Exploring In-memory HDF5 and Early Evaluations

by

Kalaranjani Vijayakumar, B.Tech

A Thesis
In

Computer Science

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

Master of Sciences

Approved

Dr. Yong Chen
Chair of Committee

Dr. Susan Mengel

Mark Sheridan
Dean of the Graduate School

May, 2015

Copyright 2015, Kalaranjani Vijayakumar

ACKNOWLEDGMENTS

I cannot express enough thanks to my committee for their continued support and encouragement: Dr. Yong Chen, my committee chair, and Dr. Susan Mengel my committee member. I offer my sincere appreciation for the learning opportunities provided by my committee. My project could not have been completed without the support from my mentor, Jialin Liu. Special thanks to my parents Mr. and Mrs. Vijayakumar. Your encouragement when the times got rough are much appreciated and duly noted. My heartfelt thanks to all of you.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

ABSTRACT v

LIST OF FIGURES vi

I.INTRODUCTION 1

Motivations 2
Challenges 4
Contributions 4
Organization 6

II.BACKGROUND 7

Scientific Dataset Management and High Level Libraries 7
HDF5 and Virtual Object Layer (VOL) 7

III.IN-MEMORY HDF5 DESIGN 10

Request Parsing 12
Object Search 14
Lineage Tracking 17

IV.IMPLEMENTATION AND RESULTS 20

1D-Contiguous IO pattern 21
1D-non-contiguous IO pattern 22
2D-Contiguous IO pattern 24
2D-non-contiguous IO pattern 25
3D-Contiguous IO pattern 27
3D-non-contiguous IO pattern 28
Initial Verification of Lineage tracking 30

V.RELATED WORK AND ITS COMPARISON 33

In-memory Cloud Design 33
In-memory Database Design 34
Existing technologies and its comparison with in-memory design 35

VI.CONCLUSION AND FUTURE WORK 37

BIBLIOGRAPHY 38

A.GLOSSARY 41

B.INSTALLATION AND COMPILATION COMMANDS........................... 43

ABSTRACT

Many scientific big data applications have iterative computations and can re-use the results from previous stages in their workflow. HDF5 is one such library that provides scientists with wide range of facilities to perform the scientific data management and computation. Like all the other existing scientific I/O libraries, HDF5 library is an entirely disk based model where the results from various stages of computation are always stored in disk. In our research, we propose to place the results in-memory and to re-use them for the future requests to avoid expensive disk accesses. As the data is residing in-memory, it is not persistent. In order to provide persistence and avoid disk read in such scenarios, lineage information that includes the source dataset and the computation that resulted in the current dataset are stored in memory. Lineage information is captured in a metadata structure as attributes of the dataset for each data block in-memory by intercepting the IO call. This captured lineage metadata can be used to re-compute the dataset without reading the disk. We have evaluated our in-memory architecture with different IO patterns, where the contiguous IO pattern proved to be efficient in a linear fashion, whereas the efficiency of non-contiguous IO pattern remains unpredictable. In addition, we have evaluated our lineage tracking module over the traditional disk based approach for re-constructing the lost datasets.

LIST OF FIGURES

1.1 Sample Analytic Cycle in NaSt3DGP 2

1.2 Scientific Workflow of NaSt3DGP 3

1.3 Lid-driven Cavity Flow Visualization. 3

2.1 Moving Data from Disk to Memory by H5Dread 8

3.1 HDF5 VOL and In-memory Plug-in 11

3.2 Metadata Organization 13

3.3 Object Search 16

3.5 Block Relationships 19

4.1 1D Contiguous IO Cost Benefit 22

4.2 Representation of holes in IO Request 22

4.3 1D Non-contiguous IO Cost Benefit 23

4.4 1D Holes Vs IO Reduction Ratio 24

4.5 2D Contiguous IO Cost Benefit 25

4.6 2D Non Contiguous Pattern 25

4.7 2D Non-Contiguous IO Cost enefit 26

4.8 2D Holes Vs IO Reduction Ratio 27

4.9 3D Contiguous IO Cost Benefit 28

4.10 3D Non-Contiguous Patterns 28

4.11 3D Non-contiguous IO Cost Benefit 29

4.12 Holes Vs IO Reduction Ratio 30

4.13 Re-computation Vs IO Reduction Ratio 31

4.14 Source Dataset size Vs IO Reduction Ratio 32

CHAPTER I

INTRODUCTION

Hierarchical Data Format 5 (HDF5) is a data model, library, and file format for storing and managing data. It supports a large variety of data types, and is designed for flexible and efficient I/O for high volume and complex data. HDF5 is portable and extensible. It includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format (The HDF Group (2014)). As HDF5 supports great range of facilities many scientific applications use it for storing their information and processing it. HDF5 provides various IO optimization techniques to reduce the data movement from the disk and to improve its performance. It has an existing optimization technique, which stores the image of the file in-memory until the file is closed by the users. This is very useful for applications which make changes to the file at lesser granularity, but applications which make changes in a greater granularity still face the issue of fetching data frequently from the disk (Operations). With the intent of solving the large data movement from the disk to main memory during the data analysis phase, we propose an in-memory HDF5 for making the scientific applications work faster without spending much time on IO operations. In this design, we tried to store the results from the previous computation in-memory rather than pushing them to disk. Applications which have a lineage of operations to be performed use the sequence of intermediary files for their next computation. Hence these applications can be benefit from this design as they avoid the disk IO. Storing intermediate results in-memory leads to another major challenge of keeping them persistent or how to re-create them when lost. With this intention of recovering the lost results, we tried to store additional metadata information to the file when it's being created, which can later be re-used to recreate the file when lost. This kind of lineage tracking greatly made the system automatically recover from the lost information. Though we are performing the re-computation on the data, the time needed for re-computing is significantly less than the amount of time the IO operations require to fetch from the

disk. This makes our design more efficient than the existing file image operations in HDF5.

In-memory architecture had proven very good results in the past in other Big Data domains e.g. Spark, proved to be 25x faster than Hadoop in machine learning algorithms (Engle). This architecture greatly inspired us to work on in-memory architecture of HDF5.

Motivations

Many scientific applications perform a sequence of transformations where the output of the previous stage is reused in subsequent stages (Moses). During each cycle, the results are stored in the intermediate files and are later retrieved for the next cycle. Writing the intermediate results to disk is redundant in this scenario, as the subsequent computation is going to re-use it immediately, either partially or fully. These applications can avoid disk IO by storing their intermediate results in-memory rather than persisting them on to the disk.

We select one typical application for case study in this thesis research, i.e., NaSt3DGP. NaSt3DGP is a C++ implementation of a solver for the incompressible, time-dependent Navier-Stokes equations in three dimensions. We show the analytics steps in Figure 1.1 (Engel) and the workflow in Figure 1.2.

```
Step 1: Bin file initialization
navsetup -s cavity.nav -b cavity.bin
Step 2: Parallel simulation
mpirun -np 2 navcalcmpi -b cavity.bin -p3
Step 3: Result generation
vtk file- navsetup -G -TC cavity.bin -o results
Step 4: Visualizization
vtk viewNaSt3DGP.tcl results.vtk.v
```

Figure 1.1. Sample Analytic Cycle in NaSt3DGP.

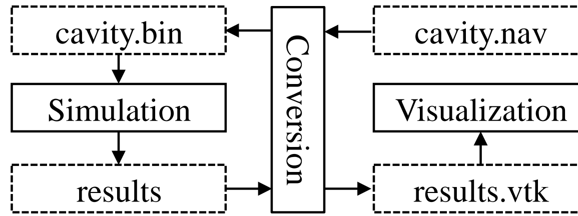


Figure 1.2. Scientific Workflow of NaSt3DGP.

In the above workflow, the *initial .nav* file is converted into the *.bin* file, which is stored in disk and then used in step 2 to run parallel simulation. The results generated from step 2 are converted to the VTK format (Visualization Toolkit) in step 3 in order to visualize them in future (Kitware Inc.). The results from step 3 create the files *results.vtk.p*, *results.vtk.u*, and *results.vtk.v*, etc. These files are later used by the VTK to produce the graphical results to the users, as shown in Figure 1.3. In the above workflow storing the intermediate results like .bin and .vtk files on to disk and retrieving them again for the next set of computation is redundant. Our approach stores these files in-memory and improves the performance of the next set of computation. In case of any loss in the data from the memory, the previous computation is performed again, rather reading the data from the disk. This can benefit the above application, as the computation takes less time than retrieving the data from the disk, when the amount of computation involved is much less compared to data movement.

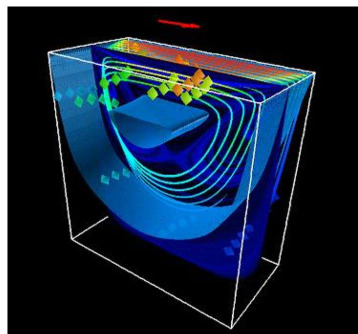


Figure 1.3. Lid-driven Cavity Flow Visualization.

Figure 1.3 shows the Lid-Driven Cavity (csar-advice@cfs.ac.uk) example, which depicts the isothermal, incompressible flow in a two-dimensional square domain.

Apart from NaSt3DGP application discussed above, we have identified other scientific applications that can iteratively run on the overlapped dataset, e.g., Machine learning algorithms (M. e. Zaharia), workflow using matrix operations, groups of statistical operations performed consecutively on a given dataset, etc. Traditional disk based data model, e.g., HDF5, cannot efficiently maintain the data in memory for fast processing. Like the way Spark complements Apache Hadoop by running mapreduce in memory (M. e. Zaharia), we believe that such in-memory data management architecture is also highly recommended in future scientific data analysis.

