SciSpark: Applying In-memory Distributed Computing to Weather Event Detection and Tracking

Rahul Palamuttam[1,3], Renato Marroquín Mogrovejo[1,4], Chris Mattmann[1,2], Brian Wilson[1], Kim Whitehall[1], Rishi Verma[1], Lewis McGibbney[1], Paul Ramirez[1]
[1]NASA Jet Propulsion Laboratory California Institute of Technology
Pasadena, CA, USA

[2]Computer Science Department, University of Southern California
Los Angeles, CA 90089 USA

[3]University of California San Diego
La Jolla, CA, USA

[4]ETH University
Zürich, Switzerland
rahulpalamut@gmail.com

***Abstract*— In this paper we present SciSpark, a Big Data framework that extends Apache™ Spark for scaling scientific computations. The paper details the initial architecture and design of SciSpark. We demonstrate how SciSpark achieves parallel ingesting and partitioning of earth science satellite and model datasets. We also illustrate the usability and extensibility of SciSpark by implementing aspects of the *Grab 'em Tag 'em Graph 'em (GTG)* algorithm using SciSpark and its Map Reduce capabilities. GTG is a topical automated method for identifying and tracking Mesoscale Convective Complexes in satellite infrared datasets.**

***Index Terms*— Apache Spark, in-memory distributed computing, large scientific datasets, mesoscale convective complexes**

I. Introduction

Google's MapReduce [1] is a widely used framework for solving large computational problems in parallel. This model adopts the 'map' and 'reduce' semantics from functional programming to express data and task decomposition for parallel computing. Apache™ Hadoop later introduced the Hadoop Distributed File System (HDFS) that is a highly fault tolerant distributed file system inspired by the Google File System (GFS) that accompanied the Google MapReduce framework. Coupling HDFS and MapReduce, makes Hadoop a general purpose system for use cases outside of its original inspiration – search – and in turn this combination has been applied to an increasing number of scientific domains such as Earth science, biomedicine, national security, etc. MapReduce jobs in Hadoop are launched as a series of map tasks followed by either additional map tasks, or by a single reduce task that combines the results. Each map task involves I/O processes to disk thereby introducing latency.

Apache Spark [2] mitigates writing to disk by keeping results in memory until data needs to be spilled to disk. Spark uses the Resilient Distributed Dataset (RDD) [3] that is an in-memory distributed data structure. Furthermore, Spark utilizes the functional programming paradigm to extend lambda expressions thus enabling users to express map and reduce tasks seamlessly. Spark achieves fault-tolerance through relying on a Directed Acyclic Graph (DAG) execution engine for its computation workflow, thus making it more efficient than the Hadoop work engine.

Spark's generic RDDs are ideal for tabular or unstructured data. Science datasets are highly structured. Reading hierarchical files from HDFS is a known challenge for Big Data applications due to how files are physically stored on distributed file systems. The dissonance between logical and physical data representation has been noted by researchers e.g. [4]. Research endeavors such as SciHadoop [5] attempt to solve the issue, however SciHadoop is built for Hadoop 0.20.2. A variety of MapReduce specific interfaces are deprecated in the later Hadoop versions. Furthermore, with the increasingly growing community around Spark and its potential to offer speed-up and advancements of nearly 1000x in-memory, our team at NASA's Jet Propulsion Laboratory - California Institute of Technology (JPL) began an exploratory effort called "SciSpark" to augment Spark with the ability to load, process, and deliver scientific data and results. SciSpark is funded by NASA's Advanced Information Systems Technology (AIST) program [6],[7]. SciSpark is grounded in two novel scientific use cases involving data reuse algorithms. The first, is a k-means clustering algorithm to compute climate extremes over decades of climate model data [8], but is outside the scope of this paper. We will instead focus on the second algorithm – a graph-based automated method for

identifying and tracking Mesoscale Convective Complexes (MCCs) [9] that can be categorized as a severe weather event.

