Prototyping a Dataflow Implementation of the CARTA System

Dylan Fouché
University of Cape Town
Department of Computer Science
fchdyl001@myuct.ac.za

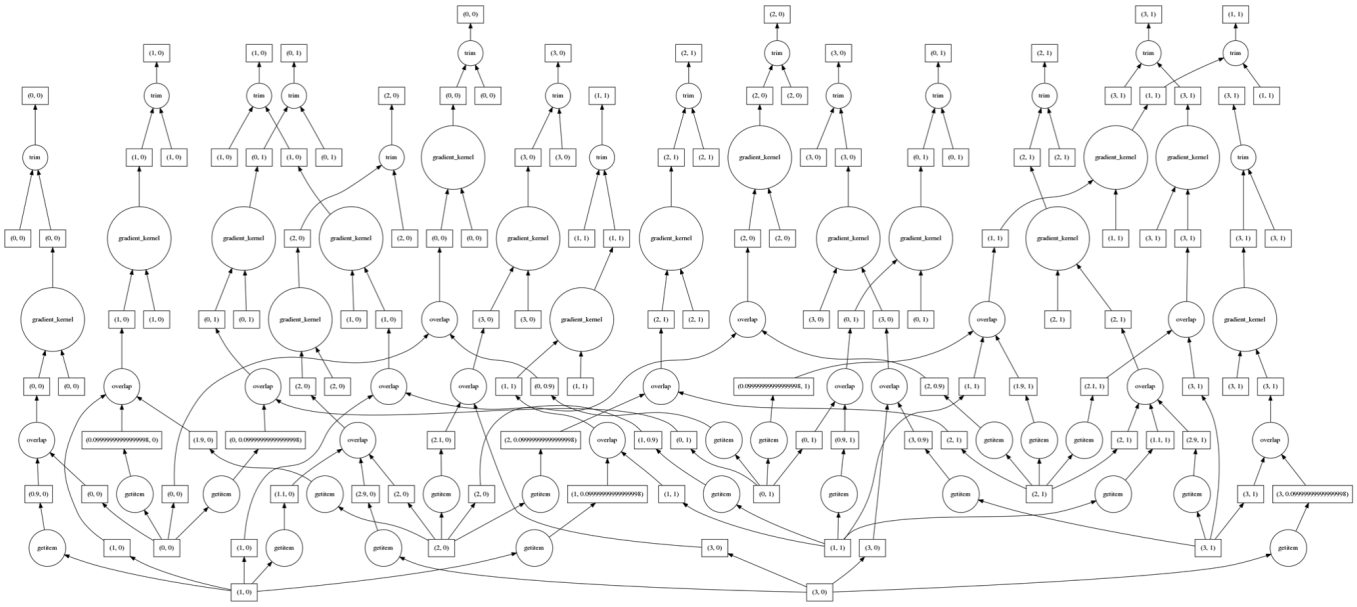

Figure 1: Dask task graph for the application of the gradient function to an image in distributed memory

ABSTRACT

The CARTA system is a tool designed to analyse and visualise large-scale astronomical imagery. The server component of the CARTA system is currently a multi-threaded C++ implementation but is undergoing a redesign to cope with exponentially increasing image sizes, and other architectures are being considered.

The dataflow architecture is a model of computation that describes functions as nodes on a directed acyclic graph that execute concurrently, while data travels asynchronously between these nodes. This model allows for scaling out to large heterogeneous distributed systems without having to consider the ordering of instructions since the lack of control flow implies there can never be race conditions or deadlocks.

In this paper, we investigate a dataflow implementation of the CARTA back-end using the Python library Dask. We implement prototype components of this dataflow back-end that would replace components of the existing CARTA back-end. We subject these prototype components to thorough testing to ensure correctness and to measure the performance and scalability of our solution.

We show that our dataflow implementation produces correct output, and analyse the data from our performance tests on a certain set of back-end functions. In the best case, Dask significantly outperforms CARTA with an approximate speedup of 10X. In the worst case, the two perform indistinguishably. We also note that the dataflow architecture offers more potential for future scaling-out than the current implementation.

Resultantly, we find that the dataflow model is a valid approach for re-implementing the CARTA back-end to ensure good performance and sufficient scalability as the computational demands on the system continue to increase.

CCS CONCEPTS

• **Software and its engineering** → *Software prototyping*; *Massively parallel systems*; *Distributed systems organizing principles*; • **Computer systems organization** → **Distributed architectures**; **Data flow architectures**.

KEYWORDS

dataflow architecture, distributed computing, radio astronomy, data visualisation, Python

1 INTRODUCTION

The Cube Analysis and Rendering Tool for Astronomy (CARTA) [10] is a tool designed to visualise and analyse data from the Atacama Large Millimetre Array (ALMA) [3], the National Radio Astronomy Observatory (NRAO) [25], and the Square Kilometre Array (SKA) [14] pathfinders.

CARTA uses a client-server model, with the back-end having access to large data cubes and performing computations over them while the front-end runs in a browser on the client machine for visualisation. This software is maintained by the Inter-University Institute for Data-Intensive Astronomy (IDIA) [17], among others. IDIA is a partnership of the University of Cape Town, the University of Pretoria and the University of the Western Cape.

As the quantity of data produced by modern telescope arrays increases rapidly, a robust and scalable solution is required to visualise and perform analysis on this imagery in near real-time. CARTA aims to meet this need with a multi-threaded imperative back-end implemented mostly in C++ [5]. While this approach is widely regarded as a highly performant solution, this software is currently undergoing an architectural redesign to better accommodate significantly larger images through a more distributed and scalable solution. Once such architecture being considered here is the dataflow model [13].

This model is becoming increasingly popular in interactive systems as we approach the exascale era of computing, with some suggesting that this is due to its simplicity, efficiency, and manageability [34]. Exascale systems are required to perform more than 10^{18} floating-point operations per seconds (FLOPS) to be able to process massive volumes of data.

There are many existing tools for dataflow computing at this scale, which are commonly applied for efficient batch processing of data on distributed systems. The *Dataflow* [19] service on Google Cloud that provides serverless stream and batch data processing is a good example of this. However, the CARTA system incorporates real-time interaction and computational steering into its use cases, and it is not clear whether existing dataflow tools will be able to accommodate this.

