9-2013

JUMMP: Job Uninterrupted Maneuverable MapReduce Platform

William Clay Moody
Clemson University, wcm@clemson.edu

Linh B. Ngo
Clemson University, lngo@clemson.edu

Edward Duffy
Clemson University, eduffy@clemson.edu

Amy Apon
Clemson University, aapon@clemson.edu

JUMMP: Job Uninterrupted Maneuverable MapReduce Platform

William Clay Moody*, Linh Bao Ngo*, Edward Duffy†, Amy Apon*
Computer Science Division of the School of Computing *
Clemson Computing and Information Technology †
Clemson University
Clemson, South Carolina
{wcm,lngo,eduffy,aapon}@clemson.edu

***Abstract*—In this paper, we present JUMMP, the Job Uninterrupted Maneuverable MapReduce Platform, an automated scheduling platform that provides a customized Hadoop environment within a batch-scheduled cluster environment. JUMMP enables an interactive pseudo-persistent MapReduce platform within the existing administrative structure of an academic high performance computing center by "jumping" between nodes with minimal administrative effort. Jumping is implemented by the synchronization of stopping and starting daemon processes on different nodes in the cluster. Our experimental evaluation shows that JUMMP can be as efficient as a persistent Hadoop cluster on dedicated computing resources, depending on the jump time. Additionally, we show that the cluster remains stable, with good performance, in the presence of jumps that occur as frequently as the average length of reduce tasks of the currently executing MapReduce job. JUMMP provides an attractive solution to academic institutions that desire to integrate Hadoop into their current computing environment within their financial, technical, and administrative constraints.**

Keywords*—*MapReduce; Hadoop; academic cluster; maneuverable applications; jummp;

I. Introduction

MapReduce [1] is an important programming paradigm for today's data-intensive computing problems. The open source Apache Hadoop [2] project is the *de-facto* baseline implementation of a MapReduce framework from which a rich and complex software ecosystem has evolved. Many industrial and governmental organizations have built large-scale production data centers with dedicated computing resources for Hadoop clusters to support their big data and scientific computation workloads. This rapid evolution and adoption of Hadoop [3] introduces challenges to system administrators in offering this service while maintaining a stable production environment. This is especially challenging in a typical centralized academic research environment where the hardware and software infrastructures are designed to accommodate a wide variety of research applications within a set of financial, technical, and administrative constraints. To address this problem, we introduce the Job Uninterrupted Maneuverable MapReduce Platform. JUMMP is an automated scheduling platform that enables the integration of Hadoop into the existing large scale computational infrastructure to support high availability and continuous computing for research and education.

978-1-4799-0898-1/13/$31.00

The strength of Hadoop comes from its design characteristics that bring computation to data stored on large local hard drives to take advantage of data locality instead of having to transfer data across the network, and to guarantee job completion in the event of frequent failures. This comes at a cost of extra computing cycles due to additional communication and metadata overhead to support very large scale transparent parallelization [4]. As a result, a typical Hadoop cluster consists of computing nodes with multiple terabyte-sized local storage [5]. Standard practice for architectural design of centralized, shared academic research environments usually places the computing components and the storage components into separate clusters. The local storage on the computing components is temporary and measured in just a few hundreds of gigabytes. Provisioning Hadoop as a separate stand-alone cluster requires the additional acquisition of new hardware, new rack placements, and additional power and cooling cost. On the other hand, the setup of a dynamic Hadoop environment is limited to standard resource scheduling policies in a shared computing environment. Examples of such policies are the maximum number of resources that can be reserved, or the maximum amount of time that a computing node can be reserved. This places a limit on the capability of Hadoop-based programs within such an environment.

Our work with the Job Uninterrupted Maneuverable MapReduce Platform provides the following contributions:

- A platform that enables the integration of Hadoop into an existing large scale academic computational infrastructure,
- Automated scheduling mechanisms that facilitate an arbitrarily long run time of MapReduce applications within the administrative constraints of a shared environment and with minimal interactions required from system administrators, and
- Experimental results that demonstrate that jumps can occur as frequently as the average length of the reduce tasks of the executing MapReduce jobs, with only modest impact to performance.

The remainder of the paper is organized as follows. Section II describes user and environmental considerations that motivate the design of the platform. Section III explains the architecture of the platform and discusses different design

trade-offs. Next, we study the performance of Hadoop running with JUMMP versus a dedicated environment in Section IV. We discuss related work in Section V, and conclude the paper with Section VI.

II. Motivation

In this section, we describe user and environmental considerations that motivate the JUMMP implementation. These considerations are observed in the typical shared high performance computing environment found in academic institutions.

