Programming Abstractions for Dynamic, Distributed, Data-intensive Computing

Vinay Sudhakaran

August 19, 2011

MSc. in High Performance Computing

The University of Edinburgh

Year of Presentation: 2011

Abstract

Processing large volumes of scientific data requires an efficient and scalable parallel computing framework to obtain meaningful information quickly. MapReduce is a programming model and an execution framework introduced by Google Inc. to facilitate processing of large datasets (Dean and Ghemawat 2004). Since its inception, MapReduce has found widespread adoption for processing massive amounts of textual data such as web pages, web request logs and crawled documents but only recently has it gained significant attention from various scientific disciplines for analyzing large volumes of scientific data.

In many cases, scientific applications may have characteristics which make it harder to apply programming models like MapReduce. The data to be analysed may come from many different sources, may be unevenly distributed or may be updated or superseded by new data. The application itself may display signs of being I/O bound or memory-bound as well as being CPU-bound.

In this dissertation, we evaluate a scientific application from the environmental sciences (*ccc-gistemp* – Python reimplementation of the original NASA GISS model for estimating the global temperature change) for its applicability to use the MapReduce framework, specifically the Hadoop implementation of MapReduce. The application consists of several stages, each of which display differing dynamic, distributed and data-intensive characteristics, and we examine the code to determine how to parallelise data and compute intensive tasks efficiently.

Three stages of the *ccc-gistemp* code have been ported using Hadoop and *mrjob* library (Python interface for Hadoop streaming jobs). Step1 is trivially data-parallel and well suited to MapReduce; Step2 is both data and compute intensive and appears suited to MapReduce but reveals issues due to uneven distribution of input data; and step3 does not initially appear to suit MapReduce but can be successfully ported by slight modification of the data access pattern. Performance bottlenecks encountered while porting and possible solutions for their resolution are highlighted, with benchmarking carried out on two commodity clusters: the St. Andrews cloud infrastructure and the University of Edinburgh EDIM1 data-intensive research machine.

This work has shown that:

1. MapReduce provides the necessary programming abstraction for parallelising data and compute intensive steps of a scientific application code.
2. The MapReduce programming framework is capable of handling applications with varying indices along the three axes of being dynamic, distributed and data-intensive.
3. Implementations of MapReduce programming model on infrastructures consisting of low-end commodity machines are cost-effective and efficient for data-intensive computing.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my project supervisor Mr. Neil Chue Hong for his advice and supervision during the project.

I would also like to extend my gratitude to Mr. Gareth Francis (EPCC) and Dr. Paolo Besana (School of Informatics, University of Edinburgh) for their assistance in setting up the Hadoop cluster on EDIM1 machine. Furthermore, I would like to thank Dr. Yuanzhi (Derek) Wang and Dr. Adam Barker at the University of St. Andrews for their willingness and support in using the St. Andrews cloud infrastructure during development and initial testing phases of this project.

I am also indebted to my family and friends who have always stood by me in every choice I make.

Chapter 1

Introduction

The ability to create rich, detailed models of natural and artificial phenomena and to process large volumes of experimental data created by new generation of scientific instruments that are themselves powered by computing, highlights the importance of computational science as a critical enabler of scientific discovery (Hey, Tansley and Tolle 2009).

1.1 Motivation and Context

Advancing the state-of-the-art research in computational science require analyses of large volumes of data collected from numerous scientific instruments and experiments conducted around the globe. Petabyte data sets are already becoming increasingly common in many High End Computing (HEC) applications from a diverse range of scientific disciplines (Mackey, et al. 2008), and this is only expected to grow in the near future. An exemplar to this is the Large Hadron Collider (LHC) which is estimated to produce roughly 15 petabytes of data a year when it is fully operational[1]. This necessitates the need for providing abstraction[2] at multiple levels for *acquiring, managing and processing* of data (Hey, Tansley and Tolle 2009), thus enabling the scientific community to focus on 'science' rather than disentangling the complexities involved in setting up and maintaining the cyber-infrastructure required to facilitate data intensive computing.

Processing the large volumes of data quickly requires efficient parallel programming models that meet the performance requirements entailed by these scientific applications. Several attributes need be considered before selecting a suitable methodology for parallelising data-intensive applications which include data volumes, data access patterns in the algorithm, computational requirements, task sharing and global synchronisation constraints, scalability, ease of programming and the underlying execution infrastructure. The main focus of this work is to evaluate if MapReduce, specifically Hadoop implementation of MapReduce, can provide the necessary high level parallel programming framework that is required for *parallelising* data and compute intensive tasks of a scientific application from the environmental sciences.

GISS Surface Temperature Analysis (GISTEMP)[3] is an open-source model from the environmental sciences for estimating the global temperature change, implemented by NASA Goddard Institute of Space Studies (GISS). GISTEMP was originally written in FORTRAN.

[1] http://public.web.cern.ch/public/en/LHC/Computing-en.html
[2] Abstraction commonly refers to the way of representing data or computation at a higher-level, hiding the underlying complexity.
[3] http://www.giss.nasa.gov

ccc-gistemp is a part of the Clear Climate Code (CCC) project from Climate Code Foundation to re-implement the NASA GISS Gistemp algorithm in Python. In this project, *ccc-gistemp* is ported to the MapReduce framework using *mrjob* API to provide the python interface for Hadoop streaming jobs.

The St. Andrews cloud infrastructure (StACC) and the University of Edinburgh EDIM1 machine are used extensively for development and testing of the ported code. EDIM1[4] is a cluster of commodity machines jointly funded by the Edinburgh Parallel Computing Center (EPCC) and the School of Informatics, primarily intended for Data-intensive research. Performance evaluations have been done on this machine configured as a sixteen node cluster (one master node, one job tracker and fourteen slave nodes).

Map-Reduce is a programming model introduced by Google Inc. to support distributed computing on large volumes of data, using clusters of commodity machines (Dean and Ghemawat 2004). Firstly, this parallel data-processing model is understood as a means of providing abstraction for distributed, data-intensive computing. Secondly, from an architectural perspective, the existing *ccc-gistemp* application code is reviewed to verify its applicability to be ported to the MapReduce framework. Data and compute-intensive steps within the code that are likely to improve the overall performance when ported are identified. Thirdly, the identified steps are ported to the MapReduce programming model in small increments using iterative technique (agile development practises), while noting performance and verifying the correctness of the algorithm. Lastly, the ported code is benchmarked to evaluate performance and scalability.

1.2 Related Work

MapReduce is extensively used within Google for processing large volumes of raw data such as crawled documents and web request logs (Dean and Ghemawat 2004). With its widespread adoption via an open source implementation called Hadoop[5] (Lin and Dyer 2009), primarily for data-intensive computing, there have been many evaluations of this programming model using large volumes of web and textual data. However, there have been only a few evaluations with scientific data (Ekanayake, Pallickara and Fox 2008).

(Zhu, et al. 2009), evaluated the feasibility of porting two applications (Water Spatial[6] and Radix Sort[7]) from the Stanford SPLASH-2[8] suite to the Hadoop implementation of MapReduce. Performance bottlenecks with porting were identified and suggestions provided for enhancing the MapReduce framework to suite these applications.

[4] http://www.epcc.ed.ac.uk/projects/research/dataintensive
[5] (Apache Hadoop framework 2008)
[6] Water Spatial is an N-body molecular dynamics application that evaluates the forces and potentials over time in a cluster of water molecules in liquid state (Zhu, et al. 2009).
[7] Integer sorting technique implemented as an iterative algorithm.
[8] The Splash-2 Suite consists of a set of complete applications and computational kernels specifically designed to facilitate the study of centralized and distributed shared-address space multi-processors (Zhu, et al. 2009).

The main attributes of the implementation strategy that were considered in porting these applications were the data access patterns and computational steps. It was identified that most scientific applications require shared data and hence synchronisation was a major source of overhead. Additionally, the probability of scientific applications using matrices and multi-dimensional arrays for their processing was much higher than simple data-structures.

Global synchronisation across all reduce tasks in a MapReduce job was achieved with a single reduce task. Suggestions to provide better support for distributing array and matrices within the HDFS to reduce communication overheads were made. Also, the advantages of directly dumping the output of first stage to the second in a multi-stage job, without the need for intermediate HDFS store were highlighted to reduce I/O overheads.

(Ekanayake, Pallickara and Fox 2008), evaluated Hadoop implementation of MapReduce with High Energy Physics data analysis. The analyses was conducted on a collection of data files produced by high-energy physics experiments, which is both data and compute intensive. As an outcome of this porting, it was observed that scientific data analyses that has some form of SPMD[9] (Single-Program Multiple Data) algorithms are more likely to benefit from MapReduce when compared to others. Additionally, MapReduce implemenations were scalable with the increase in data volume and computational nodes, minimizing the impact of overheads.

However, the use of iterative algorithms required by many scientific applications were seen as limitation to the existing MapReduce implementations. It was suggested that support for directly accessing data in binary format could benefit many scientific applications which would otherwise need some form of data transformation, reducing performance.

In this work, we study an application code from the environmental sciences to evaluate its applicability to be used with the MapReduce framwork. Implementation stratregies such as data access patterns in the algorithm, task sharing and global synchronisation constraints, scalabilty and ease of programming are considered for evaluation. Programming abstractions at various levels of execution are introduced and studied.

1.3 Organisation of the Dissertation

The rest of the dissertation work is organised as follows:

Chapter 2: A brief overview of the application domain and GISS surface temperature analysis is provided.

Chapter 3: ccc-gistemp is introduced along with a discussion on the merits of using this code for programmers.

Chapter 4: Cloud computing and the use of cloud based resources to support data-intensive computing are discussed.

[9] SPMD is a technique employed to achieve parallelism where the tasks are split and run simultaneously on multiple processors with different input data (http://en.wikipedia.org/wiki/SPMD).

Chapter 5: MapReduce framework and the Hadoop implementation of MapReduce are discussed in detail. *mrjob*, the Python interface for Hadoop streaming job is introduced along with discussion on the merits of using it in this work. An example of MapReduce programming model is provided to assist in better understanding of the overall framework.

Chapter 6: The development infrastructure which includes the execution and test environment, project hosting and the software development methodology is introduced.

Chapter 7: "In-depth" analyses of each of the steps in the original *ccc-gistemp* code are provided while identifying patterns that are suitable to be ported to the MapReduce programming model. Issues encountered while porting are discussed with possible solution for its resolution. Test cases that are executed to ensure correctness of the ported code are summarised.

Chapter 8: Performance analyses carried out on the ported *mapreduce-cccgistemp* code is reported in this chapter.

Chapter 9: The dissertation concludes by outlining some suggestions for future work. Risks that were initially identified are reviewed to assess its final impact on the completion of this work.

Chapter 2

Background

This chapter provides an overview of the three commonly used terms in this work – Dynamic, Distributed and Data-intensive. Additionally, the application from environmental sciences (GISTEMP) that is considered for porting is introduced along with its significance for estimating long term global temperature changes.

2.1 Dynamic, Distributed, Data-intensive

Technology - encompassing computers as single entity capable of efficiently storing, managing and processing data; high-speed network infrastructure supporting the Internet; software platforms and applications, has played an important role in supporting collaboration, sharing of information across geography and data management within the context of a research project.

An application can be classified along the three axes of being dynamic, distributed and data-intensive (3D). When the *processing* involves large volumes of data, called "big-data", the application is said to be *data-intensive* (E.g. the European Bioinformatics Institute (EBI)[10] holds a central repository of DNA sequence data, amounting to nearly 5 petabytes which is accessible by scientists and researchers around the globe). Much of this data arises from many different biomedical research centres scattered across the globe, and hence the genome processing involves *distributed* data. New DNA sequences are added frequently for improved genome analysis. Coping with this variability in the input data-stream constitutes the dynamicity of the application and hence *dynamic*.

An application may have varying indices along these three axes. For example, an application may process large volumes of data but the input data-set may be relatively static. An ideal programming abstraction should provide capability to handle varying indices along these three axes of being *Dynamic, Distributed and Data-intensive*.

2.2 Earth and Environmental Sciences

Earth and environmental sciences offer tremendous opportunities and challenges for data-intensive computing. Sensors and scientific instruments monitoring our planet from deep within the ocean to space based high resolution satellite imaging system, generate large volumes of data that require quick analysis to help better understand the changing environment around us. The results from these analyses could aid scientists, researchers, policy makers and general public make informed decision (Hey, Tansley and Tolle 2009).

[10] http://www.ebi.ac.uk/

Global temperature change is one of the active environmental sciences issues gaining sufficient interest from government organisations, researchers in pedagogy and industry and the general public alike. It has become increasingly important to monitor changes in global weather patterns in order to avert catastrophic loss of lives and property.

In this work we analyse GISTEMP, a model for estimating the global temperature change. GISTEMP is analysed for its applicability to use the MapReduce framework for parallelising data and compute intensive tasks. Parallelising execution could result in a model that generates the required meaningful information from large volumes of temperature data gathered across various weather stations around the globe quickly and thus make informed decisions faster.

2.3 GISS Surface Temperature Analysis (GISTEMP)

Analyses of surface air temperature and ocean surface temperature changes are carried out by several groups, including the Goddard institute of space studies (GISS) (Hansen, Ruedy and Glascoe, et al. 1999) and the National Climatic Data Center (Peterson, et al. 1998) based on the data available from a large number of land based weather stations and ship data, which forms the instrumental source of measurement of global climate change. Uncertainties in the collected data from both land and ocean, with respect to their quality and uniformity, force analysis of both the land based station data and the combined data to estimate the global temperature change. Another valuable source of global temperature data through the troposphere and lower stratosphere is provided by the radiosonde[11] stations (Hansen and Lebedeff 1987).

Estimating long term global temperature change has significant advantages over restricting the temperature analysis to regions with dense station coverage, providing a much better ability to identify phenomenon that influence the global climate change, such as increasing atmospheric CO_2. This has been the primary goal of GISS analysis.

Non climatic influence on the measured temperature change, such as urbanisation, are minimised by applying a homogeneity adjustment. The homogeneity adjustment procedure (Hansen, Ruedy and Glascoe, et al. 1999) changes long-term temperature trend of an urban station to make it agree with the mean trend of nearby rural stations. The current analysis uses satellite observed nightlights (Hansen, Ruedy and Sato, et al. 2010) to identify land based weather stations in extreme darkness and perform urban adjustments for non-climatic factors, such that urban effects on the analysed global temperature change are small. The GISS temperature analyses which include maps, graphs and tables of the results are available for download on the GISS website (http://www.giss.nasa.gov).

[11] A radiosonde is a unit for use in weather balloons that measure various atmospheric parameters and transmits them to a fixed receiver.

2.1.1 Data Sources

The current GISS analysis obtains the monthly mean station temperatures from the Global Historical Climatology Network (GHCN), available for download from the NCDC website[12]. GHCN maintains data from about 7000 stations out of which only those stations that have a period of overlap with neighbouring stations (within 1200 km) of at least 20 years are considered (Hansen, Ruedy and Sato, et al. 2010). Effectively, only 6300 stations are available for GISS analysis after this reduction. No data adjustments are done on the original GHCN data for clarity.

The data from United States Historical Climatology Network (USHCN), which is a subset of the GHCN, is however adjusted via homogenisation intended to remove effects of urbanisation and other artefacts. Bad data from GHCN are minimised at NCDC via checks for all monthly mean outliers that differ from their climatology by more than 2.5 standard deviations. About 15% of these outliers are eliminated for being incompatible with neighbouring stations, with the remaining 85% being retained. (Hansen, Ruedy and Sato, et al. 2010).

The GHCN land based temperature records would be incomplete without measurements from the Antarctic region. Credible data over long continuous period was not available until the International Geophysical year 1957 (Hansen, Ruedy and Sato, et al. 2010). Current GISS analysis uses monthly data from Scientific Committee on Antarctic Research (SCAR). Specifically, the data are from the SCAR Reference Antarctic Data for Environmental Research project (http://www.antarctica.ac.uk/met/READER/).

The ocean surface temperature measurement is an integration of the data from Met Office Hadley Centre analysis of sea surface temperatures (HadISST1) for the period 1880-1981, which was ship based during that interval, and satellite measurements of sea surface temperature for 1982 to the present (Optimum Interpolation Sea Surface Temperature version 2 (OISST.v2) [(Hansen, Ruedy and Sato, et al. 2010)]. The satellite measurements are calibrated with the help of ship and buoy data (Reynolds, et al. 2002). Uncertainties in the pre-satellite era aroused due to homogeneity issues where the temperature measurements were dependent on ships, and each had their own techniques and units of measurement. Lately, due to availability of satellite temperature measurement systems, ocean coverage has improved to a large extent providing a much better quality of sea-surface temperature. However, satellite data also have their own sources of uncertainties, despite their high spatial resolution and broad geographical coverage (Hansen, Ruedy and Sato, et al. 2010).

2.3.2 Urban effects on Global temperature

Urbanisation, which includes human-made structures and energy sources, can significantly impact the accuracy of temperature measured by stations located in or near urban areas. This has been a major concern in the analysis of global temperature change.

[12] http://www.ncdc.noaa.gov/oa/ncdc.html

Current analyses either omit urban stations or perform urban adjustments to eliminate or minimise the urban effect. A detailed study on this topic is provided by (Parker 2010). Global satellite measurements of night lights allow the possibility for an additional check on the magnitude of the urban influence on global temperature analyses (Hansen, Ruedy and Sato, et al. 2010). It has been shown in (Imhoff, et al. 1997) that all stations located in areas with night light brightness exceeding a value (32 μWm^{-2} sr^{-1} μm^{-1}) approximately divides the station into two categories, rural and urban or peri-urban. Current GISS global temperature analysis perform urban adjustments on temperature data for stations located in regions with night light brightness exceeding this limit to agree with the temperature data of nearby rural stations. If there are no sufficient numbers of nearby rural stations, the "bright" station is excluded from the analysis. Also, it has been shown that urban warming has little effect on standard global temperature analysis (Hansen, Ruedy and Sato, et al. 2010).

Figure 1 shows the satellite observed night light radiance (μWm^{-2} sr^{-1} μm^{-1}) at a spatial resolution of 0.5^{o} x 0.5^{o} that aid in the categorization of stations as rural and urban or peri-urban.

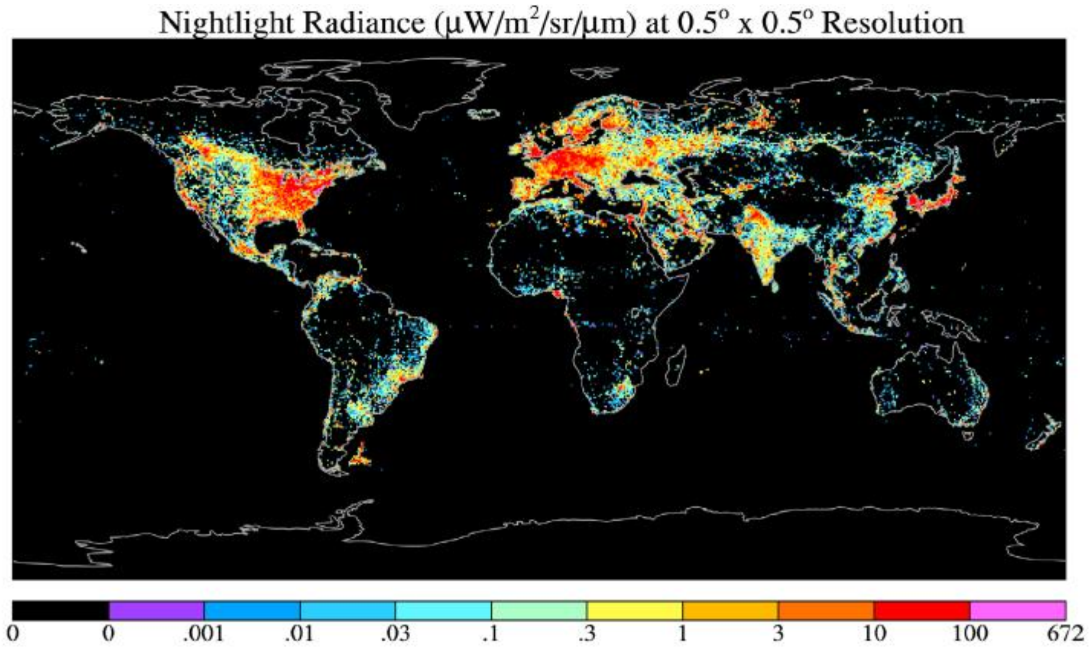


Figure 1: Satellite observed night light radiances at a spatial resolution of 0.5^{o} x 0.5^{o} [Source: (Hansen, Ruedy and Sato, et al. 2010)]

Figure 2 shows the global surface temperature anomalies for the past 4 decades, relative to the 1951 – 1980 base periods. In can be seen that on an average, the successive decades warmed by 0.17^{0}C. In addition, it is shown in (Hansen, Ruedy and Sato, et al. 2010) that warming in the recent decades is larger over land than over ocean because the ocean responds more slowly to the impacts of forced climate change due to ocean's large thermal inertia.

Warming during the past decade is enhanced, relative to the global mean warming, by about 50% in the United States, a factor of 2-3 in Eurasia, and a factor of 3-4 in the Arctic and Antarctic Peninsula. Warming of the ocean surface has been largest over the Arctic Ocean, second largest over the Indian and western Pacific oceans, and third largest over most of the Atlantic Ocean (Hansen, Ruedy and Sato, et al. 2010).

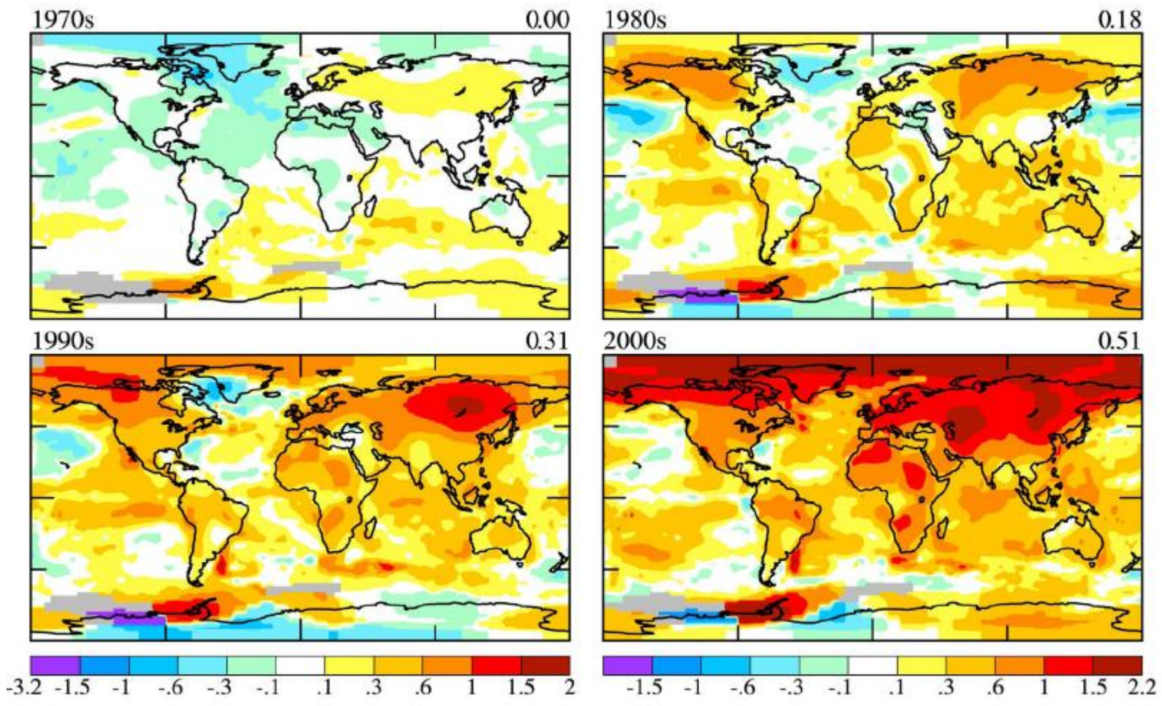


Figure 2: Decadal surface temperature anomalies relative to 1951 - 1980 base periods [Source: (Hansen, Ruedy and Sato, et al. 2010)]

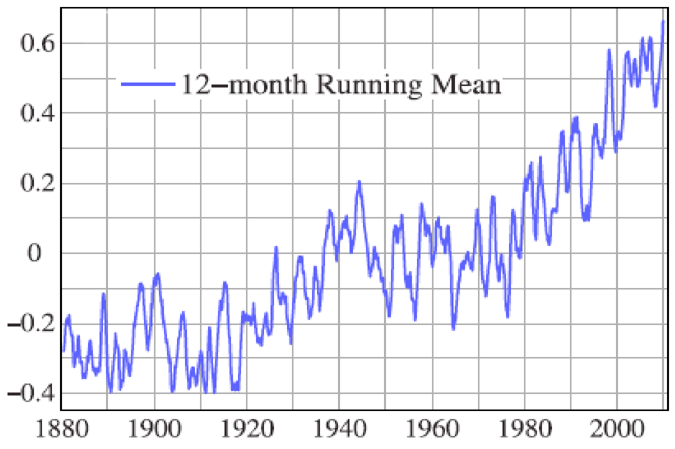


Figure 3: 12 month running mean of global temperature anomalies using data through June 2010
[Source: (Hansen, Ruedy and Sato, et al. 2010)]

Figure 3 shows a simple graph of 12 month running mean global temperature using data through June 2010.

Modifications to the GISS analysis method, if any, are available at GISS site: http://data.giss.nasa.gov/gistemp/updates/.

To summarise, Graphs, tables and maps are constructed by modifying the current GHCN, USHCN and SCAR files in two stages. In the first stage, redundant multiple records are combined into one and in the second stage the urban adjustments are performed so that their long-term trend matches that of the mean of neighbouring rural stations. Urban stations without sufficient number of rural stations in its vicinity are dropped from further analysis.

FORTRAN programs used in GISTEMP analysis along with the documentation on their use are open source and are available for download at http://data.giss.nasa.gov/gistemp/sources/.

Chapter 3

Clear Climate Code (CCC) Project

This chapter provides an overview of the *ccc-gistemp* project and outlines the merits of using this code for programmers. Stable release of *ccc-gistemp* is ported to the MapReduce programming model.

3.1 ccc-gistemp

The Climate Code Foundation *(http://www.climatecode.org/)* is a non-profit organisation to promote the public understanding of climate sciences. The Clear Climate Code project is a part of the Climate Code Foundation to re-implement NASA GISS Gistemp algorithm in Python for improved clarity called *ccc-gistemp (http://code.google.com/p/ccc-gistemp/).* ccc-gistemp release 0.6.1 is the current stable release available for download from *http://code.google.com/p/ccc-gistemp/downloads/list* which is used in the code development phase of this project.

3.1.1 A brief comparative study[13]

The combination of FORTRAN code and shell scripts in the original GISS software, for both core GISS algorithms and supporting libraries, makes it hard to perceive the code flow and the underlying algorithm. An all-python version with an 'easy-to-use' interface for program execution further exemplifies the essence of ccc-gistemp. The ccc-gistemp execution can be started from a single call to *run.py* by specifying the required step(s). Besides, by isolating the core GISS algorithms from the supporting functions (algorithms for reading input and writing output), *ccc-gistemp* has efficient code packaging further improving the readability and maintainability of the software.

The temperature "Series" anomaly, which is the primary data object in most GISS algorithms, has differing representations in different parts of the code making it hard for new users to comprehend the code easily. This has been modified in *ccc-gistemp* to have a single representation throughout the code. Additionally, the exhaustive use of intermediate files to store and manipulate temporary data has been modified to make efficient use of Python data structures and iterators, significantly reducing the I/O overheads.

