Future Generation Computer Systems 29 (2013) 1035–1048

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

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, 150 S. Woodlawn Ave., Bloomington, IN 47405, USA

A R T I C L E   I N F O

Article history:
Received 7 February 2012
Received in revised form
15 May 2012
Accepted 30 May 2012
Available online 23 June 2012

Keywords:
Iterative MapReduce
Cloud computing
HPC
Scientific applications
Azure

A B S T R A C T

Recent advances in data-intensive computing for science discovery are fueling a dramatic growth in the 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 in which scientists can perform data analytics. The challenges to large-scale distributed computations on cloud environments demand innovative computational frameworks that are specifically tailored for cloud characteristics to easily and effectively harness the power of *clouds*. Twister4Azure is a distributed decentralized iterative MapReduce runtime for Windows Azure Cloud. Twister4Azure extends the familiar, easy-to-use MapReduce programming model with iterative extensions, enabling a fault-tolerance execution of a wide array of data mining and data analysis applications on the Azure cloud. Twister4Azure utilizes the scalable, distributed and highly available Azure cloud services as the underlying building blocks, and employs a decentralized control architecture that avoids single point failures. Twister4Azure optimizes the iterative computations using a multi-level caching of data, a cache-aware decentralized task scheduling, hybrid tree-based data broadcasting and hybrid intermediate data communication. This paper presents the Twister4Azure iterative MapReduce runtime and a study of four real world data-intensive scientific applications implemented using Twister4Azure – two iterative applications, Multi-Dimensional Scaling and KMeans Clustering; and two pleasingly parallel 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.

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 has been 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 the 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, while outperforming traditional MapReduce runtimes for iterative MapReduce computations. Twister4Azure has minimal management and maintenance overheads and provides users with the capability to dynamically scale up or down the amount

[*] Corresponding author. Tel.: +1 8123914212.
E-mail addresses: tgunarat@indiana.edu (T. Gunarathne), zhangbj@indiana.edu (B. Zhang), taklwu@indiana.edu (T.-L. Wu), xqiu@indiana.edu (J. Qiu).

0167-739X/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2012.05.027

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 the complexity of the cloud services. Window Azure claims to allow users to "focus on your applications, not the infrastructure". Twister4Azure takes that claim one step further and lets users focus only on the application logic without worrying about the application architecture.

Applications of Twister4Azure can be categorized according to three classes of application patterns. The first of these are the Map only applications, which are also called pleasingly (or embarrassingly) parallel applications. Examples 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 4.5 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 the SmithWaterman-GOTOH (SWG) [2] pairwise sequence alignment and Word Count as traditional MapReduce type applications. Section 4.4 analyzes the SWG Twister4Azure application.

The third and most important type of applications Twister4-Azure 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. Sections 4.2 and 4.3 present detailed analyses of Multi-Dimensional Scaling and KMeans Clustering 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. The MRRoles4Azure [4] MapReduce framework for Azure cloud was developed based on the success of pleasingly parallel cloud frameworks and was released in late 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 (http://salsahpc.indiana.edu/twister4azure/) was made available in mid-2011.

2. Background

2.1. MapReduce

The MapReduce [5] 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 [6] MapReduce is a widely used open-source implementation of the Google MapReduce [5] distributed data processing framework. Apache Hadoop MapReduce uses the Hadoop distributed file system (HDFS) [7] 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 MapReduce 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 MapReduce 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 [8] 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 times". 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 saving 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 [9]. 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 [10] is used for the current experiments. Twister performs optimized broadcasting operations by using the chain method [11] and uses the minimum spanning tree method [12] 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 [7].

2.4. Microsoft Azure platform

The Microsoft Azure platform [13] 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 [13].

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

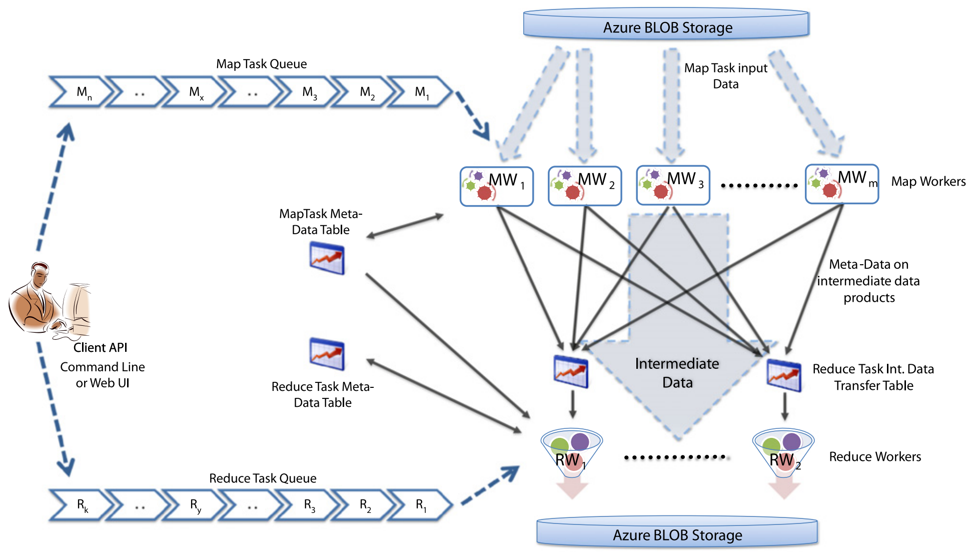


Fig. 1. MRRoles4Azure architecture [4].

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 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 (records or rows) that can be up to 1 MB. Entities contain properties (cells), that can be up to 64 KB. 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

MRRoles4Azure [4] is a distributed decentralized MapReduce runtime for the 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 Fig. 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 a natural load balancing, given a 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 descriptions of 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 [14] shown in Fig. 2 inspired the application use cases analyzed in this paper. This pipeline uses the SmithWaterman-GOTOH application, analyzed in Section 4.4, or BLAST+ application, analyzed in Section 4.5, 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.

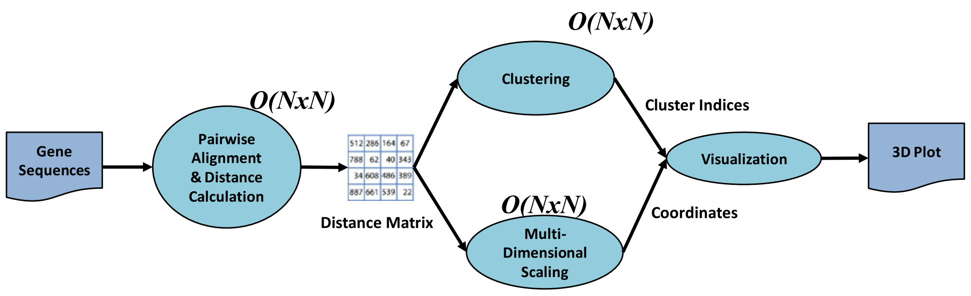


Fig. 2. Bio sequence analysis pipeline.

