DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

MARIANE: MApReduce Implementation Adapted for HPC Environments

Zacharia Fadika, Elif Dede, Madhusudhan Govindaraju, Lavanya Ramakrishnan[‡]

Department of Computer Science, State University of New York (SUNY) at Binghamton
and [‡] Lawrence Berkeley National Laboratory
(zfadika1, edede1, mgovinda)@binghamton.edu, [‡] lramakrishnan@lbl.gov

***Abstract*—MapReduce is increasingly becoming a popular framework, and a potent programming model. The most popular open source implementation of MapReduce, Hadoop, is based on the Hadoop Distributed File System (HDFS). However, as HDFS is not POSIX compliant, it cannot be fully leveraged by applications running on a majority of existing HPC environments such as Teragrid and NERSC. These HPC environments typically support globally shared file systems such as NFS and GPFS. On such resourceful HPC infrastructures, the use of Hadoop not only creates compatibility issues, but also affects overall performance due to the added overhead of the HDFS. This paper not only presents a MapReduce implementation directly suitable for HPC environments, but also exposes the design choices for better performance gains in those settings. By leveraging inherent distributed file systems' functions, and abstracting them away from its MapReduce framework, MARIANE (MApReduce Implementation Adapted for HPC Environments) not only allows for the use of the model in an expanding number of HPC environments, but also allows for better performance in such settings. This paper shows the applicability and high performance of the MapReduce paradigm through MARIANE, an implementation designed for clustered and shared-disk file systems and as such not dedicated to a specific MapReduce solution. The paper identifies the components and trade-offs necessary for this model, and quantifies the performance gains exhibited by our approach in distributed environments over Apache Hadoop in a data intensive setting, on the Magellan testbed at the National Energy Research Scientific Computing Center (NERSC).**

I. Introduction

MapReduce is an increasingly popular framework and programming model. Introduced in 2004 at the USENIX Symposium on Operating Systems Design and Implementation (OSDI) [1], the model is inspired from the functional programming construct "map()". As such, MapReduce consists of the application of uniform functions *"map"* and *"reduce"* to a set of input elements sub-divided into multiple chunks to be processed by machines, part of a distributed computing system. The appeal of the model stems from the fact that it absolves the programmer from the burden of input management, parallelism, and synchronization constraints. MapReduce programs are written as single node programs by the user, and are subsequently parallelized by the framework. The details of input distribution, synchronization of necessary data structures, as well as handling machine failures, are all abstracted away from the user, and hidden in the paradigm itself. The MapReduce model in its most popular form, Hadoop [2], uses the Hadoop Distributed File System (HDFS) [3] to serve as the Input/Output manager and fault-tolerance support system for the framework. The use of Hadoop, and of the HDFS is however not directly compatible with HPC environments such as NERSC [4], the NY state Grid [5], the Open Science Grid [6], and TeraGrid [7]. This is so because Hadoop implicitly assumes dedicated resources with non-shared disks attached. At NERSC's Magellan cluster, the system administrator has isolated a part of the large cluster for use with Hadoop. This isolation not only limits the resources available to MapReduce programs, but also produces performance penalties with MapReduce programs, as HDFS on top of an underlying filesystem introduces performance hampering layers of indirection. In contrast, applications using the rest of the cluster interact directly with GPFS. Furthermore, even though Hadoop has successfully worked at large scale for a myriad of applications , it is not suitable for scientific (legacy) applications that rely on a POSIX compliant file systems in the grid/cloud setting. The HDFS is not POSIX compliant. In this paper, we investigate the use of Global Parallel File System (GPFS) [8], Network File System (NFS) [9] for large scale data support in a MapReduce context.

For this purpose we picked three application groups, two of them, UrlRank and "Distributed Grep", from the Hadoop repository, and a third of our own: XML parsing of arrays of double, tested here under induced node failures, for fault-tolerance testing purposes. The first two being provided with the Hadoop application package, have been shown to provide scalable performance with Hadoop [10]. Even as we limit our evaluation to NFS and GPFS, our proposed design is compatible with a wide set of parallel and shared-block file systems, such as LUSTRE [11], pNFS [9], GFS2 [8], and Oracle Cluster FS [12]. We present the diverse design implications for a successful MapReduce framework in HPC contexts, and show the performance data collected from the evaluation of this approach to MapReduce along side Apache Hadoop at the National Energy Research Scientific Computing Center (NERSC) Magellan [4] cluster.

The contributions of this paper are the following:

- We design and implement a MapReduce framework, MARIANE, capable of making use of various cluster, shared-disk, POSIX, and parallel file systems.
- We show the possibility through MARIANE of the MapReduce model's applicability to a wider array of HPC

environments with different distributed file systems.

- We show the ability for MapReduce to make full use of computational resources in HPC environments, wherein computational jobs on all nodes do not have local disk access.
- We show that high performance can be achieved by offloading MapReduce file system operations to a shared file system, in so doing, leaving the MapReduce framework to only *"map"* and *"reduce"*.

