2010

Concentric Layout, A New Scientific Data Layout For Matrix Data Set In Hadoop File System

Lu Cheng
University of Central Florida

CONCENTRIC LAYOUT, A NEW SCIENTIFIC DATA LAYOUT FOR MATRIX DATA SET IN HADOOP FILE SYSTEM

By

LU CHENG

B.S. Xian Jiaotong University 2006

M.S. Xian Jiaotong University 2009

A thesis submitted in partial fulfillment of the requirements

for the degree of Master of Science

in the Department of Electric Engineering and Computer Science

in the College of Engineering

at the University of Central Florida

Orlando, Florida

Fall Term 2010

ABSTRACT

The data generated by scientific simulation, sensor, monitor or optical telescope has increased with dramatic speed. In order to analyze the raw data speed and space efficiently, data pre-process operation is needed to achieve better performance in data analysis phase. Current research shows an increasing tread of adopting MapReduce framework for large scale data processing. However, the data access patterns which generally applied to scientific data set are not supported by current MapReduce framework directly. The gap between the requirement from analytics application and the property of MapReduce framework motivates us to provide support for these data access patterns in MapReduce framework. In our work, we studied the data access patterns in matrix files and proposed a new concentric data layout solution to facilitate matrix data access and analysis in MapReduce framework. Concentric data layout is a data layout which maintains the dimensional property in chunk level. Contrary to the continuous data layout which adopted in current Hadoop framework by default, concentric data layout stores the data from the same sub-matrix into one chunk. This matches well with the matrix operations like computation. The concentric data layout preprocesses the data beforehand, and optimizes the afterward run of MapReduce application. The experiments indicate that the concentric data layout improves the overall performance, reduces the execution time by 38% when the file size is 16 GB, also it relieves the data overhead phenomenon and increases the effective data retrieval rate by 32% on average.

ACKNOWLEDGMENTS

This thesis would not have been possible without the help and support of a number of people. First and foremost, I would like to express my sincerest gratitude to my advisor, Dr. Jun Wang, for the tremendous time, energy and wisdom he invested in my graduate education. His inspiring and constructive supervision has always been a constant source of encouragement for my study. I also want to thank my other thesis committee member, Dr. Shaojie Zhang, and Dr. Jooheung Lee, for spending their time to review the manuscript and providing valuable comments.

I also would like to thank my group members for their generous help and guide. Without their selfless sharing and the inspiring discussions, I cannot finish the project smoothly.

I also want to dedicate this thesis to my family, for all their love and encouragement through my life.

TALBE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

In recent days, more and more scientific applications have been benefited from the MapReduce framework [9]. These applications share the property that they generate, collect and maintain vast volumes of data, and also require large computing resource to process data [7]. For example, earthquake prediction and analytic model collect up-dated and detailed data of earth activity around the world [8] to let geologists generate a more accurate and efficient earthquake analytic model. These data are collected in every second and delivered to computation unit for analysis. Many other scientific research applications such as bio-information model, vision simulation, climate prediction and realistic graphic animation share same properties generate, store and process multi-terabyte data. MapReduce is a good candidate for these applications as MapReduce jobs are distributed into multiple sub-jobs and processed concurrently. The distributed property improves the processing speed and meliorates the execution efficiency.

For many analytical applications, data set are generated and stored in a matrix manner naturally. For example, the weather monitor application senses and records the temperature and humidity variation in real time, and scientists analyze posted data to forecast the future weather changes.

One impelling analytic requirement is to compare the data values among different periods in the same day or the same time among different days. Apparently, storing the data set into a matrix manner will bring in performance benefit for the future analysis. Instead of reading the entire data set, the scientist just needs to read the data set in the target row to analyze the temperature change during the same day or to review the data set in the target column to analyze the humidity variation in a month. Therefore, the way the dataset is stored in a file system has an intimate relationship with how it is accessed. Besides, the same data set may be utilized by different scientists for different research works, and each scientist will process the data set in a different way. For example, the cosmic data bank is a project which a group of scientist working on cosmological simulations, which are employed in variety of projects, from mass power spectrum analysis to halo mass function. The simulation data with location and velocity information can be presented in a cube and accessed in different ways, parallel to X_Y plane or parallel to X-Z plane.

In distributed file systems like HDFS (Hadoop Distributed File System) [6] adopts MapReduce framework, the data is stored sequentially and read stream in default. Unfortunately, such storage feature breaks the aforementioned intimate relationship between data layout and data access pattern. Using the weather monitoring application as an example, when file is stored in HDFS sequentially, the data in the same column is separated and distributed among the entire file system. When data in one particular column is needed, instead of just reading one column, the whole file will be accessed. An inappropriate data layout will affect the data processing efficiency as improper data layout results in reading excess amount of data than actually needed. Meanwhile, storing the data set in a file system with one access pattern cannot fit various applications with different access patterns. After the monitor data set is generated and stored in a file system, the analytic applications with various access patterns will access the data set to perform different data analyses. For example, temperature data is used for analyzing data fluctuations in different time periods, like in a day or in a year. Based on the specific analytic requirement, the data set will be accessed in either row based or column based.

In order to deal with the aforementioned challenges, we propose a new concentric data layout scheme. Concentric data layout maintains the matrix property in chunk level. Its unique combination of row based access pattern and column based access pattern makes it works well for many scientific applications which process matrix data set. In concentric data layout, affiliated data is stored into the same chunk and hence maintain the original logical properties. As the data is stored in two dimensional manners, accessing the data in either row or column will lead to comparable performance, and realize the optimal overall performance when applications access the same matrix data set in different patterns. The concentric data layout aims to mitigate the small I/O problem, improve the data utilization rate and thus significantly improve the I/O performance by reducing the total number of chunks being accessed.

The paper is organized as follows, section 2 introduces the background of MapReduce framework and matrix related data access pattern. In section 3, we propose the concentric data layout in detail and discuss the experimental results in section 4. Section 5 introduces the related work while the conclusion and further works are discussed in section 6.

CHAPTER 2 BACKGROUND

In this section, we introduce the HDFS, MapReduce framework and data access patterns of matrix data set in brief.

2.1 Data Intensive HPC

In recent year, scientific research became increasingly rely on computing over large data sets. This phenomenon is usually referred as “e-Science” and the applications with “e-Science” property [11] require a new system design which computation and storage are coupled together.

Many scientific research problems incur large columns of data produced by different applications, different sources from numerous locations with various formats. Besides, the analytic application requires strong storage and computation power to perform further data computation and analysis.

For geographic, the more detailed and more accurate finite data will enable scientists to model the effect of a geological disturbance and the probabilities of earthquakes occurring in different regions more accurately and instantly. The analytic models are continually updated and analyzed. The continent movement, temperature, humidity as well as many other parameters are monitored all the times at numerous locations around the world and the data are collected for the forecasting of earthquakes, volcano eruptions and hurricanes.

For biological, the computational biology involves comparing genomic data from different species and different organisms [12]. The fast-accumulating data mainly consist of DNA, RNA and protein sequences due to the state-of-art sequencing technology. Large data sets are collected as new sequences are discovered and new forms of derived data are computed. The most complicated data are the relational data about the associations among the proteins, RNAs and DNAs. As a basic procedure for biologists and medical doctors, sequence alignment is a time-consuming work.

For astronomic and cosmology, the modern telescope can generate terabyte data per year and it is expects in the future, petabyte of data will be produced. The massive amounts of imagery data is collected daily and additional results are derived from computation applied to that data.

The above mentioned examples in scientific researches clearly indicates that an increasing number of data-intensive HPC problems are arising with the requirement of collecting and maintaining very large data sets and applying vast amount of computational power to the data. As these analytic applications are run on the computer cluster, data are copied from the storage cluster to the computer cluster back and forth. This data replication is extremely time consuming as significant amount of their execution time is spent on I/O transformation, so new trend is to apply these analyses in distributed MapReduce framework like Hadoop [10]. However as Figure 2-1 indicates, these kinds of applications are in the intersection of traditional HPC applications and traditional DISC. They require both storage capacity and computation ability.

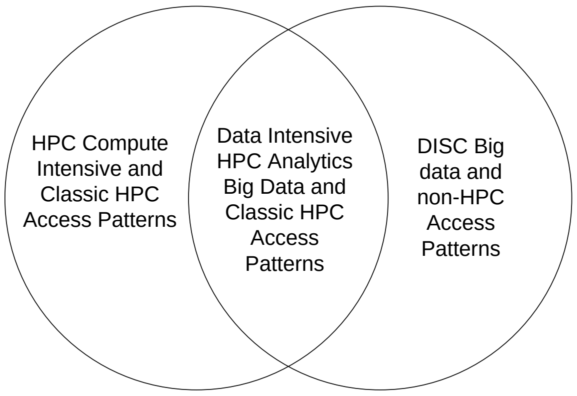
HPC Compute Intensive and Classic HPC Access Patterns
Data Intensive HPC Analytics Big Data and Classic HPC Access Patterns
DISC Big data and non-HPC Access Patterns

Figure 2-1 Data Intensive HPC Analytics Applications

2.2 HDFS and MapReduce Framework

Hadoop is inspired by MapReduce, a programming model and an associated implementation for processing and generating large data sets. MapReduce aims to let user perform the simple computations with large data set as well as hides the messy details of parallelization, fault-tolerance, data distribution and load balancing. The nature of huge amounts of data determines that the distributed computation is a better choice than the sequential computation for many problems.

