Distributed Processing Of Large Remote Sensing Images Using MapReduce

A case of Edge Detection

Ermias Beyene Tesfamariam

Dissertation submitted in partial fulfilment of the requirements
for the Degree of *Master of Science in Geospatial Technologies*

Distributed Processing of Large Remote Sensing Images using MapReduce

A case of Edge Detection

Supervisor

Dr. Theodor Foerster
Institute for Geoinformatics
Universität Münster, Germany

Co-supervisors

Katharina Henneböhl
Institute for Geoinformatics
Universität Münster, Germany

and

Prof. Dr. Mario Caetano
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa, Portugal

February, 2011
Münster, North-Rhine Westphalia, Germany

Disclaimer

Author’s Declaration

I hereby declare that this Master thesis has been written independently by me, solely based on the specified literature and resources. All ideas that have been adopted directly or indirectly from other works are denoted appropriately. The thesis has not been submitted for any other examination purposes in its present or a similar form and was not yet published in any other way.

Signed: ______________________________

Date: 28 February 2011

Acknowledgements

I would like to express my special gratitude to my supervisors Dr. Theodor Foerster, Katharina Henneböhl, and Prof. Dr. Mario Caetano for their scientific guidance and critical reviews to bring the research into shape. I have gained a lot of scientific experience throughout the period of the thesis work.

I am grateful to the European Union and the Erasmus Mundus Consortium of Universitat Jaume I, Spain; Westfälische Wilhelms-Universität Münster, Germany; and Universidade Nova de Lisboa, Portugal for awarding me the scholarship to undertake this master's course.

I am pleased to extend my sincere thanks to Prof. Dr. Joaquin Huerta, Mrs. Dolores C. Apanewicz, Dr. Christoph Brox, and Prof. Dr. Werner Kuhn for their support and hospitality during my stay in Germany and Spain. I also thank my fellow classmates and friends for sharing their knowledge and giving me inspirations during the past eighteen months. I am also grateful to members of Institute for Geoinformatics for their warm friendship.

I also would like to thank Hadoop User groups and Hadoop mailing list for their detailed replies and advices.

Finally, my heartfelt gratitude goes to my parents Beyene Tesfamariam and Weini Zekarias who raised me with endless love and values.

Abstract

Advances in sensor technology and their ever increasing repositories of the collected data are revolutionizing the mechanisms remotely sensed data are collected, stored and processed. This exponential growth of data archives and the increasing user's demand for real-and near-real time remote sensing data products has pressurized remote sensing service providers to deliver the required services. The remote sensing community has recognized the challenge in processing large and complex satellite datasets to derive customized products. To address this high demand in computational resources, several efforts have been made in the past few years towards incorporation of high-performance computing models in remote sensing data collection, management and analysis. This study adds an impetus to these efforts by introducing the recent advancements in distributed computing technologies, MapReduce programming paradigm, to the area of remote sensing.

The MapReduce model which is developed by Google Inc. encapsulates the efforts of distributed computing in a highly simplified single library. This simple but powerful programming model can provide us distributed environment without having deep knowledge of parallel programming. This thesis presents a MapReduce based processing of large satellite images a use case scenario of edge detection methods. Deriving from the conceptual massive remote sensing image processing applications, a prototype of edge detection methods was implemented on MapReduce framework using its open-source implementation, the Apache Hadoop environment. The experiences of the implementation of the MapReduce model of Sobel, Laplacian, and Canny edge detection methods are presented. This thesis also presents the results of the evaluation the effect of parallelization using MapReduce on the quality of the output and the execution time performance tests conducted based on various performance metrics. The MapReduce algorithms were executed on a test environment on heterogeneous cluster that supports the Apache Hadoop open-source software. The successful implementation of the MapReduce algorithms on a distributed environment demonstrates that MapReduce has a great potential for scaling large-scale remotely sensed images processing and perform more complex geospatial problems.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API	Application Programming Interface
CCA	Common Component Architecture
CDH2	Cloudera Hadoop 2
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFS	Distributed File System
DN	Digital Number
EC2	Amazon Elastic Compute Cloud
ETM+	Enhanced Thematic Mapper Plus
EOSDIS	Earth Observation System Data and information System
EROS	Earth Resources Observation and Science
GEOTIFF	Geographic Tag Image Input Format
GFLOPs	Giga Floating Point Operations Per Second
GFS	Google File System
GSFC	Goddard Space Flight Center
G-POD	Grid Processing on Demand
GPU	Graphic Processing Unit
GRASS	Geographic Resources Analysis Support System
HDFS	Hadoop Distributed File System
I/O	Input Output
JDAF	Java Distributed Application Framework
LAN	Local Area Network
MPI	Message Passing Interface
MPMD	Multiple Program Multiple Data
MR	MapReduce
NASA	National Aeronautic Space Agency
OpenCL	Open Compute Language
OpenMP	Open Multi-processing
QGIS	Quantum GIS
SARA	Synthetic Aperture Radar Atlas

SPOT	Satellite Pour l'Observation de la Terre
SPMD	Single Program Multiple Data
TIFF	Tag Image Input Format
TM	Thematic mapper
URI	Uniform Resource Identifier
USGS	United States Geological Survey
WAN	Wide Area Network

Chapter 1 Introduction

The research carried out in this thesis covers the topic of high performance computing in the field of remote sensing to address the computational requirement for processing of large remote sensing images. This chapter discusses the background and rationale of the research in relation to previous research efforts. Based on the rationale and identified research problems the objectives and the research questions that is research addresses are presented. Finally, the last section gives an overview of the organization of the thesis.

1.1 Background

Nowadays, with the wide support for spatial data sharing, more and more remotely sensed images are becoming publicly available where users can download them freely. Analyzing of these remotely sensed images requires high speed network connection for downloading them and powerful storage and computing resources for local processing [*Shen et al.*, 2006; *Teo*, 2003]. Therefore it might be more efficient to process satellite images remotely, preferably on a high-performance computer server that are close to the data servers [*Hawick et al.*, 2003]. However, as satellite images continue to increase rapidly in size and complexity due to increase in spatial and temporal resolution, it becomes difficult to seamlessly access and process them using the state-of-the-art services where processing is done on a stand-alone, centralized processing server.

In this context, infrastructures that employ distributed computing resources can be a potential to provide the required computational power for scaling data processing in remote sensing applications [*Aloisio and Cafaro*, 2003; *P. Votava et al.*, 2002]. Distributed computing infrastructures are suitable to store large-scale data like satellite images that have to be written only once and read frequently. The systems in distributed infrastructure can be and heterogeneous and do not need to be dedicated processing resources where their primary purpose can be other tasks and that makes it a low-cost supercomputing resource.

Within the framework of the aforementioned technologies the emerging distributed computing paradigm, MapReduce programming model, provides a potential for large-scale processing of satellite images on clusters of commodity computers. MapReduce is highly simplified distributed programming model for easy programming of applications that aim to process huge datasets in a parallel mode [*Dean and Ghemawat*, 2008]. MapReduce based large scale and high performance distributed services has gained a wide support from industrial sector through the Apache Hadoop open-source implementation and is recently gaining attention from the scientific community. The following two sections will briefly introduce the concept of Distributed computing technologies and satellite images processing, while the detail of parallel and distributed computing systems and MapReduce programming basics is covered in Chapter 2.

1.1.1 Distributed Computing Technologies

Distributed computing is a type of computing that deals with applications that run simultaneously on distributed systems that communicate through computer network in order to solve massive computational problems. Tanenbaum and Steen have defined a distributed system as "*a collection of independent computers that appears to its users as a single coherent system*" [*Tanenbaum and Steen*, 2007]. The main driving force for the development of distributed computing is the requirement for high-performance computing resources for solving massive scientific computational problems, which led to the idea of dividing the problems into smaller tasks to be processed in parallel across multiple computers [*Allan et al.*, 2006]. The development of computing and high-speed network infrastructures in the past few years has also made it possible for distributed computing systems to provide a coordinated and reliable access to high performance computational resources.

Distributed computing can be classified broadly into types. The first is high-performance computing on parallel heavy-duty systems that provide access to large-scale computational resource and are common for computationally intensive applications[*Silva*, 2006]. These resources involve high investment cost and are usually limited at few institutions and research centers. Another distributed computing solution is that is becoming increasingly popular recently is to perform computations on clusters of low-cost commodity computers connected over high speed network. The advances in high-speed network communications and its inexpensive availability made this trend more practical over expensive parallel supercomputers.

MapReduce programming model harnesses most of the requirements of distributed computing while hiding the intricate system-level details and providing highly-simplified abstractions [*Ghemawat et al.*, 2003; *Pike et al.*, 2005]. MapReduce is designed to enable automatic parallelization and distribution of large-scale data computations to achieve high-performance on clusters of low-cost commodity servers [*Dean and Ghemawat*, 2008; 2010]. This scale-out approach is perhaps the most notable feature of the MapReduce paradigm which makes it easy to develop highly scalable parallel applications [*Lin and Dyer*, 2010]. Several tests done by companies such as Google, Yahoo, New York Times etc demonstrated through their MapReduce implementation that the MapReduce programming model can achieve world record performance [*White*, 2009].

The MapReduce programming model is also a highly transparent framework as it effectively hides the details of fault-tolerance, data distribution, replication and load balancing while still able to handle failures and redundancy automatically [*Dean and Ghemawat*, 2008]. Since, many data-intensive computations do not require high processing power it is preferable for the computations to be done on the data side which saves transferring of massive datasets over the network. With this philosophy as its core principle, MapReduce is therefore creatively built on the top of a distributed file system, which takes advantage of data locality to perform the computation close to the data server [*Dean and Ghemawat*, 2010]. This is discussed in detail in section 2.4. These qualities of the MapReduce programming model make it an excellent candidate for processing of massive datasets such as satellite images where both computation and memory requirement are often expensive.

1.1.2 Distributed Remote Sensing Image Processing

Remote sensing technologies especially satellite images have a wide range applications in many areas including meteorology, global change detection and monitoring, minerals and oil exploration, natural resources management, agriculture, environmental assessment and monitoring, disaster and relief, military surveillance etc [*Schowengerdt*, 2007]. Those satellite Images are exponentially growing in size and complexity as the spatial, spectral and temporal resolution of the satellite sensors continue to develop rapidly. For example NASA's Earth Observing System Data and Information System (EOSDIS) ground stations currently receive in excess of 2 terabytes of satellite data transmitted daily from various satellite missions and an excess of 4 petabytes of earth science data products are currently archived [*Behnke et al.*, 2005; *NASA*, 2007]. This growth in acquisition technology has a put very important impetus for the amount and quality of the satellite images that are available to the geosciences community.

Along with this development comes the challenge of managing those massive satellite image databases for storage, access, processing and distribution in order to make it easily available to the users [*Petr Votava et al.*, 2002]. The computational resources needed for processing such large volume of satellite images often exceeds those available in stand-alone centralized servers and can't satisfy today's real- and near real-time requirement for image processing. This leads to the need for looking of other solutions such as grid or cluster computing.

A great deal researches and projects exist concerning parallel and distributed processing of remote sensing data. A prominent example is the Beowulf Cluster of NASA's Goddard Space Flight Center (GSFC) which uses commodity computers to build a cluster for remote sensing data processing calculations that exceeds a peak performance of 2457.6 GFLOPs [*Plaza and Chang*, 2008; *Sterling et al.*, 1995]. Another project which involves distributed systems framework implementation is the Common Component Architecture (CCA) which is used as a plug-and-play environment for construction of climate, weather, and ocean applications. It is implemented through the Ccaffeine framework to support single program multiple data (SPMD) and multiple program multiple data (MPMD) programming models [*Allan et al.*, 2006].

Among the many projects that involve the use of grid computing technology for high performance satellite data processing is the European Space Agency's Earth Observation Grid Processing on Demand (G-POD) that implements the layered approach based on the Grid-ENGINE which interfaces the application layer with different Grid middleware [*Cossu et al.*, 2009]. SARA/Digital Puglia (Synthetic Aperture Radar Atlas), is also another remote sensing application that demonstrates the application of grid technologies and high performance computing to build dynamic earth observation systems for the management and processing of large amount of satellite data [*Aloisio and Cafaro*, 2003]. This grid implementation is based on Globus Toolkit grid middleware and enables users to browse the available satellite data in distributed repositories through sophisticated resource brokers and start on-demand parallel processing on remote computational grids.

Votava et al (2002) have also demonstrated the development a flexible and extensible java based distributed framework, the Java Distributed Application Framework (JDAF), specifically designed for parallel and distributed processing of remote sensing data with flexibility and performance as its main goal. Reliability can also be included as part of this framework considering the reputable fault-tolerance capability of Java programming language. However, the authors do not believe that the Java programming language is matured enough for high performance computing of large amounts of remote sensing data.

The MapReduce programming model is a relatively new paradigm in the area of distributed computing which has only gained popularity in the area of academia recently and therefore only a handful of researches are available especially in the area of Geographic Information Systems and Remote Sensing. Rather much of the input has been from the industry sector. A noteworthy work has been done by Winslett et al. on parallel processing of spatial datasets using the MapReduce programming model [*Winslett et al.*, 2009]. The research focused how the MapReduce framework can be applied for massive parallel processing of both vector and raster data representations and it achieved a reasonable performance using MapReduce and the its open-source implementation – the Hadoop framework. Another interesting work that has been done recently is the adoption of image coaddition algorithms to MapReduce for the purpose of astronomical images processing where they used Hadoop's API to implement their algorithms [*Wiley et al.*, 2010]. [*Golpayegani and Halem*, 2009; *Lv et al.*, 2010] have also made an effort to implement some satellite image processing algorithms Hadoop's MapReduce model but they did not use the image files as a raw input to be processed by MapReduce, rather the satellite images were first converted into text format then to binary format before being processed in Hadoop environment. It is therefore obvious that the preparation of data will consume much of the computation time than the actual processing of the images especially with large multi dimensional images [*Golpayegani and Halem*, 2009]. In this study it is proposed to extend Hadoop's file management API so that it can take images as an input and that might significantly reduce the execution time.

1.2 Research Objectives

The core aim of this study is to investigate how the processing of large satellite images can benefit from distributed computing environment through massive parallelization. The study evaluates different parallelization approaches of the processing algorithms for efficient and scalable computations with primary focus on massive data parallelization. In this study the MapReduce programming model is proposed as a framework for parallel processing of remote sensing images and its features such as fault-tolerance, scalability, replica management and distributed output checking that make this approach suitable are explored. The research addresses how the MapReduce programming model can be applied and tuned to process large satellite images. The research implements various spatial transformation algorithms for enhancement and detection of edges from multispectral Landsat satellite images as a prototype. The research also experimentally evaluates and quantifies the performance of the algorithms in their execution time, scalability, and quality of output images for the different use-case scenarios on normal commodity hardware. The effort is mainly directed towards the minimization of the computation time of the algorithms and to describe and discover and describe the optimal data parallelization and communication scheme for heterogeneous network of computers.

1.3 Research Questions

The hypothesis presented in this master's thesis is that processing of large satellite images can benefit from distributed computing and the following are the main research questions of this study:

1. Can performance of large satellite image processing be augmented through implementation on a distributed environment?
2. What are the requirements to implement distributed processing of satellite images using the MapReduce programming model?
3. What is the performance of the MapReduce processes compared to sequential processes?
4. How do the different edge detection algorithms perform in a distributed environment?
5. Which data partitioning and communication scheme is preferred in order to be executed concurrently?

1.4 Thesis Structure

A brief outline of the chapters that follow in this thesis is discussed below. These chapters are structured to cover the entire spectrum of the research (Figure 1-1).

Chapter2: Fundamentals

This chapter reviews the state-of-art techniques of processing of remotely sensed images their mathematical properties and a prototype of edge detection methods is presented. Some main ideas are drawn from these concepts in order to develop an application that exploits the ease of the MapReduce without compromising the quality of the final product. This chapter also reviews the common distributed systems architecture and the current technology and trend of distributed systems. It also covers fundamentals of the MapReduce programming paradigm and the existing features of Apache Hadoop Framework and also the key factors for a successful distributed implementation using Hadoop environment.

Chapter 3: System Design

This chapter discusses the design considerations and requirements for successful implementation of remote sensing image processing algorithms on distributed environment using MapReduce. This chapter harnesses the image processing component to the implementation component of this study by following standard application design frameworks in order to achieve the desired performance.

Chapter 4: MapReduce Implementation of Satellite Image Processing

In this chapter the actual implementation of the satellite image processing algorithms of the prototype of edge detection methods based on the framework designed in the previous chapter and the underlying implementation details are described. This includes the description of how the edge detection algorithms are adopted to the MapReduce model and the various optimization strategies used are explained.

Chapter 5: Performance Evaluation

This chapter starts with the description of the test environment setup and the physical configurations followed by the description of the dataset that is going to be used in this test. It then presents the various experiments conducted and evaluation results of the prototype algorithms on the distributed environment. The benchmarking with respect to the sequential methods and strengths and limits of MapReduce found during the experiments are described. The experiments also evaluate the performance of the algorithms in terms of dependency to size of datasets and data splitting mechanisms.

Chapter 6: Conclusion and Future works

This final chapter concludes the work by providing a summary and a short outlook of the chosen approach. It includes recommendations and further trends on the application of MapReduce for distributed processing of large remote sensing images.

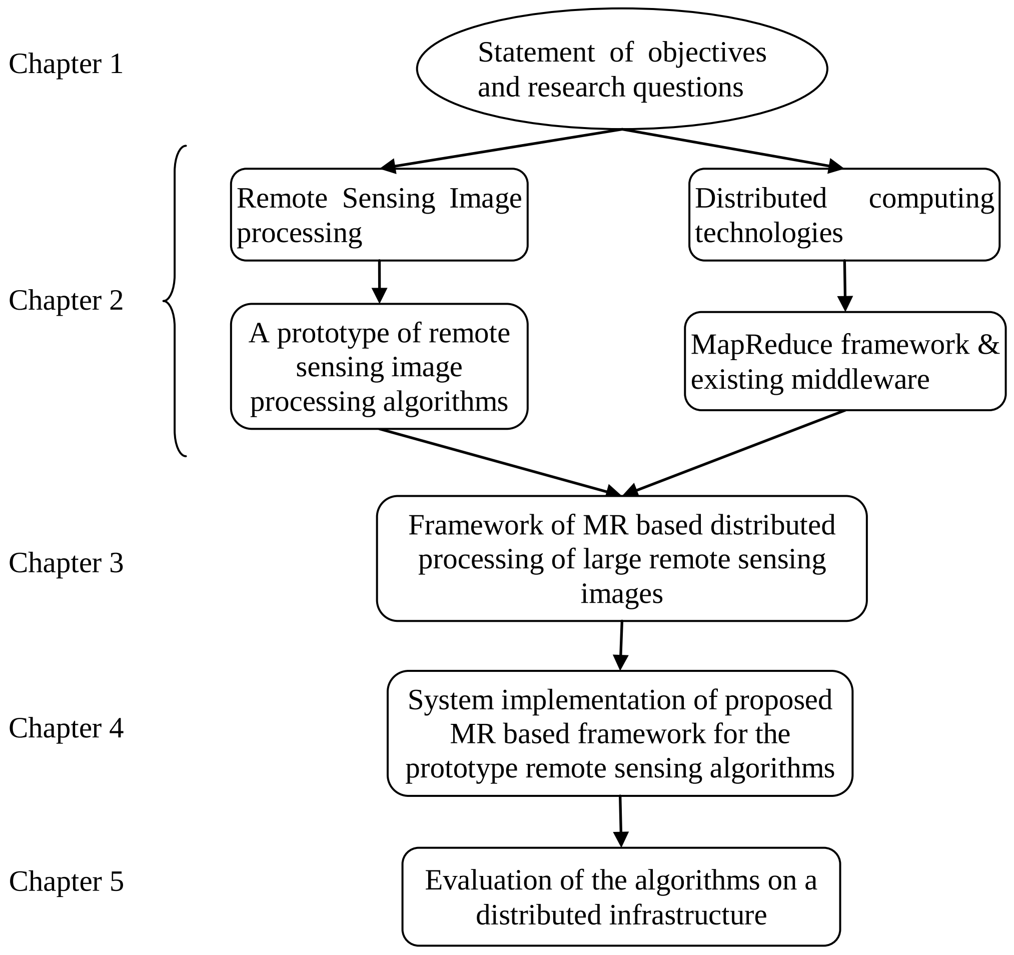

Chapter 1
Statement of objectives and research questions
Chapter 2
Remote Sensing Image processing
Distributed computing technologies
A prototype of remote sensing image processing algorithms
MapReduce framework & existing middleware
Chapter 3
Framework of MR based distributed processing of large remote sensing images
Chapter 4
System implementation of proposed MR based framework for the prototype remote sensing algorithms
Chapter 5
Evaluation of the algorithms on a distributed infrastructure

Figure 1-1 Thesis structure

Chapter 2 Fundamentals

This chapter introduces the basic concepts of distributed remote sensing image processing which are essential for understanding this study and can roughly be grouped in to two main parts. The two sections give a general overview of remote sensing data characteristics and the state-of-art remote sensing image processing techniques respectively. A theoretical background of the edge detection methods which are going to be implemented as a prototype in this thesis are also presented in the second section. In the second part, the basic ideas of distributed systems and parallel computing is discussed in the third section followed by a discussion on the detailed overview of a special kind of distributed system framework, the MapReduce programming model, in the subsequent section. Finally the open-source MapReduce implementation, Apache Hadoop software, is presented in the last section. Some main ideas are drawn from these concepts in order to develop an application that exploits the ease of the MapReduce without compromising the quality of the output product.