3. Twister4Azure-Iterative MapReduce

Twister4Azure is an iterative MapReduce framework for the 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 by utilizing the scalable, distributed and highly available Azure cloud services as the underlying building blocks. Twister4Azure employs 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) The ability to express a sufficiently large and useful subset of large-scale data-intensive and parallel computations,
(2) That it should be simple, easy-to-use and familiar to the users,
(3) That it should be 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 could be executed in the cloud environments efficiently, while using iterative MapReduce as the programming model.

3.1.1. *Data-intensive iterative computations*

There exist 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 Fig. 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 use 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 4.3). 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.

Twister4Azure extends the MapReduce programming model to support the easy parallelization of the iterative computations by adding a Merge step to the MapReduce model, and also 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 (lines 5–7 in Code 1, lines 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, lines 13–15 in Code 2) will perform any necessary computations, combine the partial results received and emit parts of the total Reduce output. A single *merge* task (lines 16–19 in Code 2) will merge the results emitted by the *reduce* tasks, and will evaluate the loop conditional function (lines 8 and 4 in Code 1), 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 Code 2). Fig. 4 presents the flow of the Twister4Azure programming model.

Code 1 Typical data-intensive iterative computation
```
1:  $k \leftarrow 0$;
2:  MAX $\leftarrow$ maximum iterations
3:  $\delta^{[0]} \leftarrow$ initial delta value
4:  **while** ($k <$ MAX_ITER$\|f(\delta^{[k]}, \delta^{[k-1]})$ )
5:      **foreach** datum in data
6:          $\beta$[datum] $\leftarrow$ process (datum, $\delta^{[k]}$)
7:      **end foreach**
8:      $\delta^{[k+1]} \leftarrow$ combine($\beta$[])
9:      $k \leftarrow k + 1$
10:   **end while**
```

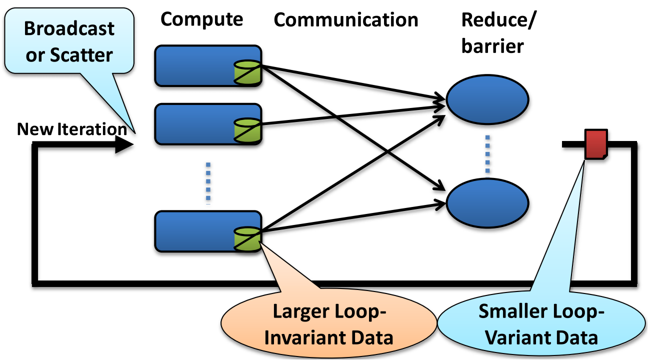


Fig. 3. Structure of a typical data-intensive iterative application.

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 [12], and avoiding the often unnecessary MapReduce jobs to perform Reduce side joins. The PageRank computation presented by Bu et al. [15] demonstrates the inefficiencies of using Map side joins for iterative computations. The Twister4Azure non-iterative computations can also use this extra input to receive broadcasts or scatter data to the Map and Reduce tasks.

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

Code 2 Data-intensive iterative computation using Twister4Azure programming model

```
1:   $k \leftarrow 0$;
2:   MAX $\leftarrow$ maximum iterations
3:   $\delta^{[0]} \leftarrow$ initial delta value
4:   $\alpha \leftarrow$ true

5:   **while** ( $k <$ MAX_ITER$\|\alpha$)
6:     distribute datablocks
7:     broadcast $\delta^{[k]}$
8:     **map** (datablock, $\delta^{[k]}$)
9:         **foreach** datum in datablock
10:            $\beta$[datum] $\leftarrow$ process (datum, $\delta^{[k]}$)
11:        **end foreach**
12:        emit ($\beta$)

13:    **reduce** (list of $\beta$)
14:        $\beta' \leftarrow$ combine (list of $\beta$)
15:        emit ($\beta'$)
16:    **merge**(list of $\beta'$, $\delta^{[k]}$)
17:        $\delta^{[k+1]} \leftarrow$ combine (list of $\beta$)
18:        $\alpha \leftarrow f(\delta^{[k]}, \delta^{[k-1]})$
19:        emit ($\alpha$, $\delta^{[k+1]}$)

20:    $k \leftarrow k + 1$
21:  **end while**
```

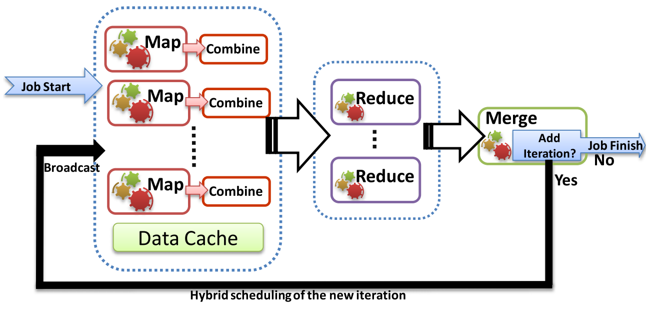


Fig. 4. Twister4Azure iterative MapReduce programming model.

3.1.3. *Merge*

Twister4Azure introduces *Merge* as a new step to the MapReduce programming model to support iterative MapReduce computations. The Merge task executes after the *Reduce* step. The *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 like the following sequence:

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

Since Twister4Azure does not have a centralized driver to make 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 KMeans Clustering. 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 the 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 provides significant speedups for the data-intensive iterative MapReduce applications. These speedups are even more significant in cloud environments, as the caching and reusing of data helps to overcome the bandwidth and latency limitations of cloud data storage.

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. The local disk cache automatically serves all the requests for previously downloaded data products. Currently, Twister4Azure does not support the eviction of the disk cached data products, and it assumes that the input data blobs do not change during the course of a computation.

The 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 the 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.

Code 3 Cache aware hybrid decentralized scheduling algorithm. (Executed by all the map workers)

```
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
```

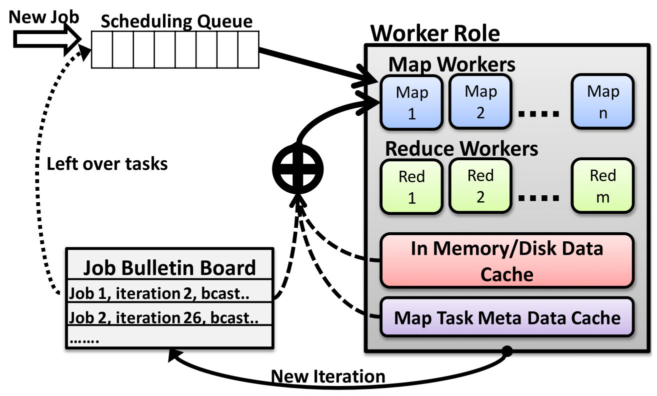


Fig. 5. Cache-aware hybrid scheduling.

