Comparing Data Flow Platforms for Astronomy Visualisation

Angus Pearce
PRCANG002@myuct.ac.za
University of Cape Town
South Africa

ABSTRACT

Image data collected from radio astronomy arrays has been increasing in size at an exponential rate due to improvements in technology, bandwidth, and the infrastructure of the arrays. This has caused a rise in the need to find better methods and structures for the reduction and processing of radio astronomy image data. The CARTA system is a tool used to process this image data, and new methods are being researched to deal with this exponential rise in data size. The dataflow model is a possible solution to this issue, with a scalable architecture that can run efficiently across massively distributed clusters, while eliminating the issues inherent in parallel programming.

This paper presents a prototype that uses a dataflow graph execution engine, DALiuGE, to implement certain features of the CARTA backend. The prototype, which includes a client-server architecture to mimic the CARTA system, is checked for numerical accuracy and subjected to efficiency testing.

The implementation of the prototype is detailed and the results of testing are shown, analysed, and discussed. The DALiuGE prototype shows promising results, outperforming the CARTA backend in certain features and achieving satisfactory results in others.

The results and discussion therefore show that the dataflow model is a plausible step forward in redesigning the CARTA backend functionality. Moreover, DALiuGE is shown to be incredibly flexible, efficient, and straightforward to integrate. We conclude that DALiuGE is worthwhile to consider as a dataflow framework, and meets the requirements for our use case.

CCS CONCEPTS

• **Computer systems organization → Client-server architectures**; **Parallel architectures**.

KEYWORDS

data flow, astronomical data processing, DALiuGE, distributed computing, Python

1 INTRODUCTION

The Cube Analysis and Rendering Tool for Astronomy (CARTA) [24] is an image visualization and analysis tool designed for the ALMA (Atacama Large Millimeter Array), VLA (Very Large Array), SKA (Square Kilometer Array) pathfinders, and the ngVLA (Next Generation Very Large Array). CARTA was developed and is maintained by the Inter-University Institute for Data-Intensive Astronomy (IDIA)[15]. It currently employs a multi-threaded C++ backend to process image data. This data is mainly in the FITS file format, the de facto format for working with image data from a data reduction pipeline. However, the data that these arrays produce has been increasing exponentially in size and will continue to do so with improvements and expansions to the current arrays. To be able to visualize and analyze this data efficiently there needs to be a change in the current methods of processing it, due to the fact that the current methods can be difficult to scale and consequently maintain at this scale.

The data flow model is a model that allows for implicit, fine-grained parallelism [20] on large amounts of data by visualizing workflow as the execution of a Directed Acylic Graph (DAG). These specific characteristics of the data flow model remove the need for a developer to worry about concurrency and parallelism issues, allowing them to focus on the processing, development, and optimization of sequential algorithms. This is of huge importance since as data scales, so must the distribution of data processing, which proves to be a nuisance should issues concerning the integrity of data arise.

This paper presents a prototype using the data flow model, implemented with the Data Activated *Liu* Graph Engine (DALiuGE) [29], a workflow graph execution framework built specifically for the reduction of interferometric radio astronomy data sets from the SKA. DALiuGE was developed and is maintained by the Data Intensive Astronomy team (DIA) at the International Centre for Radio Astronomy Research (ICRAR) [14], located in Western Australia. While it was built for astronomically large data sets and intended to run on large distributed clusters, it can be scaled down to even run locally on a single computer, making it an incredibly versatile implementation of the data flow model.

This paper will detail the design and methods used to build the prototype as well as having a focus on the reasons as to why DALiuGE is a viable option to replace the current data processing. The conclusions reached will be drawn from both the results of the prototype and the analysis of DALiuGE as a tool to implement the data flow model for our use case.

The aim of this prototype and research project is to use a data flow framework, in this case DALiuGE, to produce numerically equivalent results to CARTA's current backend and show that the framework can be more efficient and/or scalable in comparison. The research hypothesis is that DALiuGE is a powerful tool that is both flexible and scalable, and has the propensity to be the ideal framework to replace CARTA's backend. However, the system can only be as good as the underlying libraries and algorithms allow for.

This paper first discusses background work about the data flow model, DALiuGE, and Dask [6], followed by an explanation of the components within DALiuGE. The design and implementation of the prototype will be detailed, leading in to the testing methodology. Finally results will be shown with a full discussion and analysis from where conclusions will be drawn and discussed.

2 RELATED WORK

2.0.1 Data Flow. Nowadays processing power comes from making use of multiple cores and threads to run computations in parallel.

However parallelism brings forth further issues such as race conditions and deadlock, forcing developers to spend a lot of time and be very meticulous when executing operations in parallel. As the world of data gets bigger, the methods to process this data must be smarter.
Jack Dennis [7] created the first idea of a data flow model in 1974, based on the vision to create "a highly parallel computer system in which the execution of many program fragments is carried forward simultaneously". This architecture or model uses a DAG called a "task graph", which treats functions as nodes on the graph that will execute as soon as they receive input from the prerequisite functions.
A feature that stream processing brings is the compartmentalization of state within each compute unit (node), such as a thread. This means that each compute unit contains its own state and does not have to rely on any external information to know when to run, besides waiting for the prerequisite function to complete. This heavily simplifies the parallelization logic for the programmer and allows the runtime to focus on each compute unit separately while optimizing globally [2]. This removal of the programmer's concern and involvement in parallelization issues leaves more time and makes it simpler to optimize the individual parts of the program that run sequentially. An added benefit of this feature is that it encourages programmers to split their programs into logical partitions, which further helps with the optimization process.
Therefore, implementations of the data flow model contain some form of separation between application and parallelization logic, generally providing software alongside that optimizes the overall flow of the graph. Many implementations also use machine learning to be able to further optimize the work flow.

2.0.2 DALiuGE. DALiuGE allows data to trigger events, it integrates data lifecycle management within the data processing framework, and it explicitly decouples the logical view of a problem from its physical realization [3]. This explicit decoupling comes to life through the three main components of DALiuGE: EAGLE [13], the translator, and the engine. It was designed specifically for radio telescope image data processing, and so has been designed to allow stakeholders such as telescope operators, pipeline developers, and astronomers to optimize the data processing at multiple levels in a homogeneous manner [28].
DALiuGE has some advantage over other frameworks because it was first put through a test case on the Lianhe II supercomputer, with satisfactory results [18], and following that was used in two real-world use cases. The first of the real-world use cases used data from the Cosmos HI Large Extragalactic Survey (CHILES) [9] that runs at the VLA, and the second used data from the Mingantu Ultrawide Spectral Radioheliograph (MUSER) [26]. In both cases DALiuGE was mainly used for driving the data reduction pipeline. Overall, the results were satisfactory on both the scientific side and with regards to data processing performance. Moreover, the pipeline that was built based on DALiuGE was found to be simple to develop by re-using existing, mature pipeline software modules. Finally, the team at ICRAR are continuing working on DALiuGE to support it up to the levels of SKA Phase 2, which could bring about data transfers of ten times SKA1. It therefore promises continual improvements to the entire system, ensuring longevity.

