A Framework for Data-Intensive Computing with Cloud Bursting

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*—For many organizations, one attractive use of cloud resources can be through what is referred to as *cloud bursting* or the *hybrid cloud*. These refer to scenarios where an organization acquires and manages in-house resources to meet its base need, but can use additional resources from a cloud provider to maintain an acceptable response time during workload peaks. Cloud bursting has so far been discussed in the context of using additional computing resources from a cloud provider. However, as next generation applications are expected to see orders of magnitude increase in data set sizes, cloud resources can be used to store additional data after local resources are exhausted.**

In this paper, we consider the challenge of data analysis in a scenario where data is stored across a local cluster and cloud resources. We describe a software framework to enable data-intensive computing with cloud bursting, i.e., using a combination of compute resources from a local cluster and a cloud environment to perform Map-Reduce type processing on a data set that is geographically distributed. Our evaluation with three different applications shows that data-intensive computing with cloud bursting is feasible and scalable. Particularly, as compared to a situation where the data set is stored at one location and processed using resources at that end, the average slowdown of our system (using distributed but the same aggregate number of compute resources), is only 15.55%. Thus, the overheads due to global reduction, remote data retrieval, and potential load imbalance are quite manageable. Our system scales with an average speedup of 81% when the number of compute resources is doubled.

I. Introduction

Scientific and data-intensive computing have traditionally been performed using resources on supercomputers and/or various local clusters maintained by organizations. However, in the last 2-3 years, the cloud or utility model of computation has rapidly gained momentum.

Besides its appeal in the commercial sector, there is a clear trend towards using cloud resources in the scientific and HPC community. The notion of on-demand resources has prompted users to begin adopting the cloud for large-scale projects, including medical imaging [27], astronomy [7], BOINC applications [18], and remote sensing [20], to name just a few.

Various cloud providers are now specifically targeting HPC users and applications. Until recently, clouds were not very well-suited for *tightly-coupled* HPC applications. This was due to the fact that they did not initially consider high performance interconnects. Amazon, probably the single largest cloud service provider today, announced *Cluster Compute Instances* for HPC in July 2010. Shortly following that announcement, Mellanox and the Beijing Computing Center announced a public cloud that will be based on 40 Gb/s Infiniband in November 2010.

For many organizations, one attractive use of cloud resources can be through what is being referred to as *cloud bursting* or the *hybrid cloud*. These are scenarios where an organization acquires and manages in-house resources to meet its base need, but can also harness additional resources from a cloud provider to maintain an acceptable response time during workload peaks [2], [21]. Cloud bursting can be an attractive model for organizations with a significant need for HPC. But despite the interest in HPC on clouds, such organizations can be expected to continue to invest in in-house HPC resources, with considerations such providing best performance, "security" needs of certain applications, and/or desire for having more control over the resources.

At the same time, through cloud bursting, organizations can also avoid over-provisioning of base resources, while still providing users better response time. In fact, it is quite well documented that users routinely experience long delays while accessing resources from supercomputing centers. As one data point, in 2007, the ratio between wait time and execution time was nearly 4 for the Jaguar supercomputer at Oak Ridge National Lab (ORNL). Besides the need for reducing wait times for user satisfaction and productivity, another consideration is of *urgent high-end computations*, where certain compute-intensive applications rely on rapid response[1].

Cloud bursting has so far been associated with the use of additional computing resources from a cloud provider for applications [2], [21]. We do not believe that it needs to be limited in this fashion. As next generation applications are expected to see orders of magnitude increase in data set sizes, cloud resources can be used to store additional data after local resources are exhausted. While the available bandwidth to cloud-based storage is quite limited today, ongoing developments (such as building dedicated high speed connections between certain organizations and a cloud provider) are addressing this issue. Thus, one can expect efficient storage and access to data on cloud resources in the future.

In this paper, we consider the challenge of data analysis in a scenario where data is stored across local resource(s) and cloud resources. Analysis of large-scale data, or data-intensive computing has been a topic of much interest in recent years. Of particular interest has been developing data-intensive

[1]Please see http://spruce.teragrid.org

applications using a high-level API, primarily, Map-Reduce framework [3], or its variants. Map-Reduce has interested cloud providers as well, with services like Amazon Elastic MapReduce now being offered.

This paper describes a middleware that supports Map-Reduce type API in an environment where the data could be split between a local cluster and a cloud resource. Data processing is then performed using computing resources at both ends. However, to minimize the overall execution time, we allow for the possibility that the data at one end is processed using computing resources at another end, i.e., work stealing. Our middleware considers the rate of processing together with distribution of data to decide on the optimal processing of data.

We have evaluated our framework using three data-intensive applications. We have demonstrated the feasibility of data processing in a cloud bursting setting, and moreover, have evaluated our framework's scalability. Specifically, for the former, we have compared the following two data analysis scenarios: 1) *centralized processing:* a dataset of size X is stored at one location, and processed using Y computing cores at the same location, and 2) *processing with cloud bursting:* a dataset that is split (size Z at local cluster, and size $X - Z$ at Amazon S3) is processed using $Y/2$ computing cores at local cluster and $Y/2$ computing cores from Amazon EC2. The parameter Z is varied to control the *skew* or *unevenness* in the distribution of data. We observe that the average slowdown ratio of cloud bursting based data processing with our middleware (over centralized processing) is only 15.55%. Particularly, the overhead of global reduction is very small for two of the three applications, and similarly, our middleware is able to achieve good load balancing. As expected, the cost of remote data retrieval increases as Z becomes smaller. In terms of scalability, data processing with our system scales with an average speedup of 81%, everytime Y is doubled. Overall, we have shown that cloud bursting can be a feasible option for data processing, when sufficient data storage and/or computing resources are not available locally.