The Hadoop architecture consists of two servers: namenode and jobtracker and amount of other servers which function as tasktrackers and datanodes. Namenode and datanode are two main components of Hadoop. The single namenode is a master server that manages the file system namespace and regulates the access to files by clients, it not only responsible for Hadoop file system data management, but also responsible for file access and replacement. The datanode, usually one per node in the cluster, is used to manage storage attached to the nodes that they run on. It stores the file system data, manages replication tasks and services all data read/write requests from clients based on namenode's direction. The jobtracker is responsible for handling all jobs which submitted by client application. Besides, it maintains the task resiliency in the cluster by making scheduling decision and parallelizing the client applications across the cluster. The jobtracker monitors all running task on the cluster, killing and restarting tasks when they fail, hang or disappear during the operation. The tasktrackers in Hadoop is responsible for running the client application via instructions from the jobtracker. The jobtracker and the tasktrackers comprise the architecture for MapReduce programs to run on.

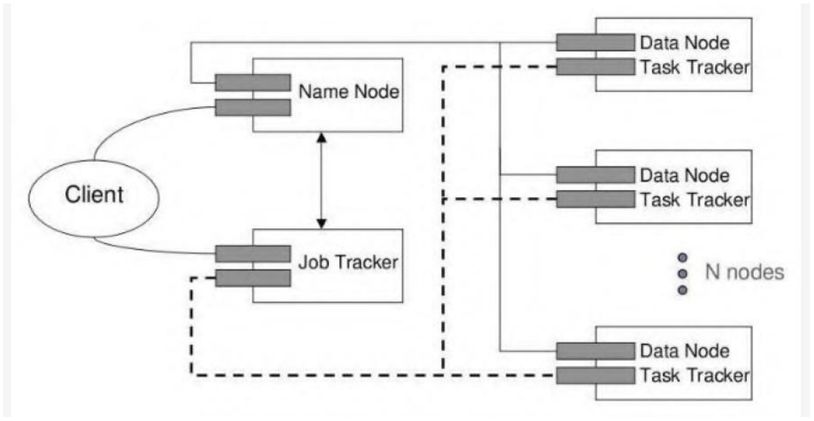

Figure 2-2 Hadoop Architecture

Figure 2-2 indicates the Hadoop architecture. It shows that in Hadoop, in order to achieve better data locality, one server is servers as namenode, one server is servers as jobtracker and other servers are configured as datanode and tasktracker. As Hadoop lacks of bandwidth needed for the cluster to function appropriately, it allows performance on commodity computing without a fast, expensive interconnect. In Hadoop, namenode becomes its own server because Hadoop keeps all file system metadata in main memory, working as an own server will not slow the file access which caused by strain on the namenode from serving data and metadata requests. Meanwhile, to ensure the task resiliency in the cluster, the jobtracker is running on multiple daemons.

The Hadoop framework consists of two main components: Hadoop Distributed File System (HDFS) and the MapReduce framework. These two important components working together to make sure Hadoop is reliable and easier for programming.

2.2.1 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is modeled very close with Google file system. It is a distributed file system designed to run on commodity hardware [1]. The approach to this file system assumes that failure in a large scale computing environment happens frequently. The HDFS stores the data across multiple nodes (default number is 3), this replicate storage ensures that in HDFS, the file stored are always intact in three separate places across a cluster. This distributed file approach guarantees the system resiliency in Hadoop without the requirement of RAID storage.

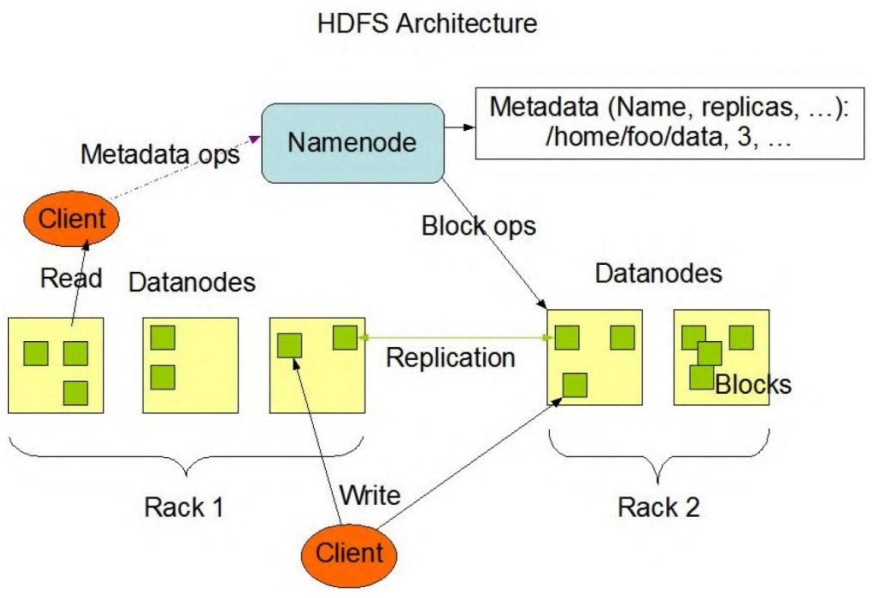

Figure 2-3 HDFS Architecture

The Figure 2-3 is a conceptual model for HDFS. It can be noticed that client first directs file queries to the namenode, namenode then directs the file request to the appropriate datanodes and the datanodes supply the client application with the data. In HDFS, the replication of the file is located across servers in a rack and across server racks. When file chunks are written to datanode across the HDFS, the namenode tries to group at least one replicated chunk on the same server rack as the primary and then another chunk to an adjacent rack of datanodes; meanwhile it ensures that no two replications of a chunk are stored to the same datanode. This mechanism applies certain data locality and fault resistance. The namenode pings every datanode periodically. Once no response is received from a datanode in a given time, the namenode marks it as failed and reassigns the job to another datanode. Therefore when a server hardware failure happens, the namenode will recover the health of the cluster without user's intervention. The ability of the HDFS to recover from system failures automatically without neither lose of service nor needs of user's intervention making HDFS a very power tool for data intensive applications.

2.2.2 MapReduce

MapReduce framework is introduced by Google to support distributed computing with large data sets on cluster of computers [2]. MapReduce has map and reduce two phases in its programming. The programmer has a map operation, one parallel operation is processed during the map operation in which results are collected at the intermediate combine phase; then reduce is performed to get together these intermediate values to form a smaller set of values before the output data becomes persistent storage [14]. The MapReduce framework works exclusively on [key, value] pairs. An input of [key, value] pairs are processed by the map operation which produces a set of intermediate key based on the input pairs, the reduce phases receive these intermediate keys and outputs a smaller possible result.

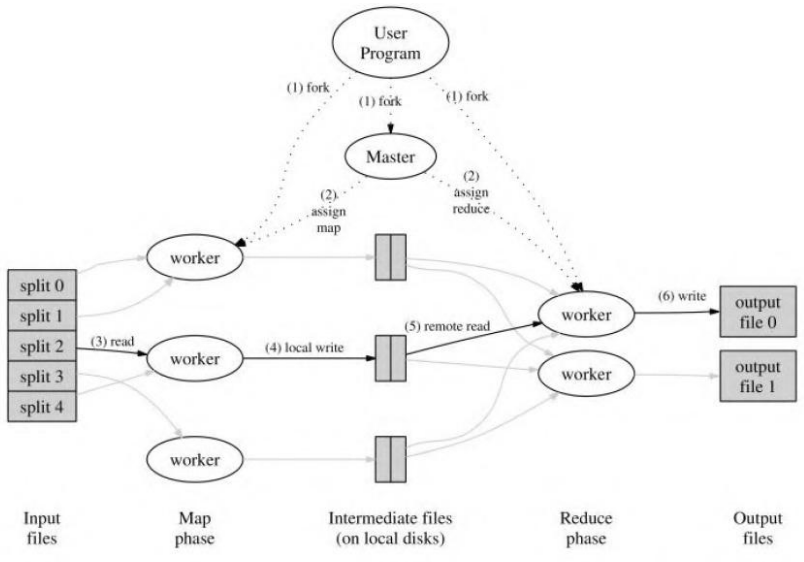

Figure 2-4 MapReduce Work Flow

The execution flow works as Figure 2-4 shows. All map and reduce operations are tasks run on the tasktrackers in the Hadoop cluster. Jobtracker monitors these map and reduce tasks from inception to completion. During the combine phase of the MapReduce operation, intermediate output data from all map tasks on an individual tasktracker is written to local storage for the reduce phases. The combine operation can result in a quick local reduce before the file is passed to a global reduce function. The Hadoop system has the share-nothing property, which means during the operation there is no intercommunication between any map task in map phase and no intercommunication between any reduce task in reduce phase. These two operations: map and reduce, allow a large parallel dataset to be operated very quickly with the assurance of task resiliency.

2.3 Data Access Pattern

Data access pattern is mainly decided by the specific application acquirement. A proper data layout will benefit the process efficiency and improve the I/O performance as the relationship between the file and the process will be determined in terms of spatial organization and temporal ordering [15].

2.3.1 Continuous Data Access Pattern

The continuous access pattern [13] is the most widely used data access pattern. In the continuous access pattern, data is stored sequentially and accessed in round-robin manner without considering data dependency. This data access pattern is widely used among applications in which the data are independent, and the task can be divided into multiple sub tasks and processed synchronously. This model fits best with HDFS because the features of streaming access and batch process match with continuous data access pattern perfectly. In HDFS, because the chunk is the smallest storage unit, the task node processes entire data in the assigned chunk, no matter the data are required to be processed or not. For data independent application, each data in the chunk is useful, and hence avoids the potential performance waste which caused by processing unnecessary data. The application with continuous data access pattern can yields the best I/O performance when processed by MapReduce framework.

