Towards a MapReduce Application Performance Model

Jared Gray
Dept. of Mathematics and Computer Science
Denison University
gray.jared.r@gmail.com

Dr. Thomas C. Bressoud
Dept. of Mathematics and Computer Science
Denison University
bressoud@denison.edu

ABSTRACT

In the modern age, our ability to generate large data sets far outpaces our capacity for analyzing them. Google's proposed solution to this fundamental problem – the MapReduce paradigm and runtime system – has recently gained traction in the scientific and "big data" industries. However, the performance characteristics of MapReduce are not well known. This paper builds on the efforts of prior research to more accurately characterize and model the performance of MapReduce applications on large-scale distributed systems.

1. INTRODUCTION

Invented by Google in 2004, MapReduce [3] is a programming runtime and software framework for distributed computing. Using the runtime, a programmer can take an embarrassingly parallel program and distribute its execution across a large cluster of computers. MapReduce enjoys a positive reputation for ease of use and general robustness, as the details of task scheduling, data partitioning, and fault tolerance are all automatically handled by the runtime system. Consequently, MapReduce has become increasingly popular for scientific computing applications and "big data" processing in recent years.

Despite its popularity, however, not a great deal is known about the performance characteristics of MapReduce, especially in the presence of machine failures. One might expect that a MapReduce application's execution time scales linearly with the number of cluster computers used for computation. However, such assumptions are often not borne out by performance benchmarks. In fact, the debate over which factors matter most to MapReduce performance has yet to be settled, due in no small part to the complex interplay of networking, cluster hardware, and operating systems. Making matters worse for analysis, there are an immense number of parameters that a user may specify to the runtime to change its behavior.

In the literature that does exist on the subject, much performance analysis of MapReduce hinges on failure-free operation assumptions. These assumptions are unrealistic in practice and are an important deficiency, for failures can cause significant delays in execution. More theoretical research often fails to accurately estimate the failure rate of computing nodes and thus the time necessary to complete execution of an application.

We addresses the deficiencies discussed above in three ways. First, through the use of our own MapReduce microbenchmarks and the results of prior work, we aim to more accurately characterize the performance of MapReduce. Second, we develop a new, more robust model for MapReduce application performance that includes provisions for machine failures. Lastly, our new model drives a discrete event simulation for MapReduce, the performance characteristics of which can be investigated thoroughly in future work. This simulation utilizes empirical cluster failure data from Los Alamos National Lab (LANL) [5] as a basis for failure injection.

2. BACKGROUND

2.1 Application Phases

A MapReduce application consists of three primary phases. In order of execution, these are: (1) *map*, (2) *shuffle and sort*, and (3) *reduce*. Prior to running an application, the user must copy all relevant input data to a distributed file system (DFS), which unifies and makes accessible the local storage of computers across a network. It is common practice to utilize the local storage of compute nodes

Proceedings of the 2012 Midstates Conference on Undergraduate Research in Computer Science and Mathematics

for DFS operations, for this allows computations to co-reside with their requisite input data. We discuss the three phases below and present a graphical representation of a MapReduce application in Figure 1.

2.1.1 Map

In the *map* phase, the MapReduce runtime hands a partition of the input data – called a "split" in MapReduce nomenclature – to each compute node participant of the map phase. We call these participants "mappers". The mappers then take their assigned splits and interpret them as a series of key-value pairs. For example, it is common for a mapper to parse an input file, creating a key-value pair for every line of text in the file; the keys are assigned the text comprising a line and their corresponding values are set to the line number at which that text appears. After the input data has been parsed, mappers run a user-specified MAP algorithm, which deterministically transforms the input key-value pairs into a set of intermediate key-value pairs. After all mappers run the MAP algorithm on their assigned input data and save the intermediate data output to local storage, the *map* phase is finished.

