Scalable Parallel Computing on Clouds Using Twister4Azure Iterative MapReduce

Thilina Gunarathne, Bingjing Zhang, Tak-Lon Wu, Judy Qiu
School of Informatics and Computing
Indiana University, Bloomington.
tgunarat, zhangbj, taklwu, xqiu@indiana.edu

***Abstract*— Recent advances in data intensive computing for science discovery are fueling a dramatic growth in use of data-intensive iterative computations. The utility computing model introduced by cloud computing combined with the rich set of cloud infrastructure and storage services offers a very attractive environment for scientists to perform data analytics. The challenges to large-scale distributed computations demand new frameworks that are specifically tailored for cloud characteristics in order to easily and effectively harness the power of *clouds*. Twister4Azure is a distributed decentralized iterative MapReduce runtime for Windows Azure Cloud. It extends the familiar, easy-to-use MapReduce programming model with iterative extensions, enabling a wide array of data mining and data analysis applications on the Azure cloud. This paper presents the Twister4Azure iterative MapReduce runtime and discusses the applicability of Twister4Azure for scientific computation with highlighted features of fault-tolerance, efficiency and simplicity. We study four data-intensive applications − two iterative scientific applications, Multi-Dimensional Scaling and KMeans Clustering; two data–intensive pleasingly parallel scientific applications, BLAST+ sequence searching and SmithWaterman sequence alignment. Performance measurements show comparable or a factor of 2 to 4 better results than the traditional MapReduce runtimes deployed on up to 256 instances and for jobs with tens of thousands of tasks. We also study and present solutions to several factors that affect the performance of iterative MapReduce applications on Windows Azure Cloud.**

Keywords- Iterative MapReduce, Cloud Computing, HPC, Scientific applications, Azure

1 INTRODUCTION

The current parallel computing landscape is vastly populated by the growing set of data-intensive computations that require enormous amounts of computational as well as storage resources and novel distributed computing frameworks. The pay-as-you-go Cloud computing model provides an option for the computational and storage needs of such computations. The new generation of distributed computing frameworks such as MapReduce focuses on catering to the needs of such data-intensive computations.

Iterative computations are at the core of the vast majority of large scale data intensive computations. Many important data intensive iterative scientific computations can be implemented as iterative computation and communication steps, in which computations inside an iteration are independent and are synchronized at the end of each iteration through reduce and communication steps, making it possible for individual iterations to be parallelized using technologies such as MapReduce. Examples of such applications include dimensional scaling, many clustering algorithms, many machine learning algorithms, and expectation maximization applications, among others. The growth of such data intensive iterative computations in number as well as importance is driven partly by the need to process massive amounts of data and partly by the emergence of data intensive computational fields, such as bioinformatics, chemical informatics and web mining.

Twister4Azure is a distributed decentralized iterative MapReduce runtime for Windows Azure Cloud that is developed utilizing Azure cloud infrastructure services. Twister4Azure extends the familiar, easy-to-use MapReduce programming model with iterative extensions, enabling a wide array of large-scale iterative data analysis and scientific applications to utilize Azure platform easily and efficiently in a fault-tolerant manner. Twister4Azure effectively utilizes the eventually-consistent, high-latency Azure cloud services to deliver performance that is comparable to traditional MapReduce runtimes for non-iterative MapReduce. It outperforms traditional MapReduce runtimes for iterative MapReduce computation. Twister4Azure has minimal management & maintenance overheads and provides users with the capability to dynamically scale up or down the amount of computing resources. Twister4Azure takes care of almost all the Azure infrastructure (service failures, load balancing, etc.) and coordination challenges, and frees users from having to deal with cloud services. Window Azure claims to allow the users to "Focus on your applications, not the infrastructure." Twister4Azure take it one step further and lets users focus only on the application logic without worrying about the application architecture.

Applications of Twister4Azure can be categorized as three classes of application patterns. First are the Map only applications, which are also called pleasingly (or embarrassingly) parallel applications. Example of this type of applications include Monte Carlo simulations, BLAST+ sequence searches, parametric studies and most of the data cleansing and pre-processing applications. Section 4.4 analyzes the BLAST+[1] Twister4Azure application.

The second type of applications includes the traditional MapReduce type applications, which utilize the reduction phase and other features of MapReduce. Twister4Azure contains sample implementations of SmithWatermann-GOTOH (SWG)[2] pairwise sequence alignment and Word Count as traditional MapReduce type applications.

The third and most important type of applications Twister4Azure supports is the iterative MapReduce type applications. As mentioned above, there exist many data-intensive scientific computation algorithms that rely on iterative computations, wherein each iterative step can be easily specified as a MapReduce computation. Section IV and V present detailed analysis of Kmeans Clustering and MDS iterative MapReduce implementations. Twister4Azure also contains an iterative MapReduce implementation of PageRank and we are actively working on implementing more iterative scientific applications using Twister4Azure.

Developing Twister4Azure was an incremental process, which began with the development of pleasingly parallel cloud programming frameworks[3] for bioinformatics applications utilizing cloud infrastructure services. MRRoles4Azure[4] MapReduce framework for Azure cloud was developed based on the success of pleasingly parallel cloud frameworks and was released in December 2010. We started working on Twister4Azure to fill the void of distributed parallel programming frameworks in the Azure environment (as of June 2010) and the first public beta release of Twister4Azure[5] was performed in May 2011.

2 BACKGROUND

2.1 MapReduce

The MapReduce[6] data-intensive distributed computing paradigm was introduced by Google as a solution for processing massive amounts of data using commodity clusters. MapReduce provides an easy-to-use programming model that features fault tolerance, automatic parallelization, scalability and data locality-based optimizations.

2.2 Apache Hadoop

Apache Hadoop[7] MapReduce is a widely used open-source implementation of the Google MapReduce[6] distributed data processing framework. Apache Hadoop MapReduce uses the Hadoop distributed file system(HDFS) [8] for data storage, which stores the data across the local disks of the computing nodes while presenting a single file system view through the HDFS API. HDFS is targeted for deployment on unreliable commodity clusters and achieves reliability through the replication of file data. When executing Map Reduce programs, Hadoop optimizes data communication by scheduling computations near the data by using the data locality information provided by the HDFS file system. Hadoop has an architecture consisting of a master node with many client workers and uses a global queue for task scheduling, thus achieving natural load balancing among the tasks. The Map Reduce model reduces the data transfer overheads by overlapping data communication with computations when reduce steps are involved. Hadoop performs duplicate executions of slower tasks and handles failures by rerunning the failed tasks using different workers

2.3 Twister

The Twister[9] iterative MapReduce framework is an expansion of the traditional MapReduce programming model, which supports traditional as well as iterative MapReduce data-intensive computations. Twister supports MapReduce in the manner of “configure once, and run many time”. Twister configures and loads static data into Map or Reduce tasks during the configuration stage, and then reuses the loaded data through the iterations. In each iteration, the data is first mapped in the compute nodes, and reduced, then combined back to the driver node (control node). Twister supports direct intermediate data communication, using direct TCP as well as using messaging middleware, across the workers without persisting the intermediate data products to the disks. With these features, Twister supports iterative MapReduce computations efficiently when compared to other traditional MapReduce runtimes such as Hadoop[10]. Fault detection and recovery are supported between the iterations. In this paper, we use the java implementation of Twister and identify it as Java HPC Twister.

Java HPC Twister uses a master driver node for management and controlling of the computations. The *Map* and *Reduce* tasks are implemented as worker threads managed by daemon processes on each worker node. Daemons communicate with the driver node and with each other through messages. For command, communication and data transfers, Twister uses a Publish/Subscribe messaging middleware system and ActiveMQ[11] is used for the current experiments. Twister performs optimized broadcasting operations by using chain method[12] and uses minimum spanning tree method[13] for efficiently sending Map data from the driver node to the daemon nodes .Twister supports data distribution and management through a set of scripts as well as through the HDFS[8].

2.4 Microsoft Azure platform

The Microsoft Azure platform [16] is a cloud-computing platform that offers a set of cloud computing services. Windows Azure Compute allows the users to lease Windows virtual machine instances according to a platform as service model and offers the .net runtime as the platform through two programmable roles called Worker Roles and Web Roles. Starting recently

Azure also supports VM roles (beta), enabling the users to deploy virtual machine instances supporting an infrastructure as a service model as well. Azure offers a limited set of instance types (Table 1) on a linear price and feature scale[14].

Table 1 Azure Instance Types

Virtual Machine Size	CPU Cores	Memory	Cost Per Hour
Extra Small	Shared	768 MB	$0.04
Small	1	1.75 GB	$0.12
Medium	2	3.5 GB	$0.24
Large	4	7 GB	$0.48
Extra Large	8	14 GB	$0.96

The Azure Storage Queue is an eventual consistent, reliable, scalable and distributed web-scale message queue service that is ideal for small, short-lived, transient messages. The Azure queue does not guarantee the order of the messages, the deletion of messages or the availability of all the messages for a single request, although it guarantees eventual availability over multiple requests. Each message has a configurable visibility timeout. Once a client reads a message, the message will be invisible for other clients for the duration of the visibility time out. It will become visible for the other client once the visibility time expires unless the previous reader deletes it. The Azure Storage Table service offers a large-scale eventually consistent structured storage. Azure Table can contain a virtually unlimited number of entities (*aca* records or rows) that can be up to 1MB. Entities contain properties (*aca* cells), that can be up to 64KB. A table can be partitioned to store the data across many nodes for scalability. The Azure Storage Blob service provides a web-scale distributed storage service in which users can store and retrieve any type of data through a web services interface. Azure Blob services supports two types of Blobs, Page blobs that are optimized for random read/write operations and Block blobs that are optimized for streaming. Windows Azure Drive allows the users to mount a Page blob as a local NTFS volume.

