CloudClustering: Toward an iterative data processing pattern on the cloud

Ankur Dave
University of California, Berkeley
Berkeley, California, USA
ankurd@eecs.berkeley.edu

Wei Lu, Jared Jackson, Roger Barga
Microsoft Research
Redmond, Washington, USA
{*weilu,jaredj,barga*}*@microsoft.com*

***Abstract*—As the emergence of cloud computing brings the potential for large-scale data analysis to a broader community, architectural patterns for data analysis on the cloud, especially those addressing iterative algorithms, are increasingly useful. MapReduce suffers performance limitations for this purpose as it is not inherently designed for iterative algorithms.**

In this paper we describe our implementation of CloudClustering, a distributed k-means clustering algorithm on Microsoft's Windows Azure cloud. The k-means algorithm makes a good case study because its characteristics are representative of many iterative data analysis algorithms. CloudClustering adopts a novel architecture to improve performance without sacrificing fault tolerance. To achieve this goal, we introduce a distributed fault tolerance mechanism called the buddy system, and we make use of data affinity and checkpointing. Our goal is to generalize this architecture into a pattern for large-scale iterative data analysis on the cloud.

***Keywords*-data-intensive cloud computing; k-means clustering**

I. Introduction

The emergence of cloud computing brings the potential for large-scale data analysis to a broader community. The MapReduce framework [1] pioneered data-parallel computation on the cloud by providing fault tolerance and data locality to applications expressed in its programming model, and Hadoop MapReduce is currently the dominant implementation of this framework. The MapReduce model is well-suited to a class of algorithms with an acyclic data flow, but it does not natively support iterative algorithms, which have broad uses in data analysis and machine learning. Such algorithms can be expressed as multiple MapReduce jobs launched by a driver program, but this workaround imposes a performance penalty in every iteration because Hadoop MapReduce jobs must write their outputs to stable disk storage on completion.

This limitation of MapReduce has given rise to iterative frameworks like Twister [2] and Spark [3]. However, our intent in this paper is not to develop a complex system like MapReduce, Twister, or Spark—rather, it is to understand the capability of the cloud and how to apply its basic functionality to the problem of data processing. To that end, in this paper we develop CloudClustering, a distributed k-means clustering algorithm on Microsoft's Windows Azure platform. We choose to study k-means as it is a well-known algorithm exhibiting iterative and data-intensive characteristics that make it representative of a broad class of data analysis algorithms. CloudClustering consists of a cloud service architecture adapted to support data affinity and fault tolerance through a mechanism that we refer to as the buddy system. Our goal is to work toward a general pattern for large-scale iterative data processing on the cloud.

Efficiently implementing a large-scale iterative algorithm using the basic cloud services encounters obstacles related to the balance between performance and fault tolerance. The recommended cloud service architecture provides fault tolerance using a pool of stateless workers which receive tasks from a reliable central queue [4], [5]. In an iterative algorithm each task would represent a partition of the input dataset, and in each iteration workers would process the partitions to produce results. However, acceptable performance over multiple iterations requires the input dataset to be cached across all worker nodes in a way that ensures each worker will process the same partition in every iteration: there must be affinity between data and workers. Yet implementing data affinity complicates fault tolerance; if a worker fails, another worker must continue its task even without a cache of the corresponding partition. We address these issues with a distributed fault tolerance mechanism that we call the buddy system.

Moreover, accommodating a non-read-only algorithm adds another challenge. Data analysis algorithms such as k-means and linear regression must iteratively modify the input dataset for best performance; with caching, each worker must modify its cached partition. We use checkpointing to maintain fault tolerance in the face of such modification.

We have used these patterns to build an experimental k-means clustering implementation, CloudClustering, on Windows Azure. The structure of this paper is as follows. In Section II we briefly discuss the Windows Azure cloud platform and the capabilities it offers. In Section III we introduce our CloudClustering prototype and describe the details of its implementation, presenting our approach to handling an iterative, data-intensive algorithm on the cloud. In Section IV we carry out a benchmark and show that CloudClustering performs well while retaining fault-tolerance. Finally, we list the related work and conclude with a summary of our iterative data processing patterns on the cloud.