II. The Architecture of a MapReduce Platform

Hadoop uses the HDFS, inspired from the GFS [13] for various background tasks such as input management, distribution, locality, output collection, performance but also fault-tolerance. As a data manager, the HDFS is tasked with dividing the input among participating nodes in the cluster, keeping a myriad of accounting tallies, including chunk size, location, and duplication counts. The HDFS insures input distribution and rally in providing the user with an interface whose role is to provide bits of given data files to cluster nodes. Among its chief advantages, the Hadoop Distributed File System provides input locality by enabling nodes hosting input shards to apply their processing on such chunks, rather than on remotely stored data. This design provides significant performance benefits as the computation is brought to the data, rather than the data to the computation [14]. In line with its data management role, the HDFS collects output data processed by nodes, and *"shuffles"* them for the reducer(s) to operate on them. Following reducing, the file system then assembles data in a coherent form and makes it available for user view, or for subsequent MapReduce processing. In one of its most central roles, the HDFS allows for the Hadoop MapReduce fault-tolerance mechanism to thrive. The HDFS in that function keeps integrity reports, block condition reports and triggers the replication of faulty or missing blocks due to node failure.

III. MapReduce for Shared-Disk file systems

As highlighted above, a successful MapReduce implementation requires parallelization, synchronization abstraction from the user, as well as fault-tolerance and an effective data management scheme. These features must however be hidden from the user's responsibility, while embodying the fabric of the framework. With today's use of MapReduce for petabyte and exabyte size data, other important factors such as scalability stem from the correct application of the tenets outlined above. With Apache Hadoop, all but synchronization abstraction rests outside of the HDFS's authority. The Hadoop Distributed File System is however not a dedicated choice within many HPC environments. In such settings including, but not limited to NERSC [4], the NY state Grid [5], the Open Science Grid [6], and TeraGrid [7], where the Message Passing Interface (MPI) [15] is widely adopted, shared-disk file systems offering the visibility of a storage solution to all nodes are favored. The use of the HDFS and by extension of MapReduce not only through Hadoop, but Twister [16], Dryad [17] need either the isolation of limited resources to that end, or the revamping of the cluster solely to MapReduce's end. For example, at NERSC the system administrators have set aside for Hadoop, 400 nodes out of 17000. Access via *ssh* and queuing policies for these nodes are different than the rest of the cluster. The latter case being impractical, wherever applicable in such settings, MapReduce users find themselves not able to make full use of existing resources. More often, this means for cluster administrators, the installation of the model on top of an already dedicated system. This approach requires traversing additional layers of indirections to the system as the HDFS must be accomodated on the host cluster. Such additional layers of indirections have been shown as performance degrading by [18], in which Hadoop's performance is shown as poor in virtualized settings, precisely because of the many layers of abstractions sitting between the native systems and the Hadoop Distributed File System. In the case of Twister and Dryad, the issue is not of HDFS' use, as such frameworks do not make use of it, but rather, the use of a similar approach requiring that each node be granted independent storage from all other nodes in the cluster. As has been our experience, such a request as in Hadoop's case leads either to resource isolation, or to resource limitation. This is because, MPI friendly clusters, such as TeraGrid, and NERSC, offer immense computing power, but also favor directly networked and POSIX file systems, as such a scheme is suitable for their operations.

IV. The case for MARIANE

MapReduce itself constitutes a model rather than an implementation. This model aiming at distributed processing of large datasets is compatible in its tenets rather than its implementation to most widely available HPC platforms. The three pillars of the model (Data management, synchronization abstraction, and fault-tolerance) can be found in the majority of HPC platforms in use today. Where not present, the latter two can be engineered. Nonetheless, the tight coupling of the MapReduce model to its implementation in various cases outlined, introduces inefficiencies not suitable for traditional, batch and legacy systems. These inefficiencies, we believe, should not handicap the model from evolving, nor should it restrict applications from using the full force of its capacity in HPC environments. To this effect, we designed the implementation of MapReduce on such systems for adaptability reasons first, and subsequently analyzed the performance aspects presented by the approach. In this work we focus on NFS and GPFS [9], [19]. Even though we limit ourselves here to NFS and GPFS, due to a common high level structure, and disk presentation, MARIANE can be installed and evaluated on a myriad of shared-disk and clustered file systems. We will focus in future work to benchmark a sizable number of such file systems with MARIANE in a performance evaluation, as this paper's scope is focused on quantifying the HDFS bottleneck along with its performance implications of MARIANE's design choices.