[13] Much of this is compiled by reading blogs and discussions among ccc-gistemp developers and users, at the ccc-gistemp Google groups.

To perform urban adjustments, all rural stations in the vicinity of a given urban station are identified (section 2.3.2). In the original GISS code, the station latitudes (northern and southern) and longitudes (western and eastern) used to compute the distances were rounded to the nearest tenths for ease of storage in intermediate files. Additionally, in the same step, the annual anomaly series that is used in the computation of adjustment to apply to an urban station were also rounded to the nearest tenths of degree Celsius (originally in hundredths of a degree). In the *ccc-gistemp*, instead of rounding, all temperature series are represented as 'float' (floating-point) providing a much better representation of the data.

Although there has been significant amount of modifications to the structure of the original code, ccc-gistemp developers have ensured that the ported algorithms are identical to the original by constant comparison of the output at every stage with the expected results.

3.2 Merits of ccc-gistemp

1. Lack of well defined coding standards and documentation made the original GISTEMP code hard to perceive. This was overcome in *ccc-gistemp* which has well defined coding standards (http://code.google.com/p/ccc-gistemp/wiki/CodingStandard) that aid programmers to quickly read and understand the code, while improving the maintainability of the code.

2. *ccc-gistemp* provides code comments for most function definitions making it easy to understand the GISTEMP algorithms and code flow.

3. The results produced by *ccc-gistemp* are almost identical to that that produced by the GISS code.

4. A script to compare the results of two executions is provided (tool/compare_results.py) to aid comparison of results obtained after code modification, without the need for any manual comparison.

5. The use of high level programming language like Python makes future code developments simpler and quicker.

6. The CCC GISTEMP discussion (http://groups.google.com/group/ccc-gistemp-discuss) group make it easier for developers to clarify doubts and share ideas.

Chapter 4

Cloud Infrastructure

This chapter provides a brief overview of Cloud computing and the use of Cloud based resources (infrastructure) to support data-intensive computing. The advantages of utility computing and pay-per-use model for the scientific community are discussed. Finally, the chapter concludes by establishing a relationship between MapReduce programming model and 'clouds' for providing cost-effective data-intensive computing.

4.1 The Definition

Numerous definitions have been coined for the term "Cloud Computing" by researchers, developers and commercial hardware and software service providers. Vaquero, Rodero-Merino, Caceres, & Lindner, 2009 define 'Clouds' as a large pool of easily usable and accessible virtual resources (such as hardware, development platforms and/or services) which can be configured and used as and when required by a pay-per-use model. What's most important to understand here is how the concept of cloud computing and the use of cloud infrastructure could shape scientific research, by making hardware and software resources available to innovative ideas, without the need for large capital investment.

4.2 Utility Computing

The pay-per-use model has led to the use of computing resources as a metered service, like electricity and natural gas, in what is called as *utility computing* where a "cloud user" can dynamically provision any amount of computing resources from a "cloud provider" on demand and only pay for what is consumed (Lin and Dyer 2009). In practical terms, the cloud user is provided access to an instance of the operating system such as Linux, called a *virtual machine*. The choice and configuration of the operating system depends upon the user and the availability of such system with the cloud provider. Provisioning of the requested physical resources is handled by the cloud provider by use of Virtualization technology, ensuring security and isolation between multiple users sharing the same hardware. Virtual machines that are no longer required are deleted, freeing up system resources that can be used by other users.

With utility computing, the cloud users with innovative ideas for a new Internet service or with significant amount of computation do not have to invest upfront in building large data centres or in manpower to maintain them. Operational costs, dominated by cost of electricity and cooling can also be avoided.

Additionally, by subscribing to a "cloud provider" the users gain an important advantage by the *property of elasticity* (Armbrust, et al. 2009) where the demand for computing resources can be varied depending on the requirements of the cloud based application. If there is an unpredicted increase in computation, for example due to increase in customers for web based applications and due to increase in problem size for scientific computation, more physical resources can be requested from the cloud without interrupting the service. As demand falls, provisioned resources can be released.

There are a number of cloud service providers around the globe, with Amazon Web Services (AWS)[14] being a dominant player. It offers various products and services that support the cloud computing paradigm, which are billed on usage.

Eucalyptus[15] offers an open source cloud computing platform that is gaining significant popularity. In this work we use the eucalyptus platform supporting the cloud infrastructure provided by University of St. Andrews, United Kingdom. (http://www.cs.st-andrews.ac.uk/stacc).

4.3 Classification of 'Clouds' – Levels of Abstraction

Current taxonomy classifies clouds based on the type of services being offered:

Infrastructure as a Service

The cloud provider offering computing resources such as storage and processing capacity to users by providing access to virtual machine instances through virtualization is *infrastructure as a service (IaaS)* scenario.

Platform as a Service

At the next higher level, the cloud systems offering computing platforms on which the user application can run, is *platform as a service (PaaS).* PaaS offerings may include facilities for application design, application development, testing, deployment and hosting (Armbrust, et al. 2009). A well known example is the Google App Engine which provides the backend data-store and APIs for anyone interested to build highly scalable web applications.

Software as a Service

At an even higher level, software applications that are of interest to a wide variety of users can be hosted on the cloud system, which is *software a service (SaaS).* An example of this is the Google Docs which allows online editing and sharing of documents, spreadsheets and presentations.

[14] http://aws.amazon.com/
[15] http://www.eucalyptus.com/

These levels of abstraction provided by the cloud services play a significant role in the scalability and elasticity of operations, thereby avoiding under-utilisation (idle resources) or over utilisation (repeated failures) of computing resources. It is therefore important to study and understand cloud infrastructure to identify appropriate levels of abstraction required for any data-intensive computing, that may as well be dynamic.

The *infrastructure* provisioned by the St. Andrews cloud for this work exemplifies the advantage of utility computing for 'tasks' whose computational requirements cannot be predetermined. With just a few mouse clicks, new instances with greater computational capacity and disk space were created for increased computational needs.

4.4 Cloud Computing and MapReduce

Cloud computing and the use of cloud infrastructure is related to MapReduce and data-intensive computing, which is one of the main areas of focus in this work. This relationship is more obvious than what it seems, as processing large volumes of distributed data with MapReduce require access to clusters with sufficient capacity and not everyone with large-data problems can afford to purchase and maintain clusters (Armbrust, et al. 2009). With utility computing, resources can be provisioned depending on the user requirements and paid for only as much as is required to solve the problem. This close coupling between MapReduce programming model and utility computing provide an ideal platform to developers in the industry and academia for data-intensive computing and research.

Barroso & Hölzle, 2009 conducted a comparative study between the communication costs associated with high-end symmetric multiprocessing (SMP) machine and low-end network-based cluster under similar workloads. It was observed that cluster of low-end servers approach the performance of the equivalent cluster of high-end servers and that the small gap was insufficient to justify the price premium of high-end servers. In addition, the cost associated with high-end SMP machine does not scale linearly with the increase in computing power (i.e., a machine with twice as many processors is often significantly more than twice as expensive). The Google implementation of MapReduce runs on a large cluster of commodity machines connected together with switched Ethernet, processing many terabytes of data on thousands of machines (Dean and Ghemawat 2004). Such systems are scalable and elastic. It appears from these observations that huge capital investment on high-end machines is unjustifiable in order to obtain the required computational power and performance for data-intensive computing. Thus many implementations of MapReduce programming model are designed around clusters of low-end commodity servers as scaling "out" is superior to scaling "up" (Lin and Dyer 2009).

System failures are a commonplace across large cluster and hence the computing platform must be resilient and fault tolerant. MapReduce is designed to cope up with system failures gracefully (detailed description in section 5.3.2). This further strengthens the point made earlier that implementations of MapReduce programming model on cloud infrastructure consisting of low-end commodity machines (Utility Computing) is cost-effective and efficient for data-intensive computing.

Chapter 5

Programming Paradigm

This chapter introduces the MapReduce programming model as a means of providing abstraction for distributed, data-intensive computing. The challenges associated with task parallelisation and data distribution are discussed along with the characteristic features of MapReduce framework that facilitate overcoming these complexities. Brief overview of the Hadoop implementation of MapReduce and Hadoop Distributed File System (HDFS) is provided.

mrjob API and its advantages as a simple abstraction for writing MapReduce jobs in Python are summarized. Finally, the chapter concludes with a weather data mining example from the environmental sciences to provide a better understanding of the MapReduce concepts.

5.1 Introduction

The challenges associated with Distributed computing[16] are greatly compounded when compared to sequential programming. In addition to algorithmic issues, language syntax and semantics, the developer has to deal with concurrency issues such as race conditions[17] and deadlocks[18] as the code executes in parallel across several machines, accessing data in unpredictable patterns. Hence, it is imperative to provide a layer of abstraction to separate the algorithmic details (implementation) from the parallelisation chores (execution). MapReduce addresses the challenges of distributed computing by providing an abstraction that isolate the developer from system-level details such as handling concurrent data access, scheduling and load balancing.

5.2 Programming Abstraction – The Basics

Within a programming system, *programming abstraction* may be viewed as a way of supporting or implementing commonly occurring modes of computation, composition and/or resource usage at a high-level (Jha, et al. 2009). In other words, abstractions hide underlying complexity and expose only the simple and well-defined interface to users of the abstraction. The novel idea behind this work is an attempt to verify if MapReduce provides an ideal programming abstraction for dynamic, distributed, data-intensive (3D) computing for real-world problems.

[16] Distributed computing refers to the use of distributed systems to solve computational problems. In general, the problem is divided into many tasks, each of which is solved by one or more computers.
[17] A *race condition* occurs when multiple threads access a shared memory location at the same time.
[18] *Deadlock* is a situation when two or more threads are waiting for the other to release a resource and none do so leading to the program being stalled.

Before delving into the details of the project, at first, it is essential to understand the nuances of MapReduce programming model and its characteristic features that make it suitable for data-intensive computing.

In computer science, the divide and conquer[19] approach is conventionally used to solve large problems by breaking down the problem into two or more sub-problems. The solutions to the sub-problems are then combined to give a solution to the original problem. This technique has found applicability in a wide range of problems across multiple domains and is also the basis for writing data-intensive applications (large-data problems). However, the complexities in applying the divide and conquer technique such as task parallelisation, work distribution and load-balancing, synchronisation and communication issues must be addressed for a particular problem in order to obtain optimum results.

Application programming interfaces (APIs) such as OpenMP[20] for shared-memory architectures and MPI[21] for distributed-memory architectures provide abstractions that facilitate parallel programming on these architectures. In shared-memory programming, the developer needs to explicitly take care of access to shared data structures using locks, barriers or critical sections which has the risk of accidentally introducing synchronisation bugs and race-conditions. With MPI, the developer is burdened with load balancing issues and minimising communication overheads. Additionally, these frameworks are ideal for compute-intensive applications and large-scale simulations but have minimal support for dealing with data-intensive problems (Lin and Dyer 2009). MapReduce framework provides a simple abstraction to the developer by automatically handling task management, concurrent data access and communication primitives.

In addition to the above mentioned issues, data-intensive computing has yet another important challenge of data locality – bringing data and code together for computation to occur (Lin and Dyer 2009). Data locality minimises network overheads resulting in improved performance and better resource utilisation. The MapReduce framework takes the location information of the input data and attempts to schedule task on the node that contains the data or at least on a node as close as possible to the input data. This is where MapReduce finds its biggest advantage, making it ideally suitable for data-intensive computation.

[19] For more information, see: Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms* (MIT Press, 2000)
[20] http://www.openmp.org/
[21] http://www.mcs.anl.gov/mpi/

5.3 Map-Reduce Framework

Map-Reduce is a programming model introduced by Google Inc. to support distributed computing on large datasets (Dean and Ghemawat 2004). The application is implemented as a sequence of Map-Reduce operations, each consisting of a Map phase and a Reduce phase. In its basic form, the user specifies a *'map'* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *'reduce'* function that merges all intermediate values associated with the same intermediate key (Dean and Ghemawat 2004).

At Google, the need for such a programming abstraction arose due to the large amount of distributed data that was required to be processed in order to obtain meaningful information in a reasonably short period of time. Examples of such data-intensive operations are processing the web request logs to identify the most frequent search query for a given day and to generate summaries of the number of pages crawled per host. Although these operations were computationally straightforward, processing large amount of distributed data brought in additional complexities pertaining to task parallelisation, data partitioning and fault tolerance.

The Map-Reduce framework enables the programmer to focus on the computation at hand while the system automatically takes care of the messy details such as parallelisation, distribution of computation, load balancing, task management and fault tolerance.

5.3.1 Map-Reduce Programming Model

To use the Map-Reduce framework, programmer specifies a *'map'* function and a *'reduce'* function. Multiple instances of these functions are run in parallel. The input data set is split into independent chunks which are then processed by these multiple instances in a completely parallel manner. The *'map'* function processes the input key/value pair to generate another key/value pair. The multiple instances of *'map'* function running in parallel, on the data partitioned across the cluster produce a set of intermediate key/value pairs which are passed to the *'reduce'* function.

The *'reduce'* function then merges all intermediate values associated with the same intermediate key to produce the next set of key/value pair. Output key/value pairs from each reduce task are written back onto the distributed file-system. The intermediate key/value pairs serve only as input to the *'reduce'* function and are not preserved.

map (*k1, v1*) → *list(k2, v2)*

reduce (*k2, list (v2)*) → *list(k3, v3)*

The framework takes care of scheduling the various *'map'* and *'reduce'* tasks to run in parallel, monitoring them and re-executing the failed task. Such programming abstraction provided by the Map-Reduce framework allows programmers without any prior experience with parallel and distributed computing to easily utilize the resources of a large distributed system.

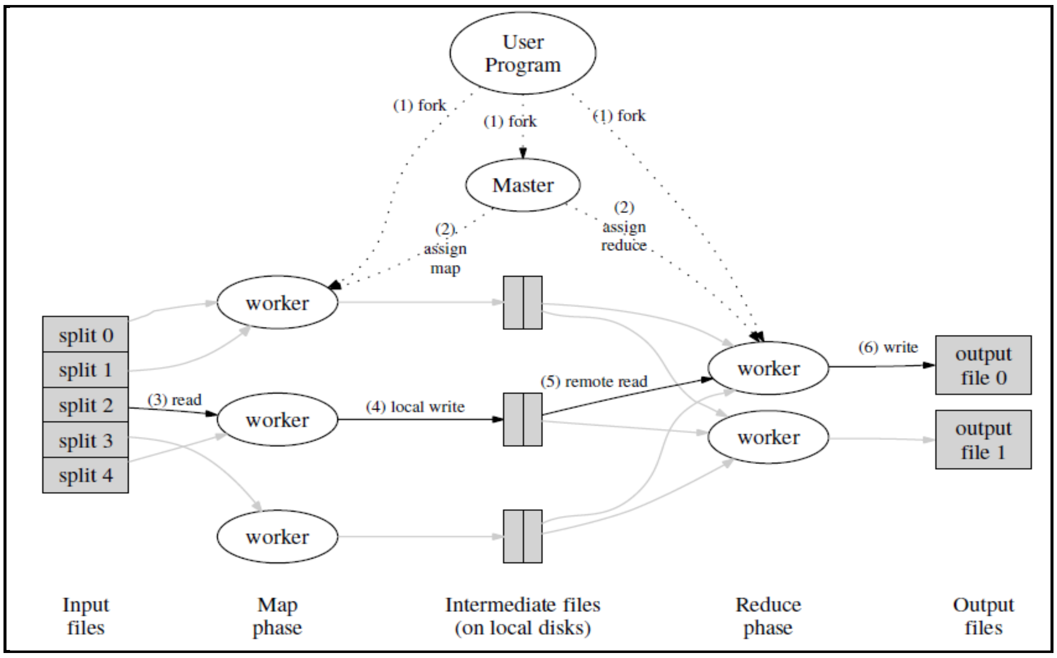


Figure 4: Map-Reduce execution overview
[Source: (Dean and Ghemawat 2004)]

Figure 4 shows the sequence of operations that occur when the user program calls the Map-Reduce function. The figure and the text below are reproduced here from (Dean and Ghemawat 2004) for completeness, with a few modifications. Also, the numbered labels in Figure 4 correspond to the numbers in the list below.

1. The Map-Reduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece which is controllable by the user via an optional parameter. It then starts up multiple copies of the program on a cluster of machines.
2. One of the copies of the program is the Master which assigns work to the rest of the copies, called workers. There are M *'map'* tasks and R *'reduce'* tasks to assign. R is either decided by the configuration specified with the user program or by the cluster wide default configuration. The master picks idle workers and assigns each one a *'map'* task or a *'reduce'* task.
3. A worker who is assigned a map task reads the contents of the corresponding input split which is parsed to obtain the key/value pairs from the input data. The key/value pairs are passed to the user-defined *'map'* function to generate the intermediate key/value pairs which are buffered in memory.
4. The buffered pairs are periodically written to local disk and partitioned into R regions by the partitioning function. The partitioning function is provided as default by the framework. However, the programmers have the flexibility to override this default function to provide custom partitioning. The locations of the buffered pairs on local disk are passed back to the master who in turn is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the users *'reduce'* function. The output of the *'reduce'* function is appended to a final output file, for this reduce partition.
7. When all *'map'* tasks and *'reduce'* tasks have been completed, the master wakes up the user program. At this point, the Map-Reduce call in the user program returns back to the user code.

5.3.2 Fault Tolerance

The Map-Reduce library provides a robust fault tolerance mechanism to handle machine failures gracefully while processing large amounts of distributed data on hundreds or thousands of machines. This is an essential feature as MapReduce was explicitly designed to operate on low-end commodity machines where failures are inevitable.

In its simplest form, the master pings every worker periodically. If no response is received from a worker in certain amount of time, the master marks the worker as failed and the task is rescheduled for execution on other available workers (Dean and Ghemawat 2004).

On failure, the completed map tasks are re-executed as their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. However, this is not the case with reduce tasks as they have their output stored in a global file system (Dean and Ghemawat 2004).

However, if the master task fails then the current implementation aborts the Map-Reduce computation. This condition can be checked by the user-program and reinitialize the operation if required.

5.3.3 Load Balancing

MapReduce employs two strategies for load balancing:

1. Dynamic load balancing is achieved when the worker nodes are assigned *map* and *reduce* tasks by the master, as and when they finish processing the current task. Slower nodes are assigned less work when compared to the faster ones.
2. Loss in performance caused by "stragglers"[22] is minimized by re-executing tasks on other available nodes. The task is marked as complete whenever either the primary or the backup execution completes.

[22] Degenerately slow workers taking unusually long time to complete one of the last few *map* or *reduce* tasks in the computation (Dean and Ghemawat 2004).

5.3.4 Data locality

Data locality – collocation of data and the node that performs computation, is a characteristic feature of MapReduce that facilitates data-intensive computing. The MapReduce master acquires information of the location of the input file from the distributed file-system and attempts to assign processing on the machine that actually contains the data. If this results in a failure, then the master reassigns the processing on a machine that is as close as possible to the input data. This has the effect of moving code to the data, improving the overall network utilisation by avoiding unnecessary data transfers.

(Butt, et al. 2009), conducted a series of experiments to evaluate the impact of data locality on application performance. It was observed that having to retrieve data over the network from remote racks significantly deteriorated the performance when compared to having the data on the same compute nodes or at least on a node within the same rack of compute nodes.

Observations in (Xie, et al. 2010) indicate that data locality is a determining factor for MapReduce performance and ignoring it can have noticeable reduction in performance, especially in heterogeneous environment such as virtualized data centres (Cloud infrastructure).

5.4 Hadoop overview

Hadoop (Apache Hadoop framework 2008) is the Apache Software Foundation open-source implementation of the Map-Reduce framework in Java. It provides tools for processing vast amounts of data using the Map-Reduce framework and, additionally, implements a distributed file system similar to Google's file system called Hadoop Distributed File System or HDFS. Hadoop can be used to process vast amounts of data in parallel on large clusters in a reliable and fault-tolerant fashion. Today, a significant amount of software development activity surrounds Hadoop with many distributed, data-intensive applications taking advantage of the open-source implementation in both industry and academia.

Although the Hadoop framework is implemented in Java, it is not required that Map-Reduce functions be written in Java. *Hadoop streaming* is a utility that allows programmers to create and run Map-Reduce jobs with executables (*map* and/or *reduce* function) written in any programming language that can read standard input and write to standard output. It uses UNIX standard streams as an interface between Hadoop framework in Java and the user program.

5.4.1 Hadoop Distributed File System (HDFS)

The fundamental idea of having a distributed file system is to divide user data (usually of the order of few gigabytes to a few terabytes) into blocks and replicate those blocks across the local disks of nodes in the cluster (Lin and Dyer 2009) such that it is easier to assign Map-Reduce job locally[23]. HDFS is designed based on this principle.

[23] See section 5.3.3 on Data locality

Additionally, data-intensive computing using MapReduce is dominated by long streaming reads and large sequential writes (batch operation involving large proportion, if not all, of the dataset). As a result, the time to read the whole dataset is important than the latency in reading the first record (White 2010).

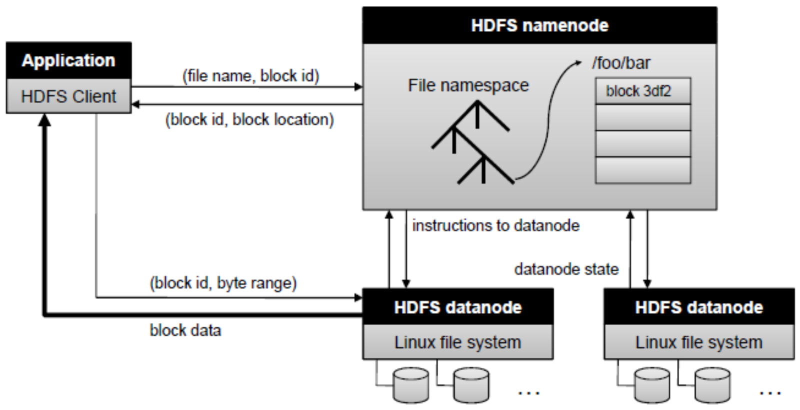


Figure 5: Architecture of HDFS.
[The *namenode* is responsible for maintaining the file namespace and directing clients to *datanodes* that actually hold blocks of user data. Source: (Lin and Dyer 2009)]

HDFS adopts master-slave architecture as shown in Figure 5. The Namenode (master) maintains the file namespace (metadata, directory structure, location of data blocks and file access permissions) and Datanode (slave) manages the actual data blocks (Lin and Dyer 2009). The Namenode is the first point of contact for any application wanting to read a file in order to obtain the physical location of the data (file metadata). In response to this request, Namenode returns the block id and the block location (Datanode) where the file is stored. The application then contacts the relevant Datanode to fetch the data. All data transfer occurs directly between the application and Datanodes, thereby improving reliability and robustness of the system (Namenode is rarely the bottleneck).

5.4.2 Hadoop Cluster

Figure 6 shows the architecture of Hadoop cluster which primarily consists of the following components, all of which are implemented as JVM daemons[24].

1. *Namenode* - Primarily responsible for controlling the HDFS and maintaining the overall health of the file system. Namenode runs the namenode daemon. Additionally, it maintains the file namespace as discussed in section 5.4.1., and hence is responsible for serving any file access requests made by the application client.

2. *Jobtracker* - Master node primarily responsible for scheduling, coordinating and monitoring the execution of MapReduce jobs[25] on various *tasktracker* nodes. It is the single point of contact for an application client wishing to execute a MapReduce job.

[24] A daemon is program that runs in the background, rather than under the direct control of a user.
[25] The MapReduce job consists of the input data, the MapReduce program (map and reduce function) and the configuration information.

As a fault-tolerant mechanism it periodically pings the slave nodes to check status. In case of a node failure, the *jobtracker* reschedules the failed job on another node.

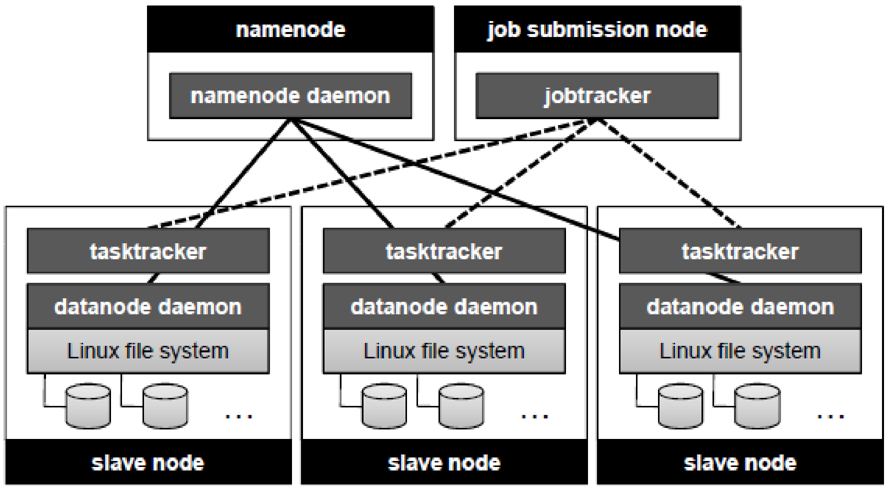


Figure 6: Hadoop Cluster
[(Lin and Dyer 2009)]

3. *Tasktracker* - Responsible for actually running the user code. It periodically sends updates back to the *jobtracker*.

4. *Datanode* - Runs the datanode daemon for serving HDFS data. The data blocks are actually stored on standard single-machine file system, like Linux and HDFS is designed to lie on top of the standard operating system stack (Lin and Dyer 2009).

5.4.3 MapReduce Job Configuration

A *JobConf* configuration file must be specified by the user application program in order to run MapReduce jobs on Hadoop cluster. It provides a primary interface for a user to describe MapReduce jobs to the Hadoop framework for execution.

JobConf is typically used for specifying:

1. The Mapper, Combiner and Reducer classes.
   Eg: JobConf.setMapperClass( ), JobConf.setReducerClass( ).

2. The number of Reducer tasks within the user program.
   Eg: JobConf.setNumReduceTasks( ).

3. Input and output files required for the MapReduce job.
   Eg: FileInputFormat.setInputPaths(conf, inputpaths), addInputPath(conf, path) and setInputPaths(conf, commaSeperatedPaths).

5.5 Python programming language and mrjob API

'mrjob' [26] is a Python package that aids in the development and execution of Hadoop streaming jobs on Amazon Elastic MapReduce (EMR) or Hadoop cluster. Version 0.2.6[27] was used in the development of this project. Some of the features of *mrjob* that were particularly useful in this project are:

1. Run jobs on EMR, Hadoop cluster and locally (for testing).
2. Write multi-step jobs (one map-reduce step feeds into the next).
3. Custom switches can be added to the map-reduce jobs, including file options.
4. Setup of map-reduce job handled transparently by *mrjob.conf* config file[28].
5. Upload source tree and automatically include it in the job's $PYTHONPATH.
6. Easy installation and *mrjob* configuration.

mrjob provides a simple abstraction for writing MapReduce jobs in Python by defining steps for specifying *'mapper'* and *'reducer'* functions, input and output file format (protocol) and paths. Additionally, it provides APIs for setting necessary parameters in the Hadoop MapReduce *JobConf* configuration file.

5.5.1 Dumbo and mrjob

Dumbo[29] is another open-source MapReduce python module which provides similar programming abstraction like *mrjob* to develop and execute Hadoop streaming jobs. *mrjob*, however being newer provides some additional functionality that makes it an ideal choice for this project.

