HADOOP IMAGE PROCESSING FRAMEWORK

By

SRIDHAR VEMULA

Master of Science in Computer Science

Oklahoma State University

Stillwater, OK

2015

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
Master of Science
May, 2015

HADOOP IMAGE PROCESSING FRAMEWORK

Thesis Approved:

Dr. Christopher John Crick

Thesis Adviser

Dr. Nohpill Park

Dr. David Cline

Name: SRIDHAR VEMULA

Date of Degree: MAY, 2015

Title of Study: HADOOP IMAGE PROCESSING FRAMEWORK

Major Field: COMPUTER SCIENCE

With the rapid growth of social media, the number of images being uploaded to the internet is exploding. Massive quantities of images are shared through multi-platform services such as Snapchat, Instagram, Facebook and WhatsApp; recent studies estimate that over 1.8 billion photos are uploaded every day. However, for the most part, applications that make use of this vast data have yet to emerge. Most current image processing applications, designed for small-scale, local computation, do not scale well to web-sized problems with their large requirements for computational resources and storage. The emergence of processing frameworks such as the Hadoop MapReduce[3] platform addresses the problem of providing a system for computationally intensive data processing and distributed storage. However, to learn the technical complexities of developing useful applications using Hadoop requires a large investment of time and experience on the part of the developer. As such, the pool of researchers and programmers with the varied skills to develop applications that can use large sets of images has been limited. To address this we have developed the Hadoop Image Processing Framework, which provides a Hadoop-based library to support large-scale image processing. The main aim of the framework is to allow developers of image processing applications to leverage the Hadoop MapReduce framework without having to master its technical details and introduce an additional source of complexity and error into their programs.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With the spread of social media in recent years, a large amount of image data has been accumulating. When processing this massive data resource has been limited to single computers, computational power and storage ability quickly become bottlenecks. Alternately, processing tasks can typically be performed on a distributed system by dividing the task into several subtasks. The ability to parallelize tasks allows for scalable, efficient execution of resource-intensive applications. The Hadoop MapReduce framework provides a platform for such tasks.

When considering operations such as face detection, image classification[7] and other types of processing on images, there are limits on what can be done to improve performance of single computers to make them able to process information at the scale of social media. Therefore, the advantages of parallel distributed processing of a large image dataset by using the computational resources of a cloud computing environment should be considered. In addition, if computational resources can be secured easily and relatively inexpensively, then cloud computing is suitable for handling large image data sets at very low cost and increased performance. Hadoop, as a system for processing large numbers of images by parallel and distributed computing, seems promising. In fact, Hadoop is in use all over the world. Studies using Hadoop have been performed, dealing with text data files[8], analyzing large volumes of DNA sequence data[10], converting the data of a large number of still images to PDF format, and carrying out feature selection/extraction in astronomy[18]. These examples demonstrate the usefulness of the Hadoop system, which can run multiple processes in parallel for load balancing and task management.

Most of the image processing applications that use the Hadoop MapReduce framework are highly complex and impose a staggering learning curve. The overhead, in programmer time and expertise, required to implement such applications is cumbersome. To address this, we present the Hadoop Image Processing Framework, which hides the highly technical details of the Hadoop system and allows programmers who can implement image processing algorithms but who have no particular expertise in distributed systems to nevertheless leverage the advanced resources of a distributed, cloud-oriented Hadoop system. Our framework provides users with easy access to large-scale image

data, smoothly enabling rapid prototyping and flexible application of complex image processing algorithms to very large, distributed image databases.

The Hadoop Image Processing Framework is largely a software engineering platform, with the goal of hiding Hadoop's complexity while providing users with the ability to use the system for large-scale image processing without becoming crack Hadoop engineers. The framework's ease of use and Java-oriented semantics will further ease the process of creating large scale image applications and experiments. This framework is an excellent tool for novice Hadoop users, image application developers and computer vision researchers, allowing the rapid development of image software that can take advantage of the huge data stores, rich metadata and global reach of current online image sources.

In the following section we will describe prior work in this area. Next we present an overview of the Hadoop Image Processing Framework including the Downloader, Processor and Extractor stages. Additionally, we describe our approach of distributing tasks for MapReduce. Finally, we demonstrate the potential of this framework with quantitative analysis and experiments performed on image processing tasks.

CHAPTER 2

LITERATURE REVIEW

2.1 Background

Apache Hadoop [5] is a platform that offers an efficient and effective method of storing and processing massive amounts of data. Unlike traditional offerings, Hadoop was designed and built from ground up to address the requirements and challenges of big data. Apache Hadoop use cases are many and show up in many industries, including : risk, fraud and portfolio analysis in financial services; behavior analysis and personalization in retail; social network, relationship and sentiment analysis for marketing; drug interaction modeling and genome data processing in health care and life sciences to name a few.

At its core, Apache Hadoop is a frame work for scalable and reliable distributed data storage and processing. It allows for the processing of large datasets across clusters of computers using a simple programming model. It is designed to scale up from single server to thousands of machines, aggregating the local computation and storage form each server., Rather than relying on expensive hardware, the Hadoop software detects and handles any failures that occur, allowing to achieve high availability on top of inexpensive commodity computes, each individually prone to failure. At the core of Apache Hadoop are the Hadoop Distributed File System or HDFS and Hadoop MapReduce, which provides a framework for distributed processing.

HDFS : The Hadoop Distributed File System [14] is a scalable and reliable distributed storage system that aggregates the storage of every node in a Hadoop cluster into single global file system. HDFS stores individual files in large blocks, allowing it to efficiently store very large ot numerous files across multiple machines and access individual chunks of data in parallel, without needing to read the entire file into a single computers memory. Reliability is achieved by replicating the data across multiple hosts, with each block of data being stored, by default, on three separate computers. If an individual node fails, the data remains available and an additional copy of any blocks it holds may be made on new machines to protect against failures. This approach allows HDFS to dependably store massive amounts of data.

MapReduce : MapReduce [3] is a programming model that allows Hadoop to efficiently process large amounts of data. MapReduce breaks large data processing problems into multiple steps, namely Maps and Reduces that can each be worked on at the same time on multiple computers. MapReduce programs are designed to compute large volumes of data in parallel fashion. This requires dividing the workload across large number of machines. This model would not scale to large clusters if the components were allowed to share data arbitrarily. The communication overhead required to keep data on the nodes synchronized at all times would prevent the system from performing reliably and efficiently at large scale. Instead all data elements in MapReduce are immutable, meaning they cannot be updated. If in mapping task you change an input(key, value) pair, it doesnt get reflected back in the input files; communication occurs only by generating new output(key, value) pairs which are forwarded by Hadoop system into the next phase of execution. Conceptually, MapReduce programs transform lists of input data elements into lists of output data elements. A MapReduce program will do this twice, using two different list processing idioms: map and reduce. The first phase of map reduce program is mapping. A list of data elements are provided, one at a time to a function called Mapper, which transforms each element individually to an output data element. The second phase of map reduce program is Reducing. Reducing lets you aggregate values together. A reducer function receives an iterator of input values form an input list. It then combines these values together, returning a single output value.

