Time and Cost Sensitive Data-Intensive Computing on Hybrid Clouds

Tekin Bicer
Computer Science and Engineering
Ohio State University
bicer@cse.ohio-state.edu

David Chiu
Engineering and Computer Science
Washington State University
david.chiu@wsu.edu

Gagan Agrawal
Computer Science and Engineering
Ohio State University
agrawal@cse.ohio-state.edu

***Abstract*—Purpose-built clusters permeate many of today's organizations, providing both large-scale data storage and computing. Within local clusters, competition for resources complicates applications with deadlines. However, given the emergence of the cloud's pay-as-you-go model, users are increasingly storing portions of their data remotely and allocating compute nodes on-demand to meet deadlines. This scenario gives rise to a *hybrid cloud*, where data stored across local and cloud resources may be processed over both environments.**

While a hybrid execution environment may be used to meet time constraints, users must now attend to the costs associated with data storage, data transfer, and node allocation time on the cloud. In this paper, we describe a modeling-driven resource allocation framework to support both time and cost sensitive execution for data-intensive applications executed in a hybrid cloud setting. We evaluate our framework using two data-intensive applications and a number of time and cost constraints. Our experimental results show that our system is capable of meeting execution deadlines within a 3.6% margin of error. Similarly, cost constraints are met within a 1.2% margin of error, while minimizing the application's execution time.

I. Introduction

Over the years, the trend of "Big Data" has prompted many organizations to acquire in-house cluster and storage infrastructures to support computing. Because these local resources are typically shared, the desired amount of computation may not always be available, which frustrates users with application deadlines. In these situations, the emergence of cloud computing has been timely. Its ability for users to immediately demand and obtain remote resources to help with computing and storage draws much interest from the computing community.

The cloud's key features include the *pay-as-you-go* model and *elasticity*. Users can instantly scale resources up or down according to the demand or the desired response time. This ability to increase resource consumption comes without the cost of *over-provisioning*, i.e., having to purchase and maintain a larger set of resources than what is needed most of the time, which is often the case for traditional in-house clusters. Some recent efforts have specifically focused on exploiting the elasticity of clouds for different services, including a transactional data store [6], data-intensive web services [11], a cache that accelerates data-intensive applications [5], and for execution of a bag of tasks [15].

In general, cloud elasticity can be exploited in conjunction with local compute resources to form a *hybrid cloud* to help meet time and/or cost constraints. For instance, some users may prefer to finish a task within a fixed deadline and may be willing to use more resources on the cloud and thus, having higher cost. Other users might prefer utilizing some cloud resources, but also have hard limits on the total cost of execution. While elasticity can be used to meet time or cost constraints, it would be desirable to have an automated and dynamic framework for such resource allocation.

This paper explores resource allocation in the aforementioned hybrid cloud environment. We describe a model-driven resource allocation framework to enable time and cost sensitive execution for data-intensive applications executed in a hybrid cloud setting. Our framework considers the acquisition of cloud resources to meet either a time or a cost constraint for a data analysis task, while only a fixed set of local compute resources is available. Furthermore, we consider the analysis of data that is split between a local cluster and a cloud storage. We monitor the data processing and transfer times to project the expected time and cost for finishing the execution. As needed, allocation of cloud resources is changed to meet the specified time or cost constraint. While the framework is dynamic, it tries to converge to a fixed number of cloud resources, so as to avoid allocating and deallocating resources during the entire execution.

We have extensively evaluated our resource allocation framework using two data-intensive applications executed with a number of different time and cost considerations. Our evaluation shows that our system is capable of meeting execution deadlines within a 3.6% margin of error. Similarly, cost constraints are met within a 1.2% margin of error, while minimizing the application's execution time.

The remainder of this paper is organized as follows. We introduce the background of this work in the next section. In Section III, we present our cost and time estimation models, as well as resource allocation algorithms guided by these models. A detailed evaluation of our system is performed using two data-intensive algorithms (KMeans clustering and PageRank). Our results are shown in Section IV. In Section V, related works are discussed, followed by our conclusions in Section VI.

II. Data-Intensive Computing on Hybrid Cloud: Motivation and Enabling Middleware

We now describe the situations where processing of data in a hybrid cloud may be desired. We also describe the needs of a framework that would support data processing within a hybrid cloud.

For a data-intensive application, co-locating data and computation on the same resource (e.g., either a cluster or a cloud environment) would clearly be ideal in terms of performance.

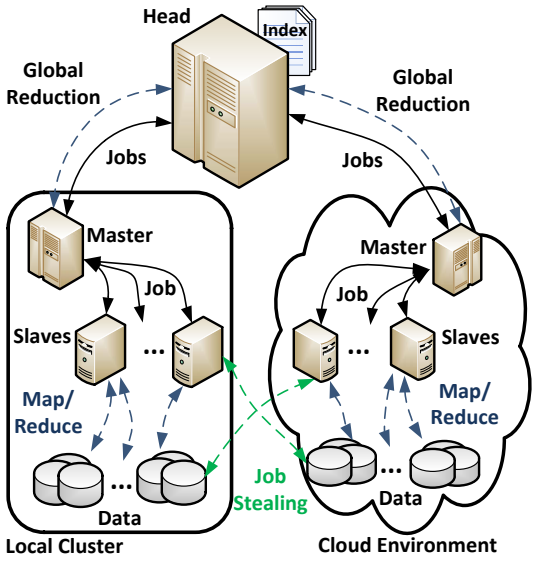


Fig. 1. Middleware for Data Processing on Hybrid Clouds

However, this is not always possible. Suppose a user wants to process data that is located in the storage nodes at a supercomputing center. When the user needs to analyze this data, compute resources at the supercomputing center may not be readily available. Rather than submitting a batch job and waiting for it to be scheduled, the user may prefer to leverage the on-demand computing resources from a cloud provider. In this particular scenario, it would not be ideal for the user to explicitly move and store the data on cloud resources. Instead, the data should be transparently moved into the cloud for processing without any effort from the user.

