XHAMI - Extended HDFS and MapReduce Interface for Big Data Image Processing Applications in Cloud Computing Environments

Raghavendra Kune[1], PramodKumar Konugurthi[1], Arun Agarwal[2], Raghavendra RaoChillarige[2], Rajkumar Buyya[3]

[1]Advanced Data Processing Research Institute,
Department of Space, India

[2]School of Computer and Information Sciences,
University of Hyderabad, India

[3]Cloud Computing and Distributed Systems (CLOUDS) Lab,
Department of Computing and Information Systems,
The University of Melbourne, Australia

Abstract–Hadoop Distributed File System (HDFS) and MapReduce model have become popular technologies for large scale data organization and analysis. Existing model of data organization and processing in Hadoop using HDFS and MapReduce are ideally tailored for search and data parallel applications, for which there is no need of data dependency with its neighbouring/adjacent data. However, many scientific applications such as image mining, data mining, knowledge data mining, and satellite image processing are dependent on adjacent data for processing and analysis. In this paper, we identify the requirements of the overlapped data organization and propose a two phase extensions to HDFS and MapReduce programming model, called XHAMI, to address them. The extended interfaces are presented as APIs and implemented in the context of Image Processing (IP) application domain. We demonstrated effectiveness of XHAMI through case studies of image processing functions along with the results. Although XHAMI has little overhead in data storage and input/output operations, it greatly enhances the system performance and simplifies the application development process. Our proposed system, XHAMI, works without any changes for the existing MapReduce models, and can be utilised by many applications where there is a requirement of overlapped data.

Keywords: Cloud Computing, Big Data, Hadoop, MapReduce, Extended MapReduce, XHAMI, Image Processing, Scientific computing, Remote Sensing.

1. Introduction

The amount of textual and multimedia data has grown considerably large in recent years due to the growth of social networking, healthcare applications, surveillance systems, earth observation sensors etc. This huge volume of data in the world has created a new field in data processing called as Big Data [1], which refers to an emerging data science paradigm of multi-dimensional information mining for scientific discovery and business analytics over large scale scalable infrastructure [2]. Big Data handles massive amounts of data collected over time, which is otherwise difficult task to analyse and handle using common database management tools [3]. Big Data can yield extremely useful information; however apart, urges new challenges both in data organization and processing [4].

Hadoop [5] is an open source framework for storing, processing, and analysis of large amounts of distributed semi structured/unstructured data [6]. The origin of this framework comes from internet search companies like Yahoo and Google, who needed new processing tools and models for web page indexing and searching. This framework is designed for data parallel processing at Petabyte and Exabyte scales distributed on the commodity computing nodes. Hadoop cluster is a highly scalable architecture, that spawns both compute and data storage nodes horizontally for preserving and processing large scale data to achieve high reliability and high throughput. Therefore, Hadoop framework and its core sub components—HDFS [7][8] and MapReduce[9][10][11]—are gaining popularity in addressing several large scale applications of data intensive computing in several domain specific areas like social networking, business intelligence, and scientific analytics, etc. for analysing large scale, rapidly growing, variety structures of data.

The advantages of HDFS and MapReduce in Hadoop eco system are – horizontal scalability, low cost setup with commodity hardware, ability to process semi-structured/ unstructured data, and simplicity in programming. However, HDFS and MapReduce, though offer tremendous potential for gaining maximum performance, but due to its certain inherent limiting features, unable to support applications with overlapped data processing requirements. Below we describe one such domain specific applications in remote sensing image processing and their requirements.

1.1.Remote Sensing Imaging Applications

Earth observation satellite sensors provide high-resolution satellite imagery having image scene sizes from several megabytes to gigabytes. High resolution satellite imagery, for example, Quick Bird, IKONOS, Worldview, IRS CARTOSAT[12], are used in various applications of information extraction and analysis in domains such as oil/gas mining, engineering construction like 3D urban/terrain mapping, GIS developments, defence and security, environmental monitoring, media and entertainment, agricultural and natural resource exploration. Due to increase in the numbers of satellites and technology advancements in the remote sensing, both the data sizes and their volumes are increasing on a daily basis. Hence, organization and analysis of such data for intrinsic information is a major challenge.

Ma et al. [13] discussed challenges and opportunities in Remote Sensing (RS) Big Data computing, focussed on RS data intensive problems, analysis of RS Big Data, and several techniques for processing RS Big Data. Two dimensional structured representation of images, and majority of the functions in image processing being highly parallelizable, the HDFS way of organizing the data as blocks and usage of MapReduce functions for processing each block as independent map function, makes Hadoop a suitable platform for large scale high volume image processing applications.

1.2.Image Representation

An image is a two-dimensional function f(x,y)as depicted in Figure 1,where x and y are spatial (plane) coordinates, and the amplitude of ‘f’ at any pair of coordinates (x,y) is called intensity or gray level of the image at that point [14]. Image data mining is a technology that aims in finding useful information and knowledge from large scale image data [15] until a pattern in the image becomes obvious. This involves the usage of several image processing techniques such as enhancement, classification, segmentation, object detection etc., which could use in turn several combinations of linear/morphological spatial filters [14] to achieve the result. Image mining is the process of searching and discovering valuable information and knowledge in large volumes of data. Image mining follows basic principles from concepts in databases, machine learning, statistics, pattern recognition and soft computing. Image mining is an emerging research field in geosciences due to the large scale increase of data which lead to new promising applications. For example, the use of very high resolution satellite images in earth observation systems now enables the observation of small objects, while the use of very high temporal resolution images enables monitoring of changes at high frequency for detecting the objects.

However, actual data analysis in geosciences or earth observation techniques suffers from the huge amount of complex data to process. Indeed, earth observation data (acquired from optical, radar, and hyper spectral sensors installed on terrestrial, airborne or space borne platforms) is often heterogeneous, multi-scale, and composed of complex objects. Segmentation algorithms, unsupervised and supervised classification methods, descriptive and predictive spatial models and algorithms for large time series would be applied to assist experts in their knowledge discovery.

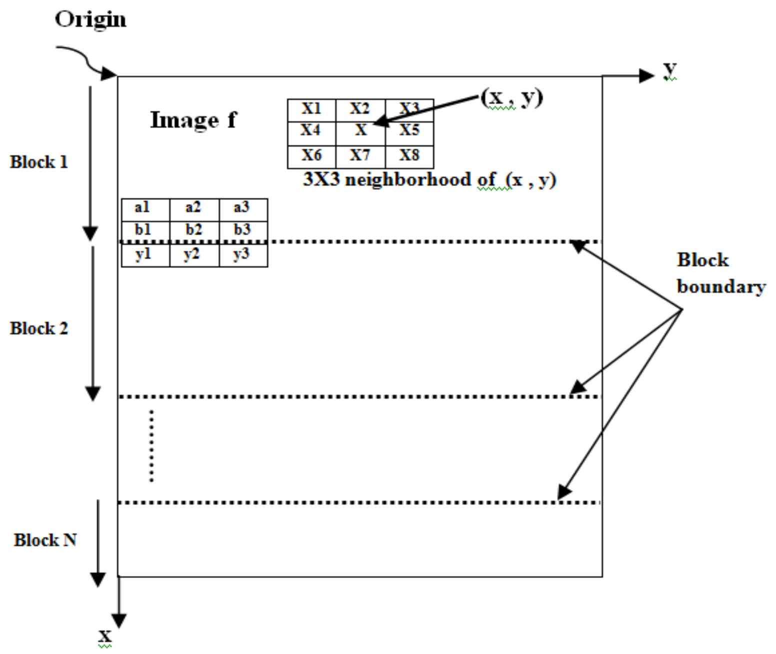


Figure 1. Image representation with segmented blocks

Many of the algorithms applied during image mining techniques such as linear/morphological spatial filters demand use of adjacent pixels for processing the current pixel. For example as depicted in Figure

1, a smoothening operation performs weighted average of a 3X3 kernel window; hence, the output of pixel X depends on the values of X1, X2, X3, X4, X5, X6, X7, and X8. If an image is represented as segmented blocks, processing the pixels those are falling on the block boundaries require the pixels from their next immediate adjacent block, as depicted in Figure 1. Therefore if the image is organized as several physical distributed data blocks, image filtering operations cannot be performed on the edge pixels, unless overlap data required is preserved in the blocks. For example processing the pixels such as b1, b2, b3 of block 1, requires its adjacent block 2 pixels such as y1, y2, y3. It is easy to get the overlapping pixels, for an image organized as a single large file, by moving the file pointer back and forth to read the data of interest. But, if the images are organized as segmented blocks, then the application demands the overlapping data at data nodes for processing, which in turn, requires moving of this overlapped data to the data nodes. As these data blocks are distributed, accounts for large I/O overheads during processing.

Hadoop and several other similar application implementations (see Section 2), split the data based on a fixed size, which results in partitioning of data as shown in Figure 1. Each of the blocks is written to different data nodes. Therefore the boundary pixels of entire line b1, b2, b3... in each block cannot be processed, as the adjacent pixels are not available at the respective data nodes. Similarly for the pixels marked as y1, y2, y3, y4,… cannot be performed straight away. To process these boundary pixels i.e., the start line and end line in each block a customized map function to read additional pixels from a different data node is essential, otherwise the output would be incorrect. These additional read operations for each block increase I/O overhead significantly.

