Comparing Data Flow Platforms for Astronomy Visualisation

Georgeo Thanathara
THNGEO002@myuct.ac.za
University of Cape Town

ABSTRACT

The need for remote astronomy visualisation is becoming more demanding as time passes. As infrastructure improves and continues to exceed expectations, there is a need for a system that can accurately analyse and render the images produced from the array telescopes. With the implementation of remote visualisation tools such as CARTA, the power of the multithreaded C++ backend is ever present.

However, as the data cubes increase in the magnitude of size, there is a need for new backend technologies that will be able to return data efficiently and promptly to the front end. This paper focuses on one technology known as dataflow models, a paradigm that makes use of Directed Acyclic Graphs where the nodes are represented as functions and the inputs are treated as dependencies.

The following paper will investigate a C++ Dataflow library known as Raftlib and will explain how the library can be implemented for a few basic features like Region Statistics, Channel Histograms, and Cube Histograms through the use of custom kernels and operator overloads to link them and produce executable graphs. This prototype has been subjected to thorough testing and the efficacy of the library as well as it's performance has been compared against CARTA.

CCS CONCEPTS

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

KEYWORDS

dataflow, astronomical data processing, RaftLib, CARTA, parallelism

1 INTRODUCTION

The ever-growing scale and the nature of high-resolution astronomical imagery that is produced by the many telescope arrays requires a need to be analysed and rendered through remote access. In particular, radio astronomers are able to create sharper, brighter, and higher resolution images using a radio telescope interferometer. This is because in contrast to a single antenna image, two or more image measurements are connected electronically by pointing the antennas in the same direction. The interferometer allows the combination of measurements simultaneously from a pair of antennas in an array resulting in high resolution measurements of a focal point [14]. As expected, these interferometric observations produce enormous cube sizes resulting in larger image file sizes. As the industry and technology continues to evolve, the standard is getting set and such cube sizes are becoming more common, especially since cube sizes increase linearly depending on the number of channels for line or polarisation observations [23].

With the planned production of the Square Kilometer array (SKA) underway, it is set to be the worlds largest radio telescope and will be more powerful and faster when it comes to mapping the sky. It is expected that the SKA will be able to discover pulsars of steller-size and super black holes which can be used to test a number of theories on gravity. Perkins et al [17] describes a number of concerns relating to visualising such images from the SKA; what is distinct to this project is the mention of the sheer size of the data files due to the above mentioned interferometric observations. It can almost be guaranteed that the file sizes could increase in size to terabytes.

As the cube sizes get larger due to the higher resolution measurements, the existing visualisation tools such as CASA-viewer [15] notices a significant increase in processing times and a reduction in efficiency. This casts a negative shadow on user experience as it is impractical to be waiting for minutes in order to perform quick tasks such as region histograms, cube histograms or even region statistics. An efficient and scalable solution that was built is known as CARTA [23], a Cube Analysis and Rendering Tool for Astronomy, which provides a solution for visualising large image cubes. The tool supports images produced from the Atacama Large Millimetre Array (ALMA) [24], MeerKat (alternatively known as the Karoo Array telescope) [7], and soon to work with the upcoming Square Kilometre Array (SKA) [8] datasets. The images used can be in CASA, FITS, MIRIAD or a unqiue HDF5 IDIA scheme [21].

The HDF5 IDIA Scheme provides useful pre-computed data, such as histograms and statistics, which are usually I/O Intensive calculations. On the other hand, Flexible Image Transport System files (FITS) do not provide such extensive data within the file. However, FITS files have been in use since the early 1970s but are now considered to be the de facto of astronomy visualisation ever since the release of better formats like the HDF5 [18]. There are a number of tools to convert FITS files to HDF5 such as fits2hdf which creates a HDFITS files which ports data models to HDF5. CARTA also has its own converter that is used to transform it into the HDF5 IDIA Scheme. It is not always possible for a client to convert a file before they make use of it on the user interface, thus CARTA allows for FITS files to be uploaded which results in slower response times from the backend due to having to perform calculations that would be usually be available in an HDF5 file.

CARTA is currently written in an imperative multi-threaded C++ backend [10, 20]. While this solution provides great results and processing times, it is necessary to look into different possible implementations which could prepare for larger image sizes in the future produced from telescopes such as the SKA. This paper propose a dataflow model and asses their suitability on being used in the CARTA backend to perform certain tasks. Dataflow architectures make use of Directed Acyclic Graphs(DAG), this is a mechanism that allows functions to be represented as nodes, and the arguments to the functions(dependencies) are treated as the

inputs to the nodes. A firing set, as described by Johnston et al [13] is whenever a node has data on its input arc. The node is executed and the result is placed on the output arc. The node then stops its execution and waits for it to receive data on its input arcs again. This is what makes the dataflow architecture unique, as it allows for multiple instructions to be executed in parallel once they have their dependencies ready.

Given the above mentioned caveat and limitations, this research project attempts to answer the following research question

> *RQ: Is there a suitable framework for implementing the data-flow model for radio astronomy image data processing, and is it scalable and efficient enough to be able to handle the astronomical data sizes that the future will bring?*

To overcome this problem, this paper will focus on the implementation of a C++ dataflow library known as RaftLib [6].The remainder of this paper will present some related work followed with the explanation and implementation of the RaftLib library on a few basic features that are computationally expensive, such as the ones that are lacking in FITS files. The suitability of the library will be evaluated and discussed by comparing the results against the CARTA backend by testing on the ilifu [12] clusters.

2 BACKGROUND AND RELATED WORK

2.1 Stream Processing and Raftlib

In contrast to batch processing, stream processing is a compute paradigm that has been around for a while. Modern applications that use as IoT devices and real-time processing are implementing stream processing as it allows for immediate execution of tasks as soon as the data arrives. Raftlib exploits the stream processing and allows a user to create and write sequential code in kernels, this allows the compartmentalisation of state within each compute kernel, which is one of its core features [1]. As described by Beard et al [6] there are a number of advantages in stream processing, for instance the ability to separate the program into logical partitions and thus allowing the programmer to think sequentially about the independent pieces whilst allowing the entire program to execute in parallel.

Three of the main optimization techniques that Raftlib provides are Dynamic Queue Optimization, Automatic Parallelisation, and Real-Time Low Overhead Performance Monitoring. The library is created such that it can eliminate bottlenecks in resource allocation by dynamically monitoring the system. This allows the programmer to rather focus on application logic. Raftlib is considered as an auto-tuned streaming system, this means that it can mitigate communication costs, adaptively schedule compute kernels, and provide low overhead instrumentation during run time. Common issues such as data races in traditional parallel code are not possible in RaftLib. Ports are also safe when accessing data from methods within the library's run method. Additionally, most streaming applications make use of different optimization techniques depending on the input, RaftLib works similarly but allows the user to specify submaps known as "synonymous kernel groupings" which swap out to optimize the computation during runtime.

2.2 Simplifying Parallelisation using Raftlib

The Raftlib github page [5] provides a variety of examples that can be used to understand how the library works. One tutorial in particular which relates to this research is a concise post that was written to compare the parallel data compression with PBZIP2 [11] against a Raftlib implementation of this [4]. The post went on to benchmark the compression speed of 10 gigabytes of data and it was clear that the Raftlib implementation was able to yield better results and proved to be slightly faster given the conditions that it was tested in. Although not the same, this tutorial showed the methodology of parallelisation using the library, this was used as a foundation in building custom kernels for this research explained hereunder.

