Riding the Elephant: Managing Ensembles with Hadoop

Elif Dede, Madhusudhan Govindaraju
State University of New York (SUNY)
Binghamton, NY, USA
{edede1, mgovinda} @binghamton.edu

Dan Gunter, Lavanya Ramakrishnan
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
{dkgunter,lramakrishnan}@lbl.gov

DRAFT

ABSTRACT

Many important scientific applications do not fit the traditional model of a monolithic simulation running on thousands of nodes. Scientific workflows – such as the Materials Genome project, Energy Frontiers Research Center for Gas Separations Relevant to Clean Energy Technologies, climate simulations, and Uncertainty Quantification in fluid and solid dynamics – all run large numbers of parallel analyses, which we call *scientific ensembles*. These scientific ensembles have a large number of tasks with control and data dependencies. Current tools for creating and managing these ensembles in HPC environments are limited and difficult to use; this is proving to be a limiting factor to running scientific ensembles at the large scale enabled by these HPC environments. MapReduce and its open-source implementation, Hadoop, is an attractive paradigm due to the simplicity of the programming model and intrinsic mechanisms for handling scalability and fault-tolerance. In this paper, we evaluate the programmability of MapReduce and Hadoop for scientific workflow ensembles.

1. INTRODUCTION

Advances in computing over the past few decades have resulted in a large number of computational models being available for simulating complex physical systems. Many of these applications do not fit the traditional model of a monolithic simulation running on thousands of nodes. For example, accurate bounds on the computational models, through Verification and Validation (V&V) and Uncertainty Quantification (UQ) are obtained by running a large number of model runs with different parameters. Similar levels of parallelism is also seen in other projects such as the Material Genome project [2] In addition, large data volumes generated by scientific simulations are resulting in exploratory data analysis that have similar workflow pattern. We use the term *scientific ensembles* to describe these class of applications where there are a large number of parallel tasks with control or data dependencies that must run in coordination to arrive at a scientifically meaningful result. These scientific ensembles have many tasks and/or operate and generate large amounts of data and essentially fits in the many-task paradigm [27].

Current approaches to managing these many-task scientific ensembles provide limited support for programming, fault-tolerance, scaling and dynamic adaptation to varying resource conditions. In fact, policies on the number of concurrent jobs from a single user in current batch queue systems hinder the scalability of these application. These challenges will be further exacerbated as we approach the exascale era due to changes in hardware that affect application failure characteristics, cost of I/O operations, limit on memory available per core/task. Thus new and innovative approaches to managing scientific ensembles are necessary.

In recent years, the MapReduce programming model and its open source implementation, Apache Hadoop, have taken traction as a means to process large volumes of data. The MapReduce [16] programming model consists of two functions familiar

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

to functional programming, *map* and *reduce*, that are each applied in parallel to a data block. The Google implementation of MapReduce remains proprietary; Apache Hadoop is an open-source Java implementation of MapReduce. The inherent support for parallelism built into the programming model enables horizontal scaling and fault-tolerance.

Scientific ensembles have many characteristics in common with MapReduce workloads, as they both employ a high degree of parallelism that must be run in coordination to arrive at a meaningful result. However these scientific workloads have a key distinguishing characteristic: multiple files per task and specific parameters per task. Thus, we must investigate further to see whether MapReduce and its open-source implementations are a convenient programming model to manage loosely coupled asynchronous scientific ensembles.

Recently, there have been a number of implementations of MapReduce and similar data processing tools [10, 14, 18, 20, 24, 30, 36]. Apache Hadoop has evolved rapidly as the leading platform and has spawned an entire ecosystem of supporting products. Thus we use Hadoop, as a representative MapReduce implementation for our evaluation. Prior work has compared the suitability of these systems for different workloads [13].

In this paper, we evaluate Hadoop's suitability as a platform for running large, complex scientific ensembles. This evaluation encompasses the following contributions:

- We compare and contrast Hadoop jobs and scientific ensembles,
- We analyze the applicability of MapReduce for six common ensemble patterns and describe the challenges from programming these patterns in Hadoop and,
- We discuss the current gaps and challenges with using Hadoop for scientific ensembles.

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Section 3, we present the background on scientific ensembles, MapReduce and Apache Hadoop and compare and contrast scientific ensembles with MapReduce jobs. Section 4 details our analysis approach. We summarize our results and identify gaps and challenges in supporting scientific ensembles with Apache Hadoop in Section 5 and finally conclude in Section 6.

2. RELATED WORK

Scientific ensembles have recently gained traction due to increased parameter space and data volumes. We are not aware of any prior work that investigates MapReduce/Hadoop as a framework for composing and managing these jobs. However the execution framework required for scientific ensembles have similar characteristics to workflow tools and other job management frameworks and we detail these systems in this section and contrast it with our approach.

Scientific Workflow Tools. Running scientific workflows in distributed environments such as grid environments has been an area of interest since the introduction of large scale systems. Pegasus [9] supports execution of workflows in distributed environments such as campus clusters, grids, clouds. Pegasus Workflow Management Service maps an application onto available resources pertaining to the cluster while keeping the internal and external dependencies of the workflow in order. Triana [35] is an open source graphical problem solving environment that allows you to assemble and run a workflow through a graphical user interface while minimizing the burden of programming [5]. Taverna [19] provides an easy to use environment to build, execute and share workflows of web services. These workflow tools focus on federation of sites and have limited or no support for automatic scaling with growth in data volumes, fault-tolerance and dynamic conditions.

Various groups have have modeled workflows using using formal semantics including lambda calculs [22, 34]. Our work is complementary to these efforts and specifically explores the representation of scientific ensembles using the MapReduce model.

MapReduce and Scientific Applications. MapReduce has been gaining popularity for use in scientific applications. With its computational power and ease of use, MapReduce provides a simple programming model for data-intensive applications. By moving the computation to the data locations, the model addresses a key performance bottleneck in data intensive scientific applications like ROOT [8], BLAST [7], statistical algorithms such as K-means, bioinformatics, and many other data mining problems. However, there is still an acknowledged lack of good examples and techniques for programming scientific applications with MapReduce. This gap has been an extensive area of research and has been targeted by other groups [1, 10, 11].