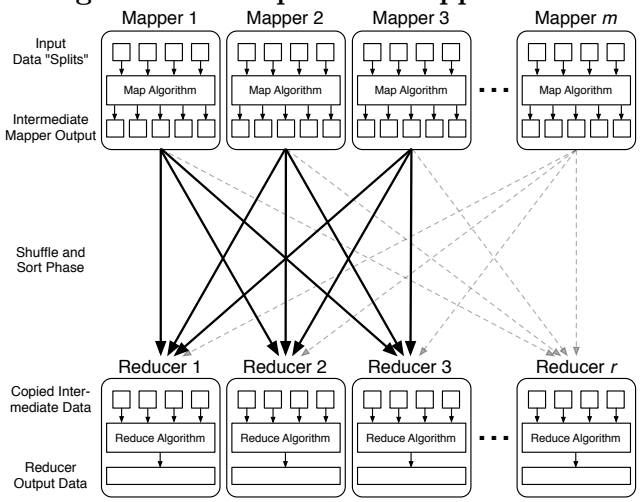
2.1.2 Shuffle and Sort

Next, in *shuffle and sort*, intermediate data is sorted and redistributed across cluster nodes. This phase technically initiates before the *map* phase completes and terminates before the *reduce* phase begins. As each mapper completes its assigned task, it partitions the intermediate keyspace into a series of key ranges and sorts each partition [8, p.175-176]. Then, compute nodes constitutive of the *reduce* phase – "reducers" – are assigned a specific range and copy the associated key-value pairs to their locally attached storage. Note that reducers will likely have to obtain key-value pairs from multiple mappers, as an individual mapper can potentially generate the full spectrum of intermediate keys. The reducers' next task is to sort their copies of the intermediate data. Given that reducers can only copy data as it becomes available, they must wait until all mappers complete their respective tasks before they can finish copying. When all reducers have copied and sorted the data pertaining to their assigned key range, the *shuffle and sort* is done.

2.1.3 Reduce

Finally, in the *reduce* phase, reducers take their sorted intermediate key-value pairs and transform

Figure 1: A MapReduce Application[1]



them into output key-value pairs. This transformation occurs according to another deterministic, user-specified algorithm, the REDUCE function. Following the completion of all REDUCE-related computation, reducers save their output back to the DFS. After all reducers finish saving, the MapReduce application is complete. A programmer can manually analyze these output files or conveniently use them as input for a subsequent MapReduce application.

2.2 WordCount Example Application

In this section, we describe a hypothetical MapReduce application that computes a word frequency histogram for a large library of text. The input data for this application consists of a series of text files. MapReduce will emit a {`line of text, line number`} pair for each line in every file, defining the input for our MAP function. We desire output data in the form of {`word, number of occurrences`} pairs.

First, in the MAP algorithm, we will count the occurrences of words, emitting a {`word, number of occurrences`} pair for each key-value pair of input data. These emitted pairs collectively constitute our intermediate data. Note that several mappers might have come across common words – such as "the", "a", and "to" – in their input data. If this is the case, the intermediate data will contain multiple key-value pairs with the same key, where

[1]The dichotomy between solid and dashed lines has no significance except to draw attention to the complete k-partite graph on two sets of 3 vertices, visualizing the network activity between mappers [1..3] and reducers [1..3].

each pair represents a unique, mapper-local count of words.

As the *map* phase nears completion, the MapReduce runtime will perform the *shuffle and sort* automatically, reorganizing the intermediate data and distributing its key-value pairs to our reducers. Based on the way in which *shuffle and sort* partitions the intermediate keyspace, all key-value pairs of the intermediate data with the same key are guaranteed to reside on the same reducer.

In order to compute our histogram, we will need to consolidate the mappers' local counts of words. The REDUCE algorithm can thus work on a partition of intermediate data, aggregating the various mapper-local word counts. This operation will produce a more consolidated set of key-value pairs of the form {`word, number of occurrences`}. These pairs define the output of our MapReduce application and represent the word frequency histogram we sought to create.

2.3 Fault Tolerance

So far, we have seen that MapReduce automatically performs data partitioning and task scheduling behind the scenes, removing a great deal of burden from programmers. All of this intelligence might be for naught, however, if MapReduce could not gracefully handle machine failures. The fault tolerance mechanism in use by the MapReduce runtime is one of re-execution. Should an individual machine fail – becoming inaccessible over the network or generally unresponsive – all map or reduce tasks running on that machine are re-executed on other cluster nodes. This mechanism is effective because the MAP and REDUCE functions are deterministic; their output should be identical given the same input data, regardless of on which machine they are executed.