2.3.2 Matrix Data Access Pattern

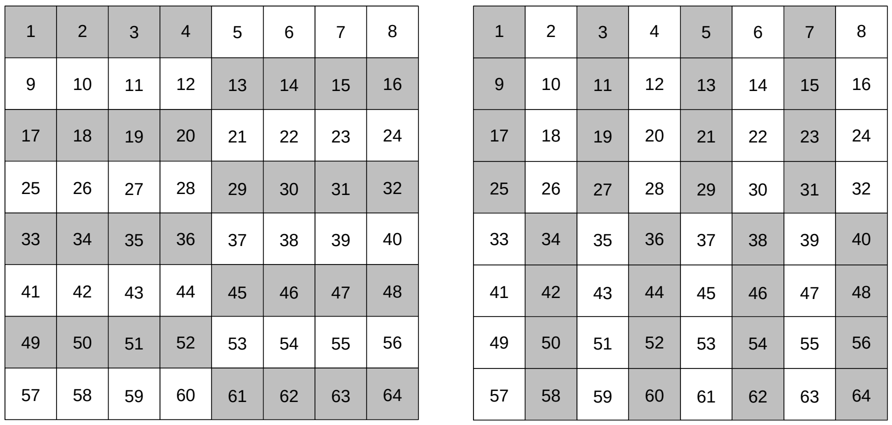

Figure 2-5 Matrix Data Access Pattern

As Figure 2-5 shows, row-based or column-based access patterns are two basic matrix access patterns for matrix data set. It is widely used in scientific analytic applications. For many scientific applications, data can be stored with dimensional manner in logical file, it helps to keep data dependency between each other. However, when the data in logical file with dimension property is stored into physical storage media, data lose their higher level property in file system and become stream bytes. The default continuous data layout cannot adapt to matrix data access pattern well. Once the data is stored continuously in row base, the data access efficient will be impacted for column data access pattern.

2.3.3 Group Data Access Pattern

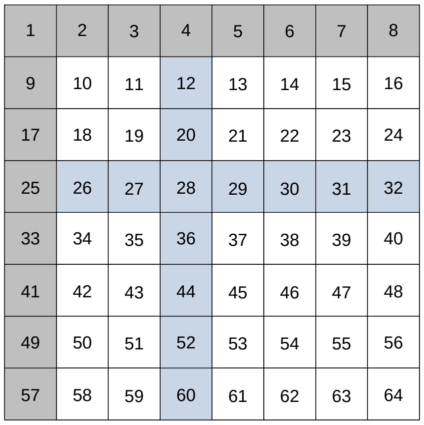

Figure 2-6 Group Data Access Pattern

Some analytic applications require complex data analysis like group access pattern. Group access pattern is a combined data access pattern which generally used in matrix computation, like matrix multiplication. For two-dimensional matrix file, group access pattern involves accessing the row and the column in same matrix set at the same time. The Figure 2-6 demonstrates one example of concentric access that the first row and first column are required. For group access pattern, continuous data layout turned out to be extremely inefficient. The data utilization rate is decreased because no matter data is stored along the row or column, only a small part of accessed data is useful for analysis.

CHAPTER 3 CONCENTRIC DATA LAYOUT

In this section we propose concentric data layout, a matrix-specific data layout optimization strategy to benefit the matrix data access pattern and group data access pattern.

3.1 Problem Description

In the file system, data is stored continuously and read as a stream by default. However, for modern scientific applications, the data access exhibits various pattern due to the nature of applications that generate the data set and the way the data is laid out in the file system. For the matrix data set, it is more often to access the data with matrix data access pattern instead of continuous data access pattern. The analytic applications may also require some complex matrix operations, such as accessing the data in the row and column at same time. Besides, with the development of research and analytic technology, the data which is generated by monitors or simulations becomes more and more complex. For a lot of scientific research, it is neither realistic nor efficient for just one specific application access the data set. After the data is collected, different analytic applications will read the data and make various data analyses. These analytic applications do not necessarily share the same data access pattern. They can exhibit various data access patterns. For example, a weather forecast application collects temperature changes over time. Logically, data is stored in a two-dimensional manner while the X-axis represents the day and the Y-axis represents the different time during a day. When a scientist tries to analyze the temperature in the same time period among different days, the row based access pattern is applied. However, when the temperature variation in same day needs to be analyzed, data in the same column will be processed. This will make different applications apply different data access patterns on the same data set. When storing the matrix file into file system, the traditional continuous storage data layout cannot adapt to the matrix access pattern well, because most of matrix data access pattern retrieve the data non-continuously.

For a matrix data set which retrieves data non-continuously with some matrix data access patterns, the small I/O problem will be generated because the file is treated as linear bytes in the file system, and loses the higher level property at the lower level of the file system. This problem is especially obvious in HDFS. In HDFS, the chunk is the smallest data storage unit which works atomically. After the data is stored in chunks, whenever data is required, the namenode will send the chunk ID which contains the required data to the client, and the client will access the chunk directly and read and process the whole data in that chunk. This structure works well when the whole data in the chunk is required, because the data will be accessed and processed sequentially. However, when data access pattern is non-continuous, the target data will be distributed into several chunks, and only part of the data in the chunk is useful. The default continuous storage data layout results in excessive chunks access with a terrible data utilization rate, and arises in extensive data overload. For example, from the user's point of view, it is natural to store the matrix data in a multidimensional way, as it is easier to explore the data dependency and other information. However, when the file is stored linearly in HDFS, the multidimensional array is flattened into one dimensional array, and the higher level information is lost at the lower level file system. For matrix data access like group access pattern, unrelated data in chunks is also retrieved and processed when the user tries to read the target data from chunks. This phenomenon will results in the small I/O problem as it reads an excess amount of data than required, and decreases the data access efficiency and impact I/O performance.

Meanwhile, another challenging question is raised by the matrix access pattern. The target data assigned to a task may map to a large number of chunks. A single map task with a large number of chunks impacts scheduling schemes. Considering in the Hadoop framework, data is replicated across three datanodes to achieve data reliability, task scheduling which selects the optimal node to perform the task becomes extremely challenging because of the large number of involved nodes. Copying data from distance datanode to local datanode also will cost a great number of resources. Therefore, when storing the matrix file into HDFS sequentially, the matrix data access pattern impacts the performance in two ways. First, it results in reading an excess amount of data than required; second, the stripes assigned to a task may map to a large number of chunks, making the task scheduling extremely challenging. In order to improve the reading efficiency for the matrix access pattern, new data layout needed to be proposed.

3.2 Concentric Algorithm for Two Dimensional Matrix Data Set

We propose the concentric data layout algorithm for the matrix data access pattern and the group data access pattern which are common access patterns in scientific applications. Concentric data layout is a data restructuring strategy which maintains the dimensional property in chunk level.

Figure 3-1 is a matrix file with continuous data layout. The matrix file is an 8×8 two dimension data set with a chunk size of 4 elements. Chunk 1 contains elements 1, 2, 3 and 4, and chunk 2 contains 5, 6, 7 and 8 and so on.

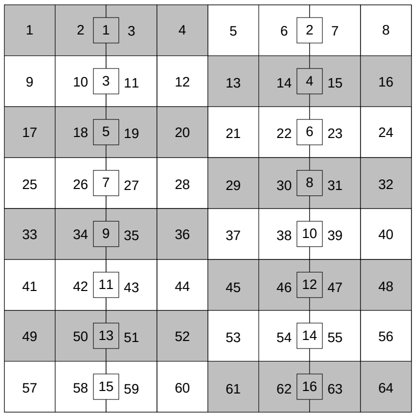

Figure 3-1 Row Based Access Pattern in Matrix Data Set

From the Figure 3-1, we can see the continuous storage method flatten the two-dimensional matrix into a linear sequence of elements. Each element just maintains the information about its peers in the same row, but loses the information about the other neighbors in its columns.

Therefore, this data layout just fits for the row based access pattern. Suppose the first row of element in the array is needed to be processed, the chunk 1 and 2 which contains these elements will be processed. Because all the data in accessed chunks are target data, the data access efficiency is 100%. However, when the data access pattern becomes vertical, the I/O performance becomes unsatisfactory. For example, when the first column needs to be processed, the chunks with even chunk ID will be processed because target data is distributed among these chunks. It greatly deteriorates the I/O overhead as the 8 chunks are retrieved, but only the first elements in chunks are useful. In this case, the data access efficiency is only 25%. When the matrix file becomes larger, the inefficiency will become more conspicuous. Considering a matrix file with a size of $64M \times 64M$ and the elements with the size of 64KB, because the default chunk size in the Hadoop file system is 64M, each row in the file will store in a chunk and there are 64M chunks in total. When only one column of data is needed, the chunks which contain the target file will be processed. Because each row is stored in one chunk, the whole file will be retrieved in order to read one column of data. The data access efficiency will become as low as 0%. The above example sufficiently shows the inflexibility of continuous data layout and demonstrates it cannot adapt the variable access patterns required by the matrix file.

Compared with the default continuously data layout, the concentric data layout maintains the multi-dimensional property in chunk level. The deployment of the matrix file can be represented as a $n \times n$, meanwhile the chunk can be treated as a $k \times k$ sub-matrix. Therefore, the whole file can be divided into multiple sub-matrices and make each sub-matrix have the same size of the chunk size.

Instead of storing the data into the chunk linearly, the concentric data layout stores the data in the same sub-matrix into one chunk. The data within the same chunk not only knows its peers in the same row, but is also aware of its neighbors in the same column. Based on concentric data layout, the big matrix file is divided into multiple sub-matrices; each sub matrix is stored into one chunk. Because the two-dimensional property is maintained in chunk level, it fits better with the matrix access pattern than continuous data layout.

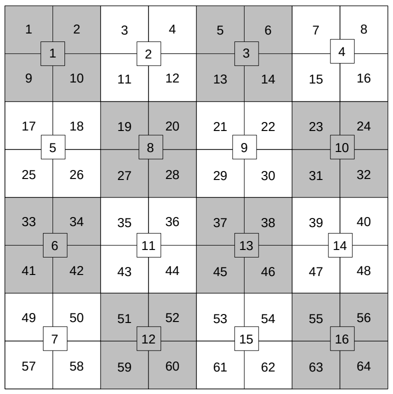

Figure 3-2 Two-Dimension Concentric Data Layout

Figure 3-2 indicates the implementation of concentric data layout in a two dimensional matrix file. It shows the concentric data layout preserves the two-dimensional property in chunks. In Figure 3-2, the file is a two-dimensional matrix with the size of 8×8 and the chunk size is 4.

