Assessing the Dependability of Apache Spark System: Streaming Analytics on Large-Scale Ocean Data

Janak Dahal[1,2], Elias Ioup[3], Shaikh Arifuzzaman[1,2(✉)], and Mahdi Abdelguerfi[1,2]

[1] Computer Science Department, University of New Orleans, New Orleans, LA 70148, USA
[2] Canizaro Livingston Gulf States Center for Environmental Informatics, New Orleans, LA 70148, USA
{jdahal,smarifuz,mahdi}@uno.edu
[3] US Naval Research Laboratory, Stennis Space Center, Hancock, MS 39529, USA
elias.ioup@nrlssc.navy.mil

Abstract. Real-world data from diverse domains require real-time scalable analysis. Large-scale data processing frameworks or engines such as Hadoop fall short when results are needed on-the-fly. Apache Spark's streaming library is increasingly becoming a popular choice as it can stream and analyze a significant amount of data. In this paper, we analyze large-scale geo-temporal data collected from the USGODAE (United States Global Ocean Data Assimilation Experiment) data catalog, and showcase and assess the dependability of Spark stream processing. We measure the latency of streaming and monitor scalability by adding and removing nodes in the middle of a streaming job. We also verify the fault tolerance by stopping nodes in the middle of a job and making sure that the job is rescheduled and completed on other nodes. We design a full-stack application that automates data collection, data processing and visualizing the results. We also use Google Maps API to visualize results by color coding the world map with values from various analytics.

Keywords: Parallel performance · Fault tolerance · Streaming analytics · Apache spark · Hadoop · Temporal data · Large-scale system

1 Introduction

Processing and analyzing data in real time can be a challenge because of its size. In the current age of technology, data is produced and continuously recorded by a wide range of sources [3,4]. According to a marketing paper published by IBM in 2017, as of 2012, 2.5 quintillion bytes of data was generated every day, and 90% of the world's data was created since 2010 [22]. With new satellites, sensors, and websites coming into existence every day, data is only bound to grow exponentially. The number of users interacting with theses mediums are producing data at an enormous rate [1,2,15]. With the Internet reaching to new nooks and corners of the world, sources of potential data are ever-growing. As more data keep coming into existence, the necessity of a system that can analyze it in real-time becomes even more imminent. Although the concept of batch

G. Wang et al. (Eds.): DependSys 2019, CCIS 1123, pp. 131–144, 2019.
https://doi.org/10.1007/978-981-15-1304-6_11

processing (using multiple commodity machines in a truly distributed setting [18]) was a revolution when it first came into existence, it might not be a complete solution to the need for real-time processing. Such on-the-fly processing has applications in many areas such as banking, marketing, and social media. For example, identifying and blocking fraudulent banking transactions require quick actions by processing vast amounts of data and producing quick results. Sensitive and illegal posts on social media can be quickly removed to nullify the adverse effects on its users. Weather data, like the one used in this research, can be analyzed in real time to detect or predict different climatic conditions.

2 Background

The notion of using commodity machines as a computational power came into existence with the advent of Google File System (GFS). It introduced a distributed file system that excelled in performance, scalability, reliability, and availability [9]. As this truly distributed and replicated file system became rigidly stable, the next step in the ladder was to be able to process the data stored in it. For this, Google introduced MapReduce as a programming model [7]. This new parallel programming model demonstrated the ability to write small programs (map and reduce classes) for processing big data. It introduced the concept of offloading computation to the data itself and thus nullifying the effect of network bottleneck on batch processing by not having to move the input data between nodes. Hadoop is the most popular MapReduce framework today, but it has its limitations. The most prominent shortcoming of Hadoop lies in the iterative data-processing [23]. To extend Hadoop beyond conventional batch processing requires various third-party libraries. Storm can be used along with Hadoop to accomplish real-time processing [10]. Other libraries such as Hive, Giraph, HBase, Flume, and Scalding are designed to tackle specific operations, e.g., querying and graphing. Managing these different libraries can be time-consuming from a development point of view.

