Lawrence Berkeley National Laboratory
Lawrence Berkeley National Laboratory

Title
Benchmarking MapReduce Implementations for Application Usage Scenarios

Permalink
https://escholarship.org/uc/item/35b550ph

Author
Fadika, Zacharia

Publication Date
2011-09-21

eScholarship.org Powered by the California Digital Library
University of California

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.

Benchmarking MapReduce Implementations for Application Usage Scenarios

Zacharia Fadika [1], Elif Dede [2], Madhusudhan Govindaraju [3], Lavanya Ramakrishnan[‡]

Grid and Cloud Computing Research Laboratory,
Department of Computer Science, State University of New York (SUNY) at Binghamton
and [‡] Lawrence Berkeley National Laboratory
[1] zfadika@cs.binghamton.edu, [2] edede1@binghamton.edu
[3] mgovinda@cs.binghamton.edu, [‡] lramakrishnan@lbl.gov

***Abstract*—The MapReduce paradigm provides a scalable model for large scale data-intensive computing and associated fault-tolerance. With data production increasing daily due to ever growing application needs, scientific endeavors, and consumption, the MapReduce model and its implementations need to be further evaluated, improved, and strengthened. Several MapReduce frameworks with various degrees of conformance to the key tenets of the model are available today, each, optimized for specific features. HPC application and middleware developers must thus understand the complex dependencies between MapReduce features and their application. We present a standard benchmark suite for quantifying, comparing, and contrasting the performance of MapReduce platforms under a wide range of representative use cases. We report the performance of three different MapReduce implementations on the benchmarks, and draw conclusions about their current performance characteristics. The three platforms we chose for evaluation are the widely used Apache Hadoop implementation, Twister, which has been discussed in the literature, and LEMO-MR, our own implementation. The performance analysis we perform also throws light on the available design decisions for future implementations, and allows Grid researchers to choose the MapReduce implementation that best suits their application's needs.**

I. INTRODUCTION

MapReduce is inspired from functional programming primitives *"map"* and *"foldr"*, whereby a programmer can apply or "map" a function to an input set, and in doing so obtain a resulting output set reflecting the transformation applied, but bearing the same length. The resulting set, as with *"foldr"* in Haskell, is then submitted to a reduce method whose role is to apply the function to the "map" output. In MapReduce, this occurs in a distributed manner across a wide array of machines, each holding a piece of a larger input file or component. The model's greatest appeal resides in the fact that it hides parallelization and synchronization responsibilities from the user. The *map* and *reduce* functions are as such written as single node programs by the programmer, then subsequently parallelized, and synchronized by the framework. In comparison to MapReduce, while distributed programming models such as MPI [1] can offer parallel data processing, such models require the user to implement data splitting, data management, parallelization, synchronization and fault-tolerance with their applications, perhaps differently for each particular application. MapReduce on the other hand takes all the aforementioned responsibilities off the programmer's slate as the framework internally implements and maintains such features.

Parallel and distributed applications place a wide range of requirements on the communication substrate and data formats. These requirements include turn around time, minimal memory footprint for improved caching efficiency, handling of CPU-intensive jobs, management of data-intensive jobs, and fault tolerance when deployed on very large clusters. Such requirements have led over time to a wide range of design and implementation choices. A comprehensive benchmark suite tailored for HPC applications can aid in determining the MapReduce framework that has the most optimized implementation for the class of applications under consideration.

In this paper, we compare and study the performance of Hadoop [2], Twister [3] and LEMO-MR [4] in various real-world application usage scenarios, including data-intensive, iterative, CPU-intensive, and memory-intensive loads. We not only compare the chosen frameworks in several real-world application scenarios, but also in real-world cluster scenarios, including fault-prone clusters/applications, physically heterogeneous clusters and load-imbalanced clusters. This performance study provides insight into the relative strengths and weaknesses of different implementations under different usage scenarios. We link the observed behavior of the frameworks to the design and implementation choices of the particular MapReduce platform being tested. Our test framework will be made available to the grid community for researchers to compare frameworks for their custom usage scenarios and data sizes of interest.

II. CHOICE OF MAPREDUCE IMPLEMENTATIONS

The MapReduce model is anchored around three central principles: data management, synchronization/parallelization abstraction, and fault-tolerance. These principles led us in the choice of Hadoop, Twister and LEMO-MR. While there exists many "MapReduce-like" implementations, very few of them implement the principles highlighted above and as a result were not selected. Octopy [5], Skynet [6] among others are an example of such cases. Other similar implementations only support very small datasets and interpreted languages, and as such would not have been competitive choices in

the context and the test environment of this study. Twister for its part does not feature a node specific fault-tolerance mechanism, but was selected for its data management and synchronization/parallelization abstraction features. Other known MapReduce implementations like Amazon EMR [7], are an adaptation of Apache Hadoop and as such were not included here. Similarly, DELMA [8] being built ontop of LEMO-MR, we did not choose to include it here as well. MapReduce frameworks like MARS [9], CGL-MapReduce [10], and FPMR [11] were also not part of this study, as they are simply MapReduce API, and not implemented MapReduce frameworks.