Azure has a logical concept of regions that binds a particular service deployment to a particular geographic location or in other words to a data center. Azure also has an interesting concept of 'affinity groups' that can be specified for both services as well as for storage accounts. Azure tries its best to deploy services and storage accounts of a given affinity group close to each other to ensure optimized communication between each other.

2.5 MRRoles4Azure

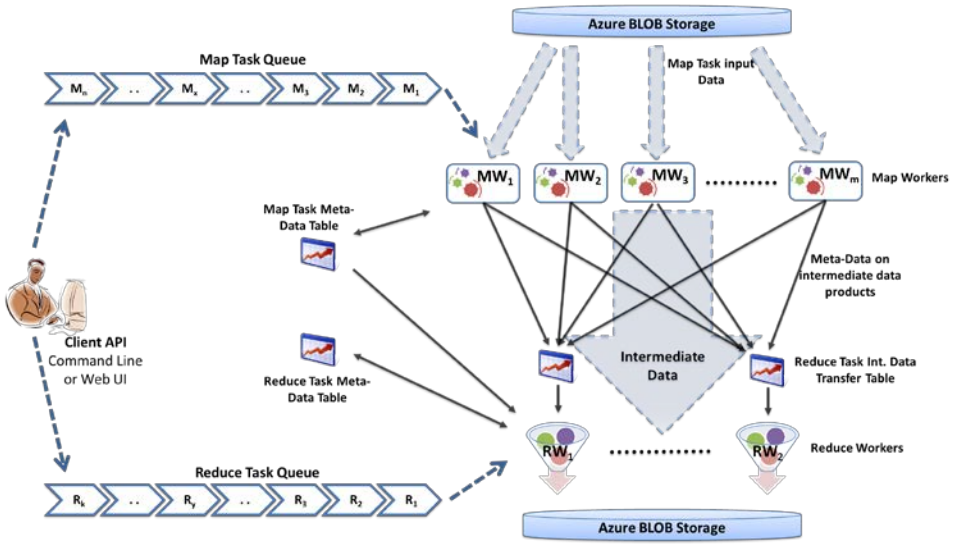

Figure 1. MRRoles4Azure Architecture [4]

MRRoles4Azure is a distributed decentralized MapReduce runtime for Windows Azure cloud platform that utilizes Azure cloud infrastructure services. MRRoles4Azure overcomes the latencies of cloud services by using sufficiently coarser grained map and reduce tasks. It overcomes the eventual data availability of cloud storage services through re-trying and explicitly designing the system so that it does not rely on the immediate availability of data across all distributed workers. As shown in

Figure 1, MRRoles4Azure uses Azure Queues for map and reduce task scheduling, Azure Tables for metadata and monitoring data storage, Azure Blob storage for data storage (input, output and intermediate) and the Window Azure Compute worker roles to perform the computations.

In order to withstand the brittleness of cloud infrastructures and to avoid potential single point failures, we designed MRRoles4Azure as a decentralized control architecture that does not rely on a central coordinator or a client side driver. MRRoles4Azure provides users with the capability to dynamically scale up/down the number of computing resources. MRRoles4Azure runtime dynamically schedules *Map* and *Reduce* tasks using a global queue achieving natural load balancing given sufficient amount of tasks. MR4Azure handles task failures and slower tasks through re-execution and duplicate executions respectively. MapReduce architecture requires the reduce tasks to ensure the receipt of all the intermediate data products from Map tasks before beginning the reduce phase. Since ensuring such a collective decision is not possible with the direct use of eventual consistent tables, MRRoles4Azure uses additional data structures on top of Azure Tables for this purpose. Gunarathne et al.[4] present more detailed description about MRRoles4Azure and show that MRRoles4Azure performs comparably to the other contemporary popular MapReduce runtimes.

2.6 Bio sequence analysis pipeline

The bio-informatics genome processing and visualizing pipeline[15] shown in Figure 2 inspired the application use cases analyzed in this paper. This pipeline uses SmithWatermann-GOTOH application, analyzed in section 4.3, or BLAST+ application, analyzed in section 4.4, for sequence alignment, Pairwise clustering for sequence clustering and the Multi-dimensional Scaling application, analyzed in section 4.1, to reduce the dimensions of the distance matrix to generate 3D coordinates for visualization purposes. This pipeline is currently in use to process and visualize hundreds of thousands of genomes with the ultimate goal of visualizing millions of genome sequences.

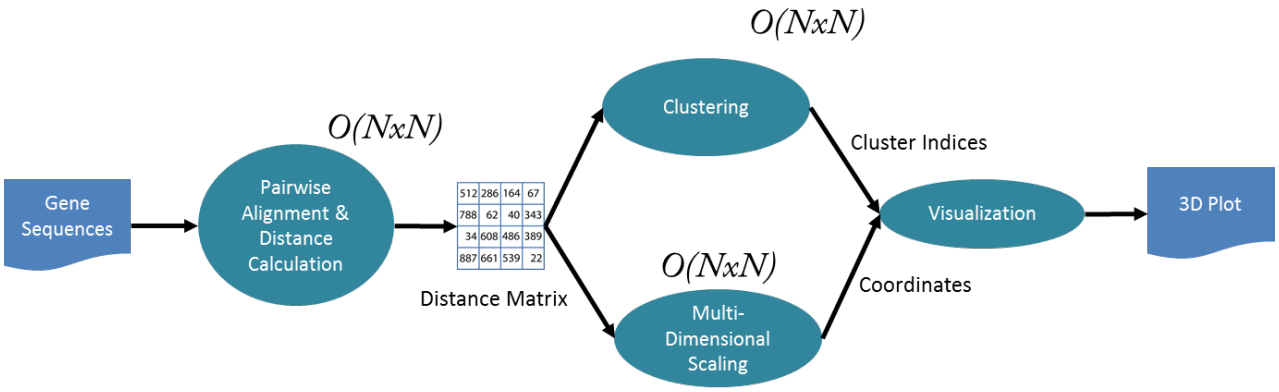

Figure 2. Bio sequence analysis pipeline

3 TWISTER4AZURE – ITERATIVE MAPREDUCE

Twister4Azure is an iterative MapReduce framework for Azure cloud that extends the MapReduce programming model to support data intensive iterative computations. Twister4Azure enables a wide array of large-scale iterative data analysis and data mining applications to utilize the Azure cloud platform in an easy, efficient and fault-tolerant manner. Twister4Azure extends the MRRoles4Azure architecture utilizing the scalable, distributed and highly available Azure cloud services as the underlying building blocks and employing a decentralized control architecture that avoids single point failures.

3.1 Twister4Azure Programming model

We identified the following requirements for choosing or designing a suitable programming model for scalable parallel computing in cloud environments.

1) Ability to express a sufficiently large and useful subset of large-scale data intensive and parallel computations,
2) Should be simple, easy-to-use and familiar to the users,
3) Suitable for efficient execution in the cloud environments.

We selected the data-intensive iterative computations as a suitable and sufficiently large subset of parallel computations that can be executed in the cloud environments efficiently, while using iterative MapReduce as the programming model.

3.1.1 Data intensive iterative computations

There exists a significant amount of data analysis, data mining and scientific computation algorithms that rely on iterative computations, where we can easily specify each iterative step as a MapReduce computation. Typical data-intensive iterative

computations follow the structure given in Code 1 and Figure 3. We can identify two main types of data in these computations, the loop invariant input data and the loop variant delta values. Loop variant delta values are the result or a representation of the result of processing the input data in each iteration. Computations of an iteration uses the delta values from the previous iteration as an input. Hence, these delta values need to be communicated to the computational components of the subsequent iteration. One example of such delta values would be the centroids in a KMeans Clustering computation (section 4.2). Single iterations of such computations are easy to parallelize by processing the data points or blocks of data points independently in parallel and performing synchronization between the iterations through communication steps.