With Hadoop's limitations in mind, another large-scale framework Spark was designed that would reuse a working set of data across multiple operations [23]. The more iterative a computation is, the more efficient is the job running on Apache Spark. Spark streaming library has become widely popular to run real-time processing jobs. This library allows applications to stream data from different sources [14]. Some of the most popular streaming sources include Kafka, Flume, Twitter, and HDFS. Data can be streamed into the streaming job from one or more sources and unified into a single stream. For the application designed for this paper, data is streamed from the Hadoop File System (HDFS).

3 Apache Spark

Introduced in a paper published in 2010, Spark is a cluster computing framework that uses a read only collection of objects called Resilient Distributed Datasets (RDDs) that let users perform in-memory calculations on large clusters [24]. RDDs are fault-tolerant, parallel data structures which makes it possible to explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators [24]. As the intermediate results are stored

in memory, iterative analytics such as PageRank calculation, k-means clustering, and linear regression become much more efficient in Spark compared to Hadoop [10].

3.1 Resilient Distributed Data (RDD)

RDD is defined as a collection of elements partitioned across different nodes in a cluster that can be operated on in parallel [24]. From a user's point of view, it looks like a data structure, but behind the scenes, it performs all the operations necessary to run in a distributed framework. Failures across large clusters are inevitable; thus, the RDDs in Spark were designed with fault tolerance in mind. Since most of the operations in Spark are lazy (no operations are run on data unless an action, e.g., collect, reduce, etc., is called), the operations on RDDs are stored in the form of a Directed Acyclic Graph (DAG). A DAG is a collection of functional lineage such as map and filter. Such awareness of the functional lineage makes it possible for Spark to handle node failures gracefully [24]. These RDDs drive the streaming framework in Apache Spark. They have the following properties that make sure the Apache Spark Streaming maintains its integrity:

Replicated. RDDs are split between various data nodes in a cluster. Replicas are also spread across the cluster to make sure that the system can recover from any aftermath of the node crash. Processing occurs on nodes in parallel, and all RDDs are stored in memory on each node.

Immutable. When an operation is performed on an RDD, the original RDD is not changed. Instead, a new RDD is created out of that operation [24]. Only two operations are performed on an RDD namely *transformation* and *action*. A *transformation* transforms the RDD into a new one whereas an *action* gets data from the RDD.

Resilient. Resiliency pertains to the replication of data and storing the lineage of operation on RDDs. When a worker node crashes, the state of the RDD can be regenerated by running the same set of transformations to reach the current state of the RDD [24].

3.2 Apache Spark Streaming

In many real-world applications, time-sensitive data can often get stale very quickly. Thus, to make the most of such data, it must be analyzed on time. For example, if a banking website starts generating piles of 500 errors, the potential of an incoming request crashing the server must be evaluated in real time. Traditional MapReduce is not a viable solution for such cases as it is mostly suited for offline batch processing where results are not associated with any latency [23]. If the input data is repeatedly produced in discrete sets, multiple passes of the map and reduce tasks would create overhead which can be eliminated by using Spark instead. Apache Spark Streaming lets the program store results in an intermediate format in memory, and when new data arrives as another discrete set, it is batched to perform transformations on them quickly and efficiently [23]. Figure 1 outlines the Apache streaming framework.

Data can be streamed into Apache Spark streaming framework from various sources like Kafka, Flume, Twitter, and HDFS [17]. A receiver must be instantiated and hooked

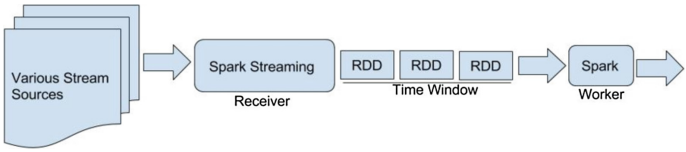

Fig. 1. Outline of Apache streaming framework used in this paper.

up with the streaming source to start the flow of data. One receiver can only stream data from one input source, and if we have multiple stream sources, then we can union them so that they can be processed as a single stream [12]. Once the receiver starts receiving the data from the streaming source, Spark stores the data into a series of RDDs delineated by a specified time window. After this time, the data is passed into the spark core for processing. To start any Spark streaming job, there needs to be at least two cores, one that receives the data as stream and one that processes the data.