Challenges

Building an in-memory data management block, however, is not trivial. The challenges include but are not limited to,

- how to leverage the scientific I/O libraries to control the data flow in a distributed shared memory environment,
- how to search the overlapped cached data block in memory,
- how to redirect the request to the traditional model if in-memory blocks decrease the contiguous fetch from the disk,
- how to group different blocks to serve the incoming request,
- how to recover from failure

Above are some major challenges which are taken into consideration in this thesis research.

Contributions

In this thesis research, we propose to utilize the shared memory resources on compute nodes to hold the intermediate data/result for efficient processing. We design

such in-memory architecture to handle all the intermediate data blocks by tracking data and computation flow, caching intermediate analysis results, and automatic recovering. We design the in-memory module based HDF5 VOL (Virtual Object Layer) with all functionality lightweight and transparent to the users. Scientific applications that use HDF5 to manage the data should be able to migrate to our platform without any source code change.

In this research we have focused on major challenges which are specified in the above section. The blocks are stored in shared memory which has to be accessed by different applications. This storage involves the introduction of a new global service thread which has to search for the data blocks even after the completion of the application. This serves the purpose of storing the data in-memory and being utilized by multiple applications rather than restricting them to only the specific applications.

Secondly, we tried to address the problem of searching the data blocks in-memory. This greatly involves the consideration of various factors like the start and end position of the dataset, count of the data blocks which has to be fetched etc. These factors greatly influence the performance which we try to improve though our in-memory design. As the larger amount of data present in-memory the performance measures have to improve linearly. But the results show the variations of what we have expected and these are discussed in Chapter IV.

Thirdly, we tried to create a lineage tracking model to avoid failure in case of the missing blocks. This model is achieved through two major design modules, namely request parsing and the lineage tracking. Request parsing module focuses on capturing the provenance information about the given request. This information is later used by the lineage tracking module, to replay those computations and re-compute the lost datasets without reading the disk. As the amount of computation is increased, the lineage becomes more complex and re-computation becomes more costly than the retrieval of the data from disk. Results for this study are provided in Chapter IV.

These are the major contributions of this research which will greatly help in improving the existing disk-based model to the next level of in-memory design

Organization

This document is organized as follows: The next chapter introduces the architecture and design of in-memory HDF5 plug-in. Chapter III explains the in-memory design architecture and its internal modules. Chapter IV provides the implementation details and the initial evaluation of our system. Chapter V provides the in-sight on the related and the existing work with its comparisons. Finally Chapter VI concludes the work and provides the details about the future work.

CHAPTER II

BACKGROUND

Scientific Dataset Management and High Level Libraries

In this section, we introduce the basic scientific data management framework, for which we will insert the in-memory module.

Scientific datasets are usually high dimensional and contain multiple variables, which are hard to load into traditional database. Instead, scientific libraries, including HDF5 (The HDF Group (2015)), NetCDF (Russ Rew), PnetCDF (Robert R. McCormick School of Engineering and Applied Science, Northwestern University), and ADIOS (Oak Ridge National Laboratory), have been well developed and widely used in recent years for scientific data management. These libraries, not only define the data model and formats, but also provide high performance parallel I/O to accelerate the processing performance.

These libraries are lightweight, high level, and easy to use. Researchers build different scientific data management toolsets (The HDF Group)on top of them, e.g., parallel format conversion, visualization tools, and parallel query utilities, etc. We also intend to support the in-memory scientific data management via the existing scientific libraries. To avoid byte-level file management, we seek a library for object-level management in memory. Among the existing libraries, we chose HDF5 due to its flexible support for user defined API at Virtual Object Layer (VOL). Before discussing the design of in-memory HDF5, we briefly introduce the HDF5 and current VOL design.

HDF5 and Virtual Object Layer (VOL)

HDF5 files are organized in a hierarchical structure, with two primary structures: *groups* and *datasets*.

- HDF5 group is a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.

- HDF5 dataset is a multidimensional array of data elements, together with supporting metadata.

Working with groups and group members is similar in many ways to working with directories and files in UNIX. As with UNIX directories and files, objects in an HDF5 file are often described by giving their full (or absolute) path names.

HDF5 uses data space to describe portions of a dataset, making it possible to do partial I/O operations on selections. Selection is supported by the data space interface (H5S). Given an n-dimensional dataset, there are currently four ways to do partial selection:

1. Select a logically contiguous n-dimensional hyperslab.
2. Select a non-contiguous hyperslab consisting of elements or blocks of elements (hyperslabs) that are equally spaced.
3. Select a union of hyperslabs.
4. Select a list of independent points.

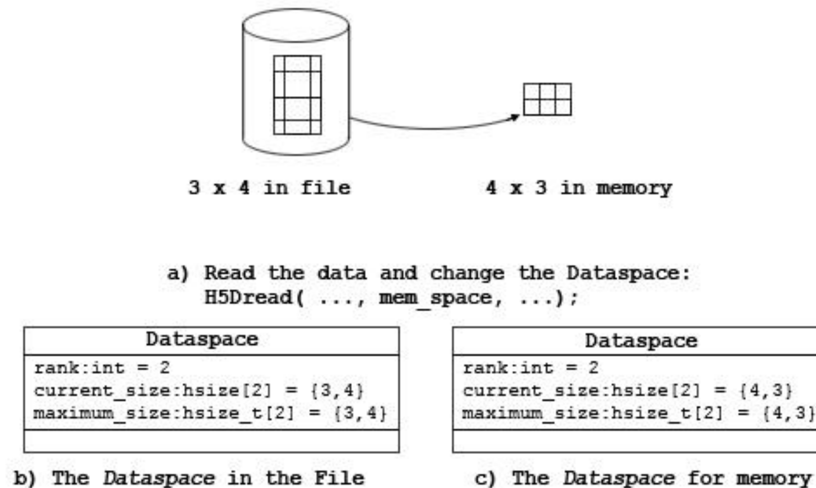


Figure 2.1. Moving Data from Disk to Memory by H5Dread

I/O functions are well supported in HDF5. For example, in HDF5 read, the users can define data space in file and memory, and then read the data from storage to memory. We show a typical *read* using HDF5 library in the Figure 2.1 (The HDF Group).

HDF5 has an abstraction layer inside HDF5 library called Virtual Object layer (VOL). VOL intercepts all HDF5 API calls that could potentially access objects in the file and forwards those calls to a plug-in 'Object Driver'. The plug-in could store the objects on disk in a variety of ways. To reduce space and for clear comparison, we illustrate the existing VOL architecture and our new plug-in in one diagram (Figure 3.1).

CHAPTER III

IN-MEMORY HDF5 DESIGN

Scientific applications that involve iterative computation, usually take the previous results to perform the recent computation. We refer the data change between consecutive steps as *transformation*. Sample transformations that can be benefited from in-memory architecture, include matrix transformations, statistical operations, creating groups in HDF5, converting the file extensions, etc. Consider an application, which involves two or more continuous transformations (T_1, T_2, $T_{3...}T_n$) on a particular dataset. In this scenario, in-memory HDF5 should store the intermediate results in-memory from each transformation for the future use. When the initial set of data blocks are retrieved from the file system and sent to the application, data has to be saved in memory for future computations. Besides, once the application starts processing the data, it can produce intermediate results after each transformation. Instead of storing the results on disk, in-memory plug-in intercepts the IO call and stores these intermediate data blocks in-memory. While storing the intermediary blocks in memory, the new plug-in in VOL layer should update the in-memory metadata which will be discussed in detail in the following sections in this chapter.

Applications can perform simple or more complex transformations but transformation consistency needs to be ensured to provide the auto-recovery of datasets when lost i.e. when the same sequence of transformations is applied on the same set of input files, the output should remain the same. This is greatly essential in designing our in-memory system as missing intermediate results need to be re-computed to avoid fetching from the file system. In order to compare and select the in-memory data block, we also have to support query over the intermediate blocks.

We demonstrate the overview of the new plug-in, i.e., In-memory Management plug-in, on the right side of Figure 3.1. In order to enable HDF5 with in-memory function, we use this new plug-in to intercept the IO call from HDF5 API and parse the request to control the data flow. The high level I/O request is now being re-

directed to the in-memory object layer rather than transferring the call to Virtual File Layer.

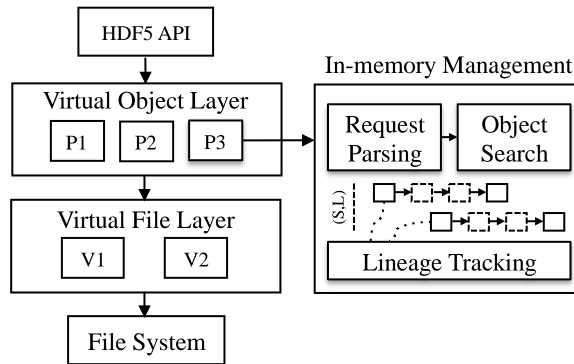


Figure 3.1. HDF5 VOL and In-memory Plug-in

Once the application sends the IO request through the HDF5 API, the request is sent to the virtual object layer, which redirects the request to the respective plug-in. To choose in-memory management plug-in from other plug-ins inside the VOL, users need to set the FAPL parameter (Koziol) while creating the file. Once the users use this in-memory plug-in as the file option in FAPL, VOL takes care of the remaining set of operations on that specific file. By taking advantage of VOL layer, our plug-in is easy to use and can be enabled or disabled anytime, based on application analysis pattern.