2.0.3 DASK. Fouché et al. [11] designed a prototype for the CARTA system using a flexible parallel computing library called Dask [6] for Python. In his paper he achieved overall fair results with Dask in comparison to CARTA, both in local and distributed systems. He concluded that it was possible to adapt existing dataflow tools or systems to integrate with CARTA and showed the simplicity of Dask in performing complex operations.
However even with Dask seemingly being very flexible it still has its own issues. For example, developers mainly have to rely on the underlying optimization that Dask performs instead of being able to control optimization at lower levels. The results that Fouché et al. show, while not slow, still under-perform slightly in comparison to CARTA. This seems to represent a possible bottleneck where a developer could be left with slower computations without the means to improve performance.
DALiuGE actually ships with a Dask emulation layer that provides support for the *delayed()* and *compute()* methods that Dask uses to create its graphs. Using *dlg.delayed()* (dlg being DALiuGE) allows users to write code just as they would for Dask, but instead run the task graph under DALiuGE. Currently there are no performance comparisons but I believe that it could be a plausible alternative to using Dask.

3 DALIUGE SYSTEM

This section describes the main components of DALiuGE as a precursor to the design and implementation of the prototype. DALiuGE has full documentation available online [8].

3.1 Drops

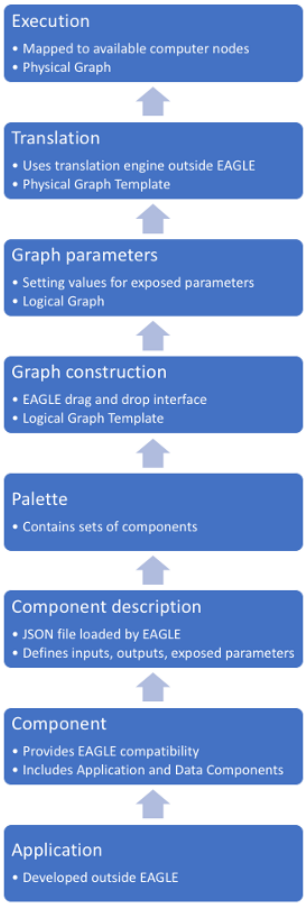
DALiuGE makes use of software objects called Drops [27], which were specifically created for the project to tackle the problems of data distribution on thousands of compute nodes. They are an abstract class that forms the nodes and edges of a DAG, and can carry either data (DataDrops) or applications (AppDrops). This is how they have fulfilled the idea of a Data Driven processing environment, with each Drop determining by itself when it should execute its own methods based on its type of class and the inputs provided. The only external control on each Drop is a data lifecycle management tool within the execution engine, which tracks each Drop and will migrate or delete the Drop automatically when required [29].

3.2 EAGLE

The *Editor for the Advanced Graph Language Environment* (EAGLE) [14] is the visual component of the Advanced Graph Language, and while it couples with DALiuGE, it is completely separate and only requires DALiuGE for the translation and execution of the graphs that it creates.
EAGLE uses components to create graphs that represent complex processing pipelines. These components are generated by wrapping the code that they represent in specialised Doxygen [19] comments and generating an XML output. This XML output can then be combined and converted into a .palette file using a Python script provided by DALiuGE. This palette represents a set of components that can be loaded into EAGLE and used to build a graph. These components and palettes are described using JSON, which has the

advantage of being readable by humans and therefore can technically be written by humans, although not advised.
EAGLE uses a simple drag-and-drop interface where components represent applications as nodes on the graph, with the edges connecting components describing the flow of data. It provides various types of application components such as Python, Bash shell and Dynamic Library applications. It also supports a variety of data components, with the most common being Memory and File (RAM and Disk).
One of EAGLE's main advantages is to be able to reuse code with immense ease, with it being as simple as loading a palette and dragging a component into the editor. This also enforces the idea of creating components that are similar to templates, allowing a single component to be used in different scenarios or with different data types.

Figure 1: Workflow from the EAGLE editor



3.3 DALiuGE Translator

The translator "unrolls" the logical graph, identifying nodes on the graph and creating all the application or data drops, as well as establishing directed edges that link all the drops together. It also shows a visualization of the Physical Graph Template so that the user can inspect the physical realisation of their logical graph for correctness. The translator also maps each node (Drop) to specific resources, creating a Physical Graph that contains all the information required for the execution engine to deploy it.
When submitting a Logical Graph to the translator there are a variety of algorithms for translating said graph. These algorithms have different focuses based on what the user wants to accomplish. For example, they can choose to minimise data movement whilst subject to load balancing or instead minimize execution time whilst subject to each partition's degree of parallelism (DoP). Some of these algorithms are expensive to run, especially on larger graphs, and they are meant to be used when the developer can set the "Execution Time" and "Data Volume" parameters for the AppDrops and DataDrops respectively.

3.4 DALiuGE Engine

The DALiuGE engine primarily uses *Drop Managers*, which receive the Physical Graph and coordinate to deploy all the Drops described and mapped by the graph. For each node there is a *Node Drop Manager*, which are grouped into a "data island" and managed by a *Data Island Manager*. If the graph is too large to be managed by one Data Island Manager, the engine will create more of them which will then be managed by a *Master Drop Manager*.
Each manager in the system exposes a REST Application Programming Interface (API) [21] which can be used by external users to interact with, deploy, and manage graph execution.
All the engine requires to deploy a session is a Physical Graph and access to the code described by the components in said graph.

3.5 Workflow

Figure 1 describes the entire process from creating application component code to executing the workflow described in the form of a graph in EAGLE. A graph begins as a *Logical Graph Template* which becomes a *Logical Graph* once the exposed component parameters have been filled in. The *Logical Graph* is then translated and becomes a *Physical Graph Template* after creating all Drops and directed edges between Drops. The graph is then partitioned per the algorithm used, subjecting the graph to specific constraints such as minimum data movement or execution time. This step produces a *Physical Graph* which can be deployed and run by the DALiuGE engine.
Examples of these graphs can be found in the appendix.