4 Streaming Analytics on Large-Scale Ocean Data

We develop an application to run queries on a large oceanographic dataset and produce results on the fly. Apache Spark is chosen for a platform to write the application because of its streaming library. We stream data into the streaming job from HDFS. We collected data from United States Global Ocean Data Assimilation Experiment (USGODAE) data catalog and then processed and stored in the HDFS. The application streams new data within the configurable window of time and run transformations and actions to generate results.

4.1 Setup and Configuration

Although Hadoop is not required to run Spark, we installed it because our application reads data from HDFS. Hadoop was first installed on a single node setting, and then other nodes were added one at a time. Each time a node was added, the sample MapReduce tasks were run to make sure that the job was making use of all the nodes. Five nodes with identical computational power were used to create the cluster.

We install Apache Spark along with SBT and Scala. SBT is used to build the Scala projects. Scala is used as the programming language of choice to write streaming jobs. We install Spark in the same way as Hadoop by starting with a single node and adding one node at a time. Two workers instances (SPARK_WORKER_INSTANCES = 2) ran on each terminal to utilize dual CPUs. Each worker is set up to utilize up to 15 GB memory (SPARK_WORKER_MEMORY = 15 GB) and up to 16 cores (SPARK_WORKER_CORES = 16). We set up Hadoop File System (HDFS) on each of the nodes. YARN, a resource manager and a dashboard to visualize and summarize the metrics, runs on the driver node. We set up REPL environment or Spark-shell in each node to make sure that the debugging is swift when a transformation needs to be performed on a set of data. Figure 2 summarizes the Apache Spark installation.

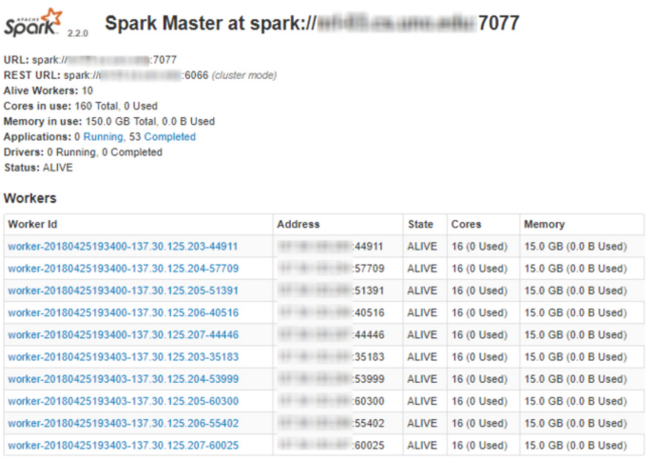

Fig. 2. Apache Spark system configuration used for our work. The IP addresses are hidden for privacy reason.

4.2 Description of Datasets

Data is generated every 6 h by an oceanographic model (NAVGEM-Navy Global Environmental Model) that predicts various environmental variables for the next 24 h to 180 h. The number of output files from the model depends on the type of variable. The data is generated for 198 different variables which cover the entire world with a precision interval of 0.5°. The model generates multiple files with the results, and each file contains only data for a single variable. The complete set of data for ten years is about 110 TB, but we have only about 4.5 TB disk space available in the distributed file storage. Therefore, we include only four variables for our experiments: ground sea temperature, pressure, air temperature, and wind speed. We use Panoply [10] as a GUI to visualize the input data and resulting data.

Our datasets cover the entire world, so the size of the data array is 361×720, where 361 represents all latitude points from $-90°$ north to $+90°$ north with 0.5° increments, whereas 720 represents all longitude points from 360° east to 0° NE with 0.5° increments as well.

Procedure for Data Collection. We collect our data using the following steps.

1. A Java program downloads the data into the filesystem.
2. NCAR Command Language was used to convert the data from GRIB (General Regularly-distributed Information in Binary form) format [11] into the NetCDF3 data format.
3. CDO (Climate Data Operators [19], written by the Max Planck Institute for Meteorology) was used to merge the data files so each file could contain data for multiple variables.
4. Files were copied to the HDFS using standard HDFS commands.

We wrote a bash script to automate the above steps and make them seamless.