3. PRIOR WORK

Previous research on MapReduce performance has yielded a number of parameters that can dramatically affect the time required for a MapReduce application to run [10] [4]. Of particular interest to us, Wottrich [9] determined that the sizes of the input, intermediate, and output data were vital considerations for application performance. Additionally, he examined the cost of managing many map and reduce tasks, an overhead which can outweigh the performance benefits derived from dividing an application into smaller parts. Unlike Wottrich, Bressoud and Kozuch [2] focused on the performance characteristics of MapReduce in the presence of machine failures. To do so, they designed and utilized a discrete event simulation of the MapReduce runtime. One of the primary goals of this project is to take that simulator, CFTsim (**C**luster **F**ault **T**olerance Simulator), and improve its underlying model of MapReduce, thus increasing its accuracy in reflecting real-world MapReduce performance.

4. EQUIPMENT

4.1 Cluster Arrangement

To run MapReduce, this research project utilizes a Beowulf cluster consisting of 35 computers. Each computer runs a 32-bit installation of Ubuntu Server 12.04 Linux. Notably, our cluster contains a heterogeneous mix of hardware. Fifteen hardware-identical computers run on older Intel Core 2 Duo processors, while a more modern set of 14 computers have Intel Core i7 2600s installed. In our testing, we determined that there exist subtle differences in hard drive and network interface card performance between the two sets of machines. As such, we carefully tailor our MapReduce benchmarks to mitigate the impact of these differences in performance.

4.2 Network

Our cluster computers are networked together by means of a fully managed gigabit ethernet switch by Cisco. All network interfaces on the switch and the compute nodes are capable of gigabit speed and full duplex connectivity. In an all-pairs network bandwidth benchmark, we found that the switch was minimally capable of delivering an average of 694 megabits per second to each cluster node.

4.3 Hadoop

Each of our cluster nodes runs Hadoop [1], the Apache Software Foundation's open-source implementation of Google's MapReduce paradigm and distributed file system. We configured our cluster to utilize 34 compute nodes for executing map and reduce tasks, along with a single "master" node in charge of managing task scheduling and the Hadoop Distributed File System [6]. The Hadoop package in use by our cluster is version 1.0.1, which was the current stable release at the onset of this project. Note that future releases of Hadoop will incur significant changes to the runtime. Thus, the discussion of MapReduce mechanics above applies to our current Hadoop installation, but may not accurately describe future (or past) versions of the runtime.

5. THE BENCHMARKS

Before we could attempt improving on the model for MapReduce, we needed to establish the impact of various performance factors on MapReduce application running time, using the results discussed in Section 3 as a starting point for inquiry. Accordingly, we developed three MapReduce benchmarks which isolate the effects of specific parameters on overall application performance. Combined with Wottrich's benchmarks for input, output, and intermediate data sizes, we had a number of utilities with which to more accurately characterize MapReduce application performance and use as the basis for a new model.

We keep the aggregate sizes of input, intermediate, and output data trivially small (< 1 MB) for each of our benchmarks. Additionally, we define a constant amount of "work" performed in our homebrew Map and Reduce functions to a specific time interval using a busy loop around the system clock. Thus, we are able to isolate the effect of each fixed-time algorithm on overall MapReduce application performance. We synchronize the system clocks of all compute nodes using a network time protocol server on the local network, which updates the compute nodes on an hourly basis. All benchmarks measure the total time elapsed from application initialization to completion.