2.3 DASK Implementation

Fouche et al [10] implemented a set of back-end components that mimic the behaviour of the CARTA backend using DASK. This was done to compare and test the efficiency and efficacy of the dataflow model and compare it to the C++ multithreaded back-end of CARTA. The region histogram computation and the region statistic computation functions were chosen and were tested when data was on both disk and memory. The test results showed that the DASK implementation significantly outperformed the CARTA when performing computations on data that is on the disk. The implementation on a distributed scheduler was almost always faster than the local scheduler. This is reasoned by mentioning that the distributed scheduler can aggregate an arbitrary number of worker nodes given some conditions, whereas the local implementation and CARTA are strictly bound to the number of cores available. We also learn that latency can cause network server issues which result in slower responses.

3 DESIGN AND IMPLEMENTATION

3.1 Architecture

It was decided that the implementation of this project would try replicate the CARTA client-server architecture wherever possible. This paper deemed it reasonable to create a simple frontend in python that shared ideas from Fouche [10] which made use of terminal inputs. This was motivated because the primary focus of this project is the backend and thus it is not required to create an actual website for the frontend. However, during the testing phase we anticipate results from the current existing CARTA backend through our frontend as to ensure that benchmarking is made as fair as possible. Simply timing the CARTA frontend and getting results from there is not a fair comparsion. It is also further motivated as this experiment aims to perform the testing on the ilifu clusters as to compare these results against CARTA on the same hardware.

The CARTA Interface Control Document [2] (ICD) and the log events from the CARTA website was studied extensively to understand how exactly the custom protocol buffers were used in order to exchange messages and acknowledgements. The connection will be established using the TCP protocol and the communication will take place through a websockets protocol. This protocol will allow for reliable, error checked and ordered transmission of data. An

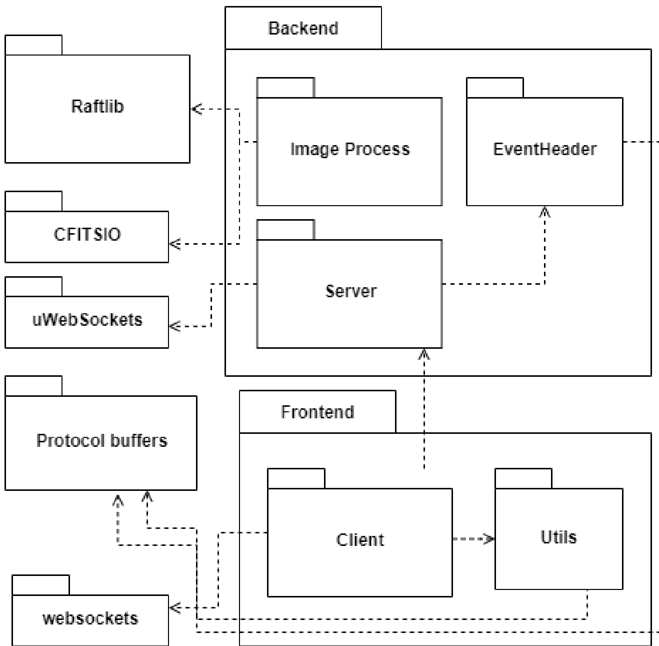

Figure 1: Packagediagram.

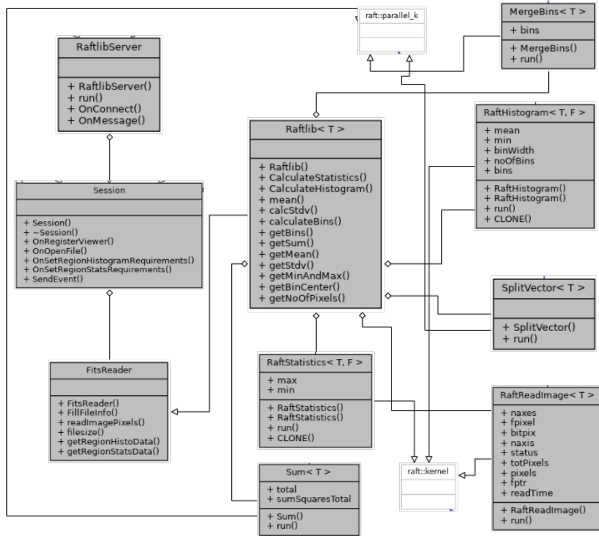

Figure 2: Class Diagram.

instance will be created when a user loads the web page and communication will take place through a shared repository of protocol buffer references. The ICD defines the protocol definitions that can be used in order to set up and establish a connection.

Figure 3 shows the sequence diagram when a connection is established between a CARTA client and server. Our python *client* has been made to closely mimic this behaviour. The *RaftlibServer* will be running and when the client connects to the backend, the server sends a response requesting the client to identify itself. The client will use a register viewer protocol in order to establish a

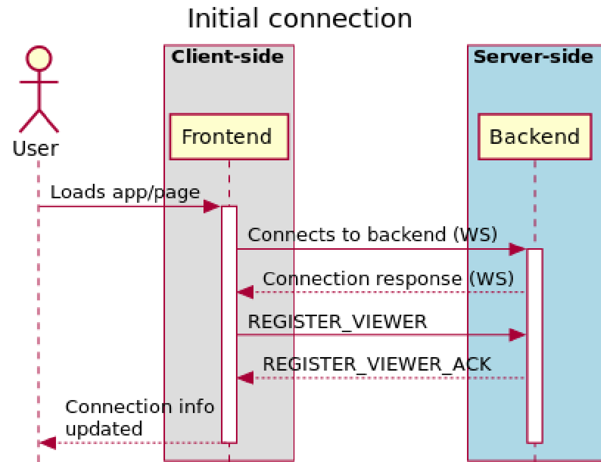

Figure 3: CARTA Connection set up using protocol buffers.

connection using a unique session ID. If the session is valid then the server creates an instance of the *Session* class which will be used to handle all requests for a particular client in a session.

The CARTA interface allows the client to see information about a file before opening it, this allows the user to see information such as the number of headers in the file. For simplicity, this project only focused on the primary HDU of all files which is the default when no HDU is specified. Once the register viewer is acknowledged by the server, the client is then prompted to open a file, figure 4 shows the protocol when the user does this. First the client will send an open file request to the server, for the scope of this project the image file will already be on the server end thus the user only has to specify the relative directory and file name. If the file is found on the server side, an open file acknowledgment protocol is used. This will include details for the FileInfo and FileInfoExtended submessages. Though this part is not actually reading the image data, it is still required to open the file using the cfitsio [16] library. The *Session* class creates an instance of *FitsReader* which is used to get relevant information from the file. Immediately after the open file acknowledgment is sent, CARTA computes the channel histogram data and sends it through, the functionalitiy of the Raftlib library is exploited here by creating separate kernels for the different features.

Three main features from the CARTA backend was implemented using Raftlib. The region statistics gathers the sum, mean, total pixels, standard deviation, min and max. The channel histogram gathers the number of bins, bin width, first bin center and the actual bins. The cube histogram gathers the same results as the channel histogram but for the entire cube. The calculations will happen on the entire image region to ensure that the largest possible calculations are performed.