II. Windows Azure

The Windows Azure cloud platform resides on large clusters of rack-mounted computers hosted in industrial-sized data centers across the world. Broadly, the machines within these clusters provide two services: the *compute fabric* and the *storage fabric*. The compute fabric runs and manages virtual machines, while the storage fabric hosts data storage services.

Within the compute fabric, Windows Azure classifies virtual machines images into two categories, or *roles*: web roles and worker roles. Web roles are specifically configured to provide a platform for hosting web services and handling HTTP or HTTPS web page requests. Worker roles have the more general purpose of carrying out long-running tasks such as data fetching and algorithmic computation.

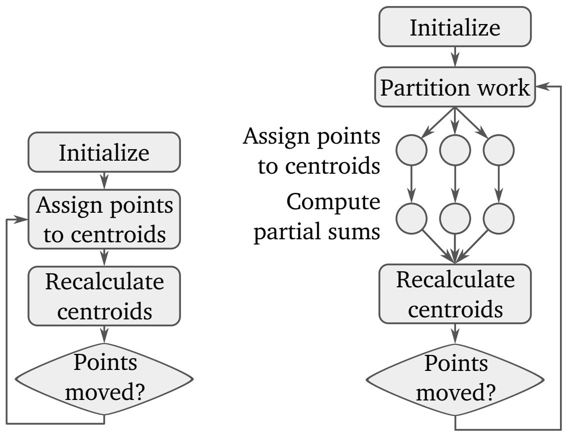
In addition to providing the capability to launch and terminate instances of these roles, a component within the compute fabric called the fabric controller manages faults within running instances. If the fabric controller detects any instance failure, it automatically attempts to recover the instance by either restarting or replacing it. Moreover, the fabric controller acts to avoid a single point of failure when deploying instances by ensuring that instances are distributed across a number of different *fault domains*. Fault domains are a feature of Windows Azure that exposes information about the likelihood of simultaneous instance failure. A fault domain is a physical unit of failure in the data center such as a rack, where a single fault (e.g., a power outage or a network disconnection) can shut down access to everything in that domain. By default, each Azure application is distributed across at least two fault domains so that the application remains available even in the event of physical failure.

The storage fabric hosts data storage services including blob storage and a reliable queue system. A blob is a file-like large object that can be retrieved by name in its entirety. Azure allows applications to store and access large blobs up to 50 GB in size each. The Azure queue service, which provides reliable messaging in the cloud, deserves elaboration because it plays a critical role in fault tolerance. When retrieving a message from a queue, Azure allows users to specify a parameter called the `visibilityTimeout`. The retrieved message remains invisible on the queue during this timeout period, but if it is not deleted by the end of the timeout period, it reappears on the queue. This feature ensures that no message will be lost even if the instance processing the message crashes. Once it has reappeared after the timeout, the message will be picked up by other running instances, providing fault tolerance for the application.

III. CloudClustering

A. Algorithm

CloudClustering implements k-means clustering, a well-known and simple iterative clustering algorithm. Given a set of n points and a desired number of clusters k, where $n >> k$, k-means finds a set of k centroids representing a locally optimal clustering of the points. Clusters are formed by the points closest to each centroid according to a given distance function. Through local optimization, k-means minimizes the sum of the squares of the distances from each point to its cluster's mean point, $\sum_{i=1}^{k} \sum_{p \in C_i} ||p - \mu_i||^2$, where C is the set of clusters.



(a) Serial k-means (b) Distributed k-means

Figure 1. k-means algorithm

Lloyd's algorithm, the classic serial implementation of k-means, begins by generating k random centroids and iteratively refines the locations of these centroids until the algorithm converges [6]. Each iteration consists of two steps: (a) assign points to closest centroids, and (b) recalculate centroids as the average of their points. This process is visualized in Figure 1a.

The typical heuristic for convergence examines the change in each points assignment to a cluster from one iteration to the next, and reports convergence if none of the points has moved between clusters. In executing this heuristic it is necessary to store the previous centroid associated with each point to evaluate whether or not it has changed. This requirement implies that it must be possible to annotate the points in order to store their previous centroid associations, leading to a non-read-only, I/O-bound algorithm. The non-read-only nature becomes a significant challenge when implementing a fault-tolerant algorithm in a distributed environment.