The first benchmark, simply named *mrbench*, aims to determine the overhead of creating many map and reduce tasks.[2] This program can be run in one of two modes, one for benchmarking map task overhead and another for reduce task overhead. If the user runs in the Map-benchmarking mode, a trivial REDUCE function is used. Should the user desire a Reduce-side benchmark, on the other hand, the program uses a trivial MAP function. The benchmark works by forcing compute nodes to run multiple non-trivial tasks at the same time. At first, each node executes a single, fixed-work map or reduce task. In the next iteration, all nodes are assigned two such asks and run them both concurrently. This continues for several dozen iterations. Because we have fixed the MAP or REDUCE algorithm running time, we are able to isolate the effect of map and reduce task overhead on application performance using this benchmark.

The next benchmark, *mrbench-waves*, works very similarly to the previous program, with one major difference. Unlike *mrbench*, which causes each cluster node to execute all assigned non-trivial map or reduce tasks concurrently, this benchmark forces every compute node to run only one such task at a time. Due to this constraint, at most n map/reduce tasks can be running simultaneously across the cluster, where n is the number of mappers or reducers running the non-trivial algorithm. Further, because the non-trivial algorithm's running time is fixed, each set of tasks currently running on the cluster initiates and completes at roughly the same time, causing "waves" of execution to occur.

Similar to *mrbench*, *mrbench-waves* operates by increasing the number of tasks handled by the MapReduce runtime. In the first iteration of *mrbench-waves*, a non-trivial task is executed once per node. In the second iteration, each node runs two tasks back-to-back; the number of tasks likewise increases for the remaining iterations. We should expect each iteration of this benchmark to take successively longer to run, proportional to the product of (a) the quantity of waves and (b) the MAP or REDUCE algorithm's fixed running time.

Our last benchmark, *mrbench-files*, examines the performance impact of opening multiple files on the DFS. In *Pro Hadoop*, Jason Venner [7, p.168] advises Hadoop developers to avoid spreading out the input or output in a MapReduce application over multiple files, as this can cause severe performance losses. This benchmark tests Venner's claim by creating multiple trivially-sized input or output files and using a fixed-time MAP or REDUCE algorithm. A single, trivial reduce task is used when benchmarking input files, a trivial mapper for the output test. Borrowing heavily from *mrbench-waves*, this benchmark simply accesses or creates one file per map/reduce task per wave, increasing the number of waves iteratively. The similarity between this benchmark and *mrbench-waves* is intentional, for it makes the results of this test directly comparable with those of *mrbench-waves*.

6. RESULTS

We present the results of *mrbench* benchmark in Figures 2 and 3. Here, the non-trivial MAP or REDUCE algorithm is fixed to a 10-minute run time. Again, because all tasks are allowed to run concurrently and the amount of work for each task's MAP or REDUCE algorithm is constant, we should expect the addition of tasks to incur no increase nor decrease in execution time, supposing no signifi-

[2]This benchmark is our own creation, different from the *mrbench* program included with the Hadoop package.

