Outline

• Data intensive computation
  – New Persistent Device Technologies
  – High Performance Storage Systems

• Scalable and parallel file systems. Lustre, GPFS, PVFS, HDFS, Ceph, GlusterFS.

• Parallel I/O libraries. MPI-IO, HDF5, NetCDF
Data intensive computation

The Digital Universe 2009 to 2020

Growing by a factor of 44

2009: 0.8 Zettabytes

2020: 35.2 Zettabytes

Equivalent to a stack of DVDs in 2009 reaching to the moon and back, now reaching halfway to Mars by 2020

Source: IDC Digital Universe Study.
The Digital Universe Gap

- Data is outpacing storage: The world’s amount of available storage capacity across all media types is growing slower than the digital universe.
- In 2013, the available storage capacity could hold just 33% of the digital universe.
- By 2020, it will be able to store less than 15%.
- Fortunately, most of the world’s data is transient (e.g. Netflix or Hulu stream, Xbox ONE game interactions, Digital TV.) and requires no storage.

Source: EMC Digital Universe study

Source: EMC Digital Universe Study, sponsored by EMC, June 2011
Other interesting facts and predictions

• Emerging markets are producing more data: Currently, 60% of data in the digital universe is attributed to mature markets such as Germany, Japan, and the United States, but by 2020, the percentage will flip, and emerging markets including Brazil, China, India, Mexico and Russia will account for the majority of.

• Data touched by the cloud will double: In 2013, less than 20% of the data in the digital universe was “touched” by the cloud. By 2020, that percentage will double to 40%.

• Consumers create data but enterprises are responsible for it: Two-thirds of the digital universe bits are created or captured by consumers and workers, yet enterprises have liability or responsibility for 85% of the digital universe.

Source: EMC Digital Universe study
The Big Problem

• Moving this amount of data from storage devices to processors and processing them will require a degree of efficiency that is not possible today

• Storage persistence necessary for dealing with failures

• Some initiatives:
  – New Persistent Device Technologies
  – High Performance Storage Systems
  – New kinds of applications (for instance, NoSQL data stores) [Out of scope]
New Persistent Device Technologies

- Solid Storage Drives (SSDs) are replacing old magnetic disks (HDD: Hard Disk Drives)
  - “Tape is dead, disk is tape, flash is disk, RAM locality is king” (Jim Gray, Dec 2006)
  - Better performance (bandwidth, latency)
- SSDs are based on non-volatile memories (NVM)
- Storage chips of a SSD:
  - Flash chips
  - Phase-Change Memory (PCM) chips
- Performance and latency:
  - (Hundreds of thousands accesses per second, latency of tens of microseconds)
  - Hundred times better performance
  - Ten times more expensive
## SSDs vs HDDs

<table>
<thead>
<tr>
<th>Features</th>
<th>SSDs vs HDDs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>Lower latency</td>
</tr>
<tr>
<td></td>
<td>Higher bandwidth</td>
</tr>
<tr>
<td></td>
<td>Supports more IOPs</td>
</tr>
<tr>
<td><strong>Energy consumption</strong></td>
<td>SSDs don’t need rotation</td>
</tr>
<tr>
<td></td>
<td>Less power</td>
</tr>
<tr>
<td></td>
<td>No heat, no noise</td>
</tr>
<tr>
<td><strong>Fragmentation</strong></td>
<td>No fragmentation in the case of SSDs</td>
</tr>
<tr>
<td><strong>Components</strong></td>
<td>SSDs don’t have moving parts</td>
</tr>
<tr>
<td><strong>Cost</strong></td>
<td>SSDs are more expensive</td>
</tr>
<tr>
<td><strong>Lifetime</strong></td>
<td>Limited lifetime in the case of SSDs</td>
</tr>
<tr>
<td></td>
<td>They support a finite number of erase cycles (flash memory wear)</td>
</tr>
</tbody>
</table>
Flash chips

• Flash chip: A set of flash cells
  – Organized by pages (512 to 4096 bytes per page)
  – Blocks (64 to 256 pages per block)
  – Sometimes arranged in multiple planes (parallelism across planes)

• Operations on flash chips:
  – Read
  – Write/program
  – Erase

• Rules:
  – Reads and writes made at page level
  – A block is erased before any of the pages it contains can be overwritten
  – Writes are sequential within a block
  – Flash chips support a limited number of erase cycles.
Flash-based SSDs

- Composed by flash chips wired in parallel to the SSD controller though multiple channels (parallelism)
- The SSD controller includes the Flash Translation Layer (FTL)
- The FTL implements Copy on Write to provide the illusion of “in-place writes” and hide the cost of block erasures.

Source: IBM

FTL is in charge of mapping I/O operations into flash chip operations
Block manager

- FTL: physical address space -> logical address space (virtualization)
- A write in a Logical BlockAddress (LBA) involves a different Flash memory page to be written
- Use of a mapping table for the translation
  - Logical pages -> Physical pages
- The target Flash memory page is selected from a pool of free pages
- When a logical page is written, the FTL writes the data to a new physical page and updates the mapping.
- Copy on Write
Garbage collection

- The FTL further maintains a list of free blocks.
- Copy on Write can provoke the presence of obsolete versions of logical pages.
- A garbage collection routine:
  - selects a block
  - copies valid pages out of the block
  - erases it
  - adds it to the free-blocks queue
- Garbage collection techniques can also be used to balance the wearing down of different blocks on the flash (wear leveling)
Wear leveling

• Technique used for prolonging the life of flash-based SSDs
• Performed in the background (transparent to the system)
• Three types:
  – No wear leveling
  – Dynamic wear leveling
  – Static wear leveling
No wear leveling

• A write to a previously written block:
  – First read
  – Erased
  – Modified
  – Re-written to the same location
• Time-consuming
• Highly written locations will wear out quickly
• If a block reach its end of life, the device is not operable anymore
Dynamic wear leveling

- All new data are written to free data blocks (blocks that do not contain user data)
- Selection of the new free data block: based on the number of program/erase cycles of the block.
- The controller updates its internal logical to physical mapping table to point to the new physical block location.
- The data block with the old data is erased and becomes a free block.
- Flash memory blocks that never get replacement data would sustain no additional wear (only dynamic data is recycled).
- Longer life than no wear leveling, but some blocks are still remaining as active when the device is not longer operable.
- Typical use: USB Flash Drives
  - the design is less complex than in the case of static wear leveling.
Static wear leveling

• Also called global wear leveling
• It behaves the same as dynamic wear leveling, except the static blocks that do not change are periodically moved so that they can be used by other data
  – by including the static data blocks in the program/erase pool
• In this way, the device is able to operate until most of the blocks are near the end of their life
• More complex design than dynamic wear leveling
• Typical use: SSDs
Phase-Change Memory

• Some disadvantages of flash chips:
  – They don’t support in-place writes to data
  – Erases only can be performed on a block of pages

• Unlike flash-based memory, PCM has the following features:
  – It is byte-addressable
  – It supports in-place writes
Storage-Class Memory (SCM)

- A system level abstraction for Phase-Change Memory
- It is PCM, but not only PCM:
  - FeRAM (Ferroelectric RAM)
  - MRAM (Magnetic RAM)
  - RRAM (Resistive RAM)
- The goal of SCM is to delete the gap between Memory (volatile, fast, expensive) and Storage (non-volatile, slow, cheap).
SCM