Therefore, the matrix can be divided into 16 2×2 sub-matrices and each of them contains 4 elements. By applying concentric data layout, instead of storing the elements 1, 2, 3 and 4 into chunk 1, elements 1, 2, 9 and 10 which form the 2×2 sub matrix are stored into chunk 1, elements 3, 4, 11 and 12 are stored to chunk 2, and so on. Compared with continuous data layout, data is stored in multi-dimensional way in concentric data layout. Suppose for matrix in Figure 3-2, when the data in the first row is required, chunks 1 to 4 are accessed. Because these 4 chunks store the data in the first two rows and only the first row of data is required by the client, the data access efficiency is 50%. Compare with the continuous data layout (Figure 3-1), the number of chunks accessed increased from 2 chunks to 4 chunks and the data access efficiency is decreased from 100% to 50%. However, the performance improved in the column access pattern and the group access pattern. When a column of data is required, the same number of chunks will be accessed with the data access efficiency of 50%. Compare with continuous data layout (Figure 3-1), the number of chunks accessed dropped observably from 8 chunks to 4 chunks and the data access efficiency increased from 25% to 50%. When data is retrieved in the group access pattern, the number of accessed chunks is reduced from 9 chunks with continuous data layout to 7 chunks with concentric data layout. The data access efficiency also improved from 41.6% to 53%. Considering the probabilities of each matrix data access pattern are independent, the average number of chunk accessed is dropped from 7 chunks per access to 5 chunks per access. The improvement becomes significant when the file size becomes bigger. For a $64M \times 64M$matrix file with the chunks size of $8k \times 8k$ $(64MB)$, when the data access pattern is row or column based, $16K$ of chunks are accessed with concentric data layout while $64M$ of chunks are accessed with continuous data layout, and the saving is astonishing.

We analyze the average performance between the concentric data layout and the continuous data layout mathematically. In our analysis, we suppose the access patterns are independent and the possibilities for each one are equal. Table 3-1 compares the number of chunks accessed with different data layouts.

Table 3-1 Chunk Amount Comparison between Continues and Concentric Data Layout with Two-Dimension Matrix File

Access Pattern	Row Based	Column Based	Group Based
Continuous	$\frac{n}{k^2}$	$\frac{\mathrm{n}^2}{k^2}$	$\frac{n}{k^2}+\frac{\mathrm{n}^2}{k^2}-1$
Concentric	$\frac{\mathrm{n}}{\mathrm{k}}$	$\frac{\mathrm{n}}{\mathrm{k}}$	$\frac{2n}{k}-1$

The size of matrix file is $N = n \times n$ and the size of the chunk is $K = k \times k$, the matrix file will be stored in $\frac{n^2}{k^2}$ chunks. For the continuous data layout which data is store sequentially, when a row of data is required, $\frac{n}{k^2}$ chunks will be involved. When a column of data is required, $\frac{\mathrm{n}^2}{k^2}$ chunks will be involved. The group access pattern which require both row and column access will require $\frac{n}{k^2}+\frac{\mathrm{n}^2}{k^2}-1$ chunks in average. The average number of chunks accessed with continuous data layout is

$$N_{continuous} = \frac{n}{k^2} \times P_{row} + \frac{\mathrm{n}^2}{k^2} \times P_{col} + \left(\frac{n}{k^2} + \frac{\mathrm{n}^2}{k^2} - 1\right) \times P_{group},$$

the P_{row} represents the probability of row based data access pattern, the P_{col} represents the probability of column based data access pattern and the P_{group} represents the probability of group access pattern. For concentric data layout, $\frac{n}{k}$ chunks will be accessed when processing a row or a column of data, the group access pattern will involve $\left(\frac{2n}{k} - 1\right)$ of chunks. So the average number of chunks accessed with concentric data layout is

$$N_{concentric} = \frac{\mathrm{n}}{\mathrm{k}} \times P_{row} + \frac{\mathrm{n}}{\mathrm{k}} \times P_{column} + \left(\frac{2n}{k} - 1\right) \times P_{group}.$$

As we have already said, the possibilities for each access pattern are independent. The possibilities for row based access, column based access and group based access are equal to $\frac{1}{3}$.

The comparison between two data layout is

$$N_{concentric}/N_{continuous} = \left({4n}/{k} - 1\right) \Big/ \left(2\frac{n}{k^2} + 2\frac{\mathrm{n}^2}{k^2} - 1\right) = \frac{2k}{1+n},$$

k indicates the size of chunk and *n* indicates the size of matrix file. As k is smaller than *n*., fewer chunks will be retrieved with matrix data access pattern when data is stored with concentric data layout. This reduces the data overhead and increases the data efficiency.

The Table 3-2 shows the pseudo code for two-dimensional concentric data layout.

Table 3-2 Two-Dimensional Concentric Data Layout Algorithm

```
Input: the matrix file size $N$;
        the chunk size $K$;
        the data size $a$;
Output: $C_{id}\ for\ each\ data$;
steps:
    Classify element within same sub matrix
    $\frac{N}{K}$, the total number of chunks for matrix file
    $\frac{\sqrt{N}}{\sqrt{K}}$, the number of chunks for each row or column
    $n = \frac{N}{datasize}$, the number of data the matrix file has
    for ($i = 0\ to\ n-1$) do
        $offset\ = i \times datasize$
        $row = \left\lfloor \frac{offset}{\sqrt{N}} \right\rfloor$, determine the row number for data $i$;
        $column = offset - row \times \sqrt{N}$, determine the column number for data $i$;
        $C_{id} = \frac{row \times \sqrt{N} + column}{k}$
```

3.3 Concentric Data Layout for Multi-Dimensional Data Set

In order to make more precise analysis and accurate simulation, scientists are collecting and analyzing more and more complex data. Scientific data formats are introduced to accelerate complex data processing efficiency. In recent research, many simulation data can be stored into matrix and accessed in matrix pattern, matrix data set with higher dimension property become more and more common. For example, in the Cosmic Data ArXiv project, the scientists are working on cosmological simulations and the simulations are employed in a variety of projects. The data which generated by simulation contains the information about the object location (x, y, z) and the velocity $(v_x,\ v_y,\ v_z)$. The data set can be stored in three dimension matrix according to its location, and velocity related processing will become the three-dimensional matrix processing problem. If we extend the two-dimensional concentric data layout into multiple-dimensional, making the data layout fits well with matrix data access pattern, it also reduces the data overhead and improves the processing efficiency for multiple-dimension matrix data set.

In this section we will use a three-dimensional matrix data set as an example to indicate how to apply concentric data layout with multi-dimensional data set.

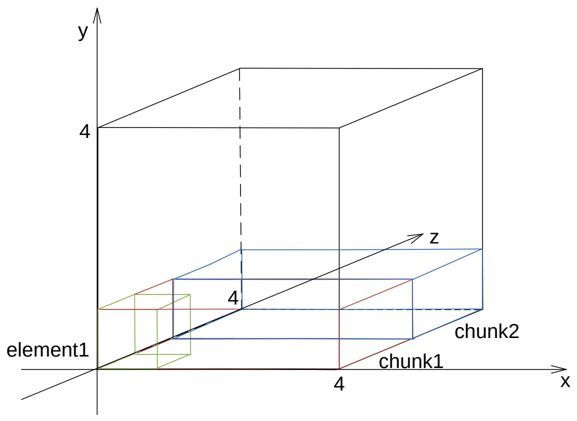

Figure 3-3Three-Dimensional Data Set with Continuous Data Layout

The Figure 3-3 represents a three-dimensional data set of temperature record, the X-axis indicates the time, Y-axis indicates the height while the Z-axis indicates the pressure. Each data in the data set represents the temperature for a given set of (pressure, height, time). Based on this matrix data set, the user can analyze the temperature variation in different time, height and pressure conditions. As Figure 3-3 shows, when data is stored with continuous data layout, data will be stored along the axis sequentially. For example, first fix the pressure and height, stores the temperature data with different time, the data will be stored along the X-axis. Then fix the pressure, stores the temperature data with time and height variation, the data will be stored along the Z-axis in Figure 3-3. At last, stores the data with different pressure, which suggests the data will be stored along the Y-axis. The continuous data storage restrains the data access pattern. For example, it is easy to analyze the temperature change with height and time variation because based on the continuous data layout, data in the same X-Z plan is stored in the same or close chunks. However, when scientists need to study the temperature change with pressure and time variation, the data access will become inefficient. Because the target data which along the Y-Z plan is stored into many different chunks, the client has to read the entire data in the chunks to get the target data. The data accessing rate becomes very inefficient. Take Figure 3-3 for example, the matrix contains 64 elements and is stored continuously. Suppose each chunk stores 8 elements, the matrix file will be stored in 8 chunks. With continuous data layout, the elements which parallel to X-Z plane will stored into the same chunk. When these elements are needed, the relevant chunks will be processed, like chunk 1 and chunk 2 will be processed when elements along the X-Z plane are required. However, when elements which paralleled to X-Y or Y-Z plane are needed, the targets are distributed into different chunks, and the whole matrix file will be processed. Apparently, when data layout does not match with data access pattern, the data processing becomes very inefficient and causes data overhead.

