LEMO-MR: Low overhead and Elastic MapReduce Implementation Optimized for Memory and CPU-Intensive Applications

Zacharia Fadika [1], Madhusudhan Govindaraju [2]

Computer Science Department, Binghamton University
P.O. Box 6000, Binghamton, NY 13902-6000, USA
[1] `zfadika@cs.binghamton.edu`
[2] `mgovinda@cs.binghamton.edu`

Abstract—**Since its inception, MapReduce has frequently been associated with Hadoop and large-scale datasets. Its deployment at Amazon in the cloud, and its applications at Yahoo! and Facebook for large-scale distributed document indexing and database building, among other tasks, have thrust MapReduce to the forefront of the data processing application domain. The applicability of the paradigm however extends far beyond its use with data intensive applications and diskbased systems, and can also be brought to bear in processing small but CPU intensive distributed applications. In this work, we focus both on the performance of processing large-scale hierarchical data in distributed scientific applications, as well as the processing of smaller but demanding input sizes primarily used in diskless, and memory resident I/O systems. In this paper, we present LEMO-MR (*L*ow overhead, *e*lastic, configurable for in-*m*emory applications, and on-*d*emand fault tolerance), an optimized implementation of MapReduce, for both on-disk and in-memory applications, describe its architecture and identify not only the necessary components of this model, but also trade offs and factors to be considered. We show the efficacy of our implementation in terms of potential speedup that can be achieved for representative data sets used by cloud applications. Finally, we quantify the performance gains exhibited by our MapReduce implementation over Apache Hadoop in a compute intensive environment.**

I. Introduction

Since its debut on the computing stage in 2004, and following its applications in various domains ranging from search engine technology [1] to astronomical data parsing [2], the MapReduce framework [3] has gained significance in the scientific computing community. The paradigm was originally inspired by a functional programming primitive called *map* [3]. In `Haskell` for instance, a programmer has the ability, given a list of input elements structured as a homogeneous dataset, to apply a given function on it. *map* takes a function and a list of elements, applies the function on each element of the list, then subsequently produces a second list resulting from the application of the function to the first list. The appeal of the method however lies in the fact that each element of the said input list can be assigned a different processing entity part of a cluster of computers, along with a copy of the *map* function to be applied on that element. This enables multiple processing entities to simultaneously apply the same *map* function on different bits of the same list, all representing a part of the overall input file, and as such reduce an application's turnaround time. MapReduce however carries its own burdens. Through previous experiments using Hadoop in the context of diverse applications, we uncovered latencies and delay conditions potentially inhibiting the expected performance of a parallel execution in CPU-intensive applications [4]. Furthermore, as it currently stands, MapReduce is favored for data-centric applications, and as such tends to be solely applied to disk-based applications. The paradigm, falls short in bringing its novelty to diskless systems dedicated to in-memory applications, and compute intensive programs processing much smaller data, but requiring intensive computations.

The contributions of this paper are the following:

- We describe an efficient and elastic MapReduce framework, which provides an abstraction of the input medium. It allows disk-based applications, as traditionally applied, diskless software systems, and compute intensive applications.
- We present a passive fault-tracking scheme, which avoids the burden of network packet abundance and possible network activity slow-down.
- We contrast latencies produced by the current MapReduce model with a direct approach to input data management, on-demand fault-tolerance and evaluate the performance, reliability, and scalability differences observed.
- We present an elastic approach to MapReduce, allowing for cluster configuration to be automatically scaled up and down in between tasks without the need for the framework to be torn down, rebuilt, and restarted.

II. The Architecture of a MapReduce Platform

Hadoop is an open-source framework developed by the Apache Software Foundation [5]. It is based on its own file system: The Hadoop Distributed File System (HDFS) [6]. Hadoop implements the Google MapReduce programming paradigm by relying on the HDFS as its foundation. The HDFS sits as a layer of abstraction between Hadoop and the Operating System. The file system administers input and output file considerations, and is also partially in charge of various framework activities such as load-balancing and fault-tolerance. Hadoop features a *splitter* module whose role is to partition and allocate file blocks or chunks to awaiting worker nodes. The blocks are then replicated according to a replication factor pertaining to Hadoop's internal settings and distributed among chosen nodes called `DataNodes` [7]. As cluster nodes fail or fall behind, `DataNodes` are given the ability to replicate their own file blocks.