4.3 SciSpark

Our application extends the functionality of the SciSpark [16] project by changing its open source code as needed. SciSpark library facilitates the process by mitigating the need to write wrapper classes to represent GRIB. The library provides a class called SciTensor that represented NetCDF data and implemented all basic mathematical operations such as addition, subtraction, and multiplication. We add new functions to SciTensor class to calculate maximum (*max*), minimum (*min*), and standard deviation. Other significant changes included logic to account for missing variables in a dataset. For multivariable analysis, we added relevant logic to create unique names for x and y axes when creating NetCDF result file with more than one variable. We create RDDs using SciTensor library and feed into the spark streaming queue.

4.4 Application in Use

Our application streams new files from a location in HDFS and writes the results back to HDFS. The job runs with a configurable time window and performs transformations and actions on all the RDDs accumulated during that time-frame. We use QueueStream API in Apache Spark to read the stream of new RDDs inside the streaming job. New RDDs are represented as a Discretized Stream (DStream) of type SciTensor. Spark Streaming API defines DStream as the fundamental abstraction in Spark Streaming and is a continuous sequence of RDDs (of the same type) [26]. Figure 3 summarizes the outline of our application.

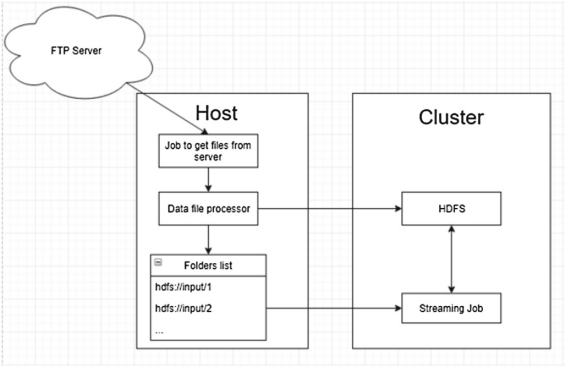

Fig. 3. A simplistic overview of our entire application. We design a full-stack application that automates data collection, data processing and visualizing the results.

A scheduled job running on the host runs every hour to download new data from the FTP server. After the download is completed, the data is processed and uploaded to HDFS. The streaming job running on the cluster processes these new files and update the result. The website running on a separate server polls the result file and visualizes the data using Google Maps.

5 Performance Evaluation

We use statistical analysis on data using Apache Spark Streaming to generate summary results in real-time. In the process, we also assess the dependability and scaling of the Spark streaming framework.

5.1 Complexity of the Operation

We evaluate and measure two significant steps in the streaming process namely *transformation* and *action*. We design multiple mathematical queries of varying complexity and run jobs to measure the performance of Apache Spark Streaming. For example, *average*, *maximum* and *minimum* are more straightforward mathematical operations, whereas *standard deviation* can be regarded as a more complex one. We perform the following statistical analyses: mean, max, min, and standard deviation. Once a user submits the streaming job, it cannot be changed for the lifetime of that job. The input sizes per streaming window for each job were approximately 180 MB, 500 MB, 1 GB, and 2 GB. The streaming window was set as 6 h for the streaming process because the input data is produced by the model every 6 h.

Variation of Each Statistical Analysis. Since GRIB1 data represents values in 361×720 2D arrays and the values are scattered across multiple files, to calculate an aggregate for each index, same indices across multiple files were aggregated. To calculate aggregate results for each latitude and longitude points, 361 and 720 more values in each file needed to be aggregated respectively. Moreover, calculating one single aggregate result for all the values across all the files increased the operation complexity as it had to aggregate more values. The variation in statistical analysis in ascending order of complexity is listed as follows: (i) one result for each combination of latitude and longitude points, (ii) one result for each latitude point, (iii) one result for each longitude point, and (iv) one single aggregate result for all data points.

Table 1 shows the average execution time for each variation of all four statistical analyses. It shows that the complexity of operation is directly proportional to the execution time. More transformations were required on data when running with variation 2, 3 and 4. Each additional transformation increased the length of the DAG and thus increased the execution time.

Multivariable Analysis of the GRIB Data. In addition to the above metrics, we also perform multivariable analysis to measure the latency of each streaming window. The same four statistical analyses were performed but with a varying number of variables. These analyses were serialized, thus increasing the number of transformations and actions for each additional variable. There was 50 GB data initially stored in HDFS which required longer execution time as each worker had to process more data. Each streaming window was once again fed with four different datasets of size 180 MB, 500 MB, 1 GB and 2 GB.

