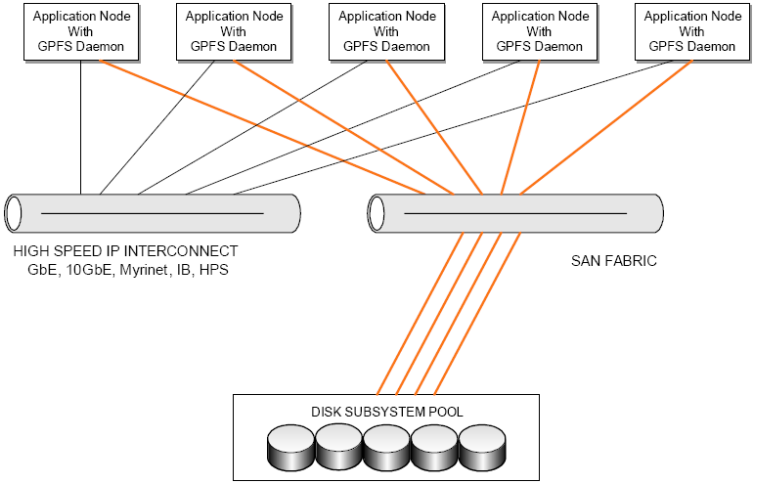
- As soon as application's write buffer is copied into the local pagepool, the write operation is complete from client's perspective
- GPFS daemon schedules a worker thread to finalize the request by issuing I/O calls to the device driver

- GPFS estimates the number of blocks to read ahead based on disk performance and rate at which application is reading the data
- Additional prefetch requests are processed asynchronously with the completion of the current read

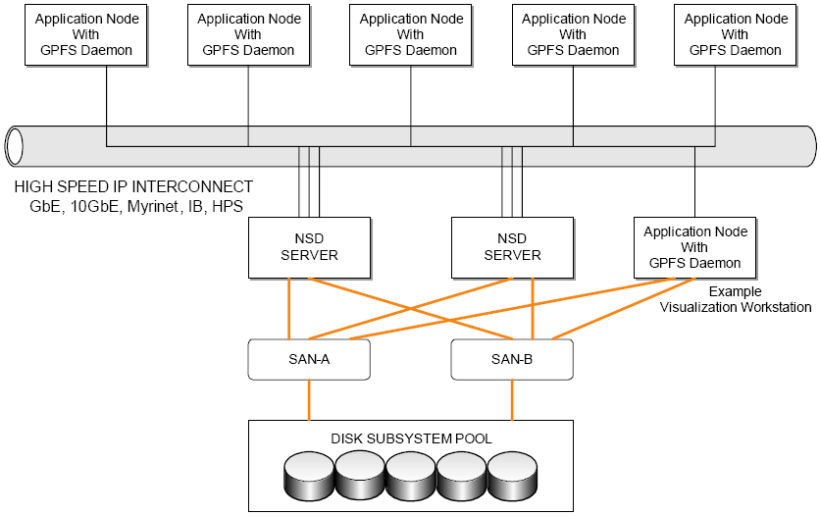
Some GPFS Cluster Models



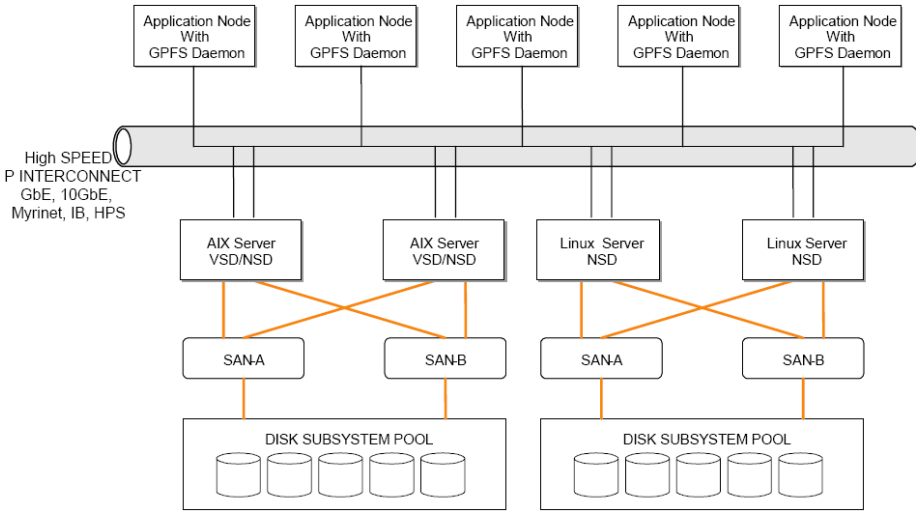
Network Shared Disk (NSD) with dedicated server model