Code 1 Typical data-intensive iterative computation

```
1:   k ← 0;
2:   MAX ← maximum iterations
3:   δ[0] ← initial delta value
4:   while ( k< MAX_ITER || f(δ[k], δ[k-1]) )
5:       foreach datum in data
6:           β[datum] ← process (datum, δ[k])
7:       end foreach
8:       δ[k+1] ← combine(β[])
9:       k ← k+1
10:  end while
```

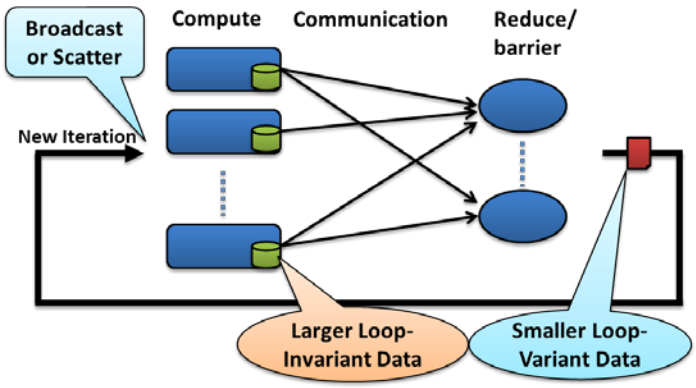

Figure 3. Structure of a typical data-intensive iterative application

Twister4Azure extends the MapReduce programming model to support easy parallelization of iterative computations by adding a Merge step to the MapReduce model and by adding an extra input parameter for the Map and Reduce APIs to support the loop-variant delta inputs. Code 1 depicts the structure of a typical data-intensive iterative application, while Code 2 depicts the corresponding Twister4Azure MapReduce representation. Twister4Azure will generate *map* tasks (line 5-7 in Code 1, line 8-12 in Code 2) for each data block and each *map* task will calculate a partial result, which will be communicated to the respective *reduce* tasks. The typical number of *reduce* tasks will be orders of magnitude less than the number of map tasks. Reduce tasks (line 8 in Code 1, line 13-15 in Code2) will perform any necessary computations, combine the partial results received and emit parts of the total reduce output. A single *merge* task (line 16-19 in Code 2) will merge the results emitted by the *reduce* tasks and will evaluate the loop conditional function (line 8 and 4 in Code1), often comparing the new delta results with the older delta results. Finally, the new delta output of the iteration will be broadcast or scattered to the map tasks of the next iteration (line 7 Code2). Figure 4. presents the flow of the Twister4Azure programming model.

Code 1 Typical data-intensive iterative computation

```
1:    k ← 0;
2:    MAX ← maximum iterations
3:    δ[0] ← initial delta value
4:    α ← true

5:    while ( k< MAX_ITER || α)
6:        distribute datablocks
7:        broadcast δ[k]
8:        map (datablock, δ[k])
9:              foreach datum in datablock
10:                 β[datum] ← process (datum, δ[k])
11:             end foreach
12:             emit (β)

13:       reduce (list of β)
14:             β' ← combine (list of β)
15:             emit (β')

16:       merge (list of β', δ[k])
17:             δ[k+1] ← combine (list of β)
18:             α ← f(δ[k], δ[k-1])
19:             emit (α, δ[k+1])

20:       k ←k+1
21:   end while
```

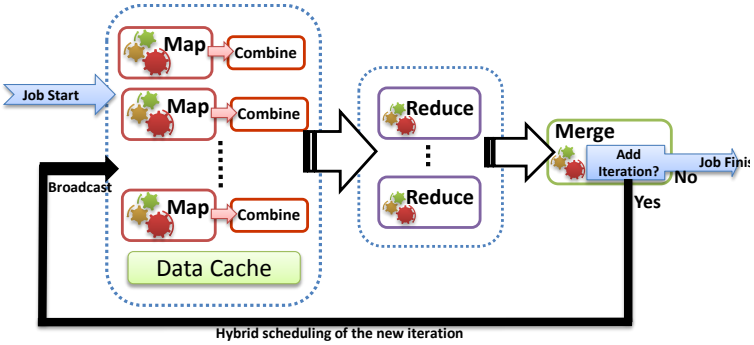

Figure 4. Twister4Azure programming model

3.1.2 Map and Reduce API

Twister4Azure extends the *map* and *reduce* functions of traditional MapReduce to include the loop variant delta values as an input parameter. This additional input parameter is a list of key, value pairs. This parameter can be used to provide an additional input through a broadcast operation or through a scatter operation. Having this extra input allows the MapReduce programs to perform Map side joins avoiding the significant data transfer and performance costs of reduce side joins[13] and avoiding the often unnecessary MapReduce jobs to perform reduce side joins. PageRank computation presented by Bu, Howe, et.al.[16] demonstrates the inefficiencies of using Map side joins for iterative computations. Twister4Azure non-iterative computations can also use this extra input to receive broadcast or scatter data to the Map & Reduce tasks.

```
Map(<key>, <value>, list_of <key,value>)
Reduce(<key>, list_of <value>, list_of <key,value>)
```

3.1.3 Merge

Twister4Azure introduces *Merge* as a new step to the MapReduce programming model to support iterative MapReduce computations. Merge task executes after the *Reduce* step. *Merge* Task receives all the *Reduce* outputs and the broadcast data for the current iteration as the inputs. There can only be one *merge* task for a MapReduce job. With *merge*, the overall flow of the iterative MapReduce computation would look as follows.

Map -> Combine -> Shuffle -> Sort -> Reduce -> Merge->Broadcast

Since Twister4Azure does not have a centralized driver to take control decisions, the *Merge* step serves as the "loop-test" in the Twister4Azure decentralized architecture. Users can add a new iteration, finish the job or schedule a new MapReduce job from the *Merge* task. These decisions can be made based on the number of iterations or by comparing the results from the previous iteration with the current iteration, such as the k-value difference between iterations for KMeansClustering. Users can use the results of the current iteration and the broadcast data to make these decisions. It is possible to specify the output of merge task as the broadcast data of the next iteration.

```
Merge(list_of <key,list_of<value>>,list_of <key,value>)
```

3.2 Data Cache

Twister4Azure locally caches the loop-invariant (static) input data across iterations in the memory and instance storage (disk) of worker roles. Data caching avoids the download, loading and parsing cost of loop invariant input data, which are reused in the iterations. These data products are comparatively larger sized and consist of traditional MapReduce key-value pairs. The caching of loop-invariant data gives significant speedups for the data-intensive iterative MapReduce applications. These speedups are even more significant in cloud environments as caching and reusing of data helps to overcome the bandwidth and latency limitations of cloud data storages.

Twister4Azure supports three levels of data caching, 1) instance storage (disk) based caching, 2) direct in-memory caching and 3) memory-mapped-file based caching. For the disk-based caching, Twister4Azure stores all the files it downloads from the Blob storage in the local instance storage. Local disk cache automatically serves all the requests for previously downloaded data products. Currently Twister4Azure do not support the eviction of the disk cached data products and assume that the input data blobs do not change during the course of a computation.

Selection of data for in-memory and memory-mapped-file based caching needs to be specified in the form of InputFormats. Twister4Azure provides several built-in InputFormat types that support both in-memory as well as memory-mapped-file based caching. Currently Twister4Azure performs least recently used (LRU) based cache eviction for these two types of caches.

Twister4Azure maintains a single instance of each data cache per worker-role shared across *map*, *reduce* and *merge* workers, allowing the reuse of cached data across different tasks as well as across any MapReduce application within the same job. Section 5 presents a more detailed discussion about the performance trade-offs and implementation strategies of the different caching mechanisms.

3.3 Cache Aware Scheduling

In order to take maximum advantage of the data caching for iterative computations, *Map* tasks of the subsequent iterations need to be scheduled with awareness of the data products that are cached in the worker-roles. If the loop-invariant data for a *map* task is present in the DataCache of a certain worker-role, then Twister4Azure should assign that particular map task to that particular worker-role. Decentralized architecture of Twister4Azure presents a challenge in this situation, as Twister4Azure does not have a central entity that has a global view of the data products cached in the worker-roles or has the ability to push the tasks to a specific worker-role.

As a solution to the above issue, Twister4Azure opted for a model in which the workers pick tasks to execute based on the data products they have in their DataCache and based on the information that is published in to a central bulletin board (an Azure table). Naïve implementation of this model requires all the tasks for a particular job to be advertised, making the bulletin board a bottleneck. We avoid this by locally storing the *Map* task execution histories (meta-data required for execution of a

map task) from the previous iterations. With this optimization, the bulletin board only advertises information about the new iterations. This allows the workers to start the execution of the map tasks for a new iteration as soon as the workers get the information about a new iteration through the bulletin board. High-level pseudo-code for the cache aware scheduling algorithm is given in Code 3. Every free map worker executes this algorithm. As shown in Figure 5. , Twister4Azure schedules new MapReduce jobs (non-iterative and 1st iteration of iterative) through Azure queues. Twister4Azure hybrid cache aware scheduling algorithm is currently configured to give priority for the iterations of the already executing tasks over new MapReduce computations.

Any tasks for an iteration that was not scheduled in the above manner will be added back to the task-scheduling queue and will be executed by the first available free worker. This ensures the eventual completion of the job and the fault tolerance of the tasks in the event of a worker failure and ensures the dynamic scalability of the system when new workers are added to the virtual cluster. Duplicate task execution can happen in very rare occasions due to the eventual consistency nature of the Azure Table storage. However, these duplicate executed tasks do not affect the accuracy of the computations due to the side effect free nature of the MapReduce programming model.

There are efforts that use multiple queues together to increase the throughput of the Azure Queue's. However, queue latency is not a significant bottleneck for Twister4Azure iterative computations as scheduling of only the first iteration depends on Azure queues.