`DataNodes` also periodically send updates and reports of block conditions to the master, such as block usage and integrity. Even though it is costly, especially for smaller data and heavy computation loads, this organization is necessary for fault-tolerance, because it is less expensive to bring the computation to the data rather than bringing the data to the computation [8]. While this provides a sense of locality, the expense of shipping data around is still incurred. Also, when this organization is not offset by a data size to computation ratio weighted heavily towards data size, the model reveals performance deficiencies that readily grow with compute intensity. As the data size increases in such scenarios, the input file blocks need to be rearranged to accommodate potential new `DataNodes`, further introducing delays and exacerbating the problem. Hadoop also uses "Shuffling" for data communication among nodes. "Shuffling" is a costly process of distributing the intermediary lists pertaining to each node among all the other nodes. We contrast these latencies with a direct approach to input data management and evaluate the performance, reliability and scalability differences observed.

A. Design Space for MapReduce Implementation

Hadoop tightly couples its input processing module to its `map` and `reduce` constructs. This introduces potential inhibiting overhead to the entire application, as the data needs to be mapped, replicated and shuffled [5]. This modus operandi limits the framework's applicability to various input sources and application behavior, such as processing intensive applications and diskless systems. In our implementation, we programmatically represent the input as an abstract stream of data. This gives the user the ability to source the data either from disk or memory then convert it into a stream, allowing the platform to be plugged in a myriad of environments.

B. Design Decisions in LEMO-MR

The input data set must be split so worker nodes may process it. The solution to this problem, however trivial it may be, constitutes the first hurdle in a successful MapReduce emulating platform. Hadoop tackles this problem by delegating it to its inherent file system. The HDFS automatically performs the splitting and allocation of input to worker nodes upon reception of such data. The operation is separated from the MapReduce application itself, but still incurs latency considerations as the framework itself cannot be launched before the file system can account for all its blocks and their replicas. In LEMO-MR, we aimed for a configurable and flexible design in an effort to maintain an important benefit of the direct MapReduce approach, but also to allow for different input sources to be usable.

1) Elastic Model for Dispatching Jobs to Workers: In our approach, we decoupled the input processing module from the input source by providing a data abstraction compatible with a stream of data, be it disk-based or memory-based. This allows for node size to be quickly increased and decreased because nodes are independent of data placement, and only needed for processing. The input module here is oblivious to the memory or file source as it is designed to work with a stream abstraction. This gives the user the ability to either fashion memory or disk input into a stream of data to be fed to the platform, thus allowing the system to process input coming either from disk or memory. In so doing, we do not preset the input files as Hadoop does, but rather we evaluate the data mid-stream as the application is started, also allowing us to elastically reconfigure our cluster. The input size is first evaluated, the number of participating *worker* machines is then accounted for, and subsequently LEMOR-MR divides the input into chunks, according to the number of participating workers.

The user can forgo this step by disabling the "Dispatch-Only-Mode" when desiring to run a filesystem-based approach to the same application. This then allows for input to be present to the worker nodes and remain resident for as many iterations of different MapReduce jobs as desired. Elasticity in this mode is still maintained at the cost of a slight delay due to recalculation, still avoiding the framework to be stopped, torn down, and restarted as Hadoop requires [5]. `Figure 1` presents an accounting of such costs for Hadoop, LEMO-MR, and LEMO-MR in Dispatch-Only-Mode, for 8, 16 and 32 node clusters respectively handling 1GB of raw data.

2) Master-Worker Communication Messages: Upon launch, LEMO-MR divides the input and simultaneously dispatches it to its various recipients in a data stream. The nodes designated as mappers also subsequently receive the *map* function, and the ones designated as reducers receive the *reduce* function from the master through the user source code sent to all the workers. The user provides these facilities as part of the structure comprising the application source code. The user is free to implement the `map` code accordingly.

3) Fault Tolerance: Hadoop also insures that each node is carrying its share of the burden by monitoring their health status in the form of a *heartbeat* signal. LEMO-MR follows this model in a low overhead fashion. A heart beat sent to the master can be a steadfast way to detect node failure and reschedule a task if necessary. Another alternative, one which avoids the burden of network packet abundance, especially when dealing with thousands of nodes, is the use of a self-reporting mechanism on the part of the workers. Upon operational 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 the communication pipe linking them. Furthermore, the master in the context of signaling its workers to begin execution, through a live