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 an 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. The decentralized architecture of Twister4Azure presents a challenge in this situation, as Twister4Azure does not have either a central entity that has a global view of the data products cached in the worker-roles, nor does it have 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 on a central bulletin board (an Azure table). Naive 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, after the previous iteration is completed. A 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 Fig. 5, Twister4Azure schedules new MapReduce jobs (non-iterative and the first iteration of the iterative) through Azure queues. The Twister4Azure hybrid cache-aware scheduling algorithm is currently configured to give priority for the iterations of the already executing iterative MapReduce computations over new computations, to get the maximum value out of the cached data.

Any tasks for an iteration that were not scheduled in the above manner will be added back in to the task-scheduling queue and will be executed by the first available free worker ensuring the completion of that iteration. This ensures the eventual completion of the job and the fault tolerance of the tasks in the event of a worker failure; it also ensures the dynamic scalability of the system when new workers are added to the virtual cluster. Duplicate task execution can happen on 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 Queues. However, queue latency is not a significant bottleneck for Twister4Azure iterative computations as only the scheduling of the first iteration depends on Azure queues.

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 computations. 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; and (2) using a combination of direct TCP and Azure blob storage. The 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 numbers of instances and does not scale well for large numbers of instances.

The second method implements a tree-based broadcasting algorithm that uses the Windows Communication Foundation (WCF) based Azure TCP inter-role communication mechanism for the data communication, as shown in Fig. 6. This method supports a configurable number of parallel outgoing TCP transfers per instance (three parallel transfers in Fig. 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

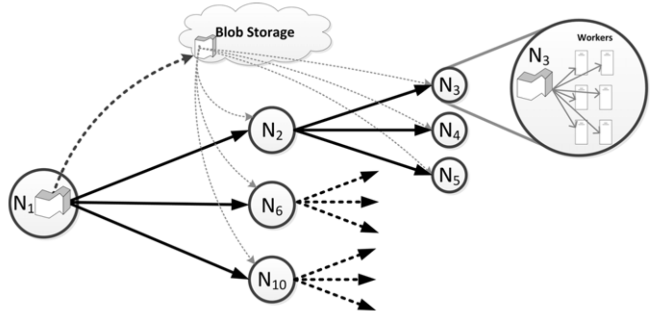


Fig. 6. Tree based broadcast over TCP with Blob storage as the persistent backup. N_3 shows the utilization of data cache to share the broadcast data within an instance.

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 the 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 exists more than one *Map*/*Reduce*/*Merge* worker per worker-role, and also, 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.

3.5. Intermediate data communication

Twister4Azure supports two types of intermediate data communication. The first is the legacy Azure Blob storage based transfer model of the MRRoles4Azure, where the Azure Blob storage is used to store the intermediate data products and the Azure tables are used to store the meta-data about the intermediate data products. The data is always persistently stored in the Blob storage before it declares the Map task a success. Reducers can fetch data any time from the Blob storage even in the cases where there are multiple waves of Reduce tasks or any re-execution of Reduce tasks due to failures. This mechanism performed well for non-iterative applications. Based on our experience, the tasks in the iterative MapReduce computations are of a relatively finer granularity, 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 optimized this method by utilizing the Azure tables to transfer smaller data products up to a certain size (currently 64 KB that is the limit for a single item in an Azure table entry) and so we could use the blob storage for the data products that are larger than that limit.

The second method is a TCP-based streaming mechanism where the data products are pushed directly from the Mapper memory to the Reducers similar to the MapReduce Online [16] approach, rather than Reducers fetching the data products, as is the case in traditional MapReduce frameworks such as Apache Hadoop. This mechanism performs a *best effort* transfer of intermediate data products to the available Reduce tasks using the Windows Communications Foundation (WCF) based Azure direct TCP communication. A separate thread performs this TCP data transfer, freeing up the Map worker thread to start processing a new Map task. With this mechanism, when the Reduce task input data size is manageable, Twister4Azure can perform the computation completely in the memory of Map and Reduce workers without any intermediate data products touching the disks offering significant performance benefits to the computations. These intermediate data products are uploaded to the persistent Blob store in the background as well. Twister4Azure declares a Map task a success only after all the data is uploaded to the Blob store. Reduce tasks will fetch the persistent intermediate data from the Blob store if a Reduce task does not receive the data product via the TCP transfer. These reasons for not receiving data products via TCP transfer include I/O failures in the TCP transfers, the Reduce task not being in an execution or ready state while the Map worker is attempting the transfer, or the rare case of having multiple Reduce task waves. Twister4Azure users the intermediate data from the Blob store when a Reduce task is re-executed due to failures as well. Users of Twister4Azure have the ability to disable the above-mentioned data persistence in Blob storage and to rely solely on the streaming direct TCP transfers to optimize the performance and data-storage costs. This is possible when there exists only one wave of Reduce tasks per computation, and it comes with the risk of a coarser grained fault-tolerance in the case of failures. In this scenario, Twister4Azure falls back to providing an iteration level fault tolerance for the computations, where the current iteration will be rolled back and re-executed in case of any task failures.

3.6. Fault tolerance

Twister4Azure supports typical MapReduce fault tolerance through re-execution of failed tasks, ensuring the eventual completion of the iterative computations. Twister4Azure stores all the intermediate output data products from Map/Reduce tasks, as well as the intermediate output data products of the iterations persistently in the Azure Blob storage or in Azure tables, enabling fault-tolerant in task level as well as in iteration level. The only exception to this is when a direct TCP only intermediate data transfer (Section 3.5) is used, in which case Twister4Azure performs fault-tolerance through the re-execution of iterations.

Recent improvements to the Azure queues service include the ability to update the queue messages, the ability to dynamically extend the visibility time outs and to provide support for much longer visibility timeouts of up to 7 days. We are currently working on improving the queue based fault tolerance of Twister4Azure by utilizing these newly introduced features of the Azure queues that allows us to support much more finer grained monitoring and fault tolerance, as opposed to the current time out based fault tolerance implementation.