2.1 Remote Sensing Data

Over the past few decades, remote sensing data especially acquired by satellite sensors, have been playing a major role in studying the earth's surface efficiently and consistently for a wide range of applications. One of the chief advantages of these remote sensing data is its availability in digital format in the form of two-dimensional images which allow us to easily process and manipulate them on computers [*Richards and Jia*, 2006]. Cost effectiveness, repetitive coverage and wide-ranging applicability are also some of the other advantages of remote sensing data compared to other forms of data collected through methods such as ground-based acquisition methods.

Remote sensing data are usually represented in raster format as discrete three dimensional data space with the two coordinates representing spatial extent and the third spectral wavelength recorded as Digital Number (DN). The characteristics of a remotely sensed data principally depend on the nature of the corresponding sensor that determines the image resolution which can mean the spectral resolution, spatial resolution, radiometric resolution or temporal resolution [*Schowengerdt*, 2007].

The spatial resolution specifies the dimension on the earth's surface that is covered by a single pixel in the acquired image and the higher spatial resolution of sensors the more detailed the acquired image and the smaller is the area covered by a single pixel [*Lillesand and Kiefer*, 2001]. The other significant characteristic of a remote sensing sensor is the spectral resolution used in the image acquisition process. A sensor's spectral resolution specifies the number of bands that a particular sensor can acquire.

Resolution can also mean the radiometric resolution of the sensor, usually expressed as bits per pixel, which indicates how the continuous data measurement is quantized into binary numbers. For example, sensors Thematic Mapper (TM) and SPOT have a radiometric resolution of 8 bits per pixel while Aqua MODIS and most hyperspectral sensors have 12 bits per pixel [*Schowengerdt*, 2007]. These three fundamental characteristics determine the amount of data that is generated by a sensor and understanding them is significant for the proper design of image processing algorithms.

If we look for example at the U.S. Geological Survey's (USGS) Earth Resources Observation and Science (EROS) data center where remote sensing data from various satellite sensors is preprocessed, archived and distributed as public domain. The Landsat data alone comprises more than 1 petabyte of archive composed of about 2.34 million scenes which is growing by 300 gigabytes daily [*NASA*, 2010]. The Landsat program began in 1971 with the launch of Landsat1 and was followed by a series of more advanced satellites until Landsat 7 was launched in 1999.

The Landsat7 satellite carries the Enhanced Thematic Mapper Plus (ETM+) instrument which comprises eight bands [*Parkinson et al.*, 2006]. Bands 1 to band 7 have a spatial resolution of 30meters while the spatial resolution of the thermal band (band 6) has increased from 120meters of Landsat 4 and 5's Thematic Mapper to 60meters. A visible panchromatic band of 15meters resolution was also introduced on ETM+ as an eights band (Table 2-1). The Landsat7 takes images of the Earth's surface by dividing it into 28,892 path/row scenes each scene of 183km X 170km coverage on the ground, and repeats any given area of the planet once every 16 days [*Goward et al.*, 2001]. The radiometric resolution all the bands of ETM+ is 8 pixels per bit and spatial resolution 30 meters except for the thermal band 6 and 8 where the resolution is 60 and 15meters respectively. From this information we can estimate the images generated by one scene of Landsat ETM+ have an uncompressed size of more than 280 Mbytes.

Table 2-1: Landsat Satellites band characteristics [*NASA*, 2010].

Satellite	Sensor	Bandwidths (µm)	Bits per pixel	Resolution (m)
LANDSATs 4-5	MSS	(4) 0.5 – 0.6	7	82
		(5) 0.68 – 0.7	7	82
		(6) 0.7 -0.8	7	82
		(7) 0.8 – 1.1	6	82
	TM	(1) 0.45 – 0.52	8	30
		(2) 0.52 – 0.60	8	30
		(3) 0.63 – 0.69	8	30
		(4) 0.76 - 0.90	8	30
		(5) 1.55 - 1.75	8	30
		(6) 10.4 – 12.5	8	120
		(7) 2.08 – 2.35	8	30
LANDSAT 7	ETM$^+$	(1) 0.45 – 0.52	8	30
		(2) 0.52 – 0.60	8	30
		(3) 0.63 – 0.69	8	30
		(4) 0.76 - 0.90	8	30
		(5) 1.55 - 1.75	8	30
		(6) 10.4 – 12.5	8	60
		(7) 2.08 – 2.35	8	30
		(8) PAN 0.50 to 0.90	8	15

2.2 Remote Sensing Image Processing

Remote sensing image processing is concerned with the extraction of measurements and information from images using various algorithms and usually involves mathematical treatment to analyze complex scenes. Image processing algorithms may be categorized into two kinds of transformations according to the space in which they operate: spectral transformation and spatial transformation [*Schowengerdt*, 2007]. Transformations based on the image data space (such as contrast enhancement and histogram equalization) that alter the spectral space of an image are called spectral transformations and these techniques are commonly applied in radiometric enhancements. These transformations are characterized by generating new pixel values based on mathematical operation on the existing pixel value and doesn't depend on the neighboring pixels therefore they are sometimes referred as point or pixel-specific operations [*Richards and Jia*, 2006]. While transforms purely on the image plane modify the spatial information of a pixel by computing the new pixel value based on the surrounding pixels. Some of these transforms are local in nature (e.g. Convolution) which use only local image information within a small neighborhood of a pixel, while other spatial transforms use global spatial information of the image (e.g. Fourier Transform) to compute the resultant images. There are also image transformations that use both the global and local information of an image such as the Wavelet transform and the Gaussian and Laplacian pyramids [*Schowengerdt*, 2007].

2.2.1 Convolution based image transformation

Convolution operation is one of the fundamental techniques in remote sensing image analysis and is most commonly used in image smoothing and blurring, edge detection and morphological processing [*Richards and Jia*, 2006]. Convolution operators use a single grey-scale image as an input and generate another grey-scale image as output. The convolution operator uses a moving kernel over the input image and it modifies the pixels that fall within that kernel. The kernel is usually positioned with its center at the pixel to be processed, and then this kernel is shifted one pixel along the row of the image to process the next pixel. Upon reaching the end of the row of the image the kernel moves down to the next row and the process is repeated (Figure 2-1). Mathematically the output *g* of a two dimensional discrete convolution operation of an impute image *f* can be represented as a weighted sum of pixels within a moving kernel *w* of size $N_x \times N_y$ [*Schowengerdt*, 2007].

$$g_{ij} = \sum_{m=0}^{N_x-1} \sum_{n=0}^{N_y-1} f_{mn} w_{i-m,j-n}$$

Equation 2-1

Where *m* and *n* are the rows and columns of an input image respectively.

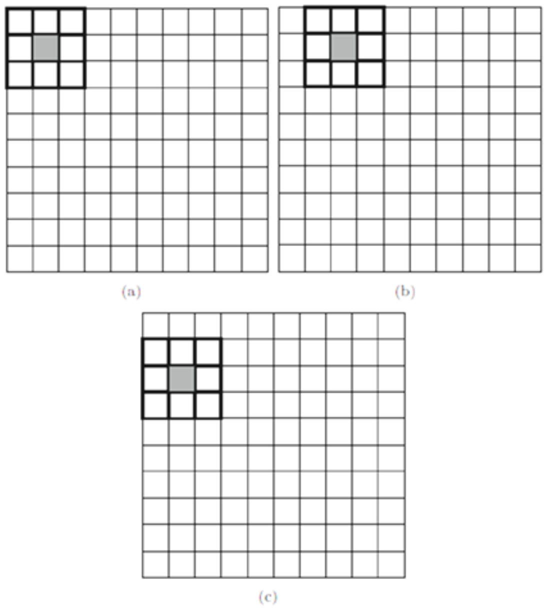


Figure 2-1: A Convolution operation for image filtering [*Schowengerdt*, 2007].

Generally, when in applying convolution operations using different sizes of kernels it is usually difficult to directly apply filtering on the border images because those pixels are the last rows and columns of the input image, they do not have neighborhood pixels on one of their sides. This is especially cumbersome when we try to apply parallel processing of images by splitting the input image into smaller chunks. There are several techniques described by [*Schowengerdt*, 2007] to compute those border pixel values to make the output image the same size as the input image which are discussed in detail in Ch. 4.

The discrete convolution operation is usually computationally expensive as it consist a set of multiplications and additions for each pixel output. The calculation of a single pixel involves not just the input pixel only but also information from the neighboring pixels by calculating multiplying each entry pixel of the input pixel with the respective kernel [*Chiarabini and Yen*, 1998]. To calculate a single pixel, the number of multiplications needed is equal to the size of the kernel. For example, a convolution operation an image of 1024×1024 size by a kernel of 3×3 dimension will involve $(3\times1024)^2$ multiplications [*Richards and Jia*, 2006].

Many image enhancement and edge detection algorithms in digital image signal processing applications usually use convolution operations with wider kernels which usually depend on computationally expensive code sections involving repetition of sequences of operations [*Bräunl*, 2001]. These applications are generally are suitable for fully parallel implementation to improve the overall exaction time of the filter operations.

2.2.2 Edge Detection

The delineation and extraction of features from remote sensing image is an important task useful for a wide range of application fields such as object recognition, image segmentation, data compression, land-water border delineation etc. Edges in an image are signified by a significant image intensity change which represents important object features and boundaries between objects in an image. Edge detection therefore has an ubiquitous interest in the field of image processing and is a fundamental pre-processing stage of feature extraction from remote sensing images [*Heath et al.*, 1998]. It is mostly done by applications that depend on local operators using convolution filters which can be a very slow process for several instances of processing large images.

There are many ways to perform edge detection using the convolution operation. However, the majority of the different edge detection methods may be grouped into two categories, first-order derivatives and second- derivatives [*Drewniok*, 1994]. The first-order derivative, commonly called gradient method, denotes methods that involve filters such as Roberts, Prewitt and Sobel operators and detects the edges by searching for the maximum and minimum values in the first derivative of the input image Fig 2. The second-order derivative, the Laplacian method, searches for zero crossings in the second derivative of the image to find edges [*Marr and Hildreth*, 1980]. If we consider an edge has one-dimensional slope of change in intensity and calculating the derivative of the original image can highlight the region of high intensity change (Figure 2-2). All these edge detection algorithms involve a single or multiple convolution filter kernels that can be of different sizes and the coefficients of these filter kernels always sum up to zero.

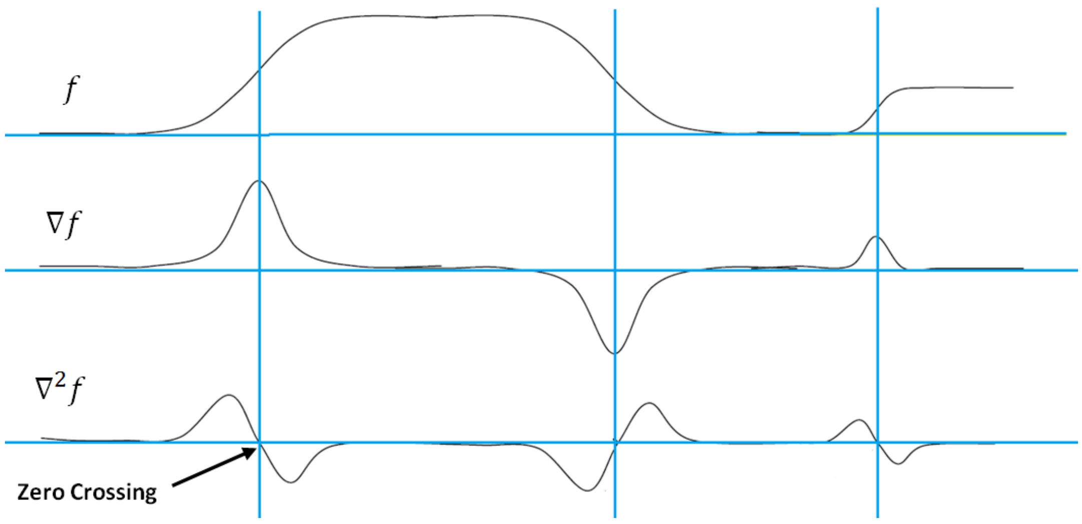


Figure 2-2: 1st spatial derivative and 2nd order spatial derivative of 1-D signal.

Sobel edge detection

The Sobel operator applies a 2-D spatial gradient measurement on an image represented by the equation below [*Kittler*, 1983]. It is typically used to find the approximate absolute gradient magnitude at each point in an input grayscale image. The discrete representation of the Sobel operator can be approximated by a pair of 3x3 convolution kernels (Figure 2-3), one estimating the gradient in the x-direction (columns) and the other estimating the gradient in the y-direction (rows). The G_X kernel highlights the edges in the horizontal direction while the G_Y kernel highlights the edges in the vertical direction [*Fisher et al.*, 1996]. The magnitude |G| of both outputs detects edges in both directions which is the brightness value of the output image.

$$G_x = \frac{\partial f}{\partial x} \quad G_y = \frac{\partial f}{\partial y}$$

Equation 2-2

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 2-3: A horizontal and vertical Sobel filter kernels

$$|G| = \sqrt{G_x^2 + G_y^2}$$

Equation 2-3

Laplacian edge detection

The Laplacian edge enhancement method is a second-order derivative of an image and it is applied by convolving the non-directional Laplacian filter. The second order derivative an edge will have a zero crossing in the region where there is the highest change in intensity [*Wang*, 2009]. Therefore the location of the edge can be obtained by detecting the zero-crossings of the second-order derivative of the image and this is known as Laplacian filter which is an effective detector for non-sharp edges where the pixel intensity level change over space slowly [*Torre and Poggio*, 1986]. A single filtering kernel of different sizes (e.g. 3x3, 5x5, 7x7, etc.) that has low values (usually negative) in the middle of the kernel surrounded by positive values can be used as Laplacian edge enhancement (Figure 2-4). Because the Laplacian is an approximation of the second-order derivative of an image preserving the high frequency components, it is very sensitive to noise and therefore it is usually applied to an image that has first been smoothed using the Gaussian filter in order to suppress noises in the image [*Fisher et al.*, 1996].

0	1	0
1	-4	1
0	1	0

0	1	0
1	-8	1
0	1	0

0	0	1	0	0
0	1	1	1	0
1	2	-16	2	1
0	1	2	1	0
0	0	1	0	0

Figure 2-4: Commonly used Laplacian filters.

Canny edge detection

The Canny edge detection is considered as the optimal and standard edge detector [*Drewniok*, 1994]. This multi-step method which was developed by

[*Canny*, 1986] with the aim to develop an optimal algorithms that satisfies three main criteria. The first one is good edge detection by maximizing the signal-to-noise ratio meaning the method should detect edges to the maximum possibility but with low probability of detecting edges falsely. The second criterion is that detected edges should be as close as possible to the real edges. The third criterion is to have minimal number of response and edges should not be detected more than once [*Canny*, 1986]. To satisfy these criteria, the algorithm can be performed in the following four separate steps.

1. *Smoothing*: this step involves smoothing the image using a Gaussian filter to suppress the noise and the degree of smoothing is controlled by the standard deviation σ of the Gaussian filter.

   $$g(x,y) = G_\sigma(x,y) * f(x,y)$$

   Equation 2-4

   Where * denotes convolution and

   $$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{[-\frac{x^2+y^2}{2\sigma^2}]}$$

   Equation 2-5

   The two dimensional Gaussian kernel can be made first by independently convolving a one dimensional Gaussian kernels in the horizontal and vertical directions and then multiplying them which makes it more suitable for computation.

   $$G_\sigma(x,y) = G_\sigma(y) * \; G_\sigma(y)$$

   Equation 2-6

2. *Gradient magnitude and direction*: the gradient magnitude of the image is computed using any of the gradient operators (e.g. Sobel, Roberts, Prewitt) by using Equation 2-3. And the direction of the gradient is calculated using the equation below and the angle is rounded to the closest 0, 45, 90 or 135 degree angle.

   $$\theta(x,y) = \; tan^{-1}\frac{G_x}{G_y}$$

   Equation 2-7

3. *Non-maximum suppression*: From the image gradient the local maxima are identified as edges based on its direction and the non-maximum image intensities are suppressed.

4. *Thresholding by hysteresis*: assuming that true edges are continuous, thresholding is done with hysteresis which requires upper and lower threshold. The upper threshold selects those edges that are strong. These edges are used then to trace the weak edges while applying the lower threshold to suppress those edges that weak and not connected to the strong edges. After completion of this process, the final image output becomes a binary format.

There are two parameters in the Canny edge detection method that affect the effectiveness of the algorithm and also the computation time. The first one is the size of the Gaussian smoothing filter which set by the parameter standard deviation σ of the Gaussian function and the greater the filter size the computationally expensive the process and the stronger the smoothing effect on the image. The second parameter is the high and low thresholds used for hysteresis and these thresholds are a function of the smoothing filter parameters, properties of the first- or second-order derivatives filter and the edge characteristics [*Ziou and Tabbone*, 1993].

2.3 Distributed Systems

The main motivation for distributed computing is low-cost resource sharing for speedup of large-scale computational problems and this is usually done through interconnection of autonomous and geographically distributed computing resources to provide a reliable access to high-end computation [*Foster et al.*, 2001; *Joseph and Fellenstein*, 2004]. Some of the main characteristics of these distributed systems are: scalability, fault-tolerance, reliability, transparency, concurrency etc [*Coulouris et al.*, 2005; *Kshemkalyani and Singhal*, 2008]. The most common distributed computing systems are cluster computing and grid computing [*Tanenbaum and Steen*, 2007]. There distinction between is these two systems is fuzzy but generally cluster computing represents to computing systems with tightly coupled commodity computing nodes with the high-speed local area network configurations. While in grid computing the systems are usually heterogeneous computers with different network configurations, administrative rights, operating systems etc [*Tanenbaum and Steen*, 2007].The main reason behind the popularity cluster computers is the high prices of supercomputers, the exponential growth of low cost processors, and the availability of high-speed network connections [*Abbas*, 2007]. Computer clusters are usually used for parallel programming for compute- or data-intensive tasks. Here parallel computing means any computing resource that runs a given task in parallel therefore even involves multi-processor machines.

2.4 MapReduce Programming Model

MapReduce is a parallel programming model for processing of large datasets on clusters of low-cost commodity computers [*Dean and Ghemawat*, 2008]. The model, originally introduced by Google, is developed on a well known principle of “divide-and conquer model” in parallel programming and has since then revolutionized the area of distributed computing. It is especially designed to process very large datasets on large cluster of low-cost commodity computers with unreliable communication and with the assumptions that storage is cheap and network communication is expensive. MapReduce is developed with a basic principles such as: the Scale-out (cluster of large number of desktops) rather than scaling up (few powerful supercomputers) solutions with a viewpoint of minimizing investment cost; moving computations to data to take advantage of data locality; fault-tolerance and reliability through data replication and distributed filing system; and hiding system-level details through clean abstractions and automatic parallelization and distribution [*Lin and Dyer*, 2010]. In MapReduce by default the codes for dividing the work, controlling the progress and merging the output is hidden from the application developer inside the framework. This abstraction comes at the expense of control over data flow and other the processes compared to grid computing APIs such as MPI [*White*, 2009]. Therefore, not all applications can be easily implemented using MapReduce. MapReduce is suitable for algorithms do not require global synchronization as the map and reduce tasks run in isolation on the computing nodes. However, algorithms such as clustering, machine learning and neural-network algorithms are challenging to implement in MapReduce as they depend on shared global state for intermediate communication during processing [*Lin and Dyer*, 2010]. This research explores how much applicable is MapReduce to remote sensing image processing algorithms and what are the bottlenecks to transform sequential image processing algorithms to parallel mode.

