Mammoth: Autonomic Data Processing Framework for Scientific State-Transition Applications*

Xin Yang, Ze Yu, Min Li
xinyang,zeyu,minli@ufl.edu

Xiaolin Li
andyli@ece.ufl.edu

Scalable Software Systems Lab
University of Florida

ABSTRACT

Scientific computing is becoming increasingly data-intensive, and more high-impact discoveries are relying on efficient processing of big scientific data. The popular MapReduce framework such as Hadoop offers an alternative to conventional solutions (e.g., MPI or OpenMP). However, they perform moderately when processing state-transition applications. There are three key challenges: (1) these applications generate the inflated intermediate data that may saturate the network; (2) they may cause substantial synchronization overheads if not managed well; (3) dynamically evolving scientific phenomena result in heterogeneous data distributions, causing significant computation skews. In this paper, we propose Mammoth, an autonomic parallel data processing framework for scientific state-transition applications. Mammoth features a MapReduce-style programming model that is familiar to users. To address the challenges mentioned, it is further enhanced with a series of optimizations that parallelize the computation automatically and efficiently. We evaluate Mammoth via a weather prediction application with real-world datasets. The experimental evaluation demonstrates that Mammoth is competitive with the MPI-based solution and at least 30% faster than the optimized Hadoop-based solution.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed Systems*

General Terms

Design, Algorithm, Experimentation

*The work presented in this paper is supported in part by National Science Foundation (grants CAREER CCF-1128805 and PetaApps OCI-0904938).

CAC'13, August 5–9, 2013, Miami, FL, USA.

Keywords

MapReduce Systems, Autonomic Systems, Programming Models, Scientific Applications, Computation Skews

1. INTRODUCTION

Scientists are embracing the era of big data. More and more significant scientific discoveries are emerging from a growing wealth of scientific data, especially for those *scientific state-transition applications*, such as the severe weather prediction and ocean prediction problems [10, 24]. These applications maintain scientific *states* that evolve when new *observations* become available. Observations are newly collected data describing current states (e.g., temperature, humidity, atmosphere pressure, diffusivity, viscosity, etc.). These observations are then incorporated into current states by using state-transition algorithms defined by scientists and trigger the states to evolve.

Scientists depend heavily on techniques of message passing (e.g., MPI) and shared memory (e.g., OpenMP) for parallelizing state-transition applications. However, efficient implementations require interdisciplinary knowledge of scientific algorithms, message passing programming models, parallel/distributed systems, large-scale data management, and so on. It is desirable to introduce an approach for scientists to express their algorithms easily and abstract the parallelization work conveniently.

Fortunately, we find that most state-transition algorithms can be decomposed into two logical phases. One defines how an observation generates influences, or *transitions*, to current states; the other specifies how transitions from multiple observations are aggregated and finalizes new states. Following this pattern, we can parallelize these applications using the popular MapReduce strategy [7], i.e., running user-defined state-transition algorithms in the map function for every observation to emit transitions, and aggregating transitions in the reduce function to finalize new states.

It remains challenging for applying existing MapReduce frameworks such as Hadoop [27] to state-transition applications. One main challenge comes from the state-transition algorithms that are generally output-inflating. One observation may trigger multiple transitions in terms of the intermediate data. This multifold expansion of the intermediate data breaks a critical assumption in MapReduce frameworks: the data volume is expected to shrink between phases [1]. For example, Hadoop expects a smaller volume of the intermediate data, compared with the input, in the shuffling phase. In our scenario, the network will be saturated by the inflated intermediate data produced by

the state-transition algorithms. For state-transition applications, placing combiners before shuffling is not applicable, since algorithms in these applications need all the individual transitions to finalize new states.

Another challenge comes from the "halo-exchange" computing model. State-transition applications are commonly modeled in a multidimensional coordinate system, and transitions only propagate to neighbors. It is desirable to range partition[1] the inputs into blocks so that interactions of propagating transitions can be performed locally, rather than being emitted globally. MapReduce frameworks simply use the global synchronization (i.e., shuffling), without considering fine-grained halo-style data dependencies.

In addition, observations reflect fluctuations of states and typically exist in a few local regions where significant phenomena occur, resulting in a sparse array. The range partitioning strategy causes computation skews since blocks contain different amounts of observations.

In this paper, we present Mammoth, an autonomic parallel data processing framework for state-transition applications. Mammoth intends to provide a user-friendly programming model similar to MapReduce for scientists to program algorithms easily, while automating the underlying parallelization work efficiently. More specifically, the programming APIs of **trigger** and **aggregate** are designed in accordance with the execution logic of state-transition algorithms. On the other hand, several crucial performance optimizations are included in Mammoth's runtime to automate the parallel execution. The input data of states and observations are partitioned for data parallelism. Data dependencies among partitioned blocks are built with the only user intervention of specifying the coverage range of observations, which is application-specific. Computation skews are detected and mitigated following the autonomic paradigm of "*sensing, deciding, and acting*" [22, 6] in literature. In summary, our work makes the following contributions:

1. We propose the Mammoth programming model: a MapReduce style programming model supporting scientific state-transition applications.

2. We design and implement the Mammoth runtime consisting of three key components: an operator placement coordinator that addresses the inflated intermediate data, a dependency-aware scheduler that builds the halo-style data dependencies automatically and shuffles the intermediate data locally, and an adaptive and autonomic load balancer that detects and mitigates computation skews.

3. We evaluate the performance of Mammoth using the real-world weather prediction application and datasets. Mammoth shows a comparable performance to the MPI-based solution and is at least 30% faster than the optimized Hadoop-based solution.

The rest of the paper is organized as follows. Section 2 describes the motivation applications. Section 3 outlines the Mammoth architecture. Section 4 presents the Mammoth programming model for users to program state-transition applications easily. We present Mammoth's runtime optimizations in Section 5. The experimental evaluation is given in Section 6. We survey the related work in Section 7 and conclude in Section 8.

[1]MapReduce frameworks such as Hadoop support range partitioning. However, it is for the intermediate data rather than the inputs.

2. MOTIVATION APPLICATIONS

In this section, we present our target applications and summarize their commonalities for designing an efficient framework to support them.

Numerical Weather Prediction. Numerical weather prediction applications [10] are used to model and predict the evolution of the meteorological system. In these applications, meteorological states are represented as multidimensional points embedded in a 3D bounding box. A typical schema describing the meteorological state is as follows:

$$\textbf{Point(x,y,z,v1,v2, ...)} \quad (1)$$

In this schema, the first three elements represent the location of the point, and the remaining elements tag the point with meteorological parameters, such as temperature, humidity, and atmospheric pressure.