To aid in this evaluation, we develop prototype back-end components with the Python-based Dask [12] dataflow environment. These components will mimic the behaviour of the CARTA back-end for a certain set of functions and will undergo thorough testing to ensure that they behave correctly and determine their performance under various conditions.

The research hypothesis is that it is possible to adapt existing dataflow tools to handle this class of high-throughput interactive visualisation and analysis workloads in a performant and scalable manner.

2 RELATED WORK

The move towards exascale computation has opened up new possibilities in the scientific community, making many previously intractable problems now effectively computable. Yet, new software architectures and design paradigms are needed to leverage the processing power of these modern computers.

Shalf et al. [29] recognise that since it is mainly an increase in the number of processing cores that is driving the increase in processing power today, high-performance software must become increasingly parallel to benefit from this. Cappello et al. [6] reason that having more threads of execution in a program leads to more potential points of failure, thus modern architecture must be more resilient to traditional software errors.

Jack Dennis [16] presented the first concept of the dataflow architecture in 1974, which differs from the traditional von Neumann architecture in that there is no traditional program counter and deterministic execution ordering. Instead, we model our computation as a directed acyclic graph called a task graph. The nodes on this graph represent some function and will fire whenever input data becomes available. Data, or tokens, will travel along the edges of this graph from node to node.

Culler [13] described this model of computation as a "machine language for parallel machines", but it has since been adapted as a software design pattern not dissimilar to the pipe and filter model. The dataflow model offers efficient use of implicit fine-grain parallelism [22], and one can easily compose more complex programs by connecting the output of one graph to the input of another [15, 30], which presents a new way of dividing a program into distinct components which implement high cohesion and low coupling.

The asynchronous nature of this model lends itself to efficient implementations on highly distributed and heterogeneous systems, which are becoming an increasingly integral part of our modern high-performance computing infrastructure. Furthermore, the architecture does not allow for any global state, which eliminates side-effects [2] and thus there is no need to consider locking and other deadlock prevention strategies for this instance of implicit concurrency.

Verdosica et al. [35] suggest in their position paper that the dataflow model is indeed a valid approach for exascale computation, and many successful implementations of this nature have been identified. Silva et al. [31], for instance, demonstrated the efficiency of this approach for performing analysis on large raw-data files over distributed systems, which is a primary use case for the CARTA system.

Mao et al. [23], developed a graph-based dataflow model to schedule jobs over a highly distributed computing network to process the exascale throughput from the SKA. The authors built upon previous work such as the open-source Celery [32] scheduler developed for the MeerKAT telescope. Furthermore, several optimisation methods are proposed for this system including a meta-heuristic approach such as genetic algorithm optimisation, as well as other critical-path aware hierarchical scheduling algorithms.

Zhang et al. [38] demonstrated the vast scalability of dataflow systems by making use of the Amazon Web Services (AWS) EC2 cloud computing infrastructure. The authors use a graph-based scheduling technique, similar to that of Mao et al. [23]. Given this, the authors report a 3.7X speedup over a naive multi-threaded implementation in a C-type language.

3 DESIGN AND IMPLEMENTATION

3.1 Solution Architecture

We implemented a set of back-end components using Python and the Dask library [12] for dataflow computing. A decision was made to replicate the client-server model that the CARTA back-end currently employs such that the two systems can communicate with each other. This communication takes place via a shared repository of protocol buffer messages [7] that are sent and received sent over the TCP protocol.

The CARTA Interface Control Document (ICD) [9] defines the protocol by which these messages are exchanged. This protocol includes control messages, request messages and their corresponding acknowledgement messages, as well as data stream messages. For instance, the initial handshake protocol on connection is defined as follows: the client will send a *REGISTER_VIEWER* message and the server should respond with a *REGISTER_VIEWER_ACK*. This is shown in Figure 2.

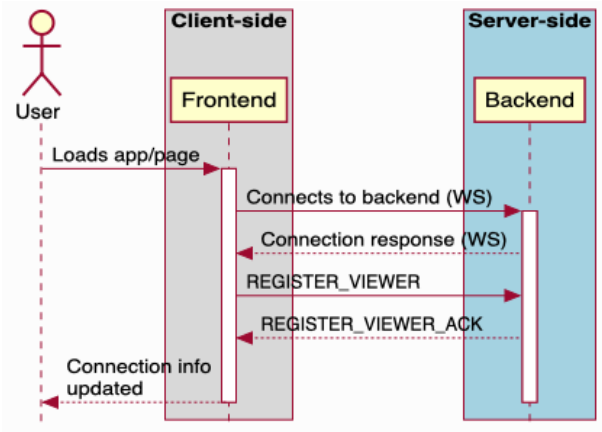

Figure 2: Initial connection message exchange

Our Python back-end will mimic the behaviour of the CARTA back-end on receipt of certain messages. The server will maintain an instance of the *Image* class that we define. This *Image* object reads in image data from disk, persists it in distributed memory, and provides the server with an interface for performing various computations over the image data. We also define a Python front-end which can connect to and exchange messages with both our Python back-end and the CARTA server. We implement a basic command-line interface for testing purposes, but an accompanying Jupyter [21] notebook is used for visualisation. This modified client-server architecture is shown as a component diagram in Figure 3.

3.2 Implementation Details

3.2.1 The Image class. The *Image* object maintained by our back-end uses Dask to store and perform computations on our image data. This class uses AstroPy [26] and h5Py [11] to abstract away from domain-specific implementation details, and to ingest *fits* and *hdf5* images efficiently. We store our image data in a member variable and provide functions to invoke various computations over our data such as *Image.get_std_dev()* or *Image.get_argmax()*. Note that these results will be cached, so a subsequent call to one of these functions will be much faster given the same parameters.