The rest of the paper is organized as follows. In the next section, we discuss the motivating scenarios for cloud bursting. In Section III, we describe the design and architecture of our middleware. We will present the evaluation of the system's feasibility, performance, and scalability in Section IV. A discussion of related works is presented in Section V, and we will conclude and discuss future directions in Section VI.

II. Data-Intensive Computing with Cloud Bursting: Motivation and Needs

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 with a hybrid cloud.

Our proposed software framework can be viewed as an implementation of Map-Reduce that can support the *transparent remote data analysis paradigm*. In this paradigm, analysis of data is specified with a high-level API (Map-Reduce or its variant), but the set of resources for hosting the data and/or processing it are geographically distributed. With the growing popularity of cloud-based solutions, one or both of the resources for hosting the data and processing it could potentially be entirely or partially on a cloud. For example, part of the data may be on a supercomputing center, and another part of the data may be hosted on Amazon Web Services. This is in clear contrast to solutions like Map-Reduce and a number of its recent variants [4], [5], [8], [22], [25], [11], [26], [29], [19], [1], [28], [17], which require that processing and data be *co-located*. While the resources are separated, the analysis must still be specified using a simple API. In particular, complexities such as data movements must not fall onto the hands of application developers. This is a discerning feature from workflow-based approaches [6] for distributed analysis of data sets.

Supporting such a system would prompt several issues involving locality. When feasible, co-locating data and computation would clearly achieve the best performance. However, in a cloud bursting environment, this would generally not be the case. Suppose a user wants to process data that is located in the storage nodes of a supercomputing center. Consider the case when the user needs to analyze this data, but computing resources at the supercomputer center are not immediately 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. For this scenario, however, it would be undesirable for the user to explicitly move and store the data on cloud resources. Instead, the local data should be transparently moved into the cloud for processing without effort from the user.

Consider another situation, where a research group has stored data at a supercomputing center. At some time, the research group may want to add data from new experiments or simulations, for which space is not available at the same supercomputing center. In this case, new data may be made available on a cloud storage system. Future users of this data may have to access data from both the supercomputing center and the cloud. Development of future data analysis applications will 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 this paper, we particularly focus on the situation where a fraction of the data is stored on a local resource, and remaining fraction is stored on a cloud resource. Similarly, resources for processing data are available on both the local cluster as well as on cloud, though the processing power available at both the ends may not be proportional to the amount of data available at the respective ends. At the same time, our solution should still be applicable if storage and/or processing is available only at either the local resource or the cloud. Similarly, our solution will also be applicable if the data and/or processing power is spread across two different cloud providers.

The implementation and the evaluation reported in this paper is specific to the Amazon Web Services (AWS) cloud, though the underlying ideas are applicable to any *pay-as-you-go* cloud provider.

III. System Design

In this section, we first describe the processing API of our system and compare its similarities and differences with Map-Reduce. Then we discuss the overall system framework to support this API with cloud bursting capabilities.

A. Data Processing API

The data processing framework in our system is referred to as *Generalized Reduction*. We show the processing structures for generalized reduction and the Map-Reduce programming model with and without the *Combine* function in Figure 1.

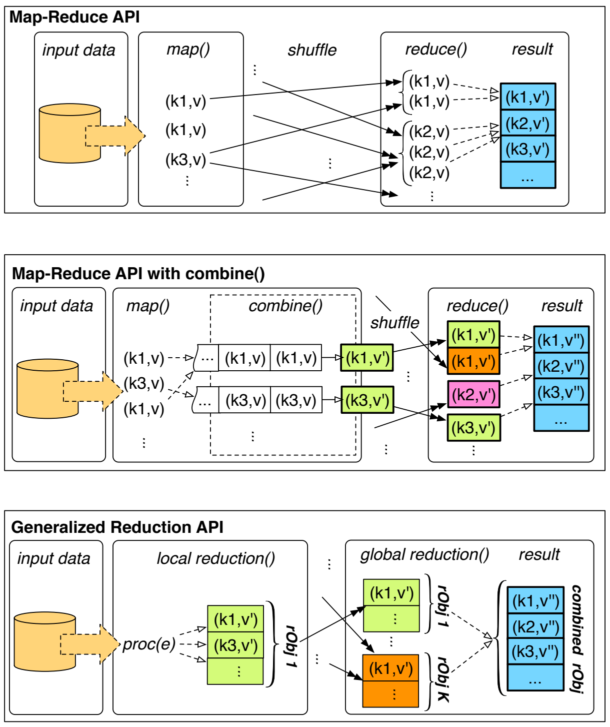


Fig. 1. Processing Structures

We will summarize the Map-Reduce API [4] as follows. The *map* function takes a set of input points and generates a set of corresponding output $(key, value)$ pairs. The Map-Reduce library then hashes these intermediate $(key, value)$ pairs and passes them to the *reduce* function in such a way that the same keys are always placed on the same *reduce* node. This stage is often referred to as *shuffle*. The *reduce* function, which is also written by the user, accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values. Typically, just zero or one output value is produced per *reduce* invocation.