Observations are collected by scientific instruments (e.g., radars, sensors) deployed in the observational space. These observations describe and drive the current states to transit. When new observations become available, they assimilate with the states using numerical prediction algorithms and finalize new states for the current analytical cycle.

The numerical weather prediction example perfectly represents state-transition applications. Meteorologists use various sophisticated estimation algorithms such as Ensemble Kalman Filter (EnKF) and Ensemble Square Root Filter (EnSRF) to assimilate observations with states accurately. We will use this example to illustrate our design of Mammoth throughout the remaining sections.

Ocean Prediction. The ocean prediction [24] is similar to the numerical weather prediction. Ideally, parameters collected from the observational network can be directly used to measure and predict the states of the ocean system. But in reality, errors are normal in such a large-scale oceanic system, e.g., the inaccurate observational data due to environmental fluctuations, imperfect measuring models. As a result, the acquired observational data needs to be calibrated by assimilating with previous oceanic states (called *data assimilation*), before infusing into ocean prediction models.

The data assimilation is an estimation procedure existing in the numerical weather prediction as well. Besides the EnKF and the EnSRF algorithms, some straightforward estimation approaches such as the direct minimization and maximum likelihood are feasible in some particular cases.

General State-Transition Applications. Generally, state-transition applications share the following commonalities. (1) The scientific states reside in a coordinate space. (2) New observations trigger the states to transit in a Markov-like manner (by Markov we mean that the states at time step t_{i+1} only depend on the states at t_i and the observations at t_{i+1}). (3) The state-transition algorithm typically consists of two logical phases: one defines how the states and the observations generate transitions; the other determines how transitions are aggregated to finalize new states for the next time step.

3. ARCHITECTURE

Figure 1 illustrates the architecture of Mammoth, which inherits the basic system design and architecture of Hadoop

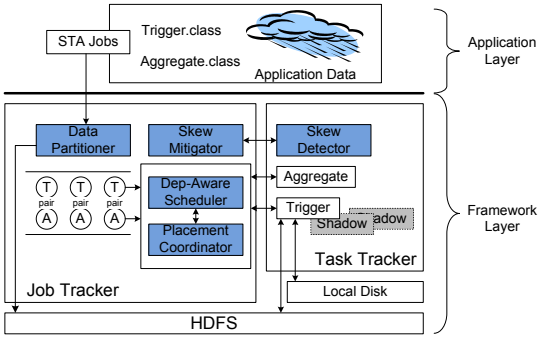

Figure 1: The architecture of Mammoth.

(ver. 0.20.203). HDFS is used for storing data. Jobs of state-transition applications are split and managed as a number of trigger and aggregate tasks.

To address the aforementioned challenges, Mammoth exposes its functionality to users through a programming model (Section 4) and adds several new components in Hadoop (colored boxes in Figure 1): a data partitioner, an operator placement coordinator, a dependency-aware scheduler, and an adaptive and autonomic load balancer. The data partitioner uniformly partitions the state-transition application as well as its data into blocks. The operator placement coordinator pairs a trigger task and an aggregate task for every block, and places them at the same node (Section 5.1). The dependency-aware scheduler is responsible for scheduling trigger tasks and aggregate tasks according to some scheduling strategies (Section 5.2). The adaptive and autonomic load balancer consists of a skew detector for detecting computation skews in trigger tasks, and a skew mitigator for alleviating the detected computation skews (Section 5.3).

4. PROGRAMMING MODEL

In this section, we present the Mammoth programming model exposed to users for expressing the state-transition algorithm.

4.1 Basic Programming APIs

From the user's perspective, the state-transition algorithm can be abstracted as the following equation:

$$\mathcal{S}^{i+1} = \mathbf{F}(\mathcal{S}^i, \mathcal{O}^{i+1}).$$

The transition of the state $\mathcal{S}$ from iteration i to iteration $i+1$ is triggered by the new observation $\mathcal{O}^{i+1}$. The operator $\mathbf{F}$ is defined by users to describe how the new observation triggers the transition of the state (i.e., state-transition algorithms). The iterative procedure in state-transition applications is driven by the incoming observations, rather than being controlled by an explicit iteration proceeding or termination logic. This characteristic of data-driven iteration makes Mammoth different from other frameworks such as HaLoop [5], Twister [9], and PrIter [31], where the iteration is a user-defined operation and terminates if predefined conditions are met or the computation starts to converge.

To enable easy programming, and more importantly, parallelize users' algorithms, we decompose the operator $\mathbf{F}$ into two APIs named **trigger** and **aggregate**. The formalized definitions of the APIs are:

$$\textbf{trigger}: \quad (\mathcal{S}^i, \mathcal{O}^{i+1}) \rightarrow \{\mathcal{T}^{i+1}\};$$
$$\textbf{aggregate}: \quad (\{\mathcal{T}^{i+1}\}, \mathcal{S}^i) \rightarrow \mathcal{S}^{i+1}.$$

Table 1: Summary of Notations

$\mathcal{S}^i$	The state in the i^{th} iteration
$\mathcal{O}^i$	The observation in the i^{th} iteration
$\mathcal{T}^i$	The set of transitions triggered by the observation in the i^{th} iteration
$\mathbf{F}$	The functional operation that incorporates the state and the observation at a point

The notations used in these APIs are summarized in Table 1. Mammoth runs these functions as follows: (1) partitions the application as well as the data into blocks (done by Mammoth's data partitioner which is not shown here), (2) within each block, applies the **trigger** function to each point (if there exists observation coverage) and generates a set of transitions, (3) applies the **aggregate** function to aggregate the transitions to produce a new state. By deliberately using the MapReduce-style APIs, Mammoth leverages the popularity of the MapReduce programming model and is user-friendly. Different from the map and the reduce APIs, the **trigger** and the **aggregate** APIs offer users the programming experience from the perspective of points instead of key-value pairs.

Figure 2 illustrates an example describing how the transitions from several observations are aggregated in the numerical weather prediction application. The circles represent the coverage of the observations and have an overlap at the gray point. The application as well as its data is partitioned into four equal blocks. In the **trigger** function, users express how the observation generates transitions affecting itself and the other points in the coverage. Mammoth APIs shield users from having to consider how to propagate transitions across blocks. In the **aggregate** function, users define how transitions at the same point (the gray point) are aggregated to generate the updated state for this point.