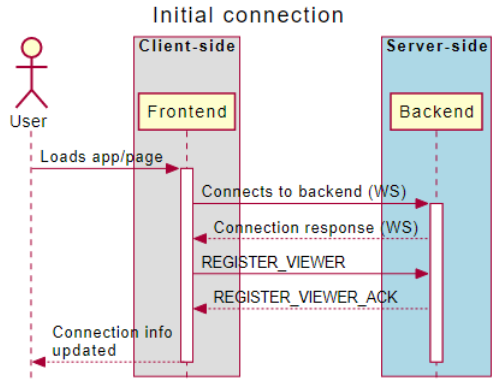
4 DESIGN AND IMPLEMENTATION

4.1 Architecture

4.1.1 Client-Server Model. To ensure a fair and controlled test environment, a Python client was built to emulate CARTA's client with only the features that the prototype implemented. This allows both the DALiuGE prototype and CARTA backend to run on the same hardware and interact with the same client. Therefore, a Python

server was also needed to communicate with the client and subsequently call the correct functions in DALiuGE. Communication between the client and server uses CARTA's protocol buffers [4] that are transferred using Python websockets [1] over the Transfer Control Protocol. Figure 2 shows the sequence of events to establish

Figure 2: Sequence diagram to start a connection in CARTA



a connection between the CARTA server and a client. Once this initial handshake is performed, the client has a consistent connection with the server and can select a file to open. CARTA's Interface Control Document (ICD) [5] contains all the necessary information for sending, receiving, and interacting with the protocol buffers.
Fouché et al. [11] implemented classes to create and read these protocol buffer messages in Python, and so their *message_header* and *message_provider* classes were used to streamline the process.

4.1.2 DALiuGE. As explained previously, DALiuGE supports multiple types of application components with the simplest being Python applications and Bash Shell commands. Python was chosen as the language of choice since DALiuGE is mainly written in Python, allowing for the easiest interaction with its API.
DALiuGE has a variety of ways in which to translate, deploy, and manage graphs. While it can deploy graphs directly via the command line or manually through the EAGLE interface, a choice was made to use the RESTful API [21] that is exposed by whichever type of *Drop Manager* is being used in the execution engine. These calls can be made with a few lines of Python code, so it was an obvious choice for simple integration between the engine and server.

4.2 Feature Implementation

While Python is incredibly simple and versatile, it suffers in efficiency due to the same factors that make it so. When working with any large amount of data, a base Python implementation would not only be too slow to use but also use up much more memory than necessary. This led to the use of Python's well-known NumPy [25] and Numba [17] libraries to provide an immense speed-up performing these calculations.
We also use the AstroPy library with its support for FITS file I/O (Input/Output). AstroPy already returns the image data in a NumPy ndarray, which makes it perfect for working with NumPy and Numba.
Both of the features were calculated over the entire FITS image range, to provide us with a case for features that will be computed often and over a large amount of data. Both features also were only used on 2D FITS files.

4.2.1 Basic Statistics. The first feature is the basic statistics. This includes the min, max, sum, mean, standard deviation, and the number of pixels in the file. When calculating these it must also be taken into account that there can be NaN values (Not-a-Number) that must be disregarded. NumPy provides all of these methods, even providing the same methods but excluding NaN values, e.g. *np.sum()* vs *np.nansum()*.

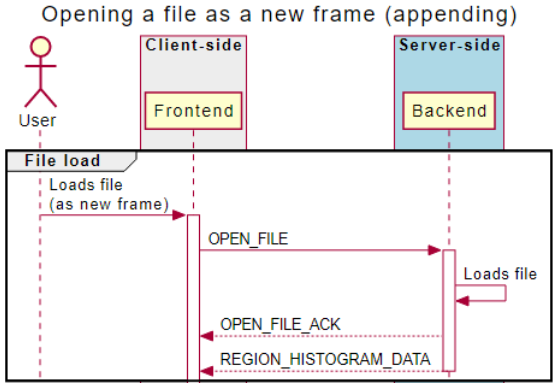
4.2.2 Region Histogram. The second feature is the region histogram. To calculate the number of bins that the histogram will generate we used CARTA's default method:

$$bins = max(\sqrt{width * height}, 2) \quad (1)$$

NumPy again does provide a method for computing a histogram, however the efficiency falls off fairly rapidly when dealing with large amounts of data. Numba, even though it has a compilation overhead at runtime, becomes faster thanks to it using various methods to speed up loops as well as caching this compilation so that it does not have to compile the method each call.

4.3 System Implementation Details

Figure 3: Sequence diagram to open a file in CARTA



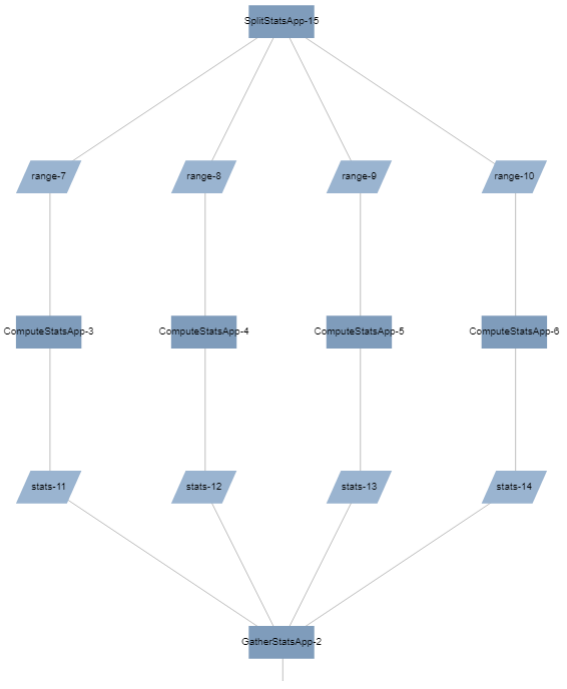
4.3.1 Computation Order. Figure 3 shows the sequence of events for opening a file in the CARTA system. Once a client has connected, registered, and been assigned a session, they can open a file. When a file is loaded, the CARTA backend automatically computes and sends the region histogram of the entire image to the client. This is done by first running through the file once to load it into cache and calculating the basic statistics. Since the file has already been loaded into the image cache, the histogram calculation can be performed very quickly by using that cache. A choice was made to mimic this flow of events by calculating the statistics and then histogram immediately after a file is opened.

4.3.2 Component Classes. To use the EAGLE editor with our code we needed to define the Doxygen comments wrapping a class based on the inputs, outputs, and purpose of the component it represents. Each class defines a single component and will require different

methods to be overwritten depending on the type of base 'Drop' class they extend. In our case we extended a class called *BarrierAppDROP*, which meant that each class only needed a single *run()* method which would only be called once every input to the class had completed. It is important to note that this class represents the simplest version of batch processing, whereas to implement stream processing you need only extend the *AppDROP* class and define a streaming compatible application.