A. Twister: Iterative MapReduce

Twister is an open-source, lightweight MapReduce framework. Twister allows for data access via local disks, and offers efficient support for iterative MapReduce computations [3]. Twister uses a publish/subscribe messaging infrastructure for communication and data transfer. The framework does not provide job specific fault-tolerance, but rather fault-tolerance in certain stages. A task along the path of a job can be restarted, but not rescued if one of the nodes performing it fails. Although unique to Twister, the iterative nature of the framework can be replicated in Hadoop or any other framework closely following the MapReduce model through scripting. We demonstrate such an approach and compare it to Twister's inherent iterative feature in section IV.

B. Hadoop MapReduce

Hadoop is an open-source MapReduce framework [2] anchored on its own file system: The Hadoop Distributed File System (HDFS) [12], inspired from the Google File System [13]. The HDFS sits as a layer of abstraction between Hadoop and the native file system. Upon copy of the data file into the Hadoop input directory, the HDFS automatically splits the file into blocks [13]. The blocks are then replicated according to a replication factor pertaining to Hadoop's internal settings and distributed among chosen nodes called `DataNodes`. As cluster nodes fail, `DataNodes` are given the ability to replicate their own file blocks. `DataNodes` also periodically send updates and reports of block conditions, such as block usage and integrity to the master node. Even though it is expensive, especially for heavy computational 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 [14].

C. LEMO-MR

LEMO-MR is a low overhead MapReduce implementation, optimized for in-memory and specially CPU-intensive applications. As shown in [4], overhead prone MapReduce implementations tend to perform poorly in CPU-intensive contexts, as that overhead is exacerbated in such settings. Cluster upkeep and fault-tolerance expectedly add overhead costs. To tackle these problems, LEMO-MR adopts a "node independence" policy, whereby nodes are not tightly coupled to their master, and as such do not require constant communication with it. Also, nodes do not "own" the data they run on and can be swiftly replaced upon failure, because their failure does not involve the disappearance of their data or the need to replicate such data. This allows the framework to avoid running and maintaining resource hungry daemons, keep constant pings between nodes, and execute performance hampering redundancy checks not at the node level, but rather at the master level. The master is in any case likely to be idle while the nodes are performing, and can take on those responsibilities. Leaving it to the nodes can impede on the performance of the task they are processing.

D. Summary of Differences

Twister, as of version 0.90 does not feature a node-specific fault-tolerance mechanism. The death of a node is not detected, and the task the node was processing is not reassigned to a healthy machine. In Twister's case, a faulty job must be restarted. This is akin to restarting a Hadoop or LEMO-MR job from the beginning because a node died 99% through the job. LEMO-MR and Hadoop are capable of detecting failures and are also capable of replacing dead nodes with healthy ones, in some cases with minimal impact on application runtime. Hadoop uses the HDFS for this purpose. The HDFS sits on top of the file system present on its nodes. It was mainly designed for fault-tolerance, and with its replication policy, speculative tasks policy, and need for constant node pinging, can cause Hadoop to incur some performance degradation. LEMO-MR features a low overhead approach, in that it reduces inter-node communication to a minimum. The framework also features a fault-tolerance mechanism completely independent of data placement. Hadoop has a full-fledged load balancer service. LEMO-MR however, much like Twister, does not feature a load balancer, allowing for slow nodes to transfer parts of their load to fast ones. Finally, Twister is iterative by nature, a feature not available to Hadoop and LEMO-MR through their framework, but rather, through elaborate user scripting.

III. PERFORMANCE TEST DESIGN

The tests Hadoop, Twister and LEMO-MR are subjected to in this paper are meant to represent the characteristics of possible problems solved using the MapReduce paradigm. Depending on the nature of the application, be it data-intensive, iterative, CPU-intensive, or fault-prone, different design choices in a MapReduce framework can lead to either good or dismal performance. Our goal with these tests is to highlight the strength and weaknesses of the tested subjects under a slew of possible domain problems they are expected to face as MapReduce frameworks.

A. Data-Intensive Tests

In contrast to a data-Intensive application, the amount of data produced by CPU or memory-intensive applications tends to be small. For instance, searching through 1 TB of files for a keyword would yield an output file with just a filename and a line number, while sorting 1 TB of data would yield a 1 TB

output file. While the first example can be considered CPU-intensive depending on the search algorithm, the latter case is clearly data-intensive. In this paper, we qualify as data-intensive, not only the processing of large input, but also the production of large output. Such applications tend to involve the reducer more than the mapper, and tend to be much slower than the other classes of MapReduce applications because of the constant I/O solicitations.