3.2 Features

3.3 Region Statistics

The region statistics calculation makes use of four custom raft kernels that extends the base raft::kernel class, namely *RaftReadImage, SplitVector, RaftStatistics and Sum*. Figure 5 shows the Directed

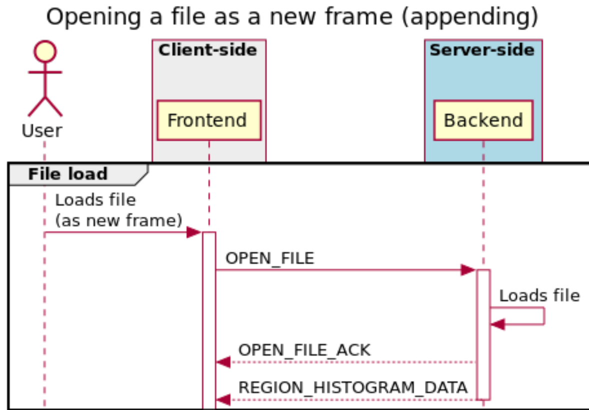

Figure 4: CARTA Opening a file using protocol buffers.

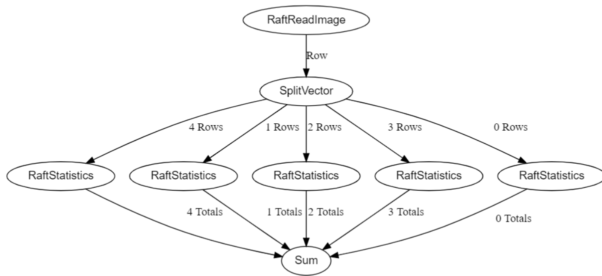

Figure 5: Statistics Directed Acyclic Graph.

Acyclic Graph of the kernels in execution. The *RaftReadImage* is responsible for sequentially reading a row at a time using the cfitsio library. Memory is allocated for a row using 'NAXIS2' value that is provided in the HDU. Once the row has been read, there is a check to eliminate any -nan pixels from the vector that is released down the stream. This is done to calculate the mean and standard deviation accurately using all non nan pixels. The *SplitVector* kernel is responsible for streaming a number of rows to its subsequent kernels. This kernel is created by making use of the pop_range method provided by Raftlib which will pop a specified number of rows from the FIFO buffer into a std::vector. The motivation behind implementing as such is to give the user the ability to stream multiple rows together to the next kernel as opposed to just one. If memory is not a concern then a user may increase the NUM_VECTORS variable allowing more data to be streamed at a time through a port.

The constructor of *SplitVector* takes in a parameter that is responsible for its number of output ports. When the split operator (<=) is used it will map each output port of SplitVector into the duplicates of the RaftStatistics kernels, which are created during runtime. This is how Raftlib allows for explicit parallelisation by linking a number of kernels together as a single pipeline stage within a streaming graph [5]. The only rule when it comes to parallelising kernels is to ensure that state should only be accessed from within a kernel or streamed in via ports. The *RaftStatistics* kernel is where the actual statistics computations take place, this kernel is duplicated as many times as there is output ports for *SplitVector*, a copy constructor is required so that the Raftlib CLONE() macro can duplicate the object. This kernel will calculate the sum and sumsquare for the data vector that has been streamed in through its input port, it will also calculate the minimum and maximum values by comparing it to a local copy for each duplicated kernel. These four values is added to a vector and is passed through the output port into the input port of kernel *Sum*.

The *Sum* Kernel is the final kernel in the RaftStatistics calculation. The same number of input ports as output ports specified for *SplitVector* is set during construction of the kernel, this is the final step to ensure that the middle kernel is duplicated and all its outputs are linked to the kernel using the join operator (>=). This run method of the sum kernel loops through its input ports and makes use of the size() method to check whether there is any data in the receiving port. What this kernel does is add the sum and sumsquare to its constructed global variables which can be accessed once the graph is finished execution in order to perform calculations such as mean and standard deviation. The *Raftlib* class makes use of the operator overloads to link the kernels together and execute the graph. The linking is as follows: *RaftReadImage » SplitVector <= RaftStatistics >= Sum.*

3.4 Channel Histogram

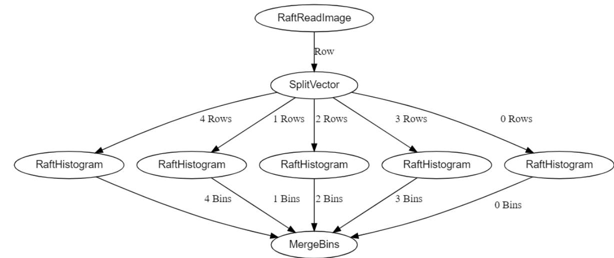

Figure 6: Histogram Directed Acyclic Graph.

The channel histogram calculations also make use of four custom raft kernels that extends the base raft::kernel class. Since we want to eliminate the need to store the entire data in memory, the same RaftReadImage and SplitVector kernels from the the region Statistics is used in order to re-read the data from the file and stream the rows through the ports. The difference is in the *RaftHistogram* and *MergeBins* kernels. Similar to the Statistics, *RaftHistoram* is duplicated the number of times that there is output ports for SplitVector. Prior to creating an object of this class, we first calculate the number of bins using the dimensions of the image to ensure that histogram is calculated for the entire region. The bin width is calculated using the minimum and maximum values computed from the statistics. A bins vector is also initialised with a size of the number of bins and will be passed into the constructor of *MergeBins*. The histogram constructor takes in the mean, minimum, bin width and the number of bins. Since this kernel is cloned, the copy constructor will create a local histogram for each duplicate kernel. The data vector that is streamed in will be looped and the bin position for each pixel is incremented in the local bins. This vector is then streamed into

the final kernel MergeBins. The local bins copy of each duplicate kernel is then reset back to zero.

MergeBins is similar to the sum kernel in the sense that it will be used in conjunction with the join operator; meaning that there is as many input ports as there is duplicates of the middle kernel. This kernel has a reference to the main bins vector that was passed in through the constructor. During execution, as the kernel receives inputs through its ports it will add the bin values from the vectors to the main referenced bin vector. Once again, the linking of the kernels happens in the CalculateHistogram method of the *Raftlib* class and the map is executed. The linking is as follows *RaftReadImage « SplitVector <= RaftHistogram >= MergeBins.*

3.5 Cube Histogram

The cube histogram is the same as the channel histogram calculation but will be for the entire cube. The existing code and the above mentioned kernels for channel histogram is compatible to calculate this. This is because the code was designed to work with either 2D or 3D imagery. The current implementation reads the header of a FITS file and checks for a depth value, if it is not found then the depth is set to 1, meaning that it is a 2D image. This is otherwise set to the depth value, which is beneficial because in the RaftReadImage kernel, we loop through the depth followed by the rows, then send each row down the stream. This means that the channel histogram and the cube histogram can be calculated in the same manner. We are currently parallelising the calculations in each channel, one suggested method was to do this and also to parallelise the reading of each channel. This would mean we would result in a graph that looks like *ReadEachChannel <= SplitVector <= RaftHistogram >= MergeBin*. This means that SplitVector and RaftStatistics would be duplicated a number of times and each output of SplitVector would go to its corresponding Histogram Kernel and the final outputs would be merged in MergeBins. However, it was found that Raftlib currently doesn't explicitly support this feature. This is because the map execution is processed from left to right and since the *ReadEachChannel <= SplitVector* returns a type kpair, there is no operator overload for the split operation that supports the first argument to be of type kpair.