Evolution of the Memory – Storage Stack

1980
- Logic: CPU
- Memory: RAM
- Active Storage: DISK
- Archival: TAPE

2008
- Logic: CPU
- Memory: RAM
- Active Storage: DISK, FLASH SSD
- Archival: TAPE

2015
- Logic: CPU
- Memory: RAM, STORAGE CLASS MEMORY
- Active Storage: DISK
- Archival: TAPE

Source: IBM
High Performance Storage Systems

• Data intensive applications
  – They require a higher number of I/O operations

• Increasing disk capacity
  – Not proportional to their performance: bandwidth and latency

• I/O crisis: unbalance between computing and I/O performance
  – Higher in multicore architectures
  – A higher impact for data intensive applications

• Solution: parallelism

• Parallel I/O
  – Data stripping in multiple storage devices
  – Parallel access
  – Due to the latency, the larger access size the better
    • Poor performance in parallel small-size accesses
Parallel I/O hierarchy

Parallel applications

Parallel libraries

Parallel file systems/Operating systems

Parallel devices
Parallel Devices

- **DAS: Direct-attached storage**
  - “Classic” solution
- **NAS: Network-attached storage**
  - A node managing a set of disks
- **SAN: Storage Area Networks**
  - Network focused on storage
    - Network disks not linked to any node
  - Direct connection node-devices
  - Direct connection device-device
    - Useful for backups, replication
  - SANs include *hubs*, *switches*, etc.
    - Usual technology: *Fibre Channel*
Direct-attached storage (DAS)
Network-attached storage (NAS)
**Storage Area Network (SAN)**

Diagram showing network connections and storage devices.
RAID systems

Original idea: RAID (Redundant Array of Inexpensive Disks).

RAID:

- Goal: Using several disks as a single device with the aim of increasing the storage system bandwidth
- Problem: The more number of disks the more failure probability
  - Solution: Providing fault tolerance
RAID 0

Pros:
- High bandwidth and capacity

Cons:
- No fault tolerance

Source: Seagate Technology
RAID 1

Pros:
- Fault tolerance
- Read optimizations are feasible

Cons:
- Wasted space
- Less parallelism than in RAID 0

Source: Seagate Technology
RAID 5

Pros:
- High bandwidth both in read and write operations
- Fault tolerance (only a fault in a single disk is allowed)

Cons:
- Small writes problem

Source: Seagate Technology
RAID 6

Pros:
- High bandwidth both in read and write operations
- It supports up to two disk failures with no data loss

Cons:
- RAID 6 synchronizing from a failed disk is slower than RAID 5
- A minimum of four disks is required to create a RAID 6 volume (wasted space for providing fault tolerance)

Source: Seagate Technology
RAID 10

Pros:
- It combines the protection of RAID 1 with the performance of RAID 0
- Exceptional data protection, allowing for two disks to fail across two RAID 1 segments
- Higher performance

Cons:
- 50% waste of space

Source: Seagate Technology
Small-write problem

• Steps:
  – Read old data, old parity
  – Compute new parity
  – Write new data and new parity ⇒ small-write problem (4 I/Os per data block)

• It is also a problem in Flash-based SSDs:
  – Slow writes and no in-place writes to Flashes can degrade the performance of these devices
  – Particularly in metadata and log-file updates
Small-write “solutions” to RAID

- Other RAID hierarchies (e.g., HP AutoRAID)
- Logs (e.g. xFS, Zebra)
- Parity logging
- Flash-based small-write solutions
Two RAID levels:

- **RAID 1:**
  - Fast reads and writes ☺
  - Fault tolerance ☻
  - It requires large storage space ☹

- **RAID 5:**
  - Fast read operations ☻
  - Fault tolerance ☻
  - Minimum storage space ☻
  - Slow small writes ☹

“The HP AutoRAID hierarchical storage system”

John Wilkes, Richard Golding, Carl Staelin, Tim Sullivan

ACM Transactions on Computer Systems (TOCS)
Volume 14, Issue 1, February 1996
Pages: 108 – 136
Parity Logging

- Use of an extra disk as log
  - Data writes equal to RAID 4/5
  - Parity writing in the log disk (sequential write)
  - Updating real parity when the log disk is full or when a threshold has been achieved

Diagram:
- Data disks
- Parity disk
- Log disk
- XOR modified data
Small-write “solutions” to Flash-based SSDs

- To combine SCM and Flash:
  - MiNVFS (Metadata in Non-Volatile RAM File System)
  - FRASH provides byte-addressability to the file system object and metadata
    - J. Jung et al. “FRASH: Exploiting storage class memory in hybrid file system for hierarchical storage”, ACM Transactions on Storage, 2010
  - Use SCM as a non-volatile cache for Flash
Using parity per stripe, distributed among the set of disks

Different organizations:
- Right-asymmetric
- Right-symmetric
- Left-asymmetric
- Left-symmetric
- Flat-left-symmetric
Right-asymmetric vs right-symmetric

Right-asymmetric

Problems with large reads. For instance, 6-10, does not use 5 disks

Right-symmetric

Higher problems with read. For instance, 1-4, does not use 4 disks
Worse than right-asymmetric
Left-asymmetric vs left-symmetric

Problems with reads again
For instance, 1-5, does not use 5 disks

Whatever we request, the maximum number of disks are used
Flat-left-asymmetric

It works very well with reads. Regarding to writes, if you use mechanisms for avoiding small-writes problems, it goes well, although worse than reads. It is used in systems with higher percentage of reads.
• Data intensive computation
• Scalable and parallel file systems. Lustre, GPFS, PVFS, HDFS, Ceph, GlusterFS
• Parallel I/O libraries. MPI-IO, HDF5, NetCDF
Scalable and Parallel File Systems

• Why not distributed file systems, as NFS?
• Problem: every file in a single server
  – No parallelism, no scalability
  – Bottleneck and single point of failure
  – No compatibility with SAN

• A parallel file system:
  – Use of stripping (similar to RAID 0, but software and several nodes)
  – Also called *Shared disk file systems*. Two levels:
    • Lower level: Distributed Storage Service, provided by SAN or I/O Nodes
    • Higher level: File system in the Computing Node

• Examples: Lustre, GPFS, PVFS, HDFS, Ceph, GlusterFS
Lustre

- Object-based storage, which operates at object level
- Traditional storage operates at block-level

**Block-based Storage**
- Operations: Read block, Write block
- Granularity: Block

**Object-based Storage**
- Operations: Create object, Delete object, Read object, Write object
- Granularity: Byte
Block-based vs Object-based Storage
Lustre

- Separation of metadata and data to increase scalability
- Scalable data serving (parallel data striping)
- Scalable metadata management
- Components:
  - Clients
    - Access to file system
    - Usually, they are the compute server running Lustre client software, allowing them to mount the Lustre file system
  - OSS
    - Object Storage Servers
    - Access to Data by means of OST
  - MDS
    - Metadata Server
    - Access to Metadata by means of MDT
  - Other components
    - LDAP
    - Kerberos
    - Routers etc.
Clients