Code 3 Cache aware hybrid decentralized scheduling algorithm. (Executes in every map worker)

```
1:   while (mapworker)
2:       foreach jobiter in bulletinboard
3:           cachedtasks[]← select tasks from taskhistories where
                    ((task.iteration == jobiter.baseiteration)  and
                    (memcache[] contains task.inputdata))
4:           foreach task in cachedtasks
5:               newtask ← new Task
                    (task.metadata, jobiter.iteration, …)
6:               if (newtask.duplicate())  continue;
7:               taskhistories.add(newTask)
8:               newTask.execute()
9:           end foreach
10:          // perform steps 3 to 8 for disk cache
11:          if (no task executed from cache)
12:              addTasksToQueue (jobiter)
13:      end foreach

14:      msg ← queue.getMessage())
15:      if (msg !=null)
16:          newTask ← new Task(msg.metadata, msg.iter, ….)
17:          if (newTask.duplicate())   continue;
18:          taskhistories.add(newTask)
19:          newTask.execute()
20:      else sleep()
21:  end while
```

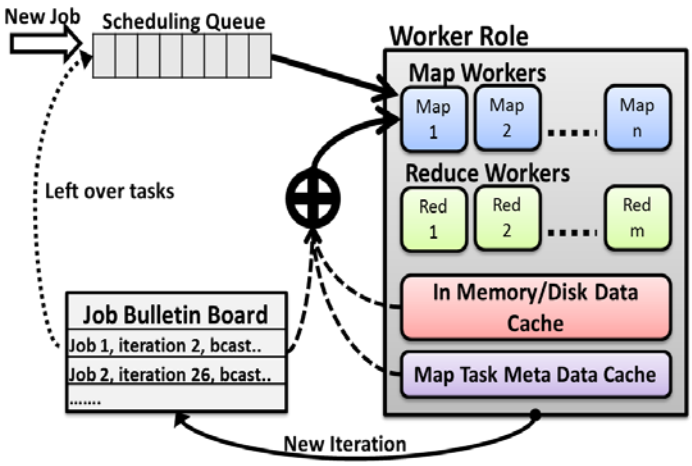

Figure 5. Cache Aware Hybrid Scheduling

3.4 Data broadcasting

The loop variant data (δ values in Code 1) needs to be broadcasted or scattered to all the tasks in an iteration. With Twister4Azure users can specify broadcast data for iterative as well as for non-iterative jobs. In typical data-intensive iterative computations, the loop-variant data (δ) is orders of magnitude smaller than the loop-invariant data.

Twister4Azure supports two types of data broadcasting methods, 1) using a combination of Azure blob storage and Azure tables, 2) Using a combination of direct TCP and Azure blob storage. First method broadcasts smaller data products using Azure tables and the larger data products using the blob storage. Hybrid broadcasting improves the latency and the performance when broadcasting smaller data products. This method works well for smaller number of instances and does not scale well for large number of instances.

The second method implements a tree based broadcasting algorithm that uses Windows Communication Foundation (WCF) based Azure TCP inter-role communication mechanism for the data communication as shown in Figure 6. This method supports configurable number of parallel outgoing TCP transfers per instance (three parallel transfers in Figure 6) , enabling the users and the framework to customize the number of parallel transfers based on the I/O performance of the instance type,

the scale of the computation and the size of the broadcast data. Since the direct communication is relatively unreliable in cloud environments, this method also supports an optional persistent backup that uses the Azure Blob storage. The broadcast data will get uploaded to the Azure Blob storage in the background and any instances that did not receive the TCP based broadcast data will be able to fetch the broadcast data from this persistent backup. This persistent backup also ensures that the output of each iteration will be stored persistently making it possible to roll back iterations if needed.

Twister4Azure supports caching of broadcast data ensuring that only a single retrieval or transmission of Broadcast data occurs per node per iteration. This increases the efficiency of broadcasting when there are more than one *map/reduce/merge* worker per worker-role and when there are multiple waves of *map* tasks per iteration. Some of our experiments contained up to 64 such tasks per worker-role per iteration.

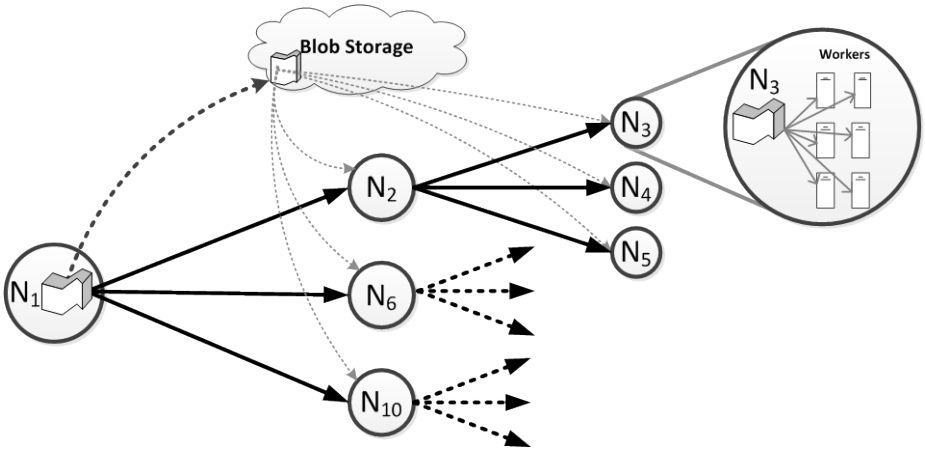

Figure 6. Tree based broadcast over TCP with Blob storage as the persistent backup. N_3 shows the utilization of data cache to share the broadcast data whithin an instance.

3.5 Intermediate data communication

MRRoles4Azure uses the Azure blob storage to store intermediate data products and the Azure tables to store meta-data about intermediate data products, which performed well for non-iterative applications. Based on our experience, tasks in iterative MapReduce jobs are of relatively finer granular making the intermediate data communication overhead more prominent. They produce a large number of smaller intermediate data products causing the Blob storage based intermediate data transfer model to under-perform. Hence, we opted for a hybrid model, in which Twister4Azure transfers smaller data products through the Azure tables. Twister4Azure uses the intermediate data product meta-data table entry itself to store the intermediate data products up to a certain size (currently 64kb that is the limit for a single item in an Azure table entry) and uses the blob storage for the data products that are larger than that limit. Twister4Azure also supports using direct TCP communication for the intermediate data communication. This intermediate communication can be made fault tolerant by uploading the data products to persistent Blob store in the background.

3.6 Fault Tolerance

Twister4Azure supports typical MapReduce fault tolerance through re-execution of failed tasks, ensuring the eventual completion of the iterative computations. Recent improvements to Azure queues service include the ability to update the queue messages, ability to dynamically extend the visibility time outs and to support for much longer visibility timeouts up to 7 days. We are currently working on improving the Queue based fault tolerance of Twister4Azure utilizing these newly introduced features of the Azure queues that allows us to support much more finer grained monitoring and fault tolerant as oppose to the time out based current fault tolerance implementation.

3.7 Other features

Twister4Azure also supports the deployment of multiple MapReduce applications in a single deployment, making it possible to utilize more than one MapReduce application inside an iteration of a single job. This also enables Twister4Azure to support workflow scenarios without redeployment. Twister4Azure also provides users with a web-based monitoring console from which they can monitor the progress of their jobs. Twister4Azure provides users with CPU and memory utilization information for their jobs and currently we are working on displaying this information graphically from the monitoring console.

Table 2. Evaluation cluster configurations

Cluster	CPU cores	Memory	I/O Performance	Compute Resource	OS
Azure Small	1 X 1.6 GHz	1.75 GB	100 MBPS, shared network infrastructure	Virtual instances on shared hardware	Windows Server
Azure Large	4 X 1.6 GHz	7 GB	400 MBPS, shared network infrastructure	Virtual instances on shared hardware	Windows Server
Azure Extra Large	8 X 1.6 GHz	14 GB	800 MBPS, shared network infrastructure	Virtual instances on shared hardware	Windows Server
EC2 XL	4 X ~2 Ghz	15 GB	High (EC2 terminology) , shared network infrastructure	Virtual instances on shared hardware	Linux
EC2 HCXL	8 X ~2.5 Ghz	7 GB	High (EC2 terminology) , shared network infrastructure	Virtual instances on shared hardware	Linux
HighMem	8 X 2.4 GHz (Intel®Xeon® CPU E5620)	192 GB	Gigabit ethernet, dedicated switch	Dedicated bare metal hardware	Linux
iDataPlex	8 X 2.33 GHz (Intel®Xeon® CPU E5410)	16 GB	Gigabit ethernet, dedicated switch	Dedicated bare metal hardware	Linux

4 Scientific Application Case Studies

4.1 Multi Dimensional Scaling

The objective of multi-dimensional scaling (MDS) is to map a data set in high-dimensional space to a user-defined lower dimensional space with respect to the pairwise proximity of the data points[17]. Dimensional scaling is used mainly in the visualizing of high-dimensional data by mapping them to two or three-dimensional space. MDS has been used to visualize data in diverse domains, including but not limited to bio-informatics, geology, information sciences, and marketing. We use MDS to visualize dissimilarity distances for hundreds of thousands of DNA and protein sequences to identify relationships.

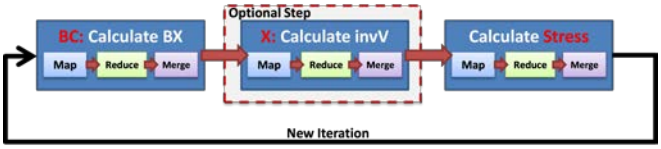

Figure 5. Twister4Azure Multi-Dimensional Scaling

In this paper we use Scaling by MAjorizing a COmplicated Function (SMACOF)[18], an iterative majorization algorithm. The input for MDS is an *N*N* matrix of pairwise proximity values, where *N* is the number of data points in the high-dimensional space. The resultant lower dimensional mapping in *D* dimensions, called the *X* values, is an *N*D* matrix.