2.2 Prior Work

With the rapid usage increase of online photo storage and social media on sites like Facebook, Flickr and Picasa, more image data is available than ever before, and is growing every day. Every minute 27,800 photos are uploaded to Instagram,[6] while Facebook receives 208,300 photos over the same time frame. This alone provides a source of image data that can scale into the billions. The explosion of available images on social media has motivated image processing research and application development that can take advantage of very large image data stores.

White et.al [16] presents a case study of classifying and clustering billions of regular images using MapReduce. It describes an image pre-processing technique for use in a sliding-window approach for object recognition. Pereira et.al [12] outlines some of the limitations of the MapReduce model when dealing with high-speed video encoding, namely its dependence on the NameNode as a single point of failure, and the difficulties inherent in generalizing the framework to suit particular issues. It proposes an alternate optimized implementation for providing cloud-based IaaS (Infrastructure

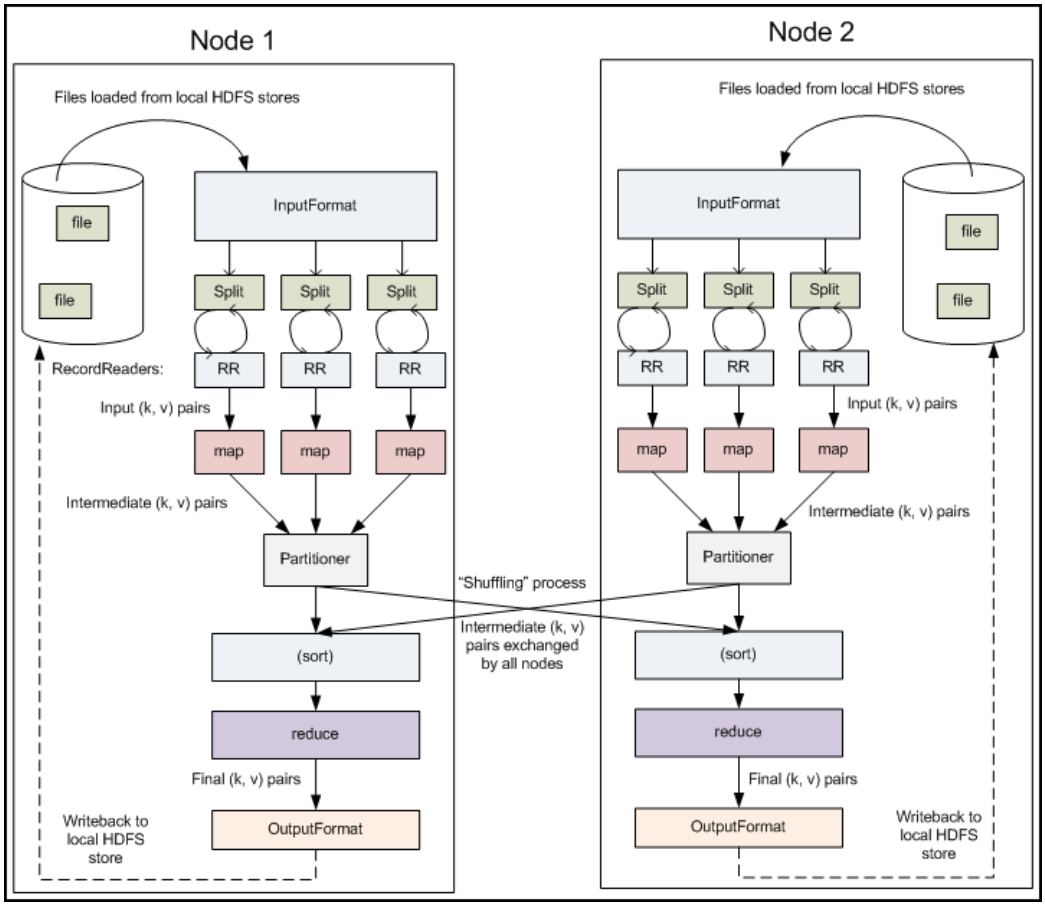

Figure 2.1: Detailed Hadoop MapReduce data flow. Source yahoo

as a Service) solutions. Lv et.al [9] describes using the k-means algorithm in conjunction with MapReduce and satellite/aerial photographs in order to find different elements based on their color.

Zhang et.al [19] presents methods used for processing sequences of microscope images of live cells. The images are relatively small (512x512, 16-bit pixels). Stored in 90 MB folders, the authors encountered difficulties regarding fitting into Hadoop DFS blocks with were solved by custom InputFormat, InputSplit and RecordReader classes. Powell et.al [13] describes how NASA handles image processing of celestial images captured by the Mars Orbiter and rovers. Clear and concise descriptions are provided about the segmentation of gigapixel images into tiles, how the tiles are processed and how the image processing framework handles scaling and works with the distributed processing. Wang, Yinhai and McCleary[15] discuss speeding up the analysis of tissue microarray images by substituting human expert analysis for automated processing algorithms. While the images were gigapixel-sized, the content was easily segmented and there was no need to analyze all of an image at once. The work was all done on a specially-built high performance computing platform using the Hadoop framework.

Bajcsy et.al [1] present a characterization of four basic terabyte-size image computations on a Hadoop cluster in terms of their relative efficiency according to a modified Amdahl's Law. The work was motivated by the fact that there is a lack of standard benchmarks and stress tests for large-scale image processing operations on the Hadoop framework. Moise et.al [11] outlines the querying of thousands of images in one run using the Hadoop MapReduce framework and the eCP Algorithm. The experiment performs an image search on 110 million images collected from the web using the Grid 5000 platform. The results are evaluated in order to understand the best practices for tuning Hadoop MapReduce performance for image search.

The above shows that there has been a great deal of intensive work in image processing using MapReduce. However, each independent project requires a complex, error-prone, one-off implementation. Such research and application development would benefit greatly from a standard, well-engineered image processing framework such as the one we provide.

CHAPTER 3

METHODOLOGY

Thesis Statement : *To design and develop an Image Library using the Hadoop MapReduce framework thereby providing a tool to users, researchers to develop of large scale image processing application with ease.*

The Hadoop Image Processing Framework is intended to provide users with an accessible, easy-to-use tool for developing large-scale image processing applications.

The main goals of the Hadoop Image Processing Framework are:

- Provide an open source framework over Hadoop MapReduce for developing large-scale image applications
- Provide the ability to flexibly store images in various Hadoop file formats
- Present users with an intuitive application programming interface for image-based operations which is highly parallelized and balanced, but which hides the technical details of Hadoop MapReduce
- Allow interoperability between various image processing libraries

3.1 Downloading and storing image data

Hadoop uses the Hadoop Distributed File System (HDFS)[14] to store files in various nodes throughout the cluster. One of Hadoop's significant problems is that of small file storage. [17] A small file is one which is significantly smaller than HDFS block size. Large image datasets are made up of small image files in great numbers, which is a situation HDFS has a great deal of trouble handling. This problem can be solved by providing a container to group the files in some way. Hadoop offers a few options:

- HAR File
- Sequence File

- Map File

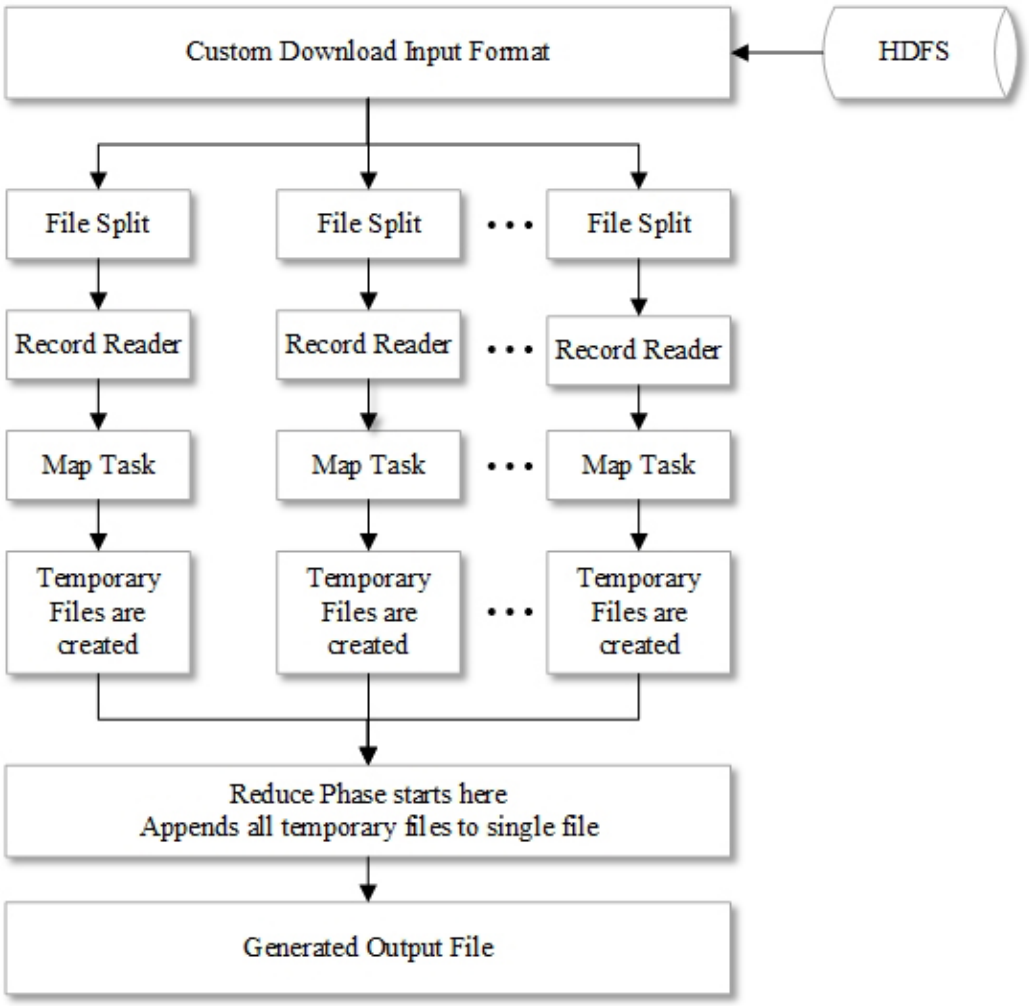

Figure 3.1: Single node running the Downloader Module (handled by the framework and transparent to the user)

The Downloader Module of our Hadoop Image Processing Framework performs the following operations:

Step 1: Input a URL List. Initially users input a file containing URLs of images to download. The input list should be a text file with one image URL per line. The list can be generated by hand, extracted from a database or a provided by a search. The framework provides an extendable ripper module for extracting URLs from Flickr and Google image searches and from SQL databases. In addition to the list the user selects the type of image bundle to be generated (e.g. HAR, sequence or map). Our system divides the URLs for download across the available processing nodes for maximum efficiency and parallelism. The URL list is split into several map tasks of equal size across

the nodes. Each node map task generates several image bundles appropriate to the selected input list, containing all of the image URLs to download. In the reduce phase, the Reducer will merge these image bundles into a large image bundle.

Step 2: Split URLs across nodes. From the input file containing the list of image URLs and the type of file to be generated, we equally distribute the task of downloading images across the all the nodes in the cluster. The nodes are efficiently managed so that no memory overflow can occur even for terabytes of images downloaded in a single map task. This allows maximum downloading parallelization. Image URLs are distributed among all available processing nodes, and each map task begins downloading its respective image set.

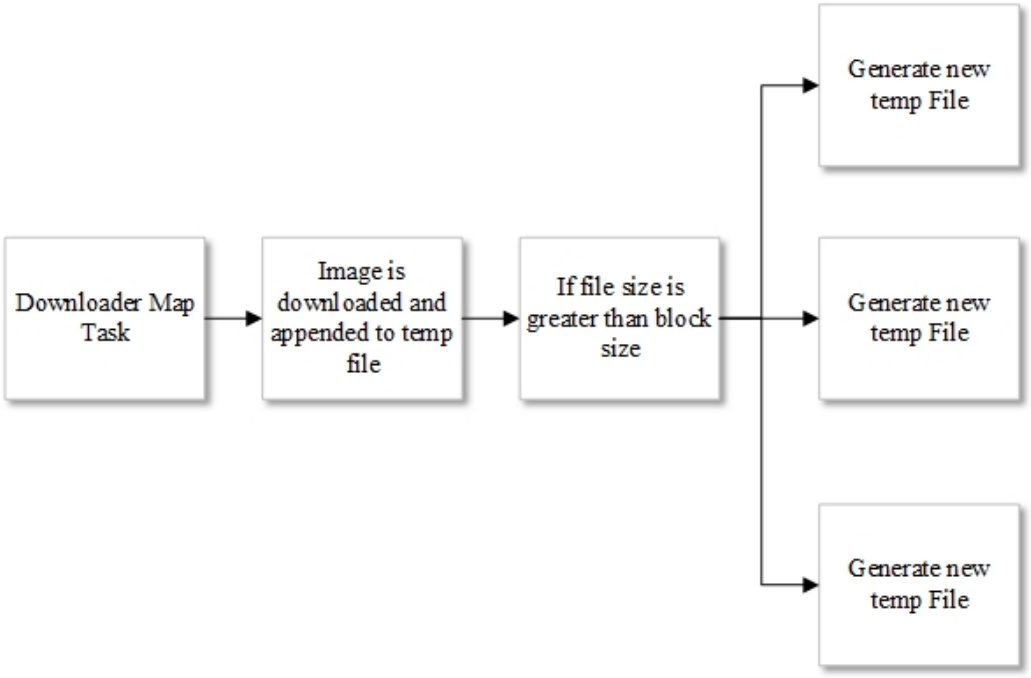

Figure 3.2: Individual map task of Downloader Module

Step 3: Download image data from URLs. For every URL retrieved in the map task, a connection is established according to the appropriate transfer protocol (e.g. FTP, HTTP, HTTPS, etc.). Once connected, we check the file type. Valid images are assigned InputStreams associated with the connection. From these InputStreams, we generate new HImage objects and add the images to the image bundle. The HImage class holds the image data and provides an interface for the user's manipulation of image and image header data. The HImage class also provides interoperability between various image data types (e.g. BufferedImage, Mat, etc.).

Step 4: Store images in an image bundle. Once an HImage object is received, it can be added to the image bundle simply by passing the HImage object to the appendImage method. Each map task generates a number of image bundles depending on the image list. In the reduce phase, all of these image bundles are merged into one large bundle.