This paper describes the initial architecture and design of SciSpark. SciSpark defines the Scientific Resilient Distributed Dataset (sRDD), a distributed-computing array structure that supports multidimensional data and processing of scientific algorithms in the MapReduce paradigm. SciSpark will produce methods to create sRDDs that preserve the logical representation of structured and dimensional data. Our early focus involves ingestion of network Common Data Form (netCDF) [10] and Hierarchical Data Format (HDF) files [11]. We follow sRDD creation by discussing a new approach to pair sequential data from separate arrays in order to implement a SciSpark version for the initial stages of the GTG algorithm (our further work will implement the full algorithm). We also illustrate the shift from the sequential programming paradigm to the MapReduce paradigm. Metrics with respect to graph development and pitfalls encountered are discussed. Our early work can be used as both a practical experience, and a research roadmap for work in Spark using structured, multi-dimensional scientific datasets going forward.

II. SciSpark

A. The SciSpark Architecture

The SciSpark architecture presented in Figure 1 consists of two components: the backend core and the front-end visualization. Within the backend are three layers that interact to complete a user-defined computation pipeline by leveraging sciTensor objects within the Scientific Resilient Distributed Dataset (sRDD) - the distributed-computing data structure. The sciTensor is a self-document array-collection developed for sRDD transformations. To interact with SciSpark's backend, an expert user utilizes the SciSpark API. A novice user will interact with SciSpark via the frontend that will leverage a RESTful web API. The functionality of the layers in the SciSpark backend will be briefly presented.

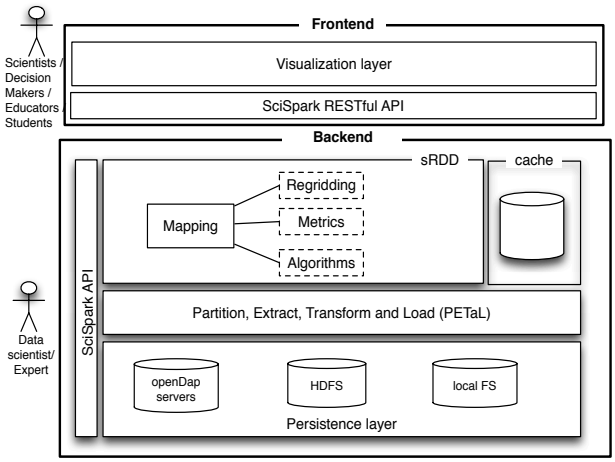

Figure 1: The SciSpark architecture

1) Persistence layer

SciSpark ingests scientific data formats, for example netCDF and HDF source files, from various local and remote data sources in a non-sequential manner. SciSpark reads these sources in the persistence layer using their uniform resource identifiers (URIs). SciSpark also uses the persistence layer to stage intermediate or output final results. The SciSpark API provides methods to access these data sources, and is extendable to other sources.

2) Partition, Extract, Transform and Load (PETaL) layer

The PETaL layer first partitions and distributes the URIs across the compute nodes. It then extracts the data and transforms it into a data type usable in SciSpark that is then loaded into the processing layer. The SciSparkContext provides the API for the PETaL layer.

3) Processing layer

In this layer of SciSpark, the user executes their computation pipeline. The sRDD methods in the API govern these computation tasks.

The aspects of the architecture that have been implemented are now presented.

B. SciSpark Implementation

SciSpark is implemented in a Java and Scala environment. The environment was chosen to avoid the known latency issues related to the communication overhead involved with copying data from the worker JVMs to Python daemon processes in the PySpark environment [12]. Furthermore after a collect call is made in PySpark, the driver JVM will write results to local disk and that is then read by the Python process.

The concept of a scientific Resilient Distributed Dataset (sRDD) couples operations on multi-dimensional arrays and distributed in-memory processing. In order to achieve this, the idea of self-documentation in hierarchical file formats was utilized to construct a self-documented array class called sciTensor as illustrated in Figure 2. The goal of the sciTensor is to provide logic that defines the data in the multi-dimensional format that scientists are accustomed to, while achieving high-performance matrix operations through the ND4J [13] and Breeze [14] linear algebra libraries. These libraries take advantage of cache locality during element-wise operations, by allocating dimensional arrays as physical linear arrays, similar to C and FORTRAN.

