Lossless compression of SKA data sets

Karthik Bharadhwaj Rajeswaran
Supervisor: Dr. Simon Winberg

A dissertation submitted to the Department of Electrical Engineering,
University of Cape Town, in fulfilment of the requirements
for the degree of Master of Science in Engineering.

Cape Town, August 2013

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author .....

Cape Town
30 August 2013

Abstract

With the size of astronomical data archives continuing to increase at an enormous rate, the providers and end users of astronomical data sets will benefit from effective data compression techniques. This dissertation explores different data compression techniques and aims to find an optimal compression algorithm to compress astronomical data obtained by the SKA, which are new and unique in the field of radio astronomy. It was required that the compressed data sets should be lossless and that they should be compressed while the data is being accessed. The project was carried out in conjunction with the SKA South Africa office.

Data compression reduces the time taken and the bandwidth used when transferring files, and can also reduce the costs involved with data storage. This is especially applicable when radio telescopes are located a long distance away from the centres where the astronomical data is processed and analysed, which is the case with the SKA project.

The SKA use the Hierarchical Data Format (HDF5) to store the data collected from the radio telescopes, with the data used in this study ranging from 29MB to 9GB in size. The compression techniques investigated in this study include SZIP, GZIP, the LZF filter, LZ4 and the Fully Adaptive Prediction Error Coder (FAPEC).

It was found that the LZ4 and PEC provided the best compression ratios and were the most time and memory efficent algorithms. A program was developed using the LZ4 algorithm which was used to compress the data sets while they were being accessed from another machine, thus simulating the environment which is

used by the SKA.

The dissertation concludes that the PEC and LZ4 are the optimal compression algorithms for the SKA data sets at this point in time, and presents suggestions for future work and discusses improvements that could be made.

Keywords: Lossless Compression, Astronomical data, Square Kilometre Array, Hierarchical Data Format, Streaming compression

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Simon Winberg who kindly agreed to supervise this project, for his invaluable advice, useful suggestions and for providing a firm support structure that I could rely upon whenever I encountered difficulties.

I would also like to thank my parents for their unwavering support, encouragement and belief in me. They motivated me when I needed it most and I owe everything that I have achieved thus far to them.

I owe thanks to Dr. Jordi Portell, Marcial Clotet and Carlos Sanchez from the Polytechnic University of Catalonia who assisted greatly with technical concepts that were new to me, and who were kind enough to aid me with elements of the testing process. A mention to the engineers at the SKA who helped me shape and develop the project and who provided feedback and information when I needed it.

A final thank you to Sandra, all of my close friends and the members of the RRSG lab who have made the last two years go by extremely quickly, and made the journey a far less arduous one.

Contents

List of Figures

List of Tables

List of Abbreviations

SKA—Square Kilometre Array

FITS—Flexible Image Transport System

HDF—Hierarchical Data Format

NASA—National Aeronautics and Space Administration

PEC—Prediction Error Coder

FAPEC—Fully Adaptive Prediction Error Coder

ESA—European Space Agency

CCSDS—Consultative Committee for Space Data Systems

MAS— Modies Airborne Simulation

JPL— Jet Propulsion Laboratory

LZW— Lambel-Ziv-Welch

LHC— Large Hadron Collider

NCSA — National Centre for Supercomputing Applicaionts

LSST — Large Synoptic Survey Telescope

API — Application Programming Interface

RAM — Random Access Memory

CPU — Central Processing Unit

OS — Operating System

KB — Kilobyte

MB — Megabyte

GB — Gigabyte

PB — Petabyte

I/O — Input and Output

Chapter 1

Introduction

This dissertation involves the development of a stream-based lossless compression system for use on astronomical data sets collected by the Square Kilometre Array (SKA) project. The system aims to provide a means to reduce data storage loads and enhance the speed with which users can access the data sets. In achieving this objective, a set of existing lossless compression algorithms were investigated. This was followed by the development of a stream-based compression module, with the best performing algorithm serving as the basis for the module.

This introduction will provide a background to the project, state the objectives of the study and present an overview of the dissertation.

1.1 Introduction to Astronomical Data Compression

Astronomical data refers to data that is collected and used in astronomy and other related scientific endeavours. In radio astronomy, data that is collected from radio telescopes and satellites is stored and analysed by astronomers, astrophysicists and scientists. Digital astronomical data sets have traditionally been stored in the Flexible Image Transport System (FITS) file format, and are very large in size. More recently, the Hierarchical Data Format (HDF5) has been

adopted in some quarters, and is designed to store and organize large amounts of numerical data.

With the size of astronomical data archives continuing to increase at an enormous rate [1], the providers and end users of these data sets will benefit from effective data compression techniques. Data compression reduces the time taken and the bandwidth used when transferring files, and can also reduce the costs involved with data storage [2]. This is particularly applicable when radio telescopes are located a long distance away from the centres where the astronomical data is processed and analysed.

1.2 Background

The MeerKAT project, which involves a radio telescope array to be constructed in the Karoo in South Africa, is a pathfinder project for the larger Square Kilometre Array (SKA) [3]. Once the MeerKAT is complete, it will be the world's most powerful radio telescope and provide a means for carrying out investigations, both in terms of astronomical studies and engineering tests, facilitating the way toward completion of the SKA.

The SKA will allow scientists to explore new depths of the Universe, and will produce images and data that could be in the order of Petabytes (PB) of size [4].

A software environment is used to analyze and extract useful information from these large data sets, which is used by scientists and astrophysicists. Improving the performance and functionality of this software environment is one of the main focus areas of research being conducted as part of the MeerKAT project.

Previous studies have discussed the big data challenges that would be faced by large radio arrays [5] and have explored the signal processing [1] [6] and data compression techniques [7] that are used in analyzing astronomical data.

1.3 Objectives

This dissertation aims to explore different lossless data compression techniques and determine the best performing option for the large datasets collected by the SKA. It then aims to explore the feasibility of stream-based compression using the selected technique, with the target being the development a lossless, stream-based compression system.

The requirements and objectives for this project stem from meetings with members of the Astronomy department at UCT [8] and with the chief software engineers working on the SKA project [9]. The meetings helped to gain a better understanding of how this project would benefit the end users (astrophysicists and scientists), and how it would reduce storage costs and allow faster access to the data sets for the SKA project.

The SKA project has a custom software environment used to process and extract information from HDF5 files. The HDF5 file format has a well-defined and adaptable structure that is becoming a popular format for storing astronomical data. These files are designed for very large data sizes.

Thus the main focus of this project is to compress the total size of the files containing the astronomical data without any loss of data, and do this while streaming the data from the source to the end user. This main objective has been divided into the following research goals:

1. Investigate and experiment with different data compression techniques and algorithms.
2. Find the optimal data compression algorithm for the given data sets.
3. Attempt to implement the algorithm while streaming the data sets from a servers.
4. Demonstrate the algorithms functionality by testing it in a similar environment to the SKA as a stand-alone program.

1.4 Overview of the research methodology

A literature based approach was used to develop the methodology of this project. This process aided in establishing and consolidating the requirements and standards to follow for the project.

The literature review consisted of reading and understanding numerous academic papers and journals on the topics of radio astronomy, astronomical data and data compression algorithms.

Previous studies, including one conducted by members of the National Aeronautics and Space Administration (NASA), tested different compression techniques and algorithms on astronomical data sets which were stored in the FITS file format, and they laid the foundation upon the which the testing and evaluation methods were developed [10].

The information gathered from this material along with the knowledge gained from conference papers and dissertations dealing with data compression [11] [12], as well as discussions with fellow students and supervisors [9] [13] , helped to shape the design phase of the project.

1.5 Proposed system design

The data that is collected from the SKA will be stored in huge data centres/servers, from which various end users will access the data. It was initially proposed that the compression should occur while the data is stored at the server end (as soon as it is collected), before the end user can access it.

This was modified at a later stage, with the compression to occur while the files were being read from the server, which is in line with the third objective of the project.

The intention is for the compression algorithm/code to be assimilated into the software stack that the SKA currently has in place. An additional functionality is that it should work as a stand-alone program.

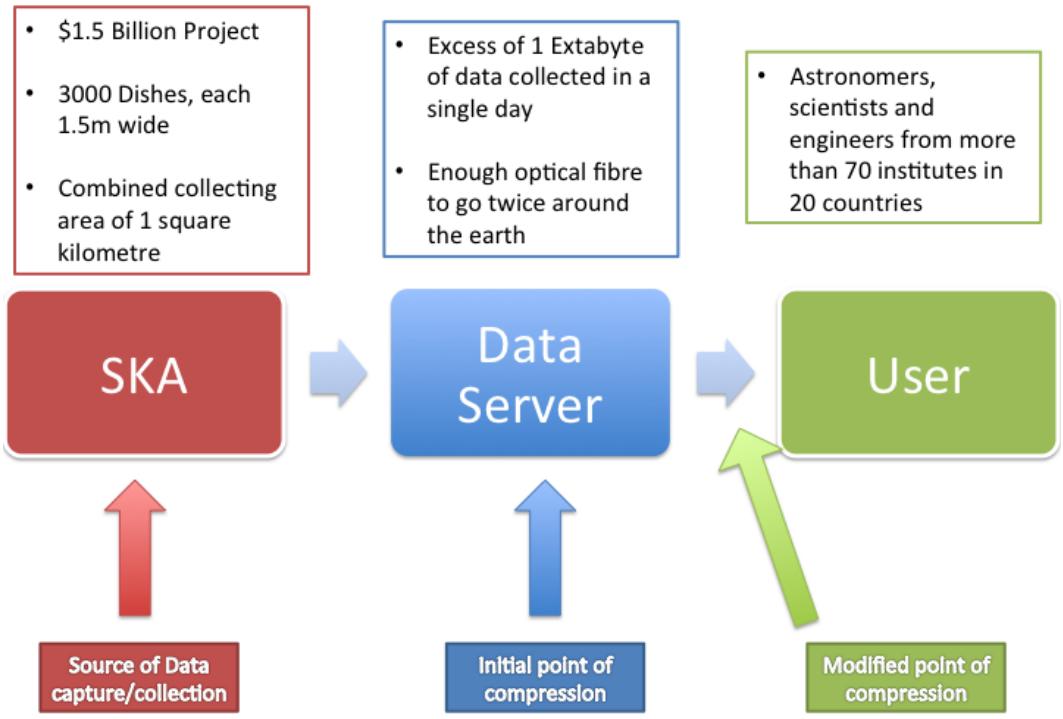

Figure 1.1: Image showing the process of data capture, storage and consumption (Adapted from [14])

From discussions with the SKA, the main priority with regards to the compression of the data is the compression ratio, with compression time and memory usage coming next. It was also mentioned all of the contents data must be preserved, including any noise, making the compression lossless. Thus two main stages of testing will be done:

1. Compressing the entire data set and attempting to obtain as the highest possible compression ratio.
2. Modifying and using different parameters within the algorithms to opitmize their performance and obtain the best results.

Various algorithms were investigated and considered, with the following 5 being used in the testing process.

1. GZIP
2. SZIP
3. LZF
4. FAPEC
5. LZ4

Testing will be carried out as follows:

- Each algorithm will be run on data sets of different sizes, across a wide range (30 MB to 9 GB).
- The compression ratio and time taken will be recorded for each test that is run.
- The results from these tests will help to determine which algorithm/technique is the best for the given astronomical data sets.
- Compression will be applied while the data sets are streamed from the server to the user, simulating the SKA environment.

1.6 Limitations and Scope

The scope of this project is to design and implement a system to investigate the performance of different compression algorithms on data sets obtained by the SKA, and to use .

Given that numerous data compression algorithms and techniques already exist, the idea was to find the one that worked best with the data sets and optimize its performance accordingly. By simulating the environment used by the SKA, the system aimed to create a testbed that would provide more accurate and relevant results. The project intended to serve as a pathfinder for further research and more rigorous investigation in the future.

One of the main challenges was the constantly evolving user requirements for this project. Since the SKA project is in its infancy, there are a variety of design changes taking place which led to modifications during the course of the project. This led to delays in developing the system design and carrying out testing.

Further difficulty was experienced with obtaining all of the data sets, as meetings with the SKA engineers were infrequent. The full set of files were only procured in April 2013. This did not allow for certain experimentation and modification of the algorithms to be carried out, and have been mentioned as avenues for further work.

1.7 Dissertation Overview

This dissertation will be organized as follows:

- Chapter 2 is a literature review of the material that was researched in order to carry out the project. It discusses the role of astronomical data in radio astronomy and describes the Hierarchical Data Format, which is the data format used by the SKA. It then looks into data compression algorithms and techniques and provides a detailed background of their functionality and usage.

- Chapter 3 presents the methodology that was used to investigate and complete this project. It lists the user requirements and the associated constraints. This is followed by a brief explanation of how the literature review was used in the methodology of the project. It ends with an overview of the system design and testing process used in this project.

- Chapter 4 describes the design process used to develop the system. The system comprises of the testbeds, datasets and the compression algorithms used.

- Chapter 5 presents and discusses the results that were obtained from the simulations using the different algorithms. This includes the results obtained from the initial testing as well as the results obtained from the second phase of testing. The results are then evaluated.

- Chapter 6 concludes the dissertation and makes recommendations for future work on data compression techniques on the SKA astronomical data sets.

Chapter 2

Background and Literature Review

This dissertation applies different compression techniques to the large astronomical data problem of the SKA. This chapter aims to present sufficient background in the fields of astronomical data and data compression, in order to aid in the understanding of their application to each other.

2.1 The Square Kilometre Array Project

The MeerKAT project, which involves a radio telescope array to be constructed in the Karoo in South Africa, is a pathfinder project for the larger Square Kilometre Array (SKA) [3]. Once the MeerKAT is complete, it will be the world's most powerful radio telescope and provide a means for carrying out investigations, both in terms of astronomical studies and engineering tests, facilitating the way toward completion of the SKA. The SKA will allow scientists to explore new depths of the Universe, and will produce images and data that could be in the order of PetaBytes (PB) of size [15].

The SKA will actually operate over a wide range of frequencies, and its size will make it 50 times more sensitive than any other radio instrument [16]. Fur-

thermore, by utilizing advanced processing technology, it will be able to survey the sky more than ten thousand times faster than ever before. With receiving stations extending out to distance of 3,000 km from a concentrated central core, it will continue radio astronomy's tradition of providing the highest resolution images in all astronomy [17].

Figure 2.1: Artist's impression of the MeerKAT array in the Karoo region [18]

The SKA will be a highly flexible instrument whose role will be to address a wide range of questions in astrophysics, fundamental physics, cosmology and particle astrophysics by probing previously unexplored parts of the distant Universe. Some of the key science projects to be undertaken by using the SKA include [17]:

- Extreme tests of general relativity
- The origins and evolution of cosmic magnetism
- Dark Matter and Dark Energy

2.2 Astronomical Data in Radio Astronomy

Astronomical data refers to data that is collected and used in astronomy and other related scientific endeavours. In radio astronomy, data that is collected from radio telescopes and satellites is stored and analysed by astronomers, astrophysicists and scientists [1]. Digital astronomical data sets have traditionally been stored in the FITS file format, and are very large in size [19]. More recently, the Hierarchical Data Format (HDF5) has been adopted in some quarters, and is designed to store and organize large amounts of numerical data.

The invention and commercialization of CCD data volumes has led to astronomical data sets growing exponentially in size [20]. The table below provides evidence of this, showing how astronomical data has grown in size from the 1970s. These results were obtained from a study carried out on the LOFAR project, which is a pathfinder to the SKA.

EPOCH	NOMINAL FILE DATA VOLUME
1970	2^{10} bytes
1980	2^{20} bytes
1990	2^{30} bytes
2000	2^{40} bytes

Figure 2.2: Increasing bit size of astronomical data [17]

Most astronomers do not want to process and analyze data and have to delete it afterwards. Variable astronomical objects show the need for astronomical data to be available indefinitely, unlike Earth observation or meteorology. The biggest problem that arises from this situation is the overwhelming quantity of data which is now collected and stored [1].

Furthermore, the storage and preservation of astronomical data is vital. The rapid obsolescence of storage devices means that great efforts will be required to ensure that all useful data is stored and archived. This adds to the necessity of using a new standard to overcome the potential break down of existing storage formats [21].

2.3 The HDF5 File Format

The Hierarchical Data Format (HDF) technology is a library and a multi-object file format specifically designed to transfer large amounts of graphical, numerical or scientific data between computers [10]. HDF is a fairly new technology and was developed by the National Centre for Supercomputing Applicaionts (NCSA), while the HDF Group currently maintains it. It addresses problems of how to manage, preserve and allow maximum performance of data which have the potential for enormous growth in size and complexity. It is developed and maintained as an open source project, making it available to users free of charge.

HDF5 (the 5th iteration of HDF) is ideally suited for storing astronomical data as it is [11] [22] :

- **Open Source**: The entire HDF5 suite is open source and distributed free of charge. It also has an active user base that provide assistance with queries.
- **Scalable**: It can store data of almost any size and type, and is suited towards complex computing environments.
- **Portable**: It runs on most commonly used operating systems, such as Windows, Mac OS and Linux.
- **Efficient**: It provides fast access to data, including parallel input and output. It can also store large amounts of data efficiently, has built-in compression and allows for people to use their own custom built compression methods.

As a result, it works as a robust, viable data framework that can handle the size, scope, diversity, distributed nature and parallel Input and Output (I/O) processing requirements of the data used by the SKA. This format also has potential use beyond the radio community, with new large scale optical telescopes such as the Large Synoptic Survey Telescope (LSST) investigating the viability of using HDF5 [17].

Furthermore, the 20 year history of HDF and its ongoing use by NASAs earth orbiting missions ensure that there will be continued support for the format. There are also a number of tools available to read and visualize HDF5 files, such as HDFView and h5py [23].

2.3.1 HDF5 Data Model

The HDF uses a model for managing and storing data. The relationship between the different models and their implementations are shown in the image below and explained on the following page.

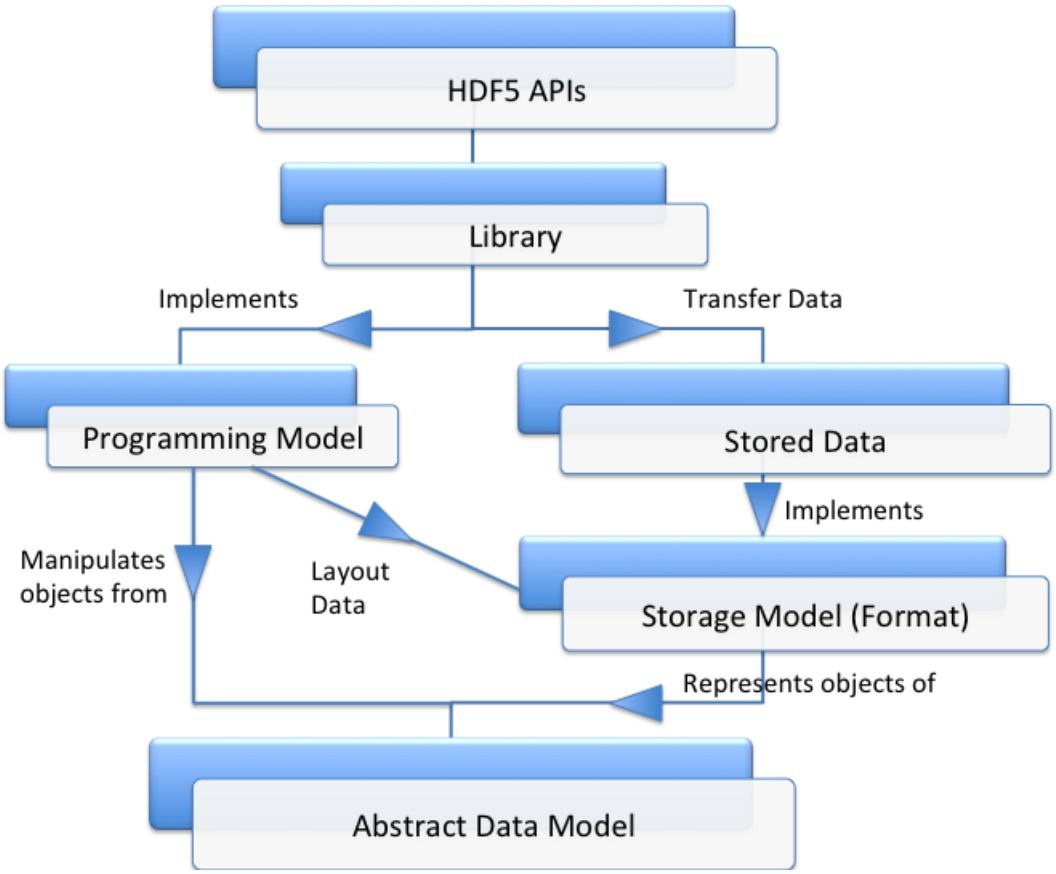

Figure 2.3: HDF5 models and implementations

As shown in the previous diagram, the HDF5 data model includes [24] :

- **An abstract data model and an abstract storage model** - The abstract data model is a conceptual model of data, data types and data organization, and is independent of storage medium or programming environment. The storage model is a standard representation for the objects of the abstract model.

- **Libraries to implement the abstract models** - This is the implementation of the programming model. The library exports the HDF5 Application Programing Interfaces (API) as its interface. In addition to implementing the objects of the abstract data model, the library manages data transfers from one stored form to another. Data transfer examples include reading from disk to memory and writing from memory to disk.

- **A programming interface** - This is a model of the computing environment and includes platforms from small single systems to large multiprocessors and clusters. The programming model manipulates (instantiates, populates, and retrieves) objects from the abstract data model.

- **A model of efficient data transfer** - This is the concrete implementation of the storage model. The storage model is mapped to several storage mechanisms including single disk files, multiple files (family of files), and memory representations.