A. The NERSC case

The National Energy Research Scientific Computing Center (NERSC) [4] hosts over 7 central clusters, as well as a myriad of specialized ”sub-clusters” hosting various energy research projects. NERSC totals approximately 17,000 available nodes setup for MPI use, used for research purposes, and within which, sit over 200,000 processing cores. NERSC also offers over 2000 Petabytes of storage space for compute and data intensive applications. Apache Hadoop is installed on its Magellan cluster and benefits from 400 processing cores and 785 TB of data space. This stems from Hadoop's requirement to operate under the HDFS and similarly structured storage systems such as S3 with Amazon MapReduce [20]. Hadoop and the HDFS however, similarly to other MapReduce implementations such as Twister and LEMO-MR [21], when operating under global filesystems as in NERSC's case, require dedicated disks, or at least, dedicated disk partitions. Each node also needs dedicated ports and active MapReduce daemons running on them. In Hadoop's case, the HDFS needs to be large enough to accommodate minimum file replication requirements for an effective fault-tolerance mechanism. This by default means 3 times the input size for each file present on the file system, and as such, 3TB of space for a 1TB set of input files. These various conditions can cause the MapReduce cluster to be spared minimal resources, as overall resources can not be solely dedicated to MapReduce. In a similar environment, MARIANE can make use of the existing global filesystem as it is configured. The framework only uses the secure shell, and does not require any dedicated ports and daemons, nor does it require file replication space for its input. In its operation, MARIANE can afford constant node deletion and addition to its host file without requiring the cluster to be reconfigured. The framework does not need to be started or stopped, and only requires installation on the ”Master” node, rather than on all participating computers as it is the case with Hadoop and Twister. The MARIANE framework is thus designed to be seamlessly setup on HPC environments with no disruption to the environment's integrity or structure.

B. Design considerations in MARIANE

The design of MARIANE rests upon of three principal modules, representing the tenets of the MapReduce model. Input/Output management and distribution rests within the `Splitter`. Concurrency management in the role of `TaskController`, while fault-tolerance dwells with the `FaultTracker`.

Figure 1 shows the design used for MARIANE

C. Input Management

1) Input Splitting: While Hadoop and most MapReduce applications apportion the input amongst participating nodes, then transfer each chunks to their destinations, MARIANE relies on the inherent shared file system it sits on top for this feat. The framework leverages the data visibility offered by shared-disk file systems to cluster nodes. This feature

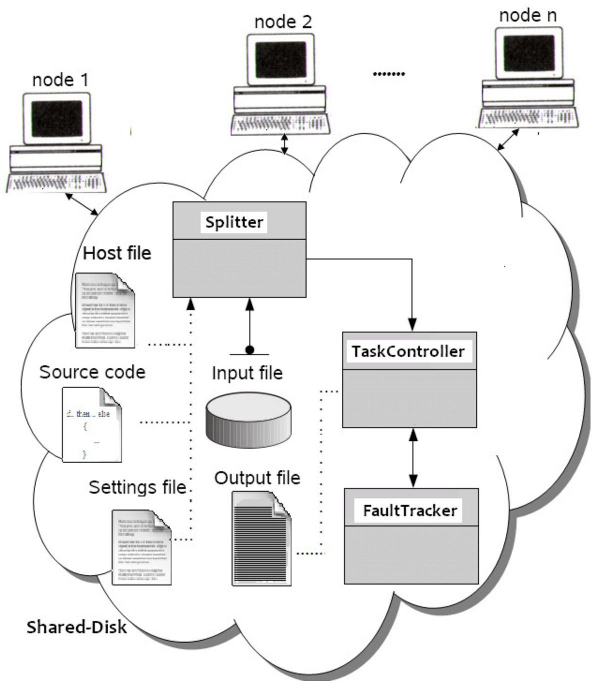

Fig. 1. Architecture used for MARIANE

exonerates MARIANE from operating data transfers, as such a task is built-in and optimized within the underlying distributed file system. Input management and split distribution are thus not performed on top of an existing file system (FS), but rather with the complicity of the latter. This absolves the application from the responsibility of low-level file management and with it, from the overhead of efficiently communicating with the FS through additional system layers. Furthermore, MARIANE in doing so, benefits not only from file system and data transfer optimizations provided by evolving shared-disk file system technology, but can solely focus on *”mapping”* and *”reducing”*, rather than data management at a lower level.

D. Input Distribution

Input distribution is directly operated through the shared file system. As the input is deposited by the user, the FS is optimized to perform caching and pre-fetching to make the data visible to all nodes on-demand. This frees the MapReduce framework from accounting, and transferring input to the diverse nodes. Another benefit of shared-disk file systems with MapReduce, one which became apparent as the application was implemented is the following: current MapReduce implementations, because of their tightly coupled input storage model to their framework require cluster re-configuration upon cluster size increase and decrease. This does not allow for an elastic cluster approach such as displayed by the Amazon EC2 cloud computing framework [20], or Microsoft's Azure [22]. Although cloud computing is conceptually separate from MapReduce, we believe that the adoption of some of its features, more specifically ”elasticity”, can positively benefit the turn around time of MapReduce applications. With the input medium isolated from the processing nodes, as MARIANE features, more nodes can be instantaneously added onto the cluster, without incurring the cost of data redistribution, or cluster re-balancing. Such operations can be highly time consuming, and only allow for the job to be started at their completion, when all data settles on the cluster nodes involved [2]. With MARIANE, input storage is independent from the diverse processing nodes. Separating the I/O structure from the nodes allows for a swift reconfiguration and a faster application turnaround time. In Hadoop's case, removing a node holding a crucial input chunk means finding a node holding a duplicate of the chunk held by the exiting node and copying it to the arriving node, or just re-balancing the cluster, as to redistribute the data evenly across all nodes. Such an operation with large scale input datasets can be time consuming. Instead, according to the number of participating workers in the cluster, nodes can be assigned file markers as to what part of the input to process. Should a node drop or be replaced, the arriving machine simply inherits its file markers, in the form of simple programming variables. This mechanism, as our performance evaluation will show, also makes for an efficient and light-weight fault-tolerant framework.