2.4.1 Basic concepts of MapReduce

The MapReduce model, inspired by functional programming languages, allows doing the computation of two distinct phases: the map and the reduce phase [*Dean and Ghemawat*, 2008]. The input data is split to be represented as a set key-value pairs which are processed in a parallel mode by the map phase to generate another intermediate key-value pairs (Listing 2-1). Those intermediate values are then processed produce the overall result. The map and reduce phase occur in an explicitly sequential mode where a reducer on an input can not start until the map phase finishes, but within these phases are executed in parallel within themselves. The application programmer specifies the map and reduces functions in the following representations:

Listing 2-1: Definition of map and reduce functions.

```
map (in_key, in_value) ->
                (out_key, intermediate_value) list

reduce (out_key, intermediate_value list) ->
                out_value list
```

The map function takes a list of values and a processing function as input. The processing function is applied to every list element and an intermediate list of processed results is returned as key-value pairs. These are intermediate results are pushed to the reducers which also receives them as key-value pairs and combines the values list to a final output result. One of the fundamental properties of MapReduce framework is locality, i.e., it tries to execute map tasks on the same machine as the physical location of the data. This property greatly reduces communication over the network.

2.4.2 Parallelism in MapReduce

The map functions run in parallel independent of each other and in isolation creating different intermediate values from different input data sets. These reduce functions also run in parallel, each working on a different output key of the intermediate values. The MapReduce framework takes care of supplying the mappers and reducers with the necessary input data and manages the exchange of intermediate results between the mappers and the reducers. But the reduce phase can't start until a map phase of an input split completely finishes [*White*, 2009].

2.4.3 Fault tolerance

Fault tolerance is one of the fundamental requirements of distributed systems [*Coulouris et al.*, 2005; *Cristian*, 1991; *Tanenbaum and Steen*, 2007], and a fault-tolerant system must be able to transparently handle failures without affecting the performance and quality of results as much as possible. Since, MapReduce is designed to work on commodity servers, it is designed with the expectation that failures occur frequently [*Lin and Dyer*, 2010]. In MapReduce framework of Hadoop for example the master server automatically detects failures in worker nodes and re-executes the task on completed map tasks or waiting reducers. It also efficiently manages failures caused by corrupted files by skipping them after they fail to execute several times[*White*, 2009]. The features enable MapReduce to provide fine-grained fault tolerance where partial failures in a job do not interrupt the overall progress of the job [*Dean and Ghemawat*, 2010]. This is particularly important if the input data are independent of each other and a specific job with multiple data inputs needs a lot of time to complete as we can achieve partial results even if the whole job did not complete successfully.

2.4.4 Comparison to other systems

MapReduce is not the first model to adopt the idea of data-intensive distributed processing; most of the issues raised now by MapReduce have been dealt to some extent efficiently by other models. But none of them enjoyed the performance and attention that MapReduce has achieved due to many reasons [*Lin and Dyer*, 2010]. This section provides a brief comparison of MapReduce to other parallel and distributed programming models that share some similarity with MapReduce.

Grid Computing

Similar to MapReduce, Grid computing technologies have been performing large scale data processing focusing on performing computations by distributing the work across several computers [*White*, 2009]. However, grid based platforms are often build servers that use a shared file system, which is practical for CPU-intensive computations but have many limitations when it comes to data-intensive computations as data sharing is by sending-receiving messages between the processes. This is the core difference between MapReduce and grid computing [*Schmidt*, 2009]. Message Passing Interface (MPI), which is considered as the lingua franca of distributed-memory applications, is extensively used in grid computing. In MPI programming the programmer need to implement mechanisms for work load partitioning, task mapping and failure handling explicitly compared to MapReduce where often only the map and reduce functions have to be implemented. [*Lin and Dyer*, 2010; *White*, 2009] has made some comparison between grid computing and MapReduce there it is stated that the main advantage of MapReduce over grid computing is data locality and simplicity.

Shared-Memory parallel computing

Shared-memory programming such as Pthreads and Open Multi-Processing (OpenMP) has been utilized both for scientific and industrial computing for many years and is being considered by many as the most extensive model compared to other parallel models [*Karavakis*, 2010]. OpenMP, which was introduced to provide a shared-memory parallelism in C, C++, python and FORTRAN, has become the platform of choice in shared-memory parallel programming since it [*Basumallik et al.*, 2007; *Karavakis*, 2010]. If we compare it to MapReduce, it is much more generic and provides a variety of solutions and is also more portable across different operating systems, architectures and compilers. However, OpenMP is primarily designed for shared-memory systems and its implementation for large-scale applications requires high investment cost as it follows the scale-up approach. Besides, there have been some successful attempts in implementing MapReduce on Shared-Memory multiprocessors which achieved a reasonable performance [*Ranger et al.*, 2007].

General Purpose Programming for Graphic Processor Units (GPGPU)

As graphic processing units have become more powerful with the primary intention of rendering images close to realism, it has also received a considerable attention from general purpose programmers to take advantage of its massively parallel architecture [*HARRIS*, 2005; *OWENS et al.*, 2007]. NVidia and ATI have been the main GPU manufactures of GPUs with different rendering power and programmability [*Advanced Micro Devices*, 2009; *NVIDIA*, 2010]. Compute Unified Device Architecture (CUDA) has been used as a standard programming platform for many years with a wide array of applications and recently with recent release of OpenCL programming language based development platform by ATI for its GPU architectures the programmability of GPUs have become more approachable.

An implementation the edge detection algorithms for large satellite images on ATI GPUs using OpenCL have showed that a computational performance of more than 20x can be achieved compared to a central processing unit (CPU) that have 5x more memory bandwidth [*Tesfamariam*, 2010]. The main constraint of GPU programming despite its high performance is that the applications programmed on GPU are hardware architecture and vendor specific. MapReduce have also been implemented on GPUs, in projects such as Mars [*He et al.*, 2008] achieving good performance. This work reveals the wide range applicability of MapReduce programming model and computationally-intensive applications can be implemented on small-scale computers.

2.5 Apache Hadoop

This study has implemented the remote sensing image processing algorithms for distributed processing on commodity clusters using MapReduce model and its open-source implementation, the Apache Hadoop. Hadoop is an Apache Software Foundation project that includes various sub-projects in it including the MapReduce implementation and Hadoop Distributed File System (HDFS) which are similar to Google's MapReduce and Google File System implementations [*Venner*, 2009]. In this study, Hadoop indicates the MapReduce programming model and execution environment along with the distributed file system. HDFS is a highly fault-tolerant distributed file system designed for storing very large files on clusters of commodity hardware nodes [*Noll*, 2004-2010; *White*, 2009]. MapReduce is build on the top of this file system but independently which consists of *JobTracker* and *Tasktrackers* which control the job execution process[*Mann and Jones*, 2008].

2.5.1 Hadoop Distributed File System

HDFS is an open-source implementation of Google File System (GFS). Externally, HDFS appears as an ordinary file system and files stored there can be deleted, moved, renamed etc [*Venner*, 2009]. But actually it is stored distributed among the data nodes. HDFS adopts a master/slave architecture in which the master, in Hadoop's case is the *Namenode*, provides the metadata service and access permissions and the slaves which are the *Datanodes* serve as storage blocks for HDFS. All the file operations of HDFS are controlled by the master server and the HDFS can only be accessed through Hadoop's. Large files stored in the distributed system are divided into blocks and replicated over multiple datanodes. The default block size of HDFS is 64MB but can be modified and files which are less than the block size are not divided [*Mann and Jones*, 2008]. This replication of files by HDFS is one of the core features of fault-tolerance and redundancy in Hadoop. And the map processes are usually performed on these data blocks on the data node which significantly reduces the amount of data that need to be transferred over the network [Yao et al., 2009]. The data transfer to and from HDFS which is done through Hadoop's API does not pass through the NameNode only metadata and log information is stored at the NameNode. Since the NameNode is usually a single server, to avoid failures there is a SecondaryNamenode that replaces the NameNode in case of failures. HDFS is used to store files that are to be used as an input for the Map phase and the results from the reduce phase; it does not store the intermediate results from the map phase [*Mann and Jones*, 2008].

2.5.2 Hadoop Data Input and Output

Hadoop have its own set of primitives for data input and output formats and simple model for processing of these data [*White*, 2009]. The *InputFormat* interface defines how and from where the map phase should read the input files. It also defines the *InputSplits* which splits the file into smaller chunks before being represented as key-value pairs. Hadoop can read and process a wide range of file formats such as text, binary, database etc formats through the base class *FileInputformat* which generalizes other file formats. The *FileInputFormat* contains methods to define which files are included as input and also an implementation for generating splits through the *InputSplits*. Hadoop also gives us an option to override the splitting of input data in case we do not want to split the data [*White*, 2009].

It also has another class for data outputs, the *OutputFormat,* which act similar to the *InputFormat*. The *FileOutputFormat* base class of *OutputFormat* is used to provide a dedicated directory where the reduce phase writes the results of the reduce tasks. This class also generates the output results in to the desired file format after the reduce computation.

Another important feature of Hadoop for managing data input and output is the *RecordReader* which provides data access mechanisms for the map phase by reading the data from its source based on the computed splits and converting it into key-value pairs. The *RecordReader* is an iterator over the record which is invoked repeatedly until the entire input splits are completely consumed by the mappers [*Venner*, 2009]. The default implementations of the above data I/O components can be replaced with customized implementations to process the desired input format, in our case image formats.

2.5.3 Hadoop Mapper and Reducer

As discussed in section 2.4.1, MapReduce is straightforward in its programming principles. The programmer has to specify a map function where one map job is performed in parallel on the file splits. Each input split generated by the *RecordReader* is assigned to each map task and after the map function has been applied to each input split the output is stored in local storage and the *Tasktracker* that is residing at the DataNode notifies the *JobTracker* of the job completion. After that the reducer pulls the groups of key-value pairs form the mapper and merges these values to produce a single key-value pair. This pulling principle is proposed by Google for more fault-tolerance and to avoid re-execution of all map tasks if one task fails [*Dean and Ghemawat*, 2010].

Before the reducers start merging the results in parallel, the output key-value pairs with the same key have to be grouped together in order for the reduce merge them to the final output. In order this to be done shuffling of intermediate results over the network is needed. This process shuffling process may take a considerable time if there are a large number of unsorted intermediate results. To minimize this communication over the network Hadoop gives an option to use a function, Combiner function, that merges the with-in the map result [*White*, 2009]. The Combiner function is an optional optimization strategy which can be ignored if considered redundant.

Chapter 3 Design of MapReduce based Remote Sensing Image Processing

The main focus of this study is to test the feasibility of distributed processing of large satellite images using the MapReduce model to solve the problem of data bulkiness. The approach is to apply parallelize the process of those large images by dividing the image into smaller images, process them by different processors in parallel and merge to yield the final output. The assumption is that we have a large amount of massive satellite images to perform computation in low-cost commodity hardware. The MapReduce programming is chosen achieve this goal because of its advantages over other distributed computing technologies discussed in section 2.4.4. One of the main rationales is that in other distributed computing services such as MPI, shared-memory models etc. the computing nodes often have a shared filesystem and data has to be moved to the nodes for computation each time a job is executed. While MapReduce built on a distributed filesystem and therefore as moving data over the network is expensive it is assumed that it is logical to process those large satellite images where they are stored.

The MapReduce programming model have also provides a highly simplified interface to the application developer compared to the other models as only the map and reduce function are needed the whole distributed processing application besides the data input and output handling. But this restricts us in having control over issues such as where and when a mapper or reducer executes, which input chunks are processed by a specific node or mapper and which intermediate data is processed by a specific reducer [*Lin and Dyer*, 2010; *Venner*, 2009; *White*, 2009]. This brings a challenge to on how to optimize our algorithms to perform well especially if the distributed computing resource is heterogeneous in nature. However, there are a number of mechanisms in manipulating the data flow by assembling intricate data structures as key-value pairs and shuffling mechanisms of intermediate results which are discussed in detail in section 3.4.

Hadoop is chosen as testing platform because it is the main open-source implementation of the MapReduce programming model and its simplicity in setting up a Hadoop based system and run MapReduce based applications [*Lin* *and Dyer*, 2010]. In this study, main issues to consider when implementing MapReduce based algorithms in Hadoop environment are the file input and output formats to be used for processing since Hadoop's primary purpose was bulk unstructured text processing. The second concern is how to divide the raster datasets into smaller chunks so that they can be efficiently processed in parallel. The third issue is how to systematically design the image processing algorithms through the map and reduce functions. This chapter discusses the design concepts to address these concerns and what needs to be implemented for Hadoop to deal with the images for processing.

3.1 Overview of Components

As part of the design of MapReduce algorithms there are also other important design and implementation that must be considered such as inducing the key-value structure on the remote sensing image datasets [*Lin and Dyer*, 2010]. The first thing to consider when designing an image processing algorithm in a Hadoop environment is how to familiarize it to read, process, and write images file formats. This should be done by extending the Hadoop API to allow images to be parsed into the BytesWritable wrapper of Hadoop which is a container for binary formats. The input image is read by from the underlying distributed file system of Hadoop by tiling into several splits and where the sub images are then processed by separate. The important issue here is how the pixel at the border of the sub-images are handled, since all the detection algorithms involve convolution operation the sub-images must have overlapping pixels at the borders based on the kernel size of the convolution to avoid gaps when the image is merged. Those tiled images are converted into key-value pair format so that the map task can understand it where the key is generated from the file name of the whole image and position of the tile in the original image and the value holds the actual image data.

The mapper which contains our edge detection algorithms is applied to every input image represented in the key-value pair to generate an intermediate edge detected image is produced in another key-value pair representation. The reducer is applied to those intermediate values after the map task is finished after being sorted and shuffled among the worker nodes. In our implementations the task of a reducer is to get the computed image tiles and use the information associated with them (key) merge them to produce the final output image.

Figure 3-1 depicts the major components of the two-stage system processing structure where the Mappers are applied to all input key-value pairs, which generate processed sub-images of intermediate key-value pairs. Combiner gathers unit images of the same files together and Reducers are applied to merge all images associated with the same key. Between the map and reduce phases there is distributed data sorting. In this scenario the codes of the mappers and reducers along with file format definition are packaged by a driver together with some configuration parameters to produce one MapReduce program where it can be submitted to a master server to process images. There is no direct relationship between the input image files and the MapReduce program and to execute this job the HDFS file path of the image files have to be submitted along with the package to the master. All other aspects of the distributed processing of the images such as distributed execution, failure-handling, job scheduling etc [*Venner*, 2009]. The following sections discuss in detail about the main components of the MapReduce program.

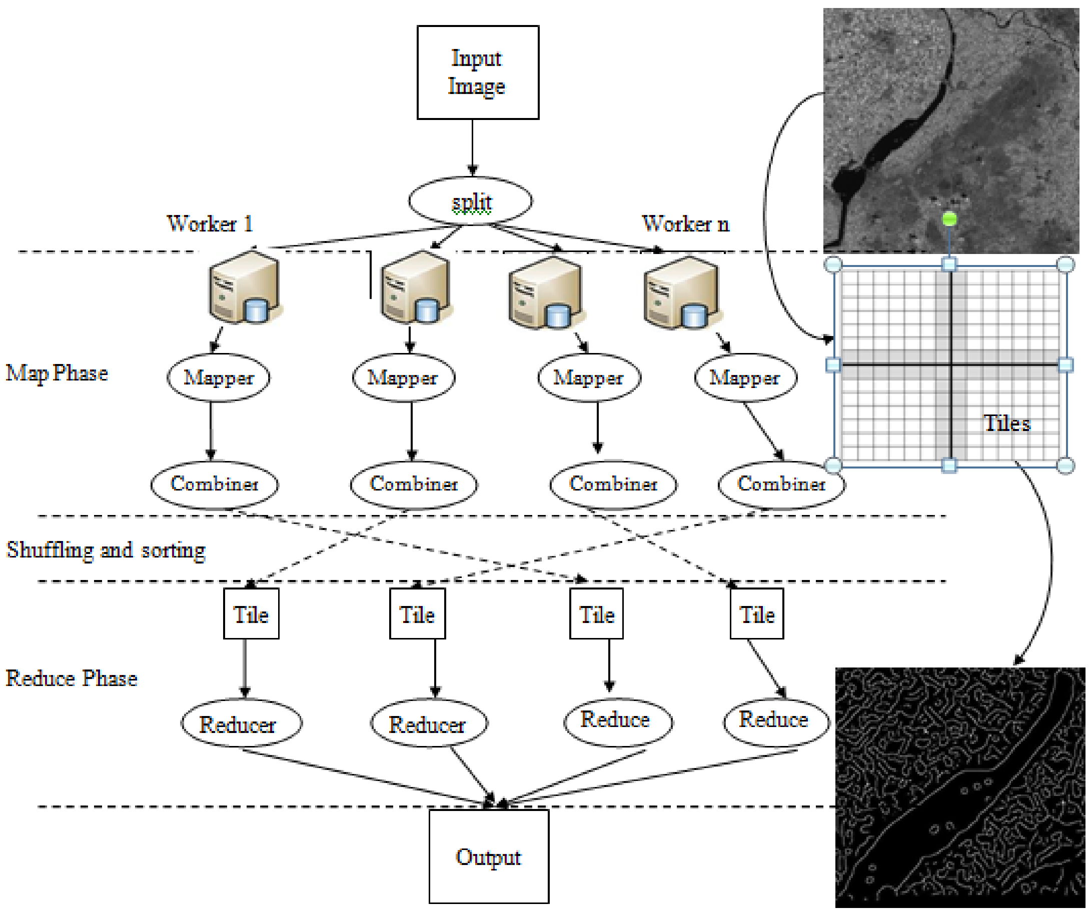


Figure 3-1 Simplified view of the MapReduce process for image processing.

3.2 Data Input and Output

The image input for our MapReduce job is typically large satellite image files stored in the Hadoop Distributed File System. Our design assumes that these images are stored in tiled TIFF format with augmented metadata where they can be readily read by tiling methods within the boundary of the whole image without restriction. Hadoop provides us with many input and output formats such as text format, binary format, database format etc through the *InputFormat* and *OutputFormat* interfaces [*White*, 2009]. These interfaces are extendable and therefore we designed our own file format on the top of Hadoop's *FileInputFormat* class that reads image files. This customized FileInputFormat defines from where the input files are to be read by taking the file input path as an argument. It also defines how the input file should be tiled for processing using the *IputSplit* interface.

The splits do not actually parse and get the tiled data, they are only a reference to the chunk data [*White*, 2009]. Their primary purpose is how and into how many will the data be sliced. But the actual loading and assigning of keys and values is performed by the *RecordReader* which is defined by our *FileInputFormat* and it is invoked iteratively on the input image until all the tiles are finished. The *RecordReader* class of Hadoop was extended in order to be able to load image data and converts it into Key-value objects that are suitable for reading by the Mappers. The UML class diagram illustrated in Figure 3-3 shows the extended *RecordReader* class with its necessary attributes and functions and we can observe here this class does the job of converting the image splits into key-value pairs by iterating using the *next()* method. We can then easily choose to use our own file format to apply to the input files when configuring the job and the mappers can use the information about the input split properties to process those splits.

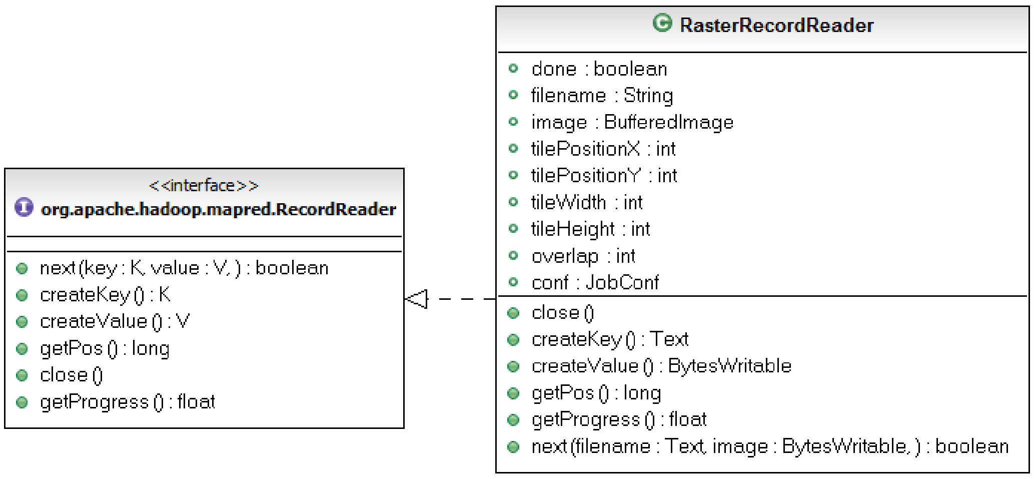


Figure 3-2: RasterRecordReader that extends Hadoop's RecordReader interface.

