SCALABLE PARALLEL COMPUTING ON CLOUDS

Thilina Gunarathne

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science of School of Informatics and Computing.

Indiana University

Month 2013

[Acceptance page]

[Copyright (optional)]

[Dedication]

[Acknowledgements]

Abstract (draft)

During the last decade three largely industry-driven disruptive trends have altered the landscape of scalable parallel computing, which has long been dominated by the HPC applications. These disruptions are, 1. Data deluge, shift to more data intensive from compute intensive. (aka big data), 2. MapReduce based compute and storage frameworks, 3. Utility computing model introduced by Cloud computing environments. The work of this thesis focuses on the intersection of these three disruptions and evaluates the feasibility of Cloud Computing environments to perform large scale data intensive computations using new generation programming and execution frameworks.

The current main issues and challenges for performing scalable parallel computing on cloud environments include identifying application patterns that are well-suited for cloud environments, identifying efficient and easy-to-use programing abstractions & primitives to represent those applications patterns, performing appropriate task partitioning & task scheduling, identifying suitable data storage & staging architectures, identifying suitable communication patterns for intermediate data communication & data broadcasting and exploring appropriate fault tolerance mechanisms.

We identify three types of applications patterns that are well suited for Cloud environments. First type is the pleasingly parallel computations and we present pleasingly parallel programming frameworks for cloud environment to support this kind of computations. Second type is the MapReduce type applications. We present a decentralized architecture and an implementation to develop MapReduce frameworks using cloud infrastructure services. Third type is the data intensive iterative applications, which can be implemented as iterative computation and communication steps, which encompasses applications such as many graph processing algorithms, many machine learning algorithms and many more. We present Twister4Azure architecture and runtime as a solution to implement this kind of applications on cloud environments. Twister4Azure architecture extends the familiar, easy-to-use

MapReduce programming model with iterative extensions and iterative specific optimizations, enabling a wide array of large scale iterative as well as non-iterative data analysis and scientific applications to utilize Cloud platforms easily and efficiently in a fault-tolerant manner.

Collective communication operations facilitate the optimized communication and coordination between groups of nodes of a distributed computations leading to many advantages. We present the applicability of collective communication operations to enrich the Iterative MapReduce with additional application pattern without sacrificing the desirable properties of MapReduce model. Addition of collective communication operations enriches the iterative MapReduce model by providing many performance and ease of use advantages such as providing efficient data communication operations optimized for the particular execution environment and use case, providing programming models that fit naturally with application patterns and allowing users to avoid overheads by skipping unnecessary steps of the execution flow.

Table of Contents

1. Introduction 17

1.1 Statement of research problem 17

1.2 Research Challenges 18

1.3 Thesis contributions 22

1.4 Thesis outline 23

2. Background 26

2.1 Cloud Environments 26

2.1.1 Microsoft Azure platform 26

2.1.2 Amazon AWS 27

2.2 Application types 30

2.2.1 Pleasingly Parallel Applications 30

2.2.2 MapReduce Type Applications 31

2.2.3 Data Intensive Iterative Applications 31

2.2.4 MPI type applications 32

2.3 Applications 32

2.3.1 Cap3 32

2.3.2 Generative Topographic Mapping Interpolation 33

2.3.3 Blast+ sequence search 34

2.3.4 Sequence alignment using SmithWaterman GOTOH (SWG) 34

2.3.5 KMeansClustering 35

2.3.6 Multi-Dimensional Scaling (MDS) 36

2.3.7 Bio sequence analysis pipeline 37

3. Related Works 38

3.1.1 Apache Hadoop 39

3.1.2 Twister 40

3.1.3 Microsoft Daytona 41

3.1.4 Haloop 41

3.1.5 Microsoft Dryad 42

3.1.6 Spark 43

3.1.7 i-MapReduce 43

3.1.8 PrItr 43

3.1.9 Apache Giraph 43

3.1.10 Dremel/Pregel 43

3.2 Summary 44

4. Pleasingly parallel computing on cloud environments 46

4.1 Pleasingly parallel application architecture 46

4.1.1 Classic Cloud processing model 47

4.1.2 Pleasingly parallel processing using MapReduce frameworks 48

4.1.3 Usability of the technologies 50

4.2 Evaluation Methodology 51

4.3 Cap3 54
4.3.1 Performance with different EC2 cloud instance types 54
4.3.2 Scalability study 55
4.4 BLAST 58
4.4.1 Performance with different cloud instance types 58
4.4.2 Scalability 61
4.5 GTM Interpolation 62
4.5.1 Application performance with different cloud instance types 63
4.5.2 GTM Interpolation speedup 63
4.6 Summary 65
5. MapReduce type applications on cloud environments 67
5.1 Challenges for MapReduce in the Clouds 68
5.2 MRRoles4Azure (MapReduce Roles for Azure) 70
5.2.1 Client API and Driver 72
5.2.2 Map Tasks 72
5.2.3 Reduce Tasks 72
5.2.4 Monitoring 73
5.2.5 Fault Tolerance 74
5.2.6 Limitations of Azure MapReduce 74
5.3 Performance evaluation 74

5.3.1 Methodology 75
5.3.2 Smith-Waterman-GOTOH(SWG) pairwise distance calculation 76
5.3.3 Sequence assembly using Cap3 79
5.4 Summary 81
6. Data intensive iterative computations on cloud environments 82
6.1 Twister4Azure – Iterative MapReduce 83
6.1.1 Twister4Azure Programming model 84
6.1.2 Data Cache 88
6.1.3 Cache Aware Scheduling 89
6.1.4 Data broadcasting 91
6.1.5 Intermediate data communication 93
6.1.6 Fault Tolerance 95
6.1.7 Other features 95
6.1.8 Development and current status 96
6.2 Twister4Azure Scientific Application Case Studies 97
6.2.1 Methodology 97
6.2.2 Multi-Dimensional Scaling - Iterative MapReduce 99
6.2.3 KMeans Clustering 101
6.3 Summary 104
7. Performance Implications for Data Intensive Parallel applications on Cloud Environments ... 105

7.1 Inhomogeneous data 105
7.1.1 SW-G Pairwise Distance Calculation 105
7.1.2 CAP3 109
7.2 Virtualization overhead 111
7.2.1 SW-G Pairwise Distance Calculation 111
7.2.2 CAP3 113
7.3 Sustained performance of clouds 113
7.4 Performance Considerations for Data Caching on Azure for Iterative MapReduce computations 114
7.4.1 Local Storage Based Data Caching 115
7.4.2 In-Memory Data Caching 116
7.4.3 Memory Mapped File Based Data Cache 117
7.5 Summary 119
8. Collective Communications Primitives for Iterative MapReduce 121
8.1 Collective Communication 122
8.2 MapReduce 124
8.2.1 MapReduce Cost Model 126
8.3 MapReduce-MergeBroadcast 127
8.3.1 API 127
8.3.2 Merge Task 128

8.3.3 Broadcast 129
8.3.4 MapReduceMergeBroadcast Cost Model 130
8.3.5 Current iterative MapReduce Frameworks and MapReduce-MergeBroadcast 130
8.4 Motivation 131
8.4.1 Motivation 131
8.5 Map-AllGather Collective 135
8.5.1 Model 136
8.5.2 Fault tolerance 139
8.5.3 Benefits 139
8.5.4 Implementations 140
8.6 Map-AllReduce Collective 141
8.6.1 Model 141
8.6.2 Fault Tolerance 144
8.6.3 Benefits 144
8.6.4 Implementations 145
8.7 Evaluation 146
8.7.1 Multi-Dimensional Scaling using Map-AllGather 146
8.7.2 K-meansClustering using Map-AllReduce 153
9. Conclusion and Future Works 158
9.1 Summary 158

9.2 Contributions 159

9.3 Future Work 160

9.4 Conclusions 160

9.5 List of publications related to this thesis 161

10. References 163

List of Figures

Figure 1 Multi-Dimensional Scaling SMACOF application architecture using iterative MapReduce ... 36

Figure 2 Bio sequence analysis pipeline ... 37

Figure 3 Classic cloud processing architecture for pleasingly parallel computations ... 48

Figure 4 Hadoop MapReduce based processing model for pleasingly parallel computations ... 49

Figure 5 Cap3 application execution cost with different EC2 instance types ... 55

Figure 6 : Cap3 applciation compute time with different EC2 instance types ... 55

Figure 7 Parallel efficiency of Cap3 application using the pleasingly parallel frameworks ... 56

Figure 8 Cap3 execution time for single file per core using the pleasingly parallel frameworks ... 56

Figure 9 : Cost to process 64 query files using the BLAST application on different EC2 instance types ... 59

Figure 10 : Time to process 64 query files using the BLAST application on different EC2 instance types ... 59

Figure 11 Time to process 8 query files using BLAST application on different Azure instance types ... 60

Figure 12 : BLAST parallel efficiency using the pleasingly parallel frameworks ... 61

Figure 13 : BLAST average time to process a single query file using the pleasingly parallel frameworks ... 61

Figure 14 : Cost of using GTM interpolation application with different EC2 instance types ... 62

Figure 15 : GTM Interpolation compute time with different EC2 instance types ... 63

Figure 16: GTM Interpolation parallel efficiency using the pleasingly parallel frameworks ... 63

Figure 17 : GTM Interpolation performance per core using the pleasingly parallel frameworks ... 64

Figure 18 MapReduceRoles4Azure: Architecture for implementing MapReduce frameworks on Cloud environments using cloud infrastructure services ... 71

Figure 19 Task decomposition mechanism of SWG pairwise distance calculation MapReduce application 76

Figure 20 (a) SWG MapReduce pure performance (b) SWG MapReduce relative parallel efficiency (c) SWG MapReduce normalized performance (d) SWG MapReduce amortized cost for clouds 78

Figure 21 Cap3 MapReduce scaling performance 79

Figure 22 Cap3 MapReduce parallel efficiency 80

Figure 23 Cap3 MapReduce computational cost in cloud infrastructures 80

Figure 24 Structure of a typical data-intensive iterative application 85

Figure 25 Twister4Azure iterative MapReduce programming model 86

Figure 26 Cache Aware Hybrid Scheduling 91

Figure 27 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. 93

Figure 28 Performance of SW-G for randomly distributed inhomogeneous data with '400' mean sequence length. 107

Figure 29 Performances of SW-G for skewed distributed inhomogeneous data with '400' mean sequence length. 108

Figure 30 Performance of Cap3 for random distributed inhomogeneous data. 110

Figure 31 Performance of Cap3 for skewed distributed inhomogeneous data 110

Figure 32 Virtualization overhead of Hadoop SW-G on Xen virtual machines 112

Figure 33 Virtualization overhead of Hadoop Cap3 on Xen virtual machines 112

Figure 34 Sustained performance of cloud environments for MapReduce type of applications 114

Figure 35 Hadoop MapReduce MDS-BCCalc histogram 151

Figure 36 H-Collectives AllGather MDS-BCCalc histogram 151

Figure 37 H-Collectives AllGather MDS-BCCalc histogram without speculative scheduling..... 151

Table 1 Azure Instance Types 26

Table 2 Sample of AWS Instance Types 28

Table 3 Summary of MapReduce frameworks 38

Table 4: Summary of pleasingly parallel cloud technology features 50

Table 5 : Cost Comparison 57

Table 6 Evaluation cluster configurations 98

Table 7 Execution time analysis of a MDS computation. 116

Table 1 Summary of patterns 135

Table 2 Execution environments 146

List of Acronyms

1. INTRODUCTION

[DRAFT]

High performance parallel computing had been for long number of years. Used MPI which worked well. Also used specialized inter connects. Lot of scientific computations. Examples are fluid dynamics such as weather predictions, molecular simulations, etc.. which are mainly compute and communication bound. Lot of communications among each other. Very long execution times. Sometimes these are parts of a larger workflow. Traditionally these type of computations are performed by phyisist, astro, etc.. Currently these kind of computations are reaching exa-scale as well.

However, in the recent years there have been 3 disruptions in this landscape, which are in some ways related to each other. 1. Data deluge, shift to more data intensive from compute intensive. 2. Cloud computing. 3. Map reduce.

Data deluge description.. cannot alwas be done using traditional HPC. Traditional HPC is not the best for these..

Cloud computing description. Different from traditional clusters. Existing solutions may not work well. Scientists can use them effectively and easily.

MapReduce. Different goals than HPC frameworks.

Our goal of this thesis is to explore these new technologies and investigate how we can use these things effectively to solve data intensive computations. In the next section we do a detailed discussion about some of the challenges and approaches. We focus mostly on scientific use cases as industry takes care of other issues very well.

1.1 Statement of research problem

In this thesis we investigate whether cloud computing environments and related application frameworks can be used to perform large-scale parallel computations efficiently with good scalability, fault-tolerance and ease-of-use. The outcomes of this work would be,

1. Understanding the challenges and bottlenecks to perform scalable parallel computing on cloud environments through experimentation and benchmarking
2. Proposing solutions to the challenges and bottlenecks identified in 1.
3. Development of scalable parallel programming frameworks specifically designed for cloud environments to support efficient, reliable and user friendly execution of data intensive computations on cloud environments.
4. Develop of data intensive scientific applications using the frameworks developed in 3. Demonstrate that these applications can be executed on cloud environments in an efficient scalable manner.

1.2 Research Challenges

1. Programming model
    - Identifying suitable application abstractions while balancing the following conditions,
        i. Ability to express a sufficiently large and useful subset of large-scale data intensive computations
        ii. Keeping the programming model simple, easy-to-use and familiar (eg: extending existing models rather than reinventing) for the end-users.
        iii. Suitable for efficient execution in cloud environments
2. Data Storage

- Overcoming the bandwidth and latency limitations, when accessing large data products from cloud and other storages.
- Strategies (where to store, when to store, whether to store) for output and intermediate data storage.
- Clouds offer a variety of storage options. We need to choose the storage option best-suited for the particular data product and the particular use case.

3. Task Scheduling
    - Schedule tasks efficiently with an awareness of data availability and locality.
    - Support dynamic load balancing of computations and dynamically scaling of the compute resources.
4. Data Communication
    - Cloud infrastructures are known to exhibit inter-node I/O performance fluctuations (due to shared network, unknown topology), which affect the data communication performance. Frameworks should be designed with considerations for these fluctuations. These may include reducing the amount of communication required, overlapping communication with computation to avoid performance bottlenecks with regards to communication, identifying communication patterns which are better suited for the particular cloud environment, etc.
5. Fault tolerance
    - Ensuring the eventual completion of the computations through framework managed fault-tolerance mechanisms. These mechanisms also should strive to restore and complete the computations as efficiently as possible.
    - Taking care of handling of the tail of slow tasks to optimize the computations.

- Node failures are to be expected whenever large numbers of nodes are utilized for computations. But they become more prevalent when virtual instances are running on top of non-dedicated hardware. Frameworks should avoid single point of failures when a node fails.

6. Scalability
    - Computations should scale well with increasing amount of compute resources. Inter-process communication and coordination overheads as well as system bottlenecks need to be kept to a minimum to ensure acceptable scalability of the computations up to hundreds of instances or cores.
    - Computations should scale well with different input data sizes.
7. Efficiency
    - Facilitate the optimized execution of the applications achieving good parallel efficiencies for most of the commonly used application patterns. In order to achieve good efficiencies, the framework overheads such as scheduling, data staging, and intermediate data transfer needs to be low relative to the compute time. In addition, the applications must be able to utilize all the compute resources of the system ensuring ideal amount of parallelism.
    - Clouds are implemented as shared infrastructures operating using virtual machines. It's possible for the performance to fluctuate based the load of the underlying infrastructure services, based on the load from other users on the shared physical node , based on the load on the network and other issues which can be unique to VM environments. Frameworks should contain mechanism to handle any tasks that take much longer to complete than others and to provide appropriate load balancing in the job level.

8. Monitoring, Logging and Metadata storage *
    - Providing the users with sufficient capabilities to monitor the progress of their computations with the ability to drill down to the task level. This should also inform the users about any errors encountered as well as an overview of the CPU and memory utilization of the system.
    - Cloud instance storage is preserved only for the lifetime of the instance. Hence, information logged to the instance storage would be lost after the instance termination. On the other hand, performing excessive logging to a bandwidth limited off-instance storage location can become a performance bottleneck for the computations.
    - Frameworks need to maintain metadata information to manage and coordinate the jobs as well as the infrastructure. This metadata needs to be stored reliably while ensuring good scalability and the accessibility to avoid single point of failures and performance bottlenecks.

9. Cost effective *
    - Ensuring the cost of cloud services to be kept to an acceptable amount.
    - Choosing a suitable instance type: Clouds offer users several types of instance options, with different configurations and price points. It's important to select the best matching instance type, both in terms of performance as well as monetary wise.
10. Ease of usage *
    - Users should be able to develop, debug and deploy programs with ease without the need for extensive upfront system specific knowledge.