Hadoop has been used to evaluate the perfor-

DRAFT

mance of various scientific applications from various domains including bioinformatics, climate simulation analysis [29].

CloudBurst [32] uses Hadoop to parallelize mapping sequence data for biological analyses while [26] uses Hadoop to reduce the time spent on human genome indexing from several hours to a handful of minutes. Lin *et al.*[23] evaluate existing MapReduce graph algorithms using Hadoop and proposes a new set of design patterns addressing the deficiencies encountered.

Hadoop Workflow Tools. CloudWF [37] and Oozie [3] are workflow systems for cloud that are built on top of Hadoop. These tools enable users to chain multiple MapReduce jobs but do not support any additional patterns. These tools also rely on Hadoop's mechanisms for fault-tolerance, scalability and thus we focus our efforts on evaluating if Hadoop would work for scientific ensembles.

UQ tools. Uncertainty Quantification software is being used for running large and complex simulations. The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) [12] is used for software optimization, parameter estimation, sensitivity and variance analysis. While providing UQ specific solutions for running large and complicated workflow based applications, DAKOTA also offers a set of concurrent computing and simulation interfacing tools [4]. PSUADE [6] is a software library developed to support UQ studies by providing features like parallel function evaluations, sampling and analysis methods, an integrated design and analysis framework and fault tolerance. However, in both these tools, details of parallelism must be explicitly specified by the user this limiting scalability.

3. OVERVIEW

In this section we describe the characteristics of a) scientific ensembles through examples (Section 3.1), b) MapReduce model (Section 3.2) c) Hadoop (Section 3.3). Finally, we compare and contrast Hadoop jobs and science ensembles (Section 3.4).

3.1 Scientific Ensembles

Scientific ensembles are applications with a large number of loosely-coupled parallel tasks that need to be managed as a single unit. Scientific ensembles can be diverse in structure. However, they do share some broad characteristics. In general, ensembles consist of one or more phases of execution, in which input is divided among a number of parallel workers, who perform computations on that input. The output from the parallel workers may be collected by one process, or recorded by each worker separately.

Unlike MapReduce workloads, which typically run on commodity clusters, scientific ensembles often run in HPC centers. However, they still share many requirements with MapReduce jobs: proximity of data, fault-tolerance and the ability to easily manage and scale up computations with growth of data.

In the Materials Genome (MG) scientific ensemble, the inputs are atomic crystal structures and the workers are each running a series of parallel Vienna Ab-initio Simulation Package (VASP) calculations to produce estimates of the crystal properties. Each worker reports its results back to a central database independently.

In the case of the CyberShake scientific ensmble, there are two phases: first, workers calculate Strain Green Tensors over a geographic area, then this becomes the input for the next phase where workers generate seismograms and peak spectral acceleration values for different rupture variations. This is an example of multiple distinct computational phases linked in series.

Sometimes scientific ensembles may grow during execution, as they do for the MG VASP jobs: when a calculation does not converge it may need to be re-run with different parameters such as additional mesh points (*k-points*).

The requirements for scientific ensembles that perform Uncertainty Quantification (UQ) are even more complex, in that the results from one simulation may either add new simulations, modify parameters, or remove other simulations from the ensemble. This is true because UQ can explore the parameter space at different granularities and with various techniques such as surrogate models, and the results of a "cheaper" simulation may either obviate or require a more expensive one.

In this paper, we select a set of common scientific ensemble patterns and implement the patterns with Hadoop, the open-source MapReduce implementation. We detail the implementation challenges and analyze the gaps and performance impact for each of the selected workflow patterns.

3.2 MapReduce Basics

MapReduce(MR) is a programming model that enables users to achieve large-scale parallelism when processing and generating large data sets. The

DRAFT

MapReduce model is at the foundation of processing large amount of index, log and user behavior data in global internet companies including Google, Yahoo!, Facebook and Twitter.

The MapReduce model was designed to be implemented and deployed on very large clusters of commodity machines. It was originally developed, and is still heavily used for massive data analysis tasks. At Google, MapReduce jobs are run on the proprietary Google File System (GFS) distributed file system [17]. GFS supports several features that are specifically suited to MapReduce workloads, such as data locality and data replication.

We differentiate between the abstract MapReduce model and the Hadoop implementation, following for our abstract representation the style of formalisms used by Karloff *et al.* [21]. In the abstract, MapReduce is defined by two functions, a `mapper` μ and `reducer` ρ. These functions operate on key/value pairs, $\langle k, v \rangle$, where both the key and value are finite binary strings. The `mapper` function takes as input one $\langle k, v \rangle$ and outputs a finite multiset of $\langle k', v' \rangle$ pairs, where the domains for k, k', v, and v' are distinct. The `mapper` must be stateless, i.e., it must operate on each key/value pair independently with no side-effects. This allows arbitrary splitting of the input set of key/value pairs across parallel mapper instances, with zero communication between them.

The `reducer` takes as input a key, k and list of N values $\{v_i : i = 1 \ldots N\}$ and outputs a multi-set of M pairs $\{\langle k, v_j \rangle : i = 1 \ldots M\}$ where again each k is the same as the input key, and the values v_j are from the same domain as the input values v_i. The values must be from the same domain because this allows reducers to be nested arbitrarily. For example, it ensures that $\rho(\rho(k, \{x\}), \rho(k, \{y, z\})) \equiv \rho(k, \{x, y, z\})$. The freedom this allows for parallel and hierarchical arrangements of reducers greatly increases parallel scalability.

In the MapReduce model all parallel instances of mappers or reducers are the same program; any differentiation in their behavior must be based solely on the inputs they operate on.

The canonical basic example of MapReduce is a word-count algorithm where the map emits the word itself as the key and the number 1 as the value; and the reduce sums the values for each key, thus counting the total number of occurrences for each word. We would notate this as follows, where M refers to the mapper and R to the reducer:

$$\begin{aligned} \text{M}: &\quad \mu(\langle \text{`word'}, \varnothing \rangle) \rightarrow \{\langle \text{word}, v \rangle\} \\ \text{R}: &\quad \rho(\text{word}, \{v_1, v_2, \ldots, v_n\}) \rightarrow \{\langle \text{word}, n \rangle\} \end{aligned}$$

Thus, scientific ensembles structures have many similarities with the MapReduce model and a scientific ensemble can be considered as a series of *map* and *reduce* phases. For example, a scientific ensemble can be considered to consist of a sequence of R mappers and reducers.

$$\langle \mu_1, \rho_1, \mu_2, \rho_2, \ldots, \mu_R, \rho_R \rangle$$

Other combinations and sequences of maps and reduces might also be possible e.g. two map phases followed by a reduce phase.