3.7. Other features

Twister4Azure 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 computation. This also enables Twister4Azure to support workflow scenarios without redeployment. Twiser4Azure also supports the capacity to have multiple MapReduce jobs inside a single iteration of an iterative MapReduce computation, enabling the users to more easily specify computations that are complex, and to share cached data between these individual computations. The Multi-Dimensional Scaling iterative MapReduce application described in Section 4.2 uses this feature to perform multiple computations inside an iteration.

Twister4Azure also provides users with a web-based monitoring console from which they can monitor the progress of their jobs as well as any error traces. 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 as well. Users can develop, test and debug the Twister4Azure MapReduce programs in the comfort of using their local machines with the use of the Azure local development fabric. Twister4Azure programs can be deployed directly from the Visual Studio development environment or through the Azure web interface, similar to any other Azure WorkerRole project.

4. Twister4Azure scientific application case studies

4.1. Methodology

In this section, we present and analyze four real-world data-intensive scientific applications that were implemented using Twister4Azure. Two of these applications, Multi-Dimensional Scaling and KMeans Clustering, are iterative MapReduce applications, while the other two applications, sequence alignment and sequence search, are pleasingly parallel MapReduce applications.

We compare the performance of the Twister4Azure implementations of these applications with the Twister [8] and Hadoop [6] implementations of these applications, where applicable. The Twister4Azure applications were implemented using C#.Net, while the Twister and Hadoop applications were implemented using Java. We performed the Twister4Azure performance experiments in the Windows Azure Cloud using the Azure instance types mentioned in Table 1. We performed the Twister and Hadoop experiments in the local clusters mentioned in Table 2. Azure cloud instances are virtual machines running on shared hardware nodes with the network shared with other users; the local clusters were dedicated bare metal nodes with dedicated networks (each local cluster had a dedicated switch and network not shared with other users during our experiments). Twister had all the input data pre-distributed to the compute nodes with 100% data locality, while Hadoop used HDFS [7] to store the input data, achieving more than 90% data locality in each of the experiments. Twister4Azure input data were stored in the high-latency off-the-instance Azure Blob Storage. A much better raw performance is expected from the Hadoop and Twister experiments on local clusters than from the Twister4Azure experiments using the Azure instances, due to the above stated differences. Our objective is to highlight the scalability comparison across these frameworks and demonstrate that Twister4Azure has less overheads and scales as well as Twister and Hadoop, even when executed on an environment with the above overheads and complexities.

Equal numbers of compute cores from the local cluster and from the Azure Cloud were used in each experiment, even though the raw compute powers of the cores differed. For example, the performance of a Twister4Azure application on 128 Azure small instances was compared with the performance of a Twister application on 16 HighMem (Table 2) cluster nodes.

We use the custom defined metric "adjusted performance" to compare the performance of an application running on two different environments. The objective of this metric is to negate the performance differences introduced by some of the underlying hardware differences. The *Twister4Azure adjusted* (t_a) line in some of the graphs depicts the performance of Twister4Azure for a certain application normalized according to the sequential performance difference for that application between the Azure (t_{sa}) instances and the nodes in the Cluster (t_{sc}) environment used for Twister and Hadoop. We estimate the Twister4Azure "adjusted performance" for an application using $t_a \times (t_{sc}/t_{sa})$, where t_{sc} is the sequence performance of that application on a local cluster node, and t_{sa} is the sequence performance of that application on a given Azure instance when the input data is present locally. This estimation, however, does not account for the framework overheads that remain constant irrespective of the computation time, the network difference or the data locality differences.

4.2. Multi-dimensional scaling—iterative MapReduce

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 onto 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.

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 by CPU power. The main objective of parallelizing MDS is to leverage the distributed memory to support the 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 Fig. 7. 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.

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). The Twister4Azure tests were performed on Azure Large instances using the Memory-Mapped file based (Section 5.3) data caching. Java HPC Twister results do not include the initial data distribution time. Fig. 8(a) presents the execution time for weak scaling, where we increase the number of compute resources while keeping the work per core constant (work $\propto$ number of cores). We notice that Twister4Azure exhibits encouraging performance and scales similar to the Java HPC Twister. Fig. 8(b) shows that the MDS performance scales well with increasing data sizes.

The HighMem cluster is a bare metal cluster with a dedicated network, very large memory and with faster processors. It is expected to be significantly faster than the cloud environment for the same number of CPU cores. The *Twister4Azure adjusted* (t_a) lines in Fig. 8 depicts the performance of the Twister4Azure normalized according to the sequential performance difference of the MDS BC calculation, and the Stress calculation between the Azure instances and the nodes in the HighMem cluster. In the above testing, the total number of tasks per job ranged from 10,240 to 40,960, proving Twister4Azure's ability to support large numbers of tasks effectively.

Fig. 9(a) depicts the execution time of individual Map tasks for 10 iterations of Multi-Dimensional Scaling of 204,800 data points on 32 Azure Large instances. The higher execution time of the tasks in the first iteration is due to the overhead of initial data downloading, parsing and loading. This overhead is overcome in the subsequent iterations through the use of data caching, enabling Twister4Azure to provide large performance gains relative to a non-data-cached implementation. The performance gain achieved by data caching for this specific computation can be estimated as more than 150% per iteration, as a non-data cached implementation would perform two data downloads (one download per application) per iteration. Fig. 9(b) presents the number of Map tasks executing at a given moment for 10 iterations for the above MDS computation. The gaps between the bars represent the overheads of our framework. The gaps between the iterations (gaps between a grey MDSStressCalc bar and a subsequent black MDSBCCalc bar) are small, which depicts that the between-iteration overheads that include Map to Reduce data transfer time, Reduce and Merge task execution time, data broadcasting cost and new iteration scheduling cost, are relatively smaller for MDS. Gaps between applications (gaps between a black MDSBCCalc bar and a subsequent grey MDSStressCalc bar) of an iteration are almost non-noticeable in this computation.

Table 2
Evaluation cluster configurations.

Cluster/instance type	CPU cores (GHz)	Memory (GB)	I/O performance	Compute resource	OS
Azure small	1 × 1.6	1.75	100 MBPS, shared network infrastructure	Virtual instances on shared hardware	Windows server
Azure large	4 × 1.6	7	400 MBPS, shared network infrastructure	Virtual instances on shared hardware	Windows server
Azure extra large	8 × 1.6	14	800 MBPS, shared network infrastructure	Virtual instances on shared hardware	Windows server
HighMem	8 × 2.4 (Intel® Xeon® CPU E5620)	192	Gigabit ethernet, dedicated switch	Dedicated bare metal hardware	Linux
iDataPlex	8 × 2.33 (Intel® Xeon® CPU E5410)	16	Gigabit ethernet, dedicated switch	Dedicated bare metal hardware	Linux

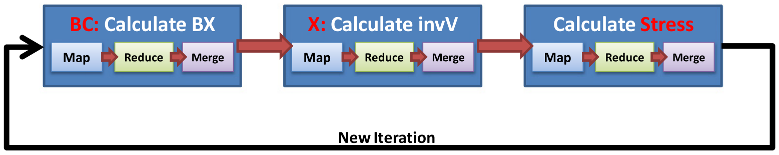


Fig. 7. Twister4Azure multi-dimensional scaling.