We are not focusing on the research issues monitoring, logging and metadata storage*, cost effectiveness* and ease of usage* in our current research. However, the solutions and frameworks we developed provide and in some cases improve the industry standard solutions for each issue.

1.3 Thesis contributions

- Architecture, programming model and prototype implementations to perform pleasingly parallel type computations on cloud environments using both cloud infrastructure services as well as using existing MapReduce frameworks. A detailed study of performance and cost of the cloud environments to perform pleasingly parallel computations.
- Decentralized architecture, programming model and prototype implementation to perform MapReduce computations on cloud environments using cloud infrastructure services. A detailed study of performance and challenges to perform MapReduce type computations on cloud environments, including the effect of inhomogeneous data and scheduling policies on the application performance.
- Decentralized architecture, programming model and prototype implementation to perform data intensive iterative MapReduce computations on cloud environments using cloud infrastructure services.
    - Multi-level data caching approach to solve the data bandwidth and latency issues of cloud storage services for iterative MapReduce.
    - High performance low overhead cache aware task scheduling algorithm for iterative applications on Cloud environments.
    - Hybrid data transfer approaches to improve the data communication performance on cloud environments without sacrificing the fault tolerance capabilities.

- Introduction of All-to-All collective communication operations to the iterative MapReduce model. Prototype implementation on Hadoop MapReduce and Twister4Azure iterative MapReduce.
- Applications performance studies using various run time environments including our prototype implementations.

1.4 Thesis outline

Chapter 2 of this thesis provides an introduction to the cloud environments and example applications used in this thesis. Chapter 2 also provides a classification of application that's used throughout in this thesis.

In chapter 3, we discuss the related works of this thesis. This includes a study of the other MapReduce and cloud oriented programming & execution frameworks. We also present a synthesis summary of these frameworks based on programming abstraction, data storage & communication mechanism, scheduling strategies, fault tolerance and several other dimensions.

Chapter 4 presents our work on performing pleasingly parallel computations on cloud environments. In this chapter, we introduce a set of frameworks that have been constructed using cloud-oriented programming frameworks and cloud infrastructure services to perform pleasingly parallel computations. We also present implementations and performance of several pleasingly parallel applications using the frameworks that were introduced in this chapter.

Chapter 5 explores the execution of MapReduce type applications on cloud environments and presents the MapReduceRoles4Azure MapReduce framework. MapReduceRoles4Azure is a novel MapReduce framework for cloud environments with a decentralized architecture built using the

Microsoft Windows Azure cloud infrastructure services. MapReduceRoles4Azure architecture successfully leverages the high latency, eventually consistent, yet highly scalable Azure infrastructure services to provide an efficient, on demand alternative to traditional MapReduce clusters. We also discuss the challenges posed by the unique characteristics of cloud environments for the efficient execution of MapReduce applications on clouds. Further we evaluate the use and performance of different MapReduce frameworks in cloud environments for several scientific applications.

Chapter 6 explores the execution of data intensive iterative MapReduce type applications on cloud environments and introduces the Twister4Azure iterative MapReduce framework. Twister4Azure is a distributed decentralized iterative MapReduce runtime for Windows Azure Cloud, which extends the familiar, easy-to-use MapReduce programming model with iterative extensions, enabling fault-tolerance execution of wide array of data mining and data analysis applications on the Azure cloud. This chapter also presents the Twister4Azure iterative MapReduce architecture for clouds, which 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 chapter also presents the implementation and performance of several real world data-intensive iterative applications.

Chapter7 study some performance implications for executing data intensive computations on cloud environments. These include the study of effect of inhomogeneous data and scheduling mechanisms on the performance of MapReduce applications, a study of the effects of virtualization overhead on MapReduce type applications and a study of the effect of sustained performance of cloud environments for the performance of MapReduce type applications. We also discuss and analyze various data caching strategies on Azure cloud environment to the performance of data intensive iterative MapReduce applications.

Chapter 8 discusses the applicability of All-to-All collective communication operations to Iterative MapReduce without sacrificing the desirable properties of MapReduce programming model and execution framework such as , fault tolerance, scalability, familiar API's and data model, etc. We show that the addition of collective communication operations enriches the iterative MapReduce model by providing many performance and ease of use advantages. We also present Map-AllGather primitive, which gathers the outputs from all the map tasks and distributes the gathered data to all the workers after a combine operation, and Map-AllReduce primitive, which combines the results of the Map Tasks based on a reduction operation and delivers the result to all the workers. MapReduceMergeBroadcast is presented as a canonical model representative of the most of the iterative MapReduce frameworks. Prototype implementations of these primitives on Hadoop and Twister4Azure as well as a performance comparison are studied in this chapter.

Finally we present a summary, conclusions and future work in chapter 9.

2. BACKGROUND

2.1 Cloud Environments

2.1.1 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. 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[12].

Table 1 Windows Azure instance types (as of July 2013)

Instance Name	CPU Cores	Memory	Cost Per Hour
Extra Small	Shared	768 MB	$0.02
Small	1	1.75 GB	$0.09
Medium	2	3.5 GB	$0.18
Large	4	7 GB	$0.36
Extra Large	8	14 GB	$0.72
Memory intensive (A6)	4	28 GB	$1.02
Memory intensive (A7)	8	56 GB	$2.04

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.

2.1.2 Amazon AWS

Amazon Web Services (AWS) [13] are a set of cloud computing services by Amazon, offering on-demand computing and storage services including, but not limited to, Elastic Compute Cloud (EC2), Simple Storage Service (S3) and Simple Queue Service (SQS).

EC2 provides users the option to lease virtual machine instances that are billed hourly and that allow users to dynamically provision resizable virtual clusters in a matter of minutes through a web service interface. EC2 supports both Linux and Windows virtual instances. EC2 follows an approach that uses infrastructure as a service; it provides users with 'root' access to the virtual machines, thus providing the

most flexibility possible. Users can store virtual machine snapshots as Amazon Machine Images (AMIs), which can then be used as templates for creating new instances. Amazon EC2 offers a variety of hourly billed instance sizes with different price points, giving users a richer set of options to choose from, depending on their requirements. One particular instance type of interest is the High-CPU-Extra-Large instances, which cost the same as the Extra-Large(XL) instances but offer greater CPU power and less memory than XL instances. Table 1 provides a summary of the EC2 instance types used in this paper. The clock speed of a single EC2 compute unit is approximately 1 GHz to 1.2 GHz. The Small instance type with a single EC2 compute unit is only available in a 32-bit environment, while the larger instance types also support a 64-bit environment.

Table 2 Sample of Amazon Web Services EC2 on-demand instance types

Instance Type	Memory	EC2 compute units	Actual CPU cores	Cost per hour
Micro	0.615GB	Variable	Shared	0.02$
Large (L)	7.5 GB	4	2 X (~2Ghz)	0.24$
Extra Large (XL)	15 GB	8	4 X (~2Ghz)	0.48$
High CPU Extra Large (HCXL)	7 GB	20	8 X (~2.5Ghz)	0.58$
High Memory 4XL (HM4XL)	68.4 GB	26	8 X (~3.25Ghz)	1.64$
Cluster GPU	22.5	33.5	8 X (~2.93Ghz)	2.10$

SQS is a reliable, scalable, distributed web-scale message queue service that is eventually consistent and ideal for small, short-lived transient messages. SQS provides a REST-based web service interface that enables any HTTP capable client to use it. Users can create an unlimited number of queues and send an unlimited number of messages. SQS does not guarantee the order of the messages, the deletion of messages or the availability of all the messages for a request, though it does guarantee eventual availability over multiple requests. Each message has a configurable visibility timeout. Once it is read by a client, the message will be hidden from other clients until the visibility time expires. The message

reappears upon expiration of the timeout as long as it is not deleted. The service is priced based on the number of API requests and the amount of data transfer.

S3 provides a web-scale distributed storage service where users can store and retrieve any type of data through a web services interface. S3 is accessible from anywhere in the web. Data objects in S3 are access controllable and can be organized in to buckets. S3 pricing is based on the size of the stored data, amount of data transferred and the number of API requests.

2.1.2.1 Amazon Elastic Map Reduce

Amazon Elastic MapReduce (EMR) [14] provides MapReduce as an on-demand service hosted within the Amazon infrastructure. EMR is a hosted Apache Hadoop [15] MapReduce framework, which utilizes Amazon EC2 for computing power and Amazon S3 for data storage. It allows the users to perform Hadoop MapReduce computations in the cloud with the use of a web application interface, as well as a command line API, without worrying about installing and configuring a Hadoop cluster. Users can run their existing Hadoop MapReduce program on EMR with minimal changes.

EMR supports the concept of JobFlows, which can be used to support multiple steps of Map & Reduce on a particular data set. Users can specify the number and the type of instances that are required for their Hadoop cluster. For EMR clusters consisting of more than one instance, EMR uses one instance exclusively as the Hadoop master node. EMR does not provide the ability to use custom AMI images. As an alternative to custom AMI images, EMR provides Bootstrap actions that users can specify to run on the computing nodes prior to the launch of the MapReduce job, which can be used for optional custom preparation of the images. EMR allows for debugging of EMR jobs by providing the option to upload the Hadoop log files in to S3 and state information to SimpleDB. Intermediate data and temporary data are stored in the local HDFS file system while the job is executing. Users can use either

the S3 native (s3n) file system or the legacy S3 block file system to specify input and output locations on Amazon S3. Use of s3n is recommended, as it allows files to be saved in native formats in S3.

The pricing for the use of EMR consists of the cost for the EC2 computing instances, the S3 storage cost, an optional fee for the usage of SimpleDB to store job debugging information, and a separate cost per instance hour for the EMR service.

2.2 Application types

For the purposes of this dissertation, we classify parallel applications in to the following four categories based on their execution patterns.

2.2.1 Pleasingly Parallel Applications

A pleasingly (also called embarrassingly) parallel application is an application that can be parallelized, thus requiring minimal effort to divide the application into independent parallel parts. Each independent parallel part has very minimal or no data, synchronization or ordering dependencies with the others. These applications are good candidates for computing clouds and compute clusters with no specialized interconnections. A sizable number of scientific applications fall under this category. Examples of pleasingly parallel applications include Monte Carlo simulations, BLAST[16] searches, parametric studies and image processing applications such as ray tracing. Most of the data cleansing and pre-processing applications can also be classified as pleasingly parallel applications. These type of applications can be mapped to MapReduce programming model as Map only applications or MapReduce applications with trivial reduce phases such as simple aggregation or collection of data.

Chapter 4 of this thesis focuses on providing solutions to executing pleasingly parallel applications on cloud environments and using cloud oriented applications frameworks. Pleasingly parallel

applications discussed in this thesis include BLAST sequence searching (section 2.3.3), Cap3 sequence assembly (section 2.3.1) and GTM interpolation (section 2.3.2).

2.2.2 MapReduce Type Applications

We define MapReduce type applications as the set of applications that consist of a pleasingly parallel step (Map) followed by a non-trivial reduction step. The non-trivial reduction can take advantage of the combining, sorting and partitioning functionalities provided by the MapReduce frameworks. Examples of MapReduce type applications include Smith-Watermann-GOTOH sequence distance calculation [17] and WordCount applications.

Chapter 5 of this thesis focuses on providing solutions to executing MapReduce type of applications on cloud environments and introduces MapReduceRoles4Azure decentralized cloud MapReduce framework for Azure cloud. MapReduce type applications discussed in this thesis include Smith-Watermann-GOTOH sequence distance calculation (section 2.3.4) application.

2.2.3 Data Intensive Iterative Applications

Many important scientific applications and algorithms can be implemented as iterative computation and communication steps, where computations inside an iteration are independent and are synchronized at the end of each iteration through reduce and communication steps. Often, each iteration is also amenable to parallelization. Many statistical applications fall in this category. Examples include clustering algorithms, data mining applications, machine learning algorithms, data visualization algorithms, and most of the expectation maximization algorithms. The growth of such iterative statistical applications, in importance and number, is driven partly by the need to process massive amounts of data, for which scientists rely on clustering, mining, and dimension-reduction to interpret the data. Emergence of computational fields, such as bioinformatics, and machine learning, have also contributed to an increased interest in this class of applications.

Chapter 6 of this thesis focuses on providing solutions to executing data intensive iterative applications on cloud environments and introduces Twister4Azure iterative MapReduce framework for Azure cloud. Also chapter **Error! Reference source not found.** of this thesis introduces the collective ommunication primitives for iterative MapReduce type applications. Data intensive iterative type applications discussed in this thesis include KMeansClustering (section 2.3.5), Multi-Dimensional-Scaling (section 2.3.6) and PageRank (section **Error! Reference source not found.**) applications.

2.2.4 MPI type applications

Applications with more complex inter-process communication and coordination requirements than the data intensive iterative applications fall in to this category. These applications often require the usage of technologies such as MPI or OpenMP together with special communications interconnects. We do not explore MPI type applications in this thesis.

2.3 Applications

Described below are some of applications that we implemented and/or parallelized, benchmarked and analyzed in the latter chapters of this thesis.

2.3.1 Cap3

Cap3 [18] is a sequence assembly program which assembles DNA sequences by aligning and merging sequence fragments to construct whole genome sequences. Sequence assembly is an integral part of genomics as the current DNA sequencing technology, such as shotgun sequencing, is capable of reading only parts of genomes at once. The Cap3 algorithm operates on a collection of gene sequence fragments presented as FASTA formatted files. It removes the poor regions of the DNA fragments, calculates the overlaps between the fragments, identifies and removes the false overlaps, joins the

fragments to form contigs of one or more overlapping DNA segments and finally through multiple sequence alignment generates consensus sequences.

The increased availability of DNA sequencers are generating massive amounts of sequencing data that needs to be assembled. Cap3 program is often used in parallel with lots of input files due to the pleasingly parallel nature of the application. The run time of the Cap3 application depends on the contents of the input file. Cap3 is less memory intensive than the GTM Interpolation and BLAST applications we discuss below. Size of a typical data input file for the Cap3 program and the result data file range from hundreds of kilobytes to few megabytes. Output files resulting from the input data files can be collected independently and do not need any combining steps.

2.3.2 Generative Topographic Mapping Interpolation

Generative Topographic Mapping **(**GTM)[19] is an algorithm for finding an optimal user-defined low-dimensional representation of high-dimensional data. This process is known as dimension reduction, which plays a key role in scientific data visualization. In a nutshell, GTM is an unsupervised learning method for modeling the density of data and finding a non-linear mapping of high-dimensional data in a low-dimensional space. To reduce the high computational costs and memory requirements in the conventional GTM process for large and high-dimensional datasets, GTM Interpolation [20] has been developed as an out-of-sample extension to process much larger data points with minor trade-off of approximation. GTM Interpolation takes only a part of the full dataset, known as samples, for a compute-intensive training process and applies the trained result to the rest of the dataset, known as out-of-samples. With this interpolation approach in GTM, one can visualize millions of data points with modest amount of computations and memory requirement.

The size of the input data for the interpolation algorithm consists of millions of data points and usually ranges in gigabytes, while the size of the output data in lower dimensions are orders of

magnitude smaller than the input data. Input data can be partitioned arbitrarily on the data point boundaries in order to generate computational sub tasks. The output data from the sub tasks can be collected using a simple merging operation and do not require any special combining functions. The GTM Interpolation application is highly memory intensive and requires a large amount of memory proportional to the size of the input data.

2.3.3 Blast+ sequence search

NCBI BLAST+ [16] is a very popular bioinformatics application that is used to handle sequence similarity searching. It is the latest version of BLAST [21], a multi-letter command line tool developed using the NCBI C++ toolkit, to translate a FASTA formatted nucleotide query and to compare it to a protein database. Queries are processed independently and have no dependencies between them. This makes it possible to use multiple BLAST instances to process queries in a pleasingly parallel manner. We used a sub-set of a real-world protein sequence data set as the input BLAST queries and used NCBI's non-redundant (NR) protein sequence database (8.7 GB), updated on 6/23/2010, as the BLAST database. In order to make the tasks coarser granular, we bundled 100 queries in to each data input file resulting in files with sizes in the range of 7-8 KB. The output files for these input data range from few bytes to few Megabytes.

2.3.4 Sequence alignment using SmithWaterman GOTOH (SWG)

SmithWaterman [22] algorithm with GOTOH [23] (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.

We used an open source implementation, named NAligner[11], of the Smith Waterman – Gotoh algorithm SW-G **Error! Reference source not found.Error! Reference source not found.** modified to ensure low start up effects by each thread, processing a large number (above a few hundred) of sequence calculations at a time. Memory bandwidth needed was reduced by storing data items in as few bytes as possible.

