Data Management of Scientific Datasets on High Performance Interactive Clusters

Faisal Ghias Mir

Master of Science Thesis

Stockholm, Sweden February 2006

Abstract

The data analysis on scientific datasets is executed in a collaborative environment. The low cost of small interactive clusters and availability of high performance system software is a promising attraction for processing medium level scientific datasets. The demanding functional and performance requirements of data analysis applications require mid layer software components that abstract away the data management and processing of scientific datasets. Active Data Repository and Data Cutter are example of such frameworks. This thesis work considers the development of a parallel middleware framework that gives necessary abstractions: rich set of library routines, processing subsets of datasets, generating new products, building complex data flows etc through the query engine. Moreover object model of framework is exposed for extending functionality. Similarly, the performance of data access, MPI-I/O over PVFS2, is reported in the context of various application access patterns. Lastly, a visualization module in VTK is developed and interfaced with framework.

Acknowledgements

This work was carried out at Department of Microelectronics and Information Technology, at the Royal Institute of Technology (KTH). Foremost, I would like to thank my advisor Prof. Scott B Baden, at University of California at Sandiego (UCSD), for allowing me to do this thesis work. I am grateful to him for promptly answering my long emails, conference calls, bundles of advices and encouragements, without all these this thesis work would not have been possible.

My special thanks to my examiner Prof. Mihail Matskin at KTH for reviewing my report, addressing our needs and giving valuable feedback on my work.

I would like to thank my parents for always giving me endless love, support and spending the last two years without me. Your encouragement, care and dedication have always been a source of motivation for me.

Finally, I would like to thank all of my friends and special one for making my time here as enjoyable as possible.

To my parents for always being there for me

Thesis Table of Contents

LIST OF FIGURES 7

CHAPTER # 1 ..... **8**

INTRODUCTION **8**

RELATED WORK 8
PROBLEM DESCRIPTION 9
REPORT OUTLINE 9

CHAPTER # 2 ..... **11**

SCIENTIFIC DATASETS: INTRODUCTION 11

STORAGE LAYOUT & APPLICATION ACCESS PATTERNS **11**

DATA STORAGE INTERFACES 11
THE GOAL: I/O PARALLELISM AND PERFORMANCE 12
STORAGE TECHNIQUES AND ACCESS PATTERN EFFECTS: 13
THE NOTION OF LOCALITY 13
Data Chunking *14*
Hilbert Space Curve Filling *15*
Indexing Schemes for Higher Dimensional Data Sets *16*
Processing Out of Core and In Core *16*
Available Libraries for Scientific Data Storage *16*

CHAPTER # 3 ..... **18**

PERFORMANCE EVALUATION WITH HIGH PERFORMANCE I/O LIBRARY: MPI-I/O **18**

I/O BOTTLE NECK 18
DISTRIBUTED VS PARALLEL FILE SYSTEMS 18
DATA FLOW IN PARALLEL FILE SYSTEM 19
PERFORMANCE GUIDELINES FOR PARALLEL FILE SYSTEMS 20
EFFICIENT USE OF FILE SYSTEM THROUGH MPI-I/O: 20
MPI-I/O OPTIMIZATION ALGORITHMS: 21
MPI-I/O FILE VIEWS: 22
MPI-I/O HINTS: 22
Performance Reporting with PVFS on Spindel *23*

CHAPTER # 4 ..... **25**

REQUIREMENTS OF FRAMEWORK **25**

DATASET USAGE SCENARIO 25
DATASET ABSTRACTION 25
LOAD BALANCING FOR SPARSE DATASETS 25
SUPPORT FOR RANGE QUERIES 26
MULTI BLOCK STRUCTURE PROCESSING 26
SUPPORT FOR USER DEFINED FUNCTIONS 26
DATA OPERATION 26

CHAPTER # 5 ..... **27**

DESIGN AND IMPLEMENTATION OF API: **27**

THE OBJECT MODEL 27
API Abstractions *27*
The Dataset *27*
Meta Data *27*

Data Buffer *28*
Region: Multi-Block Structure *28*
Call back Support *29*
Dataset Manager *29*
Object Execution Log *30*
PARALLEL IMPLEMENTATION 30
Query Engine *30*
Query Specification and Execution *30*
Query Parsing *31*
Query Plan *31*
Execution of Query Plan *33*
Each operation node stores the required information needs to execute the operation. After the operation execution conformity test of predicate is performed to check whether the operation has passed the condition with which it is performed. *33*
Parallel I/O Strategy *34*
Meta-Data Maintenance *34*
Parallel I/O Implementation *35*
PARALLEL SERVER USAGE BY CLIENTS 37
Data Transfer to Client Machines *37*
Examples: Code Snapshot *38*

CHAPTER # 6 40

PERFORMANCE MODEL OF FRAMEWORK 40

Analysis of Access Patterns *43*

CHAPTER # 7 47

DATASET VISUALIZATION 47

Usage Scenario *47*
Background *47*
Platform *48*
Datasets Visualization Pipeline: *48*
Off-Line Data Visualization *49*

CHAPTER 8 50

CONCLUSION AND FUTURE WORK 50

BIBLIOGRAPHY 51

List of Figures

Figure 2.1: Meta-information and dataset storage schemes
Figure 2.2: Contiguous file is divided in small blocks
Figure 2.3: File Stripping on I/O Servers
Figure 2.4: Stripping factor on disk controller
Figure 2.5: Access Pattern for Blocks
Figure 2.6: Access Pattern with 50% overhead in I/O times
Figure 2.7: Notion of Locality
Figure 2.8: Row-Major Chunking
Figure 2.9: Higher Dimensional Chunking
Figure 2.10: Hilbert Space Curve Filling
Figure 2.11: HDF Storage Format
Figure 3.1: MPI-I/O File Views
Figure 3.2: MPI-I/O Block size Performance
Figure 3.3: MPI-I/O Block size Performance at higher processors count
Figure 3.4: Non-Contiguous I/O Performance on Spindel
Figure 3.5: Large File Performance on Spindel
Figure 5.1: Domain of TeraShake Dataset
Figure 5.2: Chunked view of dataset, data-block and metadata object
Figure 5.3: Data Block in dataset
Figure 5.4: Single plane of Data Buffer (left)
Figure 5.5: Multi Block Region
Figure 5.6: Stencil Operations on Grid Points
Figure 5.7: TS_DataSetManager Object, Properties and Methods
Figure 5.8: The Query Engine
Figure 5.9: Phases of Query Execution
Figure 5.10: Query Parsing Algorithm
Figure 5.11: The Operation Graph
Figure 5.12: The Data Flow in Operation Graph
Figure 5.13: Query Planning Algorithm
Figure 5.14: Operation Graph: Execution
Figure 5.15: Query Execution Algorithm
Figure 5.16: Meta-Data Management
Figure 5.17: Data buffer Abstraction representation (*TS_3DFBuff*)
Figure 5.18: Dataset Blocks Access Scenario
Figure 5.19: Query Window Mapping on dataset file
Figure 5.20: Non-Contiguous Memory and File Access
Figure 5.21: Packed and Un-Packed I/O
Figure 5.22: Parallel I/O Algorithm
Figure 5.23: Data Transfer to Remote Clients
Figure 5.24: Parallel Data Transfer to Remote Clients
Figure 5.25: Code Snapshot of Framework API Calls
Figure 5.26: User Registered Callback Snapshot
Figure 6.1: Single Request
Figure 6.2: Total Time of Query: I/O and Computation Phases
Figure 6.3: Bandwidth Comparison
Figure 6.4: Access Pattern 100
Figure 6.5: Access Pattern 50
Figure 6.6: Access Pattern 25
Figure 7.1: Visualization, 3D View Window in dataset domain
Figure 7.2: General Usage of VTK Pipeline
Figure 7.3: Visualization Pipeline for TeraShake Datasets
Figure 7.4: Single Data Block, 16x30x30 (3 Floats: 172800 Bytes) in dataset
Figure 7.5: Iso-Surface of a single Data Block of Figure 4, three views
Figure 7.6: 2000 Blocks for Frame 02270

Chapter # 1

Introduction

The data management and processing of ever increasing scientific datasets on interactive cluster environments, demand understanding the use of existing software infrastructures and developing new techniques that are user centric, reliable, efficient and most importantly delivering High Performance to its users.

The data analysis on scientific datasets are executed in a collaborative environment and the Parallel File System provides users with the necessary abstractions for storage, concurrent access and writing new data products with promising high performance. The functional and performance requirements of such applications are more intricate than what these systems offer. Querying parts of datasets, heavily use of spatial indexes for boosting range queries speed, staging data from disk to process memory, user specific processing, aggregating information and data flow constructs are among the few features that scientific applications require.

The implementation of data repositories and middleware frameworks in cluster environments hide the underlying complexities of storage, retrieval and processing of datasets. These are tightly coupled systems with underlying computational and storage infrastructures and export uniform interfaces to its users. The scientists could entirely focus on developing complex analysis constructs rather than focusing on storage layouts, load allocations and scheduling issues.

Generally, middleware frameworks have generic support for common operations and standards on scientific datasets but mostly support notion of extensibility. Active Data Repository [1, 2] and Data Cutter [3] are among few examples of such frameworks. They have application in a variety of scientific disciplines like climate modeling, high energy physics, magnetic fusion, chemistry and Bioinformatics.

The user requirements are getting more demanding with continuous improvement in computational infrastructures. The ASCI initiative outlines the need of new software infrastructures with increasing storage and computational power. The Unified Parallel File System Initiative and increasingly use of iSCSI, Storage Area Network will put more demands on middleware frameworks to adapt to new technologies and deliver better performance.

Related Work

There is a need of software middleware infrastructures on parallel computer systems that is built upon new and existing software packages that efficiently stores, retrieve and process the multidimensional datasets. The middleware should exploit the inherent parallelism in task and resource management for delivering high performance. A very common usage pattern on large datasets is exploring a portion of dataset both in space and time. The exploration may span more than one datasets separated by a dependency relationship, a timestamp. Moreover, processing subsets of a group of datasets is dependent upon user operations and may cover distinct phases of filtration, transformation and writing new data products on disks. Similarly, data from different operations could be combined to build complex operation structures that the middleware engine must support. Most importantly, there must be provision for building new subsystems that use the middleware engine.

Active Data Repository [1, 2] is an example of such a system that implements this functionality with a set of services on cluster of machines. It uses a set of tightly coupled processes with disks that use de-clustering [4] and Hilbert Space Curve Filing [5] techniques to distribute multidimensional dataset blocks on disks to have better I/O performance. Exposing the lower level storage layout to middleware makes it complicated and there is no mention of fault tolerance

if a disk fails or how the system behaves once more disks are added to system, load balancing. In contrast the use of parallel file systems that use RAID [6] technology and clear distinction between the metadata and I/O nodes encapsulates these issues and leverage the architect to focus on the middleware functionality. Moreover higher level portable I/O libraries like MPI-I/O [7] are the likely candidates for utilizing these resources efficiently. Similarly, ADR lacks the abstractions for geometric operations for processing multi-block structures on data blocks and adding/deleting callbacks routines on the fly within the query operation. ADR has strong support for extensibility and its interfaces are used for implementing a Visualization Engine, The Virtual Microscope [8].

The data transfer to remote users is mostly supported by middleware engines but they have support for proprietary protocol. Generally, new emerging standards, like GridFTP [9], are not supported that makes interoperability difficult.

The Scientific Data Manager [10] combines MPI-I/O and Parallel File Systems with a small database support to mange the multidimensional datasets but lacks the support for user range queries and shipping the sub-sets to remote clients. The Data Cutter [3] is another middleware system but is inclined towards the archival storage system over WAN with the major emphasis of distributed query processing in heterogeneous environment.

Problem Description

The existing middleware frameworks and a variety of libraries for data management and processing give a range of building blocks to develop new infrastructure that address shortfalls in existing frameworks.

This thesis report primarily focuses on an incremental development of a middleware framework for the data management and processing for higher dimensional datasets. The framework design requires an in depth understanding of the storage and accessing techniques of these datasets as the major time is spent in staging the data blocks from disks to main memory of processes. The knowledge of these accesses gives chance to framework to optimize the I/O times. These datasets are accessed in small and large subsets of data blocks therefore the support for a range query interface in required to extract parts of datasets from a single request. Moreover the query must accept user defined processing constructs defined over various processing phases along with predicate support to define data workflow that could be written back to disks or feed to other tasks. The framework should have the capability to specify a multi block region in space and these operations should also be applicable to those.

The aim of such a framework is to provide users with a high performance implementation of above constructs that abstracts away the intricate details of data elements access, I/O, memory and load allocation. Similarly, the framework should export a simple Application Programming Interface, API, that can be used to link to new application.

In short, the major goal of thesis is working with a middleware framework for multidimensional datasets that can be outlined as follow:

- High Performance Storage and Access to Multidimensional Datasets
- The support for Range Queries on these datasets
- User defined processing along with predicate support to formulate new data flows
- A simple to use Application Programming Interface, API, for using the framework
- Validation of framework against a visualization module

Report Outline

The rest of this thesis report is organized as follow:

1. Chapter 2 discusses the common storage layout techniques in organizing the dataset on secondary mediums like disk arrays to have sufficient concurrency in retrieving and writing datasets. The storage of meta information in the context of spatial indexing schemes are also narrated that extracts parts of datasets and speedup the search process in locating data blocks. The impact of application access patterns in accessing data blocks is also discussed. Lastly, some of the available scientific data formats are considered for the framework.
2. Chapter 3 focus on performance evaluation of MPI-I/O library in the context of Parallel File Systems. The data flows of read and write are discussed to understand and exploit the file system read ahead and write behind optimizations. The impact of non contiguous data access in file is considered and the use of MPI derived types in issuing I/O request is discussed in optimizing I/O times.
3. Chapter 4 clearly outlines the core functionality expected from the framework. It specifies the usage scenario of framework access and detailed specification of various abstractions that is expected from the framework.
4. Chapter 5 discusses the design and implementation of the framework. The framework object model is specified along with supported functionality. It also specifies the I/O technique used for staging data blocks from disks to memory. The implementation of query engine is discussed that performs processing operations on subsets of datasets. The API and Parallel Data Server usage scenarios are explained in the context of code examples.
5. Chapter 6 explains the performance model for framework. The I/O and processing timings are reported in the context of application access patterns.
6. Chapter 7 discusses the implementation of Visualization Module that validates the working of framework and shows the use of query interface implemented in the framework.
7. Chapter 8 concludes the thesis work with further directions of future work.

Chapter # 2

Scientific Datasets: Introduction

The self describing scientific datasets consists of two types of data: The meta-information and the data arrays. The meta-information is vital for describing and accessing a dataset. The size of metadata is very small as compared to actual data arrays but the storage and access patterns of both types have performance affects.

The scientific datasets covers a broad range of application areas [11] and common characteristics of data they represent are: Multi-dimensional data, Multi-resolution data, Spatio-Temporal data, Mesh data etc. The dataset files may contain more than one type of data. Generally, dataset file is organized in a hierarchy with each portion describing some important characteristic of the dataset.

The most common ways for managing information is shown in below figures.