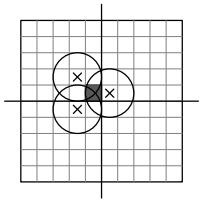

Figure 2: Transitions are aggregated at overlapping points.

4.2 Dependency API

Modeling data dependencies in a user-friendly manner is nontrivial. A straightforward way is asking users to specify the complex dependency graph on the data manually. However, the overhead is overwhelming, both for users to build such a complex graph and for the framework to maintain it. Mammoth leverages the halo-style data dependencies in state-transition applications and builds data dependencies at the block level automatically. This reduces the number of vertices and the complexity of the graph significantly. Mammoth only asks the *Influence Radius (IR)* (the radius of the observation coverage) from users using the **Initialize** API. It uses the *IR* value to determine the partitioning granularity and builds data dependencies for blocks. Controlling the partitioning granularity is for preventing transitions reaching beyond neighbor blocks. We believe that it is easy for users to specify the *IR* value since it only depends on accuracy requirements of their applications.

5. RUNTIME OPTIMIZATIONS

In this section, we present the optimization techniques applied in Mammoth's runtime: an operator placement coordinator that jointly places the trigger and the aggregate tasks for reducing the intermediate data shuffled in the network, a dependency-aware scheduler that provides the fine-grained scheduling for the trigger and the aggregate tasks, and an adaptive load balancer for detecting and mitigating computation skews in the trigger tasks automatically.

5.1 Operator Placement Coordinator

State-transition applications generate the inflated intermediate data that will saturate the network. The reason is that, an observation in the state-transition application generally covers an expanded area, which indicates that the transitions generated by the observation need to be propagated to other points. The range partitioning schema opens an opportunity for Mammoth. For the observation, it is unnecessary to propagate the transitions it generates that are in the same block (*local* block). If the observation's coverage expands across the local block, propagating the transitions beyond the boundaries of the local block will be sufficient.

As a result, the goal of the operator placement coordinator is to appropriately place the trigger and the aggregate tasks for reducing the volume of the in-network intermediate data.

Mammoth intends to prevent emitting the transitions bounded in the local block to the network. However, processing the observation and aggregating the transitions for updating the state are separately performed in the trigger and the aggregate tasks. Namely, a data transfer operation is inevitable.

Mammoth uses the local disk instead of the network for transferring this portion of the intermediate data. It allows the trigger task to store the local transitions at the local disk, and notifies the following aggregate task to fetch them thereafter. To use the local disk as the medium to transfer the transitions, the coordinator uses a technique named *joint placement* that places the trigger task and the aggregate task together. To be specific, the coordinator (1) launches the same number of trigger tasks and aggregate tasks as that of the data blocks, (2) pairs one trigger task and one aggregate task, and (3) assigns one block to each <trigger, aggregate> task pair and schedules the task pair to the node where the block locates.

5.2 Dependency-Aware Scheduler

Mammoth range partitions the application and its data, and the partitioned blocks feature halo-style spatial data dependencies. Blocks can be processed independently, and results can be produced by aggregating the intermediate data from neighbor blocks only. Remember that Mammoth controls the partitioning granularity according to the IR value specified by users in the ***Initialize*** API, so data dependencies do not exist among non-adjacent blocks.

As a result, it is unnecessary for an aggregate task to wait for the completion of the trigger tasks responsible for the non-adjacent blocks. Such halo-style spatial data dependencies imply the local synchronization, which can substitute the widely-applied global synchronization in MapReduce frameworks.

Mammoth uses a dependency-aware scheduler to enable the local synchronization. The scheduler allows an aggregate task to start as long as its dependent trigger tasks have completed. A new problem arises that the scheduling orders of trigger tasks or aggregate tasks become critical. For example, if a trigger task depended on by most aggregate tasks is scheduled last, there is probably no performance difference between the local synchronization and the global synchronization. Thus, the dependency-aware scheduler includes two scheduling considerations: the scheduling order of trigger tasks and that of aggregate tasks.

5.2.1 Scheduling Order of Trigger Tasks

The goal of organizing the scheduling order of trigger tasks is to maximize the number of runnable aggregate tasks in the trigger phase. The dependency-aware scheduler applies the following three scheduling orders for trigger tasks, and we illustrate them in Figure 3.

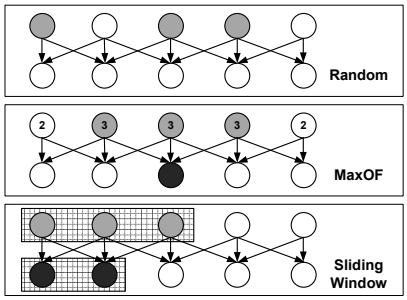

Figure 3: The scheduling orders of trigger tasks.

Random: Trigger tasks are scheduled randomly. This case is used as the baseline to evaluate other scheduling orders.

Maximum Outdegree First (MaxOF): We define the *outdegree* of a trigger task as the number of aggregate tasks that depend on it. Scheduling a trigger task that has a larger outdegree early can satisfy more aggregate tasks' dependency requirements.

Sliding Window: The sliding window scheduling strategy is conducted as follows. First, the blocks and the <trigger, aggregate> task pairs are indexed following the same continuous order, for example, the wavefront order or the line-scanning order. Then, the trigger tasks and the aggregate tasks are scheduled according to their indexing orders. We can imagine the behavior of the sliding window scheduling: a sliding window representing the completed trigger tasks is shifting from the first trigger task to the last one, as the aggregate tasks are scheduled using the same order, another sliding window of the completed aggregate tasks is catching up (as illustrated in Figure 3). Following the same indexing/scheduling order makes the two sliding windows shift continuously. The sizes of the two sliding windows are determined by the numbers of trigger slots and aggregate slots in the system, respectively.

In Mammoth, we apply the wavefront indexing/scheduling order since it is pervasively used for applications having the halo-style spatial data dependencies.

5.2.2 Scheduling Order of Aggregate Tasks

The goal of considering the scheduling order of aggregate tasks is to improve the utilization of aggregate slots. In this paper, we define the utilization of aggregate slots as the ratio of the total running time of aggregate tasks to the total opening time of aggregate slots.