More details about the Hadoop-SWG application implementation can be found in [17].

2.3.5 KMeansClustering

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 KMeans Clustering 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, 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.

2.3.6 Multi-Dimensional Scaling (MDS)

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[24]. 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.

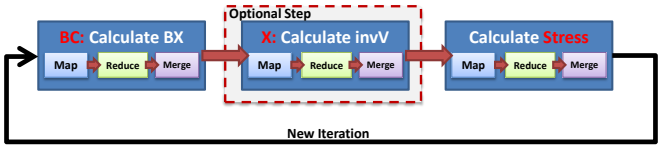


Figure 1 Multi-Dimensional Scaling SMACOF application architecture using iterative MapReduce

For the purposes of this dissertation, we use Scaling by MAjorizing a COmplicated Function (SMACOF)[25], 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[20]. This results in iterating a chain of three MapReduce jobs, as depicted in Figure 5. For the purposes of this dissertation, 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 MapReduce frameworks due to its relatively finer grained task sizes and multiple MapReduce applications per iteration.

2.3.7 Bio sequence analysis pipeline

The bio-informatics genome processing and visualizing pipeline[14] shown in Figure 2 inspired the application use cases analyzed in this paper. This pipeline uses SmithWatermann-GOTOH application, described in section 2.3.4, or BLAST+ application, described in section 2.3.3, for sequence alignment, Pairwise clustering for sequence clustering and the Multi-Dimensional Scaling application, described in section 2.3.6, 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.

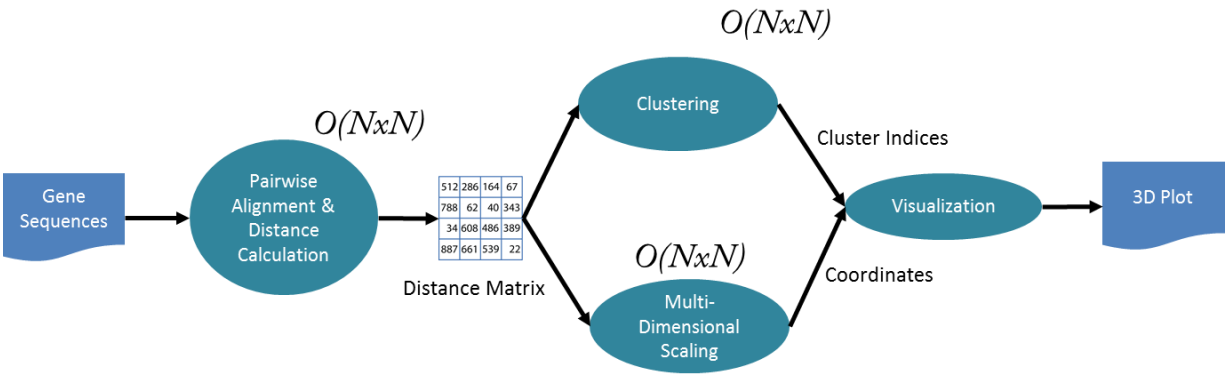


Figure 2 Bio sequence analysis pipeline[14]

3. RELATED WORKS

Table 3 Summary of MapReduce frameworks

	Google MapReduce	Hadoop	Twister	Dryad	Twister4Azure	MRRoles4Azure
Parallel Model	MapReduce	MapReduce	Map Reduce, Iterative Map Reduce	DAG; Extensible;	Map Reduce, Iterative Map Reduce	
Data Storage	GFS		Messaging frameworks,	Files (GFS like); TCP pipes; Shared memory FIFO		
Data Communication	Files					
Scheduling	Data/rack locality	Data/rack locality, Global dynamic scheduling	Reuse map tasks, preload fixed data	Data locality; hard/soft constraints; Network topology based run time graph optimizations		
Failure Handling	Rerunning of failed tasks, Duplicate executions					
Language Support	C++ base classes					
Monitoring	Extensive web interface, task/node level monitoring					

CloudMapReduce[26] for Amazon Web Services (AWS) and Google AppEngine MapReduce[27] follow an architecture similar to MRRoles4Azure, in which they utilize the cloud services as the building blocks. Amazon ElasticMapReduce[13] offers Apache Hadoop as a hosted service on the Amazon AWS cloud environment. However, none of them support iterative MapReduce. Spark[11] 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[28] 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[29] 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.

3.1.1 Apache Hadoop

Apache Hadoop[5] MapReduce is a widely used open-source implementation of the Google MapReduce[1] distributed data processing framework. Apache Hadoop MapReduce uses the Hadoop distributed parallel file system(HDFS) [30] 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 designed for deployment on commodity clusters and achieves reliability through replication of data across nodes. 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. Hadoop performs data distribution and automatic task partitioning based on the information provided in the master program and based on the structure of the data stored in HDFS. 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.

3.1.2 Twister

The Twister[31] 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[32]. 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[33] is used for the current experiments. Twister performs optimized broadcasting

operations by using chain method[10] and uses minimum spanning tree method[34] 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[30].

3.1.3 Microsoft Daytona

Microsoft Daytona[10] 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.

3.1.4 Haloop

Haloop[8] extends Apache Hadoop to support iterative applications and supports 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 doesn't have an explicit Combine operation to get the output to the master node and uses a separate MapReduce job to do the calculation (called Fixpoint evaluation) for terminal condition evaluation. HaLoop provides a high-level query language, which is not available in either Java HPC Twister or Twister4Azure.

HaLoop performs 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 with similar 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. If the associations can no longer hold due to the load, the master node will associate the data with another node. HaLoop also provides several mechanisms of on disk data caching such as reducer input cache and mapper input cache. In addition to these two, there is another cache called reducer output cache, which is specially designed to support Fixpoint Evaluations. HaLoop can also cache intermediate data (reducer input/output cache) generated by the first iteration.

3.1.5 Microsoft Dryad

Dryad [35] is a framework developed by Microsoft Research as a general-purpose distributed execution engine for coarse-grain parallel applications. Dryad applications are expressed as directed acyclic data-flow graphs (DAG), where vertices represent computations and edges represent communication channels between the computations. DAGs can be used to represent MapReduce type

computations and can be extended to represent many other parallel abstractions too. Similar to the MapReduce frameworks, the Dryad scheduler optimizes the data transfer overheads by scheduling the computations near data and handles failures through rerunning of tasks and duplicate task execution. In the Dryad version we used, data for the computations need to be partitioned manually and stored beforehand in the local disks of the computational nodes via Windows shared directories. Dryad is available for academic usage through the DryadLINQ [6] API, which is a high level declarative language layer on top of Dryad. DryadLINQ queries get translated in to distributed Dryad computational graphs in the run time. DryadLINQ can be used only with Microsoft Windows HPC clusters. The DryadLINQ implementation of the framework uses the DryadLINQ "select" operator on the data partitions to perform the distributed computations. The resulting computation graph looks much similar to the figure 2, where instead of using HDFS, Dryad will use the Windows shared local directories for data storage. Data partitioning, distribution and the generation of metadata files for the data partitions is implemented as part of our pleasingly parallel application framework.

3.1.6 Spark

3.1.7 i-MapReduce

3.1.8 PrItr

3.1.9 Apache Giraph

3.1.10 Dremel/Pregel

Another approach introduced by Lin et al[36] is to use…

Cite "MapReduce is not good enough" by jimmy lin. May be in the introduction.

3.2 Summary

There exist many studies [4, 37, 38] that benchmark existing traditionally-distributed scientific applications on the cloud. In contrast, we focused on implementing and analyzing the performance of biomedical applications using cloud services/technologies and cloud-oriented programming frameworks. In one of our earlier studies [39], we analyzed the overhead of virtualization and the effect of inhomogeneous data on the cloud-oriented programming frameworks. Ekanayake and Fox [38] analyzed the overhead of MPI running on virtual machines under different VM configurations and under different MPI stacks.

In addition to the biomedical applications discussed in this paper, we have also developed distributed pairwise sequence alignment applications using MapReduce programming models [39]. There are other biomedical applications developed using MapReduce programming frameworks, such as CloudBurst [40], which performs parallel genome read mappings. CloudBLAST [41] performs distributed BLAST computations using Hadoop and implements an architecture similar to the Hadoop-BLAST used in this paper. AzureBlast [28] presents a distributed BLAST implementation for Azure Cloud infrastructure developed using Azure Queues, Tables and Blob Storage with an architecture similar to our Classic-Cloud AzureBlast implementation.

Walker [42] presents a more detailed model for buying versus leasing decisions for CPU power based on lease-or-buy budgeting models, pure CPU hours, Moore's law, etc.. Our cost estimation in Section 4.3.2.1 is based on the pure performance of the application in different environments, the purchase cost of the cluster and the estimate of the maintenance cost. Walker also highlights the advantages of the mobility user's gain through the ability to perform short-term leases from cloud computing environments, allowing them to adopt the latest technology. Wilkening et al [43] presents a cost-based feasibility study for using BLAST in EC2 and concludes that the cost in clouds is slightly higher

than the cost of using compute clusters. They benchmarked the BLAST computation directly inside the EC2 instances without using a distributed computing framework and also assumed the local cluster utilization to be 100%.

4. PLEASINGLY PARALLEL COMPUTING ON CLOUD ENVIRONMENTS

A pleasingly parallel application is an application that can be parallelized, thus requiring minimal effort to divide the application into independent parallel parts. Each independent parallel part has very minimal or no data, synchronization or ordering dependencies with the others. These applications are good candidates for computing clouds and compute clusters with no specialized interconnections. There are many scientific applications that fall under this category. Examples of pleasingly parallel applications include Monte Carlo simulations, BLAST searches, parametric studies and image processing applications such as ray tracing. Most of the data cleansing and pre-processing applications can also be classified as pleasingly parallel applications. Recently, the relative number of pleasingly parallel scientific workloads has grown due to the emergence of data-intensive computational fields such as bioinformatics.

In this chapter, we introduce a set of frameworks that have been constructed using cloud-oriented programming models to perform pleasingly parallel computations. Using these frameworks, we present distributed parallel implementations of biomedical applications such as the Cap3 [18] sequence assembly, BLAST sequence search and GTM Interpolation. We analyze the performance, cost and usability of different cloud-oriented programming models using the above-mentioned implementations. We use Amazon Web Services [13] and Microsoft Windows Azure [44] cloud computing platforms and Apache Hadoop [5] MapReduce and Microsoft DryaLINQ [6] as the distributed parallel computing frameworks.

4.1 Pleasingly parallel application architecture

Processing large data sets using existing sequential executables is a common use case encountered in many scientific applications. Many of these applications exhibit pleasingly parallel characteristics in

which the data can be independently processed in parts. In the following sections we explore cloud programming models and the frameworks that we developed to perform pleasingly parallel computations

4.1.1 Classic Cloud processing model

Figure 3 depicts the Classic Cloud processing model. Varia [45] and Chappell [46] describe similar architectures that are implemented using Amazon and Azure processing models respectively. The Classic Cloud processing model follows a task processing pipeline approach with independent workers. It uses the cloud instances (EC2/Azure Compute) for data processing and uses Amazon S3/Windows Azure Storage for the data storage. For the task scheduling pipeline, it uses an Amazon SQS or an Azure queue as a queue of tasks where every message in the queue describes a single task. The client populates the scheduling queue with tasks, while the worker-processes running in cloud instances pick tasks from the scheduling queue. The configurable visibility timeout feature of Amazon SQS and Azure Queue service is used to provide a simple fault tolerance capability to the system. The workers delete the task (message) in the queue only after the completion of the task. Hence, a task (message) will get processed by some worker if the task does not get completed with the initial reader (worker) within the given time limit. Rare occurrences of multiple instances processing the same task or another worker re-executing a failed task will not affect the result due to the idempotent nature of the independent tasks.

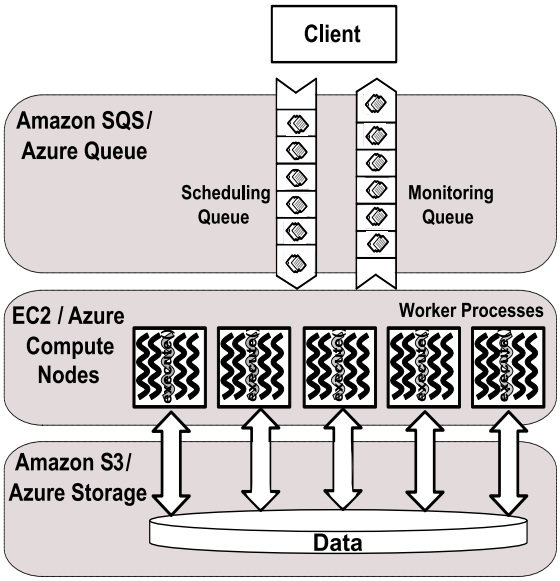


Figure 3 Classic cloud processing architecture for pleasingly parallel computations

For the applications discussed in this chapter, a single task comprises of a single input file and a single output file. The worker processes will retrieve the input files from the cloud storage through the web service interface using HTTP and will process them using an executable program before uploading the results back to the cloud storage. In this implementation user can configure the workers to use any executable program in the virtual machine to process the tasks, provided that it takes input in the form of a file. Our implementation uses a monitoring message queue to monitor the progress of the computation. One interesting feature of the Classic Cloud framework is the ability to extend it to use the local machines and clusters side by side with the clouds. Although it might not be the best option due to the data being stored in the cloud, one can start workers in computers outside of the cloud to augment compute capacity.

4.1.2 Pleasingly parallel processing using MapReduce frameworks

We implemented similar pleasingly parallel processing frameworks using Apache Hadoop[5] and Microsoft DryadLINQ[6].

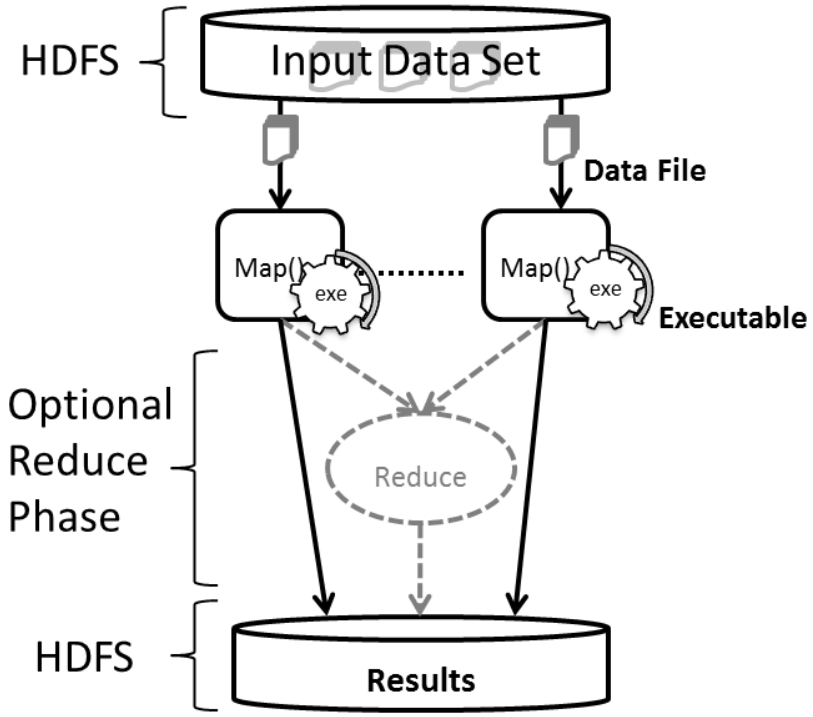


Figure 4 Hadoop MapReduce based processing model for pleasingly parallel computations

As shown in Figure 4, the pleasingly parallel application framework on Hadoop is developed as a set of map tasks which process the given data splits (files) using the configured executable program. Input to a map task comprises of key, value pairs, where by default Hadoop parses the contents of the data split to read them. Most of the legacy data processing applications expect a file path as the input instead of the contents of the file, which is not possible with the Hadoop built-in input formats and record readers. We implemented a custom InputFormat and a RecordReader for Hadoop to provide the file name and the HDFS path of the data split respectively as the key and the value for the map function, while preserving the Hadoop data locality based scheduling.

The DryadLINQ [6] implementation of the framework uses the DryadLINQ “select” operator on the data partitions to perform the distributed computations. The resulting computation graph looks much similar to the Figure 4, where instead of using HDFS, Dryad will use the Windows shared local directories

for data storage. Data partitioning, distribution and the generation of metadata files for the data partitions is implemented as part of our pleasingly parallel application framework.