At the onset of this work, it was not very obvious which cloud infrastructure would be used to test the map-reduce implementation of the original *ccc-gistemp* code. Various cloud infrastructures and cloud computing platforms were evaluated, including the Eucalyptus platform supporting the cloud infrastructure provided by the National Grid Service on campuses at the University of Edinburgh, University of Oxford and the Imperial College at London, cloud services offered by the University of St. Andrews[30] and the Amazon Elastic MapReduce[31] on Amazon Elastic Compute Cloud (Amazon EC2). Reading blogs[32] from developers and users of *dumbo* and *mrjob*, it was concluded that map-reduce jobs developed using *mrjob* integrates easily with Amazon Elastic MapReduce with little or no manual intervention. *mrjob* also provided an easy interface to launch jobs locally and on Hadoop cluster setup on any cloud infrastructure. As the choice of cloud infrastructure was not very clear at the beginning of this work, *mrjob* proved to be a better pick of the two.

[26] http://pypi.python.org/pypi/mrjob/0.2.6
[27] http://packages.python.org/mrjob/index.html
[28] More information about mrjob configuration can be found at: http://packages.python.org/mrjob/configs-conf.html#module-mrjob.conf
[29] http://klbostee.github.com/dumbo/
[30] http://www.cs.st-andrews.ac.uk/stacc
[31] http://aws.amazon.com/elasticmapreduce/
[32] Example of one of the blogs at https://github.com/Yelp/mrjob/issues/11

In addition, ccc-gistemp code has complex data-structures which are to be passed between the *map* and *reduce* functions. *mrjob* provides '*JSON*' and other communication protocols like '*Pickle*'[33] which enables easy serialisation and de-serialisation of data when passed between map and reduce functions. Although *dumbo* provides similar implementation of these protocols, *mrjob's* documentation and examples strengthened its choice.

Testing individual steps of code (map and reduce functions) is made possible with custom switches that can be specified as command line arguments. These switches enable easy testing of individual map and reduce functions without having to run the complete MapReduce job.

Further investigation and comparison of *dumbo* and *mrjob* to provide the necessary abstraction was not performed in greater detail. However, many experiments with the code and functionalities provided by *mrjob* were investigated further and in detail.

5.5.2 JSON and Pickle communication protocols

mrjob uses communication protocols to allow arbitrary values as input and output rather than just strings. JSON[34] (JavaScript Object Notation) format being simple and lightweight, is the most commonly used protocol to exchange information. JSON is also the default communication protocol within the *mrjob* framework. It is primarily used to transfer simple objects such as lists, structures and nested dictionaries. However, it does not support complex data objects such as class instances and function definitions.

When using the JSON protocol, the key/value pairs are encoded as two JSONs separated by a tab within the *mrjob*.

To facilitate transfer of complex data structures, python provides a powerful interface called Pickle[35] module. It is primarily used for serialising and de-serialising python object structure. *Pickling* results in data objects being converted into byte stream so that they can be transferred easily through a data pipe such as a network. *Un-pickling* results in the reverse operation where a byte stream is converted back into a data object. An extended version of the Pickle module is *cPickle* which is much faster and efficient than its predecessor. This improved performance is attributed to its language of development which is C, hence *cPickle*.

mrjob implements the *cPickle* module to provide the communication protocol. The key/value pairs are represented as two string-escaped pickles separated by a tab.

In this project we extensively use the *cPickle* protocol for transferring data objects to, and receiving data objects from Hadoop streaming job. The use of heavier pickle protocol is necessitated by the fact that primary data object in the original *ccc-gistemp* code are instances of class "series" (detailed explanation in section 7.1) and this object is transferred from the map task to reduce task for most computations.

[33] Refer to section 5.5.2 for more information
[34] http://www.json.org/
[35] http://docs.python.org/library/pickle.html

In addition to the above mentioned protocols, *mrjob* API provides support for two other protocols called 'repr' and 'raw' whose interface can be thought of to be somewhere in between the simple JSON to a more complex Pickle module.

The use of these protocols can be verified by checking the intermediate and final output files generated by *mrjob,* while processing the Hadoop MapReduce tasks. The input to and output from map-reduce tasks (including multi-stage reducers) can be checked for correctness by verifying these files.

The output from the reducer tasks is written in a common output directory specified by the *FileOutputFormat* in the Hadoop job configuration. This can also be specified from the *mrjob* as a command line argument using the switch *--output-dir*. The output files are typically named *'part-xxxx'*, where xxxx is the partition id associated with the reduce task. If the execution of the job succeeds then these part files are automatically cleaned by the *mrjob.* These files, if required, can be retained for testing purposes by specifying *--cleanup NONE* before running the job.

The *NullOutputFormat* generates no output (consume all outputs and put them in /dev/null). This can be specified from the *mrjob* using the switch *--no-output.*

5.6 An Example of Map/Reduce Programming Model

In this section, an example from the environmental sciences is considered to provide a better understanding of all the topics discussed so far. The program is written in Python and interacts with the Hadoop MapReduce using *Hadoop streaming*. The code is tested on the St. Andrews cloud infrastructure where Hadoop is run on single node in a pseudo-distributed mode[36]. In this mode each Hadoop daemon runs as a separate Java process on a single node.

Hadoop streaming is a utility that allows programmers to create and run Map-Reduce jobs with executables (*map* and/or *reduce* function) written in any programming language. This flexibility has led to a widespread adoption of the Hadoop MapReduce for development as well as porting of exiting data-intensive applications.

Figure 7 shows the interaction between MapReduce programs written using Python programming language and Hadoop streaming. Standard input and output streams are used in the communication between the Python process running the user program and streaming task. The input key-value pairs are read from standard input (stdin) or from the user-specified file stored in HDFS. The python process runs the input key-value pairs through the user specified map or reduce function and passes the output key-value pairs back to the Java process. The actual number of map and/or reduce task run depends on the target machine (available number of cores and the amount of memory).

[36] Detailed explanation is provided in section 6.1.1

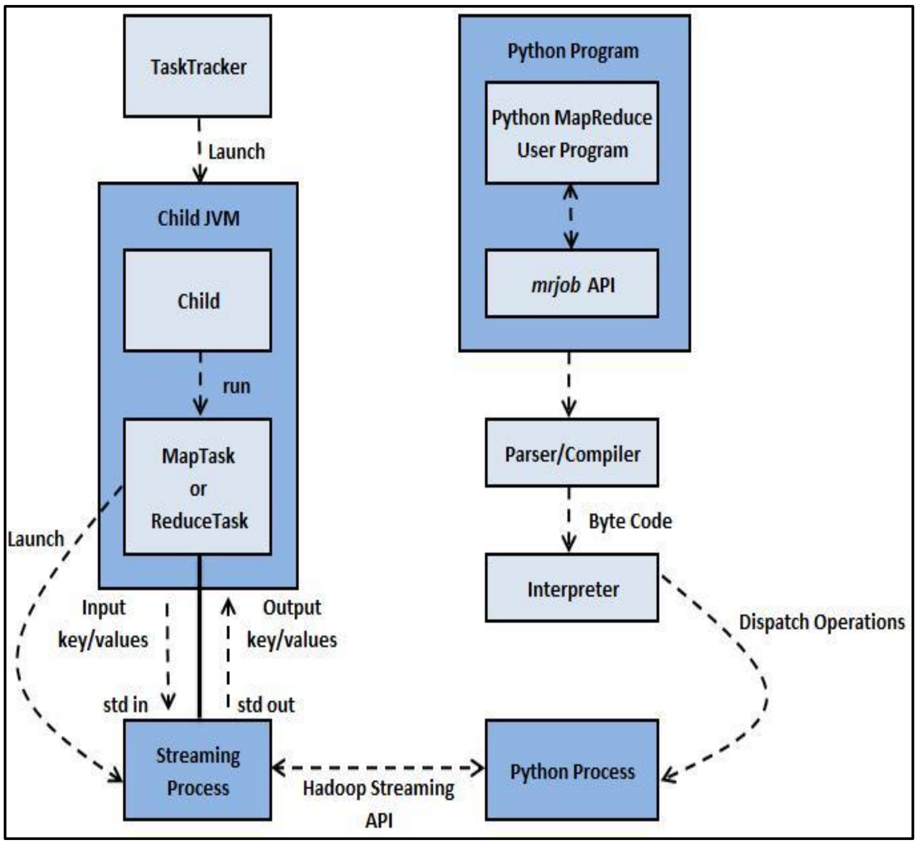


Figure 7: Block diagram showing the interaction between Python MapReduce user program, *mrjob* API & Hadoop streaming.
[Hadoop streaming reproduced from (White 2010)]

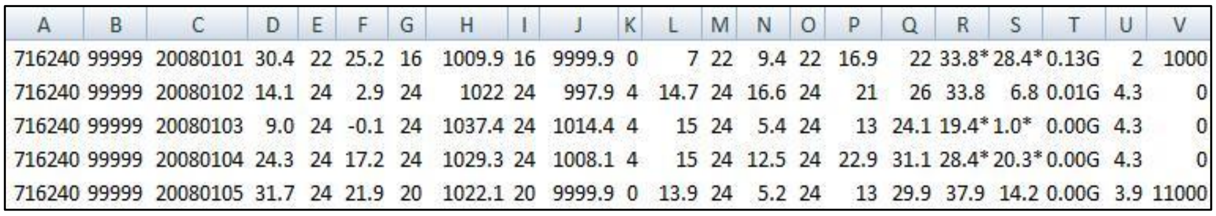
When a function is specified as a map/reduce, each mapper/reducer task will execute it as a separate process. The mapper task converts the input stream into lines and feeds these lines to the stdin of the process. The map function collects these lines from the stdout of the process and converts each line into key/value pair. By default, the *prefix of a line up to the first tab character* is the key and the rest of the line is the value (Apache Hadoop framework 2008).

The output of the mapper is given as input to the reducer task which converts the input key/value pairs into lines and feeds these lines to the stdin of the process. The reduce function then collects these lines from the stdout of the process and converts each line into a key/value pair, which is the output of the reducer. Although the map-reduce functions are executed by the Python process, for the tasktracker it is as if the map-reduce code was run by the tasktracker child process itself.

5.6.1 Weather data mining

To better understand MapReduce and the underlying data flow, let's consider a data mining example. Weather sensors around the globe record changes in temperature on an hourly basis generating large volumes of data, which is a good candidate for analysis with MapReduce. This is similar to the "A Weather Dataset" example given in (White 2010), but here the code is re-written to use the *mrjob* API. This example and the associated program in Python serve as a mechanism to test the environment and Hadoop configuration prior to executing the actual *mapreduce-cccgistemp* code on every machine. Only a small subset of the large volumes of data available on the National Climate Data Center[37] (NCDC) is used in this example.

The weather data mining example shown below returns the highest recorded monthly temperature from the recorded daily temperatures for the year 2008.



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
716240	99999	20080101	30.4	22	25.2	16	1009.9	16	9999.9	0	7	22	9.4	22	16.9	22	33.8*	28.4*	0.13G	2	1000
716240	99999	20080102	14.1	24	2.9	24	1022	24	997.9	4	14.7	24	16.6	24	21	26	33.8	6.8	0.01G	4.3	0
716240	99999	20080103	9.0	24	-0.1	24	1037.4	24	1014.4	4	15	24	5.4	24	13	24.1	19.4*	1.0*	0.00G	4.3	0
716240	99999	20080104	24.3	24	17.2	24	1029.3	24	1008.1	4	15	24	12.5	24	22.9	31.1	28.4*	20.3*	0.00G	4.3	0
716240	99999	20080105	31.7	24	21.9	20	1022.1	20	9999.9	0	13.9	24	5.2	24	13	29.9	37.9	14.2	0.00G	3.9	11000

Figure 8: Subset of the sample NCDC data for the year 2008

Figure 8 shows a subset of the sample input data from NCDC. The data consists of various fields but we are particularly interested in the month contained in column 'C' and the recorded daily temperatures in column 'D'.

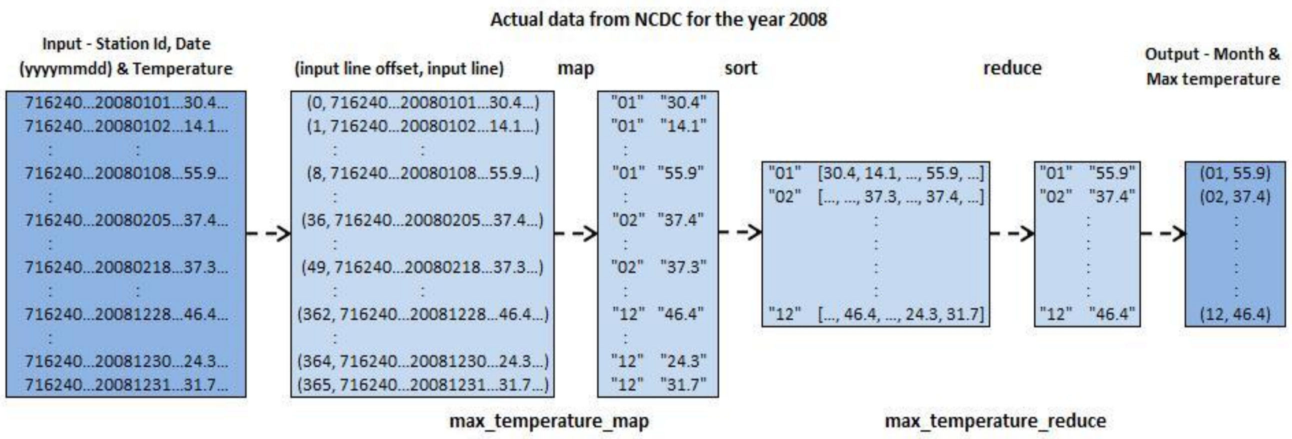


Figure 9: Data flow within the MapReduce programming model

Figure 9 illustrates the data flow within the Map-Reduce programming model for the weather data mining example. The output indicates that the maximum temperature in January was 55.9°C, February 37.4°C and December 46.4°C.

[37] ftp://ftp.ncdc.noaa.gov/pub/data/gsod/

The MapReduce framework divides the input file into fixed-sized chunks called *'splits'* and creates one *map* task for each split consisting of multiple lines, as shown in Figure 10. The *map* task then executes the user-defined *map* function for each line in the split. The input line is converted into key/value pairs consisting of line offset as the *key* and the line itself as the *value*. The *map* function extracts month and the corresponding temperature as a key/value pair and emits it as its output.

The number of *reduce* tasks can be explicitly specified in the job configuration. When there are multiple reducers, the *map* tasks partition their outputs creating one partition for each *reduce* task. It must be noted that the lines with the same *key* are all in the same partition. This is achieved by sorting and grouping the output of the *map* function, by the MapReduce framework, prior to being fed as input to the reducer.

As a result of this operation, all readings corresponding to a particular month appear in the same list with 'month' as the *key*. The *reduce* function then iterates through this list to yield the maximum temperature for a month, which is the final output.

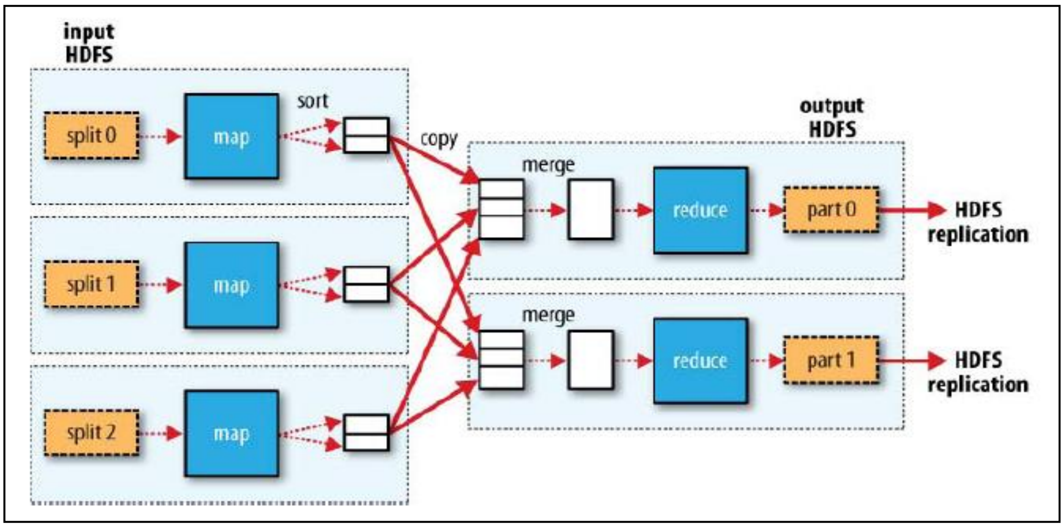


Figure 10: MapReduce framework with multiple *reduce* tasks.
[Dotted boxes indicates nodes, light arrows show data transfers within a node and the heavy arrows show data-transfers between nodes (White 2010)].

```
map (String key, String value)
{
        //<key>: line offset within the input file
        //<value>: line itself from the input file
        //Extract 'month' and 'temperature' from <value>
        EmitIntermediate('month', temperature)
}
```

```
reduce (String key, Iterator values)
{
        //<key>: month
        //<values>: Iterator over list of temperatures
        Emit(key, Max([list of temperatures])
}
```

Implementation of the above 'map' and 'reduce' functions in python are provided in Appendix-A. *'mrjob'* API provides a simple abstraction by specifying the parameters required in *JobConf*[38] and for handling the input and output file types .

[38] See section 5.4.3

Chapter 6

Development Infrastructure

This chapter provides a brief overview of the St. Andrews cloud infrastructure and the University of Edinburgh EDIM1 machine that is used extensively for the development and testing of *mapreduce-cccgistemp*. Further, project hosting and its advantages are summarised. Finally, the implications of agile development techniques in this project are discussed.

6.1 Execution and Testing Environment

The code was run and tested on two available machines viz. St. Andrews cloud infrastructure (StACC) and the University of Edinburgh EDIM1 machine. Hadoop was installed on single node in pseudo-distributed mode on the StACC, where most of the testing was done when development phase was in progress. Much of the unit-test cases (section 7.5) were executed for correctness on this machine.

After the development phase, the code was tested and benchmarked on the University of Edinburgh EDIM1 machine, which has the Cloudera distribution of Hadoop cluster setup (CDH3)[39].

6.1.1 St. Andrews Cloud Infrastructure

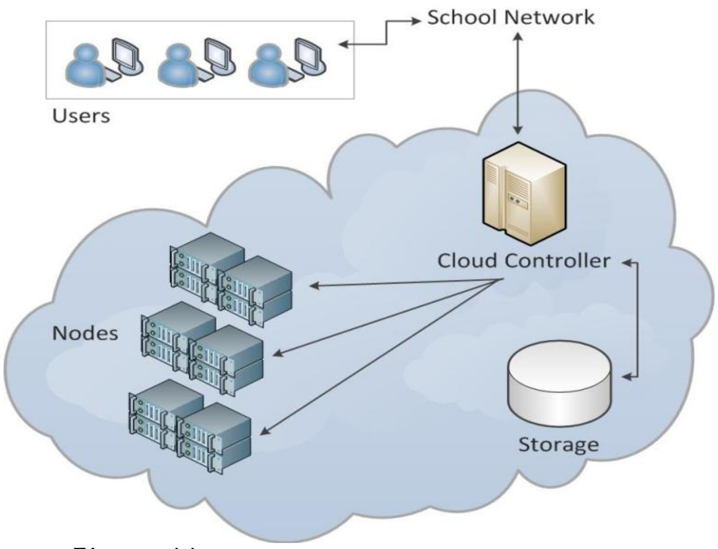


Figure 11: St. Andrews Cloud infrastructure
[Source: http://www.cs.st-andrews.ac.uk/stacc]

[39] https://ccp.cloudera.com/display/CDHDOC/CDH3+Documentation

The St. Andrews cloud infrastructure (Figure 11) was setup as a part of the St. Andrews cloud computing (StACC) research collaboration, with primary focus on becoming an international centre of excellence in research and teaching cloud computing. The procedure to setup and use the cloud services are provided in Appendix-D of this report.

6.1.2 University of Edinburgh EDIM1 machine

EDIM1[40], is cluster of commodity machines jointly funded by the Edinburgh Parallel Computing Center (EPCC) and the School of Informatics, primarily intended for Data-intensive research. The cluster is build from relatively inexpensive hardware with a dual core Intel ATOM processor on each node, which is comparatively slower to the current day high end processors. However, this machine has several fast disks connected directly to each of the 120 available nodes (distributed equally across three racks), ideally suited for data-intensive computing and research owing to its low latency and high I/O bandwidth. Various data-intensive research groups in the fields of Astronomy, Biology and Geosciences at the school of Informatics will benefit from this machine.

The Hadoop cluster setup on EDIM1 machine is based on the Cloudera distribution (CDH3) of Hadoop[41]. Steps to install Hadoop can be found at:

https://www.wiki.ed.ac.uk/display/daresearch/hadoop

Table 1 provides details of the hardware and its configuration on the EDIM1 machine. This can be used as a reference when comparing performance of *mapreduce-cccgistemp* on other machines and cloud infrastructures.

Category	Configuration
Number of Nodes	120 (3 racks of 40 nodes each)
Processors/Node	Dual-Core Intel 1.6 GHz ATOM[42] processor
Disk Storage/Node	1 x 256 MB Solid State Drive (SSD)
	3 x 2TB HDD
Network	10 Gigabit Ethernet
OS	Rocks (Clustered Linux Distribution based on CENTOS)[43] Linux Kernel Version 2.6.37
JVM	1.6.0_16
Hadoop	0.20.2 Cloudera Distribution version 3 (CDH3)

Table 1: Hardware configuration of EDIM1 machine

Performance evaluations have been done on this machine configured as a sixteen node cluster (one master node, one job tracker and fourteen slave nodes).

The results of benchmarking *mapreduce-cccgistemp* are provided in section 8.2 of this report.

[40] http://www.epcc.ed.ac.uk/projects/research/dataintensive
[41] http://www.cloudera.com/hadoop/
[42] http://en.wikipedia.org/wiki/Intel_Atom
[43] http://www.rocksclusters.org/rocks-documentation/4.2/

6.2 Project Hosting

The *Google Project hosting* services are used to host this open source project, as it is free and provides important features such as version control system and wiki. Hosting the project enables easy access to anyone who is interested in developing this project further.

Additionally, as the original *ccc-gistemp* code uses Google hosting service, it was thought to be advantageous to provide a similar interface for both the original and the ported code to aid comparison and future development.

The revision control system enables easy maintenance of the project source files and project releases. Mercurial[44] version control tool provided by Google Code is used for this purpose. All the project specific source files and project releases are maintained under this version control tool accessible through the project page at:

http://code.google.com/p/mapreduce-cccgistemp/

This also serves as a means of backing up project files and documents.

Project specific background information is added to the wiki pages to provide a good understanding of the basics to anyone interested in this project.

6.3 Integrated Development Environment

Eclipse[45] IDE was used in the code development phase of this project. Although Eclipse is well known for Java development, it provides support for other language IDEs. PyDev[46] is a well know IDE for Python development and hence used alongside Eclipse in developing code. PyDev was found to be simple to install and use.

6.4 Agile development techniques and its implications

Agile is a software development methodology[47] with emphasis on iterative, test-driven development. The principles from the Agile Manifesto[48] that were particularly useful in this project are listed below for completeness and clarity of explanation.

- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Sustainable development, able to maintain a constant pace
- Continuous attention to technical excellence and good design

[44] http://mercurial.selenic.com/

[45] Eclipse is an open source community, whose projects are focused on building an extensible development platform for building, deploying and managing software across the entire software lifecycle. (http://www.eclipse.org/)

[46] http://pydev.org/

[47] A software development methodology can be viewed as a framework to structure, plan and control the process of software development (http://en.wikipedia.org/wiki/Software_development_methodologies).

[48] http://www.agilemanifesto.org/principles.html

- Simplicity
- Regular adaptation to changing circumstances

Much of the other principles are related to team-development and are not very important in the current academic scenario.

One of the prime reasons for using agile techniques in this project is because of the flexibility offered by this model to adapt quickly to the changing realities. Major concerns during the design and development phases of the project were design deviation, as the chosen application was quite hard to be ported directly to MapReduce, and of not having a stable cloud infrastructure to test the ported code. The use of agile techniques enabled quick switching of tasks, and thus better time management to avoid missing deadlines from the original work plan (Appendix-C). As an example, when waiting on information from the open-source community for design related queries and from systems administrators for infrastructure related queries, dissertation write-up for the "Background and Motivation" section was completed. Similarly, after the design and development phase there was a considerable amount of waiting time for the Edinburgh EDIM1 machine to be setup for benchmarking. During this period, the dissertation write-up for the development phase was completed.

As the primary area of focus in this work was to evaluate the applicability of MapReduce for a scientific application code with many stages, use of agile techniques enabled incremental development and testing. Porting of code to MapReduce was done in small increments, followed by testing of these incremental steps before making progress. Thus, by not waiting for the entire development phase to complete before beginning test, risks associated with the chosen application not being suitable for dynamic, distributed, data-intensive computing (Rank 2), application domain being unsuitable (Rank 3) and the programming abstraction framework being inappropriate (Rank 4) was considerably mitigated[49].

The test-driven approach improves quality of the software, providing greater confidence in the software development at each stage. By successfully completing the first iteration (MapReduce implementation of step1), sufficient confidence was gained that the chosen application was indeed suitable to be evaluated with the MapReduce framework, covering many aspects of this model including the advantages and its limitations. Additionally, the use of agile techniques enabled thorough analysis of the chosen application (incremental development followed by testing), resulting in sufficient time towards the end for benchmarking.

The agile methods give priority to face-to-face communication over written documents. It must be emphasised that weekly/bi-weekly meetings with the project supervisor to review progress and re-evaluate priorities ensured steady progress, considerably mitigating the overall risk associated with the aggressive and ambitious project plan (Rank 1).

[49] Refer to the original risk plan included in Appendix-B

Chapter 7

Porting the ccc-gistemp

This chapter provides in-depth analyses of each of the steps in the original *ccc-gistemp* code while identifying patterns that are suitable to be ported to the MapReduce programming model. With the help of code flow diagram, a detailed summary of *mapreduce-cccgistemp* – MapReduce implementation of the original *ccc-gistemp* is provided. Further, issues encountered while porting are discussed with possible solution for their resolution. Finally, the chapter concludes with a validation study by comparing results from the original and ported code. Additionally, tests which have been performed to ensure correctness of the ported code are described.

7.1 *mapreduce-cccgistemp* – MapReduce implementation of cccgistemp

The original code was profiled on the Edinburgh EDIM1 machine to benchmark its execution pattern and runtime.

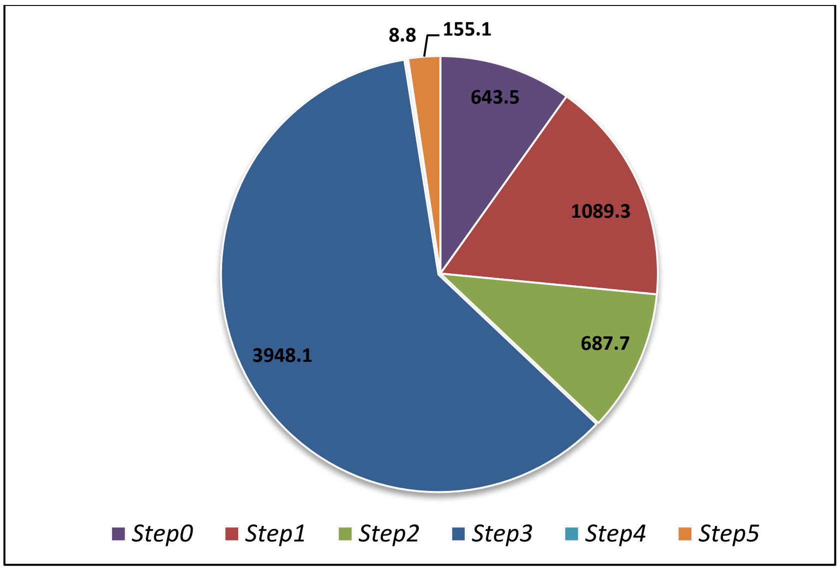


Figure 12: Profiling of the original ccc-gistemp code (seconds)

It can be observed from Figure 12 that *ccc-gistemp* exhibits a sharp profile dominated by step3. Since this step is CPU bound, efforts must be focussed on parallelising this step in order to obtain significant improvement in the overall performance. Good improvement in performance can be obtained by parallelising step1 as well. All the *ccc-gistemp* steps require intermediate storage and retrieval of files and hence should benefit from the fast disks connected to the nodes, owing to its low latency and high I/O bandwidth.

