MARISSA: MApReduce Implementation for Streaming Science Applications

E. Dede, Z. Fadika, J. Hartog, M. Govindaraju
SUNY Binghamton
Binghamton, NY 13902
Email:{edede1,zfadika,jhartog1,mgovinda}@cs.binghamton.edu

L. Ramakrishnan, D. Gunter, R. Canon
Lawrence Berekely National Lab
Berkeley, CA 94720
Email:{lramakrishnan,dkgunter, scanon}@lbl.gov

***Abstract*—MapReduce has since its inception been steadily gaining ground in various scientific disciplines ranging from space exploration to protein folding. The model poses a challenge for a wide range of current and legacy scientific applications for addressing their "Big Data" challenges. For example: MapReduce's best known implementation, Apache Hadoop, only offers native support for Java applications. While Hadoop streaming supports applications compiled in a variety of languages such as C, C++, Python and FORTRAN, streaming has shown to be a less efficient MapReduce alternative in terms of performance, and effectiveness. Additionally, Hadoop streaming offers lesser options than its native counterpart, and as such offers less flexibility along with a limited array of features for scientific software. The Hadoop File System (HDFS), a central pillar of Apache Hadoop is not a POSIX compliant file system. In this paper, we present an alternative framework to Hadoop streaming to address the needs of scientific applications: MARISSA (MApReduce Implementation for Streaming Science Applications). We describe MARISSA's design and explain how it expands the scientific applications that can benefit from the MapReduce model. We also compare and explain the performance gains of MARISSA over Hadoop streaming.**

I. Introduction

Evolving scientific instruments and the rapid sophistication of computing systems have resulted in large-scale scientific simulations and data analysis workflows. Today, scientists in a variety of disciplines such as earthquake simulation [32], bioinformatics [13], climate science [25], and astrophysics [9], generate data at increasingly larger scales than was possible before. As more and more scientific data is generated, our ability to effectively manage and process such data also needs to evolve. MapReduce, since its introduction at the 6th USENIX Symposium on Operating Systems Design and Implementation [14], has been widely used to this end. The MapReduce model is inspired from functional programming. The model allows a the uniform application of *map* and *reduce* functions to nearly equally split data amongst participating nodes. Among its most attractive qualities, the MapReduce model counts: inherent data management, parallelization/synchronization abstraction and fault-tolerance. For the scientist or the programmer, this means the advantage of being absolved from providing parallelization and synchronization features to programs, as those features are automatically managed by the framework. Similarly, data management and fault-tolerance (in case of node failures) are abstracted away from the user and are instead the responsibility of the MapReduce framework. Apache Hadoop [1], the most widely used MapReduce framework, provides these same advantages. Hadoop native does not provide support for application source code written in languages other than Java. While Hadoop streaming attempts to address this problem in enabling scripts and executable binaries to run on its framework, our previous work [19] has shown (see **Table** I and **Figure** 2 for summarized results) the negative performance impact displayed by streaming applications to be considerable. In this paper, we use the word streaming as in the context of Hadoop, which provides a mechanism to run non-Java applications from within the context of a Java-based MapReduce framework.

Hadoop MapReduce relies on the Hadoop Distributed File System (HDFS) [33], a non POSIX compliant filesystem, for its data and cluster operations. Super computing facilities such as the National Energy Research Scientific Computing Center (NERSC) [5], part of Lawrence Berkeley National Laboratory (LBNL) and scientific cluster computing centers such as TeraGrid [8] primarily rely on POSIX-compliant file systems. Thus, for scientific computing, filesystems such as GFS2 [34], GPFS [31], Lustre [3], rather than HDFS are widely adopted, making the adoption of MapReduce difficult, and reducing availability to scientists. Finally, Hadoop streaming, in its current form, although capable of generic data-intensive computing, lacks features most attractive for scientific applications. We present in this paper, MARISSA (MApReduce Implementation for Streaming Science Applications), a MapReduce framework offering better performance and faster application turnaround time than Hadoop streaming, while capable of fully supporting a variety of POSIX compliant file systems.

The contributions of this paper are the following:

- We present the design and implementation of a MapReduce streaming framework capable of running not only Java applications, but also any executable binary.
- Provide evidence illustrating a considerable performance improvement over Hadoop streaming both under normal and under availability variations.

II. Running Scientific Applications with MapReduce

The applicability of the MapReduce model for science has been widely discussed in recent years [16], [29], [30]. We identify several challenges in **Section** II-A and subsequently offer an array of solutions to each of the highlighted challenges in describing MARISSA's design and architecture.

A. Scientific Application challenges

The challenges identified in this section are based on our collaboration with scientists working with the MapReduce model for data-intensive processing applications.