The HDF5 Library is a C module that implements the Programming Model and Abstract Data Model. The figure below shows the dependencies of these modules [11].

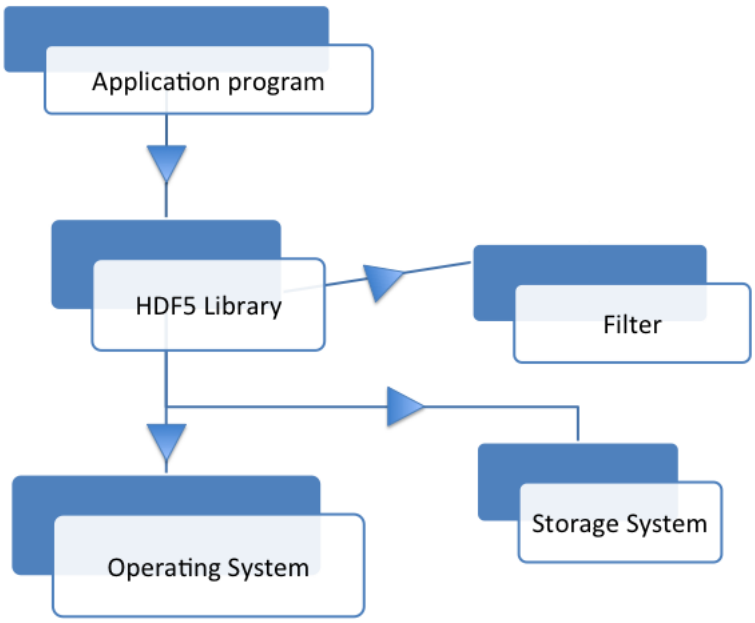

Figure 2.4: HDF5 Library Dependencies (adapted from [24])

2.3.2 The Abstract Data Model

The abstract data model helps to define and describe the data in HDF5 files, and is a very general model which aims to conceptually cover different specific models.

Only two kinds of objects are stored in HDF5 files: datasets and groups. Datasets are homogenous, regular arrays of data, while groups are containers that store datasets and other groups [10]. In essence, groups work like folders and datasets like files.

The key concepts of the Abstart Data Model include [24]:

- File - a contiguous string of bytes stored in memory
- Group - a collection of objects (including groups)
- Dataset - a multidimensional array of data elements with attributes and other metadata
- Dataspace - a description of the dimensions of a multidimensional array
- Datatype - a description of a specific class of data element including its storage layout as a pattern of bits
- Attribute - a named data value associated with a group, dataset, or named datatype
- Property List - a collection of parameters (some permanent and some transient) controlling options in the library

These key concepts are described in more detail in the following pages.

Files

A file acts as a container for a collection of objects, comprising of groups, datasets and other objects. Every file contains at least one object, which is the root group. All objects are members of the root group or descendants of the root group, creating a hierarchical system. In essence, a HDF5 file is very similar to a UNIX file system [25].

Each object has a unique identity within a single HDF5 file and can be accessed only by its name within the hierarchy of the file. Specific settings can be set when a file is created, which include version information and parameters of the global data structures. File creation properties are set permanently for the life of a file, whilst file access properties can be changed over time.

Groups

A HDF5 group is analagous to a file system directory. It is a structure containing zero or more HDF5 objects and has two parts [26]:

- A group header, which contains a group name and a list of group attributes.
- A group symbol table, which is a list of the HDF5 objects that belong to the group.

Datasets

A HDF5 dataset is a multidimensional array of data elements. The shape of the array is described by the dataspace object [27].

A data element is a single unit of data which may be a number, a character, an array of numbers or characters, or a record of heterogeneous data elements.

The dataspace and datatype are set when the dataset is created, and are fixed. The dataset creation properties are also set when the dataset is created. The dataset creation properties include the fill value and storage properties such as

chunking and compression. These properties cannot be changed after the dataset is created [11].

The dataset object manages the storage and access to all data. While the stored data is conceptually a contiguous rectangular array, it is physically stored and transferred in different ways, depending on the storage properties and the storage mechanism used. The actual storage may be a set of compressed chunks, and the access may be through different storage mechanisms and caches.

Dataspace

A dataset dataspace describes the dimensionality of the dataset. The dimensions of a dataset can be fixed (unchanging), or they may be unlimited, which means that they are extendible and can grow larger [10].

Properties of a dataspace consist of the rank (number of dimensions) of the data array, the actual sizes of the dimensions of the array and the maximum sizes of the dimensions of the array. For a fixed-dimension dataset, the actual size is set to be the same as the maximum size of a dimension [28].

A dataspace can also describe portions of a dataset, making it possible to do partial I/O operations on selections. Selection is supported by the dataspace interface (H5S). Given an n-dimensional dataset, there are currently four ways to do partial selection:

1. Select a logically contiguous n-dimensional hyperslab.
2. Select a non-contiguous hyperslab consisting of elements or blocks of elements (hyperslabs) that are equally spaced.
3. Select a union of hyperslabs.
4. Select a list of independent points.

Since I/O operations have two end-points, the raw data transfer functions require two dataspace arguments. The first one describes the application memory

dataspace or subset thereof, and the other describes the file dataspace or subset thereof [28].

Datatype

The HDF5 datatype object describes the layout of a single data element. A data element is a single element of the array; it may be a single number, a character, an array of numbers or carriers, or other data. The datatype object describes the storage layout of this data [29].

The HDF5 format makes it possible to store data in a variety of ways. The default storage layout format is contiguous, meaning that data is stored in the same linear way that it is organized in memory. Two other storage layout formats are currently defined for HDF5:

- **Compact storage** - This is used when the amount of data is small and can be stored directly in the object header.
- **Chunked storage** - This involves dividing the dataset into equal-sized 'chunks' that are stored separately.

Chunking is the preferred storage method, as it offers three main benefits:

1. It makes it possible to achieve good performance when accessing subsets of the datasets, even when the subset to be chosen is orthogonal to the normal storage order of the dataset.
2. It makes it possible to compress large datasets and still achieve good performance when accessing subsets of the dataset.
3. It makes it possible to efficiently extend the dimensions of a dataset in any direction.

Attribute

Any HDF5 Named Data Object (Group, Dataset, or Named Datatype) may have zero or more user defined Attributes. Attributes are used to document the object, and are stored within the object. An HDF5 Attribute comprises of a name and data. The data is described analogously to the Dataset, while the Dataspace defines the layout of an array of Data Elements. The Datatype then defines the storage layout and interpretation of the elements [24].

An Attribute is very similar to a Dataset, with the following limitations:

- An attribute can only be accessed via the object; attribute names are significant only within the object. Attributes cannot be shared.
- For practical reasons, an Attribute should be a small object (no more than 1000 bytes), in order to avoid large overheads.
- The data of an Attribute must be read or written in a single access (selection is not allowed).
- Attributes do not have Attributes.

Property List

HDF5 has a generic property list object. Each list is a collection of name-value pairs. Each class of property list has a specific set of properties. Each property has an implicit name, a datatype, and a value. See the figure below. A property list object is created and used in ways similar to the other objects of the HDF5 library.

Property lists are attached to the object in the library, and they can be used by any part of the library. Some properties are permanent (e.g., the chunking strategy for a dataset), and others are transient (e.g., buffer sizes for data transfer). A common use of a Property List is to pass parameters from the calling program to a VFL driver or a module of the pipeline.

Property lists are conceptually similar to attributes. They are relevant to the behavior of the library while attributes are relevant to the users data and application. They are also used to control optional behavior for file creation, file access, dataset creation, dataset transfer and file mounting [24].

2.3.3 HDF5 Tools

A variety of of tools exist which allow users of the HDF5 to access and manipulate files [30]. These include:

- **HDFView** - A visual program which allows users to open and view the entire contents of HDF files.

  The following image shows a file opened with the HDFView, with its contents and properties being displayed.

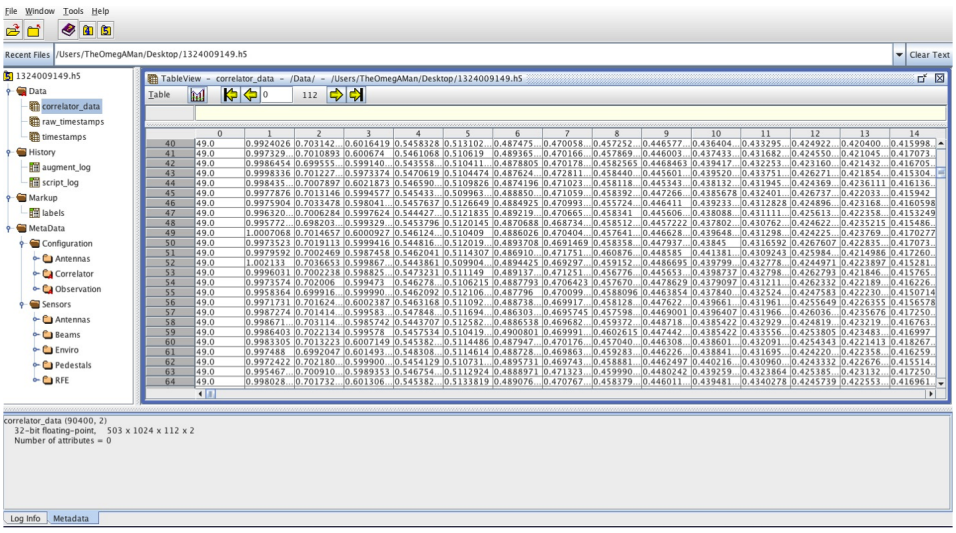

Figure 2.5: HDF5 models and implementations

- **HDF5 Command-line tools** - These include the options to import, debug, compress files and are available for most platforms
- **h5check** - Checks the validity of an HDF5 file
- **h5py** - A pythonic interface to the HDF5 format, which provides a variety of commands and functions in python

2.4 Data Compression

Data compression is the reduction in size of data in order to save space or transmission time. For data transmission, compression can be performed on just the data content or on the entire transmission unit (including header data) depending on a number of factors [31]. It works by encoding information using a fewer number of bits than the original representation [32]

There are numerous scientific projects similar to the SKA, such as the Gaia Astrometric Mission [33], Large Hadron Collider (LHC) [34] and the Meteosat [35], which require the generation of large amounts of data. However, the capacity of communications channels have not increased proportionally with time. As a result, data compression has become a crucial aspect in the design of such endeavours. Data compression reduces the time taken and the bandwidth used when transferring files, and can also reduce the costs involved with data storage [2]. In addition, many modern ground-based systems that must transfer incredible amounts of data, both between distant locations and within local networks of high-performance computers are currently in the planning or operation stages. These systems will also benefit from highly efficient data compressors [36].

Previous studies have discussed the big data challenges that will be faced by large radio arrays [5] and have explored the signal processing [1] [6] and data compression techniques [7] that are used in analyzing astronomical data.

The transmission and archiving of such an unprecedented amount of data requires tremendous computing time, computer storage, and I/O bandwidth. One technique that can reduce the data archive storage loads and network connection time requirements, without compromising data fidelity, is lossless data compression. Lossless data compression is a mature technology that has been used extensively in a variety of applications [37].

Lossless data compression techniques, such as ZIP, are commonly used on most modern computers, and are based on the Lambel-Ziv-Welch (LZW) algorithm or its variations. These common techniques generally yield poor compression ratios on data originating from spacecraft instruments [38].

Another well established technique is arithmetic coding [39]. It works on most types of data, but exhibits relatively slow speeds due to the need to update statistics along the process. The Rich algorithm has been adopted by the Consultative Committee on Space Data Systems (CCSDS) as the standard compression algorithm for space applications [40] [41] [42]. It was developed specifically for scientific data in a joint effort between NASA and the Jet Propulsion Laboratory (JPL), and has been implemented on many space missions in either instruments or data systems and baselined for many future satellites as well. The key requirements of the algorithm were high speed for real time processing, low complexity, and quick adaptation to statistics [38].

2.4.1 Data Compression Algorithms

Data compression algorithms determine the actual process of re-arranging and manipulating the contents of files and data to reduce their size. Golomb coding and Rice Coding are two of the most commonly used algorithms, and serve as the basis for numerous compression techniques.

Golomb Coding

Golomb coding is a lossless data compression method which is based on a simple model of the probability of values, independent of the content of the data . It works by reducing the average length per encoded value compared to fixed-width encoding [43].

For example, if integers from 0 to 1000 were transmitted, with a large majority of the values lying in the range between 0 and 10, most of the transmitted codes would have leading 0s that contain no information. Thus, to cover all values between 0 and 1000, a 10-bit wide encoding would be needed in standard binary. As most values would be below 10, at least the first 6 bits of every number would be 0 and would carry no information at all.

To rectify this with Golomb codes, the numbers will be divided by 10 and encoded with the quotient and the remainder separately. For most values, all that would have to be transmitted is the remainder which can be encoded using 4 bits at most. The quotient is then transmitted in unary coding, which encodes as a single 0 bit for all values below 10, as 10 for 10 to 19, 110 for 20 to 29 and so on.

For most of the values, the message size will have been reduced to a maximum of 5 bits, but it is still possible to transmit all values unambigously without separators. This comes at a high cost for the larger values, which is why the coding is optimal for 2-sided geometric distributions.

The long runs of 1 bits in the quotients of larger values can be addressed with subsequent run-length encoding. However, if the quotients consume too much space in the resulting message, this could indicate that other codes might be more appropriate than Golomb coding [44].

Rice Coding

The Rice algorithm is a subset of Golomb Coding, and consists of additions, subtractions, bit masking and shifts, making it computationally efficient [12].

It has been implemented in hardware for use on spacecraft missions and in embedded systems, and has been considered for use in compressing images from future space telescopes [45]. In its usual implementation, it encodes the differences of consecutive pixels using a variable number of bits. Pixel differences near zero are coded with few bits and large differences require more bits.

The algorithm adapts to any present noise by determining the number of pure noise bits to strip off the bottom and include directly in the output bit-stream, with no coding. The best value for this noise scale is computed independently for each block of 16 or 32 pixels. With such short blocks, the algorithm requires little memory and adapts quickly to any variations in pixel statistics across the image [2].

The CCSDS recommends the use of a data compression algorithm based on Rice codes. These codes are optimal for discrete Laplacian distributions, which are expected to occur under the premises of the CCSDS, after an adequate preprocessing stage [11].

Rice coders are a special case of Golomb coders, where the parameter m is a power of 2 (m = 2k, with k >0). Rice coders have 2k variable-length codes, starting with a minimum length of k + 1, using a very simple coding algorithm [46].

The length of a Rice code for an integer n coded using a parameter k is computed as:

$$1 + k + \left\lfloor \frac{n}{2^k} \right\rfloor .$$

Figure 2.6: The Rice equation [11]

The following figure shows rice codes for small values of k:

n	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
0	0|	0|0	0|00	0|000	0|0000	0|00000
1	10|	0|1	0|01	0|001	0|0001	0|00001
2	110|	10|0	0|10	0|010	0|0010	0|00010
3	1110|	10|1	0|11	0|011	0|0011	0|00011
4	11110|	110|0	10|00	0|100	0|0100	0|00100
5	111110|	110|1	10|01	0|101	0|0101	0|00101
6	1111110|	1110|0	10|10	0|110	0|0110	0|00110
7	11111110|	1110|1	10|11	0|111	0|0111	0|00111
8	111111110|	11110|0	110|00	10|000	0|1000	0|01000
9	1111111110|	11110|1	110|01	10|001	0|1001	0|01001
10	11111111110|	111110|0	110|10	10|010	0|1010	0|01010
11	111111111110|	111110|1	110|11	10|011	0|1011	0|01011
12	1111111111110|	1111110|0	1110|00	10|100	0|1100	0|01100
13	11111111111110|	1111110|1	1110|01	10|101	0|1101	0|01101
14	111111111111110|	11111110|0	1110|10	10|110	0|1110	0|01110
15	1111111111111110|	11111110|1	1110|11	10|111	0|1111	0|01111
16	11111111111111110|	111111110|0	11110|00	110|000	10|0000	0|10000

Figure 2.7: Rice codes for values n = 0 to 16 and parameter k = 0 to 5.

Setting the parameter k correctly will improve the compression ratios that are obtained for given data sets.

There is a rapid increase in code length for small values of k. It might occur that the expected data set only has low values and, under this condition, receiving a single high value would lead to an output code of hundreds or thousands of bits. To counter this, the CCSDS 121.0 recommendation introduces an adaptive layer to automatically select the best k parameter for each data block [40].

2.4.2 Data Compression Software

SZIP

SZIP is an implementation of the extended-Rice lossless compression algorithm. The Consultative Committee on Space Data Systems (CCSDS) has adopted the extended-Rice algorithm for international standards for space applications [37]. SZIP is reported to provide fast and effective compression, specifically for the data generated by the NASA Earth Observatory System (EOS) [47]. It was originally developed at University of New Mexico (UNM) and integrated with HDF4 by UNM researchers and developers.

As the graphs on the following page show, the primary gain with SZIP compression is in its compression speed, relative to GZIP and the other algorithms it was compared to. SZIP also provides higher compression ratios. These results, including the data presented in the graphs below, are from tests conducted by Pen-Shu Yeh, et al. [47].

SZIP and HDF5

SZIP compression software, providing lossless compression of scientific data, has been provided with HDF software products as of HDF5 Release 1.6.0.

SZIP is a stand-alone library that is configured as an optional filter in HDF5. Depending on which SZIP library is used, an HDF5 application can create, write, and read datasets compressed with SZIP compression, or can only read datasets compressed with SZIP.

Applications use SZIP by setting SZIP as an optional filter when a dataset is created. If the SZIP encoder is enabled with the HDF5 library, data is automatically compressed and decompressed with SZIP during I/O. If only the decoder is present, the HDF5 library cannot create and write SZIP compressed datasets, but it automatically decompresses SZIP compressed data when data is read [48].

The following two graphs show the performance of SZIP compared to other algorithms, as mentioned on the previous page.

The test data used in these table are real sensor data and derived products from Earth observation missions such as MODIS Airborne Simulation (MAS), with a total data volume of 930 MB [47].

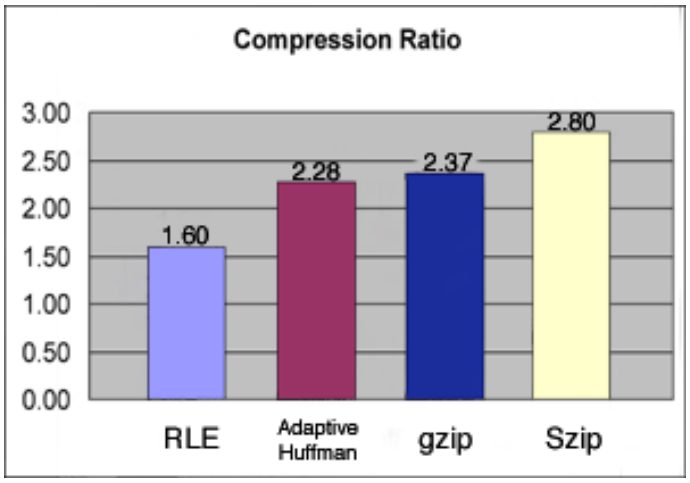

Figure 2.8: Compression ratios compared to SZIP for HDF5 [48]

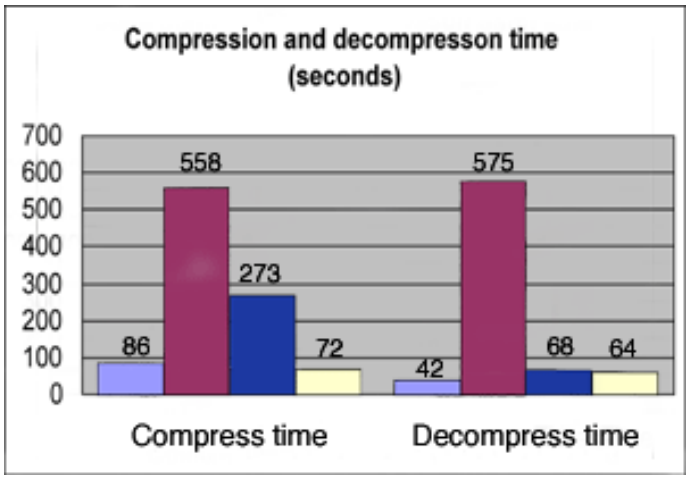

Figure 2.9: Compression and decompression times compared to SZIP for HDF5 [48]

GZIP

GZIP, is a combination of LZ77 and Huffman coding and is based on the DEFLATE algorithm. DEFLATE was intended as a replacement for LZW and other data compression algorithms which limited the usability of ZIP and other commonly used compression techniques.

GZIP is often also used to refer to the GZIP file format, which is [49]:

- A 10-byte header, containing a magic number
- Version number
- Timestamp
- Optional extra headers, such as the original file name
- A body, containing a DEFLATE-compressed payload
- an 8-byte footer containing a CRC-32 checksum and the length of the original uncompressed data

GZIP is generally used to compress single files. However, its file format also allows the multiple compression stream to be concatenated [50].

GZIP is different from the ZIP compression technique, which is also based on the DEFLATE algorithm. The ZIP format is capable of storing collections of files without the need for an external archiver. However, it does not store data in as compact a manner as tarballs, as it compresses files individually and does not take advantage of the redundancy between files.

LZF

The LZF filter is a stand-alone compression filter for HDF5, which can be used in place of the built-in DEFLATE or SZIP compressors to provide faster compression. The target performance point for LZF is very high-speed compression with an 'acceptable' compression ratio [51].

In benchmarking trials conducted with floating-point data, a filter pipeline with LZF typically provided compression times which were 3x-5x faster than DEFLATE, and provided 50% - 90% of compression ratio obtained by DEFLATE.