LDAP Server

configuration information, network connection details, & security management

Meta-data Server (MDS)

directory operations, meta-data, & concurrency

Clients

file I/O & file locking

recovery, file status, & file creation

Object Storage Targets (OST)
**Lustre Client**

- File system
- LOV
- MDC
  - OSC 1
  - OSC 2

**Meta-data Server**

- MDS

Inode A \{(O1, obj1), (O3, obj2)\}

File open request

File meta-data

Write (obj 1)

Write (obj 2)
Failover

• MDT failover:
  – Two MDSs are configured to serve the same MDT
  – Only one MDS node serves an MDT at a time

• OST failover:
  – Multiple OSSs are configured to serve the same OST
  – Only one OSS node serves an OST at a time
  – An OST can be moved between OSS nodes with access to the same storage device by using umount/mount commands

Source: wiki.lustre.org
General Parallel File System (GPFS), IBM

- Oriented to large scale clusters
- *Shared disk file system* (based on SAN or I/O nodes)
- Heterogeneous systems (AIX, Linux, Windows)
- Concurrent reads and writes
- Parallel access to both data and metadata
- POSIX interface (except *atime*)
- Client caching enabled by distributed locking
- Fault tolerance in disks, nodes and communication via logging, replication and RAID support
Shared disk file system Architecture
GPFS Architecture

- Three kinds of nodes:
  - File system nodes:
    - Run user programs
    - Read/write data from/to storage nodes
    - Implementation of the Virtual File System interface
  - Manager nodes:
    - Management activities (Global lock manager, file metadata manager, ...)
  - Storage nodes:
    - Implementation of the block I/O interface
    - data and metadata striped across multiple disks on multiple storage nodes
Distributed locking management

• A single Token Manager (TM)
  – Scalability issue and single point of failure
• Manages read/write tokens for several kinds of objects
  – Byte range, inodes, bitmaps, ...
• Use of the token:
  – Control of the parallel access to objects
  – Control of the object’s cache
    • Valid cache in case of having a token; dump updates when the token is return
• An operation in a Computing Node (CN) needs token for an object
  – It is requested to TM and kept for further operations in the CN
    • Until a conflictive operation in other CN provokes its revocation
• Scalability TM: minimize the intervention of the TM
  – Ask for multiple tokens in a single request
  – CN requiring token asks for revocation directly to CNs
  – A new file reuses the inode, keeping tokens associated to it
Coherence in data access

- Protocol based on tokens associated to byte range
- Optimization in the management of tokens
  1. A process read/write files using N calls: 1 single token
  2. M proc. write files (1/M each one) with N calls/pr.: M tokens
- Token request includes two ranges:
  - Required range: specified in read/write operations
  - Desired range: the range of bytes that could be accessed in the future
    - If sequential access, until infinite
- Request resolution:
  - Revocation of tokens which are in conflict with required range
  - Resolution: Desired range $\subset$ not in conflict
1. In first read/write for F, desired range $[0, \infty)$
   - If not any other client accesses F, no more token request
   - Thus, the process only asks for a token

2. Application with M processes creates a file: $1/M$ each one
   - P1: $d = \text{open}(F); \text{write}(d, b, TAM\_BLOQ)$
     • P1 gets token $[0, \infty)$
   - P2: $d = \text{open}(F); \text{lseek}(d, SEEK\_SET, 1/N); \text{write}(d, b, TAM\_BLOQ)$
     • P2 gets token $[1/N, \infty)$; P1 adjusts token $[0, 1/N-1)$
   - P3: $d = \text{open}(F); \text{lseek}(d, SEEK\_SET, 2/N); \text{write}(d, b, TAM\_BLOQ)$
     • P3 gets token $[2/N, \infty)$; P2 adjusts $[1/N, 2/N-1]$; P1 token $[0, 1/N-1)$
   - And so on
   - Thus, every process only asks for a token
Coherence in metadata access

- Concurrent updates to file metadata
  - Direct (chmod)
  - Indirect: write → modification date, size and block pointers
- Use of an exclusive access token per inode is not efficient
- Idea: Update of an inode in parallel and mixture of changes
- Solution: exclusive and shared write tokens
  - Write operations use shared write tokens
    - Each node modifies its inode copy (update date, size and pointers)
  - Certain operations require exclusive write tokens (stat, utime, ftruncate, ..)
    - Shared write tokens are revoked
    - Every CN involved dumps their copy of the inode and all of them are merged
    - Merge operation? File metanode
File metanode

- CN chosen like metanode (MN) of a file
  - The only node that can read/write the inode to the disk

- First access to file F in a CN: contact to the TM
  - Ask for tokens (byte range and inode)
  - Ask for metanode token (the CN offers itself as MN for F)
    - If there is not a MN for F yet, TM assigns this role to the requestor
      - First node accessing the file is the MN
    - If there is a MN for F, TM sends back its identity
      - All the tokens are asked with a single message to the TM

- Periodically and when the token is revoked, the CN sends its copy to MN
  - MN merges the concurrent changes to inodes:
    - \( \text{Mod date} = \max(\text{Mod date copies}) \) (same for size and pointers)

- CN leaves the role of MN for F when it does not use the F inode anymore
  - Reject dumps for the inode \( \rightarrow \) new MN choice for F
PVFS (Parallel Virtual File System)

- A scalable file system for data-intensive applications
- Object-based file system
  - Files are divided into multiple pieces called stripe units (by default 64KB)
  - PFVS distributes stripe units across I/O nodes using a round robin policy
- http://www.parl.clemson.edu/pvfs/
PVFS

Access to metainformation:

Access to data:
PVFS Components

- Metadata servers (mgr)
  - A single manager daemon manages metadata for PVFS files
- I/O server (iod)
  - I/O daemons are in charge of storing and retrieving file data
  - When clients open a PVFS file, the mgr informs them of location of the I/O daemons
  - Direct connection between clients and I/O daemons
- PVFS native API (libpvfs)
  - Provides user-space access to the PVFS servers (Transparent to the user)
- PVFS Linux kernel support
  - Provides the functionality to mount PVFS file systems on Linux nodes
  - Enables program access to PVFS files without any modification
  - It is not needed, although it is convenient for the interaction with PVFS

![Diagram](image_url)
HDFS

- Hadoop Distributed File System
HDFS

• A scalable, fault-tolerant and manageable file system
• Scalability:
  – With thousand of nodes and PB of data
  – Parallel processing
  – Bandwidth scales linearly with the number of nodes
  – Minimal data motion
  – Rack awareness
  – Like Google File System, optimized for large files
• Fault tolerance as essential design property
  – Redundancy (3 replicas by default)
  – Utilities for the diagnosis of the file system
  – Rollback, which enables the back to previous versions of HDFS
  – Standby NameNode (high availability and redundancy)
• Easy to manage
  – For a large number of nodes
  – Automatic addition and removal of nodes
  – According to their creators, 1 operator is enough for 3k nodes
HDFS operations

• As Google File System, HDFS:
  – Provides Read, Write, Rename and Append operations
  – Does not support Random Writes

• Optimized for large files
  – Typically, in MapReduce environments
  – MapReduce moves compute processes to the data on HDFS
  – Processing tasks can be performed on the physical node where the data resides
  – Network I/O reduction
  – Higher performance