1) Apache Hadoop: Apache Hadoop is an open-source MapReduce software framework. Hadoop relies for its functions on the Hadoop Distributed File System (HDFS) [33], a derivative of the Google File System (GFS) [20]. In its function as data management and fault-tolerance support system, the HDFS splits, replicates and hosts input blocks on cluster nodes as the user provides input data to the framework. These nodes are identified as `Datanodes`, and are responsible for holding input blocks and their replicas. As the nodes can directly process the chunks they hold, Hadoop is able to provide a sense of locality to processing nodes hosting input data. The replication methodology in Hadoop also functions as a fault-tolerance mechanism when cluster nodes experience failures. When nodes fail, replicas of their input chunk can be found on peer nodes which are also part of the cluster. In such cases, the peers take charge of processing chunks pertaining to the fallen machines. The same functions also allow Hadoop MapReduce to implement speculative execution in anticipating potential failures. While all of the above mentioned properties help in ensuring a fail safe framework, we have shown that such measures can also hinder performance [18].

Hadoop clusters additionally consist of a `JobTracker`, usually the *master* node, and `TaskTrackers`, also known as *worker* nodes. Upon job submission, the `JobTracker` reviews the job specifications and determines the number of input splits for the job. A JAR file provided by the user contains *map* and *reduce* functions written by the application programmer/user. `TaskTrackers`, depending on their status execute the *map* and *reduce* functions provided by the user. `TaskTrackers`, also known as *worker* nodes, notify the `JobTracker` of their ability to perform work in a heartbeat message, and are then assigned tasks by the *master* node. With streaming, the process is similar, with the exception that the task communicates with the executable application, written in any language. Streaming communication takes place using standard Input/Output streams, as well as network sockets. An illustration of Hadoop streaming as a comparison to Hadoop native is illustrated by **Figure** 1.

Regardless of the processing model in use, be it native or streamed, Hadoop launches *map* operations on each piece of input. The *map* function is simultaneously applied to every value in the input by each node. This process produces an intermediary list of new key/value pairs, which are, in turn, passed to the reducer: $\texttt{Map}(Key_1, v_1) \rightarrow \texttt{list}(Key_2, v_2)$.

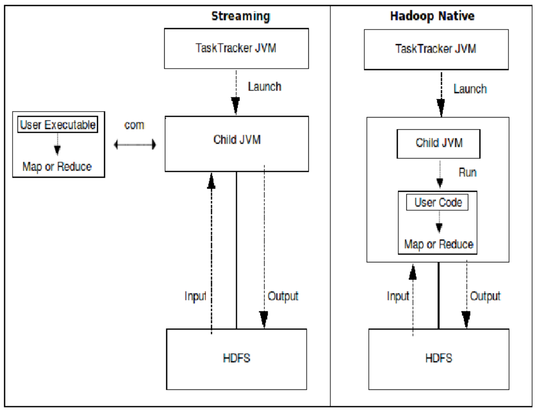

Fig. 1. Architectural comparison between Hadoop streaming and Hadoop native

The reduction process then takes place with the intermediary list produced in the previous *map* step. Key/value pairs are supplied to the *reduce* function, which sorts and rearranges them by key. The function then iterates through each pair and applies the *reduce* logic based on similar keys. $\texttt{Reduce}(Key_2, \texttt{list}(v_2)) \rightarrow \texttt{list}(v_2)$.

B. Apache Hadoop Challenges

TABLE I
Performance baseline comparison filtering Wikipedia data between C and Java programs on a single node in minutes

Input Size(TB)	C	Java
0.1	98.67	124.01
0.2	198.25	251.93

1) Streaming imposes a performance penalty: Our previous work [19] shows that the Hadoop streaming model incurs a performance penalty (See **Table** I and **Figure** 2 for summary). In our baseline experiment, we perform filtering of Wikipedia data using 1 node (specifications are described in **Section** III). Both our Java and C programs use the same algorithm. As expected we notice a faster runtime from the C program over its Java counterpart. Subsequently, the same C program is streamed through Hadoop while the Java program is provided to Hadoop native. No reducing occurs in either case. Streaming the C program proves slower than running its Java counterpart with Hadoop. We subsequently isolated the I/O performance footprint from the application turn around time as shown in **Figure** 3. While C still performs faster I/O operations while streaming, its performance when compared with Hadoop streaming can be seen as lacking. The I/O isolation shows that the performance downfall can thus be pinpointed in the execution of our C program by Hadoop streaming. This shows that streaming imposes a performance penalty, which grows with input size.

2) Lack of support for multiple executables: A scientific application might need each of its nodes to run each a different or a specific executable program as part of the *map* phase. A

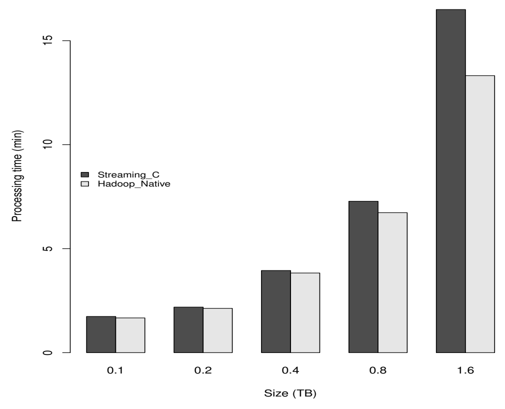

Fig. 2. Streaming counterpart of **Table** I. At 1.6TB the "C" streaming version is about 19% slower than Hadoop native. Note that the 'C" vs Java baseline experiments in **Table** I showed that 'C" was 20% faster than Java for the same applications and systems. Thus the actual overhead is even higher.