1.3.Our Contributions

To meet the requirements of applications with overlapped data, we propose an Extended HDFS and MapReduce Interface, called XHAMI, which offers ***a two phase extensions to HDFS and MapReduce programming model.*** The extended interfaces are presented as APIs and implemented in the context of Image Processing (IP) application domain. We demonstrated effectiveness of XHAMI through case studies of image processing functions along with the results. Our experimental results reveal that XHAMI greatly enhances the system performance and simplifies the application development process. It works without any changes for the existing MapReduce models, and hence it can be utilised by many applications where there is a requirement of overlapped data.

1.4.Paper Organisation

The rest of the paper is organised as follows. Section 2 describes related work in image processing with HDFS and MapReduce over Hadoop framework. Section 3 describes XHAMI system for large scale image processing applications as a two phase extension over the conventional Hadoop HDFS and MapReduce system. The first extension discusses the data organization over HDFS, and the second illustrates high level packages for image processing. XHAMI hides the low level details of the data organization over HDFS and offers a high level APIs for processing large scale images organized in

HDFS using MapReduce computations. Section 4 describes experimental results of XHAMI compared with conventional Hadoop for data organization, followed by customized Hadoop MapReduce (MR), and other related systems like Hadoop Image Processing Interface (HIPI) [16] for MapReduce computations. Section 5 presents conclusions and future work.

2. Related Work

Image processing and computer vision algorithms can be applied as multiple independent tasks on large scale data sets simultaneously in parallel on a distributed system to achieve higher throughputs. Hadoop [5] is an open source framework for addressing large scale data analytics uses HDFS for data organization and MapReduce as programming models. In addition to Hadoop, there are several other frameworks like Twister [17] for iterative computing of streaming text analytics, and Phoenix [18] used for map and reduce functions for distributed data intensive Message Passing Interface (MPI) kind of applications.

Kennedy et al. [19] demonstrated the use of MapReduce for labelling 19.6 million images using nearest neighbour method. Shi et al. [20] presented use of MapReduce for Content Based Image Retrieval (CBIR), and discussed the results obtained by using around 400,000 images approximately. Yang et al. [21] presented a system MIFAS for fast and efficient access to medical images using Hadoop and Cloud computing. Kocalkulak et al. [22] proposed a Hadoop based system for pattern image processing of intercontinental missiles for finding the bullet patterns. Almeer et al. [23] designed and implemented a system for remote sensing image processing with the help of Hadoop and Cloud computing systems for small scale images. Demir [24] et al. discussed the usage of Hadoop for small size face detection images. All these systems describe the bulk processing of small size images in batch mode over HDFS, where each map function processes the complete image.

White et al. [25] discussed the overheads that can be caused due to small size files, which are considerably smaller than the block size in HDFS. A similar approach is presented by Sweeney et al. [16] and presented Hadoop Image Processing Interface (HIPI) as an extension of MapReduce APIs for image processing applications. HIPI operates on the smaller image files, and bundles the data files (images) into a large single data file called HIPI Image Bundle (HIB), and the indexes of these files are organized in the index file.

Potisepp [26] discussed the processing small/regular images of total 48675 by aggregating them into large data set, and processed them on Hadoop using MapReduce as sequential files, similar to the one addressed by HIPI. Also, presented feasibility study as a proof-of-concept test for a single large image as blocks and overlapping pixels for non-iterative algorithms image processing. However, no design, or solution, or methodology has been suggested to either to Hadoop or MapReduce for either Image Processing applications or for any other domain, so that the methodology works for existing as well as new models which are under consideration.

Papers discussed above, demonstrated the usage of Hadoop for image processing applications, and compared the performance of image processing operations over single PC system running on a conventional file system Vs Hadoop cluster. Few other papers have demonstrated the extension of Hadoop, called HIPI to solve the small files problem, which combine the smaller images into large bundle and process them using HIPI Bundle (HIB). These systems had limitations in addressing the spatial image filters applications as the overlap data is not present among the adjacent blocks for processing. In this paper, we address the issues related to organization and processing of single large volume images, which are in general in data sizes ranging from Megabytes to Gigabytes, and their processing using MapReduce. To address the issues, we discuss the extensions proposed of the Hadoop for HDFS and MapReduce, and present those extensions XHAMI. Though, XHAMI demonstrates remote sensing/geo sciences data processing, but the same can be used for other domains where data dependencies are major requirements, e.g. bio medical imaging.

3. XHAMI- Extended HDFS and MapReduce

In this section we describe XHAMI - the extended software package of Hadoop for large scale image processing/mining applications. First we present XHAMI APIs for reading and writing (I/O), followed by MapReduce for distributed processing. We discuss two sample case studies i.e. histogram and image smoothening operations. Histogram computes the frequency of pixel intensity values in the image, and smoothening operation uses spatial filters like Sobel, Laplacian etc. [14]. Later, we discuss how XHAMI can be used for data organization, designing user specific Image related MapReduce functions, and extending the functionality for other image domain specific applications.

3.1 XHAMI –HDFS I/O extensions for domain specific applications

Figure 2 depicts the sequence of steps in reading/writing the images using XHAMI software library over Hadoop framework. Initially, client uses XHAMI I/O functions (step 1) for reading or writing the data. The client request is translated into create() or open() by XHAMI, and sent to Distributed File System (step 2). Distributed File System instance calls the name node to determine the data block locations (step 3). For each block, the name node returns the addresses of the data nodes for writing or reading the data. Distributed File System returns FSDataInput/Output Stream, which in turn will be used by XHAMI to read/write the data to/from the data nodes. XHAMI checks file format, if the format is in image type (step 4), then metadata information such as file name, total scans, total pixels, total numbers of bands in the image, and the number of bytes per pixel are stored in HBASE [28], this simplifies header information reading as and when required through HBASE queries, otherwise reading the header block by block is tedious and time consuming process.

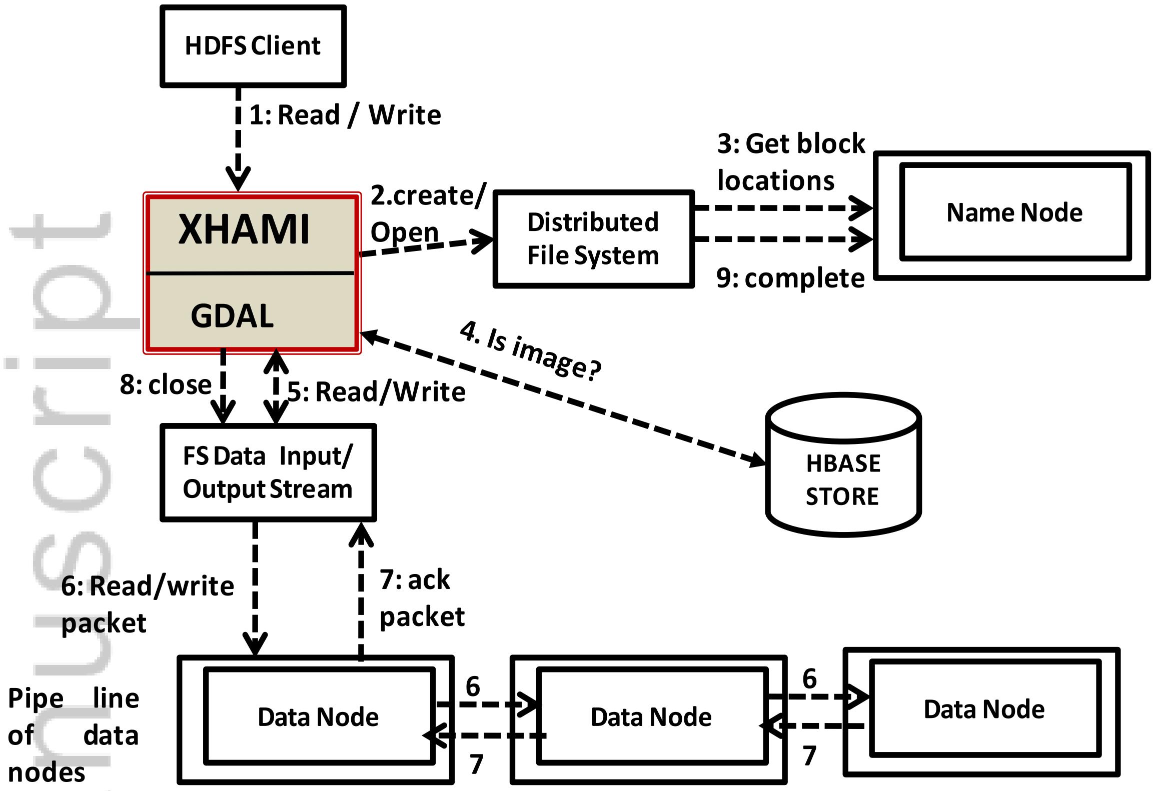


Figure 2.XHAMI for read/write operations

Later on XHAMI calls FSDataInput/Output Stream either to read/write the data to/from the respective data nodes (step 5). Steps 6 and 7 are based on standard HDFS data reading/writing in the pipelining way. Each block is written with the header information corresponding to the blocks i.e. blockid, start scan, end scan, overlap scan lines in the block, scan length, and size of the block. Finally, after the read/write operation the request is made for closing the file (step 8), and the status (step 9) is forwarded to the name node. Below we describe techniques developed for data organization followed by extended APIs for HDFS and MapReduce.