Figure 4 shows our results on the initial set of data. As expected, the execution time increases with the complexity of operation. The *standard deviation* operation took the most amount of time because the algorithm had multiple transformations to perform.

Table 1. Result for statistical analysis

Variation	Dataset size	DAG length	Execution time (s)
1	180 MB	5	22
	500 MB	5	37
	1 GB	5	49
	2 GB	5	81
2	180 MB	6	23
	500 MB	6	41
	1 GB	6	51
	2 GB	6	101
3	180 MB	6	22
	500 MB	6	43
	1 GB	6	60
	2 GB	6	117
4	180 MB	7	42
	500 MB	7	87
	1 GB	7	133
	2 GB	7	278

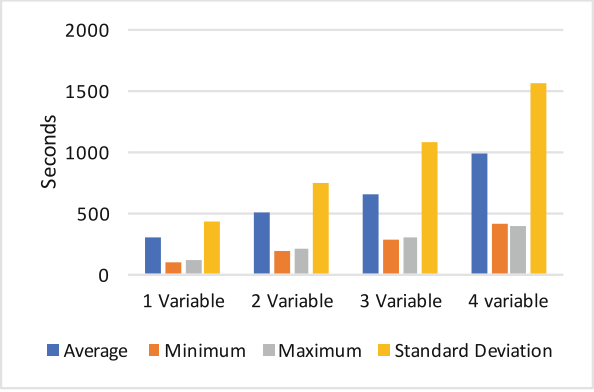

Fig. 4. Statistical Analysis of the initial set of data.

Further, as for DAG lengths, *standard deviation* has a larger DAG than those of *max* and *mean*. The length of a DAG is directly related to the latency of the corresponding streaming job. In other words, more map and filter functions are run on the dataset for operations with higher complexity.

The size of the dataset for each 6-h period was roughly 1 GB in size and latency for streaming 1 GB data was significantly smaller than the initial data. For input sources that generate discrete data at a regular interval, the streaming job is more suitable than

a batch processing job because of the lack of overhead in running an iterative job [6]. Figure 5 shows the results for batches of streams, which achieves better runtime performance than the initial set of data.

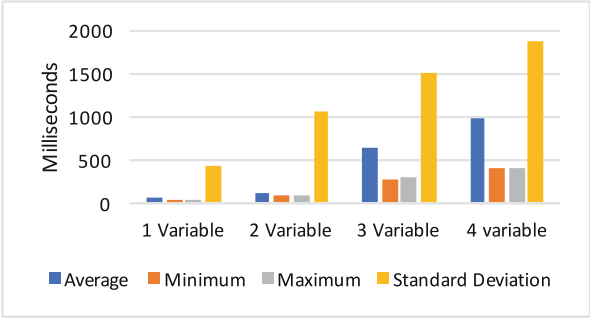

Fig. 5. Statistical Analysis of batches of stream.

5.2 Number of Executor Nodes

We ran streaming jobs with a different number of worker nodes to record the change in latency. Data was streamed from HDFS and YARN was used as a dashboard to visualize states of different worker nodes. Since Apache Spark utilizes the in-memory datasets [23], the multi-node setup outperformed the single node-setup as it could use more resources from each worker as shown in Fig. 6. It is clear from the figure that there is linear scalability in latency for a streaming job. This result shows that the efficiency of a streaming job is directly proportional to the number of workers.

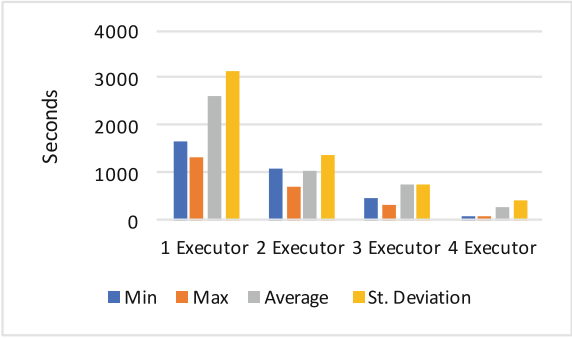

Fig. 6. Statistical analysis on initial Transformation vs. # of executors.