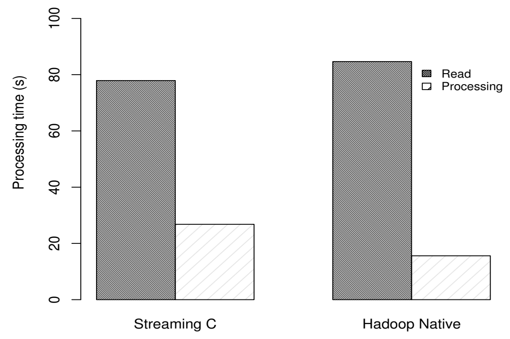

Fig. 3. Following **Table** I and **Figure** 2, this shows streaming performance as not a factor of I/O speeds in this case. In fact 'C" streaming performs faster I/O than Hadoop native. The performance losses occur in processing. This graph shows processing of 1.6TB of Wikipedia data being filtered to isolate embedded indices using 75 nodes in both cases.

variant of this problem could require all nodes to run the same executable program with different arguments. Such a feature although common and useful in scientific data processing, is not addressed by Hadoop streaming.

3) Lack of support for individual input: In a similar manner, each node might be required to process a specific file or input dataset. While some nodes have different specifications from others, it might be desirable to process specific files on given nodes. For instance, it might be beneficial to process parts of the input consisting of floating point numbers on a node specialized for fast floating point operations, while other nodes busy themselves with *String* filled parts of an input dataset. With Hadoop this feature is not possible.

4) Lack of iterative support: Scientific applications such as climate simulations [25] can be iterative in nature. As such, after a run, the application might need to assess its own output and determine if subsequent runs are needed in order to achieve the desired precision. Subsequent runs should automatically be scheduled in such cases. While MapReduce frameworks like Twister [6] offer iterations, Apache Hadoop does not. Twister, however, only provides Java program support, and no binary executable support. Similarly, Haloop [11] is a modified version of Hadoop targeting iterative applications. Haloop provides a programming interface that users can use to express the iterative nature of their applications. Haloop, however, like Twister and unlike MARISSA, does not provide executable binary support.

5) Lack of support for user specified redundant tasks: Although Hadoop launches speculative tasks, such tasks are discarded when primary tasks complete. A scientific application might require multiple nodes to execute the same tasks for result correctness comparison in the reduce phase. This feature is currently not supported by Hadoop streaming.

6) HDFS is not POSIX compliant: Most modern and legacy scientific applications require POSIX compliance on the part of the filesystem. Thus, scientific facilities and cluster centers such as NERSC and TeraGrid feature POSIX filesystems. As we have previously shown, the lack of POSIX compliance by HDFS renders its adoption in such facilities difficult, if not impossible at the moment.

7) Hadoop streaming requires `STDIN`/`STDOUT`: Scientific applications not relying on standard Input/Output for processing do not qualify to run with Hadoop streaming. The Hadoop streaming MapReduce model currently requires applications to accept input only through `STDIN` and write out only to `STDOUT`. This measure disqualifies many scientific applications from benefiting from the model. Although one could re-write or modify an application for compliance, both large and legacy applications do not always benefit from such a possibility.

C. Solutions implemented in MARISSA

Faced with the challenges highlighted above, we have implemented MARISSA, a MapReduce framework allowing for any executable program to serve as *map* and/or *reduce* functions. MARISSA also allows for:

- Full compatibility with POSIX filesystems.
- Iterative application support: The ability for an application to assess its output and schedule further executions.
- The ability to run a different executable on each node.
- The ability to run different input datasets on different nodes.
- The ability for a subset (or all) of the nodes to run the same application, allowing the reduce step to decide which result to select if applicable.
- The ability for programs not reading from `STDIN` and writing to `STDOUT` to be executed.