The LZF filter works for all datatypes on which DEFLATE works, unlike SZIP. These include compound, opaque, array and user-defined types. There are also no parameter settings to adjust [51].

The LZF filter is written in C and may be included in C++ applications. No external libraries are required. HDF5 versions 1.6 and 1.8 are both supported. The license is 3-clause BSD [52].

The compression ratios and times obtained for LZF depend on many different factors, such as the storage datatype and the range of values used. It can be used on arbitrary HDF5 types, including strings, compound and arrays in addition to scalars. However, its performance for multi-byte floating point and integer data sets is of particular importance, as they are commonly used [51].

PEC

The Prediction Error Coder (PEC) is a highly optimized entropy coder developed by researchers at the University of Barcelona in conjunction with the Gaia mission, which is a space astrometry mission of the European Space Agency (ESA) [15].

The scientific payload of the Gaia mission will generate complex sets of data. The data compression solutions initially proposed for the mission are not applicable when considering the flight realistic limitations of the mission [53]. For example, the available processing power on-board is not enough for applying standard compressors (such as ZIP) to a data stream of 5 Mbps or more, which is a requirement for the mission. Even a quick algorithm like the CCSDS 121.0 recommendation for lossless data compression, requires an excessive percentage of the on-board processor capacity [40]. Moreover, a software implementation is mandatory in this case, so the efficient hardware implementations of CCSDS 121.0 become useless [40].

An alternative is to use the Rice coder alone, with pre-calibrated values of the k parameter for each set of samples. Nevertheless, this solution is not reliable at all, because the value of k must be chosen very carefully in order to obtain good compression ratios for a given set of data. And most importantly, a single outlier in the data can lead to large expansions in the compression ratios [15] [53].

Description of the PEC

Rice codes are the adequate solution when the data to be compressed follows a geometric (or Laplacian) statistical distribution, which often arises after an adequate pre-processing stage [54]. However, any deviation from these statistics can lead to a significant decrease of the final compression ratio. This weakness is solved by the PEC. The PEC is focused on the compression of prediction errors, thus a pre- processing stage based on a data predictor plus a differentiator is needed. It is a very fast and robust compression algorithm that yields good ratios under nearly any situation.

The PEC has three coding options:

- Low Entropy (LE)
- Double-Smoothed (DS)
- Large Coding (LC)

Large Coding makes use of unary prefix codes, while Low Entropy and Double Smoothed rely on the 'minus zero' feature of signed prediction errors. This means that an unused code can be used as an escape sequence, without requiring additional bits. It should be emphasized that a pre-processing stage is required when using the PEC, and that separate sign bits are used to avoid a mapping stage.

The figure below shows a schematic view of the PEC operation and its three coding strategies.

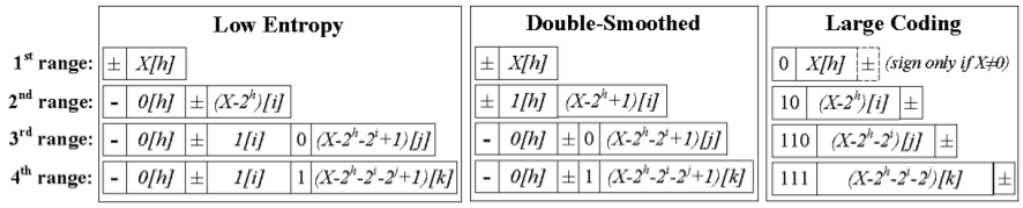

Figure 2.10: Overview of the Prediction Error Coder (PEC) and its three coding strategies [11]

The three coding options share the same principles, which is that the range of the data to be coded is split into four smaller ranges, or segments. The size of each segment is determined by its corresponding coding parameter (h, i, j, k), which indicates the number of bits required to code the values of that segment. This set of parameters is known as the coding table. Both the coding strategy and the coding table must be adequately configured before applying the PEC to a given dataset. Accordingly, the PEC can be considered a partially adaptive algorithm, where the adequate segment is selected for each one of the values [11].

The PEC automatically assumes that data values are close to zero. When this occurs, coding parameters are chosen in a way that the first segments are significantly smaller than the original symbol size, while the last segments are slightly larger. This leads to a compressed output, while the ratio is determined by the probability density function of the data combined with the selected coding table. Another advantage is that the PEC is flexible enough to process data distributions with probability peaks far from zero. With an adequate choice of its freely adjustable parameters, good compression ratios can still be reached when such distributions are processed.

An adequate coding table and coding option must be selected for the operation of the PEC. In order to easily determine the best configuration for each case, an automated PEC calibrator was developed, which only requires a representative histogram of the values to be coded. It works by analyzing the histogram and determining the optimum configuration of the PEC. This is done by testing each of the possible histogram configurations of the PEC, and selecting the one offering the highest compression ratio. In the case of space missions like Gaia, the calibrator must be run on-ground with simulated data prior to launch. The PEC is robust enough to offer good compression ratios despite of variations in the statistics of the data. Nevertheless, the calibration should be repeated periodically during the mission, re-configuring the PEC to guarantee the best results [11].

FAPEC

The FAPEC (Fully Adaptive Prediction Error Coder) is a fully adaptive model of the PEC, meaning that it automatically calibrates the necessary settings and parameters based on the type of data that needs to be compressed.

It is a proprietary solution commercialized by DAPCOM Data Services S.L., a company with expertise on efficient and tailored data compression solutions, besides data processing and data mining. The company offers not only this efficient data compression product, applicable to a large variety of environments, but also the development of tailored pre-processing stages in order to maximize the performance of the FAPEC on the kind of data to be compressed [13].

LZ4

The LZ4 algorithm was developed by Yann Collet and belongs to the LZ77 family of compression aglorithms. Its most important design criteria is simplicity and speed.

The algorithm creates compressed blocks, which are composed of sequences. Each sequence starts with a token, which is a one byte value, separated into two 4-bits fields (which range from 0 to 15) [55].

The first field is used to indicate the length of literals. Each additional byte then represents a value of 0 to 255, which is added to the previous value to produce a total length. When the byte value reaches 255, another byte is output. Since there is no limit to the size, there can be any number of bytes following the token [55].

The sequence of bytes is shown in the image below:

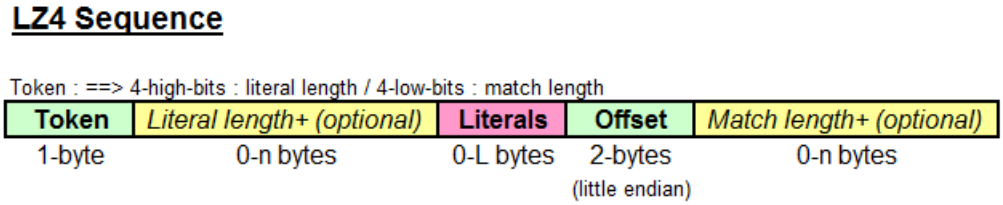

Figure 2.11: The LZ4 Format [55]

The literals come after the token and the optional literal length bytes. The literals are followed by the offset, which is a 2 byte value between 0 and 65535. The matchlength is then extracted using the second token field, which is a 4-bit value ranging from 0 to 15.

Using the offset and the matchlength, the decoder can copy the repetitive data from the buffer, which is already decoded. Once the matchlength is decoded and the end of the sequence is reached, a new sequence is initiated.

Specific parsing rules exist ir order to maintain compatibility assumptions made by the decoder. They are that:

1. The last 5 bytes are always literals
2. The last match cannot start within the last 12 bytes

As a result of this, files that have less than 13 bytes can only be represented as literals. This ensures that the decoder will never read beyond the input buffer, nor write beyond the output buffer [55].

There is no restriction to the way LZ4 searches for, and finds matches. Examples of this are standard hash chains, fast scans, 2D hash tables and almost any other known method. Advanced parsing can also be achieved while respecting the compatibility of the file format [56].

It is possible to modify the size of the hash table without affecting the compatibility of the format being used. This is an important feature for restricted memory systems, as the hash table can end up being reduced in size. This could lead to an increase in false positives, which would greatly reduce the compression power of the algorithm.

Nonetheless, LZ4 still performs as expected and remains fully compatible with more complex and memory intensive applications. The decoder functions irrespective of the method used to find the matches, and requires no additional memory [55].

The graph on the following page shows results from a simple scenario, in which a file is compressed, sent over a variable-speed pipe, and then decompressed. The total compression and decompression times are added together and compared for a variety of compression methods.

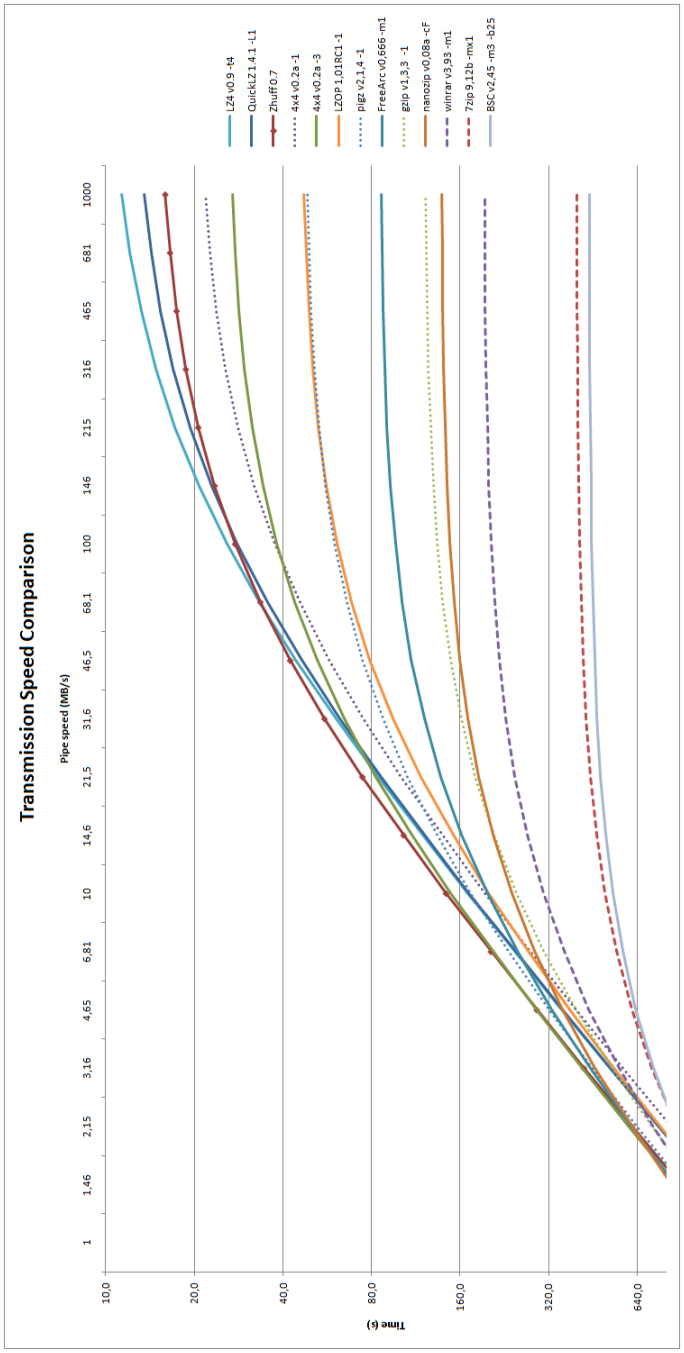

Figure 2.12: Comparison of LZ4 compression speed [57]

Chapter 3

Methodology

This project used an experimental research process, which was followed by an iterative development strategy. This chapter describes the four stages that were developed to meet the stated objectives of this project. These were:

1. Finding the user requirements through meetings and discussions
2. Performing a thorough literature review on data compression and the HDF5 file format
3. Designing a testbed system and carrying out testing on the SKA data sets
4. Evaluating the results and drawing conclusions, as well as providing suggestions for future work

The steps taken during the stage of gathering the user requirements and the challenges that were faced in this process are discussed. This is followed by describing how the literature review was used, as well as the design and implementation of the different compression algorithms. The last section details the methods that were used to test the data sets.

The following sub-sections refine each phase of the methodology followed.

The image below provides an illustration of how the different phases came together.

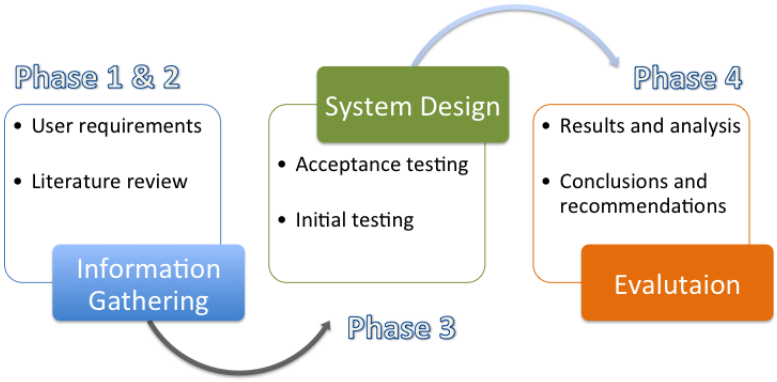

Figure 3.1: Phases of the project

3.1 Requirements and Constraints

This sub-section explains the process by which requirements were established, which were then used to guide the design of the testing system. Section 3.1.1 outlines the user requirements while section 3.1.2 discusses the design constraints.

3.1.1 User Requirements

This project was carried out in conjuction with the SKA Africa office, where numerous engineers and scientists analyse and process collected data on a daily basis [58]. The users of data collected by the MeerKAT array also include astrophysicists and radio astronomers [16]. The researcher had periodic meetings with engineers from the SKA to determine the initial requirements and necessary outcomes of the study [9]. These included finding out the need for study at the

SKA, what the results would be used for and how to carry out the research and testing.

The researcher also met with professors from the Astronomy Department at the University of Cape Town to determine how they make use of astronomical data and how important data compression is [8].

3.1.2 Constraints

An initial set of constraints was obtained from the first meeting with the SKA:

- Data compression that occurs must be lossless
- Compression algorithm that is run will occur on a server after the raw data is collected and stored
- Testing and development should be done using the h5py interface
- A program that can work as a stand-alone executable or that can be integrated into the SKA framework should be created as a final outcome.

These constraints were modified upon further meetings, due to the changes that occured in the process chain and design at the SKA [13].

- The datasets would no longer be permanently downloaded to a local access point by end users. Instead, they would only be streamed and be open as long as they were being accessed. As a result, compression should occur while the files are being streamed, and should reach the end user as a compressed file, leaving the original file on the server untouched.
- Compression time must be minimized, while compression ratios greater than 1.1 would be seen as a great result, given the size of the files eg. a 2Gb file compressed to 1.8Gb would represent a compression of 200Mb, which is a considerable saving on storage space and computer memory.

3.2 How the literature was used

This sub-section explains how specific knowledge gained while conducting the Literature Review (Chapter 2) was applied to the methodology of this project in order to meet the stated objectives.

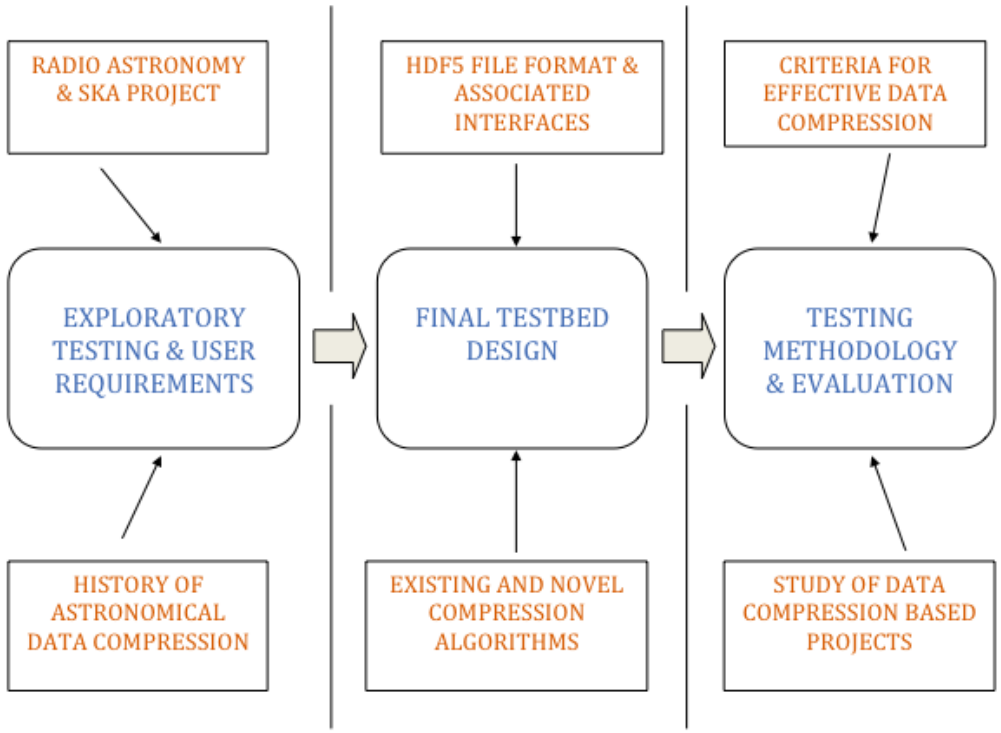

Figure 3.2: Flow Chart showing the application of the Literature Review to the Methodology

As shown in Figure 3.2, gaining a strong understanding of radio astronomy, the usage of astronomical data in the field and common compression techniques was crucial. This allowed some basic testing to be carried out, which determined the feasibility of the study and allowed for refinement of the user requirements.

This was followed by an in depth understanding of HDF5 and all of the compression techniques that exist for the format. This step aided in the initial and final design of the tested system.

Finally, the testing and evaluation processes were derived from previous studies that had been carried out on astronomical data compression.

3.3 System Design and Testing

This sub-section describes the design of the system used in this project, as well as the testing process used to obtain the necessary results.

3.3.1 System Design

Based on the stated objectives for this project and the information gathered from the literature review, the intended system for testing and evaluating the study was designed. The system comprised three parts, namely the testbeds, compression techniques and datasets. Figure 4.1 in the next chapter shows how these components work together. The system components are elaborated below:

- Testbeds - These encompass the different computer systems that were used to run and test the different compression algorithms. It aimed to simulate an environment that would be similar in nature to the SKA's processing centre.

- Compression techniques - The different compression techniques under investigation would run on the testbeds, and their performance and efficiency was monitored.

- Datasets - These consist of the raw, uncompressed datasets that were provided by the SKA, as well as the compressed files created by running them through the compression algorithms on the testbeds.

3.3.2 Testing

The testing process itself involved two stages. The first, was an exploratory phase where different existing and novel compression techniques were investigated, to understand the differences between each method and to determine which one provided the optimal functionality.

During the second stage, the best compression technique was chosen and the testbed was modified to fit the updated requirements from the SKA. This involved attempting to compress the datasets while streaming them from one computer to another, simulating the SKA environment where the end users access the datasets from a remote server.

3.4 Evaluation

The evaluation process used in this project involved determining three key metrics regarding the performances of the different compression techniques. These metrics were determined based on recommended evaluation methods used by Pence et. al [2], Yeh et. al [47] and Sanchez et al. [11] . These metrics are described below:

- **Compression Ratio** - This is the direct ratio between the size of the compressed file and the size of the uncompressed file. The compression ratio is normally the most important factor in data compression, as it shows how much storage space can be saved as a result of the compression. For each dataset, the uncompressed size would be recorded, and would then be divided by the size of the compressed file to provide the ratio.
- **Compression Time** - This is the time taken to perform the operation of compressing a file. This was calculated by running each compression technique and measuring the amount of time taken to open, compress and produce the final file.
- **Memory Usage** - This is the amount of computer memory (RAM) that is occupied by the compression program, and is determined by looking into all of the processes that are running on the computer.

These metrics were put together when determining the overall performance of each compression technique. The features of the different testbeds were considered as well to provide a clear analysis, based on the requirements of the project.

Chapter 4

System Design and Testing

This chapter presents and discusses the design of the system used in this poroject and looks at the different components that were used in its development (as described in section 3.3).

After analysing previous studies associated with data compression [11], [5],, [2], [7] [59] the system was designed and progressively refined. As was mentioned in the previous chapter, the system comprises of the:

- **Testbeds** - Computers and software used for testing
- **Datasets** - Files collected from SKA to be tested on
- **Compression techniques** - Techniques used to compress datasets on the testbeds

The first subsection presents the high level system design, which shows how these different components of the system interact with each other. Each part of the system will then be elaborated on in further detail over the following pages.

4.1 High Level System Design

The figure below shows the high level system design, which describes how the separate components come together to make the system function.