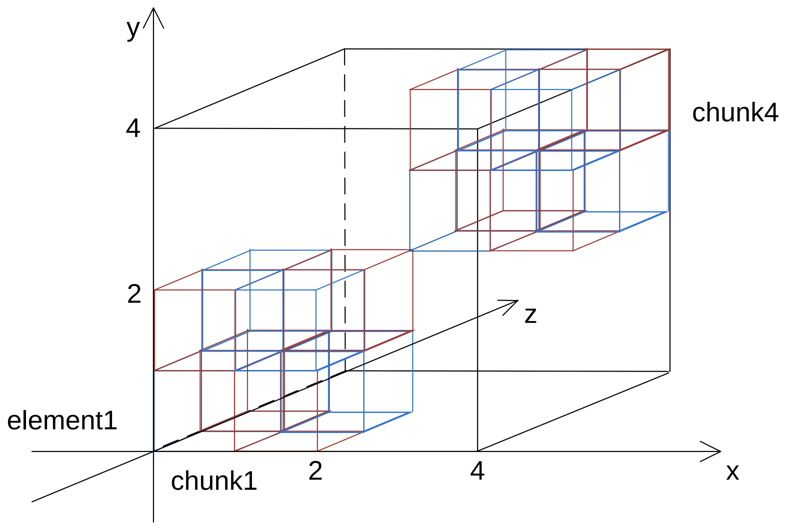

Figure 3-4 Three-Dimension Matrix with Concentric Data Layout

The concentric data layout managed to maintain the three matrix property in chunk level. When applying the concentric data layout into the three dimensional data set, the matrix file can be divided into multiple sub matrixes and each of them is stores into a chunk. In this way, data in each chunk is a sub cube, it not only stores data along the X-Z plans, it also stores data along the X-Y and Y-Z plans. Therefore, when data is accessed with different matrix access pattern, the concentric data layout will generate better performance. Take Figure 3-4 for example, the three-dimension matrix file with 64 elements can be represented as a $4 \times 4 \times 4$ matrix. Since each chunk contains 8 elements, the chunk can be represent in three-dimension way as $2 \times 2 \times 2$, and the whole file can be divided into 8 sub matrixes, data in each sub matrix will stores in one chunk. The elements 1, 2, 5, 6, 17, 18, 21 and 22 which form a small cube will be stored in chunk 1. The elements 3, 4, 7, 8, 19, 20, 23 and 24 which in another cube will be stored in chunk 2. After elements are stored with concentric data layout, if the elements along the X-Z plane are required, the 4 chunks which contains the require elements will be processed. The processing efficiency is $\frac{16}{4\times8} = 50\%$. Compare with continuous data layout, the number of accessed chunks is increased from 2 to 4 and the data efficiency is decreased from 100% to 50%. However, when data is accessed with other matrix access pattern, the concentric data layout outperforms the continuous data layout. When elements along the Y-Z plane are required, 4 chunks will be processed and the processing efficiency is 50%. Compare with continuous data layout, the amount of chunk processed is reduced from 8 to 4, and the efficiency is improved from 25% to

50%. The similar improvement can be seen when the data is accessed with group access pattern. Compared with continuous data layout which needs to process 8 chunks, 5 chunks are processed with concentric data layout. The data efficiency is increased from 43% to 70%.

We compare the average performance between concentric data layout and continuous data layout mathematically. During the comparison, we suppose the matrix access patterns are independent and the possibilities for each matrix access pattern are equal.

Table 3-3 Chunk Amount Comparison between Continuous and Concentric Data Layout with Three-Dimension Matrix File

Access Pattern	X-Y Based	X-Z and Y-Z Based	Group Based
Continuous	$\frac{n^2}{k^3}$	$\frac{n^3}{k^3}$	$\frac{n^2}{k^3}+\frac{n^3}{k^3}-\frac{n}{k^3}$
Concentric	$\frac{n^2}{k^2}$	$\frac{n^2}{k^2}$	$\frac{2n^2}{k^2}-\frac{n}{k}$

We suppose the size of matrix file is $N = n^3$ and the size of the chunk is $K = k^3$, the matrix file will be stored in $\frac{n^3}{k^3}$ chunks in total. For continuous data layout, we suppose the data is stored by the order of first along the X-axis, then along the Y-axis and at last along the Z-axis. The X-Y plane based access will require $\frac{n^2}{k^3}$ chunks in total as the access pattern fits the continuous data layout, Y-Z and X-Z plane based access will both require $\frac{n^3}{k^3}$ chunks, and the group access pattern will require $\frac{n^2}{k^3}+\frac{n^3}{k^3}-\frac{n}{k^3}$ chunks each time. The average number of chunks accessed with the continuous data layout is

$$N_{continuous} = \frac{n^2}{k^3}\times P_{x-y}+\frac{n^3}{k^3}\times P_{x-z}+\frac{n^3}{k^3}\times P_{y-z}+\left(\frac{n^2}{k^3}+\frac{n^3}{k^3}-\frac{n}{k^3}\right)\times P_{group}.$$

For the concentric data layout, the matrix based access pattern needs to process $\frac{n^2}{k^2}$ chunks and the group based access pattern needs to process $\frac{2n^2}{k^2}-\frac{n}{k}$ chunks in total. The average number of chunks accessed with concentric data layout is

$$N_{concentric}=\frac{n^2}{k^2}\times P_{x-y}+\frac{n^2}{k^2}\times P_{x-z}+\frac{n^2}{k^2}\times P_{z-y}+\left(\frac{2n^2}{k^2}-\frac{n}{k}\right)\times P_{group}.$$

The comparison between two data layout is

$$N_{concentric}/N_{continuous}=\left.\left(\frac{5n^2}{k^2}-\frac{n}{k}\right)\middle/\left(\frac{2n^2}{k^3}+3\frac{n^3}{k^3}-\frac{n}{k^3}\right)\right.=\frac{5nk-k^2}{2n+3n^2}.$$

As n is larger than k in Hadoop file system and n^2 grows fastest than k^2, the concentric data layout accessed fewer chunks accessed and reduce the data overhead.

The Table 3-4 shows the pseudo algorithm for three-dimensional concentric data layout.

Table 3-4 Two-Dimensional Concentric Data Layout Algorithm

```
Input: the matrix file size $N$;
        the chunk size $K$;
        the data size $a$;
Output: $C_{id}\ for\ each\ data$;
Steps: Classify element within same sub matrix
    $\frac{N}{K}$, the total number of chunks for matrix file
    $\frac{\sqrt[3]{N}}{\sqrt[3]{K}}$, the number of chunks for each row or column
    $n = \frac{N}{datasize}$, the number of data the matrix file has
    for ($i = 0\ to\ n - 1$) do
        $offset = i \times datasize$
        $z = \left\lfloor \frac{offset}{\left(\sqrt[3]{N}\right)^2} \right\rfloor$, determine the row number for data $i$;
        $y = \frac{offset - z \times \left(\sqrt[3]{N}\right)^2}{\sqrt[3]{N}}$
        $x = offset - z \times \left(\sqrt[3]{N}\right)^2 - y \times \sqrt[3]{N}$
        $C_{id} = \frac{x + y \times \sqrt[3]{N} + z \times \left(\sqrt[3]{N}\right)^2}{K}$
```

The concentric data layout algorithm can be extended to N-dimensional data set. When storing the data set with the concentric data layout, dividing the N-dimensional matrix file into multiple sub sets, each sub set is also an N-dimension set with the size of chunk size. Store the data in each sub set into same chunk, this makes sure the dimensional property is maintained in chunk level. In following, we compare the performance difference between the N-dimension concentric data layout and the continuous data layout.

Table 3-5 Chunk Amount Comparison between Continuous and Concentric Data Layout with Three-Dimension Matrix File

Access Pattern	X-Y Based	X-Z and Y-Z Based	Group Based
Continuous	$\frac{n^{a-1}}{k^a}$	$\frac{n^a}{k^a}$	$\frac{n^{a-1}}{k^a}+\frac{n^a}{k^a}-\frac{n^{a-2}}{k^a}$
Concentric	$\frac{n^{a-1}}{k^{a-1}}$	$\frac{n^{a-1}}{k^{a-1}}$	$\frac{2n^{a-1}}{k^{a-1}}-\frac{n^{a-2}}{k^{a-2}}$

In the comparison, we suppose the matrix file with the size of $N = n^a$ and the size of the chunk is $K = k^a$, so the N-dimension matrix file will be stored in $\frac{n^a}{k^a}$ chunks. For the continuous data layout, the a-1 matrix access pattern which adapt to the continuous data layout will require $\frac{n^{a-1}}{k^a}$ chunks in total, the rest a-1 matrix access will both require $\frac{n^a}{k^a}$ chunks, and the group access pattern will require $\frac{n^{a-1}}{k^a}+\frac{n^a}{k^a}-\frac{n^{a-2}}{k^a}$ chunks in average. The average number of chunks accessed for continuous data layout is

$$N_{continuous} = \frac{n^{a-1}}{k^a} + (a-1)\frac{n^a}{k^a} + \left(\frac{n^{a-1}}{k^a}+\frac{n^a}{k^a}-\frac{n^{a-2}}{k^a}\right).$$

For concentric data layout, the matrix access pattern needs to process $\frac{n^{a-1}}{k^{a-1}}$ chunks of data and the group based access pattern needs to process $\frac{2n^{a-1}}{k^{a-1}} - \frac{n^{a-2}}{k^{a-2}}$ chunks of data in total. The average number of chunks accessed with concentric data layout is

$$N_{concentric} = a\frac{n^{a-1}}{k^{a-1}} + \left(\frac{2n^{a-1}}{k^{a-1}} - \frac{n^{a-2}}{k^{a-2}}\right).$$

The comparison between two data layout is

$$N_{concentric}/N_{continuous} = \left.\frac{(2+a)n^{a-1}}{k^{a}} - \frac{n^{a-2}}{k^{a-2}}\right/ \frac{2n^{a-1}}{k^{a}} + \frac{an^{a}}{k^{a}} - \frac{n^{a-2}}{k^{a}} = \frac{(2+a)nk-k^{2}}{2n+an^{2}}.$$

As *n* is larger than *k* in Hadoop file system, the concentric data layout results less chunk access and relieves the data overhead.