The Map-Reduce framework also offers programmers an optional *Combine* function, which can be used to improve the performance of many of the applications. Before the $(key, value)$ pairs are emitted from the mapping phase, they are grouped according to their key values and stored in a buffer on the *map* nodes. When this buffer is flushed periodically, all grouped pairs are immediately reduced using the *Combine* function. These intermediate reduced pairs are then emitted to the *reduce* function. The use of *Combine* can decrease the intermediate data size significantly, and therefore reducing the amount of $(key, value)$ pairs that must be communicated from the *map* and *reduce* nodes.

We now explain the generalized reduction API, which also has 2-phases: The *local reduction* phase aggregates the processing, combination, and reduction operations into a single step, shown simply as `proc(e)` in our figure. Each data element e is processed and reduced immediately locally before the next data element is processed. After all data elements have been processed, a *global reduction* phase commences. All reduction objects from various local reduction nodes are merged with an *all-to-all* collective operation or a user defined function to obtain the final result.

The advantage of this design is to avoid the overheads brought on by intermediate memory requirements, sorting, grouping, and shuffling, which can degrade performance in Map-Reduce implementations [12]. Particularly, this advantage turns out to be critical for the cloud bursting scenario, since it is very important to reduce the inter-cluster communication.

At first glance, it may appear that our API is very similar to Map-Reduce with the *Combine* function. However, there are significant discerning differences. Using the *Combine* function can only reduce communication, that is, the $(key, value)$ pairs are still generated on each *map* node and can result in high memory requirements, causing application slowdowns. Our generalized reduction API integrates *map*, *combine*, and *reduce* together while processing each element. Because the updates to the reduction object are performed directly after processing, we avoid intermediate memory overheads.

The following are the components in the generalized reduction API that should be prepared by the application developer:

- `Reduction Object`: This data structure is designed by the application developer. However, memory allocation and access operations to this object are managed by the middleware for efficiency.
- `Local Reduction`: The local reduction function specifies how, after processing one data element, a *reduction object* (initially declared by the programmer) is updated. The result of this processing must be independent of the order in which data elements are processed on each processor. The order in which data elements are processed is determined by the runtime system.
- `Global Reduction`: In this function, the final results from multiple copies of a *reduction object* are combined into a single reduction object. A user can choose from one of the several common combination functions already implemented in the generalized reduction system library (such as aggregation, concatenation, etc.), or they can provide one of their own.

The generalized reduction API is motivated by our earlier work on a system called FREERIDE (FRamework for Rapid Implementation of Datamining Engines) [13], [14], [12].

B. System Structure

We now explain the architecture of our system and how the individual components interact with each other.

Data Organization

Our data organization scheme is specifically designed for maximizing the throughput of the processing units and can be analyzed in three granularity levels: *files*, *chunks*, and *units*. First, the data set is divided into several *files* to satisfy the compute units' file system requirements and can also be distributed. Secondly, the data inside the files are split into *logical chunks*. The compute units essentially read or retrieve the chunks from the files into memory. The decision

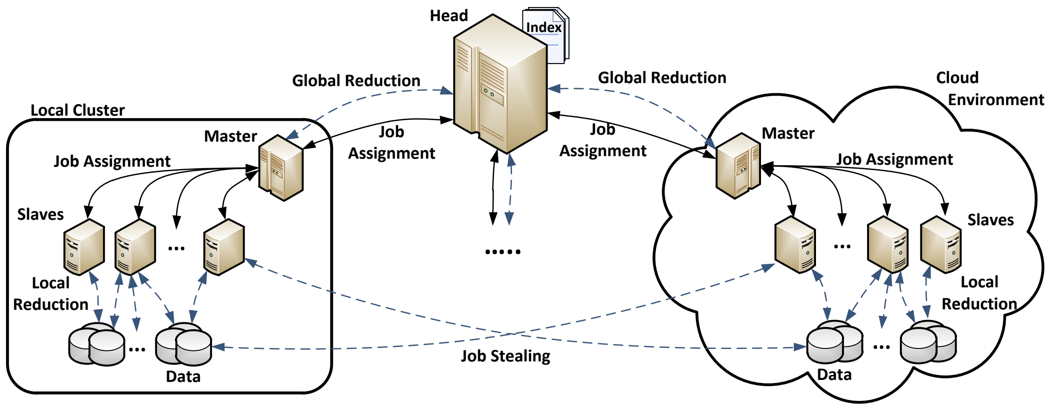


Fig. 2. System Architecture

for the size of a chunk depends on the available memory on the compute units. Finally, each chunk further consists of *data units* that need to be processed atomically. Data units represent the smallest processable data element in the system. Typically, after the data chunk is read into memory, a group of data units are sent to the local reduction layer and processed individually. The number of the data units in a group is decided according to the cache size of the processing unit, and therefore, while the logical chunks exploit the available memory on compute units, the data units maximize the cache utilization.

A data index file is generated after analyzing the data set. It holds metadata such as physical locations (data files), starting offset addresses, size of chunks and number of data units inside the chunks. When the head node (described later) starts, it reads the index file in order to generate the job pool. Each job in job pool corresponds to a chunk in data set.