B. CPU-intensive Tests

CPU-intensive applications tend to involve the CPU for longer cycles than data-intensive applications. While monitoring CPU usage on the worker nodes, with a CPU-intensive program in MapReduce, all CPUs in a multicore case tend to show very close to 100% of activity for the entire span of the application. The same type of monitoring for a purely data-intensive application, such as random number generation as we will feature later, tends to keep the CPU usage around 5 to 10% on average. CPU-intensive applications may not process very large input data, but will spend more time processing each piece of data compared to a data-intensive application which processes a lot of data, but may spend very little time on processing each piece of it.

C. Memory-Intensive Tests

A memory intensive application in a MapReduce context requires large memory space from each worker node as data needs to be entirely loaded into memory, rather than processed line by line, or in small feeds. The latter case is usually common to CPU and data intensive applications. While a CPU-intensive application might require each worker to read in one integer, and subsequently find all of its factors, then move to read another integer into the same variable to the same end, a memory intensive application will require perhaps a thousand integers be loaded into memory to determine their common divisors. Memory-intensive applications can usually not be run on single node systems because of their memory limitations. Such limitations need however to be tested in the MapReduce arena because of the differences in memory footprints imposed by the implementations. While a worker node has access to the same physical memory capacity before Hadoop, Twister or LEMO-MR is run on it, the question is: How much available memory is left for data processing on the node once the MapReduce implementation has started? The tests in this rubric are designed to answer the above question, and will quantify the memory footprints of the different implementations.

D. Cluster Heterogeneity Tests

Heterogeneity, as we define it for this benchmark, pertains to the configuration of nodes in a cluster. In our tests, we setup a cluster of 60 cores, composed of a 48 core computer, an 8 core computer and a 4 core machine. Similarly, disk speed, memory capacity and speed decrease with the machine class. The 48 core machine we feature has as such faster and more memory than the 8 core machine, which similarly dominates our 4 core machine. The goal of our heterogeneity test is to determine the extent to which each MapReduce framework adapts to such a diverse environment. The naive implementation approach of MapReduce is to allocate equal work for all machines. This common approach as shown in [15] [16], [17] works rather flawlessly for homogeneous cluster, but presents dismal performance in heterogeneous settings as the same references put it. It is good to remember that MapReduce was first presented as the ”framework” for commodity machines. Such machines could be a mix of modern and ancient machines randomly collected for the purpose of building a cluster. In such settings, a MapReduce platform with strong heterogeneous cluster support could be preferred.

E. Load-Balancing Tests

Load-balancing here pertains to how well a MapReduce implementation adapts to a homogeneous cluster turned heterogeneous with one or more of its members under load or stress. While a static load balancing strategy like advertised in LEMO-MR can help with a physically heterogeneous cluster, a homogeneous cluster prone to change due to load imbalance can offer it an unsolvable challenge. This condition is ever more popular in community owned clusters and research laboratory settings, perhaps with users running different types of applications on the same nodes. This as we will show can negatively impact the performance of a MapReduce framework devoid of an on-the-fly load re-adjustment strategy.

F. Iterative Application Tests

Iterative applications typically require multiple passes on the same data to achieve a desirable or a required output. In Hadoop and LEMO-MR cases, this would require constantly serializing and de-serializing such data, as the data needs to be loaded from disk to memory, then back to disk, as many times as there are iterations. Twister however, benefits from memory transfers between iterations, allowing more efficient iterative application processing.

G. Fault-Tolerance Tests

Fault tolerance is a key component for users of the MapReduce model, as faced with a cluster of hundreds and thousands of nodes, the probability of a faulty machine is simply heightened. The prospect of total job failure and time loss because of a single node failure is simply unacceptable. Fault-tolerance can however be realized in different ways, with performance critical consequences. Twister, being devoid of a node specific fault-tolerance mechanism, it is not capable of salvaging a job which has lost a certain number of its nodes. For this benchmark category, we solely test Hadoop and LEMO-MR for fault response and application runtime under increasingly dying nodes. We start of with a cluster of 50 nodes, 200 cores total, and progressively kill 1, 2, 3, 4 and 8 nodes, then record application slowdown in the face of those failures.

IV. Distributed Large-Scale Data Processing

In this section, we effectively test Hadoop, Twister and LEMO-MR with data-intensive, CPU-intensive, iterative, and memory-intensive applications. We subsequently test our three MapReduce frameworks in a physically heterogeneous cluster and a load-imbalanced homogeneous cluster. We used for testing purposes Apache Hadoop 0.20, Twister 0.90 and our only and LEMO-MR presented in [4]. In all the experiments showcased, we ran Twister and LEMO-MR along side Hadoop using identical nodes, identical node counts, identical input data and similar user source code. Among the applications selected, were:

- A random floating point number generator for our data-intensive tests, where between 100GB and 1TB of floating point random numbers are generated.
- Matrix multiplication for our CPU-intensive rubric.
- PigeonHole sort for our memory-intensive tests. This test loads large unsorted sets into memory from an already space-demanding algorithm. Memory exhaustion is caught and recorded for the sizes for which it occurs.
- Distributed Grep for cluster heterogeneity tests, Iterative, and Load-balancing tests.
- Wordcount from Hadoop's own package for fault-tolerance.