The local synchronization allows an aggregate task to start if its dependent trigger tasks have completed, as long as it

can acquire a free aggregate slot. But as the number of aggregate slots is much smaller than that of aggregate tasks, deciding which aggregate tasks to be scheduled when there are free slots poses a challenge to the dependency-aware scheduler. If an aggregate task is scheduled but some of its dependent trigger tasks have not completed yet, the aggregate task needs to wait while it is occupying the valuable slot resource. As a result, other aggregate tasks ready to run will be blocked and consequently reduce the utilization of slot resources.

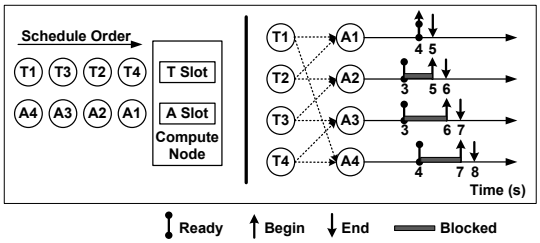

Figure 4: The head-of-line blocking in aggregate slots.

This is the head-of-line (HOL) blocking [14], and we explain it with a simple example illustrated in Figure 4. We are going to schedule four trigger tasks and four aggregate tasks to a compute node with one trigger slot (T slot) and one aggregate slot (A slot). Suppose every trigger or aggregate task takes 1 second to complete. The dependency relationship between trigger tasks and aggregate tasks is illustrated by dash lines. The four trigger tasks are scheduled to take the only T slot one by one in the order of T4, T2, T3, T1, and the four aggregate tasks are scheduled in the order of A1, A2, A3, A4. We can observe that task A1 is stalled in the only aggregate slot for 4 seconds to wait for the completion of the last trigger task T1, the subsequent aggregate tasks are blocked although their dependent trigger tasks have already completed.

Mammoth uses a conservative scheduling strategy to mitigate the HOL blocking in aggregate slots. When an "aggregate slot available" message is reported to the scheduler, it would rather leave the aggregate slot empty to wait for the satisfactory aggregate tasks whose dependent trigger tasks have already been scheduled. This strategy is similar to the delay scheduling [29] in Hadoop while the delay scheduling is designed for balancing locality and fairness among jobs.

The HOL avoidance scheduling strategy is conservative because aggregate slots are wasted at the beginning due to waiting for dependency-satisfactory aggregate tasks. Such waiting might be utilized by dependency-unsatisfactory aggregate tasks to occupy for pulling the intermediate data, which is not trivial in state-transition applications. Even though, evaluation results show that the HOL avoidance strategy is rather effective in improving the utilization of slot resources.

5.3 Load Balancer

Computation skews exist when running state-transition applications in Mammoth. A widely applied approach to solving the skew problem in literature is partitioning the input carefully before execution, so that each task gets a similar amount of workloads [17, 16]. However, it is not an ideal solution in our case. First, as state-transition applications are driven by the iteratively evolving data, computation skews often occur in different parts of the data, making the optimal partitioning plan for the current iteration becomes suboptimal for the following iterations. Second, computing an optimal partitioning plan for every iteration is too time-consuming to be feasible. The last reason is that the input of state-transition applications is from heterogeneous sources. The data of states requires more IOs while the data of observations represents the demand on the CPU resource. It is hard to reconcile an optimal partitioning plan that can satisfy both the I/O and the CPU requirements.

Mammoth uses an adaptive and autonomic load balancer to address the skew problem. The load balancer consists of two core components: a skew detector and a skew mitigator. The skew detector monitors the execution of launched tasks and identifies stragglers on the fly, and then informs the skew mitigator to activate shadow tasks to shed the load off the stragglers. Controls are conducted autonomically without any needs for users to calculate optimal partitioning plans for the data [17], speculatively execute tasks, sample the input for predicting better execution plans [20]. Note that in this paper, we only consider the skew problem for trigger tasks where the state-transition computation is mainly performed.

5.3.1 Skew Detector

The skew detector is a central component that decides which task is overloaded and by how much. It pulls execution statuses periodically from running tasks and uses them along with the statistics of finished tasks to make decisions.

The skew detector detects skews as follows. It first computes the average completion time T_{avg} of finished tasks. In the meanwhile, it keeps track of the running task's, say task i, speed S^i and predicts its remaining time T^i_{rem} according to the following formula: $T^i_{rem} = (N^i_{all} - n^i_{done})/S^i$, where N^i_{all} is the number of all the observations allocated for task i, n^i_{done} is the number of observations already processed by task i. The speed is defined as the number of observations that a task can process per second. Then the decision is made by testing the following inequality: $T^i_{rem} - (T_{avg} - t^i_{done}) > T_{avg}$, where $t^i_{done} = n^i_{done}/S^i$. The intuition is: if the estimated remaining time of task i is longer than the average completion time, it is beneficial to split its remaining workloads and allocate them to multiple tasks. Note that if it is split, task i itself needs to process $T_{avg} * S^i$ workloads at least.

Aggressiveness Control. The detecting speed is important. The faster the skew detector is, the more efficient the skew mitigator will be. However, being too aggressive tends to cause many false-positive detections and incur the unnecessary overhead. As a result, it is crucial to find a sweet point for the skew detector to be efficient. We follow two simple but effective strategies to control the skew detector's aggressiveness.

First, we set a confidence factor δ to prevent the skew detector relying on uncertain statistics due to insufficient samples. The simplest way is to define the confidence factor δ as the ratio of the number of finished tasks to that of total tasks. However, it might not be accurate. The early completed tasks may also include stragglers, thus using the average completion time of these tasks to detect skews is conservative. So the skew detector will not mark any tasks as skews until it has detected sufficient non-skewed tasks

have completed, i.e., half of the trigger tasks scheduled in the first wave report completion.

Second, we adaptively control the skew detector's sensitiveness. A task is marked as skew only if its completion time is predicted to be γ times longer than the average. This sensitiveness factor γ is adaptively tuned by the skew detector according to the following criterion. As the number of tasks is typically larger than that of slots, tasks are not launched at the same time but in waves [15, 25]. The key observation here is that the skew occurring in the early waves are "hidden" behind (overlap with) the following waves, thus γ is going to be set conservatively during these early waves. As the job execution approaches the end of the trigger phase, the skew effect is aggravated, and accelerating these skew tasks will definitely reduce the completion time of the trigger phase and consequently speed up the job execution. Thus γ is adjusted to be aggressive to accelerate "the last hurrah".