We show the in-memory HDF5 plug-in design in Figure 3.1, as P3. There are three major modules inside the in-memory HDF5 plug-in. The first module is Request Parsing. This module parses the incoming application request, and keeps track of all the metadata information about the requested data block. The second module is Object Search. This module takes care of searching the in-memory blocks for detecting overlap and optimizing the incoming IO requests. The third module is the Lineage Tracking. This module is used to re-compute the missing value and recover the data/result loss using lineage information. In the situation that the in-memory data block is transformed with multiple steps, several intermediate transformations may not always be cached in memory. The missing transformations on those non-cached data

blocks can be re-computed from lineage tracking and therefore, the disk I/O can be reduced.

Request Parsing

In order to achieve the in-memory framework, the metadata information about the file should always be made available. This information about the file should never be lost as they contain the necessary details about the origin of the file and the operations that had lead to their current state. Most of the applications that perform the operations using HDF5 file are more consistent i.e. the result from the given dataset by applying a transformation remains the same, if we apply it again on the initial dataset. Operations like finding random numbers are not consistent as they result in different sets of random numbers unless the seed value is fixed. This kind of consistency is very essential when we design a system, which is capable of recovering on its own. We tried to create a metadata structure which includes the existing metadata of the file along with lineage information. Considering the Data provenance and relationships among datasets, which are the major concern for the applications that use HDF5 libraries, we can track the dataset origin and their relationships with their parent datasets using these attributes. This metadata structure can be updated by the VOL plug-in at the time of file creation/ disk read operation. These can be later accessed by "Object Search" module to sense and re-use them for their future regeneration purposes.

The metadata information that we store involves four major components, namely pointer to parent metadata, transformation, array of blocks and flag.

- Pointer to Parent Metadata structure: This component is essential to back-track the ancestor/origin of the given file. This attribute goes in hand with transformation component of the metadata structure to retrieve the lost blocks.
- Transformation: These are the basic set of operations which a user tried to perform on a given dataset. This information is used to re-apply the

same on a given parent data block to reconstruct the dataset when lost from the main memory.

- Array of Block Range: In parent metadata structure, this information tells us the start index and length of the block that is fetched from the original file. In the child block metadata structure, it is essential to note the parent's block which is being used to compute the child block. If there are multiple parents, then there are multiple ranges which are involved, to compute the child block. This computation is briefly explained in the lineage tracking module. In order to achieve this cardinality, we use the array of block ranges.
- Flag: This information tells the status of the in-memory block whether it has data and metadata or only data or it's the original block. These status codes can help while searching the blocks in-memory without much computation.
- Pointer to the actual data: This information stores the location where the actual data blocks are located.

Elements	**Description**
Block Range	Range of data from parent
Transformation	Operation to perform
Pointer to parent structure	NULL/pointer to parent metadata
Flag	Status of the in-memory block
Pointer to the actual data	Location of the original data block

Figure 3.2. Metadata Organization

This metadata information is captured once the application program sends the IO request to the underlying Virtual File System Layer. The IO requests are intercepted by the Virtual Object Layer using FAPL. Once the FAPL is set for the

incoming request, the respective plug-in is instantiated and the IO request parameters are captured by the Request Parsing module in our plug-in. Once the information about the given IO request is extracted from the in-coming parameters, the control flows to the next stage called the Object Search.

Object Search

Once the IO call is being intercepted by the VOL layer, this module takes care of searching the blocks that are in-memory and take necessary actions if they are not available. As the users create more numbers of intermediate blocks, tracking them and reusing them is very essential. There are 3 possible statuses of a block with respect to its metadata and the actual data availability. The below section will describe briefly about these three options.

- Data with Metadata: In this case the data blocks are already available in-memory and hence, the IO operation will be completed at the VOL layer, without any call to the underlying file system. This scenario greatly reduces the IO cost.

- Only Metadata: As the data blocks grow, there wouldn't be enough memory to store the entire data about the file in-memory. In this situation, metadata is available in-memory; hence the data blocks can be re-created from the provenance information. Thus the call is transferred to the lineage tracking module, which creates the data blocks and provides it. This will help us in serving the IO request without making disk IO call.

- No Metadata: This situation can arise if the file is not fetched even once/ if the block has no parent blocks from which it is originated. These kinds of situations have to be handled by the VFL layer, which will take care of disk IO operations. Once the data is fetched its metadata is updated and are placed in-memory for future re-use.

In order to realize this searching, we tried to maintain a search flag in the metadata, which helps us to identify the status of the block. This flag keeps track of the file blocks and their current status. With this information, the data blocks can be identified. The block status flag shows 3 different categories of the block namely,

- 0 - Data and metadata is available,
- 1- Only metadata is available,
- -1 - No metadata/Original block.

No metadata/Original block situation arises if the data block is the original data block on which the computation is initiated, then the block is assumed to be absent in-memory and hence the VFL call is instantiated to fetch the data from the disk. Thus, this information can be used by the "Object Search" on module to search the blocks in-memory. In addition, if the data is not fetched even once, then the metadata information will not be available. In this scenario, the search for the block will fail and the IO request is redirected to the file system layer.

Figure 3.3 describes the control flow of the Object Search module. Once the IO request is received by this module, they check for the flag value. If the block is in-memory, then they are retrieved to the users, else if the data is not present, but if the metadata is available, the request is redirected to the lineage tracking module for further manipulations. If there is no metadata/provenance information, then the request is given to the VFL layer for further processing.

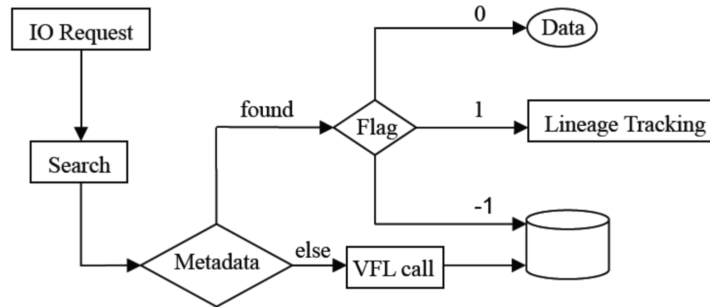


Figure 3.3. Object Search

In-memory blocks need to be combined in a useful fashion to serve the incoming request. Thus the process of efficiently combining the available in-memory blocks/blocks from the disk for a given IO request is done by *join operation*. There are various available IO patterns which may arise based on the locality of the data i.e. either in-memory/disk. These are stated as follows,

Disk and one Partial in-memory block: The portion of the IO request can be available in-memory and portion of it may be in disk. In this situation, join operation is needed to serve the IO request. Join operation combines the blocks from in-memory and disk.

Only one Partial in-memory block: In this type, the IO request needs only the portion of the in-memory data block. In this scenario, there are no two different data blocks to combine; hence no join operation is needed.

Disk and one full in-memory block: In this type, the IO request re-uses the entire data block which is in-memory with some additional data from disk. As there are two blocks one from disk and one from in-memory, join operation is needed to serve the entire IO request.

Many Partial in-memory blocks: The IO request can be of the type where the data blocks can be fetched from more than one in-memory blocks i.e. the data is fetched partially from many in-memory blocks. As there are many blocks of data,

these blocks need to be combined to serve the IO request. This type needs the help of join operation.

Many full in-memory blocks: IO requests can be served from multiple entire sets of in-memory blocks rather than partitioning them. Though the data blocks are re-used entirely, there are many data blocks in chunks, thus joining these blocks is greatly essential to serve the in-coming IO request.

Efficiently joining these data blocks are highly essential to achieve the highest IO reduction ratio.

Lineage Tracking

Lineage tracking comes into picture when the block's original data is lost from the main memory but the metadata is present. This situation arises when new results overwrite the existing data blocks. In this scenario when the file block is overwritten by certain other file blocks, then the flag is updated as 1 from the state of 0, if the data blocks have any parent blocks i.e. to indicate that these data blocks can be re-computed. If there are no parent blocks involved, then the block status is updated as -1 to indicate that these data blocks cannot be re-computed, which in-turn makes the IO to be re-directed to VFL layer. This module ensures the re-creation of the lost blocks without involving the disk IO. In order to re-trace the blocks without losing the information, we have saved enough provenance metadata information about the origin of the blocks in the module "Request Parsing". Once we could identify the parent blocks and the transformations, we can easily re-create the file/lost data blocks by re-applying them. Though this situation is computation intensive, it avoids the disk IO which will improve the performance of the application.

Figure 3.4 shows the control flow of the Lineage Tracking module. Once we know the data is not in-memory and its metadata is available, back-tracking the parent's metadata and fetching the parents is essential. This back-tracking is done by traversing the provenance information. Once the parent block is identified from the metadata of the required dataset, we need to perform the search for the parent block to

make sure whether its in-memory. If the parent blocks are present in memory (control flow marked as 0 in the Figure 3.4), re-computation can be done with the available information. If the parent data block is not available, these have to be brought in-memory before performing the child computation. This flow seems to be complex but can be achieved through recursive algorithms to compute the lost data set. This makes our system an auto-recovery model. Figure 3.4 shows these recursive methodologies by including a while loop which checks whether the re-computed dataset is the same as that of the required dataset which is being searched.