Consider another situation, where a research group has stored data on local disks. After some time, the research group may need to add data from new experiments or simulations, for which space is no longer available locally. In this case, the new data may be made available on cloud storage, such as Amazon's S3 service. Future users of this entire data set must access it from both locations, which complicates the application. Thus, development of future data analysis applications can be greatly simplified if the analysis can be specified with a familiar Map-Reduce type API, keeping the details of data location and data movement transparent to the user.

In our recent work, we have developed a middleware to facilitate Map-Reduce style processing on data that is stored across a local resource and a cloud storage resource [2]. The previous work, however, did not explore dynamic resource allocation for meeting time and cost constraints.

Figure 1 illustrates the execution paradigm facilitated by the middleware. The *head* node is responsible for inter-cluster communication and schedules jobs to be executed between clusters. Each cluster is managed by its own *master* node, which communicates directly with the *head* node and distributes the jobs to its *slaves*. The actual work is performed on the *slaves*, which retrieve and process the data.

Whenever a cluster's job pool diminishes, its corresponding *master* requests jobs from the *head* node. The *master* then assigns a group of jobs to the cluster based on data locality, e.g., if there are locally available jobs in the cluster, then those will be assigned first. Once all of the local jobs are processed, the remote jobs are selected from files which the minimum number of nodes are processing to reduce contention. Remote job processing is shown as "job stealing" in the figure. After all the jobs are processed, the *head* node enters the global reduction phase by requesting and combining the locally reduced data and forming the final result.

The job assignments in our system include the metadata information of the data chunks. Metadata information of a data chunk consists of location, offset, and size of each unit data. When a job is assigned to a *slave*, it retrieves the data chunk according to the given metadata information. If the data chunk is locally available, continuous read operations are performed. However, if the data chunk needs to be retrieved from a remote location, i.e. job stealing, multiple retrieval threads are used to utilize the available bandwidth. The processing of the data chunk begins at the slaves following data retrieval.

Load balancing is maintained through the *slaves*' on-demand job request scheme. Clearly, the *slave* nodes that have higher throughput (e.g., faster compute instances inside a cloud cluster) are expected to process more jobs. In similar fashion, a *master* node also requests a group of jobs from the *head* on demand, thus ensuring that the clusters with more computational throughput would perform more processing.

Given this processing framework [3], [2], we can focus on the techniques for *slave* node allocation in the cloud to meet deadlines and cost constraints. In the next section, we define our time and cost models, as well as the resource allocation algorithm which employs these models.

III. Resource Allocation Framework

Earlier, we stated that two well-known advantages of cloud computing are *elasticity* and the *pay-as-you-model*. The former refers to the ability to allocate and deallocate resources as needed, whereas the latter implies that a cloud user only pays for the resources it actually consumes. To exploit these aspects of cloud computing, our middleware includes a sophisticated and dynamic resource allocation framework.

In our current framework, we consider two different modes of execution, which are referred to as *cost constraint-driven* and *time constraint-driven* executions. We elaborate on the objectives associated with each mode below:

Cost Constraint-Driven Execution: Cost is a major consideration while using cloud resources. Thus, even if a user may simply want to accelerate a data-intensive task through scaling up in the cloud, the incurred cost may be prohibitive. The user may therefore be willing to accept a longer completion time for lowered costs. This would normally imply that a part of the cloud-resident data will be executed by local resources. The overall goal for this mode of execution is to minimize the time of execution while staying below a user-specified cost constraint.

It should be noted that the trade off between the cost and time of execution is nontrivial for two reasons. First, in most cloud environments today, there is a cost associated with retrieving data for processing outside of the cloud. Second, the cost is dependent upon not only the number of instances used, but how long they are used.

Time Constraint-Driven Execution: The elasticity of cloud

resources could be used to meet a time constraint for an application, by scaling either vertically (allocating faster nodes) or horizontally (acquiring more nodes). Either choice would normally imply a higher cost and would likely involve processing of some of the local data using cloud resources. The overall goal of this execution mode is to minimize the cost, while completing the execution within a user-specified deadline.

A. Detailed Model for Cost and Execution Time

To enable execution with either of the above two modes, our system uses a comprehensive model of execution time and cost incurred in the cloud environment. We now present this model, and show how it can be used for deciding the number of instances to be allocated.

In our model, we view a data-intensive application as comprising a number of *jobs*, or *data chunks*, to be processed. As we mentioned in the previous section, the dataset can be split into independent jobs. We denote the total number of jobs as j and assume each job has the same amount of data to be processed, and each job will take the same amount of time on a node or a given cloud instance type. Because we consider jobs on two independent resources (local cluster and cloud), j can be further expressed as $j = j_{local} + j_{cloud}$. We first focus on how the execution time of an application can be estimated.