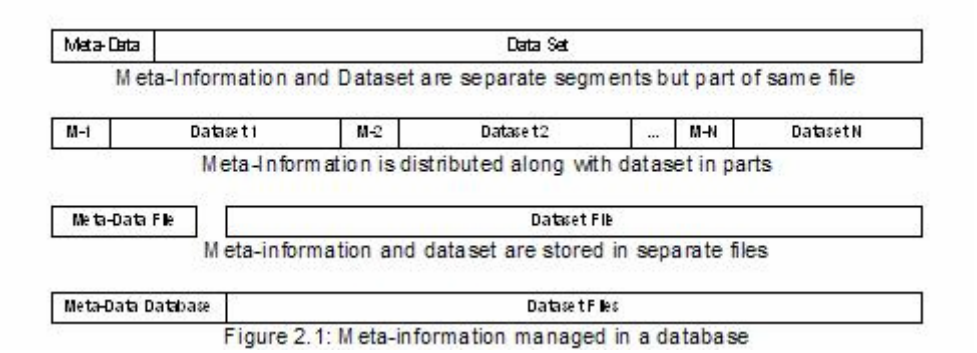


Figure 2.1: Meta-information managed in a database

Generally, data arrays are not accessed all together but some small portions of data is fetched for processing. Therefore, the way metadata and data arrays are stored and accessed have performance effects and this chapter highlights storage and application access patterns for processing scientific datasets.

Storage Layout & Application Access Patterns

The storage pattern defines the layout of datasets on storage medium, disk drives, and access pattern explains the data access by user applications. The storage pattern goals are data locality and I/O parallelism by data distribution on disk drives. The storage pattern is ***static*** but application access patterns quantify the performance benefits of a storage technique.

Data Storage Interfaces

The interactive cluster environments can have distinct or same compute and I/O nodes. The I/O nodes encapsulate the storage infrastructure and generally called storage or data servers. The storage servers use File System abstractions to provide users with data storage interfaces. The available File Systems at High Performance Facilities could be classified in to three distinct categories:

1. Network File System (NFS)
2. Storage Area Network File System (SAN)
3. Parallel File Systems

This thesis work primarily focus on using ***Parallel File System*** abstraction on small interactive cluster environment. The cluster could use attached disk drives with each node or use separate

disk arrays for storage. In general, the storage and data distribution techniques are independent of a specific system; therefore dataset processing is studied in the context of existing storage techniques. The distributed file system is used for verifying results.

The Goal: I/O Parallelism and Performance

The large data access from a single disk drive offers little I/O bandwidth and lacks parallelism. The ***File Stripping*** technique divides the file in small blocks and distribute them on available disk drives.

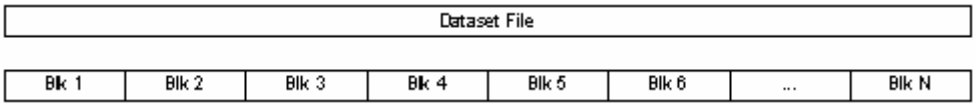


Figure 2.2: Contiguous file is divided in small blocks

Essentially, more than one disk drive controller is engaged for each large data access that result is better I/O bandwidth. Though transparent to users, the underlying technology used by Parallel File Systems to support block distribution on disk drives is:

- RAID Technology
- Attached Disk Drives with nodes in small interactive clusters

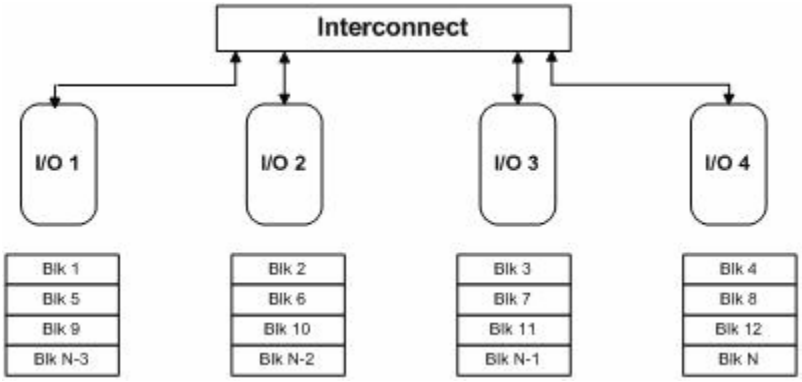


Figure 2.3: File Stripping on I/O Servers

The technique of distributing data blocks on disk drives depends upon available stripping techniques [12] on system. The linear stripping algorithm is shown in Figure 2.3. The blocks are arranged on I/O servers in increasing block numbers. The read request by user application programs is handed over to File System software that fetches in parallel the requested blocks from more than one I/O servers.

The aggregated I/O bandwidth partly depends upon amount of data transferred from each I/O server: disk controller. The major overhead in read operation is moving disk head to required data location. More seek operations with small block size results in lower I/O bandwidth. Therefore either user applications use bigger block size or put commonly accessed blocks close to each other on disks.

The stripping factor specifies data buffer transfer on each I/O server. Figure 2.4 shows stripping the same file with three contiguous blocks on same I/O server with aim of issuing large contiguous read operation with minimal seeks.

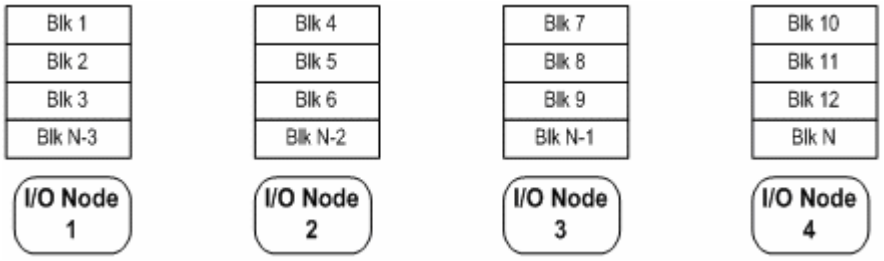


Figure 2.4: Stripping factor on disk controller

The data blocks access from I/O servers affects I/O performance if displacement in successive blocks engages less than available servers.

The performance is affected for stride access, displacement between successive blocks, where each process needs data at a specific displacement from a file location. The wrong striping unit may results in data need by process stores on the same I/O server and available concurrency is compromised. This behavior is shown in Figure 2.5.

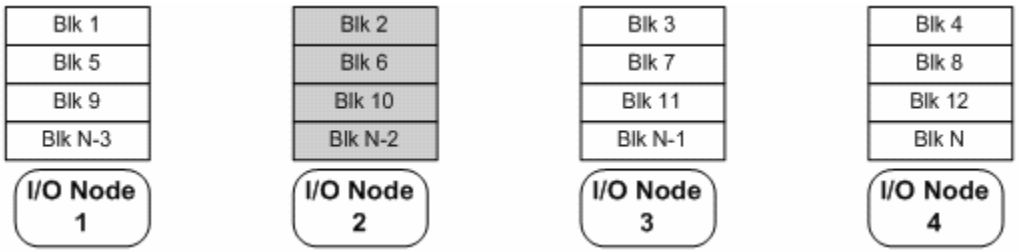


Figure 2.5: Access Pattern for Blocks: 2, 6, 10, 14..., N-2. Figure 2.5

Similarly, amount of useful data read from a block significantly affects I/O performance. If each request ends up in reading the complete block and only using half of its portion then I/O times contains overhead of 50% of reading wasted data. The Figure 2.6 shows an access pattern that fetches 4 file blocks but discards half of each block.

Figure 2.6: Access Pattern with 50% overhead in I/O times

In short, the available aggregated I/O bandwidth broadly depends on following factors:

- The block size for stripping data and allocation on disk drives
- How efficiently available disks are engaged on data access
- The fraction of useful data read from a disk/File System block
- The locality of data both in dataset file and user request
- The available storage system, interconnect and File System software

Storage Techniques and Access Pattern Effects:

The Notion of Locality

The actual data in scientific datasets consists of multidimensional arrays. The storage layout of higher dimensional arrays in memory, row major (C/C++), varies rapidly with the inner most dimension (on right) and it changes slowly as we move towards outer most dimension (on left). Therefore the locality in data is found towards the fast changing dimension.

The storage layout of modern disks drives consists of single dimension: the increasing addresses. Therefore a mechanism or technique is required to map the higher dimensional arrays on a single dimension of disk. The naive mapping approach is to use the linear memory model for higher dimensional arrays and store them on disk the way they are represented in memory (Global View of a large array). This technique has severe performance penalties [13].

Mostly, the row major technique does not provide the desired locality from a scientific application perspective. The reason is lack of locality of data in other remaining dimensions. Moreover applications does not need all of the data at once, it may be interested in only a small part of data but ranging over more than one dimension.

A rough quantification of cost of reading a block column, say 10 units wide, of 1000x1000 array stored in a file in row-major order results in 1000 seek and read operations.

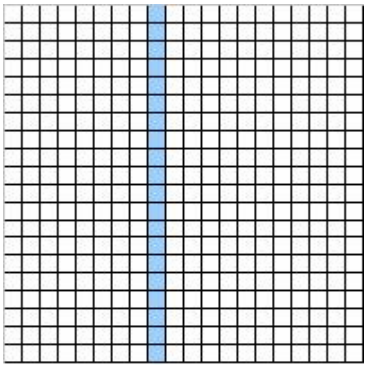


Figure 2.7: Locality, Column access for a 1000x1000 dataset

Data Chunking

As the application generally accesses parts of a data array, the higher dimensional data can be divided into small chunks [18, 19] and then stored on the disk. Chunks are used for improving the I/O access of data, since the data access is not fixed for each request. The chunk size can be tuned and optimized with I/O access. Even storing the previous example with data chunking doesn't bring significant performance improvements due to lack of locality in chunk data.

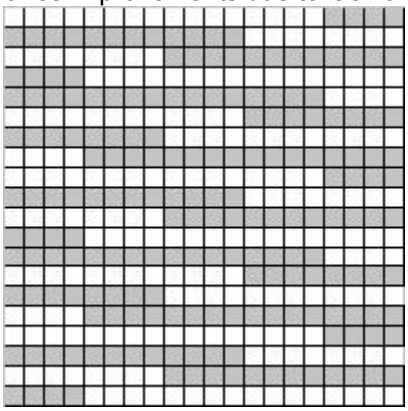


Figure 2.8: Row-Major Chunking: Lack of locality in one dimension

Instead of storing data in file in row-major order, the locality is exploited by taking some data from all dimensions and then storing the resultant block on the disk. The process is repeated over all parts of higher dimensional array to completely represent it in file. Note here that each block has locality in every dimension.

If the previous example of 1000x1000 array is divided into 10x10 chunk size then the block – column access of the whole array will result in to 100 seek and read operations in file. The result is a significant gain in performance over the previous data storage technique.

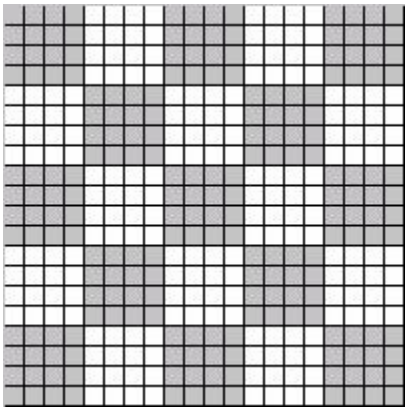


Figure 2.9: The Grey and White Blocks: 4x4 Array Stored in File in Row-Major Order

Data chunking is among the most widely used technique in scientific datasets storage [14, 15]. The notion of block size needs more attention here. It is a highly application dependent parameter, since the data is now read in chunks. Bigger chunk size results in less number of blocks on disk: compare the same example by 100x100 chunk size, resulting in fewer number of seeks and larger continuous block on disk. In contrast, the application might not need the whole block of data on each access that results in read of more bytes than required. The excessive bytes are wasted and higher cost is paid in I/O resulting in decrease in effective bandwidth.

The block representation of data sets has more opportunities to exploit parallelism inherent both in data and processing. Moreover, the chunk size has relation to the file system block size. Normally this access should also be a multiple of file system block size. This notion is explained in more detail in discussing the performance evaluation of Parallel File System.

In short, chunk size is a trade off among the conflicting goals discussed above but this is important to be aware of while designing the format and storage layout of scientific datasets.

Hilbert Space Curve Filling

The arrangement of data blocks on disk can further improve I/O performance if disk blocks that are accessed together are put close to each other. There are lesser number of seek operations on disk controller because of locality in blocks. The result is bigger I/O requests on each controller and seek overhead is minimized that gives better I/O throughput.

Normally these blocks are again mapped on the disk linearly. The previous argument again can be applied while defining an optimal layout of blocks on disk. Data array blocks row access will be more optimal than the column access, for a row-major access pattern.

Though the block arrangement is application specific, Hilbert Space Curve Filling [5] is used to map higher dimensional data points on a single dimension (storage order) preserving locality. This way the locality in space is maintained on disk.

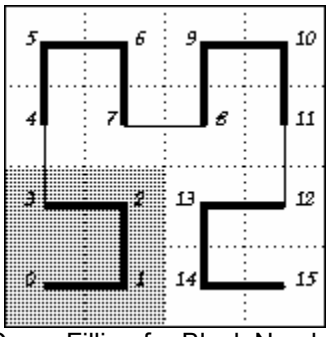


Figure 2.10: Hilbert Space Curve Filling for Block Numbering (UCSD Lecture Notes)

The important notion here is to put those blocks more close to each other that are accessed together. In other words predicting the layout of blocks and storing them on disks in form of small clusters. This gives opportunity to File Systems and I/O subsystem to exploit read-ahead optimization and there is more chance that new blocks will be found in cache. Scattered blocks on disks have more latency than the compact layout.

Indexing Schemes for Higher Dimensional Data Sets

The storage techniques of data chunking and clustering exploit locality in data blocks. An efficient data-structure [16, 17] is required to retrieve part of data from these blocks.

The index structure is widely used in database community. For example B^+-Tree[] are used for locating table rows on secondary medium. Each data request is looked up against an index structure which specifies its location on disk.

These data structures cannot be employed directly to higher dimensional data sets. These data sets generally represent properties both in space and time therefore an indexing scheme based upon spatial index is required. There are several schemes proposed for this R-Trees [20], Quad Trees, KD-Trees etc.

These index structures itself consume lots of disk space and becomes a bottle neck when many processes are accessing the same data sets. Therefore an important notion for us is to use an index structure that fits easily in a process memory and could easily be replicated to a process that is using it.

Processing Out of Core and In Core

Generally, these data sets are big enough to fit in to the main memory of a single machine therefore out of core mechanism are required to efficiently process these datasets. Here in order to support the range queries and management of metadata becomes important. We didn't want the metadata to be accesses through a fixed set of processes so that they become bottleneck. Secondly the metadata shouldn't be that much big to consume a major portion of process main memory.

Available Libraries for Scientific Data Storage

There are many available scientific data formats along with their library support used by the scientific community. The most notable are the HDF [14] and netCDF [15].

We primarily focused upon the HDF library. The HDF support hierarchical storage of data just the way the directory tree of a file system is maintained on a Unix File System. Each hierarchy describes a particular part of a data set. Many complex structures can be described through this file format.

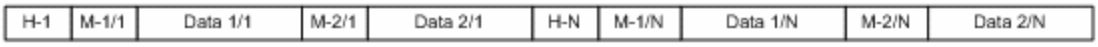


Figure 2.11: HDF Storage Format
H-N: Header information of N^{th} data segment
M-N/P: N^{th} Meta-information of P^{th} data segment

The down side of this format is that the metadata is not stored at one place in file; instead it is distributed with the hierarchies. Many small I/O requests must be issued to read this meta-information. The small I/O requests are expensive.

Secondly, the use of existing data formats binds application with specific libraries that was not needed at this stage. Subsequently, making efficient index structures on these formats require quite in depth understanding of its API and metadata structure storage and processing.

Moreover, we have a dominant ratio of read requests than the write ones. The datasets must not be densely populated. We wanted to support the remote visualization of data where the whole meta-information of dataset must be shipped to client machines that can be used to explore and navigate the dataset. Therefore it was decided to develop our own data format rather than using the existing libraries.