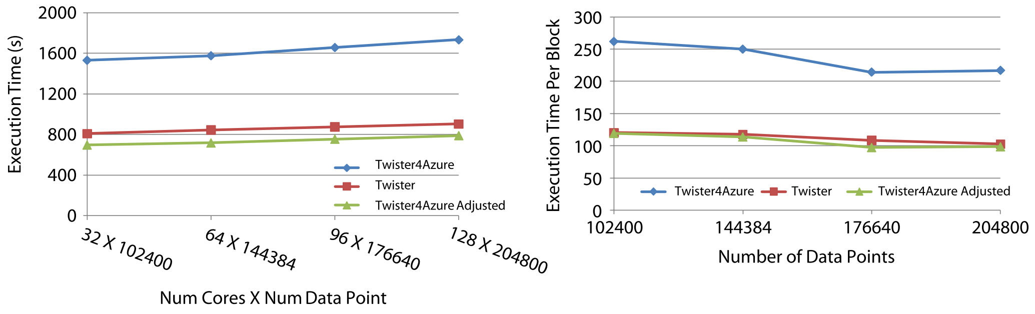


Fig. 8. *Left* (a) MDS weak scaling where the workload per core is constant. Ideal is a straight horizontal line. *Right* (b) data size scaling using 128 Azure small instances/cores, 20 iterations.

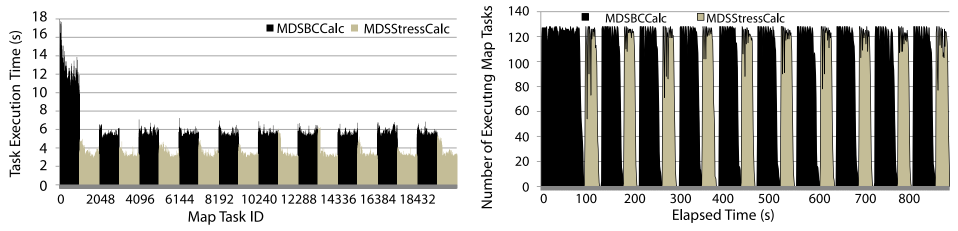


Fig. 9. Twister4Azure Map Task histograms for MDS of 204,800 data points on 32 Azure Large Instances (graphed only 10 iterations out of 20). *Left* (a) Map Task execution time histogram. Two adjoining bars (taller black MDSBCCalc bar and shorter grey MDSStressCalc bar) represent an iteration (2048 tasks per iteration), where each bar represents the different applications inside the iteration. *Right* (b) number of executing Map Tasks in the cluster at a given moment. Two adjoining bars (black and grey) represent an iteration.

4.3. KMeans clustering—iterative MapReduce

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 KMeans 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 KMeans Clustering and millions of centroids.

KMeans 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, the *assignment step* is performed in the Map task and the *update step* is performed in the Reduce task. Centroid data is broadcast 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.

Fig. 11(a) depicts the execution time of Map tasks 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. Fig. 11(b) presents the number of Map tasks executing at a given moment 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 the 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 KMeans Clustering application.

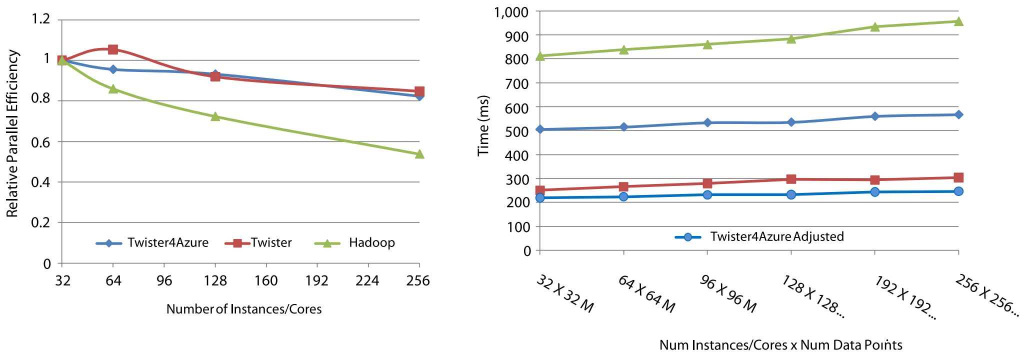


Fig. 10. KMeans Clustering scalability. *Left* (a) Relative parallel efficiency of strong scaling using 128 million data points. *Right* (b) Weak scaling. Workload per core is kept constant (ideal is a straight horizontal line).

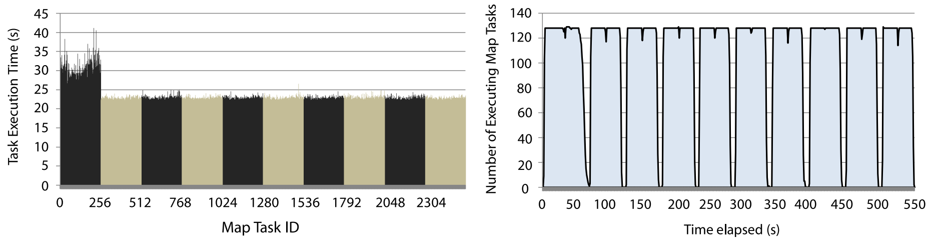


Fig. 11. Twister4Azure Map task histograms for KMeans Clustering 128 million data points on 128 Azure Small instances. 10 iterations with 256 Map tasks per iteration. *Left* (a) Map task execution time histogram. *Right* (b) Number of executing Map tasks in the cluster at a given moment.

We compared the Twister4Azure KMeans Clustering performance with implementations of the Java HPC Twister and Hadoop. The Java HPC Twister and Hadoop experiments were performed in a dedicated iDataPlex cluster (Table 2). The Twister4Azure tests were performed using the Azure Small instances that contain a single CPU core. The Java HPC Twister results do not include the initial data distribution time. Fig. 10(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. Fig. 10(b) presents the execution time for weak scaling, wherein we increase the number of compute resources while keeping the work per core constant (work $\propto$ number of nodes). We notice that Twister4Azure performance scales well up to 256 instances in both experiments. In 10(a), the relative parallel efficiency of Java HPC Twister for 64 cores is greater than one. We believe the memory load was a bottleneck in the 32 core experiment, whereas this is not the case for the 64 core experiment. We used a 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 hypothesize 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 the same, irrespective of the instance type.