4.1.3 Usability of the technologies

Table 4: Summary of pleasingly parallel cloud framework features

	AWS/ Azure	**Hadoop**	**DryadLINQ**
Programming patterns	Independent job execution, More structure possible using client side driver program.	MapReduce	DAG execution, Extensible to MapReduce and other patterns
Fault Tolerance	Task re-execution based on a configurable time out	Re-execution of failed and slow tasks.	Re-execution of failed and slow tasks.
Data Storage and Communication	S3/Azure Storage. Data retrieved through HTTP.	HDFS parallel file system. TCP based Communication	Local files
Environments	EC2/Azure virtual instances, local compute resources	Linux cluster, Amazon Elastic MapReduce	Windows HPCS cluster
Scheduling and Load Balancing	Dynamic scheduling through a global queue, providing natural load balancing	Data locality, rack aware dynamic task scheduling through a global queue, providing natural load balancing	Data locality, network topology aware scheduling. Static task partitions at the node level, suboptimal load balancing

Implementing the above-mentioned application framework using the Hadoop and DryadLINQ data processing frameworks was easier than implementing them from the scratch using cloud infrastructure services as the building blocks. Hadoop and DryadLINQ take care of scheduling, monitoring and fault tolerance. With Hadoop, we had to implement a Map function, which copy the input file from HDFS to the working directory, execute the external program as a process and finally upload the result file to the HDFS. It was also necessary to implement a custom InputFormat and a RecordReader to support file inputs to the map tasks. With DryadLINQ, we had to implement a side effect-free function to execute the program on the given data and copy the result to the output-shared directory. But significant effort had to be spent on implementing the data partition and the distribution programs to support DryadLINQ.

EC2 and Azure Classic Cloud implementations involved more effort than the Hadoop and DryadLINQ implementations, as all the scheduling, monitoring and fault tolerance had to be implemented from scratch using the cloud infrastructure services' features. The deployment process was easier with Azure as opposed to EC2, in which we had to manually create instances, install software and start the worker instances. On the other hand the EC2 infrastructure gives developers more flexibility and control. Azure SDK provides better development, testing and deployment support through Visual Studio integration. The local development compute fabric and the local development storage of the Azure SDK make it much easier to test and debug Azure applications. While the Azure platform is heading towards providing a more developer-friendly environment, it still lags behind in terms of the infrastructure maturity Amazon AWS has accrued over the years.

4.2 Evaluation Methodology

In the performance studies, we use parallel efficiency as the measure by which to evaluate the different frameworks. Parallel efficiency is a relatively good measure for evaluating the different approaches we use in our studies, as we do not have the option of using identical configurations across the different environments. At the same time, we cannot use efficiency to directly compare the different technologies. Even though parallel efficiency accounts for the system dissimilarities that affect the sequential and the parallel run time, it does not reflect other dissimilarities, such as memory size, memory bandwidth and network bandwidth. Parallel efficiency for a parallel application on P number of cores can be calculated using the following formula:

$$\text{ParallelEfficiency} = \frac{T_1}{pT_p} \quad \text{--- Equation 1[47]}$$

In this equation, T_p is the parallel run time for the application. T_1 is the best sequential run time for the application using the same data set or a representative subset. In this paper, the sequential run time for the applications was measured in each of the different environments, having the input files present in the local disks, avoiding the data transfers.

The average run time for a single computation in a single core is calculated for each of the performance tests using the following formula. The objective of this calculation is to give readers an idea of the actual performance they can obtain from a given environment for the applications considered in this paper.

$$\text{Avg. run time for a single computation in a single core} = \frac{pT(\rho)}{No.of\ computations} \quad \text{--- Equation 2}$$

Due to the richness of the instance type choices Amazon EC2 provides, it is important to select an instance type that optimizes the balance between performance and cost. We present instance type studies for each of our applications for the EC2 instance types mentioned in Table 2**Error! Reference ource not found.** using 16 CPU cores for each study. EC2 Small instances were not included in our study

because they do not support 64-bit operating systems. We do not present results for Azure Cap3 and GTM Interpolation applications, as the performance of the Azure instance types for those applications scaled linearly with the price. However, the total size of memory affected the performance of BLAST application across Azure instance types; hence we perform an instance type study for BLAST on Azure.

Cloud virtual machine instances are billed hourly. When presenting the results, the 'Compute Cost (hour units)' assumes that particular instances are used only for the particular computation and that no useful work is done for the remainder of the hour, effectively making the computation responsible for the entire hourly charge. The 'Amortized Cost' assumes that the instance will be used for useful work for the remainder of the hour, making the computation responsible only for the actual fraction of time during which it was executed. The horizontal axes of the EC2 cost figures (Figure 3 and 7) and the vertical axis labeling of the EC2 compute time figures (Figures 4 and 8) are labeled in the format 'Instance Type' – 'Number of Instances' X 'Number of Workers per Instance'. For an example, HCXL – 2 X 8 means two High-CPU-Extra-Large instances were used with 8 workers per instance.

When presenting the results used in this paper, we considered a single EC2 Extra-Large instance, with 20 EC2 compute units as 8 actual CPU cores while an Azure Small instance was considered as a single CPU core. In all of the test cases, it is assumed that the data was already present in the framework's preferred storage location. We used Apache Hadoop version 0.20.2 and DryadLINQ version 1.0.1411.2 (November 2009) for our studies.

In section 7.3, we investigated the sustained performance of Amazon AWS and Windows Azure cloud infrastructures for MapReduce type applications over a week's time during different times of the day. Performance variations we observed were very minor, with standard deviations of 1.56% for Amazon AWS and 2.25% for Windows Azure, with no noticeable correlations with the day of the week or

the time of the day. Hence, we assume that the performance results we obtained for this paper would not depend on such variables.

4.3 Cap3

We implemented a distributed parallel version of Cap3[18] sequence assembly application for Amazon EC2, Microsoft Azure, DryadLINQ and for Apache Hadoop using the frameworks that were presented in section 4.1. Cap3 program is a pleasingly parallel application that is often used in parallel with lots of input files. More details on Cap3 is given in section 2.3.1.

4.3.1 Performance with different EC2 cloud instance types

Figure 5 and Figure 6 present benchmark results for the Cap3 application on different EC2 instance types. These experiments processed 200 FASTA files, each containing 200 reads using 16 compute cores. According to these results, we can infer that memory is not a bottleneck for the Cap3 program and that performance depends primarily on computational power. While the EC2 High-Memory-Quadruple-Extra-Large instances show the best performance due to the higher clock-rated processors, the most cost effective performance for the Cap3 EC2 ClassicCloud application is gained using the EC2 High-CPU-Extra-Large instances.

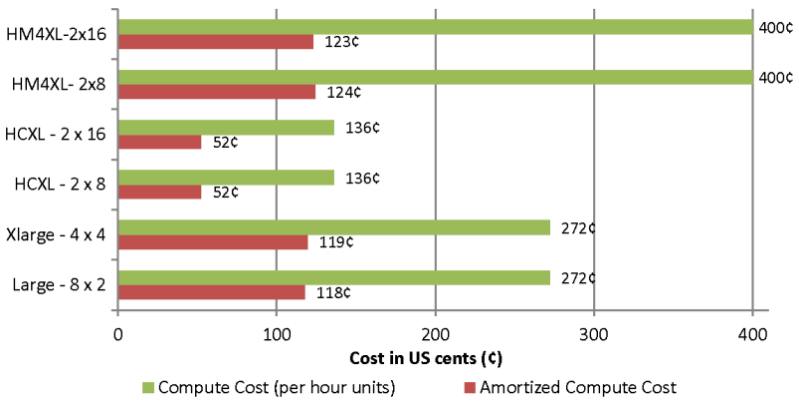


Figure 5 Cap3 application execution cost with different EC2 instance types

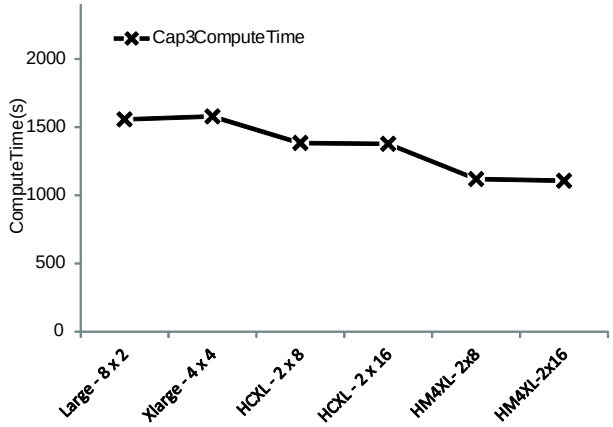


Figure 6 : Cap3 applciation compute time with different EC2 instance types

4.3.2 Scalability study

We benchmarked the Cap3 Classic Cloud implementation performance using a replicated set of FASTA-formatted data files, each file containing 458 reads, and compared this to our previous performance results [39] for Cap3 DryadLINQ and Cap3 Hadoop. 16 High-CPU-Extra-Large instances were used for the EC2 testing and 128 small Azure instances were used for the Azure Cap3 testing. The DryadLINQ and Hadoop bare metal results were obtained using a 32 node X 8 core (2.5 Ghz) cluster with 16 GB of memory on each node.

Load balancing across the different sub tasks does not pose a significant overhead in the Cap3 performance studies, as we used a replicated set of input data files making each sub task identical. We performed a detailed study of the performance of Hadoop and DryadLINQ in the face of inhomogeneous data in one of our previous studies [39]. In this study, we noticed better natural load balancing in Hadoop than in DryadLINQ due to Hadoop's dynamic global level scheduling as opposed to DryadLINQ's static task partitioning. We assume that cloud frameworks will be able perform better load balancing similar to Hadoop because they share the same dynamic scheduling global queue-based architecture.

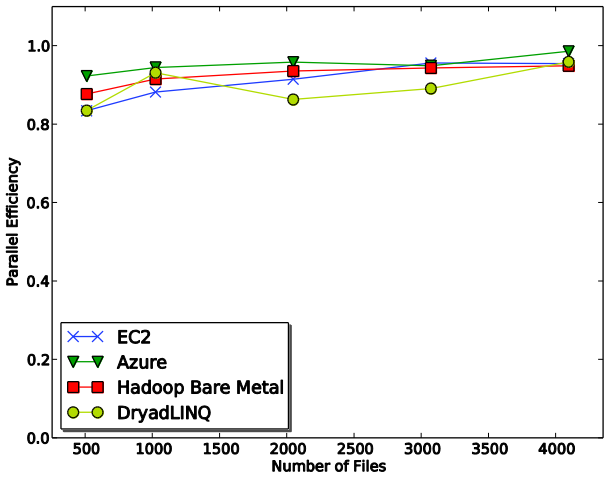


Figure 7 Parallel efficiency of Cap3 application using the pleasingly parallel frameworks

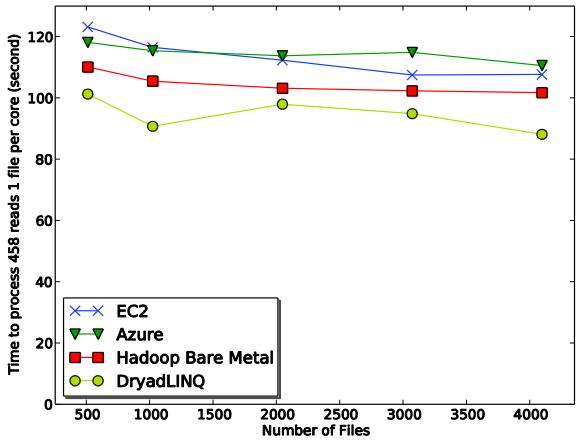


Figure 8 Cap3 execution time for single file per core using the pleasingly parallel frameworks

Based on Figure 7 and Figure 8, we can conclude that all four implementations exhibit comparable parallel efficiency (within 20%) with low parallelization overheads. When interpreting Figure 6, it should be noted that the Cap3 program performs ~12.5% faster on Windows environment than on the Linux environment. As mentioned earlier, we cannot use these results to claim that a given framework performs better than another, as only approximations are possible, given that the underlying infrastructure configurations of the cloud environments are unknown.

4.3.2.1 Cost comparison

Table 5 : Cost Comparison of Cap3 execution among different cloud environments

	Amazon Web Services	Azure
Compute Cost	10.88 $ (0.68$ X 16 HCXL)	15.36$ (0.12$ X 128 Azure Small)
Queue messages (~10,000)	0.01 $	0.01 $
Storage (1GB, 1 month)	0.14 $	0.15 $
Data transfer in/out (1 GB)	0.10 $ (in)	0.10$ (in) + 0.15$ (out)
Total Cost	11.13 $	15.77 $

In Table 5, we estimate the cost of assembling 4096 FASTA files using Classic Cloud frameworks on EC2 and on Azure. For the sake of comparison, we also approximate the cost of the computation using one of our internal compute clusters (32 node 24 core, 48 GB memory per node with Infiniband interconnects), with the cluster purchase cost (~500,000$) depreciated over the course of 3 years plus the yearly maintenance cost (~150,000$), which include power, cooling and administration costs. We executed the Hadoop-Cap3 application in our internal cluster for this purpose. The cost for computation using the internal cluster was approximated to 8.25$ US for 80% utilization, 9.43$ US for 70% utilization and 11.01$ US for 60% utilization. For the sake of simplicity, we did not consider other factors such as

the opportunity costs of the upfront investment, equipment failures and upgradability. There would also be additional costs in the cloud environments for the instance time required for environment preparation and minor miscellaneous platform-specific charges, such as the number of storage requests.

4.4 BLAST

NCBI BLAST+ [16] is a very popular bioinformatics application that is used to handle sequence similarity searching described in section 2.3.3. We implemented distributed parallel versions of BLAST application for Amazon EC2, Microsoft Azure, DryadLINQ and for Apache Hadoop using the frameworks that were presented in section 4.1. All of the implementations download and extract the compressed BLAST database (2.9GB compressed) to a local disk partition of each worker prior to beginning processing of the tasks. Hadoop-BLAST uses the Hadoop-distributed cache feature to distribute the database. We added a similar data preloading feature to the Classic Cloud frameworks, in which each worker will download the specified file from the cloud storage at the time of startup. In the case of DryadLINQ, we manually distributed the database to each node using Windows-shared directories. The performance results presented in this paper do not include the database distribution times.

4.4.1 Performance with different cloud instance types

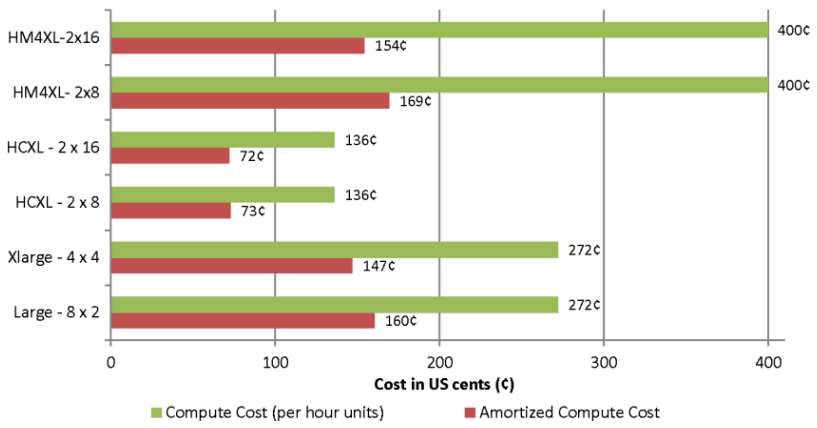


Figure 9 : Cost to process 64 BLAST query files on different EC2 instance types

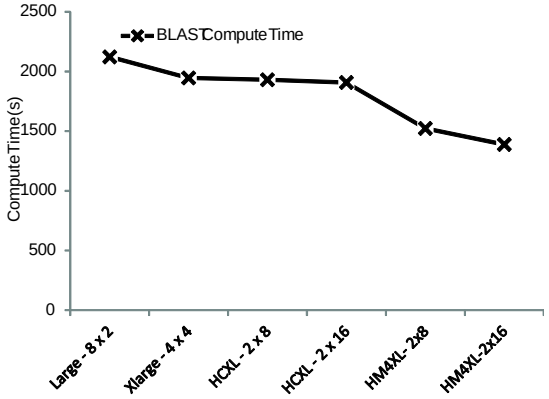


Figure 10 : Time to process 64 BLAST query files on different EC2 instance types

Figure 9 and Figure 10 present the benchmark results for BLAST Classic Cloud application on different EC2 instance types. These experiments processed 64 query files, each containing 100 sequences using 16 compute cores. While we expected the memory size to have a strong correlation to the BLAST performance, due to the querying of a large database, the performance results do not show a significant effect related to the memory size, as High-CPU-Extra-Large (HCXL) instances with less than 1GB of memory per CPU core were able to perform comparatively to Large and Extra-Large instances with 3.75GB per CPU core. However, it should be noted that there exists a slight correlation with memory size, as the lower clock rated Extra-Large (~2.0Ghz) instances with more memory per core performed similarly to the HCXL (~2.5Ghz) instances. The High-Memory-Quadruple-Large (HM4XL)

instances (~3.25Ghz) have a higher clock rate, which partially explains the faster processing time. Once again, the EC2 HCXL instances gave the most cost-effective performance, thus offsetting the performance advantages demonstrated by other instance types.