The output and log files generated by each of these steps are copied to facilitate comparison with the results obtained during subsequent executions, while porting. After the initial profiling, the original *ccc-gistemp* code was analysed to understand its behaviour and execution pattern.

Figure 13, implemented as part of this work, shows analysis of each of the steps from the original *ccc-gistemp* code. The code was analysed to understand its execution pattern, while applying the previous study of MapReduce programming model to identify areas in the code that could be suitable for porting. The "comments" provided in the code (section 3.2-Merits of ccc-gistemp) aided in faster analysis. Various approaches to porting the code are discussed while identifying the pros and cons with of these techniques.

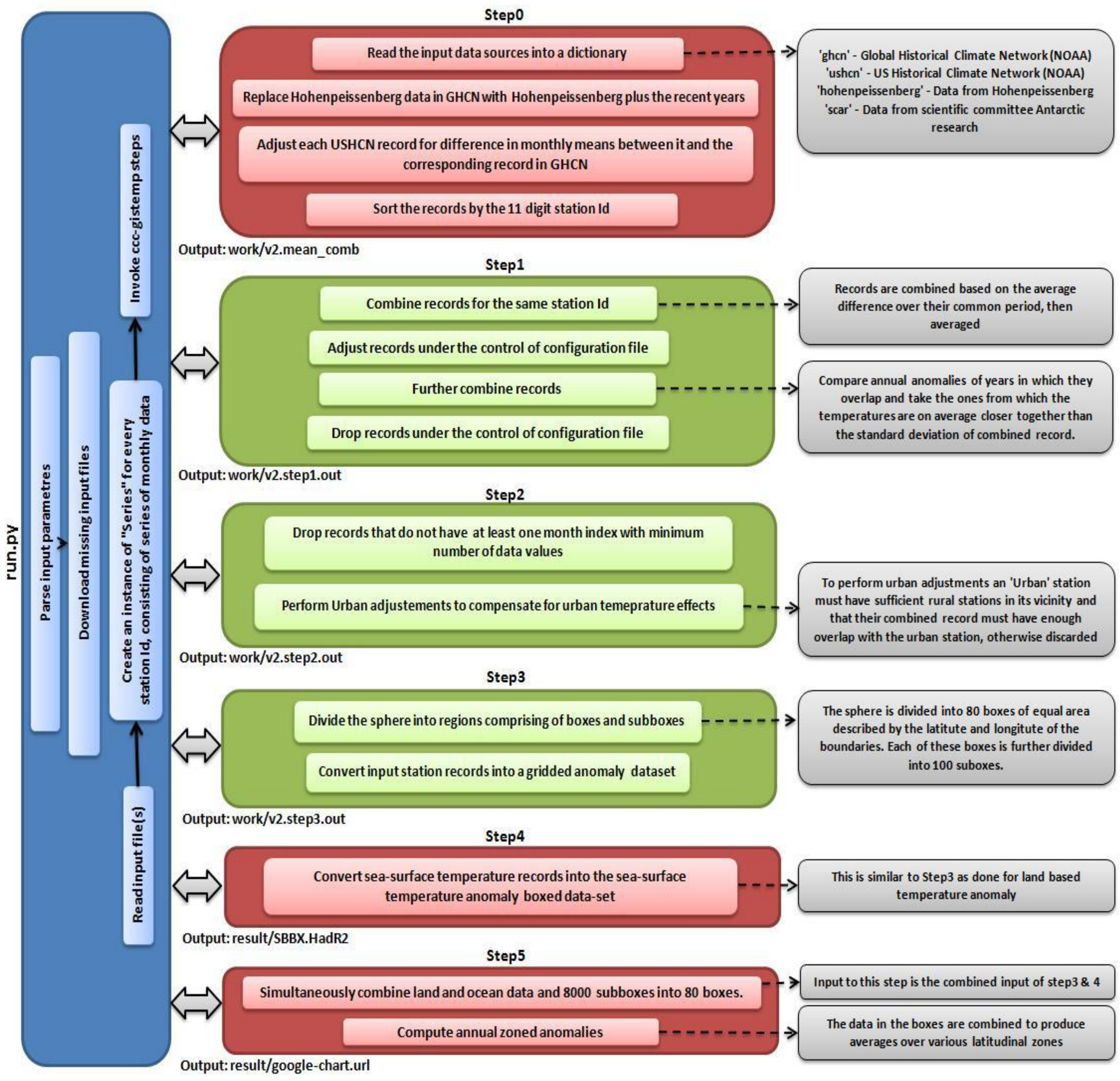

run.py
Parse input parametres
Download missing input files
Create an instance of "Series" for every station Id, consisting of series of monthly data
Invoke ccc-gistemp steps
Read input file(s)
Step0
Read the input data sources into a dictionary
Replace Hohenpeissenberg data in GHCN with Hohenpeissenberg plus the recent years
Adjust each USHCN record for difference in monthly means between it and the corresponding record in GHCN
Sort the records by the 11 digit station Id
'ghcn' - Global Historical Climate Network (NOAA)
'ushcn' - US Historical Climate Network (NOAA)
'hohenpeissenberg' - Data from Hohenpeissenberg
'scar' - Data from scientific committee Antarctic research
Output: work/v2.mean_comb
Step1
Combine records for the same station Id
Adjust records under the control of configuration file
Further combine records
Drop records under the control of configuration file
Records are combined based on the average difference over their common period, then averaged
Compare annual anomalies of years in which they overlap and take the ones from which the temperatures are on average closer together than the standard deviation of combined record.
Output: work/v2.step1.out
Step2
Drop records that do not have at least one month index with minimum number of data values
Perform Urban adjustements to compensate for urban temeprature effects
To perform urban adjustments an 'Urban' station must have sufficient rural stations in its vicinity and that their combined record must have enough overlap with the urban station, otherwise discarded
Output: work/v2.step2.out
Step3
Divide the sphere into regions comprising of boxes and subboxes
Convert input station records into a gridded anomaly dataset
The sphere is divided into 80 boxes of equal area described by the latitute and longitute of the boundaries. Each of these boxes is further divided into 100 suboxes.
Output: work/v2.step3.out
Step4
Convert sea-surface temperature records into the sea-surface temperature anomaly boxed data-set
This is similar to Step3 as done for land based temperature anomaly
Output: result/SBBX.HadR2
Step5
Simultaneously combine land and ocean data and 8000 subboxes into 80 boxes.
Compute annual zoned anomalies
Input to this step is the combined input of step3 & 4
The data in the boxes are combined to produce averages over various latitudinal zones
Output: result/google-chart.url

Figure 13: Flow diagram of the original ccc-gistemp code

ccc-gistemp steps 1-5 are triggered from *run.py* which also co-ordinates the sequence of execution. Each of these steps takes a data object as its input and produces a data object as its output. Ordinarily the data objects are iterators, either produced from the previous step or an iterator that feeds from an input file. An instance of "Series", the monthly temperature series for every year starting from the base year (set to 1880 by default, but can be changed to any value), uniquely identified by a 12-digit id is created for every station data. Multiple series can exist for a single station and hence a 12-digit id is chosen to uniquely identify the records, comprising of 11-digit station id and 1 digit series identifier. This unique id is used as the 'key' in step1 and step2 *map/reduce* functions. Step3 however has a different key/value combination.

Figure 14 is a diagrammatic representation of an instance of "Series" – the primary data object used in all the aforementioned steps for computation.

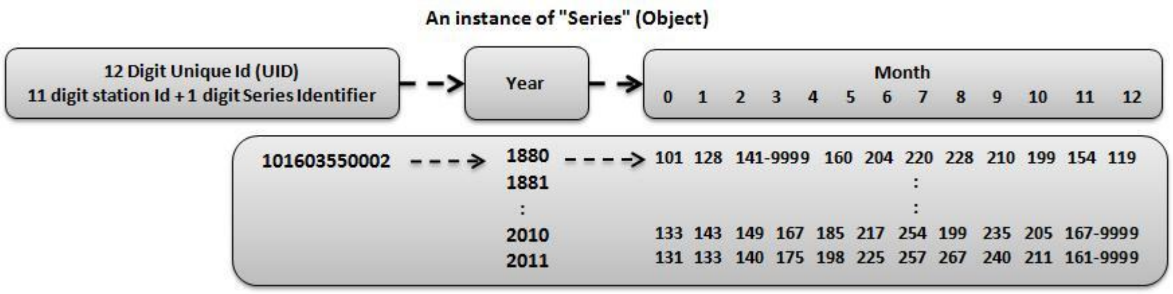


Figure 14: An instance of "Series" - The primary data structure containing Station id, Year and the monthly temperature series for all years

1. Step0

   Step0 reads the input data sources into a dictionary which primarily consists of station data, land and sea surface temperatures from GHCN and USHCN, Antarctic temperature readings from SCAR and the Hohenpeissenberg data. In the first part of this step, the Hohenpeissenberg data in the GHCN is replaced with the correct values from the actual Hohenpeissenberg data.

   In the second part of step0, the USHCN records are adjusted for difference in monthly means between it and the corresponding record in GHCN. Once adjusted, the corresponding record in GHCN is removed. Finally, the adjusted records in USHCN, remaining records in GHCN and the original SCAR records are joined together and sorted to generate the final output of step0.

   Analysis for MapReduce Programming Model

   The first part of step0 is a mere data replacement with no 'reduction' operation being performed to obtain any meaningful information. Thinking on lines of parallelisation, this step seems ideal for parallel processing with the GHCN records split among the available nodes (or reducers) and each node having access to the complete Hohenpeissenberg data to ease the compare-replace operation.

However, it must be understood that every operation (data-intensive or compute-intensive) that can be parallelised may not fit well into the MapReduce programming model. *mrjob* takes as input a single file, which is converted into key/value pairs before any processing.

In one possible approach to port step0 to MapReduce, let's assume the input file to *mrjob* are the GHCN records. The records in the input are converted as key/value pairs and yielded to the *reduce* stage which operates on its own local copy of the input data. Each instance of the reduce function should also have a copy of the Hohenpeissenberg data to facilitate compare and replace operation. The resulting data-set is yielded by each instance of the reducer, to the next stage reducer which performs the second part of step0.

The second part of step0 operates on USHCN records using GHCN records as reference but in our approach the records yielded from first part of step0 are that of GHCN and not USHCN. This change in the input stream will need rewriting of the logic from that of the original code. However, in this case, rewriting does not solve the entire problem. In addition to adjusting the USHCN records, this step also involves removing the corresponding adjusted record from another data source i.e., GHCN.

A global synchronisation across all reducer nodes will be required to combine copies of either GHCN or USHCN records (depending on the logic). Global synchronisation with compare-merge operation could be very expensive and does not fit into the MapReduce programming model. A workaround for this problem would be to load the contents of GHCN file into an external key/value store and have each reduce task concurrently access GHCN records from the store. Records that are adjusted can be removed from the store. Detailed explanation of the available key/value stores and its usage are mentioned in the step2 analysis. It must be noted that this is just a workaround and has serious performance issues. Every USHCN record processed by the reduce task will need access to the key/value store and this is done concurrently by all reduce tasks, severely degrading the performance of MapReduce job.

Additionally, it is important to note that MapReduce programming model is designed to perform operations on input data stream mapped as key/value pairs. Having to simultaneously operate on two independent input sources does not fit well into this programming model. In addition, joining of independent data sources (USHCN, GHCN and SCAR) cannot be performed within the *mrjob* framework. *mrjob* API does not offer support for operations outside the MapReduce programming paradigm. Workarounds and code hacks significantly deteriorate performance and hence does not form part of a good design. Bearing these design nuances in mind, step0 was not ported to MapReduce framework.

The final output is a stream of "Series" instances.

2. Step1

The output from step0 serves as the input to step1. In this step records from the same station (11 digit station id) are combined in a two stage process. In the *first stage* records are combined by offsetting based on the average difference over their common period, then averaged. In the *second stage*, the records are further combined by comparing the annual temperature anomalies of years in which they do overlap, and finding the ones for which the temperatures are on average closer together than the standard deviation of the combined records[50]. Finally, under the control of configuration files, a few station records are modified by adding a 'delta' to every datum for that station and a few station records are dropped from further analysis.

The records are initially grouped together by their 11-digit station id. The steps required to combine records are then applied to these groups. Algorithms designed to process data in groups make an ideal candidate to be ported to the MapReduce framework.

Analysis for MapReduce Programming Model

As the existing algorithms are written to process stations records in groups, these can be ported to MapReduce framework directly without much code changes. The input records are mapped as key/value pairs with 'key' being the 12 digit station id and 'value' being the "Series" temperature anomaly for each year. An intermediate reduce stage is used to construct the "Series" object from the input key/value pairs. This intermediate reduce stage yields the 11-digit station Id and the "Series" object, naturally resulting in data-grouping as required by the combining steps described above. The algorithms for combining records are then ported directly to the second stage reduce function.

An important point to note here is the ease of porting achieved when the algorithms are designed to use MapReduce programming model effectively. Algorithms designed to operate on groups of data rather than individual elements find ease of porting to MapReduce. These data-sets can easily be mapped into key/value pairs with *values* associated with the same *key* easily processed by algorithms in the reduce function. (Recall that the MapReduce framework assigns all *values* associated with the same *key* to a single reduce task). In some situations the existing logic may not be directly portable to MapReduce but with small changes in the data access pattern, data-intensive algorithms can be ported to use map-reduce. An example of such scenario is discussed in Step3.

Another important point that is worth mentioning here is the fact that complex logic underlying the algorithm need not be fully understood and comprehended to be able to port to MapReduce. All that is required to understand is the data-access pattern within the algorithm to be able to effectively tweak and re-use the same algorithm within the MapReduce framework. This fact was effectively used in porting step1 to MapReduce. The final output from step1 is a stream of "Series" instances.

[50] Based on the code comments by Nick Barnes and David Jones from the original ccc-gistemp code.

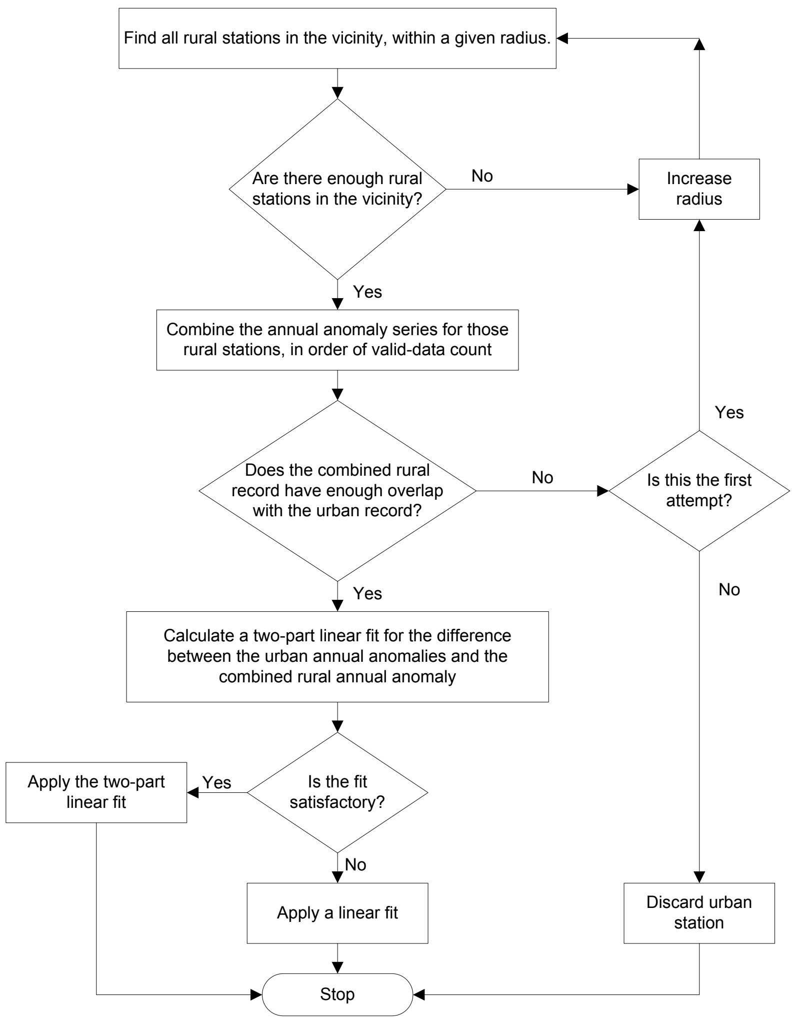
3. Step2

Output from step1 is the input to step2. An initial cleanup of the input station records is done prior to performing urban adjustments. The cleanup step is data-intensive while urban adjustment is mostly compute-intensive.

The input station records are cleaned by dropping records that do not have at least one month in a year with minimum number of data values. After the initial cleanup the remaining station records are classified and grouped as 'urban' and 'rural', with annotated objects generated for each of the records. Urban adjustment is applied to each of the urban stations to compensate for urban temperature effects[51], while rural stations remain unchanged. The urban stations that cannot be adjusted are discarded.

To perform urban adjustment, an urban station must have sufficient rural stations in its vicinity and that their combined record must have sufficient overlap with the urban station. The algorithm that performs the urban adjustment is as follows[52]:

For each urban station:


Find all rural stations in the vicinity, within a given radius.
Are there enough rural stations in the vicinity?
No
Increase radius
Yes
Combine the annual anomaly series for those rural stations, in order of valid-data count
Yes
Does the combined rural record have enough overlap with the urban record?
No
Is this the first attempt?
Yes
No
Calculate a two-part linear fit for the difference between the urban annual anomalies and the combined rural annual anomaly
Apply the two-part linear fit
Yes
Is the fit satisfactory?
No
Apply a linear fit
Discard urban station
Stop

[51] Discussed in section 2.3.2
[52] Based on code comments by Nick Barnes and David Jones from the original ccc-gistemp code.

Analysis for MapReduce Programming Model

The data-cleanup step is ideally suited for the MapReduce programming paradigm where the input station records are grouped by their 12 digit station id and processed independently by the available reducers. However, the step following the cleanup operation would require all records processed by the individual reduce tasks to be combined, so as to generate 'rural' and 'urban' classification of records. If there is no global synchronisation at this point then every reduce task will have their own copy of 'urban' and 'rural' classification for the records that were initially assigned to it. This causes issues when performing the urban adjustment.

From the above flowchart it is evident that each urban station will need access to complete rural station records in order to identify rural stations in its vicinity. This dependency between the records contained in each of the reduce tasks is not ideal for the MapReduce framework. Recollect that MapReduce is a programming model designed for processing large volumes of distributed data in parallel, by dividing the computational work into sets of *independent tasks* (Dean and Ghemawat 2004).

Additionally, the use of single reduce task to achieve synchronisation can have severe impacts on performance and scalability. Step2 being both data and compute intensive, having a global synchronisation (sequential execution in the MapReduce programming model) is a serious design flaw and must be avoided. This was also agreed by developers actively participating in the *mrjob* user community, when a query regarding this approach was posted[53]. One other option is to split the set of tasks performed in step2 into two separate stages. The two stage approach is followed in this project.

The *first stage* takes the input file and generates key/value pairs, with the 12-digit station id as the *key* and *value* being "Series" temperature anomaly for each year. The initial cleanup operation is also performed in this stage. The output of the *first stage* is a stream of "Series" instances generated from the cleaned up records (urban and rural combined).

As the records are annotated after the initial cleanup, they are ideally suited to be performed by the *map* task. The *second stage* map tasks generate the 'rural' and 'urban' classification of records for the input key/value pairs. These records, generated independently across all *map* tasks, are stored temporarily on some external key/value store. The use of external store is necessitated by the fact that MapReduce model does not provide any natural interface to store shared variables that are required for such an implementation. Additionally, it is essential to use a *key/value* store as the 'urban' records are directly referred to by their 'key' in the algorithm for adjusting urban stations.

[53] http://groups.google.com/group/mrjob/browse_thread/thread/f3b6bc74f07fd2

Key/value stores like HBase[54], PostgreSQL[55], Voldemort[56] and Redis[57] were evaluated for use in this project.

MapReduce implementations being distributed and highly scalable, the key/value store that is used in conjunction with MapReduce framework must also be distributed in nature. Also, basic database operations like ‘set’ and ‘get’ are sufficient to implement the required logic. Additionally, in the original *ccc-gistemp* code, all the annotated objects of stations classified as ‘rural’ are stored in a single list. Hence, it seems ideal to have a key/value store that allowed lists to be associated with a ‘key’.

Voldemort is an open-source distributed key/value store, but at the time of this work it did not have a stable off-the-shelf python client that could be used with ease in this project. PostgreSQL has an open-source python interface called PyGreSQL[58], but the use of PostgreSQL in this project seemed an overkill of resources as there is no need for a heavy object-relational database management system (ORDBMS). Similar conclusions were drawn for the use of HBase as well. An alternate would be to use the Amazon Web Services (AWS) which has *boto*[59] interface for Elastic MapReduce (EMR) and Simple Storage Services (S3). The *boto* interface is currently used by *mrjob* to set S3 keys and hence extending its services would not be a great challenge. Since we had already decided to benchmark on the EDIM1 machine, switching back to AWS was not a plausible option. A comprehensive study on the available SQL and NoSQL data stores can be obtained from (Cattell 2011).

Redis is an advanced open-source key/value store that allows ‘keys’ to be associated with data-structures such as strings, hashes, lists, sets and sorted sets. Also, Redis has a python client[60] that could be used with ease in this project. User-friendly documentation and it’s simple to use interface strengthened its choice for this project.

Changes to the data access pattern in the existing *ccc-gistemp* code were done to accommodate the use of key/value store. The original *ccc-gistemp* code used the “Series” object of urban stations as the key and its annotated object as the value to represent urban stations internally in a dictionary. This however seemed completely inappropriate to be used with the key/value store. Pickling and un-picking the “Series” object to be used as ‘key’ in the external store is very expensive and inefficient in terms of memory consumption. This was changed to have the 12-digit station id of urban stations as the ‘key’ and their annotated object as the ‘value’. All the annotated objects of stations classified as ‘rural’ were appended to a single list on the key/value store.

[54] http://hbase.apache.org/
[55] http://www.postgresql.org/
[56] http://project-voldemort.com/
[57] http://redis.io/
[58] http://www.pygresql.org/
[59] http://code.google.com/p/boto/
[60] http://pypi.python.org/pypi/redis/2.4.9

The output from the *map* tasks are grouped by the first 2 digits of the 12-digit station id. The use of first 2 digits results in nearly equally distributed work for each of the reduce tasks in the second stage. The reduce tasks performs urban adjustments on the input station records while referring to the key/value store for 'urban' and 'rural' records.

The key/value store is used with minimal changes to the existing code, commensurate with the ease of programming attribute of the MapReduce programming model. The use of distributed key/value store finds its greatest advantage when its 'distributed' attribute is put to use. This is achieved by the Redis *master-slave replication*, a feature that allows Redis slave servers to be exact copies of master servers. On the EDIM1 machine this can be achieved by configuring the Hadoop Namenode to also be the Redis *master* and slave nodes as the Redis *slaves*. Details of Redis replication and its configuration can be found at http://redis.io/topics/replication. Replication avoids the need for reduce tasks executing on slave nodes to connect to the master to obtain 'rural' and 'urban' records. Instead, they can connect to the Redis slave server running on each of the nodes to obtain the data, considerably reducing the bottlenecks arising from connecting to the master every time. This also results in efficient utilisation of the network resources.

Key/value stores can thus be used to share data across available reduce tasks without the need for global synchronisation with a single reduce task. However, in situations where the algorithm forces global aggregation (Zhu, et al. 2009), global synchronisation is inevitable with the current implementation of MapReduce.

The University of Edinburgh EDIM1 machine was available for use only at the end of the benchmarking phase (original work plan in Appendix-C) and hence due to time constraint, the scalability tests of *master-slave replication* have been left as a future enhancement to the project. However, the two stage approach and the use of key/value store have been tested locally on single node and Hadoop cluster, without replication.

The final output from step2 is a stream of "Series" instances.

4. Step3

Output from step2 is the input to step3. In step3, the input station records are converted into gridded anomaly data-sets represented as a box obtained by dividing the global surface (sphere) into 80 boxes of equal area as shown in the figure below. These boxes are described by a 4-tuple of its boundaries (fractional degrees of latitude for northern and southern boundaries and longitude for western and eastern boundaries), as shown in Figure 15.

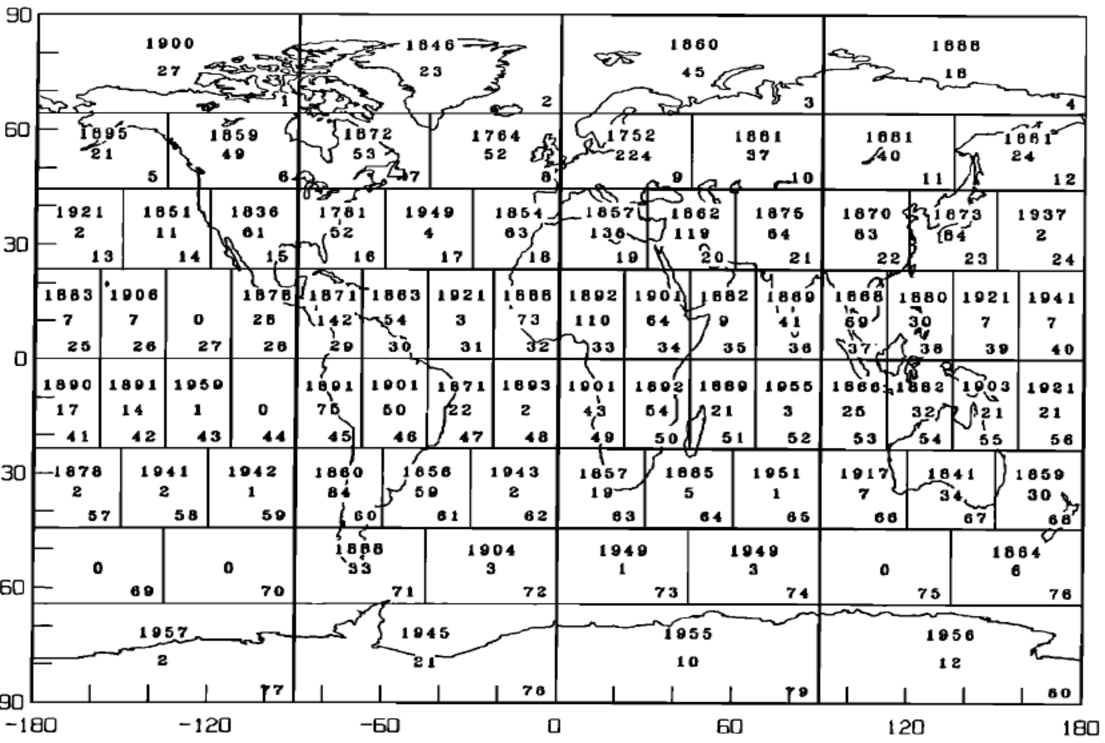


Figure 15: Global surface divided into 80 regions of equal area.
[The box is described by its co-ordinates given as fractional degrees of latitude (for northern and southern boundaries) and longitude (for western and eastern boundaries). The year within the box describes the time when continuous coverage began for that region. The number on the right corner is the box identification number. Source: (Hansen and Lebedeff 1987)]

Each of these 80 boxes is further sub-divided into 100 subboxes described by the same 4-tuple latitude/longitude representation. The input station records are iterated and assigned to the box in which they belong. Within the box, the station records are further iterated to assign them to the subbox to which they belong. The station records that belong to a grid cell are called contributors. The number of contributing records varies significantly from one region to the other. A subbox series (similar to station record "Series") consisting of monthly temperature anomaly is created for all records and returned.