3.2.2 Dask collections and functions. The data in our *Image* class is stored in a Dask array. The *dask.array* class will chunk our image data into several NumPy [33] arrays. We let Dask determine the optimal chunk size, which is usually no less than 100MB per chunk. These Dask objects are lazy: they are stored as references to objects rather than actual values. This uses Python's *Future* [28] object to

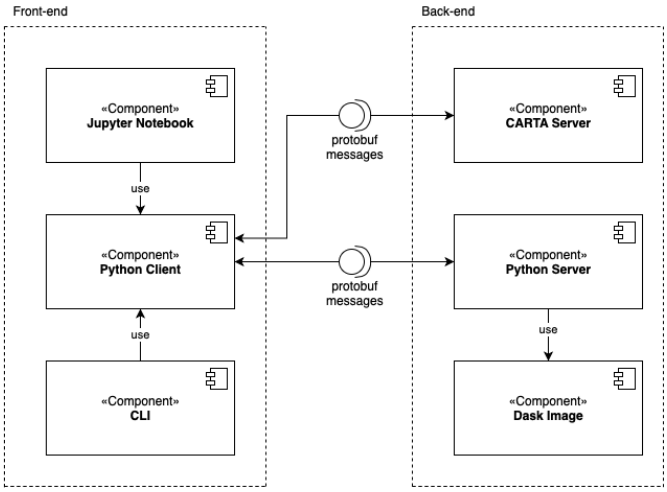

Figure 3: Component diagram showing the modified client-server architecture

represent an eventual result. To retrieve any actual result, we must first call *.compute()* on our Dask object.

Dask provides wrappers for many useful Python functions, including a large subset of the core functionality of NumPy [33] and SciPy [36]. This makes working with Dask very intuitive for a developer that is familiar with Python. We can treat our Dask array like a normal NumPy array and apply NumPy and SciPy functions to it as per usual.

3.2.3 The Dask schedulers. When we call *.compute()* on a Dask object, we are submitting a function application to our scheduler. We have two choices of schedulers, the default scheduler used for parallelism on one machine, or the *dask.distributed* asynchronous scheduler for clusters of one or more machines. The scheduler is responsible for distributing the work amongst the workers and collecting the results.

We performed tests using both the local and the distributed schedulers. We used the Dask *SSHCluster* object to instantiate an un-managed cluster for our distributed scheduler, but it should be noted that Dask provides several options to interface with managed clusters, including MPI [18] and SLURM [37] among others.

3.2.4 The Python server and client. The Python back-end uses WebSockets [4] to communicate with its clients. This is implemented with asyncio [27], a package for high-performance implicit multi-threading using the *async / await* syntax. At a high level, this allows us to define asynchronous event handlers that operate concurrently without having to implement any threading ourselves. This means that our backend can serve traffic from an unbounded number of clients concurrently. The Python client also uses these WebSockets to connect to either our Python server or the CARTA back-end.

These dependencies are summarised by Adjiet [1] in a related work as a package diagram given in Figure 4.

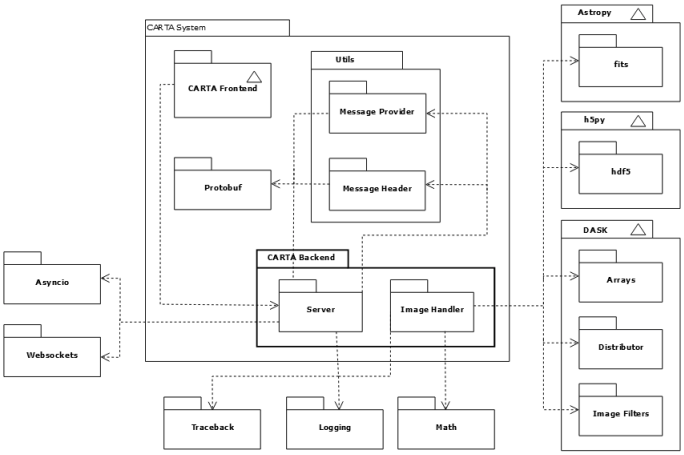

Figure 4: Package diagram for Python back-end

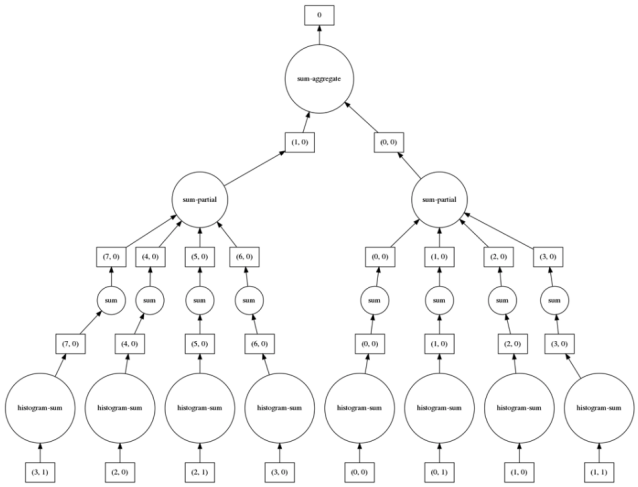

Figure 5: Dask task graph for computing image histogram

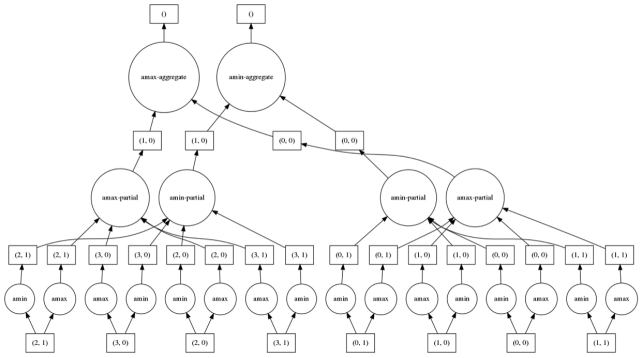

Figure 6: Dask task graph for computing image range

4 TESTING AND EVALUATION

4.1 Testing Environment

Development and testing were performed using Linux Ubuntu (18.04.5 LTS) virtual machines on the Ilifu [20] cloud computing system for data-intensive research. Each virtual machine used by Dask had 4 cores and 32GB of memory. The CARTA server was also run on this same virtual machine for consistency. The testing framework was designed and implemented with the guidance of developers from IDIA.

4.2 Test Functions

Two back-end functions were chosen for testing: computing region histograms, and computing region statistics (comprising the mean, standard deviation, minimum, maximum, and sum). These represent two of the most common use cases of the CARTA system.

In both of these computations, we use the default image channel and region corresponding to the entire image. For the region histogram computation, the range of the histogram was set to be the range of the entire image, and the number of bins set to CARTA's default given by:

$bins = \sqrt{height * width}, bins \geqslant 2.$

We can visualise the computation of these two functions by constructing the task graph that Dask would use to schedule them. The task graph for computing the histogram of an image comprising eight chunks is given in Figure 5. The task graph for computing the range of the same image is given in Figure 6, and it should be noted that the other statistics computed by our implementation have similar task graphs.