3.3 Hadoop

Hadoop is an open-source distributed computing platform that implements the MapReduce model. Hadoop consists of two core components: the job management framework and the Hadoop Distributed File System (HDFS) [33]. Hadoop's job management framework is highly reliable and available, using techniques such as replication and automated restart of failed tasks. The framework has optimizations for heterogeneous environments and workloads, *e.g.*, speculative (redundant) execution that reduces delays due to stragglers. HDFS is a highly scalable, fault-tolerant file system modeled after the Google File System. The data locality features of HDFS are used by the Hadoop scheduler to schedule the I/O intensive *map* computations closer to the data.

The scalability and reliability characteristics of Hadoop suggest that it could be used as an engine for running scientific ensembles. However, the target application for Hadoop is very loosely coupled analysis of huge data sets. It is not apparent that the Hadoop implementation, or perhaps even MapReduce itself, is sufficiently flexible to be used (without significant additional infrastructure) for scientific ensembles, which typically exhibit more complex data and computational dependency structures. The goal of this paper is to understand the feasibility and difficulty level of using the MapReduce model and specifically Hadoop to compose and manage scientific ensembles.

3.4 Hadoop Jobs and Scientific Ensembles

DRAFT

Scientific ensembles and Hadoop jobs have a number of similarities in their characteristics and requirements:

- Hadoop jobs consist of two primary phases - *map* and *reduce*. The jobs consist of a large number of maps that performs data transformation and one or more reduces that performs a combine operation to produce the final result. Scientific ensembles might have many execution phases but they can be roughly categorized as *data setup or problem decomposition, data transformation and data aggregation*. The problem decomposition is implicit in vanilla Hadoop jobs where the input is divided into blocks for the workers to operate on. The data transformation and data aggregation phases are similar to the map and reduce phases of a Hadoop job.

- Both scientific Hadoop jobs and scientific ensembles have a similar structure - large number of parallel tasks in the "map" phase and a smaller number of tasks in the "reduce" phase.

- Both scientific ensembles and Hadoop jobs exhibit data-flow parallelism i.e., they operate on either different data sets or different parameters on the same data sets.

- Scientific ensembles and Hadoop jobs both require the ability to scale up easily as more data or when additional parameters need to be analyzed.

- The large-scale parallelism makes fault-tolerance and data proximity critical for both applications.

However there are a number of differences between Hadoop workloads and typical scientific ensembles. First, Hadoop jobs consists of a map phase followed by a reduce phase whereas scientific ensembles might have a diverse set of tasks followed by a variable number of stages with additional tasks. Second, Hadoop assumes that each mapper works on a subset of the data and there is no affinity between map tasks and data blocks. In contrast, scientific applications often operate on files rather than blocks of data. Scientific applications may also require diverse codes to run as map tasks and/or may take additional parameters that might change the processing in the map tasks appropriately.

Workflow tools have also been previously used to compose scientific ensembles. Workflow tools typically have been control-flow oriented rather than data-flow oriented. Scientific ensembles however exhibit distinct data-flow characteristics, and hence workflow tools are not always appropriate for them.

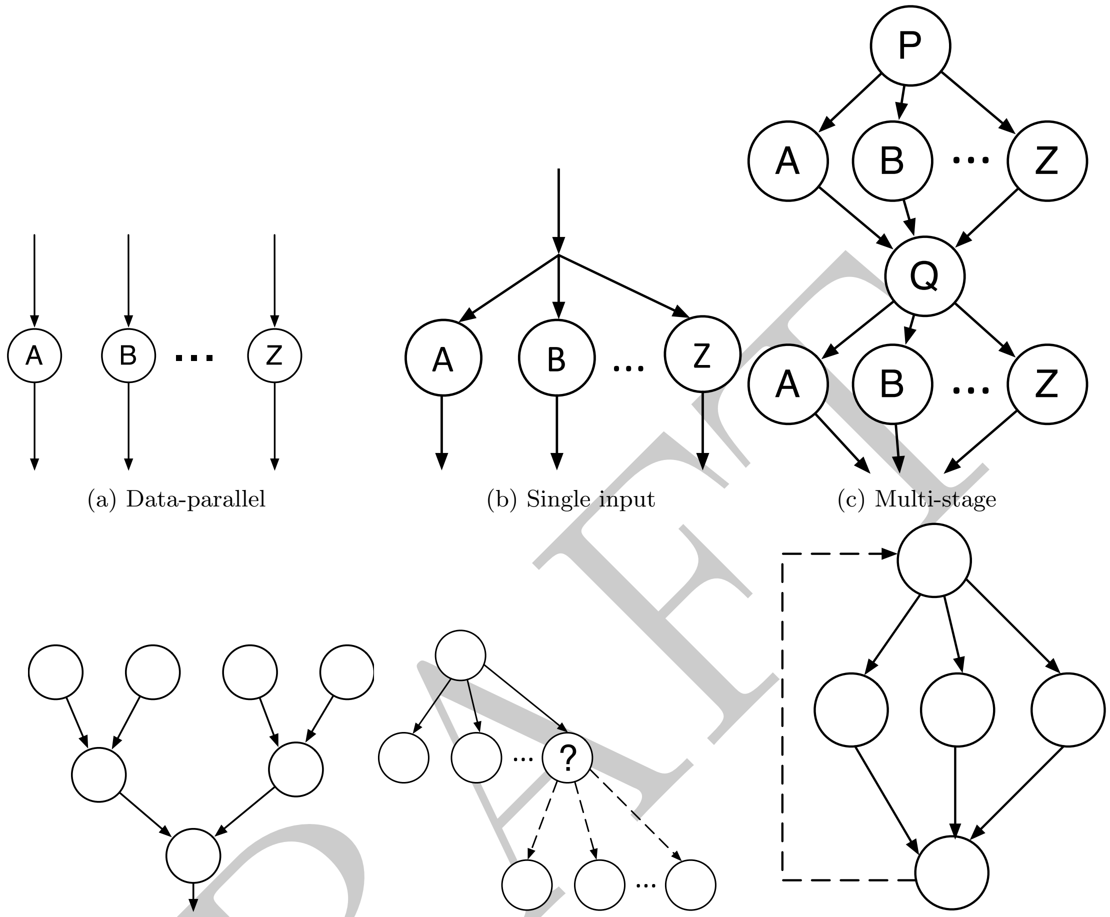
MapReduce has evolved in a completely different paradigm from scientific computations. The strengths of the model arise from its simple easy-to-learn programmability and, in implementation, its ability to adapt to failures in the underlying resources. Today, a large number of scientific computations use the Message Passing Interface (MPI) [15] as the programming model. MPI is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. Unlike in Hadoop, within standard MPI programs fault-tolerance needs to be handled explicitly by the application programmer. However, resource failures during execution are becoming an increasingly important issue for ever-larger scientific ensembles [31]. Hence, we explore the use of the MapReduce programming model for scientific ensembles.

