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

Raghavendra Kune¹, Pramod Kumar Konugurthi¹, Arun Agarwal², Raghavendra Rao Chillarige², Rajkumar Buyya³

¹Advanced Data Processing Research Institute, Department of Space, India
²School of Computer and Information Sciences, University of Hyderabad, India
³Cloud 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 as 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 an 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] have discussed challenges and opportunities in Remote Sensing (RS) Big Data computing, focused 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 examples 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, X6, X7, X8, and X9. If the image is organized as a single large file, then the blocks could be depicts the image represented as blocks, and the corresponding block boundaries, and the pixels in the block boundaries. As shown in Figure 1, for processing the pixels falling on the block boundaries, there is necessity of its adjacent block pixels. Therefore due to the dependency, these types of filtering operations cannot be performed on the edge pixels. For example, in Figure 1, processing the pixels such as b1, b2, b3 of Block 1, requires its adjacent block pixels such as y1, y2, y3. For the images, organized as a single file, it is easy to get the overlapping pixels, by moving the file pointer back and forth for processing. But, in case if, the images are organized as segmented blocks, as shown in Figure 1, and those blocks are processed at the data nodes (computing nodes at which data is available), it is essential to organize the data such that, the required data pixels can be obtained without comprising on the performance.

Hadoop and many of 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, ... also Image Processing (IP) operations 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. This additional read operations for each block increase the 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 and other related systems. 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 using HDFS and MapReduce programming models. In addition to Hadoop, there are several other frameworks like Twister [16] for iterative computing of streaming text analytics, and Phoenix [17] used for map and reduce functions for distributed data intensive Message Passing Interface (MPI) kind of applications.

Kennedy et al. [18] demonstrated the use of MapReduce for labelling 19.6 million images using nearest neighbour method. Shi et al. [19] 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. [20] presented a system MIFAS for fast and efficient access to medical images using Hadoop and Cloud computing. Kocalkulak et al. [21] proposed a Hadoop based system for pattern image processing of intercontinental missiles for finding the bullet patterns. Almeer et al. [22] designed and implemented a system for remote sensing image processing with the help of Hadoop and Cloud computing systems for small scale images. Demir [23] 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. [24] 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.
and presented Hadoop Image Processing Interface (HIPI) as an extension of MapReduce APIs for image processing applications. HIPI operates on the smaller image files, which are bundled into a large block called Hadoop Image Bundle (HIB). In HIPI each image is applied to only one map function, which has limitation in dividing the data into smaller file sets. All these said methods discussed aggregation of smaller images and mapping each image within the bundle as a whole to one single map function.

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 under consideration.

The above works focused on small files problem by combining the smaller images into large bundle like HIPI does, or they are good for handling the images and lack in performing the image related spatial filters applications as the overlap data among the adjacent blocks is not available. They compared the performance of the single PC system with the Hadoop based clusters, and do not address the data handling and processing mechanisms using Hadoop effectively. In this paper, the issues related to processing large remote sensing images which run into several Megabytes to Gigabytes are addressed, along with the several issues related to data organization over HDFS, and processing them by MapReduce using extended HDFS and MapReduce called as XHAMI library. Our proposed XHAMI is applied for describing the applications for remote sensing or geo sciences data perspective. However, the same could be either readily applied or can be extended for other domains such as medical imaging where such similar data dependencies exist in processing.

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.

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 for data organization followed by extended HDFS and MapReduce APIs.

Figure 2.XHAMI for read/write operations
3.2 Data Organization

The image is organized as blocks in HDFS with overlap among the subsequent blocks. The image as a one dimensional sequence of bytes is shown in Figure 3.a, and the construction of the blocks, where the image in one dimensional image represented as blocks and the two ways of block arrangement is described Figure 3.b and Figure 3.c. The blocks are 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.

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

$$ Sb = \left\lfloor \frac{H}{(L \times P)} \right\rfloor $$

$H$ = 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 $\alpha$ number of scan lines is

$$ T = \left\lfloor \frac{S}{Sb} \right\rfloor $$

If $T \times \alpha > Sb$ 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.

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

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

Block length is computed as below.