To simplify the presentation of our model, we assume that local compute resources can process both locally and cloud-resident jobs. Based on this assumption, we can define the following terms. We define t^l_{pl} to be the time for one local instance to retrieve and process a locally resident job. Similarly, t^c_{pc} is defined as the time for one cloud instance to retrieve and process a job stored in the cloud. Furthermore, t^c_{pl} refers to the retrieval and processing time of one job in cloud using one local instance. These values, known only at runtime, represent the computational characteristics of the application with respect to local and cloud resource types. We let j_{stolen} represent the number of jobs that are stolen from the cloud environment and consumed by the local cluster. n_{local} and n_{cloud} refer to the current number of running instances in local and cloud clusters, respectively.

Given these definitions, the execution time can be estimated as follows:

$$time_{est} = \max\left(\frac{t^l_{pl} \times j_{local} + t^c_{pl} \times j_{stolen}}{n_{local}}, \frac{t^c_{pc} \times (j_{cloud} - j_{stolen})}{n_{cloud}}\right) + time_{synch} \quad (1)$$

Equation 1 calculates the estimated time of the execution with a given cloud resource allocation, n_{cloud}, and the number of jobs to be stolen, j_{stolen}. Because the processing on the cloud and local cluster are concurrent, it suffices to take the $\max$ between the two execution times. After all jobs have been processed, the results must be reduced through a synchronization of the two clusters. This additional overhead is captured with $time_{synch}$. In our model, we assumed that the instance initializations do not introduce significant overhead. This assumption is valid for most of the applications that have long running nature where the computation time is the dominating factor.

The above equation can be directly used to determine resource allocation for the time constraint-driven execution case. This mode of execution requires $time_{est}$ to be equal or close to the user provided time constraint. This can be satisfied through adjusting the n_{cloud} parameter, but doing so affects j_{stolen}. To illustrate, when n_{cloud} is set to a lower value, then the aggregated throughput of the cloud decreases, resulting in opportunities for local compute resources to process more cloud jobs. The relationship between n_{cloud} and j_{stolen} is obtained as follows:

$$j_{stolen} = \left(j_{cloud} - \frac{(t^l_{pl} \times j_{local})/n_{local}}{t^c_{pc}/n_{cloud}}\right) \times \left(\frac{n_{local}/t^c_{pl}}{(n_{cloud}/t^c_{pc}) + (n_{local}/t^c_{pl})}\right) \quad (2)$$

The left side of the main multiplication represents the estimated number of remaining jobs in the cloud after processing all the jobs in the local cluster. The latter portion calculates the job consumption ratio by the local cluster. Therefore, the multiplication results in the estimated number of jobs that will be stolen from the cloud resources and processed by the local cluster.

A further concern is that j_{stolen} and n_{cloud} are complicated by the cost incurred on the cloud. Thus, we must also relate cost for these two factors in our cost estimate:

$$cost_{est} =$$

$$t^c_{pc} \times (j_{cloud} - j_{stolen}) \times c_{inst} + \quad (3a)$$

$$size(j_{stolen}) \times c_{trans\_out} + \quad (3b)$$

$$size(j_{cloud}) \times c_{storage} + size(j_{cloud}) \times c_{trans\_in} \quad (3c)$$

In this equation, c_{inst} refers to the cost of running an instance in the cloud for a unit of time. The cost of transferring a data unit from cloud environment to local cluster is given with c_{trans_out}. The $c_{storage}$ term returns the storage cost of a data unit. Note that most of these parameters are defined by the cloud service provider and therefore can be treated as constants. To estimate cost, first, the running cost of the cloud instances is calculated (3a). Next, the data transfer cost from the cloud environment to local cluster is shown with (3b). The storage cost, and the cost of initially uploading data to the cloud, are at last added (3c).

Finally, Equation 4 shows how the boundaries of the estimated values are determined according to the user constraints.

$$\begin{aligned} 0 \leq time_{est} \leq time \\ 0 \leq cost_{est} \leq cost \end{aligned} \quad (4)$$

TABLE I
LEGEND

Symbol	Definition
t^l_{pl}	Time for processing a local job by a local instance
t^c_{pc}	Time for processing a cloud job by a cloud instance
t^c_{pl}	Time for processing a cloud job by a local instance
n_{cloud}	Number of cloud instances
n_{local}	Number of local instances
j_{stolen}	Number of stolen jobs from cloud cluster
j_{local}	Number of jobs in local cluster
j_{cloud}	Number of jobs in cloud cluster
c_{inst}	Running cost of an instance per unit time on cloud
c_{trans_out}	Cost of transferring out unit amount of data from cloud

B. Model Implementation

In the previous subsection, we presented the parameters and equations needed for estimating the execution time, cost and the stolen number of jobs from the cloud resources. There are several practical issues in using this model, which we discuss in this subsection.

The job processing times by each type of compute instance are determined during runtime by the processing clusters. Each time a group of jobs is processed, the average processing time is updated by the master node and reported to the head node. After processing several groups of jobs, these parameters can be expected to converge. The unit cost parameters, c_{inst} and c_{trans_*}, are constant and determined by the cloud service provider.

The number of instances on the cloud, n_{cloud}, is an input for calculating the estimated time and the number of stolen jobs in the system. The different values for n_{cloud} likely affect our time and cost estimation. Therefore, n_{cloud} is computed iteratively: When the closest value to the user cost and time constraints is approached, system stops iterating and sets the n_{cloud} value.