3.2 Data Organization

Single large image data is organized as blocks, with an overlap with its next immediate blocks. The segmented techniques used for data organization are shown in Figure 3. Figure 3.a depicts an image as a one dimensional sequence of bytes, Figure 3.b depicts, block segmentation in horizontal direction, and Figure 3.c shows the organization in both horizontal and vertical directions.

Image blocks are can be constructed in two ways i.e. **(i) unidirectional**: partitioning across the scan line direction as shown in Figure 3.b and **(ii) bidirectional**: partitioning both horizontal and vertical directions as shown in Figure 3.c. while construction, it is essential to ensure that, no split take place within the pixel byte boundaries. The methods are described below.

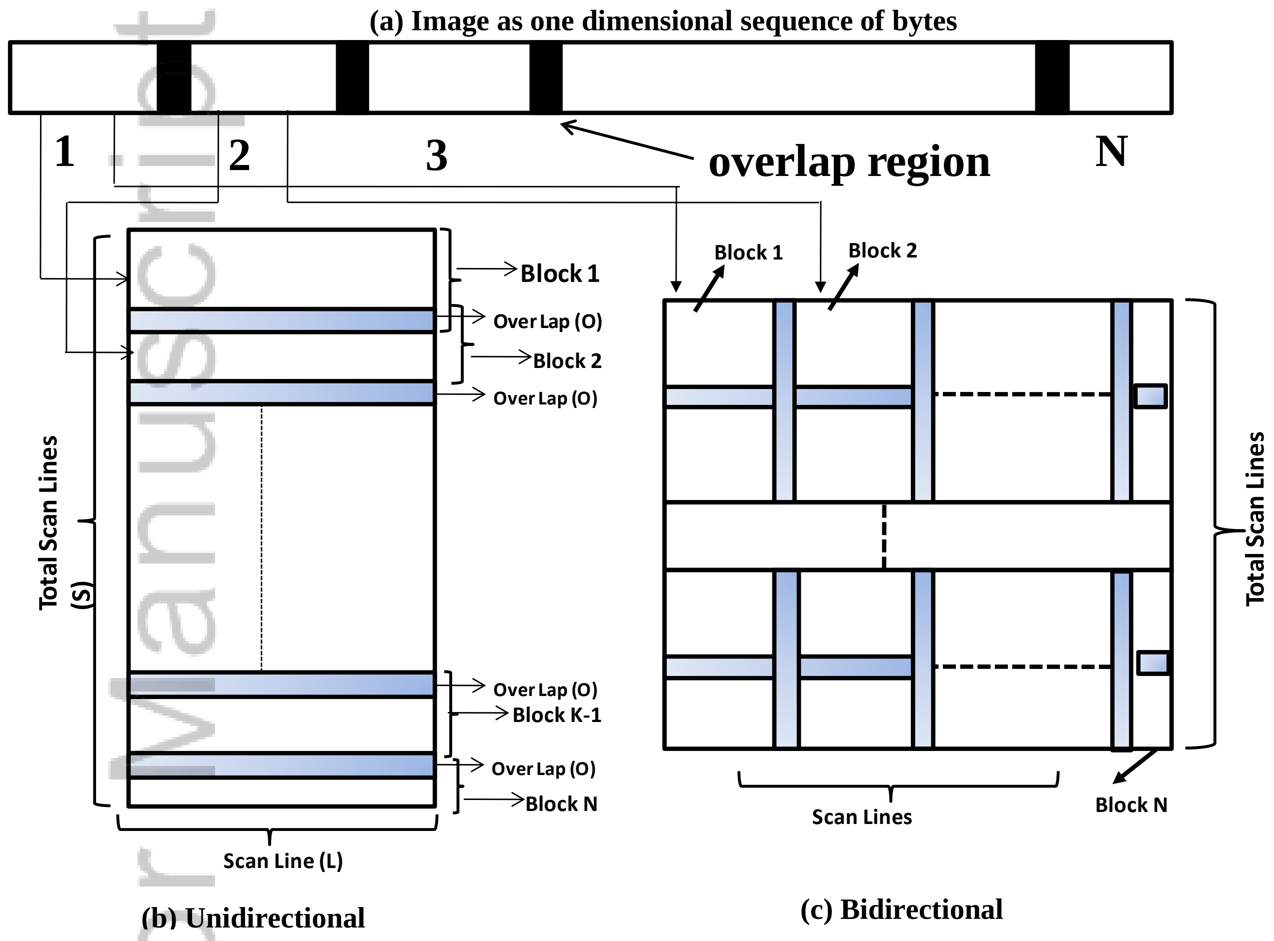


Figure 3. Block construction methods

i) Unidirectional split: blocks are constructed by segmenting the data in across scan line (horizontal) direction. Each block is written with the additional lines at the end of the block.

ii) Bi-directional split: splitting the file into blocks in both horizontal and vertical directions. The split results in the blocks, for which, the first and last blocks have overlap with their adjacent two blocks, and all the remaining blocks have overlap with their adjacent four blocks. This type of segmentation results in large storage overhead which is approximately double the size of the unidirectional segment construction. This type of organization is preferred while images have larger scan line lengths.

In the current version of XHAMI package data organization is addressed for unidirectional segmented blocks, however, it can be extended for bi-directional split. The segmentation procedure is described below.

Scan lines for each block *Sb* computed as

$$Sb = \left\lceil {}^{H}\!/_{(L * P)} \right\rceil$$

H = HDFS Default block length in Mbytes.
L = length of scan line i.e. total pixels in the scan line.
P = pixel length in bytes.
S = total number of scan lines.

Total number of blocks *T*, having overlap of *α* number of scan lines is

$$T = \left\lceil {}^{S}\!/_{Sb} \right\rceil$$

If T α >Sb then T = T+1.*

The start and end scan lines $B_{i,s}$ and $B_{i,e}$in each block is given below; N representing total scans in the image.

$$B_{i,s} = \begin{cases} 1, & i = 1 \\ B_{i-1,e-\alpha+1}, & 1 < i < T \\ B_{N-1,e-\alpha+1} & i = T \end{cases}$$

$$B_{i,e} = \begin{cases} B_{i,s} + S_b - 1 & 1 \le i < T \\ S_b i = T \end{cases}$$

Block length is computed as below.

$$R_i = (B_{i,e} - B_{i,s} + 1) * L * P , \ 1 \le i \le T$$

The blocks are constructed with metadata information in the header, such as blockid, start scan, end scan, overlap scan lines in the block, scan length, block length. Though, metadata adds small additional storage overheads, but, simplifies the processing activity during Map phase, for obtaining the total number of pixels, number of bands, bytes per pixel etc, and also helps to organize the blocks in the order during the combine/merge phase using blockid.

3.3 XHAMI package description

XHAMI offers Software Development Kit (SDK) for Hadoop based large scale domain specific data intensive applications designing. It provides high level packages for data organization and for MapReduce based processing simplifying the development and quick application designing by hiding several low level details of image organization and processing. XHAMI package description is as follows- XhamiFileIOFormat is the base class for all domain specific applications which is placed under the package xhami.io, as shown in Figure 4.