4.3 Testing for Efficacy

To prove that our implementation is correct, we compare the output from the Python back-end with that of the CARTA back-end. We construct unit tests for each function that assert the two results are within a 0.1% margin of each other, to account for errors in floating-point arithmetic.

Testing data comprised a set of astronomical images from the public domain, made available through the European Southern Observatory (ESO) Science Archive Facility [24].

All unit tests pass without error.

4.4 Testing for Efficiency

To determine how relatively efficient our implementation is, we measure the time it takes for both our implementation and the CARTA server to perform certain computations. Each test case was executed 10 times and had a mean and standard deviation computed.

Testing data comprised randomly generated Gaussian images. We used a set of 20 of these images ranging in dimensions from 1000 X 1000 pixels to 20000 X 20000 pixels and in file size from 5MB to 1.6GB. These synthetic *fits* images were generated programmatically [1].

Each test case does the following:

[1]The Python script used to generate these test images was provided by IDIA and is available at https://github.com/idia-astro/image-generator

(1) Clear the system cache
(2) Open the file
(3) Get the histogram or region statistics

Sequence diagrams describing these test cases are given in Figure 7 and Figure 8 for region statistics and histogram functions respectively.

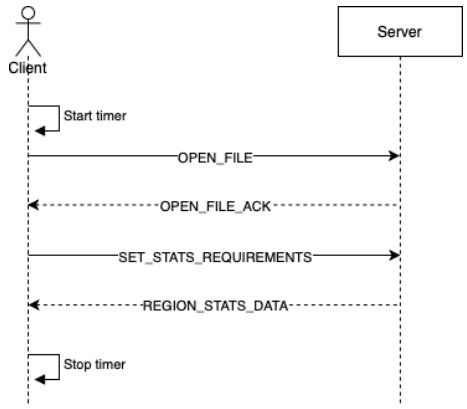

Figure 7: Sequence diagram for region statistics performance test

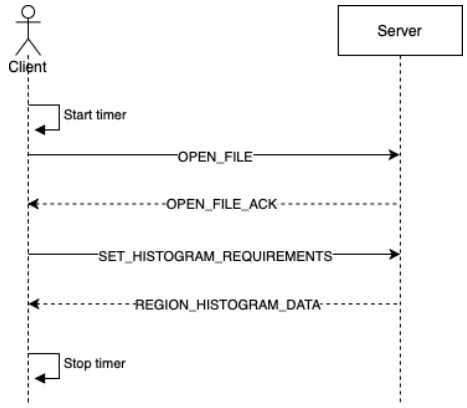

Figure 8: Sequence diagram for region histogram performance test

We benchmarked performance with these test cases against the CARTA back end, our Dask implementation running locally on one machine (with 4 threads and 32GB of memory), as well as Dask using a distributed cluster of three machines (with 12 threads and 96GB of memory, noting that the Dask scheduler shares resources with one of the three worker nodes).

Since the performance of these functions is often IO-bound, we performed each of these tests both with data from disk and data in memory. For the tests with data from disk, the time it takes to read the image into memory and distribute it across our cluster is included in our measurements. For the in-memory tests, it is assumed that the data is already in memory and has already been persisted to the cluster.

5 RESULTS AND DISCUSSION

The mean results from our performance tests are plotted in Figure 9. The raw data collected during testing is given in the appendix. While we include results from both the dask distributed implementation and the dask local implementation, the following discussion will refer to the results from the dask distributed implementation only.

All results were subjected to an unpaired t-test that yields a two-tailed p-value verifying their statistical significance. We use the conventional threshold value $\alpha = 0.05$.

5.1 Region Histogram Computation

For data from disk (Figure 9a), the Dask implementation ($\bar{x} = 3.740$, $\sigma = 2.785$) was significantly faster than CARTA ($\bar{x} = 17.86$, $\sigma = 15.68$), offering a mean speedup of 4.78X, $t(20) = 3.9641$, $p = 0.0003$ ($p < \alpha$). A large proportion of this computation time was due to disk I/O and at the time of writing it is not clear why CARTA performs significantly worse than Dask in this regard.

With data in memory (Figure 9b), the Dask implementation ($\bar{x} = 1.809$, $\sigma = 1.312$) was slightly slower than CARTA ($\bar{x} = 1.267$, $\sigma = 1.084$), $t(20) = 1.4242$, $p = 0.1626$. But since $p > \alpha$, this difference is not statistically significant, even though CARTA was 1.43X faster on average.

5.2 Region Statistics Computation

For data from disk (Figure 9c), the Dask implementation ($\bar{x} = 1.847$, $\sigma = 1.589$) was significantly faster than CARTA ($\bar{x} = 18.57$, $\sigma = 17.63$), corresponding to a mean speedup of 10.05X, $t(20) = 4.2249$, $p = 0.0001$ ($p < \alpha$). This again has to do with disk I/O latency. But with data in memory (Figure 9d), the Dask implementation ($\bar{x} = 0.3226$, $\sigma = 0.1751$) was also significantly faster than CARTA ($\bar{x} = 2.313$, $\sigma = 2.032$), with a mean speedup of 7.17X, $t(20) = 4.3632$, $p = 0.0001$ ($p < \alpha$).

5.3 Analysis of Results

In analysing Dask's performance for histogramming, we must first examine the requirements for computing the histogram. We must know the image range before computing the histogram to arrange our bins, and this is an expensive and non-trivial computation over distributed memory. Computing the range before computing the histogram contributes significantly to our overall computation time. We could avoid this by caching some statistical properties of the image such as minimum and maximum values as a header in the image file, such as is done in a new HDF5 schema designed by Comrie et al. [8] specifically for use with CARTA.

The reason that Dask outperforms CARTA significantly for computing a set of statistical values may be related to how Dask constructs its task graphs. We can combine several distinct functions into one task graph, meaning that computing all five statistical values requires only one traversal of our image data. Furthermore, Dask shares intermediary results between workers, as illustrated previously in Figure 1. For example, there is no need to compute the sum twice when evaluating the mean and sum squared error of the image at the same time. These intermediaries are shared between worker nodes using peer-to-peer data exchange to reduce bottlenecks imposed by the scheduler node's network throughput and possibly reduce the load on file servers.