Algorithm 1: Head Node

```
Input  : user_constraint, contract_params
Output: Final Result

repeat
    cluster_params := receive_request();
    jobs := prepare_jobs(cluster_params);
    numb_instances :=
        compute_numb_inst(cluster_params,
            user_constraint, contract_params);
    setup_cluster(cluster_params, numb_instances);
    assign_jobs(cluster_params, jobs);
until is_job_unavailable();
global_reduction();
```

Algorithm 1 defines how the *head* node handles the resource allocation requests. First, a cluster's *master* node requests for jobs from the head node. The head node accepts the request and prepares a group of jobs while considering locality. After the jobs are prepared, the head node determines the new number of cloud instances according to the performance of the requesting cluster so far. Next, the head node sends this information to requesting master node. The master then sets up the new number of instances in the cloud and receives prepared jobs from the head node. The compute (slave) instances then begin processing the assigned jobs using the Map-Reduce processing structure.

The calculation of the number of instances is given in Algorithm 2. The model is executed with the cluster parameters and structures containing the cloud pricing contract, which is then compared with a user's cost constraints. The pricing contract data structure represents the agreement between user and the cloud service provider. It provides the specification of the resources and the cost information of running an instance and transferring data, i.e., the constants in our model. Given a time constraint, our approach allocates the *minimum* number of instances that can execute the application on time, thereby minimizing cost. On the other hand, given a cost constraint, the algorithm maximizes the number of allocated instances to meet the cost constraint in order to minimize the execution time.

Algorithm 2: Computing Number of Instances

```
Input: cluster_params, user_constraint, contract_params
Output: numb_instances

update_average(cluster_params.clusterID,
    cluster_params.proc_time);
numb_instances := 0;
switch user_constraint.type do
    case TIME
        repeat
            time_est := estimate_time(numb_instances,
                cluster_params, contract_params);
            numb_instances := numb_instances + 1;
        until time_est < user_constraint.value;
    case COST
        repeat
            cost_est := estimate_cost(numb_instances,
                cluster_params, contract_params);
            numb_instances := numb_instances + 1;
        until cost_est > user_constraint.value;
numb_instances := numb_instances- 1;
```

In the next Section, we evaluate our models and allocation algorithms using data-intensive applications in a real cluster environment on Ohio State campus in conjunction with the Amazon Elastic Compute Cloud (EC2).

IV. Experimental Results

In this section, we present the results of a comprehensive evaluation of our model and system. Specifically, we analyzed the behavior of our model and system under different configurations, and observed whether user constraints can be met efficiently.

A. Experimental Setup

During our evaluation, we used two geographically distributed clusters: Our local cluster which is located at the Ohio State campus and a cloud-based cluster in the Virginia region. A single cloud instance is initiated for the head node role in North California region.

Our local cluster on Ohio State campus contains Intel Xeon (8 cores) compute nodes with 6GB of DDR400 RAM (with 1 GB dimms). Compute nodes are connected via Infiniband. A dedicated 4TB storage node (SATA-SCSI) is used to store data sets for our applications. For the cloud environment, we use Amazon Web Services' Elastic Compute Cloud (EC2). High-CPU Extra large EC2 instances (`c1.xlarge`) were chosen for our experiments. According to Amazon, at the time of writing, these are 64-bit instances with 7 GB of memory. High-CPU instances provide eight virtual cores, and each core further contains two and a half elastic compute units (equivalent to a 1.7 GHz Xeon processor). High-CPU Extra Large instances are also rated as having *high* I/O performance which, according to Amazon, is amenable to I/O-bound applications and suitable for supporting data-intensive applications. The cloud dataset is stored in the popular Simple Storage Service

(S3). The maximum number of allocated instances is limited to 16 for each resource. Each allocated instance has 8 cores, which corresponds to a total maximum of 256 cores in the system throughout the execution.

Two well-known applications were used to evaluate our model and system, with various characteristics:

- K-Means Clustering (`kmeans`): A classic data mining application. It has heavy computation resulting in low to medium I/O with small intermediate results. The value of k is set to 5000. The total number of processed points is 48.2×10^9.
- PageRank (`pagerank`): Google's algorithm for determining web documents' importance [18]. It has low to medium computation leading to high I/O, with large intermediate results. The number of page links is 50×10^6 with 41.7×10^8 edges.

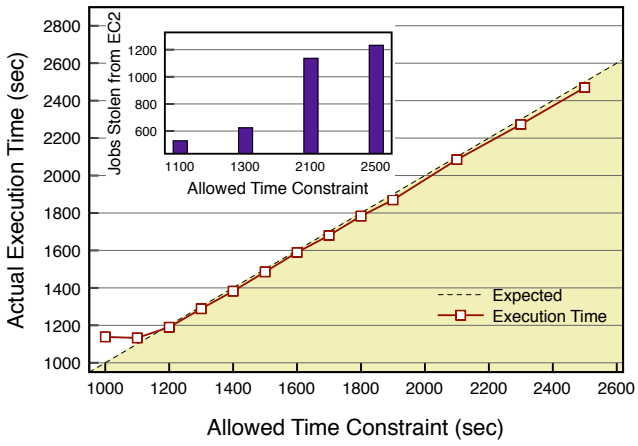
The datasets used for `kmeans` and `pagerank` are 520GB. The kmeans dataset is in binary format whereas the pagerank's data is ASCII text. The total number of generated jobs with these datasets is 4144 where each job is 128MB in size. These jobs are grouped in 16, and each job request from master node results in assigning one of these groups. To make node allocation decisions dynamically, our system invokes the prediction model after every group of jobs that is processed. These datasets are split between cloud environment and local cluster. 104GB of each dataset is stored in the local cluster, and the remaining 416GB is stored on S3.