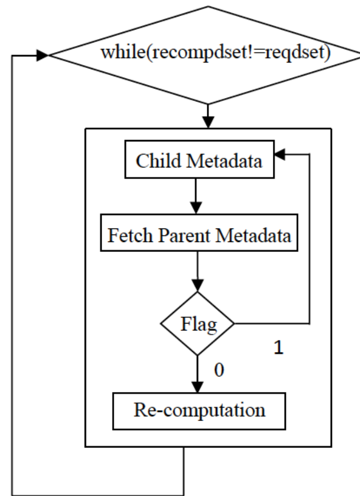


Figure 3.4. Lineage Tracking

There are three different situations which may arise when we try to back track the block creation as shown in the Figure 3.5.

(1) The block may have only one parent

(2) The block may have many parents, the presence of all the blocks in-memory is essential to re-create the file. In this situation, even if one block is not available, a recursive call to this lineage tracking module has to be performed to fetch the current block.

(3) A block can have many children. In this case, when we try to regenerate a certain set of blocks from a transformation, other blocks are created as well giving an advantage as these blocks can again be re-used by any other computation.

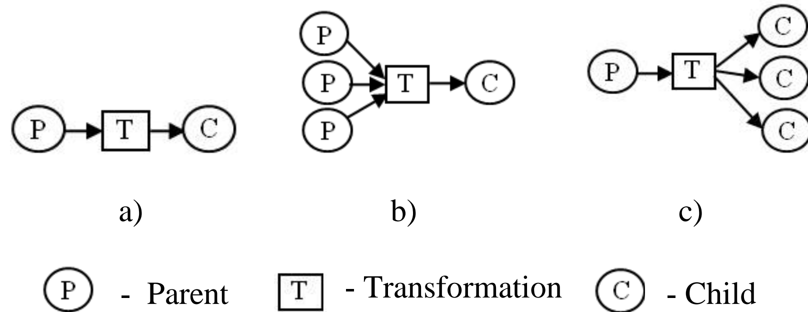


Figure 3.5. Block Relationships

a) Single parent b) Multiple parent c) Multiple children

Considering these states are highly essential when we design our system, as the amount of computation needed to re-create more parents can be higher and more complex compared to the one that has single parent. Computation leading more than one child blocks can serve the purpose of more re-usability as more blocks are generated in a single transformation. Thus considering these block relationships are highly essential while designing the lineage tracking module.

CHAPTER IV

IMPLEMENTATION AND RESULTS

This section provides the initial set of verification results about the in-memory design. These results can greatly help us in the future to refine the proposed architecture to get better results than expected. Below, a set of experiments are performed on 1D, 2D and 3D datasets of HDF5. These experiments show the IO benefit that can be achieved from the in-memory plug-in as compared to that of the disk-based model.

IO requests can be broadly classified into two categories, namely contiguous and non-contiguous IO requests. Contiguous IO requests are those in which the requested data is in a linear fashion without any interleaving blocks. Non-contiguous IO requests are those which request data by interleaving certain blocks of data, i.e., the data blocks are not continuous.

In this context, it's good to introduce a new term named holes. *Holes* are nothing but the interleaved portion of data that are either not needed by the original IO request/the data has already been present in-memory, which need not be fetched from the disk. These holes play a major role in our study as these greatly affect the IO performance benefit which we intend to achieve. With the above classifications based on the dimensions and the IO request pattern, we could get 6 different classes of IO requests, namely 1D contiguous, 1D non-contiguous, 2D contiguous, 2D non-contiguous, 3D contiguous and 3D non-contiguous. In this study, we have performed experiments on all these classes of IO requests and provided the achievable IO performance benefit from the in-memory architecture design. These experiments were conducted on the Hrothgar cluster, which is a node based system. It has 128 nodes containing two Nehalem 3.0 GHz 4-core processors with 16 GB main memory. The following sets of results are performed on the same set of hardware but with different sized input datasets which will be mentioned under each section separately.

In the below section, we have used two major terms, namely Hit ratio and IO reduction ratio/Efficiency.

Hit ratio can be defined as the amount of data which is already present in-memory.

IO reduction ratio/Efficiency can be defined as the ratio between the difference in the IO cost for fetching the entire data from the disk and that of fetching only the data that are not in-memory to the time taken to fetch the entire data from disk. This ratio can be shown in the below expression,

$E = (T_d - T_m)/T_d$ where $-1 \geq E \leq 1$

where E is the IO reduction ratio,

T_d is the time taken to fetch the entire request from the disk,

T_m is the time taken to fetch only the partial data from disk (i.e. data which is not in-memory).

1D-Contiguous IO pattern

In this kind of IO pattern, the data to be fetched doesn't contain any holes, i.e. the dataset has to be fetched in a contiguous fashion without any holes. As there is no interleaving data blocks, as per theoretical analysis, the IO performance benefit should be higher. Figure 4.1 shows the 1D-contiguous IO performance improvement. IO performance improves if the entire dataset is already in-memory and reduces gradually as there is a reduction in the hit ratio. As the amount of data to be fetched from disk is reduced, the improvement in the IO performance can be seen increasing in a linear fashion. Thus the applications which re-use the entire dataset from the previous steps can be greatly benefitted from the in-memory model. I have used 0.8GB dataset to verify the below result.

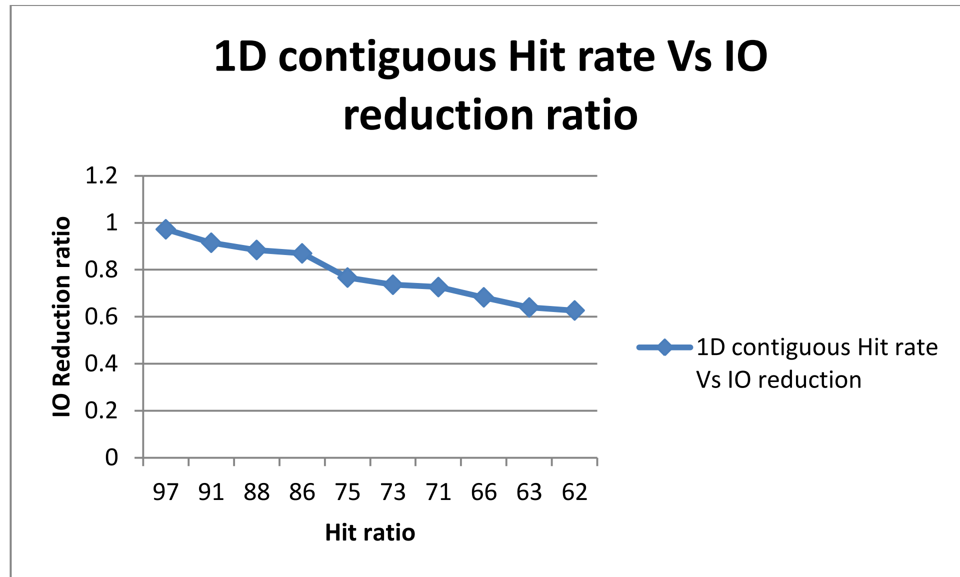


Figure 4.1.1D contiguous IO cost benefit

The above graph shows the decrease in the IO reduction ratio, expressing that the performance drops as the amount of data present in-memory reduces.

1D-non-contiguous IO pattern

In a non-contiguous IO pattern, there are certain blocks of data which are already present in-memory. Some data blocks are fetched from disk in the previous steps or computed in the previous stages. These create holes in the IO request. Figure 4.2 shows the sample 1D non-contiguous request consisting of 3 holes.

H1 H2 H3

Figure 4.2. Representation of holes in IO request

As the data blocks are not contiguous and the arrangement of holes is not predictable, the IO performance of the requests remains uncertain. Even though the amount of data present in-memory is greater, it may provide unpredictable IO benefits. This is shown

in the below graph based on the experiment performed on the 1D dataset. I have used 0.2GB dataset to verify the below result.

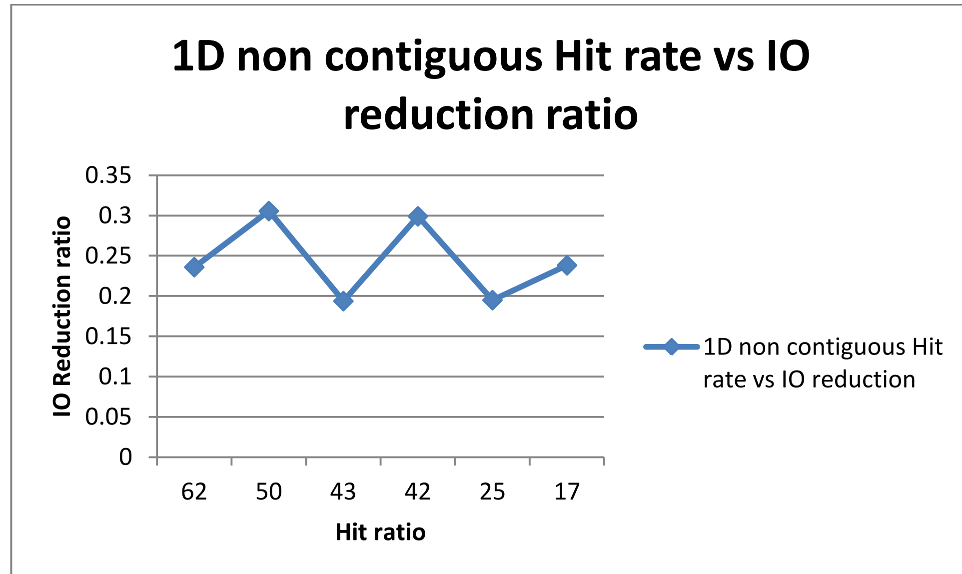


Figure 4.3.1D non-contiguous IO cost benefit

One factor that can affect the IO efficiency is the number of holes present in the IO request. Figure 4.4 shows the increase in the number of holes, potentially drops the IO performance. I have used 0.2GB dataset to verify the below result.