System Components
A major goal of our system is to enable seamless cloud bursting for application processing. To support cloud bursting, our system must be able to manage data execution over multiple compute clusters and data sources. Figure 2 illustrates our cloud bursting execution environment. Our system consists of three basic node types: *head*, *master*, and *slave* nodes. The *head* node manages the inter-cluster communication and job assignments among the clusters. The *master* node pertaining to each cluster is responsible for communicating with the head node, as well as the job distribution among its *slave* nodes. Finally, the *slaves* are responsible for retrieving and processing the data.

We now give detailed information of these node types. The *head* node has several responsibilities. The first is assigning jobs to the requesting clusters. The *head* generates jobs according to the data set index/metadata that was generated by the aforementioned data organizer (separate and not shown in the figure). Whenever a cluster's job pool is diminishing, its *master* node interacts with the *head* node to request additional jobs. Upon receiving job requests, the *head* node sends suitable jobs to the requesting cluster. The suitability of a job may depend on the locality of the data. This mechanism is similar to that of Hadoop, a popular implementation of Map-Reduce. For instance, if there are locally available jobs in the cluster, the head node assigns a group of consecutive jobs to the requesting cluster. The selection of consecutive jobs is an important optimization in our system, because it allows the compute units to sequentially read jobs from the files and increase the input utilization. Once all local jobs belonging to a cluster are processed, the jobs that are still available from remote clusters are assigned. The remote jobs are chosen from files which the minimum number of nodes are currently processing. This heuristic minimizes file contention among clusters. The *head* node's secondary responsibility is to aggregate the final result of the execution. Specifically, after all the jobs are processed by all clusters, each cluster produces a reduction object that represents the processed jobs. Subsequently, the *head* node collects these objects and reduces them into a final reduction object.

Next, the *master* is responsible for managing the *slave* nodes in its cluster and distributing the jobs among these slaves. The *master* monitors the cluster's job pool, and when it senses that it is depleted, it will request a new group of jobs from the *head*. After the *master* receives the set of jobs, they are added into the pool, and assigned to the requesting *slaves* individually.

The *slaves* retrieve and process the assigned jobs using the aforementioned processing structure. When a job is assigned, it is immediately read into the *slave*'s memory. As we mentioned before, if the data is local, a continuous read operation is performed for optimization. However, if the assigned job is in remote location, it must to be retrieved in chunks, referred to as *job stealing* in Figure 2. Each *slave* retrieves jobs using multiple retrieval threads, to capitalize on the fast network interconnects in the cluster. After the job is read into the *slave*'s memory, it is further split into groups of data units that can fit into its cache. The local reduction function is performed on each of these groups.

Load Balancing
The data organization component, along with the pooling based job distribution enables fairness in load balancing. As the *slaves* request jobs using an on-demand basis, the *slave* nodes that have higher throughput (e.g., faster compute

instances inside a cloud cluster) would naturally be ensured 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 processing power request would perform more processing.

IV. Experimental Results

We now describe results from a detailed evaluation study we performed. Particularly, we evaluated the feasibility and performance of data-intensive computing with cloud bursting (as enabled by our framework), and then focus on the scalability of the applications in this scenario. We initially describe the experimental setup used.

A. Experimental Setup

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). Large EC2 instances (`m1.large`) were chosen for for our experiments. According to Amazon at the time of writing, these are 64-bit instances with 7.5 GB of memory. Large instances provide two virtual cores, and each core further contains two elastic compute units (which are equivalent to a 1.7 GHz Xeon processor). Large instances are also rated as having *high* I/O performance which, according to Amazon, is amenable to I/O-bound applications, suitable for supporting our system. Data sets for our applications are stored in Amazon's popular Simple Storage Service (S3).

Three well-known representative data-intensive applications were used to evaluate our system, with various characteristics:

- K-Nearest Neighbors Search (`knn`): A classic database/data mining algorithm. It has low computation, leading to medium to high I/O demands and the reduction object is small. For our experiments, the value of k is set to 1000. The total number of processed elements is 32.1×10^9.
- K-Means Clustering (`kmeans`): Another celebrated data mining application. It has heavy computation resulting in low to medium I/O, and a small reduction object. The value of k is set to 1000. The total number of processed points is 10.7×10^9.
- PageRank (`pagerank`): Google's algorithm for determining web documents' importance [23]. It has low to medium computation leading to high I/O, and a very large reduction object. The number of page links is 50×10^6 with 9.26×10^8 edges.

All the data sets used for `knn`, `kmeans`, and `pagerank` are 120GB. The data sets are divided into 32 files. Moreover, the total number of jobs generated from the data sets is 960 for each application.

B. Evaluation of Cloud Bursting

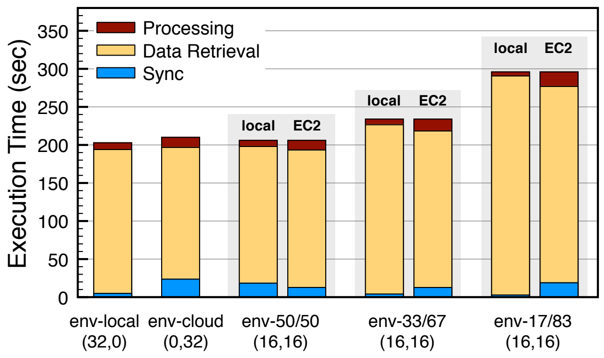
The goal of this first set of experiments is to evaluate the feasibility of using cloud bursting or hybrid configuration for data-intensive computing. Particularly, we want to see if, despite overheads like global reduction across geographically distributed clusters and remote data retrieval, application performance can be scaled.