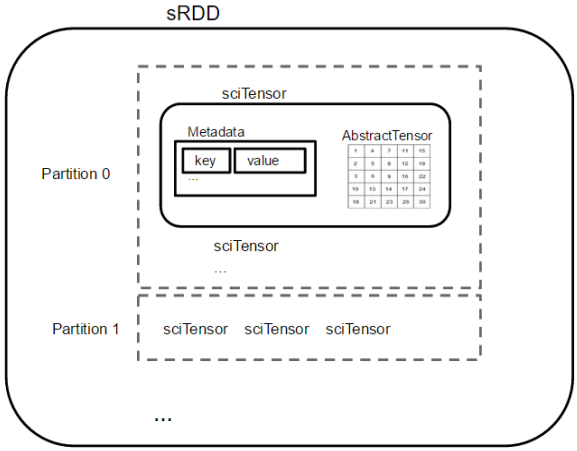

Figure 2: Illustration of the sRDD and sciTensor architectures

C. sRDD and sciTensor Architectures

The sciTensor class consists of two hash tables. The metadata hash table is used to record information about the dataset as key-value pairs. Users can query the table within an sRDD transformation function. The table can also be used to enter information such as date, area, or minimum and maximum values. Each sciTensor is constructed from arrays loaded from a unique path, irrespective of its data source from the persistent layer i.e. whether it is from OpenDAP, HDFS, or the local filesystem. Since hierarchical files can have multiple variable arrays, sciTensor also keeps a hash table of these arrays where the keys are the variable names and the values are the arrays. Users can specify which variables they want to load from the dataset and can specify which variable they want to use for computation during run-time.

D. How PETaL Materializes sRDDs

To extend RDDs to the scientific community, sRDD requires three ingredients as illustrated in Figure 3.

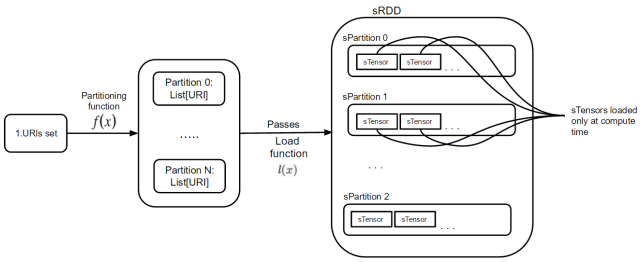

Figure 3: Illustration of the sRDD Materialization Process

a. The URI Set: The user provides a set of line-separated URI's from OpenDAP URLs, HDFS path names, or path names on the local filesystem in a file.
b. The partition function is called when computing the partition sets of sRDD. It groups the URI set after, which SciSpark distributes the subsets of URI's to different compute nodes in the cluster.
c. The user must also provide a loader function that corresponds to the specific source the URI is pointing to. For example, if given a path to a netCDF file the loader function must leverage the netCDF Java API to extract the necessary information.

The SciSparkContext of the SciSpark API provides methods to materialize sRDDs such as *NetCDFFile* and *MergFile*. These methods follow the above three-ingredient pattern.

D. Description of Tensor Libraries

Between the persistence layer and the PETaL layer, is the process of extracting the data from the native format into the sciTensors. This component of the sciTensor is the AbstractTensor whose role is to provide a common interface to different linear algebra libraries. This need arose due to ND4J's support for n-dimensional arrays but lack of maturity, whereas Breeze is a seasoned project but only supports 2-dimensional arrays.

Breeze takes advantage of Scala's operator overloading feature and is backed by Netlib-java - a wrapper library around existing BLAS [15] libraries. Instead of calling verbose function names for different levels of BLAS operations, Breeze enables code to be succinct by utilizing operator overloading. It is limited to matrix operations, and does not support beyond 2-dimensional arrays.