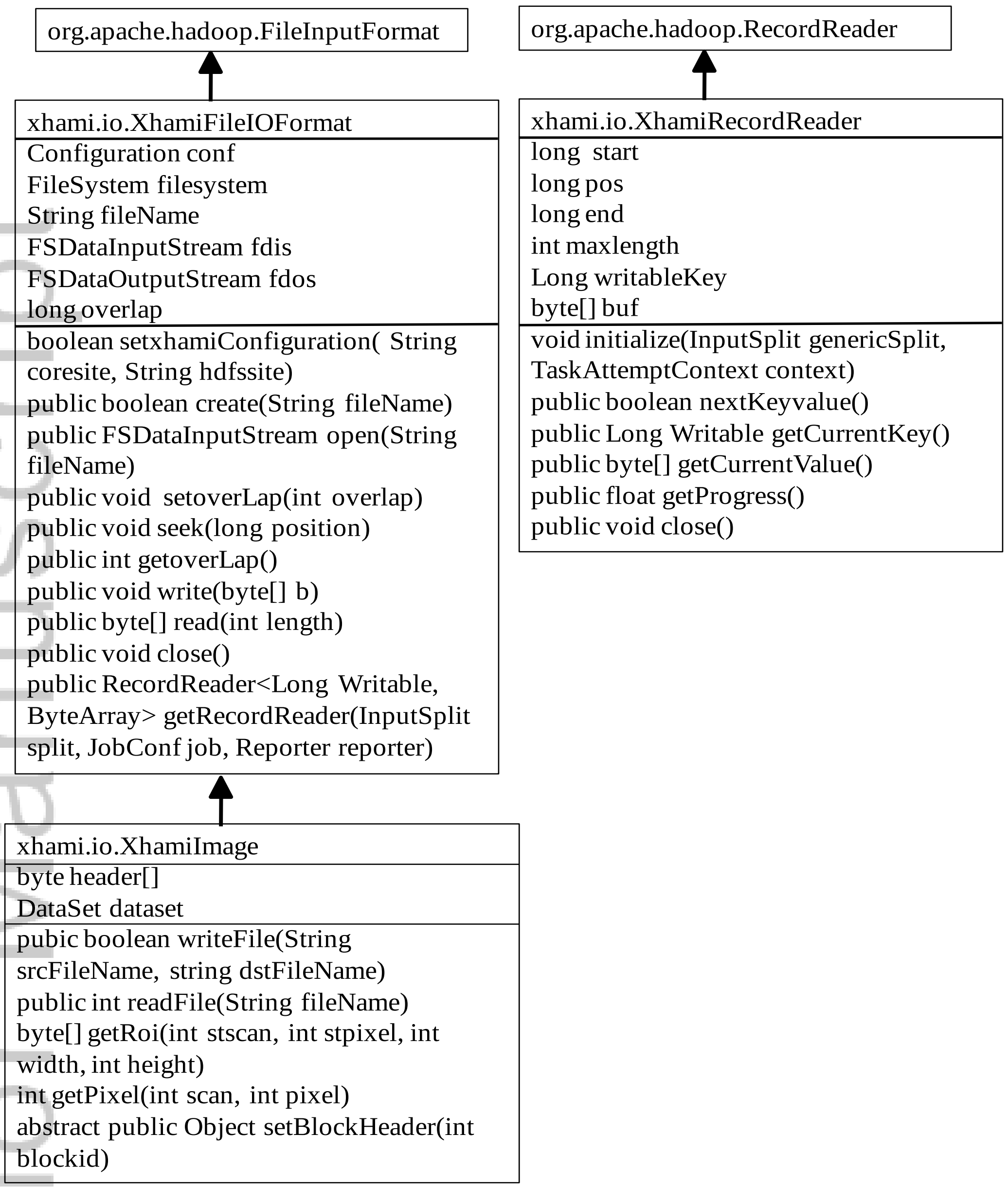

org.apache.hadoop.FileInputFormat
org.apache.hadoop.RecordReader
xhami.io.XhamiFileIOFormat
Configuration conf
FileSystem filesystem
String fileName
FSDataInputStream fdis
FSDataOutputStream fdos
long overlap
boolean setxhamiConfiguration(String coresite, String hdfssite)
public boolean create(String fileName)
public FSDataInputStream open(String fileName)
public void setoverLap(int overlap)
public void seek(long position)
public int getoverLap()
public void write(byte[] b)
public byte[] read(int length)
public void close()
public RecordReader<Long Writable, ByteArray> getRecordReader(InputSplit split, JobConf job, Reporter reporter)
xhami.io.XhamiRecordReader
long start
long pos
long end
int maxlength
Long writableKey
byte[] buf
void initialize(InputSplit genericSplit, TaskAttemptContext context)
public boolean nextKeyvalue()
public Long Writable getCurrentKey()
public byte[] getCurrentValue()
public float getProgress()
public void close()
xhami.io.XhamiImage
byte header[]
DataSet dataset
pubic boolean writeFile(String srcFileName, string dstFileName)
public int readFile(String fileName)
byte[] getRoi(int stscan, int stpixel, int width, int height)
int getPixel(int scan, int pixel)
abstract public Object setBlockHeader(int blockid)

Figure 4. XHAMI I/O package

XhamiFileIOFormat extends FileInputFormat from the standard Hadoop package, and the implementation of several methods, for hiding the low level handling of data for HDFS data organization is handled by this package. The major methods offered under XhamiFileIOFormat class are i) setting the overlap, ii) getting the overlap, iii) reading, writing, and seeking the data without knowing/knowledge of the low level details of the data.

XhamiFileIOFormat is used as base class for several application developers for file I/O functionality and implement further for domain specific operations. Xhami.io.XhamiImage is an abstract class which provides the methods for the implementation of image processing domain specific functionality by extending XhamiFileIOFormat. XhamiImage extends XhamiFileIOFormat class and implements several image processing methods for setting the header/metadata information of the image, block wise metadata information, and for read/write the image blocks into the similar format of that original file, using Geographical Data Abstraction Layer (GDAL) library [27] and offers several methods such as reading the region of interest, getting the scan lines, pixel, reading the pixel grey vale etc., by hiding the low level details of file I/O. Several methods in XhamiFileIOFormat, XhamiImage classes, and XhamiRecordReader classes are described in Table 1, Table 2 and Table 3 respectively. XhamiImage implements Geographic Data Abstraction Layer (GDAL) functionality using the DataSet object for image related operations. XhamiImage class can be extended by the Hadoop based Image processing application developer by setting up their own implementation of setBlockHeadermethod. XhamiRecordReader reads the buffer data and sends to the Map function for processing.

Table 1. Description of Methods in XhamiFileIOFormat class

Method name	Method description	Return value
boolean setxhamiConfiguration(String coresite, String hdfssite)	Sets the HDFS configuration parameters such as coresite and hdfssite. This function in turn uses the Configuration object, and calls addResource methods of its base class, to set establish the connectivity to HDFS site.	If the configuration parameters are correct, then boolean value true is returned. In case wrong supply of arguments, or if the parameter files are not available, or due to invalid credentials, or else HDFS site may be down, false will be returned.
boolean create(String fileName)	Create the file with name filename to write to HDFS. Checks if the file already exists. This function is used before the file is to be written to HDFS.	Returns true if the file is not present in HDFS, or else returns false.
FSDataInputStream open(String filename)	Checks if the file is present in the HDFS.	If the file is present FSDataInputStream having the object value is returned, otherwise FSDataInputStream with value having null is returned.
void setoverLap(int overlap)	Used to set the overlap across the segmented blocks. The supplied overlap value is an integer value corresponding to the overlap size in bytes.	-
void seek(long position)	Moves the file pointer to the location position in the file.	-

int getoverLap()	Reads the overlap value set for the file while writing to HDFS.	Return the overlap value if set for the file or else returns -1.
void write(byte[] b)	Writes b number of bytes to the file.	-
byte[] read(int length)	Reads length number of bytes from the file.	Returns the data in the byte array format.
void close()	Closes the data pointers those are opened for reading / writing the data.	-
RecordReader <Long Writable, ByteArray> getRecordReader(InputSplit split, JobConf job, Reporter reporter	Reads the Xhami compatible record reader in bytes, for MapReduce computing by overriding the RecordReader method of FileInputFormat class. The compatible here means the window size to be read for processing the binary image for processing. This would be supplied as argument value to the Image Processing MapReduce function. If the value is of type fixed, then the entire Block is read during processing.	Returns the Default Hadoop RecordReader Object for processing by the MapReduce job.

Table 2.XhamiImageclass description

Method name	Method description	Return value
boolean writeFile(String srcFileName, String dstFileName)	Used for writing the contents of the file srcFileName, to the destination dstFileName.	Boolean value true is returned if the writing is successful, or else false is returned.
int readFile(String fileName)	Set the file fileName to read from the HDFS. Returns the total numbers of blocks, that the file is organized in HDFS.	Number of blocks that the file is stored in HDFS. If the file does not exist -1 is returned.
byte[] getRoi(int stscan, int stpixel, int width, int height)	Reads the array of bytes from the file already set, starting at stscan and pixel, with a block of size width and height bytes.	If successful returns byte array read, or else returns NULL object.
int getPixel(int scan, int pixel)	Reads the pixel value at the location scan and pixel.	Returns pixel(gray) value as integer.
abstract public Object setBlockHeader(int blockid)	Abstract method which would be overwritten by XHAMI application developers for image processing domain applications.	Header information type casted to Object data type.

Package hierarchy of MapReduce for Image processing domain is shown in Figure 5 and methods in the packages are described in Table 4.