An alternative heuristic is to stop iterating when the maximum centroid movement from one iteration to the next falls below a certain threshold ϵ. This heuristic is much less challenging than the previous one because it is read-only with respect to the set of points. However, this heuristic does not guarantee that the algorithm has converged.

The serial k-means algorithm is not appropriate for datasets larger than can fit in memory on a single machine. Instead, a modification called distributed k-means is commonly used (see Figure 1b). In distributed k-means, the input point dataset is partitioned across multiple workers, each of which assigns its points to the closest centroids and computes a partial point sum for each centroid. There is a synchronization barrier after each iteration to allow a master to aggregate the partial sums, average them to form the updated set of centroids, and re-broadcast them to the workers in preparation for the next iteration.

We choose k-means as a representative algorithm because it is a very well-known iterative, data-intensive algorithm. This algorithm can be implemented in many cloud frameworks, such as Twister [2], Spark [3], and HaLoop [7], but our intention in building it directly on Windows Azure is to gain insight into the performance of data-intensive algorithms on the cloud.

B. Architecture

We have implemented the distributed k-means algorithm on Windows Azure. The architecture primarily takes advantage of the Azure blob and queue storage services in its operation. The input dataset to the distributed k-means computation is represented as a serialized sequence of Cartesian points. A pool of workers carries out the main computation, supervised by a master node.

The input dataset is stored centrally as an Azure blob. At the beginning of the computation, the master partitions the input dataset, assigning one partition to each worker. The master also generates a set of k random centroids to serve as a basis on which to iteratively improve. Once initialization is complete, the server uses the Azure queue storage to send a request containing the current centroid list and a reference to the worker's fixed partition of the input dataset; when workers receive this request, they begin downloading the partition data from the blob storage in parallel and then processing the points using the given centroids. Workers take advantage of multicore instances in processing the data, which occurs in two stages as described in the previous section: assigning points to closest centroids, and generating partial point sums for each cluster. At the end of each iteration, the workers return the requested results to the master, again by means of Azure message queue service. We use queues to do inter-instance communication rather than the conventional socket-based approach in order to avoid implementing reliable messaging from scratch. The goal is to understand the functionality provided in the cloud in the context of data-intensive processing.

The detailed schema of the inter-instance communication has implications for the performance and fault tolerance of this system. The conventional cloud service best practices recommend a single request queue connected to by all workers, each of which polls messages from it (see Figure 2a). The fault tolerance takes advantage of the reliable messaging feature provided by Azure queues. While this pattern can provide the best fault tolerance, it requires each worker to be stateless. Since a single queue does not guarantee that

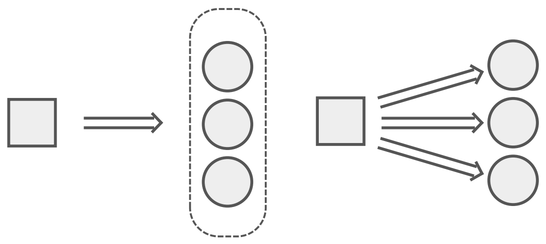

(a) Single queue for fault tolerance (b) Multiple queues for data affinity

Figure 2. Data affinity pattern

each worker will dequeue the same task partition from one iteration to the next, it is meaningless for workers to cache their partitions of the input dataset. It is therefore not possible to create cross-iteration affinity between workers and the data on which they operate, incurring a significant performance penalty.

Our solution is to adopt a pattern involving one request queue per worker, where each worker is associated with a unique queue through which the master sends a work request at the beginning of every iteration (see Figure 2b). This approach allows workers to cache their partition of the input dataset and satisfy the master's requests entirely out of cache in every iteration, leading a significant performance improvement. In order to facilitate this pattern, workers register themselves under a unique ID with the master through a separate control queue as part of the CloudClustering startup process. Each worker creates its own request queue using a hash of its unique ID, where the transformation is known to both the master and the worker. This allows the master to communicate with each worker individually and to assign it the same partition of the dataset in every iteration.

However, the shortcoming of this pattern is that when one worker fails, all messages in its queue are inaccessible and remain unprocessed until the worker instance is recovered by the Azure fabric. Because the average time to recovery of an Azure instance is 10-20 minutes [8], the entire iteration stalls for a significant period of time whenever any worker fails. We therefore introduce a buddy-based pattern to address this issue and provide fault tolerance to the application.