Direct attached model



Mixed (NSD and direct attached) model



Joined (AIX and Linux) model

Comparison of NFS and GPFS

File-System Features	NFS	GPFS
Introduced:	1985	1998
Original vendor:	Sun	IBM
Example at LC:	/nfs/tmp*n*	/p/g*x*1
Primary role:	Share files among machines	Fast parallel I/O for large files
Easy to scale?	No	Yes
Network needed:	Any TCP/IP network	IBM SP "switch"
Access control method:	UNIX permission bits (CHMOD)	UNIX permission bits (CHMOD)
Block size:	256 byte	512 Kbyte (White)
Stripe width:	Depends on RAID	256 Kbyte
Maximum file size:	2 Gbyte (longer with v3)	26 Gbyte
File consistency:		
.....uses client buffering?	Yes	Yes (see diagram)
.....uses server buffering?		Yes (see diagram)
.....uses locking?	No	Yes (token passing)
.....lock granularity?		Byte range
.....lock managed by?		Requesting compute node
Purged at LC?	Home, No; Tmp, Yes	Yes
Supports file quotas?	Yes	No

Comparison of GPFS to Other File Systems

Feature	JFS	NFS	AFS	DFS	PIOFS	GPFS
Scalability	N	N	N	Y*	Y	Y
Parallelism	N	N	N	N	Y	Y
Cross-platform	N	Y	Y	Y	N	N
Replication	N	N	Y	Y*	N	Y
Large files/file-systems	N	By partitioning only			Y	Y
Security	Y	N	Y	Y	Y	Y
Failure Recovery	N/A	N	Y	Y	N	Y
Centralized administration	N/A	N	N	N	Y	Y
Byte range locking	Y	Y	N	Y	N	Y
Physical file system	Y	N	N	N	N	Y

* Read via replicas

Topics

- Introduction
- RAID
- Distributed File Systems (NFS)
- Parallel File Systems: Introduction
- Parallel File Systems (PVFS2)
- Parallel File Systems (Lustre)
- Additional Parallel File Systems (GPFS)
- POSIX I/O API

IO Problem of the day

```
#include <stdio.h>

int main()
{
    int a = 0, b = 0;
    char buf[10];
    scanf ("%d%d", a, b);
    sprintf (buf, "%d %d");
    puts ("you entered: ");
    puts (buf);
}
```

If the user entered 3 and 17, what‘s the generated output?

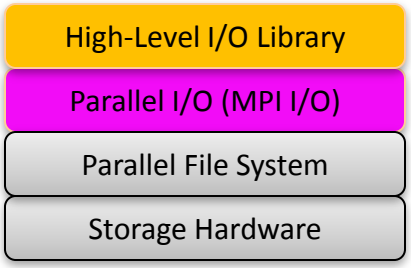
IO Problem of the day

```
#include <stdio.h>

int main()
{
    int a = 0, b = 0;
    char buf[42];           // max. 20 digits in 64bit int
    scanf ("%d%d", &a, &b);
    snprintf (buf, 42, "%d %d", a, b);
    puts ("you entered: ");
    puts (buf);
}
```

Parallel I/O: Library Layers (Review)

- Lower level interfaces may be provided by the file system for higher-performance access
- Above the parallel file systems are the parallel I/O layers provided in the form of libraries such as MPI-IO
- The parallel I/O layer provides a low level interface and operations such as collective I/O
- Scientific applications work with structured data for which a higher level API written on top of MPI-IO such as HDF5 or parallel netCDF are used
- HDF5 and parallel netCDF allow the scientists to represent the data sets in terms closer to those used in their applications, and in a portable manner