3.6 Parallel Reading

The parallel reading is an extension to the original features in the scope. Currently one of the main bottlenecks of the CARTA system is the sequential reading of data from the files. When it comes to scenarios such as cube histogram, it will take extremely long to process certain file sizes since the data wont be in memory. More about this is explained in the following section. Since it is beneficial for us to try implement the parallel reading in Raftlib, a simple but not yet complete implementation of this feature is currently in the developmental branch of the project folder. Figure 7 shows the Directed Acyclic Graph of parallel reading and the region histogram calculation for this code. In the cube histogram we spoke about the inability of Raftlib to perform a double split operations together, to circumvent this issue two graphs were combined together using the end and start kernels of each respectively. A kernel *RaftReadingRow* was created and is used to stream a number range through its multiple output ports. The kernel receives the number of rows in

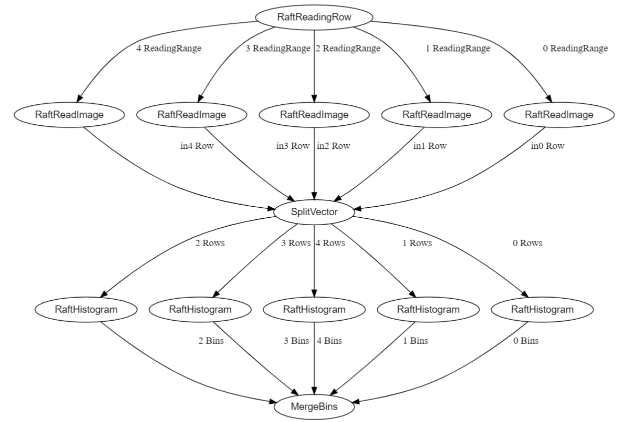

Figure 7: Parallel Reading + Histogram Directed Acyclic Graph.

the image and the number of output ports through its constructor. Before the streaming begins, a range calculation is performed using the number of rows and output ports. Each port will receive a start and end number which will be used by *RaftReadImage* to read the slice of data from the file. The *RaftReadImage* kernel is only slightly modified because now it requires a copy constructor and the clone macro since this kernel is duplicated. The *SplitVector* is slightly modified as it now has to construct multiple input ports and receive the data in parallel as opposed to sequentially receiving the data. Previously, Splitvector would loop through the outputs and if there is data on the input port then it will send data through, the new version does the same but performs a mod calculation to ensure that data from all input ports have been read. This is done to ensure that the graph is continuously streaming instead of waiting for all the inputs and then looping through the outputs and sending them.

4 TESTING AND EVALUATION

4.1 Testing Environment

The initial project scope was to test the application on multiple nodes on the ilifu cloud computing system for data intensive research [12]. However, the creator of the library had previously stated that the distributed computing code for Raftlib is currently not open sourced, but the implementation uses a very basic TCP stack that runs a process on each node that it is required to run on. It is possible that a future version of this library will support distributed computing. Thus both CARTA and the implementation of Raftlib was tested on a single node on the ilifu system using a Linux Ubuntu 20.04.3 LTS virtual machine. The virtual machine has 8 cores, 64GB RAM and runs on the intel xeon 85 processor. Even though we were allocated a few distributed disks to store the data, the performance for all I/O operations should be similar but one should expect some slight fluctuations.

4.2 Testing Features

If the reasoning for not loading the entire image into memory was not motivated well previously it is important to know why CARTA prefers this. Usually clusters can hold large chunks of data

in memory for long periods of time, however this is usually the case when multiple clients are working on a single data source. CARTA aims to allow multiple clients to work with multiple data, and is also designed to run on a single user laptop or a data center thus loading entire images into memory for every client would not be a practical solution.

Region Statistics, Channel Histogram, and Cube Histogram were the three main features that were tested on the virtual machine, Figures 5, 6 shows the Directed Acyclic Graphs for the features when in execution. Only the image from the primary HDU is tested for every file. These features were tested on the entire region as this will be the most computations that can happen at any point in time. The number of bins is also set to the default value that CARTA uses which is:

$$\sqrt{width * height}$$

This is also the same default for a cube histogram where only the width and height are used in calculating the number of bins.

4.3 Testing For Efficacy

The efficacy test is done in order to ensure that the results the program outputs will be correct and valid at all times. This is done using unit tests in the application specifically the Google C++ test framework [19] was used. We used four different FITS files for the unit tests, two files were provided by the supervisors and the other two were created using the Gaussian noise FITS image generator [9]. The actual results for each file was gathered using the CARTA interface by enabling the log events and opening the file. The statistics and histogram values were copied into the unit test file. An additional bins text file for each tested file was also created to compare the individual values of the histogram. The actual code base was modularised such that an instance of *FitsReader* can be created independently and the protocol buffers containing the program results can be called separately. The results was compared against each other using a 0.01% margin of error to account for floating point arithmetic inconsistencies. The testing files tried to cover majority of the possible data, there was a small 2D file, a cube file containing random nans, a medium 2D file and an extremely large 10GB file. The 10GB file was tested to ensure that there were no loose 32bit variables which would have affected the results of a 50000x50000 image as opposed to a 45000x45000 image.

4.4 Testing for Efficiency

Testing for efficiency is an important test to confirm whether the Raftlib library is worth implementing to replace certain features in the CARTA backend. To conduct this experiment as fairly as possible, there are a number of conditions and scenarios that have to be understood. It was decided for this project that the region statistics, channel histogram and cube histogram will be tested. Unfortunately the extended feature "parallel read" will not be tested due to its current concurrency issue that is yet to be solved. The Raftlib system works as follows: when a user sends an open file request the region statistics is calculated first. Immediately after this, the region histogram calculation is performed, both these method will read the entire image from the disk. The results for both these computations are stored in protocol buffers on the *FitsReader* object. As shown in figure 4 the histogram is sent immediately after the open file response. The region statistics data is only sent through when there is a SET_STATS_REQUIREMENT request, since the statistics values are already computed, the time here is irrelevant. The Raftlib cube histogram calculation works exactly the same way as it makes use of the same methods.

CARTA was installed on the ilifu cluster and the backend was connected to our frontend python client. The CARTA server was initiated using the command *carta –verbosity 5 –log_performance* to enable performance debug logs. The CARTA system works as follows: for a 2D image, CARTA will read an entire channel slice into memory from the disk. The *basic stats* is calculated using the channel cache, the minimum and maximum values will be used to calculate the histogram by looping through the cached memory again. The open file acknowledgement and region histogram data is sent to the front end. The performance logs here will display the time is takes to load the image to cache and the time it takes to compute the region histogram data. The user will then select the region statistics option which will send a SET_STATS_REQUIREMENT request. The Casacore [22] library is then used by CARTA to compute statistics data. It is important to note that Casacore will read the entire file again from disk in order to compute the region statistics data. However, since the image was already read from the disk recently it will be much faster than reading for the first time. The time it takes to compute this will also be displayed. The region statistics that Casacore computes will contain a handful of excess values that are not calculated in Raftlib. The cube histogram is only calculated when the SET_HISTO_REQUIREMENT has a region id of -2. The CARTA backend will then calculate histogram over the entire cube by reading each channel from disk. The Raftlib histogram time will have to be multiplied by two because CARTA goes through data twice, firstly to calculate the minimum and maximum values and then the histogram data.