Chapter # 3

Performance Evaluation with High Performance I/O Library: MPI-I/O

I/O Bottle Neck

Generally, the scientific datasets are large enough for a single machine and a cluster of workstations or SMP machines are used for processing. The logical choice is to employ Message Passing Interface, MPI-2 standard [7], over such infrastructures.

The performance gap between the existing modern microprocessors and I/O subsystems naturally underlines the bottleneck at the I/O side. The first step towards processing these datasets is to study their I/O behavior under constrain of different application access patterns. The ROMIO [21] implementation of MPI-2 standard is selected to quantify File Systems performance that stores these datasets.

The goals here are to:

1. Understanding the working of MPI-I/O
2. Quantification of I/O performance over Parallel File Systems: Though results are reported with Distributed File System to validate the I/O operations
3. Determining the I/O block access size
4. Scalability of I/O performance

Distributed Vs Parallel File Systems

It is important to understand the distinction between the Distributed and Parallel File Systems. In Distributed File System distinct clients access the file systems from different locations. A single server can export different file systems. The file system normally resides over a single server but could be mounted by different clients. Since a single server has an inherent bottleneck, file system replication can be used to increase I/O performance. The major technique to boost performance and decrease I/O latency is to heavily rely on client side file replication and server side caching. The concurrent access to files is controlled through sophisticated locking mechanism of file system.

The most popular examples of these file systems is AFS [22] and NFS [23]. It is important to understand the goal of Distributed File System to have concurrent file access, tight control over Access Control List (ACL), caching and disconnected file operations. The intensive caching operations put significant overheads, like state management, when many processes are accessing the same file of the file system. The file system access remains very general in nature.

Moreover these systems target more on high availability with multiple users working with small files that can easily be replicated. They are not optimized for larger data access.

In contrast, the Parallel File System is the one where there are many I/O servers working together to give a single coherent image of a single file system used by many clients. Among the salient features a parallel file system is the concurrent large data access, normally greater than 64KB and delivering high I/O performance to its users.

The most notable examples of Parallel File System are Parallel Virtual File System [24]: for interactive clusters), General Parallel File System (GPFS) [25] and Luster [26]. GPFS is a proprietary file system by IBM where as PVFS2 and Luster is open source software initiatives.

From this onwards, Parallel File System will be discussed and performance results of Distributed File Systems are only reported for academic reasons.

Data Flow in Parallel File System

In order to interpret the results of MPI-I/O through Parallel File Systems it is important to first understand the internal working of a Parallel File System. This effort is mainly focused on the data flow that passes from a user application to various system interfaces like network, DMA, etc finally to underlying storage device. The data flows of Parallel Virtual File System (PVFS2) and General Parallel File System (GPFS) [27] is studied and can be generalized for other Parallel File Systems.

The file system is implemented as a number of software components on a cluster of machines. Please see [27, 28] for detailed process and daemons that describe GPFS and PVFS working. Generally, the File System is configured and installed with system wide coherent cache that takes up a predefined size of memory resource on each node. The normal size is approximately 50MB and cannot be changed. This defines the notion of available "page pool" on each node. ***"File System Block Size"*** is another notable term: it is the unit of I/O access from the I/O servers to application node. Since it is also an installation time parameter and Parallel File Systems are intended for large data access its size ranges from ***64KB~512KB***. *File Locking* limits performance and scalability of Parallel File Systems on large clusters. The concurrent access is handled differently. GPFS supports fine grained locks at bytes level through a central Token Manager whereas PVFS2 puts consistency semantics on user applications.

Generally, two techniques are used for boosting I/O performance:

1. Write Behind
2. Read Ahead

Once a write is initiated at a client node roughly following list of operations take place, for GPFS only:

1. The GPFS daemon contacts the token manager to acquire the specified range of tokens
2. Once tokens are acquired, background daemon makes space in "page pool" for requested blocks.
3. The data from client memory is copied to "page pool" and at this point the write is complete from client perspective.
4. The data blocks remain in "page pool" until blocks are scheduled for writing to disks
5. Since blocks are required to transfer to I/O servers they need to pass the network interface. The blocks are further divided into IP packets that are transmitted through switch over to I/O server.
6. At the I/O server these packets are arranged in blocks and disk space and data structures are allocated for storage and DMA request is setup to transfer these blocks to disk. Here one I/O server is satisfying more than one client node therefore resources at I/O server are scarce if rate of I/O requests is more than the drain rate at servers.

Similar procedure but in opposite direction is applied for reads. The only difference from write is that GPFS detects the application behavior of reads and part of its optimization issues read request that are not still issued by client.

Irrespective of a specific File System implementation, if the client is accessing data with a specific pattern then parts of further read requests can be fulfilled by *Global Cache* and I/O performance is boosted. Random access patterns are hard to predict by File System.

Performance Guidelines for Parallel File Systems

The data flow of Parallel File System gives quite in depth understanding of various I/O performance parameters [27, 28]. These guidelines are as follow:

1. The File System Block Size is the I/O unit access for these systems. The application I/O access should also be of multiples of this size. Lets say a particular system has a block size of 256KB but the application access data in multiples of 128KB. At file system each block containing these data items will be fetched in blocks of 256KB. The result is a plenty of 50% waste of data access that will ultimately increase the I/O time and subsequently decrease the available bandwidth to application.
2. The data access should be aligned to block size. Non-aligned data access results in fetching more data blocks than actually used by application.
3. Data access should be made as large as possible: this gives file system privilege to employ read ahead optimization.
4. The file striping factor should also be made multiples of underlying RAID configuration of disk arrays. This enables the file system to put equivalent load on disk arrays.
5. The stride access should also be used carefully because bigger strides might results the data to be placed on same disk controller. It will hinder the parallelism available in blocks stored on disks.
6. It better to assign different file regions to processes to operate on. Highly interleaved access at block level results in overhead of token acquiring and shipping the data from one client machine to other.
7. The use of locking mechanism should also be avoided because it limits the scalability of applications.
8. The hints mechanism must be used to convey as much information as possible to underlying Parallel File System.

Efficient use of File System through MPI-I/O:

MPI-I/O was developed initially at IBM in 1994. Later it became the industrial standard in MPI-2 specifications. Among the major goals of MPI-I/O are: Portability, MPI style syntax, efficiency of I/O access and file interoperability in the domain of High Performance Computing.

The most important use of MPI-I/O is efficient engagement of I/O servers on each request and the way these requests are issued. The Parallel File System is optimized for larger I/O requests rather than working with large number of smaller I/O requests. Therefore smaller requests must be grouped together to form a single bigger I/O request.

The notion of packing small I/O requests in to a single big I/O request heavily based upon the use of MPI-2 Derived Data Types. Here it is important to further classify the source data location in file and destination data in memory. The possible combinations of these locations are:

1. Continuous in Memory and Continuous in File
2. Continuous in Memory and Stride in File
3. Stride in Memory and Continuous in File
4. Stride in Memory and Stride in File

MPI-2 standard defines a rich set of type constructors that are used to build new data types that can encapsulates the above complexities of data layout patterns. These type constructors are ***contiguous, vector, indexed, struct, subarray*** and ***darray***.

Moreover, it is hard for File System to predict the application access in case of processes issuing independent I/O calls. MPI-I/O use the notion of Collective I/O operation through which global information of I/O access is made. This technique is heavily effective if the file access location among participating processes is interleaved.

In short, Thakur [29] classified the way MPI-I/O calls are issued into four levels:

1. Level 0 Many Independent, Contiguous Requests
2. Level 1 Many Collective, Contiguous Requests
3. Level 2 Single Independent, Non Contiguous Request
4. Level 3 Single Collective, Non Contiguous Request

Here it is also important to understand the difference between a bigger contiguous request and smaller I/O request. If the I/O is bigger and continuous then “Level 0” and “Level 2” requests are considered to be equivalent. Since the data is brought in bigger continuous chunks therefore no need to put extra effort to combine the independent requests. Similarly, same argument applies to “Level 1” and “Level 3” type of requests.

Normally, the recommended way is to use the “Level 2” and “Level 3” requests but the paper doesn’t define the notion of at which size of I/O request should be transformed to these levels. Here a bit of testing with different call combinations is required to come up with performance measurements to see which combination is more suited for it.

MPI-I/O Optimization Algorithms:

ROMIO, MPI-I/O implementation, uses two algorithms to optimize the I/O access among participating processes.

- Data Sieving
- Two Phase I/O

The small independent non-contiguous I/O calls are grouped together to formulate a bigger I/O call at the MPI-I/O layer. MPI-I/O allocates a temporary buffer to hold the extra data in memory. The extent of buffer depends upon the start and end address of the small I/O requests that are combined into the bigger call. The memory is limited therefore it might be the case that extent of call exceeds even the available memory. MPI-I/O provides a programmatically tunable variable, ***“ind_rd_buffer_size”***, to control the extent of this buffer. If extent of request is greater than buffer than the whole process is repeated number of times of temporary buffer until the I/O extent is fulfilled.

The idea behind Data Sieving is that process read more than requested with the assumption that strides/holes in request are smaller than the overall data request. The algorithm works well if the stride in I/O access is small and a larger contiguous access out performs the overhead involves in reading the un-wanted part of data.

But, performance is compromised if the holes are bigger in each part of access. Since data is read contiguous, the result is reading in more un-wanted data and the effective bandwidth available to process is decreases. Similarly, copying of data from the temporary buffer to its destination is also imposes bandwidth compromise due to this extra copy.

In short, in using independent I/O calls the length of stride in I/O access should carefully studied and performance measures with different temporary buffers size should be taken to decide the optimal I/O call for independent calls.

The second algorithm used by MPI-I/O for optimizing I/O access is the collective I/O calls. As the name indicates Collective I/O calls require participation of all processes in I/O that has opened the parallel file. This I/O technique is highly effective if the access pattern of processes is strictly interleaved. Instead of accessing the data through a combination of seek and read requests MPI derived data types are used to set the appropriate regions of file to processes.

Collective I/O operations are used for the following two scenarios:

1. Interleaved Data Access in File
2. The performance of underlying File System doesn't scale with the number of processes

As the name indicates he two phase I/O operates in two distinct phases but in lock-steps. The first phase is the I/O phase in which processes participates and decides whether the participation of all processes can benefit from the collective I/O call or not. If there is no gain than each process reverts to independent I/O calls discussed above. Otherwise, the result is the division of I/O extent region and its assignment to individual processes. Next global ownership information is made to check which part of request is assigned to which process. The file regions, file domains that are assigned to each process actually do the I/O in this phase.

The next phase is the communication phase. The data needed by each process that is owned by the other process results in asynchronous read request by the source process. The owner process communicates the data to other processes in this phase.

Similarly, this technique also relies on the use of temporary buffers that are used in both phases. Therefore the extent of I/O request might be bigger than the temporary buffer. This forces the processes to operate at the extent of collective buffer, ***"cb_buffer_size"***, at each step. Note here that each process if it is not actually reading the data in first phase must participate in the overall procedure. This is the requirement of collective calls.

The MPI-I/O also uses a programmatically configurable variable to set the numbers of actually participating processes in the I/O call. This is a very useful parameter especially in those circumstances where the I/O bandwidth doesn't scale well with increasing number of processes.

MPI-I/O File Views:

The File View represents the regions of file that are visible to each process. The basic unit of data access from file is defined in terms of "etype" that could be a basic MPI type or a derived type. All I/O from file will be performed in this unit.

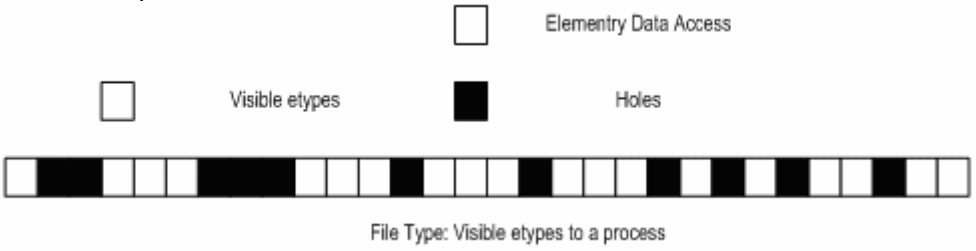


Figure 3.1 MPI-I/O File Views

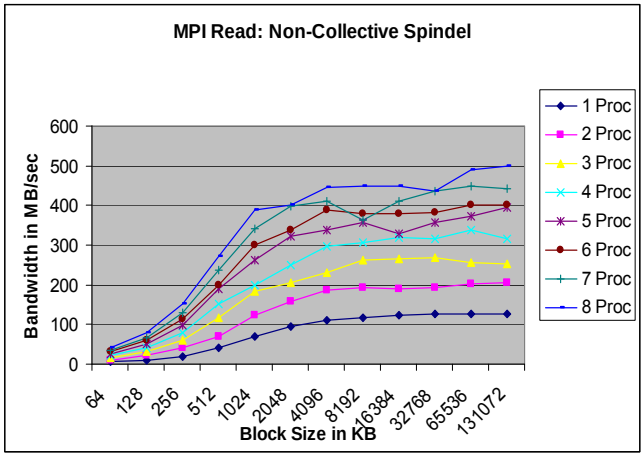
Similarly, "ftype" that represents the File Type is also a MPI type that represents the region available to each process. This view could be contiguous or with holes as mentioned before for stride access case.

The File Views are implicit hints to MPI-I/O layer which data blocks will be needed by application over a series of I/O calls. Moreover, "etypes" makes sure that data is read in a specified structure as opposed to raw stream of bytes.
The key in getting better performance from I/O is how efficiently these views are assigned to participating processes. Each I/O call is preceded by setting the appropriate view for each process.

MPI-I/O Hints:

Lastly, MPI-I/O also exposes another interface to pass the application access patterns to underlying MPI-I/O implementation and Parallel File System. The available hints varies from one system to other.