To perform the statistics and histogram computation, six classes were created to be used within DALiuGE as application components: *SplitStatsApp*, *ComputeStatsApp*, *GatherStatsApp*, *SplitHistApp*, *ComputeHistApp*, and *GatherHistApp*. The split and gather classes are used within EAGLE in scatter and gather constructs respectively. These constructs generally work in tandem, with scatter containing a 'Number of Splits' parameter and gather containing a 'Number of Inputs' parameter. The scatter's job is to replicate the middle node by the number of splits parameter, while the gather receives inputs from however many inputs it has. Figure 4 shows one scatter

Figure 4: Physical Graph of Scatter and Gather



and one gather construct in a physical graph, namely *SplitStatsApp* (top) and *GatherStatsApp* (bottom). Setting the number of splits and number of inputs parameters each to four yields this graph.

The *SplitStatsApp* class takes in a file name parameter as specified by the client when they choose the file. It computes a set of ranges over the file data, based on the file dimensions and the number of outputs it has, and passes a unique range to each output node. The nodes in between the scatter and gather applications then each receive a unique range, using it to read only the section of the file that they have been assigned. These nodes, each an instance of *ComputeStatsApp*, compute the statistics over their unique range of data, after which they pass the results downstream to the gather application to be compared and combined to a final output. Performing the reading and computations in this way not only allows for reading the FITS file in parallel but also removes the need to physically send the file data downstream to each node, increasing efficiency and decreasing memory usage.
Once the statistics have been gathered, *GatherStatsApp* sends the min and max to *SplitHistApp* to once again split the data in ranges, allowing *ComputeHistApp* to compute the histogram over their unique ranges and finally combining each histogram in *GatherHistApp*.
It should be noted that each app passes a variety of data downstream, depending on the needs of the downstream nodes. This data can be put in a normal Python list, serialised using the *pickle* library, and sent to whichever output requires it.

4.3.3 Graphs and GraphLoader. The graphs that were used were created manually in EAGLE, using a palette of components that was created using Doxygen comments wrapping the Python classes describing components, as described in section 3.2. The graphs were then exported as Physical Graphs in JSON format which can be loaded into Python and passed into the engine via the REST calls. The *GraphLoader* class was used for this purpose. It receives a file name and graph name as arguments and attempts to load the graph file using the *JSON* library. Since the graph we are using is a physical graph, it has already been mapped to specific IP addresses and been instantiated with the file name when it was translated manually through EAGLE. This can be mitigated by translating the logical graph using code by sending it to an instance of the translator, but that adds another dependency which was unnecessary for our case. To overcome this you are able to iterate through the loaded JSON object, find the Drop definition of the Drop that you wish to change, and manually change the JSON definition.
Following that, as long as the DALiuGE engine is running, the *GraphLoader* object connects to a Node Manager on localhost with a specified port. To deploy a physical graph, the only methods that need to be called on the Node Manager are *createSession()*, *addGraphSpec()*, and *deploySession()*.

5 TESTING

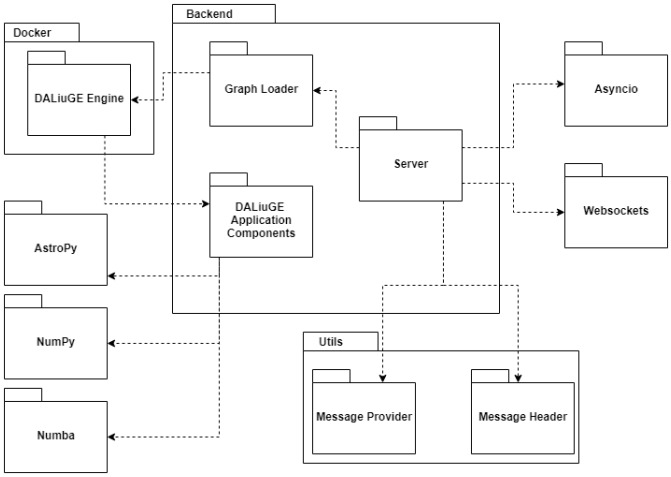
5.1 Test Environment

The testing environment consisted of two Linux Virtual Machines (VM's) running Ubuntu 20.04.03 LTS. Each virtual machine had an 8 core CPU and 64GB's of RAM. These VM's were provided by IDIA and are hosted on the ilifu HPC cluster [16].

5.2 Testing for Accuracy

Testing was required to ensure that the results from the prototype and CARTA were numerically equivalent. This involved computing the statistics and histogram over the same file using the DALiuGE prototype and CARTA system, and asserting that the results are within a certain margin of error of each other. The outputs of multiple files were compared and cross-checked for accuracy.

Figure 5: Package diagram of the prototype backend



5.3 Testing for Efficiency

Efficiency, in our case, encapsulates both the speed of the functions as well as the scalability of the framework. Scalability is especially important because it ensures that greater data sizes can still be handled by expanding the processing horizontally. This can be difficult to test as it requires being able to expand the prototype to many multiple compute nodes.

We tested eleven FITS files that were generated randomly using a tool provided by IDIA [10]. These eleven files ranged from a 5000x5000 pixel file to a 55000x55000 pixel file, with file size ranging from 100MB to 12GB.
For each test run we cleared the file system cache completely, loaded the file and ran the statistics and histogram on either the CARTA backend or the prototype. For DALiuGE the engine had to be restarted each test run after clearing the cache.
CARTA's test results are taken from their performance logging, which shows the time taken to load the image into cache (including the basic statistics calculations) and time taken to compute the histogram.
The prototype's timing begins at the beginning of the *run()* method in each scatter, and ends when the respective gather has combined the output into a final result. These results are then encoded and sent through a specific port to a File Drop to be written to disk. This is seen in both the logical graph and physical graph in the appendix.

Each file was tested using multiple different parameters, namely the physical graph (controlling the multi-processing), and the number of splits inside each process. The number of splits inside each process breaks the data into multiple pieces again, allowing the compute nodes to perform calculations on smaller subsets of their own unique range of data. This not only increases the efficiency of specific calculations but also means that only a certain amount of overall data will be loaded into memory at a time.
For example, an inside split of 4 means that each compute node only loads a quarter of their unique range of data at a time, meaning that overall only a quarter of the entire file will be loaded into memory at any given point.

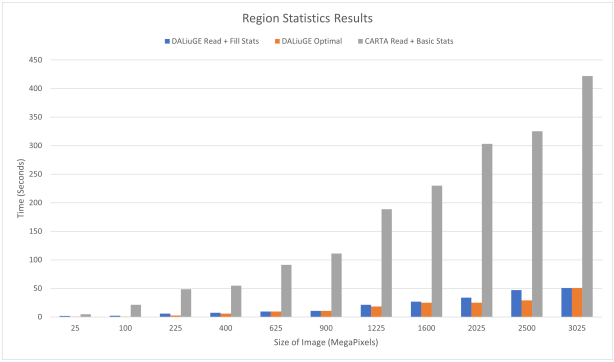
6 RESULTS AND DISCUSSION

Figure 6 shows the results of the statistics computation while Figure 7 shows the results of the region histogram computation. The table with these results can be found in the appendix, with the colours corresponding between the table and graphs.