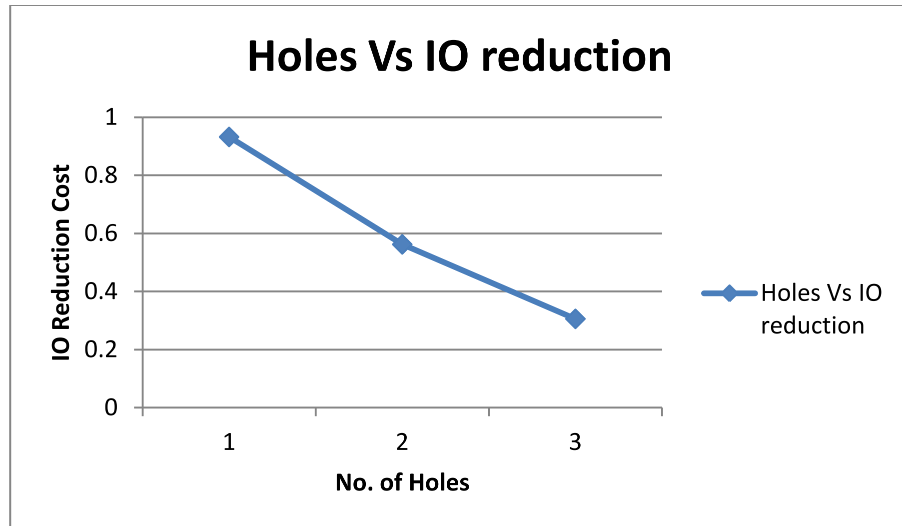


Figure 4.4.1D Holes Vs IO reduction ratio

Similar kinds of tests are performed on the 2D and 3D datasets and the results are shown in the figures below. In terms of 2D and 3D, the non-contiguous patterns can be of various types and are discussed in the forthcoming sessions.

2D-Contiguous IO pattern

In 2D contiguous data pattern, the data blocks that are needed are contiguous in both the dimensions i.e. both in X and Y. Figure 4.5 shows the IO cost benefit that can be achieved through this in-memory design.

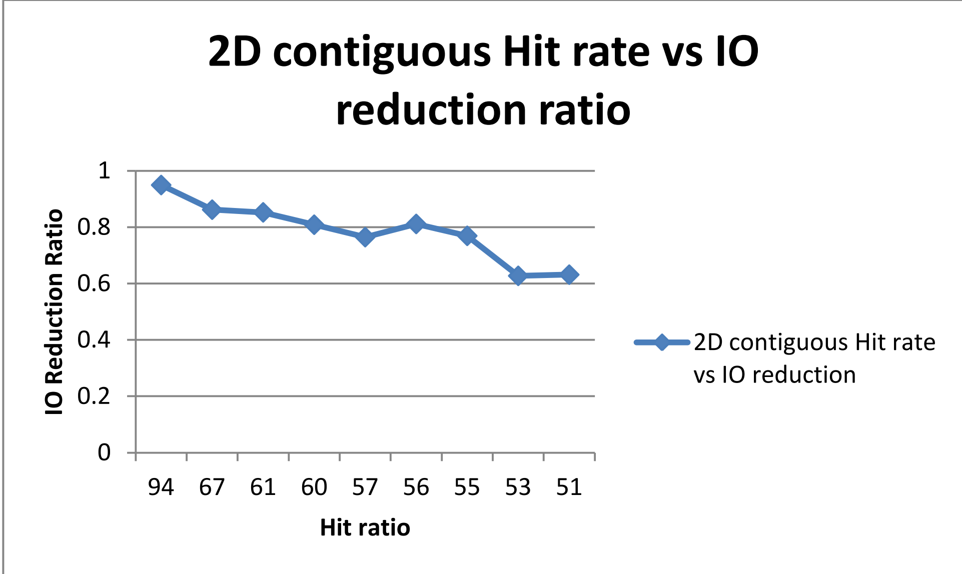
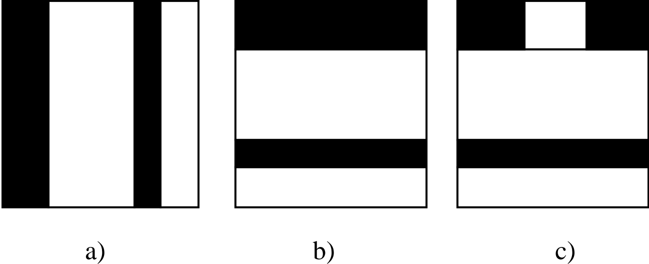


Figure 4.5.2D contiguous IO cost benefit

In the Figure 4.5, there are some scenarios where there are some unpredictable IO patterns are seen. These may arise because of the presence of the cache or the network delays. But in-general, the IO benefit tends to reduce as the hit ratio decreases. I have used 0.8GB dataset to verify the above result.

2D-non-contiguous IO pattern

In 2D non-contiguous data pattern, the data can be non-contiguous in either X, Y or on both dimensions. Figure 4.6 show the non-contiguous patterns in different dimensions.



a) b) c)

Figure 4.6. 2D Non contiguous pattern

a) X-axis b) Y-axis c) Both X and Y axis

Below, Figure 4.7 shows the non-contiguous IO benefit that can be achieved through our approach. Similar to 1D, the pattern shows uncertainty in its behavior. This can be due to many factors like the number of holes, the distance between the holes, the dimension in which the holes have occurred etc. I have used 0.8GB dataset to verify the below result.

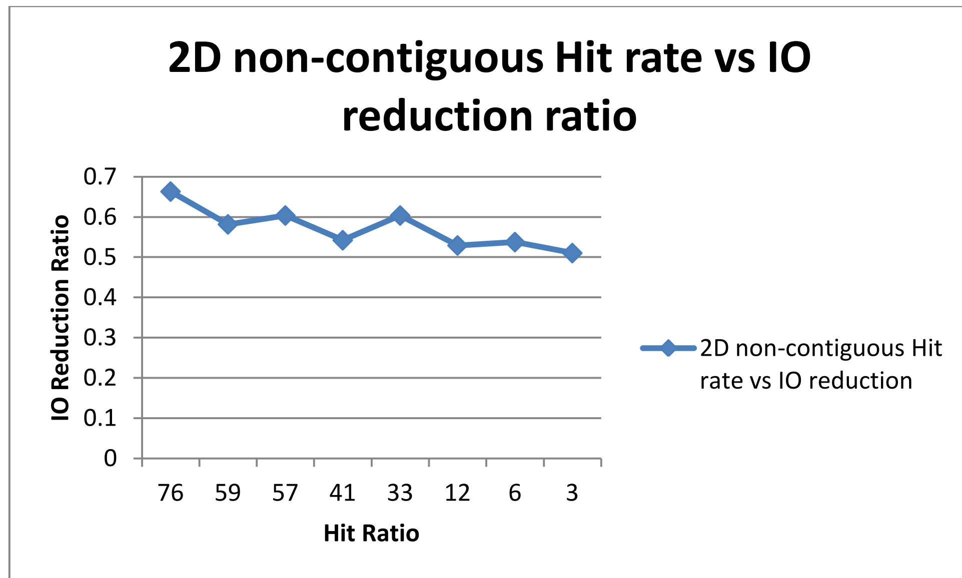


Figure 4.7. 2D non-contiguous IO cost benefit

Similar to 1D dataset, the effect of holes on the IO reduction ratio is analyzed on the 2D dataset. Figure 4.8 shows the effect of holes on a 2D dataset. I have used 0.8GB dataset to verify the below result.

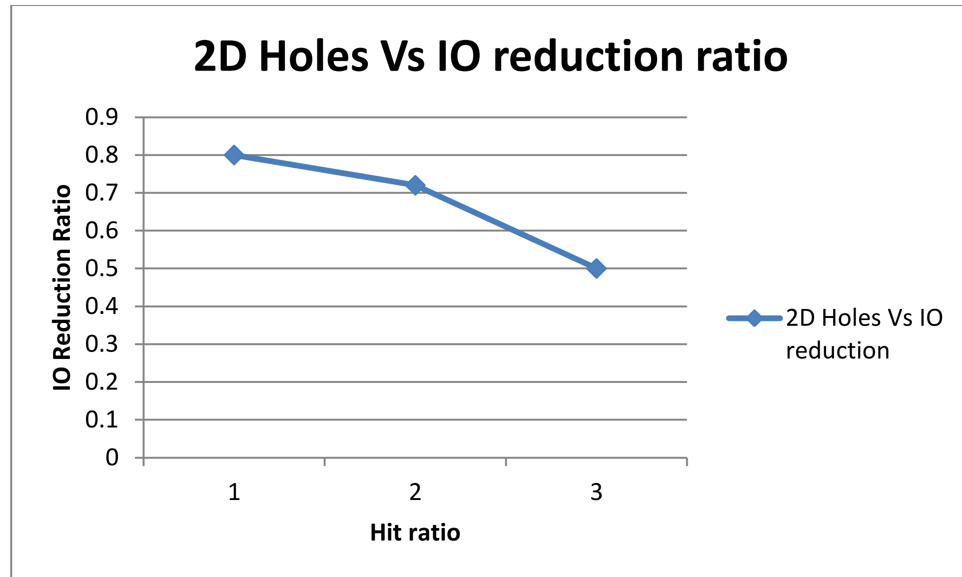


Figure 4.8. 2D Holes Vs IO reduction ratio

3D-Contiguous IO pattern

In 3D contiguous data pattern, the data blocks that are in need are contiguous in all three dimensions i.e. in X, Y and Z. Figure 4.9 shows the IO cost benefit that can be achieved through this in-memory design in a 3D contiguous IO request. I have used 1GB dataset to verify the below result.