Performance Reporting with PVFS2 on Spindel



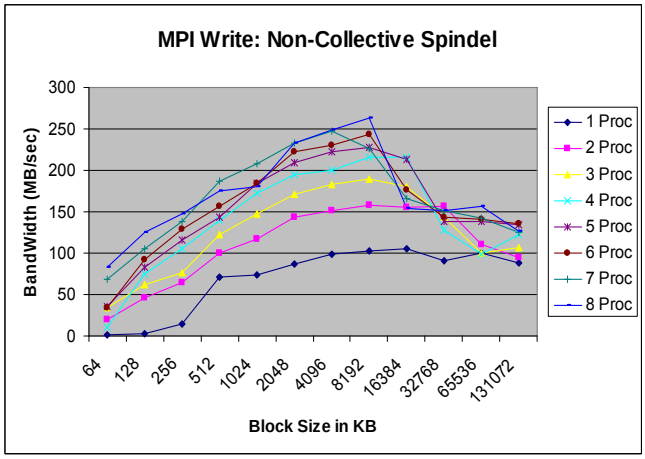
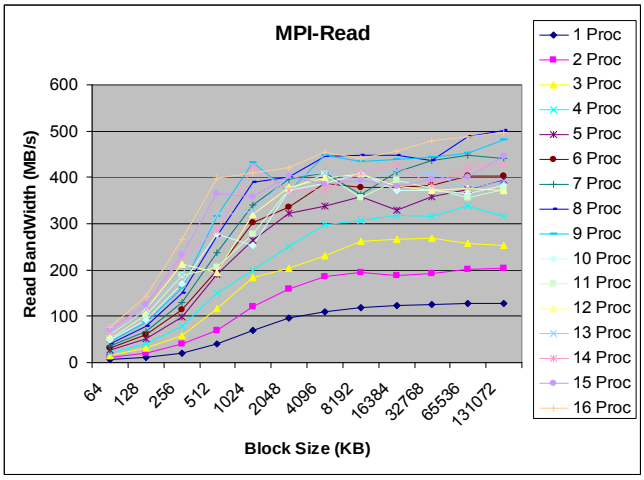


Figure 3.2: MPI-I/O Block size Performance



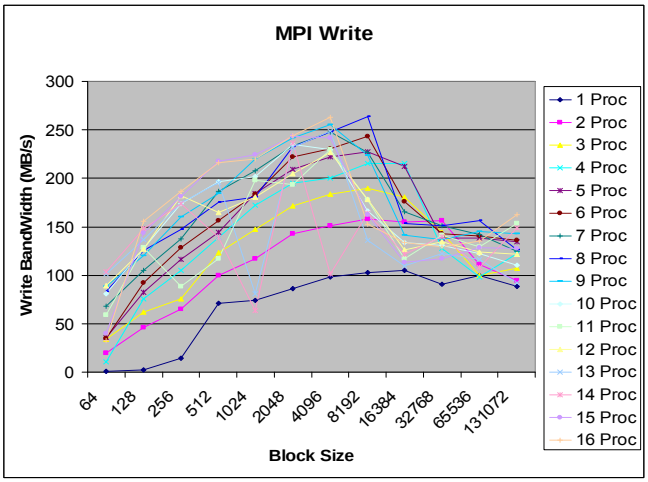
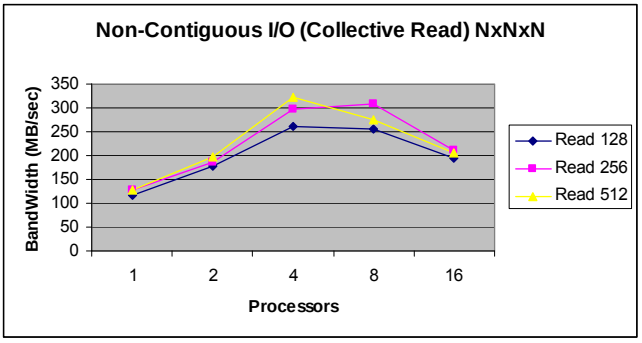


Figure 3.3: MPI-I/O Block size Performance at higher processors count

Add some text description here



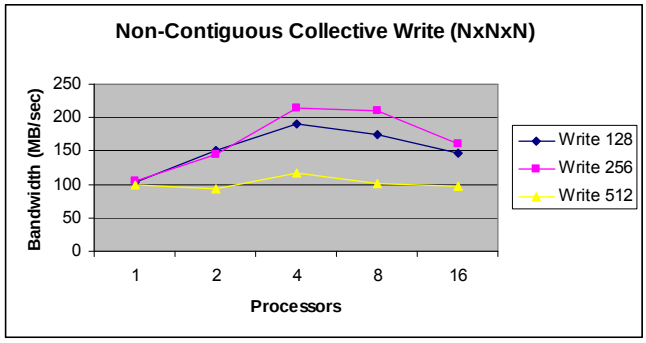


Figure 3.4: Non-Contiguous I/O Performance on Spindel

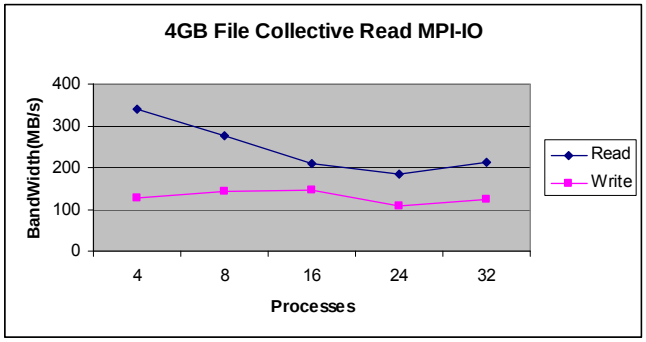


Figure 3.5: Large File Performance on Spindel

Chapter # 4

Requirements of Framework

The main goal of the framework is to develop an Application Programming Interface **(API)** through which users can navigate, explore and process any part of dataset. The API must provide valid implementation and the support for most common operations with flexibility of adding operations from users as long as they follow the syntax and semantics of the framework.

Dataset Usage Scenario

Since scientific datasets are huge, a close study of its processing reveals many key operations that are generally applicable to almost all of the datasets. The sequence of operations could briefly be stated as follow:

1. User selects a single or group of datasets from a set of given pool
2. A specific area of interest is selected from the dataset
3. User applies some filtering operations on the selected region
4. The filtered data undergoes further processing
5. The resultant, data product, might again be saved to disk or supplied to other modules like visualization for further analysis

With these generic guidelines and further discussing with the advisor, refined requirements were outlined that must be supported by the framework.

Dataset Abstraction

Though the framework can handle 3D datasets and could be generalized to 2D datasets, primarily it focuses on ***TeraShake Ground Fault Simulator*** [35] datasets. The datasets are the Simulation of seismic waves generated by large, but geo-physically plausible, earthquakes that occur in the on southern San Andreas Fault. These simulations produced 43 Terabytes of data on **240 IBM Power4** processors for 5 days and contain 2000 snapshots @ 2.1×10^9 bytes and are stored and accessible at San Diego Super Computer Centre through Storage Resource Broker, SRB interfaces.

The one time snapshot of the dataset represents a domain of **400x3000x1500** grid points. There are total 1.8×10^9 grid points in the dataset and each point represents a 3D velocity vector. This makes up more than **1 Gigabytes** of memory. The details of managing these datasets must be encapsulated by framework.

In short, the datasets are big enough to fit into main memory of a single machine the framework must support data abstraction to its users. The data arrays stored in dataset will be accessed by the users as they are residing locally on their machines. Behind the scene, the framework must take care of locating the required regions of data arrays, staging them from disk to memory and supplying users with data.

Load Balancing for Sparse Datasets

It is not necessary that all points in the domain represent a data element attached to it. This imposes holes in the dataset representation and raises load balancing issues. The naive techniques of decomposing the Global data arrays will result into unevenly loaded processors. It can severely affect performance with increased processing times and less throughput by framework. The load balancing issues must be addressed by the framework.

Support for Range Queries

In general users are not interested in all parts of the dataset in one single access. The navigation process gradually covers parts of the dataset. Therefore the framework must have support of Range Queries. The range query could be supplied to a single or set of datasets. The interested region is specified by a “View Window” that represents a minimum bounding box in N-dimensional space.

Multi Block Structure Processing

The individual dataset file is divided into chunks and the chunks are allocated to processors for load balancing. The support for geometric operations over a sub-region of domain is desired that could span over more than one block of data in space. The framework must also support these operations over any contiguous region in domain.

Support for User Defined Functions

As mentioned previously, the framework must support a set of core operations that can be applied for data filtering and processing. Users can select and de-select these set of operations during the program execution. Since it is impossible to anticipate the data flows that a scientist may require on the fly, user defined functions must also be supported by the framework.

Data Operation

In addition to various user defined operations the framework must also support data operations in well defined phases of execution. These phases are broadly classified into following categories with set of functions for each phase in execution:

1. Filtering
2. Processing
3. Data Transfer
4. Output

In short, all the performance and resource management issues for the retrieval and processing of these datasets is left on the framework providing users with a solid platform to carry out their analysis on the data.

Chapter # 5

Design and Implementation of API:

The support for required functionality is added incrementally in the API therefore an efficient strategy was required to use uniform interfaces throughout the implementation to accommodate changes occurring during the process of code implementation.

The Object Model

API Abstractions

The first step in designing the API was to incrementally refine the object model. The incremental approach is based upon prototyping and testing. The functionality is tested in isolation, verified and then made part of the object model.

The Dataset

The dataset is the key object of the framework, rest of the objects is added to support the functionality around this abstraction. The dataset represents a complete domain in space with many measurements of interest: velocity, temperature, pressure.

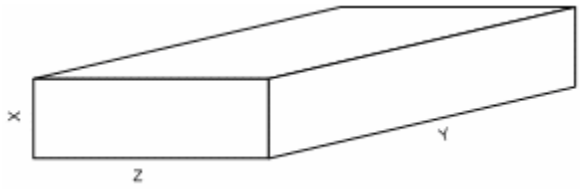


Figure 5.1: Domain of TeraShake Dataset: (x, y, z), (400x3000x1500)
Contiguous view as single entity

This object exposes a number of routines for the retrieval and management of data. The dataset object key features are:

- Metadata Management
- Data Blocks Retrieval and Caching
- Support for 3-dimensional Regions
- Call back support

Meta Data

The dataset is composed both of metadata and data arrays. The metadata defines the type of data and holds its properties. The dataset object abstracts the handling of both forms of data.

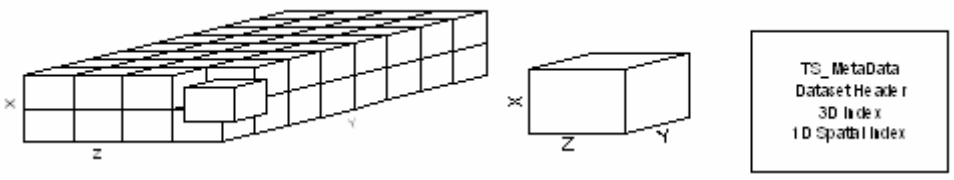


Figure 5.2: Chunked view of dataset (left), data-block: 16x30x30 (centre), metadata object (right)
Meta-information describes domain, data-blocks and spatial indexes

Since the dataset is sparse, the meta-data contains the following information:

- The dimensions of domain
- Total number of blocks in dataset
- Dataset properties: threshold, creation date
- The dimensions of data-blocks and associated components: velocity, pressure
- The location of each data-block in domain

Moreover, 3D space locations provided by meta-information is used to construct a one dimensional index to speedup the data-block lookup.

Data Buffer

The data arrays of dataset are divided in chunks of n-dimensions. The chunks define a unit of I/O and communication. Therefore at the bottom level a uniform abstraction was defined to hold the data blocks of datasets that represent the grid points in domain. The data buffer is a multi-dimensional abstraction that can hold a data block of n-dimension with its components in **X, Y, Z and T dimension**.

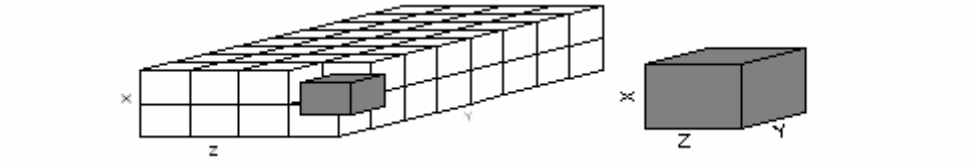


Figure 5.3: Data Block in dataset

Each point may further be attached with a physical quantity that could be a scalar or vector in n-dimension. This functionality is encapsulated in this data structure and is widely used in other parts of the API.

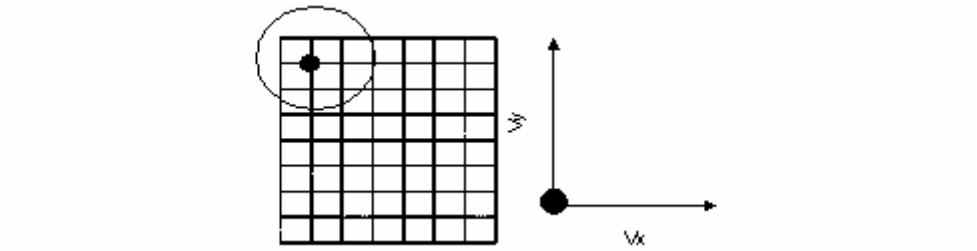


Figure 5.4: Single plane of Data Buffer (left)
Single Grid point represents 2 velocity components (right)

The layout of a single block is chosen to be contiguous because it helps in exploiting locality in memory and block transfer in communication and I/O phases. This abstraction also holds the meta-information of the block in the dataset. Helper routines and callbacks supports are added to blocks.

Region: Multi-Block Structure

The multi-block structure is another abstraction to define a Region in space: the dataset. This abstraction is build on top of the Data Buffer abstraction and holds a collection of blocks. Note here that the collection of blocks in this abstraction can have different dimensions. Since the Region is contiguous in space, there could be reference to blocks for which data blocks are missing.

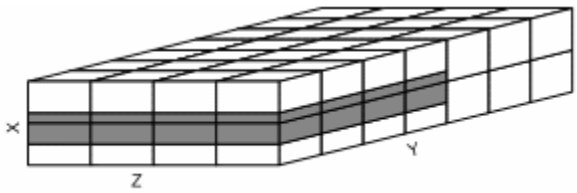


Figure 5.5: Multi Block Region

Meta headers are added to this object to define the region in space. The meta-information of the blocks is also available at this level.

Call back Support

The callback object adds user defined routines to a dataset. The callback routines must conform to the argument constrain of the framework. Primarily, following types of call back routines are supported:

1. Vector (N-dimensional component with each grid point)
2. Scalar (Single component associated with each grid point)

The API supports the above callbacks at different phases of processing. For the current implementation the callbacks at the dataset level are checked and verified. There are also a set of routines that are implemented as part of framework that a user can associate with a dataset.

The framework specifies the following phases of callbacks:

- Data Threshold
- Data Processing

The basic operation of callbacks is to support stencil operations on each grid points. The core functions are to calculate partial derivates of velocity components at each grid point: divergence and curl of vector fields. The current implementation does not support ghost cells between neighboring data-blocks.

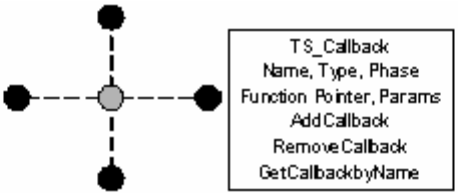


Figure 5.6: Stencil Operations on Grid Points (left)
TS_Callback Object (right): Properties and Methods

Each callback is assigned a unique name by user that could be referenced in a user defined query (details are in Query Engine section).

Dataset Manager

Normally the users will not be manipulating a single but a group of datasets in one operation. All datasets opened in a single session are managed by this object. The dataset abstraction for a single dataset is also available and a range query can span multiple datasets. The composite query execution (details are in Query Engine section) spanning more than one dataset is also managed by this object. All dataset accesses are routed through this object.

Figure 5.7: TS_DataSetManager Object: Properties and Methods

Object Execution Log

Similarly, log objects are provided within API to record the execution of Dataset and Dataset-Manger objects. These log files record the number of requests received by each object and the other performance measurements like I/O time, query execution time and processing time. The log files are used for debugging the API and reporting performance results.

Parallel Implementation

Query Engine

Instead of exposing the fine details of dataset abstractions to users, a query object is implemented to expose the datasets for data analysis and user defined modules like: visualization. This is defined as an interface between users and framework.

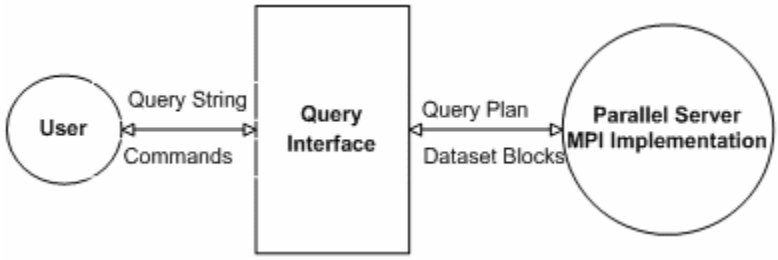


Figure 5.8: The Query Engine: Parallel Server and Remote Users

The primary goals here are:

- Easy to use interface for naive users through SQL style queries
- The management of available datasets: TeraShake 2000 Snapshots
- Range Query support:
    - Over a single or group of datasets
    - Data blocks retrieval in range query region
    - Predicate support for associated callbacks on datasets
    - Defining the dataflow over user defined processing phases

The parallel implementation is divided into two distinct phases:

1. Query Specification and Execution
2. Parallel I/O Strategy

Query Specification and Execution

The query specification is based upon SQL style constructs and user submits a query string to the query interface. The query execution is divided into three phases [30]:

Figure 5.9: Phases of Query Execution

1. Parsing
2. Planning
3. Execution

Query Parsing

#define TS_QUERY "SELECT * WHERE ('Curl_CB' > 7.50e-05 AND 'Divergence_CB' < -1.60e-05) AND 'User_CB' > 0.017 "

A query parser was implemented to accept user queries. The Query Parser parses the query string and identifies tokens. The result from this module is a parse tree that is input to query planning.