As stated previously and shown in the graphs in the appendix, the prototype first calculates statistics while reading the file for the first time, and then calculates the histogram when reading the file a second time. This means that the first read, which includes calculating the statistics, does not use any cache whatsoever. The second read, including the histogram, now has the file stored in cache but still must read once more from the disk and not directly from memory.
In each graph the blue bars represent the times for the prototype, all using the same graph file and parameters. In this case, the graph that was used contained 16 splits for both the statistics and histogram, with each node splitting their unique range of data into 8 pieces. This obviously does not show the true performance, and is instead used as a control. The red bars represent the optimal times for the statistics or histogram. Optimal here simply means that this was the lowest time that was recorded during testing. These optimal times come from a variety of graphs and parameters that were tried during testing, and still do not represent the most optimal solution, seeing as there are a variety of factors and too many possible parameter combinations to perform this manually.
The physical difference between using 4 and 8 splits can be seen on pages 2 and 3 in appendix A.

6.1 Statistics

Figure 6: Region Statistics

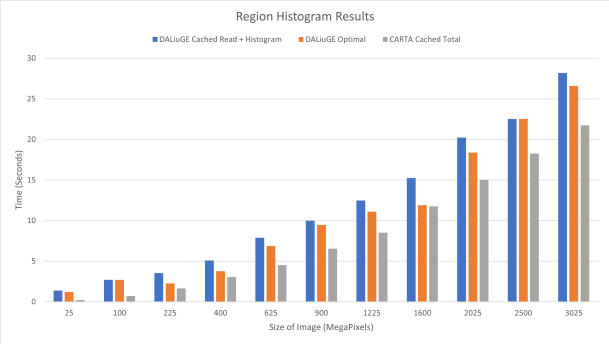


As noted, the times displayed here are using no cache whatsoever. The time used for CARTA is the time taken to load the image into cache and calculate basic statistics, which is a near exact comparison to the prototype.
As shown in the graph, both the control and optimal times far outperform CARTA's time. On average we see around a 7 times speedup for the control, with outliers such as the optimal time for

the 100 MegaPixel file (10000x10000 pixels) achieving a 26 times speedup.
These stark differences in times can be attributed to the parallel reading performed by the prototype. Each compute node reads their unique 1/16th slice of the file while the other nodes do the same. This drastically increases performance and when compared to CARTA, which reads mostly sequentially, the difference is clear. Something not shown here is the memory usage. The DALiuGE prototype only used around 8 to 10 gigabytes of memory for the largest file we tested, with CARTA reaching around 14 gigabytes. However, CARTA used substantially more memory for smaller files in comparison to DALiuGE, and its to be noted that the prototype can easily be adapted to use more or less memory albeit at the possible loss of performance.

6.2 Histogram

Figure 7: Region Histogram



To make sure that the comparison for the histogram computation was fair, the times for the prototype to read the file a second time and calculate the histogram were compared against the CARTA backend reading the file from cache and then performing the histogram. This does not represent the flow of CARTA but provides a fair comparison. Using this comparison shows that CARTA outperforms the prototype, but only by a few seconds at best and around a tenth of a second at worst.
The explanation as to why the prototype falls short here is fairly straightforward. Numba was used to perform the histogram computation. Numba can be very fast and efficient when working with larger arrays but it also has a compilation overhead. While it still proved to be faster than using NumPy for the histogram, each compute node had to compile the function once at runtime (Numba uses Just-In-Time compilation), which technically brings more overhead with each split. Using Numba more efficiently for the histogram would require more testing to figure out a balance between the size of the data with regards to the efficiency of the function and the amount of memory used.

6.3 Distributed Computing

Part of the aims of this project included being able to distribute the system onto multiple nodes. Due to some issues surrounding the virtual machines, results of this were not achieved. However, the current system can be completely distributed without a single change in the code as long as the FITS file can be accessed from each virtual machine or node.
Distributing the system only requires the deployment version of the engine to be built on each node which, when run, will each start up a Node Manager on their respective nodes. Following that a Data Island Manager needs to be started on one of these nodes, which will be fed the IP addresses of the other nodes, creating a distributed system.
Finally, the graphs used must be translated using the *metis* algorithm, which has parameters indicating the number of nodes and number of islands, with a feature to balance the load on each node.

6.4 Possible Changes and Thoughts

While Python was simple to work with and had some advantages in that sense, it should not be used if straight efficiency is key. C and C++ are supported languages in DALiuGE through the Dynamic Library component. Multiple systems that are using DALiuGE in production currently make use of C++ application components. Since CARTA's backend is built in C++ it makes sense to use it. It is more than likely that a C++ implementation, using the same architecture and graphs, would be faster given its underlying efficiency and lower-level control.
DALiuGE has a remarkably extensive list of supported application and data components. This list encapsulates a variety of methods for running a large number of different applications through DALiuGE. Components such as the Message Passing Interface (MPI)[12] component could be taken advantage of when dealing with parallel computing, while the Docker component allows any Docker container [22] to be run through DALiuGE. These showcase the flexibility of the system in comparison to most other models which do not nearly have the same functionality.

7 CONCLUSIONS

We implemented a software prototype in Python using DALiuGE, a data flow graph execution framework designed to support large scale distributed computing. We built a set of components that can be used to create a graph describing the workflow of these components.
We implemented a client-server model in Python that mimics certain functionality from the CARTA system, with the client being able to interact with both the Python server and CARTA's backend.

The results from testing show that DALiuGE outperforms CARTA heavily when reading a file for the first time. The statistics and histogram results, while showing very different comparisons, are entirely explainable and prove a case as to why DALiuGE is worthwhile to use as a data flow framework.
Pulling from the related work done on DALiuGE and the understanding of how the distribution works, we can conclude that DALiuGE is a robust and massively scalable implementation of the data flow or work flow model. In contrast to other systems such as Dask or RaftLib [2], it allows developers to fully utilise the data flow system while taking advantage of an extensive number of other libraries or systems. This being said, DALiuGE is only as good as

the underlying systems that it makes use of.
We can also conclude that while Python performed admirably in the tests, the histogram computation does show the shortcoming of using Python libraries, introducing either an overhead or only providing support for specific functionality.
We therefore find that DALiuGE is a suitable framework that could be used in the CARTA system. For this, we recommend that further testing be done using C or C++ application components to ensure the highest levels of efficiency and memory management.

8 FUTURE WORK

DALiuGE was just fully released and is currently in version 1.0.0. It is currently receiving big overhauls and optimizations in many areas, while updating support for new components and systems. Notably, support for using Ray [23] to implement distributed computing is being worked on. Ray, a framework for scaling Python workloads, should make distribution both easier and more efficient.