E. Task tracker and task control

The task tracker, also known as "master" makes the *"map"* and *"reduce"* code written by the user, available to all participating nodes through the shared file system. This results on the application level to a one time instruction dispatch, rather than *"map"* and *"reduce"* instructions streaming to as many participating nodes as there are in the cluster. Upon launch, the nodes designated as mappers also subsequently use the *"map"* function, while those designated as reducers, use the *"reduce"* function. The task tracker 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's data structure. Upon failure, the completion list is communicated to the `FaultTracker`. Slow nodes are similarly accounted for, and their work is re-assigned to completed and available machines. In a redundant situation caused by two nodes running similar jobs, in the case perhaps 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.

F. Fault-tolerance

While Hadoop uses task and input chunk replication to fault-tolerance ends, we opted for a node specific fault-tolerance mechanism, rather than an input specific one. With this approach, node failure does not impact data availability, and new nodes can be assigned failed work with no need for expensive data relocation. Upon failures, we elected for an exception handler to notify the master before terminating, or in the case of sudden death of one of the workers, the rupture of a communication pipe. Furthermore, the master receives the completion status of the *"map"* and *"reduce"* functions from all its workers. Should a worker fail, the master receives notification of the event through a return code, or a broken pipe signal upon sudden death of the worker. The master then updates its node availability and job completion data structures to indicate that a job was not completed, and that a node has failed. We later evaluate this low overhead fault-tolerant component along with Hadoop's data replication and node heartbeat detection capability, and assess job completion times in the face of node failures.

1) FaultTracker: The Fault-Tracker consults the task completion table provided by the master node, and reassigns failed tasks to completed nodes. Unlike Hadoop, the reassignment does not include locating relevant input chunks to the task and copying them, if those chunks are not local to the rescuing node. The task reassignment procedure rather provides file markers to the rescuing node so it can process from the input, the section assigned to the failed node. As the input is visible to all nodes, this is done without the need to transfer huge data amounts, should massive failures occur. The `FaultTracker`'s operation is recursive; should a rescuing node fail, the rescuer is itself added to the completion table and the module runs until all tasks are completed, or until all existing nodes die or fail. In the interim, dead nodes are pinged as a way to account for their possible return and inscribe them as available again. Hadoop uses task replication regardless of failure occurrence. It also does not keep track of nodes that might have resurfaced after suddenly failing. MARIANE for its part only uses task replication in the case of slow performing nodes, when the sloth is detected, and not before. To this effect, whichever version of the replicated sub-job completes first is accepted.

V. Distributed Large-Scale Data Processing

In this section, we test the applicability of MARIANE to traditional MapReduce problems along side Hadoop MapReduce. We not only test the framework in various node addition schemes with constant input sizes to account for cluster increase scalability and speed-up, but also test the cluster for increasing input sizes faced with static cluster sizes. Our experiments were conducted on the National Energy Research Scientific Computing Center's cluster (NERSC), and in the Binghamton University Grid and Cloud Computing Research Lab. On NERSC, we performed our tests on the Magellan cluster where MARIANE was installed on top of GPFS, and tested along side the local Apache Hadoop v.20 installation running on the same test bed. The Binghamton University Grid and Cloud Computing Research Lab benefits from the same Hadoop version, and hosts MARIANE using NFS. In all the experiments showcased, we ran MARIANE along side Hadoop using identical nodes, identical node counts, identical input data and similar user source code.

We chose for these experiments the application of traditional MapReduce problems, available from Apache Hadoop's example repository. One being UrlRank, a popular application similar to Google pagerank [23], and used at Yahoo! with Hadoop MapReduce to categorize web pages by request frequency. Similar versions of this application are also used for data mining and user statistical analysis at Facebook [24]. The second application we tested was ”Distributed Grep”, where typically, documents containing word or sequences patterns are searched and returned from Terabytes of raw data, or potentially hundreds of millions of individual documents, making processing on a single computer, an unacceptable option. Variations to this application include pattern appearance frequencies and count. As a final application group, we tested a compute intensive application, one of our own: distributed parsing and processing of XML elements, using the AxisJava parser [25]. AxisJava is a web services-based toolkit consisting of a SOAP engine capable of creating SOAP processors allowing data content parsing. Resulting tokens from this parsing can then subsequently be streamed into application-friendly content and returned to the user. Our input data for testing purposes is composed of arrays of floating point numbers. In [26], [27], we have shown that AxisJava scales poorly for applications that require processing arrays of floating point numbers. The computation is intensive and demanding on the systems involved, regardless of overall data size. With our XML data parsing experiments, we test the performance of both frameworks' fault-tolerance mechanisms in the face of node failures.

VI. Performance Results

We run our tests on a selection of two clusters:

NERSC Magellan cluster

- 1× 8 core – Intel Nehalem machines, with 2.6Ghz and 3 GB of ECC RAM, running Linux The file system in use here is GPFS. Results on this class of machines are taken by averaging the timings produced on these nodes.