ND4J claims to provide Python NumPy-like syntax and performance. The library gives the option to choose from a variety of backend BLAS libraries to use. ND4J, like Breeze, supports operations on complex numbers. ND4J also provides a backend called Nd4j-x86 that uses BLAS for linear algebra operations and uses C-level for-loops for element-wise operations. ND4J is a young project and is not as mature as Breeze.

We performed evaluations of the Breeze and ND4J libraries compared to NumPy for two time periods, weeks apart. It was found that the ND4J metrics changed drastically between these periods - further indicating the immaturity of this project.

III. A SciSpark use case

The code for our experiments can be found at the GitHub repo https://github.com/SciSpark/SciSpark. The data used in this study are the National Centers for Environmental Prediction / Climate Prediction Center (NCEP/CPC) 4 km Global (60N – 60S) Infrared Dataset (also known as MERG – merged dataset) [16].

A. The Grab 'em, Tag 'em, Graph 'em (GTG) method

The Grab 'em, Tag 'em, Graph 'em (GTG) method illustrated in Figure 4, automates identification of a particular weather phenomenon by searching for cloud elements in consecutive time data frames (acquired from files) of brightness temperature, correlating them in a graph, and analyzing the graph [15]. The cloud elements are identified via a criteria based on shape, size and absolute values of brightness temperatures of contiguous points within a data frame, and are denoted as the vertices in the graph. Consecutive frames are then checked for overlapping cloud elements. Two cloud elements that overlap are represented as an edge in the graph. The graphs are then analyzed via graph methods to determine the type of weather phenomenon. The GTG method presents an opportunity to explore parallelizing the sequential approach of constructing the final graph with inherent chronology from multi-dimensional arrays within the SciSpark framework. We present this as the Distributed GTG.

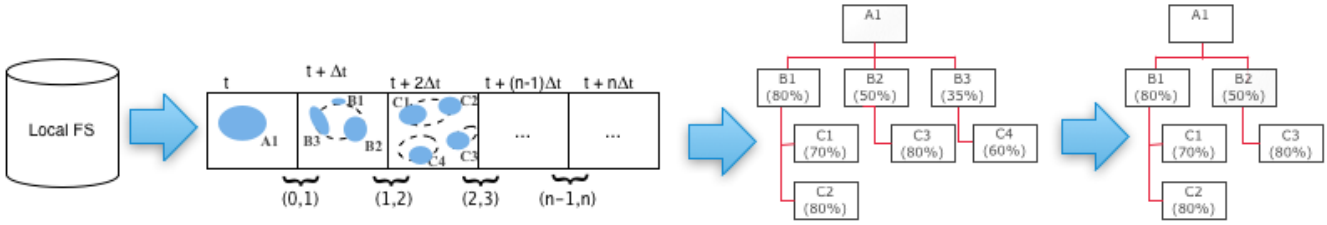

Figure 4: The (sequential) GTG workflow. The GTG reads files sequentially from a local filesystem and places the data into an array. Cloud elements are identified in each time frame - these are the graph nodes. Spatial overlapping between sequential time frames determines the graph edges. The graph is then analyzed to determine the type of weather feature.

B. The Distributed GTG

The initial steps of sequential GTG are to load the files sequentially, then find cloud elements and overlapping cloud elements across consecutive frames. Once cloud elements are identified, the sequential implementation relies on a "for loop" to iterate through each data frame to determine graph edges. Since the array is shared on each iteration, array elements can be referenced by index. The sequential GTG will operate in O(*n*) time as it iterates through all elements in a single thread (as illustrated in Figure 4).

sRDDs on the other hand are computed by executing transformation calls on all partitions simultaneously which makes directly porting the sequential version of GTG to a distributed implementation infeasible. The primary limitation is that map tasks cannot access sciTensors in other sRDD partitions. In order to overcome this lack of shared-memory in distributed system tasks, the approach of the sequential GTG required redesigning. The Distributed GTG is presented in Figure 5.