Since the Raftlib implementation does not make use of an initial basic stats compute, a fair comparison would be to run the program twice with the same file and compare the region statistics time of Raftlib against the Casacore statistics time. This will ensure that both programs are exploiting the disk cache because the data was already read once from disk. The channel histogram computation will always be using data from memory in CARTA, the read time for the Raftlib implementation will not be used in the comparison to ensure that the histogram comparison is fair. Finally, for Cube Histogram, it was noticed that the virtual machine was producing unusual behaviours when reading data from the disk. The other comparisons are not affected by this because they were exploiting the disk and memory cache. A disk ram was thus created and the files were loaded from here. Every file was tested by clearing the file system cache, creating a file (this ensured that the disk cache will be exploited by Raftlib in the first run), test Raftlib and finally test CARTA. Files with image pixels ranging from 25 megapixels to 3600 megapixels were tested. All recorded time is in seconds.

5 RESULTS AND DISCUSSION

5.1 Statistics Histogram

Figure 8 tabulates the Raftlib *total time*, sequential read time, statistics compute time and the CARTA casacore statistics time. Figure 9

Region Statistics 2D Results					CARTA Performance logs
Size	Megapixels	Total Time	Read Time	Stats Time	(Read+StatsCompute)
5000x5000	25	0,19074	0,151634	0,039106	0,199054
10000x10000	100	0,711865	0,613133	0,098732	0,711581
15000x15000	225	1,5191	1,29777	0,221328	1,491807
20000x20000	400	2,27217	1,95335	0,318817	2,930523
25000x25000	625	3,22784	2,68992	0,537919	4,205752
30000x30000	900	4,53414	3,71309	0,821047	5,72977
35000x35000	1225	7,86157	6,64035	1,22122	8,40015
40000x40000	1600	8,75448	7,59075	1,16373	10,697664
45000x45000	2025	10,3881	8,89966	1,48842	13,735692
50000x50000	2500	11,9196	10,0642	1,85541	16,887703
55000x55000	3025	15,3503	12,9755	2,37477	19,441191
60000x60000	3600	17,9945	15,233	2,76152	24,191909

Figure 8: CARTA v Raftlib Statistics Table.

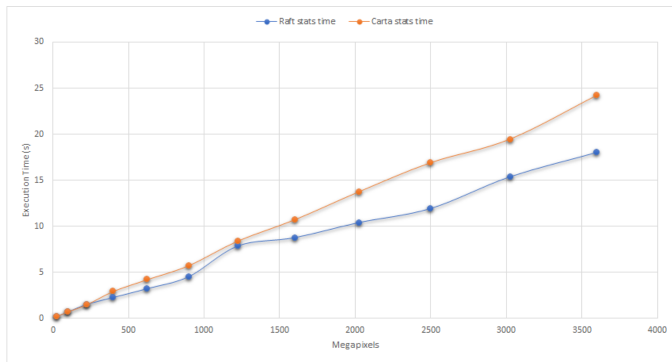

Figure 9: CARTA v Raftlib Statistics Graph.

is the graph showing the region statistics calculation using Raftlib versus CARTA. The graph shows that the Raftlib computation of statistics performs slightly better than the casacore implementation. It is evident that for greater file sizes, the Raftlib implementation increases linearly but still performs better than CARTA. One probable cause for this is because cassacore statistics was written for a more general approach where statistics calculations would be needed for any part of the image. Another reason could be that because only a row is streamed at a time, the actual computation is very quick, meaning that the streaming is not halted at any point because of filling up the buffer size. This can perhaps be considered as a positive outcome and can provide as foundation for CARTA to replace the casacore library to perform simple calculations by exploiting the streaming advantages.

5.2 Channel Histogram

Figure 10 tabulates the Raftlib total time, sequential read time, *channel histogram compute time* and the CARTA channel histogram time. Figure 11 is the graph showing the channel histogram using Raftlib versus CARTA. Both, the graph and the tables shows that Raftlib histogram was slower than the CARTA histogram. As explained in section 3.4, each duplicate kernel will calculate its own histogram when data is streamed through its port. The slower time could be a result of resetting the local histogram back to 0s in the duplicate kernels. Additionally, the mergeBins kernel loops through each histogram that is streamed in, while adding the values to the respective indexes of the global histogram vector. A different approach could be implemented and tested to see whether this would make a difference. It is also possible to increase the number of rows which are streamed in at a time. This will reduce the number of histograms that will have to be computed and streamed, keeping in mind the overhead to perform the same computations on a bigger vector.

Region Histogram 2D Results					CARTA Performance logs
Size	Megapixels	Total Time	Read Time	Histo Time	HistoCompute
5000x5000	25	0,20669	0,162172	0,044518	0,026264
10000x10000	100	0,595846	0,5111	0,084746	0,068421
15000x15000	225	1,38236	1,1206	0,261762	0,133231
20000x20000	400	2,28987	1,90825	0,381619	0,225904
25000x25000	625	3,31974	2,76874	0,550996	0,356978
30000x30000	900	4,66182	3,8454	0,816415	0,5055
35000x35000	1225	6,48464	5,42946	1,05519	0,667475
40000x40000	1600	7,61724	6,46048	1,15677	0,891584
45000x45000	2025	10,3552	8,76405	1,59119	1,118076
50000x50000	2500	12,7009	10,7464	1,95449	1,363033
55000x55000	3025	15,1527	12,8952	2,25741	1,648768
60000x60000	3600	17,2349	14,5929	2,642	1,980318

Figure 10: CARTA v Raftlib Histogram Table.

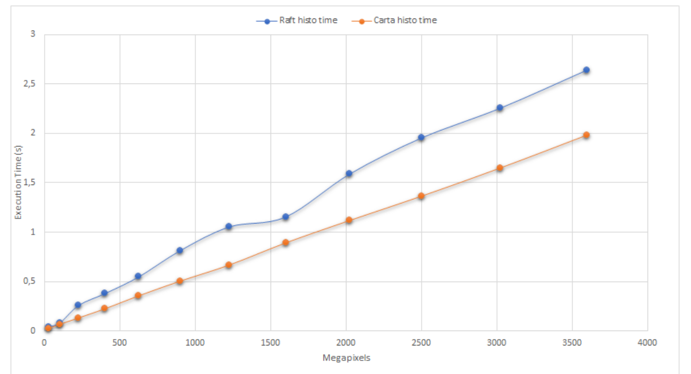

Figure 11: CARTA v Raftlib Histogram Graph.

5.3 Cube Histogram

Cube Histogram 3D Results						CARTA Performance logs
Size	Megapixels	Total Time	Read Time *2	Read Time	Histo Time	Read+HistoCompute
250x250x250	15,625	0,630384	0,4611	0,23055	0,169284	0,383128
500x500x250	62,5	2,079848	1,718944	0,859472	0,360904	1,12559
750x750x250	140,625	4,594215	3,89824	1,94912	0,695975	2,067949
1000x1000x250	250	6,804832	6,07824	3,03912	0,726592	3,393014
1250x1250x250	390,625	8,163866	7,46088	3,73044	0,702986	4,904003
1500x1500x250	562,5	11,37291	10,51678	5,25839	0,85613	7,181119
1750x1750x250	765,625	17,87781	16,35686	8,17843	1,52095	9,428738
2000x2000x250	1000	19,90947	17,98774	8,99387	1,92173	11,528804
2250x2250x250	1265,625	22,16545	20,1802	10,0901	1,98525	15,001133
2500x2500x250	1562,5	26,44298	23,9468	11,9734	2,49618	19,690254
2750x2750x250	1890,625	32,16957	28,9192	14,4596	3,25037	23,262502
3000x3000x250	2250	36,76033	33,4738	16,7369	3,28653	32,974548
3250x3250x250	2640,625	40,59294	36,8948	18,4474	3,69814	41,773345
3500x3500x250	3062,5	42,05489	36,10978	18,05489	5,94511	49,118

Figure 12: CARTA v Raftlib Cube Histogram Table.