Grid and Cloud Computing Research Lab Cluster at Binghamton University

- 1× dual core – One desktop-class machine, which has a single 2.4Ghz Intel Core 6600 with 2 GB of ECC RAM, and quad cores running Linux 2.6.24. The file system in use here is NFS v.4.
- 2× quad core – 1U nodes in a cluster, each of which has two 3.2Ghz Intel Xeon CPUs, 4 gigabytes of RAM 8 cores, and run a 64 bit version of Linux 2.6.15. Results on this class of machines are taken by averaging the timings produced on these nodes. The file system in use in the test directory is NFS v.4

Experiments 1, 2, 3 were run on NERSC, under GPFS, whereas experiment 4, 5 and 6 were run on at Binghamton University under NFS.

Figure 2 compares Hadoop and MARIANE in a data intensive scenario. This experiment being more data intensive than compute intensive, MARIANE can comfortably rely on the shared-disk file system for data management, while the

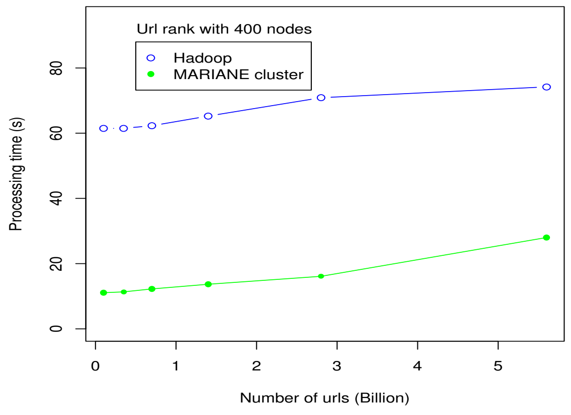

Fig. 2. 400-node MARIANE and Hadoop clusters, each ranking over 5 billion given urls on the National Energy Research Scientific Computing Center's (NERSC) cluster. Both MapReduce frameworks gradually process diverse increasing input loads ranging from 0.1 to 5.6 billion urls.

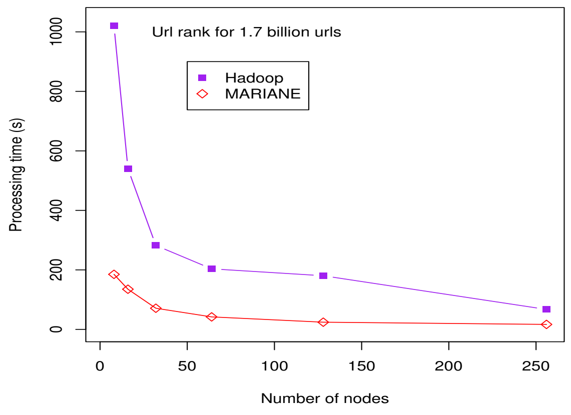

Fig. 3. MARIANE and Hadoop clusters processing each 1.7 billion urls with diverse cluster sizes ranging from an 8 node cluster to a 256 node cluster. This experiment, like the previous one, was run on the National Energy Research Scientific Computing Center's (NERSC) cluster. Hadoop is considerably slower for small clusters as its data management operation needs more nodes to replicate and disseminate data, in an effort to make it local to each of its nodes. MARIANE automatically benefits from this feature even with smaller cluster sizes, as the shared-disk file system makes its input local and visible to the entire cluster. Despite the convergence of curves towards 256 nodes, it is worth pointing out that with 256 nodes, Hadoop runs the 1.7 billion urls in 67.09 seconds while MARIANE runs the same load in 16.723 seconds, thus 4 x faster.

platform solely focuses on ”mapping” and ”reducing”. Hadoop on its end, working in tight concert with the HDFS for fault-tolerance and data integrity purposes, needs to traverse the HDFS to access its input data. What's more, Hadoop needs to operate and maintain the HDFS as it performs *”map”* and *”reduce”* operations on all its nodes. Such tasks include among others, chunk integrity checks, datanode heartbeat checks, and chunk replication count threshold verifications. As MapReduce processes huge datasets, the result of this experiment is also heavily dependent on the data transfer ability exhibited by both technologies. In our tests, we measured GPFS's transfer speed at the time of the experiments, to be ranging around 0.83GB/s, while HDFS showed 0.12GB/s.

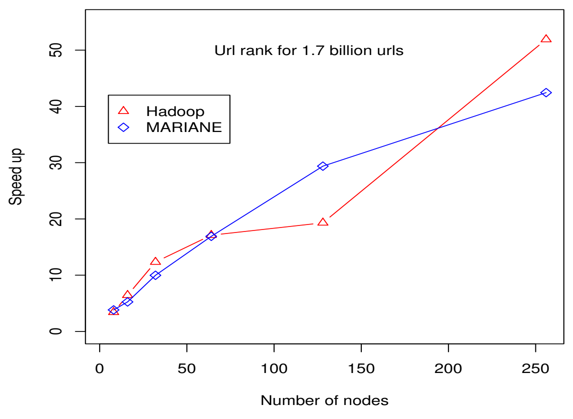

Fig. 4. Speed-up computed from MARIANE and Hadoop clusters processing 1.7 billion urls with diverse cluster sizes ranging from 8 nodes to 256 nodes. Speed-up is computed as $\frac{T_1}{T_p}$ and represents how fast each cluster performs relative to its previous sized versions. Both clusters roughly scale in a similar fashion. Hadoop however scales slightly better size-wise vis-a-vis itself as it shows slower performance in earlier runs.