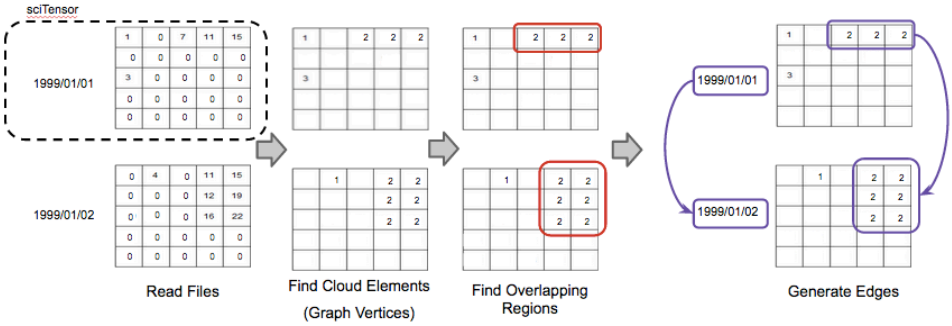

Figure 5: The Distributed GTG workflow. The original data is read into sciTensors into a sRDD. The cloud elements and overlapping regions are determined from each sciTensor pair.

Within the Distributed GTG, the workflow is:

a. Identify the files required for an experiment on a given file system. A path to the HDFS was used. A hash table mapping of chronological dates to indices of the file list found at the path was generated.
b. Partition the file list, extract, transform and load the data frame into SciSpark's sciTensors on a sRDD. SciSparkContext leverages SparkContext's binaryFiles function to read binary data from HDFS into sciTensors.
c. Build the graph in the processing layer utilizing sRDD mapping capabilities. By finding ways to pair sciTensors in the sRDD, we can compute the initial stages of the GTG in parallel.

C. Structuring input sRDD for Distributed GTG graph construction

We wish to achieve a chronological frame pairing in order to replicate the original graph construction in the GTG. This section provides the algorithmic assessment of the approaches explored. We do not evaluate run-time since that is specific to how the task function is implemented.

1) Naïve port of GTG

We attempted to port the sequential GTG by utilizing the sRDD filter operation to reference specific frames. However this exploited little parallelism, as the filtered sRDDs represented one frame each and it was required to execute the collect operation at the end of each iteration (See Figure 6).

```
for(i = 0; i < n - 1; i++){
   FrameOneRDD = sRDD.filter(p => p_f == i)
   FrameTwoRDD = sRDD.filter(p => p_f == i + 1)
         …
}
```

Figure 6: Implementation of the naive approach to represent the initial stages of the sequential GTG in SciSpark

Our naïve approach suffered from network latency in SciSpark resulting from overhead communications with all the partitions.

2) Cartesian product approach

SciSpark extends the ability to create Cartesian products between two sRDDs. Naturally a Cartesian product of a sRDD with itself would compute all pairs of frames. The frame-pair set is then filtered for frame pairs that are consecutive as illustrated in Figure 7. This is the simplest solution to achieve parallelism, but has the highest performance cost. When SciSpark computes the Cartesian product, it scales the partition space, and consequently the number of tasks and memory requirements, by N2. This means that for each frame there are N - 1 copies which are never used. Furthermore, if an entire sRDD partition is empty after the filter call, it is not eliminated from the SciSpark DAG execution pipeline. Thus unnecessary network latency is incurred when tasks communicate back to the master node.

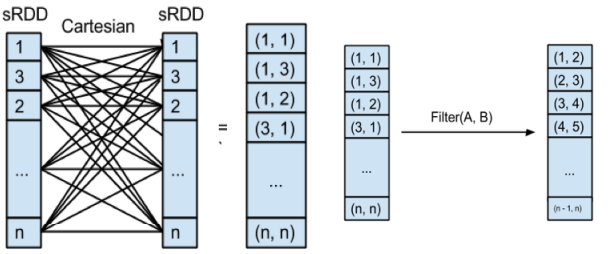

Figure 7: Illustration of the Cartesian product between two sRDDs. Note the elements are not ordered. Filtering on the frame-pair set obtains the chronological frame pairs necessary for edge mining. The implementation is shown at the bottom.