D. Architecture and Design of MARISSA

As a MapReduce framework, MARISSA needs to adhere to the three main pillars of the model:

- Data management
- Synchronization/parallelization abstraction
- Fault-tolerance

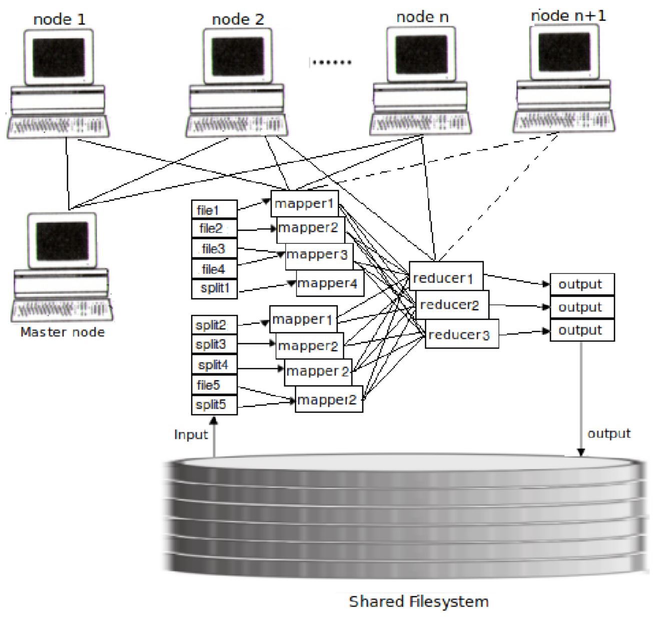

Fig. 4. MARISSA Architecture. The shared file system is used for both input and output placement. Executables accessible by individual workers can be used for map or reduce functions.

This is achieved through three principal modules, namely the `Input/Output Manager`, the `Job Controller`, and the `FaultTracker` or Fault-Tolerance module. **Figure** 4 illustrates the architecture of the framework.

1) The case for shared-disk Filesystems: Most supercomputing centers today use POSIX compliant filesystems. As Hadoop depends on HDFS for its operations, the adoption of the MapReduce model for large scale, data-intensive, scientific processing in such centers is hindered by the requirement of HDFS adoption. This change would mean either entire cluster reconfiguration to install HDFS, or the isolation of a handful of resources for the same purpose. MARISSA allows the use of the shared-disk filesystems, without the need for modifying the host environment. This feature increases the accessibility of the MapReduce model to be used by a wider array of scientific applications.

2) Input Management: Input management traditionally encompasses both input splitting as well as input distribution, or solely splitting as is the case with MARISSA. As with MARIANE [17], while Hadoop and most MapReduce frameworks [6] apportion the input amongst participating nodes and then transfer all chunks to their destinations, MARISSA relies on the shared filesystem as all nodes benefit from file system visibility. The framework leverages the data visibility offered by the shared-disk filesystem to provide each node with the data needed for processing. Thus, no transfer occurs on the application level, but it rather directly leverages the optimizations at the filesystem level. This model allows us to focus on *mapping* and *reducing*, without the need for I/O management by the MapReduce framework during application runtime as is the case with Hadoop. Instead of forcing all files to be split across the cluster, this modus operandi enables us to direct nodes in the cluster to use specific files. This allows MARISSA to direct nodes suited to work on given types of inputs to do so, or similarly, machines geographically close to certain resources to process them. In Hadoop's case, directing nodes to given input datasets is not possible, as nodes do not have visibility of the entire input, but rather of pieces of it. MARISSA's model thus permits for redundant task execution, whereby two nodes can be directed to perform tasks on the same file or input dataset. This may be needed in an effort to confirm a result, or ensure its precision. Additionally, through the `Input Manager`, we opted for both providing input to applications through standard I/O as with Hadoop, but also through file access, allowing applications not espousing the standard I/O model to be applicable to the framework. Finally, MARISSA, by working with shared-disk filesystems, allows for the ability to work with a myriad of POSIX filesystems, and the ability for the framework to benefit performance-wise from highly parallel and optimized filesystems such as GPFS-SNC [22] and GPFS [31].

3) Job Control: Job control consists of scheduling each node to spawn a process tied to the specified input file or dataset present on the shared disk. Nodes are assigned applications to run on a given input dataset. They execute the binary provided by the user against the local data they hold. This enables each of them to execute a different binary. Unlike Hadoop streaming, MARISSA can either start different applications on each node or use different parameters for the same application on each node.

Hadoop applies the same *mapper* code and argument to all workers, and as such is unable to tailor execution statements for each of its nodes. For example, in a heterogeneous cluster, it might be desirable to assign tasks by matching the task requirements to the nodes that have the desired configuration in terms of memory capacity and processor speed. This is a pertinent case with BLAST [24], a protein search and comparison application that allows for the number of cores it uses to be passed through the command line.

Through the `Job Controller`, nodes can iteratively re-execute their own tasks should they find it necessary. The `Job Controller` monitors task progress from the cluster nodes, and records broken pipes and non-responsive nodes as failed. A completion list of the different sub-tasks performed by the nodes is kept in the *master* node's data structure. Upon failure, the completion list is communicated to the `FaultTracker`, also known as the `Fault-tolerance` module. Slow nodes are similarly accounted for, and their work is re-assigned to available machines that have completed their work. In a situation caused by two nodes running similar jobs, in the specific case of a slow node's job being rescheduled, the system registers whichever sub-job completes first. This particular scheme is akin to Hadoop's *straggler* suppressing mechanism, and serves as a load balancing maneuver.