A. User Considerations

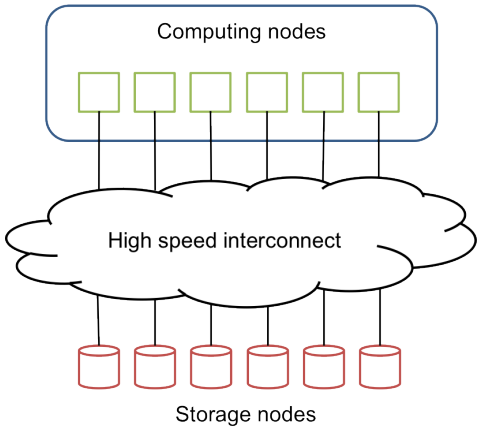
In an academic environment, we can classify the usage of Hadoop into three different categories. The first category includes research applications that use Hadoop MapReduce as a tool for research purposes. These projects can either use MapReduce programs exclusively or use MapReduce programs as part of a larger workflow in a programming framework. The researchers may spend some time developing MapReduce programs and other necessary components and then focus on executing the programs to achieve the final results. As these are research applications, researchers will typically alternate between running the computations and analyzing the produced outputs.

The second category is the study of the Hadoop MapReduce ecosystem itself. This includes studies of Hadoop MapReduce under different hardware and software configurations, development of improvements to Hadoop MapReduce, and implementations of different alternative parallel frameworks to Hadoop MapReduce. While these projects do not usually use as many computing hours as the first category, they need to frequently test different setups of the Hadoop cluster for performance analysis purposes. This testing of dynamic execution environments for Hadoop is difficult to do in most dedicated production facilities.

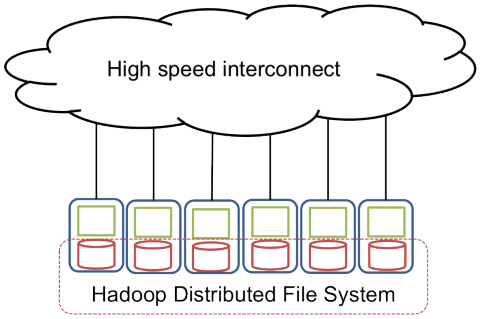
The third category includes small projects such as those submitted by students doing course projects. While these projects usually have short run times and use small data, the nature of the students' behavior can lead to unintended issues. In our experiences teaching Hadoop MapReduce during the Fall 2012 and Spring 2013 semesters, we observed multiple problems, such as crashing of the Hadoop MapReduce core processes, corruption of data on the cluster's HDFS, and overloading of the cluster as students rush to complete the work before deadlines.

With these categories, we have the following design objectives:

- Support guaranteed execution of MapReduce programs until completion regardless of run time (currently, there is no check-pointing mechanism for MapReduce jobs),
- Allow reconfiguration of Hadoop environment settings such as block size, number of map tasks, and number of reduce tasks, for different performance evaluation experiments,



(a) A typical HPC cluster with compute nodes separated from storage nodes



(b) A Hadoop cluster with storage on the compute nodes for data locality

Fig. 1: Comparing a design choice for computing and storage placement between typical HPC cluster and Hadoop cluster

- Facilitate isolated execution of MapReduce programs, and
- Support rapid movement to different hardware when impending failures are detected on current hardware.

B. Environment Considerations

While considering the needs of the users, we also seek to address other constraints of a typical academic high performance computing environment. The first constraint is cost. The costs of purchasing new computing hardware, building new floor space, and maintaining power make it difficult for many academic computing centers to provision dedicated resources for Hadoop.

A second constraint is the difference in technology between Hadoop MapReduce and a typical high performance computing cluster. Standard accepted practice for building high performance computing clusters (HPC) is to separate compute nodes from storage nodes. The compute nodes handle CPU-intensive parts of parallel applications while the storage nodes handle I/O demands of the applications, typically through a parallel file system [6]. Parallelization mechanisms are facilitated by specialized libraries at run time (e.g. MPI libraries for C-based applications), and data movement between computation and storage is done through a high-speed

interconnect. While Hadoop MapReduce also relies on specialized libraries to facilitate parallelism, its performance comes from the data locality enabled by the Hadoop Distributed File System (HDFS). HDFS divides data files into equally sized blocks and stores them on the local hard disks of compute nodes. Instead of transferring data through the network, the computation processes are spawned locally and assigned to individual blocks. This difference in computation and storage placement is illustrated in Figure 1.

A third constraint is the support of the run time environment of Hadoop. Hadoop maintains two permanent Java daemon processes, called the DataNode and the Task Tracker. These process are on all of the compute nodes at all times in order to handle both HDFS and MapReduce. The continous execution of dedicated persistent Java virtual machines makes Hadoop unsuited for a centralized installation on an HPC cluster where users also frequently run non-Hadoop applications.

Our approach is to provision Hadoop as a dynamic execution environment that can be set up, executed, and shut down on demand by a user. This is feasible from user space since Hadoop's core Java processes do not require any root-level privileges to execute and communicate. However, scheduling dynamic setups of Hadoop environments creates overhead due to run-time configuration, data staging, output write, and shutdown of the environment. Related work shows that for data-intensive applications, the total overhead could be as high as the execution time itself [7]. To make dynamic provisioning effective, there must be a mechanism that allows dynamic Hadoop environments to reserve the resources beyond the standard policy of per-user limits in a semi-persistent manner during the computation phases of application execution. An obvious solution to this problem, permanently assigning a set of compute nodes to specific users, places additional responsibilities on the administrators such as how to set a scheduling policy for users who want to execute both Hadoop and non-Hadoop applications, and how to make sure that the allocations of computing resources are fair.