• Large block sizes
  – Typically, 128 MB

• Used in companies like Twitter, Facebook or Yahoo!
HDFS Namenode and Datanodes

• The Namenode keeps the namespace in memory
  – Design must be easy
• The Namenode is in charge of monitoring the number of replicas of a block
• If a replica of a block is lost (Datanode failure or disk failure), the Namenode creates another replica of the block
• The Namenode does not directly send requests to Datanodes. It sends instructions to the Datanodes by replying to heartbeats sent by them.
  – Examples of instructions: replicate blocks to other nodes, remove local block replicas, re-register and send an immediate block report, or shut down the node
• More info: http://hortonworks.com/hadoop/hdfs/
Ceph

• Goals:
  – Scalability
    • To hundreds of petabytes and beyond
  – Reliability
    • “…failures are the norm rather than the exception…”
  – Performance

Source: S.A. Weil et al., Ceph: A Scalable, High-Performance Distributed File System, OSDI 2006
First Key Idea: Object-based Storage

Traditional Storage

Operating System

Applications

System Call Interface

File System

Logical Block Interface

Block I/O Manager

Hard Drive

Object-based Storage

Operating System

Applications

System Call Interface

File System Client Component

Object Interface

File System Storage Component

Block I/O Manager

Object-based Storage Device (OSD)
Second Key Idea: Decoupled Data and Metadata

Source: S.A. Weil et al., Ceph: A Scalable, High-Performance Distributed File System, OSDI 2006
Ceph Overview

Source: S.A. Weil et al., Ceph: A Scalable, High-Performance Distributed File System, OSDI 2006
Main Features

• Decoupled data and metadata
  – CRUSH
    • CRUSH maps objects to storage devices

• Dynamic Distributed Metadata Management
  – Dynamic subtree partitioning
    • Distributes metadata amongst MDSs

• Object-based storage
  – OSDs handle migration, replication, failure detection and recovery

Source: S.A. Weil et al., Ceph: A Scalable, High-Performance Distributed File System, OSDI 2006
Dynamic Subtree Partitioning

Busy directory hashed across many MDS's

Source: S.A. Weil et al., Ceph: A Scalable, High-Performance Distributed File System, OSDI 2006
Distributed Object Storage

Source: S.A. Weil et al., Ceph: A Scalable, High-Performance Distributed File System, OSDI 2006
CRUSH

• Controlled Replication Under Scalable Hashing
• \text{CRUSH}(x) \rightarrow (\text{osd}_{n1}, \text{osd}_{n2}, \text{osd}_{n3})
  – Inputs
    • x is the placement group
    • Hierarchical cluster map
    • Placement rules
  – Outputs a list of OSDs
• Advantages
  – Anyone can calculate object location
  – Cluster map infrequently updated

Source: S.A. Weil et al., Ceph: A Scalable, High-Performance Distributed File System, OSDI 2006
GlusterFS

- Migration from proprietary, monolithic server architectures to virtualized and open architectures
- Storage has not kept pace with computing
- Gluster tries to overcome this issue
- Gluster is a file-based scale-out NAS platform
  - Using both on-premise commodity hardware and public cloud storage infrastructure
  - Better price and performance

“It was much nicer before people started storing all their personal information in the cloud.”
Gluster Design Goals

- Elasticity: addition or deletion of data resources to a storage pool as needed, without interrupting the working
- Linear scaling:
  - “twice the amount of storage systems will deliver twice the observed performance”
  - Traditional file system models and architectures are unable to scale in this manner (usually logarithmic scalability)
- How to achieve this:
  - The elimination of metadata
  - Effective distribution of data to achieve scalability and reliability
  - The use of parallelism to maximize performance via a fully distributed architecture
Elimination of metadata

• Location metadata enables to find the logical and physical location of data

• First solution: a centralized metadata server
  – Single point of failure: The operations depend on the working of the metadata server
  – Performance bottleneck: Specially when the number of files and file operations increases

• An alternative: Distributed metadata servers (metadata is spread among a large number of storage systems)
  – Performance overhead: locking and synchronization mechanisms
  – Corruption issues: more chances of a corrupted storage system, since the increase of the number of systems
Elimination of metadata

• Gluster overcomes these issues by means of:
  – No separation between metadata and data
  – No use of any separate metadata server, whether centralized or distributed

• Use of Elastic Hash Algorithm
  – Inputs: the pathname and filename
**Elastic Hash Algorithm (EHA)**

- **Benefits:**
  - The algorithm makes Gluster faster
  - No contention for metadata
  - No need to synchronize metadata:
    - Linear scaling for distributed environments
    - Safer in distributed environments (no risk of corruption)

- **A hash-based algorithm**
  - Mathematical function that converts a string of an arbitrary length into a fixed length values
    - Determinism: same input (pathname+filename), same hash
    - Uniformity: Results tend to be uniformly distributed
  - EHA is based on the Davies-Meyer hashing algorithm

- **But, what’s happened if some disk fail, if a disk is full, if files need to be redistributed? Elasticity**
Elastic Hash Algorithm (EHA)

- **Elasticity:**
  - Gluster sets up a very large number of virtual volumes
  - The EHA assigns files to virtual volumes
  - An additional process assign virtual volumes to multiple physical devices (virtualization layer)

- **What’s happened if the file is renamed?**
  - The EHA results in a different value
  - Files can be large (rewrite and move files are not real-time operations)
  - Solution: creating a pointer when a file is renamed, which redirects to the old logical volume location. A background process migrates these files and only when this operation is complete, the pointers are removed
  - Exactly the same if files need to be moved or reassigned (e.g., a disk has been degraded)
    - Reassignment decisions made in real-time
    - Physical migration of files as a background process
Scalability and Reliability

- Gluster provides a global namespace with access to a Virtual Storage Pool.
- Gluster supports n-way synchronous replication.
- A storage system server is replicated to another storage system server using synchronous writes.
  - Fault tolerance
  - Reads spread across all members of the mirror.
- Write operations slightly slower than read operations for small number of storage nodes.
- At large numbers of storage nodes, writes can be faster than reads, especially with non-sequential reads.
- No write caching.

Source: [http://www.gluster.com](http://www.gluster.com)
Outline

• Data intensive computation
• Scalable and parallel file systems. Lustre, GPFS, PVFS, HDFS, Ceph, GlusterFS
• Parallel I/O libraries. MPI-IO, HDF5, NetCDF
Application require more software than just a parallel file system

Break up support into multiple layers with distinct roles:

- **Parallel file system** maintains logical space, provides efficient access to data (e.g. PVFS, GPFS, Lustre)
- **Middleware layer** deals with organizing access by many processes (e.g. MPI-IO, UPC-IO)
- **High level I/O library** maps app. abstractions to a structured, portable file format (e.g. HDF5, NetCDF)

*Source: Parallel NetCDF, Argonne National Laboratory*
Middleware layer

- Enables the optimization of parallel I/O with the goal of reducing the latency
- Techniques:
  - Data sieving
  - Collective operation
  - High level interfaces
  - Access patterns
  - Caching and prefetching
  - Hints
Data sieving

- Used to improve the requests of non contiguous data
- Reducing I/O latency by means of the reduction of the number of requests to the file system
  - Few large and contiguous requests
  - Extraction, in memory, of data actually needed
  - More data than required is read

- E.g., ROMIO
Collective operations

- Used to improve the request of non contiguous data by multiple processes
- I/O requests from different processes are combined in order to get a single larger I/O request
- Increment effective I/O bandwidth
- E.g., ROMIO
High level interfaces