4) Fault-tolerance: Hadoop uses task and input chunk replication as a means to ensure fault-tolerance. With MARISSA, the approach used employs a node specific fault-tolerance mechanism, rather than an input specific one. Should a node fail with Hadoop, replicas of the chunk it held are located on other nodes, and the nodes found to be holding them are tasked with running a *rescue* job. This implies a high disk space requirement as file sizes grow. Further more, given large failures, or failures involving a chunk and its replicas, a total failure is a possibility on Hadoop's part. With MARISSA's approach, node failure is inherently decoupled from data availability as the data is held by the shared filesystem. This means that a user code exception or failure does not jeopardize the job. Upon failures, the *master* is notified through an exception handler, or a broken pipe in the case of sudden node failures. The master then resubmits the job to be executed by the first available node. As all nodes benefit from full visibility on the disk, re-execution simply consists of task re-assignment with no application-level data movement from the failed node to the rescuing node. Given MARISSA's fault tolerance model, should massive failures occur, or should the rescuing node itself fail, the fault-tolerance operation is recursive. Failed rescuers are themselves added to a completion table and the `Fault-tolerance` module runs until all tasks are completed, or until all existing nodes die or fail. This means that unlike Hadoop, losing a substantial number of nodes, typically more than half, will not affect the whole job, but rather rescue attempts are made until the last living node dies.

III. Performance Results

In this section we compare the performance of MARISSA and Hadoop streaming for two classes of scientific applications: BLAST and K-means clustering. BLAST (Basic Local Alignment Search Tool) [24], is used for comparing primary biological sequence information, such as the amino-acid sequences of different proteins or the nucleotides of DNA sequences. A BLAST search enables a researcher to compare a query sequence with a library or database of sequences, and identify library sequences that resemble the query sequence above a certain threshold. We chose both K-means and BLAST as K-means is CPU-intensive, and BLAST presents substantial memory demands. Choosing both a CPU and a Data-intensive application ensures coverage of a wide array of scientific applications.

The following experiments were performed on the Grid and Cloud Computing Research Lab Cluster at Binghamton University.

- Dual core – One desktop-class machine, which has a single 2.4Ghz Intel Core 6600 with 2 GB of ECC RAM, running Linux 2.6.24.
- Quad core – Nodes in a cluster, each of which has two 2.6Ghz Intel Xeon CPUs, 8 GB of RAM 4 cores, and run a 64 bit version of Linux 2.6.15
- 8 core – Nodes in a cluster, each of which has two 2.6Ghz Intel Xeon CPUs, 8 GB of RAM 8 cores, and run a 64 bit version of Linux 2.6.15.
- 48 core – Nodes in a cluster, each of which has two 2.6Ghz Intel Xeon CPUs, 16 GB of RAM 48 cores, and run a 64 bit version of Linux 2.6.15.

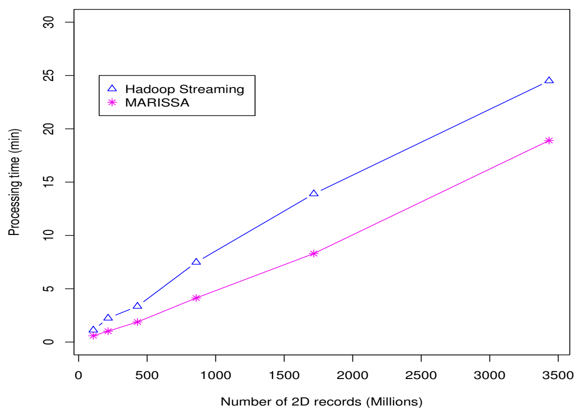

Fig. 5. 80-core Hadoop streaming and MARISSA clusters, each performing K-means clustering on 0.1 Billion to 3.4 Billion 2D-records, using 20 nodes. MARISSA here, given the CPU-intensive nature of the application performs up to 47% faster than Hadoop streaming.

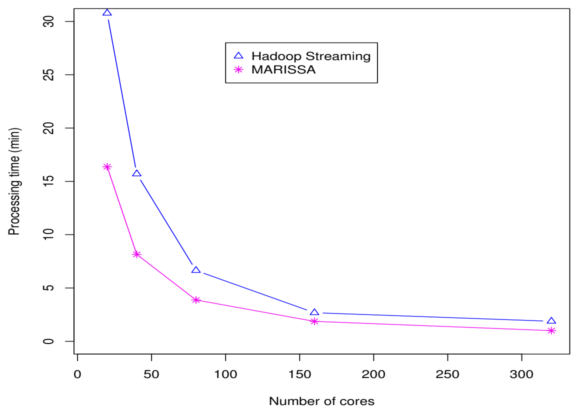

Fig. 6. K-means clustering on both streaming MapReduce frameworks, processing 858 million 2D-records given different cluster sizes ranging from 5 to 80 nodes.