5.3 Scalability

As data grows and higher processing speed is desired, new nodes should be easily added to the cluster. During our experiment, nodes were killed and started in the cluster with a fair speed and easiness. We wrote bash scripts to control the state of a node and used YARN dashboard to verify the state. Figure 7 shows the state of the cluster after multiple nodes were killed. Further, Figs. 8 and 9 showcase how different metrics of a streaming job can be visualized using Apache Spark's dashboard. Figure 8 plots the scheduling delay and Fig. 9 plots the processing time for batches ran with the different number of executor nodes. Yellow represents six executors, brown five executors, and purple three executors. The sizes of datasets in different batches were 180 MB, 500 MB, and 1 GB. The scheduling delay and processing time are both directly proportional to the size of the data and inversely proportional to the number of worker instances.

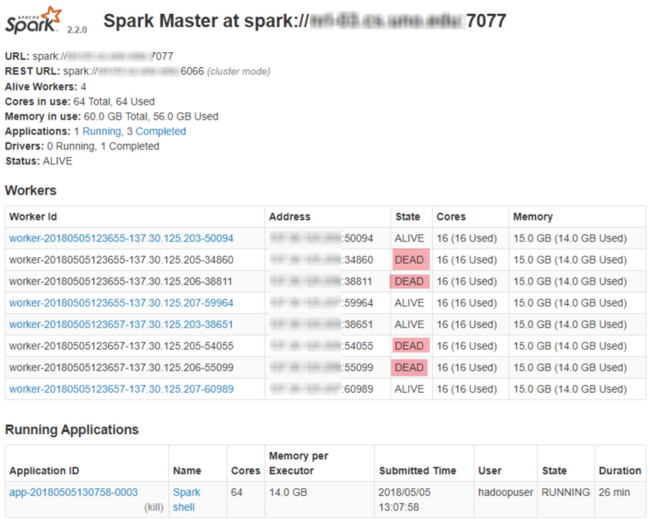

Fig. 7. Status of dead workers on YARN dashboard.

Fig. 8. Scheduling delay for different datasets with the varying number of executors. Yellow represents six executors, brown five executors, and purple three executors. (Color figure online)

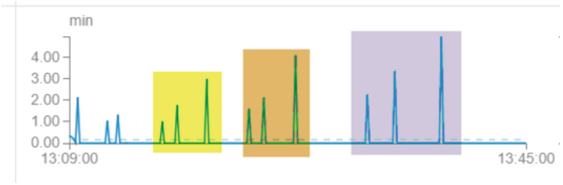

Fig. 9. Processing time for different datasets with the varying number of executors. Yellow represents six executors, brown five executors, and purple three executors. (Color figure online)

5.4 Fault Tolerance

Spark can reconstruct the RDDs using lineage information stored in the RDD objects when a node falls apart [23]. Since the data is already replicated across nodes in HDFS, lost partitions can be reconstructed in parallel across multiple nodes without much overhead. If the node running receiver fails, then another node is spun up with the receiver which can continue to read from HDFS. If the receiver was using Kafka or Flume as a source instead of HDFS, then a small amount of data may be lost which hasn't been replicated to other nodes in the cluster [6]. We measure performance of a system running streaming job with various node failures to access the fault tolerance capability of the Apache Spark streaming. Spark's dashboard interface was used to visualize the difference in latency for different batches running with and without node failures. Figure 10 shows that if some nodes fail while running a batch, it will take longer to account for the lost nodes and reschedule those jobs in different node/s. For instance, stage ID 520 lost a node with two workers, and the driver had to reschedule seven tasks running on that node somewhere else. As a result, the latency increased from 2.9 to 5.1 min.

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
520	Streaming job from [output operation 0, batch time 13:52:20] reduce at <console>:38 +details	2018/05/05 13:52:20	5.1 min	19/19 (7 failed)	997.1 MB
352	Streaming job from [output operation 0, batch time 13:38:20] reduce at <console>:38 +details	2018/05/05 13:38:20	4.5 min	13/13	997.1 MB
214	Streaming job from [output operation 0, batch time 13:26:50] reduce at <console>:38 +details	2018/05/05 13:26:54	2.9 min	19/19	997.1 MB

Fig. 10. Difference in processing time for node failures. The first row demonstrates the execution time with node failures.