Our goal is to facilitate the setup of user-controlled dynamic Hadoop environments that execute within existing scheduling policies including resource limitation, maximum walltime, and priority preemption, without administrative intervention.

III. Architectural Design

A. Hadoop Overview

Hadoop MapReduce is a network-based parallel programming paradigm for implementing data computation over big data using a cluster of commodity computer systems. The network file system for the cluster is the Hadoop Distributed File System (HDFS). This file system is composed of a single centralized management node, *NameNode*, which maintains all the metadata for the cluster along with multiple storage nodes, *DataNodes*, which contain all of the data in large block sizes (default 64 MB). Large files are divided into blocks and the data blocks are replicated across the cluster to provide redundancy.

The MapReduce computation model includes a single central management node, *JobTracker*, and multiple computation nodes, *TaskTrackers*. The JobTracker is responsible for delegating the specific map and reduce task for a submitted MapReduce job to a subset of the TaskTrackers in the cluster. The JobTracker further monitors the status of the TaskTrackers to ensure redundancy and timely completion of the job. A single TaskTracker can be assigned both map and reduce tasks within the same job. DataNodes and TaskTrackers exist on the same physical computing system, thus providing the computation and storage integration that is critical to the performance of MapReduce.

A MapReduce job is made up of multiple map tasks and multiple reduce tasks. A mapper takes an input set of data in a key-value pair (K, V) and outputs an intermediate key-value pair (K', V'). The input to the reducer is an intermediate key and the set of intermediate values for that key $(K', \{V'_1, V'_2, V'_3, ...\}$ from all the mappers across the cluster. The reducer performs some computation on the set of intermediate values and produces the final output value for each key for which it was responsible for reducing. The entire possible set of intermediate key values are partitioned and each partition is assigned to a reducer. During the shuffle phase between map and reduce, a reducer pulls its respective partition from each mapper before beginning the reduce computation.

B. Portable Batch System

The Portable Batch System, PBS, was developed by the Numerical Aerodynamic Simulation Facility at NASA Ames Research Center in the 1990s [8]. PBS provides an external batch scheduler for execution of shared supercomputing resources. PBS maintains different priority job queues with designed authorized users and hardware. This allows administrators to allow some jobs to be given preferential scheduling over other jobs and to limit the runtime of user submitted jobs. Users submitting jobs to PBS are allowed to specify the number of computing nodes, the number of processors, and memory needed on each computing resource, among other options.

PBS jobs can be interactive or can run in batch mode, executing designed programs or scripts that are provided as part of the job submission. Typical parallel programming models involve the creation of a set of parallel processes, a designated set of data, and the location for storing the output. Jobs are submitted to a queue. When serviced the job is assigned to a set of nodes to be executed. The user does not have to concern himself or herself with when the job will run. That is the domain of the scheduler. While a job is running, there is the potential for the job to be preempted by a job submitted to a queue with a higher priority that needs the preempted job's computing resource.

C. JUMMP Design

JUMMP is designed to operate within the Palmetto High Performance Computing Cluster at Clemson University [9] using the PBS Professional scheduler, although the system can easily be adapted to use with other schedulers. Hadoop version 1.1.2 is used in our system. Any 1.x version can be used. The Hadoop cluster uses a single dedicated node for both the NameNode and JobTracker. This dedicated node resides outside the control of the scheduler. Each DataNode/TaskTracker is scheduled as an individual PBS job, which

allows for preemption or failure of a PBS job to only affect a single node of the Hadoop Cluster.

The design choice to use a dedicated persistent node as the master of the Hadoop cluster is based on an assumption that securing a single node for dedicated use from an HPC administrator is more achievable than securing all the nodes required for the cluster. Also, the transition of the head node of an active Hadoop cluster during the middle of a MapReduce job is technically more difficult than the transition of the slave nodes. Including the JobTracker and name node in the maneuvering portion of the cluster is a topic for future work.

Each DataNode and TaskTracker is established within its own PBS job. The PBS job starts the Hadoop daemons and connects them to the persistent head node. Each daemon waits for its trigger to jump. When the trigger to jump is received (PBS preemption or expiration of the jump timer), the PBS job schedules its replacement PBS job. Next, the DataNode decommissions itself from the name node and the TaskTracker blacklists itself from the JobTracker. Finally, the Hadoop daemons are stopped and the PBS job completes. The newly spawned replacement node repeats this process.

When notified of an upcoming jump of a slave node, the name node begins to reassign the blocks replicated on the outgoing node to the remaining nodes in the cluster. The incoming node receives a subset of the blocks its previous node possessed. The JobTracker immediately kills any task assigned to the outgoing TaskTracker and reassigns those tasks to available TaskTrackers.