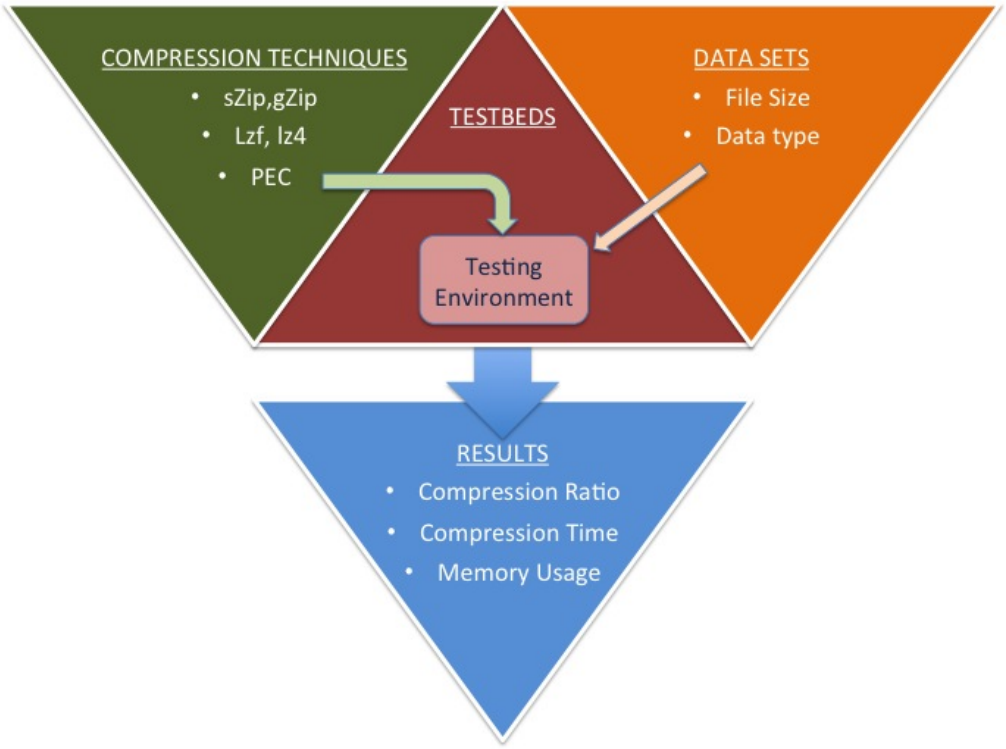

Figure 4.1: High Level System Design

The different compression techniques were installed on the testbeds. When the datasets were loaded or read, the testing environment was activated and the algorithms were run on the files being tested. The achieved compression ratio, time taken and memory used (in certain scenarios) then formed the results for this project.

4.2 Testbeds

The testbeds consist of the computers, and the software applications and tools on them, that are used to run and test the performance of the different compression techniques on the datasets.

In order to obtain relevant results, it was necessary to attempt to simulate the computing environment used by the SKA. This included:

- **Computers running Linux-based operating systems** - The majority of the machines at the SKA run versions of the Ubuntu operating system. Given that Ubuntu is open source and that the researcher had previous experience using it, it was chosen as operating system to use.
- **h5py Interface** - The h5py interface is a python module designed for the HDF5 format. It allows users to easily access and manipulate HDF5 files using python commands.
- **Streaming files from a server** - This involved streaming the datasets from another computer (which acted as the server) and attempting to compress them as they were being read.

The main computer (primary testbed) that was used had the following specifications:

- Intel Core 2 Duo Processor, E6550 @ 2.33 GHz
- 2GB of RAM
- Windows XP and Ubuntu 12.04
- 32 Bit OS

The second machine (server testbed) was used to host and send the data when re-creating the streaming environment. This machine had the same specifications as the main computer.

Since the code for the FAPEC could not be used due to copyright and commerical reasons, it was tested using a machine by Dr. Jordi Portell, one of the FAPEC's developers, which had the following specifications:

- Intel i7-2640M Processor @ 2.8 GHz
- 8GB of RAM
- Gentoo Linux
- 64 Bit OS

The following images show how the primary testbed and server testbed were connected up to each other using a network cable.

Figure 4.2: Front view of the two machines

Figure 4.3: Rear view

Modules that were needed to carry out the testing were installed on the primary testbed. These included compilers for different languages, including C, C++ and Python, as well as the h5py interface. HDFView was also installed and used to open the data sets to view and analyze their contents. It also acted as a method of verification to confirm that the compressed files are the same as the original files.

The following image shows how the the h5py interface is invoked after running python. From this stage, a file can be accessed using python commands.

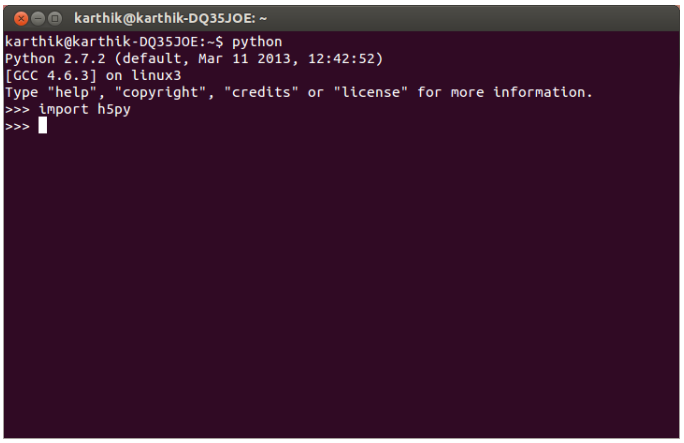

Figure 4.4: The h5py Interface invoked within Python

This image shows a dataset open in HDFView, which allows the user to view the entire contents of the file. This includes the antenna and correlator data, as well as metadata associated with the entire file eg. which antenna it was captured with.

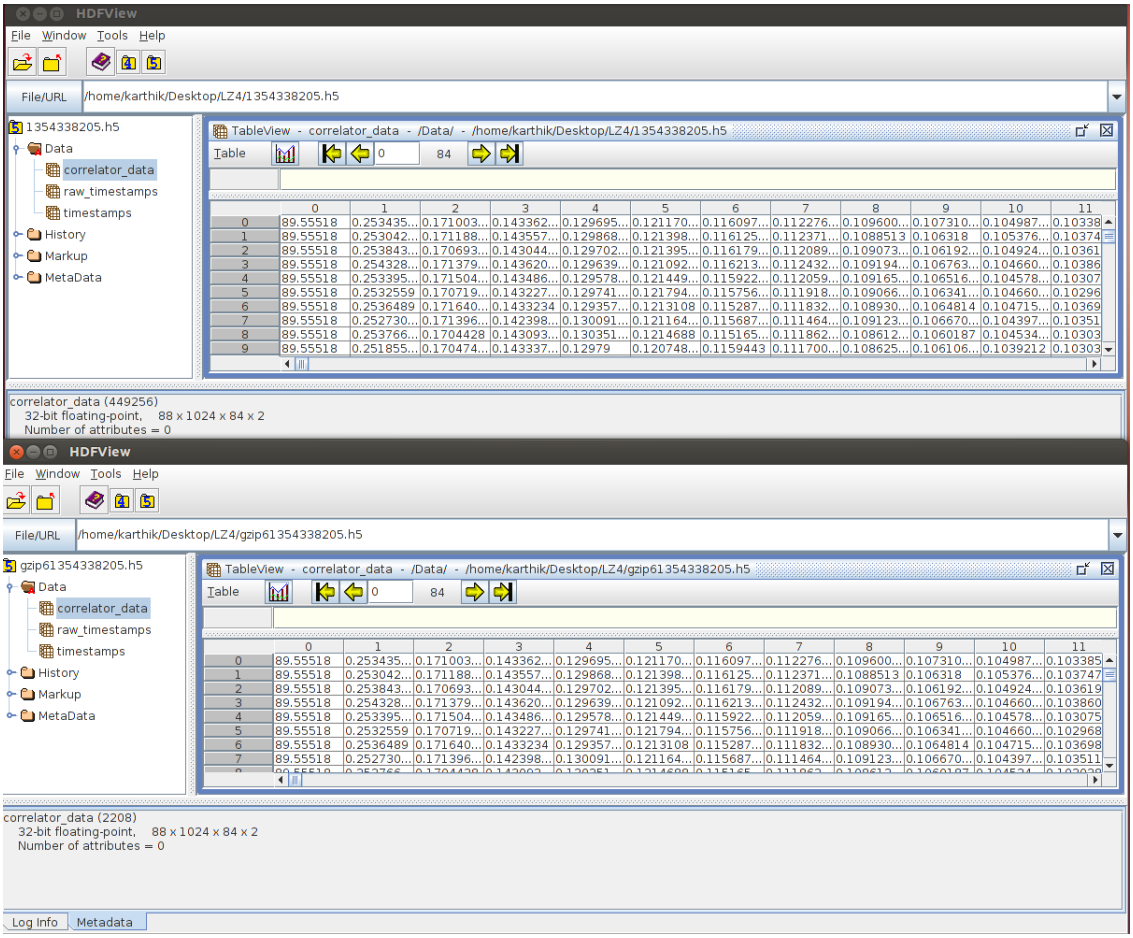

Figure 4.5: Original and compressed dataset opened in HDFView

Every compressed file was opened with HDFView to ensure that its contents were the same as the original file. The image above shows a dataset along with a compressed version of it (using GZIP), verifying that the contents of the correlator are the same, as well as the dataspace dimensions.

4.3 Datasets

A total of 11 datasets were obtained from the SKA. They ranged from 30MB to 9.35GB in size, of which 10 were collected during a 24 hour period on the 1st of December, 2012. File number 6 was collected on the 16th of December 2011. The table below shows each dataset and their corresponding details.

Table 4.1: List of the datasets

File Number	Filename	File size (MB)	Dataspace Dimensions
1	1354388597	29.8	37X1024X84X2
2	1354388388	43.9	55X1024X84X2
3	1354338205	69.6	88X1024X84X2
4	1354333641	75.9	96X1024X84X2
5	1354373685	342.4	105X4096X84X2
6	1324009149	466.5	503X1024X112X2
7	1354329284	2,720	3483X1024X84X2
8	1354333815	3,390	4347X1024X84X2
9	1354323010	4,690	6009X1024X84X2
10	1354338467	5,590	7020X1024X84X2
11	1354373850	9,350	2868X4096X84X2

Each file had an automatically generated filename, and they have been listed in ascending file size. The dataspace dimensions indicate the number of data entries collected and stored from the correlator in each dataset.

4.4 Compression Techniques

A considerable amount of exploratory testing was carried out using different compression algorithms and techniques in the early stages of this project. This involved:

- Becoming familiar with the process of compiling, installing and running different compression programs.
- Testing as many available compression programs to determine which ones worked best on the given data sets. This process was carried out in conjuction with the literature review.

Once the researcher became familiar with the process, the optimal compression programs were chosen and more rigorous testing began. It should be noted once again that the programs/algorithms had to provide lossless compression.

The selected programs were:

- SZIP
- GZIP
- LZF
- FAPEC
- LZ4

The following charts and images highlight some of the differences between the algorithms. They also highlight important lines of code from the programs that were run to test the different algorithms. The default programs were modified where needed, changing certain parameters and methods in which the files were processed.

SZIP

The flowchart below gives an overview of how the SZIP program works. The following pages show lines of code which expand on the this.

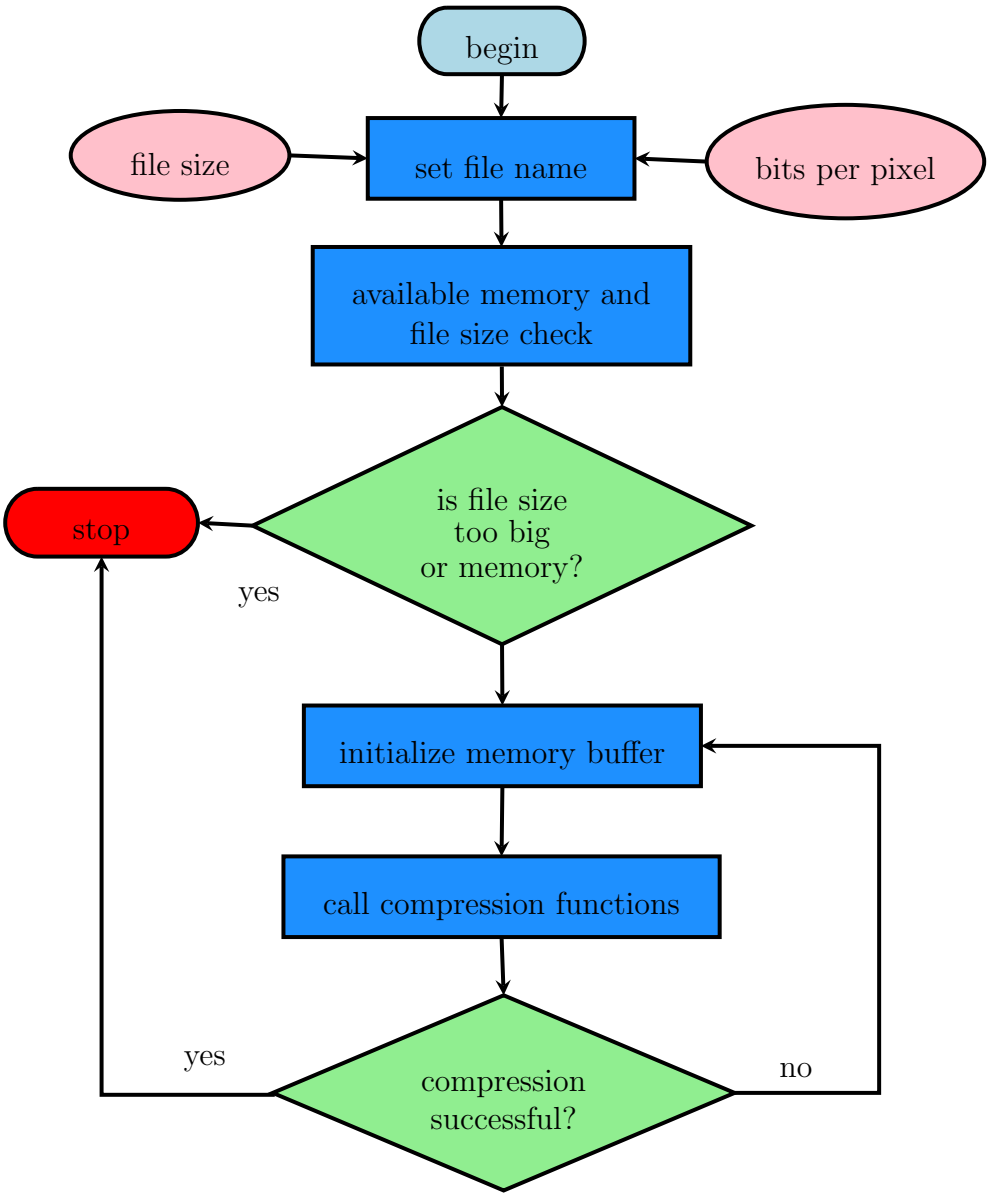

Figure 4.6: SZIP program flowchart

The main SZIP program is designed to operate on images. The file name, size and bits per pixel (set to a default value when not an image) are declared. The memory buffer is then initiated, calling the encoding and decoding functions.

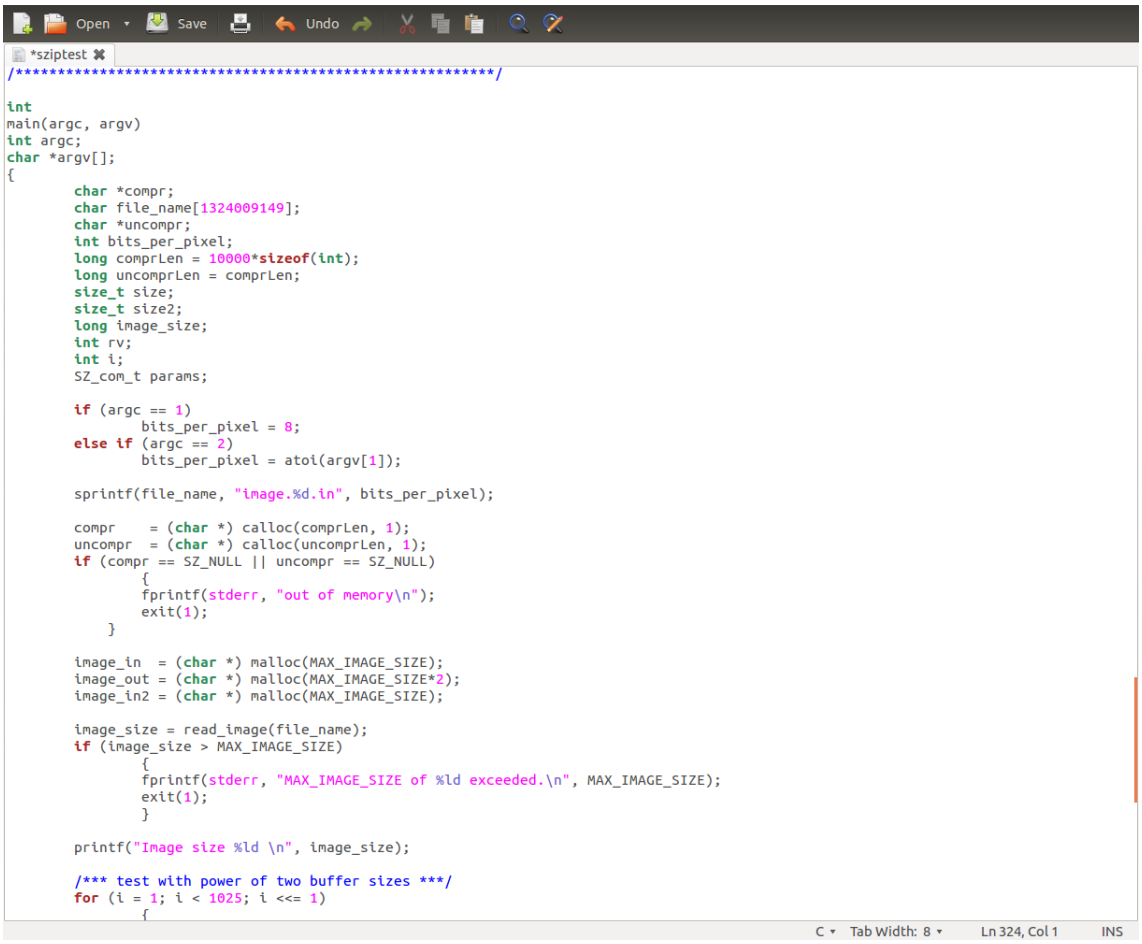

Figure 4.7: SZIP program Code Part 1

The 'for' loop shows how the functions needed to encode and process the file are called, with the compression process being deemed a success or failure at the end.

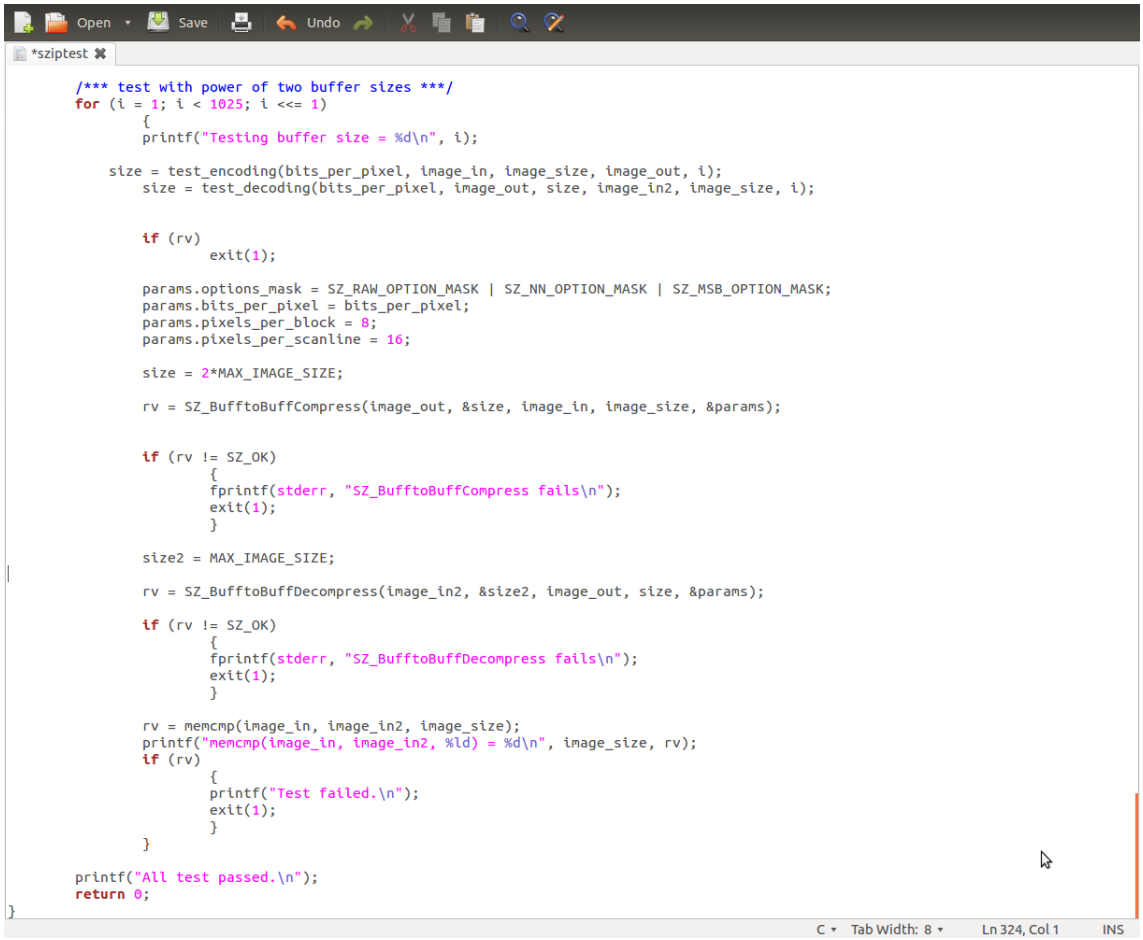

Figure 4.8: SZIP program code Part 2

LZF

The flowchart below shows how the LZF testing program worked.