We conclude that while the Cartesian product combined with the filter is powerful and easy to use, it does not scale for larger problems as it creates wasteful data and unnecessary tasks.

3) The group-by approach

Another solution involved making a copy of each frame to map to the next frame thus creating key-value pairs between original frame and the next frame. The key-value pairs were then grouped by the key (i.e. the original frame), thus generating consecutive frame pairs as illustrated in Figure 8. This implementation did not change the partition space and consequently the number of tasks. At this point we treat each pair as input to a single sequentially implemented function.

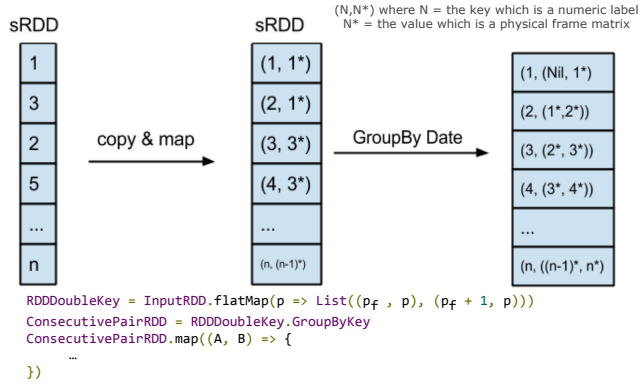

Figure 8: Illustration of the copy and mapping of the two sRDDs and the groupBy operation leading to the chronological frame pairs necessary for edge mining. The implementation is shown at the bottom.

Table 1 summarizes the results of our approaches to accomplish the first stages of the sequential GTG. We concluded from our analysis the group-by approach is favorable for constraining the memory footprint.

	Original GTG	**Naïve port**	**Cartesian product**	**Group-by**
Level of Parallelism	1	<1	p	p
Number of Tasks	1	n	k	k
Total Memory	*O(n)*	*O(n)*	$O(n^2)$	2n = O(n)
Input Memory per task	*O(n)*	*2 = O(1)*	$O(n^2/k^2)$	2n/k = O(n/k)

Table 1: Algorithmic assessment of the approaches used to achieve the chronological frame pairing necessary for replicating the graph construction in the GTG

Within the Distributed GTG implementation in SciSpark we utilize the groupby approach. The next steps involve defining the task that identify the cloud elements with a user defined criteria. Then implementing a depth first search to find and label these cloud elements. The labeled component arrays are referred to as component frames within the Distributed GTG implementation. We now look at reducing the runtime for a single task in the graph construction job. A key point to note is that we needed to increase our input size in order to restructure the problem into a parallel workflow.

D. Graph vertex and edge mining from a single pair

Once the sRDD of consecutive frame-pairs is computed, each pair can be computed on independently. Reducing computations to independent tasks is paramount to MapReduce programming, as well as monitoring memory consumption associated with individual tasks. This section analyzes task runtime and the impact on overall job runtime as it relates to the graph construction in the GTG. The tests were run on a 4-node cluster, each node has 32 cores, 240 GB memory and 100 GB disk space. The tests record job runtimes to process 6000 frames of data type double, split over 350 partitions. The frame size was varied.

1) Cartesian product approach

In our initial approach, we extracted unique labeled components from each frame into masked component frames. From a single pair of frames, we extracted all component frames, then called a Cartesian product on the two lists of component frames. For small frames with few components this is a feasible operation. However for large frames with several components, the component-pair space becomes infeasible for a single task as shown in Table 2.

2) In-place Iteration approach

As we are considering a single task, we can rely on the shared-memory state of sequential programming since the enclosed function of a map operation is independent of other executing functions. This approach generated two component-labeled arrays for each pair, introduced a list to record overlapping labels, and stored the overall properties of each component in a hash table (as defined by the criteria used in the GTG). The two labeled matrices are multiplied to obtain a product matrix with non-zero values in positions of overlap. Since the dimensions of the labeled matrix and the product matrix are the same, we iterate over the product matrix and update our list of component edges for each non-zero value. For each component label encountered in the loop, we update the corresponding properties in the hash table. At this stage of the Distributed GTG implementation we leverage sequential programming techniques to write algorithms that consume fewer resources (see Table 2).