4. EVALUATION

The scientific ensemble examples in Section 3.1 illustrate the diverse characteristics of scientific pipelines. In our evaluation, we use six representative ensemble patterns to capture the variation found in scientific ensembles. The patterns were selected to represent the different structures possible in scientific ensembles. We describe these patterns and discuss how they fit in the MapReduce model and discussion implementation challenges in Hadoop and corresponding impact on performance and reliability.

Workload Patterns. A schematic of each pattern is shown in Figure 1. We consider a data-parallel ensemble pattern that is closest to the default MapReduce pattern with just a map phase and zero reduces(Figure reffig:disjoint). The single input (Figure 1b) ensemble is representative of a scientific ensemble where a single input file might trigger multiple simulation or analysis in parallel with different parameters.The multi-stage pattern (Figure 1c) has three phases in stage one (a problem decomposition, computation or data transformation and a data aggregation phase) and the final stage triggers additional computation and data aggregation steps. The inverse tree pattern (Figure 1d) is selected to represent a mult-stage data

DRAFT



(a) Data-parallel (b) Single input (c) Multi-stage

(d) Inverse tree (e) Dynamic (f) Iteration

Figure 1: Workflow Patterns

DRAFT

aggregation. The dynamic (Figure 1e) and iteration patterns (Figure 1f) capture the run-time variability that might result in change in the workflow structure. These workflow patterns also exhibit diverse characteristics in terms of number of stages, number of initial inputs, number of outputs and hierarchy depth.

We evaluate only the basic patterns and other derived patterns that are a combination of one or more of the give basic patterns are possible. Our goal is to evaluate Hadoop for scientific ensembles and thus the wider class of generic scientific workflows is outside the scope of this paper [28].

Metrics. We analyze each workflow pattern discussed above along four dimensions:

- ***MapReduce Realization.*** We compare the given workflow pattern with the MapReduce model to understand the difference of each of the patterns from the base model.
- ***Hadoop Implementation.*** Next, we outline our experience and identify strengths and gaps with composition of the workflow pattern in Hadoop's implementation of the MapReduce model.
- ***Data management.*** We evaluate the ease in managing the inputs and outputs for a given workflow pattern in Hadoop's imple-

mentation.

- ***Performance and Reliability.*** Finally, we outline the impact of our implementation on the performance, fault-tolerance, efficiency and scalability relative to what is achievable with respect to native Hadoop jobs.

In our patterns, we only consider ensembles where each first-stage task takes a single input. The MapReduce model by default is designed to work such that each map task works on single input block. There is limited or no support in tools such as Hadoop for tasks that might take one or more inputs. In the following discussion, $\mathcal{W}$ stands for the number of workers, i.e. the desired parallelism. If the workflow has multiple distinct "stages" then the parallelism at stage i will be written as $\mathcal{W}_i$.

4.1 Data-parallel

In this pattern, each task operates on a separate file or block of data. These tasks have high degree of parallelism inherently and often run together as they are part of the same scientific experiment or have input data arriving at the same time. These tasks may or may not have common infrastructure they access such as backend databases that might limit the level of parallelism possible. For example, the content maintenance cycle of the Integrated Microbial Genomes (IMG) system [25] involves running BLAST [7] for identifying pair-wise gene similarities between new metagenome and reference genomes. This activity can be highly parallelized as each task works on a set of input metagenomes that it compares against the reference genomes.

MapReduce Realization. The data-parallel workflow pattern (see Figure 1a) is easily represented in the MapReduce model. The input value δ_i is the appropriate input for each process $A, B, \ldots, Z$ and the reduce ρ simply returns a copy of its inputs as a multi-set, i.e., the reduce is an identity function. The key k output by the mapper may be randomized or empty.

$$\begin{aligned} \mathrm{M:}\quad & \mu(\langle \varnothing, \delta_i \rangle) \rightarrow \langle k, \{v_1, v_2, \ldots, v_n\} \rangle \\ \mathrm{R:}\quad & \rho(\langle k, \{v_1, v_2, \ldots, v_n\} \rangle) \rightarrow \{\langle k, v_1 \rangle, \langle k, v_2 \rangle, \ldots, \langle k, v_n \rangle\} \end{aligned}$$

The program would consist of one mapper and one reducer, $\langle \mu, \rho \rangle$.

Hadoop Implementation. First, we consider the simple case where $A = B = \cdots = Z$. The mapper phase is straightforward but there are a couple of alternatives to handling the reduce phase. This is most naturally implemented with $\mathcal{W}$ `Identity` reducers. Each mapper associates all its output with a single unique key that is then passed to each reducer. Alternatively zero reducers can be specified allowing for mapper output to be the output of the job.

When the processing is heterogeneous, i.e. when $A \neq B \neq C \ldots, \neq Z$, additional logic needs to be included in the Mapper implementation to handle the different possible functions of the tasks $A, B, \ldots, Z$. This is discussed in more detail in (4.2), but we note that it is true to some extent for all the workflow patterns.

Data management. The only difference in the data management is associated with the fundamental division of unit of work between MapReduce jobs and scientific ensembles. Each task in this pattern operates on a different file and thus Hadoop needs to instructed to not split the file into blocks. This can be achieved in Hadoop by specifying the input to be "Non-Splittable".