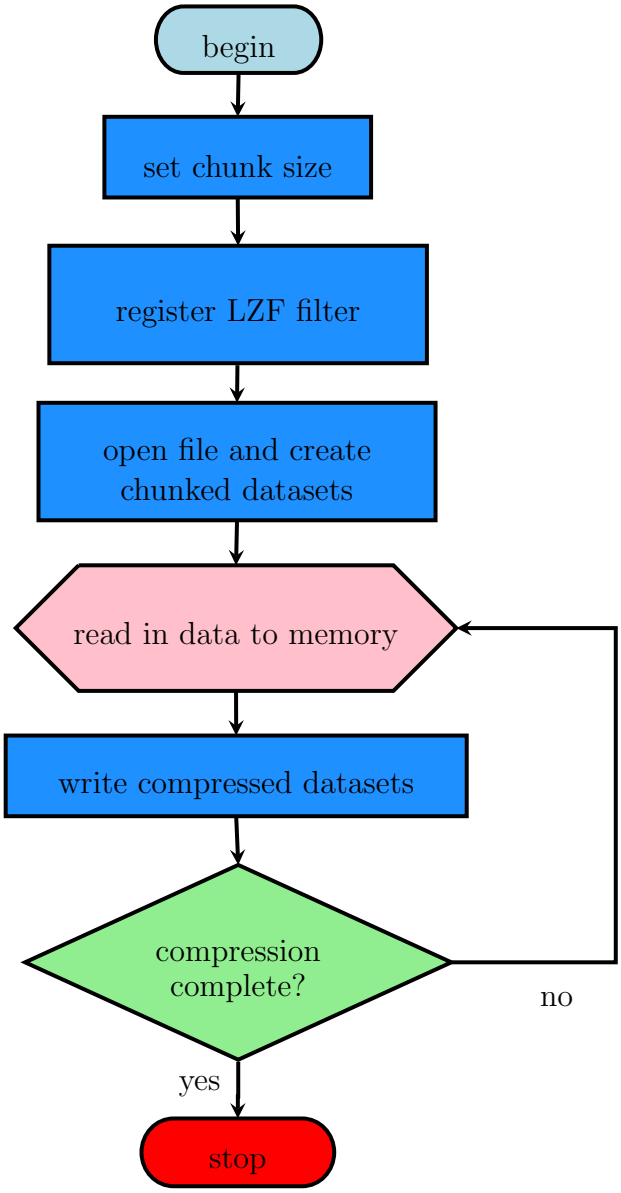

Figure 4.9: LZF program flowchart

This section of code from the LZF filter shows how smaller datasets are created within the file, with the compression (through the different 'shuffle' and 'filter' functions) being applied to the datasets.

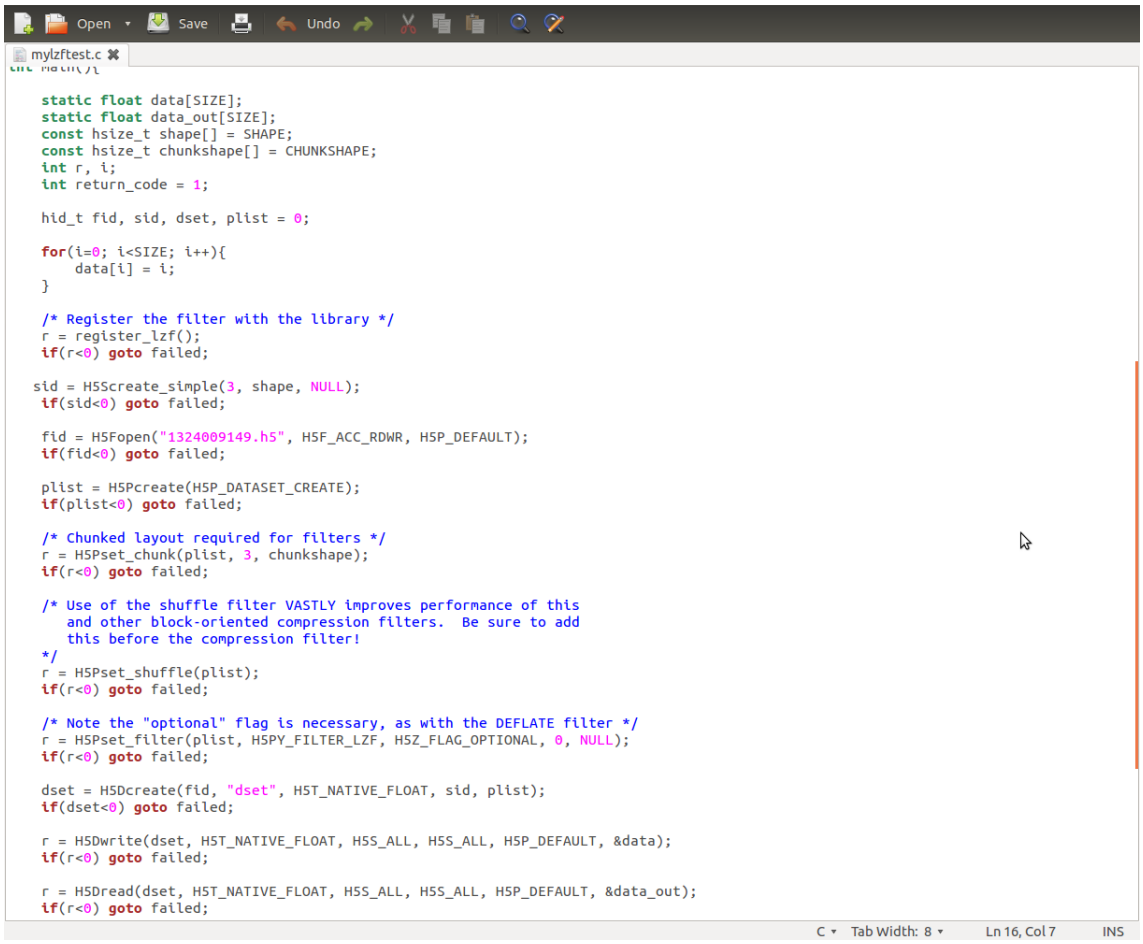

Figure 4.10: LZF program code

FAPEC

As the entire code for the FAPEC could not be obtained, the following image containing code will show how the program is initiated.

main.c (~/Dropbox/Masters Work/Screenshots/redatasets) - gedit

Open Save Undo

main.c

```
32 int main(void)
33 {
34
35     hid_t       file, dataset;         /* File and dataset           */
36     hid_t       dataspace;             /* Dataspace handles          */
37         hid_t           memspace;
38     hid_t       cparms;
39     hsize_t     dims[2];              /* Dataset dimensions          */
40     herr_t      status;                /* Error checking             */
41     int         data[ROWS][COLS];            /* Data to fill dataset        */
42     int         rank, rank_chunk;
43     hsize_t     chunk_dims[2];
44     int         data_out[ROWS][COLS];  /* buffer for dataset to be read */
45
46
47     int         i, j;                  /* Looping control vars       */
48
49         H5Z_class_t dummyFilter;           /* dummy Filteraset           */
50
51         /* Setup dummy filter */
52         dummyFilter.version = 1;
53     dummyFilter.id = FILTER_DUMMY;
54     dummyFilter.name = "DUMMY FILTER";
55     dummyFilter.can_apply= &dummy_can_apply;
56     dummyFilter.set_local= &dummy_set_local;
57     dummyFilter.filter =   &dummy_filter;
58         dummyFilter.encoder_present = 1;
59         dummyFilter.decoder_present = 1;
60
61         /* Register dummy filter */
62         status = H5Zregister(&dummyFilter);
63         assert(status>=0);
64
65     /* Data and output buffer initialization.                         */
66     for (j = 0; j < ROWS; j++) {
67                 for (i = 0; i < COLS; i++){
68                     data[j][i] = i + j;
69                 }
70     }
```

C Tab Width: 8 Ln 1, Col 1 INS

Figure 4.11: FAPEC program code Part 1

Lines 35-44 show how the FAPEC is designed to specifically work on HDF5 files, as it takes in the dataset dimensions as key parameters before creating the buffer for the dataset to be read.

These brief descriptions of the SZIP, LZF and FAPEC testing programs show how they slightly vary from each other. SZIP uses 'bits per pixel' as a key parameter for compression, while LZF works by creating chunks within the file and applying compression on the smaller datasets, lending credence to it's compression speed. And the FAPEC, as shown in the previous page, is tailored to the HDF5 format and modifies it's compression based on the structure of the files.

4.4.1 h5repack testing

The h5repack tool was used initially to determine the performance of the SZIP and GZIP algorithms, which were written and compiled in C. The two images below show how the input parameters and compression settings were set.

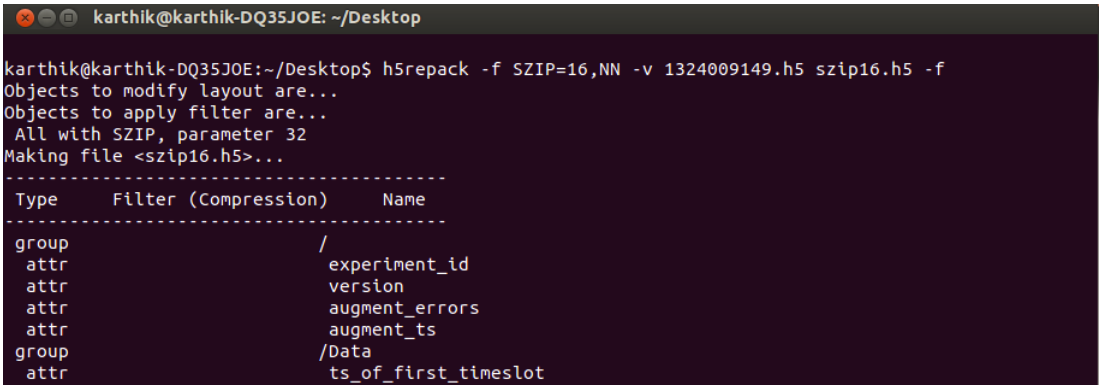

Figure 4.12: File being compressed in C using SZIP

Figure 4.13: File being compressed in C using GZIP

Figure 4.13 shows how all of the datasets within the file are compressed one by one using GZIP.

```
karthik@karthik-DQ35JOE: ~/Desktop
 attr                        type
group                       /MetaData/Sensors/Pedestals/ped6
dset     GZIP   (0.667:1)   /MetaData/Sensors/Pedestals/ped6/rfe3.rfe15.noise.coupler.on
 attr                        name
 attr                        description
 attr                        units
 attr                        type
dset     GZIP   (0.667:1)   /MetaData/Sensors/Pedestals/ped6/rfe3.rfe15.noise.pin.on
 attr                        name
 attr                        description
 attr                        units
 attr                        type
group                       /MetaData/Sensors/Pedestals/ped7
dset     GZIP   (0.667:1)   /MetaData/Sensors/Pedestals/ped7/rfe3.rfe15.noise.coupler.on
 attr                        name
 attr                        description
 attr                        units
 attr                        type
dset     GZIP   (0.667:1)   /MetaData/Sensors/Pedestals/ped7/rfe3.rfe15.noise.pin.on
 attr                        name
 attr                        description
 attr                        units
 attr                        type
group                       /MetaData/Sensors/RFE
dset     GZIP   (0.889:1)   /MetaData/Sensors/RFE/center-frequency-hz
dset     GZIP   (0.719:1)   /MetaData/Sensors/RFE/rfe7.lo1.frequency
 attr                        name
 attr                        description
 attr                        units
 attr                        type
karthik@karthik-DQ35JOE:~/Desktop$
```

Figure 4.14: Datasets within the file being compressed using GZIP

4.4.2 h5py testing

The h5py interface was then used to run and test and SZIP, GZIP and the LZF filter. Different parameters were used, specifying the number of chunks, coding method and and dataset shapes.

The image below shows an instance of compression using GZIP, with the dataset shape set at 100X100.

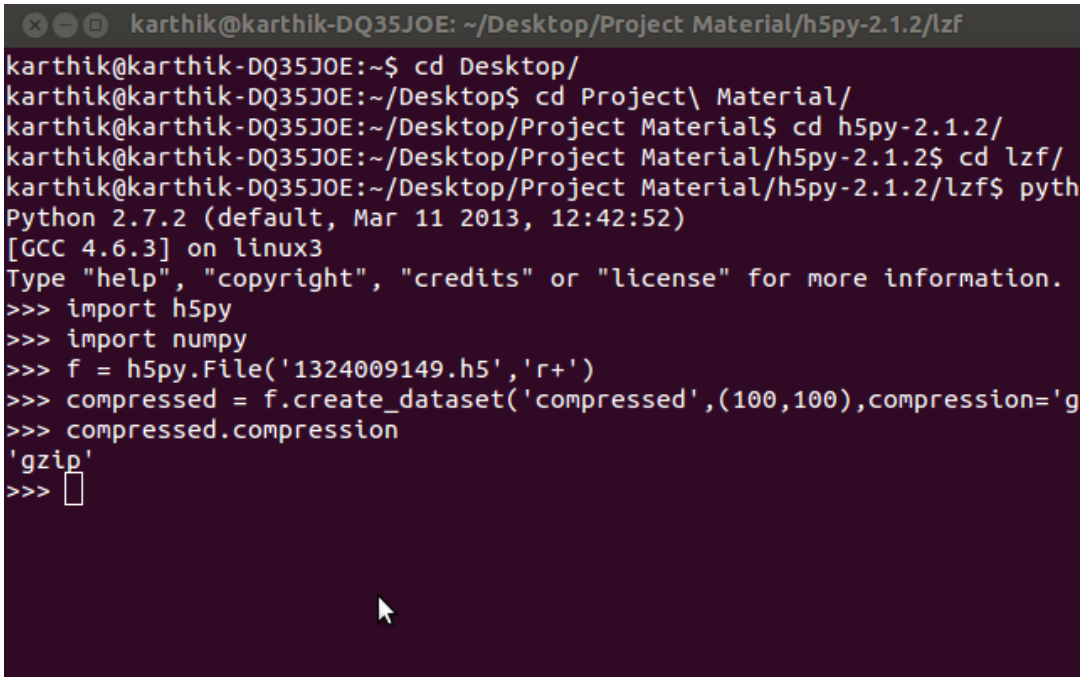

Figure 4.15: GZIP test run using h5py interface

4.4.3 LZ4 Testing

The LZ4 algorithm worked as a stand alone program and was executed as such. It takes in the input filename and output filename as parameters.

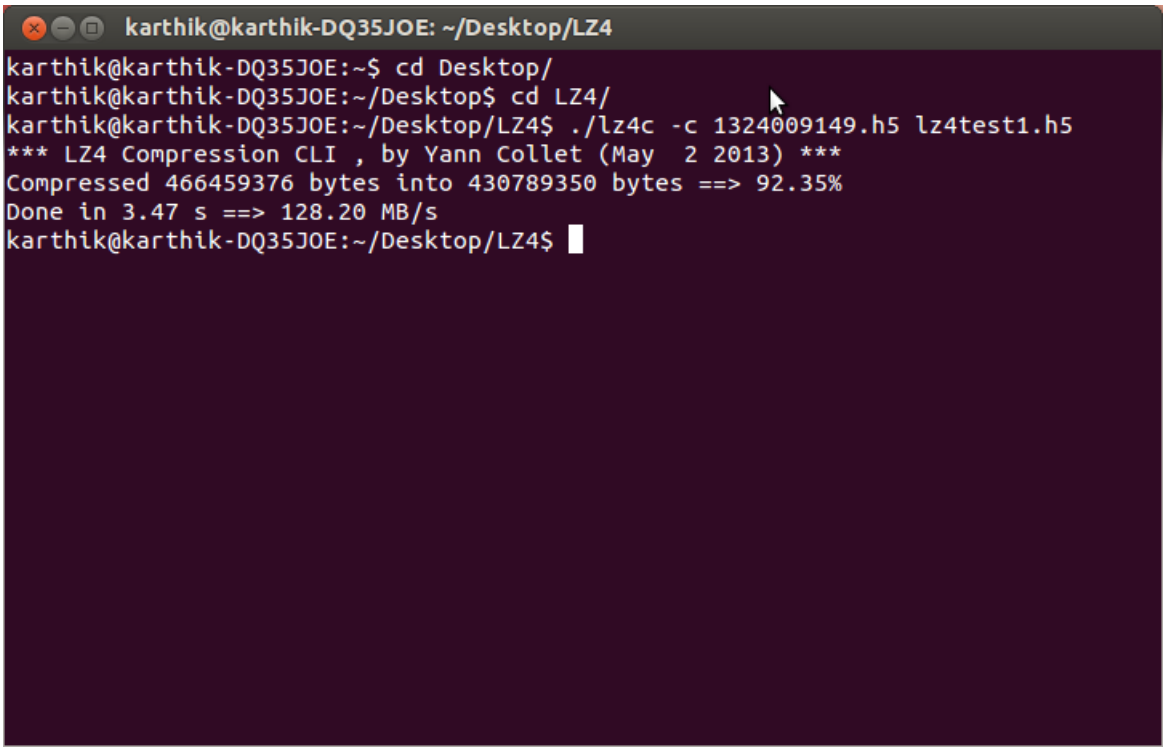

Figure 4.16: LZ4 test run

It displays the size of the compressed file as a percentage of the original file, while also showing the speed of compression.

4.4.4 FAPEC testing

Due to the commercial restrictions related to the FAPEC data compressor, as well as the unavailability of a fully-HDF5-integrated PEC version ready for float values, this study has only tested a preliminary compression option in the integrated environment (whereas the float-based FAPEC compression has been tested separately, in the Barcelona/DAPCOM environment).

Specifically, the 'SZIP' compression for the 'float' arrays has been activated, as has been previously demonstrated [11] that CCSDS 121.0 (i.e. SZIP) and PEC/FAPEC offer quite similar results - although FAPEC would perform significantly better in this case owing to its float-ready pre-processing stage [60].

Only datasets 4-8 were used while testing the FAPEC due to time restrictions, as they had to be sent to Dr. Jordi Portell in Barcelona.

The following command line excerpt shows the FAPEC being tested on file number 8 (as shown in table 4.1) using the FAPEC testing machine [61].

```
Testing FAPEC on the correlator data:

jportell@powertank /mnt/win/Dades/ska time ./fapec 1354329284-
correlatordata-First256MB.bin -m fp -32 -le -il 168 -bl 1024

Fully Adaptive Prediction Error Coder (FAPEC) v1.0 Beta

Loading file into memory...

File 1354329284-correlatordata-First256MB.bin loaded.
Size = 268435456 bytes
Launching PEC/FAPEC data compressor
Initializing FAPEC...
Starting data compression...
```

```
Input file size:  268435456 bytes
Number of samples processed:  134213632
Output size:  226710790 bytes
File compression ratio achieved:  1.1840 (InFileSize/OutFileSize)

Compression finished.
Output file:  1354329284-correlatordata-First256MB.bin.fapec

Done.

real 0m7.209s
user 0m6.070s
sys 0m0.227s
```

The overall compression speed was approximately 35.5MB/s, but it should be noted that the FAPEC implementation was not optimized. Specifically, this FAPEC implementation loaded the complete file to memory, instead of compressing that progressively. The operation of the FAPEC is being changed to a progressive loading + compressing + storing operation, which will be even quicker, while the core compressor is being optimized.

4.4.5 Streaming Compression

The aim of streaming compression is to compress a file while it is being transferred or accessed. This normally involves loading the file that is being read into memory and then applying the compression algorithm to the file. The effectiveness of this process relies heavily on the amount of RAM that is available and the size of the file that is being compressed.

Given the requirements for the project, two important factors had to be considered:

- The amount of time taken to compress the data while streaming
- The time taken to send the file (network throughput).

These two metrics are crucial to the process of streaming compression as the trade-off between the time taken to compress the file and the time taken to send it would determine the effectiveness of streaming compression.

As a result, the following equations were established:

$$T_o = S_o \tag{4.1}$$

$$T_c = C + S_c \tag{4.2}$$

Where:

- T_o is the total time taken to transfer the original file
- S_o is the time taken to stream the original file
- T_c is the total time taken to transfer the compressed file
- C is the time taken to the compress the file
- S_c is the time taken to stream the compressed file

For streaming compression to be effective, T_{sc} would always have to be less than T_o.

SKA Compress

In order to explore the feasibility of streaming compression, a program was written which would perform the tasks shown in the following image. The program was named 'SKA Compress'. This was the algorithm which was used to design the program.

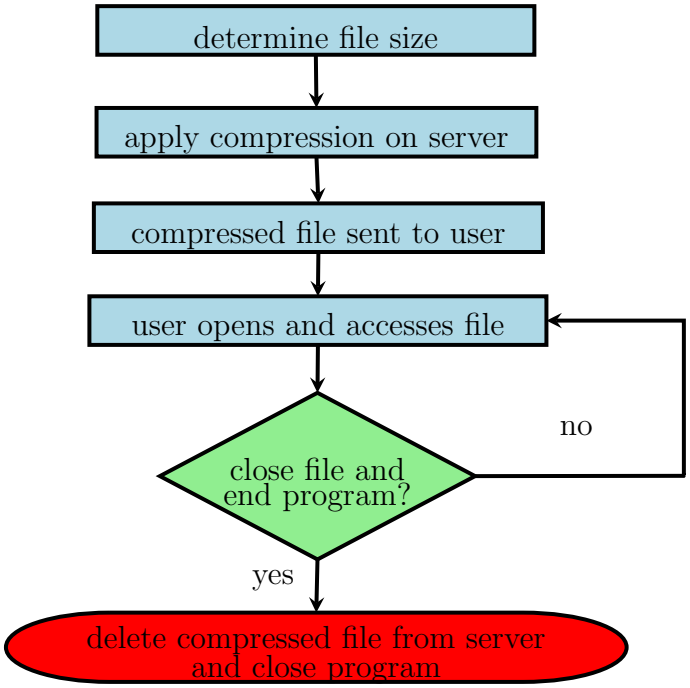

Figure 4.17: SKA Compress Flowchart

The threshold of the file size would need to be set in the program depending on which compression algorithm was being used and the available network speed.

For example, if it took 30 seconds to transfer File A, and a total time of 40 seconds to compress and then transfer the compressed version of File A, the program would not compress the file and simply transfer. However, if it took 60 seconds to transfer a larger file (File B), and 50 seconds to compress and transfer the compressed version of File B, then the program would go ahead and compress the file and send it to the user.