5.3.2 Skew Mitigator

Informed by the skew detector, the skew mitigator sheds the load off the skew. There are three major design goals of the skew mitigator. First, we want to quickly respond to the detected skews. Second, when launching skew-mitigating tasks, we want to minimize the interference to other running tasks thus minimizing the visible side effect to users. Last, we want to make the skew mitigation of a task transparent to its downstream work.

To meet these goals, the skew mitigator launches multiple shadow tasks for the skew task. To be specific, when a new task is launched, the skew mitigator creates multiple replicas of this task. The main difference between shadow tasks and normal tasks is that, when there is no skew, shadow tasks do not really run user-defined functions, thus producing no output and incurring little overhead. When a skew is detected, say task i, the skew mitigator first sends a stop signal s_i to task i, informs task i to stop after processing s_i input records. It then sends start signals to task i's shadow tasks, activates them, and tells them where to start processing. Re-partitioning the skew task i's unprocessed inputs and allocating them to shadow tasks are done by the skew detector when detecting skews. After shadow tasks complete, the skew mitigator stitches their outputs together according the order of their inputs and commits task i as usual. There are several merits of this design: first it minimizes the warm-up time of shadow tasks by overlapping them with the normal execution; second, shadow tasks are lightweight and interfere little with other running tasks if no skew occurs; finally, it is transparent to other parts of the system.

The next question is where to put shadow tasks. The key idea is to take the advantage of data replication of the underlying HDFS. As HDFS exploits replica to achieve fault tolerance, we can put shadow tasks where the corresponding input data is replicated, so that shadow tasks can embrace the data locality while running. We do not cap the number of shadows a task could have and set the default number of shadow tasks to the replication level of the underlying HDFS. If the skew detector really needs more shadow tasks than those launched by default (say it needs k shadow tasks where $k > r$, and r is the replication factor), the skew mitigator exploits a greedy strategy. It first launches r shadow tasks on the r compute nodes hosting replicas, and then dispatches the remaining $k-r$ shadow tasks to the least loaded $k-r$ nodes.

Algorithm 1 Skew Detect/Mitigate Algorithm

Require: *stat*: statistical info
  S: statuses of running tasks
  δ: confidence factor
  γ: sensitiveness factor
Ensure: $List < task > T$: set of shadow tasks
  T $\leftarrow \varnothing$
  if *stat*.confidence() $> \delta$ **then**
    $T_{avg} \leftarrow$ *stat*.computeAvg()
    for all task s_i in S **do**
      $t^i_{done} \leftarrow s_i$.getElapsedTime()
      $T^i_{rem} \leftarrow s_i$.getRemainingTime()
      if $T^i_{rem} > (1+\gamma)T_{avg} - t^i_{done}$ **then**
        $k \leftarrow \lceil \frac{T^i_{rem}}{T_{avg}} \rceil$
        $T^{'} \leftarrow$ activateShadowTasks(s_i,k)
        $T \leftarrow T \cup T^{'}$
      end if
    end for
  end if
  return T

We summarize the adaptive load balancer's algorithm of detecting and mitigating skews in Algorithm 1. The algorithm accepts inputs of the statistical information of all the completed and running tasks *stat*, running tasks S, the confidence factor δ, and the sensitiveness factor γ, and returns the set of shadow tasks to be activated T. The statistical information is checked first for its reliability, then every running task in S is checked and activates its shadow tasks by adding them into T, if the task is a skew.

6. EVALUATION

In this section, we present the experimental results for a representative state-transition application implemented using Mammoth, Hadoop, and MPI. The goals of our evaluation are two-fold: (1) we compare Mammoth with Hadoop and MPI on the state-transition workload; (2) we evaluate the optimization techniques in Mammoth runtime.

6.1 Setup

Applications and Datasets. As Mammoth is a specialized framework, we use a representative state-transition application, the weather data assimilation [28], in the evaluation. The application models states of the environment and observations of the atmosphere in a 3D bounding box. The data assimilation algorithm is conducted layer by layer along the z-axis and point by point within each layer. At every point, the state and the observation are assimilated, and new states are generated, which update not only the point itself but also the neighbor points.

Two datasets are used, and each consists of states and observations represented as 3D arrays and stored as HDF4 files. Observations reflect fluctuations of the atmosphere and indicate computations. As shown in Figure 5, the observations in *dataset 1* have a hotspot distribution in the center, indicating severe computation skews. In contrast, the observations in *dataset 2* distribute broadly, indicating less computation skews but a large amount of computations. Each dataset contains 50 snapshots, but we only use one snapshot

for the following experiments due to the limited hardware resources.

For dataset 1, the snapshot has twelve layers, and each layer is represented as a 1323×963 2D array and consists of 13.2GB data of states and 25MB data of observations. Processing one layer produces the same size of outputs as that of inputs, namely, 158GB inputs and 158GB outputs for twelve layers. The snapshot in dataset 2 has twelve layers as well. Each layer is represented as a 400×400 2D array with 2.2GB data of states and 3MB data of observations. That are 26GB inputs and 26GB outputs for all the twelve layers. In summary, the state-transition workload with dataset 1 is more data-intensive and suggests severe computation skews, and that with dataset 2 is more computation-intensive.

For both datasets, we use Mammoth to uniformly partition each layer into 100 blocks (132×96 and 40×40 per block, respectively). For Mammoth and Hadoop, the blocks are distributed across the cluster through HDFS. The "dfs.block.size" is set to 256MB to prevent splitting the blocks of dataset 1. For the MPI experiment, the blocks are stored in a dedicated Amazon EBS volume mounted onto a shared directory using NFS.

Implementation. The implementation work of Mammoth is on top of Hadoop (ver. 0.20.203) and consists of two major parts: the data adaptor and the runtime optimizer. The data adaptor preprocesses the input to support the abstraction of the **trigger** and the **aggregate** programming APIs. It converts the input from some specific formats (e.g., the HDF4 format for scientific multidimensional arrays) to the key-value format, and partitions the input into regular blocks before storing it in the HDFS. For each state-transition job, the runtime optimizer launches trigger and aggregate tasks according to the number of blocks, one pair of trigger and aggregate tasks for each block. The trigger and aggregate tasks are indexed using their corresponding block's index. Shadow trigger tasks are created but not launched unless computation skews in trigger tasks are detected. Trigger tasks piggyback their task execution progresses in heartbeats for the jobtrack to detect computation skews. Shadow tasks are activated if the corresponding trigger tasks are detected as skews. Every aggregate task monitors the "trigger task completes" event and starts to process the intermediate data as long as: (1) all of its neighbor trigger tasks are completed, and (2) all the intermediate data required is pulled.