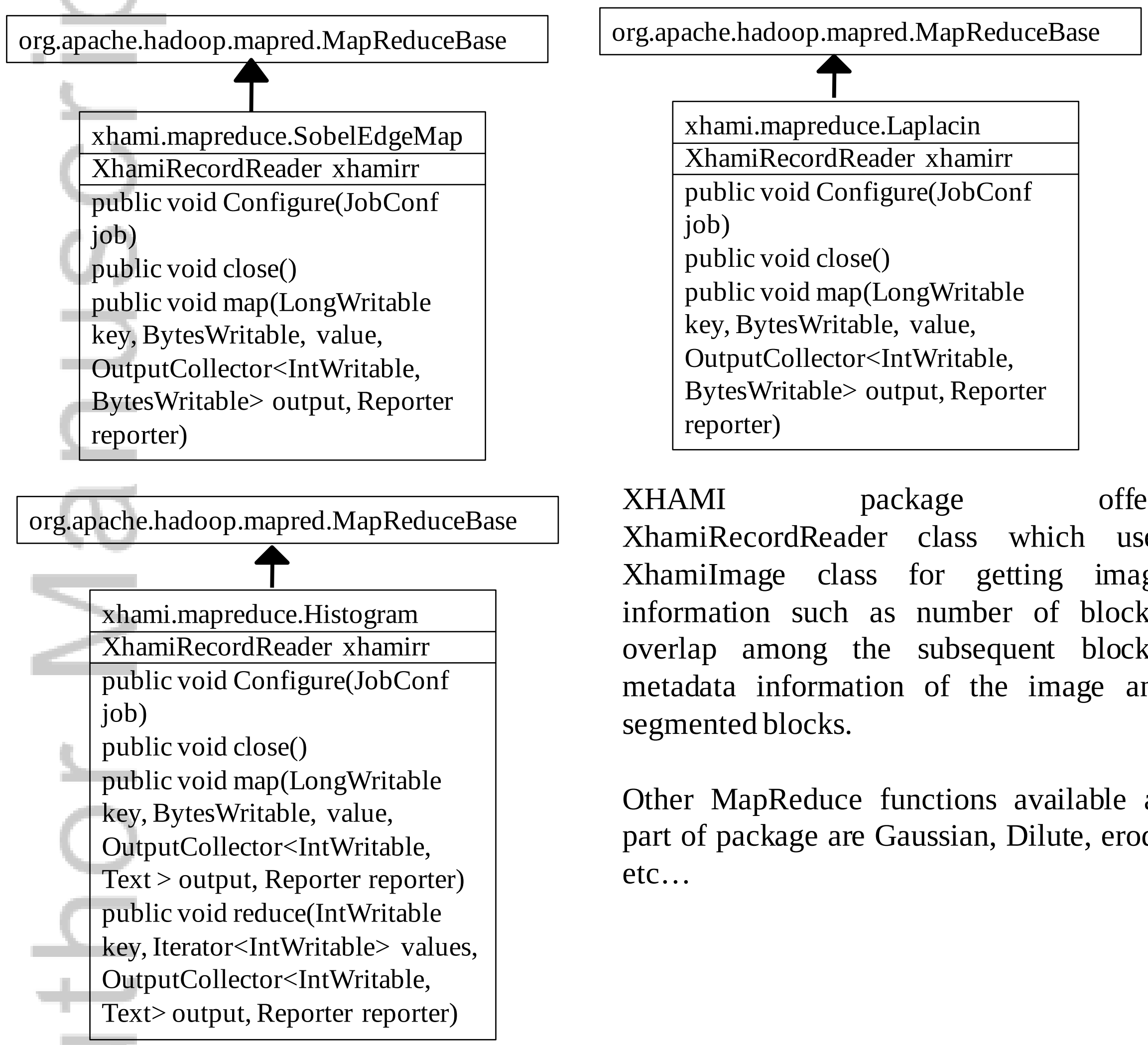
Table 3. Methods in XhamiRecordReader

Method name	Method description	Return value
Initialize(InputSplit genericSplit, TaskAttemptContext context)	Overrides the Initialize method of standard Hadoop Record reader method. Implements the own split method, which reads the content which is compatible with the Xhami File data format, hiding the overlap block size.	-
boolean nextKeyvalue()	Overrides the nextKeyvalue method of its base class RecorReader.	Returns true if it can read the next record for the file, or else return false.
Long Writable getCurrentKey()	getCurrentKey method of its base class is overridden.	Return Writable Object of the record recorder method.
float getProgress()	Overriding method, to send the progress of the data read.	Return float false representing the percentage of data read so far from the XhamiRecordRecorder, corresponding to the InputSplit.
byte[] getCurrentValue()	Reads the bytes array to be sent for computing for Map function.	Return byte array if true, else returns NULL object.
void close()	Closes the record reader object.	-

Table 4. Description of XHAMI MapReduce classes for Image Processing domain

MapReduce Class	Description	Return value
Sobel	Implementation of Sobel spatial edge detection filter. It has map function implementation only, and the Reduce is not required, as the output of the map itself is directly written, as it does not required any collection of the map inputs for processing further. This implementation hides the several details such as overlapping pixels across the blocks, and the kernel window size to be read for processing. Output is written to the HDFS file system.	Output Images with the detected edges.
Laplacian	Implementation of Laplacian differential edge detection filter. It has map function implementation but not reduce method. Reduce is not required, as the output of the map itself is directly written, as it does not required any collection of the map inputs for processing further. This implementation hides the several details such as overlapping pixels across the blocks, and the kernel	Output Images with the detected edges.

	window size to be read for processing. Output is written to the HDFS file system.	
Histogram	Implements both Map and Reduce functions. Map collects the count of the pixel (gray) value, and reduce does the aggregation of the collected numbers from the map functions. While processing it does not consider the overlapping pixels across the blocks.	Histogram of the image.



```
org.apache.hadoop.mapred.MapReduceBase
xhami.mapreduce.SobelEdgeMap
XhamiRecordReader xhamirr
public void Configure(JobConf job)
public void close()
public void map(LongWritable key, BytesWritable value, OutputCollector<IntWritable, BytesWritable> output, Reporter reporter)

org.apache.hadoop.mapred.MapReduceBase
xhami.mapreduce.Laplacin
XhamiRecordReader xhamirr
public void Configure(JobConf job)
public void close()
public void map(LongWritable key, BytesWritable value, OutputCollector<IntWritable, BytesWritable> output, Reporter reporter)

org.apache.hadoop.mapred.MapReduceBase
xhami.mapreduce.Histogram
XhamiRecordReader xhamirr
public void Configure(JobConf job)
public void close()
public void map(LongWritable key, BytesWritable value, OutputCollector<IntWritable, Text > output, Reporter reporter)
public void reduce(IntWritable key, Iterator<IntWritable> values, OutputCollector<IntWritable, Text> output, Reporter reporter)
```

XHAMI package offers XhamiRecordReader class which uses XhamiImage class for getting image information such as number of blocks, overlap among the subsequent blocks, metadata information of the image and segmented blocks.

Other MapReduce functions available as part of package are Gaussian, Dilute, erode etc…

Figure 5. XHAMI MapReduce Package for Image processing domain

Below we describe sample implementation of XHAMI MapReduce functions for image processing domain.

3.4 XHAMI –MapReduce Functions

In this section we describe the extensions for Map and Reduce functions for image processing applications. Based on the image processing operation either map function alone, or both map and reduce functions are implemented. For example, edge detection operation does not require the reducer, as the resultant output of the map function is directly written to the disk. Each map function reads the block numbers and metadata of the corresponding blocks.

Read operations can be implemented in two ways in HDFS, one way is to implement own split function, ensuring the split does not happen across the boundaries, and other one is to use FIXED LENGTH RECORD of FixedLengthInputFormat class. The package offers the implementations for both FIXED LENGTH RECORD and custom formatter XhamiRecordRecorder.

(1) MapReduce Sobel edge detection sample implementation

Edges characterize boundaries in images are areas with strong intensity contrasts- a jump in intensity from one pixel to the next. There are many ways to perform edge detection. However, the majority of different methods may be grouped into two categories, gradient, and Laplacian. The gradient method detects the edges by looking for the maximum and minimum in the first derivative of the image. The Laplacian method searches for zero crossings in the second derivative of the image to find the edges. An edge has the one-dimensional shape of a ramp and calculating the derivative of the image can highlight its location. In the map function, for edge detection, the combiner and reduce functions are not performed, as there is no need of aggregation of the individual map functions.

The differences between conventional Hadoop implementation and the XHAMI implementations are as follows- in the former, the data is organized as segmented as blocks, and there is no overlap of the line pixels across the blocks. Hence, it would be difficult to process the edge pixels of the blocks, and to process the edge pixels, one should get the two blocks, and compute the overlap pixels before it is sent to the map function for processing. Also, it would be difficult to ensure that the split does not happen within the pixel while reading. But, XHAMI hides all such low level details of data organization such as lines or pixels overlap, no split within the pixels, number of blocks the image is organized as blocks, and header information of the blocks.

(2)MapReduce Histogram sample implementation

Histogram operation computes frequency count of the pixel in the image. The histogram is computed as follows, first, the block and length of the block is read, and each block is mapped to one map function. The difference between the conventional implementation and the XHAMI MapReduce Histogram implementation are – in the former, it is necessary to ensure that the split does not happen within the pixels. The later overcomes this problem by using XHAMI Image implementation for data organization, and ensures that overlap lines (pixels) are vomited during processing by the Map function.

3.5 Writing domain specific applications by extending XHAMI package

XhamiFileIOFormat class is the base class, which hides the low level details of data organization and processing for the several applications of binary data handling. This class offers several methods for reading, writing, seeking to the particular block of the image, getting the overlap information among the subsequent blocks etc. XhamiImage class is an extended class of XhamiFileIOFormat, which offers several methods for handling the data for several applications in image processing domain. XhamiImage could be used for development of HDFS based data organization readily, or else, one can extend the class XhamiFileIOFormat for handling similar kind of image processing domain applications of their own interest. Below, we describe the procedure for writing and reading the images in HDFS format using by extending XhamiImage for writing XHAMI based applications.

- Extend XhamiImage class and implement setBlockHeader method.
- Define header class and the necessary data types for implementation of the Image processing application.
- Implement setBlockHeader method using the FSDataInputStream available in XhamImage class as member variable.
- Set the overlap required among the adjacent blocks using setoverLap method.
- Assign the source file and destination files, using the writeFile method. Internally, this method computes the total file size, by computing the total numbers of blocks that the image gets divided into and writes the corresponding block header.
- The contents of the file using getBlockCount and getBlockData methods.