ACKNOWLEDGMENTS

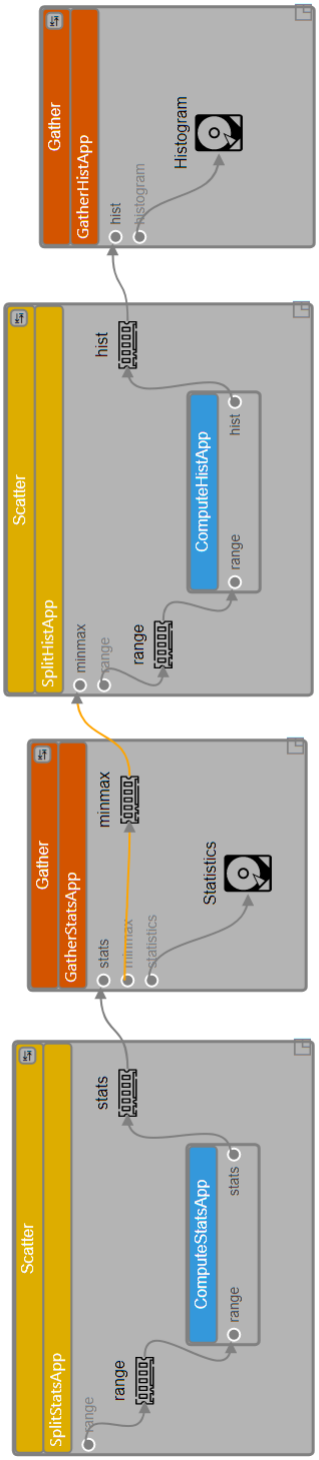
A big thanks to Professor Andreas Wicenec, the head of the Data Intensive Astronomy program at the International Center for Radio Astronomy Research, for his overwhelming assistance and communication when working with DALiuGE.
This project is complemented in part by the work done by Georgeo Thanathara, who assisted in the research and understanding of DALiuGE.
To the supervisors Professor Rob Simmonds, Ms Adrianna Pińska, and Dr Angus Comrie, for their guidance and assistance in solving various issues.
This work made use of the CARTA (Cube Analysis and Rendering Tool for Astronomy) software (DOI 10.5281/zenodo.3377984 – https://cartavis.github.io).
We acknowledge the use of the ilifu cloud computing facility – www.ilifu.ac.za, a partnership between the University of Cape Town, the University of the Western Cape, the University of Stellenbosch, Sol Plaatje University, the Cape Peninsula University of Technology and the South African Radio Astronomy Observatory. The Ilifu facility is supported by contributions from the Inter-University Institute for Data Intensive Astronomy (IDIA – a partnership between the University of Cape Town, the University of Pretoria, the University of the Western Cape and the South African Radio astronomy Observatory), the Computational Biology division at UCT and the Data Intensive Research Initiative of South Africa (DIRISA).
The author would like to acknowledge the use of existing code, taken from a project done by Dylan Fouché [11], which provided functionality for the client-server model.