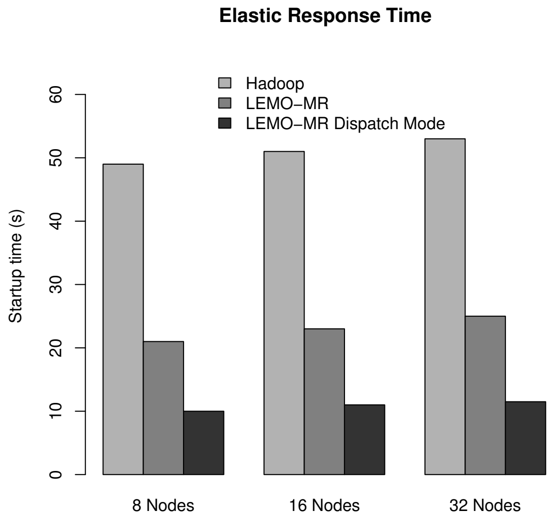

Fig. 1. System readiness response time vis-a-vis cluster size increase for 1GB of data. This graph shows the time-to-readiness exhibited by both systems in the face of cluster expansion. This analysis is also applicable to cluster shrinkage. It is worth noting that these delays grow with cluster size as shown, but also with input data size. The turn around time exhibited by LEMO-MR is much lower even in its capacity as input file manager, because the file manager interacts directly with the underlying file system. Hadoop on the other hand, in addition to requiring input chunk replication, also relies on the HDFS as a layer of indirection between it and the file system, thus causing delays. In dispatch-Only-Mode, LEMO-MR shows faster turn around times, as input streaming is performed during the application runtime. The system *(dispatcher)* decouples the input placement scheme from the nodes, and thus allows for the cluster size to be dynamically scaled. This provides an elastic framework, capable of growing or shrinking on-demand.

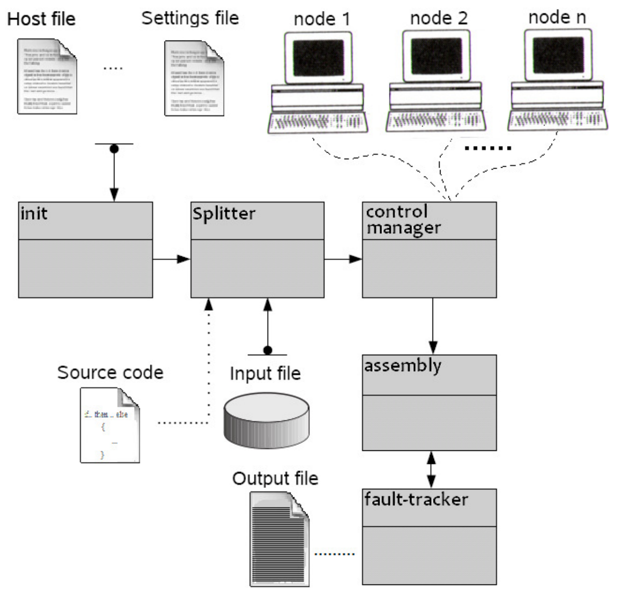

Fig. 2. Architecture of LEMO-MR, implemented in Java. The user provides a list of hosts to the host file, and a settings file. Upon launch, the *init* module parses both files, updates its data structure with relevant variables including input directory location, source code path, and number of hosts to use from the host file. The relevant information is then passed on to the *splitter* which splits the input and dispatches it to the chosen nodes. The *MapReduce Control Manager* then provides the worker nodes with running instructions. Once the *MapReduce Control Manager* (MRC) accounts for all the necessary completion signals, the *assembly* module consults the fault table through the *fault-module* and returns the flow to the MRC for re-runs if necessary. Alternatively, the *assembly* module assembles and provides the output to the user through the output directory specified in the settings file.

execution thread, receives the completion status of the *map* functions. Should the 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.

The architecture of this model is featured in `Figure 2`, which shows the different parts of the model, along with their interactions.

III. LEMO-MR VS HADOOP: SIMILARITIES AND DIFFERENCES

In our prior work, we noticed that for jobs bearing a moderate data load, Hadoop tended to perform poorly, and sometimes stood out-staged by a single operating node [4]. Our architecture is based on the traditional framework, following the contours to the original concept of MapReduce, and designed for low overhead.

The LEMO-MR framework consists of five principal modules: the *initialization* module, the *splitter* module, the *MapReduce Control Manager* (MRC), the *assembly* module, and finally the *fault-tracker* module.

A. The init module

The *initialization* module is tasked with parsing the *settings* file and gathering valuable settings provided by the user. Such variables range from the location of the *host* file, to the diverse input and output directory paths to the number of hosts to pluck out of the *host* file. The user provided source code is compiled on the master side and sent as an executable archive so it can both run and be instantiated, if need be. Doing so allows independence of the nodes, as all workers are sent an executable version of the user code, allowing them to work on their input split independently without requiring constant *map* and *reduce* instructions from the master node. Hadoop on the other hand streams its instructions to its workers, and does not allow the user to pick and choose which nodes should participate in a given job. Additionally, it does not allow the user to pick how many nodes in the host file should be selected for each particular job, but instead selects for scheduling all the machines present in its host file. Because of this structural organization, the Hadoop cluster, unlike LEMO-MR, cannot be elastically scaled, as it would require the system to be stopped, reconfigured, and then restarted.