5.5 Visual Application

We develop a web interface to demonstrates a sample usage of our application. The web page uses Google Maps and its developer API to visualize the results generated by our application. The web application is written in .NET MVC framework. The server-side code grabs the latest result from the cluster by using the WinSCP library (this was used to avoid installing FTP on the master in the cluster), then converts the results into text format using ncl_dump. A text dump of the resulting NetCDF file was processed and sent to the view. A JavaScript function regularly polls for the result, and once the Spark application generates the result, it is visualized on the web (Fig. 11).

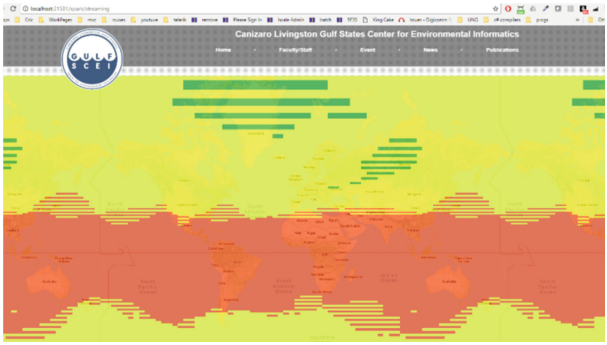

Fig. 11. Screenshot of color-coded representation of the result. This UI visualizes a single variable result file by color coding the latitude and longitude over google map based on the value of the variable for that coordinate.

6 Additional Observations and Findings

Spark Streaming vs. Hadoop's Batch processing vs. Storm Trident. An iterative job like the one used in this experiment can be expressed as multiple Map and Reduce operations in Hadoop. However, different MapReduce jobs cannot share data. So for iterative analysis, the same dataset must be read from HDFS multiple times, and results would need to be written to HDFS many times as well [5]. These iterations create much overhead because of the I/O operations and other unwanted computations [8]. Spark tackles these issues by storing intermediate results in memory. Spark Streaming uses D-Streams or discretized streams of RDDs which provides consistent, "exactly-once" processing across the cluster [25] and thus significantly increases the performance for iterative analysis. Apache Storm can process unbounded streams of data in real time, and it can be used alongside Hadoop, but it only guarantees "at-least-once" processing [20]. Trident bolsters Storm by providing micro-batching and other abstractions that would ensure "exactly-once" processing [21]. It would take three different libraries to work seamlessly to accomplish what Spark Streaming can accomplish by itself. Time and effort required to setup and maintain Storm Trident application along with Hadoop can hamper the production and deployment. In contrast, Spark's Streaming library is directly written over its core and maintained by the same group who maintain the core's code base. Thus, Spark streaming outshines both Hadoop and Storm Trident combination for streaming scientific data.

Limitations of Spark. When a dataset is large enough not to allow any more RDDs to be stored in memory, Sparks starts to replace RDDs, and such frequent replacement degrades the latency [13]. However, for this work, we needed a framework that would seamlessly stream a relatively large datasets, and Spark Streaming was able to handle it efficiently.

7 Conclusions

We use SciSpark successfully with Apache Spark to stream GRIB1 data in a streaming application. The bulk of the logic in this application lies in the ability to convert the statistical analysis into transformations and actions that would run upon the DStream of RDDs of type SciTensor. Datasets ranging from 180 MB to 50 GB were used in the application without running into any memory issues. Various properties of a streaming application like operation complexity, scalability and fault tolerance were assessed, and results were summarized using simple mathematical operations like mean, min/max and standard deviation. Based on these results and other properties of apache Spark Streaming, we are confident that Spark Streaming is a better solution to stream the scientific data over Hadoop or Storm Trident.

Acknowledgments. Part of this work was supported by ONR contracts N00173-16-2-C902 and N00173-14-2-C901, and Louisiana Board of Regents RCS Grant LEQSF(2017-20)-RD-A-25.