We execute our three applications over five configurations. These configurations involve the same aggregate computing power. In the first two configurations, which are `local` and `cloud`, the computing resources and the datasets are at the same location. In other words, these two configurations involve centralized storage and processing, and are used as the baseline. The next three configurations involve a 50-50 split of computing power across local and cloud resources. Moreover, within these three configurations, there is a varying amount of *skew* or *unevenness* in the distribution of data. For example, the data distribution for `env-33/67` is 33% (40GB) of the data is hosted locally, while 67% (80GB) is being hosted in Amazon S3. By varying the amount of data skew, we increase the amount of remote data retrieval that may be needed, and thus can observe its impact on the overall performance. The five configurations are summarized below:

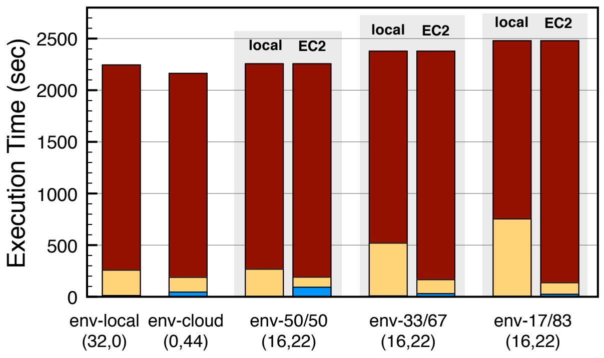
Env.	**Data Dist. All app.**		**Cores**			
			knn & pagerank		**kmeans**	
	Local	S3	Local	EC2	Local	EC2
`local`	100%	0%	32	0	32	0
`cloud`	100%	0%	0	32	0	44
`50/50`	50%	50%	16	16	16	22
`33/67`	33%	67%	16	16	16	22
`17/83`	17%	83%	16	16	16	22

On the bottom of Figures 3(a), 3(b), and 3(c), the environment, `env-*` reflect these setting labels. Furthermore in the figures, the pair (m, n) below the `env-*` setting denotes that m cores were used in the local cluster and n cores were employed in the cloud. The number of cores for each application and cluster is empirically determined according to the computational power they provide. We set the throughput power of each cluster as close as possible and evaluated the overhead of usage of the cloud with the local resources. So for instance, while the compute cores are equally halved for `knn` and `pagerank`, the `kmeans` application requires slightly more cores on the cloud. Our experience determined that 22 EC2 cores resulted in a more equal comparison with 16 local cluster nodes due to the compute intensive nature of `kmeans`.

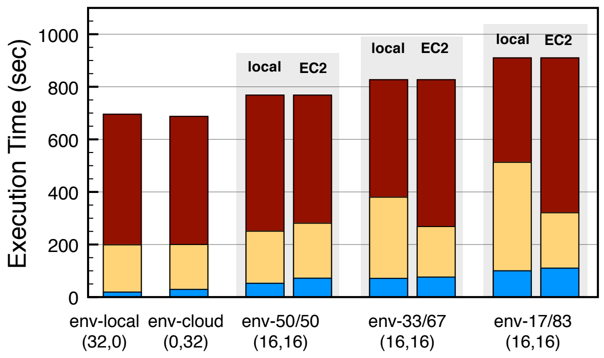
Due to the performance variability of EC2 during certain times, each execution configuration was repeated at least three times and the shortest execution time is being presented in the figures. In Figure 3, we report the overall execution time as the sum of processing time, time spent on data retrieval, and *sync.* time. The *sync.* time can be viewed as a "barrier" wait time, which is observed when the threads must synchronize on either the cluster or cloud to perform a global reduction. Another contribution to the sync. time is the time that either system must wait for the other to finish before the final result can be produced. For the first two configurations, `env-local` and `env-cloud`, there are no inter-cluster communication synchronization costs. However, there are still synchronization overheads due to local combination and intra-cluster idle time due to the slight variations in processing throughput among the *slave* nodes. Recall that for the remaining configurations, `env-50/50`, `env-33/67`, and `env-17/83`, we halved the processing power of the local and cloud environment configurations. Therefore, the total throughput of the system was



(a) K Nearest Neighbors



(b) KMeans



(c) Page Rank

Fig. 3. Cloud Bursting Execution over Various Environment

not changed. These configurations, however, still introduce additional overheads due to the global combination: inter-cluster load imbalance to the synchronization times and other system overheads such as transferred messages among *head* and *master* nodes.

Figure 3(a) shows the results of the `knn` application, which requires low amounts of computation per job processed. Comparing `env-local` with `env-cloud`, we can see that the cloud-based version has a higher synchronization overhead. The main reasons for this can be attributed to (1) the slight differences in processing throughputs among the *slave* nodes, which results in intra-cluster load imbalance, and (2) higher network delays between the *master* and *head* nodes in the `env-cloud` configuration. We can also observe that the `env-cloud` configuration has shorter retrieval time than `env-local`. This indicates that the available bandwidth between the EC2 instances and S3 was efficiently utilized by our multi-threaded data retrieval approach.

The next three environment configurations, `env-50/50`, `env-33/67`, and `env-17/83` run the applications over the cloud and the local cluster in parallel. Notice that the overall execution time tends to increase over these configurations. Since `knn` is data intensive, the data retrieval time dominates both processing and synchronization times. Furthermore, as the proportion of data increases in S3, the retrieval time on both clusters increases.