B. Meeting Time Constraints

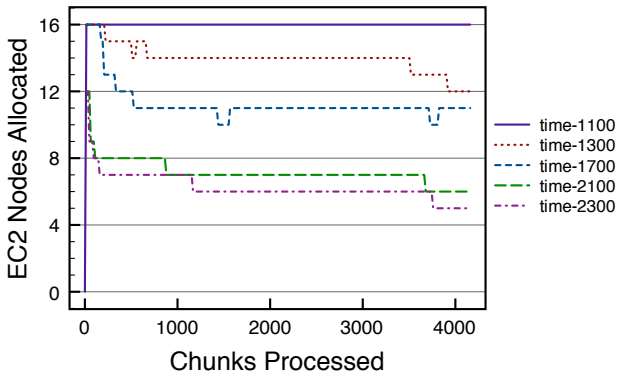
In our first set of experiments, we show how our model meets different time constraints from the user. We analyzed each of these experiments in two ways: 1) we observed how close the actual execution time of the system to the user' allowed time constraint. 2) We also observed the cloud resource allocation behavior throughout the execution in order to meet these time constraints.

Figure 2(a) compares the actual execution time with user time constraint for varying configurations. In this set of experiments, the number of local instances is fixed to 16 nodes. In the main plot, for the first two points at 1000 and 1100 seconds, the system cannot meet the time constraints. The reason is due to the fact that, even as we have reached the maximum available number of EC2 instances (16 nodes), the processing cannot finish on time. For all remaining configurations, our model successfully decides the correct number of instances throughout the execution. The error range between actual execution times and the time constraints is below 1.5%. In the subgraph, we show the number of jobs stolen by the local cluster off the cloud. As the time constraint increases, the number of stolen jobs should also be expected to increase. Since less number of EC2 nodes should be allocated for processing, there is more opportunity for local cluster to process remote jobs.

The node allocation sequence during the execution is presented in Figure 2(b). Recall that our system calls the prediction model after every group of jobs processed, and thus, the x-axis varies on the number of jobs processed. The y-axis plots the active number of cloud instances. For clarity in the plot, we show only five representative series, which associate with certain time constraints. Our first observation is made on the declining number of instances at the beginning of the execution. The reason for this is due to the selection of the



(a) Meeting Time Constraints



(b) Node Allocation

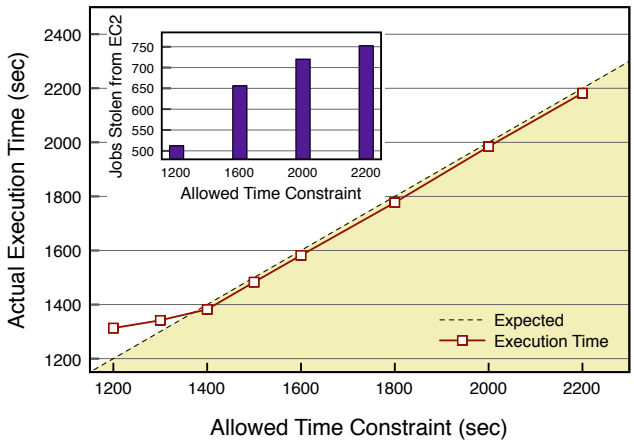
Fig. 2. KMeans under Time Constraints

initial processing time parameters, namely t^l_{pl}, t^c_{pl}, and t^c_{pc}. These parameters are normalized during the execution with cluster feedbacks.

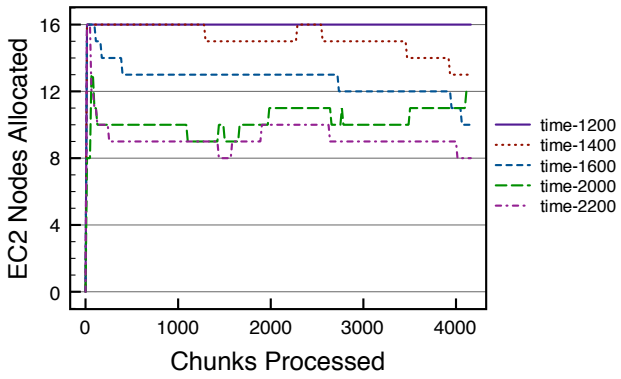
When the time constraint is set to *time-1100*, the system allocates the maximum number of available cloud instances. This also explains why real execution times of *1000* and *1100* configurations in Fig. 2(a) cannot meet the time constraints. Another decreasing trend in Fig. 2(b) can be seen at the end of execution. This is due to our preemptive approach for deciding the number of instances on the cloud side. More specifically, our system accepts the situations where $time_{est} \leq time_{user}$. However, $time_{est} > time_{user}$ is not accepted even in the case of $time_{est}$ is close to $time_{user}$.

We repeated the same set of experiments with pagerank application and presented the results in Figure 3(a) and 3(b). Pagerank application follows a similar pattern to kmeans. If we analyze the first two data points in Fig. 3(a) with the *time-1200* and *time-1400* series in Fig. 3(b), it can be seen that the system again shows a best effort to meet the time constraints. However, as with kmeans, due to the limit in maximum number of available cloud instances, it is unable to meet the constraints. For the other time constraints, the delta in actual execution time versus the given constraints is below 1.3%.