Table 5 shows a sample code describing how to extend XhamiImage class for writing the images along with the image specific header while storing the image into HDFS.

Table 5.XhamiImage extension for writing block header

```
class ImageHeader implements Serializable{
int blockid, startscan, endscan, overlapscan, scanlength, blocklength, bytesperpixel;
}
public class XhamiImageIOOperations extends XhamiImage{
  //other implementation specific to the application
@override
public object setBlockHeader(int blockid){
ImageHeader ih = new ImageHeader();
 //set the details and write to FSDataOutputStream
}
```

4. Performance Evaluation

In this section we present the experiments conducted for large size images of remote sensing data having different dimensions (scans, pixels) and sizes varying approximately from 288 Megabytes to 9.1 Gigabytes. First we discuss data organization and I/O overheads for read and write, followed by performance comparison of image processing operations for histogram and Sobel edge detection filter. We conduct the experiments both on Hadoop using conventional APIs and XHAMI libraries, and discuss how XHAMI simplifies the programming complexity, and increases the performance when applied to a large scale images. The experiments are conducted to analyse the two important factors; i) performance gain/improvement of proposed XHAMI system Vs. Hadoop and similar systems, as discussed in section 4.1, and ii) I/O performance overheads for read and write operations, of XHAMI Vs. Hadoop based HDFS data organization, discussed in Section 4.2.

Table 6. System configuration

Type	Processor type	hostname	RAM (GB)	Disk (GB)
Name node	Intel Xeon 64 bit , 4 vCpus, 2.2 GHz	namenode	4	100
Job tracker	-do-	jobtracker	2	80
Data node 1	-do-	datanode1	2	140
Data node 2	Intel Xeon 64 bit , 4 vCpus, 2.2 GHz	datanode2	2	140
Data node 3	Intel Xeon 64 bit , 2 vCpus, 2.2 GHz	datanode3	2	140
Data node 4	-do-	datanode4	2	100

For the experimental study, virtualization setup based Xen hypervisor with a pool of four servers of Intel Xeon 64 bit architecture are used. The configuration of the nodes for RAM, disk storage capacity, and virtual CPUs for each node in the Hadoop cluster are shown in Table 6. Hadoop version 2.7 is configured in the fully distributed mode on all these virtual machines running with 64 bit 'Cent OS' operating system.

4.1 Performance comparisons of XHAMI, Hadoop (HDFS, MapReduce), and HIPI

Sample data sets used for experiments are from the Indian Remote Sensing (IRS) satellite series i.e. CARTOSAT-1, and CARTOSAT-2A are shown in Table 7. In the table, the columns; Image size represents the original image size in bytes in regular file system, and the resulted image size indicates the size in bytes in HDFS with overlapping of 5 scan lines using the block length computation algorithm in unidirectional approach illustrated in section 3.2. A sample image with overlap of 5 scan lines shown in red colour is depicted in Figure 6. The results show a maximum of 0.25% increase in the image size, which is negligible.

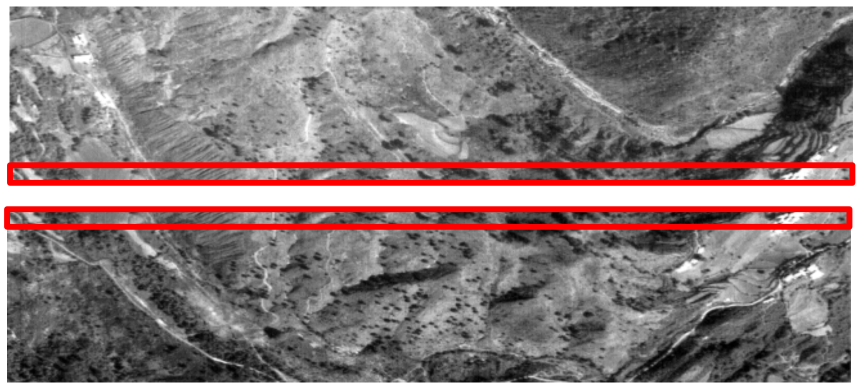


Figure 6.Image blocks with overlap highlighted in rectangular box

Table 7. Sample data sets used and the resultant image size

S.No	Image size (in bytes)	Scan line length	Total Scan lines	Resulted Image size (in bytes)
1	288000000	12000	12000	288480000
2	470400000	12000	19600	471240000
3	839976000	12000	34999	841416000
4	1324661556	17103	38726	1327911126
5	3355344000	12000	139806	3361224000
6	9194543112	6026	762906	9202738472

Here, we discuss the I/O performance of HDFS data organization in XHAMI and default Hadoop for single large volume image handling, followed by MapReduce image processing in XHAMI, customized MapReduce for Hadoop, and HIPI. The data sets used for the experiments shown in Table 7. Here, Hadoop, in its native form cannot be used for MapReduce Image processing, due to the overlapped data requirements as discussed in Section 1, hence, we few customizations are done, to default Hadoop MapReduce, like locating the neighbour adjacent block data and its location, and migrating them to the node where Map functioning is to be computed. We present the results for image processing operations such as histogram and Sobel filter, followed by read and write overheads. The I/O overhead comparisons are presented for both Hadoop, and XHAMI. The advantage of HIPI over customised Hadoop is the use of Java Image Processing Library. HIPI comes with processing the image formats like jpg, and png files, hence do not require the additional implementation functions for image handling. As HIPI uses HIB formats to create a single bundle for smaller image files, but, the data sets used for the experiments are very large, hence, here HIB consists of a single image itself, unlike the smaller files bundled into a single image.

(a) Histogram operation

Histogram operation counts the frequency of the pixel intensity in the entire image, which is similar to counting the words in the file. The performance results of histogram operation for customized Hadoop MapReduce (MR), HIPI and XHAMI are shown in Figure 7. For customized Hadoop MR, data is organized as non overlapped blocks, for XHAMI, data blocks are with overlapping data, and extensions of HDFS and MapReduce APIs for processing. For HIPI processing, the data blocks are retrieved using HIB files, and for XHAMI the data is retrieved from the data blocks having overlap region with their corresponding immediate subsequent block. The results show that, all the three systems Customized Hadoop, HIPI, and XHAMI performance are more or less similar, and has XHAMI has little overhead which is less than 0.8% with customized Hadoop, which is due to skipping of the overlapped scan lines while processing.

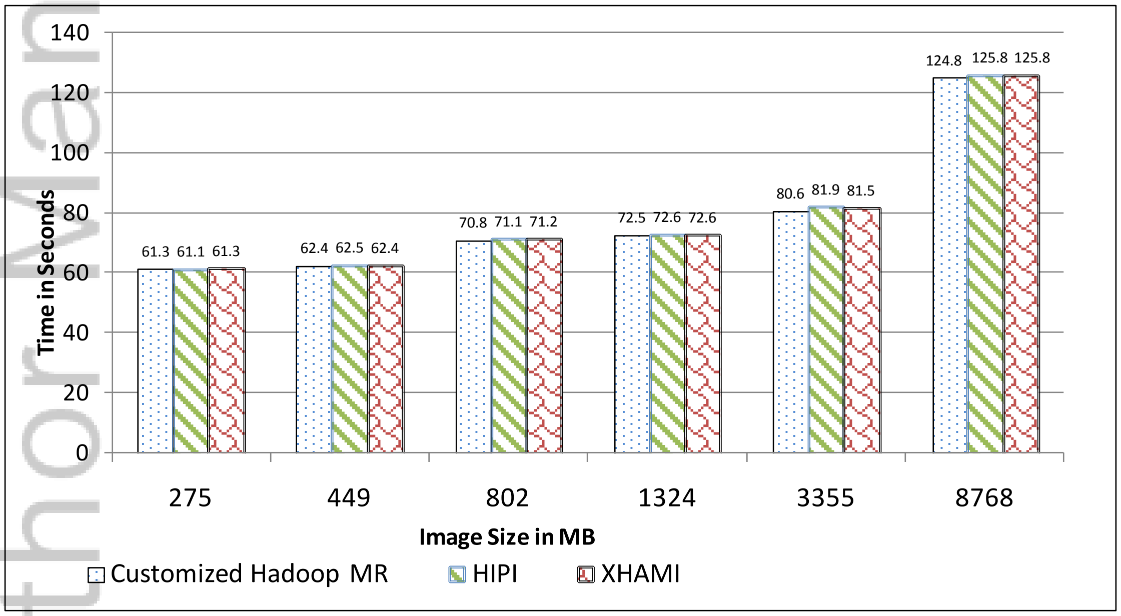


Figure 7. Histogram performance

(b) Fixed mask convolution operation

Convolution is a simple mathematical operation which is fundamental to many common image processing operators. The convolution is performed by sliding the kernel over the image, generally starting at the top left corner, so as to move the kernel through all the position where the kernel fits entirely within the boundaries of image. Convolution methods are the most common operations used in image processing which uses the mask operator i.e. kernel for performing windowing operations on the images. Sobel operator is one of the commonly used methods for detecting edges in the image using convolution methods. In case if the image is organized as physical partitioned distributed blocks, then, convolution operations cannot process the edge pixels of such blocks, due to the non availability of the adjacent blocks data on the same node. In conventional Hadoop based HDFS and MapReduce processing, the data is organized as physical partitioned blocks, hence, the edge pixels cannot be processed directly, and demands the additional I/O overheads for processing the edge pixels of each block.