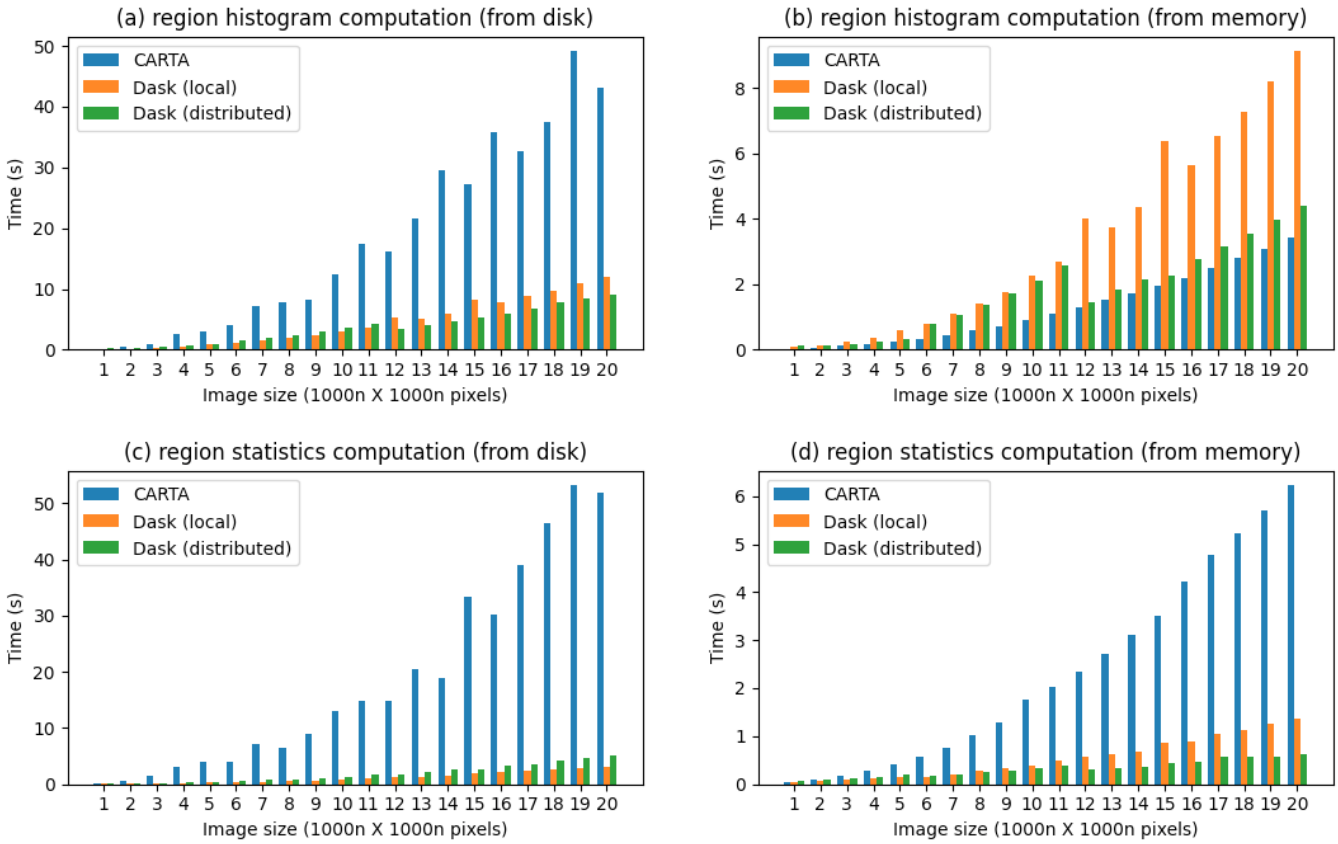

Figure 9: Mean computation times for the histogram and statistics functions, with data from disk and in memory.

We notice that Dask is considerably slower in some cases with smaller images. This is due to Dask's 100MB optimal chunk size: for images smaller than this, the entire image may be in one chunk implying that all computation will happen sequentially. We also notice that in some cases, it is faster to use Dask on a single machine than on a cluster of three machines. This is likely due to the latency encountered in passing data between the scheduler node and the worker nodes over the network. For less expensive computations, the network overhead can be larger than the speed-up achieved on the cluster, so it can be more efficient to perform the computation locally. This network latency could potentially be reduced by experimenting with message-passing protocols that may be more efficient than the standard SSH protocol, with MPI as an example.

6 CONCLUSIONS

We implemented a set of prototype components for a dataflow implementation of the CARTA back-end using Dask in Python. We developed a back-end server that responds to protocol buffer messages in the same manner as the CARTA implementation. We implemented a front-end that can interface with both our back-end and the CARTA back-end. We conducted tests on these components to ensure correctness and measure performance.

Our test results show that the Dask implementation significantly outperforms the CARTA implementation in computing region statistics both with data from disk and data in memory. The Dask implementation outperforms the CARTA implementation in computing region histograms with data from disk, while it performs comparably with the CARTA implementation in performing this computation with data in memory.

Notably, the Dask solution provides a much more scalable approach to performing these computations. The distributed scheduler can aggregate an arbitrary number of worker nodes, and given that the network overhead is not too large, we can continue to decrease our compute times by adding more nodes to our cluster. This is not the case with CARTA and the local Dask implementation, where we are strictly bounded by how many cores our one processor has. Furthermore, we can introduce heterogeneity into our Dask solution, for example by having some nodes with CPU compute resources and other nodes with GPU compute resources in our cluster.

Thus, we conclude that it is indeed possible to adapt existing dataflow tools to handle the high-throughput interactive visualisation and analysis workloads of the CARTA system in a performant and highly scalable manner.

As a result of this, we find it would be feasible to move forward with a dataflow implementation of the CARTA back-end. Specifically, we find that a hybrid approach with Dask performing smaller computations locally and larger computations across a cluster may offer the best performance in these use cases.

The prototype software developed in this project would form the basis around which a new production back-end could be built. Future work would involve extending the Python server to respond to the additional set of protocol buffer messages for which event handlers have not yet been implemented. The Python front-end would be replaced with the existing CARTA front-end (perhaps maintaining the existing performance and correctness testing framework). Additionally, ongoing work is required to optimise the performance of the Dask cluster, as such is the nature of distributed computation.