B. The splitter

The *splitter* or *input-splitter* also operates as a dispatcher. Upon splitting the data into the necessary number of chunks, the *splitter* sends the shards to the different workers through thread spawning. This ensures that the dispatch operates in a concurrent way. The *splitter* also provides to the workers the user source code in the same manner. Split organization among the workers is kept track of, giving the splitter the alternate role of distributed file system manager. The data passed by our splitter can be sourced from either disk or memory, or even streamed into the splitter module from a remote machine, which Hadoop MapReduce does not permit.

C. The MapReduce Control Manager

The *MapReduce Control Manager* is tasked with starting the workers with their work. In LEMO-MR, a remote execution command is sent to all workers through individual and grouped threads. The workers upon reception of the command commence their execution and upon failure or termination return a code describing the outcome of the execution, be it successful or not. Upon sudden death, the workers break the communication pipe, allowing the master to take note of the fatal failure of a given worker node. This scheme dwells as a low overhead method for communication with the workers while still allowing them to work independently. During the run, the nodes do not burden themselves, the network, and the master by advertising their status. In fact they only do so before dying, or fail to do so when the master inquires after receiving most of the completion stubs sent by other successful nodes. Hadoop MapReduce maintains active communication with its nodes to detect failures and communicates processing instructions by seemingly streaming *map* and *reduce* routines as they do not physically appear on the workers' local storage. The large size of a typical MapReduce cluster and the repetitive nature of compute intensive code warrants constant function calls across the network. This coupled with a centralized error and progress reporting mechanism can band to create delays and inherent inefficiencies negatively impacting the performance of CPU intensive applications. In [9], Glimcher et. al. show the negative impact of network resource constriction in distributed environments. Network resource clogging has subsequently been shown by [10] to have worse consequences on performance in cloud environments. This analysis is corroborated by [11], in which Hadoop MapReduce's performance is shown as poor in virtualized settings, and thus in most cloud environments.

D. The assembly

The output assembly module in LEMO-MR is triggered by the master which keeps a tally of completed nodes and decides on user instructions to replicate slow jobs. However, upon receiving all output bits, it assembles them to produce the final output. The assembly goes through the reducer, which may contain user code providing special assembly information such as sorts, triages, or other operations depending on the goal of the job. The assembly however consults the fault tracker module before proceeding further.

E. Fault-Tracker

The *Fault-Tracker* module traverses a node availability table produced by the master. The table keeps a record of the tasks that need to be restarted, and which nodes can be entrusted with such tasks. Upon encountering failed tasks, the module dispatches free nodes and instructs them to accomplish the jobs. The module itself is recursive, meaning should a rescue fail, the rescuing node is added to the table and the module runs again, parsing the table for the next available node to pickup the failed tasks. Dead nodes are pinged in the interim in an effort to update the table should they have come back to life. This effort continues until all output pieces are recovered, or until either all nodes die, or a user specified timeout is reached. Hadoop uses a heartbeat monitor and task replication regardless of failure occurrence. It also does not keep track of nodes that might have resurfaced after suddenly failing. LEMO-MR, on the other hand, migrates tasks as long as one or more nodes are alive, and constantly updates its availability table to register revived nodes.

IV. Distributed Large-Scale Data Processing

Prior benchmarking results on the processing of data intensive Web services applications showed that for Web services toolkits scalability is adversely affected as the size of the datasets increase [12], [13]. While traditional approaches of MapReduce focus entirely on processing time savings on scaled dataset sizes, it has also been shown that the same is true for memory sized datasets requiring significant amount of processing cycles [4]. The use of parallelism as MapReduce offers is a benefit in both scenarios where processing is in high demand, even with a small input dataset.

This case applies well to in-memory applications where the input is small enough to spend its entire life-cycle in memory, but intensive enough to demand significant processing per each single element contained in the input. The original approach of MapReduce is rather well known, and constitutes the traditional demand for the paradigm whenever the input sits too large to fit in memory and needs to be parallelized as a manner to exercise the resources of additional nodes.