3.3 Data Splitting and Merging

Two kinds of data partitioning mechanisms can be performed on a multispectral data: partition based on the image plane (the spatial domain) or on the spectral space. Partitioning on the spectral space refers to when the input image is portioned based on the spectral bands where the different bands of the same image are to be processed in parallel [*Valencia et al.*, 2008]. While partitioning on the image plane splits the input image into smaller chunks on the spatial domain (in terms of width and height of the image). In this study since the edge detection algorithms are spatial filters that operate on the spatial domain and one band is going to be used independently to enhance edges of the image, we are going to be mainly concerned on how to apply partition of the satellite image on the spatial domain. The performance of parallel implementation of image processing algorithms is highly dependent on how the data is partitioned.

One of the key concerns that arise with partitioning images for parallel processing in the spatial domain is the issue of accessing pixels outside the spatial domain of the image splits available in the computing node. This is usually managed by border handling strategy by replication of pixels at the processors to avoid border effects. For instance, if we take the example of Sobel operator which uses 3x3 filtering kernel as a convolution operation, the number of pixels that have to be replicated P_r in the processing of an image can be computed given by the equation below [*Valencia et al.*, 2008].

$$P_r = 2 * \left[\left(2\left\lceil\frac{\log_2 N}{2}\right\rceil t\right) - 1\right] * I_R + 2 * \left[\left(2\left\lceil\frac{\log_2 N}{2}\right\rceil\right) - 1\right] * I_C$$

Where N is the number of partitions, I_R is the number of rows in the original image, and I_C is the number of columns in the original image.

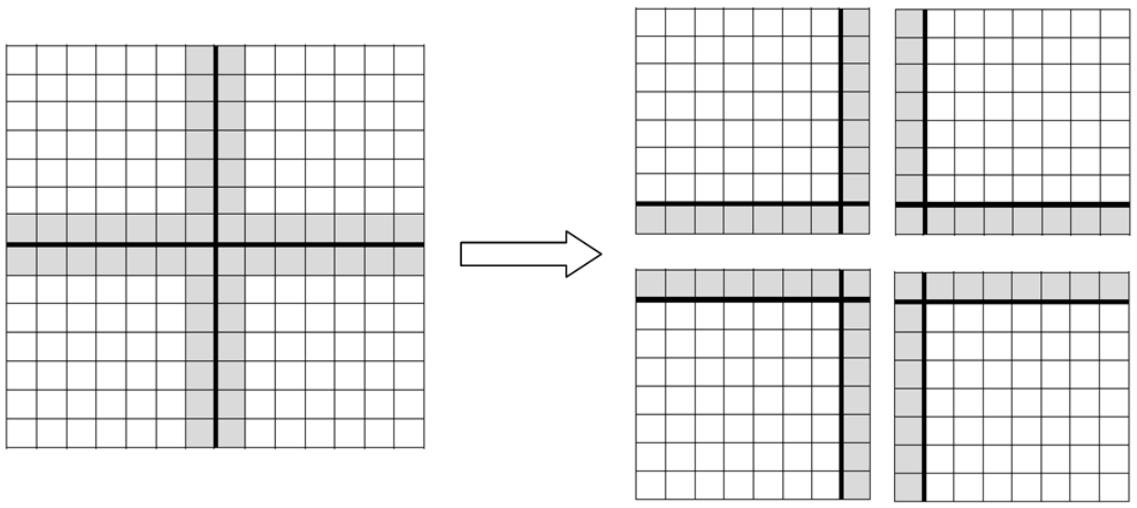


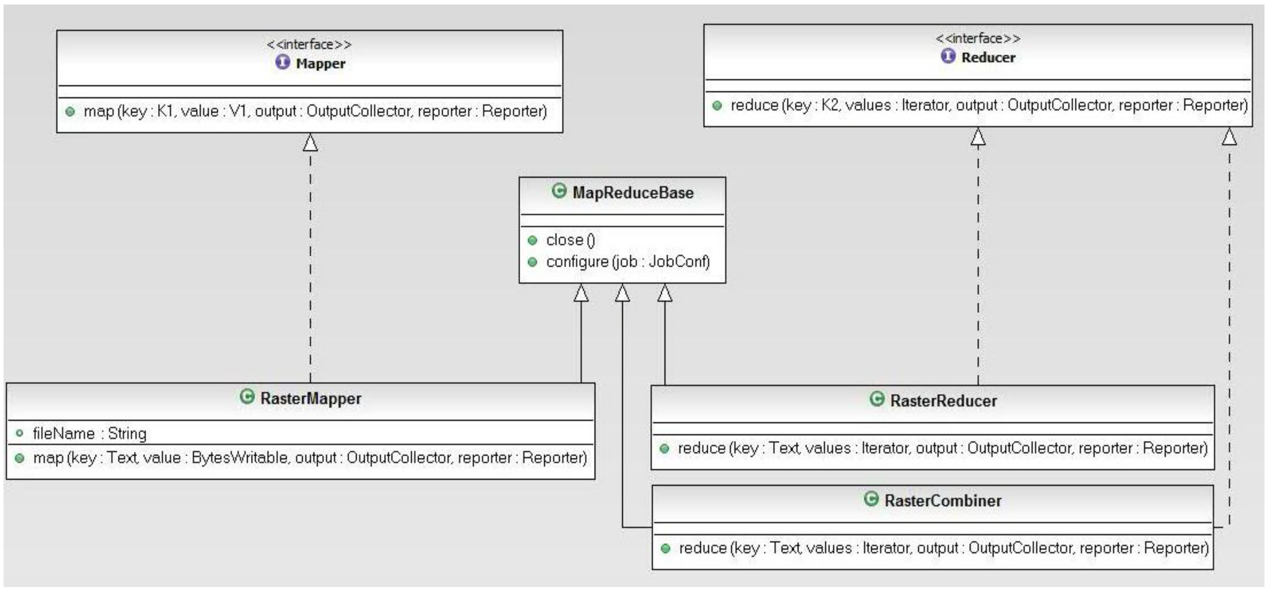
Figure 3-3: Input image splitting strategy to handle border pixels for a 3x3 filter.

The sub-images are generated by splitting the image into regular chunks of the same dimension in horizontal and vertical direction until the end of the whole image. For the Sobel filter which the kernel size is 3x3 a two pixels wide gap forms in the middle of the output image and the outer border pixels are also lost. This is remediated by overlapping the sub-images by two pixels during the slicing of the input image (Figure 3-4). The same strategy must be applied for the 3x3 Laplacian filter. As for the 5x5 Laplacian filter a four pixel overlap was need to remove the pixel gap.

The merging of the of the result images is straight forward as the edge detection algorithms will produce tiles that exactly match at their borders. As outside for the borders of the whole image, they can be ignored if the kernel size is small but for the Canny method where Gaussian smoothing is applied with large kernel sizes (up to 25 pixels width) some mechanism have to be devised to get values for these pixels.

3.4 MapReduce functions

After the image splits are generated and represented in key-value format by our RecordReader the map function is invoked for every key-value pair by implementing the mapper class. The key-value pairs are the basic data structure and are the only arguments that the map function needs to start processing (Figure 3-4) [*Lin and Dyer*, 2010]. The key holds the image file name and the sub-image ID and the value is the sub-image itself. The map function produces an edge detected image as an intermediate value along with the new output keys. In this study the task of the map function is to do the complete edge processing task on the image tiles and give edge image outputs where they are merged by the reduce function. The following sections discuss in detail about the execution overview of the map, reduce, combiner functions and methods involved within these functions.



<<interface>>
Mapper
map (key : K1, value : V1, output : OutputCollector, reporter : Reporter)
<<interface>>
Reducer
reduce (key : K2, values : Iterator, output : OutputCollector, reporter : Reporter)
MapReduceBase
close ()
configure (job : JobConf)
RasterMapper
fileName : String
map (key : Text, value : BytesWritable, output : OutputCollector, reporter : Reporter)
RasterReducer
reduce (key : Text, values : Iterator, output : OutputCollector, reporter : Reporter)
RasterCombiner
reduce (key : Text, values : Iterator, output : OutputCollector, reporter : Reporter)

Figure 3-4: Implements of Mapper and Reducer.

3.4.1 Mapper

It is clear that the *map* method is a pure function with a sole purpose to process the data splits represented as key-value pairs in parallel mode with no communication with among the other map processes. The map method also receives two more parameters beside the key and value. The first one is the *OutputCollector* which writes the output images in another key-value format to be forwarded to the reduce tasks. The second instance is the Reporter which reports information about the map task. The first thing our map function does is grab the key and then decompress and read the value associated with it using java image reader classes. After this image is read the image processing algorithm is performed on this image tile. In this study the edge detection algorithms which are explained in section 2.2 are implemented as a use case scenario. The edge detection process is explicitly performed from start to end at this stage. The sequential implementation of the edge detection algorithms can be easily plugged into the map function with no or slight modifications. The processed images are then compressed into array of bytes where they are represented as key-value pairs and sent to the reducers. When a map task finishes, then the master forward these key-value pairs to the reduce workers. At this stage, in order to reduce the volume of the intermediate images to be transferred to the reduce workers over the network, grouping of the intermediate computed image tiles is made by the map worker nodes locally by the combiner function (similar to the reduce function) which we will discuss it in section 3.3.3. Since a MapReduce job have mappers and reducers running in parallel and sharing the distributed file system, special attention is taken in the file naming system of the intermediate files to avoid conflicts and synchronization between the mappers. The mappers run until all the key-value pairs are processed and after the finishing of all the mappers we can terminate the code and close the data output streaming. Figure 3-5 shows a Unified Modeling Language (UML2) sequence diagram depicting the main execution overview of the map phase. It also shows how the execution framework instantiates the map tasks.

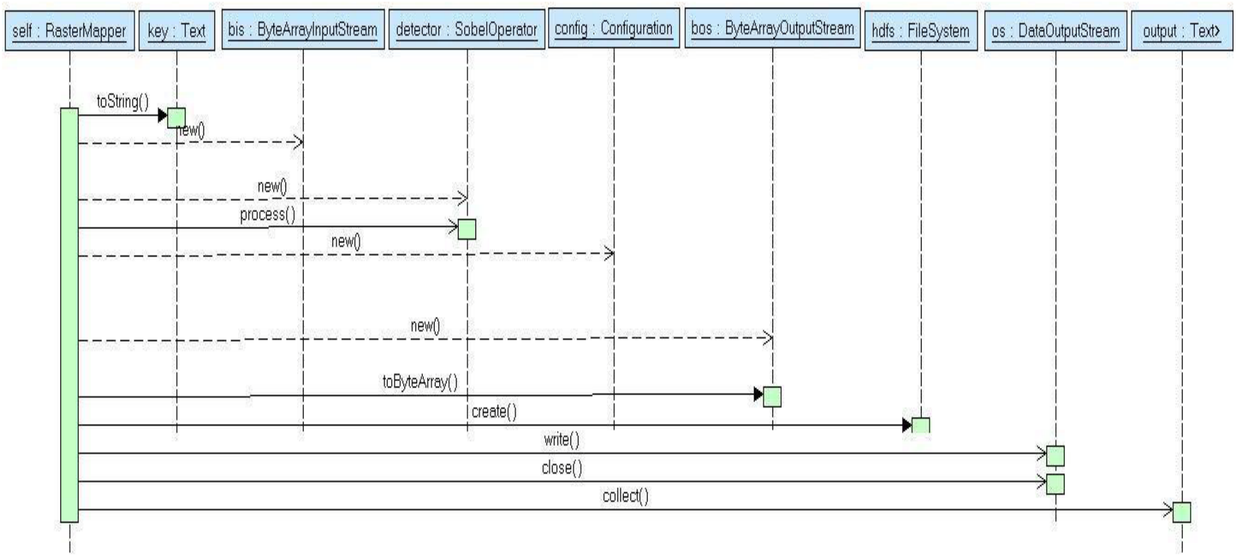


Figure 3-5: Sequence diagram for the map phase.

3.4.2 Combiner

To minimize the communication over network during the sorting and shuffling process a *combiner* function is used to aggregate the image out puts from the mappers locally before being sent to the reduce workers. This *combiner* function does exactly the same process as the *reduce* function but only aggregates output image tiles if they are found next to each other in the whole image. This brought a slight challenge to the reduce function as some of the image tiles are now bigger in size but it greatly improves the performance of both the reduce and the shuffle processes. This combiner function is especially important for the canny method because the output tile is much smaller (more than 20% of the original image) than the input tile as it is in binary format.

3.4.3 Reducer

The task of our reducer is straightforward, only to merge the result images from the map tasks and to give the final image. The Hadoop API provides us with options for initialization and closing of the reduce task and these reduce tasks start as soon as each map task is finished. The first step is to set the destination where the final aggregated image is going to be put. Here we use the information from the original input image to create an output image with the same dimension as the original image. We can then grab the key-value pairs that have been produced by the mappers and sort them according their key id. From this information we can identify original location of the tiled image within the whole image and then the value can be read, decompressed and streamed to its identified location after checking its boundary conditions. The reduce tasks are run in parallel to write the images to the distributed file system until all image tiles are stacked to their respective location in the complete image. We can observe from the sequence diagram in Figure 3-6 that the major part of the reduce tasks is reading and writing image files.

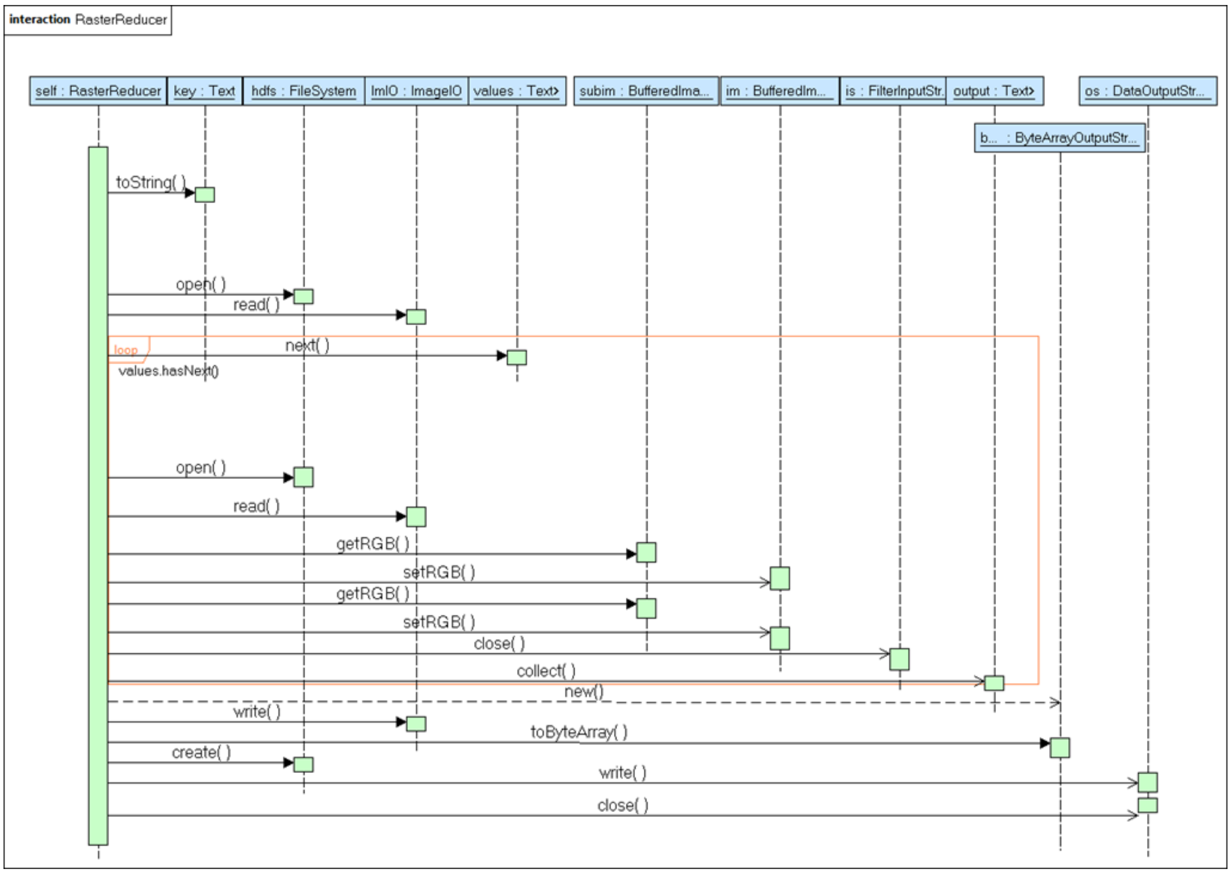


Figure 3-6: Sequence diagram for the reduce phase.

3.5 MapReduce Driver

Having specified the MapReduce jobs and how the image data format is going to be read, final step is to design a driver program where we specify is how the job is run on a Hadoop environment and also which data input and output formats and from where to use data for processing. Hadoop API provides us a clear-cut mechanism to set the parameters and other configurations that are necessary to run a MapReduce job through the *JobConf* object [*Venner*, 2009; *White*, 2009]. By creating the job to be done through this driver, we can specify the input and output paths, input and output formats as well as our own parameters related to the map job. How many reducers must the MapReduce job run is also set in this driver. This class will be the main driver after the codes are packaged in JAR compression and it will be responsible for job submission, running, and progress reporting.

Chapter 4 Implementation

The Apache Hadoop MapReduce framework is favored as a distributed platform for processing of large remote sensing images because of its simplicity in setup and deployment and its high-level java development tools. The edge diction algorithms are implemented using the apache Hadoop MapReduce framework for as a map and reduce functions. All the development of the algorithms and associated codes are developed in java. By extending the current API in the Hadoop library a system is built that allows for parallel implementation of the image processing algorithms. This chapter discusses the MapReduce implementation details of these algorithms on Apache Hadoop environment.

4.1 Extending Hadoop API

The Hadoop API allows for creation an extension of input formats, by implementing the *FileInputFormat* and *RecordReader* interfaces where it becomes possible to describe the way the input image is split and sent to the Mappers for processing. The BytesWritable container of Hadoop API was used in this implementation to allow images to be parsed by the FileInputFormat. The BytesWritable is a wrapper for an array of binary data and its serialized format is an integer field that specifies the number of bytes to follow, followed by the bytes themselves [*White*, 2009]. Therefore the input image tiles can be easily parsed and wrapped using this container.

First, a class was created that extends the Hadoop *RecordReader* class to deliver the image file contents as the value of the record. The *RecordReader* is an important class that is used by the *Map* function to generate record key-value pairs. This *RasterRecordReader* inherited class created is also used to generate record key-value pairs for the image splits to be processed by the map function. Since the input images are too big to be read at once for splitting, the Java Advanced Imaging (JAI) API was used to read the image from the file system one image split at a time and convert it into key-value pair format. The *ImageReader* class of the API allows us to read rectangular tiles of the image in the horizontal and vertical direction by specifying the starting x and y direction and the width and length of the desired image tile. After decoding the image tile from the stream it is converted in to an array of bytes and distinctive filename (*tilePositionX+_tilePositionY+_tileWidth+_tileHeight*) which is the key part in the key-value pair is given to it based on its location in the original image. The process continues to read the next tile until the whole image is tiled. This naming system is crucial for the reduce function which sorts and merge the image tiles together based on this information.

This image splitting method is used inside our extended *RasterRecordReader* which is able to split the image convert it to key-value pairs in the form of an array of binary data and make it ready for the map function to process those image split. The main methods that are implemented in this class can be seen in Figure 3-3. It is also here where we define the number of pixel rows and columns that are going to be overlapped among the image tiles depending on which edge detection algorithm is being implemented. The pseudo code in Listing 4-1 shows the important sections of code the adopted image reading and splitting mechanism by the *RasterRecordReader*. Here, we can observe that the minimum dimension of an image in order to be split is 1024x1024, images below that size are directly converted to key-value pairs.

A second class that is extended in this study is the *FileInputFormat* (Figure 3-2) which is the base class for any file-based input formats that provides a place to define which files are included as the input to a job and also an implementation for generating splits for the input files [*White*, 2009]. In our case, both are both these are implemented by the *RasterRecordReader* class, therefore the main purpose our extension of the FileInputFormat is to the instantiate the *RasterRecordReader*. Another important argument is also passed in this implementation: the *FileInputFormat* also by default splits data that are larger than the HDFS block size. Since we have implemented our own splitting mechanism in the within the *RecordReader*, any splitting by that might be done by default by *FileInputFormat* is disabled by overriding the *isSplittable()* method.