The limits of MDS are more bounded by memory size than the CPU power. The main objective of parallelizing MDS is to leverage the distributed memory to support processing of larger data sets. In this paper, we implement the parallel SMACOF algorithm described by Bae et al[19]. This results in iterating a chain of three MapReduce jobs, as depicted in Figure 5. For the purposes of this paper, we perform an unweighted mapping that results in two MapReduce jobs steps per iteration, BCCalc and StressCalc. Each BCCalc Map task generates a portion of the total X matrix. MDS is challenging for Twister4Azure due to its relatively finer grained task sizes and multiple MapReduce applications per iteration.

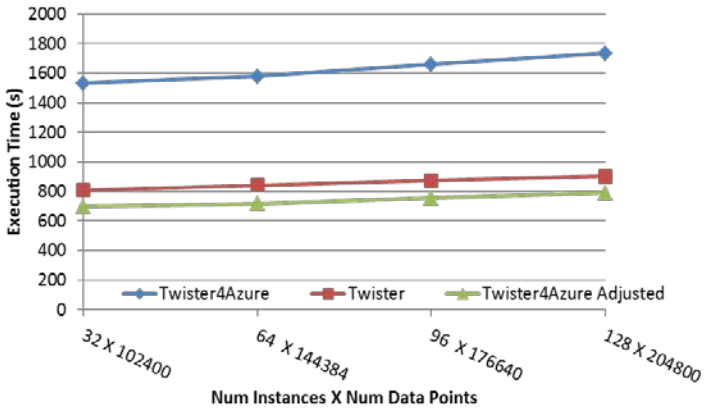

Figure 6. MDS weak scaling where we the workload per core is constant. Ideal is a straight horizontal line.

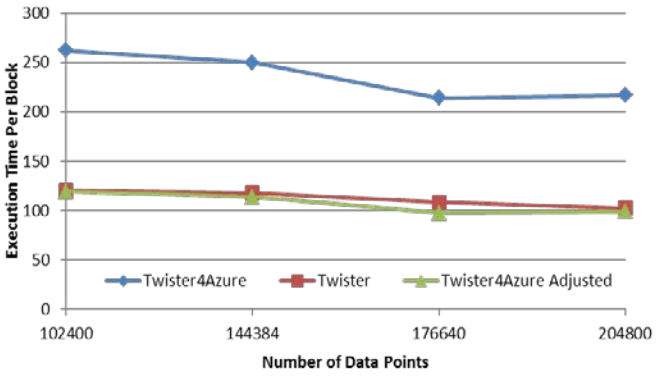

Figure 7. Data size scaling using 128 Azure small instances/cores, 20 iterations.

We compared the Twister4Azure MDS performance with Java HPC Twister MDS implementation. The Java HPC Twister experiment was performed in the HighMem cluster (Table 2). Twister4Azure tests were performed using Azure Large instances. Java HPC Twister results do not include the initial data distribution time. Figure 6 presents the execution time for weak scaling, where we increase the number of compute resources while keeping the work per core constant (work ~ number of cores). We notice that Twister4Azure exhibits encouraging performance and scales similar to the Java HPC Twister. Figure 7 shows that MDS performance scales well with increasing data sizes.

The JavaHPCTwister cluster is a bare metal cluster with dedicated network and with faster processors. It is expected to be significantly faster than the cloud environment for the same number of CPU cores. The adjusted performance estimate the performance of the two systems when the underlying hardware differences are negated. The *Twister4Azure adjusted* (t_a) line in Figure 6 and 7 depicts the performance of Twister4Azure normalized according to the sequential MDS BC calculation and Stress calculation performance difference between the Azure(t_{sa}) instances and the nodes in Cluster(t_{sc}) environment used for Java HPC Twister. It is calculated using t_a x (t_{sc}/t_{sa}). This estimation however does not account for the overheads that remain constant irrespective of the computation time. In the above testing, the total number of tasks per job ranged from 10240 to 40960, proving Twister4Azure's ability to support large number of tasks effectively.

4.2 KMeans Clustering

Clustering is the process of partitioning a given data set into disjoint clusters. The use of clustering and other data mining techniques to interpret very large data sets has become increasingly popular, with petabytes of data becoming commonplace. The K-Means clustering[20] algorithm has been widely used in many scientific and industrial application areas due to its simplicity and applicability to large data sets. We are currently working on a scientific project that requires clustering of several TeraBytes of data using KMeansClustering and millions of centroids.

K-Means clustering is often implemented using an iterative refinement technique, in which the algorithm iterates until the difference between cluster centers in subsequent iterations, i.e. the *error*, falls below a predetermined threshold. Each iteration performs two main steps, the cluster *assignment step*, and the centroids *update step*. In the MapReduce implementation, *assignment step* is performed in the Map Task and the *update step* is performed in the Reduce task. Centroid data is broadcasted at the beginning of each iteration. Intermediate data communication is relatively costly in KMeans clustering as each Map Task outputs data equivalent to the size of the centroids in each iteration.

Figure 8(a) presents the Twister4Azure KMeansClustering performance on different types of Azure compute instances, with the number of map workers per instance equal to the number of cores of the instance. We did not notice any significant performance variations across the instances for KMeansClustering. Figure 8(b) shows that the performance scales well with the number of iterations. The performance improvement with a higher number of iterations in Figure 8(b) is due to the initial data download/parsing overhead distributing over the iterations. Figure 8(c) presents the number of map tasks executing at a given time throughout the job. The job consisted of 256 map tasks per iteration, generating two waves of map tasks per iteration. The dips represent the synchronization at the end of iterations. The gaps between the bars represent the total overhead of the intermediate data communication, reduce task execution, merge task execution, data broadcasting and the new iteration scheduling that happens between iterations. According to the graph, such overheads are relatively small for the KMeansClustering application. Figure 9(c) depicts the execution time of MapTasks across the whole job. The higher execution time of the tasks in the first iteration is due to the overhead of initial data downloading, parsing and loading, which is an indication of the performance improvement we get in subsequent iterations due to the data caching.

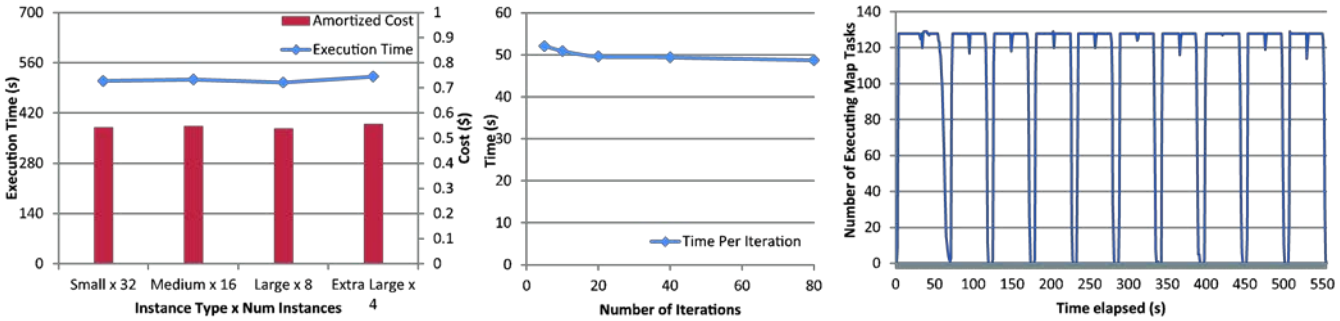

Figure 8. Twister4Azure KMeansClustering (20D data with 500 centroids, 32 cores). **Left(a)**: Instance type study with 10 iterations 32 million data points **Center(b)** : Time per iteration with increasing number of iterations 32 million data points. **Right(c):** Twister4Azure executing Map Task histogram for 128 million data points in 128 Azure small instances

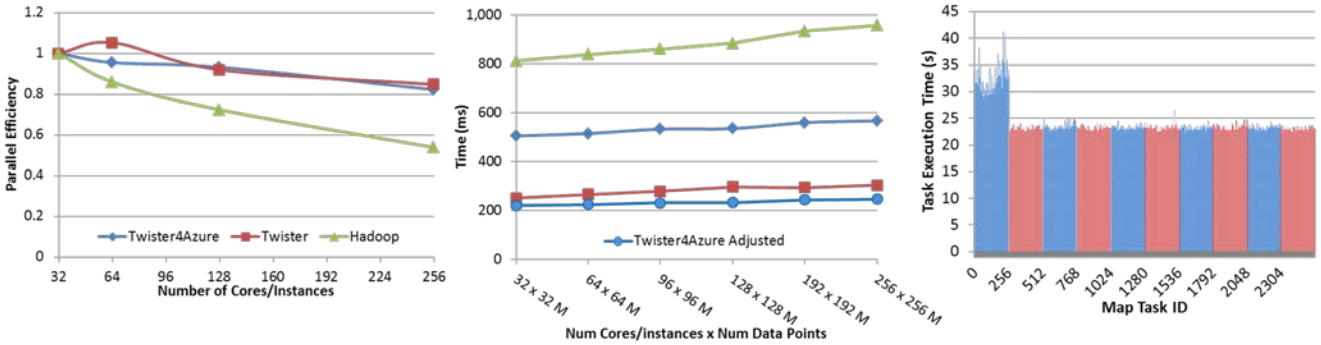

Figure 9. KMeansClustering Scalability. **Left(a):** Relative parallel efficiency of strong scaling using 128 million data points. **Center(b):** Weak scaling. Workload per core is kept constant (ideal is a straight horizontal line). **Right(c)** :Twister4Azure Map task execution time histogram for 128 million data points in 128 Azure small instances

We compared the Twister4Azure KMeans Clustering performance with implementations of Java HPC Twister and Hadoop. The Java HPC Twister and Hadoop experiments were performed in a dedicated iDataPlex cluster (Table 2). Twister4Azure tests were performed using the Azure small instances that contain single CPU core. Java HPC Twister results do not include the initial data distribution time. Figure 9(a) presents the relative (relative to the smallest parallel test with 32 cores/instances) parallel efficiency of KMeans Clustering for strong scaling, in which we keep the amount of data constant and increase the number of instances/cores. Figure 9(b) presents the execution time for weak scaling, wherein we increase the number of compute resources while keeping the work per core constant (work ~ number of nodes). We notice that Twister4Azure performance scales well up to 256 instances in both experiments. In 9(a), the relative parallel efficiency of JavaHPCTwister for 64 cores is greater than 1. We believe the memory load was a bottleneck in the 32 core experiment, while it's not the case for 64 core experiment. We used direct TCP intermediate data transfer and Tree based TCP broadcasting when performing these experiments. Tree based TCP broadcasting scaled well up to the 256 Azure small instances. Using this result, we can hypothesis that our Tree based broadcasting algorithm will scale well for 256 Azure Extra Large instances (2048 total number of CPU cores) as well, since the workload, communication pattern and other properties remain same irrespective of the instance type.