The pseudo algorithm of concentric data layout for N dimension data set is displayed in Table 3-6,

Table 3-6 Concentric Data Layout Algorithm for N-Dimensional Matrix file

```
Input: the matrix file size $N = n^a$;
        the data size $a$;
        the chunk size $K = k^n$;
```

```
Output: $C_{id}\ for\ each\ data$;
```

```
Steps: $t = \frac{n^a}{k^a}$, the total number of chunks for matrix file
        $m = \frac{n}{k}$, the number of chunks for each row or column
        $e = \frac{N}{s}$, the number of data the matrix file has
        for ($i = 1\ to\ e$) do
            $offset\ = i \times datasize$
            for each dimension $j$
                $i_j = \frac{offset - \sum_{j+1}^{a} m \times (\sqrt[a]{N})^m}{K}$
                $C_{id} = \frac{\sum_{1}^{a} i_j \times (\sqrt[a]{N})^j}{K}$
```

CHAPTER 4 EXPERIMENTAL METHODOLOGY AND EVALUATION

In this section we evaluate the performance of the concentric data layout against the continuous data layout. Because most of the HPC analytics applications with group access patterns still need to be developed, there are no established benchmarks available to test our design. We carry out a prototype implementation with matrix data layout on Hadoop File System based on the previously discussed data layout algorithm. We analyze the experiment result in following sections and demonstrate the concentric data layout reduces the amount of data accessed, relieves the data overhead, solves the small I/O problem and improves the processing efficiency.

4.1 Experimental Setup

In our experiment, we access to a 14 node cluster with Hadoop 0.20 installed on it. In our setup, the cluster's master node is used as the namenode and jobtracker, while the 13 slave nodes are configured to be the datanodes and tasktrackers. In the experiment, we are mainly concerned about the number of data retrieved and number of map task processed.

During experiment, we write a MapReduce program to process the data set with matrix data access patterns by two different data layouts, the original continuous data layout and the optimized concentric data layout. In the map phase each process reads contiguous chunks and marks all the required data. In the reduce phase, all the data required by a single process are combined together. We analyze the performance in aspects of executing time, amount of accessing data, data access efficiency and number of map tasks.

4.2 Experimental Analysis

We perform a series of tests on the Hadoop cluster to compare the performance on different layout strategies. We first compare the performance with two-dimensional matrix data set. We write the MapReduce program to process two dimensional files with the size of 1GB, 4GB, and 16GB by using different data layout respectively. These files are originally stored in the HDFS with continuous data layout, and then they are processed by concentric data layout and stored in the HDFS with concentric data layout. In our experiment, the default chunk size is 64MB.

First, the experiments are conducted to indicate the improvement on the execution time of the applications using MapReduce program to access data between concentric data layout and continuous data layout. In the experiments, we have the application to access the data with different matrix access patterns. Figure 4-1 shows the performance of the execution time when accessing data in row based, column based and group access pattern by using the concentric data layout and the continuous data layout.

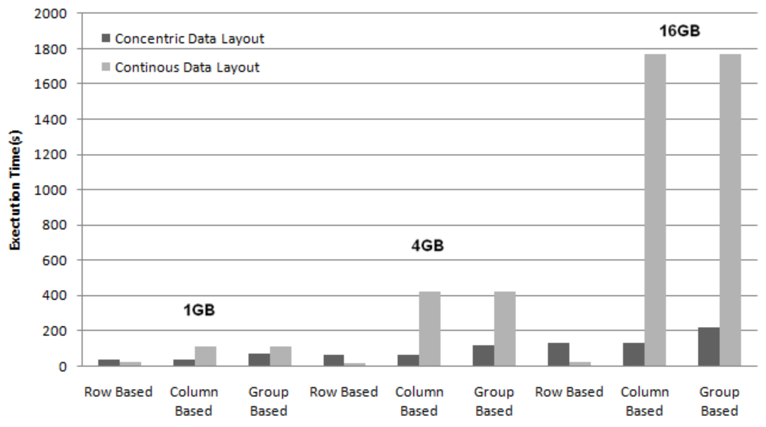

Figure 4-1 Executing Time Comparison for Two Dimension Matrix File

From the Figure 4-1, we can see that the concentric data layout outperforms the continuous data layout when data access pattern is column based or group based, but the continuous data layout works better when data access pattern is row based. The experiment result matches our theoretical analysis. According to the continuous data layout, data is stored row by row. When the accessing pattern is column based, the target data is stored among all chunks. Therefore, the entire matrix file with continuous data layout has to be processed when access pattern is column based or group based. Take 16GB file for example, when data is stored by the continuous data layout, accessing a column of data will required to process the whole data set. The processing time is about 1772s. The group based data access also required the same amount of processing time because in the group base access pattern, both row and column of data is required. The processing time is reduced when data is stored with concentric data layout, because after data is stored with concentric data layout, related data which in the same row or column is stored in the same or near chunks. Therefore it reduces the number of chunks which needed to be processed and reduces the processing time. Take 16GB file for example, the processing time for group access pattern is 217s while the processing time for column access pattern is 134s. The improvement can be observed in other files with size of 1GB, 4GB as well. From Figure 4-1, we can see when data access pattern is row based, the processing time for the continuous data layout is less than the concentric data layout because when data is stored with continuous data layout, the data in the same row will be stored in the same chunk. When a row of data is required, just one chunk is processed. However, when data is stored with concentric data layout, the data in the same row will be stored into several chunks. When a row of data is required, several chunks are required to be processed. Therefore, when data access pattern is row based, the processing time for continuous data layout is better than that of concentric data layout. However, considering the possibilities for each access pattern are independent and equal, we can get the conclusion that the execution time with concentric data layout is better than that with continuous data layout. This is consistent with our model and analysis in chapter 3. In theoretical analysis, we draw the conclusion that the data processing ratio between concentric data layout and continuous data layout is $\frac{2k}{1+n}$, as the processing time is proportional to the amount of accessed data. Take 4GB file for example, the average processing time when data is stored with concentric data layout is

83s while the average processing time when data is stored with the continuous data layout is 288s. The processing ratio between the concentric data layout and the continuous data layout is $\frac{83}{288} \approx \frac{1}{4}$, which match our analysis. Based on the experiment and the above analysis, we can see that data with the concentric data layout fits better with matrix access pattern than data with the continuous data layout. It reduces the execution time and improves I/O system performance.

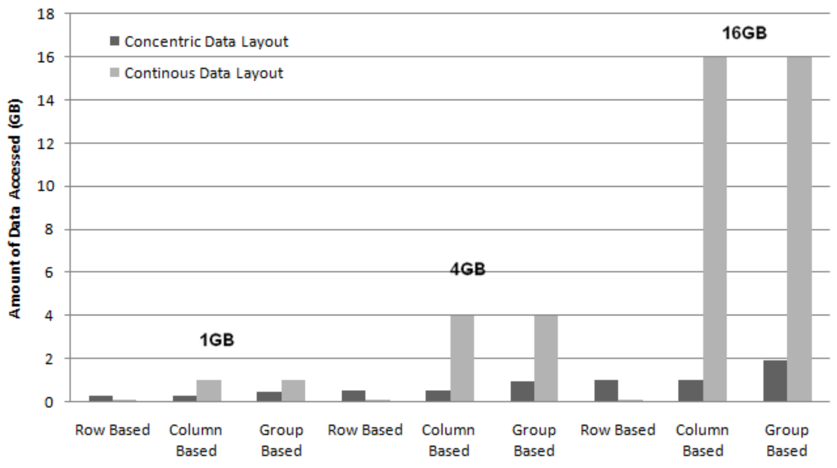

Figure 4-2 Amount of Data Accessed for Two Dimension Matrix File

Second, the experiment compares the amount of data accessed when data is stored with the concentric data layout and the continuous data layout. From Figure 4-2, it clear to see that during the processing, less data is accessed when data is stored with concentric data layout. Take 1GB file for example, when data is accessed with group based access pattern, 448MB of data is retrieved when the data set is stored in the concentric data layout while 1GB of data is retrieved when the data set is stored in the continuous data layout. When the data access pattern is column based, the data set retrieved with the concentric data layout is 256MB while the data set retrieved with the continuous data layout is still 1GB. As the file gets larger, the difference becomes more obvious. In 16GB file, when data access pattern is group based, 1.93GB of data is accessed with concentric data layout while 16GB of data is accessed with continuous data layout. In theoretical analysis, we draw the conclusion that the data processing ratio between concentric data layout and continuous data layout is $\frac{2k}{1+n}$, the experiment validates our conclusion. Take 4GB file for example, the average amount of data accessed when data is stored with concentric data layout is 0.6458GB while the average amount of data accessed when data is stored with continuous data layout is 2.6875GB. The data processing ratio between concentric data layout and continuous data layout is $\frac{0.6458}{2.6875} = \frac{1}{4}$, which match our analysis. The improvement is caused by the fact that in order to access all the required data, the client needs to access the chunks which contains the target data. Compare with continuous data layout, concentric data layout reconstructs the data and keep the matrix property in chunk level. Therefore, compared with the continuous data layout, the concentric data layout makes client accesses fewer chunks and reduces the data overhead.