V. Performance Results

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 – 1U 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 – 1U 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 – 1U 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.

The top plot of **Figure 1** shows Hadoop, Twister and LEMO-MR each running 64-core clusters with identical nodes, producing identical output sizes, and using similar source code in doing so. Twister operates *map* to *reduce* transfers through memory. This allows Twister to start stronger than Hadoop with 40 and 50 MB of generated random numbers. Twister then quickly slows down as its memory gets progressively consumed, and finally exhausted at the 93 MB mark. LEMO-MR uses a hybrid approach capable of starting with a memory-based transfer scheme between *map* and *reduce*, then progressively dump its memory content to file, making it capable to use its memory longer, and as such run efficiently as the figure shows. Hadoop uses solely a file-based approach. Twister's design can be fast and efficient with jobs requiring much *map* side data processing and very limited *reduce* side data

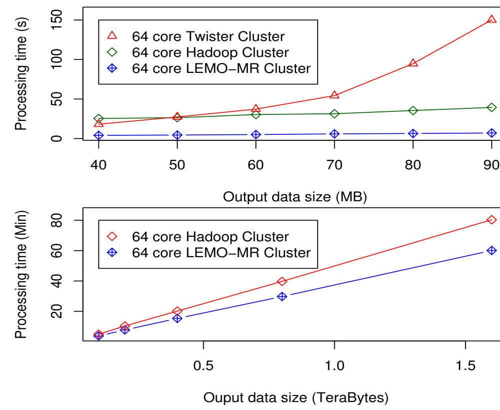

Fig. 1: 64-core Hadoop, Twister and LEMO-MR clusters each producing from 40 MB to 1.5 TB of random floating point numbers. The maximum output scale of the top graph is very small with 90 MB of data for a data-intensive experiment, because, per our research and experiments, Twister is not capable of passing more than 93 MB of data from its mapper to its reducer, as the transfer is memory-based, rather than file-based like Hadoop and memory/file-based like LEMO-MR.

processing, such as CPU and memory-intensive jobs. In the realm of data intensity however, any processing necessitating more than 93 MB of data to be transferred from the *mappers* to the *reducers* would result in memory insufficiency errors on Twister's part. It should be noted that as we studied Twister's source code, we noticed memory threshold increase instructions, preceding job starting instructions. The 93 MB limit we observed here could be raised with RAM upgrades on the worker nodes, and higher memory allocations in the Twister source code itself. However, despite all the mitigating measures laid out above, Twister will ultimately be limited in data-intensive applications by the physical memory size of its nodes. LEMO-MR and Hadoop can however process terabytes, if not petabytes of data from their mappers to their reducers, if the disk-space capacity on the worker nodes allows it. In the bottom plot of **Figure 1** , Hadoop and LEMO-MR generate random number filed output files from 100 GB up to 1.6 TB. Hadoop uses "shuffling" for passage of intermediate data from mapper to reducers [18], [19]. Shuffling consists of providing each datanode with the intermediate result of every other datanode [19]. This mechanism provides fault-tolerance support for failure occurring during the transitional phase, but pays a performance penalty for growing data sizes, as the overhead incurred by the method is exacerbated.

In **Figure 2**, CPU-intensity is tested. This job is CPU-intensive as the frameworks in presence here perform little data reading and witting operations, but rather lengthy loops and floating point multiplications, one matrix after another. For this experiment, all cores in the cluster were recorded with an average of 93% utilization throughout the length of each run, with the exception of Hadoop with a lower mark at 89%. This condition is explained in [19] where Hadoop is shown to lose CPU cycles due to skipped cycle meant for processing data, to the benefit of cluster maintenance. Contrarily to our previous data-intensive tests, and even though millions of matrices are read in and multiplied, a single resulting matrix is passed from mapper to reducer from each mapper. The cumulus of data

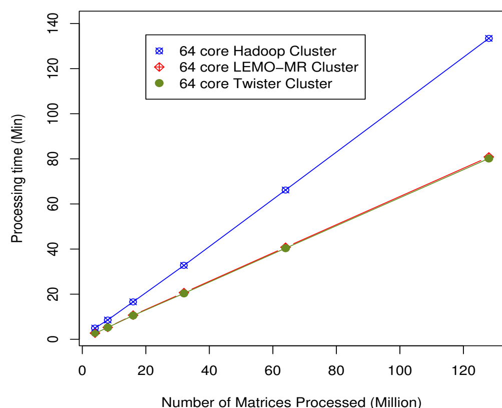

Fig. 2: 64-core Hadoop, Twister and LEMO-MR clusters, each multiplying from 2 to 120 million 33 x 33 matrices.