Performance and Reliability. This workflow pattern is able to take advantage of all Hadoop features ensuring scaling and performance similar to a pure MapReduce job.

4.2 Single Input

In this pattern (Figure 1b), a single input is fed into a number of parallel tasks. The tasks differ in the parameter that it takes and/or the operation performed on the data itself might be different. For example, UQ analysis might have a set of tasks operate on the same input deck with different parameters.

MapReduce Realization. From the perspective of the MapReduce model, this pattern is similar to the Data-parallel pattern. The difference is that instead of multiple input data blocks δ_i each mapper operates on the same input data block δ. This has an important implication: the behavior of the mapper can no longer be controlled by different input data. If each mapper must perform one of the functions in $A, B, \ldots, Z$ then this decision would need additional information, for example a different key k_i for each type of function. This key may be propagated to the reducers.

$$\begin{aligned} \mathrm{M:}\quad & \mu(\langle k_i, \delta \rangle) \rightarrow \langle k_i, \{v_1, v_2, \ldots, v_n\} \rangle \\ \mathrm{R:}\quad & \rho(\langle k_i, \{v_1, v_2, \ldots, v_n\} \rangle) \\ & \rightarrow \{\langle k_i, v_1 \rangle, \langle k_i, v_2 \rangle, \ldots, \langle k_i, v_n \rangle\} \end{aligned}$$

DRAFT

The program would consist of one mapper and one reducer $\langle \mu, \rho \rangle$.

Hadoop Implementation. Hadoop assumes that each mapper operates on a separate data item, which is selected by "splitting" the original input data. To implement a single data source being processed in different ways by the same mapper a mechanism would be needed to communicate the type of processing to the mapper. We could encode the key of the key-value pair or introduce a parameter p_i. There is no support in Hadoop to communicate additional parameters to the mappers. Additionally, as our goal was to find mechanisms with minimal programming effort, we artificially replicated the input to represent multiple inputs that were then consumed by the mappers, similar to the Data-parallel pattern.

As in the Data-Parallel pattern (4.1), we set the number of reducers to either $\mathcal{W}$ or zero.

Data management. Hadoop is designed to operate on many different input "splits" of an input file. In this pattern, each worker must operate on the same input, but produce separate outputs. To implement this in Hadoop, we replicated the file $\mathcal{W}$ times by copying the input file in HDFS, with different names. Thus, file F became files $F_1, F_2, \ldots F_{\mathcal{W}}$.

Performance and Reliability. The data management implementation for this workflow pattern results in wasted storage especially for large input files. Each file in Hadoop is replicated three times, thus resulting in $3 * W$ copies of the input file. These files are identical, and since the copies are managed by the user Hadoop cannot efficiently use all the copies for data locality and fault tolerance decisions.

4.3 Scatter-Gather

The Scatter-Gather pattern shown in Figure 1c captures a two-level task process where the input is "scattered" to many tasks whose output is "gathered" into a single task, which may then re-scatter the input to the next stage. Such a workflow pattern would be common in exploratory data analysis where a first stage data analysis is fed as input to the second stage of data analysis and so forth.

MapReduce Realization. As discussed earlier, the pattern's data setup task P does not have an equivalent in the MapReduce model. Thus, that task can be represented as either a map or a reduce task. If we were to express P as a map task, the first map task M_1 can be expressed as a series of functions f_i applied to a single input δ_s, generating multiple key/value pairs with one key per reducer task ($A \ldots Z$ in Figure 1c). The rest of the pattern is a series of MapReduce stages. Tasks $A \ldots Z$ correspond to a data-parallel map phase and the task Q corresponds to a single reduce task.

$$\begin{aligned}
&\mathrm{M}_P : \mu(\langle \kappa, \delta_s \rangle) = f_1(\delta_s), f_2(\delta_s), \ldots f_n(\delta_s) \rightarrow \\
&\qquad \{\langle k_1, values_1 \rangle, \langle k_2, values_2 \rangle, \ldots, \langle k_n, values_n \rangle\} \\
&\mathrm{M}: \quad \mu(\langle \varnothing, \delta_i \rangle) \rightarrow \langle k, \{v_1, v_2, \ldots, v_n\} \rangle \\
&\mathrm{R}: \quad \rho(\langle k, \{v_1, v_2, \ldots, v_n\} \rangle) \rightarrow \{\langle k, v_1 \rangle, \langle k, v_2 \rangle, \ldots, \langle k, v_n \rangle\}
\end{aligned}$$

Thus the pattern can be represented as $\{\langle M_P, M, R \rangle^*\}$.

Alternatively, this pattern can be considered a sequence of two types of *map* and *reduce* tasks. At a given step, s, in the sequence the first map task M_1 (P in the figure) can be expressed as a series of functions f_i applied to a single input δ_s, generating multiple key/value pairs with one key per reducer task ($A \ldots Z$ in the figure).

The next stage mapper M_2 simply copies the inputs with the same empty or constant key κ so that a single type of next stage reducer R_2 will receive all the outputs.

$$\begin{aligned}
&\mathrm{M}_1 : \mu(\langle \kappa, \delta_s \rangle) = f_1(\delta_s), f_2(\delta_s), \ldots f_n(\delta_s) \rightarrow \\
&\qquad \{\langle k_1, values_1 \rangle, \langle k_2, values_2 \rangle, \ldots, \langle k_n, values_n \rangle\} \\
&\mathrm{R}_1 : \rho(\langle k_i, values_i \rangle) \rightarrow \langle k_i, v_i \rangle \\
&\mathrm{M}_2 : \mu(\langle k_i, v_i \rangle) \rightarrow \langle \kappa, v_i \rangle \\
&\mathrm{R}_2 : \rho(\langle \kappa, \{v_1, v_2, \ldots, v_n\} \rangle) \rightarrow \{\langle \kappa, \delta_{s+1} \rangle\}
\end{aligned}$$

The extended MapReduce program then becomes a repeated sequence of these four $\{\langle M_1, R_1, M_2, R_2 \rangle^*\}$.