The parse tree contains two types of nodes: *Operator* and *Operand*. The type of operator defines the associated operands in parse tree. The unary, binary and territory operator has one, two and three operands associated with it.

The predicate support for callback functions is added through the *Relational Operators*: “>”, “<”, “!”. The user callback names are separated from query string and then are checked against the registered callbacks with the dataset.

```
string query;
TS_Parse_Node node;
TS_Parse_Node_List list;
while(query.length > 0) {
        symbol = ReadNextSymbol(query);
        if(symbol == Operator) Then
                node = GetNewOperatorNode(symbol)
                if(stack.empty then stack.push(node)) Then
                elseif (precd(node.symbol > precd(stack.top)) Then
                        stack.push(node)
                else
                        list.push_back(stack.pop());
                        stack.push(node);
        else if(symbol == Operand)
                node = GetNewOperandNode(symbol)
                list.push_back(node);
}
```

Figure 5.10: Query Parsing Algorithm

Query Plan

The query parse tree is processed to convert it in to the query operation graph. Each node of the operation graph represents an operation. The operation node could be of any type:

- Unary Operation
- Binary Operation
- Leaf Operation

The operation node degree represents the dependency of the operation on its children. Leaf nodes comes at the lowest level hence do not have any children therefore they are the independent nodes in graph. The unary operation nodes only have one child operation. Similarly the dependency of operations increases with degree of operation node.

The supported operations from the query engine are:

- Logical Operations
    - AND
    - OR
    - NOT
- Callback Operations

The edges of the graph represent the flow of data from the child node to parent. Unary node will accept a single set of data. Similarly binary and higher degree node will accept two and more than two set of data.

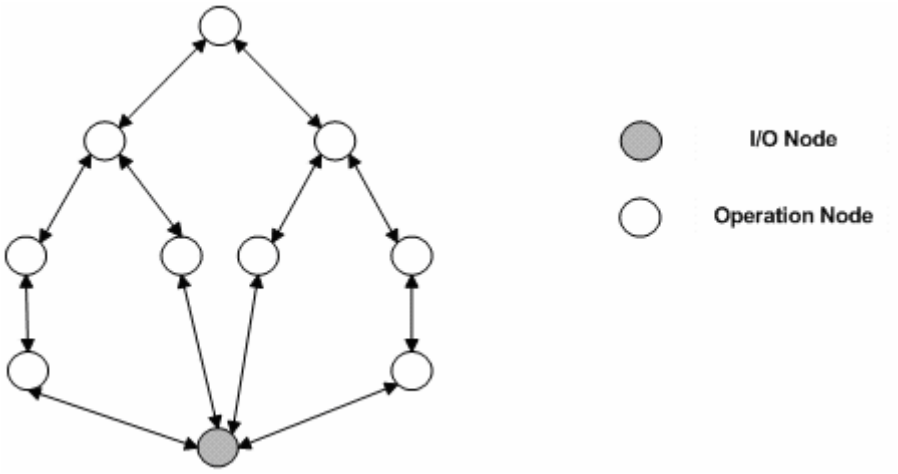


Figure 5.11: The Operation Graph

Lastly, the view is specified separate from the query in terms of bounding box in 3D. It is added to the operation graph as a leaf node and all the leaf nodes are changed to unary node and the operation node is complete at this stage with one leaf node.

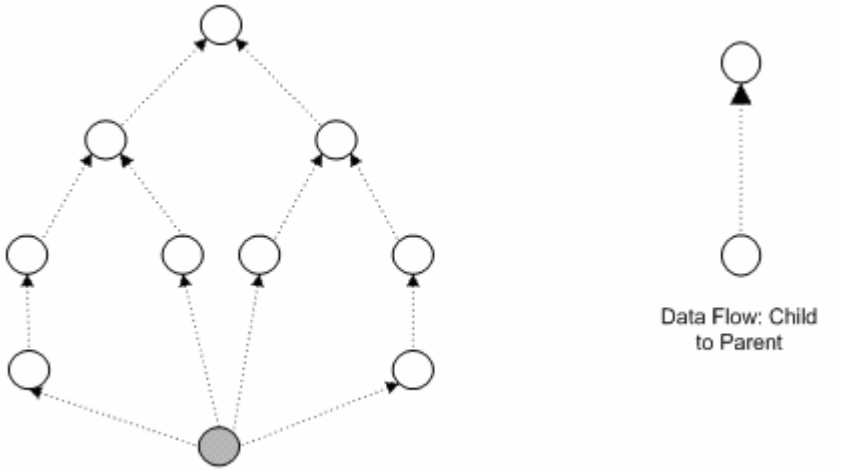


Figure 5.12: The Data Flow in Operation Graph

```
if(node.type == TS_OPERATOR) Then
        if(not stack.empty()) then
                op1 = parsestack.pop();
                op2 = parsestack.pop();
                if(op1 or op2 == TS_NAMED_OPERAND) Then
                        op_node = GetNewOperationNode(TS_NAMED_OPERAND);
                        SetOperationParameters(op_node, op1, op2);
                        SetOperationParamValues(op_node, op1, op2);
                        operationstack.push(op_node);
                End
        else
                op1 = operationstack.pop();
                op2 = operationstack.pop();
                op_node = GetNewOperationNode(TS_LOGICAL_OPERATOR)
                op_node.childlist.push_back(op1);
                op_node.childlist-push_back(op2);
        End
SetLeafOperationNode(root, view);
```

Figure 5.13: Query Planning Algorithm

Execution of Query Plan

The operation graph preserves the data and operation dependency in query execution. The query is executed by initiating a post order traversal of graph. The operation node can only be executed once all of its children operation nodes are executed therefore post order traversal is the logical choice of implementation.

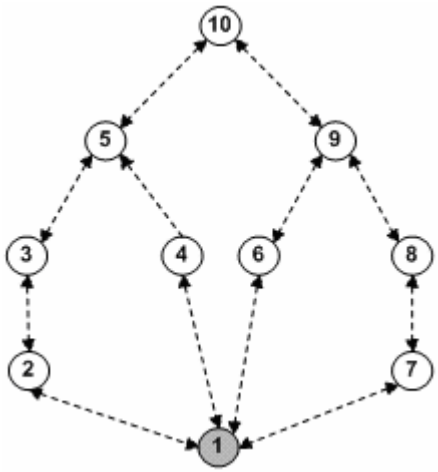


Figure 5.14: Operation Graph: Execution
Numbers represents sequence of operations

Each operation node stores the required information needs to execute the operation. After the operation execution conformity test of predicate is performed to check whether the operation has passed the condition with which it is performed.

```
TS_3DFBuff scalar_buffer, vector_buffer;
TS_PostOrder post_order(operation_graph_root);
TS_Operation_Node op_current, operands[];

op_current = GetNextOperationNode(post_order);
While(op_current) Begin
        operands = GetDataFromChildren(op_current.childlist)
        if(op_current.type == TS_OP_USER_DEFINED) then
                for each dataset.cached_blocks Begin
                        TS_CallBack call    = dataset.GetCallback(op_current.name);
                        TS_3DFBuff  block = dataset.GetCachedBlock(operand.index);
                        if(callback.datatype == TS_VECTOR) then
                                callback.Function(block, vector_buffer);
                                result = verifyvectortest(block, operands.paramlist, operands.valuelist);
                        else if(callback.datatype == TS_SCALAR) then
                                callback.Function(block, scalar_buffer);
                                result = verifyscalartest(block, operands.paramlist, operands.valuelist);
                        end
                        if(result) then
                                AddBlockindex(op_current.index_list);
        else if(op_current.type == TS_OP_LOGICAL) then
                LogicalFunctionCall(operands.index_list);
        else if(op_current.type == TS_OP_IO)
                dataset.GetViewWindow();
                op_current.index_list = dataset.cached_list;
        op_current = GetNextOperationNode(post_order);
End
```

Figure 5.15: Query Execution Algorithm

Parallel I/O Strategy

After outlining the object model of the API the next step was to decide how to design the I/O interface for fetching dataset blocks in to memory. The chapter 3 outlines the performance results of MPI-I/O on PVFS and give design guidelines in formulating the I/O request.

Meta-Data Maintenance

In parallel I/O more than one process operates on the same file. The dataset cannot be accessed without the metadata. The metadata was explicitly designed to easily fit in to memory of a single process. This means the memory requirements of metadata is very small as compared to number of blocks it is representing.

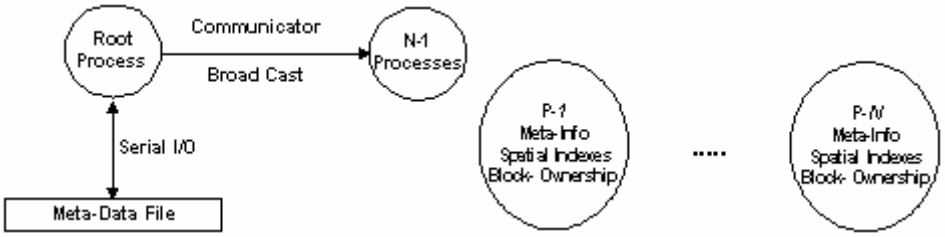


Figure 5.16: Meta-Data Management (left)
Meta-Data Replication in MPI-Communicator (right)

Though it depends on dataset size, on average metadata takes approximately ***100KB*** of memory. On a parallel file system that is even less than a single file system block size. Instead of using the parallel, serial interface is used to read in the data in process memory. The root process does this

job and the metadata is replicated over all remaining processes that are involved in opening the dataset.

Parallel I/O Implementation

The dataset block layout in file suggests that data blocks are contiguous in file around **169KB** in size. The data buffer abstraction ***(TS_3DFBuff)*** is used both for data processing, I/O and communication with other MPI processes. The user applications can access data blocks in any access pattern. Therefore, it is decided to implement the data access through different levels of I/O requests and check the performance gained through each technique.

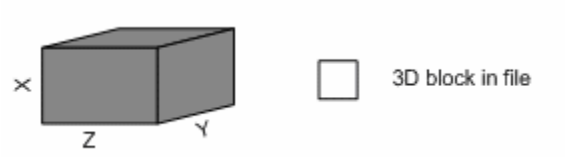


Figure 5.17: Data buffer Abstraction representation (*TS_3DFBuff*)

The I/O request is accepted by API in 3D bounding box of a region in space. The box has its extents in each dimension and the starting location tells start of region stretched to its extents.

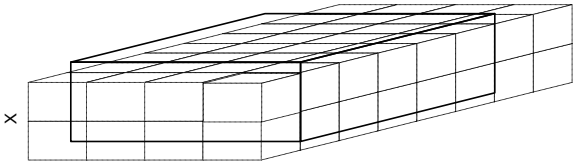


Figure 5.18: Dataset Blocks Access Scenario
Domain: (background), Query Window: (foreground)

The query window extents are used to identify data blocks in file. The overlap of window and blocks is calculated by meta-information that is replicated over MPI processes. It translates the 3D window into linear space of a dataset file. The difference between start and end block index gives extent of an access: it is the region accessed in dataset file that might not be contiguous.

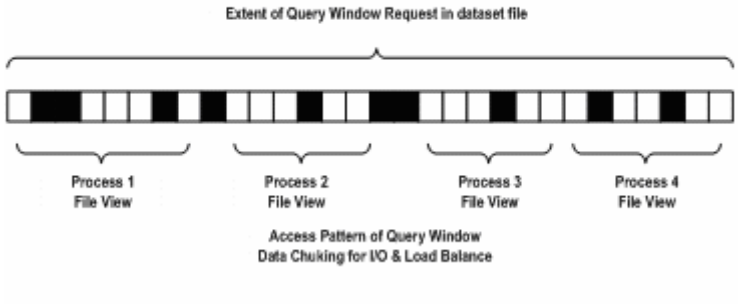


Figure 5.19: Query Window Mapping on dataset file (left)

The numbers of requested blocks could be randomly distributed (without pattern) for a query window. The block indexes are sorted for achieving locality in access. The load is distributed by allocating closely located blocks to each process. The processes have non-overlapping views of same dataset file. Figure 5.19, shows an example of such scenario.

The data buffer abstraction induces non-contiguity in memory of a MPI process. Inherently, the I/O mechanism has non-contiguous memory and file access. It is simple for POSIX style I/O calls. If each process has **"N"** blocks to fetch then ***MxN*** I/O calls are issued for a set of **"M"** MPI processes to complete request.

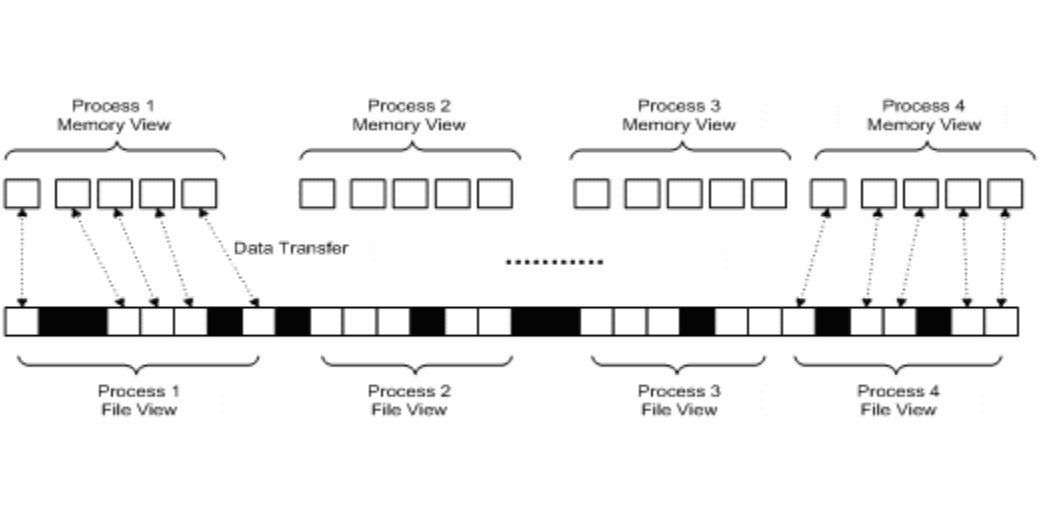


Figure 5.20: Non-Contiguous Memory and File Access

The ***"MxN"*** I/O calls can be reduced to "N", if each process use MPI derived types for packing Memory and File regions for each window request [29, 31, 32, 33]. To evaluate performance measures I/O requests are divided in two distinct ways:

- Packed I/O Request
- Unpacked I/O Request

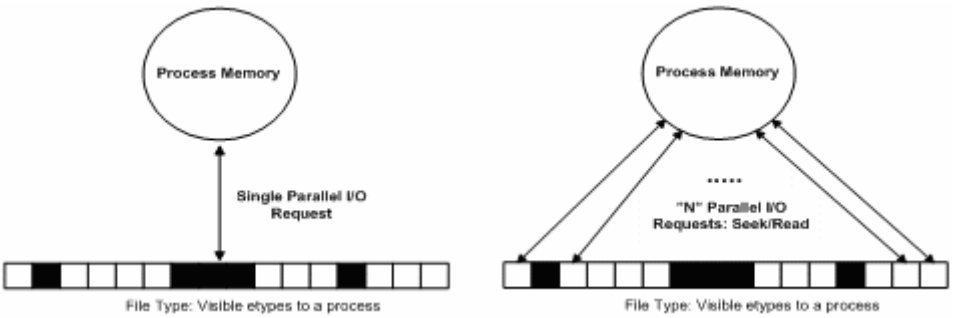


Figure 5.21: Packed I/O (left), Un-Packed (right)

Moreover these calls are further classified as:

- Independent I/O calls
- Collective I/O Calls

In independent I/O calls, there is no process participation in I/O and fewer chances are there for MPI-I/O implementation to optimize the access. In contrast, collective operations try to optimize access if there is an overlap of region requested by any two participating processes. Moreover, the associated overhead of File Pointer Management is eliminated if direct offset calls are used. The first step is to register a derived data type with MPI that represents a single block of dataset. It is unit of data access from parallel file and among participating processes. It uses ***"MPI_Type_contiguous"*** and all further operations are defined in terms of this type.