However, it must be noted that the subbox series are represented as a 4-tuple of its boundaries when compared to the station records which were uniquely identified by their 12-digit station id.

Analysis for MapReduce Programming Model

The first thought would be to map the input station records as key/value pairs to be used for processing in step3, as done previously in steps 1 and 2. However, this mapping has severe drawbacks.

The original code is written to parse the input station records and assign then to the correct box and then subbox. If the input records are split across available reducers, each of them would process their own subset of the original records and assign them to grids created within each reducer. At the end of this step every reducer will have its own copy of the gridded anomaly data-set. There are 2 issues with this approach. Firstly, each of the subbox “Series” object is created with only the partial data available within each reduce function and secondly, there is no way of combing these independent subbox “Series” objects for the same station as the objects are already fully constructed within each reducer. Writing methods to mutate the read-only objects of “Series” is a serious design flaw.

The second approach would be to split the regions (boxes) across available reducers and have each reduce function independently read the input station records and assign records that belong to its region (box). The second approach is followed in this project. However, it is evident that this approach can be viewed more as a parallelisation strategy for compute-intensive step rather than data-intensive computing using MapReduce.

The ‘key’ is selected from one of the 4-tuples (latitude/longitude representation) and the ‘value’ is a tuple consisting of the region (box) and the subboxes within that region. Each of available reducers will compute the contributors for the region that was assigned to them, identified by the 4-tuples representation and yield the gridded anomaly dataset.

The final output is a stream of subbox “Series” instances.

An important point to note here is the fact that in order to port step3 to MapReduce, the underlying algorithm that parses each of the station records and assign them to the corresponding box and then subboxes, need not be fully comprehended. This further strengthens the ease of porting characteristic feature of the MapReduce programming model.

Although it was first observed that step3 is not ideal for porting to MapReduce, by slight modification of the data-access pattern, it was made possible to port step3. The conventional approach would have been to specify the output of step2 as an input to the MapReduce job which would then convert into key/value pairs. By modifying this pattern, no input was specified to the MapReduce job. Instead, the regions were directly read from within the map function and converted into key/value pairs consisting of one of the 4-tuples (latitude/longitude representation) as the ‘key’ and a tuple of region (box) and subboxes within that region as ‘value’. All regions associated with a ‘key’ will be processed by the same reducer yielding the gridded anomaly data-sets.

Additionally, it can be observed from Figure 15 that grouping by longitude will result in many more unique ‘keys’ than by latitude, which has just 8 unique numbers. As we already know that MapReduce assigns all ‘values’ associated with the same ‘key’ to a single reduce task, using latitude as the key will result in a maximum of 8 reduce tasks, severely impacting the scalability of the implementation.

Hence we choose the 'key' to be one of either eastern or western longitude. The results of benchmarking with both the combinations of keys are presented in section 8.2 of this report.

The final output from step3 is a land based gridded anomaly dataset of 8000 boxes.

5. Step4

Step4 converts the recent sea-surface temperature records into the sea-surface temperature anomaly boxed dataset. The initial steps are I/O intensive and the overall execution takes ~1.3 seconds on the St. Andrews machine (primarily used for development and initial testing).

Analysis for MapReduce Programming Model

The Hadoop implementation of MapReduce incurs considerable start-up costs which is usually amortised when processing large amounts of data (key/value pairs) in parallel across available nodes. However, if the data-set is small, these initial start-up costs dominate even when executed on large number of nodes.

As this step is neither data-intensive nor compute-intensive, it is not ported to MapReduce.

Output from step4 is an ocean based gridded anomaly dataset of 8000 boxes.

6. Step5

Output files from step3 (land data) and 4 (ocean data) zipped together forms the input to this step. These input files contain Subbox metadata as the header, followed by the gridded anomaly dataset ("Series" data) for the 8000 subboxes. The tasks performed by step5 can be enlisted as follows:

1. Assign weights to the input tuple, consisting of land and ocean records. The weight is set to 1 when referring to land records and 0 when referring to ocean records. This process is also called *masking* as we are attempting to mask land based records.
2. The masked data along with land and ocean records are output to an external file /work/step5mask.
3. Simultaneously combine land and ocean series in each of the subboxes and combine subboxes into boxes. As a result the 8000 subboxes are combined into 80 boxes.
4. Output the box data to result/BX.Ts.ho2.GHCN.CL.PA.1200.
5. Combine the box data to produce average over 14 latitudinal zones including northern hemisphere, southern hemisphere and global.

Analysis for MapReduce Programming Model

From the list above, steps 2 and 4 are I/O operations. By altering the sequence of operations slightly, steps 1, 3 and 5 can be grouped together to be analysed for the MapReduce framework. I/O steps 2 and 5 can be performed at the end but care must be taken to ensure storage of intermediate results.

Splitting the input land and ocean records across the available reduce tasks has drawbacks as already mentioned in Step3 analysis for the MapReduce programming model. Instead, the regions (boxes) could be split across available reducers with each reduce function independently read the land and ocean records and assign records that belong to its region (box). Additionally, with this approach each reducer will have the Subbox metadata that is required for processing all gridded anomaly dataset.

It must be noted that the input dataset will always be a tuple of land and ocean records consisting of 8000 lines each irrespective of the number of stations considered initially in step0 input. Hence, step5 output is not scalable in terms of the input dataset. The only gain in performance obtained is by parallelising the operations across available reduce tasks. This parallelisation can however be achieved in a manner similar to that of step3.

In the final step, the 14 latitudinal zones are obtained from 8 basic bands and 6 combined zones made from basic bands. The 80 boxes are decomposed into 8 bands with the number of boxes in each band explicitly specified in a list. This decomposition of boxes into bands with each band having variable number of boxes makes it hard to split the box data as key/value pairs to be used within the MapReduce framework.

The only gain in performance obtained is by parallelising a single step and that can be done in a manner to similar to step3. Since the goal of this project is not parallelising and optimising *ccc-gistemp*, and that the primary area of focus is to analyse various data patterns for its applicability to the MapReduce programming model, step5 was not ported.

Figure 16 shows the flow diagram of the ported *mapreduce-cccgistemp* code with the individual *map* and *reduce* functions for each of the steps that were ported to the MapReduce framework.

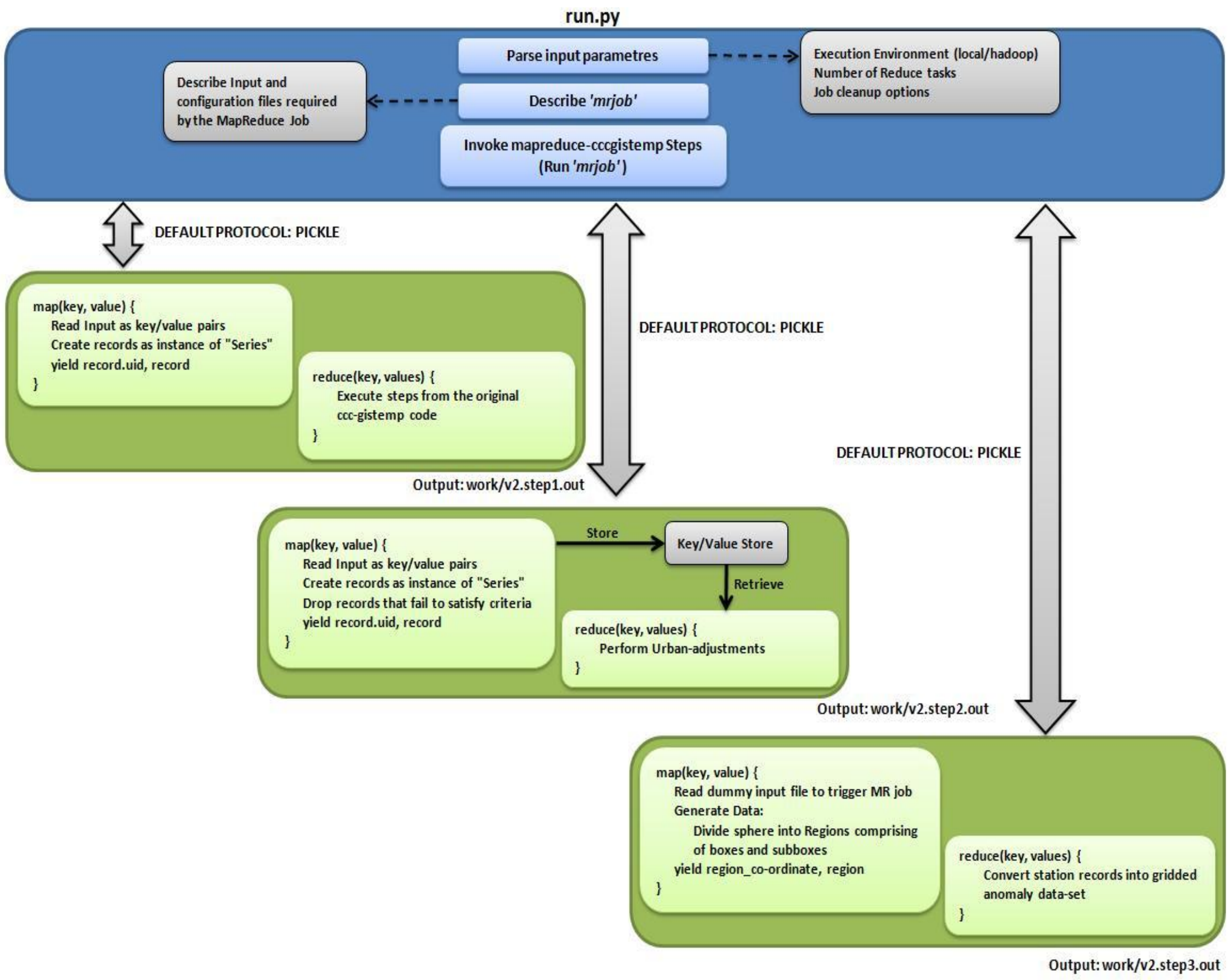

run.py
Parse input parametres
Describe 'mrjob'
Invoke mapreduce-cccgistemp Steps (Run 'mrjob')
Describe Input and configuration files required by the MapReduce Job
Execution Environment (local/hadoop)
Number of Reduce tasks
Job cleanup options
DEFAULT PROTOCOL: PICKLE
map(key, value) { Read Input as key/value pairs Create records as instance of "Series" yield record.uid, record }
reduce(key, values) { Execute steps from the original ccc-gistemp code }
DEFAULT PROTOCOL: PICKLE
Output: work/v2.step1.out
DEFAULT PROTOCOL: PICKLE
map(key, value) { Read Input as key/value pairs Create records as instance of "Series" Drop records that fail to satisfy criteria yield record.uid, record }
Store
Key/Value Store
Retrieve
reduce(key, values) { Perform Urban-adjustments }
Output: work/v2.step2.out
map(key, value) { Read dummy input file to trigger MR job Generate Data: Divide sphere into Regions comprising of boxes and subboxes yield region_co-ordinate, region }
reduce(key, values) { Convert station records into gridded anomaly data-set }
Output: work/v2.step3.out

Figure 16: Flow diagram of the ported mapreduce-cccgistemp code

Overcoming challenges while porting code to the MapReduce framework require good understanding of the data access patterns and its usage within the algorithms. It is not however very essential to comprehend the entire algorithm to be able to port efficiently. In situations where the existing algorithm cannot be ported to MapReduce, due to dependency between tasks, it is advisable to rewrite the algorithm bearing MapReduce framework in mind for maximizing performance. However, this contradicts the 'easy of porting' quality provided by the MapReduce framework.

7.2 Issues and bottlenecks with porting

1. Most algorithms in *ccc-gistemp* require the monthly temperature series sorted in the increasing order of 'year' for a particular station id. This however, is not guaranteed within the MapReduce framework. The *values* associated with a particular *key* can appear in any order within the reduce task for processing. In many data-intensive tasks, this is acceptable, where the sequence of operation is not very important.

In this project the values (monthly temperature series for every year starting from the base year, set as default to 1880) for a particular key (12 digit station id) was sorted in the increasing order of the available years before being processed by the algorithms in the reduce task. Assertion error results if sorting is not done when the current 'year' being processed are checked for 'previous year+1'. The maximum number of records that would be sorted for a given station id is 131 [current year – 1880], increasing by 1 every year. This is not a significant overhead on the reduce tasks.

2. As mention in Step5, the input to this step is a tuple of land records from step3 and ocean records from step4. It is important that a tuple is created with the same station id for both land and ocean records. However, the output from step3 can be in any arbitrary order depending on the records yielded by the reduce tasks. When these records are zipped together with ocean records from step4 using iterttools.izip, the station id's in the aggregated tuple isn't the same, leading to assertion error in step5.

   To overcome this problem, the records from step3 and 4 are independently sorted by their unique id (uid). Each of these steps will sort a total of 8000 records (80 boxes each having 100 subboxes). The overheads incurred by sorting is however not dependent on the initial number of stations. To better understand this let's consider an example.

   Let's assume we begin step0 with only 25% of the overall global temperature stations, generating temperature data. These input station records are converted into gridded anomaly datasets represented by 80 boxes (dividing the sphere into 80 boxes of equal area) in step3. If the input is increased to 75% of the overall global stations, the representation in step3 will still be 80 boxes. Hence the overheads incurred by sorting remains constant and become insignificant when compared to overall gain in performance obtained by running the MapReduce job in parallel across all available nodes. This observation can also be verified from the step3 benchmarking results presented in Appendix-E of this report.

   Sorting the data however will result in a format that is not acceptable by some functions in *compare_results.py* (script used to compare results of two executions). At this point of time, comparison of box data is ignored and updating the *compare_script.py* has been left as a future enhancement to the project.

From these issues it is evident that order of output from map and reduce tasks must not be assumed if this is important from the algorithmic perspective. Additionally, some issues may not be obvious when run locally on a single node machine. It is therefore important to test for all possible input combination and scenarios on multiple maps and reduce tasks to ascertain the port to MapReduce programming model.

7.3 Designing MapReduce Jobs

Based on the experiences from porting *ccc-gistemp* to MapReduce framework, the following summary can be drawn to facilitate developing and executing MapReduce jobs:

1. Examine the data input format and the underlying data structures within the code. Understand the code flow and data objects required by various algorithms, i.e. the data access patterns within the algorithm. Verify if the data can be grouped and mapped as key/value pairs.

2. Sample the original dataset to obtain data subsets for all testing purposes. Ensure that the data sampled is a good representation of the actual data to perform both black box and white box testing (systematic sampling in section 8.1).

3. Write map-reduce functions.

4. Identify dependency between data contained in each of the map/reduce tasks and verify if synchronisation is absolutely required to overcome this dependency. The two approaches discussed in this work (use of single reduce task for global aggregation and key/value store for sharing data between tasks) can be used to overcome dependency.

5. Configure MapReduce jobs to run in Hadoop environment (Hadoop Pseudo-distributed or Hadoop cluster setup).

6. Run job in Hadoop pseudo-distributed mode (locally) and compare output with that of the original code. Also, verify intermediate results (map and/or reduce outputs) to confirm that it is consistent with the expected output.

7. After ensuring that the ported code is performing as expected when run on single and multiple reduce tasks, run MapReduce job on Hadoop cluster.

8. Verify scalability (increase in the number of nodes and/or data).

Improvements in performance can be obtained by re-writing certain compute-intensive algorithms bearing MapReduce programming model in mind. This however is time consuming.

7.4 Building and executing the Project

Hadoop v0.20.2 was used in this project. The source code is available for download from the project website[61]. The installation and configuration details are provided in Appendix-D of this report.

When executing in Hadoop, all source files required for execution must be archived with path specified in the mrjob configuration file[62]. An example mrjob configuration (~/.mrjob) is as follows:

```
{"runners": {
 "hadoop": {
  "python_archives": [
    "/home/s1053340/Project/ccc-gistemp/mapreduce-cccgistemp.tar.gz",
    "/home/s1053340/UtilitySoftwares/redis-2.4.9/redis.tar.gz"
  ]
  }
 }
}
```

It is advisable to delete .pyc files from the source tree before archiving files, to avoid the unwanted side effects resulting from new changes being not picked up when executing Hadoop streaming jobs. A build script (./build_archive.sh) is created to automate this process which in turn is called by default in *run.sh*. The script *run.sh* is used to launch all jobs in the *mapreduce-cccgistemp* project.

This script accepts job configuration parameters such as execution environment (local or Hadoop), number of mapper tasks, number of reducer tasks and the job cleanup options as command line arguments.

7.5 Verification

In this section the various unit tests that were performed to ensure correctness of the ported *mapreduce-cccgistemp* code are described. Similar test cases are executed for all the steps that were ported, in addition to verifying the final result and output *google-chart*.

To aid comparison of intermediate results, the output from steps 1, 2 and 3 in both the original *ccc-gistemp* and the ported *mapreduce-cccgistemp* are sorted by their unique 12-digit station id. This is required as there is no guarantee of ordering of the results obtained from MapReduce in any particular fashion. Sorting by a common *key* will result in data being ordered with increasing values of station id.

[61] http://code.google.com/p/mapreduce-cccgistemp/downloads/list
[62] More information of mrjob configuration can be found at: http://packages.python.org/mrjob/configs-conf.html#module-mrjob.conf

A simple *diff* on both the output files (original *ccc-gistemp* and ported *mapreduce-cccgistemp*) will now compare and return lines if they are different. It was observed that the output of *diff* from comparing step1 intermediate files showed no difference in the output.

However, the output of *diff* from step2 intermediate files showed slight discrepancy in the values for some station ids, particularly for years lesser than 1910. A brief discussion of this error is provided in Appendix-G of the report. However, the presence of this minor discrepancy does not significantly alter the behaviour of the model and our benchmarking results and discussions.

Whilst mere inspection of intermediate files using *diff* is sufficient to check the stability of the ported code, a quantitative approach is required to ensure that the overall ported code has not in some way broken the original GISS model. Script provided in the *ccc-gistemp* code (section 3.2) can be used for this purpose.

A bug identified while using the compare script (compare_results.py) was fixed and also reported to the *ccc-gistemp* community to be incorporated in future releases.

Figure 17 shows the graph of global annual temperature anomaly by comparing the original ccc-gistemp implementation with the ported mapreduce-cccgistemp. Although the trends appear to be identical, minor difference in the plots can be observed in years ranging from 1880 to 1910 due to presence of discrepancy in step2 output.

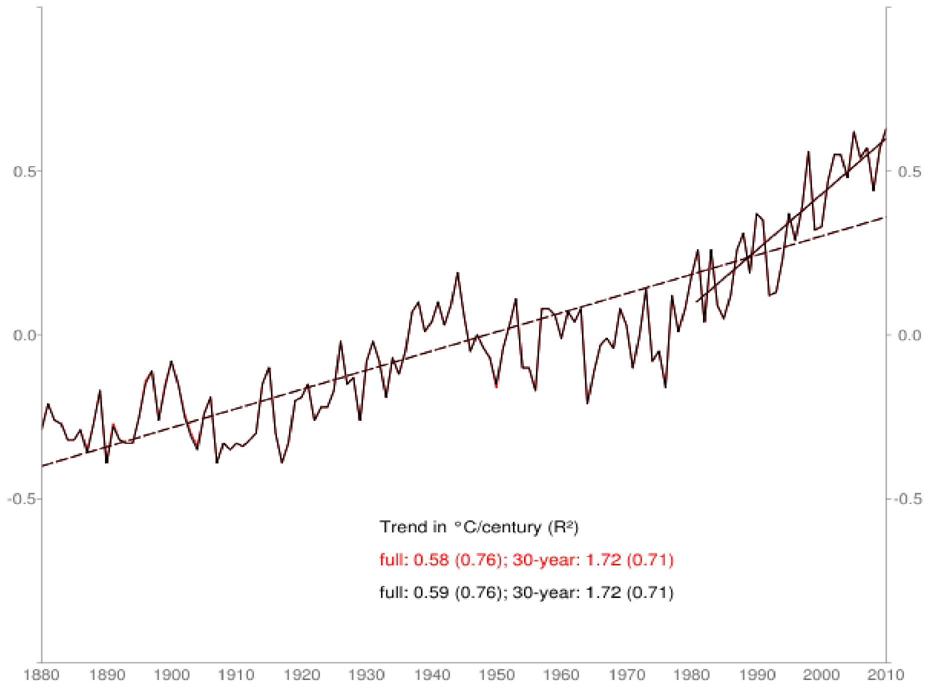


Figure 17: Graph comparing the global temperature anomaly of original ccc-gistemp code and ported mapreduce-cccgistemp
[mapreduce-cccgistemp in 'Red' and original ccc-gistemp in 'Black']

This difference is more obvious in Figure 18, where the plot show a difference of 0.01 for the same years where the plotted lines in Figure 17 are not identical.

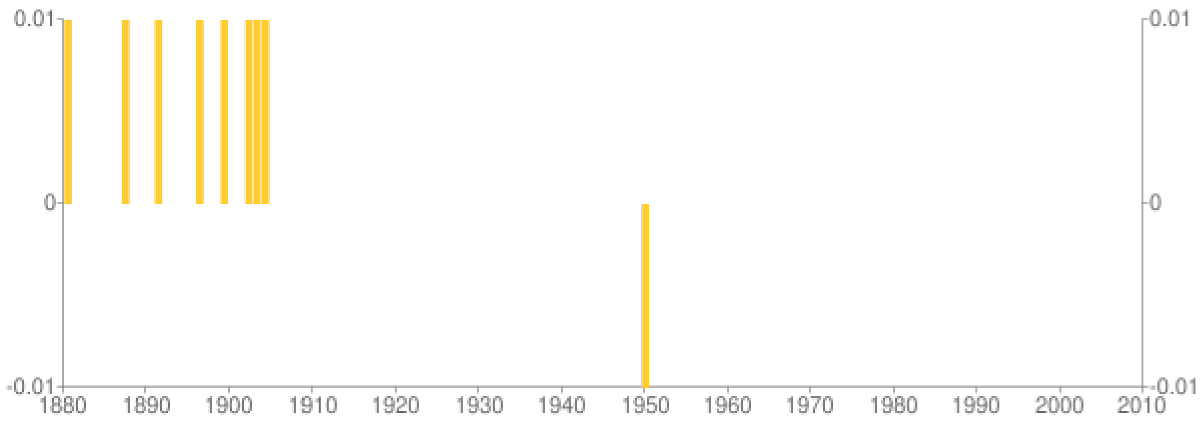


Figure 18: Plot indicating a difference of 0.01 for some years in the output of mapreduce-cccgistemp.

Chapter 8

Performance Evaluation

Performance and scalability analyses of the steps ported to MapReduce are discussed in this chapter. A brief overview of the sampling technique used to obtain data subsets for this analysis is also presented. Finally, the results obtained from porting and benchmarking exercise are summarized.

8.1 Data Sampling

Although a random sampling technique could be employed to randomly select lines from the input file, this technique is avoided as monthly temperatures for years that do not have an entry in the input file is assumed 'Missing'. This adversely alters the computation of global temperature change for a particular station. Hence we select all data available for a particular station and instead sample the input file based on the station id. Such use of systematic sampling[63] can easily be achieved as the input file to the MapReduce job is already sorted in Step0. Sorting of the input file in step0 is based on a compound key of station id and year, resulting in all available stations grouped together in ascending order of the compound key. Hence it is easy to pick all available data for a particular station. However, the choice of station id is done in random.

As an example, let's assume the input file v2.mean_comb has a total of 586173 lines. We sample this input file to obtain 25% of the total data which results in approximately 146543 lines. The simplest is when the first 146543 lines are picked up as a subset of the data, but care must be taken to ensure that all available years for a particular station id are chosen, even if it exceeds or results in lesser number of lines than 146543. The best choice is when the number of lines is as close to 146543 with all the information for a station id sampled.

In this project the first 25%, 50% and 75% of the total number of lines are yanked and copied to new file using vi commands to create the sample data sets, while ensuring that the yanked lines contain all information for the station id's. This check is done manually prior to executing the vi copy/paste commands. When sampling the first 50% of the lines, it was observed that the above mentioned criteria would be violated and hence the first 49.996% of the lines were yanked instead of 50% which is a fair approximation of the required data set. Similarly, 75.004% was chosen instead of the first 75%.

[63] Systematic sampling relies on arranging the target population according to some ordering scheme and then selecting elements at regular intervals through that ordered list.

It must be noted that the accuracy of final output obtained is proportional to the percentage of input data considered, when using sample datasets. As an example, the obtained global temperature anomaly maybe deviated from the desired output and the extent of deviation is proportional to the number of stations considered. Higher the number of stations, greater is the accuracy of the final output. However, experimenting with sampled datasets is an important aspect of the scalability test to assess performance of code and the underlying infrastructure, if in case new stations are added in the future.

8.2 Benchmarking

The timing results obtained is an average over two consecutive executions. It was decided that two executions were sufficient to determine the consistency of results as the variation obtained during each consecutive execution was small (less than +/- 1% always). However, anomalous results were executed again to verify its credibility. The sixteen nodes on the EDIM1 machine were dedicated exclusively for this benchmarking and no other user-specific tasks were run when the MapReduce job was in progress.

The timing results of step1, 2 and 3 executions are presented in Appendix-E of this report. The results were obtained by executing these steps on 2, 4, 8, 16, 20 and 24 cores respectively by varying the size of the input dataset.

Speedup was calculated using the formula: $Speedup = \frac{\textbf{MapReduce Sequential Run Time } (T_1)}{\textbf{Parallel Run Time } (T_P)}$

Efficiency was calculated using the formula: $Efficiency = \frac{\textbf{Speedup}}{\textbf{Number of Processing units}}$

1. Step1

   Table 3, Table 4, Table 5 and Table 6 presents timing result of step1 with respect to 100%, 75%, 50% and 25% of the dataset respectively. Figure 19 and Figure 20 present benchmarks of step1 to see how the ported MapReduce implementation scales with the increase in the number of processing units, processing cores of a node to be specific (Recall from section 6.1.2, that each processing node contains a dual-core processor).

   It is evident from the tables that the overall execution time decreases with the increase in the number of processing units. The tables also show that the time required to perform IO operations (IO time) remains nearly constant for a given dataset, indicating that the influence of IO on increasing processing units is minimum. MapReduce job time, which is a significant component of the overall execution time, is inversely proportional to the number of processing units clearly indicating that the MapReduce implementation is *efficient and scalable across processing units* with data size being the only limiting factor.

   The speedups were calculated by measuring the time it takes to process the datasets on a single core with one Map-Reduce task. It can be observed from Figure 20 that speedup increases with the increase in number of cores.