Listing 4-1: Hadoop's *RecordReader* interface expanded to deal with images.

```
RecordReader<Text, BytesWritable>
declare variables
overlap; tile width; tile height;

  // start the splitting and key-value generation process
  initialize(FileSplit, Configuration);
  getHDFSFilePath();
  getFileSystem();
  hdfs.open(); //opens HDFS
  hdfs.mkdir(hdfs_path+"filename"+"output"); //create directory for tiles

  // Split the image only if it is > 1024x1024pixels
  if( width * height is less greater than 1024*1024 ){
    key(filename); &// URI format
    value(wholeImage);
  }
  else{
    count=0;
    for(i=0;i<horizontalNumberofTiles;i++) {
      for(j=0;j<verticalNumberofTiles;j++)
          {
        // read the desired portion of the image
            Read using ImageReader (0, TileBox);
            BufferImage(tileWidth+overlap, tileHeight+overlap, pxlValue);

   // create the key in the form of URI based on its original position
            key(filename+tilePostionX+tilePositionY);
   // write image tile to byte array
            write to byte array (tiledImage, "jpg", byteOutputStream);

   // set the value to the byte array with a starting offset and length
            image.set( byteTile, offset, byteTile.length );
          }
      }
  count++;

  getProgress();
  closeInputStream();
```

4.2 Edge Detection Algorithms

All the edge detection processes are run by the map function. The sequential edge detection algorithms developed in java can be easily adopted to be plugged into map function with little or no modification. The main difference between the Sobel, Laplacian, and Canny algorithms in terms of coding and implementation is when considering the width of the overlap of the image tiles. Since the image splitting is done by the *RasterRecordReader* class, a separate record reader class was developed for each of the algorithms. Besides that class all the other codes can be used by the three algorithms without modification.

4.2.1 Sobel edge detection

The Sobel operator uses two 3x3 filter kernels in order to compute the magnitude and direction of edges. To compute the Sobel operator at the border pixels of the image tiles, the number of pixel rows and columns that must be overlapped is two which is set in the record format during splitting. Figure 4-1 shows the main attributes and methods of the Sobel operator class. To further optimize the Sobel operator during the process the computationally expensive square root function that calculates the magnitude of the output as function of the horizontal and vertical detected edges was replaced by the summation absolute values of the horizontal and vertical edges. After processing the image with Sobel operator the resulting magnitude image is compressed into an array of bytes, ready to be processed by the reduce workers.

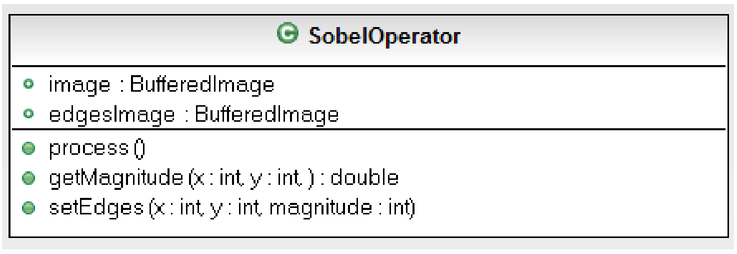

SobelOperator
image : BufferedImage
edgesImage : BufferedImage
process ()
getMagnitude (x : int, y : int) : double
setEdges (x : int, y : int, magnitude : int)

Figure 4-1: Sobel operator class diagram.

4.2.2 Laplacian edge detection

Similar principle to the Sobel operator was adopted to implement the Laplacian edge detector Hadoop except by changing the values of the filtering kernels and also using only one filter kernel instead of two. Two separate Laplacian filters were implemented one with a kernel size of 3x3 and another one with 5x5. The same splitting mechanism was performed in order to process the tile images by the 3x3 sized kernel while for the 5x5 kernel a different record reader was used to where the image is split with an overlap four pixel rows and columns.

4.2.3 Canny edge detection

The Canny edge detection method is a multi-step method where many operations that involve setting parameters are involved. The main methods in the code of the Canny algorithm was adopted from the sequential implementation in java by [*Gibara*, 2009] with some modifications in parameters setting and how to handle the border pixels. One of the challenges that affect the border handling of the image tiles was setting the kernel width of the Gaussian smoothing filter. This kernel width is computed by *Equation 2-5* based on the given standard deviation σ parameter and therefore a separate *RecordReader* class was implemented to create overlapping of borders between the image tiles. The other parameters to be set are the low and high hysteresis thresholds which trace the edges based on the gradient and connectivity of the edges. Various statistical methods were attempted to automate the selection of the thresholds of the hysteresis based on the value of the image pixels such as using the 1st and 3rd quantiles of the pixel values. But this resulted some degeneration in the quality of the edges detects which is discussed in the next chapter.

Listing 4-2: Canny edge detection method

```
declare variables
Gaussian Cutoff; Gaussian Standard deviation;
lowThreshold; highThreshold;

getSourceImage();
process();
// set hysteresis threshold
setLowThreshold(threshold);
setHighThreshold(threshold);
//set the Gaussian kernel width
getGaussianKernelWidth();
// Sets the strength of the Gaussian convolution kernel for smoothing
setGaussianKernelRadius(gaussianSigma);
gaussianSmooth( gaussianSigma, gaussiankernelWidth);
sobelGradient();

supressNonMaxima();
performHysteresis(lowThreshold, highThreshold);
writeEdges();
```

4.3 MapReduce functions

The MapReduce implementation consists of two separate classes for a mapper and reducer. Since the splitting of the image and converting into key-value pairs structure has already been done by the *RasterRecordReader* the task of the mapper is to take the key-value pair for processing by the edge detection algorithms. The reducer will collect those edge detected image tiles and merge them into one output image. In these functions the input key is particularly important as is also hold location information of the image tiles in the original image. In this context, there are a series of steps and methods have to be implemented with close coordination between the two functions. The detail implementation of the map and reduce functions is discussed below.

4.3.1 Mapper

As mentioned section 4.1 key-value pair represents the image splits in the form of an array of bytes wrapped by Hadoop's *ByteWritable* class. Each map task processes a single image split that has been generated by our *RecordReader*. The map function basically have three major steps: at the start of the process the value is decoded using the java image reader and converted into a buffered image before being processed by the edge detection algorithms. Then the edge detection algorithm is called to process the decoded image and produce an edge detected image as shown in listing 4-3. Finally, this edge image is converted back to an array of bytes and represented as new key-value pair. The new assigned key to the edge image is the file name of the original image.

Listing 4-3: mapper for edge detection

```
map (key, value) // key: URI to image tile, value: image in byte array
        // we decode the byte array to buffered image
        read byte array (value);

        // apply edge algorithms to detect edges
        detect edges (get edge image);

        //get the bytes of the buffered edge image
        to byte array (edge image);
        //generate output key
        key (image filename);

        // output key-value pair
        output.create (key, image bytes);
```

4.3.2 Combiner

The combiner phase simply collects the map output for each mapper and combines those key-value pairs that are from the same image before being pushed to the reducers. This class implements the reduce function using the Reducer interface the same as the reduce phase (Appendix 1) but collects values are only within one mapper node. Our implemented reduce function was used in the combiner phase since the objective is to merge neighboring image splits but the output of the combiner phase is in key-value pair since they are going to be further processed by the reduce phase. By implementing this class we are reducing the communications between the map and reduce phases and therefore reducing the network load.

4.3.3 Reducer

The reduce phase collects the edge detected image tiles from the intermediate map queue and combines them to the final image. The image subsets, sorted by key, are dynamically assigned to the reducers by the scheduler of the execution framework. Then the reducers merge those images to a final output image by recursively writing to the distributed file system based on the location information in the original image. Listing 4-4 depicts the main processes involved during the reduce phase.

Listing 4-4: reducer for merging image tiles

```
reduce (key, Iterator values):
        // create a buffered output image at HDFS
        bufferedImage(hdfsPath);
        // go through the values and collect the images
                while( values has next ())
                {
                // get the image tiles
                imageTile.getBytes();
                //allocate image tile to original image box
                setRGB (startX, startY, witdth, height, byteArray, offset);

                //close the sub image
                close();
                // collect the output
                output.collect( key, list values);
                }
                // Create a data streaming to HDFS
                FSDataOutputStream (hdfsPath );
                // write the image file at hdfs
                DataOutputStream.write(byteArray, offset, array.length);
                output.collect (key, result);

                // close the streaming
                close();
```

4.4 MapReduce Driver

The driver program handles the housekeeping of our MapReduce job so that it can be submitted to a Hadoop environment. First a job configuration was created using the *JobConf* object followed by the required parameters for the job such as the input and output formats that the data input and output format, directory where to look for data and the map and reduce classes. For the input format the customized *RecordReader* class is used which is used as an input for the map phase. The output format is set to *TextOutputFormat* but the reduce phase overrides this format and writes the output images directly to the distributed file system.

The next step is to configure the map, reduce, and combiner phases the setting methods of the Job configuration. Here the number of reduce tasks that must run in this job can also be set by the *setNumbReduceTask()* which determines how the final image is going to be merged. Finally we call the static *runJob()* which submits the job, reports the progress of the map and reduce phases. The Tool interface in this class also gives us to use some optimization options for passing arbitrary parameters from the command line interface such as setting the number of reducer, putting files to the distributed cache.

Chapter 5 Evaluation

There are three primary objectives for evaluation conducted in this study. Firstly, to demonstrate if the reliability and performance of the remote sensing applications has been improved through the implementation on distributed environment as proposed in the previous chapters. Secondly, that the MapReduce programming framework is a feasible model for efficient processing of large satellite images. Finally, the quality of the end product is not affected by implementation of the remote sensing applications on a MapReduce framework. This chapter will start in the first section with the description hardware and software configurations of the testing environment where the prototype framework and applications are run. The following section discusses the characteristics of the test datasets that has been used in this study. The third section discusses the qualitative evaluation of the output images from the various algorithms tested. Finally, the last section illustrates the benchmarking results obtained from the performance tests done to evaluate the computational performance.

5.1 Testing Environment

All the experiments in this study are done on normal commodity PCs and notebooks. One of the main characteristics of the test environment is its heterogeneous configuration. The test bed is built on Linux operating system (Ubuntu 10.10) on 4 computers each of different hardware architecture and performance. Moreover, the two desktop PCs are connected through high-speed Gigabit network connections and these are connected to the master workstation (notebook 2) through wide area network (WAN). This heterogeneity in hardware and network connection can be seen be one of the parameters for evaluating Apache Hadoop’s transparency and fault-tolerance. But this type of infrastructure also brings latency and unpredictability to the computational performance.

Table 5-1: Test environment system hardware information

	Notebook1	**Notebook2**	**PC2**	**PC1**
OS	Ubuntu 10.10	Ubuntu 10.10	Ubuntu 10.10	Ubuntu 10.10
Processor	Intel Core 2 Duo 2.3Ghz	Intel Core 2 Duo 1.8Ghz	AMD Dual Core, 2.6 GHz	Pentium 4, 2.40Ghz
Cache	3072KB	2048KB	512KB	512KB
RAM	4GB	3GB	1.7GB	748.2MB

The Hadoop framework used in this experiment is Cloudera's Distribution of Hadoop 0.20.1 which is the settable version CDH2. It is deployed on each node on the top of Java SE Runtime Environment version 1.6. Notebook 1 is used as both as a masternode and datanode with *jobtracker* and namenode on it that controls all MapReduce jobs and datanodes respectively Figure 5-1. The other machines act as worker nodes with datanode and *tasktracker* on them. Each node is set to be capable of running 2 Map jobs and 2 Reduce jobs concurrently except for the last node which runs 1 Map and 1 Reduce job at a given time since its processor is single-core. Therefore, in total if all the nodes are active in the system have a total of 14 task capacity.

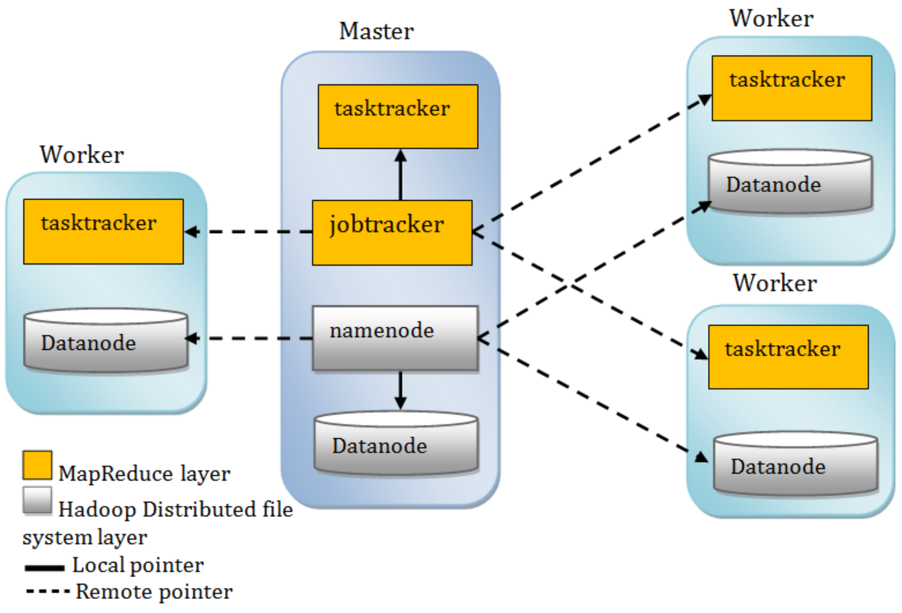


Figure 5-1: Test environment setup.

5.2 Test Datasets

The satellite images used in this experiment 4 scenes of a 30meters resolution Landsat 7 Enhanced Thematic Mapper plus of Muenster area taken on the year 1999 and were downloaded from United States Geological Survey's (USGS) Earth Explorer portal (Figure 5-2). The Band 4 is used during the experiment as it does not need contrast enhancement compared to the other bands. The original data are Level-1 processed image, each scene with a dimension of 8251x7591 in an uncompressed 8bits per pixel Geographic Tag Image Input Format (GeoTIFF) and a size of about 60MB. The image is fed to the HDFS along with its metadata file to be read by the MapReduce applications. It is worth noting that the transfer of the input file to the Hadoop Distributed system is not part of the performance test as we are assuming that the image database is stored using HDFS.

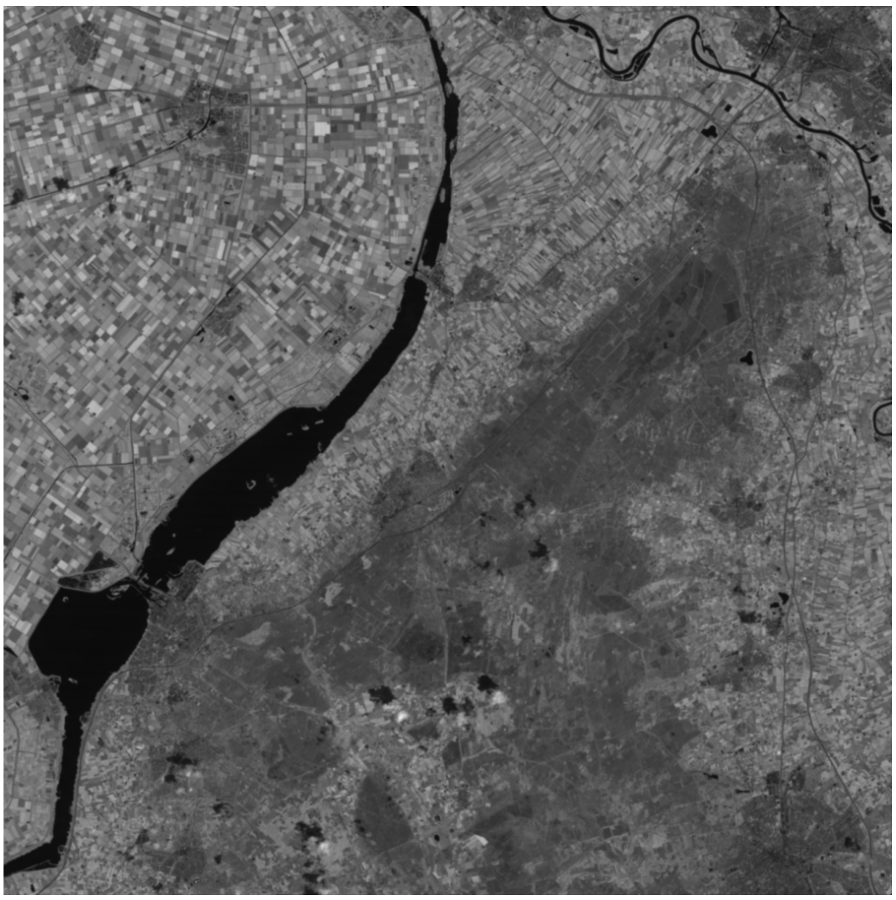


Figure 5-2: Band 4 Landsat image (*courtesy of the U.S.G.S.*)

5.3 Qualitative Evaluation

Assessment was made on the quality of images produced by the Sobel, Laplacian and Canny edge detection methods to evaluate if the parallel implementation of the algorithms using MapReduce affects the quality of the image and also to evaluate the performance between the three methods. The quality of output images are evaluated in according to three criteria set for visual interpretation [*Canny*, 1986]. Firstly one is good edge extraction in terms of the probability of detecting edges the maximum possible level with minimum falsely detected edges (noises). Secondly is to evaluate how well are the edges connected and look real. And finally, the detected edges must not be duplicated and their position should be as close as possible to the actual edges.

It is known that quality of the detected edges is highly dependent on the image characteristics and also the characteristics of the object of interest which influences the parameters to be set for the edge detection algorithms [*Canny*, 1986; *Drewniok*, 1994; *Heath et al.*, 1998; *Liu and Jezek*, 2004]. This suggests that it difficult to generalize these parameters across all algorithms and all images. Therefore, the quality of the detected edges from the three detection methods is evaluated for a single image that has a fixed size and resolution and the most relevant parameters are identified and adaptively tuned for each detection method.

5.3.1 Sobel method

For the Sobel method, since a fixed 3x3 kernel size is used for convolution and the final pixel value is the gradient magnitude of the Sobel operator, there are no parameters to be set, therefore the output images can be visually compared and evaluated. Figure 5-3 the out image of the implementation of Sobel edge detector on MapReduce. It can be seen that the Sobel method does are reasonably good job in detecting most of the edges with minimal inclusion of false edges or noises. The edges are also well connected with few disconnected lines. But the method seem to enhance mostly those edge that have sharp contrast and this suggests that wider and non sharp edges are difficult to detect using the Sobel operator. The MapReduce implementation of the Sobel method provided the same results in terms of image output characteristics and quality as the corresponding sequential implementation.

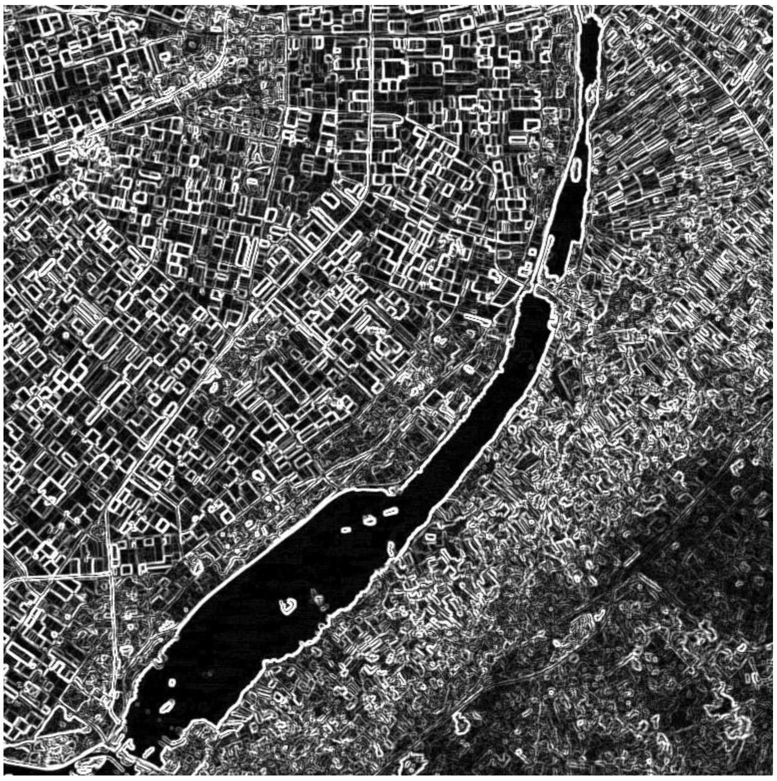


Figure 5-3: Edges detected by Sobel method

5.3.2 Laplacian method

The Laplacian method was evaluated for different kernel sizes (Figure 2-4) and generally the Laplacian operators produce low contrast images where it is difficult to identify most edges. This is due to the low contrast of the grey-scale images of the Landsat bands and poor detection capability of the Laplacian operator for low contrast images. Therefore to remedy this, the Laplacian method was tuned to enhance the contrast of the output image until the most of the edges are clear enough for visual inspection. The result after the contrast enhancement shows that the Laplacian method recognizes many edges for all the kernel sizes but the quality of the edges is low compared to that of Sobel method. The detected edges are wider than the actual width of the edges resulting less fine grained detail and also a considerable amount noise is introduced by the operators. Many fragmented edges were also observed from the result especially for the images processed with a 3x3 dimensioned kernel Figure 5-4 (left). The edges from the image processed by the 5x5 kernel have also thinner width compared to image processed by 3x3 kernel.