Implementing the data assimilation application using Mammoth consists of: initializing Mammoth with the paths to the data files of observations and states and the influence radius in the **initialize** API; programming the data assimilation algorithm from the perspective of a single point in the **trigger** API; and defining how the transitions having the same coordinate are aggregated in the **aggregate** API.

The Hadoop implementation is similar. The data assimilation algorithm is simulated in the map function, and how the overlapped transitions are aggregated is defined in the reduce function. The default FileInputFormat is used, and the coordinate information contained in each input line is used as the output key of map tasks and the input key of reduce tasks.

The MPI implementation follows that in [28]. In our scenario, the input is uniformly partitioned to 100 blocks, one for each MPI processor. Due to the one-file-per-processor I/O pattern, we use the standard fstream library in C++ instead of parallel I/O libraries such as MPI-IO for I/Os. Each processor computes the block point by point: assimilating the observation and the state, and producing transitions. Processors exchange transitions in the "halo" with others responsible for the neighbor blocks using asynchronous communications (i.e., MPI_Irecv and MPI_Isend).

For the purpose of clarity, we refer to the three implementations using the names of Mammoth, Hadoop, and MPI, respectively, in the experiments.

Computer Cluster. Experiments are conducted in Amazon EC2. Due to the limited budget, we use 20 "m1.large" instances (40 cores) to build a virtual cluster for all the experiments. Each instance features 2 virtual Xeon 2.26GHz cores, 7.5GB memory, and 850GB ephemeral storage. All the instances reside in the same availability zone (us-east-1a). The AMI installed in the instances is Ubuntu 11.10 x86-64.

To mitigate the possible randomness caused by the virtualization resources in Amazon EC2, every experiment is repeated ten times, and the results are reported as average values with error bars. Note that the error bars are pretty small in some figures.

6.2 Results

6.2.1 *Performance Comparison*

As we mentioned previously, the state-transition application generates the inflated intermediate data in Hadoop, which will saturate the network at shuffling and overload the reduce tasks. Even though, we can distribute the inflated intermediate data to more reduce tasks.

Figure 6 shows the performance change with respect to the number of reduce tasks in Hadoop. The improvement is obvious when launching more reduce tasks until the turning point: the number of reduce tasks equals to the cluster size. After that, the overhead of bookkeeping reduce tasks starts to overweigh the performance gain of distributing the intermediate data. In the following comparison, we will set the number of reduce tasks to the cluster size for the optimized performance of Hadoop.

Figure 7 and Figure 8 show the results comparing Mammoth with Hadoop and MPI on the two datasets. Mammoth and Hadoop illustrate good scalability with respect to the cluster size, no matter the state-transition workload is data-intensive or computation-intensive. However, the performance of MPI is constant. That is because MPI directs all the I/Os to the centralized storage of the EBS volume, and its performance is bounded by the I/O. In contrast, Mammoth and Hadoop use HDFS that distributes the input across the cluster. They start to outperform MPI at some "sweet points". It is predictable that the performance improvement will be more significant with the increase of the cluster size.

MPI's I/O performance can be improved by techniques such as installing fast I/O devices and high-performance file systems, using libraries supporting parallel I/Os, or chunking and distributing data explicitly. However, as we mentioned before, they require considerable interdisciplinary knowledge for scientists. Mammoth and Hadoop encapsulate such tedious work and present an easy-to-program yet performance-efficient experience.

Although the Hadoop setting is already optimized, Mammoth consistently outperforms Hadoop by at least 30% due

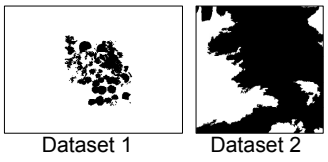

Figure 5: Observations in the datasets.

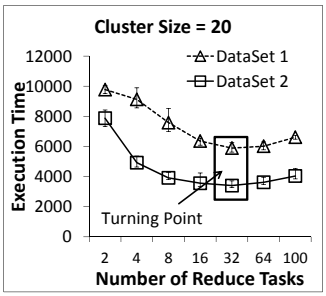

Figure 6: Turning point of the number of reduce tasks

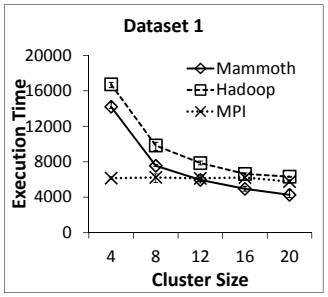

Figure 7: Mammoth vs. Hadoop vs. MPI, Dataset 1

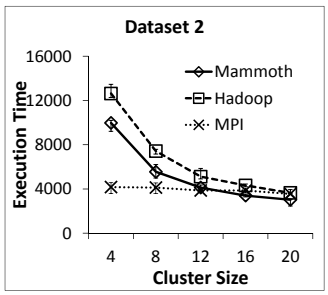

Figure 8: Mammoth vs. Hadoop vs. MPI, Dataset 2

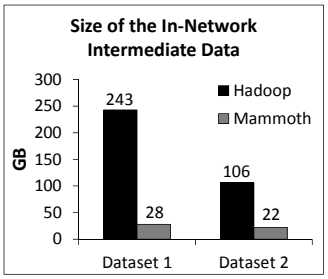

Figure 9: In-network intermediate data size

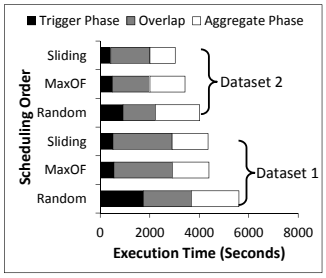

Figure 10: The scheduling orders of trigger tasks

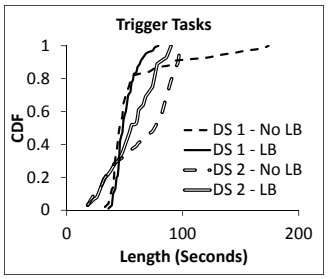

Figure 11: CDF of trigger tasks

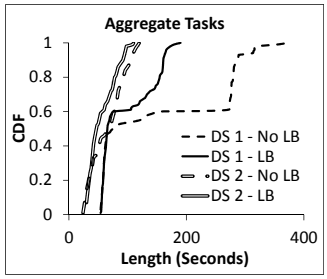

Figure 12: CDF of aggregate tasks

to the optimization techniques specialized for state-transition applications. In the following, we evaluate and analyze each optimization, respectively.