In our earlier implementations of JUMMP, we stopped the TaskTracker daemons without notifying the JobTracker. After the standard heartbeat failure timeout, tasks would then be reassigned. This behavior greatly degraded performance. The proactive measure of blacklisting the TaskTracker allows the JUMMP to transition much faster.

JUMMP is implemented using a combination of Bash and Python scripts to control execution of the daemons, logging of actions, and trapping of signals to ensure the rescheduling of replacement nodes during jumps.

We describe the configuration of the JUMMP with the following variables:

- The number of DataNodes and TaskTrackers in the system is n.
- The scheduled time between node jumps is t_j.

When the JUMMP system is initialized, the user is able to specify n and t_j. After a stabilization period, a single node will live for nt_j minutes. A node jumps somewhere in the cluster every t_j minutes.

In the event of a jump caused by an PBS scheduler preemption, the jumping node could alert its new node of the remaining time until the regularly scheduled jump, thus allowing the timing of the cluster transitions to be resumed. This topic is an area of future work.

The flexible and maneuverable design of JUMMP allows us to support our user considerations as presented in Section II. While allowing preempted or failed nodes to submit PBS jobs for their replacement, we have guaranteed survival of the Hadoop cluster and continuation of running jobs until completion. Since each instantiation of JUMMP specifies a Hadoop directory with executables and configurations, users are able to tailor their JUMMP instance to the needs of their long running Hadoop environment on an isolated set of nodes within the shared cluster environment. Lastly, administrators can "blacklist" nodes that are experiencing prolonged failure or which have upcoming maintenance. Blacklisted nodes are not listed as available resources for the scheduler, thus allowing JUMMP to avoid and maneuver around those resources to more suitable hardware.

IV. Performance Analysis

Adding maneuverability to a Hadoop cluster incurs some degradation of performance. We attempt to evaluate this degradation to understand the trade-offs of such a system and to estimate the rate at which a jump could efficiently occur. With each jump of a DataNode, additional overhead is occurred over a non-jumping cluster. This overhead comes from two separate sources: the scheduler overhead and replication of HDFS blocks. As previously mentioned, JUMMP schedules each DataNode and TaskTracker as separately scheduled jobs within the supercomputing environment. Scheduling and queuing delays will result in fewer TaskTrackers being available to perform map and reduce tasks. We refer to the JUMMP as being "undersized" during the time when a node is awaiting the scheduler to place the slave node on an available node in the datacenter. Once an existing DataNode leaves the cluster, all blocks that were stored on its local storage must be replicated across the cluster. The NameNode begins this immediately upon the decommissioning of the outgoing node. This data replication consumes processing, network bandwidth, and disk drive seeks that normally would be used for computation for the currently running MapReduce job.

A. Experimental Design

In order to capture, measure and quantify this degradation, we conducted two separate experiments on the performance of JUMMP on the Palmetto High Performance Computer Cluster. In each experiment, we establish a baseline performance metric for a stationary Hadoop cluster by repeatedly running the same MapReduce job 100 times over a static dataset. We rerun the job three additional times while varying the jump time for the cluster. We record the times of the jumps, the individual task start and stop times, and the overall job run time. With these results we quantify the overhead of jumping during the execution of a MapReduce job.

We ran our experiments on a homogeneous set of nodes within Palmetto to ensure uniformity of test results. All tests are run on an isolated pool of 96 nodes for DataNodes and TaskTrackers. The hardware of these nodes is shown in Table I. A datablock size of 256 MB is used throughout the experiments and each TaskTracker has 8 map task slots and 4 reduce task slots. 2GB of memory is allocated for each task. At the allocation of the initial nodes and creation of the JUMMP, the dataset for the experiment is imported, the nodes start jumping, and the MapReduce job begins to run.

For our experiments, we use the dataset and benchmarks from PUMA, Purdue's MapReduce Benchmark Suite [10].

Node	HP SL250s
CPU	Intel Xeon E5-2665 (2)
Cores	16
Memory	64 GB
Local Storage Capacity	900 GB
Networking	Infiniband

TABLE I: Node Configuration

Application	Wordcount	Terasort
Dataset Size	50 GB	300 GB
Node Count	8	32
Jump Times [mins]	7/10/15	20/40/60

TABLE II: Experiment Parameters

PUMA is developed as a benchmark suite to represent a broad range of MapReduce applications exhibiting application characteristics with high/low computation and high/low shuffle volumes. The parameters of our two experiment runs are shown in Table II. The jump times are calculated based on the average running time of the baseline MapReduce job. Jump times are configured to jump during every second, third, and fourth job runs.

B. Characteristics of the Impact

Whenever an active TaskTracker jumps, all tasks currently executing and some previously completed tasks will have to be rerun. Since all reduce tasks must retrieve their partition of intermediate data from all map tasks, any map task previously completed by a jumping TaskTracker will have to be rerun if all reduce tasks have not finished the shuffling phase. Furthermore, any reduce tasks being executed by the jumping TaskTracker will have to be reassigned and rerun. If any reduce task is restarted, then then the map task whose intermediate data is unavailable will also have to be rerun. This re-execution of in-progress and completed tasks adds delay and overhead into the overall execution of the MapReduce job.