3.2 Processing image bundle using MapReduce

Hadoop MapReduce program handles input and output data very efficiently, but their native data exchange formats are not convenient for representing or manipulating image data. For instance, distributing images across map nodes require the translation of images into strings, then later decoding these image strings into specified formats in order to access pixel information. This is both inefficient and inconvenient. To overcome this problem, images should be represented in as many different formats as possible, increasing flexibility. The framework focuses on bringing familiar data types directly to user.

As distribution is important in MapReduce, images should be processed in the same machine where the bundle block resides. In a generic MapReduce system, the user is responsible for creating InputFormat and RecordReader classes to specify the MapReduce job and distribute the input among nodes. This is a cumbersome and error-prone task; the Hadoop Image Processing Framework provides such InputFormat and RecordReaders for system's ease of use.

Images are distributed as various image data types and users have immediate access to pixel values. If pixel values are naively extracted from the image byte formats, valuable image header data (e.g. JPEG EXIF data or IHDR [4] image headers) is lost. The framework holds the image data in a special HImageHeader data type before converting the image bytes into pixel values. After processing pixel data, image headers are reattached to the processed results. The small amount of storage overhead required for this functionality is a trade-off worth making for preserving image header data after processing.

The functionality of the framework's Processor module is described below:

Step 1: Devise the algorithm. We assume that the user writes an algorithm which extends the provided GenericAlgorithm class. This class is passed as an argument to the processor module. The framework starts a MapReduce job with the algorithm as an input. The GenericAlgorithm holds an HImage variable; this allows user to write an algorithm on a single image data structure, which the framework then iterates over the entire image bundle. In addition to the algorithm, the user should provide the image bundle file that needs to be processed. Depending on the specifics of

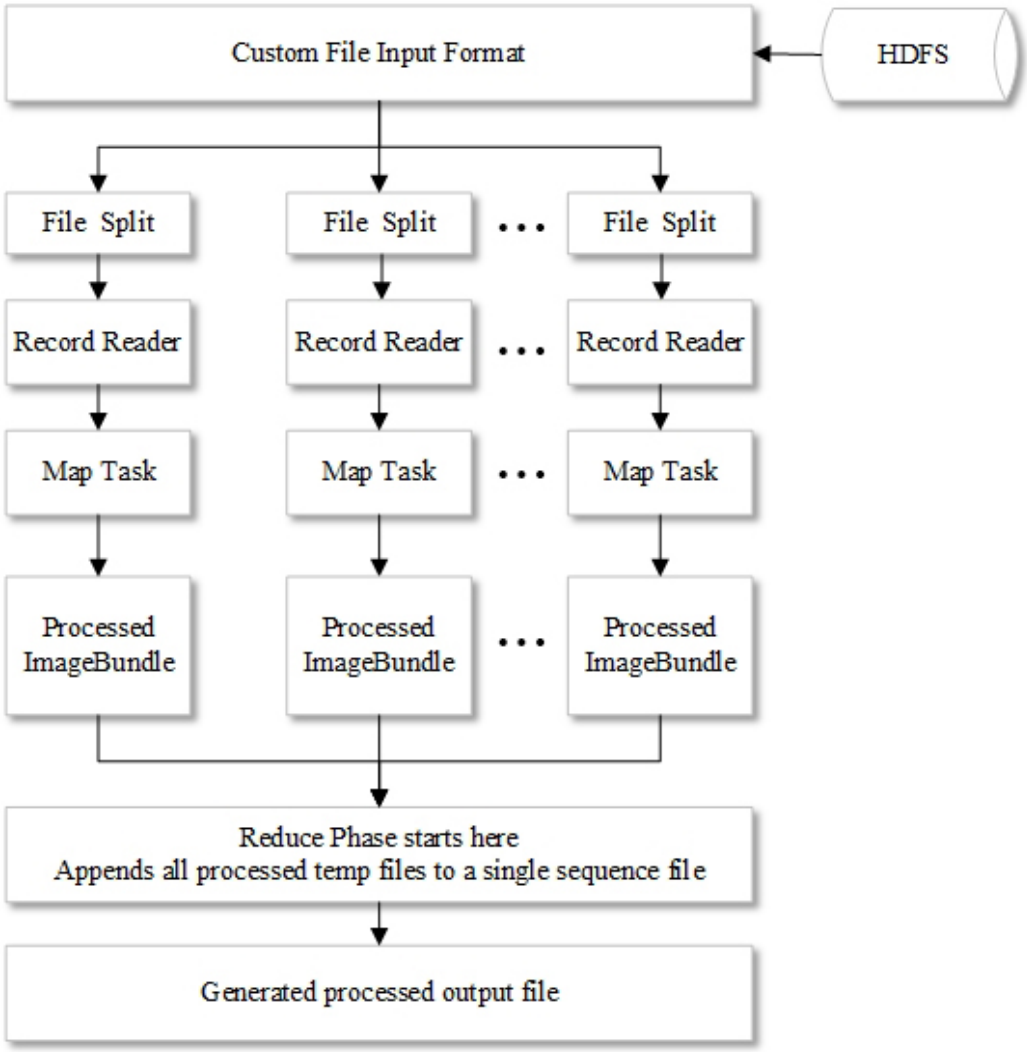

Figure 3.3: Single node running the Processor Module (handled by the framework and transparent to user)

the image bundle organization and contents, the bundle is divided across nodes as individual map tasks. Each map task will apply the processing algorithm to each local image and append them to the output image bundle. In the reduce phase, the Reducer merges these image bundles into a large image bundle.

Step 2: Split image bundle across nodes. The input image bundle is stored as blocks in the HDFS. In order to obtain maximum throughput, the framework establishes each map task to run in the same block where it resides, using custom input format and record reader classes. This allows maximum parallelization without the problem of transferring data across nodes. Each image bundle now applies different map tasks to the image data for which it is responsible.

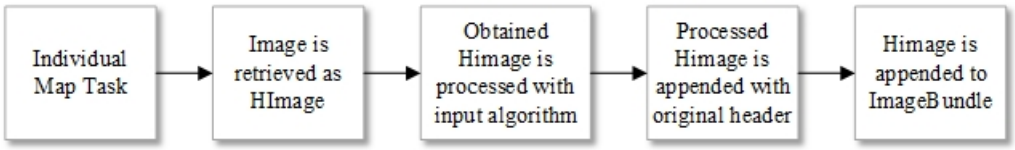

Figure 3.4: Individual map task of Processor Module

Step 3: Process individual image. The processing algorithm devised by the user and provided as input to the Processing Module is applied to every HImage retrieved in the map task. The HImage provides its image data in the data format (e.g. Java BufferedImage, OpenCV Mat, etc.) requested by the user and used by the processing algorithm. Once the image data type is retrieved, processing takes place. After processing, the preserved image header data from the original image is appended to the processed image. The processed image is appended to the temporary bundle generated by the map task.

Step 4: Store processed images in an image bundle. Every map task generates an image bundle upon completion of its processing. Once the map phase is completed there are many bundles scattered across the computing cluster. In the reduce phase, all of these temporary image bundles are merged into a single large file which contains all the processed images.