The Twister4Azure adjusted (t_a) line in Figure 9(b) depicts the performance of Twister4Azure normalized according to the ratio of Kmeans sequential performance difference between Azure (t_{sa}) instances and the Kmeans sequential performance in the cluster (t_{sc}) nodes calculated using the t_a x (t_{sc}/t_{sa}) equation. This estimation, however, does not take into account the overheads that remain constant irrespective of the computation time. All tests were performed using 20 dimensional data and 500 centroids.

4.3 Sequence alignment using SmithWaterman GOTOH

SmithWaterman [21] algorithm with GOTOH [22] (SWG) improvement is used to perform pairwise sequence alignment on two FASTA sequences. We use SWG application kernel in parallel to calculate the all-pairs dissimilarity of a set of *n* sequences resulting in *n*n* distance matrix. Set of map tasks for a particular job are generated using the blocked decomposition

of strictly upper triangular matrix of the resultant space. Reduce tasks aggregate the output from a row block. In this application, the size of the input data set is relatively small, while the size of the intermediate and the output data are significantly larger due to the n^2 result space, stressing the performance of inter-node communication and output data storage. SWG can be considered as a memory-intensive application. More details about the Hadoop-SWG application implementation can be found in [23]. The Twister4Azure SWG implementation also follows the same architecture and blocking strategy as in the Hadoop-SWG implementation. Twister4Azure SWG uses NAligner [24] as the computational kernel.

We performed the SWG weak scaling test from Gunarathne et al.[4] using Twister4Azure to compare the performance of Twister4Azure SWG implementation on Azure Small instances (Table 2) with Amazon ElasticMapReduce using EC2 XL (Table 2) instances and Apache Hadoop implementation on iDataPlex cluster (Table 2). Figure 10 shows that the Twister4Azure SWG performs comparable to the Amazon EMR and Apache Hadoop. The performance of Twister4Azure SWG lied between +/- 2% of MRRoles4Azure SWG performance[4], confirming that the extra complexity of Twister4Azure has not adversely affected the non-iterative MapReduce performance.

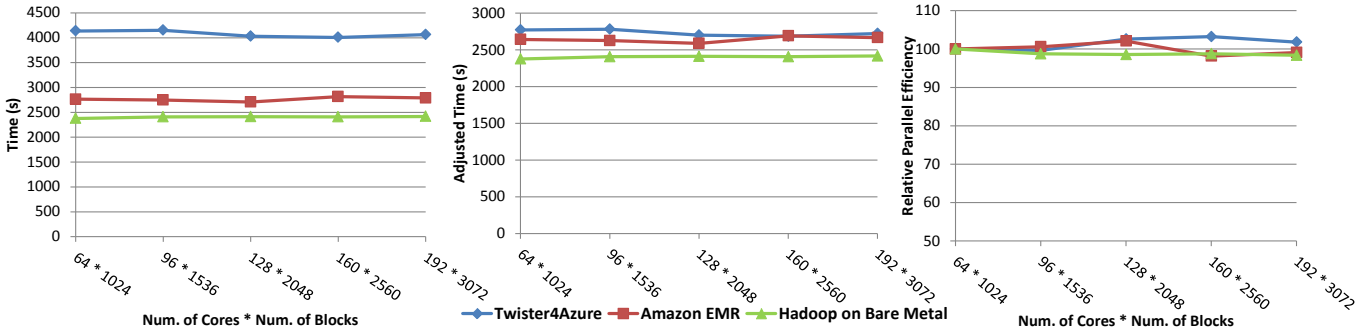

Figure 10. Twister4Azure SWG performance. **Left** : Raw execution time. **Center** : Execution time adjusted to compensate the sequential performance difference in the environments relative to Hadoop Bare Metal **Right**: Parallel efficiency relative to 64*1024 test case

4.4 Sequence searching using Blast

NCBI BLAST+ [1] is the latest version of popular BLAST program, that is used to handle sequence similarity searching. Queries are processed independently and have no dependencies between them making it possible to use multiple BLAST instances to process queries in a pleasingly parallel manner. We performed the BLAST+ scaling speedup performance experiment from Gunarathne, et al[3] using Twister4Azure Blast+ to compare the performance with Amazon EC2 classic cloud with EC2 and Apache Hadoop BLAST+ implementations. We used Azure Extra Large instances (Table 2) with 8 Map workers per node for the Twister4Azure BLAST experiments. We used a sub-set of a real-world protein sequence data set (100 queries per map task) as the input BLAST queries and used NCBI's non-redundant (NR) protein sequence database. All of the implementations downloaded and extracted the compressed BLAST database to a local disk of each worker prior to processing of the tasks. Twister4Azure's ability to specify deploy time initialization routines was used to download and extract the database. The performance results do not include the database distribution times.

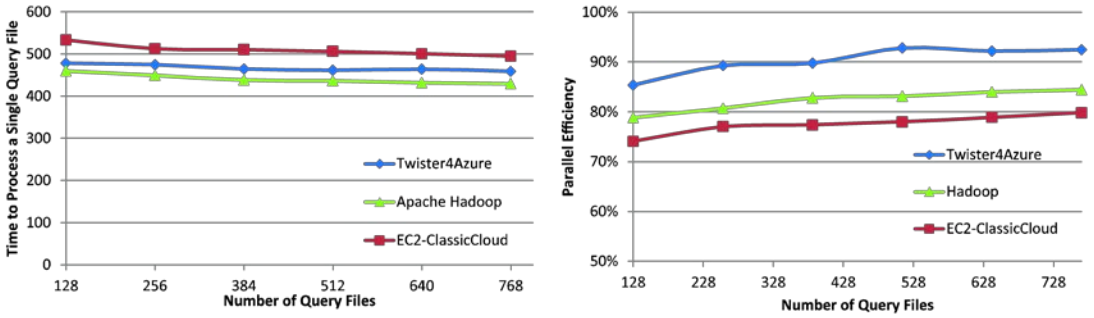

Figure 11. Twister4Azure BLAST performance. **Left** : Time to process a single query file. **Right**: Absolute parallel efficiency

The Twister4Azure BLAST+ absolute efficiency (Figure 11) was better than the Hadoop and EMR implementations. Additionally the Twister4Azure performance was comparable to the performance of the Azure Classic Cloud BLAST results that we had obtained earlier. This shows that the performance of BLAST+ is sustained in Twister4Azure, even with the added complexity of MapReduce and iterative MapReduce.

5 Performance Considerations For Large Scale Iterative Applications on Azure

In this section, we perform a detailed performance analysis of several factors that affect the performance of large-scale parallel iterative MapReduce applications on Azure, in the context of Multi-Dimensional-Scaling application presented in Section 4.1. These applications typically perform tens to hundreds of iterations. Hence, we focus mainly on optimizing the performance of the majority of iterations, while giving a lower priority in optimizing the initial iteration.

In this section we use a dimension-reduction computation of 204800 * 204800 element input matrix, partitioned in to 1024 data blocks (number of map tasks is equal to the number of data blocks), using 128 cores and 20 iterations as our use case. We focus mainly on the BCCalc computation as it is much more computationally intensive than the StressCalc computation. Table 2 presents the execution time analysis of this computation under different mechanisms. The 'Task Time' in Table 2 refers to the end-to-end execution time of BCCalc Map Task including the initial scheduling, data acquiring and the output data processing time. The 'Map F^n Time' refers to the time taken to execute the Map function of the BCCalc computation excluding the other overheads. In order to eliminate the skewedness of 'Task Time' introduced by the data download in the first iterations, we calculate the averages and standard deviations excluding the first iteration. '# of slow tasks' is defined as the number of tasks that take more than twice the average time for that particular metric. We used a single Map worker per instance in the Azure small instances and four Map workers per instances in the Azure Large instances.

5.1 Local Storage Caching

As discussed in section 3.1, it is possible to optimize iterative MapReduce computations by caching the loop-invariant input data across the iterations. We use the Azure Blob storage as the input data storage for Twister4Azure computations. Twister4Azure supports local instance (disk) storage caching as the simplest form of data caching. Local storage caching allows the subsequent iterations (or different applications or tasks in the same iteration) to reuse the input data from the local storage rather than fetching them from the Azure Blob Storage. This resulted in speedups of more than 50 % (estimated) over a non-cached MDS computation of the sample use case. However, local storage caching causes the applications to read and parse data from the instances storage each time the data is used. On the other hand, on-disk caching puts minimal strain on the instance memory.