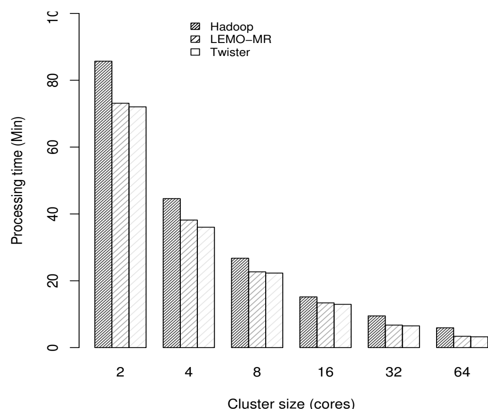

Fig. 3: 5 million 33 x 33 matrices processed by an increasing number of cores ranging from 2 to 64. This graph shows how much processing time is reduced as each framework gets more processing power added to its ranks.

passed from mapper to reducer being very small, Twister can be seen here to thrive. In fact, LEMO-MR and Twister have similar performance, even as Twister shows slightly faster runs than LEMO-MR. Hadoop, here proves to be slower in this CPU-intensive scenario. Hadoop uses a number of overhead-prone operations such as data chunk replication, constant worker node pings along with speculative and redundant jobs, mainly for fault-tolerance reasons [20]. These measures can however impede performance in any application setting [3] However, in [4] this situation is shown to be further aggravated in CPU-intensive scenarios where the pre-dominance of CPU operations and the long processing nature of tasks make duplicating tasks and speculative jobs launched by Hadoop more costly than in data-intensive or memory-intensive cases.

In **Figure 3**, we show how and at what pace each MapReduce framework is capable of reducing runtime with the addition of more processing cores. The trends noticed in **Figure 2** hold true here as well. Twister is faster than LEMO-MR in this context, while Hadoop shows to be slowest overall. **Figure 4**, next, shows the speedup of all three frameworks.

In **Figure 4**, Twister shows better scalability than LEMO-MR and Hadoop. Even as the advantage is minimal versus LEMO-MR, it displays better CPU usage. Further more, the use of a brokering medium in Twister's case with NaradaBro-

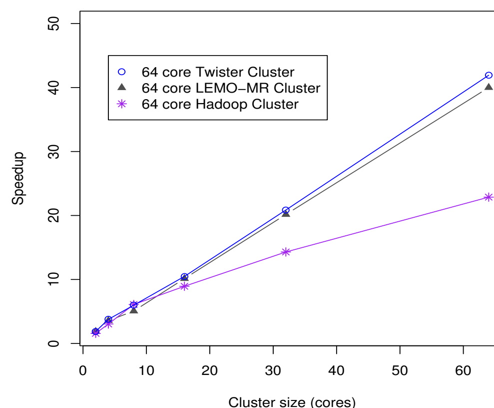

Fig. 4: Speed-up computed from Twister, Hadoop and LEMO-MR clusters processing 5 million 33 x 33 matrices, with diverse cluster sizes ranging from 2 to 64 cores. Speed-up is computed as $\frac{T_1}{T_p}$ and represents how fast each cluster performs relative to a single node system. Twister runs and scales slightly better than LEMO-MR, and much better than Hadoop for this CPU-intensive test.

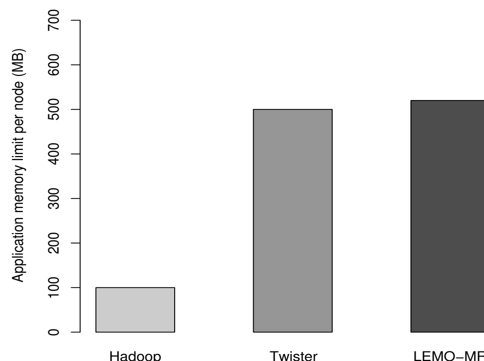

Fig. 5: Shows how much memory Twister, Hadoop and LEMO-MR make available for similar user programs on each node. Hadoop shows at 100 MB memory limit, while Twister shows 500 MB, and LEMO-MR, a 520 MB limit, before causing a memory exhaustion error.

kering or Active MQ [21] [22] allows the framework to decouple communication from processing, making processing more efficient. Heavily communicating nodes need to spare some of their cycles to that end, thus reducing the overall amount of cycles available for user data processing. This condition worsens further in CPU-intensive cases. With LEMO-MR, communication is kept at a minimum, but is not brokered like with Twister, thus causing worker nodes to still "burden" themselves with it.