	Cartesian product	**In-place Iteration**
Average Number of Components per Frame	n	n
Number of Matrix Products	n^2	1

Table 2: Algorithmic analysis of the single task to the GTG construction approaches

Figure 9 provides the job runtime achieved on the 4-node cluster between the two task implementations.

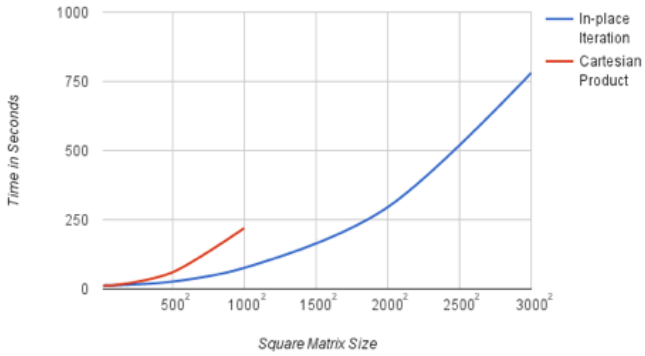

Figure 9: The job runtimes of the task approaches to the graph construction in the GTG. The matrices are randomly generated, factoring out the I/O latency incurred by ingesting physical data. Note that the component-wise Cartesian product implementation cannot scale beyond frames of 1000 x 1000. The in-place iteration approach scales to frames of 3000 x 3000. This is close to a terabyte of data being processed.

The final Distributed GTG is presented using the MapReduce paradigm in Figure 10. It is worth noting that optimizations, such as pre-aggregations over partitions containing consecutive time-frames, can be achieved by using an extra data structure to perform an In-Mapper aggregation. We do not describe such optimizations here as Apache Spark optimizes the DAG execution plan it generates.

```
1: class Mapper

2: // a time-frame id represents the actual
date/timestamp when the data was gathered.

3: method MAP(key frameId, sciTensor r)

4: EMIT(frameId, r)

5: // the (frameId + 1) operation outputs the next
sequential frameID

6: EMIT(frameId + 1, r)

1: class Reducer

2: method Reduce(key frameId, sciTensors [r1, r2, . .
.])

3: // Avoids using first and last frames

4: If sciTensors.length == 2 Then

5:         // label the components inside each sciTensor

6:         AL = labelComponents(sciTensors[0])

7:         BL = labelComponents(sciTensors[1])

8:          // outputs the edge found between those
consecutives time-frames

9: EMIT (overlappingComponents(AL, BL))
10: EndIf
```

Figure 10: The Distributed GTG algorithm using the MapReduce paradigm in SciSpark

E. Latency Trade-Offs with Apache Spark

We observed that Spark successfully mitigates I/O latency by only spilling to disk when necessary. However, Spark's reliance on the JVM combined with its greedy usage of memory has introduced another significant source

of latency in SciSpark, namely Java's "stop-the-world" garbage collection. We observed for matrices larger than 2500 x 2500 approximately 20 - 25% of the task time is waiting on garbage collection.

IV. Conclusions and Future Work

SciSpark provides an API that abstracts the methods to ingest scientific data into a distributed pipeline away from the end scientist user. For future work a solution that does not tightly couple reading hierarchical files with the HDFS version is required. One idea is to use binaryFiles to read the entire NetCDF files or binaryRecords to read specific offsets. The netCDF API would need to be integrated into these methods.

SciSpark abstracts the ND4J and Breeze linear algebra libraries behind a common interface for evaluation purposes. The performance of current and future multi-dimensional array Java libraries needs to be consistently evaluated. Towards this effort, SciSpark's design will provide seamless access to operations on multi-dimensional scientific datasets.

The SciSpark API provides developers with a clean architecture for contributing new methods to partition, extract, transform and load data from different formats. Partitioning in time was tested in this research. For future work, partition and extraction methods in the SciSparkContext of the API will be explored to achieve range partitioning in other dimensions.