Another observation is the jitters in the range of 1000 and 2000 chunks processed, particularly for *time-2000* and *time-2200*. When the local cluster finishes its local jobs, it



(a) Meeting Time Constraints



(b) Node Allocation

Fig. 3. PageRank under Time Constraints

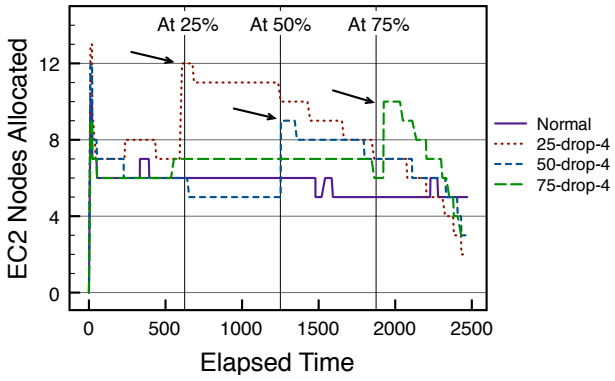


Fig. 4. KMeans Meeting Time Constraints: Varying Local Nodes

begins stealing cloud jobs. At this point, if the t^c_{pl} parameter was not sampled well enough, then the system's decisions become tenuous. This parameter normalizes after processing enough number of chunks. For instance, after 2000 chunks are processed, straight allocation lines are observed. While the application approaches the end of the execution, we again see declining trend. This follows the same reasoning with kmeans application, i.e., preemptive compute instance allocation on the cloud site.

In the previous set of experiments, the number of local instances were fixed. However, because local clusters are often shared among a number of users, there may be situations where we would want to deallocate local cluster nodes during execution. In such scenarios, we would expect an increased number of cloud nodes to help with processing in order to meet time constraints. We show this situation for kmeans in Figure 4. The series *25-drop-4*, *50-drop-4*, and *75-drop-4* refer to dropping local nodes from 16 to 4 after 25%, 50%, and 75% (shown as vertical bars in the plot) of the allowed execution time has elapsed. The time constraint for the execution is set to *2500* seconds.

For each of the **-drop-4* settings, a sharp increase in the number of allocated instances on cloud can be seen at the reflected elapsed time. For *25-drop-4*, the number of allocated cloud nodes increases by 5 up to 12 total only seconds after the local cluster instances are dropped down to 4. For *50-drop-4* and *75-drop-4*, the cloud nodes increase by 4 each, up to 9 and 10 total nodes respectively. The arrows in the figure refer to the points when our model adapts the cloud environment.

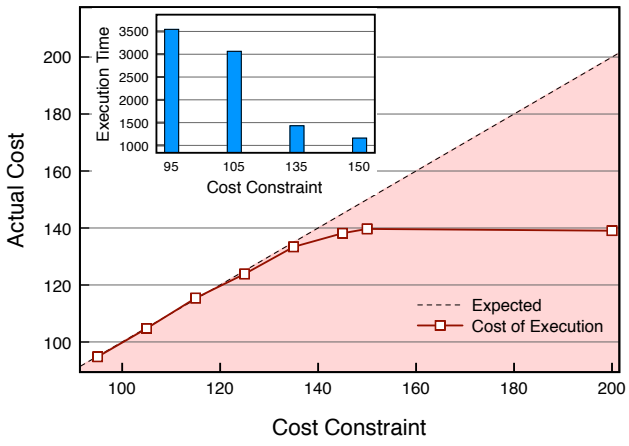
The interesting observation here is that the $(16 - 4) = 12$ local cluster nodes can be effectively replaced by only 4 or 5 cloud nodes and still meeting the 2500 second constraint. This is due to our system's accurate modeling of data locality (recall the majority of data is cloud-based). Therefore, excessive cloud node allocation is avoided, which saves on costs. The delta between the true execution time and the constraint is below 1.9% for *25-drop-4* and *50-drop-4*, and 3.6% for *75-drop-4*. The reason for higher error rate for *75-drop-4* is due to the shorter time period given to adapt to the reduced local cluster environment.

Another observation can be made on the steady decrease in the number of allocated nodes on the cloud environment. We believe this is due to the stabilization of the new t^c_{pl} value. The reason is the available bandwidth that can be consumed by an instance on the local cluster. Before the number of nodes is decreased, the same bandwidth is being consumed by a larger number of nodes. When the local instances are decreased, the available bandwidth for an instance is increased, and t^c_{pl} became smaller than expected. Since the system adapts considering the average processing time throughout the execution, a steady decrease is observed.

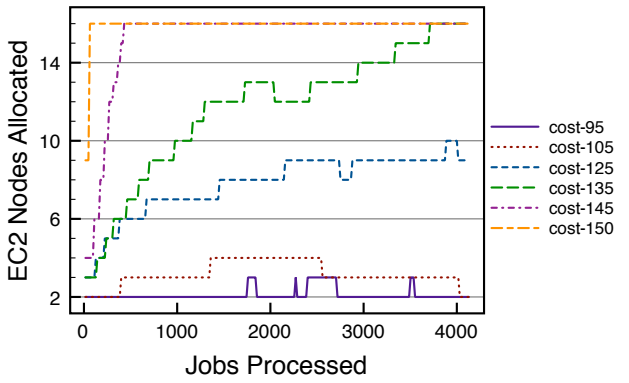
C. Meeting Cost Constraints

In this subsection, we run similar experiments to those done above, but in the context of meeting cost constraints. Particularly, we compare the user-specified cost constraints with real execution costs, and present allocation sequence of the instances on cloud environment. The goals of these experiments are to show that: 1) the system successfully meets the user specified cost constraints, and 2) the execution time is *minimized* within the available cost constraint range.

In Fig. 5(a) we present the user cost constraints against actual costs for kmeans. If we focus on the cost constraints from 0 to 140, we can see that the real costs are very close to the user provided constraints, i.e., the system effectively decides the number of cloud instances. After 140, we observe that the actual cost line is fixed and does not change. The reason for this behavior is because the system reaches the maximum number of cloud instances. If we further analyze the *cost-150* node allocation sequence in Fig. 5(b), we also see that maximum number of nodes are allocated at the beginning and kept that way throughout the execution.