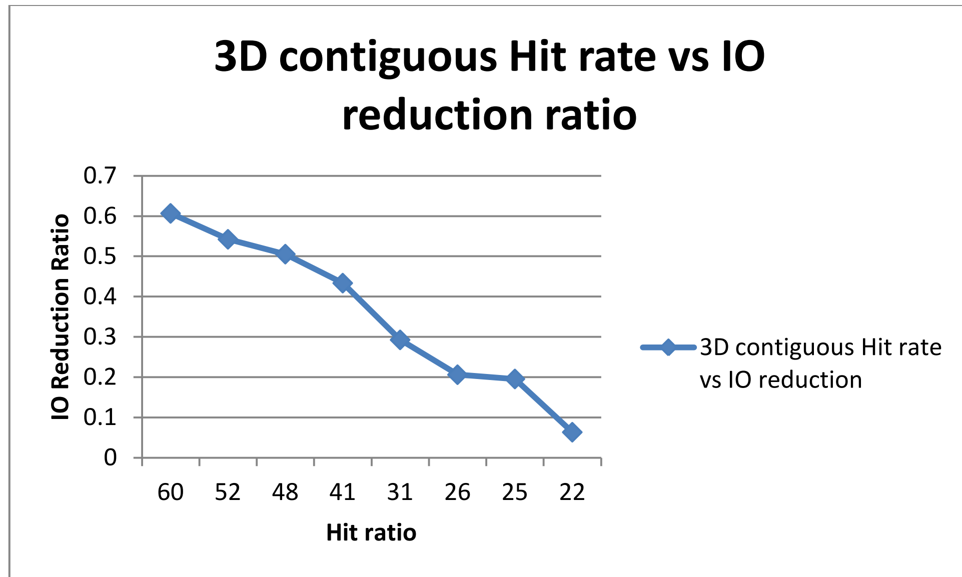


Figure 4.9.3D contiguous IO cost benefit

3D-non-contiguous IO pattern

In 3D non-contiguous IO pattern, the data blocks that are in need are not contiguous in at least one dimension. There are many different scenarios that can arise in case of the 3D non-contiguous IO requests. Figure 4.10 shows some sample IO requests on a 3D dataset.

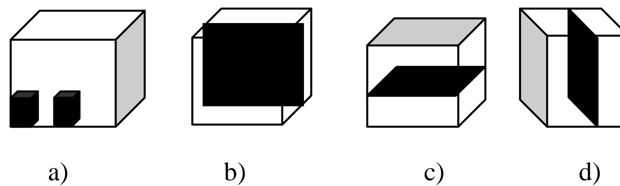


Figure 4.10. 3D Non-contiguous patterns

a) All the dimensions b) Z-axis c) Y-axis d) X-axis

In the Figure 4.10 the shaded region shows that the data plane/data block is already present in-memory. Thus these are called holes. Below graph shows the benefit for these kinds of IO patterns. I have used 1GB dataset to verify the below result.

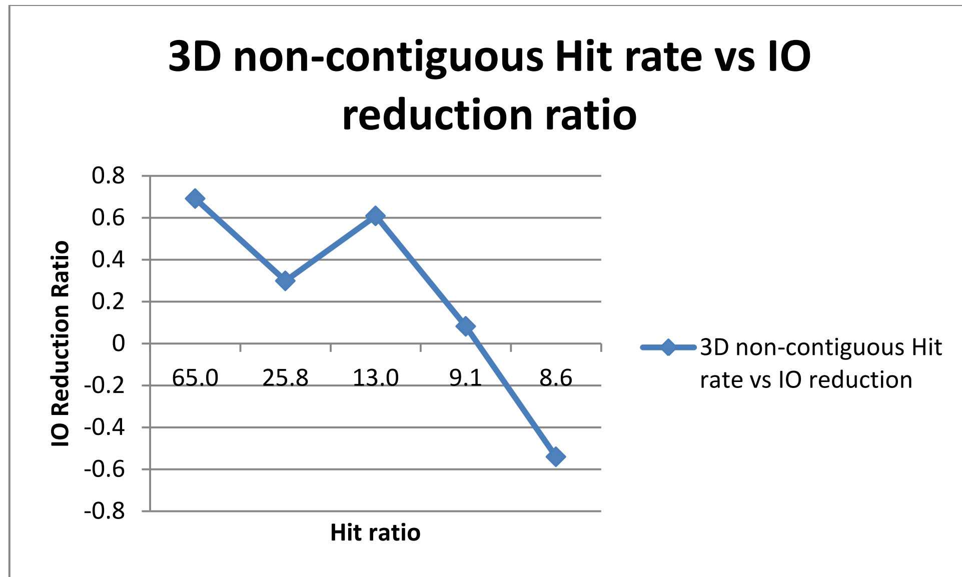


Figure 4.11.3D non-contiguous IO cost benefit

The graph shows an uncertainty in the improvement of the efficiency as there are many factors to be considered in 3D. In 3D the non-contiguous nature can be across any directions and there are many permutations in which the non-contiguous nature can appear. Thus there are various factors that can affect the IO reduction ratio.

As discussed above, number of holes is one such factor that affects the performance or the IO reduction ratio. Figure 4.12 describes the effect of holes in a 3D dataset. I have used 1GB dataset to verify the below result.

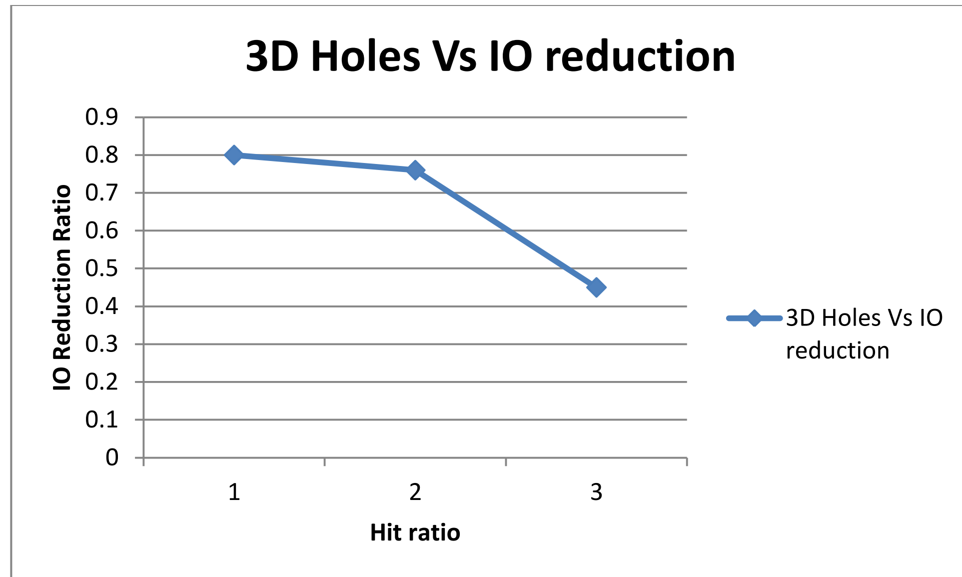


Figure 4.12. Holes Vs IO reduction ratio

Initial Verification of Lineage tracking

Lineage tracking module performs the re-computation of the previous steps in-order to fetch the lost dataset. This module is essential for automatic retrieval of the lost datasets. As per our claim, the computations can be much faster than retrieving the data from the disk. This claim proves to be true when the amount of computation is less. But as the amount of computation is increased/if the computation is much more complex, the behavior of re-computation is not as expected. This is because the amount of computation time drastically increases in comparison to the time needed to fetch the data from disk. Consider a sample workflow where step 1 involves selecting the subset of a file and step 2 is to compute the maximum on the selected subset.

In a traditional model, the subset is computed and it is stored onto disk. This file is later read by step 2 and the maximum is computed.

By in-memory design, we stored the subset in memory. As in-memory data is not persistent, assume this portion of the dataset is lost. In order to re-compute the lost portion of the dataset, we stored the lineage information, which provides us the details

about the operation and its parent dataset. With this information, we performed the re-computation and placed the subset again in-memory and performed the maximum computation.

With these two experiments, we could identify the effectiveness of our proposed lineage model. Figure 4.13 shows the relationship between the amount of re-computation and the IO reduction ratio.