Twister4Azure also supports the 'in-memory caching' of the loop-invariant data across iterations. With in-memory caching, Twister4Azure fetch the data from the Azure Blob storage, parse and load them in to the memory during the first iteration. After the first iteration, these data products remain in memory throughout the course of the computation for reuse by the subsequent iterations, eliminating the overhead of reading and parsing data from disk. As shown in Table 2, in-memory caching improved the average run time of the BCCalc map task by approximately 36% and the total run time by approximately 22% over disk based caching. Twister4Azure performs cache-invalidation for in-memory cache using Least Recently Used (LRU) policy. In a typical Twister4Azure computation, the loop-invariant input data stays in the in-memory cache for the duration of the computation, while Twister4Azure caching policy will evict the broadcast data for iterations after the particular iterations.

As mentioned in section 3.3, Twister4Azure supports cache-aware scheduling for data cached in-memory as well as for data cached in local-storage.

Table 2 Execution time analysis of a MDS computation. 20480 * 20480 input data matrix, 128 total cores, 20 iterations. 20480 BCCalc map tasks.

Mechanism	Instance Type	Total Execution Time (s)	Task Time (BCCalc)			Map F^n Time (BCCalc)		
			Average (ms)	STDEV (ms)	# of slow tasks	Average (ms)	STDEV (ms)	# of slow tasks
Disk Cache only	small * 1	2676	6,390	750	40	3,662	131	0
In-Memory Cache	small * 1	2072	4,052	895	140	3,924	877	143
	large * 4	2574	4,354	5,706	1025	4,039	5,710	1071
Memory Mapped File (MMF) Cache	small * 1	2097	4,852	486	28	4,725	469	29
	large * 4	1876	5,052	371	6	4,928	357	4

5.2 Non-Deterministic Performance Anomalies

When using in-memory caching, we started to notice occasional non-deterministic fluctuations of the Map function execution times in some of the tasks (143 slow Map F^n time tasks in row 2 of Table 2). These slow tasks, even though few,

affect the performance of the computation significantly because the execution time of a whole iteration is dependent on the slowest task of the iteration. Even though Twister4Azure supports the duplicate execution of the slow tasks, duplicate tasks for non-initial iterations are often more costlier than the total execution time of a slow task that uses data from a cache, as the duplicate task would have to fetch the data from the Azure Blob Storage. Figure 12 top row gives an example of an execution trace of a computation that shows this performance fluctuation. The performance fluctuations cause occasional unusual high task execution times in the left graph while the tail of the bars in the right hand graph shows the effect of performance fluctuations for the execution time of iterations. With further experimentation, we were able to narrow down the cause of this anomaly to the use of large amount of memory, including the in-memory data cache, within a single .NET process. One may assume that using only local storage caching would perform better as it reduces the load on memory. We in fact found that the Map function execution times were very stable when using local storage caching (zero slow tasks and smaller standard deviation in Map F^n time in row 1 of Table 2). However, the 'Task Time' that includes the disk reading time is unstable when local-storage cache is used (40 slow 'Task Time' tasks in row 1 of Table 2).

A memory-mapped file contains the contents of a file mapped to the virtual memory and can be read or modified directly through memory. Memory-mapped files can be shared across multiple processes and can be used to facilitate inter-process communication. .NET framework version 4 introduces first class support for memory-mapped files to .NET world. .NET memory mapped files facilitate the creation of a memory-mapped file directly in the memory, with no associated physical file, specifically to support inter-process data sharing. We exploit this feature by using such memory-mapped files to implement the Twister4Azure in-memory data cache. In this implementation, Twister4Azure fetch the data directly to the memory-mapped file and the memory mapped file will be reused across the iterations. The Map function execution times become stable with the memory-mapped file based cache implementation (row 4 and 5 of Table 2).

With our earlier in-memory cache implementation, the performance on larger Azure instances (with number of workers equal to the number of cores) was very unstable (row 3 of Table 2). On the contrary, when using memory-mapped caching, the execution times were more stable on the larger instances than in smaller instances (row 4 vs 5 in Table 2). The ability to utilize larger instances effectively is a significant advantage as usage of larger instances improves the data sharing across workers, facilitate better load balancing within the instances, provide better deployment stability, reduce the data broadcasting load and simplify the cluster monitoring.

The memory-mapped file based caching requires the data to be parsed (decoded) each time the data is used adding an overhead to the task execution times. In order to avoid duplicate loading of data products to memory, we use real time data parsing in the case of memory-mapped files. Hence, the parsing overhead becomes part of the Map function execution time. However, we find that the execution time stability advantage outweighs the added cost. In Table 2, we present results using Small and Large Azure instances. Unfortunately, we were not able to utilize Extra Large instances during the course of our testing due to an Azure resource outage bound to our 'affinity group'. We believe the computations will be even more stable in Extra Large instances. Middle row of Figure 12 presents an execution trace of a job that uses Memory Mapped file based caching.

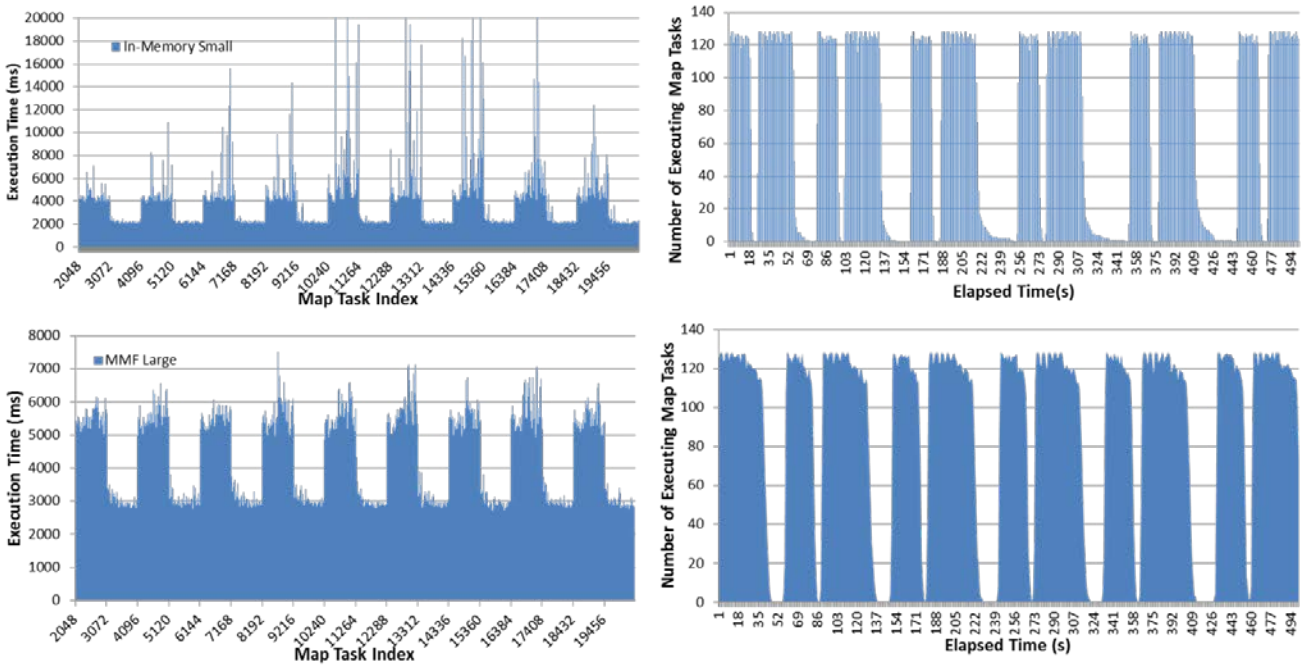

Figure 12: Execution traces of MDS iterative MapReduce computations using Twister4Azure. TOP ROW: Using in-memory caching on small instances. MIDDLE ROW: Using Memory-Mapped file based caching on Large instances.. LEFT COLUMN: shows the execution time of tasks in each iteration. The taller bars represent the MDSBCCalc computation, while the shorter bars represent the MDSStressCalc computation. A pair of BCCalc and StressCalc bars represents an iteration. RIGHT COLUMN: Number of active map tasks of the computation at a given time (A 500 second view from the 3rd iteration onwards). The wider bars represent BCCalc computations, while the narrower bars represent StressCalc computations. The gaps between the computations represent the overhead of task scheduling, reduce task execution, and merge task execution and data broadcasting.

6 Related Work

CloudMapReduce[25] for Amazon Web Services (AWS) and Google AppEngine MapReduce[26] follow an architecture similar to MRRoles4Azure, in which they utilize the cloud services as the building blocks. Amazon ElasticMapReduce[27] offers Apache Hadoop as a hosted service on the Amazon AWS cloud environment. However, none of them support iterative MapReduce. Spark[28] is a framework implemented using Scala to support interactive MapReduce like operations to query and process read-only data collections, while supporting in-memory caching and re-use of data products.