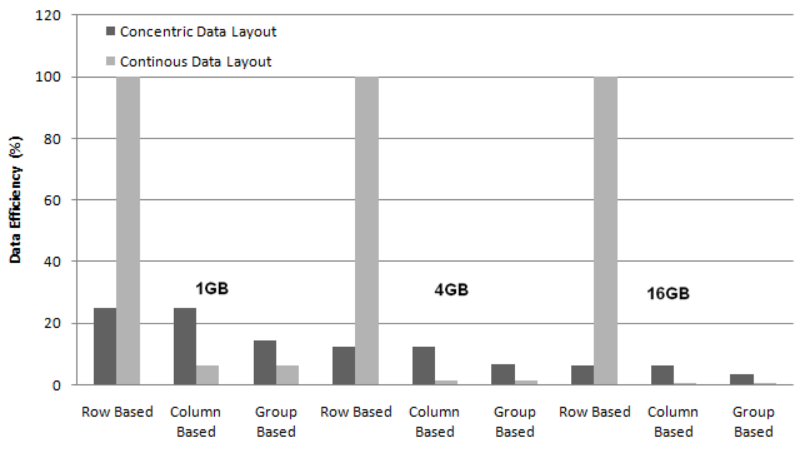

Figure 4-3 Data Efficiency Comparison for Two Dimension Matrix File

Meanwhile, we compare the data efficiency, which indicates how many data is the target data among all the data we processed. We suppose the amount of data we required is 64MB. From Figure 4-3 we can see that compares with continuous data layout, the concentric data layout improves the data efficiency. Take 1GB file for example, the data efficiency for both column based access pattern and group based access pattern is 6.25% when data layout is continuous, while the data efficiency for the column based access pattern is improved to 25% and the data efficiency for the group based access pattern is improved to about 14% with concentric data layout, The same trend can be seen in 4GB file and 16GB file, and the data efficiency improvement becomes more evidence as the file become larger. The experiment results are consistent with our theoretical analysis. According to our analysis, when data is stored with continuous data layout, data is stored in chunks sequentially. The target data is stored in different chunks and each chunk only contains a small part of target data. When data is stored with concentric data layout, the two dimension property is maintained in chunk level. Data in the same row or column is stored in same or close chunks. Therefore, the client is able to retrieve fewer chunks to get the target data. Since the client requires the same amount of data, the fewer chunks retrieved the higher efficiency the data layout provides, so the data efficiency in the concentric data layout is better than that in the continuous data layout.

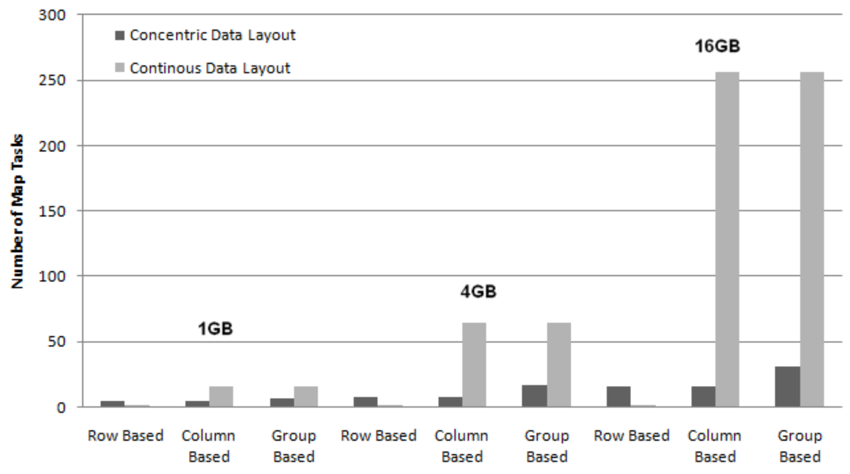

Figure 4-4 Number of Map Tasks for Two Dimension Matrix File

At last, we compare the amount of map tasks during the processing when data is stored with the concentric data layout and the continuous data layout respectively. From Figure 4-4, it is clear to see that the concentric data layout has reduced the number of map task dramatically. For example with group based access pattern, accessing data in concentric data layout with 16GB file requires 31 map tasks, while accessing data in continuous data layout with 16GB file requires 256 map tasks. The same improvement can be seen when the data access pattern is column based. For column based access pattern, 16 map tasks are required when processing the data with the concentric data layout while 256 map tasks are required when processing same amount of data with the continuous data layout. The improvement is caused by the fact that concentric data layout keep the dimension property in chunk level. When matrix data access patterns are required, fewer chunks are accessed to get all target data. In our experiment each map task processes one split which has the size of 64MB. Therefore, the fewer chunks the application retrieved, the less map tasks it generated. Compare with continuous data layout, concentric data layout reduce the task amount during the processing, and relieves the task scheduling problem.

We also conduct the experiment for concentric data layout with three-dimensional matrix file. We write the MapReduce program to process three-dimensional files with the size of 512 MB and 4GB by using different data layout respectively. We analyze several performance aspects like the processing time, amount of accessing data, data efficiency and so on.

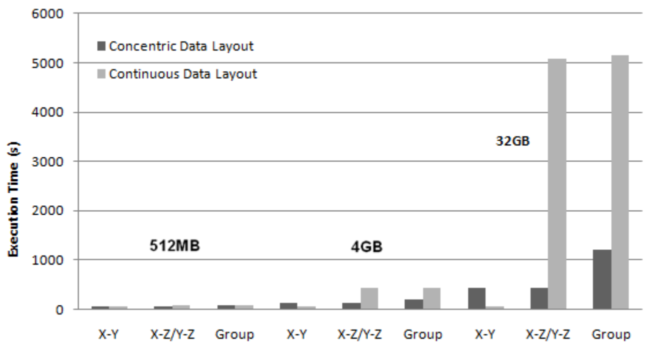

Figure 4-5 Execution Time for Three-Dimensional Matrix File

From the Figure 4-5, we can see that concentric data layout outperforms the continuous data layout when data access pattern is X-Z/Y-Z plane based or group based, but the continuous data layout works better when data access pattern is X-Y plan based. The experiment result matches our theoretical analysis. According to continuous data layout, the data is stored first along the X axis, then along the Y axis and Z axis. When accessing pattern is group based, the target data is stored among all chunks. Therefore, the entire matrix file with continuous data layout has to be processed when access pattern is column based or group based. Take 4GB file for example, when data is stored by continuous data layout, accessing data which parallel to Y-Z plane will required to process the whole data set. The processing time is about 429s. The group based data access also required the same amount of processing time because in group base access pattern, both row and column of data is required. The processing time is reduced when data is stored with concentric data layout. This is because after data is stored with concentric data layout, related data which in the same row or column is stored in the same or near chunks. Therefore it reduces the number of chunks which needed to be processed and hence reduces the processing time. In 4GB file, the processing time for group access pattern is 197s while the processing time for matrix access pattern is 127s. This improvement can be observed in other files with 512MB, 32GB as well. From Figure 4-5 we can see when data access pattern is parallel to X-Y plane, the processing time for continuous data layout is less than concentric data layout because when data is stored with continuous data layout, the data on the same X-Y plane will be stored in the same chunk. When a row of data is required, just one chunk is processed. However, when data is stored with concentric data layout, the data in the same row will be stored into several chunks. When a row of data is required, several chunks are required to be processed. Therefore, when data access pattern is parallel to X-Y plane, the processing time for continuous data layout is better than that of concentric data layout. However, considering the possibilities for each access pattern are independent and equal, we can get the conclusion that the MapReduce program execution time with concentric data layout is better than that with continuous data layout, the concentric data layout has better performance than continuous data layout on I/O system performance with execution time.

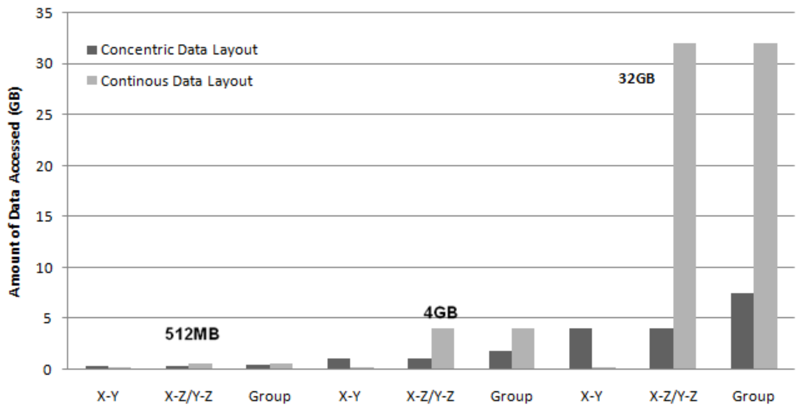

Figure 4-6 Amount of Data Accessed for Three-Dimensional Matrix File

Second, the experiment compares the amount of data accessed when data is stored with concentric data layout and continuous data layout. From Figure 4-6, it clear to see that during the processing, less data is accessed when data is stored with concentric data layout. Take 512MB file for example, when the data access pattern is group based, 358MB of data is retrieved when the data set is stored in concentric data layout while 512MB of data is retrieved when the data set is stored in continuous data layout. When the data access pattern is X-Z/Y-Z based, the data set retrieved with concentric data layout is 256MB while the data set retrieved with continuous data layout is still 512MB. As the file gets larger, the improvement becomes more evidence. In 4GB file, when data access pattern is group based, 1.75GB of data is accessed with concentric data layout while 4GB of data is accessed with continuous data layout. In theoretical analysis, we draw the conclusion that the data processing ratio between concentric data layout and continuous data layout is $\frac{5nk-k^2}{2n+3n^2}$. Take 4GB file for example, the average amount of data accessed when data is stored with concentric data layout is 1.26GB while the average amount of data accessed when data is stored with continuous data layout is 3GBs. The processing ratio between concentric data layout and continuous data layout is $\frac{1.26}{3} \approx \frac{2}{5}$, which matches our analysis. The improvement is caused by the fact that in order to access all required data, the client needs to access all the chunks which contains the target data. Compare with the continuous data layout, the concentric data layout reconstructs the data and keep the matrix property in chunk level. Therefore, during the process, it accesses fewer chunks and reduces the data overhead.