Figure 5 shows K-means clustering on data increases from 0.1 Billion 2D-records to 3.4 Billion. MARISSA, because it is free of data management during runtime, performs best here as the framework can solely focus on *mapping* and *reducing*. Hadoop also supports `Datanodes` on its `TaskTrackers`, potentially inhibiting the worker nodes. MARISSA performs up to 47% faster than Hadoop streaming.

Figure 6 shows K-means clustering on cluster sizes ranging from 20 to 320 cores, which corresponds to a range of 5 to 80 nodes. MARISSA here performs up to 46.8% faster than Hadoop with 5 nodes, and up to 47% with 80 nodes.

In **Figure** 7, we show the increasing performance disparity between Hadoop streaming and MARISSA while both are running BLAST in the face of data increases. MARISSA runs up to 13.6% faster than Hadoop streaming for 300,000 sequences,

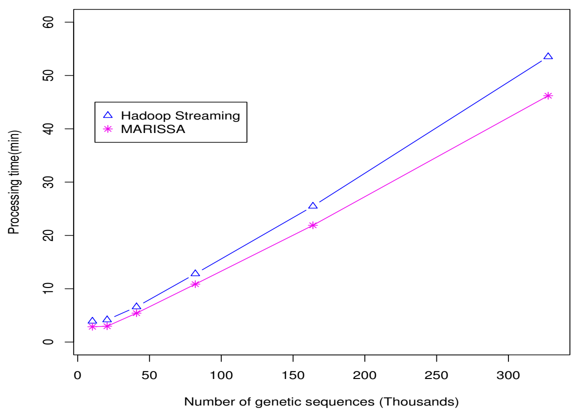

Fig. 7. 160-core Hadoop streaming and MARISSA clusters, each performing gene sequencing and similarity comparisons using BLAST. Each framework processes from 10,240 to 327,680 genetic sequences, using 80 nodes.

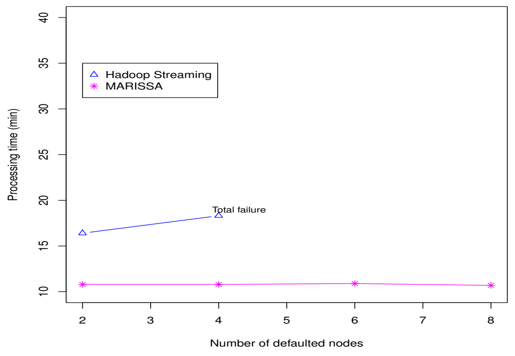

Fig. 9. This graph shows how Hadoop streaming and MARISSA behave faced with defaulting or dying nodes while running BLAST, processing 10240 genetic sequences. Both clusters start with 20 nodes, and progressively, in separate runs lose 2, 4, and 6 nodes. The Hadoop streaming framework however experiences a total failure beyond a 4 node loss. This occurs as worker nodes also hold vital data for processing.

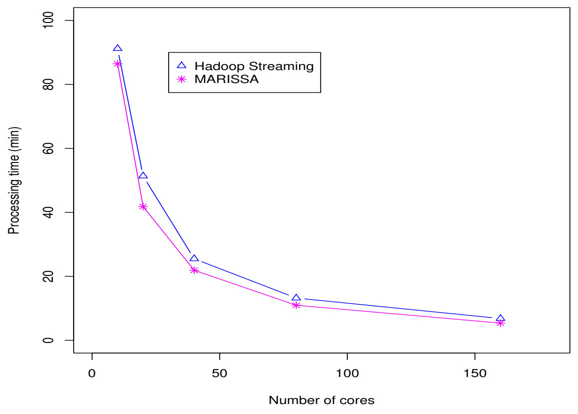

Fig. 8. MARISSA and Hadoop streaming cluster running BLAST, performing similarity comparisons on 40,960 genetic sequences. The number of cores used here varies from 10 to 160 cores.

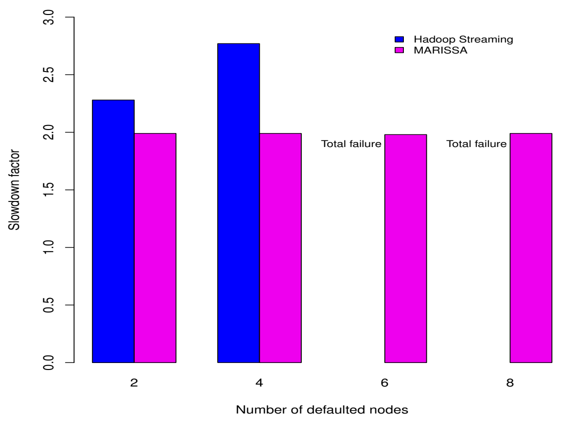

Fig. 10. This graph shows how Hadoop streaming and MARISSA slowdown given node losses while running BLAST. Both frameworks start with 20 nodes and process 10240 genetic sequences. Where comparable, Hadoop performs up to three times as slow, while MARISSA only slows down by a factor of 2.

and subsequently provides up to 26.6% better performance for 10,240 sequences. BLAST shows a lower performance disparity over K-means between both frameworks, because BLAST is more data-intensive than CPU-intensive. As we have shown in [18] our MapReduce processing model fairs better in CPU-intensive contexts.