The software developed in this work is all open source and available online [2] under the GNU general public license (GPLv3).

ACKNOWLEDGMENTS

This work is based on the research supported wholly or in part by the National Research Foundation of South Africa (grant number MND190716456261).

The author acknowledges the use of the Ilifu cloud computing facility, 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 and the University of the Western Cape), the Computational Biology Division at UCT and the Data-Intensive Research Initiative of South Africa (DIRISA).

The author would also like to acknowledge and thank our supervisor Professor Robert Simmonds for the academic guidance and support, as well as the developers from IDIA, Ms Adrianna Pińska and Dr. Angus Comrie, for their technical advice and suggestions.

REFERENCES

[1] Zainab Adjiet. 2020. Exploring a Dataflow Design of the CARTA System.
[2] Tilak Agerwala et al. 1982. Data Flow Systems: Guest Editors' Introduction. *Computer* 15, 2 (1982), 10–13.
[3] A Worldwide Collaboration ALMA. 2020. Atacama Large Millimeter/submillimeter Array. https://www.almaobservatory.org/en/home/
[4] Aymeric Augustin. 2019. Getting started. https://websockets.readthedocs.io/en/stable/intro.html
[5] British Standards Institute. 2003. *The C++ Standard: incorporating Technical Corrigendum 1: BS ISO* (second ed.). Wiley, New York, NY, USA. xxxiv + 782 pages.
[6] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. 2009. Toward exascale resilience. *The International Journal of High Performance Computing Applications* 23, 4 (2009), 374–388.
[7] CARTAvis. 2020. CARTA Protobuf Messages. https://github.com/CARTAvis/carta-protobuf
[8] A Comrie, A Pińska, R Simmonds, and AR Taylor. 2020. Development and application of an HDF5 schema for SKA-scale image cube visualization. *Astronomy and Computing* (2020), 100389.
[9] Angus Comrie and Rob Simmonds. 2020. CARTA Interface Control Document. https://carta-protobuf.readthedocs.io/en/latest/index.html
[10] Angus Comrie, Kuo-Song Wang, Pamela Ford, Anthony Moraghan, Shou-Chieh Hsu, Adrianna Pińska, Cheng-Chin Chiang, Hengtai Jan, Rob Simmonds, Tien-Hao Chang, and Ming-Yi Lin. 2020. *CARTA: The Cube Analysis and Rendering Tool for Astronomy.* https://doi.org/10.5281/zenodo.3746095
[11] Andrew Collette & contributors. 2020. *HDF5 for Python.* https://www.h5py.org
[12] James Crist. 2016. Dask & Numba: Simple libraries for optimizing scientific python code. In *2016 IEEE International Conference on Big Data (Big Data).* IEEE, 2342–2343.
[13] David E Culler. 1986. Dataflow architectures. *Annual review of computer science* 1, 1 (1986), 225–253.
[14] David Davidson. 2012. MeerKAT and SKA phase 1. *2012 10th International Symposium on Antennas, Propagation and EM Theory, ISAPE 2012*, 1279–1282. https://doi.org/10.1109/ISAPE.2012.6409014
[15] Alan L Davis and Robert M Keller. 1982. Data flow program graphs. (1982).
[16] Jack B Dennis. 1974. First version of a data flow procedure language. In *Programming Symposium.* Springer, 362–376.
[17] Inter-University Institute for Data Intensive Astronomy. 2020. *Inter-University Institute for Data Intensive Astronomy.* https://www.idia.ac.za
[18] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. USA.
[19] Google. 2020. Dataflow. https://cloud.google.com/dataflow
[20] ilifu. 2020. Cloud computing for data-intensive research. http://www.ilifu.ac.za/
[21] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows.. In *ELPUB.* 87–90.
[22] Ben Lee and Ali R Hurson. 1994. Dataflow architectures and multithreading. *Computer* 27, 8 (1994), 27–39.
[23] Yishu Mao, Yongxin Zhu, Tian Huang, Han Song, and Qixuan Xue. 2016. DAG Constrained Scheduling Prototype for an Astronomy Exa-Scale HPC Application. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS).* IEEE, 631–638.
[24] European Southern Observatory. 2020. The ESO Science Archive Facility. https://archive.eso.org/cms.html
[25] National Radio Astronomy Observatory. 2020. *National Radio Astronomy Observatory.* https://public.nrao.edu/
[26] A. M. Price-Whelan, B. M. Sipőcz, H. M. Günther, P. L. Lim, S. M. Crawford, S. Conseil, D. L. Shupe, M. W. Craig, N. Dencheva, A. Ginsburg, J. T. VanderPlas, L. D. Bradley, D. Pérez-Suárez, M. de Val-Borro, (Primary Paper Contributors, T. L. Aldcroft, K. L. Cruz, T. P. Robitaille, E. J. Tollerud, (Astropy Coordination Committee, C. Ardelean, T. Babej, Y. P. Bach, M. Bachetti, A. V. Bakanov, S. P. Bamford, G. Barentsen, P. Barmby, A. Baumbach, K. L. Berry, F. Biscani, M. Boquien, K. A. Bostroem, L. G. Bouma, G. B. Brammer, E. M. Bray, H. Breytenbach, H. Buddelmeijer, D. J. Burke, G. Calderone, J. L. Cano Rodríguez, M. Cara, J. V. M. Cardoso, S. Cheedella, Y. Copin, L. Corrales, D. Crichton, D. D'Avella, C. Deil, É. Depagne, J. P. Dietrich, A. Donath, M. Droettboom, N. Earl, T. Erben, S. Fabbro, L. A. Ferreira, T. Finethy, R. T. Fox, L. H. Garrison, S. L. J. Gibbons, D. A. Goldstein, R. Gommers, J. P. Greco, P. Greenfield, A. M. Groener, F. Grollier, A. Hagen, P. Hirst, D. Homeier, A. J. Horton, G. Hosseinzadeh, L. Hu, J. S. Hunkeler, Ž. Ivezić, A. Jain, T. Jenness, G. Kanarek, S. Kendrew, N. S. Kern, W. E. Kerzendorf, A. Khvalko, J. King, D. Kirkby, A. M. Kulkarni, Astropy Contributors, et al. 2018. The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package. *Astronomical Journal* 156, Article 123 (Sept. 2018), 123 pages. https://doi.org/10.3847/1538-3881/aabc4f
[27] Python. 2020. asyncio - Asynchronous I/O. https://docs.python.org/3/library/asyncio.html
[28] Python. 2020. Futures. https://docs.python.org/3/library/asyncio-future.html
[29] John Shalf, Sudip Dosanjh, and John Morrison. 2011. Exascale Computing Technology Challenges. In *High Performance Computing for Computational Science – VECPAR 2010,* José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25.
[30] Jurij Silc, Borut Robic, and Theo Ungerer. 1998. Asynchrony in parallel computing: From dataflow to multithreading. *Parallel and Distributed Computing Practices* 1, 1 (1998), 3–30.
[31] Vitor Silva, Daniel de Oliveira, and Marta Mattoso. 2014. Exploratory analysis of raw data files through dataflows. In *2014 International Symposium on Computer Architecture and High Performance Computing Workshop.* IEEE, 114–119.
[32] Ask Solem et al. 2013. Celery: Distributed Task Queue. *URL http://docs. celeryproject. org/en/latest/index. html* (2013).
[33] 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.
[34] Brad Vander Zanden. 1996. An Incremental Algorithm for Satisfying Hierarchies of Multiway Dataflow Constraints. *ACM Trans. Program. Lang. Syst.* 18, 1 (Jan. 1996), 30–72. https://doi.org/10.1145/225540.225543