- More functionality than in traditional I/O interfaces
  - E.g., matrix read operation
- E.g., MPI-IO
Access patterns

- Data layout is critical in the performance improvement
- Knowing access patterns enables the use of such information in the most efficient way
- nCUBE and Vesta provides higher control over the layout
- Some parallel I/O systems are oriented to a specific field. For instance, ChemIO (chemistry data)
- Panda hides physical details to applications
- MPI-IO uses MPI datatypes to describe the data layout both in memory and in file (`MPI_File_set_view()`)
Caching and prefetching

- Temporal and spatial locality
- Two kinds of caches:
  - Data block cache
  - Metadata cache
- Improvement in read and write operations:
  - Read: Prefetching
  - Write: Delayed write
- Problems of coherence -> Coherence algorithms resolution. E.g., cooperative caches
Prefetching

- **Sequential read-ahead**: Exploits sequential I/O operations
  - Very limited use for small files
  - No efficient for no sequential access patterns

- **One-block look-ahead**
  - Predict block i+1

- **Infinite-block look-ahead**
  - Predict all the future blocks

- **Portion-recognition (PORT)**
  - Recognizes regular data portions
Hints

Goal: Improve the performance of the I/O system
Lampson, 1983
- Operating systems: Alto, Pilot
- Networks: Arpanet, Ethernet
- Languages: Smalltalk

To infer future access from past access
To expose advanced knowledge about a system component (e.g., virtual memory system, cache)
Hints

- **Advising hints:**
  - Recommendation about the resource management in order to increase the performance

- **Disclosing hints:**
  - Describing the knowledge about the application
  - Advantages:
    - Information does not depend on the implementation. Portable I/O optimization mechanism.
    - Different policies can be selected
    - It is expressed in terms of the system interface
E.g., MPI-IO

MPI_Info:
- Pairs (key, value) -> Additional information about I/O operations
- Set of fixed keys
- E.g., key `striping_unit`, unit for the distribution of files in the I/O devices
Parallel I/O interface

Operations:
- Basic operations
- Non contiguous read and write operations
- Non blocking read and write operations
- Collective operations
- Operations with shared pointers
- Other operations

E.g., MPI-IO
Most parallel file systems use a UNIX-like API (POSIX)

- open: Open a file
- creat: Create a file
- read: Read n bytes from the current position of the file pointer
- write: Write n bytes from the current position of the file pointer
- close: Close a file
- lseek: Seek the file pointer
Other I/O operations

*readv, writev*

- Read/write from/to multiple buffers in memory
- It is assumed that data are stored in a contiguous way in the file
- Most implementations have a limit of 16 buffers of memory in a call
- Very limited use, because a user needs specify no adjacency in files more often than in memory
Asynchronous I/O

- **aio_read, aio_write** in POSIX
  - Non-blocking: The user can check if one operation has finished or not, or the implementation uses a signal
  - Most implementations have a limit of concurrent asynchronous operations; this limit is small, although it is not predefined
  - The performance is not good enough
I/O operations list

\textit{l/io\_listio} in POSIX

- Specify a list of I/O operations
- This list can be a mixture of reads and writes
- Every operation is treated internally as an \textit{aio\_read}, \textit{aio\_write} operation
- It is possible to specify if \textit{l/io\_listio} is non-blocking or not
Drawbacks of *lio_listio*

- The list is not treated as a single request
- No collective I/O
- No portable (it is not implemented in all the systems)
I/O access patterns in parallel applications

- Different from sequential patterns
- Sequential programs usually access data in a contiguous way
- In the case of several processes, every process can access to non-contiguous zones of a file
  - E.g., bytes 0–10, 20–30, 40–50,...
- Groups of processes can access to the same file simultaneously and these access can be non-contiguous
Problems of the UNIX API in parallel I/O

- It is not possible to express non-contiguous accesses in a single function call.
- Every piece of data must be accessed independently. Too many system calls and therefore low performance.
- No collective I/O.
Communicator:
- Set of processes interconnected by messages
- Set of processes that can access a file with I/O operations

File access:
- Independent (no coordination between processes)
- Collective (every process associated with the communication must participate in the collective access)

No contiguous access in both memory and file

MPI datatypes
- Basic MPI datatypes:
  - Those that correspond to basic data types in the programming language: MPI_INT, MPI_FLOAT, ...
- Derived MPI datatypes:
  - To express data distribution in files and partitioning between processes
MPI-IO API

- MPI_File_open()
- MPI_File_close()
- MPI_File_read() and family
- MPI_File_write() and family
- MPI_File_seek()
- MPI_File_set_view()
MPI_File_open() : Open a file

Header:

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh);
```

Arguments:

- `comm`: communicator
- `filename`
- `amode`: access mode
- `info`: info object
- `fh`: file handle

Description:

- Open the file, establish the view by default in the file, and establish the access mode `amode`. It is a collective operation. `comm` is a valid communicator. The values specified in `amode` must be identical for all the participants.
MPI_File_open()

**amode:**
- **MPI_MODE_APPEND:** The initial pointer is the end of the file
- **MPI_MODE_CREATE:** Create the file if it does not exist
- **MPI_MODE_DELETE_ON_CLOSE:** Delete the file on close
- **MPI_MODE_EXCL:** It returns an error if the file exists and MPI_MODE_CREATE is used
- **MPI_MODE_RDONLY:** read only
- **MPI_MODE_RDWR:** read and write
- **MPI_MODE_SEQUENTIAL:** the file only can be accessed in a sequential way
- **MPI_MODE_UNIQUE_OPEN:** the file cannot be opened in a concurrent way
- **MPI_MODE_WRONLY:** write only
Example:

```c
MPI_File fh;
MPI_File_open(MPI_COMM_SELF,
    "test.txt", MPI_MODE_CREATE|
    MPI_MODE_RDWR,
    MPI_INFO_NULL, &fh);
```
MPI_File_close()  

Header

\[ \text{int } \text{MPI}\_\text{File}\_\text{close}(\text{MPI}\_\text{File }*fh); \]

Arguments:
- \( fh \): file handle

Description:
- Close the file handled by \( fh \) and free the internal data structures associated. It is a collective operation.
- Example:

\[ \text{MPI}\_\text{File}\_\text{close}(\&fh); \]
**MPI_File_read()** : Read from a file

**Header:**

```
int MPI_File_read (MPI_File fh, void *buf,
                  int count, MPI_Datatype datatype,
                  MPI_Status *status);
```

**Arguments:**

- `fh`: file handle
- `buf`: buffer address
- `count`: number of elements
- `datatype`: datatype of every buffer element
- `status`: status of the operation

**Description:**

Read from a file managed by `fh` `count` elements of `datatype` in the buffer `buf`, starting from the pointer of the file.

**Example:**

```
MPI_File_read(fh, buf, count, MPI_INT, &status);
```
MPI_File_read_all() : Collective read

Header:

```
int MPI_File_read_all (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
```

Arguments:

- **fh**: file handle
- **buf**: buffer address
- **count**: number of elements
- **datatype**: datatype of every buffer element
- **status**: status of the operation

Description:

Collective version of `MPI_File_read()`

Example:

```
MPI_File_read_all(fh, buf, count, MPI_INT, &status);
```
MPI_File_iread() : Non-blocking read

**Header:**

```c
int MPI_File_iread (MPI_File fh, void *buf,
                     int count, MPI_Datatype datatype,
                     MPI_Request *request);
```

**Arguments:**

- `fh`: file handle
- `buf`: buffer address
- `count`: number of elements
- `datatype`: datatype of every buffer element
- `request`: status request

**Description**

This routine is the non-blocking version of MPI_File_read(). The same as MPI_File_read(), except that returns immediately and stores a request object.

**Example:**

```c
MPI_File_iread(fh, buf, count, MPI_INT, &request);
```
MPI_Test() and MPI_Wait()

- int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status);
- int MPI_Wait (MPI_Request *request, MPI_Status *status);

**Description:**
- MPI_Test() returns flag = true if the operation identified by request has finished. The object status stores information about the end of the operation.
- MPI_Wait() returns after the operation identified by request has finished. If the object associated with request was created with a non-blocking operation, the object is free and request is set to MPI_REQUEST_NULL.
int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status);

int MPI_Wait (MPI_Request *request, MPI_Status *status);

Description:

- MPI_Test() returns flag = true if the operation identified by request has finished. The object status stores information about the end of the operation.

- MPI_Wait() returns after the operation identified by request has finished. If the object associated with request was created with a non-blocking operation, the object is free and request is set to MPI_REQUEST_NULL.
**MPI_File_read_shared() : Shared file pointer**

**Header:**

```c
int MPI_File_read_shared (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
```

**Arguments:**

- `fh`: file handle
- `buf`: buffer address
- `count`: number of elements
- `datatype`: datatype of every buffer element
- `status`: status of the operation

**Description:**

Read from a file handled by `fh count` elements of datatype in the buffer `buf`, starting from the shared file pointer.

**Example:**

```c
MPI_File_read_shared(fh, buf, count, MPI_INT, &status);
```
MPI_File_write() : Write a file

Header:
int MPI_File_write (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);

Arguments:
- fh: file handle
- buf: buffer address
- count: number of elements
- datatype: datatype of every buffer element
- status: status of the operation

Description:
Write in a file handled by fh, count elements of datatype of the buffer buf, starting from the file pointer

Example:
MPI_File_write(fh, buf, count, MPI_INT, &status);
MPI_File_write_all() : Collective write

**Header:**

```c
int MPI_File_write_all (MPI_File fh,
    void *buf, int count, MPI_Datatype datatype,
    MPI_Status *status);
```

**Arguments:**
- `fh`: file handle
- `buf`: buffer address
- `count`: number of elements
- `datatype`: datatype of every buffer element
- `status`: status of the operation

**Description:**
Collective version of MPI_File_write()

**Example:**

```c
MPI_File_write_all(fh, buf, count, MPI_INT, &status);
```
**MPI_File_iwrite()**: Non-blocking write

**Header:**

```c
int MPI_File_iwrite (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request);
```

**Arguments:**

- `fh`: file handle
- `buf`: buffer address
- `count`: number of elements
- `datatype`: datatype of every buffer element
- `request`: status request

**Description:**

Non-blocking version of `MPI_File_write()`. The same as `MPI_File_write()`, except that returns immediately and stores a request object.

**Example:**

```c
MPI_File_iwrite(fh, buf, count, MPI_INT, &request);
```
**Header:**

```c
int MPI_File_write_shared
    (MPI_File fh, void *buf, int count,
     MPI_Datatype datatype,MPI_Status *status);
```

**Arguments:**

- `fh`: File handle
- `buf`: Buffer address
- `count`: Number of elements
- `datatype`: Datatype of every buffer element
- `status`: status of the operation

**Description:**

Write to a file handled by `fh count` elements of datatype in the buffer `buf`, starting from the shared file pointer.

**Example:**

```c
MPI_File_write_shared(fh, buf, count, MPI_INT, &status);
```
## MPI-IO data access functions

<table>
<thead>
<tr>
<th>positioning</th>
<th>synchronism</th>
<th>coordination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>noncollective</td>
</tr>
<tr>
<td>explicit offsets</td>
<td>blocking</td>
<td>MPI.FILE.READ AT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI.FILE.WRITE AT</td>
</tr>
<tr>
<td></td>
<td>nonblocking &amp; split</td>
<td>MPI.FILE.READ AT</td>
</tr>
<tr>
<td></td>
<td>collective</td>
<td>MPI.FILE.WRITE AT</td>
</tr>
<tr>
<td></td>
<td>blocking</td>
<td>MPI.FILE.READ AT.All</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI.FILE.WRITE AT.All</td>
</tr>
<tr>
<td></td>
<td>nonblocking &amp; split</td>
<td>MPI.FILE.READ AT.All begin</td>
</tr>
<tr>
<td></td>
<td>collective</td>
<td>MPI.FILE.WRITE AT.All begin</td>
</tr>
<tr>
<td></td>
<td>nonblocking &amp; split</td>
<td>MPI.FILE.READ AT.All end</td>
</tr>
<tr>
<td></td>
<td>collective</td>
<td>MPI.FILE.WRITE AT.All end</td>
</tr>
<tr>
<td>individual file</td>
<td>blocking</td>
<td>MPI.FILE.READ AT</td>
</tr>
<tr>
<td>pointers</td>
<td></td>
<td>MPI.FILE.WRITE AT</td>
</tr>
<tr>
<td></td>
<td>nonblocking &amp; split</td>
<td>MPI.FILE.READ AT</td>
</tr>
<tr>
<td></td>
<td>collective</td>
<td>MPI.FILE.WRITE AT</td>
</tr>
<tr>
<td></td>
<td>nonblocking &amp; split</td>
<td>MPI.FILE.READ AT.All begin</td>
</tr>
<tr>
<td></td>
<td>collective</td>
<td>MPI.FILE.WRITE AT.All begin</td>
</tr>
<tr>
<td></td>
<td>nonblocking &amp; split</td>
<td>MPI.FILE.READ AT.All end</td>
</tr>
<tr>
<td></td>
<td>collective</td>
<td>MPI.FILE.WRITE AT.All end</td>
</tr>
<tr>
<td>shared file pointer</td>
<td>blocking</td>
<td>MPI.FILE.READ_SHARED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI.FILE.WRITE_SHARED</td>
</tr>
<tr>
<td></td>
<td>nonblocking &amp; split</td>
<td>MPI.FILE.READ_SHARED</td>
</tr>
<tr>
<td></td>
<td>collective</td>
<td>MPI.FILE.WRITE_SHARED</td>
</tr>
<tr>
<td></td>
<td>nonblocking &amp; split</td>
<td>MPI.FILE.READ_Ordered begin</td>
</tr>
<tr>
<td></td>
<td>collective</td>
<td>MPI.FILE.WRITE.Ordered begin</td>
</tr>
<tr>
<td></td>
<td>nonblocking &amp; split</td>
<td>MPI.FILE.READ_Ordered end</td>
</tr>
<tr>
<td></td>
<td>collective</td>
<td>MPI.FILE.WRITE_Ordered end</td>
</tr>
</tbody>
</table>

Source: Peter Corbett et al. “Overview of the MPI-IO Parallel I/O Interface”, IPPS 95
Positioning

- Two concepts to describe locations in a file:
  - MPI datatypes
  - Offsets
    - new_file_position=old_position+(size(buftype)*bufcount)/size(etype)
- The file pointer is updated at the beginning of each access by the amount of data requested.
The use of nonblocking data access functions can improve performance, through the explicit overlap of computation with I/O.

A separate request complete call (MPI_Wait or MPI_Test) is needed to complete the I/O request (certify that data has been read/written).

The only way to guarantee data is actually written to storage is using the MPIO_File_sync call.
Coordination

- Collective operations: “All” processes in the communicator group which opened the file must participate in the operation.
- Differences collective-independent:
  - Semantic viewpoint: Potential synchronization.
  - Performance viewpoint: Collective version can be much faster.
**MPI_File_seek() : Sets a file pointer**

- **Header:**
  
  ```c
  int MPI_File_seek (MPI_File fh, MPI_Offset offset, int whence);
  ```

- **Arguments:**
  - `fh`: is the file handle
  - `offset`: is the file offset
  - `whence`: is the update mode

- **Description:**
  - This routine updates the individual file pointer according to `whence`.

- **whence values:**
  - `MPI_SEEK_CUR`: the file pointer is set to its current position plus `offset`
  - `MPI_SEEK_END`: the file pointer is set to the end of the file position plus `offset`
  - `MPI_SEEK_SET`: the file pointer is set to `offset`

- **Example:**
  - ```c
    MPI_File_seek(fh, -10, MPI_SEEK_CUR);
    MPI_File_seek(fh, 0, MPI_SEEK_SET);
    ```
MPI_File_set_view() : Establish a file view