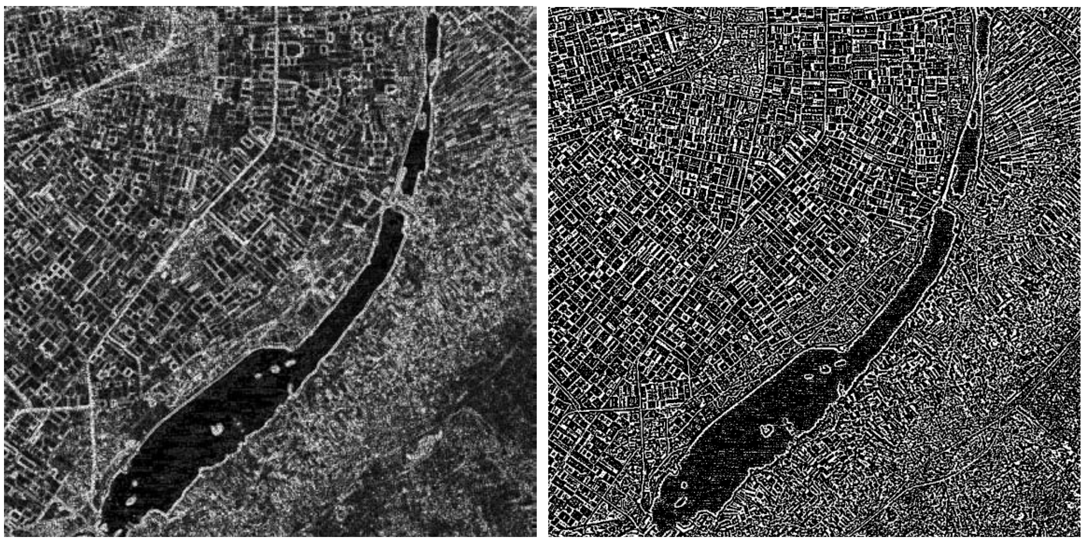


Figure 5-4: Edges detected by Laplacian filter with 3x3 kernel (left) and 5x5 kernel (right)

Similar to the Sobel method the MapReduce implementation of the Laplacian method provided exactly the same image output quality when compared the corresponding sequential implementation.

5.3.3 Canny edge detection method

For the Canny edge detection method there are three parameters that must be tuned in order to obtain optimal edge detection [*Canny*, 1986; *Heath et al.*, 1998]. The first one is the standard deviation σ of the Gaussian filter which controls the degree of smoothing and the kernel width. The second and third parameters are the values of the lower and upper hysteresis thresholds respectively. Various levels of smoothing were tested to identify the best smoothing Gaussian filter using filter kernels with σ levels of 1, 1.4, 2.0 and 2.5. The filtering kernels were computed using *Equation 2-5* to generate kernels of different sizes and the Gaussian filter with σ = 2.0 which have kernel size of 9 gave the best result of detected edges (Figure 5-5).

The optimal hysteresis parameters values for the high and low thresholds were estimated from the gradient magnitudes by setting them to 1st and 3rd quintiles (25% and 75%) of the magnitude. This gave a good quality detection of edges when implemented on a sequential mode but when this algorithm was implemented on MapReduce, the result was poor quality edges of the output image with considerable amount of falsely detected edges and noises in some regions of the image (Figure 5-6 left).

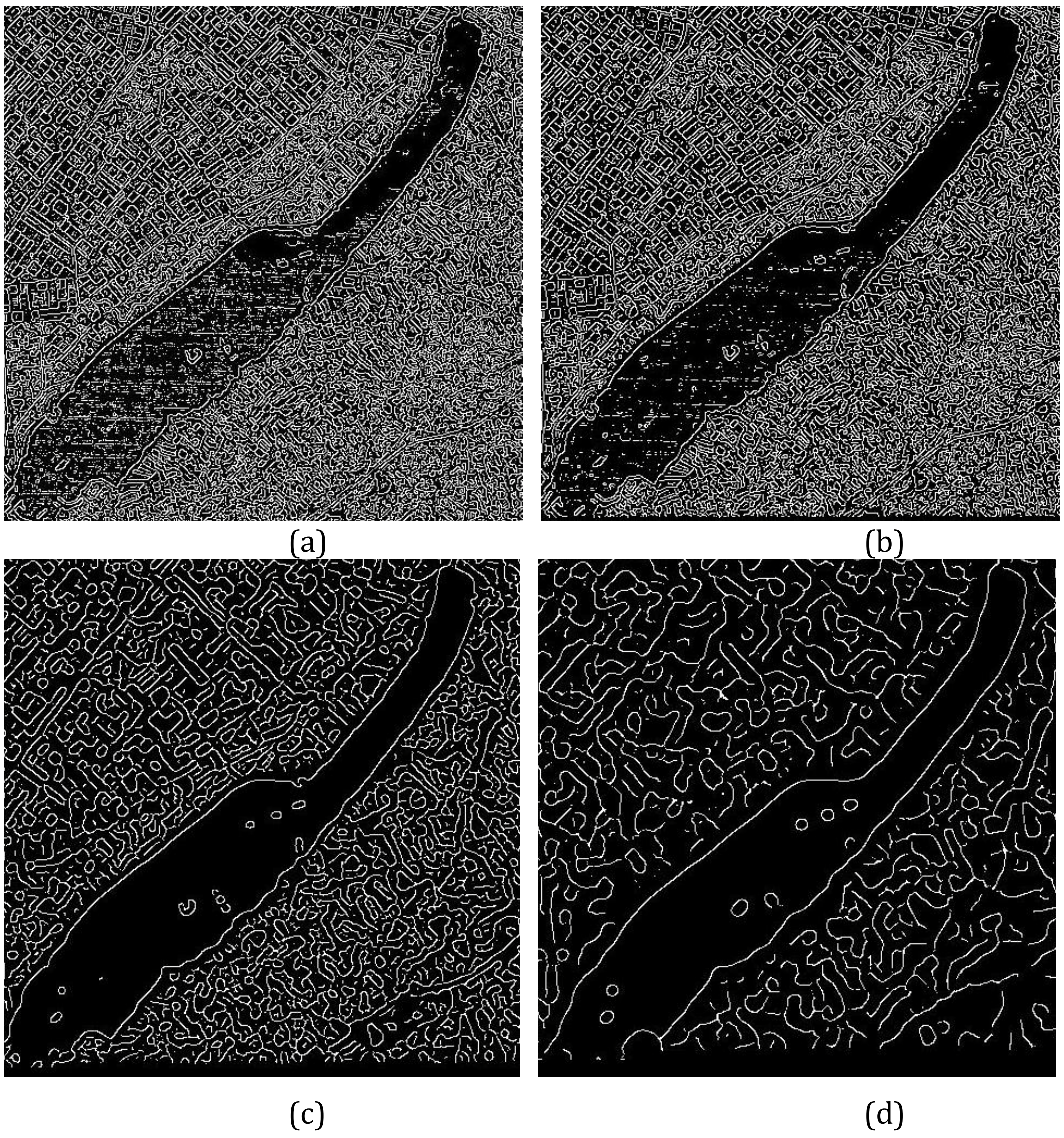


Figure 5-5: Edges detected by Canny method with Gausian filter of (a) σ=1, kernel size=3, (b) σ=1.4 kernel size=7, (c) σ=2.0 kernel size=9 (d), and σ=2.5 kernel size=16.

This is because the sub-images are processed independent of each other and estimating the upper and lower thresholds using the quantiles will result false detection of edges especially in those sub-images that have few actual edges. Therefore for the MapReduce implementation fixed higher and lower thresholds were set after doing some tuning and the best value of the high threshold for hysteresis was found to be 0.8 where gradient magnitudes above this threshold are unambiguously considered as edges. The optimal best value of the low threshold was found to be 0.3 and magnitudes below this level are considered as non-edges and set to zero. The magnitudes that have a value between 0.8 and 0.3 are considered as edges only if they are connected the edges that has been selected by the high threshold otherwise they are considered as non-edges.

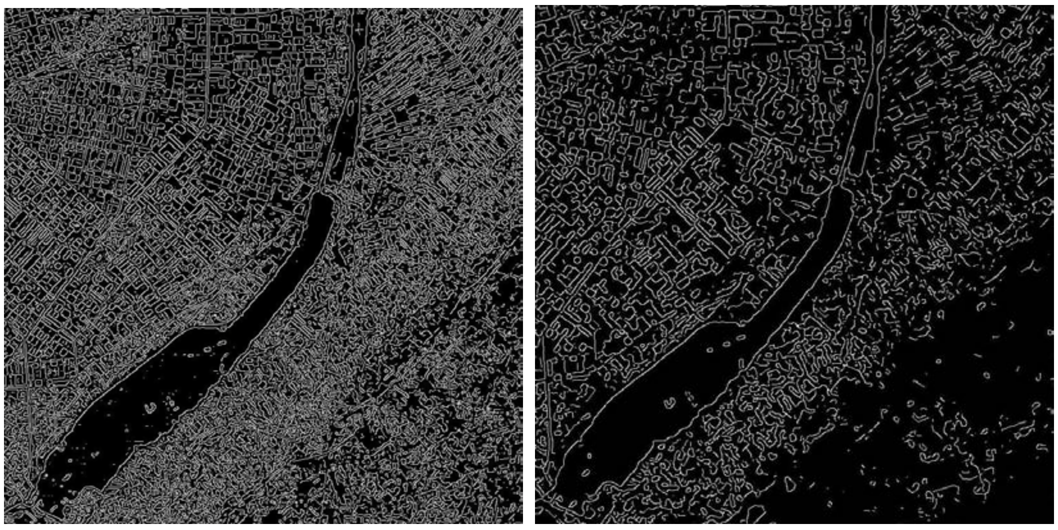


Figure 5-6: Edges detected using Canny method by quantiles parameterization of thresholds (left) and by fixed T_{high}=0.8, T_{low}=0.3 thresholds (right).

Generally the Canny method showed good edge detection, good localization but in some cases poor localizations were observed especially the edge corners and this is due to the use of Gaussian smoothing which blurs the borders of objects in an image.

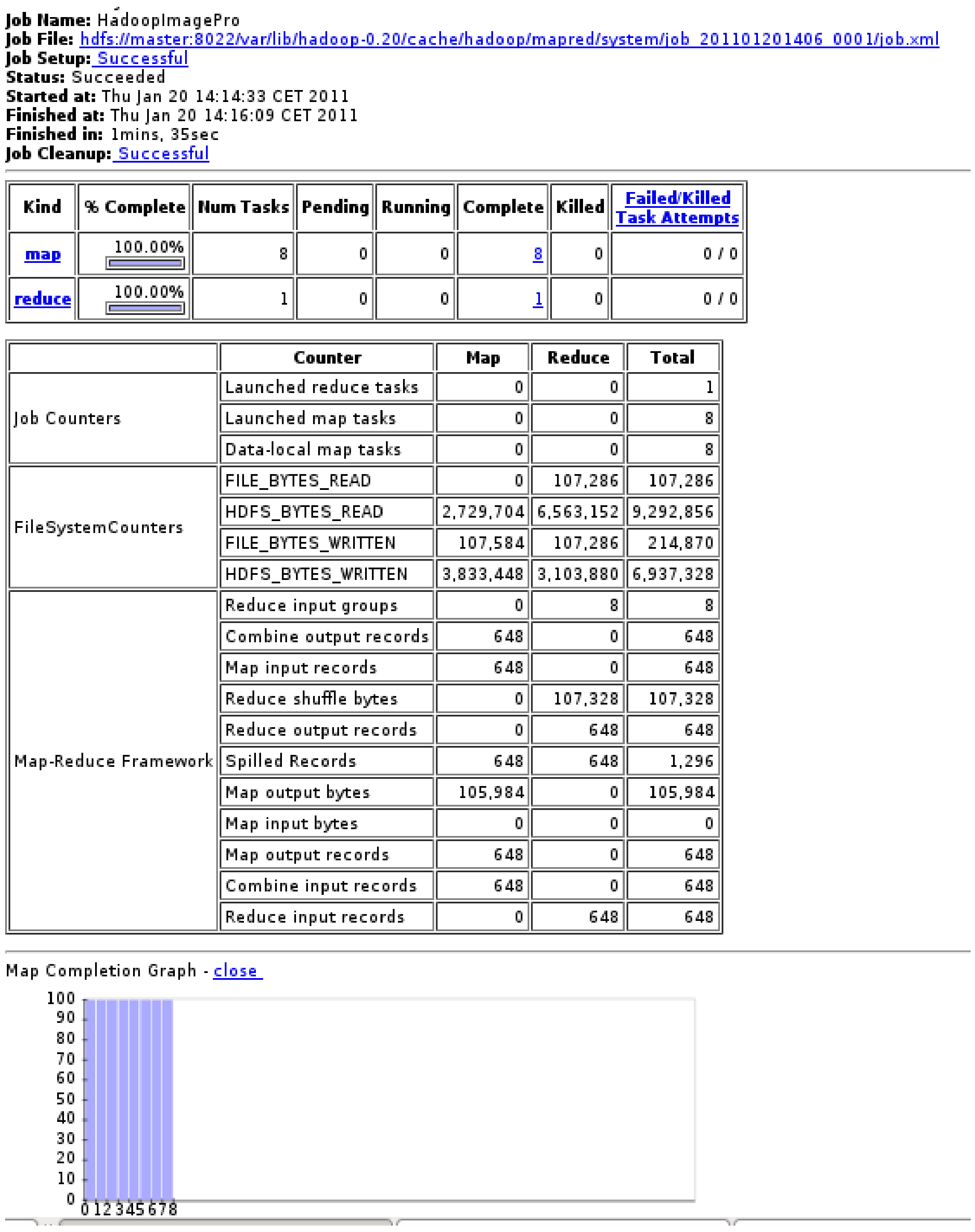
5.4 Performance Tests

The performance evaluation generally addresses the following parameters: the involved number of worker nodes, size of the satellite image, and the number of map and reduce jobs [*Ranger et al.*, 2007; *Winslett et al.*, 2009]. The principal metric for the evaluation of the computational performance is execution time of a job ordered by the master node. The performance in the job completion times were experimentally evaluated for the Sobel, Laplacian (with 5x5 kernel size) and the Canny detection methods using the above parameters.

Hadoop provides an integrated multi-paged web user interface where information about the job progress, completion time, failed tasks, job history and other statistics of the job can be tracked (Figure 5-7). The job completion time here means the total time consumed by the job for the worker nodes to process the input images and send the result to the master node. It must be noted that during the performance experiments the time of hour and day the job is executed have an impact on the computation time as the nodes are connected through WAN. Therefore, to minimize this effect of network latency and traffic, each job order was repeated more than 10 times and the average completion time is computed to capture the global performance of the algorithms. However, it is difficult to judge if the peak performance has been explored using this highly heterogeneous and few computing nodes but generally a good range of performance has been observed.

First a sequential implementation of the Sobel, Laplacian and Canny algorithms was made using java to obtain a control case for the performance evaluation MapReduce implementations. In the following sections, the performance of 1node denotes to the sequential implementation of the algorithms. To evaluate MapReduce programming model four performance metrics are used which are discussed in detail in the sections below [*Ranger et al.*, 2007; *Winslett et al.*, 2009]:

a. Number of nodes
b. Size of data,
c. Kernel sizes of the filtering.
d. Number of mappers and reducers.



Job Name: HadoopImagePro
Job File: hdfs://master:8022/var/lib/hadoop-0.20/cache/hadoop/mapred/system/job_201101201406_0001/job.xml
Job Setup: Successful
Status: Succeeded
Started at: Thu Jan 20 14:14:33 CET 2011
Finished at: Thu Jan 20 14:16:09 CET 2011
Finished in: 1mins, 35sec
Job Cleanup: Successful

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	8	0	0	8	0	0 / 0
reduce	100.00%	1	0	0	1	0	0 / 0

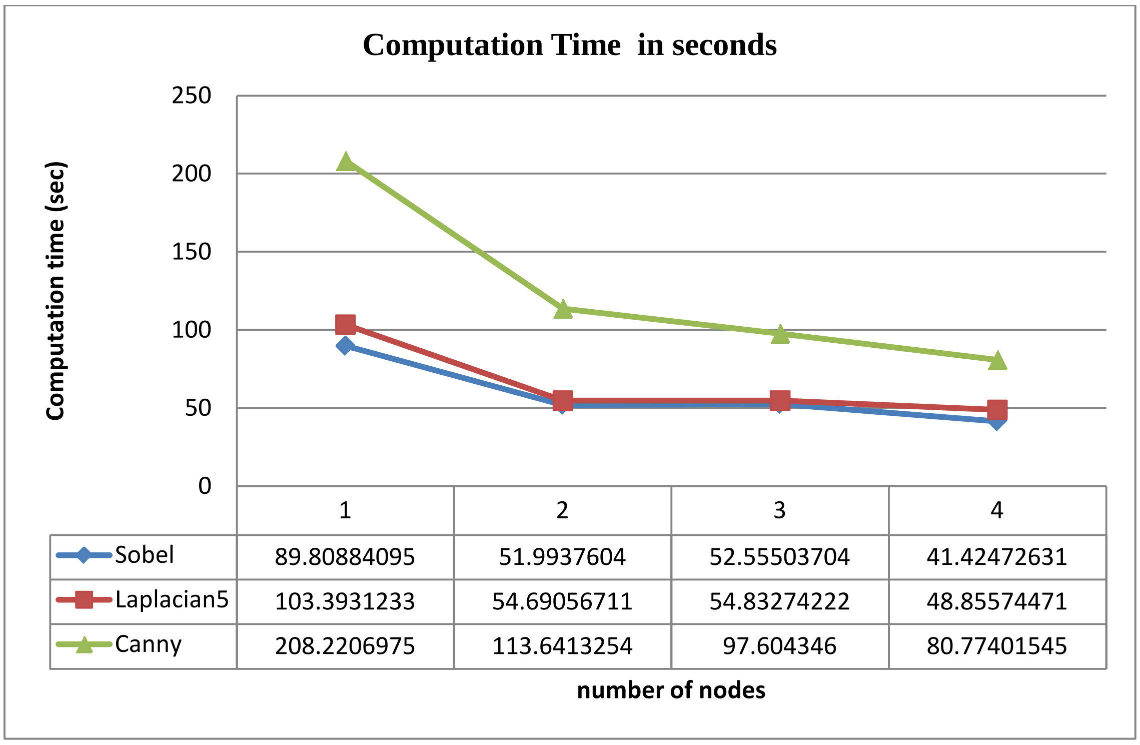
	Counter	Map	Reduce	Total
Job Counters	Launched reduce tasks	0	0	1
	Launched map tasks	0	0	8
	Data-local map tasks	0	0	8
FileSystemCounters	FILE_BYTES_READ	0	107,286	107,286
	HDFS_BYTES_READ	2,729,704	6,563,152	9,292,856
	FILE_BYTES_WRITTEN	107,584	107,286	214,870
	HDFS_BYTES_WRITTEN	3,833,448	3,103,880	6,937,328
Map-Reduce Framework	Reduce input groups	0	8	8
	Combine output records	648	0	648
	Map input records	648	0	648
	Reduce shuffle bytes	0	107,328	107,328
	Reduce output records	0	648	648
	Spilled Records	648	648	1,296
	Map output bytes	105,984	0	105,984
	Map input bytes	0	0	0
	Map output records	648	0	648
	Combine input records	648	0	648
	Reduce input records	0	648	648

Map Completion Graph - close

Figure 5-7: A screenshot of *Jobtracker* page at Hadoop web based interface.