[2] The prototype software, documentation, and Python scripts used for data analysis and visualisation are available at https://github.com/DylanFouche/CADaFloP

[35] Lorenzo Verdoscia and Roberto Vaccaro. 2013. Position paper: Validity of the static dataflow approach for exascale computing challenges. In *2013 Data-Flow Execution Models for Extreme Scale Computing*. IEEE, 38–41.

[36] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[37] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60.

[38] Zhao Zhang, Kyle Barbary, Frank Austin Nothaft, Evan Sparks, Oliver Zahn, Michael J Franklin, David A Patterson, and Saul Perlmutter. 2015. Scientific computing meets big data technology: An astronomy use case. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 918–927.

Table 1: Performance test results for region histogram computation with data on disk

Image dimensions	Runtime (seconds)			
		CARTA	Dask (local)	Dask (distributed)
1000X1000	mean	0.083973482	0.147940915	0.259091207
	std dev	0.028463537	0.00329015	0.010272032
2000X2000	mean	0.607651015	0.175108006	0.357176841
	std dev	0.110343836	0.02147193	0.02016716
3000X3000	mean	0.86734242	0.376615192	0.481420768
	std dev	0.20533837	0.004927366	0.027400818
4000X4000	mean	2.570225082	0.535928135	0.665745026
	std dev	0.460740268	0.018099902	0.037117642
5000X5000	mean	2.957107015	0.832367673	0.930730314
	std dev	0.397043372	0.019392305	0.02520691
6000X6000	mean	4.010597545	1.130542752	1.482484095
	std dev	0.490310095	0.031909439	0.055183845
7000X7000	mean	7.095431598	1.51607524	1.902063001
	std dev	1.240013319	0.04187532	0.085091269
8000X8000	mean	7.887402703	1.878722592	2.377365647
	std dev	0.752159264	0.043613406	0.080923525
9000X9000	mean	8.270056809	2.459515017	2.975669905
	std dev	0.812064063	0.055989485	0.116232823
10000X10000	mean	12.42829339	2.94449417	3.563897172
	std dev	0.979725634	0.062832722	0.164335016
11000X11000	mean	17.46711974	3.596476388	4.292373152
	std dev	1.538639366	0.06717249	0.196846427
12000X12000	mean	16.17364813	5.218936241	3.366203461
	std dev	1.167979582	0.027736309	0.14202732
13000X13000	mean	21.5736837	5.107077898	4.126811227
	std dev	2.268034955	0.087151372	0.148521066
14000X14000	mean	29.59556534	5.935723358	4.737114021
	std dev	9.849811971	0.083223989	0.267186259
15000X15000	mean	27.31874898	8.207481606	5.242815395
	std dev	2.587735857	0.045072005	0.267549734
16000X16000	mean	35.90970596	7.780746745	5.985637632
	std dev	3.239538383	0.133728753	0.211510508
17000X17000	mean	32.70255804	8.8383282	6.771232129
	std dev	1.649885592	0.1706576	0.302698392
18000X18000	mean	37.53867875	9.777181336	7.767570164
	std dev	2.577100447	0.141552742	0.383065295
19000X19000	mean	49.09826756	11.05436623	8.453593761
	std dev	5.399490261	0.102890502	0.50356986
20000X20000	mean	43.01979401	12.04781072	9.064930938
	std dev	1.546950995	0.173946126	0.628402189

Table 2: Performance test results for region histogram computation with data in memory

Image dimensions	Runtime (seconds)			
		CARTA	Dask (local)	Dask (distributed)
1000X1000	mean	0.032419875	0.104704935	0.127036522
	std dev	0.001904326	0.002369486	0.001722356
2000X2000	mean	0.073248547	0.123145655	0.15168232
	std dev	0.006864479	0.013052706	0.007382914
3000X3000	mean	0.133431528	0.26540361	0.168605557
	std dev	0.012737338	0.003703564	0.010304487
4000X4000	mean	0.182269339	0.371761606	0.25558966
	std dev	0.010264456	0.009769408	0.008042513
5000X5000	mean	0.25203988	0.595180977	0.338630372
	std dev	0.005627782	0.005742249	0.011531267
6000X6000	mean	0.342695104	0.809570203	0.79696593
	std dev	0.006314159	0.030759888	0.020811725
7000X7000	mean	0.454550814	1.091878952	1.061936385
	std dev	0.003889051	0.026310852	0.025942936
8000X8000	mean	0.609884823	1.421095017	1.365454931
	std dev	0.081340316	0.035455509	0.025074087
9000X9000	mean	0.723701357	1.756271151	1.724159806
	std dev	0.007111379	0.033428578	0.04891792
10000X10000	mean	0.909897896	2.257816234	2.105707183
	std dev	0.010131014	0.026651872	0.021638323
11000X11000	mean	1.092638929	2.696698318	2.557195692
	std dev	0.031593127	0.057053812	0.049233472
12000X12000	mean	1.291618937	4.012829536	1.459340244
	std dev	0.049993635	0.012858078	0.025358946
13000X13000	mean	1.535468281	3.728905378	1.851271295
	std dev	0.066075737	0.072568225	0.007985509
14000X14000	mean	1.735440321	4.372448187	2.133318104
	std dev	0.067607721	0.062938563	0.017268317
15000X15000	mean	1.955368664	6.369780679	2.256568447
	std dev	0.019847627	0.055126368	0.023807977
16000X16000	mean	2.207667474	5.644412089	2.774302989
	std dev	0.01433429	0.082365295	0.021035003
17000X17000	mean	2.482743686	6.521540911	3.162728916
	std dev	0.0184552	0.120978351	0.030532345
18000X18000	mean	2.799750769	7.271051937	3.538650655
	std dev	0.064698161	0.085549201	0.017309601
19000X19000	mean	3.088372432	8.196474144	3.959471981
	std dev	0.048841464	0.115216887	0.029045585
20000X20000	mean	3.43028668	9.115087942	4.384508413
	std dev	0.044723569	0.129096793	0.023662407