Figure 2: *mrbench* Map-Side Results

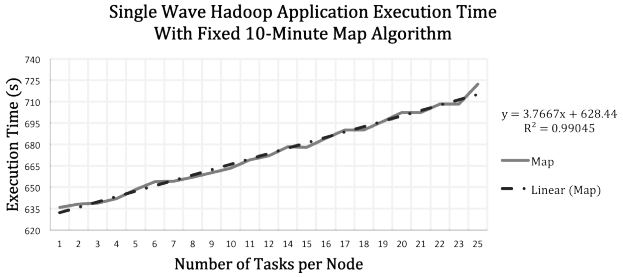


Figure 3: *mrbench* Reduce-Side Results

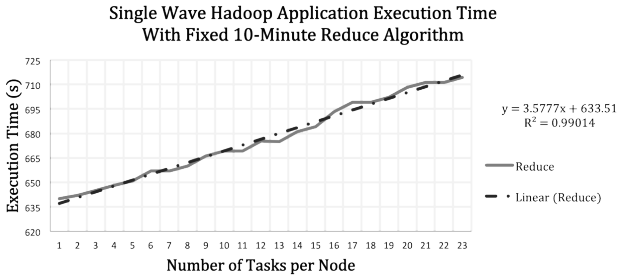


Figure 4: *mrbench-waves* Results

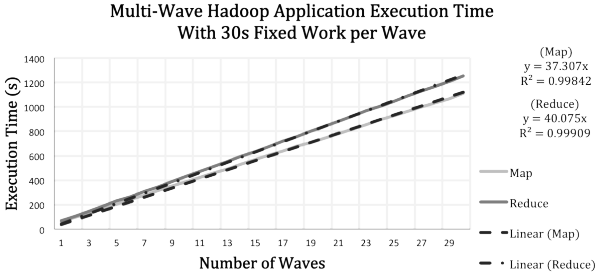
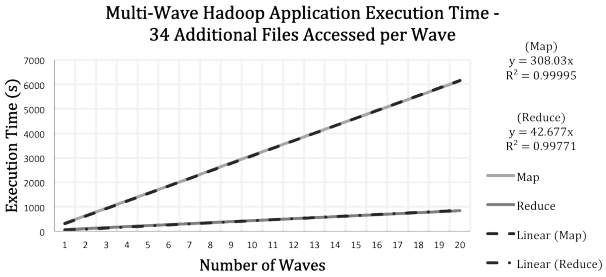


Figure 5: *mrbench-files* Results



cant overhead exists for map/reduce task creation. However – based on the line of best fit – we observe a clear trend of 3.77s overhead for the Map plot, 3.58s for the Reduce-side benchmark. Because we have 34 compute nodes, each time we increase the number of tasks per node by one, we are adding 34 map or reduce tasks. Then the map task overhead is $\frac{3.77s}{34} \approx 0.111s$ and reduce task overhead is $\frac{3.58s}{34} \approx 0.105s$.

These overheads signify an important component of MapReduce performance. Typical MapReduce clusters are comprised of hundreds or thousands of computers. Most commodity machines today can also efficiently run multiple map or reduce tasks in parallel, thanks to the prevalence of multi-core systems. It would be fairly typical, then, for a MapReduce cluster to be configured for hundreds or thousands of map and/or reduce tasks. Moreover, with such large clusters, the possibility for individual machine failures escalates, leading to further performance costs down the road. Finally, with the pervasiveness of "small" MapReduce jobs that take only a few hours to run, this overhead can be a significant component of overall execution time.

The *mrbench-waves* results appear in Figure 4. For this benchmark, the non-trivial algorithm has a fixed 30-second running time. A null hypothesis for this benchmark predicts that each successive run takes an additional 30s over the previous one. However, we see that each successive Map run takes roughly 37.3s longer to complete than its predecessor and each Reduce run takes an additional 40.1s to complete. This result indicates that the map task overhead for each wave is around 7.3 seconds, and reduce task overhead is approximately 10.1 seconds. As each wave is comprised of 34 tasks in our case, this amounts to approximately 0.215 seconds and 0.297 seconds and of overhead per map task and reduce task, respectively.

Interestingly, although this benchmark also aims to capture the overhead of map/reduce task creation and maintenance, its results differ significantly from those of the previous test. Both map and reduce task overhead, as captured by this benchmark, exceed the overheads measured by *mrbench*. This result makes intuitive sense, as there is some management overhead inherent in performing job scheduling. When an individual task completes, the master node must free the compute node to run another task. Then, if there are multiple tasks yet to be assigned, the master node must also per-

form scheduling to determine the ideal computer on which to run the next task.[3] In *mrbench*, all tasks complete at virtually the same time – minimizing the overheads for scheduling and task management. However, in *mrbench-waves*, tasks complete one wave at a time. Because each wave completes at the same time, the master node is likely overwhelmed with such scheduling tasks, and a significant overhead results.

The other difference in measured performance between the two benchmarks pertains to the relative overhead of map and reduce tasks. Unlike *mrbench*, *mrbench-waves* indicates that reduce task overhead is greater than map task overhead, as shown by the steeper line of best fit for the result data. Unfortunately, we have not been able to come up with as exact an explanation for why this is the case. It is possible that software changes on our cluster between the run of *mrbench* and *mrbench-waves* are to blame, a scenario which we could rule out by re-running the benchmarks. There might also be an issue related to the MAP or REDUCE algorithm code or the Java interpreter's optimization thereof, but nothing apparent stands out to us. Because most real-world MapReduce applications will exhibit this wave-type behavior, however, we believe the results of *mrbench-waves* should be preferentially used over those of *mrbench*.