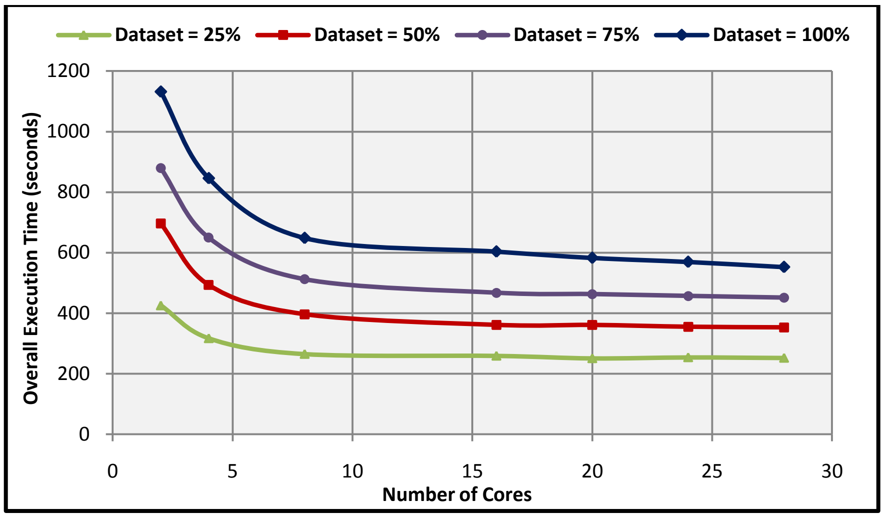


Figure 19: Plot of overall execution time to the number of cores for input dataset 100%, 75%, 50% and 25% respectively.
[Timing results presented in Appendix-E, Table 3, 4, 5 and 6 respectively]

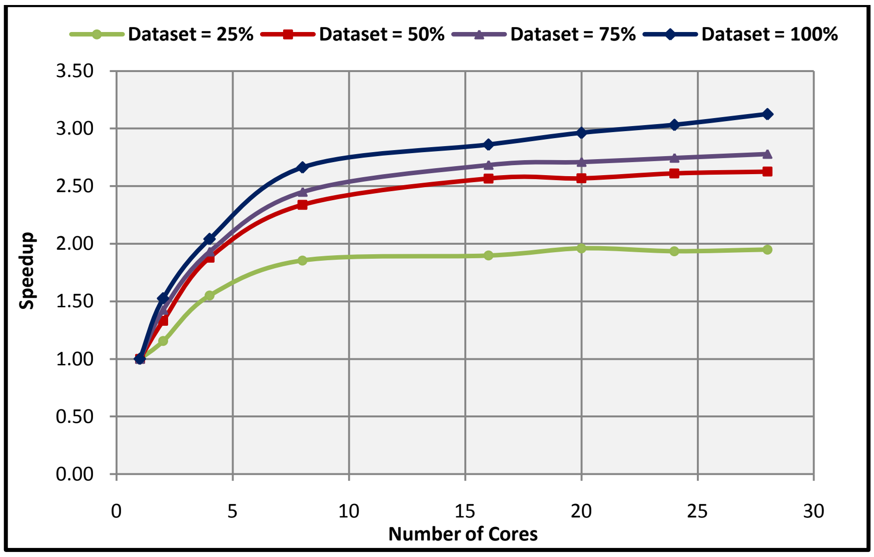


Figure 20: Plot of Speedup to the number of cores for input dataset 100%, 75%, 50% and 25% respectively.
[Timing results presented in Appendix-E, Table 3, 4, 5 and 6 respectively]

Additionally, it is evident from Figure 20 that for smaller datasets the speedup remains nearly constant beyond a certain number of processing cores. At this point the overheads associated with MapReduce implementation either negates or becomes nearly equivalent to the gain that MapReduce framework can provide with parallelisation.

To overcome this limit on the speedup that parallelisation can provide the data size was increased.

The Hadoop implementation of MapReduce incurs considerable start-up costs which is usually amortised when processing large amounts of data (key/value pairs) in parallel across available nodes. However, if the data-set is small, these initial start-up costs dominate even when executed on large number of nodes. The overheads are usually associated with reading/writing files to/from HDFS for the MapReduce job, sorting and shuffling the intermediate Map/Reduce output and transformation of the output as required by subsequent Map/Reduce stages.

Due to time constraints, quantitative analysis of overheads associated with every stage of the MapReduce job is left as a future work to this project.

Figure 21 and Figure 22 present benchmark of step1 to see how the ported MapReduce implementation scales with the increase in the data size. The speedup increases with the increase in the data size, clearly indicating that this implementation is scalable.

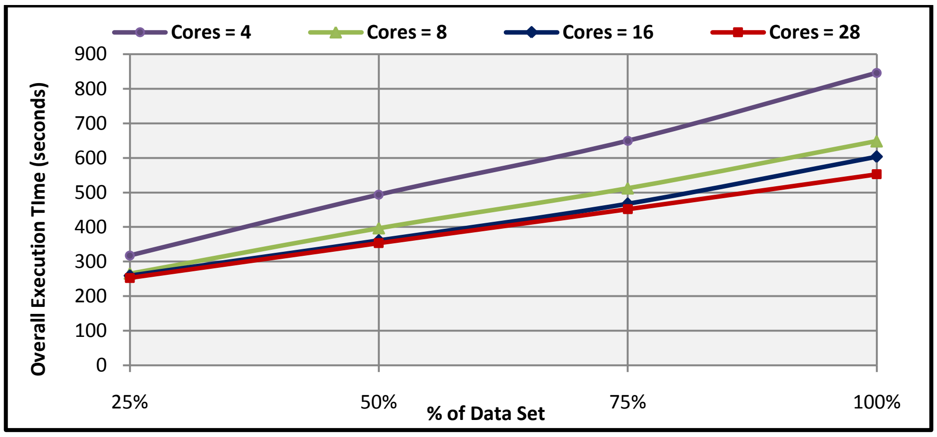


Figure 21: Plot of overall execution time to the % of dataset for processing units 4, 8, 16 and 28 cores.
[Timing results presented in Appendix-E, Table 3, 4, 5 and 6 respectively]

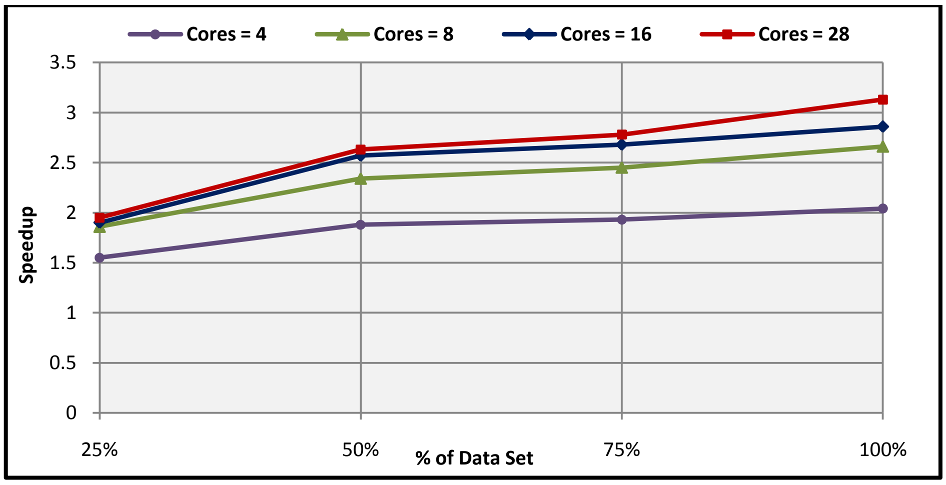


Figure 22: Plot of speedup to the % of dataset for processing units 4, 8, 16 and 28 cores.
[Timing results presented in Appendix-E, Table 3, 4, 5 and 6 respectively]

However, it can be observed that the improvement in speedup obtained is limited by the size of the original dataset (Recall that MapReduce is a programming model for processing very large datasets). Replicating every data point in the current implementation will definitely create a larger dataset but not necessarily a meaningful one that fits well with the GISS model. Hence, the scalability test is limited to the maximum available input data (number of meteorological stations currently contributing to station data) for the current GISS model.

2. Step2

Table 7, Table 8 and Table 9 present timing results of step2 with respect to the original ported code at 100% dataset, optimised code at 100% dataset and optimised code at 50% dataset respectively. 50% dataset is considered for scalability analysis.

While benchmarking step2 it was observed that the overall run time of MapReduce task was dominated by a single reduce task as shown in Figure 23 (task 4 in this example). By reviewing the input dataset it was identified that grouping values (station records) associated the first two characters of the key (12-digit station id) created severe imbalance in the number of records processed by each reduce task (Recall that MapReduce assigns all values associated with the same key to a single reduce task). Further investigations revealed that the number of records associated with the station id beginning with '42' (USA), particularly '42572#######', were very large compared to other station ids causing this imbalance.

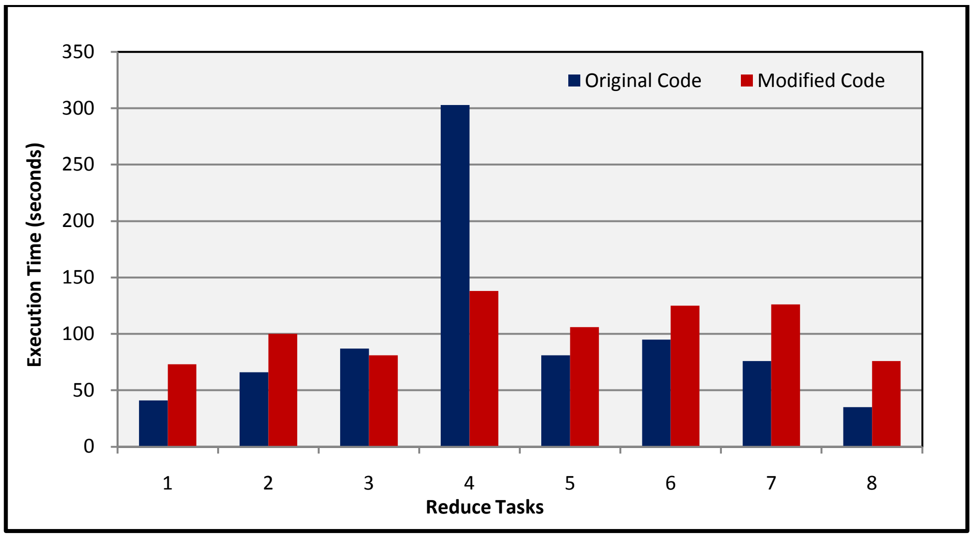


Figure 23: Distribution of reduce task runtimes on eight cores

The original ported code was then modified to account for this unequal distribution of values associated with the 'key' starting with '42'. It must be noted that the current implementation of MapReduce scheduler does not account for number of values associated with a 'key' before distributing work, which causes severe imbalance in applications with data skew. The assumptions of MapReduce scheduler is ideal for workloads that are evenly distributed. Current implementation of the MapReduce scheduler and its limitations are highlighted in step3 analysis, where its effects are more prominent.

Figure 24 and Figure 25 present benchmarks of step2 to see how the ported MapReduce implementation (original and optimised code) scales with the increase in the number of processing cores.

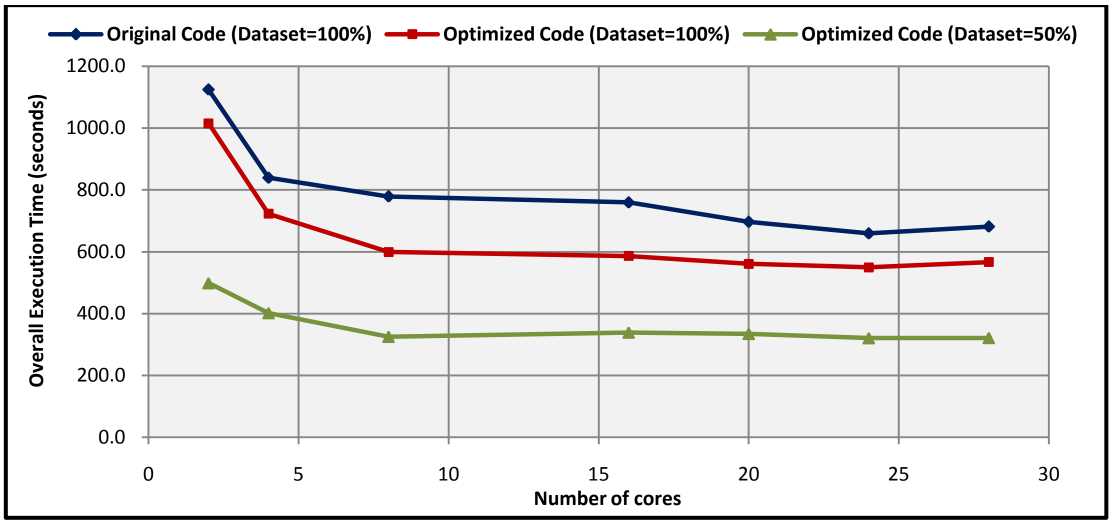


Figure 24: Plot of overall execution time to the number of cores for the original and optimised code with dataset=100% and optimised code with dataset=50%
[Timing results presented in Appendix-E, Table 7, 8 and 9 respectively]

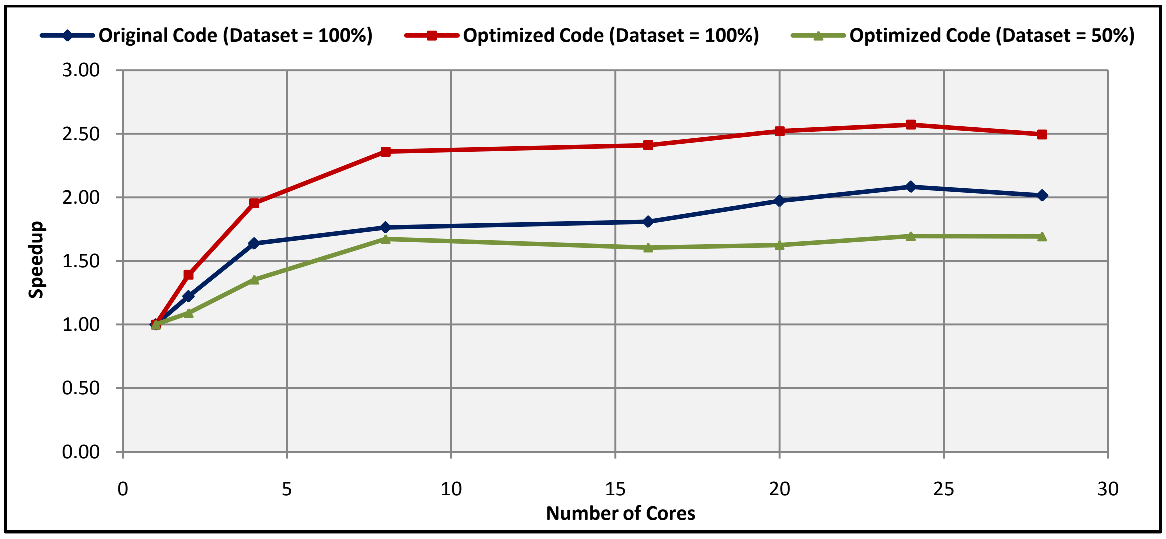


Figure 25: Plot of speedup to the number of cores for original and optimised code with dataset=100% and optimised code with dataset=50%.
[Timing results presented in Appendix-E, Table 7, 8 and 9 respectively]

It can be seen that the scalability of step2 is rather poor when compared to step1 with the increasing number of processing units. A number of reasons can be attributed to this poor scaling behaviour:

1. The number of unique keys in step2 is limited, and hence scaling beyond the maximum number of reduce tasks that can be created causes significant decline in performance due to the presence of idle processing units. The start up costs associated with the MapReduce programming model can only be amortised when all the processing nodes are busy performing nearly the same amount of work all the time.
2. Uneven distribution of workload due to uneven distribution of values associated with a 'key'.
3. Redis key/value store is not used with Master-Slave replication.
4. The input dataset is not large enough to overcome the above mentioned overheads.

However, it can be observed from Figure 25 that speedup improves with the increase in dataset (50% to 100%), indicating that this implementation is scalable with increase in data sizes.

In addition to step-wise evaluation, we attempt to evaluate the gain in performance obtained by avoiding the intermediate storage and retrieval of step1 output by combining MapReduce steps 1 and 2. The aim of this study is to merely identify the impacts of I/O overheads at an intermediate stage of MapReduce job, although this attempt is a deviation from the original *ccc-gistemp* architecture. Figure 26 presents the results of this experiment.

It can be observed that a significant gain in performance can be obtained by avoiding the intermediate storage and retrieval. An important point to note from this study is the fact the I/O operations are performance inhibitors to a scalable system like MapReduce and must be minimised as much as possible.

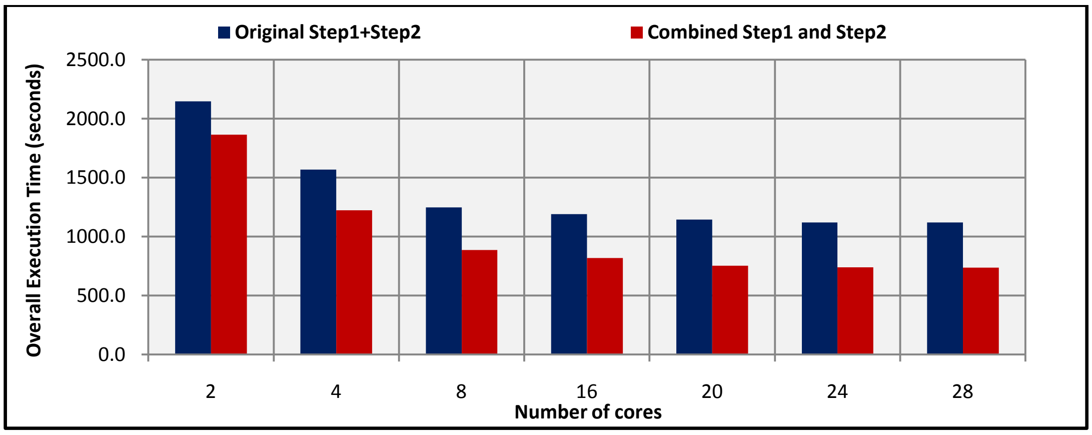


Figure 26: Performance improvement obtained by combining steps 1 and 2
[Timing results presented in Appendix-E, Table 10]

Figure 27 shows the gain in speedup obtained by combining steps 1 and 2. Additionally, it can be observed that speedup increases with the increase in processing units clearly indicating that this *implementation is efficient and scalable*, with data size being the only limiting factor. Additionally, when the processing units are increased beyond 20, the improvement in speed diminishes due to the poor scaling attributes listed previously.

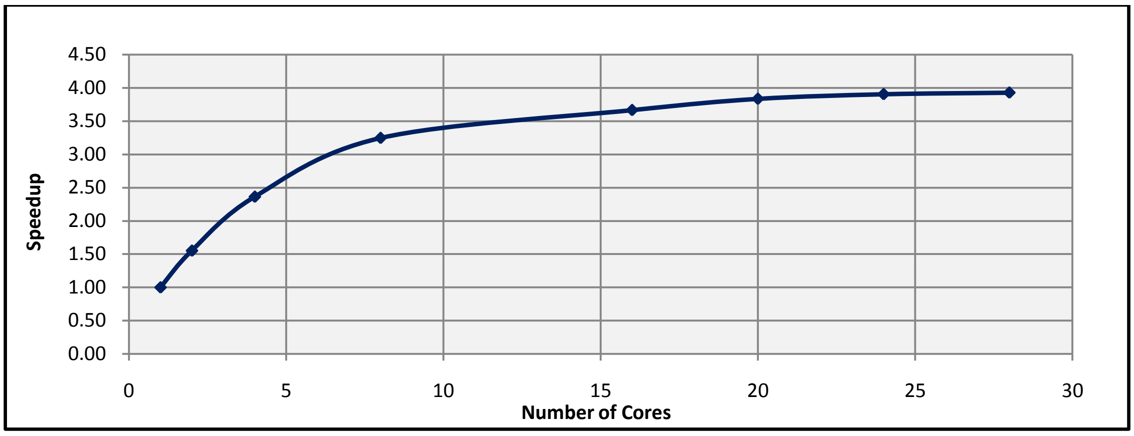


Figure 27: Plot of speedup to the number of cores with 100% dataset for combined MapReduce steps 1 and 2.
[Timing results presented in Appendix-E, Table 10]

3. Step3

Table 11, Table 12, Table 13 and Table 14 present timing results of step3 for keys western and eastern longitude at 100% and 50% datasets respectively.

Figure 28 and Figure 29 present benchmarks of step3 to see how the ported MapReduce implementation scales with the increase in the number of processing cores.

It is evident from the plots that the overall execution time decreases with the increase in the number of cores. MapReduce Job time, which is the dominant factor of the overall execution time, is inversely proportional to the number of processing units clearly indicating that this MapReduce implementation is *efficient and scalable across processing units.*

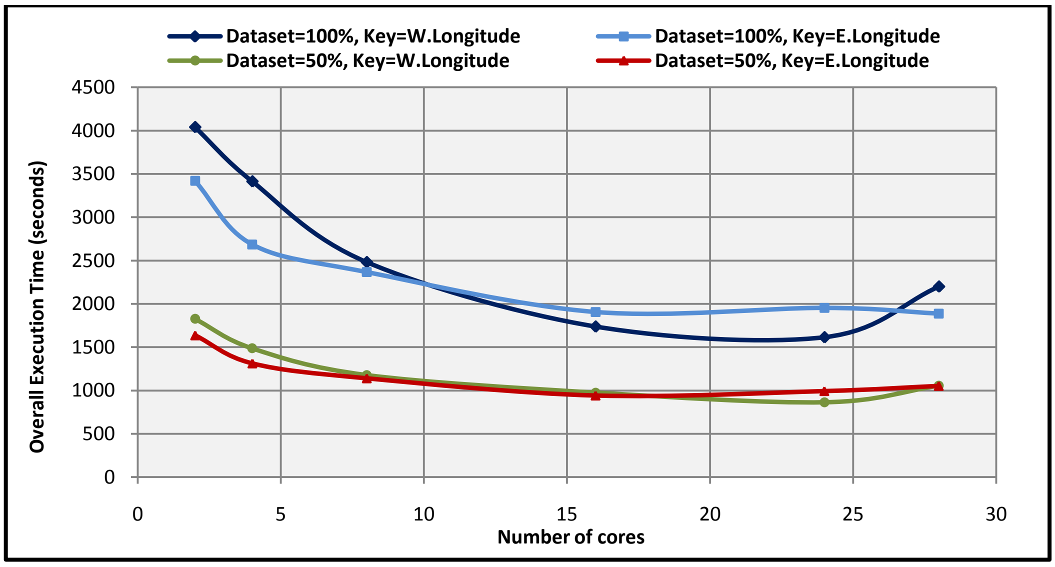


Figure 28: Plot of overall execution time to the number of cores for keys W. Longitude and E. Longitude at 100% and 50% dataset respectively.
[Timing results presented in Appendix-E, Table 10, 11, 12 and 13 respectively]

The speedup increases with the increase in processing units for up to 20 cores and then diminishes for reasons enlisted in the summary of step3 analysis. The plots also indicate that MapReduce implementation of step3 is scalable with the input data size.

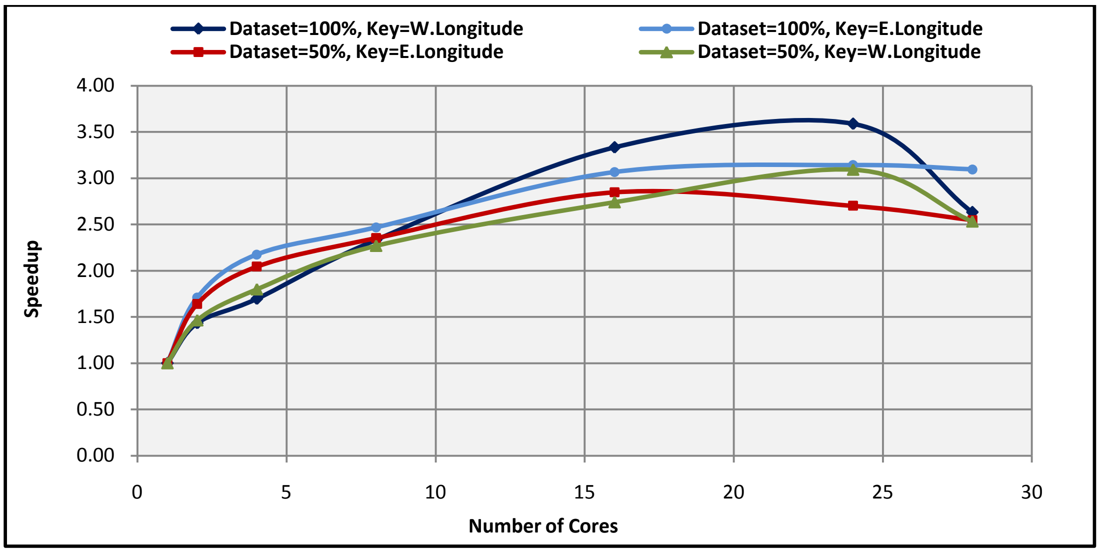


Figure 29: Plot of speedup to the number of cores for keys W. Longitude and E. Longitude at 100% and 50% dataset respectively.
[Timing results presented in Appendix-E, Table 10, 11, 12 and 13 respectively]

It can be observed from the plots that the choice of 'key' has impacts on performance, although not very significant but closely related to one of the assumptions made by the Hadoop scheduler[64] within the MapReduce paradigm. As already mentioned, any of the two coordinates (fractional degrees of longitude for eastern or western boundaries) could be used as the 'key'. Changing the 'key' results in regions being grouped differently to be processed by the reduce task, which in turn alters the amount of computation performed by each of the reduce tasks (Recollect from step3 analysis that the number of contributing stations vary significantly from one region to the other).

The dynamic load balancing strategy of MapReduce distributes tasks (map or reduce) to nodes as and when they finish processing the task at hand (Section 5.3.3). This strategy ensures that the regions, grouped by the 'key' are processed continuously and concurrently with no idle time. However, the scheduler assumes that the amount of work done by each reduce task is roughly the same (Zaharia, et al. 2008). This assumption sometimes causes unequal work load distribution which is particularly prominent in this example as the amount of computation required within a region depends on the number of contributing stations within that region. Currently, there is no way for the scheduler to obtain this piece of information while scheduling *reduce* tasks. Hence, the last set of tasks may finish at unequal times depending on the workload, causing an overall reduction is performance due to this 'slow' task.

[64] A process that handles the scheduling mechanism in Hadoop for distributing work across the cluster.

Figure 30 indicates the runtime distribution of reduce tasks in step3, dominated by a single reduce task (task 14 in this example). The plot also indicates unequal workload distribution across all available reduce tasks.