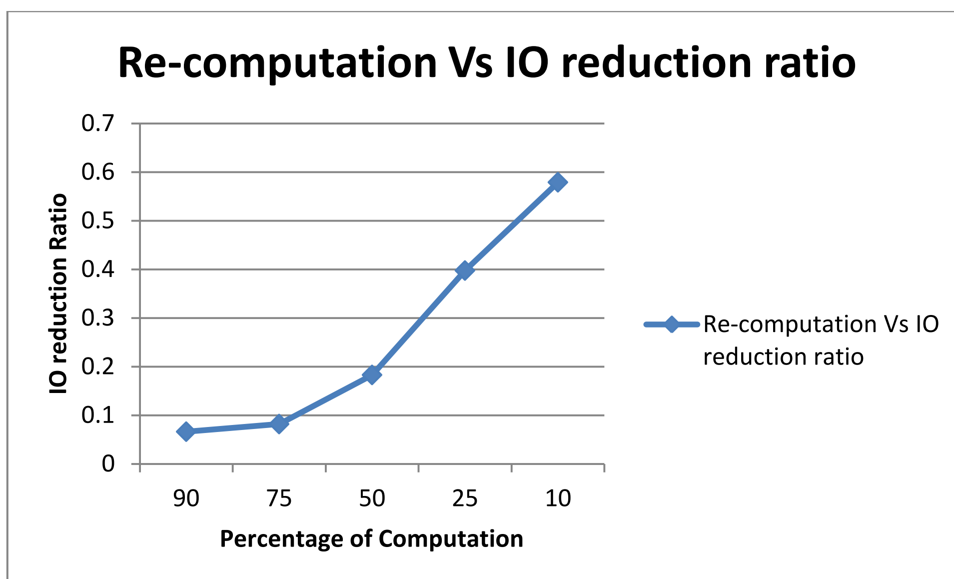


Figure 4.13. Re-computation Vs IO reduction ratio

As the amount of computation decreases, the IO reduction ratio/efficiency is more. But, when we compare the same percentage with the large amount of dataset, there is a decrease in the IO benefit. Thus the amount of re-computation not only depends on the data to be fetched, it also depends on the total size of the data set in the worst case scenario when the original dataset is not in-memory. The dataset size used for this evaluation is 0.8GB. Thus, the read operation to fetch the entire base dataset from the disk in the in-memory computation takes much time which shows the variation though the amount of computation is the same. This computation is the worst case of our in-memory design where the base dataset itself is lost.

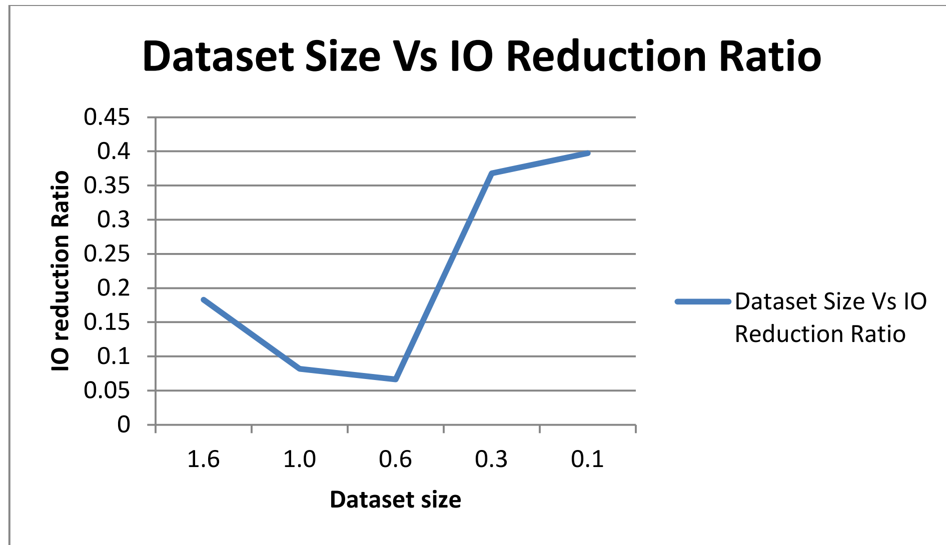


Figure 4.14. Source Dataset size Vs IO reduction ratio

As we discussed previously, the size of the source dataset plays a major role in determining the IO reduction ratio. Figure 4.14 shows a general trend of increasing IO reduction ratio with the decrease in the dataset size. This is mainly because the amount of data to be fetched for the re-computation is reducing as the dataset size decreases.

CHAPTER V

RELATED WORK AND ITS COMPARISON

Growth in technical advancement demands the faster and easy access of information to make reliable and smart decisions. In this regard, in-memory processing is an emerging technology that is gaining the attention of many computer scientists. This research is aimed at creating an in-memory plug-in for HDF5 in Scientific Data Management domain. Though these kinds of architectures exist in the other domains like databases and cloud, these architectures have not gained much attention in scientific domain. These models have rich set of information, which we can always look up and improve the in-memory designs in the scientific data management domain.

In-memory Cloud Design

Considering the cloud environment, Apache Spark (M. e. Zaharia) is one such example for in-memory framework. It is a parallel data processing framework which helps in real-time analytics and iterative calculations by storing the results in-memory. Storing the results in-memory lowers the latency in the computation of these kinds of applications. A spark application consists of a driver program which runs the main function and executes the parallel operations. Spark achieves in-memory framework by introducing an interface named Resilient Distributed Datasets, which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel (Spark Programming Guide - Spark 1.2.0 Documentation). RDDs provide the information about the re-computation of the lost datasets and keep the dataset in-memory. RDDs are ephemeral i.e. they are lazy and they live for short span of time. RDDs support two operations: transformations create a new dataset from existing ones, and actions return a value to the driver program after running the computation on the dataset. Transformations in spark are lazy, i.e., they do not compute their results immediately. Instead, they remember the transformations that have to be applied to any base dataset. The transformations are only applied when an action requires a result to be returned to the driver program making spark more

efficient. Though the computation cost is more, the use/fetch of large dataset from the disk is avoided until the action phase. Once the data is fetched for the computation, they are cached in memory for future computation. This caching avoids the disk read again when the same data is requested for different transformations/actions. The intermediate RDDs after applying the transformations can also be saved in memory to avoid the re-computation. But since the main memory storage is very limited, chances exist that these RDDs can be lost. Fetching these lost datasets is highly essential in this in-memory framework as RAM is not persistent. Spark handles this persistency issue in a great way by storing their computation steps in the RDDs as lineage information. They re-used this information to reconstruct the RDDs again when they are lost from memory rather than fetching it from disk. This module in spark has motivated and provided a way to implement in-memory architectures in our field of study.

In-memory Database Design

A Database Management System is another place where in-memory architectures are greatly used these days. In-memory database (IMDB)/Main Memory Database (MMDB)/Memory resident Database are Database Management Systems that rely primarily on the main memory for storing the databases. These database systems are faster than the traditional disk based systems as their internal optimization algorithms are simpler and are executed with fewer CPU instructions. When a query is executed, if the data is already present in-memory, time taken to search the disk i.e. seek time is eliminated providing faster and predictable performance than disk.

TimesTen is an in-memory, relational database management system with features like persistence and recoverability. The idea of the TimesTen database system is to make the data reside in main memory during the run-time to reduce the response time which in-turn improves the throughput even in commodity hardware. TimesTen is designed to operate more efficiently in the application's address space. TimesTen can be configured to operate entirely in-memory or it can be configured for disk-based environments to log and checkpoint data to disk. In the in-memory design of TimesTen, the data is brought entirely into the application memory to avoid the IPC

call to the server. This avoidance benefits the application as it need not look for the data from other address spaces/ from the server to avoid the communication overhead that is involved in invoking the IPC call. In case of the disk-based model of TimesTen, they tried to capture the log information about the events on the TimesTen database and are written to disk. The snapshots of the TimesTen database are also placed on the disk as a checkpoint to retrieve them back in case of any failure. This model of combining the in-memory design with the disk-based model has motivated the introduction of a similar feature in the in-memory design scheme of Scientific Data Management domain. (Oracle TimesTen).

Existing technologies and its comparison with in-memory design

Below are some of the existing technologies along with their comparisons with our in-memory design which are aimed at decreasing the IO cost.

Firstly, HDF5 has an existing technology of capturing the entire image (Operations) of the file in-memory, but it does not provide any fault-tolerant technology for retrieving the lost portion of the file from the parent file. This kind of storage can greatly hinder the application performance when there is a loss in the in-memory data.

Secondly, HDF5 has an existing feature of storing the metadata in the cache (The HDF5 Group (2008)). Though caching helps in retrieving the dataset faster from the underlying disk, it might not be very efficient for those applications where the results are re-used by the next set of operations. This approach of storing metadata is not specifically addressing applications of the above type. But, our approach of storing the results focuses on the major set of scientific applications to provide increased IO performance.

Thirdly, HDF5 has a feature of storing one dataset in different formats redundant on-disk (The HDF5 Group (2008)) to fetch it faster. Though this feature provides the improved performance, tracking the IO patterns and achieving greater performance depends on the IO request pattern and the patterns in which we have

stored. We can select only the most optimal one among the stored patterns. To avoid such redundancy, we can store the results in-memory and if lost we can re-compute them from the previous stages.

Existing applications like Fast Analysis with Statistical Metadata (FASM) (J. a. Liu), are aimed at integrating sub-setting and applying statistical metadata to improve the query and the data analysis performance. Though this application helps many data intensive applications to retrieve the data faster, they don't make use of the in-memory storage schemes to store their sub-setting results for future re-use. Storing the sub-setting results for future re-use for the same request, can greatly improve the performance in the retrieval of datasets. In addition, if the dataset itself is stored in-memory then fetching it from the disk can also be avoided. So, our in-memory design can be potentially more beneficial than FASM, which involves much computation and retrieval of data from disk.