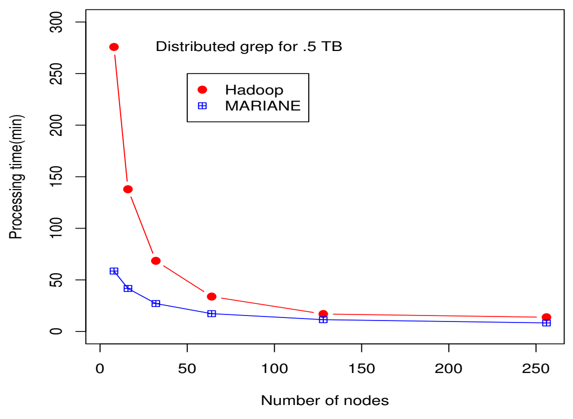

Fig. 5. This figure shows the operation of "Distributed Grep" on 0.5TB of data by both frameworks, with diverse cluster sizes ranging from an 8 node cluster to a 256 node cluster for both Hadoop and MARIANE. MARIANE here benefits from the support of NFS, as this experiment was run on the cluster at the Binghamton University Grid and Cloud Computing Research Lab.

Figure 3 shows a performance trend similar to that shown in **Figure 2**. In this case however, the cluster size is gradually increased from 8 nodes to 256 nodes, while the input sits at a cumulus of 1.7 billion urls processed per cluster run. The slower performance exhibited by Apache Hadoop for small clusters here stems from the need for intermediate results to more often be checked onto disk (HDFS), rather than held in memory; this is so, as the small number of nodes does not permit the latter. As a result, this circumstance causes repeated HDFS reads and writes in the midst of *"map"* and *"reduce"* operations, thus further negatively impacting performance. MARIANE also suffers from the inability to make use of faster memory storage offered by a wide array of nodes. In MARIANE's case however, the disk penalty is simply not as high, for the framework is devoid of file management responsibilities, and the file system in play appears more efficient.

In **Figure 4**, speed-up is computed as a measure of how both clusters scale relative to themselves. Even though both scale in a similar fashion vis-a-vis themselves, Hadoop speed-ups slightly better when more nodes are added to the cluster. When such is the case, HDFS transfers can diminish as data can be held in greater size in memory as a result of more nodes' presence in the cluster.

In **Figure 5**, both clusters process word pattern searches on 0.5TB of data with cluster size ranging from 8 nodes to 256 on the Binghamton University cluster. As a direct consequence of those cluster sizes, processing runtimes range from 5 hours with 8 nodes to 8 minutes with 256 nodes. Here, as with GPFS , performance trends are maintained, even as Hadoop's performance considerably improves in this context. This is perhaps due to the nature of the application run, because "Distributed Grep" relies heavily on memory operations, and can do without a constant need for disk (HDFS) access, especially disk writes. In such an environment, Hadoop emerges less penalized by HDFS housekeeping and operations during its runtime. Towards 256 nodes, despite the appearance of proximity of both plots in the Figure, MARIANE runs the input in 8 minutes, whereas Hadoop does so in 13. With NFS on our test bed, at the time of the experiments, we recorded transfer speeds of 0.08GB/s, while HDFS showed 0.04GB/s for internal HDFS transfers and 0.03GB/s for NFS to HDFS transfers. Note the closer NFS and HDFS rates, compared to GPFS, as perhaps a factor in Hadoop's closer performance numbers to MARIANE's.

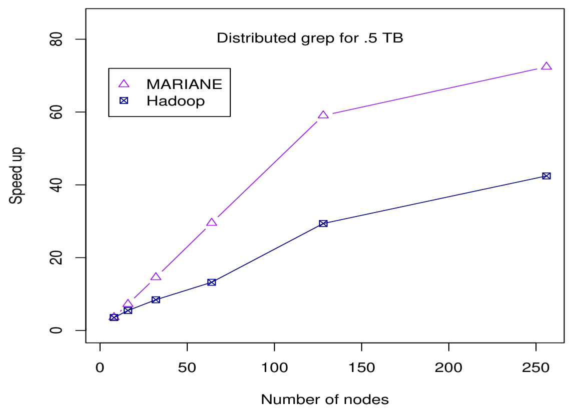

Fig. 6. Speed-up computed from MARIANE and Hadoop clusters processing 0.5TB of data with diverse cluster sizes ranging from 8 nodes to 256 nodes. Speed-up is computed as $\frac{T_1}{T_p}$ and represents how fast each cluster performs relative to itself. Hadoop scales vis-a-vis itself better as it starts slow.