6.2.2 *Optimization Techniques*

Joint Placement. The joint placement optimization is responsible for reducing the volume of the intermediate data shuffled through network. Figure 9 shows the comparison results between Hadoop and Mammoth. We can see a significant reduction of the volume of the in-network intermediate data, from 243GB in Hadoop to 28GB in Mammoth for dataset 1, and from 106GB to 22GB for dataset 2. More than 200GB data for dataset 1 and 80GB data for dataset 2 is transferred from trigger tasks to aggregate tasks using local disks. Although the aggregate task needs to combine the intermediate data from the network and that at the local disk, the combining is much faster than shuffling massive data through the network.

Dependency-Aware Scheduling. The dependency-aware scheduling leverages the halo-style data dependencies in state-transition applications and allows the aggregate task to proceed as long as the trigger tasks in its halo have completed.

Figure 10 illustrates the performance change of applying different scheduling orders of trigger tasks in Mammoth. Breaking the all-to-all communication pattern allows a considerable amount of aggregate tasks to start before the trigger phase ends. In Figure 10, every bar is divided into three parts to represent the trigger phase, the aggregate phase, and the overlap of the two. Note that the aggregate phase starts from the actual processing of the intermediate data, otherwise (i.e., starts when the first aggregate task is launched) the aggregate phase will enclose the entire trigger phase. We can see that the MaxOF and the sliding window strategies improve the random strategy by about 25% for both datasets. That is because the MaxOF and the sliding window strategies show more overlaps of the trigger and the aggregate phases. The sliding window strategy does not outperform the MaxOF strategy significantly, since most aggregate tasks in our application have five (four side-neighbor blocks and one local block) dependent trigger tasks, and the scheduling orders of the two strategies are similar.

Table 2: the HOL avoidance scheduling

	Dataset 1		Dataset 2	
	No HOL	HOL	No HOL	HOL
Execution Time	5256s	4360s	3295s	3031s
Utilization	61.4%	66.7%	62.5%	77.8%

For the scheduling order of aggregate tasks, we evaluate the effectiveness of the HOL avoidance scheduling on reducing the execution time and improving the utilization of aggregate slots. Table 2 shows that the HOL avoidance scheduling shortens the execution time by 13% for dataset 1 and 8% for dataset 2. In the experiment, we can observe a few aggregate tasks are stalled after occupying the aggregate slots, if the HOL avoidance scheduling is disabled.

The HOL avoidance scheduling also improves the utilization of aggregate slots. We say an aggregate task is effectively using the aggregate slot if it is pulling or processing the intermediate data. In-slot waiting is a waste of resources. Table 2 also illustrates that the utilization of aggregate slots increases from 61.4% to 66.7% on dataset 1 and from 62.5% to 77.8% on dataset 2, if the HOL avoidance scheduling is enabled. Although the utilization on dataset 2 improves more significantly, the execution time on dataset 1 improves more. That is because the state-transition workload with dataset 2 is computation-intensive, and the aggregate phase contributes less to the execution time. In addition, the improvement is conservative because the HOL avoidance scheduling

postpones the aggregate phase until there exist satisfactory aggregate tasks.

Load Balancer. In the experiments, the observations in dataset 1 feature a hotspot distribution. As Mammoth uniformly partitions the data into blocks, a few trigger tasks responsible for the hotspot blocks contain intensive computations. Thus, we can evaluate the effectiveness of the load balancer on shedding off the computation from the skewed trigger tasks using dataset 1. In contrast, the observations in dataset 2 distribute broadly, and there exist few computation skews.

Figure 11 compares the CDFs of the trigger task length in Hadoop and Mammoth. Due to the lacking of the load balancer in Hadoop, for dataset 1, the median length (46s) of the trigger task is much shorter than the longest one (147s). As a result, the load balancer in Mammoth significantly shortens the lengths of the skewed trigger tasks, and all the trigger tasks become shorter than 100s. However, for dataset 2, the change of CDFs is not as obvious as that for dataset 1 since few computation skews exist.

Figure 12 illustrates the change of CDFs of the aggregate task length in Hadoop and Mammoth. Although the load balancer does not account for computation skews in aggregate tasks, the experiment result of dataset 1 indicates a significant mitigation of aggregate task skews. In fact, these long aggregate tasks are not stragglers. They are launched as early as the very beginning of the trigger phase and then wait the completion of the trigger tasks. The improvement is due to shortening the trigger phase rather than mitigating skews in aggregate tasks. For dataset 2, the observation as well as the reason about aggregate tasks are similar to that about trigger tasks: few computation skews, small performance improvements.

7. RELATED WORK

Addressing Computation Skews. SkewReduce [17] is the most related work to Mammoth. It is designed and implemented for extracting features from massive scientific data. Based on Hadoop, SkewReduce features a static partitioner that uses user-defined functions to measure the costs of data blocks. Blocks are adjusted (divided further or merged) if computation skew exists. In contrast, Mammoth tackles such skew issue in an autonomic manner. Mammoth applies the uniform partitioning no matter how data and computation distribute. Skews are addressed by the load balancer adaptively and automatically. This design frees users from describing complex cost functions as that in SkewReduce.

The user-transparent and autonomic design has already been investigated in SkewTune [19] for mitigating skews in general MapReduce applications. The skew problem in MapReduce systems has been widely studied in [18, 2, 1, 16]. Among them, [1, 16] are inspired by the idea of handling skew in parallel joins in database [8, 26] and focus on mitigating skews by appropriately partitioning data. However, Mammoth and [19, 2] emphasize solving skews automatically at runtime with minimal user intervention.

Much recent work [13, 11, 23] focuses on profiling MapReduce environments as well as on automatically tuning and optimizing the runtime behaviors. Mammoth borrows their concepts of fine-granularity profiling, self tuning, and autonomic controls, and focuses on controlling computation skews in a lightweight manner. It uses the shadow task mechanism to minimize the warm-up overhead, and introduces a confidence factor and a sensitive factor to control the detection accuracy and the mitigation effectiveness, respectively. The techniques of offline run [20] and sampling data for application signatures [13] are relatively heavy-weighted and not suitable of state-transition applications (i.e., the input data of observations changes iteration by iteration).

Supporting Iterative Operations. To accelerate the processing of iterative applications in MapReduce systems, major effects focus on speeding up iteration convergence, eliminating repeated accessing to identical inputs between iterations and subsequent re-processing of them, and reducing data movement in the shuffle phase [31, 9, 5, 30].