This restarting overhead is visualized in Figure 2. Figures (a) and (d) show a typical execution of the *wordcount* and *terasort* experiments in the absence of a node jump. The charts plot the number of currently executing tasks, grouped by type, during the time of the job execution. The job starts with multiple map tasks conducted in parallel. Once the first set of map tasks completes, the shuffle phase of the first set of reduce tasks begins downloading their partitions of the intermediate data from the completed map tasks. This process continues until all map tasks have completed and the shuffling is done, at which time the reduce actions begin. The reducing continues until all reduce tasks have completed and the output is written to HDFS.

Figure 2 (b) and (e) present the cost of jumps occurring before all the mapping and shuffling phases have completed. In these examples, a spike in the number of map tasks at the time of the jump (indicated by dashed vertical lines) shows that additional mapping and shuffling acitivities are performed due to the loss of work from the jumped tasktracker (shown in yellow). The intermediate data from the map tasks completed by the jumping TaskTracker is no longer available for the future reduce tasks and thus that work must be repeated. As illustrated in Figures 2 (c) and (f), when jumps occur during the reduce phase, map tasks must be re-executed and reduce tasks must begin shuffling intermediate data to complete the MapReduce job.

C. Performance Measurements

There are two important measures in understanding the performance of a jumping cluster. The first measure, the "effort" overhead, is the overhead of work that must be re-executed because of a jump. The second measure, "performance" overhead, is the additional wall-time needed to complete the job. We record the performance of our experiments and present the results with a focus on these two measures in this section.

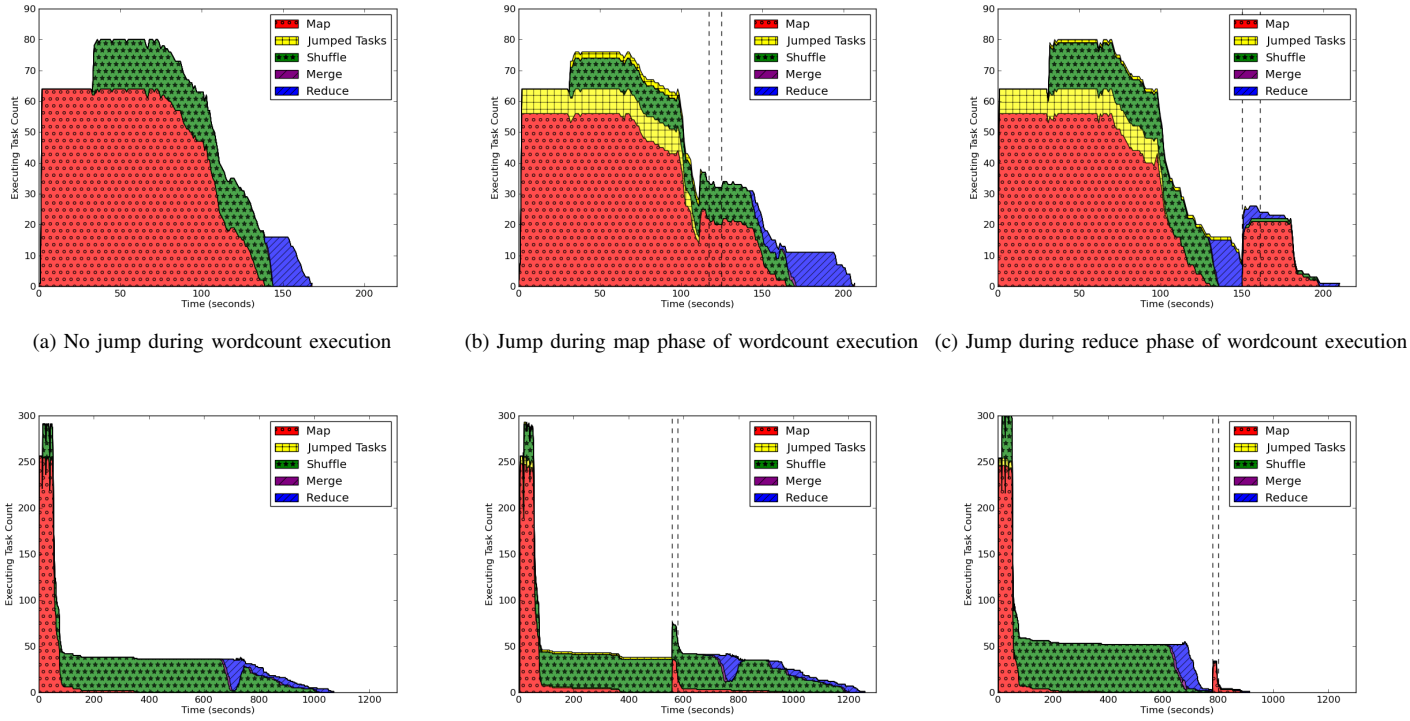
Observing the information shown in Figure 2, we calculate the effort overhead as the area of the jumped map and reduce tasks. This extra computation is wasted since it must be repeated after the jump. We can quantify taskseconds for all phases of the job by taking the area under the curve of each type of task.