Table 3: Performance test results for region statistics computation with data on disk

Image dimensions	Runtime (seconds)			
		CARTA	Dask (local)	Dask (distributed)
1000X1000	mean	0.16134228	0.045422638	0.077280223
	std dev	0.053836874	0.006742844	0.004058448
2000X2000	mean	0.522890772	0.07508293	0.129107816
	std dev	0.130160572	0.004320071	0.009515398
3000X3000	mean	1.36874004	0.122545418	0.205302701
	std dev	0.338995673	0.00580008	0.010094922
4000X4000	mean	3.042096641	0.171456093	0.309386703
	std dev	0.499744551	0.007473159	0.047115707
5000X5000	mean	3.872838514	0.260820402	0.441022826
	std dev	0.591391459	0.010865779	0.028997669
6000X6000	mean	4.034074716	0.308464873	0.553232597
	std dev	0.613456093	0.007858133	0.050323597
7000X7000	mean	7.226235466	0.443988365	0.700420164
	std dev	0.615678404	0.027526743	0.054970557
8000X8000	mean	6.442994514	0.560725101	0.835247515
	std dev	0.646599168	0.034143028	0.045428861
9000X9000	mean	8.950019369	0.663626459	1.055898435
	std dev	0.669570535	0.034997067	0.07968626
10000X10000	mean	12.90927582	0.808643103	1.244698408
	std dev	0.731311911	0.053703041	0.029541021
11000X11000	mean	14.82100849	0.990791499	1.631916062
	std dev	0.648569976	0.061534084	0.153189387
12000X12000	mean	14.88973671	1.186967599	1.697564436
	std dev	0.620804937	0.025442022	0.153722487
13000X13000	mean	20.46376944	1.351859	2.190974714
	std dev	0.852097928	0.04948981	0.205142623
14000X14000	mean	18.88624544	1.519707447	2.489731997
	std dev	1.048797602	0.053116254	0.322029462
15000X15000	mean	33.25728028	1.824996882	2.653845446
	std dev	2.005775948	0.042188788	0.411781344
16000X16000	mean	30.14348859	2.089427313	3.261382149
	std dev	1.940487803	0.100482101	0.347641872
17000X17000	mean	38.92950441	2.264839355	3.611563147
	std dev	2.960348566	0.057861187	0.383452969
18000X18000	mean	46.43081268	2.525616601	4.192326819
	std dev	7.411646947	0.080536692	0.620512141
19000X19000	mean	53.16148639	2.750385896	4.541751385
	std dev	9.156237687	0.094577817	0.420168994
20000X20000	mean	51.82220803	3.108858252	5.122935691
	std dev	4.542524637	0.125182723	0.719616838

Table 4: Performance test results for region statistics computation with data in memory

Image dimensions	Runtime (seconds)			
		CARTA	Dask (local)	Dask (distributed)
1000X1000	mean	0.043788905	0.040383491	0.069368549
	std dev	0.003319475	0.003274622	0.00386439
2000X2000	mean	0.103659071	0.06002607	0.082282744
	std dev	0.008089353	0.00381433	0.004792538
3000X3000	mean	0.172277242	0.085049056	0.107407824
	std dev	0.006895824	0.005747465	0.006383513
4000X4000	mean	0.266348011	0.109607858	0.133318396
	std dev	0.004656603	0.008807267	0.007114832
5000X5000	mean	0.406708261	0.153304119	0.196463476
	std dev	0.009293859	0.009582305	0.022353219
6000X6000	mean	0.562052773	0.15223546	0.167803796
	std dev	0.019508248	0.005316572	0.01047157
7000X7000	mean	0.756451507	0.191777584	0.202121843
	std dev	0.006086149	0.002333368	0.021059842
8000X8000	mean	1.012833191	0.277093173	0.250448937
	std dev	0.055870166	0.025336	0.031035124
9000X9000	mean	1.283358262	0.328220077	0.288305248
	std dev	0.076588205	0.039344915	0.033346087
10000X10000	mean	1.752214023	0.380241508	0.330313641
	std dev	0.113030933	0.043223674	0.037800247
11000X11000	mean	2.039371797	0.491069002	0.378730664
	std dev	0.074384341	0.050459914	0.029234504
12000X12000	mean	2.342358662	0.577141999	0.314808366
	std dev	0.120243491	0.011546824	0.016341557
13000X13000	mean	2.711349631	0.623599295	0.326472967
	std dev	0.106822752	0.04368001	0.007794566
14000X14000	mean	3.100263616	0.681589623	0.351518111
	std dev	0.053334206	0.046408886	0.014382425
15000X15000	mean	3.520058804	0.850479839	0.448140785
	std dev	0.150215799	0.025298952	0.046321125
16000X16000	mean	4.2285058	0.896898769	0.463455375
	std dev	0.356011701	0.076559054	0.059671443
17000X17000	mean	4.776374464	1.036016547	0.571273714
	std dev	0.074351944	0.05932181	0.062965594
18000X18000	mean	5.228070524	1.122260022	0.566539339
	std dev	0.097616965	0.072122125	0.083764575
19000X19000	mean	5.721162193	1.258089667	0.582226971
	std dev	0.249753599	0.067166665	0.065375858
20000X20000	mean	6.226993544	1.354669118	0.621698569
	std dev	0.187102944	0.068224591	0.017389287