In Figure 3(b), we can clearly see that this application is dominated by computation. Even high fraction of the data is stored in S3, the computation time takes longer than the data retrieval time in local cluster. The overheads of the hybrid configurations are quite low. In fact, `env-17/83` is still performing at around 90% efficiency of `env-local`.

Next, we consider the `pagerank` application, shown in Figure 3(c). As mentioned earlier, and as we can see, `pagerank` is quite balanced between computation and data retrieval. Again, we can observe that data retrieval times are increasing across the varying data proportions. In the `env-17/83` case, the application is operating at 76% efficiency, which is expected due to the increased requirements of data retrieval. Another significant contribution to the overhead is the larger sync time due to the large reduction object.

The displacement of stored data from local clusters to S3 is identical with moving locally available jobs to the cloud. Therefore, the local cluster finishes its local jobs sooner and fetches more from S3, which results in higher retrieval times on local cluster. For clarity, Table I provides detailed information on the number of processed and stolen jobs by the clusters according to the applications and environment configurations. The right-most column beyond the dotted line shows the number of jobs the local cluster *stole* from S3 after processing all of its locally stored jobs.

	env-*	Jobs Processed		
		EC2	Local	(stolen)
knn	50/50	480	480	0
	33/67	576	384	64
	17/83	672	288	128
kmeans	50/50	480	480	0
	33/67	512	448	128
	17/83	544	416	256
pagerank	50/50	480	480	0
	33/67	528	432	112
	17/83	560	400	240

TABLE I
JOB ASSIGNMENT PER APPLICATION

We further analyze system performance in Table II, by focusing on the specific overheads and overall slowdowns. The *global reduction* refers to the elapsed time for combining and calculating the final reduction object. The idle time refers to the elapsed time in the case where one cluster is waiting for the other to finish processing at the end of execution and cannot steal jobs. The *total slowdowns* are derived for `env-50/50`, `env-33/67`, and `env-17/83` by comparing with `env-local` as the baseline.

	env-*	Global Reduction	Idle Time		Total Slowdown
			Local	EC2	
knn	50/50	0.072	16.212	0	6.546
	33/67	0.076	0	10.556	34.224
	17/83	0.076	0	15.743	96.067
kmeans	50/50	0.067	0	93.871	20.430
	33/67	0.066	0	31.232	142.403
	17/83	0.066	0	25.101	243.312
pagerank	50/50	36.589	0	17.727	72.919
	33/67	41.320	0	22.005	131.321
	17/83	42.498	0	52.056	214.549

TABLE II
SLOWDOWNS OF THE APPLICATIONS WITH RESPECT TO DATA DISTRIBUTION (IN SECONDS)

In `knn`, for `env-50/50`, the local cluster moves on to the global reduction phase before the EC2 cluster, and therefore local cluster must idle. Notice that the total slowdown is smaller than the idle time. This was a somewhat surprising result, and we believe the reason for this is that the clusters finish processing their assigned jobs very close to each other and the systems cannot steal jobs. Thus, the idle time might be maximized and total job processing time is minimized. Opposite behaviors are observed in all other environments where the EC2 cluster initiates global reduction before the local cluster. The elapsed time during the global reduction phase is short because of the small reduction object size. The ratios of total slowdown with respect to the total execution times are 1.7%, 15.4% and 45.9% for `env-50/50`, `env-33/67`, and `17/83` respectively. The data retrieval time makes up the largest portion of the total slowdown because of the increasing data imbalance between the clusters.

For `kmeans`, the slowdowns increase as the data set is disproportioned from local cluster to EC2. The reason is essentially the same with `knn`. The additional cost is due to remote processing at the local cluster and the extra jobs stored in S3. The worst case total slowdown ratio for execution time is 10.4%, which is significantly smaller than `knn`. Synchronization overheads range from only 1% to 4.1% for all configurations. The global reduction does not introduce significant overhead to the system. The overheads for the cloud bursting environments are small compared to the base versions, suggesting that compute-intensive applications can exploit cloud bursting with a very little penalty.

Finally, we analyze `pagerank` application. The slowdown ratios range from 10.5% to 30.8%. The reason of these higher slowdown ratios is because of the synchronization times. Considering the base and hybrid cloud environments, the synchronization times of base environments are relatively smaller than the hybrid cloud. In the base environments, where inter-cluster communication is avoided, the reduction object need not be transferred between the *master* and *head* node. However, in the case of hybrid cloud environments, the reduction object needs to be exchanged.

If we consider the large size of the reduction object ($\sim$ 300 MB), the introduction of the inter-cluster communication significantly increases the synchronization times of the distributed data configurations, resulting in overheads from 6.8% to 12.1%. Our conclusion is that, if the reduction object size is fixed, then the computation and data retrieval times typically dominate transfer time, minimizing the synchronization time. However, if the reduction object size increases relative to input data size, it may not be feasible to use cloud bursting due to the increasing costs of transferring the reduction object.