Figure 12 tabulates the Raftlib total time, sequential read time, *cube histogram compute time* and the CARTA cube histogram time. Figure 13 is the graph showing the cube histogram using Raftlib

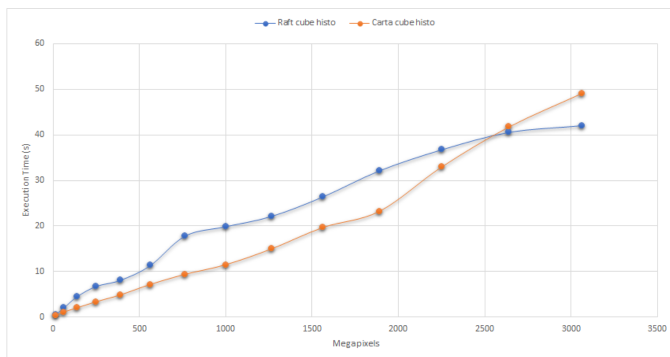

Figure 13: CARTA v Raftlib Cube Histogram Graph.

versus CARTA. The read times shown for both CARTA and Raftlib are the read times of the files from a ramdisk. CARTA goes through the entire data twice to calculate cube histogram. Raftlib also does this but calculates the entire statistics in the first run. Thus the read time for the histogram was multiplied by two for the Raftlib time to achieve the most fair comparison. This is not the most exact comparison, but it is assumed that the reading of the histogram twice is equivalent to multiplying one iteration by two. At the time of writing this paper, it is unknown whether the CARTA implementation optimises cube histogram calculation when reading the data twice. Similar to the channel histogram, CARTA seems to perform much better than the Raftlib implementation, this can be due to the same reasons. However, as the files sizes get larger, it can be seen that Raftlib starts to perform slightly better. It may be beneficial to set up a more fair comparison and test for extremely large file sizes. It is also important to note that the depth remained as a constant value of 250 while the channel size was increased for each tested file.

5.4 Final Discussion and Future Work

It can be seen from the above results and discussion that the current CARTA system outperformed the Raftlib implementation for both channel and cube histogram data. The region statistics performed better using the Raftlib implementation. Overall, this paper is confident that a fair experiment was conducted given the conditions in which it was tested in. However, the potential bottlenecks that were mentioned could be revised to possibly yield better results. Currently a row is being read at a time from the FITS file, each row has to be copied into a vector which will be passed down the stream. This computation itself is contributing to majority of time consumption during reading. One important observation is the sequential reading time from the first kernel. This is one of the main bottlenecks that causes a reduction in computation speed. As mentioned previously, the parallel read could be fully implemented in a future version of this program. It is almost guaranteed that successfully implementing this would result in faster execution times in a streaming graph. We can also find an optimal NUM_VECTOR value using the file size to move more data down the stream at a time. Another option would be to read bigger chunks of data from the file at a time or even entire channels for cube histogram calculations. The initial project scope hoped to test the library on multiple nodes, however it was found that the library currently doesn't support prime time distributed computing, this could also be an extension of this project when the library allows this feature. Nevertheless, one could still use the Message Parsing Interface (MPI) [3] to allow execution on multiple nodes.

6 CONCLUSIONS

We implemented a set of logically simple backend features that CARTA uses using the Raftlib library in c++. The prototype that was developed made use of the same protocol buffer messages that is used by the CARTA application. By doing so, we were able to set up a fair comparison to measure the correctness and performance of both implementation since they were both compatible with our frontend.

The Raftlib library allowed the implementations of the features using the dataflow architecture through stream processing. Although the results do not point highly in favour of the library, the CARTA region statistcs computation currently makes use of the casacore library, and we saw that the Raftlib implementation performed better than this. It can be concluded that it would be worth re-implementing this feature using Raftlib along with any other features that may be computationally inexpensive but needs a good portion of the image data.

The region histogram results did not show to be better than its CARTA equivalent. This can be due to the implementation strategy that was explained above. We can conclude that a different approach such as streaming more than a single row could be a more efficient solution. It is also seen that the performance of Raftlib becomes more of a constant as the file sizes increase for cube histogram while CARTA continuous to increase linearly. It could be possible that the streaming application performs better for extremely large file sizes.

One feature that is mentioned throughout the paper is the parallelisation of the reading of the FITS file. This feature was tackled as an extension to this research to see whether there would be an increase in performance. It would certainly be beneficial to complete the implementation of this and test the hypothesis. Currently, the lack of assistance, documentation, and examples is a limiting factor in creating a functioning code. It can be viewed in the developmental branch of the project repository.

There is a slight learning curve when it comes to understanding how the library works. Even though major components of the library is completed, it is still being developed and features such as multi node execution or certain split overloads are not available. There is no official documentation on the library other than the wiki that is provided in the Raftlib github. However, the wiki does provide some detailed explanations of the different methods that the input or output ports can use. One potent advantage of Raftlib that has been observed is that it gives the user the ability to write code within a kernel without having to take into account low level details of resources mapping or allocation. The linking operators also allow the user to not worry about how the data is processed and where certain locks need to be placed. Once the data from the input ports were captured, the algorithm implementation could be thought of as a sequential application. The map object allows

the runtime to choose whether to execute the kernels in separate processes, threads, thread pools, or fibers.

Given the above, it can be concluded that a dataflow architecture is indeed feasible to replace certain features in the CARTA backend. The simplicity of the Raftlib interface allows the programmer to think in a sequential manner whilst still implementing parallel programming. This makes the implementation of kernels simpler. Thus, to simply answer the research question, a dataflow architecture, specifically this library can be used to replace specific features in the CARTA backend to increase performance.

The entire system that was developed for this project can be viewed on the GitHub page[1]. The unit test files can be requested from the author since they cannot be uploaded due to the file size.

ACKNOWLEDGMENTS

The author acknowledges that this work made use of the CARTA (Cube Analysis and Rendering Tool for Astronomy) software.

The author acknowledges that this work made use of RaftLib: A C++ Template Library for High Performance Stream Parallel Processing.

The author acknowledges 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 acknowledges and would like to thank the three supervisors, Professor Rob Simmonds, Dr. Angus Cormie, and Ms Adrianna Pińska for showing great interest and providing support and help throughout the duration of the entire project.

The author would like to thank and acknowledge Angus Pearce (who is doing the complement of this project using a dataflow library known as Daliuge) for the collaboratory work done for components of this research.

The author would like to acknowledge Dylan Fouche for the previous work provided that made use of the DASK library and comparing it against CARTA.

REFERENCES

[1] William B Ackerman. 1979. Data flow languages. In *1979 International Workshop on Managing Requirements Knowledge (MARK)*. IEEE, 1087–1095.

[2] Adrianna Pinska Angus Comrie, Rob Simmonds. 2020. CARTA Interface Control Document. (2020). https://carta-protobuf.readthedocs.io/en/latest/index.html

[3] Brandon Barker. 2015. Message passing interface (mpi). In *Workshop: High Performance Computing on Stampede*, Vol. 262.