References

1. Arifuzzaman, S., Khan, M.: Fast parallel conversion of edge list to adjacency list for large-scale graphs. In: Proceedings of the 23rd High Performance Computing Symposium (HPC 2015), Alexandria, VA, USA, pp. 17–24, April 2015
2. Arifuzzaman, S., Khan, M., Marathe, M.: A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. In: 2015 IEEE BigData Conference (2015)
3. Arifuzzaman, S., Khan, M., Marathe, M.V.: PATRIC: a parallel algorithm for counting triangles in massive networks. In: Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM 2013), San Francisco, CA, USA, pp. 529–538, October 2013
4. Arifuzzaman, S., Pandey, B.: Scalable mining, analysis, and visualization of protein-protein interaction networks. Int. J. Big Data Intell. (IJBDI) **6**(3/4), January 2019. https://doi.org/10.1504/IJBDI.2019.10019036
5. Bu, Y., et al.: Haloop: efficient iterative data processing on large clusters. Proc. VLDB Endow. **3**(1–2), 285–296 (2010). https://doi.org/10.14778/1920841.1920881
6. Cordava, P.: Analysis of real time stream processing systems considering latency. White paper (2015)
7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. Ekanayake, J., et al.: Twister: a runtime for iterative mapreduce. In: 19th ACM International Symposium on High Performance Distributed Computing, pp. 810–818 (2010). https://doi.org/10.1145/1851476.1851593
9. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. SIGOPS Oper. Syst. Rev. **37**(5), 29–43 (2003). https://doi.org/10.1145/1165389.945450
10. Gopalani, S., Arora, R.: Comparing apache spark and map reduce with performance analysis using K-means. Int. J. Comput. Appl. **113**(1), 8–11 (2015)
11. GRIB: Converting grib (1 or 2) to netcdf. www.ncl.ucar.edu/Applications/griball.shtml (2018). Accessed 9 Dec 2018
12. Grulich, P.M., Zukunft, O.: Bringing big data into the car: Does it scale? In: 2017 International Conference on Big Data Innovations and Applications (Innovate-Data), pp. 9–16 (2017). https://doi.org/10.1109/Innovate-Data.2017.14

13. Gu, L., Li, H.: Memory or time: performance evaluation for iterative operation on hadoop and spark. In: 2013 IEEE 10th International Conference on High Performance Computing and Communications, pp. 721–727 (2013). https://doi.org/10.1109/HPCC.and.EUC.2013.106
14. Krob, J., Krcmar, H.: Modeling and simulating apache spark streaming applications. Softwaretechnik-Trends **36**, 1–3 (2016)
15. Motaleb Faysal, M.A., Arifuzzaman, S.: A comparative analysis of large-scale network visualization tools. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 4837–4843, December 2018. https://doi.org/10.1109/BigData.2018.8622001
16. Palamuttam, R., et al.: SciSpark: applying in-memory distributed computing to weather event detection and tracking. In: 2015 IEEE International Conference on Big Data (Big Data), pp. 2020–2026 (2015). https://doi.org/10.1109/BigData.2015.7363983
17. Salloum, S., et al.: Big data analytics on apache spark **1**(3), 145–164 (2016). https://doi.org/10.1007/s41060-016-0027-9
18. Sattar, N.S., Arifuzzaman, S.: Overcoming mpi communication overhead for distributed community detection. In: Majumdar, A., Arora, R. (eds.) Software Challenges to Exascale Computing, pp. 77–90. Springer, Singapore (2019)
19. Schulzweida, U., et al.: CDO user's guide: Climate data operators, April 2018
20. Toshniwal, A., et al.: Storm@ twitter. In: 2014 ACM SIGMOD International Conference on Management of Data, pp. 147–156 (2014). https://doi.org/10.1145/2588555.2595641
21. Trident, A.: Trident tutorial (2018). https://storm.apache.org/documentation/Tridenttutorial.html. Accessed 9 Dec 2018
22. Winans, M., et al.: 10 key marketing trends for 2017 (2017). www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WRL12345USEN. Accessed 9 Dec 2018
23. Zaharia, M., et al.: Spark: cluster computing with working sets. In: 2nd USENIX Conference on Hot Topics in Cloud Computing, 22–25 June 2010
24. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: 9th USENIX Conference on Networked Systems Design and Implementation, p. 2 (2012)
25. Zaharia, M., et al.: Discretized streams: fault-tolerant streaming computation at scale. In: Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 423–438 (2013). https://doi.org/10.1145/2517349.2522737
26. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664