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

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 some additional storage overhead, 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.

```
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()

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()
```

```
org.apache.hadoop.FileInputFormat
xhami.io.XhamiFileIOFormat
Configuration ... stpixel, int width, int height)
int getPixel(int scan, int pixel)
abstract public Object setBlockHeader(int blockid)
```

```
xhami.io.XhamiImage
byte header[]
DataSet dataset

public 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 setBlockHeader method. XhamiRecordReader reads the buffer data and sends to the Map function for processing.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Method description</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean setxhamiConfiguration(String coresite, String hdfssite)</td>
<td>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.</td>
<td>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.</td>
</tr>
<tr>
<td>boolean create(String fileName)</td>
<td>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.</td>
<td>Returns true if the file is not present in HDFS, or else returns false.</td>
</tr>
<tr>
<td>FSDataInputStream open(String filename)</td>
<td>Checks if the file is present in the HDFS.</td>
<td>If the file is present FSDataInputStream having the object value is returned, otherwise FSDataInputStream with value having null is returned.</td>
</tr>
<tr>
<td>void setoverLap(int overlap)</td>
<td>Used to set the overlap across the segmented blocks. The supplied overlap value is an integer value corresponding to the overlap size in bytes.</td>
<td>-</td>
</tr>
<tr>
<td>void seek(long position)</td>
<td>Moves the file pointer to the location position in the file.</td>
<td>-</td>
</tr>
<tr>
<td>int getoverLap()</td>
<td>Reads the overlap value set for</td>
<td>Return the overlap value if set for</td>
</tr>
</tbody>
</table>
the file while writing to HDFS. the file or else returns -1.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>void write(byte[] b)</td>
<td>Writes b number of bytes to the file.</td>
<td>-</td>
</tr>
<tr>
<td>byte[] read(int length)</td>
<td>Reads length number of bytes from the file.</td>
<td>Returns the data in the byte array format.</td>
</tr>
<tr>
<td>void close()</td>
<td>Closes the data pointers those are opened for reading / writing the data.</td>
<td>-</td>
</tr>
<tr>
<td>RecordReader&lt;Long Writable, ByteArray&gt;</td>
<td>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.</td>
<td>Returns the Default Hadoop RecordReader Object for processing by the MapReduce job.</td>
</tr>
</tbody>
</table>

Table 2. XhamiImage class description

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

Package hierarchy of MapReduce for Image processing domain is shown in Figure 5 and methods are described in Table 4.
Table 3. Methods in XhamiRecordReader

<table>
<thead>
<tr>
<th>Method name</th>
<th>Method description</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize(InputSplit genericSplit, TaskAttemptContext context)</td>
<td>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.</td>
<td>-</td>
</tr>
<tr>
<td>boolean nextKeyValue()</td>
<td>Overrides the nextKeyValue method of its base class RecordReader.</td>
<td>Returns true if it can read the next record for the file, or else return false.</td>
</tr>
<tr>
<td>Long Writable getCurrentKey()</td>
<td>getCurrentKey method of its base class is overridden.</td>
<td>Return Writable Object of the record recorder method.</td>
</tr>
<tr>
<td>float getProgress()</td>
<td>Overriding method, to send the progress of the data read.</td>
<td>Return float false representing the percentage of data read so far from the XhamiRecordRecorder, corresponding to the InputSplit.</td>
</tr>
<tr>
<td>byte[] getCurrentValue()</td>
<td>Reads the bytes array to be sent for computing for Map function.</td>
<td>Return byte array if true, else returns NULL object.</td>
</tr>
<tr>
<td>void close()</td>
<td>Closes the record reader object.</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4. Description of XHAMIMapReduce classes for Image Processing domain

<table>
<thead>
<tr>
<th>MapReduce Class</th>
<th>Description</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sobel</td>
<td>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.</td>
<td>Output Images with the detected edges.</td>
</tr>
<tr>
<td>Laplacian</td>
<td>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 window size to be read for processing. Output is written to the HDFS file system.</td>
<td>Output Images with the detected edges.</td>
</tr>
<tr>
<td>Histogram</td>
<td>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.</td>
<td>Histogram of the image.</td>
</tr>
</tbody>
</table>
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. The sample job configuration for MapReduce operation in java language is shown Table 5.

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.

---

**Figure 5. XHAMI MapReduce Package for Image processing domain**

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…

---

```java
// XhamiRecordReader for SobelEdgeMap
public void configure(JobConf job) {}
public void close() {}
public void map(LongWritable key, BytesWritable value, OutputCollector<IntWritable, BytesWritable> output, Reporter reporter) {}