The mean taskseconds for each cluster for both the wordcount and terasort applications are shown in Figure 3. Tasks that are executed by a node that will jump during the job execution are refered to as "doomed" since they will be lost and re-executed. The results are interesting in that they show the doomed reduced tasks incurr no additional taskseconds than the Hadoop normal effort overhead of speculative execution of straggler reduce tasks. Furthermore, in both cases of (a) wordcount and (b) terasort, the doomed map taskseconds increase as jump times decrease, but this effort overhead is still two orders of magnitude less than the taskseconds of the successful map tasks.

The running times of the MapReduce jobs themselves are extracted from the normal logging system of Hadoop. Since our experiments use the same job running on the same static dataset, we have control over many variables that affect the run time of MapReduce jobs. Those variables that are outside of our control are locations on disk where intermediate values and reduce outputs are written, network traffic from adjacent nodes not part of the Hadoop cluster, and variations in task scheduling from the JobTracker.

The map task run time cumulative distribution function (CDF) in Figure 4 for both the (a) wordcount and (b) terasort jobs show that jumping has minimal impact on the map task time. This is due to the fact that the system only reports the time for the successful attempt of each map task. For instance, in the case of the wordcount experiment, there are 200 HDFS blocks on which map tasks are placed. If there is a 50% failure rate on map tasks, the time spent during those failed attempts for the 100 map tasks would not be included in the time to complete those tasks. As such, viewing the map task time does not help to show the imposed overhead of a jumping node. That is where information in Figure 3 is more relevant.

The job run time CDFs in Figure 5 for both (a) wordcount and (b) terasort jobs show the expected behavior of longer run times as the frequency of the jumps increase. Each jump time plot is observed to follow the non-jumping plot up until the point where it reaches the percentage of jobs that are under jump conditions. This is directly related to the ratio of average run time and the jump time for the cluster. For instance, the



(a) No jump during wordcount execution (b) Jump during map phase of wordcount execution (c) Jump during reduce phase of wordcount execution

(d) No jump during terasort execution (e) Jump during map phase of terasort execution (f) Jump during reduce phase of terasort execution

Fig. 2: MapReduce job task execution during jump

wordcount job, when not jumping, has an average run time of 3 minutes. When jumping every 7 minutes, 60% of the jobs are unaffected by a jump. Observing the CDF plot will show that the slope of the plot is near vertical up to 60% and the remaining 40% begins to flatten as overhead is observed.

D. Optimizing Jump Time

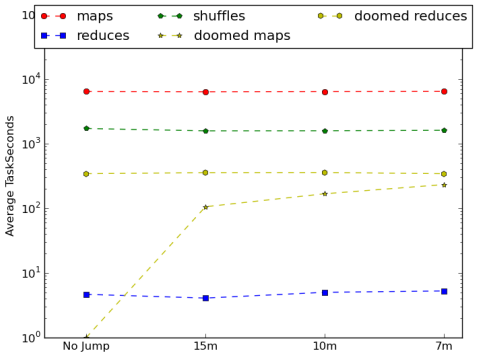
Due to the observed overhead of JUMMP, a very short jump time is not preferred for a maneuverable Hadoop cluster. As the cluster' size increases and reservation window decreases, it becomes necessary for JUMMP to be configured such that a node jumps before the expiration of its reservation window. An optimal jump time is one that minimizes the overhead of the jumps but also allows the size of the cluster to be maximized for improving parallel processing. Equation 1 shows the derivation of the maximum time to jump (t_j) based on reservation window ($RESV$) and size of the cluster (n).

$$\begin{array}{rcl} n \times t_j & \leq & RESV \\ t_j & \leq & \frac{RESV}{n} \end{array} \quad (1)$$

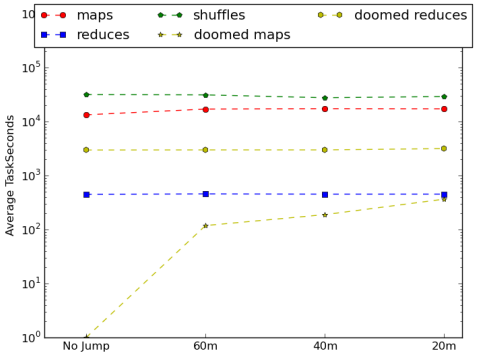
Characteristics of the behavior of Hadoop also contribute to the calculation of the minimal jump time. These factors involve data replication speeds and MapReduce job execution timing. We will look at each of these individually to discuss how they contribute to minimizing the jump time.

The default replication factor in Hadoop is three. This means that a data block will exist within the local storage of three DataNodes. This is configurable by the cluster administrator at start up or on a case-by-case basis when data is written into HDFS. When the NameNode detects a block is under-replicated, it will assign another replication to be stored throughout the cluster. JUMMP initiates this replication proactively when a jumping node decommissions itself from the Hadoop cluster, as previously mentioned. If the jump time is so small that nodes are jumping faster than the blocks can be replicated, the JUMMP could become unstable due to missing blocks. The speed at which data is replicated is related to average number of blocks on the node, the speed of the network interconnect, and the workload of the NameNode to reassign block locations. Calculating the minimum jump time as related to data replication is an area of future work.