In **Figure 6**, Hadoop shows faster speed-up than MARIANE. Starting slow, the framework gains faster speeds compared to its previous runs. MARIANE however keeps a constant increase throughout the experiment. The difference in speed-up stems from Hadoop's comfort with growing cluster sizes. In such settings, memory from the cumulus of participating nodes abounds, and as such HDFS access from intermediate data storage can be bypassed. This gives the framework a boost of performance as overhead prone

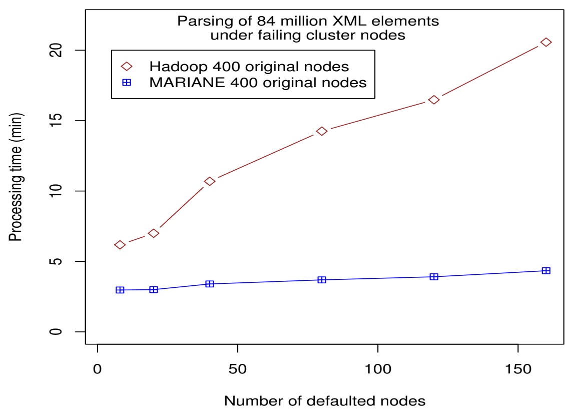

Fig. 7. Parsing of 84 million XML elements under failing conditions. Both MARIANE and Hadoop start with 400 nodes each. Gradually, we pull the power plug on 8, 20, 40, 80, 120 and 160 processing nodes, causing them to fail. As its input data is held by its nodes, upon failure, Hadoop is subjected to input data integrity checks, replication checks, data replication, relocation and cluster re-balancing. These operations in HDFS can be time consuming and as a result negatively impact performance. MARIANE, under failing nodes, instructs its `TaskController` to pass file markers from the dead nodes to the rescuing nodes for them to commence work. This involves simple argument passing, as data values are exchanged, and explains the performance observed under failing conditions.

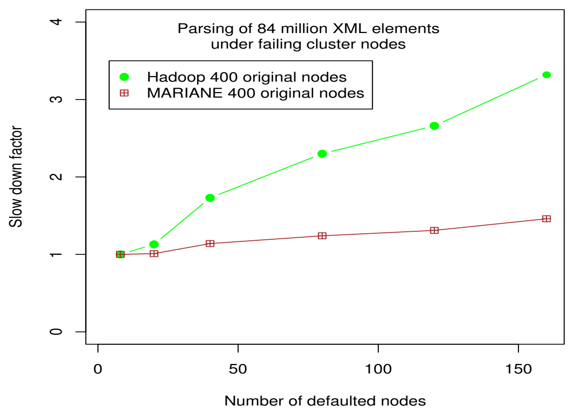

Fig. 8. Slow-down factor computed from MARIANE and Hadoop clusters parsing XML elements with the AxisJava engine, using diverse cluster sizes ranging from 8 nodes to 256 nodes. Slow-down, as speed-up is computed as $\frac{T_1}{T_p}$ and represents here how much slower vis-a-vis itself, each framework performs as more nodes fail. Hadoop's processing slows down more as the platform loses more nodes. The same is true for MARIANE, at a slower rate however.

operations are avoided. MARIANE, even as it is the subject of the same advantages and disadvantages, does not suffer as greatly from constant NFS access as Hadoop does from HDFS access. **Figure 7** shows in action MARIANE's fault tolerance mechanism outlined in the design section (IV-B). As nodes are decoupled from the input in MARIANE's design, their failure does not take with them input chunks necessary for the application's well-being. As such, the input does not need to be relocated or even replicated. Rescuing nodes in MARIANE's case are simply told where to find the input left by the failed nodes. Such input, visible to the entirety of the processing cluster, can be readily accessed by any rescuing node lending help to a failed node. The hand-off as such, consists of file marker passing, and as a result, occurs relatively fast, explaining the performance observed in Figure 7 under failing conditions. It should be noted that should the storage solution fail, the cluster will find itself inoperable. This however would entail the cluster as a whole, and not only MARIANE's operations. Similarly, should the HDFS itself or even the master node fail in Hadoop's case, the Hadoop cluster will find itself in an inoperable state as well. **Figure 8** shows how much slower vis-a-vis itself, each framework performs as more nodes fail. This graph is a different perspective on Figure 7. In Hadoop's case, we can observe a progressively faster slow-down trend as more nodes fail, due to cluster balancing, data replication and relocation features, happening within the framework. MARIANE, on the other hand, proceeds to file marker hand-offs to rescuing nodes, allowing them to swiftly act upon the occurring node failures.

VII. Related Work

In the High-Performance Distributed Computing community, many systems such as Sphere/Sector [28] have been designed and developed for distributed data, application management and processing. Sphere is however not a MapReduce application, and as such does not offer all the advantages embodied in the model and in MARIANE. Amazon has produced EC2 [20], a cloud computing framework allowing for MapReduce applications to be implemented. In a similar fashion, Microsoft has produced Azure [22]. EC2 and Azure are proprietary applications, and hence an insightful analysis on their design decisions is not possible. Twister [16] and DELMA [29] are MapReduce frameworks, one espousing an iterative approach, and the other an elastic approach to solving MapReduce problems. Both however require each of their nodes to benefit from individual storage units, as a shared storage option is not yet supported by both implementations. The same can be said for Microsoft Dryad [17]. These measures to storage management cast both Twister and Dryad, as Hadoop and DELMA, with the inability as of yet to flourish on traditional HPC frameworks.