REFERENCES

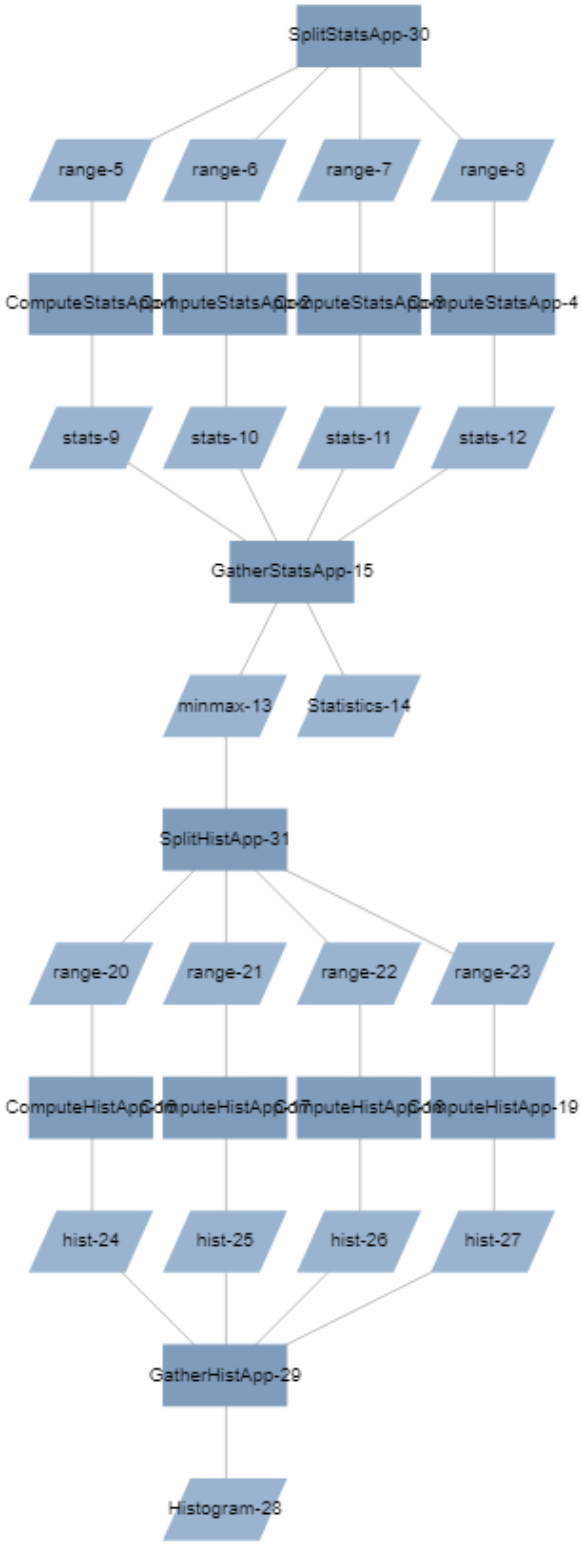
[1] Aymeric Augustin. 2021. Aaugustin/Websockets: Library for building websocket servers and clients in Python. https://github.com/aaugustin/websockets
[2] Jonathan C. Beard, Peng Li, and Roger D Chamberlain. 2016. RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. *International Journal of High Performance Computing Applications* (2016). https://doi.org/10.1177/1094342016672542 arXiv:http://hpc.sagepub.com/content/early/2016/10/18/1094342016672542.full.pdf+html
[3] Mark Boulton, Ian Cooper, Richard Dodson, Markus Dolensky, Dave Pallot, Rodrigo Tobar, Kevin Vinsen, Andreas Wicenec, and Chen Wu. 2019. Imaging SKA-Scale Data on Cloud and Supercomputer Infrastructure Using Drops and DALiuGE. *Astronomical Data Analysis Software and Systems XXVI* 521 (2019), 628.
[4] CARTAvis. 2021. CARTAvis/carta-protobuf: Protocol Buffer Messages. https://github.com/CARTAvis/carta-protobuf
[5] Angus Comrie, Rob Simmonds, and the CARTA development team. 2021. CARTA Interface Control Document. (2021). https://carta-protobuf.readthedocs.io/en/latest/index.html
[6] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling.* https://dask.org
[7] Jack B Dennis. 1974. First version of a data flow procedure language. In *Programming Symposium*. Springer, 362–376.
[8] DALiuGE Docs. 2021. DALiuGE Documentation. https://daliuge.readthedocs.io/en/latest/index.html
[9] Ximena Fernandez, Jacqueline H van Gorkom, Emmanuel Momjian, and Chiles Team. 2015. The COSMOS HI Large Extragalactic Survey (CHILES): Probing HI Across Cosmic Time. In *American Astronomical Society Meeting Abstracts# 225*, Vol. 225. 427–03.
[10] Inter-University Institute for Data Intensive Astronomy (IDIA). [n.d.]. Gaussian noise FITS image generator. ([n. d.]). https://github.com/idia-astro/image-generator#readme
[11] Dylan Fouché and Zainab Adjiet. 2020. Prototyping a Dataflow Implementation of the CARTA System. (2020). https://github.com/DylanFouche/CADaFloP
[12] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.
[13] ICRAR. 2021. ICRAR/EAGLE: Editor for the Astronomical graph language environment. https://github.com/ICRAR/EAGLE
[14] ICRAR. 2021. International Center for Radio Astronomy Research. https://www.icrar.org/
[15] IDIA. 2021. Inter-University Institute for Data Intensive Astronomy. https://www.idia.ac.za
[16] ilifu. 2021. http://www.ilifu.ac.za/
[17] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.
[18] Baoqiang Lao, Tao An, Chen Wu, and Rodrigo Tobar. 2018. SDP Memo 078: Scalability Testing using DALiuGE on Tianhe-2 and Pawsey.
[19] Robert S Laramee. 2011. *Bob's Concise Introduction to Doxygen.* Technical Report. Technical report, The Visual and Interactive Computing Group, Computer ....
[20] Ben Lee and Ali R Hurson. 1994. Dataflow architectures and multithreading. *Computer* 27, 8 (1994), 27–39.
[21] Mark Masse. 2011. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* "O'Reilly Media, Inc.".
[22] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.
[23] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.
[24] J. A. Ott and Carta Team. 2020. CARTA: Cube Analysis and Rendering Tool for Astronomy. In *American Astronomical Society Meeting Abstracts #235 (American Astronomical Society Meeting Abstracts, Vol. 235)*. Article 364.11, 364.11 pages.
[25] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering* 13, 2 (2011), 22–30.
[26] F Wang, Y Mei, H Deng, W Wang, CY Liu, DH Liu, SL Wei, W Dai, B Liang, YB Liu, et al. 2015. Distributed data-processing pipeline for mingantu ultrawide spectral radioheliograph. *Publications of the Astronomical Society of the Pacific* 127, 950 (2015), 383.
[27] A Wicenec, D Pallot, R Tobar, and C Wu. 2017. DROP Computing: Data Driven Pipeline Processing for the SKA. *Astronomical Data Analysis Software and Systems XXV* 512 (2017), 319.
[28] Chen Wu, Rodrigo Tobar, Kevin Vinsen, Andreas Wicenec, Dave Pallot, Baoqiang Lao, Ruonan Wang, Tao An, Mark Boulton, Ian Cooper, et al. 2019. DALiuGE: Data Activated Liu Graph Engine. *Astrophysics Source Code Library* (2019), ascl–1912.
[29] Chen Wu, Rodrigo Tobar, Kevin Vinsen, Andreas Wicenec, Dave Pallot, Baoqiang Lao, Ruonan Wang, Tao An, Mark Boulton, Ian Cooper, Richard Dodson, Markus Dolensky, Ying Mei, and Feng Wang. 2017. DALiuGE: A Graph Execution Framework for Harnessing the Astronomical Data Deluge. *CoRR* abs/1702.07617 (2017). arXiv:1702.07617 http://arxiv.org/abs/1702.07617