Many applications which deal with the provenance information, the metadata about the lineage/tracking information is lost after the application is closed. But here, we tried to store this information in-memory as a persistent store globally, to allow other applications re-use them in the future to avoid the unwanted retrieval of data from the disk when the same data is re-used by a different sets of applications. This in-memory storage is the major difference between our approach and the existing provenance related proposals used in the computation jobs of Mapreduce (Dean) and Dryad (Isard)

HDF5 in-memory plug-in has made use of the existing Virtual Object Layer which will make it easy to integrate existing applications with the new in-memory architecture without much modification in the code to help many scientific applications like NaSt3DGP (csar-advice@cfs.ac.uk) to migrate easily to our in-memory HDF5.

CHAPTER V

CONCLUSION AND FUTURE WORK

With respect to growing scientific applications and their workflow, storing the intermediate results and re-using them is very essential. With this re-use as the major consideration, we have proposed a new architecture named *In-memory HDF5* built on top of HDF5 library. We have provided our initial evaluations for the proposed idea and certain other feasibility study experiments to support the claim that in-memory designs prove more efficient than the disk-based model. Our study had helped us to come up with the conclusions below.

- In-memory design for contiguous IO requests proves to be more efficient and more predictable than the non-contiguous IO requests.
- Holes in the IO request can drastically reduce the efficiency of in-memory design as they reduce the contiguous nature of the IO request.
- Distance between the holes also show variations in the efficiency achieved through our in-memory design.
- Lineage tracking proves to be more efficient when the amount of re-computation is less.
- Lineage tracking proves to be less efficient if the size of base dataset to be retrieved from the disk is higher.

Below are the two major directions in which we intend to work in the future to achieve a great improvement in our in-memory design.

- How can we select the best in-memory block when there are two or more blocks can satisfy the in-coming IO request?
- If two in-memory blocks need to be combined for a given IO request, how can we join these IO blocks to produce a better result?

BIBLIOGRAPHY

Robert R. McCormick School of Engineering and Applied Science, Northwestern University. Parallel netCDF (PnetCDF). 19 February 2015. 2 February 2015. <http://cucis.ece.northwestern.edu/projects/PnetCDF/>.

Chang, Yingli, Dongmei Huang, and Yu Liu. "Introduction and Application of the Parallel 3D Fluid Dynamics Software Package NaSt3DGP." IEEE International Conference on High Performance Computing and Communications. 2008.

csar-advice@cfs.ac.uk. Visualization Support Examples. 23 August 2004. 2 February 2015. <http://www.csar.cfs.ac.uk/services/support/vis_support/vis_examples.shtml>.

Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM (2008): 107-113.

Engel, Martin. Research Group of Prof. Dr. M. Griebel . 15 March 2014. 2 February 2015. <http://wissrech.iam.uni-bonn.de/research/projects/NaSt3DGP/documentation/userguide/index.html>.

Engle, Cliff, et al. "Shark: fast data analysis using coarse-grained distributed memory." Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012.

Folk, Mike, Robert E. McGrath, and Nancy Yeager. "HDF: an update and future directions." Geoscience and Remote Sensing Symposium, 1999. IGARSS'99 Proceedings. IEEE 1999 International. IEEE, 1999.

Howells, David. "Fs-cache: A network filesystem caching facility." Proceedings of the Linux Symposium. 2006.

Installing h5utils on MAC. n.d. <https://www.underworldproject.org/forums/viewtopic.php?t=6&p=10>.

Isard, Michael, et al. "Dryad: distributed data-parallel programs from sequential building blocks." ACM SIGOPS Operating Systems Review 41.3 (2007).

Kallman, Robert, et al. "H-store: a high-performance, distributed main memory transaction processing system." Proceedings of the VLDB Endowment 1.2 . 2008. 1496-1499.

Kernert, David, Frank Köhler, and Wolfgang Lehner. "Bringing Linear Algebra Objects to Life in a Column-Oriented In-Memory Database." In Memory Data Management and Analysis 2015: 44-55.

Kitware Inc. VTK - The Visualization Toolkit. n.d. 2 February 2015. <http://www.vtk.org/>.

Koziol, M. Chaarawi and Q. "HDF5 Virtual Object Layer." Technical Report. n.d.

Lahiri, Tirthankar, Marie-Anne Neimat, and Steve Folkman. "Oracle TimesTen: An In-Memory Database for Enterprise Applications." IEEE Data Eng. Bull.36.2 (2013): 6-13.

Liu, Jialin, and Yong Chen. "Improving data analysis performance for high-performance computing with integrating statistical metadata in scientific datasets." igh Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion. 2012 SC Companion:. IEEE, 2012.

Liu, Jialin, Surendra Byna, and Yong Chen. "Segmented analysis for reducing data movement." Big Data, 2013 IEEE International Conference on. IEEE, 2013. 2013.

Moses, Edward I. "Ignition on the National Ignition Facility: a path towards inertial fusion energy." Nuclear Fusion. 2009.

Oak Ridge National Laboratory. Oak Ridge Leadership Computing Facility. n.d. 2 February 2015. <https://www.olcf.ornl.gov/center-projects/adios/>.

Operations, The HDF Group - File Image. "File Image Operations." May 2012. http://www.hdfgroup.org/. 2 February 2015. <http://www.hdfgroup.org/HDF5/doc/Advanced/FileImageOperations/HDF5FileImageOperations.pdf>.

Oracle TimesTen. "TimesTen-Documentation ." Vers. 6.0. March 2006. www.oracle.com. Oracle. 19 2 2015. <http://download.oracle.com/otn_hosted_doc/timesten/603/TimesTen-Documentation/arch.pdf>.

Russ Rew, Glenn Davis, Steve Emmerson, Harvey Davies, Ed Hartnett, Dennis Heimbigner and Ward Fisher. NetCDF: Overview. 12 February 2015. <https://www.unidata.ucar.edu/software/netcdf/docs/index.html>.

Schaffner, Jan, et al. "Predicting in-memory database performance for automating cluster management tasks." Data Engineering (ICDE), 2011 IEEE 27th International Conference. 2011.

Spark Programming Guide - Spark 1.2.0 Documentation. n.d. <http://spark.apache.org/docs/1.2.0/programming-guide.html>.

The HDF Group (2014). HDF5. 13 November 2014. 2 February 2015. <http://www.hdfgroup.org/HDF5/>.

The HDF Group (2015). The HDF Group - Information, Support, and Software. February 2015. 2 February 2015. <http://www.hdfgroup.org/>.

The HDF Group. Chapter 7: HDF5 Dataspaces and Partial I/O. n.d. 2 February 2015.

The HDF5 Group (2008). "HDF5-Tutorial-PDF." 9 September 2008. http://www.speedup.ch/. 2 February 2015. <http://www.speedup.ch/workshops/w37_2008/HDF5-Tutorial-PDF/HDF5-Cach-Buf.pdf>.

The HDF5 Group (H5Utils). n.d. <http://ab-initio.mit.edu/wiki/index.php/H5utils_release_notes>.

The HDF5 Group Developers. HDF5 specification and Developer's Guide Version 4.2.11. 2015.

Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010.

Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

APPENDIX A

GLOSSARY

Term	Definition
Contiguous IO requests	Contiguous IO requests are those in which the requested data is in a linear fashion without any interleaving blocks.
HDF5	Hierarchical Data Format 5 (HDF5) is a data model, library and file format for storing and managing data.
HDF5 dataset	HDF5 dataset is a multidimensional array of data elements, together with supporting metadata.
HDF5 group	HDF5 group is a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.
Holes	Holes are nothing but the interleaved portion of data that are either not needed by the original IO request/the data has already been present in-memory, which need not be fetched from the disk.
IO reduction ratio/Efficiency	IO reduction ratio/Efficiency can be defined as the ratio between the difference in the IO cost for fetching the entire data from the disk and that of fetching only the data that are not in-memory to the time taken to fetch the entire data from disk.
NaSt3DGP	NaSt3DGP is a C++ implementation of a solver for the incompressible, time-dependent Navier-Stokes equations in three dimensions.
Non-contiguous IO requests	Non-contiguous IO requests are those which request data by interleaving certain blocks of data.

TERMS	EXPLANATION
Transformation	Data change between consecutive steps e.g. transformations, statistical operations, creating groups in HDF5, converting the file extensions, etc
Transformation consistency	When same sequence of transformations is applied on the same set of input files, the output should remain the same.

APPENDIX B

INSTALLATION AND COMPILATION COMMANDS

PHDF5 Installation

$ CC=/lustre/work/apps/openmpi/bin/mpicc ./configure --prefix=<install-directory>

$ make # build the library

$ make check # verify the correctness

$ make install

HDF5Utils Installation **(Installing h5utils on MAC)**

$CC=/lustre/work/apps/openmpi/bin/mpicc ./configure

CFLAGS=I/home/kvijayak/phdf5/hdf5-1.8.12/hdf5/include/

CPPFLAGS=I/home/kvijayak/phdf5/hdf5-1.8.12/hdf5/include/

LDFLAGS=L/home/kvijayak/phdf5/hdf5-1.8.12/hdf5/lib/

$ make # build the library

$ make check # verify the correctness

$ make install

File Conversion h5 to VTK

$h5tovtk -o vtkfile h5file

Compilation of HDF5 code in C

$/home/kvijayak/phdf5/hdf5-1.8.12/hdf5/bin/h5pcc -o <<objectfile>> <<sample.c>>

Run the object file

$ ./<<objectfile>>