Figure 11 presents the benchmark results for BLAST Classic-Cloud application on different Azure instance types. These experiments processed 8 query files, each containing 100 sequences using 8 small, 4 medium, 2 large and 1 Extra-Large instances respectively. Although the features of Azure instance types scale linearly, the BLAST application performed better with larger total memory sizes. When sufficient memory is available, BLAST can load and reuse the whole BLAST database (~8GB) in to the memory. BLAST application has the ability to parallelize the computations using threads. The horizontal axis of Figure 11 depicts 'Number of workers (processes) per instance' X 'Number of BLAST threads per worker'. The 'N' stands for the number of cores per instance in that particular instance type. According to the results, Azure Large and Extra-Large instances deliver the best performance for BLAST. Using pure BLAST threads to parallelize inside the instances delivered slightly lesser performance than using multiple workers (processes). The costs to process 8 query files are directly proportional to the run time, due to the linear pricing of Azure instance types.

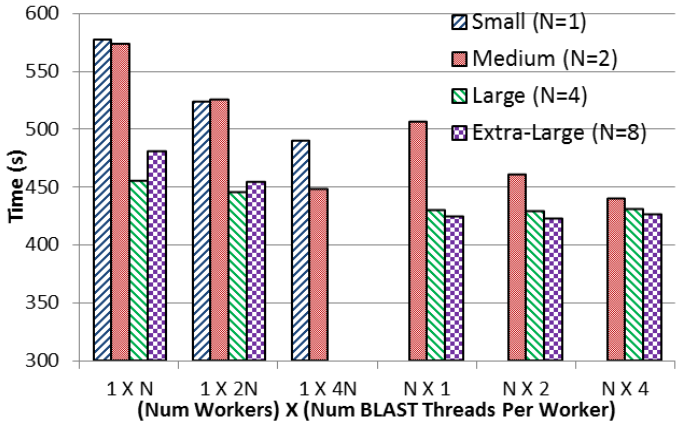


Figure 11 Time to process 8 query files using BLAST application on different Azure instance types

4.4.2 Scalability

For the scalability experiment, we replicated a query data set of 128 files (with 100 sequences in each file), one to six times to create input data sets for the experiments, ensuring the linear scalability of the workload across data sets. Even though the larger data sets are replicated, the base 128-file data set is inhomogeneous. The Hadoop-BLAST tests were performed on an iDataplex cluster, in which each node had two 4-core CPUs (Intel Xeon CPU E5410 2.33GHz) and 16 GB memory and was inter-connected using Gigabit Ethernet. DryadLINQ tests were performed on a Windows HPC cluster with 16 cores (AMD Opteron 2.3 Ghz) and 16 GB of memory per node. 16 High-CPU-Extra-Large instances were used for the EC2 testing and 16 Extra-Large instances were used for the Azure testing.

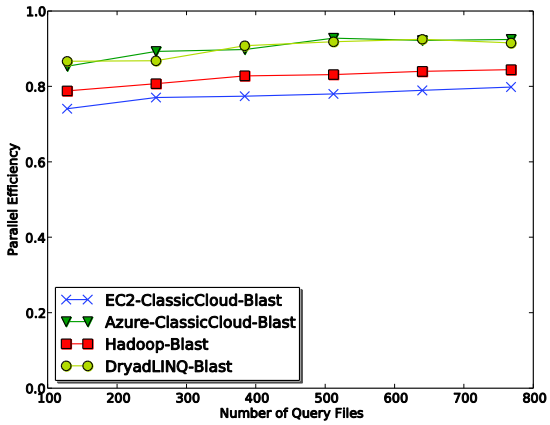


Figure 12 : BLAST parallel efficiency using the pleasingly parallel frameworks

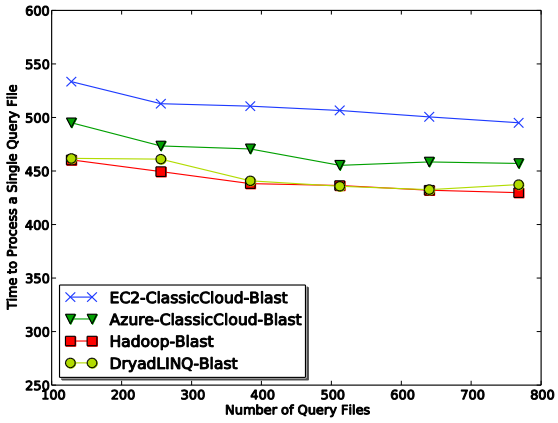


Figure 13 : BLAST average time to process a single query file using the pleasingly parallel frameworks

Figure 12 depicts the absolute parallel efficiency of the distributed BLAST implementations, while Figure 13 depicts the average time to process a single query file in a single core. From those figures, we can conclude that all four implementations exhibit near-linear scalability with comparable performance (within 20% efficiency), while BLAST on Windows environments (Azure and DryadLINQ) exhibit the better overall efficiency. The limited memory of the High-CPU-Extra-Large (HCXL) instances shared across 8 workers performing different BLAST computations may have contributed to the relatively low efficiency of EC2 BLAST implementation. According to figure 8, use of EC2 High-Memory-Quadruple-Extra-Large instances would have given better performance than HCXL instances, but at a much higher cost. The amortized cost to process 768*100 queries using Classic Cloud-BLAST was ~10$ using EC2 and ~12.50$ using Azure.

4.5 GTM Interpolation

We implemented distributed parallel versions of GTM interpolations application described in section 2.3.2 for Amazon EC2, Microsoft Azure, DryadLINQ and for Apache Hadoop using the frameworks that were presented in section 4.1.

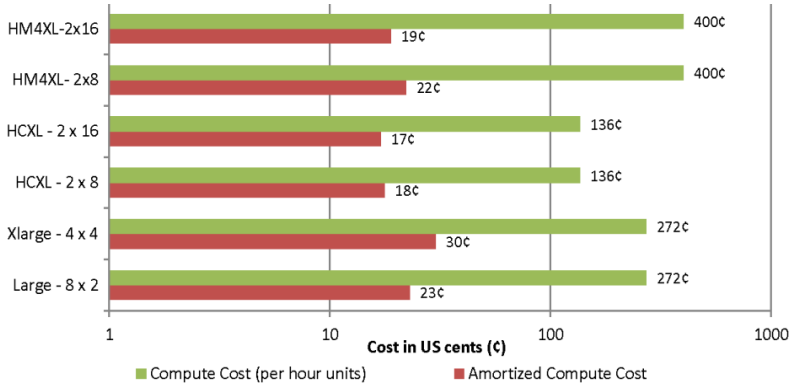


Figure 14 : Cost of using GTM interpolation application with different EC2 instance types

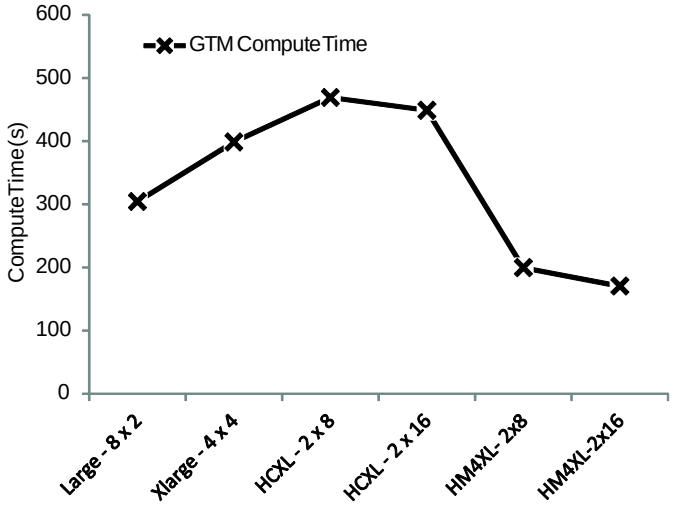


Figure 15 : GTM Interpolation compute time with different EC2 instance types

4.5.1 Application performance with different cloud instance types

According to Figure 15, we can infer that memory (size and bandwidth) is a bottleneck for the GTM Interpolation application. The GTM Interpolation application performs better in the presence of more memory and a smaller number of processor cores sharing the memory. The high memory quadruple Extra-Large instances give the best performance overall, but the High-CPU-Extra-Large instances still appear to be the most economical choice.

4.5.2 GTM Interpolation speedup

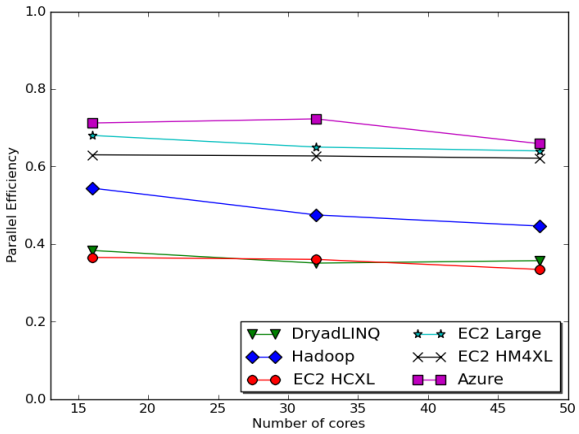


Figure 16: GTM Interpolation parallel efficiency using the pleasingly parallel frameworks

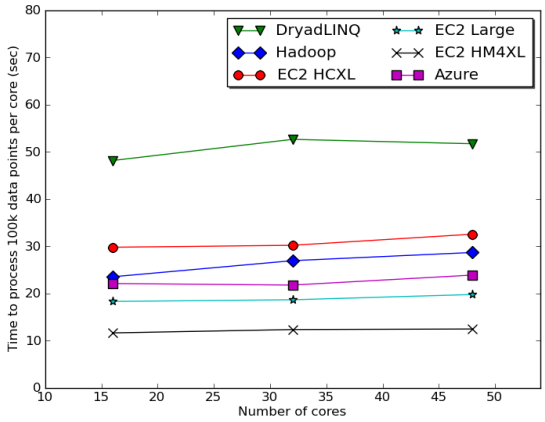


Figure 17 : GTM Interpolation performance per core using the pleasingly parallel frameworks

We used the PubChem data set of 26 million data points with 166 dimensions to analyze the GTM Interpolation applications. PubChem is an NIH funded repository of over 60 million chemical molecules, their chemical structures and their biological activities. A pre-processed subset of 100,000 data points were used as the seed for the GTM Interpolation. We partitioned the input data into 264 files, with each file containing 100,000 data points. Figure 16 and Figure 17 depict the performance of the GTM Interpolation implementations.

DryadLINQ tests were performed on a 16 core (AMD Opteron 2.3 Ghz) per node, 16GB memory per node cluster. Hadoop tests were performed on a 24 core (Intel Xeon 2.4 Ghz) per node, 48 GB memory per node cluster which was configured to use only 8 cores per node. Classic Cloud Azure tests we performed on Azure Small instances (single core). Classic Cloud EC2 tests were performed on EC2 Large, High-CPU-Extra-Large (HCXL) as well as on High-Memory-Quadruple-Extra-Large (HM4XL) instances separately. HM4XL and HCXL instances were considered 8 cores per instance while 'Large' instances were considered 2 cores per instance.

Characteristics of the GTM Interpolation application are different from the Cap3 application as GTM is more memory-intensive and the memory bandwidth becomes the bottleneck, which we assume to be the cause of the lower efficiency numbers. Among the different EC2 instances, Large instances achieved

the best parallel efficiency and High-Memory-Quadruple-Extra-Large instances gave the best performance while High-CPU-Extra-Large instances were the most economical. Azure small instances achieved the overall best efficiency. The efficiency numbers highlight the memory-bound nature of the GTM Interpolation computation, while platforms with less memory contention (fewer CPU cores sharing a single memory) performed better. As noted, the DryadLINQ GTM Interpolation efficiency is lower than the others. One reason for the lower efficiency would be the usage of 16 core machines for the computation, which puts more contention on the memory.

The computational tasks of GTM Interpolation applications were much finer grain than those in the Cap3 or BLAST applications. Compressed data splits, which were unzipped before handing over to the executable, were used due to the large size of the input data. When the input data size is larger, Hadoop and DryadLINQ applications have an advantage of data locality-based scheduling over EC2. The Hadoop and DryadLINQ models bring computation to the data optimizing the I/O load, while the Classic Cloud model brings data to the computations.

4.6 Summary

We have demonstrated the feasibility of Cloud infrastructures for three loosely-coupled scientific computation applications by implementing them using cloud infrastructure services as well as cloud-oriented programming models, such as Hadoop MapReduce and DryadLINQ.

Cloud infrastructure services provide users with scalable, highly-available alternatives to their traditional counterparts, but without the burden of managing them. While the use of high latency, eventually consistent cloud services together with off-instance cloud storage has the potential to cause significant overheads, our work in this paper has shown that it is possible to build efficient, low overhead applications utilizing them. Given sufficiently coarser grain task decompositions, Cloud

infrastructure service-based frameworks as well as the MapReduce-based frameworks offered good parallel efficiencies in almost all of the cases we considered. Computing Clouds offer different instance types at different price points. We showed that selecting an instance type that is best suited to the user's specific application can lead to significant time and monetary advantages.

While models like Classic Cloud bring in operational and quality of services advantages, it should be noted that the simpler programming models of existing cloud-oriented frameworks like MapReduce and DryadLINQ are more convenient for the users. Motivated by the positive results presented in this paper, we are working on developing a fully-fledged MapReduce framework with iterative-MapReduce support for the Windows Azure Cloud infrastructure using Azure infrastructure services as building blocks, which will provide users the best of both worlds. The cost effectiveness of cloud data centers, combined with the comparable performance reported here, suggests that loosely-coupled science applications will be increasingly implemented on clouds and that using MapReduce frameworks will offer convenient user interfaces with little overhead.

5. MAPREDUCE TYPE APPLICATIONS ON CLOUD ENVIRONMENTS

The MapReduce distributed data analysis framework model introduced by Google [1] provides an easy-to-use programming model that features fault tolerance, automatic parallelization, scalability and data locality-based optimizations. Due to their excellent fault tolerance features, MapReduce frameworks are well-suited for the execution of large distributed jobs in brittle environments such as commodity clusters and cloud infrastructures. Though introduced by the industry and used mainly in the information retrieval community, it is shown [2-4] that MapReduce frameworks are capable of supporting many scientific application use cases, making these frameworks good choices for scientists to easily build large, data-intensive applications that need to be executed within cloud infrastructures.

The lack of a distributed computing framework on Azure platform at the time (circa 2010) motivated us to implement MRRoles4Azure (MapReduce Roles for Azure), which is a decentralized novel MapReduce run time built using Azure cloud infrastructure services. MRRoles4Azure implementation takes advantage of the scalability, high availability and the distributed nature of cloud infrastructure services, guaranteed by cloud service provider, to deliver a fault tolerant, dynamically scalable runtime with a familiar programming model for the users.

Several options exist for executing MapReduce jobs on cloud environments, such as manually setting up a MapReduce (e.g.: Hadoop [15]) cluster on a leased set of computing instances, using an on-demand MapReduce-as-service offering such as Amazon ElasticMapReduce (EMR) [48] or using a cloud MapReduce runtime such as MRRoles4Azure or CloudMapReduce [49]. In this chapter we explore and evaluate each of these different options for two well-known bioinformatics applications: Smith-Waterman GOTOH pairwise distance alignment (SWG) [22, 23], Cap3 [50] sequence assembly. We have performed experiments to gain insight about the performance of MapReduce in the clouds for the

selected applications and compare its performance to MapReduce on traditional clusters. For this study, we use an experimental version of MRRoles4Azure.

Our work was motivated by an experience we had in early 2010 in which we evaluated the use of Amazon EMR for our scientific applications. To our surprise, we observed subpar performance in EMR compared to using a manually-built cluster on EC2 (which is not the case anymore), which prompted us to perform the current analyses. In this paper, we show that MapReduce computations performed in cloud environments, including MRRoles4Azure, has the ability to perform comparably to MapReduce computations on dedicated private clusters.