Figure 5 tests the memory footprint of each MapReduce implementation. Should an input file need to be completely loaded into memory at once in order to be efficiently processed, a single node would not be a good candidate, if the file's size sits beyond its memory limits. In such a case, the file might need to be broken down, and sequentially processed. With a cluster of machines, the break down still occurs, but in smaller sizes, and the processing in this case is in parallel. This case however implies that the MapReduce framework involved leaves enough space in memory after its own maintenance operations for user data to reside in memory. As **Figure 5** shows, Hadoop leaves 100MB per node, Twister leaves

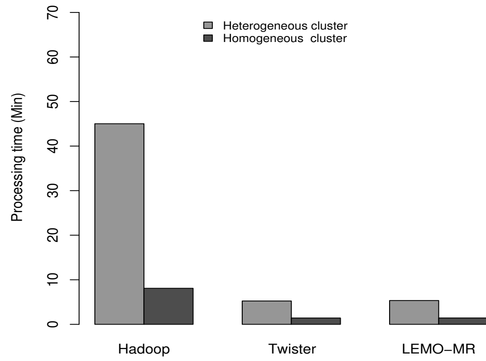

Fig. 6: Hadoop, Twister and LEMO-MR running on performance equivalent 60 core homogeneous and heterogeneous clusters. The application tested here is distributed grep for 36 billion words. Hadoop shows a bigger disparity and loss of performance between its homogeneous and heterogeneous runs.

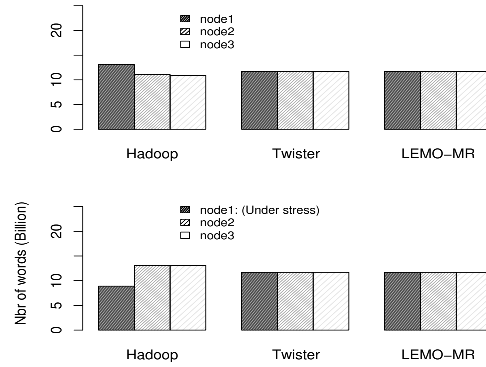

Fig. 7: Shows two graphs. The first, on top, portrays a homogeneous cluster of 3 nodes for Hadoop, Twister and LEMO-MR. All 3 nodes operate freely and are under no external load. Hadoop shows a slight load disparity between nodes, even as they operate freely. The bottom graph however depicts the same cluster, but this time with node1 under immense stress. Hadoop here, because of its speculative algorithm, can be seen shifting load to less stressed machines. Nearly 4.2 billion words in Hadoop case are shifted from the overloaded machine, to the "free" machines.

500MB per node, and LEMO-MR, 520MB per node. In our previous data-intensive test, in **Figure 1**, Twister was shown to default with 93 MB of data sent to its reducer. The limit shown in that experiment pertains to how much data each node is capable of passing from *mapper* to *reducer*, and not how much memory space each mapper or worker node disposes of for "mapping" as this particular experiment shows.

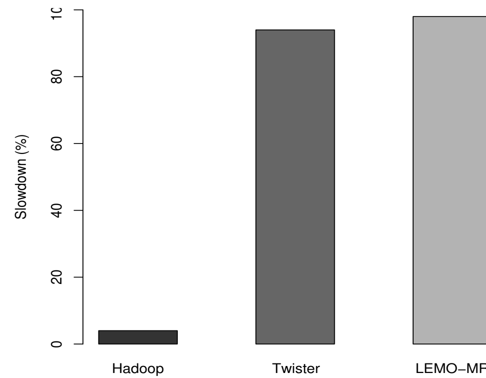

Fig. 8: This graph shows how each framework represented slowed down faced with CPU stressful conditions applied to node1 in **Figure 7**. All four cores on node1, the candidate node, were stressed to 99%-100%.

Figure 6: [15], [16] and [17] have shown Hadoop to perform rather slowly in heterogeneous environments. Cluster Heterogeneity in our tests pertains to the diverse processor and memory configurations of cluster nodes. The cumulus of cores however in both clusters (homogeneous and heterogeneous) is the same. This condition, as it was previously shown [15] [16], [17], can greatly impact MapReduce performance if mitigating design decisions are not effectively taken in a given MapReduce implementation. While it would stand to intuition for embarrassingly parallel applications, that a 60 core cluster composed of a 4 core machine, an 8 core machine and a 48 core machine would perform equally as a 60 core cluster with 3 20 core machines, previous work, and tests in **figure 6** show otherwise. In brief, the uniformity of MapReduce input chunk, and task definitions, cause the MapReduce model to be slower on heterogeneous clusters. This test simply shows how slower each framework tested is. Hadoop implements speculative execution for load-balancing purposes. Load-balancing is however not akin to heterogeneity in Hadoop's context because in the former, the homogeneous nature of the cluster makes lackluster performance on a node's part easily detected, and as a result, load is transferred. Not having a homogeneous lineup means not being able to set a standard as to what "slow" is. Slow for a 48 core machine might means very fast for a 4 core machine. As a result, load transfer in a heterogeneous cluster under Hadoop is bound to occur excessively and counter-productively, thus negatively impacting performance as shown here.