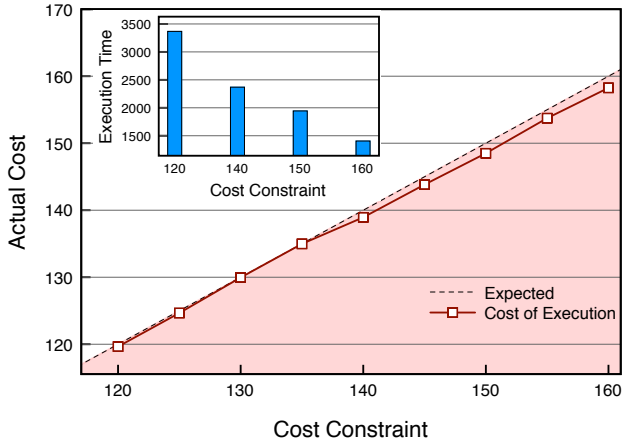


(a) Meeting Cost Constraints

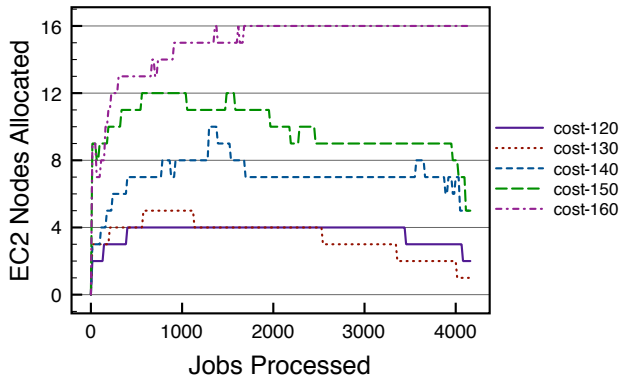


(b) Node Allocation

Fig. 5. KMeans under Cost Constraints



(a) Meeting Cost Constraints



(b) Node Allocation

Fig. 6. PageRank under Cost Constraints

Considering only Fig. 5(b), it can be seen that the allocated number of nodes shows steady increase while time passes. This is because the system tries to allocate as many instances as it can within the available cost range. Thus, the minimum execution time is satisfied. The error range for the costs below 140 is between 0.2% and 1.2%.

If we focus on the Fig. 5(a) subgraph, the execution times of increasing cost constraints show a decreasing trend given higher cost constraints. This is expected because more relaxed cost constraints result in a larger number of node allocation.

In Fig. 6(a) and 6(b) we repeat and present the same experiments using pagerank. We can see a similar pattern to the kmeans application. In Fig. 6(a), the actual costs again increase with higher cost constraints. This shows that there is still some opportunity for increasing cost and decreasing execution time. The error range of the cost constraints against execution costs is below 1.1% for all configurations.

Considering Fig. 6(b), we observe more node allocation between 0 and 2000 jobs processed, after which the node allocation is stabilized. The reason is as follows: Initially, the system tries to adapt the environment before 1000. However, when it approaches the stabilization point, the local cluster finishes its local jobs and starts stealing from cloud environment. This creates additional jitters until around 2000 jobs have been processed. Then the system approaches the optimal values for job processing times, resulting in steady node allocation.

V. Related Work

Analysis of large-scale data, or data-intensive computing has been a topic of much interest in recent years. Of particular interest is developing data-intensive applications using a high-level API, primarily, Map-Reduce framework [8], or its variants. Map-Reduce has interested cloud providers as well, with services like Amazon Elastic MapReduce now being offered. Very recently, there has been interest in use of Map-Reduce paradigm for analysis of highly distributed data. Cardosa *et al.* proposed different architectures for MapReduce which enable different widely-distributed computations [4]. Their Distributed MapReduce solution shares similarities with our system [2] in which the reduce operations are performed in a hierarchical manner. A similar approach was also developed by Luo *et al.* [13] where several clusters perform concurrent MapReduce operations and their results are reduced with a final global reduction operation. These efforts do not consider meeting user constraints and dynamic resource provisioning.

A recent effort by Deelman *et al.* [9] examined application performance and cost for workflows when data is deployed on various cloud storage options: S3, NFS, GlusterFS, and PVFS. Mao and Humphrey proposed an approach where they dynamically (de)allocate cloud instances in order to meet user constraints [14]. They consider only single environment for the computation whereas our work exploits cloud as well as local resources. Amazon's Auto Scaling [1] is a core service for enabling elasticity on their cloud. Auto Scaling allows users to

define certain rules, e.g., *scale down by one node if the average CPU usage dips below 60%*. Oprescu and Kielmann's BaTS system [17] addresses the problem of executing bag-of-tasks in the cloud while dynamically meeting cost constraints. Unlike many schedulers with similar goals, BaTS does not require *a priori* knowledge and learns application performance during runtime. Mao *et al.* focuses on auto-scaling cloud nodes to meet cost and time constraints in the cloud [15]. The authors additionally model the cost and performance effects of various cloud instance-types, rather than simply changing the number of instances allocated. These works differ from our system in that, they do not address the effects from an integration of local compute resources. Our work is distinct in considering data-intensive applications on a hybrid cloud.