5.1 Challenges for MapReduce in the Clouds

As mentioned above, MapReduce frameworks perform much better in brittle environments than other tightly coupled distributed programming frameworks, such as MPI [51], due to their excellent fault tolerance capabilities. However, cloud environments provide several challenges for MapReduce frameworks to harness the best performance.

- Data storage: Clouds typically provide a variety of storage options, such as off-instance cloud storage (e.g.: Amazon S3), mountable off-instance block storage (e.g.: Amazon EBS) as well as virtualized instance storage (persistent for the lifetime of the instance), which can be used to set up a file system similar to HDFS [30]. The choice of the storage best-suited to the particular MapReduce deployment plays a crucial role as the performance of data intensive applications rely a lot on the storage location and on the storage bandwidth.
- Metadata storage: MapReduce frameworks need to maintain metadata information to manage the jobs as well as the infrastructure. This metadata needs to be stored reliability

ensuring good scalability and the accessibility to avoid single point of failures and performance bottlenecks to the MapReduce computation.

- Communication consistency and scalability: Cloud infrastructures are known to exhibit inter-node I/O performance fluctuations (due to shared network, unknown topology), which affect the intermediate data transfer performance of MapReduce applications.
- Performance consistency (sustained performance): Clouds are implemented as shared infrastructures operating using virtual machines. It's possible for the performance to fluctuate based the load of the underlying infrastructure services as well as based on the load from other users on the shared physical node which hosts the virtual machine (see Section 7.3).
- Reliability (Node failures): Node failures are to be expected whenever large numbers of nodes are utilized for computations. But they become more prevalent when virtual instances are running on top of non-dedicated hardware. While MapReduce frameworks can recover jobs from worker node failures, master node (nodes which store meta-data, which handle job scheduling queue, etc) failures can become disastrous.
- Choosing a suitable instance type: Clouds offer users several types of instance options, with different configurations and price points (See Table 1 and Table 2). It's important to select the best matching instance type, both in terms of performance as well as monetary wise, for a particular MapReduce job.
- Logging: Cloud instance storage is preserved only for the lifetime of the instance. Hence, information logged to the instance storage would be lost after the instance termination. This can be crucial if one needs to process the logs afterwards, for an example to identify a software-caused instance failure. On the other hand, performing excessive logging to a

bandwidth limited off-instance storage location can become a performance bottleneck for the MapReduce computation.

5.2 MRRoles4Azure (MapReduce Roles for Azure)

MRRoles4Azure is a distributed decentralized MapReduce runtime for Windows Azure that was developed using Azure cloud infrastructure services. The usage of the cloud infrastructure services allows the MRRoles4Azure implementation to take advantage of the scalability, high availability and the distributed nature of such services guaranteed by the cloud service providers to avoid single point of failures, bandwidth bottlenecks (network as well as storage bottlenecks) and management overheads. In this paper, we use an experimental, pre-release version of MRRoles4Azure.

The usage of cloud services usually introduces latencies larger than their optimized non-cloud counterparts and often does not guarantee the time for the data's first availability. These overheads can be conquered, however, by using a sufficiently coarser grained map and reduce tasks. MRRoles4Azure overcomes the availability issues by retrying and by designing the system so it does not rely on the immediate availability of data to all the workers. In this paper, we use the pre-release implementation of the MRRoles4Azure runtime, which uses Azure Queues for map and reduce task scheduling, Azure tables for metadata & monitoring data storage, Azure blob storage for input, output and intermediate data storage and the Window Azure Compute worker roles to perform the computations.

Google MapReduce [1], Hadoop [15] as well as Twister [7] MapReduce computations are centrally controlled using a master node and assume master node failures to be rare. In those run times, the master node handles the task assignment, fault tolerance and monitoring for the completion of Map and Reduce tasks, in addition to other responsibilities. By design, cloud environments are more brittle than the traditional computing clusters are. Thus, cloud applications should be developed to anticipate

and withstand these failures. Because of this, it is not possible for MRRoles4Azure to make the same assumptions of reliability about a master node as in the above-mentioned runtimes. Due to these reasons, MRRoles4Azure is designed around a decentralized control model without a master node, thus avoiding the possible single point of failure. MRRoles4Azure also provides users with the capability to dynamically scale up or down the number of computing instances, even in the middle of a MapReduce computation, as and when it is needed. The map and reduce tasks of the MRRoles4Azure runtime are dynamically scheduled using a global queue. In a previous study [52], we experimentally showed that dynamic scheduling through a global queue achieves better load balancing across all tasks, resulting in better performance and throughput than statically scheduled runtimes, especially when used with real-world inhomogeneous data distributions.

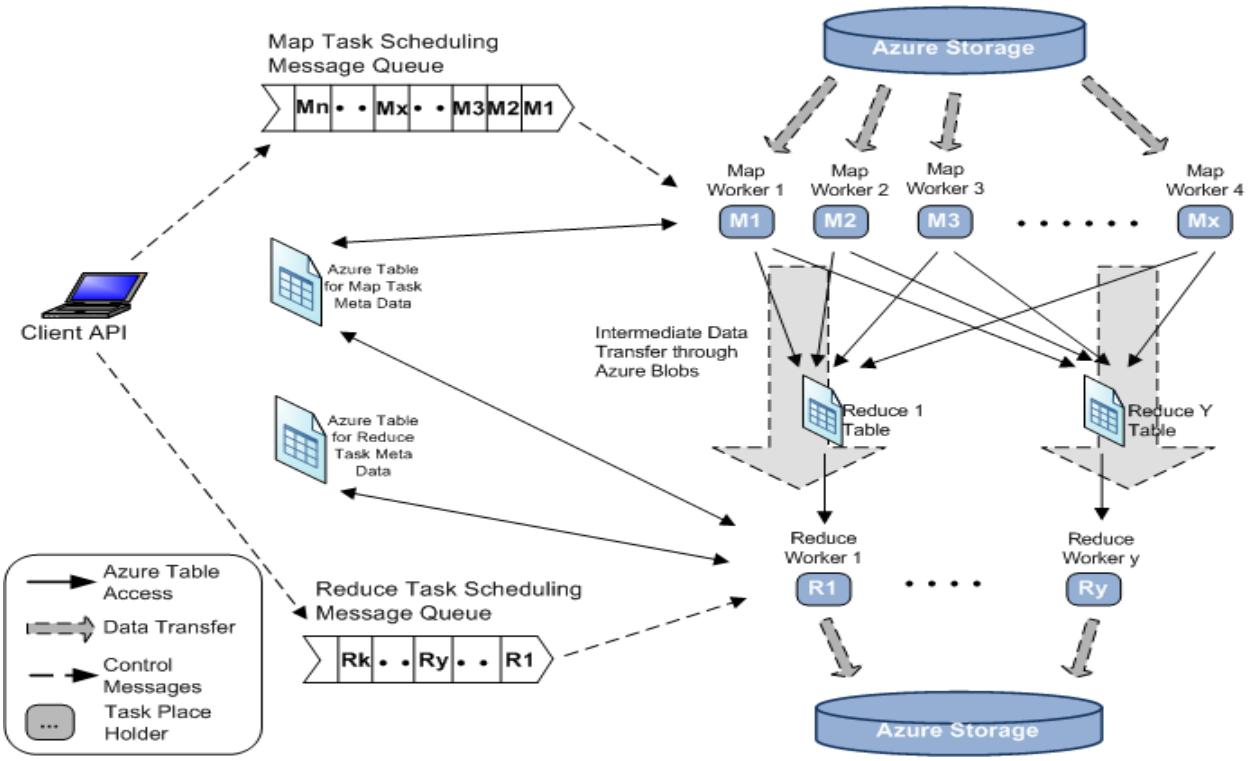


Figure 18 MapReduceRoles4Azure: Architecture for implementing MapReduce frameworks on Cloud environments using cloud infrastructure services

5.2.1 Client API and Driver

Client driver is used to submit the Map and Reduce tasks to the worker nodes using Azure Queues. Users can utilize the client API to generate a set of map tasks that are either automatically based on a data set present in the Azure Blob storage or manually based on custom criteria, which we find to be a very useful feature when implementing science applications using MapReduce. Client driver uses the .net task parallel library to dispatch tasks in parallel overcoming the latencies of the Azure queue and the Azure table services. It's possible to use the client driver to monitor the progress and completion of the MapReduce jobs.

5.2.2 Map Tasks

Users have the ability to configure the number of Map workers per Azure instance. Map workers running on the Azure compute instances poll and dequeue map task scheduling messages from the scheduling queue, which were enqueued by the client API. The scheduling messages contain the meta-data needed for the Map task execution, such as input data file location, program parameters, map task ID, and so forth. Map tasks upload the generated intermediate data to the Azure Blob Storage and put the key-value pair meta-data information to the correct reduce task table. At this time, we are actively working on investigating other approaches for performing the intermediate data transfer.

5.2.3 Reduce Tasks

Reduce task scheduling is similar to map task scheduling. Users have the ability to configure the number of Reduce tasks per Azure Compute instance. Each reduce task has an associated Azure Table containing the input key-value pair meta-data information generated by the map tasks. Reduce tasks fetch intermediate data from the Azure Blob storage based on the information present in the above-mentioned reduce task table. This data transfer begins as soon as the first Map task is completed, overlapping the data transfer with the computation. This overlapping of data transfer with computation

minimizes the data transfer overhead of the MapReduce computations, as found in our testing. Each Reduce task starts processing the reduce phase; when all the map tasks are completed, and after all the intermediate data products bound for that particular reduce task is fetched. In the MRRoles4Azure, each reduce task will independently determine the completion of map tasks based on the information in the map task meta-data table and in the reduce task meta-data table. After completion of the processing, reduce tasks upload the results to the Azure Blob Storage and update status in the reduce task meta-data table.

Azure table does not support transactions across tables or guarantee the immediate availability of data, but rather guarantees the eventual availability data. Due to that, it is possible for a worker to notice a map task completion update, before seeing a reduce task intermediate meta-data record added by that particular map task. Even though rare, this can result in an inconsistent state where a reduce task decides all the map tasks have been completed and all the intermediate data bound for that task have been transferred successfully, while in reality it's missing some intermediate data items. In order to remedy this, map tasks store the number of intermediate data products it generated in the map task meta-data table while doing the task completion status update. Before proceeding with the execution, reduce tasks perform a global count of intermediate data products in all reduce task tables and tally it with the total of intermediate data products generated by the map tasks. This process ensures all the intermediate data products are transferred before starting the reduce task processing.

5.2.4 Monitoring

We use Azure tables for the monitoring of the map and reduce task meta-data and status information. Each job has two separate Azure tables for map and reduce tasks. Both the meta-data tables are used by the reduce tasks to determine the completion of Map task phase. Other than the

above two tables, it is possible to monitor the intermediate data transfer progress using the tables for each reduce task.

5.2.5 Fault Tolerance

Fault tolerance is achieved using the fault tolerance features of the Azure queue. When fetching a message from an Azure queue, a visibility timeout can be specified, which will keep the message hidden until the timeout expires. In Azure Map Reduce, map and reduce tasks delete messages from the queue only after successful completion of the tasks. If a task fails or is too slow processing, then the message will reappear in the queue after the timeout. In this case, it would be fetched and re-executed by a different worker. This is made possible by the side effect-free nature of the MapReduce computations as well as the fact that MRRoles4Azure stores each generated data product in persistent storage, which allows it to ignore the data communication failures. In the current implementation, we retry each task three times before declaring the job a failure. We use the Map & Reduce task meta-data tables to coordinate the task status and completion. Over the course of our testing, we were able to witness few instances of jobs being recovered by the fault tolerance.

5.2.6 Limitations of Azure MapReduce

Currently Azure allows a maximum of 2 hours for queue message timeout, which is not enough for Reduce tasks of larger MapReduce jobs, as the Reduce tasks typically execute from the beginning of the job till the end of the job. In our current experiments, we disabled the reduce tasks fault tolerance when it is probable for MapReduce job to execute for more than 2 hours. Also in contrast to Amazon Simple Queue Service, Azure Queue service currently doesn't allow for dynamic changes of visibility timeouts, which would allow for richer fault tolerance patterns.

5.3 Performance evaluation

5.3.1 Methodology

We performed scalability tests using the selected applications to evaluate the performance of the MapReduce implementations in the cloud environments, as well as in the local clusters. For the scalability test, we decided to increase the workload and the number of nodes proportionally (weak scaling), so that the workload per node remained constant.

All of the MRRoles4Azure tests were performed using Azure small instances (one CPU core). The Hadoop-Bare Metal tests were performed on an iDataplex cluster, in which each node had two 4-core CPUs (Intel Xeon CPU E5410 2.33GHz) and 16 GB memory, and was inter-connected using Gigabit Ethernet network interface. The Hadoop-EC2 and EMR tests for Cap3 application were performed using Amazon High CPU extra-large instances, as they are the most economical per CPU core. Each high CPU extra-large instance was considered as 8 physical cores, even though they are billed as 20 Amazon compute units. The EC2 and EMR tests for SWG MapReduce applications were performed using Amazon extra-large instances as the more economical high CPU extra instances showed memory limitations for the SWG calculations. Each extra-large instance was considered as 4 physical cores, even though they are billed as 8 Amazon computing units. In all the Hadoop-based experiments (EC2, EMR and Hadoop bare metal), only the cores of the Hadoop slave nodes were considered for the number of cores calculation, despite the fact that an extra computing node was used as the Hadoop master node.

Below are the defined parallel efficiency($\mathrm{ParallelEfficiency} = \frac{\mathrm{T_1}}{\mathrm{pTp}}$ --- Equation 1) and relative parallel efficiency calculations used in the results part of this paper.

$$Parallel\ Efficiency\ (Ep) = \frac{\mathrm{T}(1)}{\mathrm{pT}(\rho)}$$

T(1) is the best sequential execution time for the application in a particular environment using the same data set or a representative subset. In all the cases, the sequential time was measured with no

data transfers, i.e. the input files were present in the local disks. T(ρ) is the parallel run time for the application while "p" is the number of processor cores used.

We calculate that the relative parallel efficiency when estimating the sequential run time for an application is not straightforward. α = *p/p*1, where p1 is the smallest number of CPU cores for the experiment.

$$Relative\ Parallel\ Efficiency\ (Ep) = \frac{T(\rho 1)}{\alpha T(\rho)}$$ --- Equation 3

5.3.2 Smith-Waterman-GOTOH(SWG) pairwise distance calculation

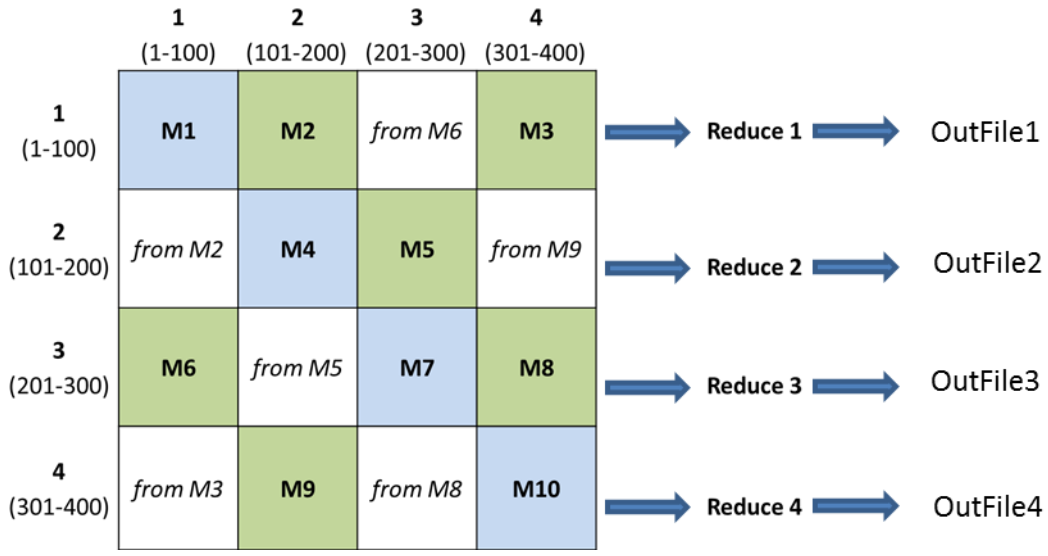


Figure 19 Task decomposition mechanism of SWG pairwise distance calculation MapReduce application

In this application, we use the Smith-Waterman [22] algorithm with GOTOH [23] (SWG) improvement to perform pairwise sequence alignment on FASTA sequences. Given a sequence set we calculate the all-pairs dissimilarity for all the sequences. When calculating the all-pairs dissimilarity for a data set, calculating only the strictly upper or lower triangular matrix in the solution space is sufficient, as the transpose of the computed triangular matrix gives the dissimilarity values for the other triangular matrix. As shown in **Error! Reference source not found.**, this property, together with blocked

ecomposition, is used when calculating the set of map tasks for a given job. 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 are given in [52]. The MRRoles4Azure implementation also follows the same architecture and blocking strategy as in the Hadoop-SWG implementation. Hadoop-SWG uses the open source JAligner [53] as the computational kernel, while MRRoles4Azure SWG uses the C# implementation, NAligner [53] as the computational kernel. The results of the SWG MapReduce computation get stored in HDFS for Hadoop-SWG in bare metal and EC2 environments, while the results get stored in Amazon S3 and Azure Block Storage for Hadoop-SWG on EMR and SWG on MRRoles4Azure, respectively.