Figure 7 shows load balancing tests whereas, homogeneous machines operate first freely, followed by tests in which one node, node1 here is put under extreme stress. Hadoop implements speculative execution, and as such benefits here from the duplicate tasks running on the "free" machines. The framework then discards the late arrivals from the slow node, and moves onto reducing, thus completing its run faster, relative to its previous run. LEMO-MR, and Twister do not feature active mechanisms to deal with this condition and find themselves outstaged by Hadoop. Hadoop merely shifts 4.2 billion words to both node2 and node3 from node1, allowing it to perform better in homogeneous imbalanced clusters. In this test, conditions imposed on node1 can however be considered extreme, as all 4 cores were made busy to 99%-100%. In less intense scenarios, as might be the generic case, the disparity might be less severe between Hadoop, LEMO-MR and Twister. As **Figure 8** shows, Hadoop was slowed down by only 5% relative to its free run, versus 94% for Twister, and 98% for LEMO-MR. It is however worth mentioning that Twister and LEMO-MR still ran much faster in their free and stressed runs than Hadoop in both, even considering its mitigating algorithm. Relative to its free run, however, Hadoop performed overwhelmingly better in its stressed run vis-a-vis

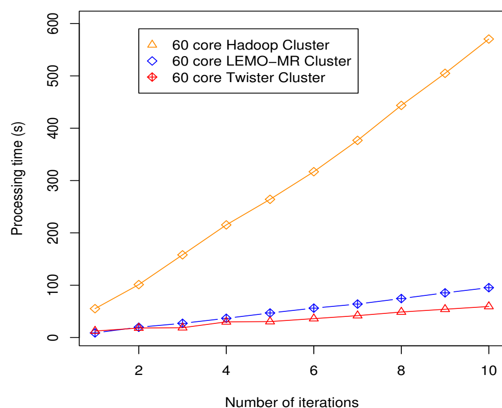

Fig. 9: Hadoop, LEMO-MR and Twister performing distributed grep in an iterative manner. Twister here benefits from its iterative nature, and departs further from Hadoop and LEMO-MR as the number of iterations applied increases.

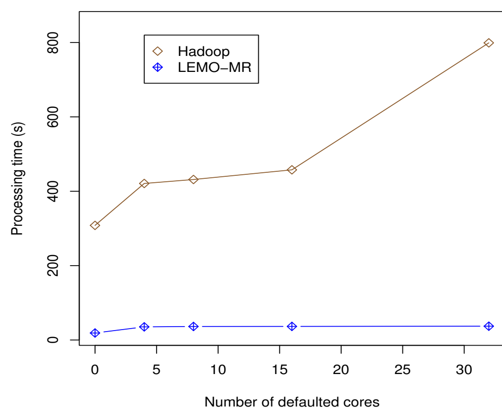

Fig. 10: This graph shows how Hadoop and LEMO-MR behave faced with defaulting or dying nodes. Both clusters start with 200 cores, and progressively, in separate runs lose 4, 8, 16, and 32 cores. Hadoop slows down much faster than LEMO-MR.

LEMO-MR and Twister.

In **Figure 9**, despite the clear advantage exhibited by Twister, it is good to note that for much larger datasets, Twister would require constantly increasing number of nodes to offset the memory limit observed in **Figure 1**.

Figure 10 shows fault-tolerance tests between Hadoop and LEMO-MR. The absence of Twister in these tests is due to the fact that Twister does not feature a node specific fault-tolerance mechanism [3], and as such killing nodes mid-processing would leave the framework to hang, as the master would wait indefinitely for the dead node to send its output for the job to be reduced and completed. In these tests, LEMO-MR takes the upper hand over Hadoop as its fault-tolerance mechanism is independent of file chunk location but rather, node availability [4]. The overhead involved for Hadoop in replicating chunks from node failures and verifying the integrity of its input structure is leading to its demise in this particular case. Note however that LEMO-MR's advantage is sanctioned by overhead incurred in providing the rescuer nodes with the input chunk processed by the faulty nodes. This one time penalty is however almost constant throughout the entire experiment.

VI. Related Work

MRBench [23] is a benchmarking tool implemented for the comparison Hadoop to relational database systems. MRBench is based on TPC-H which is used to evaluate database systems that have realistically complex queries. TPC-H queries are converted into MapReduce jobs and each job includes multiple steps of MapReduce. The output produced from each step becomes the input for the next step except for the last step which basically collects the result of the operation. [24] compares MapReduce with parallel SQL DBMS's in terms of performance and development complexity. The authors show that tested parallel DBMS's out-perform Hadoop on data-intensive analysis benchmarks while recognizing the ease of use and advanced fault-tolerance that Hadoop provides over the DBMS's. In Cogset [25], data storage is distributed over the cluster through partitioning and replications, while data access is done through traversals. In [19], Cogset is compared with Apache Hadoop and shown to be outperforming Hadoop with database specific operations. The authors highlight the reasons for their results and offer some minor modifications to Hadoop on the causes of the performance gap.

VII. Conclusions and Future Work

In this study, we identified six applications and cluster categories pertinent to MapReduce performance and tested them with three well known MapReduce variants: Hadoop [2], Twister [3] and LEMO-MR [4]. We not only provide performance relevant design details of three MapReduce implementations, but also highlight throughout our tests, the best current choice of the three frameworks for a given HPC application scenario.