The Twister4Azure adjusted line in Fig. 10(b) depicts the KMeans Clustering performance of Twister4Azure normalized according to the ratio of the sequential performance difference between the Azure instances and the iDataPlex cluster nodes. All tests were performed using 20-dimensional data and 500 centroids.

4.4. Sequence alignment using SmithWaterman GOTOH—MapReduce

The SmithWaterman [21] algorithm with GOTOH [22] (SWG) improvement is used to perform a pairwise sequence alignment on two FASTA sequences. We used the SWG application kernel in parallel to calculate the all-pairs dissimilarity of a set of n sequences, resulting in an $n * n$ distance matrix. A set of Map tasks for a particular job are generated using the blocked decomposition of a 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. Ekanayake et al. [2] presents more details about the Hadoop-SWG application implementation. The Twister4Azure SWG implementation also follows the same architecture and blocking strategy as in the Hadoop-SWG implementation. Twister4Azure SWG uses NAligner [23] as the computational kernel.

We performed the SWG weak scaling test from Gunarathne et al. [4] using Twister4Azure to compare the performance of the Twister4Azure SWG implementation on Azure Small instances (Table 2) with a Apache Hadoop implementation on the iDataPlex cluster (Table 2). Fig. 12 shows that the Twister4Azure SWG performs comparably to the Apache Hadoop SWG. The performance of the Twister4Azure SWG fell between +/−2% of the MRRoles4Azure SWG performance [4], confirming that the extra complexity of Twister4Azure has not adversely affected the non-iterative MapReduce performance of Twister4Azure.

4.5. Sequence searching using BLAST

NCBI BLAST+ [1] is the latest version of the 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 that of the Apache Hadoop BLAST+ implementation. We used Azure Extra Large instances (Table 2) with 8 Map workers per node for the Twister4Azure BLAST

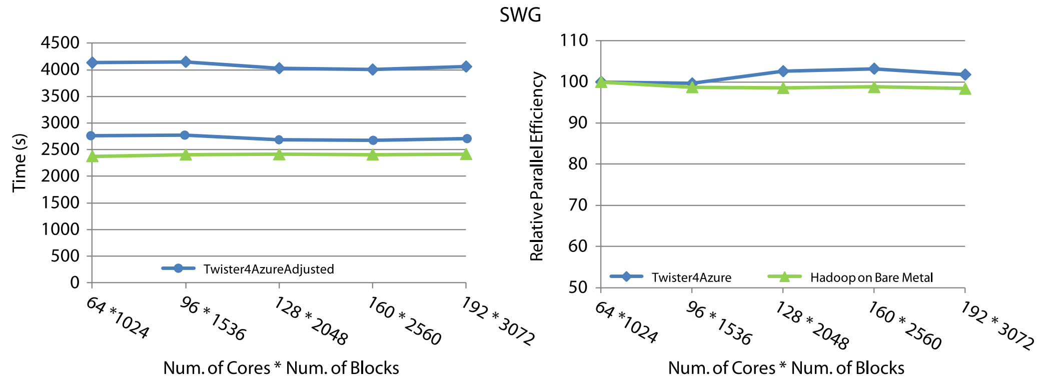


Fig. 12. Twister4Azure SWG performance. *Left* (a) Raw and adjusted execution times. *Right* (b) Parallel efficiency relative to the 64* 1024 test case.

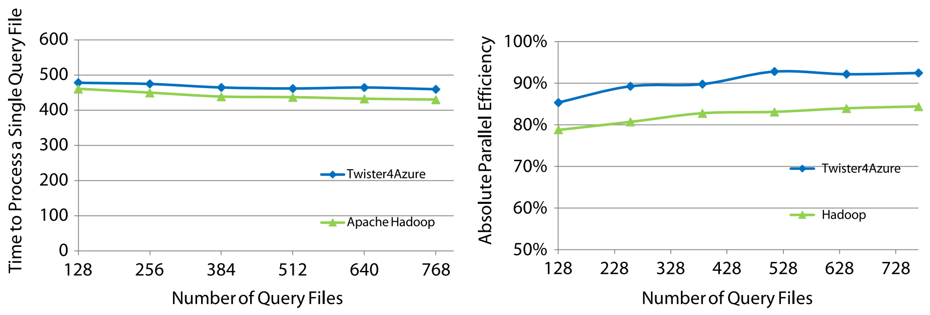


Fig. 13. Twister4Azure BLAST performance. *Left* (a) Time to process a single query file. *Right* (b) Absolute parallel efficiency.

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. Both the implementations downloaded and extracted the compressed BLAST database to the local disk of each worker prior to processing of the tasks. Twister4Azure's ability to specify deployment-time initialization routines was used to download and extract the database. The performance results do not include the database distribution times.

The Twister4Azure BLAST+ absolute efficiency (Fig. 13) was better than the Hadoop implementation. 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 data caching on Azure

In this section, we present a performance analysis of several data caching strategies that affect the performance of large-scale parallel iterative MapReduce applications on Azure, in the context of a Multi-Dimensional Scaling application presented in Section 4.2. These applications typically perform tens to hundreds of iterations. Hence, we focus mainly on optimizing the performance of the majority of iterations, while assigning a lower priority to optimizing the initial iteration.

In this section, we use a dimension-reduction computation of a 204800 * 204800 element input matrix, partitioned into 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 3 presents the execution time analysis of this computation under different mechanisms. The "Task Time" in Table 3 refers to the end-to-end execution time of the 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 the "Task Time" introduced by the data download in the first iterations, we calculated the averages and standard deviations excluding the first iteration. The "# 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 based data caching

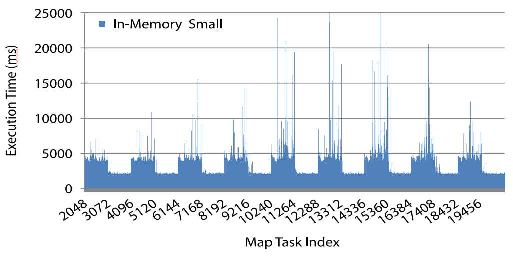
As discussed in Section 3.2, it is possible to optimize iterative MapReduce computations by caching the loop-invariant input data across the iterations. We use the Azure Blob storage as the input data storage for the Twister4Azure computations. Twister4Azure supports local instance (disk) storage caching 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 disk based 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.