We now summarize the above observations. The performance of our framework is high with compute intensive applications, and we showed that cloud bursting is feasible since the typical system overhead is manageable. We observe that the proportion of data distribution and allocated throughput are important parameters, especially for data intensive applications where the execution is sensitive to data retrieval. Because data retrieval accounts for the largest portion of overheads, having a perfect distribution would likely minimize the total slowdown. We also showed that the size of the reduction object can result in additional overhead during inter-cluster communication. In our experience running experiments, the virtualized environment of EC2 can occasionally cause variability in performance, which exacerbates overheads. Our pooling based load balancing system and long running nature of the target applications help normalizing these unpredictable performance changes.

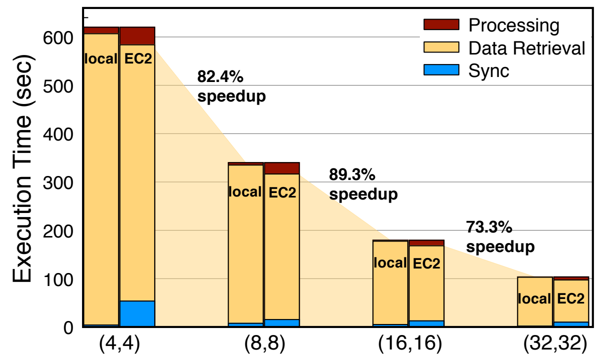
C. Scalability of the System

In this section, we present the results on the scalability of our cloud bursting data execution model. We placed *all* data sets in S3, and to show scalability, we vary the amount of cores used. In the bottom of Figures 4(a), 4(b) and 4(c), the pair (m, n) denotes the number of cores used locally and in the cloud respectively. For this set of experiments, we fixed $m = n$ and then varied $m = n =$ 4, 8, and 16.

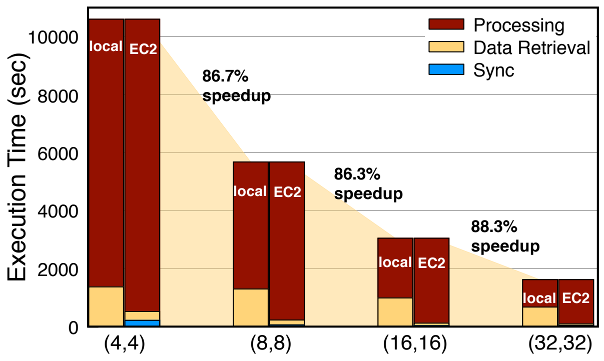
Let us first discuss `knn`, in Figure 4(a). Focusing on the $(4, 4)$ configuration, the synchronization time of EC2 is significantly higher than the local cluster. The reason for this increase is due to the imbalance in job distribution. The small number of active cores result in small processing throughput, thus larger synchronization overhead. The synchronization overheads become smaller while the number of cores increase. For all configurations, the synchronization overheads of cloud are higher than the local cluster, meaning that the cloud finishes processing all the assigned jobs and start waiting for the local cluster. Although `knn` is data intensive and all the data is located in S3, the synchronization overheads of our system only range between 0.01% to 0.03%. Furthermore, the speedups of the configurations change from 73.3% to 82.4%.

In Figure 4(b), we show the scalability results of `kmeans`. Because `kmeans` is compute-intensive and the data set is fairly large, the overall computation time is quite high. Compared with `knn`, the scalability of the system is much more dependent on processing throughput. Hence, increasing the number of cores is much more effective here, leading to greater speedups. The synchronization overhead in this set of experiment ranges from 0.1% to 2.5% and the maximum synchronization overhead is seen in the $(4, 4)$ configuration on cloud cluster. The reason is similar to `knn`: The small number of cores results in longer job processing times and the idle time of the clusters increases.

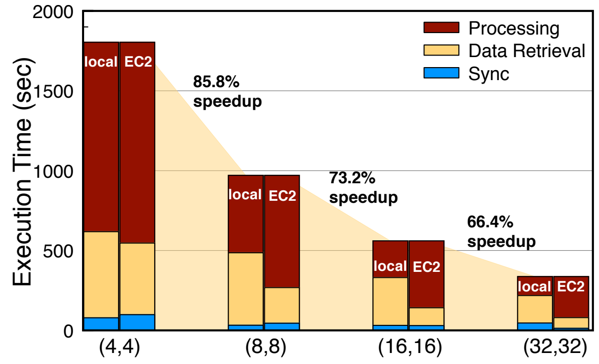
The scalability of `pagerank` is presented in Figure 4(c). Interestingly, comparing with the other applications, `pagerank` does not perform as well in terms of scalability. There are several reasons for this: First, `pagerank` has moderate to high I/O requirement, and the data retrieval from S3 to



(a) K Nearest Neighbors



(b) KMeans



(c) Page Rank

Fig. 4. System Scalability

local cluster slows down the performance. Secondly, recall that the reduction object size is quite large. For each configuration the reduction object must be exchanged during the global reduction phase. While the number of cores increases, the transfer cost remains the fixed and this negatively impacts scalability. The synchronization overheads range from 3.3% to 13.3%, and the worst case synchronization time is observed in $(32, 32)$ configuration. The reason is due to the short execution time of the application and the high overhead of the reduction object exchange.

Finally, we summarize our scalability results. Overall, we have made several observations on the scalability of our system: (1) Data intensive applications are less scalable than the compute intensive applications. Furthermore, if large portions of data is hosted in the cloud, additional slowdown is expected due to remote transfer. (2) If the inter-cluster communication cost is high, it can cause a fixed overhead which cannot be reduced through increasing the number of cores. We observed this issue with `pagerank`, and therefore, the scalability can be limited in this way. (3) If the application is compute-intensive and the data set size is large, then all the other overheads are dominated by the computation and the the system scales quite well, as observed for `kmeans`. Since our data intensive middleware takes advantage of the cloud environment, long execution times and processing large data sets are desirable.