Due to the all-pairs nature and the block-based task decomposition of the SWG MapReduce implementations, it's hard to increase the workload linearly by simply replicating the number of input sequences for the scalability test. Hence, we modified the program to artificially reuse the computational blocks of the smallest test case in the larger test cases, so that the workload scaling occurs linearly. The raw performance results of the SWG MapReduce scalability test are given in Figure 20(a). A block size of 200 * 200 sequences is used in the performance experiments resulting in 40,000 sequence alignments per block, which resulted in ~123 million sequence comparisons in the 3072 block test case. The MRRoles4Azure SWG performance in Figure 20(a) is significantly lesser than the others. This is due to the performance of NAligner core executing in windows environment being slower than the JAligner core executing in Linux environment.

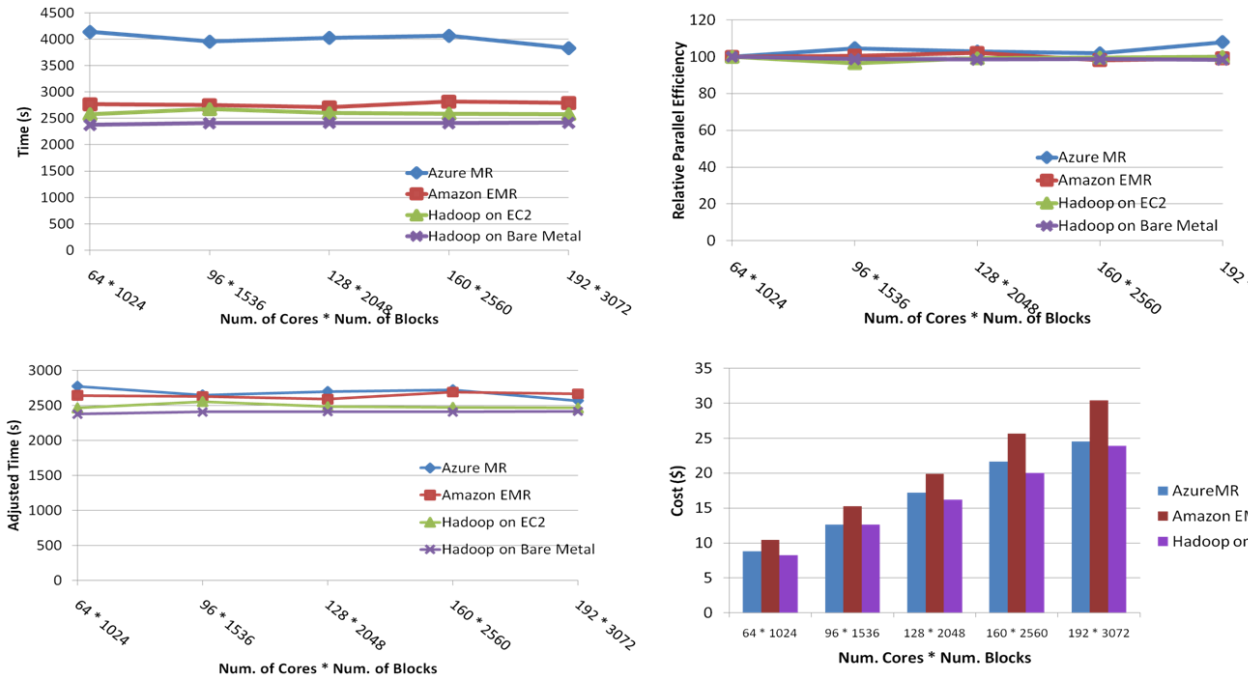


Figure 20 (a) SWG MapReduce pure performance (b) SWG MapReduce relative parallel efficiency (c) SWG

Due to the sheer size of even the smallest computation in our SWG scaling test cases, we found it impossible to calculate the sequential execution time for the SWG test cases. Also, due to the all-pairs nature of SWG, it's not possible to calculate the sequential execution time using a subset of data. In order to compensate for the lack of absolute efficiency (which would have negated most of the platform and hardware differences across different environments), we performed a moderately-sized sequential SWG calculation in all of the environments and used that result to normalize the performance using the Hadoop-bare metal performance as the baseline. The normalized performance is depicted in figure 2(c), where we can observe that all four environments show comparable performance and good scalability for the SWG application. Figure 20(b) depicts the relative parallel efficiency of SWG MapReduce implementations using the 64 core, 1024 block test case as *p1* (see section V-A).

In Figure 20(d) we present the approximate computational costs for the experiments performed using cloud infrastructures. Even though the cloud instances are hourly billed, costs presented in 2(d) are amortized for the actual execution time (time / 3600 * num_instances * instance price per hour),

assuming the remaining time of the instance hour has been put to useful work. In addition to the depicted charges, there will be additional minor charges for the data storage for EMR & MRRoles4Azure. There will also be additional minor charges for the queue service and table service for MRRoles4Azure. We notice that the costs for Hadoop on EC2 and MRRoles4Azure are in a similar range, while EMR costs a fraction more. We consider the ability to perform a large computation, such as ~123 million sequence alignments, for under 30$ with zero up front hardware cost, as a great enabler for the scientists, who don't have access to in house compute clusters.

5.3.3 Sequence assembly using Cap3

Cap3 [50] is a sequence assembly program which assembles DNA sequences by aligning and merging sequence fragments to construct whole genome sequences. More details about the Cap3 are given in section 2.3.1 and more details about Cap3 Hadoop implementation are given in section 4.3.

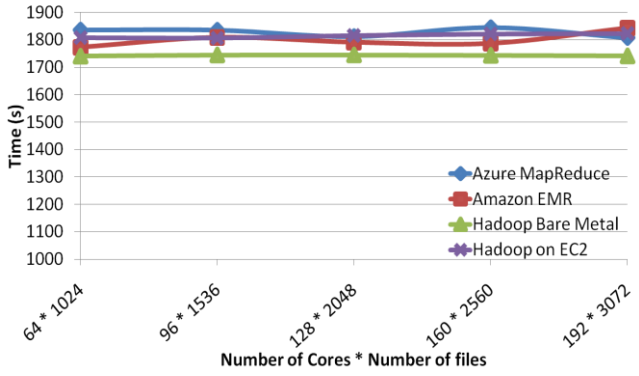


Figure 21 Cap3 MapReduce scaling performance

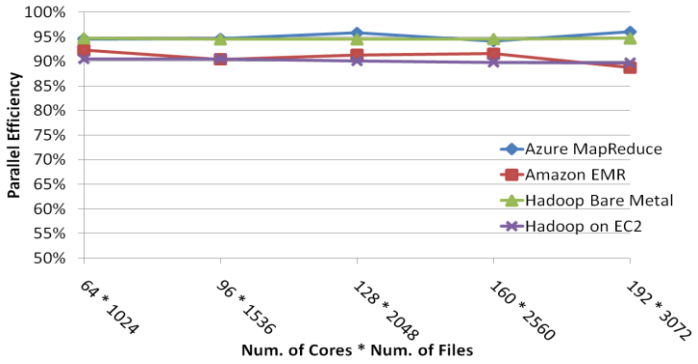


Figure 22 Cap3 MapReduce parallel efficiency

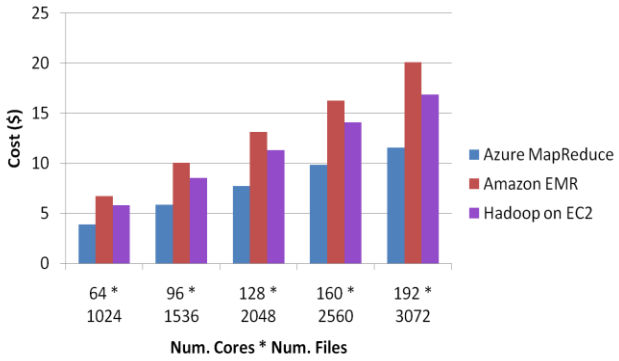


Figure 23 Cap3 MapReduce computational cost in cloud infrastructures

We used a replicated set of Fasta files as the input data in our experiments. Every file contained 458 reads. The input/output data was stored in HDFS in the Hadoop BareMetal and Hadoop-EC2 experiments, while they were stored in Amazon S3 and Azure Blob storage for EMR and MRRoles4Azure experiments respectively. Figure 21 presents the pure performance of the Cap3 MapReduce applications, while Figure 22 presents the absolute parallel efficiency for the Cap3 MapReduce applications. As we can see, all of the cloud Cap3 applications displayed performance comparative to the bare metal clusters and good scalability, while MRRoles4Azure and Hadoop Bare metal showed a slight edge over the Amazon counterparts in terms of the efficiency. Figure 23 depicts the approximate

amortized computing cost for the Cloud MapReduce applications, with MRRoles4Azure showing an advantage.

5.4 Summary

We introduced the novel decentralized controlled MRRoles4Azure framework, which fulfills the much-needed requirement of a distributed programming framework for Azure users. MRRoles4Azure is built using Azure cloud infrastructure services that take advantage of the quality of service guarantees provided by the cloud service providers. Even though cloud services have higher latencies than their traditional counter parts, scientific applications implemented using MRRoles4Azure were able to perform comparatively with the other MapReduce implementations, thus proving the feasibility of MRRoles4Azure architecture. We also explored the challenges presented by cloud environments to execute MapReduce computations and discussed how we overcame them in the MRRoles4Azure architecture.

We also presented and analyzed the performance of two scientific MapReduce applications on two popular cloud infrastructures. In our experiments, scientific MapReduce applications executed in the cloud infrastructures exhibited performance and efficiency comparable to the MapReduce applications executed using traditional clusters. Performance comparable to in house clusters, on demand availability, horizontal scalability and the easy to use programming model together with no upfront cost makes using MapReduce in cloud environments a very viable option and an enabler for the computational scientists, especially in scenarios where in-house compute clusters are not readily available. From an economical and maintenance perspective, it even makes sense not to procure in-house clusters if the utilization would be low.

6. DATA INTENSIVE ITERATIVE COMPUTATIONS ON CLOUD ENVIRONMENTS

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 & 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 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. 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 **Error! Reference source not found.** 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 SmithWatermann-GOTOH (SWG)[2] pairwise sequence alignment and Word Count as traditional MapReduce type applications. Section **Error! Reference source not found.** analyzes the SWG wister4Azure application.

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 6.2.2 and 6.2.3 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.

6.1 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 employing a decentralized control architecture that avoids single point failures.

6.1.1 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.

6.1.1.1 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 **Error! Reference source not found.**We can identify two main types of data in these computations, the

oop 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 6.2.3). 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.

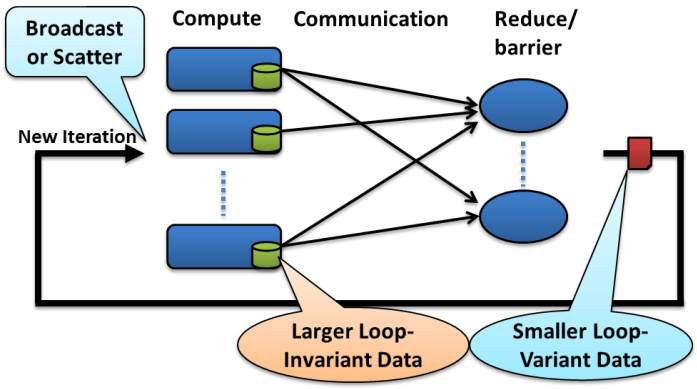


Figure 24 Structure of a typical data-intensive iterative application

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
```

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 (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 25 presents the flow of the Twister4Azure programming model.

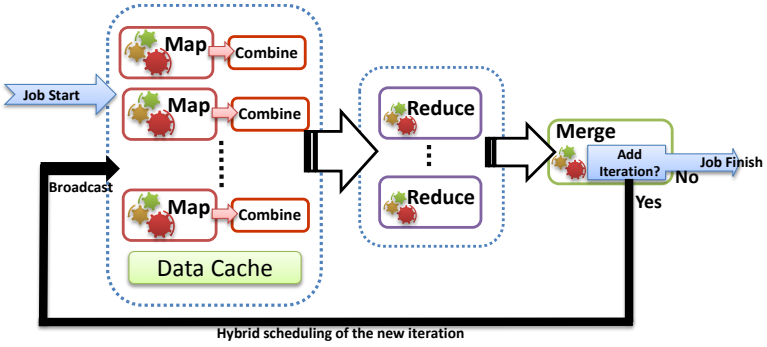


Figure 25 Twister4Azure iterative MapReduce programming model

Code 2 Data-intensive iterative computation using Twister4Azure programming model

```
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
    end while
```

6.1.1.2

6.1.1.3 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, Howe, 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 & Reduce tasks.

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

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

6.1.1.4 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>)

6.1.2 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.

6.1.3 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). 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, 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 **Error! Reference source ot found.**, Twister4Azure schedules new MapReduce jobs (non-iterative and the first iteration of the iterative) through Azure queues. 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.

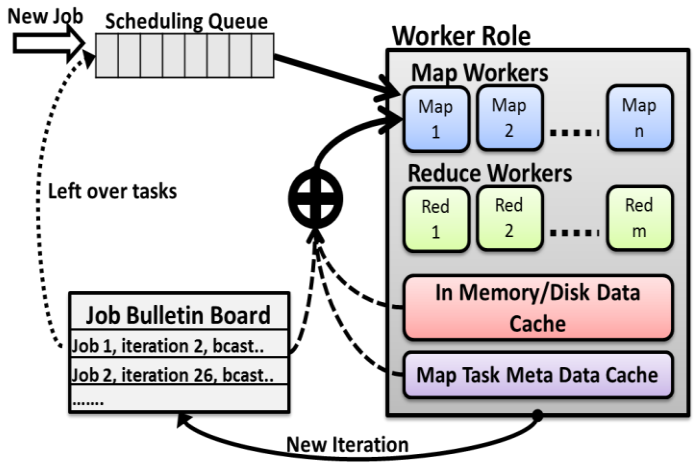


Figure 26 Cache Aware Hybrid Scheduling

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
```

6.1.4 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 number of instances and does not scale well for large number 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 **Error! Reference source not found.**This method supports a configurable umber 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 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.

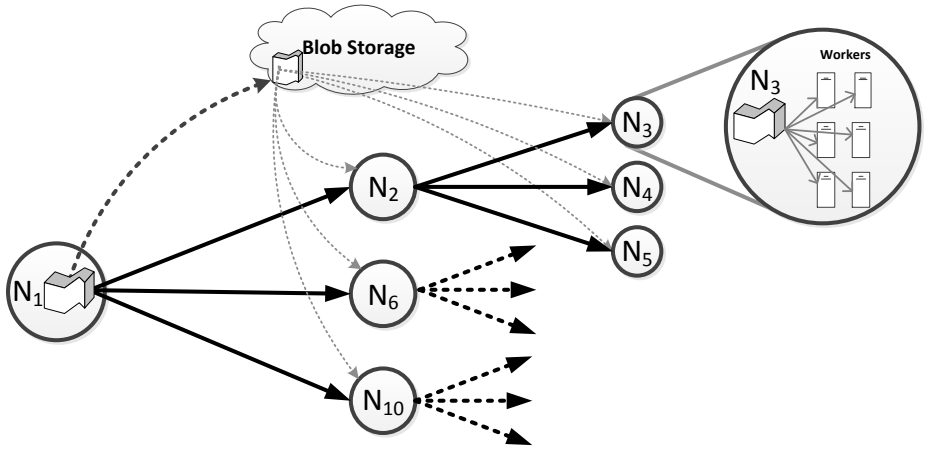


Figure 27 Tree based broadcast over TCP with Blob storage as the persistent backup. N¬3 shows the utilization of data cache to share the broadcast data within an instance.

6.1.5 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 persisted 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 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 optimized this method by utilizing the Azure tables to transfer smaller data products up to a certain size (currently 64kb 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 persisted 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 in 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.

6.1.6 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 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 tolerant, as opposed to the current time out based fault tolerance implementation.

6.1.7 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.

6.1.8 Development and current status

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 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 was made available in mid-2011.