Finally, optimal jump time is tied to the timing of the currently executing MapReduce job. As shown in Figure 2, when a TaskTracker jumps, its currently executing and previously completed tasks in the running MapReduce jobs must be reassigned. The biggest impact is observed when a TaskTracker with an active reduce task jumps. This causes the reduce task to start over and have any unavailable map intermediate data to be recalculated. If nodes are jumping so frequently as to never allow a rescheduled reducer to complete execution, then JUMMP will become unstable and jobs will be interrupted. We thus find that the cluster remains stable in the present of jumps as frequent as the average execution time of the reduce tasks of the currently executing MapReduce job.

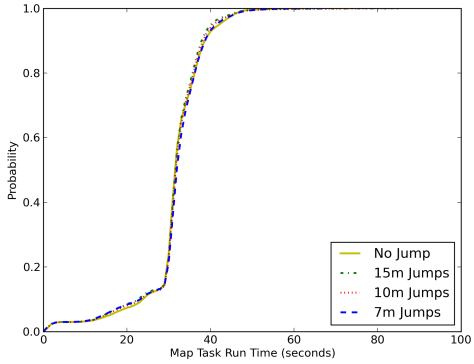


(a) Wordcount TaskSecond Averages

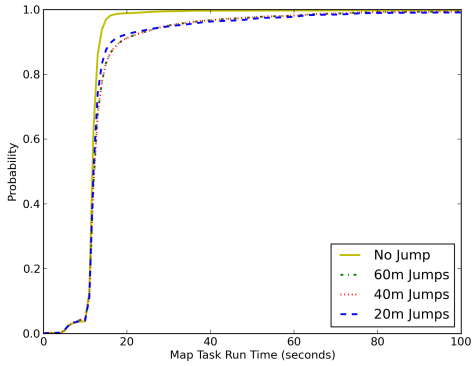


(b) Terasort TaskSecond Averages

Fig. 3: Map, shuffle and reduce taskseconds and doomed map and reduce tasks for wordcount and terasort jobs under various jump times



(a) Wordcount Map Task Time CDF



(b) Terasort Map Task Run Time CDF

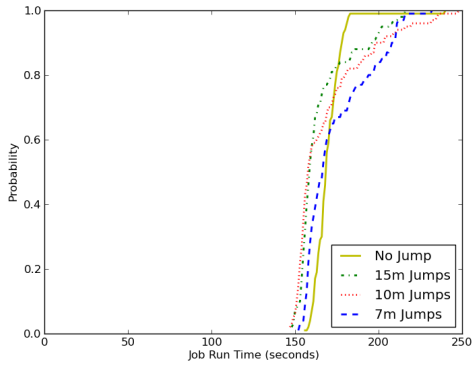
Fig. 4: CDF of Map Task times for WordCount and TeraSort

V. Related Work

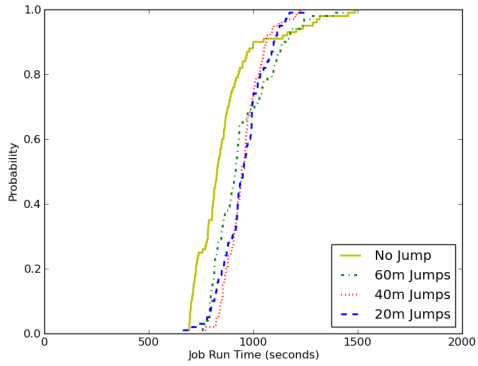
A common solution to integrating Hadoop into existing HPC clusters is the use of virtualization. For clusters that support virtualization interfaces, users can reserve a set of compute nodes and set up a cluster of virtual machines (VMs) on these nodes. This provides users with an isolated environment in which they have full root-level control over the set up of Hadoop. Notable HPC clusters that take advantage of this approach are the FutureGrid distributed testbed [11] and the Amazon Elastic MapReduce Cloud [12]. A drawback of this approach is the degradation in I/O performance for data-intensive MapReduce applications, which has been observed on FutureGrid [13]. Another potential issue is the interaction between Hadoop and non-Hadoop jobs that are part of a workflow using non-Hadoop software packages installed on the physical hardware.

Another approach is to dynamically set up Hadoop environments in users' workspace. A solution provided by Apache is the Apache Hadoop on Demand (HOD) system [14]. HOD enables the quick provision and management of multiple Hadoop MapReduce instances on a shared cluster with PBS Torque scheduler. However, HOD requires access to a static external HDFS cluster. The myHadoop system extends HOD by including a dynamic HDFS with user-generated Hadoop environments [7]. This HDFS could either be placed on the local storage of the compute nodes in non-persistent mode or on a permanent external parallel file system in persistent mode. The approach of HOD and myHadoop enables the concurrent and isolated creation of Hadoop environments with dedicated resources through reservations. However, the implementations of HOD and myHadoop do not allow users to automatically deal with administrative and policy issues such as walltime limitation and priority preemption of computing resources.