3.3 Extracting image bundles using MapReduce

In addition to creating and processing image bundles, the framework provides a method for extracting and viewing these images. Generally, Hadoop extracts images from an image bundle iteratively, inefficiently using a single node for the task. To address this inefficiency, we designed an Extractor module which extracts images in parallel across all available nodes. Distribution plays a key role in MapReduce; we want to make effective and efficient use of the nodes in the computing cluster. As previously mentioned in the description of the Processor module, a user working in a generic Hadoop system must again devise custom InputFormat and RecordReader classes in order to facilitate distributed image extraction. The Hadoop Image Processing Framework provides this functionality for the extraction task as well, providing much greater ease of use for the development of image processing applications.

Organizing and specifying the final location of extracted images in a large distributed task can be confusing and difficult. Our framework provides this functionality, and allows the user simply to specify whether images should be extracted to a local file system or reside on the Hadoop DFS.

The process of the Extractor module is explained below:

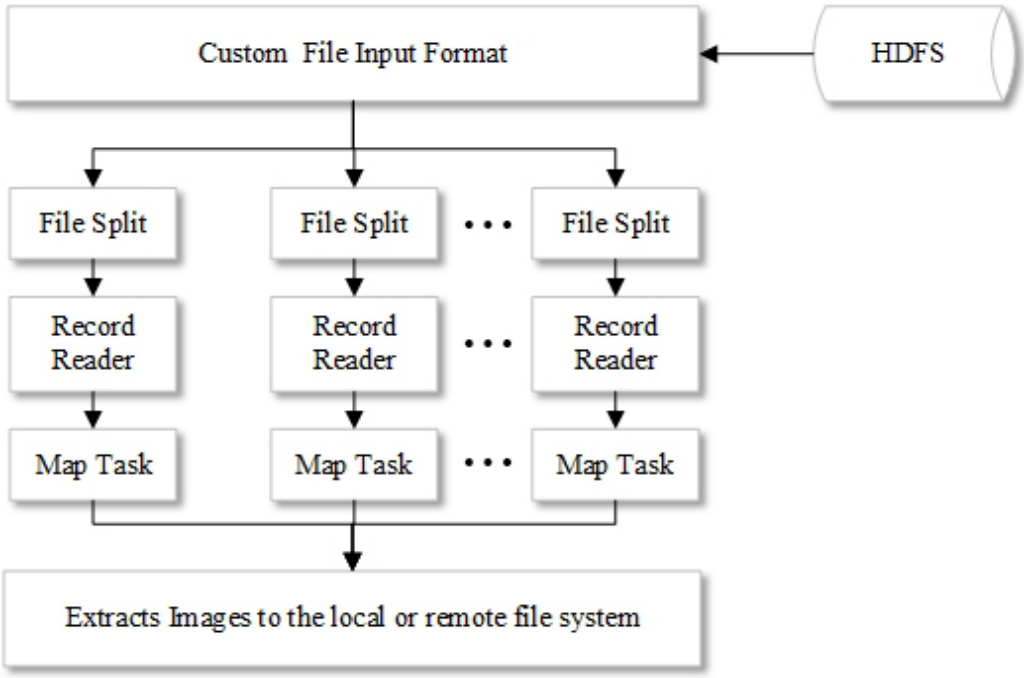

Figure 3.5: Single node running the Extractor Module (handled by the framework and transparent to the user)

Step 1: Input the image bundle to be extracted. The image bundle specified for extraction is passed as a parameter to the Extractor module. In addition, the user can include an optional parameter specifying the images' final location (defaults to local file system). The image bundle is then distributed across the nodes as individual map tasks. Each map task will extract the requisite images onto the specified filesystem.

Figure 3.6: Individual map task of Extractor Module

Step 2: Split Image bundle across nodes. The input image bundle is split across available nodes using the framework's custom input format and record reader classes for maximum throughput. Once a map task starts, HImage objects are retrieved.

Step 3: Extract individual image. The image bytes obtained from each HImage object are stored onto the filesystem in the appropriate file format based on its image type. The Extractor

module lacks a reduce phase.

CHAPTER 4

IMAGE PROCESSING ALGORITHMS

To explore the Hadoop Image Processing Framework's capabilities and performance, we implemented multiple variations of existing widely-used image processing algorithms:

- A stand-alone implementation running on a single node with no distributed processing capability
- A generic Hadoop implementation which requires a high level of software engineering expertise within the Hadoop framework
- An implementation that employs the Hadoop Image Processing Framework

We chose the following algorithms: Laplacian filtering, Canny edge detection and k-means image segmentation. These are widely-used, computation-intensive and data-intensive algorithms of varying complexity which require large distributed systems for timely operation on hundreds of thousands of images.

4.1 Laplacian Filter

The Laplacian is a 2-D isotropic measure of the 2nd spatial derivative of an image. The Laplacian of an image highlights regions of rapid intensity change and is therefore often used for edge detection. The Laplacian is often applied to an image that has first been smoothed with an approximation of a Gaussian filter in order to reduce sensitivity to noise. The operator normally takes a single gray level image as input and produces another gray level image as output.

The Laplacian $L(x, y)$of an image with pixel intensity values $I(x, y)$ is given by

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \tag{4.1}$$

This is calculated using a convolution filter.

Since the input image is represented as a set of discrete pixels, we have to find a discrete convolution kernel that can approximate the second derivatives in the definition of the Laplacian. This is a very simple computation that consists of a few additions and multiplications.

4.2 Canny Edge Detection

Canny edge detection[2] is an multi-stage algorithm to detect a wide range of edges in images. Edge detection, especially step edge detection, is an important technique to extract useful structural information from visual information, dramatically reducing the amount of data to be processed. Among existing edge detection methods, the Canny edge detection algorithm is considered to provide reliable edge detection without relying on context-specific heuristics.

The process of the Canny edge detection algorithm can defined in five stages:

- Apply a Gaussian filter to smooth the image and remove noise.
- Find the intensity gradients of the image.
- Apply non-maximum suppression to get rid of spurious responses to edge detection
- Apply a double threshold to determine potential edges.
- Track edges by hysteresis. Finalize the detection of edges by suppressing all edges that are "weak" and not connected to strong edges.

4.3 Image segmentation using k-means clustering

The k-means algorithm is an unsupervised clustering algorithm that classifies input data points into multiple classes based on their Euclidean distance from each other. The algorithm assumes that the data features form a vector space and tries to find natural clusterings within it. The points are clustered around centroids $\mu_i \forall i = 1...k$ which are obtained by minimizing the objective

$$V = \sum_{i=1}^{n} \sum_{x_j \in S_i} (x_j - \mu_i)^2 \tag{4.2}$$

where there are k clusters $S_i, i = 1, 2, ..., k$ and μ_i is the centroid or mean point of all the points $x_j \in S_i$.

For the purposes of our experiment, we implemented iterative versions of the algorithm. The algorithm takes a 2-dimensional image as input. The steps of the algorithm are as follows:

- Compute the intensity distribution (also called the histogram) of the image.
- Initialize the centroids with k random intensities.
- Repeat the following steps until the cluster labels of the image stop changing from one iteration to the next:

– Cluster the points based on the distance of their intensities from the centroid's intensity.

$$c^i := \text{argmin}_j ||x^i - \mu_j||^2 \tag{4.3}$$

– Compute the new centroid for each of the clusters.

$$\mu_i = \frac{\sum_{i=1}^{m} 1\{c_i = j\} x^i}{\sum_{i=1}^{m} 1\{c_i = j\}} \tag{4.4}$$

where k is a parameter of the algorithm (the number of clusters to be found), i iterates over all the intensities, j iterates over all the centroids and μ_i are the centroid intensities.