The application domain we have chosen for testing our implementation is processing for large XML datasets. Scalable processing of datasets is a critical concern as the size of data used by applications has steadily increased over the years in both scientific and business applications. For example, the MetaData Catalog Service (MCS) [14] provides access via a Web service interface to store and retrieve descriptive XML information (metadata) on millions of data items. While the Web service approach of MCS provides interoperability, widely used toolkits such as Axis [15], Libxml [16], and Piccolo [17], are typically designed to process small-sized datasets. Additionally, scalable processing of information is now of critical importance in scientific applications where the data size is large, and processing on a single node is not an acceptable option. Our framework for implementation and experimentation with the MapReduce model can be used for large-scale data-intensive processing, but also in CPU-intensive scenarios where the input is smaller, either disk-based or solely resides in memory, as is sometimes the case. Even though we use XML as the data format for performance comparisons, our framework can be used for different scientific data formats such as HDF5 [18] and NetCDF [19], when the MapReduce model is applicable for the distributed application. In the context of performing data processing, we use AxisJava, a widely used web services-based application toolkit. It is a processing intensive SOAP engine capable of creating SOAP processors, which in turn can be used in parsing data content. The resulting tokens can then be streamed into application friendly content and returned to the user. For input, we had the choice of different XML-represented documents. It has however been shown that AxisJava tends to require multiple CPU cycles in intensive computations when it comes to parsing and processing arrays of type double [12], [20]. The computation is CPU intensive regardless of overall data size. This fact led us to choose this particular data set for our input, as good performance behaviors if any in our comparison, will be clearly contrasted with low performance behaviors. We also chose this experimentation path because XML is widely being used as an encoding of choice for many scientific, healthcare, and financial applications. For our experiments, we extracted AxisJava's parsing module and integrated it into our applications. Both Hadoop and LEMO-MR run off similar source code powered by the AxisJava parsing engine. We tested both frameworks in a myriad of cluster computing scenarios regarding not only performance but also fault tolerance. While a high overhead prone system can be a detriment to its own performance, overhead generating design decisions such as data replication, shuffling and constant network chatter as in Hadoop's case contribute to the reliability of its framework. A key goal of our experiments is to assess the impact of reduced overhead approaches on reliability in the face of job run time. We achieved this by purposely causing gradual node failures as both systems performed. In all the experiments, we ran tests for LEMO-MR along side Hadoop using identical node counts and input data.

V. Performance Results

We run our tests on a selection of two machine classes:

- $1\times$ dual core – One desktop-class machine, which has a single 2.4Ghz Intel Core2 6600 with 2GB of ECC RAM, running Linux 2.6.24. The filesystem in use here is ext3fs.
- $2\times$ uniprocessor – 1U nodes in a cluster, each of which has two 3.2Ghz Intel Xeon CPUs, 4 gigabytes of RAM 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 filesystem in use in the test directory here is reiserfs.

Figure 3 shows how data analysis time can be saved by our framework wherein primary operational concern is harnessing the power of additional processing units while leaving as little overhead footprint as feasible. This is possible through our

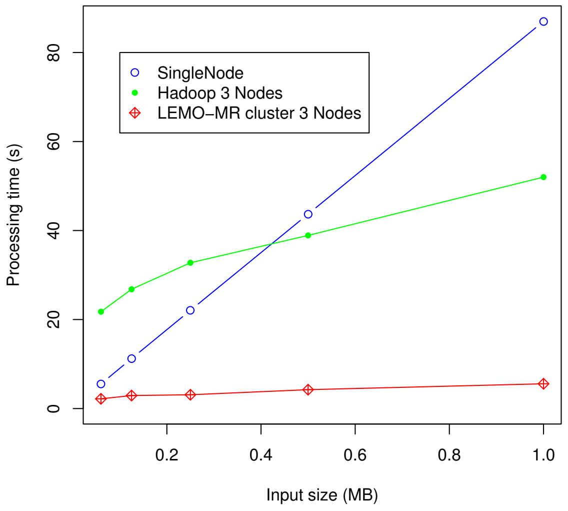

Fig. 3. Shows the performance of a 3 node Hadoop cluster compared with our designed framework running the same code with 3 nodes as well, up against a single node computer. The experiment shows relatively small input sizes as to illustrate the performance of a single computer system in the same processing scenario as the Hadoop cluster and LEMO-MR cluster. This affords the reader a baseline as to the performance of a single desktop computer in this environment. The graph also shows on a smaller scale, the behavior of all the systems involved. Even as it is rightly so to assume that MapReduce systems like Hadoop and our design are better suited with large input data sizes, this experiment allows us to illustrate the amount of overhead still taking place with small inputs as a means to later showcase and explain this prevailing trend as input files grow larger. Up to 400KB of data, the single node performs better than the Hadoop cluster. This is in part because in the case of Hadoop, every node in the cluster is not only vying for one single collection file in HDFS, but due to cost incurred from shuffling, data replication, and intensive node to node streaming communication, while the input size is not large enough to offset those performance dampening factors.

design by establishing node communication only when necessary, and by making use of the underlying system as much as possible, thus limiting layers of abstractions contributing to performance slowdown in Hadoop. The Hadoop cluster, running three nodes, is out staged by a single system for data files containing less than 400KB of input. Admittedly, MapReduce is commonly used with large input sizes in the scale of PetaBytes of data, and as such while a 400KB to 1MB data range seems like a small feed for Hadoop. This however allows us to point out that even for small input sizes, a pattern of processing latencies that will undoubtedly grow as data and CPU intensity grow in compute intensive scenarios. Due to the computation vs communication trade-off in distributed systems, a large input data size does not always warrant cluster use. A single desktop system can still sequentially process as much data as its memory would allow and run faster than a cluster if the work load per input unit is low [4].