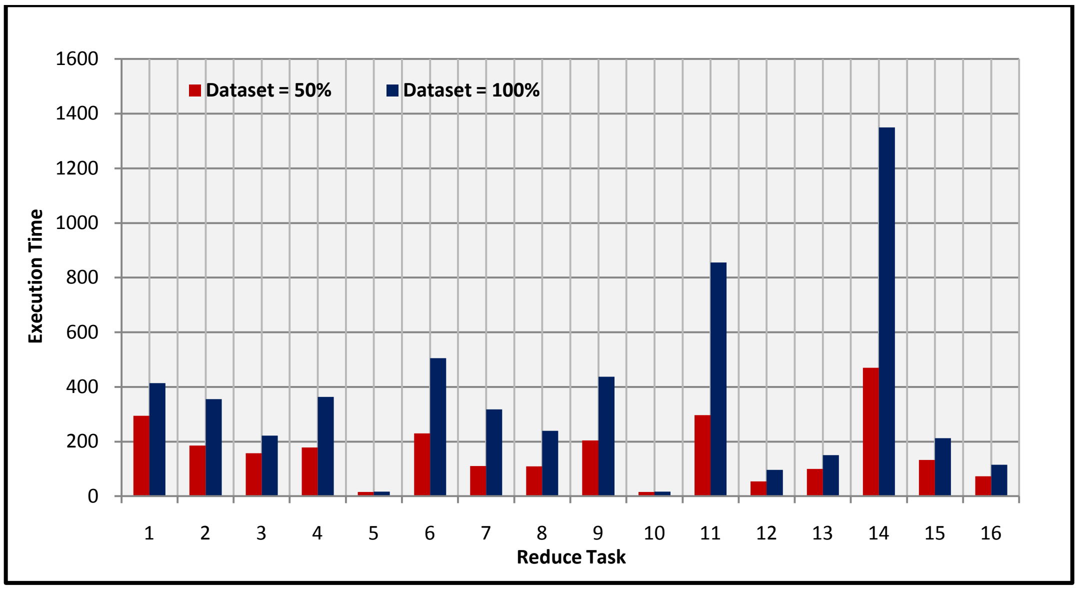


Figure 30: Runtime distribution of reduce tasks in step3 of mapreduce-cccgistemp

At this juncture, a question could be asked as to why the second load balancing strategy mentioned in section 5.3.3, also called “speculative execution”, is not helping to solve this issue. To answer this question it is important to understand how speculative execution works in Hadoop.

The execution for a reduce task is divided into three phases (Zaharia, et al. 2008), as shown in Figure 31:

1. The *copy* phase, where the outputs from the map task is fetched.
2. The *sort* phase, where the fetched map outputs are sorted by ‘key’.
3. The *reduce* phase where the reduce function is applied to the list of key/value pairs.

The reduce task progress is monitored using a progress score between 0 and 1 where each of the above mentioned phases contribute to 1/3 of the score. In each phase, the score is a fraction of the data processed and the total score is the sum of the scores for each of the defined phases. For example, a task quarter way through the reduce phase has a total score of 1/3 + 1/3 + (1/4 x 1/3) = 3/4. Based on the average progress of each reduce task, a threshold for speculative execution is defined. When the task has run for at least a minute and its progress score is less than the average minus 0.2, it is marked as a straggler (Zaharia, et al. 2008).

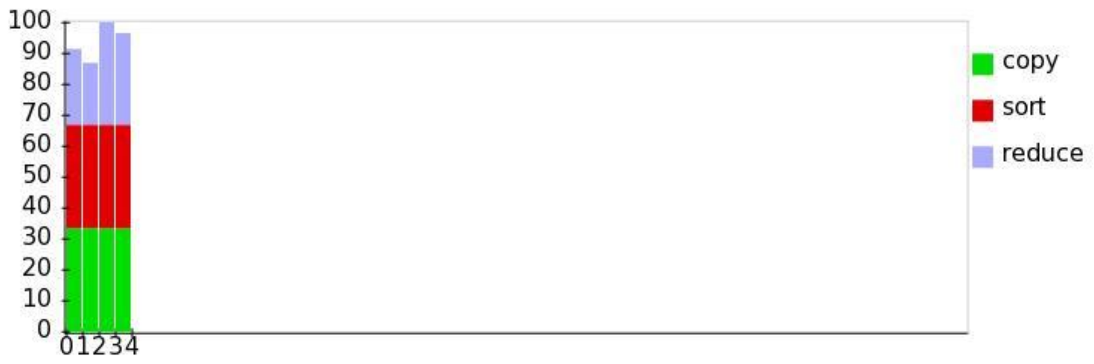


Figure 31: An example of reduce task execution indicating the three phases Copy, Sort and Reduce.

It must be understood that each of the reduce task is performing a lot of computation independently and there is no way of predicting the depth of computation for each reduce task. From the reduce task completion graph and '% complete field' of *jobtracker* it was noted that the task progress is high and hence not identified as a straggler. As a result of which the speculative execution does not kick in to solve the above problem. Thus it can be concluded that MapReduce is ideally suited for jobs that are large, but can be divided into smaller units of nearly equal size. A single large task can slow the overall performance.

In this example, the optimum performance is determined by a combination of the choice of 'key' and the available number of reduce tasks within the MapReduce framework for the job. If for example, a reducer task having all stations contributing in its region, will take longer to complete when compared to a reduce task having fewer contributing stations, even though the total number of regions are almost evenly distributed among the available reducers.

Hence, summarizing the reasons for diminishing speedup observed in Figure 29:

1. The number of unique keys in step3 is limited by the longitudes dividing the sphere (Figure 15). Thus, scaling beyond the maximum number of reduce tasks that can be created causes significant decline in performance due to the presence of idle processing units. The start up costs associated with the MapReduce programming model can only be amortised when all the processing nodes are busy performing nearly the same amount of work all the time.
2. Uneven distribution of workload due to processing of uneven number of contributing stations by each reduce task.
3. Current implementation of Hadoop load balancing strategy does not distribute workload based on the granularity of 'values' associated with a 'key', creating imbalance in the task execution times.

Kwon, Balazinska and Howe, 2011 and Gufler, et al., 2011 study the impact of variable task runtimes in MapReduce applications. (Gufler, et al. 2011), propose two new load balancing approaches that can deal with variable task runtimes while (Kwon, Balazinska and Howe 2011) suggest best practises to minimize the impacts of non-linear reduce tasks. Fine partitioning of the reduce tasks such that there are more partitions than the number of available reducers (currently, the number of partitions is equal to the number of reducers) can distribute chunks of complex reduce tasks evenly among the available reducers, significantly minimizing the impact of variable task runtimes. The application knowledge was however utilized in the selection of keys for the MapReduce programming model but redesigning a complex compute intensive algorithm for the MapReduce framework requires domain expertise.

Additionally, in step 2 and 3 analyses we observed the impacts of balancing load based on key granularity. The overall execution time in step2 was however improved by using the knowledge of application and input dataset but it may not always be possible to incorporate this level of granularity at the coding level, for example in step3. It would have been much better to have a holistic load balancing strategy that automatically handled load distribution based on the granularity of 'values' associated with a 'key', as suggested in (Gufler, et al. 2011).

Figure 32 shows the profiling of the ported *mapreduce-cccgistemp*, where the MapReduce steps 1, 2 and 3 are parallelised across 16 cores. The CPU bound steps 1 and 3 have found significant improvement in performance by distributing the compute-intensive tasks across 16 cores when compared to the original *ccc-gistemp* profiling chart in Figure 12. Although the I/O intensive step 0 has not been ported to MapReduce it finds benefits executing on the EDIM1 machine owing to its low latency and high I/O bandwidth. The improvement in performance of step2 is however not as significant as that of steps 1 and 3, for reasons already explained in the benchmarking analysis. Thus it has become a more dominant part of the entire application profile.

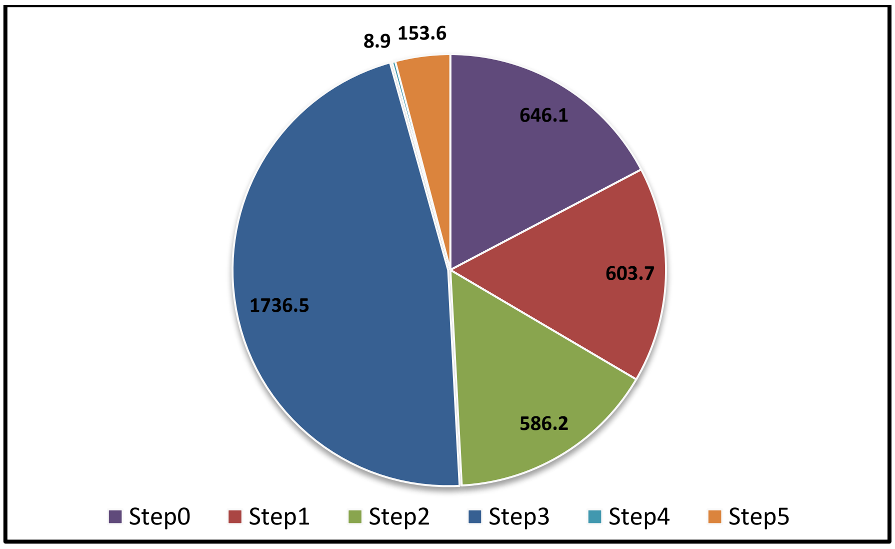


Figure 32: Profiling of the ported mapreduce-cccgistemp code on 16 cores (seconds)

8.3 Analyses of Results

This section outlines the analysis that can be drawn from the porting exercise and benchmarking results.

8.3.1 Impacts on Scalability

Scalable algorithms are highly desirable in both compute-intensive and data-intensive applications. Lin & Dyer, 2009 define scalability along two dimensions ideally applicable for data-intensive computing. First in terms of data: Given twice the amount of data, the same algorithm should take at most twice as long to run. Second, in terms of computing resource: Given a cluster twice the size, the same algorithm should take no more than half as long to run. In addition, such scaling characteristics must remain constant across various data ranges, from gigabytes to terabytes, and on clusters consisting of few tens of nodes to a few thousands. However, in reality, algorithms with such linear scalability are unobtainable.

It is evident from step1 and step3 analysis that MapReduce programming model is efficient and scalable across processing units and data sizes. Increasing the data and/or computation negates the impact of overheads induced by MapReduce programming model, thereby improving the overall speedup.

In Hadoop streaming jobs, the python process reads data from another running process (typically JVM or storage system process) through certain inter-process communication schemes such as TCP/IP and JDBC (Jiang, et al. 2010). The use of streaming I/O results in reduced performance when compared to executing the job directly on the target process (in this case JVM). However, for jobs written in Python programming language to use the Hadoop MapReduce framework, this loss in performance due to I/O overheads is unavoidable.

Hadoop MapReduce uses block scheduling scheme for assigning input data to the available nodes for processing, dynamically at runtime. This runtime scheduling strategy enables MapReduce to offer elasticity and remain fault tolerant by dynamically adjusting resources (adding nodes for scalability and removing failed nodes for fault tolerance) during job execution. However, it introduces runtime overheads that may slow down the execution of MapReduce job.

Additionally, it was observed that MapReduce implementations are ideally suited for processing large amounts of data, an important attribute for data intensive computing. Also, skewed data in compute intensive processing can have significant impact on the overall performance. Improved load balancing strategies can mitigate the impacts of skew, thus enabling MapReduce to provide an ideal programming abstraction for processing data and compute intensive scientific applications.

8.3.2 Time and ease of Porting

In distributed memory architectures, parallelising sequential code with MPI would require significant amount of time to alter the existing code structure to use the MPI library. It is also very essential to have a thorough understanding of the existing logic to be able to efficiently port the code.

In this work, it was observed during the porting exercise that it is not very essential to comprehend the entire algorithm to be able to port to MapReduce. However, it is essential to understand the data-access patterns within the algorithm to be able to modify the algorithm to operate on key/value pairs. Additionally, it was observed that algorithms designed to operate on groups of data find ease of porting to MapReduce. These datasets can easily be mapped as key/value pairs with values associated with the same key processed by algorithms ported to the reduce function.

Algorithms that induce dependency between tasks while processing find it hard to be ported to MapReduce (Recollect that MapReduce programming model operates by dividing computational work into sets of *independent tasks*). Since MapReduce framework does not provide any direct interface to share data between dependent tasks, alternate techniques such as synchronisation with a single reduce task and use of external key/value store for shared data can be incorporated to overcome this limitation.

Various limitations of MapReduce were studied and overcome in a fairly small amount of time, in addition to comprehending the existing *ccc-gistemp* architecture and data-access patterns within the algorithm. It is also worth mentioning the fact that MapReduce framework enabled the author of this work to focus on the computation at hand while the framework automatically handled the messy details of parallelisation, distribution of computation, load balancing, task management and fault tolerance.

Thus it can be concluded from this porting exercise that the time and effort required to port the code when compared to the scalability obtained is quite low, when compared to other parallelisation techniques like MPI.

Chapter 9

Conclusions and Scope for future work

This chapter presents the conclusions that have been drawn from porting a scientific application code to the MapReduce programming model. Also, the risks that were identified initially during the project preparation phase are reviewed to assess their impact on the satisfactory completion of this work. Finally, the scope for further work is proposed by outlining some suggestions to improve the performance of ported code and to continue evaluation of other programming abstractions to provide a comprehensive study on this topic.

9.1 Conclusions

The following conclusions were drawn from porting a scientific application code to the MapReduce programming model:

MapReduce provides the necessary programming abstraction for parallelising data and compute intensive steps of a scientific application code.

A scientific application code from the environmental sciences was chosen to evaluate the applicability of MapReduce to parallelise data and compute-intensive tasks. At the onset of this work, there was no evidence of any prior evaluation of this application to verify its credibility to use with the MapReduce programming framework. Attributes of the implementation strategy such as data volume and data-access patterns in the algorithm were applied to evaluate feasibility of parallelising data and compute intensive tasks.

The results obtained as part of this work were encouraging for a large part of the application, which was either data and/or compute intensive. Porting of certain tasks was however made difficult with the current implementation of Hadoop MapReduce, but alternate techniques were employed to overcome this limitation. Additionally, the Map-Reduce framework enabled the author of this work to focus on the computation at hand while ignoring the underlying complexities pertaining to partitioning the input data-set across the clusters, scheduling, handling machine-failures and communication which were automatically handled by the framework.

We believe the choice of application in this project was instrumental, as it provided a good insight into what code semantics and algorithm designs are best suited for MapReduce and what isn't. In addition, experimenting with alternative approaches and workarounds proved useful in overcoming some of the limitations imposed by the current implementation of MapReduce, while some were not suitable for MapReduce at all.

The MapReduce programming framework is capable of handling applications with varying indices along the three axes of being dynamic, distributed and data-intensive.

Analyses of results from scalability tests combined with the ease of porting attribute of MapReduce proved that this model provides the necessary abstraction for handling applications with large volumes of distributed data. It was also observed that the input data did not have to conform to a well definite schema as necessitated by relational databases. This is also confirmed by a study conducted by Pavlo, et al. 2009.

Application whose input dataset changes with time is said to be dynamic. We have already seen that MapReduce implementations are scalable with the increase in data size. However, it was observed from step2 analysis that a sudden increase in input data associated with a particular station created severe imbalance in the amount of computation performed by each of the available reduce tasks. This was seen as a limitation of the current Hadoop load balancing strategy which is based on the granularity of keys. Suggestions and alternative implementation techniques were discussed. Hence the current implementation of Hadoop MapReduce is not ideally suited for handling dynamic variation in input data that creates imbalance in grouping based on key/value pairs. However, if there is a sudden increase in the input such that they can be uniformly distributed, then MapReduce implementations are indeed ideal abstractions for dynamic, distributed, data-intensive computing.

Implementations of MapReduce programming model on infrastructures consisting of low-end commodity machines are cost-effective and efficient for data-intensive computing.

The scalability tests were performed on EDIM1 machine, which is a cluster of commodity machines built from inexpensive hardware. Additionally, the implementation was scalable both in terms of data and computation.

The MapReduce programming model can cope with system failures gracefully by executing the failed tasks on other available nodes. Thus it can be concluded that huge investments on high-end machines are indeed unjustifiable to obtain the necessary computational power for data-intensive computing and that MapReduce implementations perform equally well on low-end commodity servers.

9.1.1 Risk Assessment

A number of risks to the successful completion of this work were identified initially during the project preparation phase. Some of these risks whose probability of occurrence was high are reviewed here to assess its final impact on the satisfactory completion of this work. Additionally, the unlikely risk of not having a basic infrastructure to run and test applications which became apparent during the course of the project is reviewed. The original risks are included in Appendix-B of this report.

1. Aggressive and ambitious project plan

   This work was started with absolutely no prior knowledge of either the programming model (MapReduce), application domain (Environmental sciences) or the programming language (Python) and hence posed a significant risk to the overall completion of the project. A reasonable amount of time from the project preparation phase was allocated to understand the MapReduce programming model, in addition to finding a suitable application to be used with this model.

   After selecting the application, time from the "application evaluation" phase had to be dedicated to quickly comprehend the basics of Python programming language before actually attempting to port the code. Instabilities in the StACC cloud infrastructure required reinstallation of the necessary software and application code many a times before actually deciding to develop code locally and transfer it to the cloud infrastructure only for testing. The use of Google project hosting aided in this, while ensuring backup of the code. Additionally, the use of agile development techniques mitigated much of the risks associated with aggressive and ambitious project plan (section 6.4).

2. Absence of basic infrastructure to execute and test scalability of ported MapReduce application.

   The impact of this risk became apparent during the course of the project. The single node setup on the StACC was sufficient for code development and basic testing. Much of the scalability tests were planned on the EDIM1 machine. However, this machine was not available until the first week of August, 2011, two weeks later than expected (Original work plan in Appendix-C). Numerous issues with the machine setup and availability of system administrators caused the delay. The remaining time was just sufficient to perform all the required scalability tests and analyse their results, in addition to concluding the dissertation with these results. Thus, much of the work such as benchmarking the EDIM1 machine for its suitability to run MapReduce jobs, fine tuning the infrastructure to make it suitable for the ported MapReduce application and evaluating the impacts of various individual contributors in a MapReduce job (input/output, sort and shuffle) had to be suspended and added as a future enhancement to this work.

The impact of this risk could have been severe if the waiting time had not been utilized fully to write and complete other sections of the dissertation. However, it was quite hard to conclude on certain aspects of the project without the result from scalability analysis. Additionally, some parts of the dissertation had to be rewritten based the results obtained and optimisation performed.

3. Chosen application not suitable for Dynamic, distributed, data-intensive computing and incorrect choice of programming abstraction framework

   GISTEMP, a model for estimating the global temperature change from the environmental sciences, consisted of steps that were both data and compute intensive, ideally suiting our requirement for evaluation with the MapReduce programming model. The impact of this risk was mitigated after completing the first iteration (agile implementation). Sufficient confidence was gained that the chosen application was indeed suitable to be evaluated with the MapReduce framework, encompassing many aspects of this model including the advantages and its limitations.

4. Incorrect software development paradigm

   The use of agile software development techniques proved very useful in the satisfactory completion of this work. Many of the above mentioned risks were mitigated by the 'incremental development followed by testing' approach attributed by the agile software development methodology. Efficient time management and constant review of progress helped mitigate the overall risk associated with the aggressive and ambitious project plan.

5. Design deviation

   Thorough evaluation before implementation prevented any negative impacts of design deviation. Optimisations were performed based on the results from scalability analysis.

6. Poor schedule

   The original work plan (Appendix-C) was followed as closely as possible but inconsistencies in the development and testing infrastructure had significant impacts on the time available for benchmarking and optimising the *mapreduce-cccgistemp* code.

9.2 Scope for Future work

The scope for future enhancement in presented in this section.

The main area of focus in this project has been to evaluate the applicability of data and compute intensive tasks of *ccc-gistemp* to MapReduce programming model. Various approaches and techniques to efficiently parallelise these tasks have been discussed. Although most steps have been parallelised to use MapReduce efficiently, they are not fully optimised.

It is essential to perform quantitative analysis of the overheads associated with MapReduce framework to ascertain with confidence that a particular implementation is scalable. Certain overheads have significant impact on the overall performance and hence must be studied in detail to negate their effects. Due to unavailability of EDIM1 machine in time to perform a quantitative analysis of the overheads, this has been left as a future work to the project.

Porting of *ccc-gistemp* to other scalable systems intended for data-intensive computing such as Dryad (Isard, et al. 2007), All-Pairs (Moretti, et al. 2010) and Pregel (Malewicz, et al. 2009) will provide a comparative study of the various programming abstractions that are suitable for dynamic, distributed and data-intensive computing. The advantages and limitations of each of these abstractions will provide a comprehensive study on the subject which will aid in classification of applications that are particularly suitable for a class of programming abstraction.

Benchmarking the EDIM1 machine to ascertain the various system properties such as CPU consistency, disk throughput, memory bandwidth, network bandwidth and latency will aid in better comparison of the execution environment and cost-model of machines intended for cost-effective data-intensive computing.

Further investigations into the minor discrepancy observed in step2 output can be carried out in conjunction with other ccc-gistemp developers, to find a stable solution to this problem.

Distributed key/value stores like Voldemort, HBase, PostgreSQL and Redis were evaluated for use in this project. However, their performance with respect to the execution environment was not compared prior to making the selection. As previously stated, MapReduce being a highly scalable system, the key/value store using in conjunction with MapReduce must also be scalable. The choice of Redis in this project was thoroughly based on the simplicity of its interface which satisfied the requirements of its usage in step2. A comparative study of all the applicable key/value stores based on its performance on the EDIM1 machine would have further strengthened the choice, in addition to simplicity and ease of use. Additionally, a high throughput low-latency key/value store would improve the overall efficiency of the system.

The University of Edinburgh EDIM1 machine was available for use only at the end of the benchmarking phase and hence due to time constraint, the scalability tests of *master-slave replication* have been left as a future enhancement to the project.

(Ogawa, et al. 2010), implemented a key-value store based MapReduce framework to overcome some of the limitations imposed by the current implementation of MapReduce. This new implementation is particularly aimed at improving the performance of HPC applications intended to use the MapReduce framework. One of the limitations that this new implementation aims to address is associated with the sharing of data between *map* and *reduce* tasks during execution of MapReduce jobs. Dynamic applications whose input data stream changes in real-time can also benefit from such implementation. As a future enhancement to this work, it is advisable to explore alternate implementations to ascertain the ideal programming abstraction suitable for dynamic, distributed and data-intensive computing.

Bibliography

Apache Hadoop framework. 2008. http://hadoop.apache.org/ (accessed June 27, 2011).

Armbrust, Michael, et al. *Above the Clouds: A Berkeley View of Cloud Computing.* Berkeley, U.S.A.: EECS Department, University of California, Berkeley, 2009.

Barroso, Luiz André, and Urs Hölzle. *The Datacenteras a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan and Claypool Publishers, 2009.

Butt, Ali R., Prashant Pandey, Karan Gupta, and Gunaying Wang. "A Simulation Approach to Evaluating Design Decisions in MapReduce Steps." *IEEE International Symposium on Modeling Analysis Simulation of Computer and Telecommunication Systems.* 2009: IEEE, 2009. 1-11.

Cattell, Rick. "Scalable SQL and NoSQL Data Stores." *Rick Cattell Home Page.* 12 June 2011. http://www.cattell.net/datastores/Datastores.pdf (accessed August 06, 2011).

Dean, Jeffrey, and Sanjay Ghemawat. "Mapreduce: Simplified Data Processing on Large Clusters." *OSDI'04.* 2004. 137-150.

Ekanayake, Jaliya, Shrideep Pallickara, and Geoffrey Fox. "MapReduce for Data Intensive Scientific Analyses." *IEEE Fourth International Conference on eScience.* IEEE, 2008. 277-284.

Gufler, Benjamin, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. "Handling Data Skew in MapReduce." *CLOSER 2011 - International Conference on Cloud Computing and Services Science.* 2011.

Hansen, J., R. Ruedy, J. Glascoe, and M. Sato. "GISS analysis of surface temperature change." *J. Geophys. Res.,104*, 1999: 30,997-31,022.

Hansen, J., R. Ruedy, M. Sato, and K. Lo. "Global Surface Temperature Change." *J. Geophys. Res, 48*, 2010: 1-29.

Hansen, James, and Sergej Lebedeff. "Global Trends of Measured Surface Air Temperature." *J. Geophys. Res., 92*, 1987: 13,345-13,372.

Hey, Tony, Stewart Tansley, and Kristin Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Redmond, Washington: Microsoft Research, 2009.

Imhoff, M. L., W. T. Lawrence, D. C. Stutzer, and C. D. Elvidge. "A technique for using composite DMSP-OLS "city-lights" satellite data to map urban area." *Remote Sens. Environ.,61*, 1997: 361-370.

Isard, Michael, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. “Dryad: distributed data-parallel programsfrom sequential building blocks.” *ACM SIGOPS Operating Systems Review.* 2007. Volume: 41, Issue: 3, Pages: 59.

Jha, Shantenu, Murray Cole, Daniel S Katz, Manish Parashar, Omer Rana, and John Weissman. “Abstractions for Large-Scale Distributed Applications and Systems.” *ACM Computing Surveys*, 2009.

Jiang, Dawei, B.C Ooi, L Shi, and S Wu. “The Performance of MapReduce : An In-depth Study.” *Proceedings of the VLDB Endowment .* VLDB Endowment, 2010. 472-483, Vol.: 3, Issue:1.

Kwon, YongChul, Magdalena Balazinska, and Bill Howe. “A Study of Skew in MapReduce Applications.” *Open Cirrus Summit 2011.* Russia, 2011.

Lin, Jimmy, and Chris Dyer. “Data Intensive Text Processing with MapReduce.” *Proceedings of Human Language Technologies, The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics .* Association for Computational Linguistics, 2009. 1-2.

Mackey, Grant, Saba Sehrish, John Bent, Julio Lopez, Salman Habib, and Jun Wang. “Introducing map-reduce to high end computing.” *3rd Petascale Data Storage Workshop.* IEEE, 2008. 1-6.

Malewicz, Grzegorz, et al. “Pregel : A System for Large-Scale Graph Processing.” *28th ACM Symposium on Principles of Distributed Computing (PODC 2009).* Calgary, Alberta, Canada: ACM, 2009. Volume: 9, Pages: 6-6.

Moretti, Christopher, Hoang Bui, Karen Hollingsworth, Brandon Rich, Patrick Flynn, and Douglas Thain. “All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids.” *IEEE Transactions on Parallel and Distributed Systems.* IEEE, 2010. Volume: 21, Issue: 1, Pages: 33-46.

Ogawa, Hirotaka, Hidemoto Nakada, Ryousei Takano, and Tomohiro Kudoh. “An Implementation of Key-value Store based MapReduce Framework.” *2010 IEEE Second International Conference on Cloud Computing Technology and Science.* IEEE, 2010. 754-761.

Parker, D. E. “Urban heat island effects on estimates of observed climate change.” *Wiley Interdiscip. Rev. Clim. Change, 1*, 2010: 123-133.

Pavlo, Andrew, et al. “A comparison of approaches to large-scale data analysis.” *ACM SIGMOD, Volume: 12, Issue: 2.* 2009. 165-178.

Peterson, T.C., T.R. Karl, P.F. Jamason, R. Knight, and D.R. Easterling. "First difference method: Maximizing station density for the calculation of long-term global temperature change." *J. Geophys. Res.,103*, 1998: 25,967-25,974.

Reynolds, R. W., N. A. Rayner, T. M. Smith, D. C. Stokes, and W. Wang. "An improved in situ and satellite SST analysis for climate." *J. Clim., 15*, 2002: 1609-1625.

Vaquero, Luis. M, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. "A break in the Clouds: Towards a Cloud Definition." *ACM SIGCOMM Computer Communication Review, Vol. 39, No.1*, 2009: 50-55.

White, Tom. *Hadoop: The Definitive Guide, Second Edition.* O'Reilly, 2010.

Xie, Jiong, Shu Yin, Xiaojun Ruan, Zhiyang Ding, and Yun Tian. "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters." *2010 IEEE International Symposium on Parallel Distributed Processing Workshops.* IEEE, 2010. 1-9.

Zaharia, Matei, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. "Improving MapReduce Performance in Heterogeneous Environments." *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008.* San Diego, 2008. 29-42.