In **Figure** 8, In contrast to K-means, BLAST is more memory intensive than CPU-intensive. The addition of computing nodes, and the ability for MARISSA to leverage its unbridled processing model therefore only minimally shows. MARISSA runs up to 5.2% faster than Hadoop streaming with 10 cores, and 21% faster with 160 cores. In **Figure** 9, both frameworks gradually experience induced node failures. Hadoop streaming is seen here experiencing an unrecoverable failure beyond an eight node loss. This is because `Datanodes` with Hadoop also hold the role of `TaskTrackers`. With MARISSA, as the input is decoupled from the processing nodes, the gradual loss of nodes only slightly affects the processing runtime of the underlying application. As data does not require being shifted from failing node to rescuing node, fault-tolerance in MARISSA's case incurs little overhead if any. Data availability is the responsibility of the filesystem, and not MARISSA's. Rescuing nodes are simply instructed by the `master` to process the data left by the failed nodes. This simply involves pointer passing which is very inexpensive, explaining the good performance exhibited.

In **Figure** 10, we quantify both frameworks' slowdown factors with respect to node losses. Over 6 and 8 node losses, Hadoop experiences a total job failure, leaving us unable to draw a comparison beyond 4 lost nodes.

IV. Related Work

While a fair body of work exists with regards to MapReduce implementations, the questions of multi-language support and application streaming has not been widely addressed. Here we summarize work related to our research into two categories:

MapReduce comparisons: Our previous work, MARIANE [17], is a traditional MapReduce framework, whose architecture presents the basis for MARISSA. Twister [15] is an iterative MapReduce framework. Just like Hadoop, Twister only supports user code written in Java. LEMO-MR [18] is a lightweight, low-overhead MapReduce framework. LEMO-MR also can only support Java applications. Amazon has produced EMR [10], a cloud computing framework allowing for MapReduce programs to be implemented. In a similar fashion, Microsoft has produced Azure [4]. While EMR borrows from Hadoop's model, Azure is limited by applications developed inside its own environment, precluding legacy and pre-existing scientific applications. Similarly, Twister4Azure [21] affords the Azure framework the ability to apply iterative MapReduce applications. MARS [23] is a MapReduce framework on graphics processors. MARS does not feature streaming nor multi-language support, as it offers only C/C++ APIs. Phoenix [35] is a shared-memory implementation of the MapReduce model. Similar to MARS, Phoenix features APIs solely for C/C++ applications.

MapReduce applications in Science: CGL-MapReduce is a set of MapReduce APIs for use with Hadoop MapReduce in processing binary data formats in use with High Energy Physics [16]. While CGL-MapReduce work focuses on MapReduce for science, the APIs offered only focus on High Energy Physics, and do not cover other scientific applications. Similarly, Twister-BLAST [7] allows for the application of BLAST on the Twister platform. Nguyen et.al. [28] expose the use of MapReduce to climate satellite applications, and much like CGL-MapReduce, only focus on a single application suited for a given scientific discipline. AzureBlast [26] applies the bioinformatics tool BLAST to the Azure cloud and evaluates the advantages and trade-offs of such an approach. Buck et. al [12] introduce a Hadoop plug-in allowing scientists to specify logical queries over array-based NetCDF data models. Mackey et. al [27], [36], highlight how the MapReduce model can be useful to a myriad of disciplines, ranging from bioinformatics to astrophysics, to cyber-security and discuss extensions such as arbitrary input format support required for scientific applications. While Lu et. al [26] address challenges in their quest to develop scientific applications on the cloud, the authors solely focus on Microsoft Azure [4] in their study.

V. Conclusion

According to CERN [2], every second of the Large Hadron Collider's operation produces more than 40 million data elements. Similar trends are being observed in other scientific facilities. The utilization of data-intensive frameworks capable of managing and processing such large-scale data is increasingly becoming a critical need to the scientific community. While MapReduce presents an adequate model for this challenge, its current implementation, in the form of Apache Hadoop, does not yet fully allow scientists to make use of the model for their purposes. In this paper, we first discussed the problems and shortcomings of existing MapReduce implementations as they apply to scientific applications, and subsequently addressed these needs in MARISSA (MapReduce Implementation for Streaming Scientific Application). Specifically, MARISSA allows for:

- Iterative application support: The ability for an application to assess its output and schedule further executions.
- The ability to run a different executable on each node of the cluster.
- The ability to run different input datasets on different nodes.
- The ability for all or a subset of the nodes to run duplicates of the same task, allowing the reduce step to decide which result to select.