• Header:

   int MPI_File_set_view
   (MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
    MPI_Datatype filetype, char *datarep, MPI_Info info);

• Arguments:

  – fh: is the file handle
  – disp: is the displacement
  – etype: is the elementary datatype
  – filetype: is the filetype
  – datarep: is the data representation
  – info: is the info object

• Description:

  – This routine associates a new view defined by disp, etype, filetype, and datarep with the open file referred to by fh. This is a collective operation. All participating tasks must specify the same values for datarep and the same extents for etype.
• **datarep values:**
  
  - **external32**: States that read and write operations convert all data from and to the external32 representation that is documented in the MPI-2 standard.
  
  - **Internal**: can be used for I/O operations in a homogeneous or heterogeneous environment. IBM has defined its internal format with the intent that any implementation of MPI provided by IBM can use this format.

  - **native**: should be used in most situations. Data in this representation is stored in a file exactly as it is in memory. This representation is always suitable in a homogeneous MPI environment and does not incur conversion costs.

• **Example:**
  
  - `MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);`
MPI derived datatypes

- To describe how data is laid out in:
  - The user’s buffer: buftype
  - The file: filetype

*Goals:*
- Consistency
- Portability
• It defines a data pattern that is replicated throughout the file.

Common distribution patterns:
• broadcast/reduce
• scatter/gather
• The filetype and etype are specified at file open time:
  – Data layout during file creation or filesystem creation: Too restrictive!!:
    • it prohibits accessing a file using multiple patterns simultaneously.
    • Static data layout information must be stored as file metadata, inhibiting file portability.
  – Data layout during data access (read/write): It is cumbersome and it is expected that filetypes will not be changed too often.
Buftype

- It describes the layout in the user’s buffer.

*Process buffer*
Noncontiguous access

- Noncontiguous data in both memory and file is specified using MPI datatypes, both predefined and derived.
- Data layout in memory specified on each call, as in message-passing.
- Data layout in file is defined by a file view.
- A process can access data only within its view.
- Views can be changed.
Example (2-D matrix)

File structure:

Distribute and transpose

Source: Peter Corbett et al. “Overview of the MPI-IO Parallel I/O Interface”, IPPS 95
Example (2-D matrix)

Implementation using filetypes and buftypes:

process 1 filetype:  

process 2 filetype:

process 3 filetype:

actual layout in the file:

buftype (all processes):
Example (2-D matrix)

File structure:

```
1 2 3
4 5 6
7 8 9
```

Distribute and transpose

```
1 4 7
2 5 8
3 6 9
```

Process 1 buffer

Process 2 buffer

Process 3 buffer

**ftype** = set of rowtype

**buftype** = set of columntype
Example (2-D Matrix) (Code I)

read_matrix(
    char *fname, /* File containing matrix “A[n][n]” */
    int n, /* Number of rows (columns) of matrix */
    MPI_Datatype etype, /* Matrix element type */
    void *localA) /* Target for transposed matrix */
{
    MPI_File fh;
    MPI_Datatype ftype, buftype;
    MPI_Status status;
    MPI_Datatype column_t, rowtype;
    int m, rank, nrows, sizeofetype, sizeofrowtype;
Example (2-D Matrix) (Code II)

/* Create row-cyclic filetype for data distribution */
MPI_Type_extent(etype, &sizeofetype);
MPI_Type_vector(n, 1, sizeofetype, etype, &rowtype);
MPI_Type_extent(rowtype, &sizeofrowtype);
MPI_Type_hvector(n, 1, n*sizeofrowtype, rowtype, &ftype);
MPI_Type_commit(&ftype);
MPI_Type_free(&rowtype);

/* Create buftype to transpose matrix into process memory */
MPI_Comm_size(MPI_COMM_WORLD, &m);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_vector(n, 1, n*sizeofetype, etype, &column_t);
MPI_Type_hvector(n, 1, sizeofetype, column_t, &buftype);
MPI_Type_commit(&buftype);
MPI_Type_free(&column_t);
/* Read, distribute and transpose the matrix (and cleanup) */
MPI_File_open(MPI_COMM_WORLD, fname,
  MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, rank*n*sizeofetype, etype, ftype,"native", MPI_INFO_NULL);
MPI_File_read_all(fh, localA, 1, buftype, &status);
MPI_File_close(&fh);
MPI_Type_free(&ftype);
MPI_Type_free(&buftype);
}
int main(int argc, char *argv[]) {
    char buffer[20];
    int i, j, m, rank;

    if (argc != 2) {
        fprintf(stderr, "Error. Usage: matrix filename \n");
        exit(1);
    }

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &m);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    read_matrix(argv[1], m, MPI_CHAR, buffer);
}
Example (2-D Matrix) (Code V)

```c
sleep(rank*5);
printf("The transposed matrix of the process
%d is:\n",rank);
for (i=0;i<m;i++) {
    for (j=0;j<m;j++){
        printf("%c ",buffer[i*m+j]);
    }
    printf("\n");
}
MPI_Finalize();
exit(0);
```
Example (2-D Matrix)

Data file:

1234abcd$%&/zyxw5678efgh$
%&/zyxw1234ijkl$%&/zyxw567
8mnop$%&/zyxw
Example (2-D Matrix)
MPI-IO Optimizations

• Given complete access information, an implementation can perform optimizations such as:
  – Data sieving: Read large chunks and extract what is really needed
  – Collective I/O: Merge requests of different processes into larger requests
  – Improved prefetching and caching
High Level Libraries

• APIs more appropriate for computational science
  – Multidimensional datasets
  – Typed variables
  – Attributes

• Provide self-describing, structured files

• Map to middleware interface
  – Encourage collective I/O

• Implement optimizations that middleware cannot, such as
  – Caching attributes of variables
  – Chunking of datasets

Source: Parallel NetCDF. Argonne National Laboratory
HDF5

• HDF: Hierarchical Data Format: Model for managing and storing data
  – Abstract Data Model
  – Abstract Storage Model

• HDF5: library that provides a programming interface to a concrete implementation of the abstract models

Source: http://hdfgroup.org
HDF5 Library

- Providing access data in HDF5 format
- Especially oriented to large and complex sets of data
- Platform independent
- HDF5 Library is linked to an application program (C, C++, Fortran, Java)
HDF5 History

- Origin: HDF4 (called HDF)
  - last major release: version 4
- Changes to format, library and Data Model: HDF5
- HDF6 not foreseen
The Abstract Data Model

- Key concepts:
  - File
  - Group
  - Dataset
  - Dataspace
  - Datatype
  - Attribute
  - Property List
  - Link
An HDF5 file is a **container** that holds data objects.
HDF5 Groups and Links

HDF5 groups and links organize data objects.