5.4.1 Dependency on number of nodes

To measure the performance of the MapReduce implementations with regard to computation time, the total image size was kept constant while increasing the number of nodes in the cluster [*Xu et al.*, 1999]. We first evaluate the computation time on a single computer and then increase number of computers in the system until 4 computers. Figure 5-8 shows the computation time consumed to process the five image files each with a dimension of 8251x7591 pixels using the Sobel method, Laplacian method with 5x5 kernel size, and the Canny method with Gaussian filter of 9x9 kernel size. Those images are processed in parallel as a single MapReduce job. From this figure it can be observed that the computation time of the MapReduce implementation significantly decreases with increasing number of computers for all the algorithms. The Canny method achieved better performance compared to the Sobel and Laplacian methods. This is because generally the Canny method is more data-intensive and needs more processing power than the others since its algorithm is a multi-step process involving more mathematical operations compared to the Sobel and Lapalcian methods.



	1	2	3	4
Sobel	89.80884095	51.9937604	52.55503704	41.42472631
Laplacian5	103.3931233	54.69056711	54.83274222	48.85574471
Canny	208.2206975	113.6413254	97.604346	80.77401545

Figure 5-8: Computation performance for the Sobel, Laplacian, and Canny methods.

This observation can be clearly seen when we compute the speedup and efficiency from the above results. The speedup is measured as the ratio of the computation time of the sequential implementation on single computer to that of the distributed implementation on n computers while efficiency is the measure of the utilization of n computers in the system [*Eager et al.*, 1989].

$$\text{Speedup}(n) = \frac{\text{Time on single computer}}{\text{Time on } n \text{ computers}} \qquad Efficiency = \frac{Speedup(n)}{n}$$

The Speedup is described in terms of the number of computing nodes used in the distributed implementation and

Figure 5-9 and Figure 5-10 show the graph of the speedup and efficiency versus the number of nodes for the three edge detection methods respectively and it can be observed that generally a good speed up is achieved especially by the Canny method.

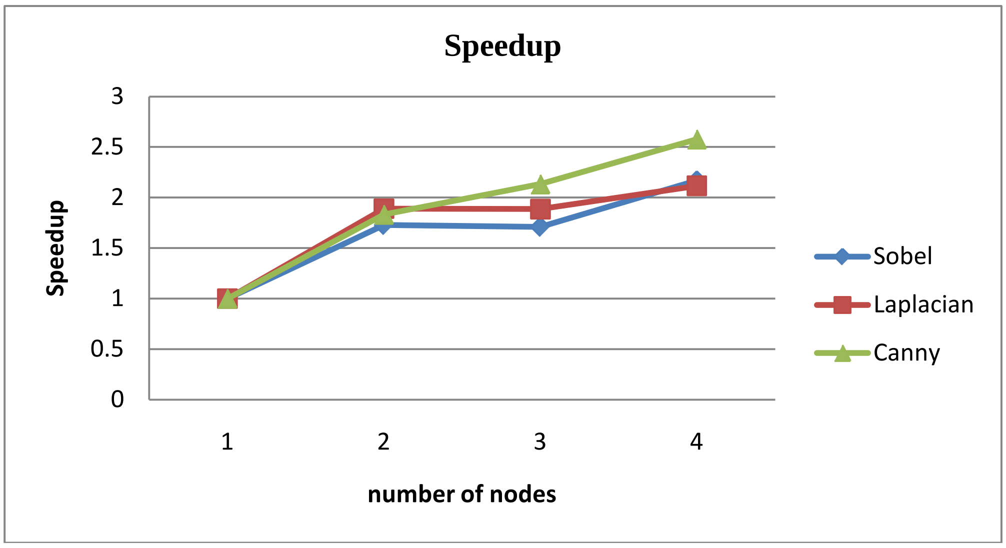


Figure 5-9: Speed up

In ideal situation the speedup increases linearly with increase in number of computing processors but this is difficult to achieve since the cost of communication increases with increase in processors [*DeWitt and Gray*, 1992]. According to [*Goller et al.*, 2001] processes with greater than 0.5 (50%) efficiency are considered to have achieved good performance. However, a significant deterioration of speed up and efficiency is observed when the third node was added. The reason for this deterioration is that the 2nd node connected to the 1st node (master server) through Local Area Network (LAN) dedicated to both computers while the 3rd and 4th node are connected to the master through Wide Area Network which is of lower speed. And also although there communication cost is not that much great during the MapReduce processing, a considerable amount is consumed due to the shuffling process after the finishing of the map processes especially for larger datasets. The speedup increased again when the 4th node is added to the system though not at the previous rate. From this it can be deduced that better performance can be achieved if the number of computing nodes are increased or if there is a high-speed network connection between the nodes.

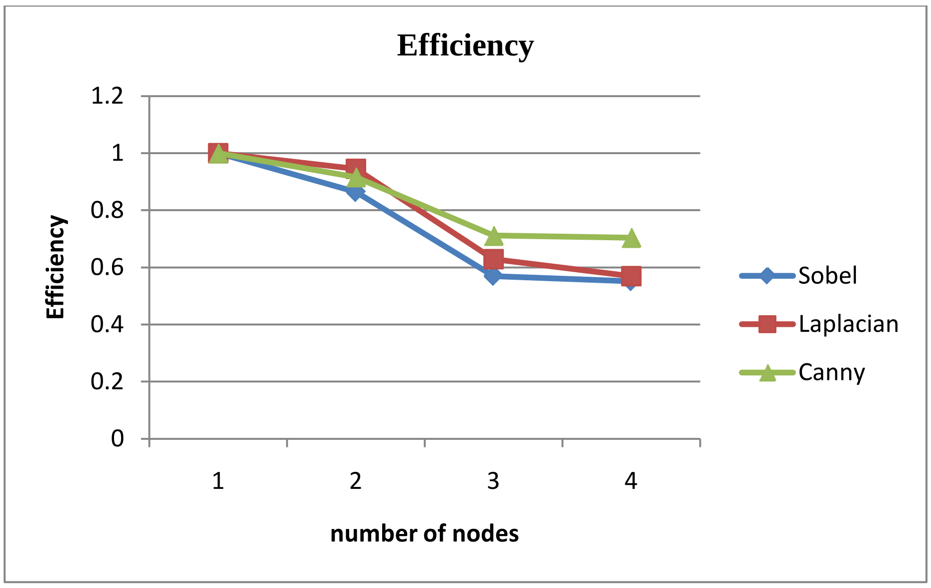


Figure 5-10: Efficiency

5.4.2 Dependency on size of the data

To investigate how the MapReduce programming model reacts to very large satellite images, the scaleup approach was evaluated where the computation time was measured while increasing the data size and number of nodes by the same fold. The concept of scaleup is to keep the amount of job on each computing node constant. By using the scaleup approach we can investigate the bottlenecks in the distributed system as the each computer node is processing the same amount of data and the effect load balancing problem and inefficiencies caused by the distributed method is avoided [*DeWitt and Gray*, 1992; *Goller et al.*, 2001]. This metric is a good way of evaluating the capability of the MapReduce implementation to cope with different sizes of data. Ideally the graph of scaleup will have a straight horizontal line where data size makes no impact with the computation time staying constant[*Goller et al.*, 2001]. In this experiment started with data size of 59.7 megabytes and increased the size by adding 59.7MB every time we add a computer node. Figure 5-11 shows the performance results and all the Sobel, Laplacian, and Canny methods scale well until the second computing node but it increases when the third computer is added to the system indicating that the network connection bandwidth is the main bottleneck of the distributed system. We can infer from this that had the datanodes been connected through high speed network, they would have scaledup well and the graph would have looked more flat.

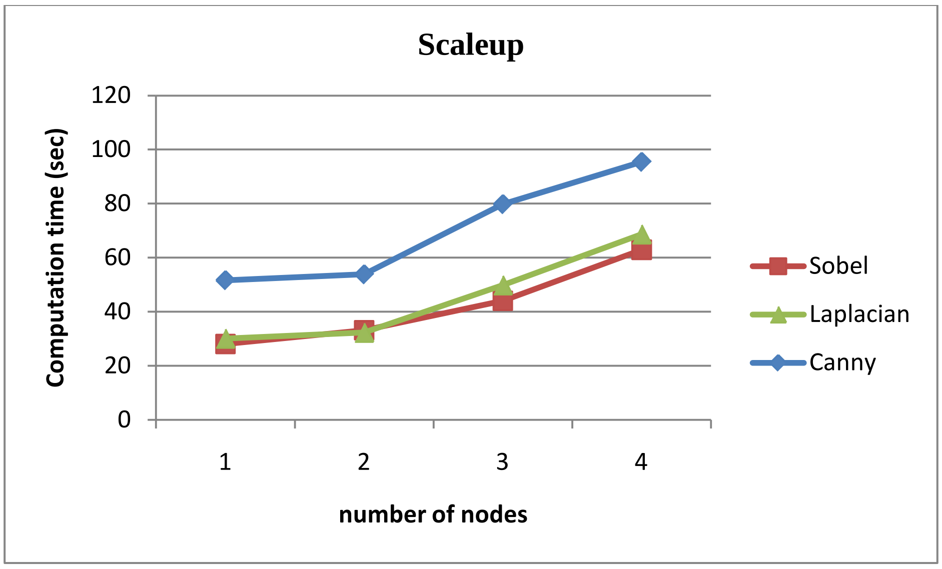


Figure 5-11: Scaleup

The sizeup approach was also explored for the Canny method where the number of computing nodes is held constant while the size of the data set is increased by some factor. We tested the computing performance for each group of nodes while increasing the dataset by doubling it starting from 59.7MB and increasing the size by 59.7 MB up to 358 MB.

Table 2-1 shows the elapsed computation time for different sizes of data and it can be seen that for processing on a single computer the computation time increases steeply while for the parallel processing the slope of increase is considerably lower. This can be further observed when the sizeup metric is computed from Table 5-2 using by the ratio of increased data size to the original data size which is 59.7MB for each of the computing nodes [*Xu et al.*, 1999].

$$\text{Sizeup} = \frac{\text{Time of increased data size}}{\text{Time of original data size}}$$

Table 5-2: performance of the Canny method with increasing data size.

	Computation time (sec)			
Data Size (MB)	1node	2node	3node	4node
59.7	86.16515	47.49669	39.62422	33.95762
119	174.5037	89.77618	85.88436	78.97992
179	249.6478	154.4281	128.4724	114.3664
239	351.1648	225.1332	214.3879	163.6579
298	439.9121	295.8607	272.3931	210.0497
358	537.3904	344.4746	338.3546	245.0497

Figure 5-12 shows the graph of sizeup for the Canny edge detection method and it can be observed that the sizeup is reasonable for all the nodes for considered the increase in communication cost with increase in dataset. If we see the example for the 4nodes cluster to process image data size that is 6 times larger than the original data 8 times more computation time is needed. From this it can be deduced that increasing the dataset achieves better performance and this is due to the fact that the processing time at the workers compensates the overheads at the startup of the process and network communications during shuffling process.

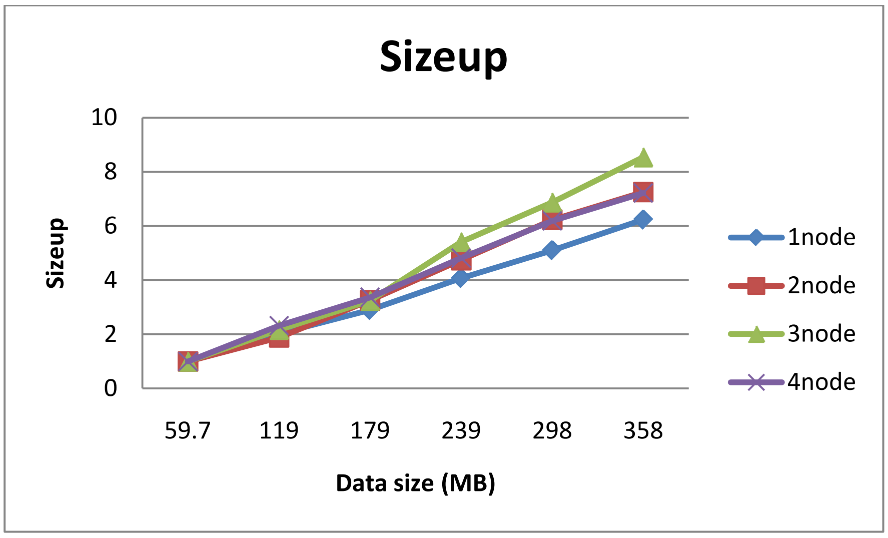


Figure 5-12: Sizeup for the Canny method.

5.4.3 Performance based on Neighborhood size

One observation seen during the performance tests of the edge detection algorithms implemented using MapReduce was that the computation time varies significantly with varying size of neighborhood of the convolution filter operators especially for the Canny method since smoothing step uses Gaussian filters of large kernel size. Therefore, performance evaluation was made to test dependence of the computation time on the neighborhood size. For this test the Canny method with Gaussian filters of 5x5, 7x7, 9x9, and 16x16 kernel size were evaluated on the 4node cluster and the results can be seen in Figure 5-13. When the test was done for two images each with a dimension of 8251x7591pixels, there is no significant difference in computation time between the different kernel sizes. But when the number of images was increased to 4 images to have a total data size of about 239MB, as significant difference in the elapsed computation time was observed with varying kernel size. And the computation time generally increases with increase in kernel size mainly due to the reason that more multiplication operations are involved with bigger kernels but also the number of overlapping pixels at the borders of image partitions also significantly higher.

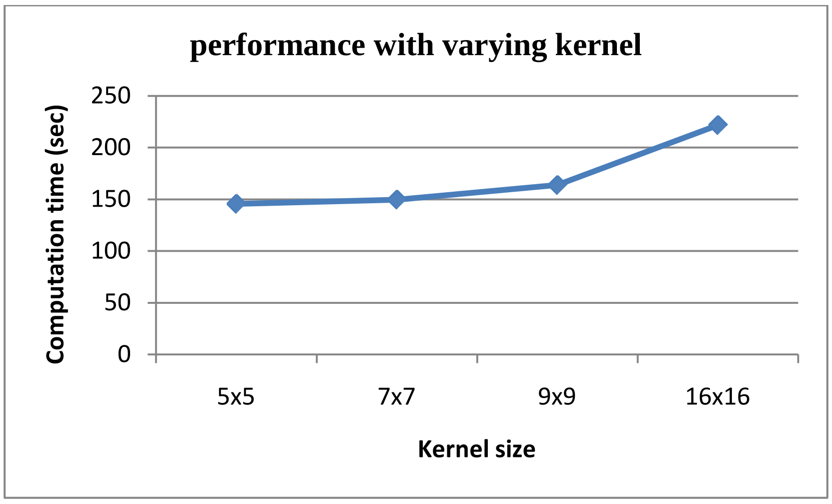


Figure 5-13: Performance of Canny method with varying Gaussian filters kernel size.

5.4.4 Dependency on number of mappers and reducers

In this experiment performance test was conducted to evaluate the dependency of the MapReduce algorithms on the number of mappers and reducers used to process an image of given size. Because in MapReduce we cannot directly control the number of mappers that is run by a specific job as is it determined by the number of image subsets, the performance was done by tuning the number of image subsets. To test the computation performance the dataset is kept constant while the numbers of mappers are increased from 4 up to 36. Figure 5-14 shows the computation time breakdown for the Canny method and it can be clearly seen that much of the computation time is consumed during the map phase while the shuffle (sort) and reduce phases share less of the total time. This is logical as the task of the reducer is only to shuffle and merge the output images from the mapper.

The important observation is this experiment is that the computation time greatly decreases with increase in the mappers up to 12 mappers but beyond that the performance deteriorates significantly. The reason for this is that, since we have four nodes each which is capable of running only 7 mappers in parallel, the mappers cannot be completed within on round and therefore an overhead occurs. It was expected that the performance will deteriorate as the number of mappers becomes greater than the number of nodes. But this decrease in performance did not occur instantly after the number of mappers surpassed the resources available, the deterioration occurred when the mappers are more than 12. This is mainly due the heterogeneous nature of the test environment where some nodes have significantly inferior computing power and Hadoop's dynamic load balancing and rescheduling mechanism handles this issue reasonably well in distributing the work efficiently. But eventually the performance deteriorates with greater number of mappers. This is because we are running more mappers for the same data size meaning the size of sub-images fed to each mapper are smaller. Hence, Hadoop is spending more time setting up the mappers than processing the input data.

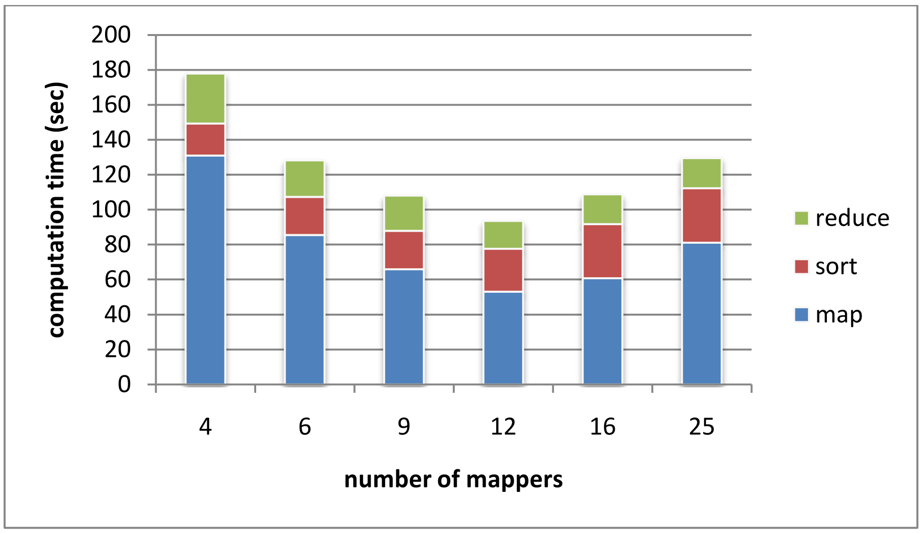


Figure 5-14: Computation time breakdown with increasing number of mappers for the Canny method.

Chapter 6 Conclusions and Future Works

6.1 Conclusions

The research undertaken is an effort to address one of the core issues in remote sensing studies –solving large-scale computational demanding problems in remote sensing image processing. Advances in sensor technology and their ever increasing repositories of the collected data are revolutionizing the mechanisms remotely sensed data are collected, stored and processed. This exponential growth of data archives and the increasing user's demand for real-and near-real time remote sensing data products has pressurized remote sensing service providers to deliver the required services. The remote sensing community has recognized the challenge in processing large and complex satellite datasets to derive customized products. To address this high demand in computational resources, several efforts have been made in the past few years towards incorporation of high-performance computing models in remote sensing data collection, management and analysis. This study adds an impetus to these efforts by introducing the recent advancements in distributed computing technologies, MapReduce programming paradigm, to the area of remote sensing. The general conclusions derived from this research and the arguments for the research questions raised in section 1.3 are discussed below.

Research question 1: Can performance of large satellite image processing be augmented through implementation on a distributed environment?

This question has been addressed in section 5.4 and based on the results of the performance evaluation conducted using standard metrics MapReduce programming paradigm has been proved to be a simple and efficient framework for large-scale processing of remote sensing images in low-cost distributed environments.

In the experiments done the edge detection algorithms scaled well when the volume of data to be processed is significantly large. This establishes that MapReduce is suitable for processing large-scale archives of remote sensing images and also high resolution images are usually massive in size. However further optimization strategies have to be explored also to increase the computational performance on smaller datasets.

Research question 2: What are the requirements to implement distributed processing of satellite images using the MapReduce programming model?

This question has been discussed in sections 2.4 and partially in chapter 3. Considering the complexity of distributed computing technologies, MapReduce have few prerequisites that must be satisfied in order to successfully implement image processing algorithms. The first one is to identify which algorithms to implement as not all computational problems related with remote sensing image processing can be solved through MapReduce especially those algorithms that need global communication between the distributed processors. It has been seen that the map tasks run concurrently in isolation with no communication between them.

The other issue to consider is that MapReduce was primarily designed for large scale text batch processing. Therefore there need to be some tasks to familiarize image formats to a MapReduce environment.

The third is that data must be large enough in order to achieve considerable performance improvement as have been observed in section 5.4.2.

The final requirement is regarding hardware configurations. It has been observed that high speed network connection between the nodes is more important than high processing power in the computing nodes. Therefore, the scale out approach seems a feasible option for distributed computing of large satellite images. Furthermore, MapReduce job scheduling and load balancing mechanism is based on the expectation that all data partitions will be computed equally fast, which is not practical when we have heterogeneous computing nodes. Therefore either the computing nodes should have similar processing power

Research question 3: What is the performance of the MapReduce processes compared to sequential processes?

This question is addressed in section 5.4 and the performance results show that MapReduce has improved task completions times reasonably well for most of the algorithms despite the test environment was made of only 4 computing nodes.