Based on the results obtained from the initial phases of testing that were carried out (as shown in chapter 5.1 and 5.2), the LZ4 algorithm was chosen to serve as the basis for the program as it provided good compression ratios but more crucially, in drastically quicker times than the other algorithms.

The program was designed to take in the file that was to be opened as an input parameter. It would then compress the file, creating a compressed file named 'tempfile', which would then be sent to the user. The first version of the program simply compressed and transferred the file, while the second version would allow the user to open and access the compressed file. Once the users have finished accessing the file, they could close the program, upon which the temporary file would be deleted. Although the program would not strictly be compressing the file while it was being streamed to the user, the intention was for the program to operate so quickly that it would give the impression that stream-only compression was being achieved.

The following image shows an initial version of the running program.

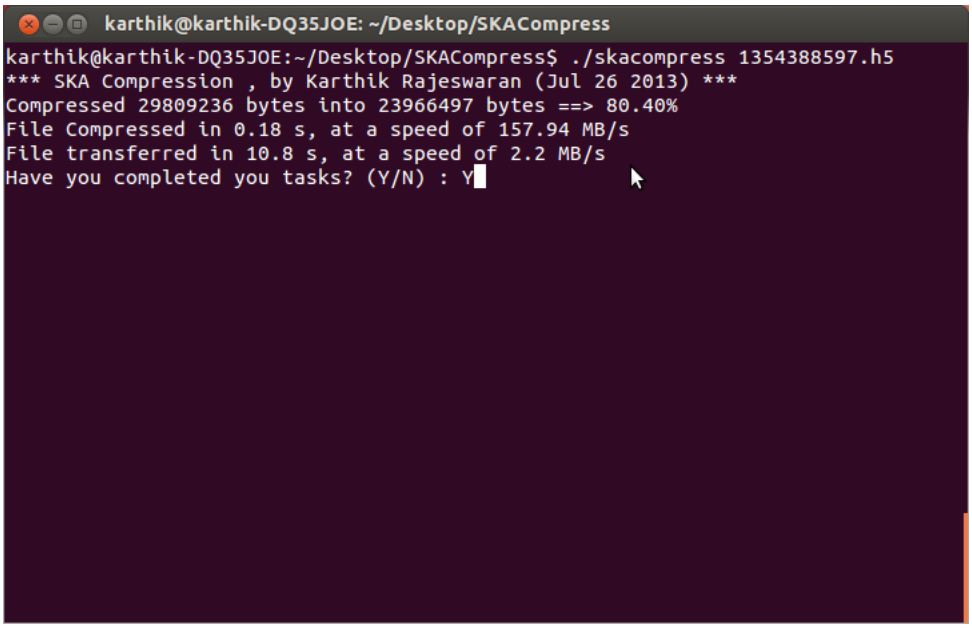

Figure 4.18: Initial version of SKA Compress

SKA Compress with split files

An intermediary step between the initial version of SKA Compress and the stream-only version was attempted. This was done by splitting the datasets into equally sized segments, compressing the individual segments and then transferring them to the user.

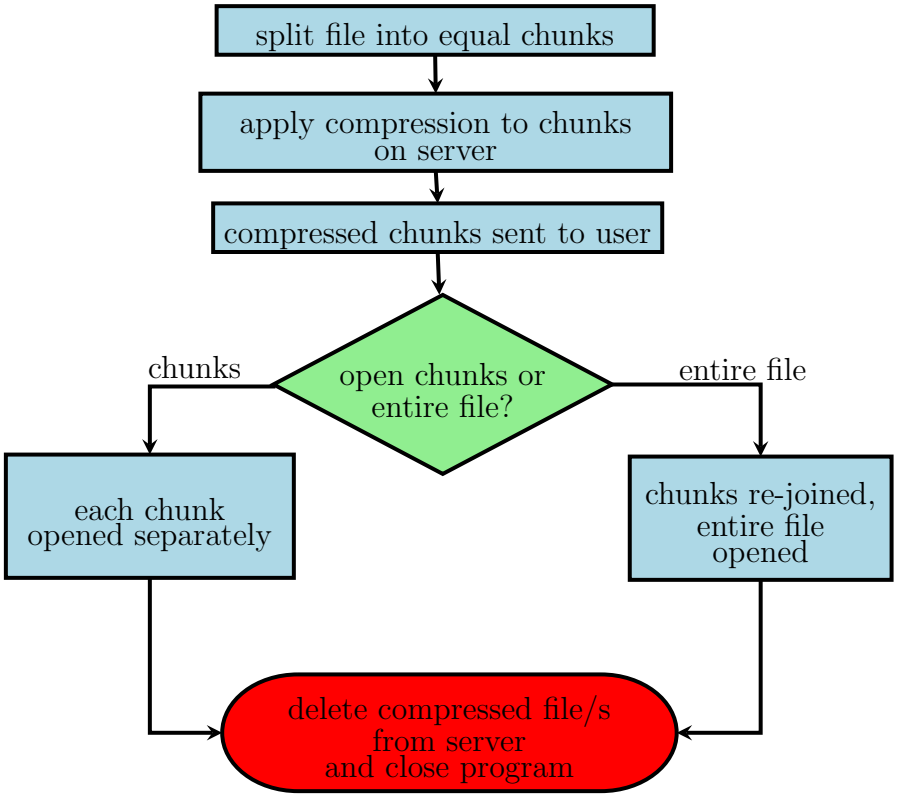

Figure 4.19: SKA Compress with split files Flowchart

The user would then have the option of choosing whether to open each individual chunk or whether to re-join all of the chunks and access the file as a whole.

Stream-only compression

The final step was to implement stream-only compression i.e. applying the compression only while the file was being streamed from the primary testbed to the secondary testbed, neither before nor after.

The LZ4 algorithm implemented a function to carry out such a process, which was modified slightly to improve it's performance. However, it was not tailored to suit the specific content and format of the files. Thus, it carried out a more generic approach towards the stream-only compression.

The following images show important lines of code for the function.

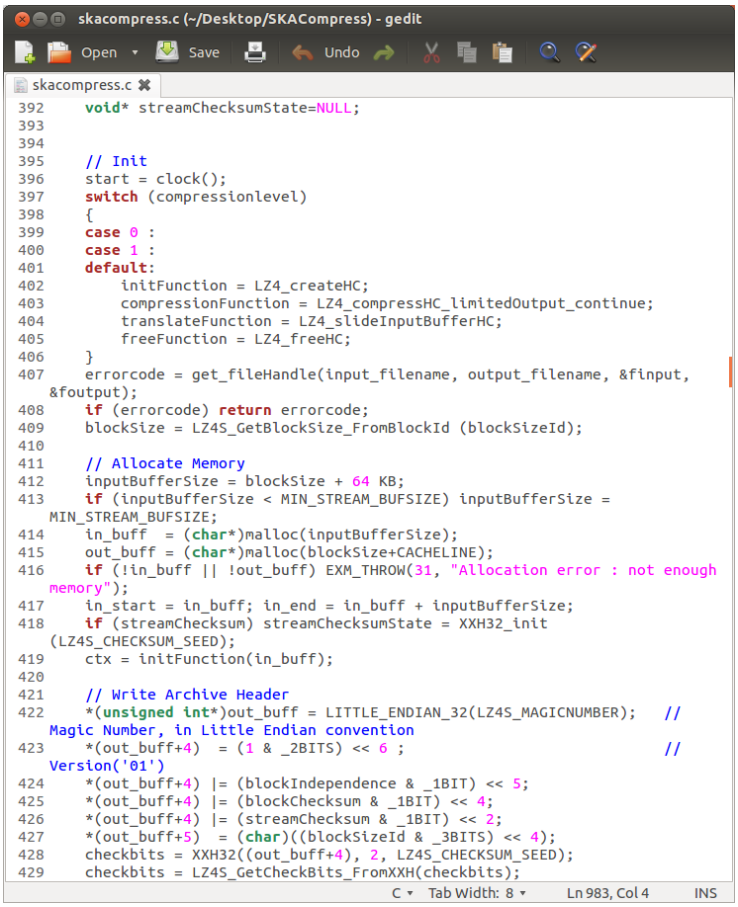

Figure 4.20: Stream-only compression function Part 1

Lines 412 - 419 on the previous image show how the memory buffers are initialized in order to begin the transfer of data.

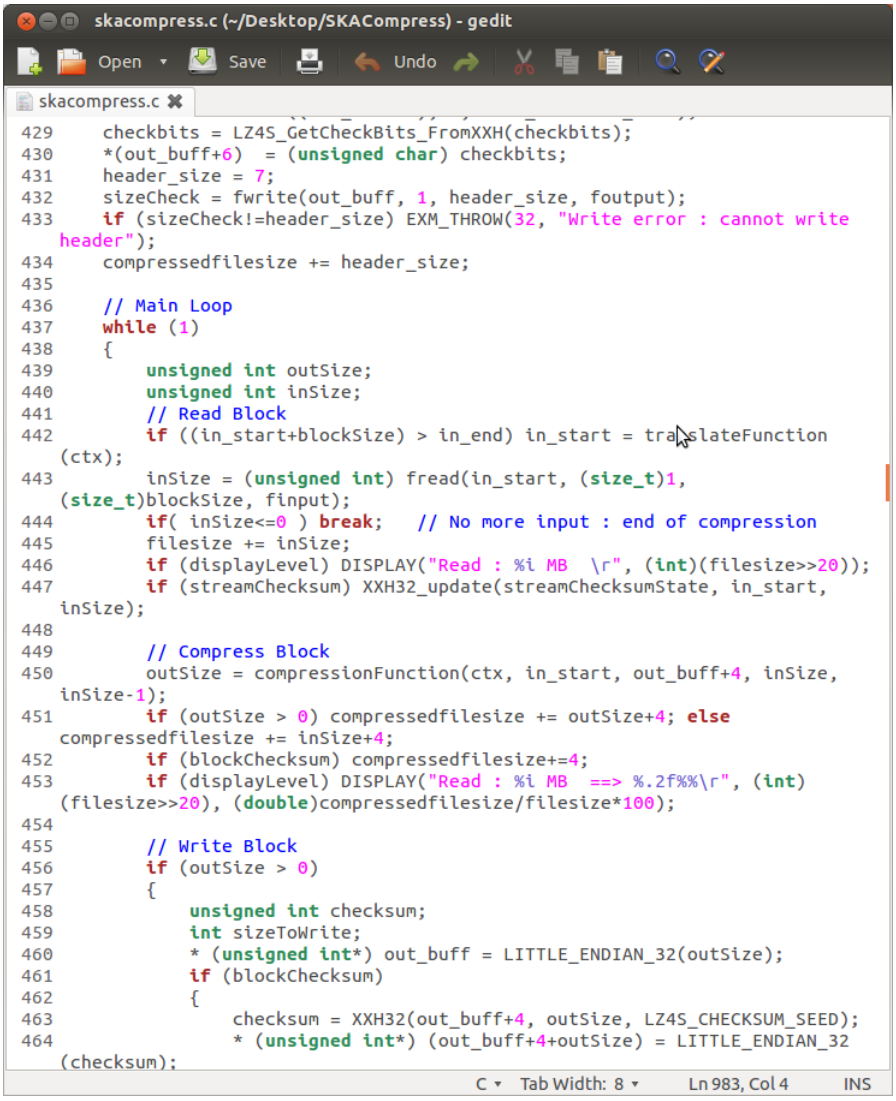

Figure 4.21: Stream-only compression function Part 2

The main loop begins on line 437 and shows how each block that is being streamed is read in, compressed and then written to the output.

Lines 489 - 501 show how the program closes the stream and determines if the compressed file was written successfully or not.

skacompress.c (~/Desktop/SKACompress) - gedit

Open Save Undo

skacompress.c

```
468             if (sizeCheck!=(size_t)(sizeToWrite)) EXM_THROW(33, "Write
    error : cannot write compressed block");
469
470         }
471         else   // Copy Original
472         {
473             unsigned int checksum;
474             * (unsigned int*) out_buff = LITTLE_ENDIAN_32(inSize|
    0x80000000);   // Add Uncompressed flag
475             sizeCheck = fwrite(out_buff, 1, 4, foutput);
476             if (sizeCheck!=(size_t)(4)) EXM_THROW(34, "Write error : cannot
    write block header");
477             sizeCheck = fwrite(in_start, 1, inSize, foutput);
478             if (sizeCheck!=(size_t)(inSize)) EXM_THROW(35, "Write error :
    cannot write block");
479             if (blockChecksum)
480             {
481                 checksum = XXH32(in_start, inSize, LZ4S_CHECKSUM_SEED);
482                 * (unsigned int*) out_buff = LITTLE_ENDIAN_32(checksum);
483                 sizeCheck = fwrite(out_buff, 1, 4, foutput);
484                 if (sizeCheck!=(size_t)(4)) EXM_THROW(36, "Write error :
    cannot write block checksum");
485             }
486         }
487         in_start += inSize;
488     }
489
490     // End of Stream mark
491     * (unsigned int*) out_buff = LZ4S_EOS;
492     sizeCheck = fwrite(out_buff, 1, 4, foutput);
493     if (sizeCheck!=(size_t)(4)) EXM_THROW(37, "Write error : cannot write
    end of stream");
494     compressedfilesize += 4;
495     if (streamChecksum)
496     {
497         unsigned int checksum = XXH32_digest(streamChecksumState);
498         * (unsigned int*) out_buff = LITTLE_ENDIAN_32(checksum);
499         sizeCheck = fwrite(out_buff, 1, 4, foutput);
500         if (sizeCheck!=(size_t)(4)) EXM_THROW(37, "Write error : cannot
    write stream checksum");
501         compressedfilesize += 4;
502     }
503
504     // Status
```

C Tab Width: 8 Ln 983, Col 4 INS

Figure 4.22: Stream-only compression function Part 3

Chapter 5

Results and Evaluation

This chapter presents and evaluates the results obtained from the three phases of testing that were carried out. These were:

1. Initial performance testing
2. Final performance testing
3. Streaming compression testing

Each phase of testing served to refine and improve the researcher's knowledge of the performance of the algorithms, and aided in the design process of the stream-based compression system. The results are presented in a series of tables, which are then summarised using graphs and charts, followed by an evaluation of the results.

5.1 Initial Performance Testing

As was mentioned in the previous chapter, an inital set of testing was carried out on a few, smaller data sets to compare the performances of the different algorithms. These were files 1-6 as shown in table 4.1, ranging from 29MB to 466.5 MB in size.

This testing helped to give an indication of the algorithms' compatibility with the specific arrangement and structure of these files, so that the best performing algorithms could be tested on the larger files.

These results are shown in the tables and graphs on the following pages.

It should be noted that the the best results for GZIP were obtained using level 6 compression, while the best results for SZIP were obtained by setting the blocks per pixel parameter at 16, using the NN coding method. Thus, only the results obtained using those settings are shown.

The percentages indicated in the memory usage column reflect the percentage of CPU occupation, regardless of the physical memory being used.

Table 5.1: Initial Performance Testing: GZIP-6

File Number	Compression Ratio	Compression Time (s)	Memory Usage (MB)
1	1.419	2.46	16.568 (32%)
2	1.444	3.10	36.692 (49%)
3	1.571	6.62	81.802 (50%)
4	1.491	7.87	88.376 (49%)
5	1.479	155.8	409.400 (50%)
6	1.381	205.22	447.620 (50%)

Table 5.2: Initial Performance Testing: SZIP-16 NN

File Number	Compression Ratio	Compression Time (s)	Memory Usage (MB)
1	1.546	1.21	18.835 (27%)
2	1.532	1.84	37.836 (50%)
3	1.529	3.78	78.621 (50%)
4	1.516	4.66	90.593 (50%)
5	1.485	139.02	402.749 (50%)
6	1.495	187.8	451.115 (50%)

Table 5.3: Initial Performance Testing: LZF

File Number	Compression Ratio	Compression Time (s)
1	1.114	0.42
2	1.105	0.57
3	1.102	1.24
4	1.092	1.41
5	1.080	33.93
6	1.075	49.03

Table 5.4: Initial Performance Testing: FAPEC

File Number	Compression Ratio	Compression Time (s)
4	1.154	1.15
5	1.148	6.3
6	1.088	9.7

Table 5.5: Initial Performance Testing: LZ4

File Number	Compression Ratio	Compression Time (s)	Memory Usage (MB)
1	1.208	0.18	0.5
2	1.240	0.26	0.6
3	1.267	0.46	0.9
4	1.242	0.48	1.3
5	1.224	2.87	7.5 (8%)
6	1.083	3.72	8 (11%)

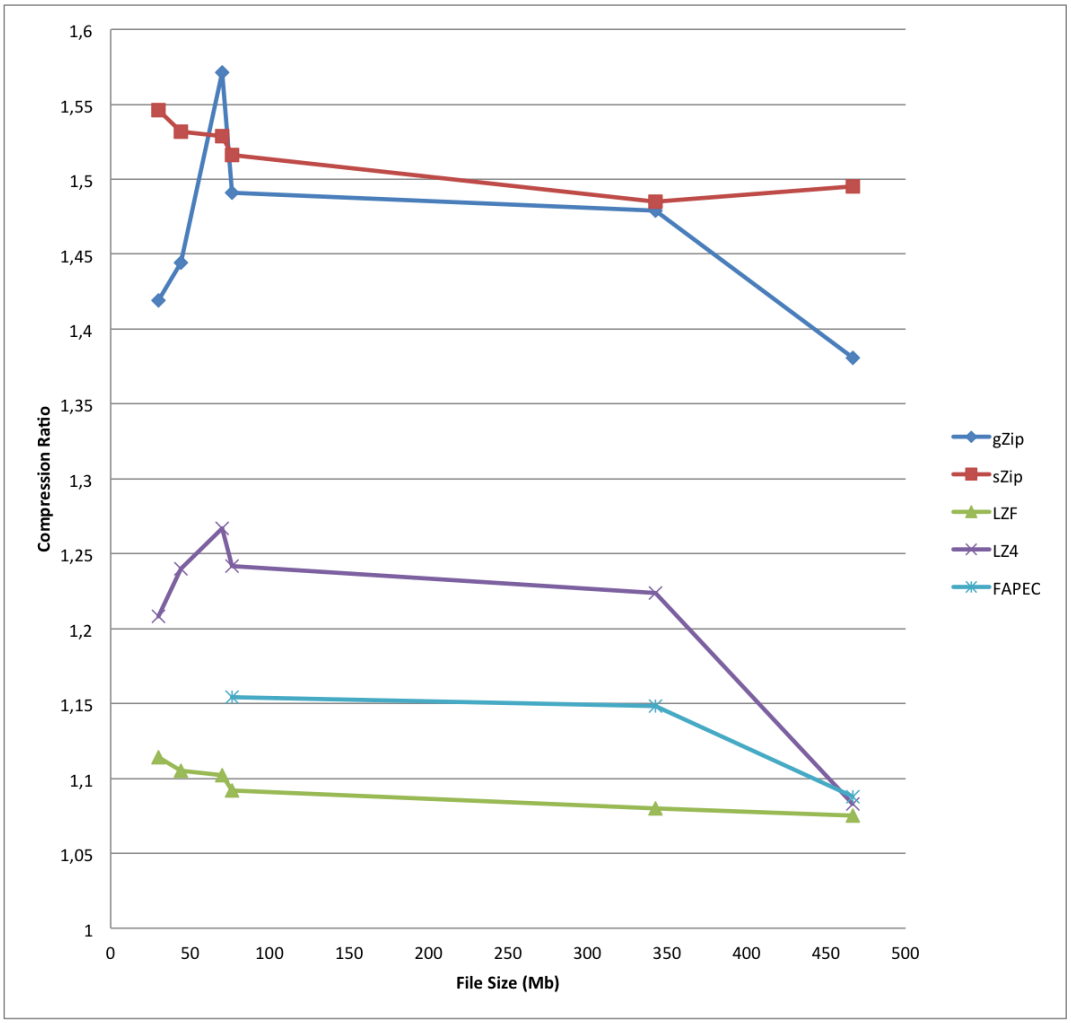

Figure 5.1: Compression ratios for initial performance testing

The compression ratios, with the odd exception, gradually decreased as the file sizes increased, which is to be expected [15]. SZIP and GZIP provided the highest ratios, ranging between 1.4-1.6, while the FAPEC and LZ4 provided lower but consistent ratios. The LZF filter provided the lowest ratios. The relative compression ratios for SZIP, GZIP and LZF were in line with those that were found in the studies conducted by Yeh et. al [47] and Collette [51]. This was reflected further in the compression times.