Every HDF5 file has a root group.

Experiment Notes:
Serial Number: 99378920
Date: 3/13/09
Configuration: Standard 3

Parameters
10; 100; 1000

Timestep 36,000
HDF5 Datasets

- HDF5 datasets **organize and contain** “raw data values”.
  - HDF5 datatypes describe individual data elements.
  - HDF5 dataspaces describe the logical layout of the data elements.

![HDF5 Datatype](integer_32bit_LE)

**HDF5 Dataspace**

- **Rank**: 3
- **Dimensions**:
  - Dim_0 = 4
  - Dim_1 = 5
  - Dim_2 = 7

Specifications for single data element and array dimensions

Multi-dimensional array of identically typed data elements

Source: [http://hdfgroup.org](http://hdfgroup.org)
HFD5 Dataspace

- HDF5 datasets **organize and contain** “raw data values”.
  - HDF5 dataspaces describe the logical layout of the data elements.
HDF5 Datatypes

- HDF5 datasets **organize and contain** “raw data values”.
  - HDF5 datatypes describe individual data elements.

Source: http://hdfgroup.org
HDF5 Attribute

- A HDF5 object may have zero or more attributes.
- Attributes are stored with the object.
- An HDF5 attribute:
  - A name
  - Data (value): Similar to a dataset
    - Dataspace
    - Datatype
HDF5 Property List

- Conceptually similar to attributes.
- Whereas attributes are relevant to the user’s data and application, property lists are relevant to the behavior of the library.

<table>
<thead>
<tr>
<th>Property List Class</th>
<th>Used</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>H5P_FILE_CREATE</td>
<td>Properties for file creation</td>
<td>Set size of user block</td>
</tr>
<tr>
<td>H5P_FILE_ACCESS</td>
<td>Properties for file access</td>
<td>Set parameters for VFL driver. An example is MPI I/O.</td>
</tr>
<tr>
<td>H5P_DATASET_CREATE</td>
<td>Properties for dataset creation</td>
<td>Set chunking, compression, or fill value.</td>
</tr>
<tr>
<td>H5P_DATASET_XFER</td>
<td>Properties for raw data transfer (read and write)</td>
<td>Tune buffer sizes or memory management</td>
</tr>
<tr>
<td>H5P_FILE_MOUNT</td>
<td>Properties for file mounting</td>
<td></td>
</tr>
</tbody>
</table>

Source: http://hdfgroup.org
Parallel HDF5

Application

Parallel computing system (IBM AIX)

Compute node

Compute node

Compute node

Compute node

I/O library (HDF5)

Parallel I/O library (MPI-I/O)

Parallel file system (GPFS)

Switch network/I/O servers

Disk architecture & layout of data on disk

PHDF5 built on top of standard MPI-IO API

Source: http://hdfgroup.org
NetCDF

• NetCDF is a set of data formats, programming interfaces, and software libraries for scientific data
  – http://www.unidata.ucar.edu/software/netcdf/
  – API: http://www.unidata.ucar.edu/software/netcdf/docs/modules.html
• C, C++, Fortran 77, Fortran 90 and Java APIs
• Classic NetCDF Data Model:
  – Variables: N-dimensional arrays of data. Variables in netCDF files can be one of six types (char, byte, short, int, float, double).
  – Dimensions: describe the axes of the data arrays. A dimension has a name and a length.
  – Attributes: scalar values or 1D arrays metadata
• This model has been expanded in NetCDF 4.0
  – Called Common Data Model
A file has named variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One dimension may be of unlimited length.
A file has a top-level unnamed group. Each group may contain one or more named subgroups, user-defined types, variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One or more dimensions may be of unlimited length.
What’s new in NetCDF-4

- Interoperability with HDF5
  - Reading and editing NetCDF-4 Files with HDF5
  - Reading and editing HDF5 Files with NetCDF-4

- Use of groups:
  - NetCDF-4 files can store attributes, variables, and dimensions in hierarchical groups

- Compound types:
  - A data type which corresponds to a C struct

- Parallel I/O
  - Many processes can read/write netCDF data at the same time
Parallel I/O with NetCDF

• Two choices:
  – For parallel read/write access to classic and 64-bit offset data
    • Parallel netCDF developed by Northwestern University and Argonne National Laboratory
  – For parallel read/write access to netCDF-4/HDF5 files
    • Use the netCDF-4 API:
      – nc_open_par and nc_create_par functions
      – Support either collective or independent
      – Support MPI-IO via parallel HDF5
Option 1. Parallel netCDF

- Based on original netCDF work from Unidata
  - Derived from their source code
  - Argonne, Northwestern, and community

- Features:
  - C and Fortran interfaces
  - Portable data format (identical to netCDF)
  - Noncontiguous I/O in memory using MPI datatypes
  - Noncontiguous I/O in file using sub-arrays
  - Collective I/O

  - `ncmpi_create()`
  - `ncmpi_open()`
  - `ncmpi_get_att_text()`
  - `ncmpi_put_att_text()`
  - `ncmpi_close()`
Option 2. Parallel I/O netCDF-4 API

• nc_create_par
  (const char *path, int mode, MPI_Comm comm, MPI_Info info, int *ncidp)
modes: NC_NETCDF4|NC_MPIIO or NC_NETCDF4|NC_MPIPOSIX
• nc_var_par_access
  (int ncid, int var_id, int data_access)
data_access: NC_COLLECTIVE or NC_INDEPENDENT
• nc_open_par
  (const char *path, int mode, MPI_Comm comm, MPI_Info info, &ncid)
modes: NC_MPIIO or NC_MPIPOSIX
Parallel I/O programming model

Data writing:
/* 1. Initialize MPI. */
MPI_Init(&argc,&argv)
/* 2. Create a parallel netcdf-4 file. */
nc_create_par(FILE, NC_NETCDF4|NC_MPIIO, comm, info, &ncid)
nc_var_par_access(ncid, v1id, NC_COLLECTIVE)
/* 3. Write data. */
nc_put_vara_int(ncid, v1id, start, count, data)
/* 4. Close the file */
nc_close(ncid);
/* 5. Shut down MPI. */
MPI_Finalize();

Data reading:
Use nc_open_par() instead of nc_create_par() and nc_get_vara_int() instead of nc_put_vara_int()