VI. Acknowledgements

This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Additional support was provided by NSF grants OCI-0943705 and CNS 0958501.

References

[1] Apache Hadoop. http://hadoop.apache.org.

[2] European organization for nuclear research. http://public.web.cern.ch/public/en/lhc/lhc-en.html.

[3] Lustre File System.

[4] Microsoft Research.

[5] National Energy Research Scientific Computing Center. http://www.nersc.gov.

[6] Pervasive Technology Institute, Indiana University. http://www.iterativemapreduce.org/.

[7] SALSA Group, Indiana University .

[8] TeraGrid Information Services.

[9] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Kalasky, K.-L. Ma, V. Pascucci, J. Ahrens, E. W. Bethel, H. Childs, J. Huang, K. Joy, Q. Koziol, G. Lofstead, J. Meredith, K. Moreland, G. Ostrouchov, M. Papka, V. Vishwanath, M. Wold, N. Wright, and K. Wu. Scientific discovery at the exascale: Report from the doe ascr 2011 workshop on exascale data management, analysis, and visualization. http://www.olcf.ornl.gov/wp-content/uploads/2011/01/Exascale-ASCR-Analysis.pdf, 2011.

[10] AMAZON. Amazon Elastic MapReduce.

[11] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[12] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. A. Brandt. Scihadoop: array-based query processing in hadoop. In *SC*, page 66, 2011.

[13] S. Canon, S. Cholia, J. Shalf, K. Jackson, L. Ramakrishnan, and V. Markowitz. A performance comparison of massively parallel sequence matching computations on cloud computing platforms and hpc clusters using hadoop. In *Using Clouds for Parallel Computations in Systems Biology Workshop, Held at SC09*, 2009.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[15] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.

[16] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *IEEE Fourth International Conference on eScience*, pages 277–284, December 2008.

[17] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. Mariane: Mapreduce implementation adapted for hpc environments. *Grid Computing, IEEE/ACM International Workshop on*, 0:82–89, 2011.

[18] Z. Fadika and M. Govindaraju. Lemo-mr: Low overhead and elastic mapreduce implementation optimized for memory and cpu-intensive applications. *Cloud Computing Technology and Science, IEEE International Conference on*, 0:1–8, 2010.

[19] Z. Fadika, M. Govindaraju, S. Canon, and L. Ramakrishnan. Evaluting hadoop for data-intensive scientific operations. *IEEE Cloud Computing*, 2012.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[21] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu. Portable parallel programming on cloud and hpc: Scientific applications of twister4azure. In *UCC*, pages 97–104, 2011.

[22] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, P. Sarkar, M. Seaman, and D. Subhraveti. Gpfs-snc: An enterprise storage framework for virtual-machine clouds. *IBM Journal of Research and Development*, 55(6):2:1 –2:10, nov.-dec. 2011.

[23] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[24] I. Korf, M. Yandell, and J. Bedell. *BLAST*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[25] Lavanya Ramakrishnan, Piotr T Zbiegiel, Scott Campbell, Rick Bradshaw, Richard Shane Canon, Susan Coghlan, Iwona Sakrejda, Narayan Desai, Tina Declerck, Anping Liu. Magellan: Experiences from a Science Cloud. In *ScienceCloud 2011: 2nd Workshop on Scientific Cloud Computing*, San Jose, California, USA, 2011.

[26] W. Lu, J. Jackson, and R. Barga. Azureblast: a case study of developing science applications on the cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 413–420, New York, NY, USA, 2010. ACM.

[27] G. Mackey, S. Sehrish, J. Lopez, J. Bent, S. Habib, and J. Wang. Introducing Map-Reduce to High End Computing. In *Petascale Data Storage Workshop at SC08*, Austin, Texas, Nov. 2008.

[28] P. Nguyen and M. Halem. A mapreduce workflow system for architecting scientific data intensive applications. In *Proceeding of the 2nd international workshop on Software engineering for cloud computing*, SECLOUD '11, pages 57–63, New York, NY, USA, 2011. ACM.

[29] L. Ramakrishnan, K. R. Jackson, S. Canon, S. Cholia, and J. Shalf. Defining future platform requirements for e-science clouds. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 101–106, New York, NY, USA, 2010. ACM.

[30] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu. Magellan: experiences from a science cloud. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, pages 49–58, New York, NY, USA, 2011. ACM.

[31] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pages 231–244, 2002.

[32] S. W. S. Shimin Chen. Map-reduce meets wider varieties of applications. May 2008.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –10, May 2010.

[34] S. R. Soltis, G. M. Erickson, K. W. Preslan, M. T. O'keefe, and T. M. Ruwart. The global file system: A file system for shared disk storage, 1997.

[35] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[36] C. Zhang, H. De Sterck, A. Aboulnaga, H. Djambazian, and R. Sladek. Case study of scientific data processing on a cloud using hadoop. *High Performance Computing Systems and Applications*, 5976(2):400–415, 2010.