VIII. Conclusions

With its applications in search engine technology, space monitoring and data mining technology, among others, MapReduce has slowly grown to be a successful and widely acclaimed data processing model. A testament to the model's effectiveness dwells in, to cite a few, among Yahoo!, Facebook, and Google's use of it, for large scale data processing. Before the application of MapReduce, Yahoo! took 26 days of processing to build automatic completion indexes for their search engine. After MapReduce, the same operation was reduced to 20 minutes with a cluster of computing nodes [30]. MapReduce has however in so far as it has been presented required dedicated disk-space for its applications. Founded on the

HDFS, in Hadoop's case, and in the HDFS's image in auxiliary implementations of the paradigm, MapReduce's adoption from our observations, shows impractical if not impossible on many batch, legacy and traditional HPC systems. Such systems as NERSC, TeraGrid, NYS Grid, Open Science Grid, to list a few do not harbor a file storage system in line with the Hadoop Distributed File System, but rather one which favors providing shared disk resources to all nodes. In this paper, we built upon such file systems, and presented MARIANE. A MapReduce implementation adapted and designed to work with existing and popular distributed file systems. With this framework, we eliminate the need for an additional file system and along with it, the involvement of MapReduce in expensive file system maintenance operations. By doing so, we show a significant increase in performance and decrease in application overhead along with a dramatic speed up in fault tolerance response as we compare our results to Apache Hadoop. Not only does MARIANE allow for traditional benefits offered by MapReduce such as ease of programming, synchronization and parallelization abstractions, the platform also allows for:

- High performance under working and failing conditions.
- The applicability and expansion of the MapReduce paradigm to a wider array of HPC environments espousing different distributed file systems.
- The evolution and adaptability of the MapReduce model, while still preserving the tenets that popularized it.

IX. Future Work

In future work, we plan to benchmark a wide array of shared-disk file systems with MARIANE.

X. Acknowledgements

This research used resources of the National Energy Research Scientific Computing Center, supported by the Office of Science of the U.S. Department of Energy, and was funded in part by the Advanced Scientific Computing Research (ASCR) in the DOE Office of Science under contract number DE-AC02-05CH11231.

The work presented here was also funded in part by NSF grant CNS 0958501.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] Apache Hadoop. [Online]. Available: http://hadoop.apache.org

[3] 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*, May 2010, pp. 1 –10.

[4] National Energy Research Scientific Computing Center. [Online]. Available: http://www.nersc.gov

[5] NYSGrid. [Online]. Available: www.nysgrid.org

[6] Open Science Grid. [Online]. Available: http://www.opensciencegrid.org.

[7] TeraGrid Information Services. [Online]. Available: http://info.teragrid.org/

[8] 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.

[9] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation or the sun network filesystem," 1985.

[10] 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.

[11] Lustre File System. [Online]. Available: http://wiki.lustre.org/index.php/Main_Page

[12] ORACLE Cluster File System. [Online]. Available: http://oss.oracle.com/projects/ocfs/

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/945445.945450

[14] G. Mackey, S. Sehrish, J. Lopez, J. Bent, S. Habib, and J. Wang, "Introducing mapreduce to high end computing in Petascale Data," in *Storage Workshop Held in conjunction with SC08.*

[15] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," 1994.

[16] Pervasive Technology Institute, Indiana University. [Online]. Available: http://www.iterativemapreduce.org/

[17] Microsoft Research. [Online]. Available: http://research.microsoft.com/en-us/projects/Dryad/

[18] S. Ibrahim, H. Jin, B. Cheng, H. Cao, S. Wu, and L. Qi, "Cloudlet: towards mapreduce implementation on virtual machines," in *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2009, pp. 65–66.

[19] 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*, 2002, pp. 231–244.

[20] Amazon, "Amazon Elastic Compute Cloud." [Online]. Available: http://aws.amazon.com/ec2

[21] 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*, vol. 0, pp. 1–8, 2010.

[22] Microsoft Research. [Online]. Available: http://www.microsoft.com/windowsazure/

[23] Google.com, "Google Web APIs," http://www.google.com/apis/.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Z. 0002, S. Anthony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *ICDE*, 2010, pp. 996–1005.

[25] AxisJava, "The Apache Project," 2002, http://ws.apache.org/axis/.

[26] M. R. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. J. Lewis, "A Benchmark Suite for SOAP-based Communication in Grid Web Services," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 19.

[27] W. Zhang and R. A. van Engelen, "A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services," in *ICWS '06: Proceedings of the IEEE International Conference on Web Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 197–204.

[28] Y. Gu and R. L. Grossman, "Sector and sphere: the design and implementation of a high-performance data cloud." *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, vol. 367, no. 1897, pp. 2429–2445, June 2009. [Online]. Available: http://dx.doi.org/10.1098/rsta.2009.0053

[29] Z. Fadika and M. Govindaraju, "Delma: Dynamically elastic mapreduce framework for cpu-intensive applications," in *CCGRID*, 2011, pp. 454–463.

[30] G. Orenstein, "Digging Deeper Into Data With Hadoop," *Available at http://gigaom.com/2009/06/07/digging-deeper-into-data-with-hadoop*, 2009.