CHAPTER 5

EXPERIMENTAL RESULTS

We present execution time and programming complexity results for the three image processing algorithms described above, demonstrating the effectiveness of the Hadoop Image Processing Framework. These algorithms are not particularly challenging to implement on a single node, but producing a distributed Hadoop implementation is quite challenging and requires a great deal of familiarity with Hadoop concepts and distributed software engineering expertise. On the other hand, our framework provides all of the performance and scalability benefits of the Hadoop system while retaining the simplicity of the single-node implementation. In this section we describe the dataset, hardware and software specifications used in our experiments, and discuss the results.

5.1 Characteristics of Image Dataset, Hardware and Software

We performed image processing computations on a dataset of about 1 TB in size. The dataset was obtained by performing a Flickr search based on the keyword "flight". The dataset was extracted using the FlickrRipper, an instance of the Ripper interface provided by our Hadoop Image Processing Framework. The framework currently provides rippers for Flickr and Google image searches, and more will be provided in the future. A Ripper interface extracts image URLs from search results, and can be implemented for any set of results in any format. The experimental dataset is composed of 220,000 images at 4.76 MB/image, or about 1 TB.

5.2 Computer hardware and software characteristics

We ran the image processing computations on a 6-node distributed cluster and on a single-node desktop computer. The table below summarizes the cluster and desktop hardware and software specifications. The cluster nodes differ in terms of CPU speed and RAM. We installed Hadoop, Cloudera CDH5 and Java 1.7 on the cluster to support Java code execution. The desktop computer had a similar software configuration to the cluster.

The desktop implementations of the algorithms are Java programs for processing a small image dataset. As soon as the data set is processed the program halts. The desktop implementation starts

Table 5.1: Computer hardware and software characteristics.

	Specs	**Cluster**	**Desktop**
Hardware	Cluster Nodes	6 data nodes with from 2 to 4 virtual processors and 4 GB RAM and name node with 8 GB RAM	Intel Xeon @ 2Ghz 8 cores, 16GB of RAM and hyperthreading activated
	Networking	1 Gbit/second	
Software	Java Virtual Machine	Java version "1.7.0_45" Java SE Runtime Environment	Java version "1.7.0_45" Java SE Runtime Environment
	Hadoop	Hadoop 2.5 Cloudera CDH5	
	Operating System	CentOS 6.5	CentOS 6.5

as a single thread, and images are processed one at a time in succession. When the dataset size is low, each program runs smoothly and with excellent performance. However, as the size of the data to be processed reaches the terabyte scale, such a single-process approach fails utterly. Data on this scale must be processed in distributed fashion using Hadoop.

Each algorithm is also implemented on a Hadoop cluster without using the Hadoop Image Processing Framework. These implementations overcome the problem of handling large datasets but they suffer from extremely complex code. The user needs to write a great deal of code and understand the technical details of Hadoop. Writing such code, including custom InputFormats and RecordReaders, is cumbersome and error-prone. Figure 5.2 compares the actual lines of code a user needs to write in order to run our experimental algorithms (or any similar image processing application) on a Hadoop cluster.

Finally, we compare the same algorithms implemented on a Hadoop cluster using the framework. These implementations overcome the problems of handling large datasets and managing interoperability between image datatypes while holding code complexity down to the same level as the single-process, local implementation. Users enjoy automated creation of custom InputFormats and RecordReaders, and can tune settings for the desired output. The framework is so transparent that

the user needs only a few lines of code to process a large image data set. It hides the technical details of MapReduce and runs the code in highly parallelized fashion. The code required to implement each image processing algorithm is no more complex than the same algorithm implemented on a local, single-threaded system, but the performance improvement for large datasets is immense.

Table 5.2: Characteristics of computational elasticity, coding complexity and data locality for the three experimental platforms.

Computational Platform	**Computational Elasticity**	**Code Complexity**	**Data Locality**
Desktop	Low : limited by RAM and CPU of the executing computer	Normal : Complexity lies in writing the algorithm	All data are on local disk
Hadoop: without using framework	High: Nodes can be requested as needed	Heavy : Complexity lies in writing every module in framework including custom InputFormats and RecordReaders. User requires high technical expertise with Hadoop.	All data resides in HDFS. Custom InputFormats need to be created to launch computations in appropriate locations
Hadoop: using framework	High: Nodes can be requested as needed	Normal : Complexity lies in writing algorithm and is equivalent to writing on desktop platform	All data resides in HDFS. Highly parallelized.

5.3 Observations

We have explored the space of image processing algorithms, applying different hardware and software configurations to the various data sets. Specifically, we compared:

- Three image processing algorithms described in Section 4
- Datasets of varying image counts
- Configurations of hardware cluster used for computations
- Coding complexity - measured in Lines of Code (LOC)

The numerical results are shown in Figures 5.1 through 5.4.

Figure 5.1 illustrates the performance comparison of Canny edge detection on a single node, on Hadoop using our framework and Hadoop without using the framework.

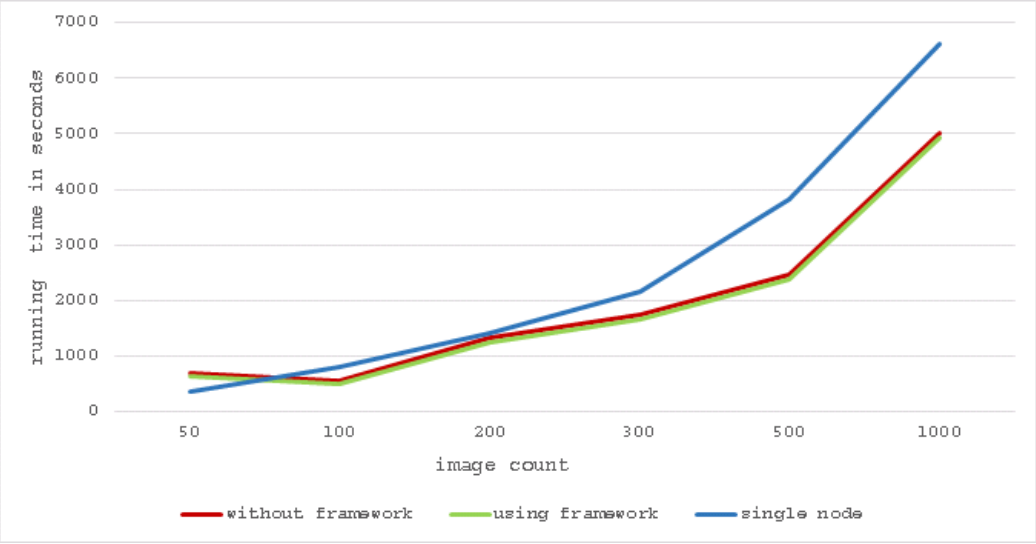

Figure 5.1: Canny edge detection computation executed on a single node and on a Hadoop cluster with and without using framework

It can be observed that the Hadoop cluster performance is nearly identical whether the Hadoop Image Processing Framework is used or not. The single node performance is satisfactory on small input sizes; in fact it outperforms Hadoop when the amount of data is very small owing to the lack of MapReduce overhead. However, as the amount of data grows the single node performance becomes increasingly poor. The same pattern is observed with any image processing algorithm.