Hadoop Implementation. We chose to implement the first MapReduce realization for our implementation. Task P in Figure 1c is the data preparation phase and we model it as a single task Single Input MapReduce job since this step is implicit in traditional MapReduce jobs. Subsequent stages in the workflow are modeled as series of data-parallel patterns with a single reducer (e.g., task Q). The execution flow across these stages are managed through a script that submits the jobs to Hadoop. The inputs to the second and subsequent MapReduces are the output files generated from the first MapReduce.

Data management. In MapReduce jobs, each reduce task creates a single output file. However in our case, task Q needs to emit multiple output files that drive the second stage of the workflow. To implement this, we overrode a Hadoop method

DRAFT

called `generateFileNameForKey` in the `MultipleTextOutputFormat` class to create a different output file for each key. There is no easy way to control the output splits from task P or Q without embedding the logic in the task implementation.

Performance and Reliability. In this case, we could have a load imbalance problem since task Q might have more outputs for one key value resulting in one task processing more data than the others.

4.4 Inverse Tree

This workflow pattern (see Figure 1d) begins with $\mathcal{W}$ workers, then recursively merges their output to $\mathcal{W}/2$ workers in each subsequent stage until there is only one worker left. In the MapReduce model this pattern can be achieved by encoding the position in the tree into the key itself, as described below.

MapReduce Realization. To implement this in the abstract MapReduce model, we first observe that if each node at the top of the Inverse Tree is numbered consecutively from left to right, then integer division of each of these numbers by 2 will group the outputs correctly for the next level. The binary representation of that number can be thought of as the path from the bottom (root) to the leaf where 0 means "left" and 1 means "right". An example with $\mathcal{W} = 5$ is shown in Figure 2.

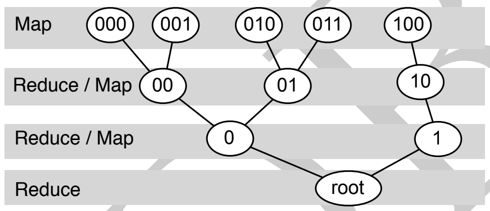


Figure 2: Numbering of 5-node tree

If the keys k_i for inputs δ_i are these binary numbers, every mapper will simply divide the keys by two and pass through the input to the reducer. A minor additional complexity is that the first phase mapper M_0 must also do the work f_* that is done in subsequent phases within the reducer. Thus, the representation in the MapReduce model uses a different mapper and reducer for the first phase, M_0 and R_0, than for the subsequent phases:

$$\begin{aligned} \mathrm{M}_0 &: \mu(\langle k_i, \delta\rangle) \rightarrow \{\langle k_j = \frac{k_i}{2}, values_j = f_j(\delta)\rangle\} \\ \mathrm{R}_0 &: \rho(\langle k_i, values_i\rangle) \rightarrow \langle k_i, v_i\rangle \\ \mathrm{M} &: \mu(\langle k_i, v_i\rangle) \rightarrow \langle \frac{k_i}{2}, v_i\rangle \\ \mathrm{R} &: \rho(\langle k_i, values_i\rangle) \rightarrow \langle k_i, f_j(values_i)\rangle \end{aligned}$$

Hadoop Implementation. Traditional MapReduce applications have one phase of maps followed by a reduce phase. Hadoop has an additional feature called *chaining* that allows job dependencies to be programmatically specified. Hadoop chaining in essence allows key-value pairs to be processed by multiple mappers and with a single reducer. The pattern could also be represented as multiple map phases followed by a single reduce phase but then the maps need to be able to handle multiple inputs that is also not supported in the model. Fundamentally, the MapReduce model does not allow for a selected set of tasks to be merged and thus the logic of selectively merging the appropriate inputs will need to be implemented in the application layer.

Data management. The data management is performed by a script. The script's first MapReduce phase is identical to the Data-Parallel pattern (4.1). For subsequent phases, the script must manually specify the two inputs needed by each mapper. However, in Hadoop it is not possible to control specific outputs from maps to specific reducers or to control parameters to different mappers. This needs to be handled in the appled in the application layer. We solved this problem by numbering the $\mathcal{W}_i$ output files at stage i with the mapper's task ID, creating files $F_0, F_1, \ldots, F_{\mathcal{W}_i - 1}$. Then, in stage $i + 1$ each mapper task ID t processes the two input files F_{2*t} and F_{2*t+1} from its parent. In other words, mapper 0 processes files F_0 and F_1, mapper 1 processes files F_2 and F_3, etc.

Note that in all MapReduces except the first, the input files are not assigned by Hadoop, but rather read manually by the mapper. Hadoop still requires that each mapper be assigned an input file. We implemented a work-around by creating an input directory with $\mathcal{W}$ dummy input files (containing only 1 line), each of which is set to be "non-splittable". An alternative approach, which we did not implement, would be to override Hadoop's input splitter to work with no input files.

Performance and Reliability. We note that

DRAFT

directly reading data from HDFS means Hadoop's scheduler has no insight into data locality. This can impact performance since this goes against the MapReduce model's key optimization of moving code to data. Therefore we expect that this workflow pattern would have poor performance for large input files.

4.5 Dynamic

In the dynamic pattern, a task might decide to spawn additional tasks at run-time if a particular condition is met. The number of tasks spawned, or whether the tasks are going to be spawned at all, is not known *a priori*. Any of the scientific ensemble patterns we have discussed might have dynamic elements resulting in run-time changes to the structure; for example, the Materials Genome workflows may need to re-run calculations at higher mesh resolutions (more *k-points*) if the crystal's atomic energies have not yet converged. We consider the pattern shown in Figure 1e as an example to evaluate the support for dynamic conditions in Apache Hadoop.

MapReduce Realization. There is no support in the MapReduce model to mark certain tasks as dynamic.

Hadoop Implementation. There is no built-in property in Hadoop that provides dynamic tasks in MapReduce jobs. This needs to be performed manually. We at first tried launching Hadoop jobs from within the reducers (i.e. within Hadoop jobs), but ran into a permissions issue. This stems from the fact that at execution time all jobs run under user `hadoop` instead of the original system user. Therefore, an external script was required to coordinate the workflow stages.