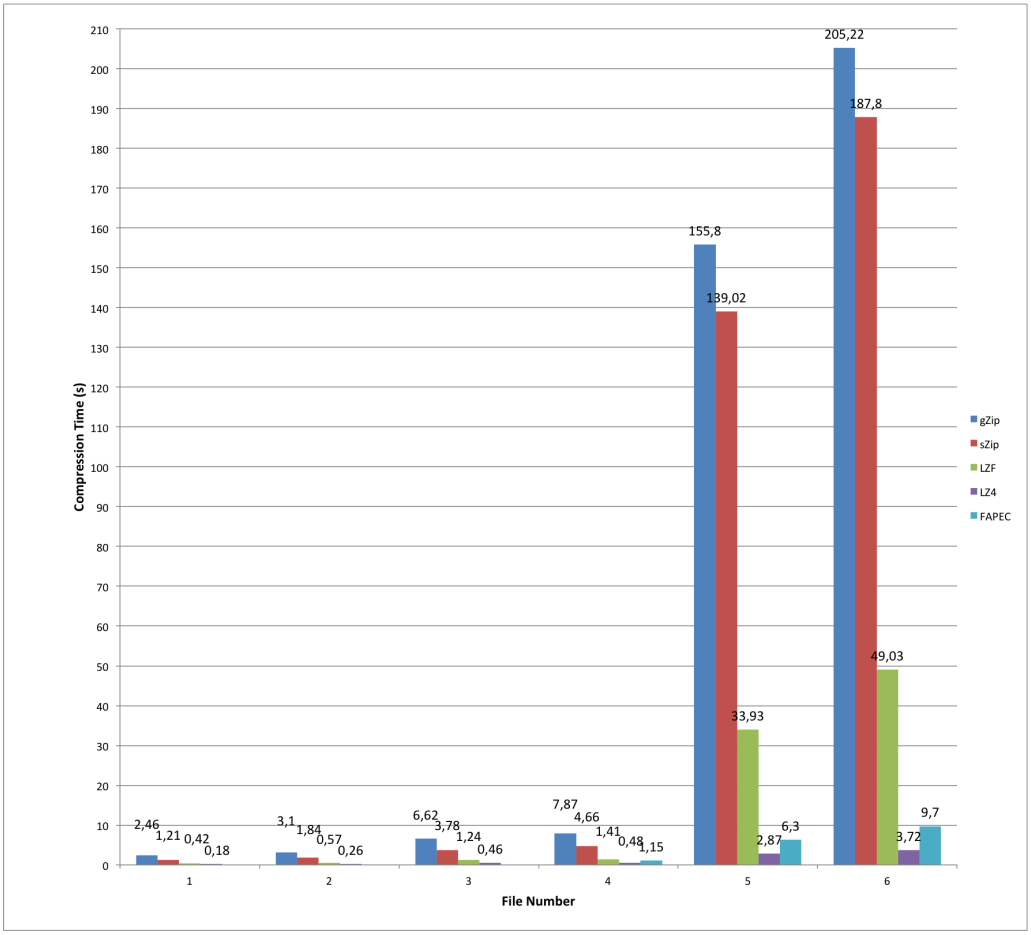

Figure 5.2: Compression times for initial performance testing

Figure 5.2 compares the compression times for the five algorithms, showing how long they took, relative to each other, to compress each file. The FAPEC and LZ4 were drastically faster than the other three algorithms. The LZ4 filter was also relatively quick while SZIP took longer, with GZIP being by far the slowest.

The memory usage could only be monitored for three of the algorithms, as shown in tables 5.1 - 5.5. Of those three, SZIP and GZIP placed an extremely high load on the system, using 50 percent of the available memory for all of the files except the smallest one. In comparison, LZ4 used negligible memory for files 1-4, while it increased to 8 and 11 percent respectively for files 5 and 6, which was a very acceptable amount.

5.2 Final Performance Testing

The best three compression algorithms were selected from the initial testing in this stage. Based on the three key metrics, it was intended that the algorithms which provided the best ratio, quickest compression times and had the least memory usage would be selected. However, LZ4 provided the quickest compression speeds as well as the least memory usage, thus the FAPEC was chosen, as its ratio and timing results were very similar to those of LZ4. the final algorithm that was selected was SZIP, which provided the highest compression ratios.

Datsets 7-11 were used for LZ4 and SZIP, but only 7 and 8 could be used for the FAPEC due to time restrictions in sending the datasets to the researchers at the University of Barcelona.

Table 5.6: Final Performance Testing: FAPEC

File Number	Compression Ratio	Compression Time (s)
7	1.153	52.8
8	1.182	63.6

Table 5.7: Final Performance Testing: LZ4

File Number	Compression Ratio	Compression Time (s)	Memory Usage (MB)
7	1.231	16.53	7.8 (11 %)
8	1.234	24.17	7.6 (11 %)
9	1.228	33.12	7.5 (9 %)
10	1.216	39.01	7.5 (8 %)
11	1.153	72.74	8.6 (15 %)

Table 5.8: Final Performance Testing: SZIP-16 NN

File Number	Compression Ratio	Compression Time (s)	Memory Usage (MB)
7	1.484	673.5	462.320 (50 %)
8	1.513	848.2	462.840 (50 %)
9	1.346	1187.3	462.620 (50 %)
10	1.215	1534.7	462.582 (50 %)
11	1.185	2093.2	462.740 (50 %)

The graph below compares the compression ratios obtained from the final performance testing. As was with the inital stage, the FAPEC and LZ4 provided steady and similar ratios, ranging between 1.15 and 1.25. SZIP initially provided high ratios close to 1.5 for files 7 and 8, but its performance drastically declined on the three files greater than 4GB in size, reaching a similar level to that of LZ4.

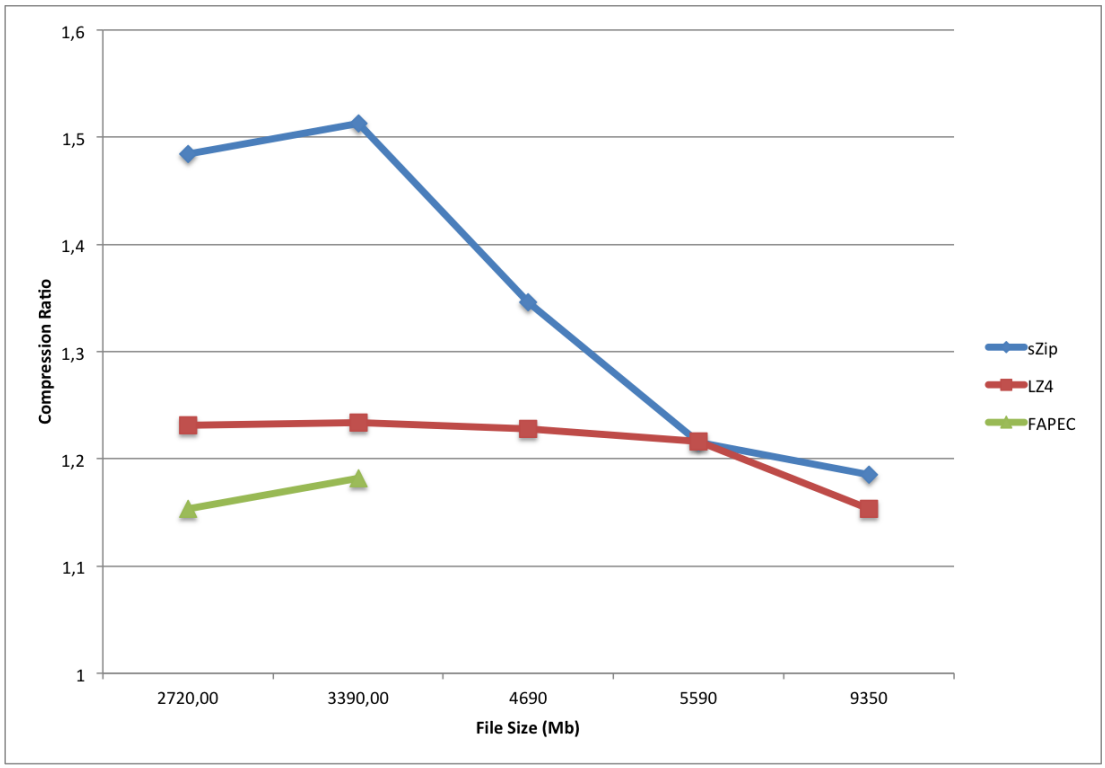

Figure 5.3: Compression ratios for final performance testing

The memory usage for both SZIP and LZ4 did not differ much from the initial performance testing, with SZIP occupying 50% of the system memory and slowing the computer down drastically, while LZ4 occupied a maximum of 15%.

The chart below compares the compression times obtained from the final performance testing. SZIP was considerably slower than the FAPEC and LZ4. In the most extreme case SZIP took almost 34 minutes longer than LZ4 to compress dataset 11. LZ4 was extremely quick, with the FAPEC performing slightly slower.

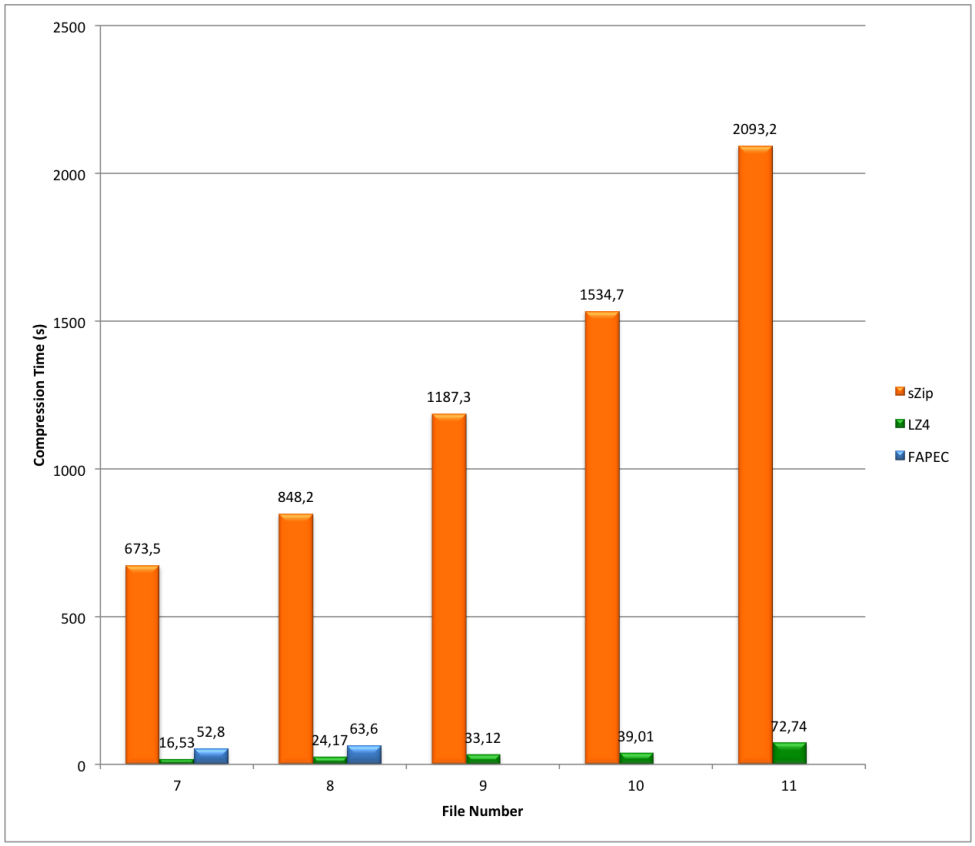

Figure 5.4: Compression times for final performance testing

The results from this section clearly showed that LZ4 provided the best overall performances for the given data sets. It performed considerably faster than the other two algorithms, as well as being memory efficient and providing ratios greater than 1.1, as specified in the constraints (Section 3.1.2). As a result, it was chosen as the optimal algorithm to develop the stream-based system. The FAPEC provided promising results which were likely to be close to those achieved by LZ4.

5.3 Streaming compression Testing

The results in this section are split into two sub-sections, those from the initial stage of the SKA Compress program, and those when stream-only compression was integrated and attempted.

5.3.1 SKA Compress

The table on the following page shows the results that were obtained from running the SKA Compress program, comparing the the critical parameters shown below (as described in section 4.4.5): Where:

- T_o - the total time taken to transfer the original file
- S_o - the time taken to stream the original file
- T_c - the total time taken to transfer the compressed file
- C - the time taken to the compress the file
- S_c - the time taken to stream the compressed file

The results obtained from splitting the files with the SKA Compress came out to be exactly the same as the results without splitting them, thus they have been omitted.

The transfer speed from the primary testbed to the secondary testbed (Fig 4.2 and 4.3) averaged between 2.5 MB/s and 2.8 MB/s.

It should be noted that during the first run of the program on dataset 11, a compression ratio of 1.543 was obtained in 3042.28 seconds. However, this compressed file was not identical to the original and some of the data had been corrupted. As a result, the program was run again and the successfull result is shown in the table.

Table 5.9: SKA Compress Testing

File Number	Compression Ratio	T_o (s)	C (s)	S_c (s)	T_c (s)
1	1.208	13.50	0.18	10.9	11.18
2	1.240	17.56	0.26	14.16	14.42
3	1.267	23.92	0.46	21.97	22.43
4	1.242	29.19	0.48	23.50	23.98
5	1.224	127.76	2.87	107.59	110.46
6	1.083	162.6	3.72	165.67	169.39
7	1.231	1088.00	16.53	849.84	866.37
8	1.234	1356.00	24.17	1098.86	1113.03
9	1.228	1876.00	33.12	1527.68	1560.80
10	1.216	2150.00	39.01	1838.81	1877.82
11	1.153	3596.15	72.74	3243.71	3316.43

The following two graphs show the time taken to transfer the original, uncompressed files and the compressed files. The first graph shows the results for dataset 1-6, and the second graph shows the results from dataset 7-11.

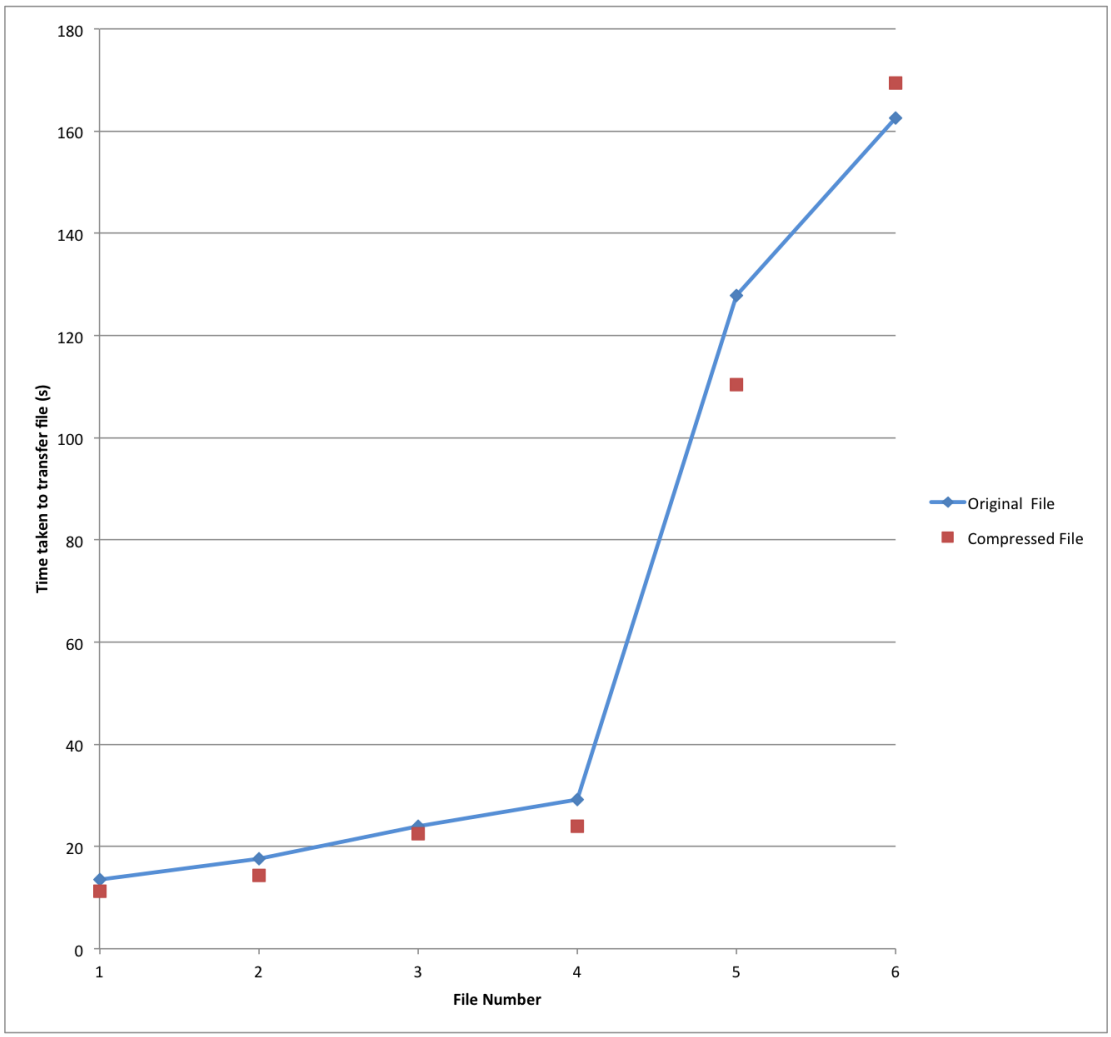

Figure 5.5: Comparing the streaming compression times for datasets 1-6

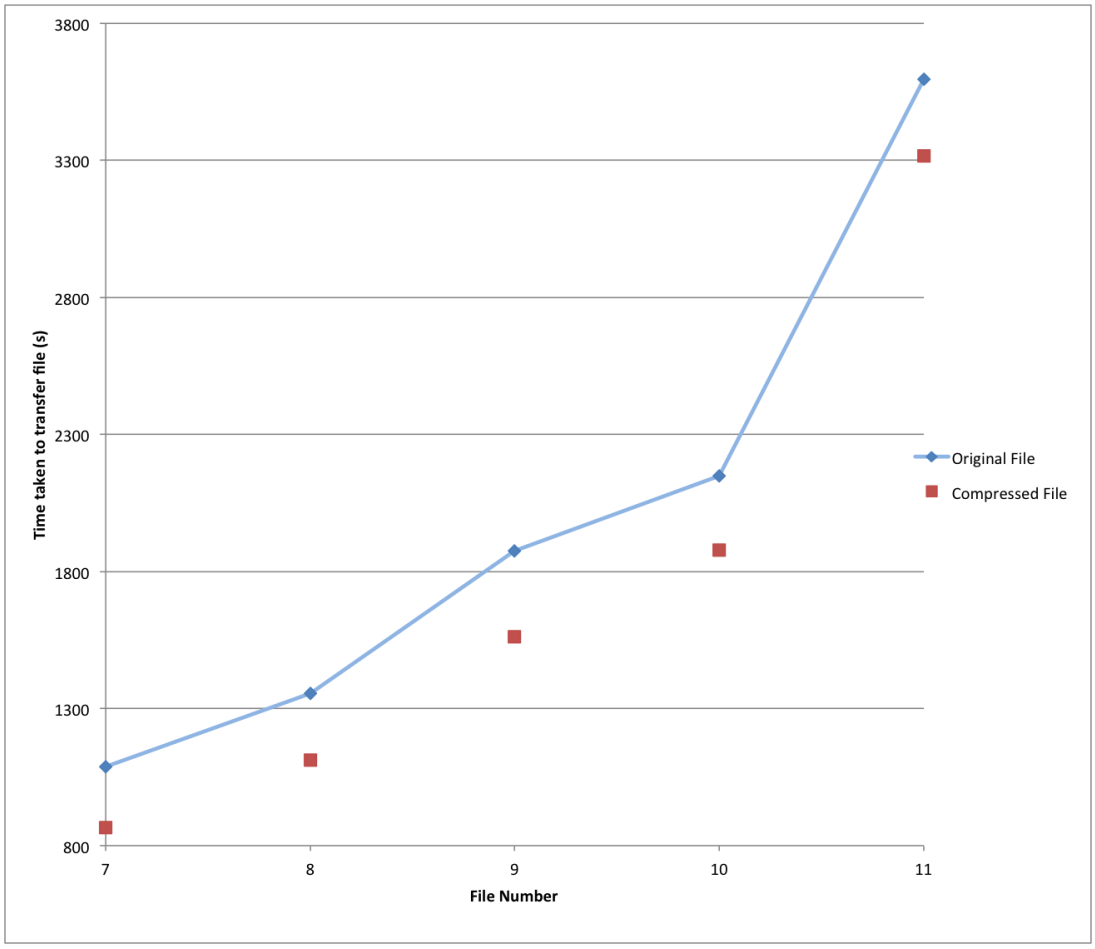

Figure 5.6: Comparing the streaming compression times for datasets 7-11

As was mentioned in section 4.4.5 and using equations 4.1 and 4.2, for streaming compression to be effective, T_c needed to be less than T_o. The two graphs show that T_c was less than T_o for all of the datasets except number 6, where a lower compression ratio of 1.083 was obtained. The difference in time between T_o and T_c increased as the files got larger in size, while the difference was small (less than 5 seconds on average) with files smaller than 100MB.

This showed that SKA Compress performed successfully and met the objective of providing results where T_c was consistently less than T_o.

5.3.2 Stream-only compression

The table below shows the results that were obtained from carrying out stream-only compression, as was described in the last section of chapter 4.4.5. It compares T_o and S_c, as well as indicating whether the compressed file that reached the user was read-able or not.

Table 5.10: Stream-only Compression Results

File Number	Compression Ratio	T_o (s)	S_c (s)	Output file readable
1	1.217	13.50	10.87	Yes
2	1.214	17.56	12.52	Yes
3	1.207	23.92	22.18	Yes
4	1.202	29.19	35.95	Yes
5	1.128	127.76	153.16	Yes
6	1.116	162.6	202.38	Yes
7	1.065	1088.00	1208.64	No
8	1.039	1356.00	1720.86	No
9	1.018	1876.00	2361.01	No
10	1.014	2150.00	2639.38	No
11	1.003	3596.15	4324.40	No

The graph below compares the key metrics shown in table 5.10.