- Twister is efficient for jobs requiring significant *map* side data processing and very limited *reduce* side data processing, such as CPU and memory-intensive jobs. For our test infrastructure, Twister was better than Apache Hadoop when data transfer between the Mapper and Reducer was less than 93 MB, because it uses memory based transfers as opposed to the file based transfers used by Hadoop. The 93 MB threshold can be increased slightly by allocating more memory to the worker processes, but is still limited by the maximum memory footprint of a Java process on a given node.
- Hadoop is designed to make use of the maximum disk space available to each node and hence is known to scale seamlessly for very large data sizes. LEMO-MR uses a hybrid approach starting with a memory-based transfer scheme between Map and Reduce, then progressively changing to the Hadoop model for larger data sizes.
- Twister and LEMO-MR are optimized for CPU-intensive Map and Reduce jobs compared to Hadoop, which spends extra CPU cycles on high overhead prone fault tolerance related tasks. For the matrix multiplication experiment, the CPU utilization for Twister and LEMO-MR was 93% while it was 89% for Hadoop.
- As the processing power is increased for a given application, Twister's performance improvement is best, closely

followed by LEMO-MR. Hadoop has the highest overhead per node, and as such, when nodes were increased from 8 to 64 nodes, Twister's speedup improvement was by a factor of 7.5, LEMO-MR's was 7, and Hadoop's was by a factor of 4.
- LEMO-MR has the least memory footprint allowing for 520MB of data to be processed per node, closely followed by Twister which allowed 500 MB of data to be loaded in to memory for processing, per node. Hadoop has the highest memory footprint and it allowed only 100 MB of data to be processed per node, before throwing an *out of memory* error.
- The uniformity of input chunk to each node renders Hadoop unable to make efficient use of a heterogeneous cluster. It is unable to determine what "slow" means for tasks mapped to nodes with different memory and processor configurations, and as a result load transfer operations occur excessively and inefficiently.
- In a homogeneous cluster, Hadoop's load management significantly outperforms both LEMO-MR and Twister. When random nodes experience stress, induced by our tests, Hadoop benefits from its speculative execution model which duplicates tasks and is not affected by the slow nodes.
- For applications requiring multiple iterations, Twister showed to be best. Apart from its ease in running iterative applications, compared to Hadoop and LEMO-MR, Twister also offers significant performance advantages to such applications. This is illustrated with Twister being from 2 up to 5 times faster than Hadoop and LEMO-MR.

In future work, we will release the scripts, data, and code associated with this benchmark project, and maintain an updated website including performance data for new MapReduce implementations as they become available.

VIII. Acknowledgements

This research was 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] M. P. I. Forum, "Mpi: A message-passing interface standard," 1994.
[2] Apache Hadoop. [Online]. Available: http://hadoop.apache.org
[3] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *HPDC*, 2010, pp. 810–818.
[4] 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.
[5] Octopy, Easy MapReduce for Python. [Online]. Available: http://code.google.com/p/octopy/
[6] Skynet. [Online]. Available: http://rubyforge.org/projects/skynet
[7] AMAZON, "Amazon Elastic MapReduce." [Online]. Available: http://aws.amazon.com/ec2
[8] Z. Fadika and M. Govindaraju, "Delma: Dynamically elastic mapreduce framework for cpu-intensive applications," in *CCGRID*, 2011, pp. 454–463.
[9] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 260–269.
[10] 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.
[11] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in *FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2010, pp. 93–102.
[12] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project*, 2007.
[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] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," in *IPDPS Workshops*, 2010, pp. 1–9.
[16] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI*, 2008, pp. 29–42.
[17] C. Tian, H. Zhou, Y. He, and L. Zha, "A dynamic mapreduce scheduler for heterogeneous workloads," in *GCC '09: Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 218–224. [Online]. Available: http://dx.doi.org/10.1109/GCC.2009.19
[18] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[19] S. V. Valvag, D. Johansen, and A. Kvalnes, "Cogset vs. hadoop: Measurements and analysis," *Cloud Computing Technology and Science, IEEE International Conference on*, vol. 0, pp. 768–775, 2010.
[20] 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.
[21] S. Pallickara and G. Fox, "Naradabrokering: A distributed middleware framework and architecture for enabling durable peer-to-peer grids," in *in Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003, Rio Janeiro*, 2003, pp. 41–61.
[22] Apache ActiveMQ. [Online]. Available: http://activemq.apache.org/
[23] K. Kim, K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom, "Mrbench: A benchmark for mapreduce framework," in *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 11–18. [Online]. Available: http://portal.acm.org/citation.cfm?id=1491261.1491574
[24] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2009, pp. 165–178. [Online]. Available: http://database.cs.brown.edu/sigmod09/benchmarks-sigmod09.pdf
[25] S. V. Valvag and D. Johansen, "Cogset: A unified engine for reliable storage and parallel processing," in *Proceedings of the 2009 Sixth IFIP International Conference on Network and Parallel Computing*, ser. NPC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 174–181. [Online]. Available: http://dx.doi.org/10.1109/NPC.2009.23