Data management. Similar to the Inverse Tree pattern (see 4.4), the script works by manipulating input and output files. In this case, we used a hardcoded "marker directory" to act as a queue of new jobs to be processed. While mappers are running they come to a point of decision whether to add a new set of tasks or not. If they decide to do so they create a file in the marker directory in which they write the path to the their output files. Logic inside the mapper can dynamically decide which paths to write. After the first set of MapReduce job completes, the script checks the marker directory for new files. If it finds any, then new MapReduce jobs are spawned for each output file it finds. A similar logic can be implemented in a reducer if the dynamic elements might be spawned from reduce tasks. This implementation is essentially a simplistic task manager since there is no support for dynamic tasks in Hadoop.

Performance and Reliability. The result of our work-around is that Hadoop's job scheduler has no knowledge of the intermediate data products and is not able to leverage data locality scheduling impacting performance and fault-tolerance.

4.6 Iterative

Many scientific processes require iterative simulations where the parameters for the next run is determined based on the results from the current run. For example, Uncertainty Quantification workflows may run the same simulation with many different sets of parameters that are chosen using Latin Hypercube or some other orthogonal sampling method in the large multi-dimensional parameter space. Such iterative processes are controlled by real-time decisions and there is no prior knowledge of the number of iterations or if the iteration will occur at all. Figure 1f shows the workflow pattern.

MapReduce Realization. The MapReduce model has no support for iterations.

Hadoop Implementation. The iterative pattern is very similar to the dynamic pattern, and required the same type of scripted management. For our specific pattern we implemented the conditional logic for subsequent iterations in the (single) reducer of the MapReduce job.

Data management. Input and output files were assigned to the jobs using the "marker directory" approach from the Dynamic workflow pattern. However, only the reducer (instead of all mappers) write a file in this directory.

Performance and Reliability. The QoS impact for this pattern is similar to the dynamic pattern.

5. DISCUSSION

There are a number of challenges when using Hadoop for scientific applications in current HPC environments [29], [14], [13]. In this section, we summarize our evaluation and discuss the gaps in current MapReduce frameworks in supporting scientific ensembles.

5.1 Pattern-based Analysis

In this section we summarize the implementation challenges we described in Section 4 using a three-point difficulty scale. We rate the Hadoop Implementation and Data Management categories

DRAFT

Workflow Type	Hadoop Impl.	Data Management	MR Realization	Perf. & Reliability	Total Score
Data Parallel	○	○	○	○	0
Single Input	○	◓	◓	◓	1.5
Scatter-Gather	◓	◓	◓	◓	2
Inverse Tree	◓	●	●	◓	3
Dynamic	●	○	●	●	3.5
Iteration	●	○	●	●	3.5

Key: ○= Easy/Minimal ◓= Moderate ●= Difficult/Significant

Table 1: Visual Summary of Hadoop Applicability by Workflow. To calculate the total score, we assign a value of 0 (easy), 0.5 (moderate) and 1 (difficult).

as `easy`, `moderately difficult`, or `very difficult`. Similarly, we rate the MapReduce Realization and Performance/Reliability categories as `minimal`, `moderate`, or `significant` impact. While our criteria for comparison is largely qualitative, we summarize the suitability of Apache Hadoop/MapReduce by assigning scores of 0, 0.5, 1 (higher score indicating more difficult or more impact) to each of the categories for better understanding of the relative difficulty of each of the categories and patterns. We intentionally use a distinct scoring scheme rather than continuous scoring since our intention is to understand the difficulty of managing scientific ensembles in Hadoop. A quantitative performance analysis is outside the scope of this discussion.

A visual summary of the scores for all type of workflows is given in Table 1. We sum the scores for each pattern in the column *Total score*. Our criteria for the rating is based on how easy it would be for an application scientist to plug-in their particular workflow in Hadoop. We could implement all patterns within Apache Hadoop but some of the patterns required significant programming or wrappers for manipulation. These manual mechanisms can be difficult for application scientists and also tend to be error-prone. The dynamic and iteration patterns present the most challenges for implementation in Hadoop. The inverse tree present some challenges while the data-parallel and single input are the easiest to implement with Apache Hadoop.

5.2 Language

Apache Hadoop and a number of the other open source MapReduce implementations are in Java. Scientific codes are often written in Fortran, C, C++ or use languages such as Python for analysis. The Hadoop streaming model allows one to create map-and-reduce jobs with any executable or script as the mapper and/or the reducer. This is the most suitable model for scientific applications that have years of code in place capturing complex scientific processes. For an application to be able to use this model, it needs to read input through *stdin* and send output to *stdout*. Thus, legacy applications are limited to using the streaming model that may not harness the full benefits of the MapReduce framework.

5.3 Filesystem.

The Hadoop Distributed File System (HDFS) does not have a POSIX compliant interface severely restricting the adoptability for legacy applications. HDFS's data locality features can be useful to applications that need to process large volumes of data. However, Hadoop considers only the data locality for a single file and does not handle applications that might have multiple input sets. It is possible to implement work-arounds by merging multiple input files into one before staging them into HDFS. This adds potentially large I/O overheads at the beginning of execution. It also reduces the usability of Hadoop, since it requires preprocessing or changes to the application.

5.4 Data Formats.

Apache Hadoop considers inputs as blocks of data where each map task gets a block of data. Scientific applications often work with files where the logical division of work is per file. Apache Hadoop has internal support to handle text data and certain other data formats. Hadoop also provides an extensible Java API to implement new data types in the framework. However this will require multiple Java classes to be implemented to define the

DRAFT

data format and define how to split a map task's inputs. Additionaly, code modules for readers and writers are needed to load the data and convert it to a key/value pair.

5.5 Diverse Tasks.

Traditionally, all mapper and reducer tasks are considered identical in function roughly working on equal sized workloads. Implementing different mapper and reducer requires logic in the tasks that differentiate the functionality since there is no easy way to specify it in the higher-level programming model. In addition, differences in inputs or algorithms can cause worker processing times to vary widely. This could result in timeouts and restarted tasks due to the speculative execution in Hadoop. If there is a large difference in processing time between tasks, this will cause load imbalance. This may dramatically impact the horizontal scalability.