Since the displacement between successive disk blocks doesn't follow any pattern ***"MPI_Type_indexed"*** is used for packing file access. The indexed access is expressed in terms of dataset block. Optimal view setup in MPI-I/O effectively use Parallel File System read ahead feature. Thus only needed blocks are visible to a process in its view.

Similarly, MPI process memory is packed for one to one mapping with corresponding file regions. The data block memory regions are distributed in process address space. ***"MPI_Type_struct"*** expresses relation between successive memory regions. This special buffer is passed in ***"MPI_File_read"*** call.

```
TS_Box view_window, overlap_index;
List block_index, process_list;
Integer chunk, rank, displacement;
/* Mark block extents that view_window cuts in overlap_index */
CalculateOverlapRegion(view_window, overlap_index);

/* Meta information access for getting locations */
for(int i = overlap_index.minX; i <= overlap_index.maxX; i++)
        for(int j = overlap_index.minY; i <= overlap_index.maxY)
                for(int k = overlap_index.minZ; i <= overlap_index.maxZ)
                        if(index(i, j, k) exists) then
                                block_index.push(index(i,j,k);
block_index.sort();
chunk = block_index.size()/total_processors;
AdjustProcessorList(chunk, rank, block_index, process_list);

Allocate buffer_list
Commit MPI_Type_contiguous(block_elements);
for(iterate over process_list) Begin
        Allocate TS_3DFBuff buffer: buffer_list.add(buffer)
        Mark displacement for file access: process_list[iterator];
        Mark displacement for memory access: buffer
End
Commit MPI_Type_indexed for file access: file_map
Commit MPI_Type_struct for Memory access: memory_map

MPI_File_set_view(block_elements, file_map)
MPI_File_read_all(buffer_list, memory_map, 1, file_pointer)
```

Figure 5.22: Parallel I/O Algorithm

The log objects record timing information for I/O and computation times. Every MPI process has an associated serial file that records it execution and results of all requested queries.

Parallel Server Usage by Clients

Data Transfer to Client Machines

Though framework API is available for linking user applications, standalone implementation is tested for transferring data from cluster to remote clients. The client residing in the same domain of cluster is tested due to non-accessibility of computation nodes from outside. Another solution could be to execute a ***proxy*** on login node that routes traffic to backend nodes but due to time constraint only local clients is tested.

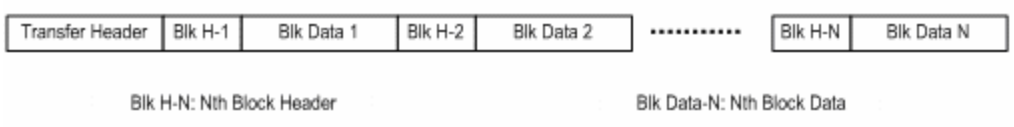


Figure 5.23: Data Transfer to Remote Clients

Figure 5.22 specifies a simple protocol for data transfer. The header information is transmitted in beginning followed by a stream of headers and data for individual blocks.

Each MPI process is equipped with additional Pthread that takes care of data transfer. The communication between MPI process and Pthread happens by queues. The MPI process after competing I/O puts the references of data blocks in Pthread queue. The queue objects have complete information regarding client and data.

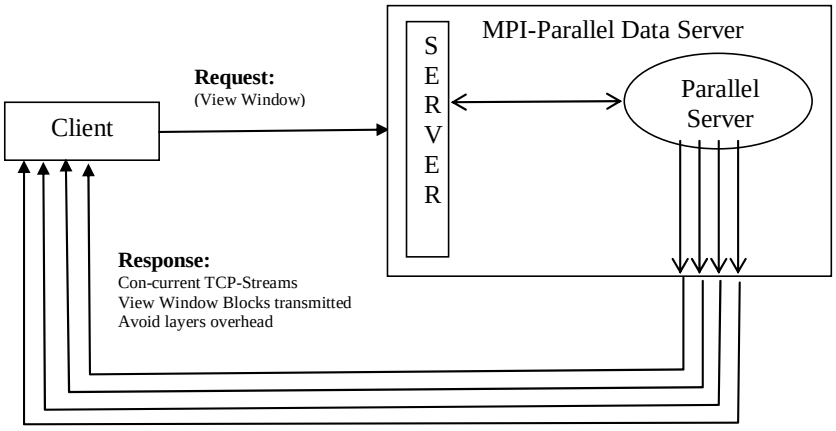


Figure 5.24: Parallel Data Transfer to Remote Clients

Similarly, the client is a multithreaded application and requested regions of file are owned by separate MPI processes. Each process that owns the data establish a separate TCP/IP stream [34] with client and transfer starts from Server to client machines.

Examples: Code Snapshot

The user programs can access the framework through direct function calls to its Application Programming Interface, API. A small set of helper routines are implemented to abstract away the direct access to object model. The following steps are needed to use the API:

1. Include TeraShake header file
2. Open the desired dataset frame
3. Add or Remove Callbacks
4. Formulate a Query String
5. Execute Query

```
#include <mpi.h>
#include "ts.h"

Int main(int argc, char **argv)
{
        MPI_Init(&argc, &argv);
        TS_Init(MPI_COMM_WORLD);
        TS_OpenDataSet_All(handler, "/home/pvfs/fgmir/T3/02270", MPI_COMM_WORLD);
        if(handler < 0) {
                cout<<"DataSet Cannot be initialized : Handler = "<<handler<<endl;
                MPI_Abort(MPI_COMM_WORLD, -10);
        }
        TS_AddCallBack(handler, "Curl_CB",        TS_CALLBACK_THRESHOLDING,  TS_VECTOR,  curl);
        TS_AddCallBack(handler, "Divergence_CB",  TS_CALLBACK_THRESHOLDING,  TS_SCALAR,  divergence);
        TS_AddCallBack(handler, "User_CB",        TS_CALLBACK_PROCESSING,    TS_SCALAR,  User);
        TS_AddCallBack(handler, "User2_CB",       TS_CALLBACK_PROCESSING,    TS_VECTOR,  User2);

        Char *query = "SELECT * WHERE ('Curl_CB' > 7.50e-05 AND 'Divergence_CB' < -1.60e-05) AND 'User_CB' > 0.017
                        ThenApply User2_CB"
        TS_ExecuteQuery(handler, query, 0, 399, 0, 2999, 0, 1499);
        MPI_Finalize();
        return 0;
}
```

Figure 5.25: Code Snapshot of Framework API Calls

```
#define P(i, j, k)      *(yourblk->pBlk + i*m*n + j*n + k)
int User(TS_3DFBuff *block, TS_3DFBuff *yourblk)
{
      int l  = block->l;
      int m = block->m;
      int n  = block->n;

      for(int i = 0; i < l; i++) {
            for(int j = 0; j < m; j++) {
                  for(int k = 0; k < n; k++) {
                        float x = PX(i,j,k);
                        float y = PY(i,j,k);
                        float z = PZ(i,j,k);
                        float mg = sqrtf((x*x)+(y*y)+(z*z));
                        P(i,j,k) = mg;
                  }
            }
      }
      return 0;
}
```

Figure 5.26: User Registered Callback Snapshot

Chapter # 6

Performance Model of Framework

Note:
All the performance measures reported in this chapter are taken on **"Spindel Cluster"** located at University of California, Sandi ego. The cluster consists of 9 nodes with 8 compute nodes and one login node. The experiments are executed on 8 compute nodes which are equipped with dual Itanium processors and **2 Giga-bytes** of main memory. The cluster has PVFS2 file system installation that is accessible from any node.

The TeraShake datasets are executed on the framework and a single dataset has a domain size of 400x3000x1500. The dataset is divided into contiguous blocks of dimensions: 16x30x30. The queries are generated for opened dataset and the view window selects volume in space.

The performance model defines the set of experiments executed on the framework to quantify framework specific performance parameters. The performance indicators are scrutinized in the context of Application Access Patterns that are defined by a view window in domain:

- The volume selection in 3D space
- The layout of selected blocks in file
- The movement of view window

Since the dataset is sparse, the same view window at different positions will select different number of data blocks. The amount of work allocated to processors at runtime will significantly vary as user moves from a low dense block area to a higher one.

Secondly, the layout of data blocks in file against the view window will result into different strides between successive data blocks. The absence of a pattern in file access results into non-optimal use of File System Cache. The I/O performance is at its best if the blocks on the disk are contiguous.

Moreover, it is important to quantify performance as a user moves the view window in domain. Though it is hard to mimic the user behavior as both the view size and window are runtime adjustable parameters, roughly performance against three axes is reported. The results are also compared with a single query that accesses the complete dataset.

The four application access patterns for sweeping the complete domain against a view window that sweep through three axes and that accesses dataset in one attempt are:

- Single request
- X-axis
- Y-axis
- Z-axis

The access patterns fetch the same amount of data from I/O servers. The query window size is varied in all requests as access, for axis only, is planer along the axis. Moreover the access results in different number of queries issued to server for processing. The average response time and server throughput are quantified against the number of queries.

The framework performance parameters in the context of Application Access Patterns are outlined as follow:

1. Parallel I/O Performance

2. Time spent is different execution phases:
    a. I/O and Computation
3. Scalability
4. Load Balancing
5. Response Times
6. Server throughput

The series of results gives time distribution of Parallel Server in two distinct execution phases:

- I/O Phase
- Computation Phase

The I/O phase fetches data blocks from Parallel File System and MPI-I/O performance is reported for the Parallel Server. The computation phase measures time spent in executing user callback function.

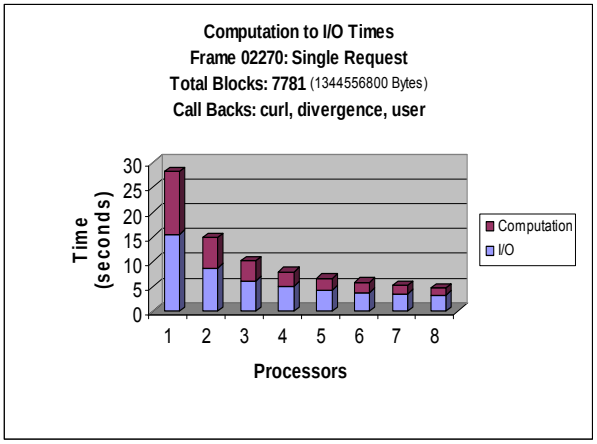


Figure 6.1: Single Request

The Graph [GGG] measures I/O and computation times for accessing a complete frame by the Parallel Server. Almost **1.3 Giga-Bytes** of data is read by the server running with different configurations of MPI processes.

The ***Single Request*** result is to calculate upper bound on computation and I/O times. A single request represents a single query for accessing complete domain: **"Frame 02270"**.

The computation times are roughly equal to I/O times at a single process. The I/O times are high at a single process because 8 I/O servers are feeding data to a single MPI process. Moreover, the reason for higher computation time is due paging activity that thrashes the memory hierarchy of a single machine.

The server shows good performance on higher process count. The computation times keeps on decreasing. The computation time is lowest at 8 processors.

The I/O times are also decreased with increasing processors. The effect is more significant at small number of processors, 4 processor count. There is further decrease in I/O times at higher processors but the effect is not much significant. Readings are taken at more than 8 processors but the times are not further decreased. At 8 processors is the lowest reported I/O time. It is quantified as lower bound on I/O times for parallel server.

The parallel server shows good scalability. The computation and I/O phases show increased performance for increasing processors. Since there are only 8 Nodes on Spindel therefore scalability results are shown only up to 8 processors.

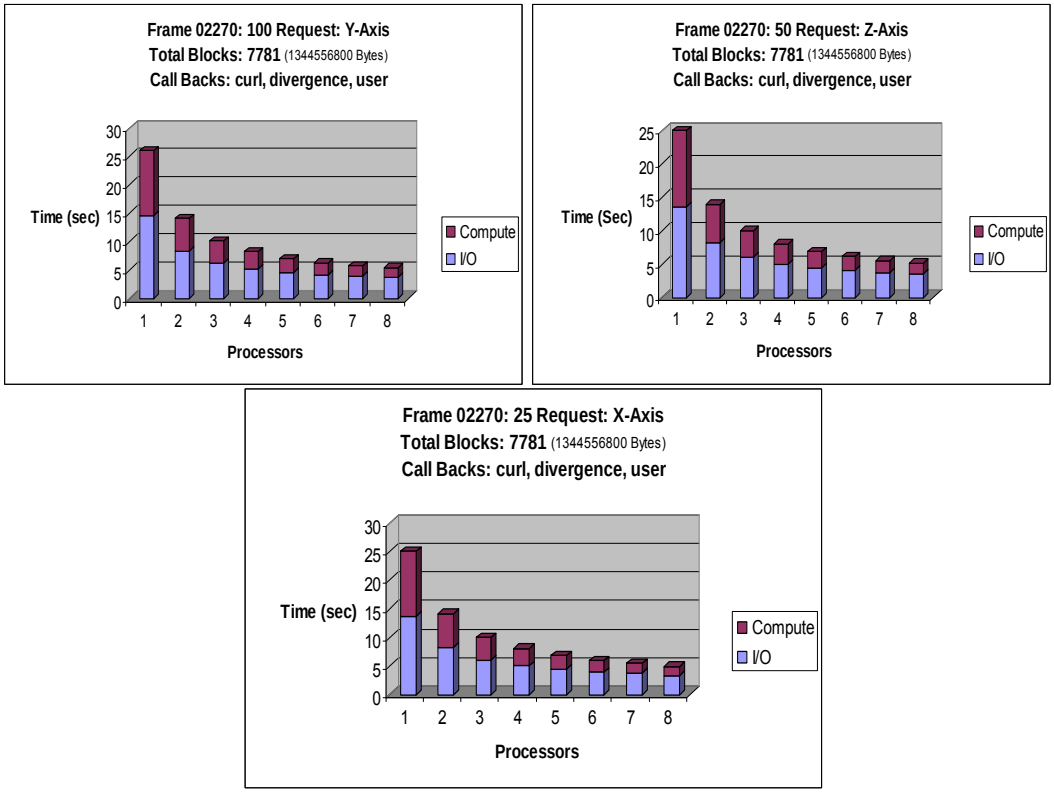


Figure 6.2: Total Time of Query: I/O and Computation Phases

The quantification of ***Single Request*** result gives upper bounds on Parallel Server phases. The above graphs give I/O and computation ratios of accessing domain along X, Y and Z axis. It takes ***25***, ***100*** and ***50*** requests to access the complete domain. The window size along each axis is:

- X-axis
- Y-axis
- Z-axis

The I/O and computation times are calculated by accumulating timing information of each query during both phases only.