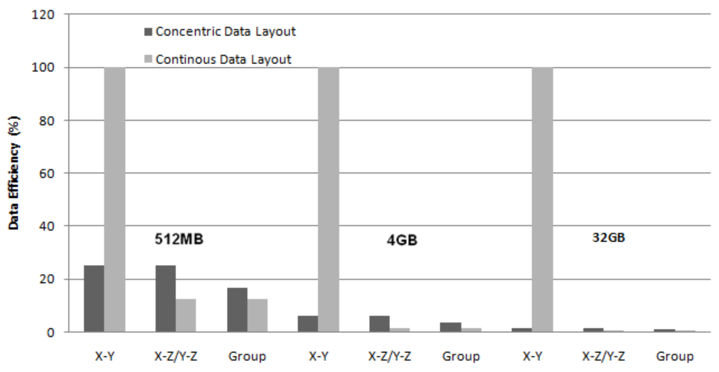

Figure 4-7 Data Efficiency Comparison for Three Dimension Matrix File

The Figure 4-7 compares the data efficiency between the concentric data layout and continuous data layout. We suppose the amount of data we required is 64MB. From Figure 4-7 we can see that compares with continuous data layout, the concentric data layout improves the data efficiency. Take 512MB file for example, the data efficiency for X-Z/Y-Z plan access pattern and group based access pattern is 12.5%. when data layout is continuous, the data efficiency for X-Z/Y-Z plan based access pattern is improved to 25% while data efficiency for group based access pattern is improved to about 16.7% with the concentric data layout, The same trends can be seen with 4GB file, and the data efficiency improvement becomes more evidence as the file become larger. The experiment results are consistent with our theoretical analysis. When data is stored with the continuous data layout, data is stored in chunks sequentially. The target data is stored in different chunks and each chunk only contains a small part of target data. When data is stored with concentric data layout, the three-dimensional property is maintained in chunk level. Data in the same row or column is stored in same or close chunks. Therefore, the client is able to retrieve fewer chunks to get the target data. Since the client requires the same amount of data, the fewer chunks retrieved the higher efficiency the data layout provides, so the data efficiency in the concentric data layout is better than that in the continuous data layout.

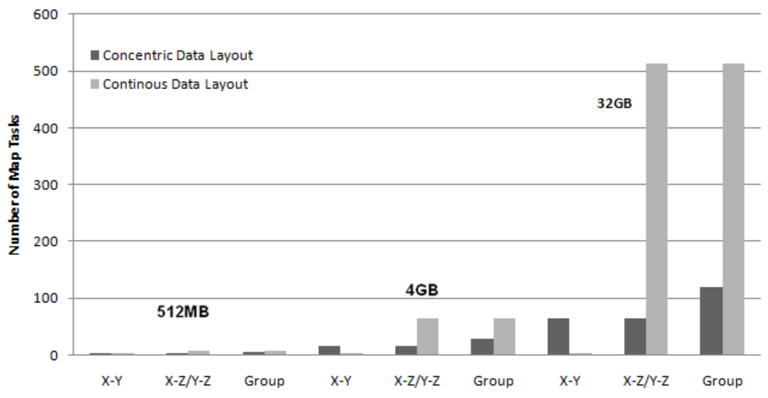

Figure 4-8 Number of Map Tasks for Three Dimension Matrix File

At last, we compare the amount of map tasks during the processing when data is stored with the concentric data layout and the continuous data layout respectively. From Figure 4-8, it is clear to see that concentric data layout has reduced the number of map task dramatically. For example with group based access pattern, accessing data which stored with concentric data layout with 512MB file requires 6 map tasks, while accessing data which stored with continuous data layout requires 8 map tasks. The same improvement can be seen when the data access pattern is X-Z/Y-Z plan based. For these data access pattern, 4 map tasks are required when processing the data with concentric data layout while 8 map tasks are required when processing same amount of data with the continuous data layout. The improvement is caused by the fact that the concentric data layout keeps the dimensional property in chunk level and fits better than continuous data layout. When matrix data access patterns are required, fewer chunks are accessed to get all target data. In our experiment each map task processes one split which has the size of 64MB. Therefore, the fewer chunks the application retrieved, the less map tasks it required. Compare with continuous data layout, concentric data layout reduce the task amount during the processing, and relieves the task scheduling problem.

CHAPTER 5 RELATED WORK

Many approaches have been adopted to relieve the small I/O problem in HPC application, especially for applications using MPI/MPI_IO. Data sieving[2] is an optimization technique to deal with small I/O problem. According to data sieving algorithm, instead of accessing each contiguous portion of data separately, a single contiguous chunk of data which start from the first requested byte up to the last requested byte is read into a temporary buffer in memory. The advantage of this algorithm is that data is always accessed in large chunks. However, the limitation of this simple algorithm is obvious. The data sieving requires the temporary buffer into which data is first read must be as large as the total number of chunk, and generates excessive amount of unnecessary data. Collective I/O[2] also allows client to read a contiguous chunk of data but it redistributes the data among multiple processes as required by them. Besides, applying collective I/O with two-phase implementation in large scale system will result in communication overhead among processes. PLFS[3] is another approaches to solve small I/O problem. PLFS is a file system which mounted on the top of an existing parallel file system and re-maps an applications' write access pattern to be optimized for the under-laying file system. DFS[4] provides striping mechanisms that divides a file into small pieces and distributed them across multiple storage devices for parallel data access. Our work is different from the above mentioned approaches. In our work, we reconstruct the data layout and processes do not need to communicate with others due to the data reorganization. Our work successfully maintains the shared-noting architecture for scalability. DPFS[5] also proposed a multi-dimension data layout to process matrix data set. But the scale of file considered is different. The sizes of files which DPFS is focusing on are relative smaller, from megabytes to gigabytes. Concentric data layout is focusing a large data file which the size is from terabytes to petabytes. Besides, the layout is implemented on parallel file system and the strips which contain the target data are stored in same sub file. This method is not flexible because it only fits well for one data access pattern. When other applications access the data set with different access patterns, the strips which store the target data are distributed to different sub files. In order to read related splits, client needs to go through all the sub files to get the related splits. Our work is more flexible, when data is required, the client only needs to access the chunks which contain the target data. Besides, in DPFS, it just considers the row and column based data access pattern. In our work, we consider the complex matrix access pattern and the situation which the same data set is processed by different applications. Compare with DPFS, the concentric data layout is more flexible and fits well with complex matrix data access patterns.

CHAPTER 6 CONCLUSION

In this paper we analyzed the matrix access patterns and the problems caused by matrix access patterns. We presented the concentric data layout to support data analytics applications processing matrix data set. Concentric data layout is an optimization strategy which works well with various matrix access patterns. It maintains the dimensional property in chunk level. In concentric data layout, instead of storing the data into chunks continuously, data located within the same sub-matrix is stored into the same chunk, when data is required by different access patterns, fewer chunk will be accessed. The concentric data layout is able to significantly boost the I/O performance for data analytics programs by matching with their mixed row-based and column-based access patterns. Our experiments on two-dimensional matrix file and three-dimensional matrix file shows that when data is stored in concentric data layout, the client will accesses fewer chunks, it reduces the amount of process data and improves the processing efficiency, and thereby significantly improves the I/O performance.

ACKNOWLEDGEMENTS

We would like to thank the US NSF for sponsoring this work under grants CCF-0811413 and CAREER CCF-0953946. Additionally, we thank the US Department of Energy for sponsoring us under Early Career Principal Investigator Award: ED-FG02-07ER25747.

REFERENCES

[1]. http://hadoop.apache.org/common/docs/current/hdfs_design.html.

[2]. Rajeev Thakur, William Gropp, Ewing Lusk, Data Sieving and Collective I/O in OMIO," frontiers, pp.182, The 7th Symposium on the Frontiers of Massively Parallel omputation, 1999.

[3]. John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A checkpoint filesystem for parallel applications. In Supercomputing, 2009 ACM/IEEE Conference, Nov. 2009.

[4]. JH Howard, ML Kazar, SG Menees. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, Volume 6, Issue 1, 1988.

[5]. Xiaohui Shen, Alok N, Choudhary. Dpfs: A distributed parallel file system. In ICPP 02: Proceedings of the 2001 International Conference on Parallel Processing, pages 533-544, Washington, DC, USA, 2001.

[6]. D. Borthakur. The Hadoop Distributed File System: Architecture and Design. Apache Software Foundation, 2007.

[7]. Bryant, R. E. Data-Intensive Supercomputing: The Case for DISC. Tech. Rep. CMU-CS-07-128, Carnegie Mellon University, May 2007.

[8]. V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. R. O'Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terasacale computers. In Proceedings of SC2003, November 2003.

[9]. G. Mackey, S. Sehrish, J. Lopez, J. Bent, S. Habib, and J. Wang, Introducing mapreduce to high end computing, in Petascale Data Storage Workshop, 2008. PDSW'08. 3rd, pp.1-6, Nov.2008.

[10]. J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implemention, San Francisco, CA, Dec. 2004.

[11]. Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox, MapReduce for Data Intensive Scientific Analyses, Proceedings of the IEEE International Conference on e-Science, Indianapolis, 2008. December, 2008.

[12]. Bryant, R. E, Data-Intensive Supercomputing: The Case for DISC. Tech. Rep. CMU-CS-07-128, Carnegie Mellon University, May 2007.

[13]. J. Worringen, J.L. Traeff, and H. Ritzdorf, Fast Parallel Non-Contiguous File Access. In Supercomputing 2003: The International Conference for High Performance Computing and Communication, Nov. 2003

[14]. G. Wang, A.R.Butt, P.Pandey, and K.Gupta, A Simulation Approach to Evaluating Design Decisions in MapReduce Setup, in International Symposium

on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, London, UK, Sep.2009.

[15]. Han, H. Rivera, G. and Tseng, Compiler and Run-Time Support for Improving Locality in Scientific Code. Proceedings of Languages and Compilers for Parallel Computing, Twelfth International Workshop, Srpinger-Verlag, 1999.