5.6 Summary

Our analysis shows that it was possible to implement the general scientific ensembles patterns in Apache Hadoop with varying degrees of difficulty. However, in the case of some patterns we required a significant amount of custom code making it difficult for scientists to use Hadoop without significant programming expertise. Thus, there are a number of gaps and challenges when supporting scientific ensembles through current MapReduce implementations. Our analysis shows that there are opportunities to generalize some of the features required by applications into the programming model and execution framework. We summarize them here:

- Our initial analysis shows the formalizations of these workflow patterns in the MapReduce programming model. The MapReduce model was developed to process large volumes of data for operations such as search, data mining, log processing etc. The difficulty of implementing these patterns in MapReduce stem from the differences in the nature of the tasks compared to traditional MapReduce jobs. There is further research needed into appropriate programming model abstractions or extensions to MapReduce that can effectively support these ensemble models while providing the ease of programming and scaling that MapReduce provides. Support for task diversity, parameters, multiple inputs will need to be considered.

- Data locality is increasingly becoming important for scientific applications as data volumes increase due to advances in computing hardware and sensor technologies. Similarly, the cost of I/O transactions, size of I/O storage subsystems, and disparity in growth between computation, memory and storage on exascale systems will make data locality more critical. However new advances are needed to handle data locality of multiple files.

- There is additional support needed in execution frameworks that can handle dynamic tasks and iterations.

- Additionally, MapReduce implementations that are conducive to handling scientific codes and languages will be needed in the near future.

6. CONCLUSION

Many-task scientific ensembles require programming support for automatic scalability, fault-tolerance and dynamic adaptation in diverse resource environments. In this paper, we evaluate the difficulty in representing common scientific ensemble patterns - data-parallel, single input, multi-stage, inverse tree, dynamic and iterative in Apache Hadoop's MapReduce implementation. Our evaluation considers four criteria including differences from the MapReduce model, Hadoop implementation, data management and performance and reliability impact. While it was possible to implement all the patterns in the framework, the level of difficult varied with dynamic and iteration being the most difficult and the Data-parallel being the easiest. Our analysis identifies the gaps and challenges in MapReduce and specificially in Apache Hadoop in supporting many-task scientific ensembles. In addition, our evaluation will help applications identify if their workflows might be suitable to run in MapReduce frameworks.

7. REFERENCES

[1] MapReduce-MPI Library. http://www.sandia.gov/ sjplimp/mapreduce.html.
[2] Materials Genome. http://www.materialsgenome.org/.
[3] Oozie: Workflow engine for Hadoop. http://yahoo.github.com/oozie/.
[4] The DAKOTA Project Large-Scale Engineering Optimization and Uncertainty Analysis. http://dakota.sandia.gov.

DRAFT

[5] Triana The Open Source Problem Solving Environment. http://www.trianacode.org/index.html.

[6] Uncertainty Quantification. https://computation.llnl.gov/casc/uncertainty_quantification/.

[7] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[8] R. Brun. Root âĂŤ an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1-2):81–86, 1997.

[9] E. Deelman, J. Blythe, A. Gil, C. Kesselman, G. Mehta, S. Patil, M. hui Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. pages 11–20, 2004.

[10] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[11] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[12] M. Eldred, A. Giunta, and B. van Bloemen Waanders. Multilevel parallel optimization using massively parallel structural dynamics. *Structural and Multidisciplinary Optimization*, 27:97–109, 2004. 10.1007/s00158-003-0371-y.

[13] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. Benchmarking mapreduce implementations for application usage scenarios. *Grid 2011: 12th IEEE/ACM International Conference on Grid Computing*, 0:1–8, 2011.

[14] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. Mariane: Mapreduce implementation adapted for hpc environments. *Grid 2011: 12th IEEE/ACM International Conference on Grid Computing*, 0:1–8, 2011.

[15] M. P. I. Forum. Mpi: A message-passing interface standard, 1994.

[16] S. Ghemawat and J. Dean. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDIÕ04), San Francisco, CA, USA*, 2004.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[18] Y. Gu and R. L. Grossman. Sector and sphere: the design and implementation of a high-performance data cloud. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 367(1897):2429–2445, June 2009.

[19] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web Server issue):729–732, July 2006.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[21] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[22] P. M. Kelly, P. D. Coddington, and A. L. Wendelborn. Lambda Calculus as a Workflow Model. *Practice*, 21(July 2009):1999–2017, 2008.

[23] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM.

[24] H. Liu and D. Orban. Cloud mapreduce: A mapreduce implementation on top of a cloud operating system. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 464–474, Washington, DC, USA, 2011. IEEE Computer Society.

[25] V. M. Markowitz, F. Korzeniewski,

DRAFT

K. Palaniappan, E. Szeto, N. Ivanova, and N. C. Kyrpides. The integrated microbial genomes (img) system: a case study in biological data management. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1067–1078. VLDB Endowment, 2005.

[26] R. K. Menon, G. P. Bhat, and M. C. Schatz. Rapid parallel genome indexing with mapreduce. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 51–58, New York, NY, USA, 2011. ACM.

[27] I. Raicu, I. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1 –11, nov. 2008.

[28] L. Ramakrishnan and B. Plale. Multidimensional classification model for scientific workflow characteristics. In *1st International Workshop on Workflow Approaches to New Data-centric Science (WANDS'10)*, Indianapolis, IN, 06/2010 2010.

[29] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu. Magellan: experiences from a science cloud. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, pages 49–58, New York, NY, USA, 2011. ACM.

[30] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[31] D. A. Reed, C. da Lu, and C. L. Mendes. Reliability challenges in large systems. *Future Generation Computer Systems*, 22(3):293 – 302, 2006.

[32] M. C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics (Oxford, England)*, 25(11):1363–1369, June 2009.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –10, May 2010.

[34] J. Sroka, J. Hidders, P. Missier, and C. Goble. A formal semantics for the taverna 2 workflow model. *Journal of Computer and System Sciences*, 76(6):490 – 508, 2010.

[35] I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana Workflow Environment: Architecture and Applications. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 320–339. Springer, New York, Secaucus, NJ, USA, 2007.

[36] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[37] C. Zhang, H. De Sterck, M. Jaatun, G. Zhao, and C. Rong. CloudWF: A Computational Workflow System for Clouds Based on Hadoop. In M. G. Jaatun, G. Zhao, and C. Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 393–404. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

DRAFT