Recent work focuses on creation of a new resource management mechanism that can handle both Hadoop and non-Hadoop parallel jobs. Significant efforts include the Apache Mesos project [15] and Apache Hadoop YARN (Yet Another Resource Negotiator) [16]. The main principle for this approach is the separation of resource management and task scheduling mechanism in Hadoop's JobTracker. The new stand-alone resource manager can be used to handle both Hadoop and non-Hadoop processes. This keeps the users from having to configure a new Hadoop environment every time while still maintaining dynamic and isolated execution of MapReduce applications. However, the above projects are still being developed, with Mesos adapted by only a few institutions and YARN yet to be considered stable for production [17].



(a) Wordcount Job Run Time CDF



(b) Terasort Job Run Time CDF

Fig. 5: CDF of Job times for WordCount and TeraSort

VI. Conclusion

We have presented JUMMP, the Job Uninterrupted Maneuverable MapReduce Platform, an automated scheduling platform that provides a customized Hadoop environment within a batch-scheduled cluster environment. Through its design and redundant nature, JUMMP enables an interactive pseudo-persistent MapReduce platform within the existing administrative structure of an academic high performance computing center. Our experimental evaluation shows that JUMMP can be as efficient as a persistent Hadoop cluster on dedicated computing resources, depending on the jump time. Additionally, we show that the cluster remains stable, with good performance, in the presence of jumps that occur as frequently as the average length of reduce tasks of the currently executing MapReduce job.

In the future, we plan to explore how JUMMP can be used to provide a moving target defense approach to Hadoop cluster security, to integrate Hadoop with a distributed network file system available in many large campus supercomputing clusters, and to integrate Software Defined Networking to build dynamic network topologies to minimize network congestion for MapReduce traffic. We also plan to investigate smart jumping, determined by coordination with the NameNode and JobTracker to minimize overhead and the inclusion of the head node into the maneuverable portion of the cluster.

Acknowledgments

We would like to acknowledge the use of Clemson University's Palmetto Cluster and assistance from Randy Martin and Corey Ferrier, Clemson Computing and Information Technology. This work was supported in part by National Science Foundation Grant #1228312.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[2] Apache Hadoop, http://hadoop.apache.org, 2013.

[3] J. Y. Monteith, J. McGregor, and J. Ingram, "Hadoop and its evolving ecosystem," *Proceedings of IWSECO*, 2013.

[4] 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 *Proceedings of the ACM SIGMOD*, 2009.

[5] Hortonworks, "Best practices for selecting apache hadoop hardware," http://hortonworks.com/blog/best-practices-for-selecting-apache-hadoop-hardware/.

[6] Z. Sebepou, K. Magoutis, M. Marazakis, and A. Bilas, "A comparative experimental study of parallel file systems for large-scale data processing," in *Proceedings of the 1st USENIX Workshop of Large-Scale Computing*, 2008.

[7] S. Krishnan, M. Tatineni, and C. Baru, "myHadoop - Hadoop-on-Demand on Traditional HPC Resources," http://www.sdsc.edu/pub/techreports/SDSC-TR-2011-2-Hadoop.pdf, 2011, San Diego Supercomputing Center Tech Report.

[8] R. L. Henderson, "Job scheduling under the portable batch system," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson and L. Rudolph, Eds. Springer Berlin Heidelberg, 1995, vol. 949, pp. 279–294.

[9] Clemson University, "Palmetto cluster - high performance computing resource," http://citi.clemson.edu/palmetto/.

[10] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," http://web.ics.purdue.edu/ fahmad/benchmarks.htm.

[11] J. Diaz, G. von Laszewski, F. Wang, A. J. Younge, and G. Fox, "Futuregrid image repository: A generic catalog and storage system for heterogeneous virtual machine images," in *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science*, 2011.

[12] Amazon Elastic MapReduce, http://www.aws.amazon.com/elasticmapreduce/, 2013.

[13] Y. Kang and G. C. Fox, "Performance evaluation of mapreduce applications on cloud computing environment, futuregrid," in *Grid and Distributed Computing*, T. Kim, H. Adeli, H. Cho, O. Gervasi, S. S. Yau, B. Kang, and J. G. Villalba, Eds. Springer Berlin Heidelberg, 2011, vol. 261.

[14] Apache Hadoop on Demand, http://hadoop.apache.org/docs/r1.1.2/hod_scheduler.html, 2013.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked System Design and Implementation*, 2011.

[16] Apache Hadoop NextGen MapReduce, http://hadoop.apache.org/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2013.

[17] Deploying MapReduce v2 (YARN) on a Cluster, http://www.cloudera.com/content/cloudera-content/cloudera-docs/CDH4/4.2.0/CDH4-Installation-Guide/cdh4ig_topic_11_4.html, 2013.