Azure HPC scheduler is a new Azure feature that enables the users to launch and manage high-performance computing (HPC) and other parallel applications in the Azure environment. Azure HPC scheduler supports parametric sweeps, Message Passing Interface (MPI) and LINQ to HPC applications together with a web-based job submission interface. AzureBlast[29] is an implementation of parallel BLAST on Azure environment that uses Azure cloud services with an architecture similar to the Classic Cloud model, which is a predecessor to Twister4Azure. CloudClustering[30] is a prototype KMeansClustering implementation that uses Azure infrastructure services. CloudClustering uses multiple queues (single queue per worker) for job scheduling and supports caching of loop-invariant data.

6.1 Microsoft Daytona

Microsoft Daytona[12] is a recently announced iterative MapReduce runtime developed by Microsoft Research for Microsoft Azure Cloud Platform. It builds on some of the ideas of the earlier Twister system. Daytona utilizes Azure Blob Storage for storing intermediate data and final output data enabling data backup and easier failure recovery. Daytona supports caching of static data between iterations. Daytona combines the output data of the Reducers to form the output of each iteration. Once the application has completed, the output can be retrieved from Azure Blob storage or can be continually processed by using other applications. In addition to the above features, which are similar to Twister4Azure, Daytona also provides automatic environment deployment and data splitting for MapReduce computations and claims to support a variety of data broadcast patterns between the iterations. However, as oppose to Twister4Azure, Daytona uses a single master node based controller to drive and manage the computation. This centralized controller substitute the 'Merge' step of Twister4Azure, but makes Daytona prone to single point of failures.

Currently Excel DataScope is presented as an application of Daytona. Users can upload data in their Excel spreadsheet to the DataScope service or select a data set already in the cloud, and then select an analysis model from our Excel DataScope research ribbon to run against the selected data. The results can be returned to the Excel client or remain in the cloud for further processing and/or visualization. Daytona is available as a "Community Technology Preview" for academic and non-commercial usage.

6.2 Haloop

Haloop[16] extends Apache Hadoop to support iterative applications and supports on-disk caching of loop-invariant data as well as loop-aware scheduling. Similar to Java HPC Twister and Twister4Azure, Haloop also provides a new programming model, which includes several APIs that can be used for expressing iteration related operations in the application code.

However Haloop does not have an explicit Merge operation similar to Twister4Azure and uses a separate MapReduce job to perform the Fixpoint evaluation for the terminal condition evaluation. HaLoop provides a high-level query language, which is not available in either Java HPC Twister or Twister4Azure.

HaLoop performs centralized loop aware task scheduling to accelerate iterative MapReduce executions. Haloop enables data reuse across iterations, by physically co-locating tasks that process the same data in different iterations. In HaLoop, the first iteration is scheduled similarly to traditional Hadoop. After that, the master node remembers the association between data and node and the scheduler tries to retain previous data-node associations in the following iterations. HaLoop also supports on-disk caching for reducer input data and reducer output data. Reducer input data cache stores the intermediate data generated by the map tasks, which optimizes the Reduce side joins. Twister4Azure additional input parameter for Map API eliminates the need for such reduce side joins. Reducer output data-cache is specially designed to support Fixpoint Evaluations using the output data from older iterations. Twister4Azure currently do not support this feature.

7 Conclusions and Future Work

We have developed Twister4Azure, a novel iterative MapReduce distributed computing runtime for Azure cloud. We implemented four significant scientific applications using Twister4Azure – MDS, Kmeans Clustering, SWG sequence alignment and BLAST+. Twister4Azure enables the users to easily and efficiently perform large-scale iterative data analysis for scientific applications on a commercial cloud platform.

In developing Twister4Azure, we encounter the challenges of scalability and fault tolerance unique to utilizing the cloud interfaces. We have developed a solution to support multi-level caching of loop-invariant data across iterations as well as caching of any reused data (e.g. broadcast data) and proposed a novel hybrid scheduling mechanism to perform cache-aware scheduling.

We presented MDS and Kmeans Clustering as iterative scientific applications of Twister4Azure. Experimental evaluation shows that MDS using Twister4Azure on public cloud environment scales similar to the Java HPC Twister MDS on a local cluster. Further, the Kmeans Clustering using Twster4Azure with virtual instances outperforms Apache Hadoop in a local cluster by a factor of 2 to 4 and exhibits performance comparable to Java HPC Twister running on a local cluster. We consider the results presented in this paper as one of the first study of Azure performance for large-scale iterative scientific applications.

Twister4Azure and Java HPC Twister illustrate our roadmap to a cross platform new programming paradigm supporting large scale data analysis, an important area for both HPC and eScience applications.

Acknowledgment

This work is funded in part by the Microsoft Azure Grant. We would also like to thank Geoffrey Fox and Seung-Hee Bae for many discussions.

References

[1] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. Papadopoulos, K. Bealer, T.L. Madden, BLAST+: architecture and applications, BMC Bioinformatics 2009, 10:421, (2009).

[2] J. Ekanayake, T. Gunarathne, J. Qiu, Cloud Technologies for Bioinformatics Applications, Parallel and Distributed Systems, IEEE Transactions on, 22 (2011) 998-1011.

[3] T. Gunarathne, T.-L. Wu, J.Y. Choi, S.-H. Bae, J. Qiu, Cloud computing paradigms for pleasingly parallel biomedical applications, Concurrency and Computation: Practice and Experience, (2011) n/a-n/a.

[4] T. Gunarathne, W. Tak-Lon, J. Qiu, G. Fox, MapReduce in the Clouds for Science, in: Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, 2010, pp. 565-572.

[5] Twister4Azure, Retrieved May 10, 2012, http://salsahpc.indiana.edu/twister4azure/.

[6] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM, 51 (2008) 107-113.

[7] Apache Hadoop, Retrieved May 10, 2012, from ASF: http://hadoop.apache.org/core/.
[8] *Hadoop Distributed File System HDFS*, Retrieved May 10, 2012; http://hadoop.apache.org/hdfs/.
[9] J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.Bae, J.Qiu, G.Fox, Twister: A Runtime for iterative MapReduce, in: Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010, ACM, Chicago, Illinois, 2010.
[10] B. Zhang, Y. Ruan, T.-L. Wu, J. Qiu, A. Hughes, G. Fox, Applying Twister to Scientific Applications, in: CloudCom 2010, IUPUI Conference Center Indianapolis, 2010.
[11] Apache, ActiveMQ, Retrieved May 10, 2012, from :http://activemq.apache.org/.
[12] Microsoft Daytona, Retrieved Feb 1, 2012, from : http://research.microsoft.com/en-us/projects/daytona/.
[13] J. Lin, C. Dyer, Data-Intensive Text Processing with MapReduce, Synthesis Lectures on Human Language Technologies, 3 (2010) 1-177.
[14] Windows Azure Compute, Retrieved July 25th 2011; http://www.microsoft.com/windowsazure/features/compute/.
[15] Judy Qiu, Thilina Gunarathne, Jaliya Ekanayake, Jong Youl Choi, Seung-Hee Bae, Hui Li, Bingjing Zhang, Yang Ryan, Saliya Ekanayake, Tak-Lon Wu, Scott Beason, Adam Hughes, Geoffrey Fox, Hybrid Cloud and Cluster Computing Paradigms for Life Science Applications, in: 11th Annual Bioinformatics Open Source Conference BOSC 2010, Boston, 2010.
[16] Yingyi Bu, Bill Howe, Magdalena Balazinska, Michael D. Ernst, HaLoop: Efficient Iterative Data Processing on Large Clusters, in: The 36th International Conference on Very Large Data Bases, VLDB Endowment, Singapore, 2010.
[17] J.B. Kruskal, M. Wish, *Multidimensional Scaling*, Sage Publications Inc., 1978.
[18] J. de Leeuw, Convergence of the majorization method for multidimensional scaling, Journal of Classification, 5 (1988) 163-180.
[19] S.-H. Bae, J.Y. Choi, J. Qiu, G.C. Fox, Dimension reduction and visualization of large high-dimensional data via interpolation, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, ACM, Chicago, Illinois, 2010, pp. 203-214.
[20] S. Lloyd, Least squares quantization in PCM, Information Theory, IEEE Transactions on, 28 (1982) 129-137.
[21] T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology, *147* (1981) 195-197.
[22] O. Gotoh, An improved algorithm for matching biological sequences, Journal of Molecular Biology, 162 (1982) 705-708.
[23] J. Ekanayake, T. Gunarathne, J. Qiu, G. Fox, Cloud Technologies for Bioinformatics Applications, Accepted for publication in Journal of IEEE Transactions on Parallel and Distributed Systems, (2010).
[24] JAligner., Retrieved December, 2009; http://jaligner.sourceforge.net.
[25] cloudmapreduce, Retrieved Sep. 20, 2010: http://code.google.com/p/cloudmapreduce/.
[26] AppEngine MapReduce, Retrieved July 25th 2011; http://code.google.com/p/appengine-mapreduce.
[27] Amazon Web Services, Retrieved Mar. 20, 2011, from Amazon: http://aws.amazon.com/.
[28] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: Cluster Computing with Working Sets, in: 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10), Boston, 2010.
[29] Wei Lu, Jared Jackson, Roger Barga, AzureBlast: A Case Study of Developing Science Applications on the Cloud, in: ScienceCloud: 1st Workshop on Scientific Cloud Computing co-located with HPDC 2010 (High Performance Distributed Computing), ACM, Chicago, IL, 2010.
[30] A. Dave, W. Lu, J. Jackson, R. Barga, CloudClustering: Toward an iterative data processing pattern on the cloud, in: First International Workshop on Data Intensive Computing in the Clouds, Anchorage, Alaska, 2011.