The *mrbench-files* benchmark also yielded interesting results (Figure 5). As mentioned above, much of this benchmark's source code was derived from *mrbench-waves*. Due to the additional work of creating files, the *reduce* phase takes an additional 2.6s per wave, a 20% increase in overhead. The *map*-side results, however, are even more dramatic. Map task overhead is nearly 8.5 times greater here than in the previous benchmark, demonstrating that spreading out the input to a MapReduce application over multiple files can degrade performance significantly. Hence, we are able to confirm that the number of input/output files accessed via the DFS has a significant impact on MapReduce application execution time.

7. MODELING

Having identified a number of parameters impacting MapReduce application performance, we propose an enriched version of the Bressoud-Kozuch model for MapReduce. Our new model, much like the original, operates on two different yet overlapping levels. The first level is that of the map and reduce task instances. At this level, we examine the state transitions that individual tasks must progress through on their way to completion. The second level is that of the MapReduce application itself. In this latter level, we track the progress of the application through the *map*, *shuffle and sort*, and *reduce* phases.

At the task instance level of our new model, displayed in Figures 6 and 7, there are separate finite state machines for map and reduce task instances, necessary to denote the difference in behavior between them. These task instance models progress through more precise states than before; map tasks transition through READ, WORK, WRITE, and COMPLETE stages, while reduce tasks have an additional SORT state between READ and WORK. The amount of time spent in each phase is denoted by a series of parameters supplied by the user to the simulation. Should no failures occur in a given time interval, the task will move along the solid arrow to the next state. If, on the other

Figure 6: Map Task Instance

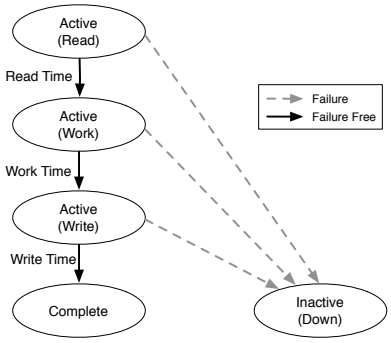
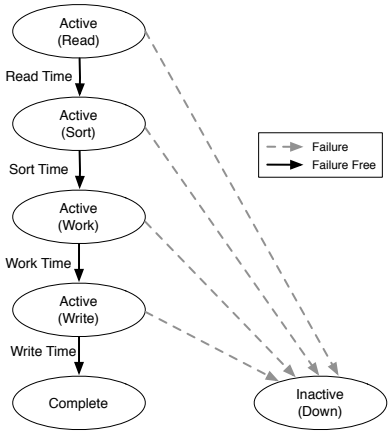


Figure 7: Reduce Task Instance



[3]The proportion of task input data local to the machine is the most important factor in scheduling.

hand, the machine on which the task is executing does fail, the task instance will itself fail and be re-executed later, possibly on a different machine.

The model for MapReduce applications has also grown more complex than its predecessor. The new model, shown in Figure 8, progresses through state transitions realistically; we must wait for all mappers to complete writing data to local disk before reducers can begin making a copy. When all reducers have completed copying, they begin executing the REDUCE algorithm on their input key/value pairs. After all reducers have completed running the algorithm and writing results to the DFS, all map and reduce tasks are complete and the application is done.