// XhamiRecordReader for Laplacin
public void configure(JobConf job) {}
public void close() {}
public void map(LongWritable key, BytesWritable value, OutputCollector<IntWritable, BytesWritable> output, Reporter reporter) {}
```
Table 5. Sample job configuration for MapReduce

1. JobConf conf = new JobConf(XhamImageMROperations.class); //setting MapReduce job configuration
2. conf.setWorkingDirectory(new Path("hdfs://namenode/user/hduser"); //setting configuration working directory.
3. conf.addResource(newPath("/home/hduser/hadoop-2.7.0/etc/hadoop/core-site.xml"); // adding the core site file.
4. conf.addResource(newPath("/home/hduser/hadoop-2.7.0/etc/hadoop/hdfs-site.xml"); // adding the hdfs site information file.
5. conf.setInt(XhamiFileIOFormat.RecordReader); //setting XhamiFileIOFormat recorder class
6. conf.setInputFormat(XhamiFileIOFormat.class); //setting XhamiFileIOFormat class

(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. A sample map function implementation of XHAMI for Sobel edge detection using 3X3 image gradient operator is discussed in Table 6.

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.
Table 6. Sample map function of Sobel gradient edge operator

```
int[][] GX=
{-1,0,1},{-2,0,2},{-1,0,1}
};
int[][] GY=
{1,2,1},{0,0,1},{-1,-2,-11}
};
public void map(LongWritable key, BytesWritable value,
OutputCollector<IntWritable, BytesWritable> output,
Reporter reporter) throws IOException {
//key contains the value of the block number to be processed and
value is the buffer to be processed
byte[] data = value.getBytes(); //data array to process
//apply the kernel on the variable value using GX and GY operator on data array. The resultant output
written to a file from output data.
byte[] output = new byte[data.length]; //output array to be written.
//output buffer is written to individual files, with the sequence number as name. Once all the map functions
are completed, they would be merged to generate a single large output.
}
```

(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.
Sample code for histogram of map and reduce function is described in Table 7 and Table 8 respectively.
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.

Table 7. Histogram map function