5.2. In-memory data caching

Twister4Azure also supports the "in-memory caching" of the loop-invariant data across iterations. With in-memory caching, Twister4Azure fetches the data from the Azure Blob storage, parses and loads them into 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

Table 3
Execution time analysis of a MDS computation. 204800 * 204800 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	6390	750	40	3662	131	0
In-memory cache	Small*1	2072	4052	895	140	3924	877	143
	Large*4	2574	4354	5706	1025	4039	5710	1071
Memory-mapped file (MMF) cache	Small*1	2097	4852	486	28	4725	469	29
	Large*4	1876	5052	371	6	4928	357	4



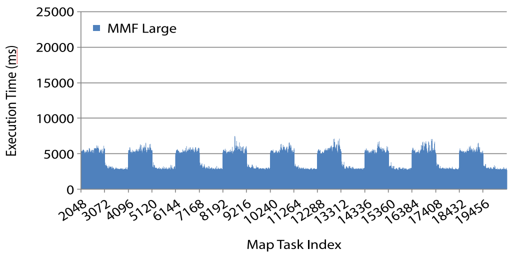


Fig. 14. Execution traces of MDS iterative MapReduce computations using Twister4Azure showing the execution time of tasks in each iteration. The taller bars represent the MDSBCCalc computation, while the shorter bars represent the MDSStressCalc computation. A pair of BCCalc and StressCalc bars represents an iteration. *Left* (a) Using in-memory caching on small instances. *Right* (b) Using Memory-Mapped file based caching on Large instances.

subsequent iterations, eliminating the overhead of reading and parsing data from the disk. As shown in Table 3, this 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 a 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 the Twister4Azure caching policy will evict the broadcast data for iterations from the data cache after the particular iterations.

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

5.2.1. *Non-deterministic performance anomalies with in-memory data caching*

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 3). 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. Fig. 14(a) offers an example of an execution trace of a computation that shows this performance fluctuation where we can notice occasional unusual high task execution times. Even though Twister4Azure supports the duplicate execution of the slow tasks, duplicate tasks for non-initial iterations are often more costly 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. With further experimentation, we were able to narrow down the cause of this anomaly to the use of a 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 offer a better performance, 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 3). However, the "Task Time" that includes the disk reading time is unstable when a local-storage cache is used (40 slow "Task Time" tasks in row 1 of Table 3).

5.3. *Memory-mapped file based data cache*

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 fetches 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 (rows 4 and 5 of Table 3).

With the Twister4Azure in-memory cache implementation, the performance on larger Azure instances (with the number of workers equal to the number of cores) was very unstable (row 3 of Table 3). By contrast, when using memory-mapped caching, the execution times were more stable on the larger instances than for the smaller instances (rows 4 vs 5 in Table 3). The ability to utilize larger instances effectively is a significant advantage, as the usage of larger instances improves the data sharing across workers, facilitates better load balancing within the instances, provides better deployment stability, reduces the data-broadcasting load and simplifies the cluster monitoring.

The memory-mapped file based caching requires the data to be parsed (decoded) each time the data is used; this adds an overhead to the task execution times. In order to avoid a duplicate loading of data products to memory, we use real time data parsing in the case of the memory-mapped files. Hence, the parsing overhead becomes part of the Map function execution time. However, we found that the execution time stability advantage outweighs the added cost. In Table 3, 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. Fig. 14(b) presents an execution trace of a job that uses Memory Mapped file based caching.

6. Related work

CloudMapReduce [24] for Amazon Web Services (AWS) and Google AppEngine MapReduce [25] follow an architecture similar to that of MRRoles4Azure, as they utilize cloud services as their building blocks. Amazon ElasticMapReduce [26] offers Apache Hadoop as a hosted service on the Amazon AWS cloud environment. However, none of them supports iterative MapReduce. Spark [27] 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 the re-use of data products.

The 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. The Azure HPC scheduler supports parametric sweeps, Message Passing Interface (MPI) and LINQ to HPC applications together with a web-based job submission interface. AzureBlast [28] is an implementation of a parallel BLAST on an Azure environment that uses Azure cloud services with an architecture similar to the Classic Cloud model, which is a predecessor to Twister4Azure. CloudClustering [29] is a prototype KMeans Clustering implementation that uses Azure infrastructure services. CloudClustering uses multiple queues (single queue per worker) for job scheduling and supports caching of loop-invariant data.

Microsoft Daytona [11] is a recently announced iterative MapReduce runtime developed by Microsoft Research for the 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 the 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 data splitting for MapReduce computations and claims to support a variety of data broadcast patterns between the iterations. However, as opposed to Twister4Azure, Daytona uses a single master node based controller to drive and manage the computation. This centralized controller substitutes the "Merge" step of Twister4Azure, but makes Daytona prone to single point of failures. Daytona is available as a "Community Technology Preview" for academic and non-commercial usage.

Haloop [15] 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. The Twister4Azure additional input parameter for Map API eliminates the need for such Reduce side joins. The Reducer output data-cache is specially designed to support Fixpoint Evaluations using the output data from older iterations. Twister4Azure currently does not support this feature.

7. Conclusions and future work

We presented Twister4Azure, a novel iterative MapReduce distributed computing runtime for Windows Azure Cloud. Twiser4-Azure enables the users to perform large-scale data-intensive parallel computations efficiently on Windows Azure Cloud, by hiding the complexity of scalability and fault tolerance when using clouds. The key features of Twiser4Azure presented in this paper include the novel programming model for iterative MapReduce computations, the multi-level data caching mechanisms to overcome the latencies of cloud services, the decentralized cache-aware task scheduling utilized to avoid single point of failures and the framework managed fault tolerance drawn upon to ensure the eventual completion of the computations. We also presented optimized data broadcasting and intermediate data communication strategies that sped up the computations. Users can perform debugging and testing operations for the Twister4Azure computations in their local machines with the use of the Azure local development fabric. We also analyzed the performance anomalies of Azure instances with the use of in-memory caching; we then proposed a novel caching solution based on Memory-Mapped Files to overcome those performance anomalies.

We discussed four real-world data-intensive scientific applications which were implemented using Twister4Azure so as to show the applicability of Twister4Azure; we compared the performance of those applications with that of Java HPC Twister and Hadoop MapReduce frameworks. We presented Multi-Dimensional Scaling (MDS) and KMeans Clustering as iterative scientific applications of Twister4Azure. Experimental evaluation showed that MDS using Twister4Azure on a shared public cloud scaled similar to the Java HPC Twister MDS on a dedicated local cluster. Further, the KMeans Clustering using Twister4Azure with shared cloud virtual instances outperformed Apache Hadoop in a local cluster by a factor of 2–4, and also exhibited a performance comparable to that of Java HPC Twister running on a local cluster. These iterative MapReduce computations were performed on up to 256 cloud instances with up to 40,000 tasks per computation. We also presented sequence alignment and BLAST sequence searching pleasingly parallel MapReduce applications of Twister4Azure. These applications running on the Azure Cloud exhibited performance comparable to the Apache Hadoop on a dedicated local cluster.