[4] Jonathan Beard. 2017. Simplifying parallel applications for C++, an example parallel Bzip2 using RaftLib with performance... https://medium.com/cat-dev-urandom/simplifying-parallel-applications-for-c-an-example-parallel-bzip2-using-raftlib-with-performance-f69cc8f7f962

[5] Jonathan Beard. Cloned:08-2021. Raftlib/Raftlib: The RAFTLIB C++ library, streaming/dataflow concurrency via C++ iostream-like operators. https://github.com/RaftLib/RaftLib

[6] Jonathan C Beard, Peng Li, and Roger D Chamberlain. 2017. Raftlib: A C++ template library for high performance stream parallel processing. *The International Journal of High Performance Computing Applications* 31, 5 (2017), 391–404.

[7] RS Booth and JL Jonas. 2012. An overview of the MeerKAT project. *African Skies* 16 (2012), 101.

[8] David B Davidson. 2014. The SKA and the MeerKAT precursor—Extreme antenna engineering. In *The 8th European Conference on Antennas and Propagation (EuCAP 2014)*. IEEE, 1216–1219.

[9] Inter-University Institute for Data Intensive Astronomy (IDIA). Date Accessed: 09-2021. Gaussian noise FITS image generator. (Date Accessed: 09-2021). https://github.com/idia-astro/image-generator#readme

[10] Dylan Fouché and Zainab Adjiet. 2020. Prototyping a Dataflow Implementation of the CARTA System. (2020).

[11] Jeff Gilchrist. 2004. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, Vol. 16. Citeseer, 559–564.

[12] ilifu. Accessed: 09-2021. Cloud computing for data-intensive research. (Accessed: 09-2021).

[13] Wesley M Johnston, JR Paul Hanna, and Richard J Millar. 2004. Advances in dataflow programming languages. *ACM computing surveys (CSUR)* 36, 1 (2004), 1–34.

[14] Jeff Magnum. Viewed:09-2021. How does a radio interferometer work? https://public.nrao.edu/ask/how-does-a-radio-interferometer-work/

[15] JP McMullin, BSDYWGK Waters, D Schiebel, W Young, K Golap, RA Shaw, F Hill, and DJ Bell. 2007. Astronomical Data Analysis Software and Systems XVI. In *ASP Conf. Ser*, Vol. 376. 127.

[16] William D Pence. 2010. CFITSIO: a FITS file subroutine library. *Astrophysics Source Code Library* (2010), ascl–1010.

[17] Simon Perkins, Jacques Questiaux, Stephen Finniss, Robin Tyler, Sarah Blyth, and Michelle M Kuttel. 2014. Scalable desktop visualisation of very large radio astronomy data cubes. *New Astronomy* 30 (2014), 1–7.

[18] Danny C Price, Benjamin R Barsdell, and Lincoln J Greenhill. 2015. HDFITS: Porting the FITS data model to HDF5. *Astronomy and Computing* 12 (2015), 212–220.

[19] Arpan Sen. 2010. A quick introduction to the Google C++ Testing Framework. *IBM DeveloperWorks* 20 (2010), 1–10.

[20] Bjarne Stroustrup. 2003. The C++ Standard: Incorporating Technical Corrigendum No. 1. *British Standards Institute* (2003).

[21] AR Taylor, A Comrie, and A Pińska. 2020. Development and application of an HDF5 schema for SKA-scale image cube visualization. (2020).

[22] Casacore Team. 2019. casacore: Suite of C++ libraries for radio astronomy data processing. *Astrophysics Source Code Library* (2019), ascl–1912.

[23] K-S Wang, A Comrie, P Harris, A Moraghan, S-C Hsu, A Pinska, C-C Chiang, H Jan, R Simmonds, Q Pang, et al. 2020. CARTA: Cube Analysis and Rendering Tool for Astronomy. In *Astronomical Society of the Pacific Conference Series*, Vol. 527. 213.

[24] Alwyn Wootten. 2003. The Atacama large millimeter array (ALMA). In *Large ground-based Telescopes*, Vol. 4837. International Society for Optics and Photonics, 110–118.

[1] The system, documentation and ReadMe can be found on https://github.com/anguspearce/CompDaF.

A RAFTLIB 2D RESULTS

—————————————-

Num of threads: 5
5000 5000 1
Raft Total Statistics Time: 0.19074s
Raft Read Image Time: 0.151634s
Raft Statistics Time: 0.039106s
Finsihed Raft stats through reading in raft
5000 5000 1
Raft Total Histogram Time: 0.20669s
Raft Read Image Time: 0.162172s
Raft Histogram Time: 0.044518s
Finsihed Raft Histo through reading in raft
—————————————-
Listening on port 9001
Num of threads: 5
10000 10000 1
Raft Total Statistics Time: 0.711865s
Raft Read Image Time: 0.613133s
Raft Statistics Time: 0.098732s
Finsihed Raft stats through reading in raft
10000 10000 1
Raft Total Histogram Time: 0.595846s
Raft Read Image Time: 0.5111s
Raft Histogram Time: 0.084746s
Finsihed Raft Histo through reading in raft
—————————————-
Num of threads: 5
15000 15000 1
Raft Total Statistics Time: 1.5191s
Raft Read Image Time: 1.29777s
Raft Statistics Time: 0.221328s
Finsihed Raft stats through reading in raft
15000 15000 1
Raft Total Histogram Time: 1.38236s
Raft Read Image Time: 1.1206s
Raft Histogram Time: 0.261762s
Finsihed Raft Histo through reading in raft
—————————————-
Num of threads: 5
20000 20000 1
Raft Total Statistics Time: 2.27217s
Raft Read Image Time: 1.95335s
Raft Statistics Time: 0.318817s
Finsihed Raft stats through reading in raft
20000 20000 1
Raft Total Histogram Time: 2.28987s
Raft Read Image Time: 1.90825s
Raft Histogram Time: 0.381619s
Finsihed Raft Histo through reading in raft
—————————————-
Num of threads: 5
25000 25000 1
Raft Total Statistics Time: 3.22784s
Raft Read Image Time: 2.68992s
Raft Statistics Time: 0.537919s
Finsihed Raft stats through reading in raft
25000 25000 1
Raft Total Histogram Time: 3.31974s
Raft Read Image Time: 2.76874s
Raft Histogram Time: 0.550996s
Finsihed Raft Histo through reading in raft
—————————————-
Num of threads: 5
30000 30000 1
Raft Total Statistics Time: 4.53414s
Raft Read Image Time: 3.71309s
Raft Statistics Time: 0.821047s
Finsihed Raft stats through reading in raft
30000 30000 1
Raft Total Histogram Time: 4.66182s
Raft Read Image Time: 3.8454s
Raft Histogram Time: 0.816415s
Finsihed Raft Histo through reading in raft
—————————————-
Num of threads: 5
35000 35000 1
Raft Total Statistics Time: 7.86157s
Raft Read Image Time: 6.64035s
Raft Statistics Time: 1.22122s
Finsihed Raft stats through reading in raft
35000 35000 1
Raft Total Histogram Time: 6.48464s
Raft Read Image Time: 5.42946s
Raft Histogram Time: 1.05519s
Finsihed Raft Histo through reading in raft
—————————————-
Num of threads: 5
40000 40000 1
Raft Total Statistics Time: 8.75448s
Raft Read Image Time: 7.59075s
Raft Statistics Time: 1.16373s
Finsihed Raft stats through reading in raft
40000 40000 1
Raft Total Histogram Time: 7.61724s
Raft Read Image Time: 6.46048s
Raft Histogram Time: 1.15677s
Finsihed Raft Histo through reading in raft
—————————————-
Num of threads: 5
45000 45000 1
Raft Total Statistics Time: 10.3881s
Raft Read Image Time: 8.89966s
Raft Statistics Time: 1.48842s
Finsihed Raft stats through reading in raft
45000 45000 1
Raft Total Histogram Time: 10.3552s
Raft Read Image Time: 8.76405s
Raft Histogram Time: 1.59119s
Finsihed Raft Histo through reading in raft
—————————————-
Num of threads: 5
50000 50000 1