A GRAPHS

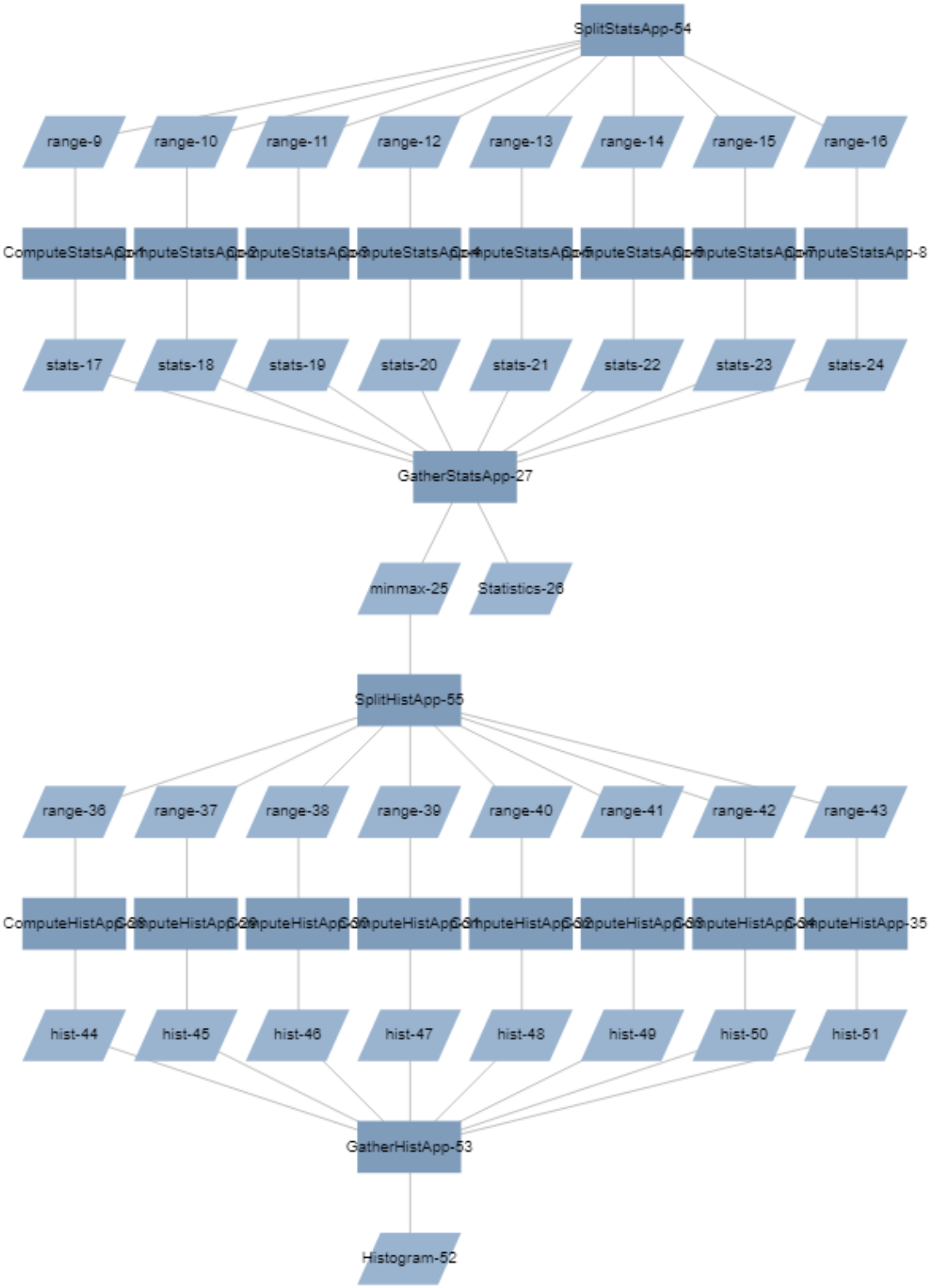
1. Logical Graph in the EAGLE editor



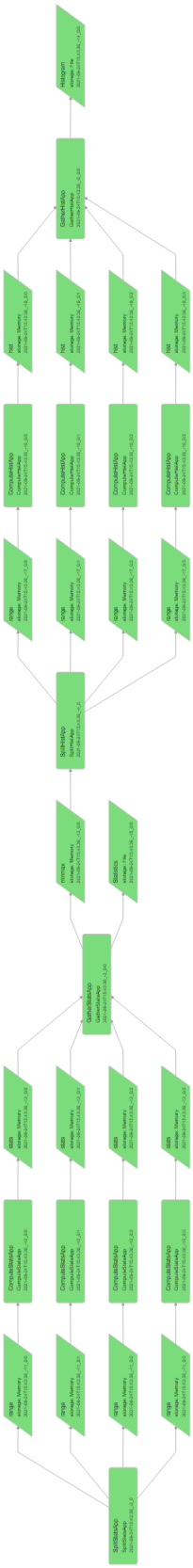
2. Physical Graph with four splits and four inputs



3. The same Physical Graph as above, but with the splits and inputs set to 8



4. Visualisation of a graph being executed

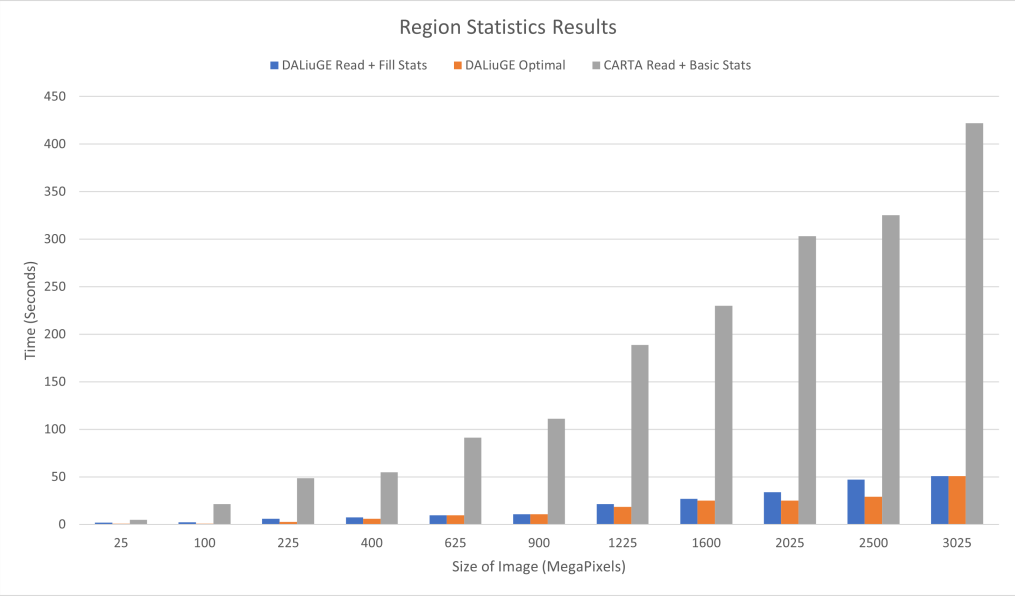


B RESULTS

1. Table of results from region statistics and region histogram

Region Statistics Results					
Size	DALiuGE Read + Stats	DALiuGE Optimal	CARTA Read + Basic Stats	CARTA Fill Stats	CARTA Total
25	2.027028227	0.754018002	4.744052	0.382762	5.126814
100	2.373027858	0.79010334	21.352039	0.883816	22.235855
225	6.024401802	2.798460564	48.700653	1.624627	50.32528
400	7.378395238	5.841263578	54.77333	2.818183	57.591513
625	9.545787312	9.545787312	91.261641	4.391783	95.653424
900	10.69808986	10.69808986	111.097504	5.704692	116.802196
1225	21.45605907	18.60635566	188.697594	8.689184	197.386778
1600	27.03918563	25.06860012	230.029924	11.390597	241.420521
2025	34.0205701	25.13846165	303.343661	14.731795	318.075456
2500	47.00231403	28.95704573	325.343537	18.729208	344.072745
3025	50.79092291	53.91362823	421.949305	19.414292	441.363597
Region Histogram Results					
Size	DALiuGE Cached Read + Histogram	DALiuGE Optimal	CARTA Cached Read + Basic Stats	CARTA Fill Histogram	CARTA Total
25	1.381894662	1.212634273	0.177042	0.032393	0.209435
100	2.700731946	2.700731946	0.651878	0.069397	0.721275
225	3.546727064	2.266059596	1.503406	0.151729	1.655135
400	5.099853701	3.776211333	2.800355	0.238964	3.039319
625	7.893039323	6.873221383	4.154806	0.369488	4.524294
900	9.98751429	9.463366911	6.038611	0.49354	6.532151
1225	12.47261516	11.11222251	7.815061	0.702225	8.517286
1600	15.2445905	11.89571494	10.796861	0.951859	11.74872
2025	20.21066948	18.38305343	13.641072	1.387439	15.028511
2500	22.53471505	23.05027648	16.678984	1.566836	18.24582
3025	28.17286294	26.60591661	19.920376	1.806338	21.726714

2. Region Statistics Chart



3. Region Histogram Chart