Figure 4 displays the performance output produced in the processing of a 2.5GB input file as the number of nodes in charge of the processing is gradually increased. The argument indicting operational overhead is magnified in this graph.

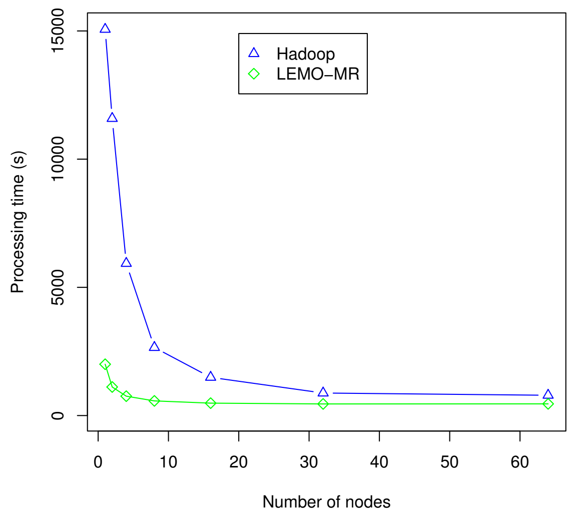

Fig. 4. Performance displayed by the different platforms as the number of participating worker nodes is gradually increased from 2 to 64 on a 2.5 Gigabyte dataset. LEMO-MR outperforms Hadoop by 173% for 64 nodes, while running up to 7 times faster for smaller configurations

For a low number of nodes, the difference in performance between the two frameworks is significant, but as more nodes participate in the work, Hadoop regains some stability. The same can be said of our framework. However, in our case, the stabilization point of the curve occurs sooner and for a lower number of nodes involved. This points out that as more operational cost is produced, either more nodes are necessary for significant performance speedup, or more data processing needs to occurs to the same end. It is thus natural in this scenario that our low overhead framework would yield better performance. However, it is interesting to notice the consistency of the pattern as increases in data and nodes are induced. With less than 32 nodes present in the cluster, both curves can be observed to be quite steep. Over 32 nodes, however, they flatten due to the fact that shipping data around the cluster and managing the nodes begins to dominate the overall work-load as the optimum number of machines necessary for the job at hand has been reached and exceeded. In fact, at this point, adding more machines will not help improve performance and can instead hinder it. The file size is 2.5GB, which roughly corresponds to over 240 million data items.

Figure 5 presents data similar to that in figure 4, using speedup (computed as $\frac{T_1}{T_p}$) and a 2.5GB input file. The LEMO-MR cluster not only performs better, but also scales better due to low overhead considerations. In this case, T_1 is computed by running the test against a Hadoop cluster, and LEMO-MR.

Figure 6 presents the performance of both clusters as nodes are led to fail. 1, 2 , 4, 8, and then 16 nodes fail in independent runs within the 60 node cluster. Both platforms triggered their recovery mechanisms in order to complete the work left by

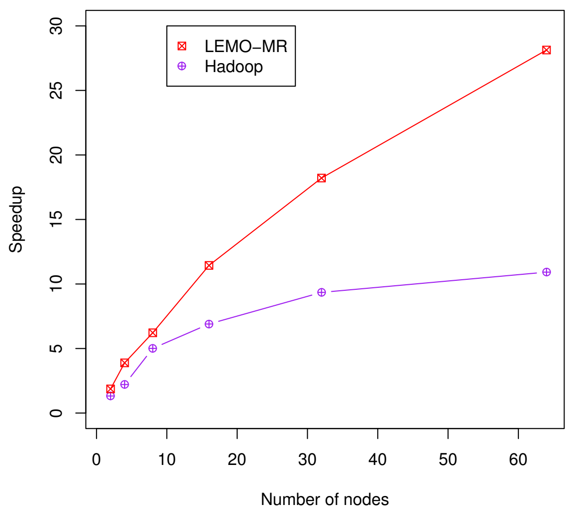

Fig. 5. Speedup achieved by both frameworks over a single node on a 2.5GB dataset, roughly 240 million items. LEMO-MR scales much better because it carries a lower overhead cost per additional node.