Raft Total Statistics Time: 11.9196s
Raft Read Image Time: 10.0642s
Raft Statistics Time: 1.85541s
Finsihed Raft stats through reading in raft
50000 50000 1
Raft Total Histogram Time: 12.7009s
Raft Read Image Time: 10.7464s
Raft Histogram Time: 1.95449s
Finsihed Raft Histo through reading in raft
————————————————-
Num of threads: 5
55000 55000 1
Raft Total Statistics Time: 15.3503s
Raft Read Image Time: 12.9755s
Raft Statistics Time: 2.37477s
Finsihed Raft stats through reading in raft
55000 55000 1
Raft Total Histogram Time: 15.1527s
Raft Read Image Time: 12.8952s
Raft Histogram Time: 2.25741s
Finsihed Raft Histo through reading in raft
————————————————-
Num of threads: 5
60000 60000 1
Raft Total Statistics Time: 17.9945s
Raft Read Image Time: 15.233s
Raft Statistics Time: 2.76152s
Finsihed Raft stats through reading in raft
60000 60000 1
Raft Total Histogram Time: 17.2349s
Raft Read Image Time: 14.5929s
Raft Histogram Time: 2.642s
Finsihed Raft Histo through reading in raft

B RAFTLIB 3D RESULTS

————————————————-
Num of threads: 5
250 250 250
Raft Total Statistics Time: 0.339793s
Raft Read Image Time: 0.217901s
Raft Statistics Time: 0.121892s
Finsihed Raft stats through reading in raft
250 250 250
Raft Total Histogram Time: 0.399834s
Raft Read Image Time: 0.23055s
Raft Histogram Time: 0.169284s
Finsihed Raft Histo through reading in raft
————————————————-
Num of threads: 5
500 500 250
Raft Total Statistics Time: 0.955669s
Raft Read Image Time: 0.739118s
Raft Statistics Time: 0.216551s
Finsihed Raft stats through reading in raft
500 500 250
Raft Total Histogram Time: 1.22038s
Raft Read Image Time: 0.859472s
Raft Histogram Time: 0.360904s
Finsihed Raft Histo through reading in raft
————————————————–
Num of threads: 5
750 750 250
Raft Total Statistics Time: 2.52638s
Raft Read Image Time: 1.89342s
Raft Statistics Time: 0.632957s
Finsihed Raft stats through reading in raft
750 750 250
Raft Total Histogram Time: 2.64509s
Raft Read Image Time: 1.94912s
Raft Histogram Time: 0.695975s
————————————————–
Num of threads: 5
1000 1000 250
Raft Total Statistics Time: 3.27418s
Raft Read Image Time: 2.59799s
Raft Statistics Time: 0.676194s
Finsihed Raft stats through reading in raft
1000 1000 250
Raft Total Histogram Time: 3.76571s
Raft Read Image Time: 3.03912s
Raft Histogram Time: 0.726592s
Finsihed Raft Histo through reading in raft
————————————————–
Num of threads: 5
1250 1250 250
Raft Total Statistics Time: 5.1775s
Raft Read Image Time: 4.26887s
Raft Statistics Time: 0.908626s
Finsihed Raft stats through reading in raft
1250 1250 250
Raft Total Histogram Time: 4.43343s
Raft Read Image Time: 3.73044s
Raft Histogram Time: 0.702986s
—————————————————
Num of threads: 5
1500 1500 250
Raft Total Statistics Time: 7.96052s
Raft Read Image Time: 6.68656s
Raft Statistics Time: 1.27396s
Finsihed Raft stats through reading in raft
1500 1500 250
Raft Total Histogram Time: 6.11452s
Raft Read Image Time: 5.25839s
Raft Histogram Time: 0.85613s
Finsihed Raft Histo through reading in raft
————————————————–
Num of threads: 5
1750 1750 250
Raft Total Statistics Time: 9.22147s
Raft Read Image Time: 7.89963s
Raft Statistics Time: 1.32184s
Finsihed Raft stats through reading in raft
1750 1750 250

Raft Total Histogram Time: 9.69937s
Raft Read Image Time: 8.17843s
Raft Histogram Time: 1.52095s
—————————————–
Num of threads: 5
2000 2000 250
Raft Total Statistics Time: 9.84035s
Raft Read Image Time: 8.54186s
Raft Statistics Time: 1.29849s
Finsihed Raft stats through reading in raft
2000 2000 250
Raft Total Histogram Time: 10.9156s
Raft Read Image Time: 8.99387s
Raft Histogram Time: 1.92173s
Finsihed Raft Histo through reading in raft
—————————————–
Num of threads: 5
2250 2250 250
Raft Total Statistics Time: 11.1045s
Raft Read Image Time: 9.33453s
Raft Statistics Time: 1.76996s
Finsihed Raft stats through reading in raft
2250 2250 250
Raft Total Histogram Time: 12.0754s
Raft Read Image Time: 10.0901s
Raft Histogram Time: 1.98525s
—————————————–
Num of threads: 5
2500 2500 250
Raft Total Statistics Time: 12.282s
Raft Read Image Time: 10.4811s
Raft Statistics Time: 1.8009s
Finsihed Raft stats through reading in raft
2500 2500 250
Raft Total Histogram Time: 14.4696s
Raft Read Image Time: 11.9734s
Raft Histogram Time: 2.49618s
Finsihed Raft Histo through reading in raft
—————————————–
Num of threads: 5
2750 2750 250
Raft Total Statistics Time: 16.1832s
Raft Read Image Time: 13.4978s
Raft Statistics Time: 2.68544s
Finsihed Raft stats through reading in raft
2750 2750 250
Raft Total Histogram Time: 17.7099s
Raft Read Image Time: 14.4596s
Raft Histogram Time: 3.25037s
—————————————–
Listening on port 9001 Num of threads: 5
3000 3000 250
Raft Total Statistics Time: 19.6074s
Raft Read Image Time: 16.341s
Raft Statistics Time: 3.26643s
Finsihed Raft stats through reading in raft
3000 3000 250
Raft Total Histogram Time: 20.0234s
Raft Read Image Time: 16.7369s
Raft Histogram Time: 3.28653s
Finsihed Raft Histo through reading in raft
—————————————–
Listening on port 9001
Num of threads: 5
3250 3250 250
Raft Total Statistics Time: 21.6476s
Raft Read Image Time: 18.1115s
Raft Statistics Time: 3.53609s
Finsihed Raft stats through reading in raft
3250 3250 250
Raft Total Histogram Time: 22.1455s
Raft Read Image Time: 18.4474s
Raft Histogram Time: 3.69814s
—————————————–
Listening on port 9001
Num of threads: 5
3500 3500 250
Raft Total Statistics Time: 23.1142s
Raft Read Image Time: 18.59161s
Raft Statistics Time: 4.52259
Finsihed Raft stats through reading in raft
3500 3500 250
Raft Total Histogram Time: 24s
Raft Read Image Time: 18.05489
Raft Histogram Time: 5.94511s
Finsihed Raft Histo through reading in raft