Here, we present the performance of the Sobel edge detection implementation in XHAMI, and compare it with customized Hadoop MapReduce (MR) and HIPI. In customized Hadoop MR; data is physically partitioned as non overlapping data blocks, and for HIPI data is organized as single large block for the file stored in HIB data format. For MapReduce processing; customized Hadoop MR, uses an additional functionality is included in Map function for retrieving the adjacent block information corresponding to the block to be processed. In the case of HIPI the logic cannot be added, due to the non availability of HIPI APIs to know corresponding adjacent block information. The results are depicted in Figure 8, compare the performance of XHAMI with customized Hadoop MR and HIPI. The results indicate that, the performance of XHAMI is much better, and which is nearly half of the time taken by Customized Hadoop MR, and it is extremely better over HIPI. Customized Hadoop MR is implemented with standard Java Image Processing Library, with few customized features over default Hadoop, like retrieving the adjacent blocks information in Map functions for processing the edge pixels of the blocks. This customization requires additional overheads, increasing both the programming and computational complexities. The additional overheads are mainly due to the transfer of whole data block which is located in different data nodes, than the one where Map function to be processed.

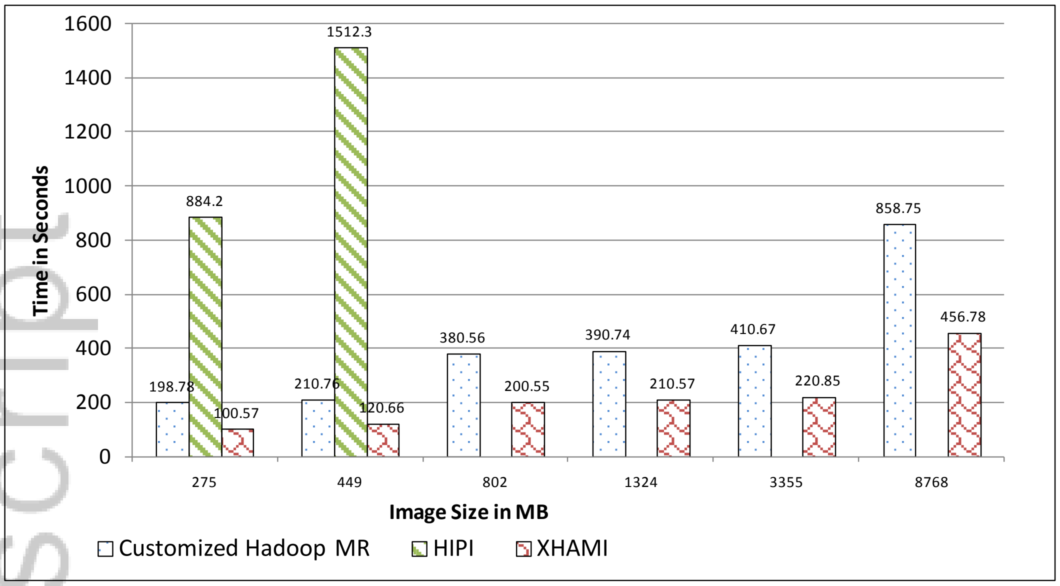


Figure 8.Sobel filter performance

HIPI has in built Java Processing APIs for processing the jpg and png image formats. For HIPI, the data is organized a single large block equivalent to the size of the image, as there is no functionality readily available for retrieving the adjacent blocks information. Due to this reason, the experiments for the larger image size data sets starting from Serial numbers 3 and above mentioned in the Table 8 could not be conducted. XHAMI overcomes the limitations of both Hadoop, and HIPI, by extending the Hadoop HDFS and MapReduce functionalities with the overlapped data partitioned approach, and MR processing using high level APIs with the integrated open source GDAL package for handling several types of image formats. XHAMI not only simplifies the programming complexity, but also allows the development of image processing applications quickly over Hadoop framework using HDFS and MapReduce.

Table 8. Read/write performance overheads

S.No	Image size (MB)	Write (Sec)		Read (Sec)	
		Default Hadoop	XHAMI	Default Hadoop	XHAMI
1	275	5.865	5.958	10.86	10.92
2	449	14.301	14.365	19.32	19.45
3	802	30.417	30.502	40.2	40.28
4	1324	44.406	77.153	50.28	50.95
5	3355	81.353	88.867	90.3	90.6
6	8768	520.172	693.268	550.14	551.6

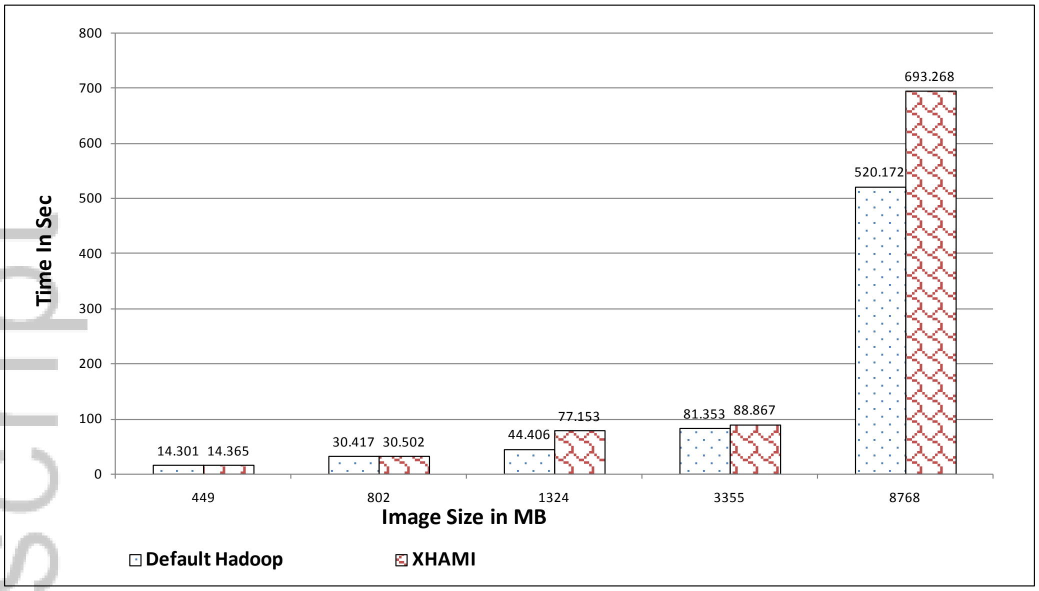


Figure 9. Image write performance

4.2 Read / write overheads

Performance of read and write function in default Hadoop and XHAMI with overlap of 5 scan lines in horizontal partition direction is shown in Figure 10 and Figure 9 respectively. The results shown in Figure 10 represents a negligible read overhead, as the scan lines to be skipped are very few, and also the position of those lines to skip are known prior. Write performance overheads are shown in Figure 9, indicate that, XHAMI has minimal overheads compared with default Hadoop, and it is observed that data sets in serial nos. 1, 2, 3 and 5 is less than 5%, and for other data sets it is 33%.

The data sets 4 and 6 are with larger scan line lengths, this in turn has consumed more storage disks space, resulting more read and writes overheads. Performing the vertical direction partition also has resulted in more write overheads for these two data sets, as the number of pixels in vertical direction is more compared to the scan direction. Read performance for all the data sets is less than 0.2% which is very negligible. Hence, the better mechanism to choose the partitioning approach is use to compute the number of blocks during data partition either in horizontal or vertical direction, and subsequently compute the storage overhead for each blocks followed by all the blocks. Based on the storage overheads, a data partitioning approach selection can be made, for the one resulted in minimal write overheads. But it is to be observed that for image processing explorations, in general the images are written once and read several times, hence, the writing overhead is one time activity, which is negligible, while compared with the overall performance achieved while processing.