Figure 5.2 compares the coding complexity of the different image handling and processing tasks described in Section **??**, for the Canny edge detection task. The Downloader module and Extractor module are simple to use and require few lines of code to configure. The processor module includes the specific algorithm devised by the user, which can be significant amount of code, but using our framework to configure the algorithm for MapReduce requires almost no additional coding.

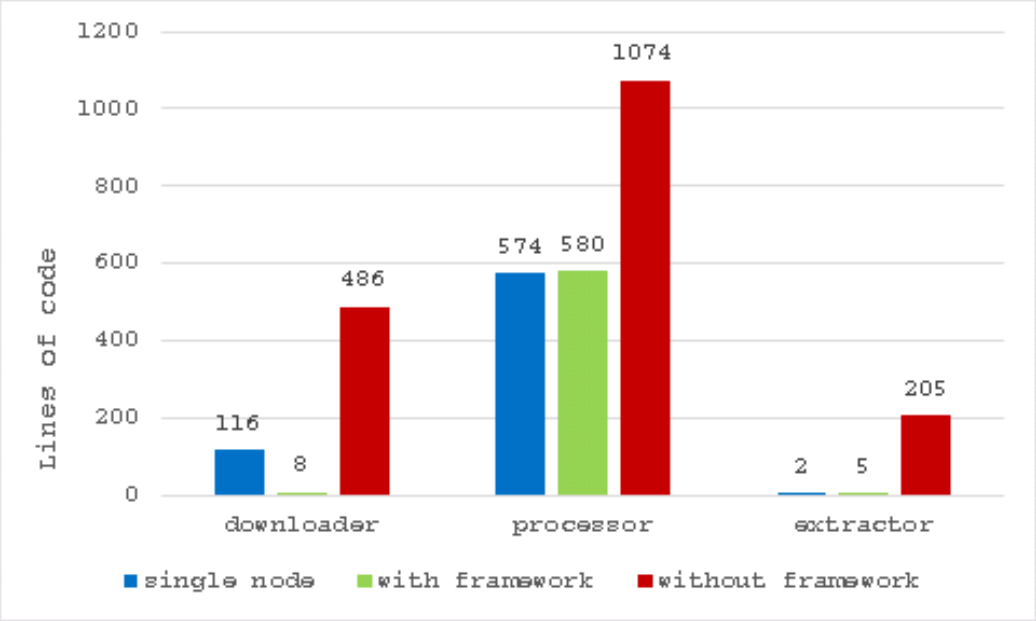

Figure 5.2: Comparing coding complexities of modules in different environments

Figure 5.3 shows the coding complexity of the processor portion of our three experimental image processing algorithms. The naive single-process, single-node algorithm requires a similar amount of code to using our highly parallel, distributed framework, while direct implementation of the algorithm using Hadoop and MapReduce requires a great deal more effort. The Hadoop Image Processing Framework hides all of this complexity from the user.

The Hadoop Image Processing Framework provides transparency on multiple levels. Users can use the predefined MapReduce modules (Downloader, Processor and Extractor) for processing the image modules, and most image processing tasks require no more than this. Users may also write custom MapReduce tasks within the framework, taking advantage of its hierarchical construction. This still protects the user from the full complexity and error-prone nature of the full Hadoop framework, but requires additional knowledge and expertise on the user's part.

The Hadoop Image Processing Framework is intended to be extremely simple to use. The framework strictly adheres to Java file writing and reading techniques, extending those conventions to the Hadoop framework's notion of reading and writing bundle files. In this way, working with Hadoop image processing is exactly like working on a single system. The main aim of the framework is to provide useful software abstractions such that programming on a Hadoop cluster is equivalent to programming on a single computer. Listing 1 demonstrates the similarity between Java code written for a single machine and the same code using the image processing framework. The framework's use of ordinary Java conventions helps in understanding the software engineering process and re-

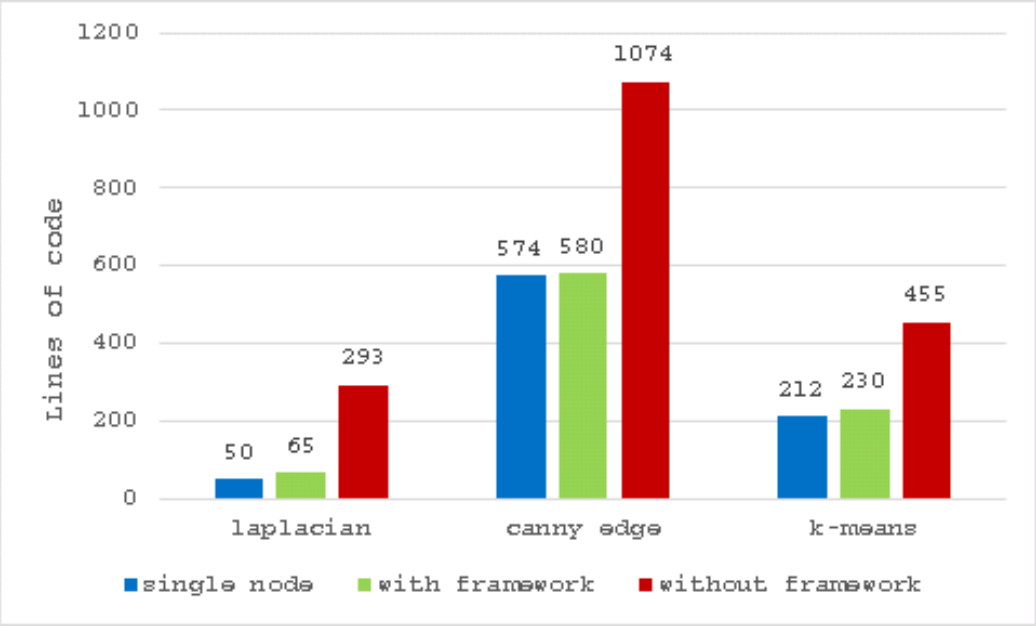

Figure 5.3: Comparing coding complexity of Laplacian filter, Canny edge detection and k-means clustering in different environments

duces complexity. Listing 2 demonstrates the mechanism for setting the image headers of processed images.