C. Buddy system

Separating the I/O channels by moving from one central request queue to per-worker request queues creates the need for an additional mechanism to handle fault tolerance. With a central request queue, the Windows Azure queue service guarantees that any tasks that a failed node was working on are automatically returned to the queue after a timeout and subsequently restarted by a new node. Per-worker request queues do not benefit from this fault-tolerant behavior; each worker polls only its own queue, so tasks that are returned to the queue of a failed worker will never be processed, and

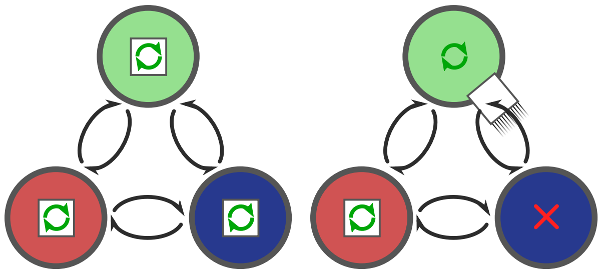

(a) Buddy group before worker failure (b) Buddy group after worker failure

Figure 3. Buddy system. Colored circles represent workers in different fault domains. White boxes represent tasks. Arrows represent polling for instance failure. When a worker fails, one of its buddies takes over its task.

the iteration will stall.

To address this issue, we introduce an active monitoring mechanism that we refer to as the *buddy system*. The master assigns workers into buddy groups at the beginning of the computation, as well as whenever the set of workers changes due to failure or scaling. Whenever each worker in a buddy group is idle, it polls the task queues of its buddies. Therefore, when a worker instance fails, one of its buddies will observe that there is a message in the worker's queue that has existed for longer than a specified time interval, and will automatically take over that message and run it on behalf of the failed instance. Consequently, the iteration can progress without waiting for the recovery of the failed instance. This process is illustrated in Figure 3.

Note that if the number of buddies per buddy group is one, this pattern degrades to the one queue per worker pattern that maximizes data affinity. On the other hand, if the number of buddies per buddy group is the same as the total number of workers, then this pattern is equivalent to the single-queue pattern, which maximizes fault tolerance. The buddy system is therefore in some sense a trade-off to balance data affinity and fault tolerance.

The decentralized nature of the buddy system makes it simple to implement compared to centralized fault tolerance mechanisms such as heartbeating used by Hadoop. It relies solely on the basic cloud queue service rather than requiring the use of additional communication mechanisms such as network sockets. Because the buddy system avoids unnecessary duplication of processing and workers poll their buddies' queues only when they are otherwise idle, it is optimally efficient when no failures occur.

A limitation of the buddy system is that it is vulnerable to multiple simultaneous worker failures. Although the master regroups workers as soon as a failure is detected in order to ensure that each buddy group contains at least two workers, if all workers in a buddy group fail simultaneously, then the failure will go unreported and the job will stall. One solution is to increase the number of buddies so that the possibility of simultaneous failures becomes very low. However, large buddy groups cause increased network traffic because each worker has to poll an increasing number of buddy queues. Because Windows Azure charges for storage transactions at the rate of $0.01 per 10,000 transactions, this traffic can incur a monetary cost. In addition, with more buddies per group the expected cache utilization in the event of failure decreases, because it is nondeterministic which buddy picks up a failed worker's task in any given iteration.

Instead, we address this drawback by taking advantage of Windows Azure's support for user-specified fault domains mentioned before to reduce the chance that an entire buddy group could fail simultaneously. Fault domains expose information about the likelihood of simultaneous failure; if Windows Azure reports two workers as members of different fault domains, then they are guaranteed to be in different physical units of failure (for example, different racks in the datacenter) and guaranteed higher collective uptime as part of the service level agreement. When assigning workers into buddy groups, the master always distributes the workers within each group across as many fault domains as possible. This provides a high degree of fault tolerance even within a small buddy group.

D. Checkpointing

While the buddy pattern works well for the read-only convergence heuristic, it becomes insufficient to handle the non-read-only heuristic, which requires each worker to modify its cached partition and to keep the cache across iterations. In this case the workers become stateful and non-interchangeable, and the system is vulnerable to data loss in the event of a failure. In the absence of any system to prevent it, the entire computation might have to begin from scratch.