```
public void map(LongWritable key, BytesWritable value,
OutputCollector<IntWritable, Text> output,
Reporter reporter) throws IOException {
byte[] data = value.getBytes();
byte pixelValue=0; //skip overlap scan lines
for(inti=0;i<data.length-(overLapScanLines*scanLineLength);i++){
    pixelValue= data[i];
    output.collect(new IntWritable(pixelValue), new Text(""+1));
}
```
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 XhamiImage 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 9 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.
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 the read and write performance, storage overheads of the conventional system, both with and without overlapping scan lines, followed by performance comparison of histogram and Sobel edge detection filter operations. We conduct the experiments both on conventional APIs and XHAMI libraries, and discuss how XHAMI simplifies the programming complexity and also increases the performance when applied to a large scale image over Hadoop framework. The experiments are conducted to analyse the two important factors such as i) performance gain/improvement of proposed XHAMI system over conventional Hadoop based, and similar systems ii) read/write performance disk overheads while the overlapping data approach is used over the conventional HDFS data organization. These experiments are discussed in Section 4.1, and 4.2 respectively.

<table>
<thead>
<tr>
<th>Type</th>
<th>Processor type</th>
<th>hostname</th>
<th>RAM (GB)</th>
<th>Disk (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name node</td>
<td>Intel Xeon 64 bit, 4 vCpus, 2.2 GHz</td>
<td>namenode</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>Job tracker</td>
<td></td>
<td>jobtracker</td>
<td>2</td>
<td>80</td>
</tr>
<tr>
<td>Data node 1</td>
<td></td>
<td>datanode1</td>
<td>2</td>
<td>140</td>
</tr>
<tr>
<td>Data node 2</td>
<td>Intel Xeon 64 bit, 4 vCpus, 2.2 GHz</td>
<td>datanode2</td>
<td>2</td>
<td>140</td>
</tr>
<tr>
<td>Data node 3</td>
<td>Intel Xeon 64 bit, 2 vCpus, 2.2 GHz</td>
<td>datanode3</td>
<td>2</td>
<td>140</td>
</tr>
<tr>
<td>Data node 4</td>
<td></td>
<td>datanode4</td>
<td>2</td>
<td>100</td>
</tr>
</tbody>
</table>

For the experimental study, we have used virtualized environment running on Xen hypervisor with a pool of four servers of Intel Xeon 64 bit architecture, with 2TB internal storage. Hadoop version 2.7 is configured in the fully distributed mode, running on the server pool of four virtual machines with 64 bit ‘Cent OS’, the nodes configuration is shown in Table 10.

4.1 Performance Comparisons of XHAMI and Customized Hadoop

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 11, the columns in the table, 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 way as illustrated in section 3.2. A sample image with overlap of 5 scan lines shown in red color 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](image)

<table>
<thead>
<tr>
<th>S.No</th>
<th>Image size (in bytes)</th>
<th>Scan line length</th>
<th>Total lines</th>
<th>Scan size (in bytes)</th>
<th>Resulted Image size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2880000000</td>
<td>12000</td>
<td>12000</td>
<td>288480000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4704000000</td>
<td>12000</td>
<td>19600</td>
<td>471240000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8399760000</td>
<td>12000</td>
<td>34999</td>
<td>841416000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1324661556</td>
<td>17103</td>
<td>38726</td>
<td>1327911126</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3355344000</td>
<td>12000</td>
<td>139806</td>
<td>3361224000</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9194543112</td>
<td>6026</td>
<td>762906</td>
<td>9202738472</td>
<td></td>
</tr>
</tbody>
</table>

We discuss the performance comparison of default Hadoop, HIPI and XHAMI for histogram and Sobel filter on the data sets mentioned in Table 11. The only advantage is achieved using 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 coding. As HIPI uses HIB formats to create a single bundle for smaller image files, but, here, the image sizes are large, hence, we have used only one single image is supplied to create the HIB format.

(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. However, due to XHAMI data organization the overlapped pixels need to be counted only once which may incur additional overheads.
The performance results of histogram operation of default Hadoop, HIPI and XHAMI system is shown in Figure 7. The results show that, there is no significant difference in the execution timings, which is less than 0.8%.

(b) Fixed mask convolution operation

Convolution methods are most common operation done in image processing. Sobel operator is one of the commonly used methods for detecting edges in the image. It involves multiplication of 3X3 mask around each pixel. It is to be noted for the reasons mentioned in the section 1, either default Hadoop, and HIPI implementation cannot produce required result. To overcome this limitation, it is necessary to introduce additional I/O operations from the adjacent blocks.
The performance of the Sobel edge detection shown in Figure 8, illustrates that execution time of XHAMI. It is to be noted that XHAMI implementation nearly takes the half of the time compared to default Hadoop. The comparison with HIPI is not possible; as HIPI does not organize the data blocks with the overlapping of pixels, hence the edge pixels cannot be processed. Apart from this, the default Hadoop system requires more programming complexity like reading the image, writing it to the blocks, and reading the overlapping neighbourhood pixels, etc., these processes are offered as high level APIs by XHAMI system, which not only simplifies the programming complexity but also allows the development of image processing applications rapidly on Hadoop framework, and overcomes the limitations of other frameworks such as HIPI, which is meant for processing smaller image sizes as a single large bundle, which is primarily designed to overcome smaller files problems of Hadoop. Also, XHAMI supports the GDAL APIs for handling several types of image formats, however, HIPI handles only jpg, and png formats only.

Table 12. Read/write performance overheads

<table>
<thead>
<tr>
<th>S.No</th>
<th>Image size (MB)</th>
<th>Write (Sec)</th>
<th>Read (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Default Hadoop</td>
<td>XHAMI</td>
</tr>
<tr>
<td>1</td>
<td>275</td>
<td>5.865</td>
<td>5.958</td>
</tr>
<tr>
<td>2</td>
<td>449</td>
<td>14.301</td>
<td>14.365</td>
</tr>
<tr>
<td>3</td>
<td>802</td>
<td>30.417</td>
<td>30.502</td>
</tr>
<tr>
<td>4</td>
<td>1324</td>
<td>44.406</td>
<td>77.153</td>
</tr>
<tr>
<td>5</td>
<td>3355</td>
<td>81.353</td>
<td>88.867</td>
</tr>
<tr>
<td>6</td>
<td>8768</td>
<td>520.172</td>
<td>693.268</td>
</tr>
</tbody>
</table>

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 9 and Figure 10 respectively. The results shown in Figure 9 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 10, 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 having larger scan line lengths, has resulted more storage disks space, this has resulted in increased disk space, and resulting in the more write 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 resulted while data is portioned into 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 which results in minimal write overheads. But it is to be observed that, in general images are written...
only once, and read several times for explorations, hence, the writing overhead is one time activity, which is negligible, while we compare it with the system with the overall performance.

Figure 9. Image read performance

Figure 10. Image write 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 conventional Hadoop system, the experimental results show that, though the proposed methodology incurs marginal read and write overheads, due to overlapping of data, 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. Venkat Raman, 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