The results are pretty much consistent with the single request case. The Un-Packed I/O requests are issued for accessing each block in domain. There are total of *7781* blocks in frame 02270 that means same number of Independent I/O requests are issued to File System. Independent I/O access stands little chance to be optimized by MPI-I/O data sieving algorithm because the access information is provided in pieces to File System.

Moreover, Un-packed requests access single block in-contrast to Packed Requests that access all I/O in one request. The larger number of requests saturates the File System.

Lastly, the block size of 16x30x30 with 3 floats represents 172800 Bytes (**169 KB**) that almost represents 3 PVFS2 blocks, with **64K** block size. Each access represents fetching quite a significant amount of data from I/O servers.

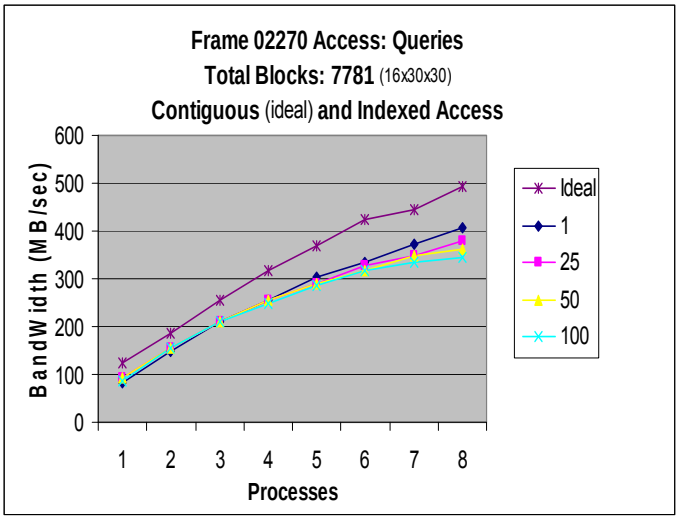


Figure 6.3: Bandwidth Comparison

The data for each access pattern is same: the complete domain of frame 02270. The cumulative I/O times are calculated for each access patterns and the above graph reports the I/O bandwidths of each access. There is not much difference between bandwidths for process count up to 6. The gap further widens at 7 and 8 processes. The general behavior is: lower number of queries result into better throughput of server.

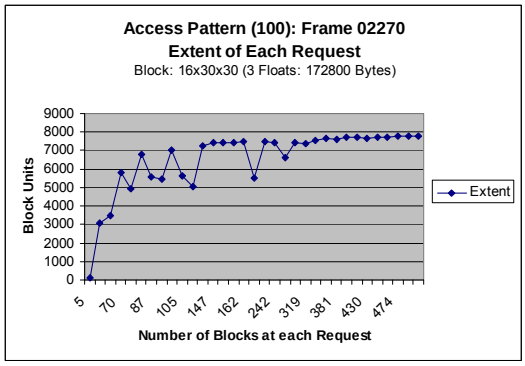
The only distinction is the “ideal case”: that use packed I/O requests. It is an upper bound on I/O bandwidths. The gap between “ideal” and rest of cases represents the potential that could be attained through Packed I/O requests which are not yet taken due to problems with Non-contiguous Memory access with PVFS2.

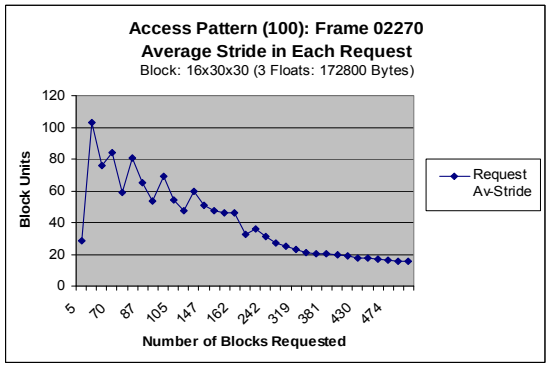
Analysis of Access Patterns

The MPI-I/O performance of Parallel Server is analyzed in the context of access patterns. Each query of an access pattern is scrutinized for following parameters:

- Extent of each request in dataset file
- The average stride between successive blocks
- I/O Bandwidth through MPI-I/O and PVFS2

Each query represents a set of blocks that must be fetched through MPI-I/O interface. The extent of each request is reported in Block Units (1 Block Unit = 172800 Bytes). The extent is calculated by taking difference of end and start block in an access. Similarly, the average stride is calculated by taking average distance between successive blocks in file. The access pattern is characterized by a single process access because single process outlines the pattern that must be fetched from file for a given query. The average strides are also reported in block units. Lastly, the aggregated bandwidth of each query is plotted against the set of requested blocks.





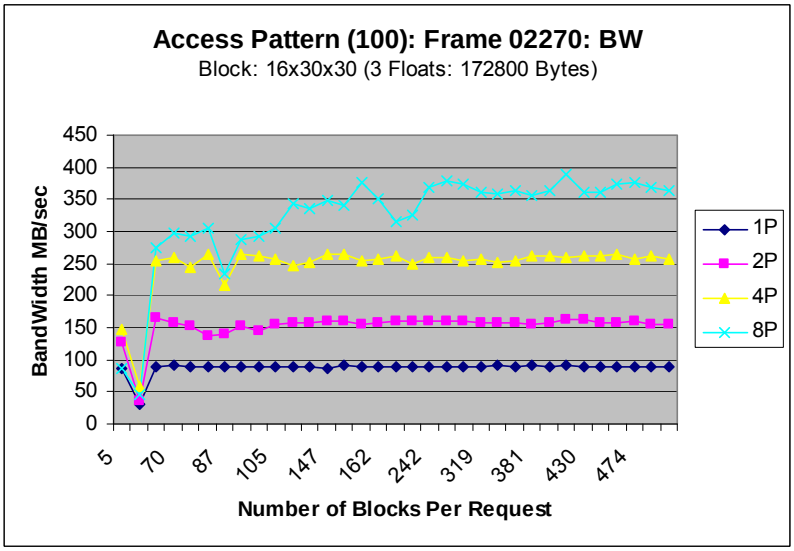


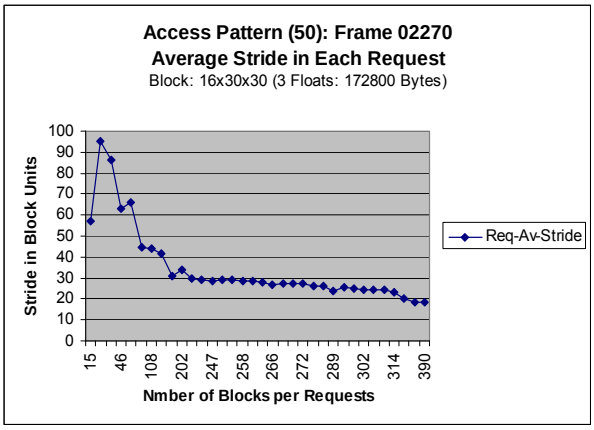
Figure 6.4: Access Pattern 100

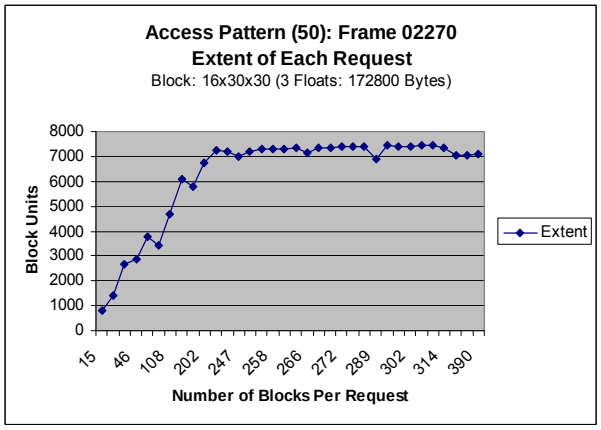
The above graphs show the access pattern of 100 requests that fetch data along the Y-axis. The 3D view window is same for each request:

```
for (int i = 0; i < 100; i++) {
        int cury  = i*30;
        int curyy = 30*(i+1)-1;
        TS_ExecuteQuery(handler, query, 0, 399, cury, curyy, 0, 1499);
}
```

The average stride is high for requests with lower block count. As the number of block count increases in a 3D window the average stride between successive blocks in decreased.

The performance of Parallel Server scales well as the number of processes are increased. Almost 100 MB/sec is added by increasing the process count by a factor of 2. The bandwidth remains consistent at lower process count but starts to oscillate at higher processors. The possible reason is inconsistency in average strides in blocks. Since MPI-I/O use data sieving for optimizing independent I/O different strides at each access will fetch less useful blocks in sieving buffer.





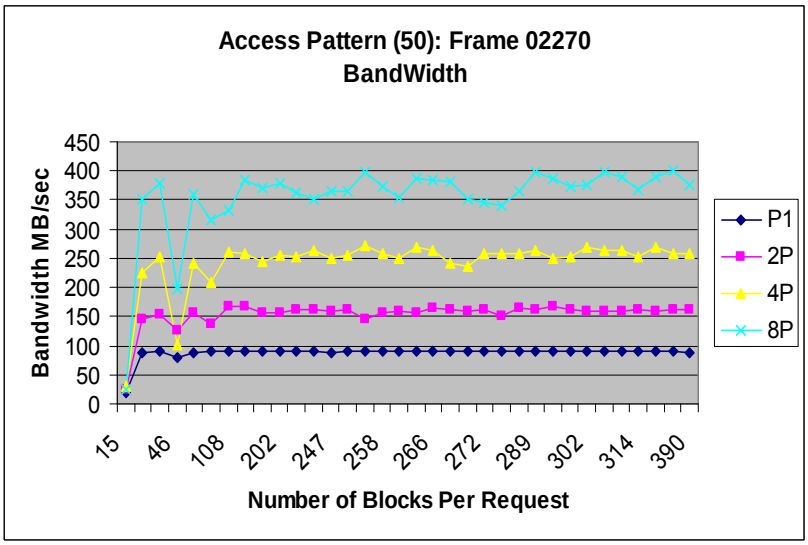
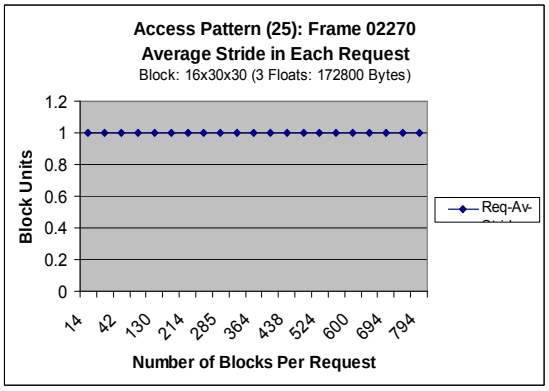


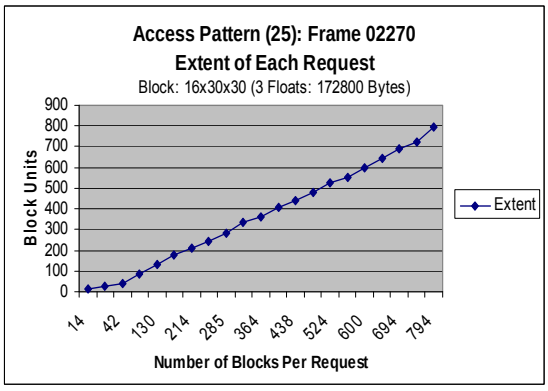
Figure 6.5: Access Pattern 50

The above graphs shows I/O performance results for 50 requests along Z-axis.

```
for(int i = 0; i < 50; i++) {
    int curz = i*30;
    int curzz = 30*(i+1)-1;
   TS_ExecuteQuery(handler, query, 0, 399, 0, 2999, curz, curzz);
}
```

The important observation here is the performance hit at block count 46. The performance chokes at Block count of 46 where the start block is 12 and end block is 2858. The average stride is 63 is this request. The possible reason for this much drop is bandwidth is due to layout of blocks on a single disk controller. Since the file is striped on disks and there are 8 Nodes in total therefore this behavior can be attributed to this





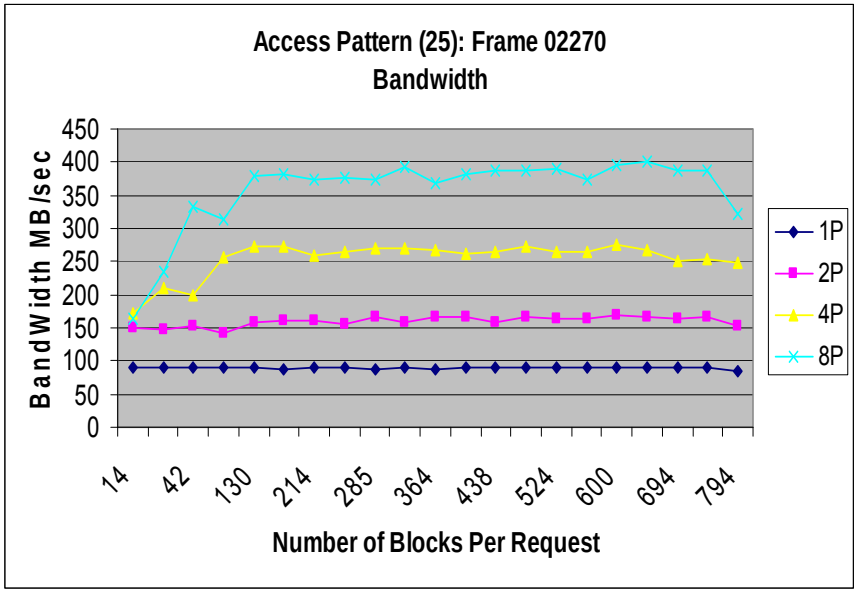


Figure 6.6: Access Pattern 25

The last set of graphs shows 25 requests access pattern along x-axis. The 3D query window is:

```
for (int i = 0; i < 25; i++) {
    int curx  = i*16;
    int curxx = 16*(i+1)-1;
   TS_ExecuteQuery(handler, query, curx, curxx, 0, 2999, 0, 1499);
}
```

The interesting observation here is that each query accesses contiguous blocks from file. The average stride is exactly one for each access. The bandwidth results also become consistent with constant stride in each access. The reasons are both data-sieving and Parallel File System “Read Ahead” optimization. There is a greater possibility that future blocks needed by application are present in File System cache.

In short, the block size of domain is big enough to absorb the access patterns effects especially at higher block numbers. The performance is affected for smaller block counts with bigger average strides. Moreover, at processor count the bandwidth oscillates due to different average strides between successive blocks.

Chapter # 7

Dataset Visualization

Usage Scenario

The Parallel Server data management and query interface is validated against a visualization module. The visualization module implements the iso-surface generation of TeraShake datasets. The datasets are large enough to be visualized at once. The query interface supports window operations that operate on parts of datasets. The visualization requires implementation of a "Visualization Window" within dataset domain. The user moves the visualization window in domain and data fetching requests are translated and passed on to query interface of Parallel Server.

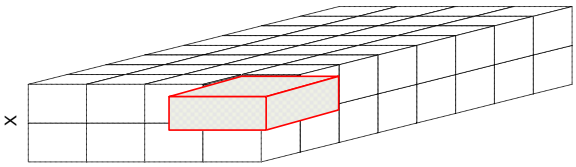


Figure 7.1: Visualization, 3D View Window in dataset domain

Background

Since TeraShake datasets belong to Computational Fluid Dynamics, at first, Mayavi [37] an open source implementation was used to visualize datasets. But the Mayavi software has its limitations: it is suited for visualizing very small datasets files. The software crashes for visualizing large datasets. Moreover, the requirement for selecting parts of datasets for visualizing was not directly supported by Mayavi therefore it was decided to develop our own visualization interface.