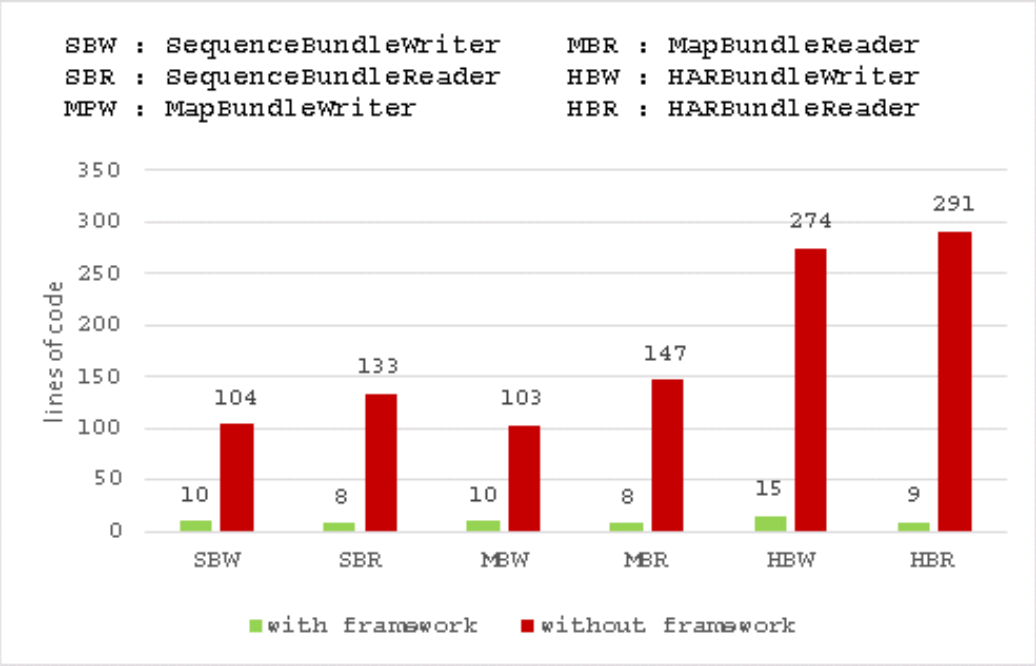

Figure 5.4: Comparing file writers and readers with and without using framework

Listing 5.1: Comparing FileWriter instance in java and SequenceBundleWriter instance in Hadoop Image Processing framework

```
//Sample File Writing in Java
File file = new File("test");
FileWriter fw = new FileWriter(file);
fw.append(val)
fw.close();

//Sample BundleFile Writing in Hadoop using Framework
BundleFile file = new BundleFile("test_bundle.seq");
SequenceBundleWriter sbw = new SequenceBundleWriter(file);
sbw.append(himage);
sbw.close();
```

Listing 5.2: Setting image headers for processed images using Hadoop Image Processing Framework

```
//input - HImage is sent as input
CannyEdge canny = new CannyEdge(input);
```

```
canny.process();

HImage himage = canny.getProcessedImage();

himage.setImageHeader(input.getImageHeader());
```

CHAPTER 6

FUTURE WORK

This paper has described our Hadoop Image Processing Framework for implementing large scale image processing applications. The framework is designed to abstract the technical details of Hadoop's powerful MapReduce system and provide an easy mechanism for users to process large image datasets. We provide software machinery for storing images in the various Hadoop file formats and efficiently accessing the Map Reduce pipeline. By providing interoperability between different image data types we allow the user to leverage many different open-source image processing libraries. Finally, we provide the means to preserve image headers throughout image manipulation process, retaining useful and valuable information for image processing and vision applications.

With these features, the framework provides a new level of transparency and simplicity for creating large-scale image processing applications on top of Hadoop's MapReduce framework. We demonstrate the power and effectiveness of the framework in terms of performance enhancement and complexity reduction. The Hadoop Image Processing Framework should greatly expand the population of software developers and researchers easily able to create large-scale image processing applications.

CHAPTER 7

CONCLUSION

In the near future we hope to extend the framework into a full-fledged multimedia processing framework. We would like to improve the framework to handle audio and video processing over Hadoop with similar ease. We also intend to add a CUDA module to allow processing tasks to make use of machines' graphics cards. Finally, we intend to develop our system into a highly parallelized open-source Hadoop multimedia processing framework, providing web-based graphical user interfaces for image processing applications.

BIBLIOGRAPHY

[1] P. Bajcsy, A Vandecreme, J. Amelot, P. Nguyen, J. Chalfoun, and M. Brady. Terabyte-sized image computations on hadoop cluster platforms. In *Big Data, 2013 IEEE International Conference on*, pages 729–737, Oct 2013.

[2] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, Nov 1986.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[4] David Duce. Portable network graphics (png) specification.

[5] Apache Hadoop. Hadoop, 2009.

[6] Stan Horaczek. How many photos are uploaded to the internet every minute? *http://www.popphoto.com/news/2013/05/how-many-photos-are-uploaded-to-internet-every-minute*, 2013.

[7] Yunpeng Li, D.J. Crandall, and D.P. Huttenlocher. Landmark classification in large-scale image collections. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 1957–1964, Sept 2009.

[8] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[9] Zhenhua Lv, Yingjie Hu, Haidong Zhong, Jianping Wu, Bo Li, and Hui Zhao. Parallel k-means clustering of remote sensing images based on mapreduce. In *Proceedings of the 2010 International Conference on Web Information Systems and Mining*, WISM'10, pages 162–170, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo. The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20:1297–1303, 2010.

[11] D. Moise, D. Shestakov, G. Gudmundsson, and L. Amsaleg. Terabyte-scale image similarity search: Experience and best practice. In *Big Data, 2013 IEEE International Conference on*, pages 674–682, Oct 2013.

[12] R. Pereira, M. Azambuja, K. Breitman, and M. Endler. An architecture for distributed high performance video processing in the cloud. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 482–489, July 2010.

[13] M.W. Powell, R.A. Rossi, and K. Shams. A scalable image processing framework for gigapixel mars and other celestial body images. In *Aerospace Conference, 2010 IEEE*, pages 1–11, March 2010.

[14] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

[15] Yinhai Wang, David McCleary, Ching-Wei Wang, Paul Kelly, Jackie James, DeanA. Fennell, and PeterW. Hamilton. Ultra-fast processing of gigapixel tissue microarray images using high performance computing. *Cellular Oncology*, 34(5):495–507, 2011.

[16] Brandyn White, Tom Yeh, Jimmy Lin, and Larry Davis. Web-scale computer vision using mapreduce for multimedia data mining. In *Proceedings of the Tenth International Workshop on Multimedia Data Mining*, MDMKDD '10, pages 9:1–9:10, New York, NY, USA, 2010. ACM.

[17] Tom White. The small files problem. *http://blog.cloudera.com/blog/2009/02/the-small-files-problem/*, 2009.

[18] Keith Wiley, Andrew Connolly, Jeff Gardner, S Krughoff, Magdalena Balazinska, Bill Howe, Y Kwon, and Yingyi Bu. Astronomy in the cloud: using mapreduce for image co-addition. *Astronomy*, 123(901):366–380, 2011.

[19] Chen Zhang, Hans De Sterck, Ashraf Aboulnaga, Haig Djambazian, and Rob Sladek. Case study of scientific data processing on a cloud using hadoop. In DouglasJ.K. Mewhort, NatalieM. Cann, GaryW. Slater, and ThomasJ. Naughton, editors, *High Performance Computing Systems and Applications*, volume 5976 of *Lecture Notes in Computer Science*, pages 400–415. Springer Berlin Heidelberg, 2010.

VITA

Sridhar Vemula

Candidate for the Degree of

Master of Science

Thesis: HADOOP IMAGE PROCESSING FRAMEWORK

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Masters degree with a major in Computer Science at Oklahoma State University in May, 2015.