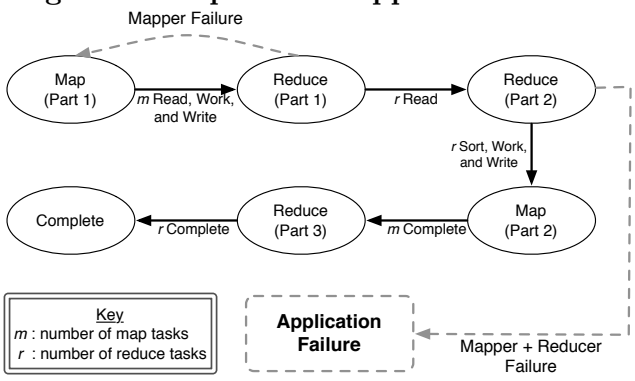
The failure behavior described by the new model follows a uniformly distributed key assumption: we assume that each mapper generates intermediate data such that every reducer copies at least one key-value pair from each mapper. Thus, if a mapper should fail before the read portion of *shuffle and sort* – "Reduce (Part 1)" in Figure 8 – is complete, that mapper must be re-executed before the application can continue. In the simulation, all reducers are subsequently re-executed to ensure the availability of intermediate data.

Another interesting component of MapReduce behavior is that a mapper failure does not result in re-execution after the *shuffle and sort* is complete. This can be problematic if a reducer failure follows a mapper failure anytime in the *reduce* phase. Assuming uniform key distribution of intermediate data and no DFS data replication, this situation would result in a permanent loss of intermediate data. Unless the user specifies that the MapReduce application should compute a partial result based on the remaining data, the application will fail. Our model reflects the default case.

8. CONCLUSIONS

This project contributes to our understanding of MapReduce application performance on large-scale distributed systems. First, our microbenchmarks isolate important performance factors affecting MapReduce. With the *mrbench* and *mrbench-waves* benchmarks, we observe that the overheads incurred by creating and maintaining multiple map and reduce tasks is significant. Furthermore, the *mrbench-files* benchmark results indicate a strong decrease in performance as we increase the number of files managed by the DFS. Extrapolating these results,

Figure 8: MapReduce Application Model



we present a more robust model for MapReduce, which takes machine failures into account. Finally, we provide a discrete event simulation for MapReduce based on real-world failure data. This simulation supplies the vital groundwork for future statistical analysis of MapReduce performance on large-scale clusters.

9. FUTURE DIRECTIONS

A significant amount of work lies ahead in analyzing the performance of MapReduce applications via the new CFTsim. This analysis should examine performance as we scale the numbers of map/reduce tasks, compute nodes, and processor cores. A future simulation might also consider the role of networking in application performance, as well as the overheads and redundancy-related advantages of distributed file system operations. Lastly, there is work to be done in optimizing the efficiency of the simulator itself. Much of its complexity could be eliminated by removing its checkpointing and rollback recovery components.

10. ACKNOWLEDGMENTS

We gratefully acknowledge the generous support of the Denison Undergraduate Research Foundation, which made this project possible.

I would like to thank Dr. Thomas Bressoud for advising me during this project. His advice and guidance were invaluable.

11. REFERENCES

[1] Apache Hadoop. `http://hadoop.apache.org`, retrieved 1 November 2012.

[2] Thomas C. Bressoud and Michael Kozuch. "Cluster Fault-Tolerance: An Experimental

Evaluation of Checkpointing and MapReduce Through Simulation". In *Proceedings of the 2009 IEEE International Conference on Cluster Computing*. IEEE, 2009.

[3] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation - Volume 6*, OSDI'04. USENIX Association, 2004.

[4] Florin Dinu and T.S. Eugene Ng. "Understanding the Effects and Implications of Compute Node Related Failures in Hadoop". In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12. ACM, 2012.

[5] Los Alamos National Lab (LANL). "Operational Data to Support and Enable Computer Science Research". `http://institutes.lanl.gov/data/fdata/`, retrieved 1 November 2012.

[6] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System". In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10. IEEE Computer Society, 2010.

[7] J. Venner. *Pro Hadoop (Expert's Voice in Open Source)*. Apress, New York, NY, 2009.

[8] T. White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale*. O'Reilly, Sebastopol, CA, 2nd edition, 2011.

[9] K. Wottrich and T. Bressoud. "The Performance Characteristics of MapReduce Applications on Scalable Clusters". In *Proceedings of the Midstates Conference for Undergraduate Research in Computer Science and Mathematics (MCURCSM)*, 2011.

[10] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. "Improving MapReduce Performance in Heterogeneous Environments". In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08. USENIX Association, 2008.