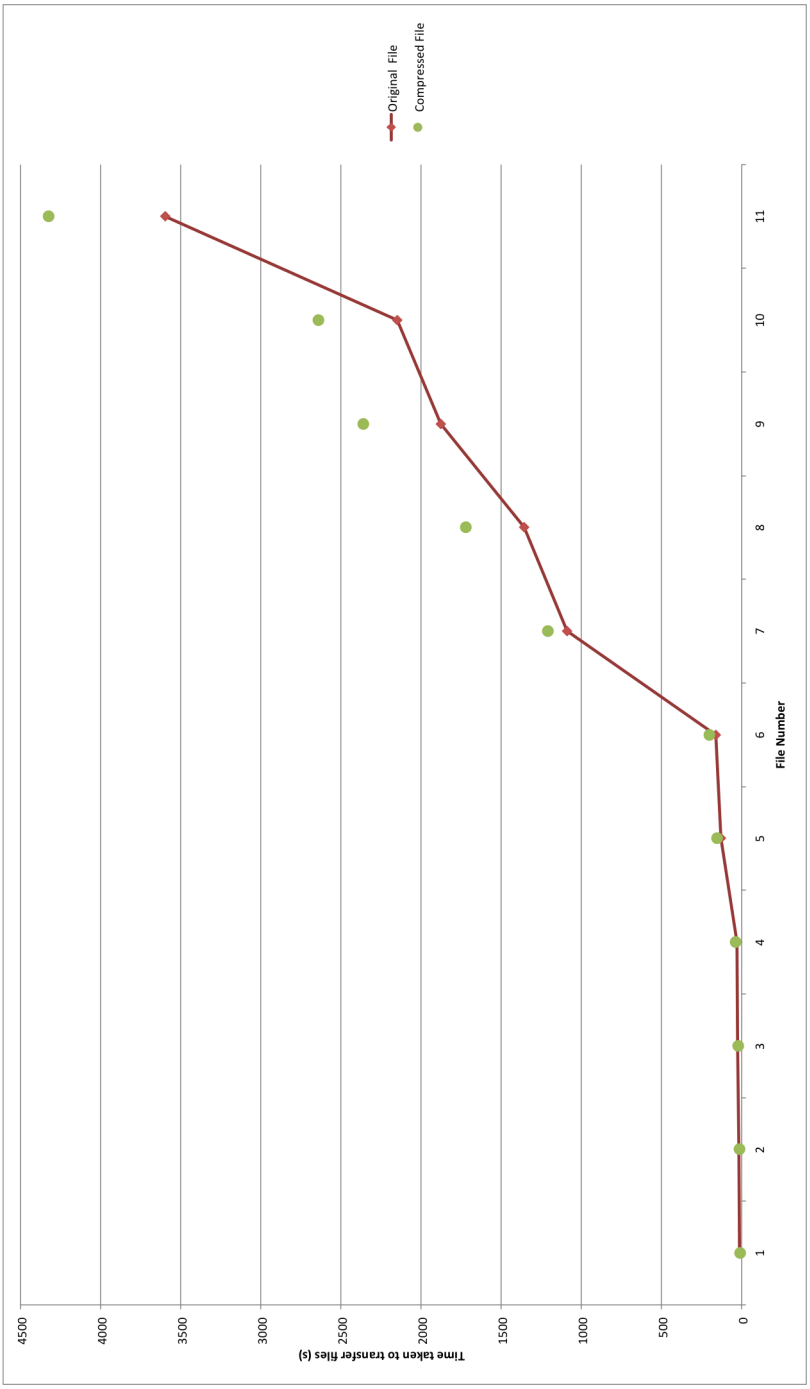

Figure 5.7: Comparing the compression times between T_o and S_c

As shown in table 5.10 and figure 5.7, the time taken to compress and stream the files generally took much longer than it would to simply transfer the original, uncompressed files. The stream-only compression program performed in a time efficient manner for the first three files. For every file after that, the differences in the time taken were progressively longer.

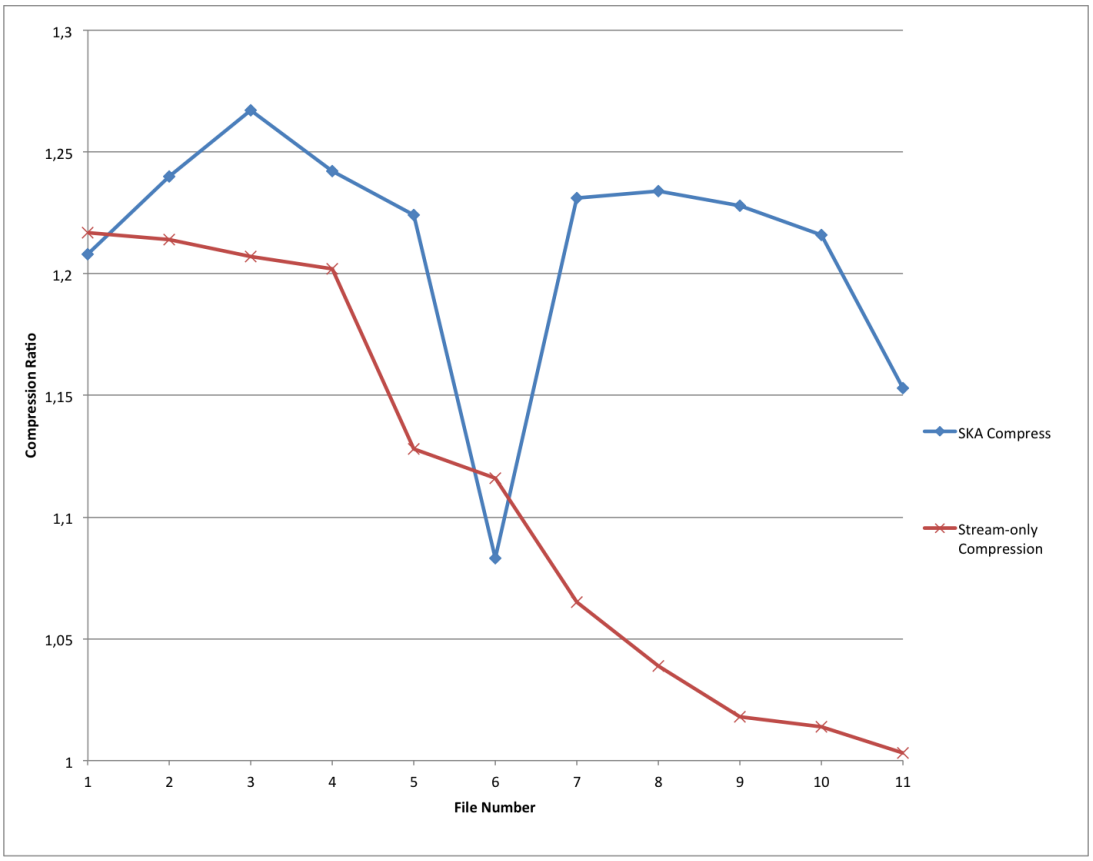

Figure 5.8: Comparing the compression ratios between SKA Compress and stream-only compression

The compression ratios obtained through stream-only compression were comparable to those from SKA Compress for the first four files. However, there was a drastic decrease for every file after that. The compression ratio dropped to a very low level, such that the ratio obtained for file 11 was entirely negligible, with the compressed file only 003% smaller than the original. This can be attributed to the larger block sizes that had to be sent through the memory buffer.

The memory usage, when carrying out stream-only compression, was extremely high. It's percentage of CPU usage would always be at 50%, but the actual amount of memory fluctuated rapidly, thus definitive results could not be obtained.

The higher memory usage was expected, given that the blocks that would be loaded into the streaming buffer would be greater in size. The fluctuations in memory usage were also expected, as different parts of the file with different content would be compressed uniquely.

One of the major problems that came up from stream-only compression was that the compressed versions of files 7-11 could not be read and were corrupted. Given that file 7 is 2.7GB in size, it was suspected that the reason for those outcomes was a result of the header of the files and the end of the files not reaching the user in the same session. It was observed that when the memory usage became extremely high (greater than 50%), the disk would be described as being in a 'sleeping' state and thus the process would be paused for a short period time, before resuming. In that period, it is likely that the files greater in size than half of the available RAM (which would be 1GB) caused the disk to enter the 'sleeping' state, thus not concatenating the header with the end of the file.

As a result, although the entire compressed files would reach the user, they were not in the exact same format and structure as the original, causing them to be corrupted.

Chapter 6

Conclusions and Recommendations

This chapter discusses the results that were presented in chapter 5, and provides conclusions based on those results. It then provides recommendations on how to improve this system and discusses future work that could be done as a follow up to the topic covered in this report and other similar topics.

6.1 Summary

This dissertation has described the design, implementation and evaluation of a select group of compression algorithms on data sets collected by the SKA. By testing well known compression algorithms as well as novel ones, a thorough study was carried out, which provided a wide range of relevant results.

The results in section 5.1 have shown that the most commonly known compression algorithms for HDF5, which are GZIP, SZIP and LZF, provided results which were consistent with previous studies that were conducted. Their specific performances were affected by the large sizes and complex contents of the datasets, as SZIP and GZIP normally provide compression ratios greater than 2.

However, relative to each other, they followed the expected pattern, with SZIP and GZIP providing much higher compression ratios with slow times, while LZF was the opposite, with very low ratios but siginificantly faster compression times.

The FAPEC and LZ4 provided the best overall results. As was mentioned in section 3.1.2, compression ratios above 1.1 would be considered as a great result. Both algorithms achieved this while being significantly faster and using less memory than the other three. The commercial restrictions of the FAPEC did not allow for more rigorous testing, however the nature of the algorithm and the results it provided were very promising. The LZ4 results provided the best performance when considering all three metrics (section 3.4) and was successfully used to develop the SKA Compress program.

The results using SKA Compress were excellent and met most of the criteria detailed in the requirements for the project. They showed that it was possible to run a program which eased the load on the available computing resources (storage space and memory) and allowed users to access the datasets in a time efficient manner. It was also evident that certain thresholds would need to be set within the program based on the network speed, in order to maximize its performance.

The outcome of attempting stream-only compression was not entirely successfull, as it was extremely memory intensive and regularly created corrupted datasets which could not be read by the end user. However, the few successfull attempts indicate that it is an aspect which needs to be worked on and could provide hugely beneficial outcomes if improved upon.

Referring back to the objectives listed in Chapter 1, the outcomes of the study can be summarised as shown below:

1. Five different algorithms were rigorously tested, which laid a strong foundation for further investigation.

2. The FAPEC and LZ4 were found to provide the best performances of the algorithms that were tested on the given datasets.

3. Streaming compression was implemented in two stages. The initial stage compressed the file on the server and sent it to the user, who could open the file and access it. The compressed file would be deleted from the server once the user closed the stream, while the second stage implemented stream-only compression.

4. The testing environment was similar in structure to that of the SKA. The OS and software used were identical, with the major differences being in the hardware of the computers that were used. The likely difference in results that would be obtained by the more powerful SKA computers is that stream-based compression would be much faster and less memory intensive. The SKA Compress program can function as a stand-alone application or it could be assimilated into the software environment of the SKA if necessary.

As per the results mentioned above, this dissertation has met the objectives stated in Chaper 1. The first two objectives were met by investigating five algorithms, and determining those which provided the best performances through rigorous testing. The implementation of streaming compression on a system which simulated the SKA environment, ensured that objectives three and four were met.

The promising results as presented suggest a likely potential benefit for further investigation of the the performance of the FAPEC and LZ4 compression algorithms in terms of their application to the SKA data sets and streaming compression.

6.2 Recommendations for further work

The following list of recommendations are intended to provide a guideline to build upon the work carried out in this study, in order to improve certain facets and possibly discover new avenues in the future.

1. **Test on greater number of datasets** - While this study managed to cover a wide range of datasets, more accurate results could be obtained by testing a greater number of files within the same range. This would provide clearer indications on how to optimize parameters in the compression programs, particularly with regards to pure streaming compression.

2. **Refinement of LZ4 to be more specific to the datasets** - The LZ4 algorithm provided excellent results in all three key metrics. However, the algorithm was only slightly modified and was run without considering the format or contents of the files. Tailoring the performance of the LZ4 algorithm to the HDF5 format and treating dataspaces within the datasets based on the nature of the data, could yield much better results. It could also eradicate the problems that occured when attempting pure streaming compression.

3. **Obtain final version of FAPEC** - The FAPEC provided very similar results to LZ4, but was slightly slower. However, the nature of the FAPEC means that it is designed to progressively compress the HDF5 files as they are being transferred/streamed, and that it is fully adaptive. This would indicate that the final commercial version of the FAPEC could provide even better and more specifically applicable compression. In future, it is likely to become the standard compression technique for the large data sets stored in the HDF5 format.

The researcher hopes that the foundations laid by the work in this project will be used and developed further in the future to produce a more robust and efficient data compression system for the SKA.

Bibliography

[1] L.-L. Stark and F. Murtagh, Handbook of Astronomical Data Analysis. Springer-Verlag, 2002.

[2] W. D. Pence, R. Seaman, and R. L. White, "Lossless astronomical image compression and the effects of noise," Publications of the Astronomical Society of the Pacific, vol. 121, pp. 414 – 427, March 2009.

[3] "South africa's meerkat array." [ONLINE]: `http://www.ska.ac.za/download/fact_sheet_meerkat_2011.pdf.`, May 2012.

[4] R. Mittra, "Square kilometer arraya unique instrument for exploring the mysteries of the universe using the square kilometer array," IEEE Computer Society, September 2009.

[5] D. L. Jones, K. Wagstaff, D. R. Thompson, L. DAddario, R. Navarro, C. Mattmann, W. Majid, J. Lazio, R. Preston, and U. Rebbapragada, "Big data challenges for large radio arrays," 2011.

[6] J. Starck and F.Murtagh, "Astronomical image and signal processing," IEEE Signal Processing Magazine, 2002.

[7] S. de Rooij and P. M. Vitanyi, "Approximating rate-distortion graphs of individual data: Experiments in lossy compression and denoising," IEEE Transactions on Computers, vol. 23, pp. 14 – 15, March 2012.

[8] K. Rajeswaran, "Meeting - dept. of astronomy,university of cape town," June 2012.

[9] K. Rajeswaran, S. Ratcliffe, and J. Horrell, "Meeting - ska office, cape town," May 2012.

[10] "Hdf5 for python." [ONLINE]: `www.alvfen.org/wp/hdf-5for-python`, March 2013.

[11] C. E. Sanchez, "Feasibility study of the pec compressor in hdf5 file format," Master's thesis, Universitat Politecnica de Catalunya, 2011.

[12] R. F. Rice, P.-S. Yeh, and W. H. Miller, "The development of lossless data compression technology for remote sensing applications," in Geoscience and Remote Sensing Symposium,1994.

[13] K. Rajeswaran and J. Manley, "Meeting - ska office, cape town," April 2013.

[14] "From big bang to big data: Astron and ibm collaborate to explore origins of the universe." [ONLINE]: `http://www-03.ibm.com/press/us/en/pressrelease/37361.wss`, June 2012.

[15] J. Portell, E. GarciaBerroad, C. E. Sanchez, J. Castaneda, and M. Clotet, "Efficient data storage of astronomical data using hdf5 and pec compression," SPIE High-Performance Computing in Remote Sensing, vol. 1, April 2011.

[16] P. Hall, R. Schilizzi, P. Dewdney, and J. Lazio, "The square kilometre array (ska) radio telescope: Progress and technical directions," Radio Science Bulletin, vol. 326, September 2008.

[17] K. R. Anderson, A. Alexov, L. Bahren, J. M. Griemeier, M. Wise, and G. A. Renting, "Lofar and hdf5: Toward a new radio data standard," in SKAF2010 Science Meeting - ISKAF2010 June 10-14, 2010, June 2010.

[18] "Visuals and media." [ONLINE]: `http://www.ska.ac.za/media/meerkat.php`, February 2013.

[19] S. Finniss, "Using the fits data structure," Master's thesis, University of Cape Town, 2011.

[20] K. Borne, "Data science challenges from distributed petascale astronomical sky surveys," in DOE Conference on Mathematical Analysis of Petascale Data, 2008.

[21] B. B. C. I. of Technology, "Astronomy needs new data format standards." [ONLINE]: http://astrocompute.wordpress.com/2011/05/20/astronomy-needs-new-data-format-standards/, May 2011.

[22] "What is hdf5." [ONLINE]: www.hdfgroup.org/hdf5/whatishdf5.html, May 2013.

[23] "Ibm completes software research project for the square kilometre array global telescope." [ONLINE]: http://www-03.ibm.com/press/nz/en/pressrelease/36116.wss, June 2012.

[24] "The hdf5 data model and file structure." [ONLINE]: http://www.hdfgroup.org/HDF5/doc/UG/03_DataModel.html, May 2013.

[25] "The hdf5 file." [ONLINE]: http://www.hdfgroup.org/HDF5/doc/UG/08_TheFile.html, May 2013.

[26] "Hdf5 groups." [ONLINE]: http://www.hdfgroup.org/HDF5/doc/UG/09_Groups.html, May 2013.

[27] "Datasets." [ONLINE]: http://www.hdfgroup.org/HDF5/doc/UG/10_Datasets.html, May 2013.

[28] "Hdf5 dataspaces and partial i/o." [ONLINE]: http://www.hdfgroup.org/HDF5/doc/UG/UG_frame12Dataspaces.html, May 2013.

[29] "Hdf5 datatypes." [ONLINE]: http://www.hdfgroup.org/HDF5/doc/UG/UG_frame11Datatypes.html, May 2013.

[30] "Hdf5 tools." [ONLINE]: http://www.hdfgroup.org/products/hdf5_tools/, May 2013.

[31] D. A. Lelewer and D. S. Hirschberg, "Data compression," ACM Computing Surveys, vol. 19, pp. 261–296, 1987.

[32] O. A. Mahdi, M. A. Mohammed, and A. J. Mohamed, "Implementing a novel approach an convert audio compression to text coding via hybrid techniques," International Journal of Computer Science Issues, 2012.

[33] M. A. C. Perryman, K. S. de Boer, G. Gilmore, E. Hg, M. G. Lattanzi, L. Lindegren, X. Luri, F. Mignard, O. Pace, and P. T. de Zeeuw, "Gaia: Composition, formation and evolution of the galaxy," in Astron. Astrophys,April 2001.

[34] L. Evans, "The large hadron collider: a marvel of technology," in Fundamental sciences, EPFL Press 2009.

[35] J. Cermak, J. Bendix, and M. Dobbermann, "Short note: Fmet-an integrated framework for meteosat data processing for operational scientific applications," in Comput. Geosci., November 2009.

[36] J. P. de Mora, A. G. Villafranca, and E. Garca-Berro, "Quick and robust data compression for space missions," SPIE Newsroom, vol. 1, April 2011.

[37] P.-S. Yeh, W. Xia-Serafino, L. Miles, B. Kobler, and D. Menasce, "Implementation of ccsds lossless data compression in hdf," Space Operations 2002 Conference, 2002.

[38] T. Welch, "A technique for high performance data compression," IEEE Computer Society, vol. 17, June 1984.

[39] B. Langdon, "An introduction to arithmetic coding," IBM J. Res Develop, vol. 28, March 1984.

[40] CCSDS, "Losless data compression," Blue Book, vol. 1, May 1997.

[41] CCSDS, "Losless data compression," Green Book, vol. 1, May 1997.

[42] "Introduction to hdf5." [ONLINE]: `http://www.hdfgroup.org/HDF5/doc1.6/H5.intro.html`, May 2013.

[43] "Golomb-rice coding." [ONLINE]: `http://urchin.earth.li/~twic/Golomb-Rice_Coding.html`, February 2013.

[44] Data Compression. Springer, 2004.

[45] M. A. Nieto-Santisteban, D. J. Fixsen1, J. D. Offenberg, and R. J. H. ans H. S. (Peter) Stockman, "Data compression for ngst," in Astronomical Data Analysis Software and Systems VIII, June 1999.

[46] M. Clotet, J. Portell, A. G. Villafranca, and E. Garcia-Berro, "Simple resiliency improvement of the ccsds standard for lossless data compression," SPIE Journal of Applied Remote Sensing, vol. 4, July 2010.

[47] P.-S. Yeh, W. Xia-Serafino, L. Miles, B. Kobler, and D. Menasce, "Implementation of ccsds lossless data compression in hdf," in Earth Science Technology Conference2002, Pasadena, California, June 2002.

[48] "Szip compression." [ONLINE]: `http://www.hdfgroup.org/doc_resource/SZIP`, March 2013.

[49] "Gnu gzip." [ONLINE]: `http://www.gnu.org/software/gzip/manual/html_node/Advanced-usage.html`, May 2013.

[50] "Can gzip compress several files into a single archive?." [ONLINE]: `http://www.gzip.org/#faq16`, Jan 2010.

[51] "Lzf compression filter for hdf5." [ONLINE]: `http://www.h5py.org/lzf/`, July 2013.

[52] "Liblzf." [ONLINE]: `http://oldhome.schmorp.de/marc/liblzf.html`, July 2013.

[53] J. Portell, E. GarciaBerroad, and A. G. Villafranca, "Designing optimum solutions for lossless data compression in space, 2008," in Proceedings of the On-Board Payload Data Compression Workshop.

[54] J. Portell, E. GarciaBerroad, and A. G. Villafranca, "Quick outlier-resilient entropy coder for space missions," SPIE Journal of Applied Remote Sensing, vol. 4, July 2010.

[55] "Lz4 explained." [ONLINE]: `http://fastcompression.blogspot.com/2011/05/lz4-explained.html`, March 2013.

[56] Y. Collet, LZ4 Streaming Format, 2013.

[57] "lz4 - extremely fast compression algorithm." [ONLINE]: `https://code.google.com/p/lz4/`, July 2013.

[58] "Meerkat and kat-7." [ONLINE]: `http://www.ska.ac.za/meerkat/index.php`, February 2013.

[59] Experimentation in software engineering: an introduction. Kluwer Academic Publishers, 2000.

[60] K. Rajeswaran and J. Portell, "E-mail conversation," June 2013.

[61] M. C. from DAPCOM Data Services S.L., "E-mail conversation," July 2013.

Appendix A

Included CD

The CD included this dissertation has the following contents:

- An electronic copy of the entire dissertation
- Software source code for the SKA Compress program
- Sample compressed datasets