Our solution is to augment the buddy pattern with checkpointing, a well-known strategy for fault tolerance in distributed computations, to periodically back up the locally stored centroid associations into central blob storage with the intent of restoring it to a buddy worker. The history of calculated centroids is also stored because it is necessary in order for the buddy worker to perform all the iterations since the previous checkpoint. Modifying how frequently checkpointing occurs shifts the balance between performance and fault tolerance—the more frequently, the less work is necessary to recover from failure, but the greater is the overhead from I/O.

IV. Evaluation

Windows Azure compute instances come in four unique sizes to enable complex applications and workloads. Each Azure instance represents a virtual server; the hardware configurations of each size are listed in Table I. In terms of software, every Azure instance, no matter the size, runs a specially tailored Microsoft Windows Server 2008 Enterprise

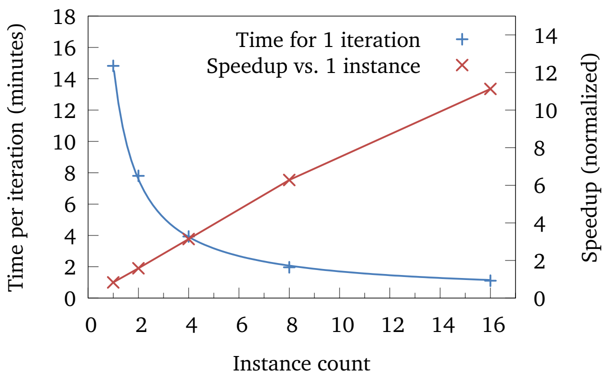


Figure 4. Scale-out: Time and speedup for varying worker counts

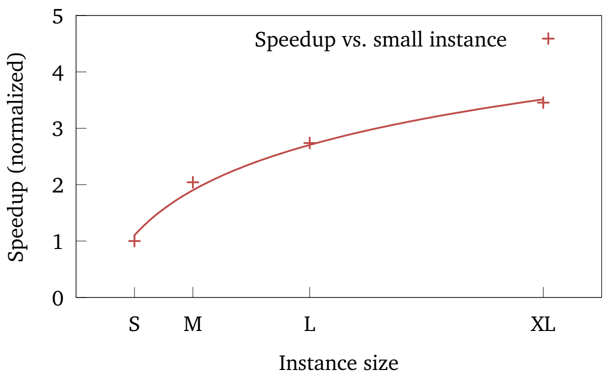


Figure 5. Scale-up: Speedup for varying instance size

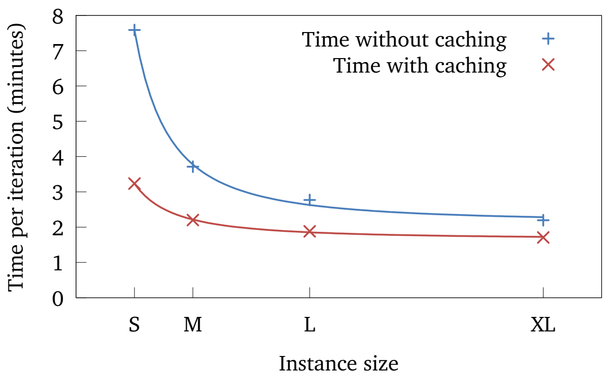


Figure 6. Caching: Time for varying instance size, with and without caching

operating system as the guest OS, referred to as the Azure Guest OS.

We benchmarked CloudClustering on Windows Azure with varying numbers of workers and instance types, as well as with and without caching. Benchmarking shows a linear speedup when scaling out the number of workers in the computation (Figure 4). When scaling up the number of CPU cores and amount of memory per worker through

Table I
WINDOWS AZURE INSTANCE SIZE AND CONFIGURATION

Instance Size	**CPU**	**Memory**	**Storage**	**I/O**
Small	1.6 GHz	1.75 GB	225 GB	Moderate
Medium	2 x 1.6 GHz	3.5 GB	490 GB	High
Large	4 x 1.6 GHz	7 GB	1,000 GB	High
Extra Large	8 x 1.6 GHz	14 GB	2,040 GB	High

Azure's instance size parameter, we observe a sublinear speedup (Figure 5). This result correlates with the total amount of I/O bandwidth available in each trial run, because the algorithm is I/O bound due to checkpointing every iteration as a result of using the non-read-only convergence heuristic. The more instances the application is using, the greater the total available bandwidth. Caching also improves performance, but the improvement narrows with increasing instance size, as illustrated by the narrowing difference between the two benchmark data sets (Figure 6). Larger instance sizes deliver higher bandwidth guarantees, reducing the potential performance gain from caching.