V. Related Work

There have been a number of recent efforts focusing on the use of cloud computing for scientific and data-intensive applications.

Deelman, *et al.* described their initial foray into using of AWS resources to support the Montage application [7]. A more recent effort by these authors [15] examined application performance and cost for workflows when data is deployed on various cloud storage options: S3, NFS, GlusterFS, and PVFS. Hill and Humphrey performed a quantitative evaluation of 64-bit Amazon EC2 as a replacement of Gigabit Ethernet Commodity Clusters for small-scale scientific applications [10]. They found that EC2 is not the best platform for tightly coupled synchronized applications with frequent communication between instances because of high network latency. However, its on-demand capabilities with no queuing front-end (unlike traditional HPC environment) makes for a compelling environment.

Several closely-related efforts have addressed the "cloud bursting" compute model, where local resources elastically allocate cloud instances for improving application throughput/response time. An early insight into this model came from Palankar *et al.*. They extensively evaluated S3 for supporting large-scale scientific computations [24]. 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 [2]. 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* [21], 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 work stealing.

Another set of related works involve the celebrated class of Map-Reduce [3] applications. Driven by its popularity, Cloud providers including Google App Engine [9], Amazon Web Services, among others, began offering Map-Reduce as a service.

Several efforts have addressed issues in deploying Map-Reduce over the Cloud. For instance, the authors propose a preliminary heuristic for cost-effectively selecting a set of Cloud resources [16]. Related to performance, Zaharia, *et al.* analyzed *speculative execution* in Hadoop Map-Reduce and revealed that its assumption on machine homogeneity reduces performance [29]. 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) [19], which further considers scenarios where cycles available on each node can continuously vary. The dynamic load balancing aspect of our middleware has a similar goal, but our work is specific to AWS.

VI. Conclusion

Hybrid clouds are emerging as an attractive environment for high performance and data-intensive computing. This paper describes a framework for enabling the development and execution of data-intensive computations. Our extensive evaluation has shown that the combination of local and cloud resources can be effectively used, with only a small slowdown over processing using the same aggregate computing power at one location. Thus, cloud bursting can allow flexibility in combining limited local resources with pay-as-you-go cloud resources, while maintaining performance efficiency. Particularly, 1) inter-cluster communication overhead is quite low for most data-intensive applications, 2) our middleware is able to effectively balance the amount of computation at both ends, even if the initial data distribution is not even, and 3) while remote data retrieval overheads do increase as the disproportion in data distribution increases, the overall slowdown is modest for most applications.

References

[1] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *NSDI*, pages 313–328, 2010.

[2] 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.

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

[4] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*, pages 137–150, 2004.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[6] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping Abstract Complex Workflows onto Grid Environments. In *Journal of Grid Computing*, pages 9–23, 2003.

[7] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The Cost of Doing Science on the Cloud: The Montage Example. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[8] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *IEEE Fourth International Conference on e-Science*, pages 277–284, Dec 2008.

[9] Google App Engine, http://code.google.com/appengine.

[10] Z. Hill and M. Humphrey. A Quantitative Analysis of High Performance Computing with Amazon's EC2 Infrastructure: The Death of the LocaL Cluster? In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing*, Banff, Alberta, Canada, Oct 2009.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72. ACM, 2007.

[12] W. Jiang, V. T. Ravi, and G. Agrawal. A Map-Reduce System with an Alternate API for Multi-core Environments. In *CCGRID*, pages 84–93, 2010.

[13] R. Jin and G. Agrawal. A Middleware for Developing Parallel Data Mining Implementations. In *Proceedings of the first SIAM conference on Data Mining*, Apr. 2001.

[14] R. Jin and G. Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, Apr. 2002.

[15] 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.

[16] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning in the cloud. In *HotCloud'09: Proceedings of the 2009 conference on Hot topics in cloud computing*, Berkeley, CA, USA, 2009. USENIX Association.

[17] K. Kambatla, N. Rapolu, S. Jagannathan, and A. Grama. Relaxed Synchronization and Eager Scheduling in MapReduce. Technical Report CSD-TR-09-010, Department of Computer Science, Purdue University, 2009.

[18] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson. Cost-benefit Analysis of Cloud Computing versus Desktop Grids. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[19] 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.

[20] J. Li, *et al.* eScience in the Cloud: A MODIS Satellite Data Reprojection and Reduction Pipeline in the Windows Azure Platform. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.

[21] 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.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of SIGMOD Conference*, pages 1099–1110. ACM, 2008.

[23] 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.

[24] 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.

[25] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[26] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of 13th International Conference on High-Performance Computer Architecture (HPCA)*, pages 13–24. IEEE Computer Society, 2007.

[27] C. Vecchiola, S. Pandey, and R. Buyya. High-Performance Cloud Computing: A View of Scientific Applications. *Parallel Architectures, Algorithms, and Networks, International Symposium on*, 0:4–16, 2009.

[28] A. Verma, N. Zea, B. Cho, I. Gupta, and R. H. Campbell. Breaking the MapReduce Stage Barrier. In *CLUSTER*. IEEE, 2010.

[29] 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.