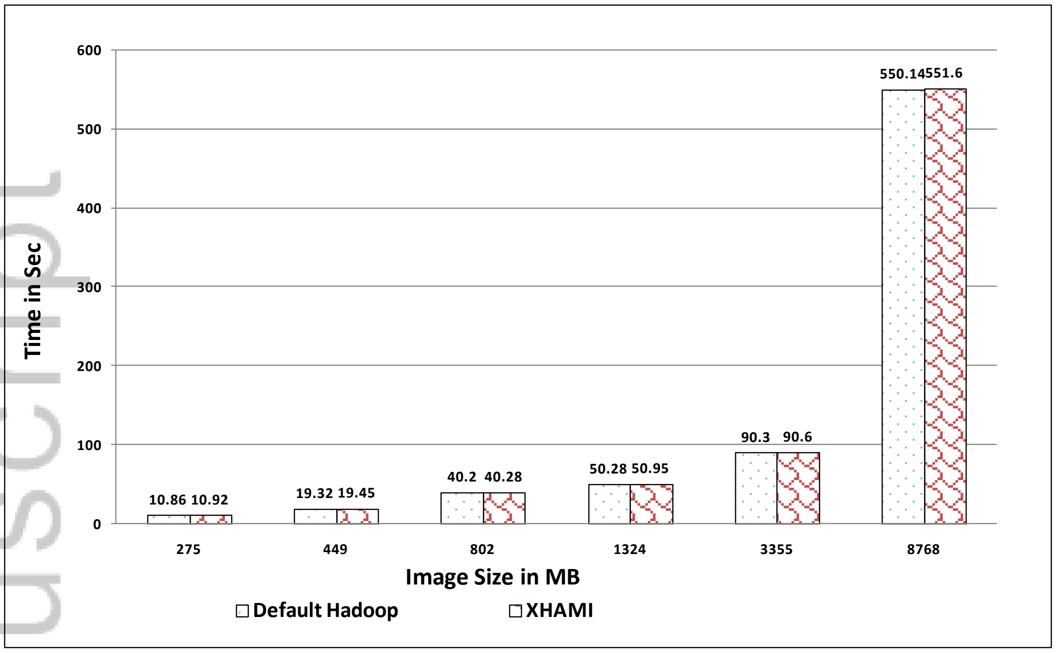


Figure 10. Image read performance

5. Conclusions and Future Work

Image processing applications deal with processing of pixels in parallel, for which Hadoop and MapReduce can be effectively used to obtain higher throughputs. However many of the algorithms in Image Processing, and other scientific computing, require use of neighbourhood data, for which the existing methods of data organization and processing are not suitable. We presented an extended HDFS and MapReduce interface, called XHAMI, for image processing applications. XHAMI offers extended library of HDFS and MapReduce to process the single large scale images with high level of abstraction over writing and reading the images. APIs are offered for all the basic forms Read/Write and Query of images. Several experiments are conducted on sample of six data sets with a single large size image varying from approximately 288 MB to 9.1 GB.

Several experiments are conducted for reading and writing the images with and without overlap using XHAMI. The experimental results are compared with the Hadoop system, and HIPI, shows that, though the proposed methodology incurs marginal read and write overheads, due to overlapping of data, however, the performance has scaled linearly and also programming complexity is reduced significantly. Currently XHAMI has functions for the data partitioning in horizontal direction, it needs to be extended for both in vertical direction, and bi-directional (both horizontal, and vertical).

The system is implemented with both the fixed length record and the customized split function which hides the low level details of handling and processing, spawning more map functions for processing. However, challenges involved in organizing the sequence of executed map functions for aggregations need to be addressed. We plan to implement the bi-directional split also in the proposed system, which would be the requirement for large scale canvas images. The proposed MapReduce APIs could be extended for many more Image processing and Computer vision modules. It is also proposed to extend the same to multiple image formats in the native format itself.

Currently, image files are transferred one at a time from the local storage to Hadoop cluster. In future, Data aware scheduling discussed in our earlier work [29] will be integrated for the large scale data transfers from the replicated remote storage repositories and performing group scheduling on the Hadoop cluster.

Acknowledgements

We express our thanks to Mr. Nooka Ravi of ADRIN for discussion on GDAL library, and Mr. Rama Krishna Reddy V. for discussions on image filters based on spatial and frequency domains. We thank Smt. GeetaVaradan, Ex-Director ADRIN, Sri. Santanu Chowdhury, Director ADRIN, and Dr. S. VenkatRaman, Group Director ADRIN, for their support and encouragement in pursuing the research. This paper is a substantially extended version of our 9-page conference paper [30].

References

[1] IDC, *The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest growth in the Far East*, www.emc.com/leadership/digital-universe/index.htm.

[2] R. Kune, P. K. Konugurthi, A. Agarwal, R. Chillarige, and R. Buyya, The Anatomy of Big Data Computing, *Software: Practice and Experience (SPE)*, 46(1): 79-105, Wiley Press, New York, USA, Jan. 2016.

[3] K.Bakshi, *Considerations for Big Data: Architecture and Approach*, Proceedings of the 2012 IEEE Aerospace Conference- Big Sky, Montana, USA, March 2012.

[4] K. Michael, and K. W. Miller, Big Data: New opportunities and New Challenges, *IEEE Computer*, 46 (6) (2013): 22-24.

[5] The Apache Hadoop Project, http://hadoop.apache.org

[6] J. Kelly, *Big Data: Hadoop, Business Analytics and Beyond*, Wikibon White paper, 27th August 2012, http://wikibon.org/wiki/v/Big Data: Hadoop, Business Analytics and Beyond.

[7] S. Ghemawat, H. Gobioff, and S. T. Leung, *The Google File System*, In Proc. 9th ACM Symposium on Operating System Principles (SOSP 2003),NY, USA 2003.

[8] K. Schvachko, H. Kuang, S. Radia, R. Chansler, *The Hadoop Distributed File System*, Proceedings of the26thIEEE Symposium on Mass Storage Systems and Technologies (MSST 2010), Incline Village, Nevada, USA, May 2010.

[9] J. Dean, and S. Ghemat, MapReduce: Simplified Data Processing on Large Cluster, *Communications of the ACM*, 51(1) (2008): 107-113.

[10] C. Jin, and R. Buyya, *MapReduce Programming Model for .NET-Based Cloud Computing*, Euro-Par 2009 Parallel Processing, Lecture Notes in Computer Science, 5704 (2009): 417-428.

[11] J. Ekanayake, S. Pallickara, and G. Fox, *MapReduce for Data Intensive Scientific Analyses*, Proc. IEEE fourth International Conference on e-Science, Indiana Police, Indiana, USA, Dec 2008.

[12] Satellite imaging corporation, http://www/satimagingcorp.com/satellite-sensors

[13] Y. Ma, H. Wu, L. Wang, B. Huang, R. Ranjan, A. Zomaya, and W. Jie, Remote Sensing Big Data computing: Challenges and opportunities, *Future Generation Computer Systems*, Volume 51, October 2015, Pages 47–60.

[14] R. C. Gonzalez, and R. E. Woods, *Digital Image Processing*, 3rd Edition, 2007 (chapters 1, and 3).

[15] X. Cao, S. Wang, *Research about Image Mining Technique*, Journal of Communications in Computer and Information Sciences, 288(2012): 127-134.

[16] C. Sweeney, L. Liu, S. Arietta, and J. Lawrence, *HIPI: A Hadoop Image Processing Interface for Image-based MapReduce Tasks*, undergraduate thesis, University of Virginia, USA.

[17] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox, *Twister: A Runtime for Iterative MapReduce*, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010),Chicago, Illinois, USA 2010.

[18] W. Jiang, V. T. Ravi, G. Agarwal, *A Map-Reduce System with an Alternate API for Multi-Core Environments*, Proceedings of the10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID 2010), Melbourne, May 2010.

[19] L. Kennedy, M. Slaney, and K. Weinberger, *Reliable Tags using Image Similarity: Mining Specificity and Expertise from Large-Scale Multimedia Databases*, Proceedings of the1stWorkshop on Web-Scale Multimedia Corpus, Beijing, October 2009.

[20] L. L. Shi, B. Wu, B. Wang, and X. G. Yang, *Map/Reduce in CBIR Applications*, Proceedings of the International Conference on Computer Science and Network Technology (ICCSNT), Harbin, December 2011.

[21] C. T. Yang, L. T. Chen, W. L. Chou, and K. C. Wang, *Implementation on Cloud Computing*, Proceedings of the IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS 2011), Beijing, September 2011.

[22] H. Kocalkulak, and T. T. Temizel, *A Hadoop Solution for Ballistic Image Analysis and Recognition*, Proceedings of the International Conference on High Performance Computing and Simulation (HPCS 2011), Istanbul, July 2011.

[23] M. H. Almeer, *Cloud Hadoop MapReduce For Remote Sensing Image Analysis*, Journal of Emerging Trends in Computing and Information Sciences, vol. 3, 637-644.

[24] I. Demir, and A. Sayar, *Hadoop Optimization for Massive Image Processing: Case Study of Face Detection*, International Journal of Computers and Communications and Control, 9(6) (2014), 664-671.

[25] T. White, The Small files problem: 2009, http://www.cloudera.com/blog/2009/02/02/the-small-files-problem.

[26] K. Potisepp, *Large Scale Image Processing Using MapReduce*, MSc. Thesis, Institute of Computer Science, Tartu University, 2013.

[27] GDAL, Geospatial data abstraction library. http://www.gdal.org/

[28] Apache HBASE project, http://www.hbase.apache.org

[29] R. Kune, P. Konugurthi, A. Agarwal, C. Rao, and R. Buyya, *Genetic Algorithm based Data-aware Group Scheduling for Big Data Clouds*, Proceedings of the International Symposium on Big Data Computing (BDC 2014), London, December 2014.

[30] R. Kune, P. Konugurthi, A. Agarwal, C. R. Rao, and R. Buyya, *XHAMI - Extended HDFS and MapReduce Interface for Image Processing Applications*, Proceedings of the4th International Conference on Cloud Computing for Emerging Markets (CCEM 2015, IEEE Press, USA), Bangalore, India, November 25-27, 2015.