The Visualization Tool Kit [36], vtk, library from Kitware is used for visualization. The vtk library is popular for scientific data visualization and supports data streaming and parallel data visualization on small cluster environments.

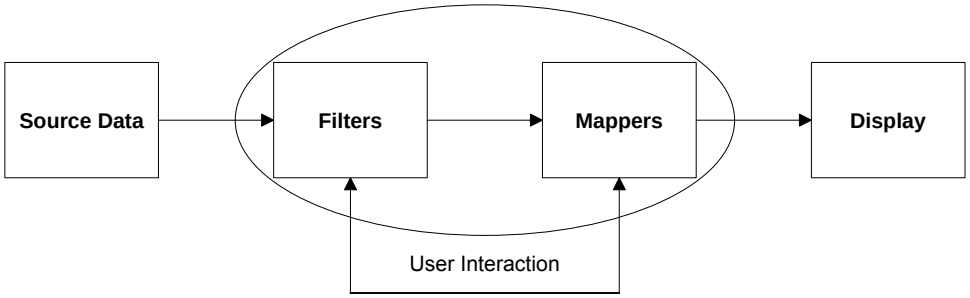


Figure 7.2: General Usage of VTK Pipeline

VTK is cross platform 3D visualization library that consists of lower level Graphics subsystem and higher level visualization pipeline subsystem. The source data represents the dataset. There are many possible types of data elements supports in VTK but we use **Structured Points/Grid** for TeraShake datasets visualization.

The Filters consists of visualization algorithms that extract parts of a dataset for a particular visualization type. The filters output is connected with Mappers that holds the geometry of a 3D object in scene. The Actors are the sink of visualization pipeline and represent display characteristics, position, orientation, textures etc.

The connection of above objects types represents a visualization pipeline in vtk. The pipeline depicts data flow within vtk subsystem. The multiple connections between Filters and Mappers connectivity gives interactivity to visualization. The pipeline is fired at each instant of Render function call. The pipeline calculates output for each object and the final geometry is displayed through available Graphical system like OpenGL, Direct-X.

Platform

The open source vtk library is compiled with Python Wrappers along with TCL and TK support. The vtk applications are written in python and more importantly debugging support is available under Python Interpreter through vtk Python Wrappers. The TCL and TK support User Interface widgets for interactive visualization applications.

Datasets Visualization Pipeline:

The primary requirement for TeraShake datasets is the 3D visualization of Iso-Surface. Since the datasets are large enough for remote visualization on a single machine, the view window is added for displaying parts of iso-surface. The view window calls the query interface of Parallel Server to fetch data from the dataset file.

The Visualization pipeline for displaying iso-surface in vtk is as follow:

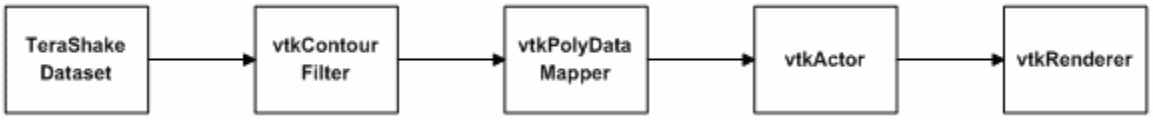


Figure 7.3: Visualization Pipeline for TeraShake Datasets

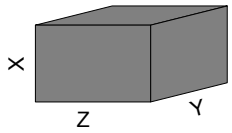


Figure 7.4: Single Data Block, 16x30x30 (3 Floats: 172800 Bytes) in dataset

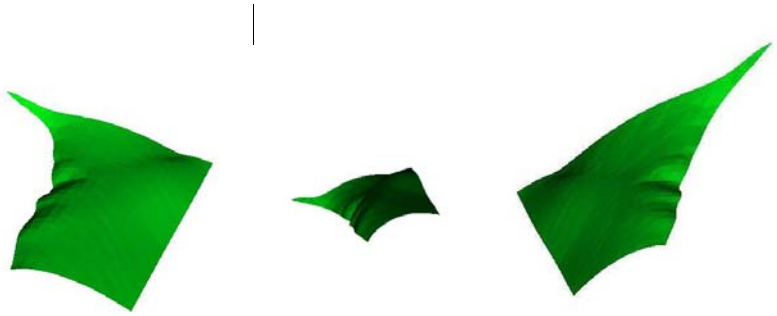


Figure 7.5: Iso-Surface of a single Data Block of Figure 4, three views

The metadata is explicitly designed to fit in to main memory of a single machine therefore meta-information is transmitted to clients []. The view window identifies the required blocks from file. The iso-surface geometry of each individual block is calculated through the dataset pipeline. Meta-information is used for identifying location and dimensions of each block in complete domain.

Off-Line Data Visualization

The data block approximately takes **169KB** of main memory. The source data type is ***vtkStructuredPoints***. Each Grid Point needs a scalar value to generate the iso-surface. The scalar value for current implementation is taken to be magnitude of velocity.

$$V = V_x^2 + V_y^2 + V_z^2$$

The offline visualization of datasets is preferred over online due to following reasons:

- Slow internet connections
- Large data transmission
- Memory issues on client machines

Instead of transmitting **169KB** of data block**,** that represents 3 floats on each Grid Point; new data product is written from source data that reduce data blocks size by one third factor: **53KB**. This sufficiently reduces memory and processing requirements on a single client machine.

Moreover, the pipeline executes in a sequence and the output of one object is made input to next in pipeline until it is terminated on sink: vtk Actor. The intermediate objects in pipeline holds temporary results along with final geometry object that is displayed in screen. The iso-surface geometry takes another **48KB** main memory.

Similarly, the movement of 3D objects in scene fires the pipeline and is executed again to update any changes from the previous execution of pipeline. This consumes considerable processing and memory resources. The result is at higher blocks plotting the client machine memory thrashing start.

In short, due to above reasons the offline data visualization is selected. Secondly, the client machine cannot hold all 7781 blocks of iso-surface. Instead we managed to plot up to 2000 blocks of data.

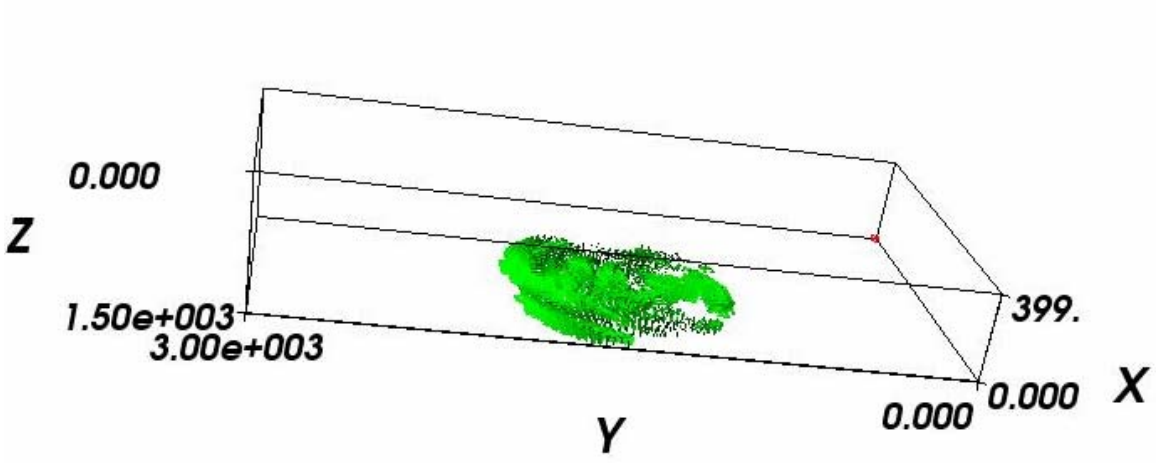


Figure 7.6: 2000 Blocks for Frame 02270
Domain 400x3000x1500

Chapter 8

Future Work

The PVFS2 gives scalable I/O performances with MPI-I/O implementation in interactive cluster environments. The current Parallel Server has ***synchronous*** implementation. The next step for framework is converting current version to an ***asynchronous System*** with I/O and computation overlap will surely, result into better throughput. Moreover, data processing on dataset blocks doesn't consider **Ghost Cells** into account. It will be interesting to note framework computation and I/O performance with Ghost Cells.

The major enhancement is to parallelize the Visualization Pipeline for iso-surface generation by using vtk data streaming feature. It will alleviate memory and processing requirements on remote client machines. Moreover, efficient Data Compression techniques can be applied for online remote visualizations over **Wide Area Networks** (WAN).

The datasets are sparse and when many blocks are plotted in vtk the resultant iso-surface is not contiguous but has cracks in it. It is needed to investigate how the individual surfaces are merged together to generate a contiguous plot.

The TeraShake datasets have associated timestamps. Once the individual plot of datasets are combined in a specific area in domain the resultant visualization will step through the velocity components through time dependency.

The implementation of Visualization Module validates the data management of Parallel Server. Any data analysis or visualization module could be developed by using the query interface of Parallel Server.

Bibliography

[1] Active Data Repository Project: http://www.cs.umd.edu/projects/hpsl/ResearchAreas/ADR.htm
[2] http://larva.cs.umd.edu/misc/publist.asp

[3] Michael Beynon, Renato A. Ferreira, Tahsin M. Kurc, Alan Sussman, Joel H. Saltz, "DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems", Eighth NASA Goddard Conference on Mass Storage Systems and Technologies/Seventeenth IEEE Symposium on Mass Storage Systems, 2000, pp. 119-133.

[4] Moon, J.H. Saltz, Scalability analysis of declustering methods for multidimensional range queries, IEEE Trans. on Knowledge and Data Eng. 10 (2) (1998) 310--327.

[5] D Hilbert. U¨ ber die stetige Abbildung einer Linie auf Fla¨chenstu¨ck. Math. Ann, 38:459–460, 1891.

[6] Patterson D. A., Gibson G., and Katz R.H., "A Case for Redundant Arrays of Inexpensive Disks (RAID)," Proceedings ACM Sigmod, Chicago, June 1988.

[7] Message Passing Interface Forum. MPI-2: Extensions to Message Passing Interface, July 1997. http://www.mpi-forum.org/docs/docs.html

[8] Umit V. Catalyurek, Michael Beynon, Chialin Chang, Tahsin M. Kurc, Alan Sussman, Joel H. Saltz, "The Virtual Microscope", IEEE Transactions on Information Technology in Biomedicine, Vol. 7, No. 4, 2003, pp. 230-248.

[9] W.Allcock, J.Bester, J.Bresnahan, S.Beder, P.Plaszczak and S.Tueecke, GridFTP Protocol Extensions to FTP for Grid, GWD-R (Recommendation), 2002. Revised Apr 2003 http://www-isd.fnal.gov/gridftp-wg/draft/GridFTPRev3.htm

[10] Jaechun No, Rajeev Thakur, Alok Choudhary. "Integrating Parallel File I/O and Database Support for High-Performance Scientific Data Management," sc, p. 57, ACM/IEEE SC 2000 Conference (SC'00), 2000

[11] W. T. C. Kramer, A. Shoshani, D. A. Agarwal, B. R. Draney, G. Jin, G. F. Butler, and J. A. Hules: Deep scientific computing requires deep data. IBM J.Res & Dev. Vol.48 no.2 March 2004

[12] Xiaohui Shen and Alok Choudhary. A high-performance distributed parallel file system for data-intensive computations. Journal of Parallel and Distributed Computing, Volume 64, Issue 10, October 2004, Pages 1157-1167.

[13] HDF Performance Issues, Chapter 14, http://hdf.ncsa.uiuc.edu/UG141r3_html/Perform.fm.html

[14] HDF5. http://hdf.ncsa.uiuc.edu/hdf5/

[15] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, Michael Zingale. "Parallel netCDF: A High-Performance Scientific I/O Interface," sc, p. 39, ACM/IEEE SC 2003 Conference (SC'03), 2003.

[16] C. B¨ohm, S. Berchtold, and D. Keim. Searching in high dimensional spaces, index structures for improving the performance of multimedia databases. ACM Computing Surveys, 33(3):322-373, Sept. 2001.

[17] E. Ch´avez, G. Navarro, R. Baeza-Yates, and J. Marroqu´in. Searching in metric spaces. ACM Computing Surveys, 33(3):273-321, Sept. 2001.

[18] M. S. Sunita Sarawagi. Efficient organization of large multidimensional arrays. In Proceedings of the Tenth International Conference on Data Engineering, pages 328-336. IEEE Computer Society, February 1994.

[19] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In Proceedings Supercomputing '94, pages 650-659. IEEE Computer Society Press, Nov. 1994.

[20] A. Guttman. R-trees: A dynamic index structure for spatial searching. In Proceedings of SIGMOD'84, pages 47-57. ACM Press, May 1984.

[21] ROMIO: A High-Performance, Portable MPI-IO Implementation. http://www-unix.mcs.anl.gov/romio/

[22] Wide Area File Service and the AFS Experimental System, Unix Review, 7 1989

[23] Network Programming Guide, Sun Microsystems Inc 1990.

[24] Parallel Virtual File System, PVFS2: http://www.pvfs.org/pvfs2/

[25] General Parallel File System, IBM Inc, http://www-03.ibm.com/servers/eserver/clusters/software/gpfs.html

[26] Lustre, www.lustre.org

[27] General Parallel File System (GPFS) 1.4 for AIX Architecture and Performance- November 2001. http://www-03.ibm.com/servers/eserver/clusters/whitepaper/gpfs_aix.pdf

[28] PVFS2 Performance measures and Guidelines

[29] Rajeev Thakur, William Gropp, Ewing Lusk. Optimizing Noncontiguous Accesses in MPI-IO. Parallel Computing 28(1) (January 2002), pp. 83-105

[30] Donald Kossmann, The state of art in Distributed Query Processing. ACM Computing Surveys, Vol 32, No.4, December 2000, pp.411-469

[31] Avery Ching, Alok Choudhary, Wei-keng Liao, Robert Ross, William Gropp. "Efficient Structured Data Access in Parallel File Systems," cluster, p. 326, IEEE International Conference on Cluster Computing (CLUSTER'03), 2003.

[32] Avery Ching, Alok Choudhary, Kenin Coloma, Wei-keng Liao, Robert Ross, William Gropp. "Noncontiguous I/O Accesses Through MPI-IO," ccgrid, p. 104, 3rd International Symposium on Cluster Computing and the Grid, 2003.

[33] Beomseok Nam, Alan Sussman. "Improving Access to Multi-dimensional Self-describing Scientific Datasets," ccgrid, p. 172, 3rd International Symposium on Cluster Computing and the Grid, 2003.

[34] Nawab Ali, Maria Lauria. SEMPLAR: A high performance remote parallel I/O over SRB

[35] The TeraShake Project. http://epicenter.usc.edu/cmeportal/TeraShake.html

[36] VTK: Visulization Toolkit from Kitwate Inc. http://public.kitware.com/VTK/

[37] The MayaVi Data Visualizer. http://mayavi.sourceforge.net/

[38] Andrei Hutanu, Andre Merzky, Ralf K□ahler, Hans-Christian Hege, Brygg Ullmer, Thomas Radke, and Ed Seidel. Progressive Retrieval and Hierarchical Visualization of Large Remote Data. In Proceedings of the 2003 Workshop on Adaptive Grid Middleware, pages 60-72, September 2003.