Several closely-related efforts have addressed the "cloud bursting" compute model, where local resources elastically allocate cloud instances for improving application throughput or response time. An early insight into this model came from Palankar *et al.*. They extensively evaluated S3 for supporting large-scale scientific computations [19]. In their study, they observed that data retrieval costs can be expensive for such applications, and the authors discussed possibility of instead processing S3 data in EC2 (where data transfers are free) in lieu of downloading data sets off site. De Assunção *et al.* considered various job scheduling strategies which integrated compute nodes at a local site and in the cloud [7]. Each job (which may include a time constraint) is vetted on submission according to one of the strategies, and their system decides whether to execute the job on the cluster or redirect it to the cloud. Marshall *et al.* proposed *Elastic Site* [16], which *transparently* extends the computational limitations of the local cluster to the cloud. Their middleware makes calculated decisions on EC2 node (de)allocation based on the local cluster's job queue. In contrast, we consider scenarios where data sets might be also hosted on remote clouds. Our system supports pooling based dynamic load balancing among clusters, and allows for job stealing.

Several efforts have addressed issues in deploying MapReduce over the cloud. Kambatla *et al.* focused on provisioning the MapReduce jobs on the cloud therefore the cost of the execution can be minimized while the best performance is gained [10]. Related to performance, Zaharia, *et al.* analyzed *speculative execution* in Hadoop Map-Reduce and revealed that its assumption on machine homogeneity reduces performance [20]. They proposed the *Longest Approximate Time to End* scheduling heuristic for Hadoop, which improved performance in heterogeneous environments. In another related effort, Lin *et al.* have developed MOON (MapReduce On Opportunistic eNvironments) [12], which further considers scenarios where cycles available on each node can continuously vary. Our model and its implementation are distinct considering the aforementioned efforts.

VI. Conclusion

In this paper, we focused on cost and time sensitive data processing in hybrid cloud settings, where both computational resources and data might be distributed across remote clusters. We developed a model for the class of Map-Reducible applications which captures the performance efficiencies and the projected costs for the allocated cloud resources. Our model is based on a feedback mechanism in which the compute nodes regularly report their performance to a centralized resource allocation subsystem. The resources are then dynamically provisioned according to the user constraints.

We have extensively evaluated our system and model with two data-intensive applications with varying cost constraints and deadlines. Our experimental results show that our system effectively adapts and balances the performance changes during the execution through accurate cloud resource allocation. We show that our system is effective even when one of the involved clusters drastically and instantly reduces its compute nodes. The error margins of our system's ability to meet different cost and time constraints are below 1.2% and 3.6% respectively.

Acknowledgments: This work is supported by NSF grants CCF-0833101 and IIS-0916196

References

[1] Amazon auto scaling, http://aws.amazon.com/autoscaling/.

[2] T. Bicer, D. Chiu, and G. Agrawal. A framework for data-intensive computing with cloud bursting. In *CLUSTER*, pages 169–177, 2011.

[3] T. Bicer, D. Chiu, and G. Agrawal. Mate-ec2: A middleware for processing data with aws. In *Proceedings of the SC'11 Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*, 2011.

[4] M. Cardosa, C. Wang, A. Nangia, A. Chandra, and J. Weissman. Exploring mapreduce efficiency with highly-distributed data. In *MapReduce and its Applications (MAPREDUCE)*, 2011.

[5] D. Chiu, A. Shetty, and G. Agrawal. Elastic cloud caches for accelerating service-oriented computations. In *Proceedings of SC*, 2010.

[6] S. Das, D. Agrawal, and A. E. Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *Proceedings of Workshop on Hot Topics in Cloud (HotCloud)*, 2009.

[7] M. de Assuncao, A. di Costanzo, and R. Buyya. Evaluating the Cost-Benefit of Using Cloud Computing to Extend the Capacity of Clusters. In *Proccedings of High Performance Distributed Computing (HPDC)*, pages 141–150, June 2009.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[9] G. Juve, E. Deelman, K. Vahi, G. Mehta, G. B. Berriman, B. P. Berman, and P. Maechling. Data Sharing Options for Scientific Workflows on Amazon EC2. In *SC*, pages 1–9, 2010.

[10] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning in the cloud. In *1st Workshop on Hot Topics in Cloud Computing*, 2009.

[11] H. Lim, S. Babu, and J. Chase. Automated Control for Elastic Storage. In *Proceedings of International Conference on Autonomic Computing (ICAC)*, June 2010.

[12] H. Lin, X. Ma, J. S. Archuleta, W. chun Feng, M. K. Gardner, and Z. Zhang. MOON: MapReduce On Opportunistic eNvironments. In S. Hariri and K. Keahey, editors, *HPDC*, pages 95–106. ACM, 2010.

[13] Y. Luo, Z. Guo, Y. Sun, B. Plale, J. Qui, and W. W. Li. A hierarchical framework for cross-domain mapreduce execution. In *The International Emerging Computational Methods for the Life Sciences Workshop*, 2011.

[14] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *SC*, page 49, 2011.

[15] M. Mao, J. Li, and M. Humphrey. Cloud Auto-Scaling with Deadline and Budget Constraints. In *Proceedings of GRID 2010*, Oct. 2010.

[16] P. Marshall, K. Keahey, and T. Freeman. Elastic Site: Using Clouds to Elastically Extend Site Resources. In *Proceedings of Conference on Cluster, Cloud, and Grid Computing (CCGRID)*, May 2010.

[17] A.-M. Oprescu and T. Kielmann. Bag-of-tasks scheduling under budget constraints. In *CloudCom*, pages 351–359, 2010.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[19] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon S3 for Science Grids: A Viable Solution? In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64, New York, NY, USA, 2008. ACM.

[20] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, pages 29–42, 2008.