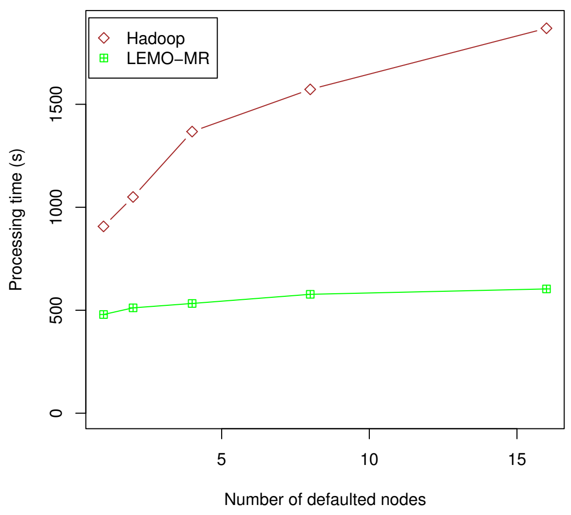

Fig. 6. Processing time produced over gradual failure of nodes. Running times are recorded for a 60 node cluster in both cases processing a 2.5GB file while 1, 2, 4, 8, and 16 nodes are brought to fail, thus triggering both systems' fault tolerance mechanisms.

the failed nodes. The first observation made is that despite the on-demand approach of our framework, performance is not hampered by the potential delay in rescue response. This is primarily because, as we hypothesized earlier, a homogeneous cluster does not benefit much in terms of real-time failure reports. The reason of this being, in this type of design, remaining nodes are likely to still be performing their tasks, and as such will not be able to readily assist fallen nodes. In fact, such nodes should only be free of their own prerogatives slightly after the fault is reported, and not when it occurs. Even if the failure is reported as it occurs, no node may yet be available for the re-runs. The latency in our case exists, but is minimal compared to Hadoop. Furthermore, the gain of not having to constantly query nodes, ship and duplicate tasks across the network, proves valuable in this scenario. This is shown in the graph as the processing differences are maintained even under failing node conditions. The LEMO-MR framework however shows a steadier graph, and a smaller increase of processing delays overall from 1 node defaulting to a 16 node fall out. This exemplifies the fact that restarting a process on another node, will suffer the same performance degradation as on the original node. Hadoop might also suffer from a worse case scenario because failures in its case do not have the same impact as LEMO-MR. The failure of the wrong combination of nodes, meaning nodes all holding a particular data block, might cause its replication factor to drop below the acceptable threshold thus triggering new rounds of replication, thus taking more time. The failure of inconsequential nodes might, on the other hand, simply not trigger the same level of activities leading to costly operations taking place. It is also difficult to track those chunks in targeting only inconsequential nodes for the benefits of only inducing best case scenario failures. In fact, in real application scenarios, just as in our tests, failures themselves may not always take place in the best of combinations.

Figure 7 presents data representing the performance of both clusters as the required processing time on each one of their individual input items is increased. Using speedup (computed as $\frac{T_1}{T_p}$), and a 10 million item input file, demanding processing times are applied per each input element parsed from the input file. As can be observed, Hadoop falls further below the LEMO-MR cluster as work load increases on each input item. This goes to corroborate the fact that workload aggravates already existent performance hampering overhead. In general, processing load for a job must trump the different overhead considerations of that job, for the use of the cluster to be fully efficient. Speedup is computed against a single worker system, and speedup is also shown between increasingly sized configurations ranging from 16 nodes to 64.

VI. Related Work

Microsoft Dryad [21] is a parallel execution framework that allows for the execution of computations expressed as direct acyclic graphs. Disco [22] is a MapReduce implementation inspired from a functional language, Erlang [23]. Disco espouses the concept of a lightweight framework as well as lightweight programs. The framework carries Python applications, facilitating application development, but has limited capabilities compared to Hadoop, or compared to the LEMO-MR framework. Ekayana et al. present `CGL-MAPREDUCE` [7], and like LEMO-MR leverages the power of input data streaming, but fails to provide direct distributed file system support and fault-tolerance. None of the works highlighted above as of yet provide the full set of capabilities provided by LEMO-MR.