Within SciSpark, we are able to process high resolution grids using a complex sequential-based algorithm without compromising on the original matrix size.

Our case study demonstrated that copying of data can lead to better use of resources in distributed applications. For the distributed implementation of GTG, it was found that creating a copy of the input data allowed for maintaining the chronological order necessary for the graph creation. This finding supports SciSpark's architectural design in the processing layer of creating a cache space for large jobs.

We found that SciSpark's architecture supports leveraging the advantages of both distributed and sequential programming to complete user-defined problems. The recommended approach is to construct jobs for parallel work while utilizing the shared-memory state of each independent task.

While the Cartesian product coupled with the filter is a powerful API feature for generating pairs, it is infeasible for Big Data applications. End users framing a Big Data problem should reformulate it within the MapReduce paradigm from the onset.

Acknowledgment

We acknowledge the AIST for the funding of this research under NASA proposal number 14-AIST-14-0034. Rahul Palamuttam and Renato Marroquín Mogrovejo would like to acknowledge the JPL Summer Internship Program for the opportunity and group 398M at JPL. We would also like to acknowledge the Apache Spark community and the ND4J developers.

References

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Communications of the ACM* 51, no. 1 (2008): 107-13.

[2] Zaharia, Matei, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets." *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*, 2010, 10.

[3] Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*. 2012, 2-2.

[4] Zitting, Jukka L., and Chris A. Mattmann. Tika in Action. Manning, 2012.

[5] Buck, Joe B., Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. "SciHadoop: Array-based Query Processing in Hadoop." Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11, 2011.

[6] Wilson, B. D., C. A. Mattmann, D. E. Waliser, J. Kim, P. Loikith, H. Lee, L. J. McGibbney, and K. D. Whitehall. "SciSpark: Highly Interactive and Scalable Model Evaluation and Climate Metrics." In AGU Fall Meeting Abstracts, vol. 1, p. 3772. 2014.

[7] Mattmann, C.A. "SciSpark: Interactive and Highly Scalable Climate Model Analytics". Presentation. Earth Science Technology Office, 2015.

[8] Loikith, P. C., B. R. Lintner, J. Kim, H. Lee, J. D. Neelin, and D. E. Waliser. "Classifying reanalysis surface temperature probability density functions (PDFs) over North America with cluster analysis." *Geophysical Research Letters* 40, no. 14 (2013): 3710-3714.

[9] Whitehall, Kim, Chris A. Mattmann, Gregory Jenkins, Mugizi Rwebangira, Belay Demoz, Duane Waliser, Jinwon Kim et al. "Exploring a graph theory based algorithm for automated identification and characterization of large mesoscale convective systems in satellite datasets." Earth Science Informatics: 1-13.

[10] Rew, Russ, and Glenn Davis. "NetCDF: an interface for scientific data access."*Computer Graphics and Applications, IEEE* 10, no. 4 (1990): 76-82.

[11] NCSA HDF Calling Interfaces and Utilities, Version 3.0, National Center for Supercomputing Applications, Univ. of Illinois at Urban-Champaign, Nov. 1989.

[12] Rosen, Joshua. "PySpark Internals". https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals (accessed September 1, 2015).

[13] Skymind. "ND4J: Scientific Computing for Java". http://nd4j.org/about.html (accessed September 1, 2015).

[14] Hall, David. "Scala NLP: Scientific Computing, Machine Learning, and Natural Language Processing". http://www.scalanlp.org/documentation/ (accessed September 1, 2015).

[15] Goto, Kazushige, and Robert Van De Geijn. "High-performance implementation of the level-3 BLAS." ACM Transactions on Mathematical Software (TOMS) 35, no. 1 (2008): 4.

[16] Kempler, Steve. "NCEP/CPC 4km Global (60N – 60S) IR Dataset Product Description". http://mirador.gsfc.nasa.gov/collections/MERG__001.shtml (accessed September 30, 2015).