POSIX File Access API

- Widespread standard
- Available on any UNIX-compliant platform
  - IBM AIX, HP HP-UX, SGI Irix, Sun Solaris, BSDi BSD/OS, Mac OS X, Linux, FreeBSD, OpenBSD, NetBSD, BeOS, and many others
  - Also: Windows NT, XP, Server 2003, Vista, Windows 7 (through C runtime libraries)
- Simple interface: six functions from POSIX.1 (core services) provide practically all necessary I/O functionality
  - File open
  - File close
  - File data read
  - File data write
  - Flush buffer to disk
  - Adjust file pointer (seek)
- Two interface variants, provide roughly equivalent functionality
  - Low-level file interface (file handles are integer *descriptors*)
  - C stream interface (streams are represented by FILE structure; function names prefixed with “f”)
- But: no parallel I/O support

File Open

Function: open()

int open(const char **path*, int *flags*);
int open(const char **path*, int *flags*,
mode_t *mode*);

Description:

Opens the file identified by *path*, returning a non-negative descriptor on success. The *flags* argument must contain one of the following access modes O_RDONLY, O_WRONLY, or O_RDWR; additional file creation flags may be bitwise or'd: O_CREAT, O_EXCL, and O_TRUNC. The optional *mode* specifies access permissions when the file is created.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
...
/* create empty writable file with default access
permissions, storing its descriptor in fd */
int fd = open("test", O_WRONLY|O_CREAT|O_TRUNC);
if (fd < 0) {/* handle error here */}
```

Function: fopen()

FILE *fopen(const char **path*,
const char **mode*);

Description:

Opens the file identified by *path*, associating a stream with it and returning non-zero pointer if successful. The *mode* string is one of: "r" (reading), "r+" (reading and writing), "w" (creating or truncating an existing file for writing), "w+" (reading and writing, with creation or truncating), "a" (appending: writing at the end of file), or "a+" (reading and appending, with creation if the file doesn't exist).

```
#include <stdio.h>
...
/* replicate open() example on the left, storing
file handle in f */
FILE *f = fopen("test", "w");
if (f == NULL) {/* handle error here */}
...
```

File Close

Function: close()

int close(int *fd*);

Description:

Closes file descriptor *fd* making it available for reuse, returning zero on success. OS resources associated with the open file descriptor are freed. Note that a successful close does not guarantee that file data have been saved to the disk.

```
#include <unistd.h>
...
/* open a file */
int rc;
int fd = open(...);
...
/* file is accessed here */
...
rc = close(fd);
if (rc != 0) {/* handle error here */}
```

Function: fclose()

int fclose(FILE **fp*);

Description:

Flushes the stream pointed to by *fp* and closes the underlying file descriptor returning zero on success. Note that buffer flush affects only data implicitly managed by the C library, not the kernel buffers.

```
#include <stdio.h>
...
/* open a file */
int rc;
FILE *f = fopen(...);
...
/* file is accessed here */
...
rc = fclose(f);
if (rc != 0) {/* handle error here */}
```

File Read

Function: read()

int read(int *fd*, void **buf*, size_t *count*);

Description:

Attempts to read at most *count* sequential bytes from file descriptor *fd* into the buffer starting at *buf*. Returns the number of bytes read (zero indicates end of file). On error, -1 is returned.

```
#include <unistd.h>
...
int bytes;
char buf[100];
/* open an existing file for reading */
int fd = open(...);
...
bytes = read(fd, buf, 100);
if (bytes < 100) {/* handle EOF or error here */}
...
```

Function: fread()

size_t fread(void **ptr*, size_t *size*, size_t *n*, FILE **stream*);

Description:

Reads *n* sequential elements of data, each *size* bytes long from the stream identified by **stream*, storing them in location pointed to by *ptr*. Returns the number of items (not bytes!) successfully read. On error, or if end of file is reached, the return value is less than *n*.

```
#include <stdio.h>
...
size_t items;
char buf[100];
/* open an existing file for reading */
FILE *f = fopen(...);
...
items = fread(buf, 1, 100, f);
if (items < 100) {/* handle EOF or error here */}
...
```

File Write

Function: write()

int write(int *fd*, void **buf*, size_t *count*);

Description:

Writes sequentially at most *count* bytes from the buffer pointed to by *buf* to the file identified by descriptor *fd*. Returns the number of bytes written; if less than *count*, it means that either the underlying device is out of space, or an interrupt occurred. On error, -1 is returned.