Twister4Azure presents a viable architecture and a programming model for scalable parallel processing in the cloud environments: this architecture can also be implemented for other cloud environments as well. We are currently working on improving the Twister4Azure programming model by introducing novel iterative MapReduce collective communication primitives to increase the usability prospects, and also to further enhance the performance of Twister4Azure computations.

Acknowledgments

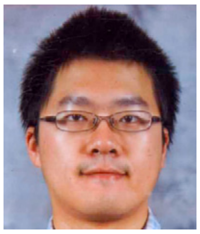
This work is funded in part by the Microsoft Azure Grant. We would also like to thank Geoffrey Fox and Seung-Hee Bae for many discussions. TG was supported by a fellowship sponsored by Persistent Systems.

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 (2009) 421.

[2] J. Ekanayake, T. Gunarathne, J. Qiu, Cloud technologies for bioinformatics applications, IEEE Transactions on Parallel and Distributed Systems 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 23 (2011) 2338–2354.
[4] T. Gunarathne, W. Tak-Lon, J. Qiu, G. Fox, MapReduce in the clouds for science, IEEE Second International Conference on Cloud Computing Technology and Science, CloudCom-2010, 2010, pp. 565–572.
[5] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (2008) 107–113.
[6] Apache Hadoop, Retrieved Mar. 20, 2012, http://hadoop.apache.org/core/.
[7] Hadoop Distributed File System HDFS, Retrieved Mar. 20, 2012, http://hadoop.apache.org/hdfs/.
[8] 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, Chicago, Illinois, 2010.
[9] Z. Bingjing, R. Yang, W. Tak-Lon, J. Qiu, A. Hughes, G. Fox, Applying twister to scientific applications, IEEE Second International Conference on Cloud Computing Technology and Science, CloudCom-2010, 2010, pp. 25–32.
[10] Apache ActiveMQ open source messaging system; Retrieved Mar. 20, 2012. http://activemq.apache.org/.
[11] Microsoft Daytona, Retrieved Mar. 20, 2012. http://research.microsoft.com/en-us/projects/daytona/.
[12] J. Lin, C. Dyer, Data-intensive text processing with MapReduce, Synthesis Lectures on Human Language Technologies 3 (2010) 1–177.
[13] Windows Azure Compute, Retrieved Mar. 20, 2012. http://www.microsoft.com/windowsazure/features/compute/.
[14] J. Qiu, T. Gunarathne, J. Ekanayake, J.Y. Choi, S.-H. Bae, H. Li, B. Zhang, Y. Ryan, S. Ekanayake, T.-L. Wu, S. Beason, A. Hughes, G. Fox, Hybrid Cloud and Cluster Computing Paradigms for Life Science Applications, 11th Annual Bioinformatics Open Source Conference, BOSC-2010, Boston, 2010.
[15] Y. Bu, B. Howe, M. Balazinska, M.D. Ernst, HaLoop: efficient iterative data processing on large clusters, The 36th International Conference on Very Large Data Bases, VLDB Endowment, Singapore, 2010.
[16] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, K. Elmeleegy, R. Sears, MapReduce online, In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, San Jose, California, 2010, p. 21–21.
[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, IEEE Transactions on Information Theory 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] JAligner., Retrieved Mar. 20, 2012. http://jaligner.sourceforge.net.
[24] L. Huan, D. Orban, Cloud MapReduce: a MapReduce implementation on top of a cloud operating system, 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011, 2011, pp. 464–474, (http://dx.doi.org/10.1109/CCGrid.2011.25).
[25] AppEngine MapReduce, Retrieved Mar. 20, 2012. http://code.google.com/p/appengine-mapreduce.
[26] Amazon Web Services, Retrieved Mar. 20, 2012. http://aws.amazon.com/.
[27] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: Cluster Computing with working sets, 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud '10, Boston, 2010.
[28] W. Lu, J. Jackson, 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, ACM, Chicago, Illinois, 2010, pp. 413–420.
[29] A. Dave, W. Lu, J. Jackson, R. Barga, CloudClustering: toward an iterative data processing pattern on the cloud, First International Workshop on Data Intensive Computing in the Clouds, Anchorage, Alaska, 2011.

Thilina Gunarathne is a Ph.D. candidate at the School of Informatics and Computing at Indiana University. Thilina has engaged in research in the fields of distributed & parallel computing, cloud computing, many/multicore systems and SOA. His current research focuses on exploring architectures and programming models for scalable parallel computing on cloud environments. He has contributed to several open source projects in Apache Software Foundation as a committer and a PMC member starting from 2004. He received his B.Sc. (Computer Science and Engineering) from the University of Moratuwa, Sri Lanka in 2006 and M.Sc. (Computer Science) from the Indiana University in 2009.



Bingjing Zhang is a Ph.D. candidate at the School of Informatics and Computing at Indiana University. His current research focuses on design and optimization of iterative MapReduce framework. He is one of main contributors to Twister iterative MapReduce framework. He received his M.Sc. (Computer Science) from Indiana University in 2011, M.E. (Software Engineering) from Nanjing University in 2009, and B.E. (Software Engineering) from Nanjing University in 2007.

Tak-Lon (Stephen) Wu is a graduate student pursuing Ph.D. degree of School of Informatics and Computing at Indiana University, Bloomington. Besides attending the school, he is working as a Graduate Research Assistant of SalsaHPC Team, Community Grids Lab at Indiana University Bloomington. In addition, he has been an associate instructor of Dr. Judy Qiu courses, Distributed System and Cloud Computing, for four semesters. He received his B.Sc. (Computer Science and Information Engineering) from National Central University, Taiwan in 2008 and M.Sc. (Computer Science) from the Indiana University in 2010.

Dr. Judy Qiu is an Assistant Professor of Computer Science at Indiana University. Her areas of study include parallel and distributed systems, Cloud/Grid computing and high performance computing. She has extensive research experience in multicore computing and the use of cloud platforms and systems for scientific data analysis. Her research thrust is Data-Enabled Discovery Environments for Science and Engineering with novel technologies driven by applications. She has pioneered the use of Iterative MapReduce on both HPC and commercial cloud environments. She collaborates in several applications to motivate and validate her computer science systems work. Current work is in Genomics, Proteomics and Network Science. Her work has been funded by NSF, NIH, Microsoft, and Indiana University Faculty Research Support Program.