HaLoop [5] uses an iteration-aware scheduler that co-places map and reduce tasks and stores iteration-invariant data in caches, in order to reduce the data transferred via network and prevent reruns of the repeated data. iMapReduce [30] proposes using persistent tasks to reduce the initialization overhead between iterations and thus eliminate the data in the shuffle phase. Twister [9] keeps its map and reduce tasks alive through multiple iterations in distributed memory caches, as a consequence, repeatedly reading inputs from the disk for map tasks is avoided.

The iterative computing model in state-transition applications is different from others. The iterative computation is driven by newly incoming observations and terminates when there are no observations rather than meeting predefined convergence conditions. As the computation determined by observations changes iteration by iteration, it is hard to detect incremental changes from inputs to eliminate the repeated input reading and the subsequent re-processing. Even though, the technique of reducing the intermediate data in the shuffling phase does inspire the idea of jointly placing operators and using local disks for shuffling.

Addressing Synchronization Issues. Local synchronization is investigated in [32], and the dependency scheduling is designed to tolerate network jitters in clouds for scientific applications. Their parallel processing framework leverages the abstraction in database that application states are represented as tables and dependencies are translated into queries on these tables. Both Mammoth and [32] have the similar target applications featuring the local synchronization characteristic, and they leverage this characteristic to reduce the global synchronization overhead. Nevertheless, Mammoth is designed and implemented based on Hadoop while [32] is implemented with MPI.

The local synchronization feature in specific applications, such as mesh partitioning, offers the new opportunity of the fine-grained task scheduling in Mammoth. Different from the job scheduling problem [29, 25, 12] in sharing computing resources among users fairly and efficiently (e.g., locality, deadline driven), Mammoth schedules tasks for maximizing the overlap between phases and improving the utilization of computing resources.

Many graph processing frameworks [21, 3, 4] consider the local synchronization for speeding up computation convergences or optimizing data flows in conventional MapReduce systems. However, Mammoth does not benefit from the local synchronization feature as much as them. Synchronization in Mammoth is driven by the newly arriving data rather than computations. All the data blocks kick off an iteration simultaneously, although they can be constructed as a graph that every node links to a few neighbor nodes only.

8. CONCLUSION

We proposed an autonomic data processing framework Mammoth for scientific state-transition applications. Mammoth features a user-friendly programming model with simplistic APIs: **trigger** and **aggregate**. It is implemented by modifying Hadoop internals but keeps most good features inherited from Hadoop. To enhance the runtime performance, we further proposed three optimization mechanisms: (1) a joint placement coordinator to handle the inflated intermediate data; (2) a dependency-aware scheduler to automatically build data dependencies and localize synchronization; and (3) an adaptive and autonomic load balancer to detect and mitigate computation skews. Experimental results demonstrate that Mammoth achieves a satisfactory performance compared with the solutions based on MPI and Hadoop. Using the real-world weather application and datasets, we showed that Mammoth reduces the in-network intermediate data by near 90%. The performance of the Mammoth-based solution is comparable with the MPI-based solution. Mammoth outperforms Hadoop by at least 30%. For the future work, we plan to extend Mammoth to (1) support more general block-oriented scientific applications and (2) be capable of adapting to computation changes in many iterative scientific applications. Furthermore, we will make it available as a specialized platform as a service on our GatorCloud SDN-enabled campus cloud services. Mammoth will be available as open source for the community.

9. REFERENCES

[1] S. Agarwal, S. Kandula, N. Bruno, M. Wu, I. Stoica, and J. Zhou. Re-Optimizing data-parallel computing. In *NSDI*. ACM, 2012.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, pages 1–16. USENIX Association, 2010.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130. ACM, 2010.

[4] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[5] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. Haloop: Efficient iterative data processing on large clusters. *VLDB Endowment*, 3(1-2):285–296, 2010.

[6] A. Computing. An architectural blueprint for autonomic computing. *IBM White Paper*, 2006.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–27. IEEE, 1992.

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818. ACM, 2010.

[10] M. Fisher, J. Nocedal, Y. Trémolet, and S. Wright. Data assimilation in weather forecasting: A case study in pde-constrained optimization. *Optimization and Engineering*, 10(3):409–426, 2009.

[11] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.

[12] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, number November. ACM, 2009.

[13] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning in the cloud. In *HotCloud*, 2009.

[14] M. Karol, M. Hluchyj, and S. Morgan. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communications*, 35(12):1347–1356, 1987.

[15] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *CCGrid*, pages 94–103. IEEE, 2010.

[16] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *HotOS*, pages 13–13. USENIX Association, 2011.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC*, pages 75–86. ACM, 2010.

[18] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in MapReduce applications. *Open Cirrus Summit*, 2011.

[19] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. Technical report, Technical Report UW-CSE-12-03-03, University of Washington, 2012.

[20] P. Lama and X. Zhou. Aroma: automated resource allocation and configuration of mapreduce environment in the cloud. In *ICAC*, pages 63–72. ACM, 2012.

[21] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[22] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Decision making in autonomic computing systems: comparison of approaches and techniques. In *ICAC*, volume 11, pages 201–204, 2011.

[23] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for mapreduce dags. In *SIGMOD*, pages 507–518. ACM, 2010.

[24] A. Robinson and P. Lermusiaux. Overview of data assimilation. *Harvard reports in physical/interdisciplinary ocean science*, 62, 2000.

[25] A. Verma, L. Cherkasova, and R. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *ICAC*. IEEE/ACM, 2011.

[26] C. Walton, A. Dale, and R. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *VLDB*. IEEE, 1991.

[27] T. White. *Hadoop: The definitive guide*. Yahoo Press, 2010.

[28] M. Xue, D. Wang, J. Gao, K. Brewster, and K. Droegemeier. The Advanced Regional Prediction System (ARPS), storm-scale numerical weather prediction and data assimilation. *Meteorology and Atmospheric Physics*, 82(1):139–170, 2003.

[29] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278. ACM, 2010.

[30] Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1112–1121. IEEE, 2011.

[31] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: a distributed framework for prioritized iterative computations. In *SoCC*, page 13. ACM, 2011.

[32] T. Zou, G. Wang, M. Salles, D. Bindel, A. Demers, J. Gehrke, and W. White. Making time-stepped applications tick in the cloud. In *SoCC*, page 20. ACM, 2011.