Research question 4: How do the different edge detection algorithms perform in a distributed environment?

This question is addressed in section 5.3 and 5.4 the performance of the Sobel, Laplacian, and Canny edge detection algorithms has been evaluated in two ways. The first one is in terms of the output image quality and the qualitative evaluation done demonstrates that no difference occurs in the quality of the output images from the distributed environment compared to the sequential implementation.

The second evaluation is performance in terms of execution time. There a significant difference is observed between the different edge detection algorithms with the Canny method performing well in terms of speedup and improvement as a result of implementation on distributed environment. The key conclusion that can be drawn from this observation is that the algorithms perform well if they are data intensive with high demand for local processing.

Research question 5: Which data partitioning and communication scheme is preferred in order to be executed concurrently?

This question is addressed in section 3.3 and 5.4. Data partitioning mechanisms of image data is one of the core issues that needs careful designing based on the algorithms are being implemented so that not to affect the quality of the final output images also loss of pixels at the borders. This is particularly important if the algorithms involve spatial transformations of the input image.

The other data partitioning consideration is with regard to processing in a distributed environment using MapReduce. As MapReduce is best at processing fewer and bigger data rather than many small sized images [*White*, 2009]; considerable care must be taken on the split size of the images in order to achieve optimal performance.

6.2 Future Works

This study is a preliminary work in an effort to integrate remote sensing and GIS applications and distributed computing using the MapReduce programming paradigm. Therefore there a lot of issue that can be improved in this work and also some future direction for distributed processing of spatial datasets using MapReduce. The following are some of the identified issues for further research.

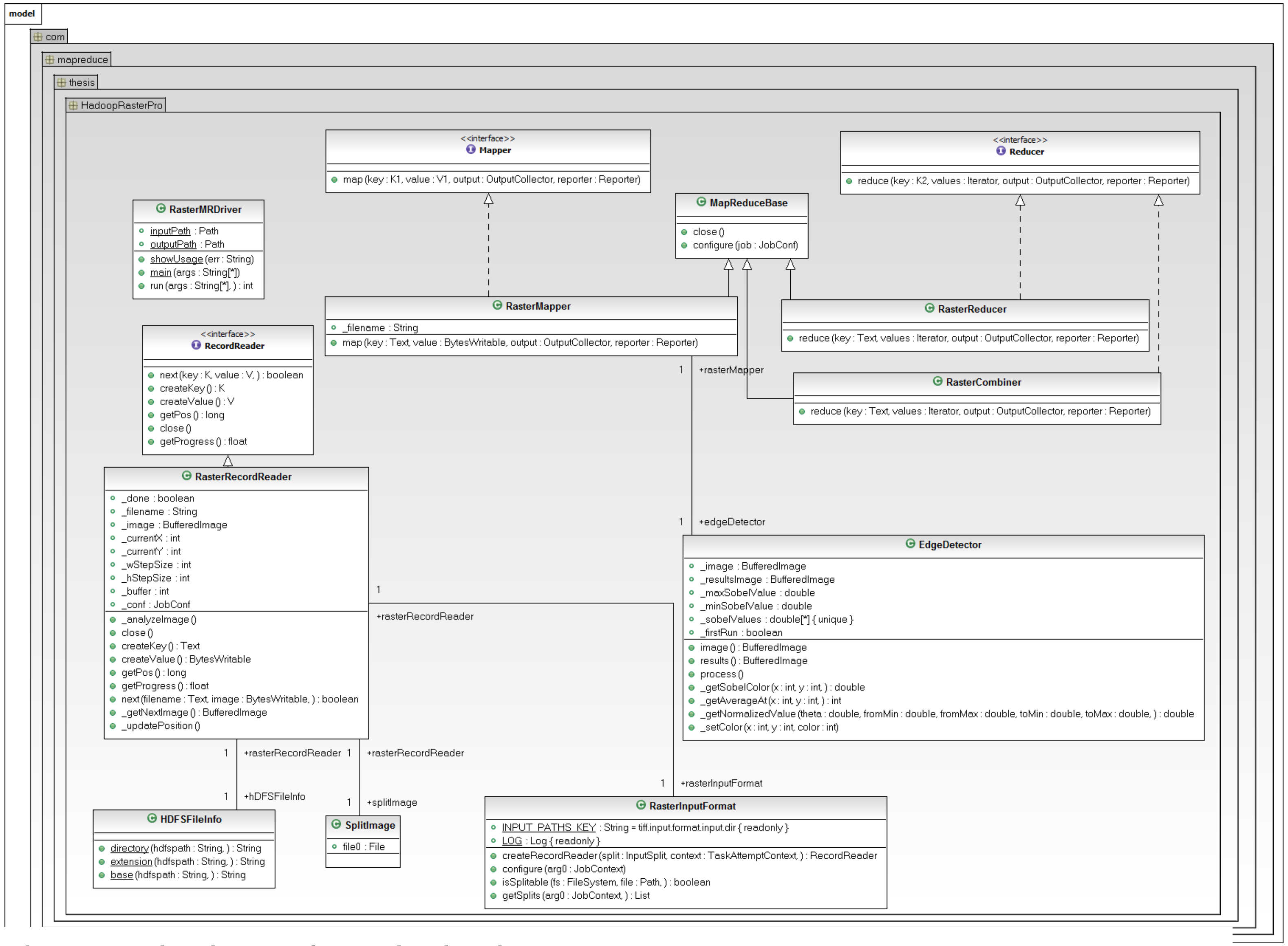
Further works can be done in this subject such as improving the scheduling mechanism for better load balancing and data locality and other optimization strategies such as intermediate values compression, serialization of the image input formats, proportion the java virtual memory used etc. It can also be investigated the impact of task granularity on the performance of the execution time and also dynamic load balancing.

This study have shown that MapReduce distributed programming paradigm is good for scaling the processing of large satellite images and has a substantial potential to support more complex remote sensing and GIS problems. Therefore, it is an interesting future direction to extend the MapReduce framework implemented in this study to include more complex and data-intensive algorithms and also remote sensing datasets from other sensor instruments.

Another future direction is to migrate these applications to high-end computing cluster and cloud platforms such as Amazon Elastic Compute Cloud (EC2) as Hadoop has already been implemented there.

It was also observed that no substantial modification has been made to the sequential algorithms for implementation in Hadoop MapReduce. Therefore it is highly suitable to use the codes of existing open-source remote sensing applications such as GRASS and QGIS. Hadoop can also be seamlessly integrated to other existing server based geoprocessing functionalities through Java API and Hadoop’s streaming features.

Appendices


model
com
mapreduce
thesis
HadoopRasterPro
<<interface>> Mapper
map (key : K1, value : V1, output : OutputCollector, reporter : Reporter)
<<interface>> Reducer
reduce (key : K2, values : Iterator, output : OutputCollector, reporter : Reporter)
MapReduceBase
close ()
configure (job : JobConf)
RasterMRDriver
inputPath : Path
outputPath : Path
showUsage (err : String)
main (args : String[*])
run (args : String[*],) : int
RasterMapper
_filename : String
map (key : Text, value : BytesWritable, output : OutputCollector, reporter : Reporter)
RasterReducer
reduce (key : Text, values : Iterator, output : OutputCollector, reporter : Reporter)
RasterCombiner
reduce (key : Text, values : Iterator, output : OutputCollector, reporter : Reporter)
<<interface>> RecordReader
next (key : K, value : V,) : boolean
createKey () : K
createValue () : V
getPos () : long
close ()
getProgress () : float
RasterRecordReader
_done : boolean
_filename : String
_image : BufferedImage
_currentX : int
_currentY : int
_wStepSize : int
_hStepSize : int
_buffer : int
_conf : JobConf
_analyzeImage ()
close ()
createKey () : Text
createValue () : BytesWritable
getPos () : long
getProgress () : float
next (filename : Text, image : BytesWritable,) : boolean
_getNextImage () : BufferedImage
_updatePosition ()
1 +rasterMapper
1 +edgeDetector
EdgeDetector
_image : BufferedImage
_resultsImage : BufferedImage
_maxSobelValue : double
_minSobelValue : double
_sobelValues : double[*] { unique }
_firstRun : boolean
image () : BufferedImage
results () : BufferedImage
process ()
_getSobelColor (x : int, y : int,) : double
_getAverageAt (x : int, y : int,) : int
_getNormalizedValue (theta : double, fromMin : double, fromMax : double, toMin : double, toMax : double,) : double
_setColor (x : int, y : int, color : int)
1 +rasterRecordReader
1 +rasterRecordReader 1 +rasterRecordReader
1 +rasterInputFormat
1 +hDFSFileInfo
1 +splitImage
HDFSFileInfo
directory (hdfspath : String,) : String
extension (hdfspath : String,) : String
base (hdfspath : String,) : String
SplitImage
file0 : File
RasterInputFormat
INPUT_PATHS_KEY : String = tiff.input.format.input.dir { readonly }
LOG : Log { readonly }
createRecordReader (split : InputSplit, context : TaskAttemptContext,) : RecordReader
configure (arg0 : JobContext)
isSplitable (fs : FileSystem, file : Path,) : boolean
getSplits (arg0 : JobContext,) : List

Appendix 1: UML2 Class diagram of MapReduce based Remote Sensing Image Processing

References

Abbas, A. (2007), *Grid Computing—A Practical Guide to Technology and Applications*, Design and Production Services, Inc.

Advanced Micro Devices (2009), OpenCL™ and the ATI Stream SDK v2.0., edited, AMD.

Allan, B. A., et al. (2006), A component architecture for high-performance scientific computing, *Int J High Perform C*, *20*(2), 163-202.

Aloisio, G., and M. Cafaro (2003), A dynamic earth observation system, *Parallel Comput*, *29*(10), 1357-1362.

Basumallik, A., S.-J. Min, and R. Eigenmann (2007), Programming Distributed Memory Sytems Using OpenMP, paper presented at International Parallel and Distributed Processing Symposium, IEEE.

Behnke, J., T. H. Watts, B. Kobler, D. Lowe, S. Fox, and R. Meyer (2005), EOSDIS Petabyte Archives: Tenth Anniversary, paper presented at 13th NASA Goddard Conference on Mass Storage Systems and Technologies, IEEE.

Bräunl, T. (2001), Tutorial in Data Parallel Image Processing, *Australian Journal of Intelligent Information Processing Systems*, *6*(3), 164–174.

Canny, J. (1986), A computational approach to edge detection, *IEEE Trans. Pattern Anal. Mach. Intell.*, *8*(6), 679-698.

Chiarabini, L., and J. Yen (1998), Complexity reduction on two-dimensional convolutions for image processing, SPIE, San Jose, CA, USA.

Cossu, R., E. Schoepfer, P. Bally, and L. Fusco (2009), Near real-time SAR-based processing to support flood monitoring, *J Real-Time Image Pr*, *4*(3), 205-218.

Coulouris, G. F., J. Dollimore, and T. Kindberg (2005), *Distributed systems : concepts and design*, 4th ed. ed., xiv, 927 p. pp., Addison-Wesley, Harlow.

Cristian, F. (1991), Understanding Fault-Tolerant Distributed Systems, *Commun Acm*, *34*(2), 56-78.

Dean, J., and S. Ghemawat (2008), Mapreduce: Simplified data processing on large clusters, *Commun Acm*, *51*(1), 107-113.

Dean, J., and S. Ghemawat (2010), MapReduce: A Flexible Data Processing Tool, *Commun Acm*, *53*(1), 72-77.

DeWitt, D. J., and J. Gray (1992), Parallel Database Systems: The Future of High Performance Database Processing, *Commun Acm*, *36*(6).

Drewniok, C. (1994), Multi-spectral edge detection. Some experiments on data from Landsat-TM, *Int J Remote Sens*, *15*(18), 3743-3765.

Eager, D. L., J. Zahorjan, and E. D. Lazowska (1989), Speedup versus efficiency in parallel systems, *Computers, IEEE Transactions on*, *38*(3), 408-423.

Fisher, R., S. Perkins, A. Walker, and E. Wolfart (1996), *Hypermedia Image Processing Reference*, JOHN WILEY & SONS LTD, New York.

Foster, I., C. Kesselman, and S. Tuecke (2001), The anatomy of the grid: Enabling scalable virtual organizations, *Int J High Perform C*, *15*(3), 200.

Ghemawat, S., H. Gobioff, and S.-T. Leung (2003), The Google file system, paper presented at 9th ACM symposium on Operating systems principles.

Gibara, T. (2009), Canny Edge Detector Implementation, edited.

Goller, A., I. Glendinning, D. Bachmann, and R. Kalliany (2001), Parallel and Distributed Processing, in *Digital Image Analysis*, edited by W. Kropatsch and H. Bischof, pp. 135-153, Springer New York.

Golpayegani, N., and M. Halem (2009), Cloud Computing for Satellite Data Processing on High End Compute Clusters, in *IEEE International Conference on Cloud Computing*, edited.

Goward, S. N., J. G. Masek, D. L. Williams, J. R. Irons, and R. J. Thompson (2001), The Landsat 7 mission - Terrestrial research and applications for the 21st century, *Remote Sens Environ*, *78*(1-2), 3-12.

HARRIS, M. (2005), Mapping computational concepts to GPUs, paper presented at GPU Gems 2, Addison-Wesley.

Hawick, K. A., P. D. Coddington, and J. H. A. (2003), Distributed frameworks and parallel algorithms for processing large-scale geographic data, *Parallel Comput*, *29*(10).

He, B., W. Fang, Q. Luo, N. Govindaraju, and T. Wang (2008), Mars: a MapReduce framework on graphics processors, ACM.

Heath, M., S. Sarkar, T. Sanocki, and K. Bowyery (1998), Comparison of Edge Detectors: A Methodology and Initial Study, *Computer Vision and Image Understanding*.

Joseph, J., and C. Fellenstein (2004), *Grid computing*, Prentice Hall PTR.

Karavakis, E. (2010), A Distributed Analysis and Monitoring Framework for the Compact Muon Solenoid Experiment and a Pedestrian Simulation, 216 pp, Brunel University.

Kittler, J. (1983), On the accuracy of the Sobel edge detector, *Image and Vision Computing*, *1*(1), 37-42.

Kshemkalyani, A. D., and M. Singhal (2008), *Distributed computing : principles, algorithms, and systems*, xvii, 736 p. pp., Cambridge University Press, Cambridge.

Lillesand, T. L., and R. W. Kiefer (2001), *Remote Sensing and Image Interpretation*, 4 ed., John Wiley & Sons, Inc., New York.

Lin, J., and C. Dyer (2010), Data-Intensive Text Processing with MapReduce, *Synthesis Lectures on Human Language Technologies*, *3*(1), 1-177.

Liu, H., and K. Jezek (2004), Automated extraction of coastline from satellite imagery by integrating Canny edge detection and locally adaptive thresholding methods, *Int J Remote Sens*, *25*(5), 937-958.

Lv, Z., Y. Hu, H. Zhong, J. Wu, B. Li, and H. Zhao (2010), Parallel K-Means Clustering of Remote Sensing Images Based on MapReduce, *Springer-Verlag Berlin Heidelberg*, 162–170.

Mann, K., and T. Jones (2008), Distributed computing with Linux and Hadoop, edited, IBM Corporation.

Marr, D., and E. Hildreth (1980), Theory of Edge Detection, *Royal Society of London Proceedings Series B*, *207*, 187-217.

NASA (2007), *Earth System Science Data Resources*, National Aeronautics and Space Administration.

NASA (2010), Landsat 7 Science Data User's Handbook.

Noll, M. G. (2004-2010), Running Hadoop On Ubuntu Linux (Single-Node Cluster), edited, http://www.michael-noll.com/wiki/Running_Hadoop_On_Ubuntu_Linux.

NVIDIA (2010), CUDA (Compute Unified Device Architecture), edited.

OWENS, J. D., D. LUEBKE, N. GOVINDARAJU, M. HARRIS, J. KR¨UGER, A. E. LEFOHN, and T. J. A. PURCELL (2007), Survey of general-purpose computation on graphics hardware, in *Computer Graphics Forum*, edited.

Parkinson, C. L., A. Ward, and M. D. King (2006), *Earth Science Reference Handbook: A Guide to NASA's Earth Science Program and Earth Observing Satellite Missions*, National Aeronautics and Space Administration, Washington D.C.

Pike, R., S. Dorward, R. Griesemer, and S. Quinlan (2005), Interpreting the data: Parallel analysis with sawzall., *Sci.Prog.*, *13*(4).

Plaza, A. J., and C.-I. Chang (2008), *High performance computing in remote sensing*, xxvi, 466 p. pp., Chapman & Hall/CRC, Boca Raton, Fla. ; London.

Ranger, C., R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis (2007), Evaluating MapReduce for multi-core and multiprocessor systems, paper presented at 13th International Symposium on High-Performance Computer Architecture (HPCA 2007), Phoenix, Arizona.

Richards, J. A., and X. Jia (2006), *Remote sensing digital image analysis : an introduction*, 4th ed. ed., xxv, 439 p. pp., Springer, Berlin.

Schmidt, S. (2009), The Holumbus Framework: Distributed computing with MapReduce in Haskell, 103 pp, FHWedel University of Applied Sciences, Schenefeld, Germany.

Schowengerdt, R. (2007), *Remote sensing: models and methods for image processing*, Academic Pr.

Shen, Z., J. Luo, G. Huang, D. Ming, W. Ma, and H. Sheng (2006), Distributed computing model for processing remotely sensed images based on grid computing, *Elsevier Inc.*, *177*, 504-518.

Silva, V. (2006), *GRID COMPUTING FOR DEVELOPERS*, Charles River Media, Inc., Hingham, Massachusetts.

Sterling, T., D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. (1995), BEOWULF : A parallel workstation for scientific computation, paper presented at In International Conference on Parallel Processing, CRC Press, Boca Raton, USA.

Tanenbaum, A. S., and M. v. Steen (2007), *Distributed systems : principles and paradigms*, 2nd ed. ed., xviii, 686 p. pp., Prentice Hall, Harlow.

Teo, Y. M. (2003), Distributed Geo-Rectification of Satellite Images Using Grid Computing.

Tesfamariam, E. (2010), Efficient Satellite Image Filtering Algorithms on Graphics Processors using OpenCL, in *Geomundus Symposium*, edited, Castellon de la plana, Spain.

Torre, V., and T. A. Poggio (1986), On Edge Detection, *Pattern Analysis and Machine Intelligence, IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8*(2), 147-163.

Valencia, D., P. Mart´inez, A. Plaza, and J. Plaza (2008), Parallel Wildland Fire Monitoring and Tracking Using Remotely Sensed Data, in *High Performance Computing In Remote Sensing*, edited, pp. 151-182, Chapman & Hall/CRC, Boca Raton, Fla.; London.

Venner, J. (2009), *Pro Hadoop*, Apress, Bekerley, CA.

Votava, P., R. Nemani, K. Golden, D. Cooke, H. Hernandez, and C. Ma (2002), Parallel distributed application framework for earth science data processing, paper presented at International Geoscience and Remote Sensing Symposium (IGARSS), IEEE, Toronto, Canada.

Votava, P., R. Nemani, C. Bowker, A. Michaelis, A. Neuschwander, and J. Coughlan (2002), Distributed Application Framework for Earth Science Data Processing, *Ieee Commun Mag*.

Wang, R. (2009), Sharpening and Edge Detection, edited, http://fourier.eng.hmc.edu/e161/lectures/gradient/node9.html.

White, T. (2009), *Hadoop : the definitive guide*, xix, 501 p. pp., O'Reilly, Beijing.

Wiley, K., A. Connolly, J. Gardner, and S. Krughoff (2010), Astronomy in the Cloud: Using MapReduce for Image Coaddition, *University of Washington*, *http://arxiv.org/abs/1010.1015v1*.

Winslett, M., A. Cary, Z. Sun, V. Hristidis, and N. Rishe (2009), Experiences on Processing Spatial Data with MapReduce, in *Scientific and Statistical Database Management*, edited, pp. 302-319, Springer Berlin / Heidelberg.

Xu, X., J. J, and H.-P. Kriegel (1999), A Fast Parallel Clustering Algorithm for Large Spatial Databases, *Data Min. Knowl. Discov.*, *3*(3), 263-290.

Yao, K.-T., R. F. Lucas, C. E. Ward, G. Wagenbreth, and T. D. Gottschalk (2009), Data Analysis for Massively Distributed Simulations, in *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*, edited, pp. 1-7.

Ziou, D., and S. Tabbone (1993), A multi-scale edge detector, *Pattern Recognition*, *26*(9), 1305-1314.

Masters Program in **Geospatial Technologies**

Supported by:

Education and Culture