V. RELATED WORK

Existing cloud frameworks for iterative algorithms can currently implement the k-means algorithm. It has been adapted for Hadoop MapReduce by Zhao et al. [9], whose work focuses on implementing the clustering algorithm with the read-only convergence heuristic in the Hadoop MapReduce pattern. However, as Hadoop does not support iterative processing by design, this implementation has little choice regarding data affinity even with the read-only convergence heuristic, and the implementation's fault tolerance is delegated to the Hadoop framework.

Twister [2] and HaLoop [7] provide modifications to MapReduce that add support for iterative computation. Spark [3] provides a general cluster computing framework that supports cache persistence across iterations. Rather than using a high-level programming model, however, CloudClustering is built directly on the basic cloud services that Windows Azure provides so that we can examine how each building block affects the performance and fault-tolerance of the system.

Other algorithms on Windows Azure include AzureBlast [10], a study of benchmarks and best practices from implementing NCBI-BLAST on Azure. Building on the experience gained from AzureBlast, we examine the performance of iterative and bandwidth-intensive algorithms on Azure. In CloudClustering, unlike in AzureBlast, caching and data affinity are essential, and performance is constrained to a far greater extent by I/O bandwidth. As a result, we observe different performance characteristics and tradeoffs, as reliance on I/O bandwidth fundamentally affects the application's performance characteristics.

VI. Conclusion

In this paper we have introduced a general pattern to balance data affinity and fault tolerance for iterative data analysis algorithms in the cloud. We have used it to implement k-means clustering and demonstrate its performance. Compared with other patterns, the buddy system is simpler but more robust, as it makes use solely of the reliable cloud messaging service. Furthermore, it can naturally take advantage of the powerful fault domain features provided by Windows Azure, achieving a very efficient balance between data locality and fault tolerance for iterative algorithms. We have also incorporated the checkpointing pattern to address non-read-only data processing scenarios.

Future improvements to CloudClustering include investigating fault tolerance of the master and improving the efficiency of the existing caching and multicore parallelism. The master is currently vulnerable to failure, but storing its state in a reliable storage system such as Azure table storage would allow the entire system to become resilient. In addition, when the number of workers changes due to an added worker or a failure, the input dataset is currently repartitioned to keep the work evenly distributed, causing each worker's cache to be flushed; it may be possible to take advantage of overlap between the old and the new partitions to improve efficiency. Finally, increasing the number of I/O channels opened on each worker to read and write the data partitions could significantly improve application performance.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," December 2004, pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

[2] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *HPDC*, S. Hariri and K. Keahey, Eds. ACM, 2010, pp. 810–818. [Online]. Available: http://dblp.uni-trier.de/db/conf/hpdc/hpdc2010.html#EkanayakeLZGBQF10

[3] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-53, May 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.html

[4] D. Chappell, "Introducing Windows Azure," March 2009.

[5] J. Varia, "Architecting for the cloud: best practices," January 2010.

[6] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," in *36th International Conference on Very Large Data Bases*, Singapore, September 14–16, 2010.

[8] W. Lu, J. Jackson, J. Ekanayake, R. S. Barga, and N. Araujo, "Performing large science experiments on Azure: Pitfalls and solutions," in *CloudCom*, 2010, pp. 209–217.

[9] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on MapReduce," in *Cloud Computing*, ser. Lecture Notes in Computer Science, M. Jaatun, G. Zhao, and C. Rong, Eds. Springer Berlin / Heidelberg, 2009.

[10] W. Lu, J. Jackson, and R. Barga, "AzureBlast: a case study of developing science applications on the cloud," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 413–420. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851537