Zhu, Shengkai, Zhiwei Xiao, Haibo Chen, Rong Chen, Weihua Zhang, and Binyu Zang. "Evaluating SPLASH-2 Applications Using MapReduce." *APPT09.* 2009. 452-464.

Appendix A - Implementation of the weather data mining example

The *mrjob MaxTemperature* class, containing the map and reduce functions, is called from *max_temperature_wrapper.py*. It is not always necessary to call the Map/Reduce job from another external script, but here it is implemented intentionally in that fashion in order to test the behaviour of code when called from an external script.

This test proved useful when porting the original *cccgistemp* code, where the *mrjob* could only be invoked from an external script due to architectural dependence. The original code is written as a series of five steps, each being called from an external script (tool/run.py). The aforementioned approach was followed to remain consistent with the implementation of the original code. Additionally, this improves readability and maintainability of the code and does not pose any performance issues.

max_temperature.py contains the definitions of the *map* and *reduce* functions. The logic and behaviour is as explained in 6.1.1.

max_temperature_wrapper.py

```
#!/usr/bin/python

import sys

#http://packages.python.org/mrjob/
from mrjob.job import MRJob

from max_temperature import MaxTemperature

def main():

  # Input from a specified location
  argsArray = ['2008_NOAA.dat']

  # Provide input arguments to MRJob (e.g. '-r hadoop')
  argsArray.extend(sys.argv[1:])

  mr_job = MaxTemperature(args=argsArray)

  with mr_job.make_runner() as runner:
      # Call MapReduce task
      runner.run()
      for line in runner.stream_output():
          # Extract key/value pairs from reducer output and print
          key, value = mr_job.parse_output_line(line)
          print key, value

if __name__ == '__main__':
          sys.exit(main())
```

max_temperature.py

```
#!/usr/bin/python

#Implementing Max_temperature based on NCDC data (year='2008'), using MRJob

import sys

#http://packages.python.org/mrjob/
from mrjob.job import MRJob

class MaxTemperature(MRJob):

      DEFAULT_PROTOCOL = 'json'

      # Mapper function
      def max_temperature_map(self, _, line):
         token = line.strip()
         (month, temp) = (token[18:20], token[26:30])
         if (temp != "9999.9"):
            yield month, temp

      # Reducer function
      def max_temperature_reduce(self, month, temp):
         yield month, max(temp)

      # Map/Reduce job steps
      def steps(self):
         return [
            self.mr(
               mapper=self.max_temperature_map,
               reducer=self.max_temperature_reduce
            )
         ]

      def __init__(self, **kwargs):
            super(MaxTemperature, self).__init__(**kwargs)


if __name__ == '__main__':
   MaxTemperature.run()
```

Appendix B - Original Risk Assessment

Table 2: Risks identified initially during the project preparation phase.

Rank	Risk	Likelihood	Impact	Risk mitigation
1	Aggressive and ambitious project plan	Certain	High	Reduce by re-planning and planning early.
2	Chosen application not suitable for Dynamic, Distributed, Data-intensive (3D) computing.	Highly Probable	Moderate	Avoid by thoroughly evaluating the chosen application.
3	Application domain unsuitable	Probable	High	Reduce by evaluating other suitable domains as options.
4	Incorrect choice of programming abstraction framework.	Probable	Moderate	Avoid by choosing applications suitable for MapReduce.
5	Incorrect software development paradigm	Unlikely	High	Reduce the number of iterations if agile techniques fail.
6	Design deviation	Probable	Moderate	Avoid by evaluating thoroughly before making a decision.
7	Poor schedule	Probable	Moderate	Reduce by re-adjusting time (Agile).
8	Nothing working	Unlikely	Critical	Reduce by planning early.
9	Data loss	Unlikely	Critical	Avoid by taking regular backups of assets.
10	Unable to find open-source software	Unlikely	Severe	Reduce by evaluating early in project and change application domain to where open-source software is available.
11	Programming model did not improve throughput of the application on the cloud.	Highly Probable	High	Reduce by thoroughly analysing application model and code structure.
12	Available benchmarking unsuitable for 3D computing	Probable	High	Assume. Benchmarking models for clouds are relatively new and experimental.
13	Backtracking if dead-ends.	Probable	Moderate	Reduce by planning in advance and reviewing progress.
14	Absence of basic infrastructure on cloud platforms to run applications	Unlikely	High	Avoid by allocating sufficient time during term break to get the infrastructure up and running.
15	Porting application code to cloud platform	Highly Probable	High	Reduce by choosing right application domain and well written code.

Figure 33 shows the risk assessment diagram for the above mentioned risks.

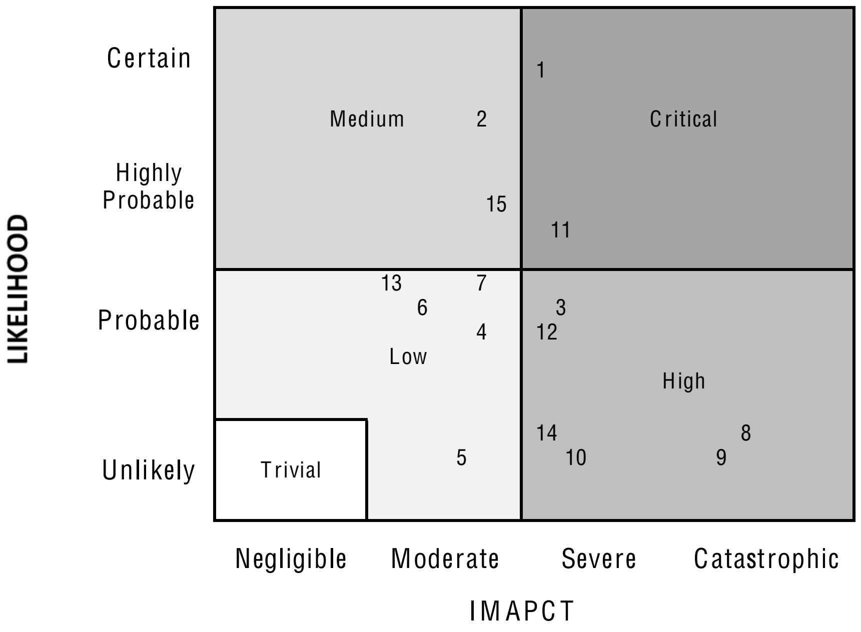
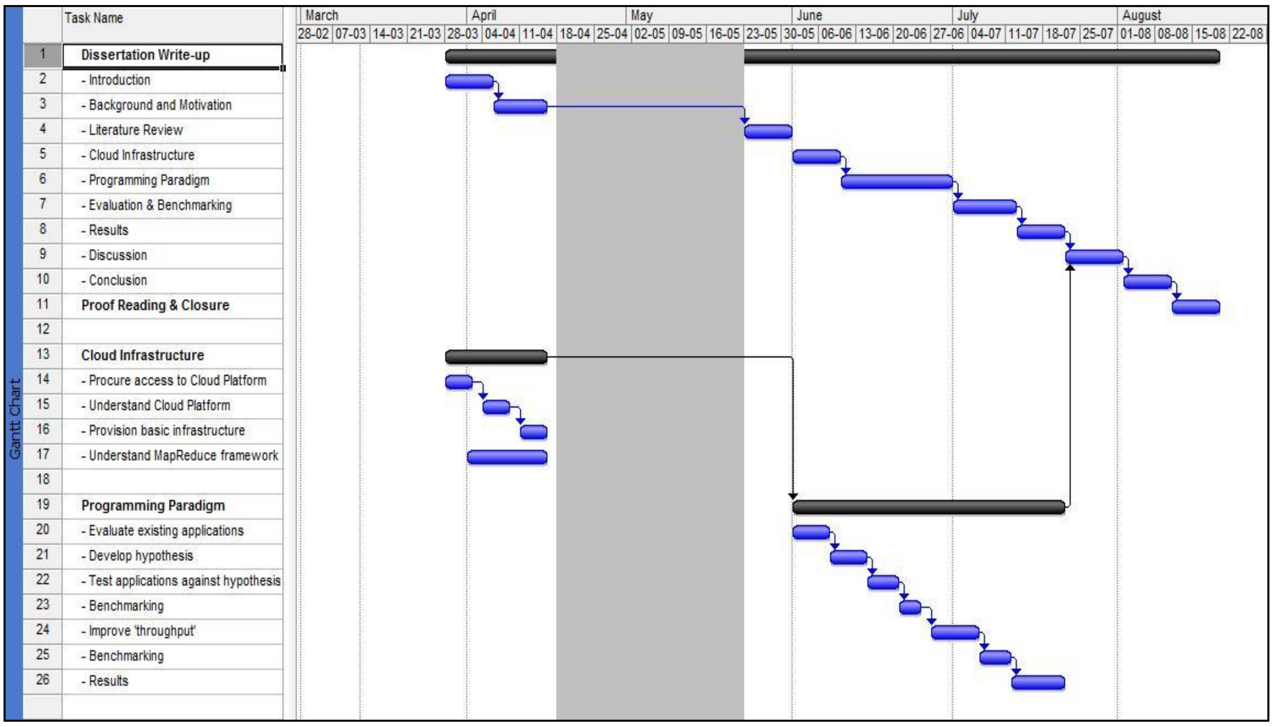


Figure 33: Original risk assessment diagram indicating the significance attributed to each of the identified risks.

Appendix C - Original Work Plan

Figure 34 shows the original work plan identified during the project preparation phase.



	Task Name
1	**Dissertation Write-up**
2	- Introduction
3	- Background and Motivation
4	- Literature Review
5	- Cloud Infrastructure
6	- Programming Paradigm
7	- Evaluation & Benchmarking
8	- Results
9	- Discussion
10	- Conclusion
11	**Proof Reading & Closure**
12	
13	**Cloud Infrastructure**
14	- Procure access to Cloud Platform
15	- Understand Cloud Platform
16	- Provision basic infrastructure
17	- Understand MapReduce framework
18	
19	**Programming Paradigm**
20	- Evaluate existing applications
21	- Develop hypothesis
22	- Test applications against hypothesis
23	- Benchmarking
24	- Improve 'throughput'
25	- Benchmarking
26	- Results

Figure 34: Original project work plan identifying the major milestones along with sub-tasks that are required to be completed for each Milestone

Appendix D - Hadoop in Pseudo-distributed mode on St. Andrews cloud infrastructure (StACC)

The following steps enlist the procedure to setup Hadoop in Pseudo-distributed mode on the St. Andrews cloud infrastructure. It must be noted that most steps are independent of the cloud service provider and can be executed on any machine with similar architecture.

1. Create a user account at https://cloud.cs.st-andrews.ac.uk:8443 to obtain the login credentials.

2. Setup the Hybridfox Firefox plugin to access the StACC cloud. The procedure for the same is provided at: http://stacc.trac.cs.st-andrews.ac.uk/wiki/HybridfoxStacc
   [Note: Instance of any operating system can be selected from the Images tab. However, in this project Ubuntu-10.10 was selected and hence the commands in the following steps are specific to Ubuntu. If any other distribution is selected, please change the commands accordingly.]

3. Once logged into the running instance, create a username with the following command:
   useradd –m –s /bin/bash <username>
   Change password with the command: passwd <command>

4. Add the username to the sudoers list with the command:
   su –
   echo '<username> ALL=(ALL) ALL' >> /etc/sudoers

5. The creation of userid is not mandatory. However, it is recommended to prevent accidental damage to system files while code and testing the application.

6. Java^{TM} is required for Hadoop setup. The latest version can be downloaded and installed from: http://www.oracle.com/technetwork/java/javase/downloads/index.html

7. The detailed procedure for downloading and configuring Hadoop in pseudo-distributed mode can be found here:
   http://hadoop.apache.org/common/docs/current/single_node_setup.html
   Some of the issues identified during installation can be solved by following the procedure described in http://wiki.apache.org/hadoop/HowToSetupYourDevelopmentEnvironment

8. Most releases of Ubuntu have Python preinstalled. Check for the correct version of Python by executing 'python' at the command prompt. If the required version (2.6+) is not found, then download and install the latest release of python from:
   http://www.python.org/getit/

9. Latest releases of *mrjob* can be found at: http://pypi.python.org/pypi/mrjob/.
   Download and install *mrjob*. Version 0.2.6 was used in this project.

10. Download and install *redis* by following the procedure given in http://redis.io/download
    Latest releases of redis python client can be found at: http://pypi.python.org/pypi/redis/
    Version 2.4.9 was used in this project.

11. After downloading the redis python client, required redis packages found in <REDIS_HOME>/redis must be archived to be used by the Hadoop streaming job. The path for this archive must be specified correctly in *.mrjob* configuration file.

12. Start Redis server by executing ./redis-server from REDIS_HOME/src directory.

13. Once all the required software packages are installed, download the latest version of *mapreduce-cccgistemp* and follow the execution instructions provided in the release notes.
    http://code.google.com/p/mapreduce-cccgistemp/downloads/list

Appendix E – Timing Results

Step1 Analysis

Local MapReduce Execution (1 Map, 1 Reduce Task): 1727.3

Number of Cores	Stage 1			Stage 2			Total MapReduce Job Time	Total IO Time	Overall Execution Time	Speedup	Efficiency
	Map	Reduce	MapReduce Job Time	Map	Reduce	MapReduce Job Time					
2	39	157	207	67	471	531	738	394.5	1132.5	1.53	76%
4	24	90	125	58	244	312	437	409.2	846.2	2.04	51%
8	16	56	83	32	137	179	262	386.6	648.6	2.66	33%
16	18	47	76	24	89	123	199	404.7	603.7	2.86	18%
20	18	43	69	21	78	108	177	405.9	582.9	2.96	15%
24	17	41	68	19	69	98	166	403.6	569.6	3.03	13%
28	16	39	65	18	62	91	156	396.6	552.6	3.13	11%

Table 3: Timing results of step1 analysis with dataset=100%

Local MapReduce Execution (1 Map, 1 Reduce Task): 1254.9

Number of Cores	Stage 1			Stage 2			Total MapReduce Job Time	Total IO Time	Overall Execution Time	Speedup	Efficiency
	Map	Reduce	MapReduce Job Time	Map	Reduce	MapReduce Job Time					
2	32	120	162	67	374	405	567	312.7	879.7	1.43	71%
4	20	71	102	44	185	237	339	310.6	649.6	1.93	48%
8	18	46	71	25	106	140	211	301.2	512.2	2.45	31%
16	18	43	69	24	70	92	161	306.6	467.6	2.68	17%
20	16	39	65	19	64	91	156	307.1	463.1	2.71	14%
24	16	38	64	18	58	86	150	307.1	457.1	2.75	11%
28	15	37	62	17	54	79	141	310.5	451.5	2.78	10%

Table 4: Timing results of step1 analysis with dataset=75%

Local MapReduce Execution (1 Map, 1 Reduce Task): 927.7

Number of Cores	Stage 1			Stage 2			Total MapReduce Job Time	Total IO Time	Overall Execution Time	Speedup	Efficiency
	Map	Reduce	MapReduce Job Time	Map	Reduce	MapReduce Job Time					
2	23	93	127	64	276	349	476	220.6	696.6	1.33	67%
4	16	56	83	34	146	190	273	220.5	493.5	1.88	47%
8	14	38	60	23	86	117	177	219.6	396.6	2.34	29%
16	15	36	61	21	63	84	145	216.4	361.4	2.57	16%
20	16	36	61	16	55	79	140	221.2	361.2	2.57	13%
24	15	36	60	20	53	74	134	221.2	355.2	2.61	11%
28	15	34	59	16	47	72	131	222.2	353.2	2.63	9%

Table 5: Timing results of step1 analysis with dataset=50%

Local MapReduce Execution (1 Map, 1 Reduce Task): 492.4

Number of Cores	Stage 1			Stage 2			Total MapReduce Job Time	Total IO Time	Overall Execution Time	Speedup	Efficiency
	Map	Reduce	MapReduce Job Time	Map	Reduce	MapReduce Job Time					
2	16	58	85	37	155	204	289	137.3	426.3	1.16	58%
4	12	40	62	23	88	118	180	137.5	317.5	1.55	39%
8	11	30	50	16	54	77	127	138.4	265.4	1.86	23%
16	14	30	56	16	43	65	121	138.5	259.5	1.90	12%
20	14	32	54	16	41	64	118	133.2	251.2	1.96	10%
24	13	32	54	16	41	61	115	139.4	254.4	1.94	8%
28	14	30	53	13	38	60	113	139.6	252.6	1.95	7%

Table 6: Timing results of step1 analysis with dataset=25%

Step2 Analysis

Local MapReduce Execution (1 Map, 1 Reduce Task): 1373.8

Number of Cores	Stage 1			Stage 2			Total MapReduce Job Time	Overall Execution Time	Speedup	Efficiency
	Map	Reduce	MapReduce Job Time	Map	Reduce	MapReduce Job Time				
2	46	201	243	58	455	524	767	1124.0	1.22	61%
4	24	110	141	33	317	361	502	839.1	1.64	41%
8	16	65	90	21	303	333	423	779.0	1.76	22%
16	15	58	85	18	297	324	409	759.5	1.81	11%
20	15	32	76	15	220	267	343	696.8	1.97	10%
24	14	28	73	18	206	243	316	659.4	2.08	9%
28	13	26	70	13	218	263	333	681.7	2.02	7%

Table 7: Timing results of step2 analysis with Dataset=100% (Original Code)

Local MapReduce Execution (1 Map, 1 Reduce Task): 1413.3

Number of Cores	Stage 1			Stage 2			Total MapReduce Job Time	Overall Execution Time	Speedup	Efficiency
	Map	Reduce	MapReduce Job Time	Map	Reduce	MapReduce Job Time				
2	31	188	230	58	380	449	679	1015	1.39	70%
4	20	112	142	34	204	142	284	723.1	1.95	49%
8	15	66	90	21	138	169	259	599.1	2.36	29%
16	18	58	84	18	138	165	249	586.2	2.41	15%
20	16	50	76	16	114	140	216	560.8	2.52	13%
24	15	48	73	15	100	125	198	549.4	2.57	11%
28	15	46	70	14	118	143	213	566.4	2.50	9%

Table 8: Timing results of step2 analysis with Dataset=100% (Optimised Code)

Local MapReduce Execution (1 Map, 1 Reduce Task): 544.4

Number of Cores	Stage 1			Stage 2			Total MapReduce Job Time	Overall Execution Time	Speedup	Efficiency
	Map	Reduce	MapReduce Job Time	Map	Reduce	MapReduce Job Time				
2	19	44	132	32	144	186	318	498.9	1.09	55%
4	14	62	86	20	103	132	218	402.2	1.35	34%
8	12	41	61	15	63	85	146	325.5	1.67	21%
16	14	40	65	15	72	92	157	339.1	1.61	10%
20	14	39	63	14	65	87	150	334.9	1.63	8%
24	14	37	60	12	56	78	138	321	1.70	7%
28	14	36	59	19	61	83	142	321.7	1.69	6%

Table 9: Timing results of step2 analysis with Dataset=50% (Optimised Code)

Local MapReduce Execution (1 Map, 1 Reduce Task): 2894.3

Number of Cores	Stage 1 MapReduce Job Time	Stage 2 MapReduce Job Time	Stage 3 MapReduce Job Time	Stage 4 MapReduce Job Time	Total MapReduce Job Time	Total IO Time	Overall Execution Time	Speed up	Efficiency	Original Step1+ Step2 Execution Time
2	204	528	226	527	1485	379.4	1864.4	1.55	78%	2147.5
4	124	310	145	284	863	361.5	1224.5	2.36	59%	1569.3
8	81	175	90	168	514	373.0	887	3.26	41%	1247.7
16	73	117	79	166	435	354.3	789.3	3.67	23%	1189.9
20	70	111	72	139	392	362.4	754.4	3.84	19%	1143.7
24	68	98	69	130	365	376.0	741	3.91	16%	1119.0
28	66	89	66	142	363	373.6	736.6	3.93	14%	1119.0

Table 10: Timing results of combined step1 and step2 execution with dataset=100%

Step3 Analysis

Local MapReduce Execution (1 Map, 1 Reduce Task): 5842.30

Number of Cores	MapReduce Job Time	Sorting and Writing Output	Overall Execution Time	Speedup	Efficiency
2	2889	427.9	3419.4	1.71	85%
4	2149	433.1	2686.2	2.17	54%
8	1830	432.1	2367.5	2.47	31%
16	1370	433.4	1905.9	3.07	19%
24	1416	340.5	1860.2	3.14	13%
28	1340	431.0	1887.6	3.10	11%

Table 11: Timing results of step3 analysis with Dataset=100% and Key=Eastern Longitude

Local MapReduce Execution (1 Map, 1 Reduce Task): 5787.90

Number of Cores	MapReduce Job Time	Sorting and Writing Output	Overall Execution Time	Speedup	Efficiency
2	3502	433.2	4039.9	1.43	72%
4	2878	431.6	3413.2	1.70	42%
8	1946	432.9	2480.7	2.33	29%
16	1196	437.7	1736.5	3.33	21%
24	1074	435.4	1613.1	3.59	15%
28	1662	433.4	2198.7	2.63	9%

Table 12: Timing results of step3 analysis with Dataset=100% and Key=Western Longitude

Local MapReduce Execution (1 Map, 1 Reduce Task): 2682.50

Number of Cores	MapReduce Job Time	Sorting and Writing Output	Overall Execution Time	Speedup	Efficiency
2	1189	344.5	1636.4	1.64	82%
4	863	348.4	1312.5	2.04	51%
8	693	344.3	1140.2	2.35	29%
16	489	350.0	941.7	2.85	18%
24	546	344.5	993	2.70	11%
28	605	345.7	1052.8	2.55	9%

Table 13: Timing results of step3 analysis with Dataset=50% and Key=Eastern Longitude.

Local MapReduce Execution (1 Map, 1 Reduce Task): 2674.20

Number of Cores	MapReduce Job Time	Sorting and Writing Output	Overall Execution Time	Speedup	Efficiency
2	1383	343.5	1826.7	1.46	73%
4	1037	348.2	1486.6	1.80	45%
8	730	347.2	1178.2	2.27	28%
16	525	347.6	975.5	2.74	17%
24	417	344.7	864.5	3.09	13%
28	612	342.1	1055.8	2.53	9%

Table 14: Timing results of step3 analysis with Dataset=50% and Key=Western Longitude.

Appendix F – Code overview of the ported mapreduce-cccgistemp

mapreduce-cccgistemp has the following directory structure[65]:

/code/	Source code for the GISTEMP algorithm only
/config/	Configuration files
/doc/	Internal ccc-gistemp developer documentation
/input/	Input data files
/log/	Log files (Used only by non-MapReduce steps)
/tool/	Tools - sources other than the GISTEMP algorithm
/work/	Intermediate data files
/result/	Final result files

ccc-gistemp & mapreduce-cccgistemp uses input data in the subdirectory input/ which includes files of temperature records from GHCN, USHCN and sea surface data, small files of additional temperature records and station tables from GISS. The code /tool/preflight.py is used to fetch this data over the internet.

Steps 1, 2 and 3 of the original ccc-gistemp are ported to MapReduce. Primary code changes for the porting exercise impact the following files:

/code/step1.py
/code/step2.py
/code/step3.py
/code/read_config.py
/tool/run.py
/tool/giss_io.py

/code/Mapreduce.py was added as part of this project for verifying the impacts of I/O by combining steps 1 and 2.

All the GISS MapReduce class definitions begin with GissAnalysisStep<#>, where '#' refers to the ported ccc-gistemp step and contains the definitions of *map* and *reduce* functions. The required GISS algorithms are invoked from within the map/reduce function. Step2 however has modifications within the GISS algorithm to incorporate the use of Redis key/value store. The records are added to the store in function annotate_records and retrieved for use in urban_adjustments. Both the definitions are in /code/step2.py.

The function definitions that initially read input files for steps 1, 2 and 3 are now commented out in /tool/giss_io.py as these files are now fed directly to the *map* function by specifying their path while creating the *mrjob* class. *V2MapReduceMeanReader*, the modified 'MeanReader' that invokes creation of "Series" object is defined in /tool/giss_io.py.

[65] Identical to the original ccc-gistemp code structure

/tool/run.py, the code that initiates the execution, has significant changes to incorporate the calling of *mrjob* class for the corresponding MapReduce step(s). The definitions however follow a similar pattern. First, the input file is specified in the arguments array followed by all the supporting GISS configuration files. Secondly, the Hadoop MapReduce job configuration parameters are specified. Next, *mrjob* is invoked with the following statement:

```
with mr_job.make_runner() as runner:
  runner.run()
```

Finally, the output of MapReduce job is retrieved and written to the specified output file.

The following command can be executed to verify the complete execution on Hadoop cluster:

```
./run.sh -r hadoop -mt <No. of map tasks> -rt <No. of reduce tasks> -bin <Python_binary>
```

To execute only a specified step on the Hadoop cluster:

```
./run.sh -s 1 -r hadoop -mt <No. of map tasks> -rt <No. of reduce tasks> -bin <Python_binary>
```

This command executes step1, assuming that step0 has already been run and the output is available for use by this step.

After executing all the six steps (steps 0 to 5), the resulting GISTEMP outputs are all in the /result/ directory. A simple graphical chart[66] created using the Google Chart API, showing the global mean surface temperature anomaly is available for verification at /result/google-chart.url

[66] Chart created as part of the original ccc-gistemp code.

Appendix G – Brief review of the rounding error observed in Step2 verification

Excerpt from the output of '*diff*' showing variation in the temperature anomaly of last month (December) for certain years. This discrepancy then creeps into the subsequent years up until November. Similar trend is observed for all stations that exhibit this discrepancy.

```
< 6510324200101947  27  -8   21  79   120  146  164  175  143  108  61  50
< 6510324200101948  41  45  77  85   99   124  151  139  130  97   66  46
< 6510324200101949  47  53  50  100  107  139  164  161  155  111  63  53
---
> 6510324200101947  27  -8   21  79   120  146  164  175  143  108  61  51
> 6510324200101948  42  46  78  86   100  125  152  140  131  98   67  47
> 6510324200101949  48  54  51  101  108  140  165  162  156  112  64  53
391287,391288c391298,391299
< 6510324200101960  34  30  51  86   114  151  143  142  126  98   59  35
< 6510324200101961  29  60  82  85   99   136  140  145  141  98   54  12
---
> 6510324200101960  34  30  51  86   114  151  143  142  126  98   59  36
> 6510324200101961  30  61  83  86   100  137  141  146  142  99   55  12
```

Analogous rounding error was also observed by developers of ccc-gistemp when porting the original GISS Gistemp code to Python and they have discussed its behaviour at the ccc-gistemp discussion groups[67].

This issue is obvious when run on multiple cores and was observed during the benchmarking phase of the project. Investigations into this issue revealed that the error was most likely creeping in the function annual_anomaly, which is called from annotate_record. Here the month December of final year is neglected followed by taking the average of the monthly means of global temperature series. The seasonal anomalies and annual anomalies are then computed from the monthly means. An error induced in the monthly mean computation is most likely to affect results of seasonal and annual anomalies and hence the observed cumulative error across years.

As this issue was observed during the benchmarking phase when EDIM1machine was finally available for testing, very less time was available to fix this bug and run sufficient test cases to ensure its correctness and impacts on other areas of the code. Hence, it was thought of as best to report this issue and not make code fixes at the final stages of the project. The issue has also been reported to ccc-gistemp developers, as this has been prevailing in the base code that was used for porting.

[67]http://groups.google.com/group/ccc-gistemp-discuss/browse_thread/thread/3386c3c814584e0/9bf7dcbba12b4ebb?lnk=gst&q=annual_anomaly#9bf7dcbba12b4ebb