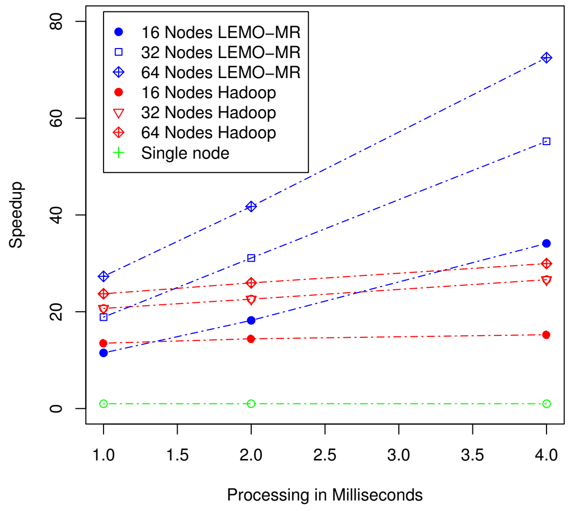

Fig. 7. Speedup reached by the Hadoop cluster and the LEMO-MR framework over a single node run while faced with increasing load intensification. The amount of processing shown is the amount in milliseconds each input item requires to be processed for. By increasing this variable, we assess the performance of the system in CPU intensive scenarios, and we show the wide disparity between both systems as CPU-intensity pervades throughout the task. The 100MB input file contains a little over 10 million data elements going through the cluster. The green line shows the performance threshold of a single machine system facing the same tests.

VII. Conclusions

Concurrent exploitation of compute nodes focused on single tasks can provide faster processing times for data and computation-intensive applications. While MapReduce has long been associated with data size, the paradigm can be similarly efficacious when redesigned and applied to processing-intensive applications and diskless systems. MapReduce however as it currently stands does not provide for structures allowing CPU-intensive applications to thrive. The assessment of this situation is however an implementation issue. The model itself, as we showed is well suited for such environments. Most MapReduce implementations are suited for data-intensive scenarios, offering little processing on such data, and as such suffer when processing per items on the data processed increases. LEMO-MR shows that available computing resources can be effectively used while requiring minimal commitments and changes to underlying systems and hosts. While fault-tolerance appears as a high overhead construct, it can still be achieved in a low cost manner on an on-demand basis while still producing fault quenching capabilities and minimally generating performance impacting overhead. Our experiments showed not only the merits of a minimalistic implementation, but also the lack of impediment on a MapReduce application's fault-tolerance capability even as it operates in a low overhead context. MapReduce when implemented through indirect layers of abstractions can suffer from a high overhead to work ratio, not suitable for processing-intensive applications, and diskless systems. The repetitive nature of operations in a cluster implementing this model can furthermore aggravate this condition. In LEMO-MR we opted for:

- The abstraction of the input interface to allow disk-based, stream-based, or memory-based input.
- Compute node independence and decoupling, allowing cluster elasticity at very low cost.
- Compute node elasticity by allowing nodes to be seamlessly added and removed from the cluster.
- Reducing Master to node communication in allowing nodes to directly instantiate the user code.
- On demand fault-tolerance response, allowing the system to increase efficiency in dramatically reducing network traffic and communication.

VIII. Future Work

For applications that require processing of Terabyte size datasets, mass storage is more likely to be arranged in higher performance configurations such as RAID, NAS, and SAN. These configurations are likely to efficiently feed multiple data streams to concurrent threads. We plan to study the benefits and limitations of LEMO-MR when applied to such cases. We also plan to study the threshold points for our custom implementation for a wider range of application datasets used in cloud environments.

References

[1] 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

[2] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner, "Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey," *SIGMOD*, 2000.

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

[4] Z. Fadika, M. R. Head, and M. Govindaraju, "Parallel and Distributed Approach for Processing Large-Scale XML Datasets," in *10th IEEE/ACM International Conference on Grid Computing*, October 2009, pp. 5–7.

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

[6] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project*, 2007.

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

[8] 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*.

[9] L. G. V. T. Ravi and G. Agrawal, "Supporting Load Balancing for Distributed Data-Intensive Applications," in *IEEE International Conference on High Performance Computing (HiPC'09)*, Kochi, India, December 2009.

[10] H. Qiming, K. Shujia, and T. Duffy, "Case study for running hpc applications in public clouds," in *ScienceCloud '10: the 1st Workshop on Scientific Cloud Computing*. Chicago, IL, USA: ACM, 2010.

[11] 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.

[12] 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.

[13] M. R. Head, M. Govindaraju, R. van Engelen, and W. Zhang, "Benchmarking XML Processors for Applications in Grid Web Services," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, p. 121.

[14] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman, "A Metadata Catalog Service for Data Intensive Applications," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 33.

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

[16] D. Veillard, "The XML C parser and toolkit of Gnome," 2006, http://xmlsoft.org/.

[17] Y. Oren, "Piccolo is a small, extremely fast XML parser for Java," 2006, http://piccolo.sourceforge.net/.

[18] *Hierarchical Data Format (HDF)*, http://hdf.ncsa.uiuc.edu/.

[19] *Network Common Data Form (netCDF)*, http://www.unidata.ucar.edu/packages/netcdf/.

[20] 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.

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

[22] Disco project. [Online]. Available: http://discoproject.org

[23] Erlang programming language. [Online]. Available: http://www.erlang.org