```
#include <unistd.h>
...
int bytes;
char buf[100];
/* open a file for writing or appending */
int fd = open(...);
...
/* initialize buffer data */
...
bytes = write(fd, buf, 100);
if (bytes < 100) {/* handle short write */}
...
```

Function: fwrite()

size_t fwrite(void **ptr*, size_t *size*, size_t *n*, FILE **stream*);

Description:

Writes sequentially *n* elements of data, each *size* bytes long to the stream identified by **stream* from location pointed to by *ptr*. Returns the number of items successfully written. On error, or if end of file is reached, the return value is less than *n*.

```
#include <stdio.h>
...
size_t items;
char buf[100];
/* open a file for writing or appending */
FILE *f = fopen(...);
...
/* initialize buffer data */
...
items = fwrite(buf, 1, 100, f);
if (items < 100) {/* handle short write */}
...
```

File Seek

Function: lseek()

off_t lseek(int *fd*, off_t *offs*, int *whence*);

Description:

Adjusts the offset of the open file associated with the descriptor *fd* to the argument *offs* in accordance to *whence*, which may assume the following values: SEEK_SET (sets offset to *offs* bytes), SEEK_CUR (offset is set to the current location plus *offs*), or SEEK_END (sets offset to the sizeof file plus *offs*). Returns the resultant offset value measured from the beginning of file, or (off_t) -1 on error.

```
#include <sys/types.h>
#include <unistd.h>
...
/* open file for read/write access */
int fd = open("/tmp/myfile", O_RDWR);
...
/* write some file data */
...
/* "rewind" to the beginning of file to check
   the written data  */
lseek(fd, 0, SEEK_SET);
/* start reading... */
```

Function: fseek()

int fseek(FILE **stream*, long *offs*, int *whence*);

Description:

Sets the file position indicator for the stream identified by *stream*. The meaning of the *offset* and *whence* arguments is the same as for lseek(). Returns the current file offset in bytes, or -1 on error.

```
#include <stdio.h>
...
/* open file for reading and writing */
FILE *f = fopen("/tmp/myfile", "r+");
...
/* to start appending data at the end of file: */
fseek(f, 0, SEEK_END);
fwrite(...);
...
```

File Data Flushing

Function: fsync()

int fsync(int *fd*);

Description:

Transfers all modified in-core data and metadata (such as file size) of the file referred to by descriptor *fd* to permanent storage device. The call blocks until the transfer is complete. Returns zero on success, -1 on error.

```
#include <unistd.h>
...
/* open file for writing */
int fd = open("checkpt.dat", O_WRONLY|O_CREAT);
...
/* write checkpoint data */
...
/* make sure data are flushed to disk before
   starting the next iteration */
fsync(fd);
...
```

Function: fflush()

int fflush(FILE **stream*);

Description:

Forces write of all user-space buffered data of the output stream identified by **stream*, or all open output streams if *stream* is NULL. Returns zero on success, or EOF on error.

```
#include <stdio.h>
...
/* open file for appending */
FILE *f = fopen("/var/log/app.log", "a");
...
/* special event happened: output a message */
fprintf(f, "driver initialization failed");
/* make sure message reaches at least kernel
   buffers before application crashes */
fflush(f);
...
```

Problems with POSIX File I/O

- Too simplistic interface
  - Operates on anonymous sequences of bytes
  - No preservation of type or information structure
  - Cumbersome access to optimized/additional features (fcntl, ioctl)
  - Designed for sequential I/O (even regularly strided accesses require multiple calls and may suffer from poor performance)
- Portability issues
  - Must use specialized reader/writer created for a particular application
  - Compatibility checks dependent on application developers (possibility of undetected failures)
  - No generic utilities to parse and interpret the contents of saved files
  - Cross platform endianness and type representation problem if saving in binary mode
  - Significant waste of storage space if text mode is used (for portability or readability of transferred data)
- Permit access only to locally mounted storage, or remote storage via NFS (which has its share of problems)
- Parallel and concurrent access issues
  - Lack of synchronization when accessing shared files from multiple nodes
  - Atomic access to shared files may not be enforceable, has unclear semantics, or has to rely on the programmer for synchronization
  - Uncoordinated access of I/O devices shared by multiple nodes may result in poor performance (bottlenecks)
  - Additional performance loss due to suboptimal bulk data movement (e.g., no collective I/O)
  - On the other hand, without sharing, the management of individual files (i.e. with at least one data file per I/O node) is complicated and tedious