In August 2012, we open sourced Twister4Azure under Apache License 2.0 at http://twister4azure.codeplex.com/. We performed Twister4Azure 0.9 release on September 2012 as the first open source release. Currently all the developments of Twister4Azure are performed in an open source manner.

6.2 Twister4Azure Scientific Application Case Studies

6.2.1 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 instances 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.

Table 6 Evaluation cluster configurations

Cluster/ Instance Type	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
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

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 Cluster(t_{sc}) environment used for Twister and Hadoop. We estimate the Twister4Azure "adjusted performance" for an application using t_a x (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.

6.2.2 Multi-Dimensional Scaling - Iterative MapReduce

We implemented the SMACOF algorithm for Multi-Dimensional Scaling (MDS) described in section 2.3.6 using Twister4Azure. 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. MDS application results in iterating a chain of three MapReduce jobs, as depicted in **Error! Reference source not found.**. For the purposes of this paper, we perform an unweighted mapping hat results in two MapReduce jobs steps per iteration, BCCalc and StressCalc. MDS is challenging for Twister4Azure due to its relatively finer grained task sizes and multiple MapReduce applications per iteration.

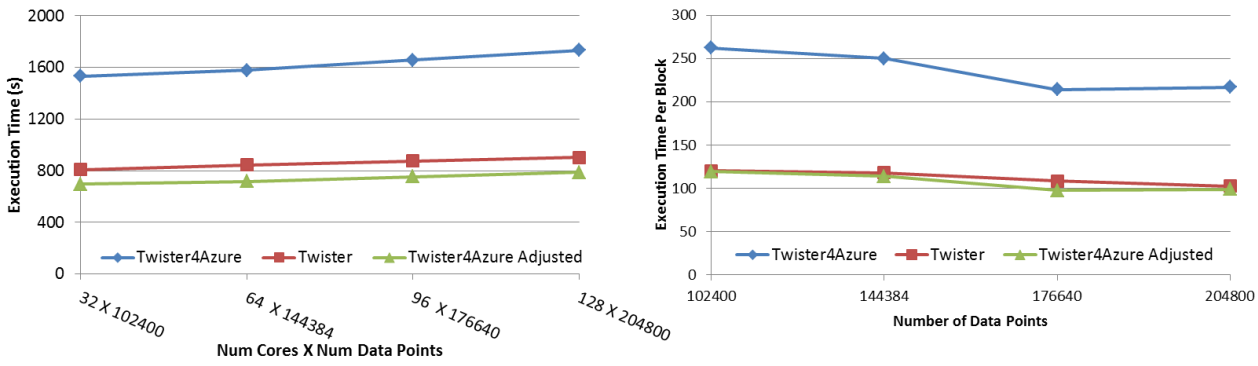


Figure 28 Left (a): MDS weak scaling where we the workload per core is constant. Ideal is a straight horizontal line. Right (b):Data size scaling using 128 Azure small instances/cores, 20 iterations

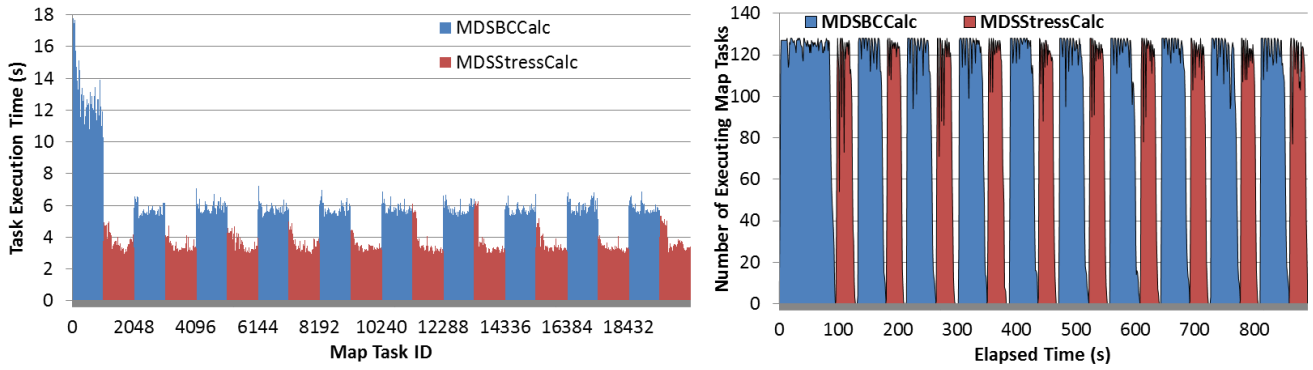


Figure 29 Twister4Azure Map Task histograms for MDS of 204800 data points on 32 Azure Large Instances (graphed only 10 iterations out of 20). Left (a): Map task execution time histogram. Two adjoining bars (blue and red) represent an iteration (2048 ta

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. Figure 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 ~ number of cores). We notice that Twister4Azure exhibits encouraging performance and scales similar to the Java HPC Twister. Figure 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 Figure 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 10240 to 40960, proving Twister4Azure's ability to support large number of tasks effectively.

Figure 9(a) depicts the execution time of individual Map tasks for 10 iterations of Multi-Dimensional Scaling of 204800 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. Figure 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 red and subsequent blue bars) 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 blue and subsequent red bars) of an iteration are almost non-noticeable in this computation.

6.2.3 KMeans Clustering

The K-Means Clustering[20] algorithm described in section 2.3.5 has been widely used in many scientific and industrial application areas due to its simplicity and applicability to large data sets. In this section we compare the performance of Twister4Azure, Hadoop and Twister KMeansClustering iterative MapReduce implementations.

Figure 11(a) 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 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.

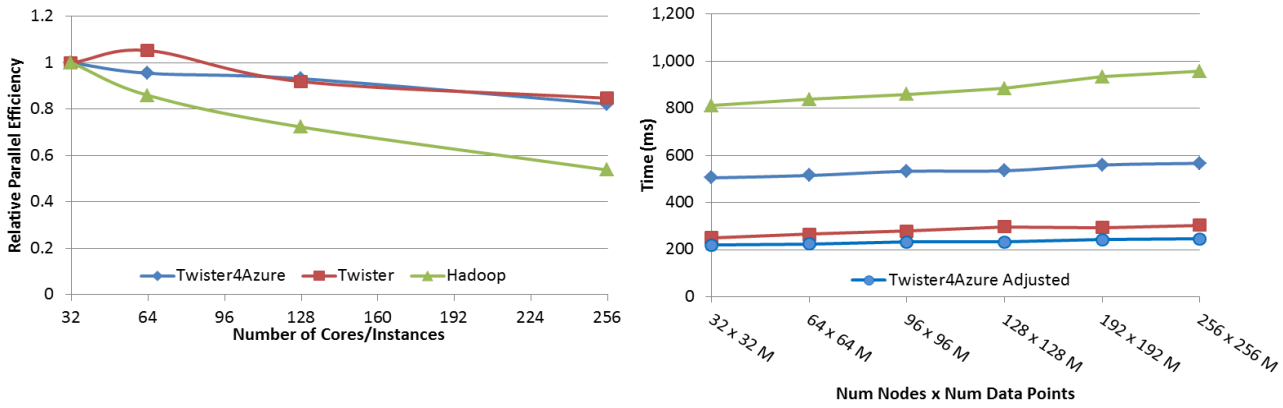


Figure 30 KMeans Clustering Scalability. Left (a): Relative parallel efficiency of strong scaling using 128 million data points.

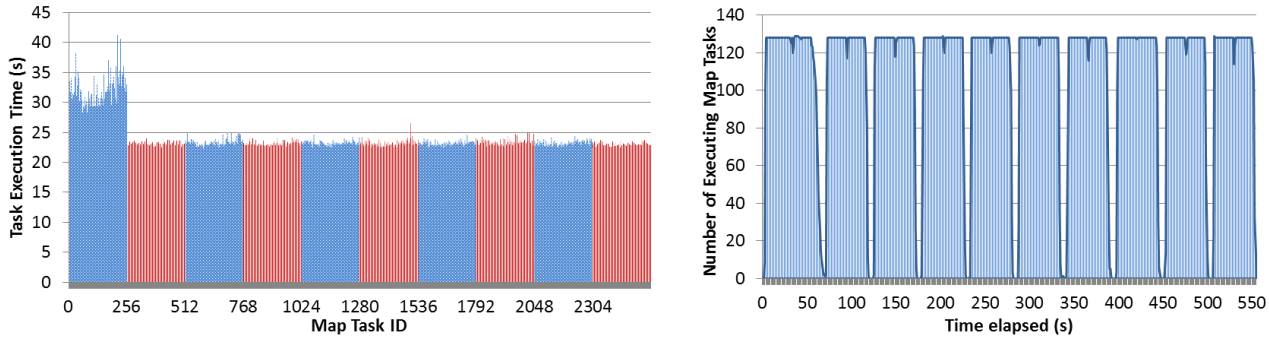


Figure 31 Twister4Azure Map Task histograms for KMeans Clustering 128 million data points on 128 Azure small instances. Left (a): Map task execution time histogram Right (b): Number of executing Map Tasks in the cluster at a given moment

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. Figure 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. Figure 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 ~ 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 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 the same, irrespective of the instance type.

The Twister4Azure adjusted line in Figure 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.

6.3 Summary

We presented Twister4Azure, a novel iterative MapReduce distributed computing runtime for Windows Azure Cloud. Twiser4Azure 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 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 to 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.

7. Performance Implications for Data Intensive Parallel applications on Cloud Environments

7.1 Inhomogeneous data

New generation parallel data processing frameworks such as Hadoop and DryadLINQ are designed to perform optimally when a given job can be divided in to a set of equally time consuming sub tasks. Most of the data sets we encounter in the real world however are inhomogeneous in nature, making it hard for the data analyzing programs to efficiently break down the problems in to equal sub tasks. At the same time we noticed Hadoop & DryadLINQ exhibit different performance behaviors for some of our real data sets. It should be noted that Hadoop & Dryad uses different task scheduling techniques, where Hadoop uses global queue based scheduling and Dryad uses static scheduling. These observations motivated us to study the effects of data inhomogeneity in the applications implemented using these frameworks.

7.1.1 SW-G Pairwise Distance Calculation

The inhomogeneity of data applies for the gene sequence sets too, where individual sequence lengths and the contents vary among each other. In this section we study the effect of inhomogeneous gene sequence lengths for the performance of our pairwise distance calculation applications.

$$SWG(A,B) = O(mn)$$

The time complexity to align and obtain distances for two genome sequences A, B with lengths m and n respectively using Smith-Waterman-Gotoh algorithm is approximately proportional to the product of the lengths of two sequences (O(mn)). All the above described distributed implementations of Smith-

Waterman similarity calculation mechanisms rely on block decomposition to break down the larger problem space in to sub-problems that can be solved using the distributed components. Each block is assigned two sub-sets of sequences, where Smith-Waterman pairwise distance similarity calculation needs to be performed for all the possible sequence pairs among the two sub sets. According to the above mentioned time complexity of the Smith-Waterman kernel used by these distributed components, the execution time for a particular execution block depends on the lengths of the sequences assigned to the particular block.

Parallel execution frameworks like Dryad and Hadoop work optimally when the work is equally partitioned among the tasks. Depending on the scheduling strategy of the framework, blocks with different execution times can have an adverse effect on the performance of the applications, unless proper load balancing measures have been taken in the task partitioning steps. For an example, in Dryad vertices are scheduled at the node level, making it possible for a node to have blocks with varying execution times. In this case if a single block inside a vertex takes a longer amount of time than other blocks to execute, then the entire node must wait until the large task completes, which utilizes only a fraction of the node resources.

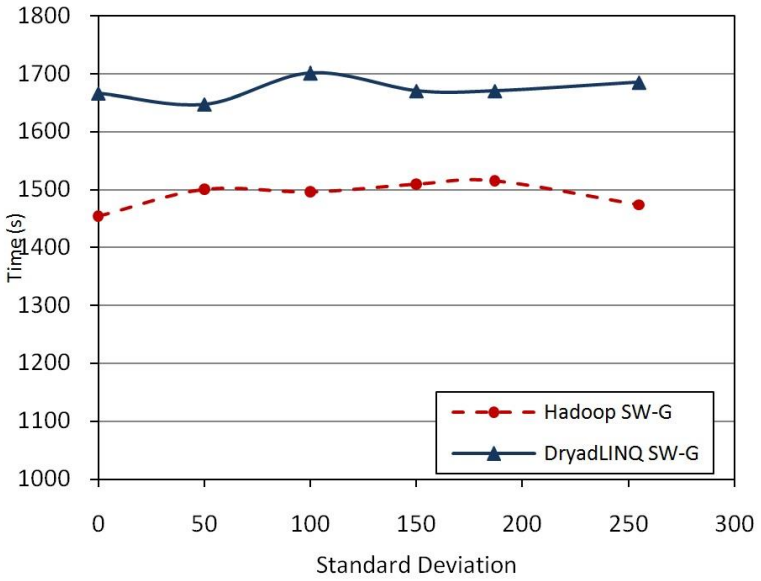


Figure 32 Performance of SW-G for randomly distributed inhomogeneous data with '400' mean sequence length.

For the inhomogeneous data study we decided to use controlled inhomogeneous input sequence sets with same average length and varying standard deviation of lengths. It's hard to generate such controlled input data sets using real sequence data as we do not have control over the length of real sequences. At the same time we note that the execution time of the Smith-Waterman pairwise distance calculation depends mainly on the lengths of the sequences and not on the actual contents of the sequences. This property of the computation makes it possible for us to ignore the contents of the sequences and focus only on the sequence lengths, thus making it possible for us to use randomly generated gene sequence sets for this experiment. The gene sequence sets were randomly generated for a given mean sequence length (400) with varying standard deviations following a normal distribution of the sequence lengths. Each sequence set contained 10000 sequences leading to 100 million pairwise distance calculations to perform. We performed two studies using such inhomogeneous data sets. In the first study the sequences with varying lengths were randomly distributed in the data sets. In the second study the sequences with varying lengths were distributed using a skewed distribution, where the sequences in a set were arranged in the ascending order of sequence length.

Figure 32 presents the execution time taken for the randomly distributed inhomogeneous data sets with the same mean length, by the two different implementations, while Figure 33 presents the executing time taken for the skewed distributed inhomogeneous data sets. The Dryad results depict the Dryad performance adjusted for the performance difference of the NAligner and JAligner kernel programs. As we notice from the Figure 32, both implementations perform satisfactorily for the randomly distributed inhomogeneous data, without showing significant performance degradations with the increase of the standard deviation. This behavior can be attributed to the fact that the sequences with varying lengths are randomly distributed across a data set, effectively providing a natural load balancing to the execution times of the sequence blocks.

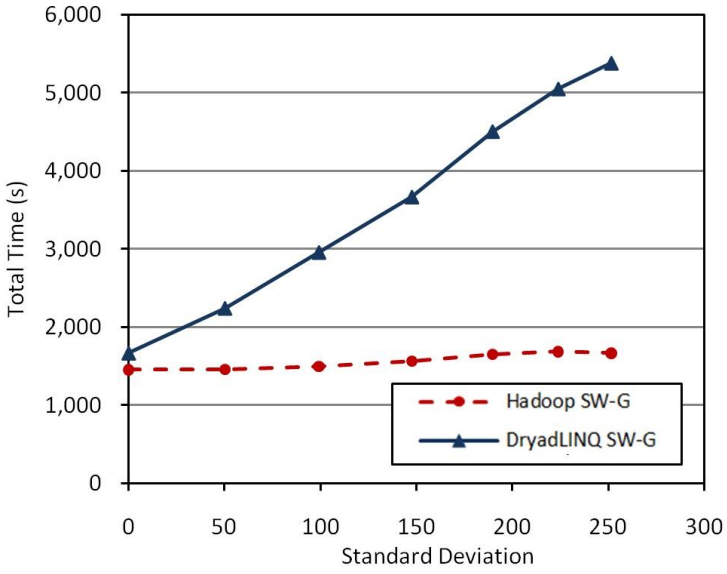


Figure 33 Performances of SW-G for skewed distributed inhomogeneous data with '400' mean sequence length.

For the skewed distributed inhomogeneous data, we notice clear performance degradation in the Dryad implementation. Once again the Hadoop implementation performs consistently without showing significant performance degradation, even though it does not perform as well as its randomly distributed counterpart. The Hadoop implementations' consistent performance can be attributed to the

global pipeline scheduling of the map tasks. In the Hadoop Smith-Waterman implementation, each block decomposition gets assigned to a single map task. Hadoop framework allows the administrator to specify the number of map tasks that can be run on a particular compute node. The Hadoop global scheduler schedules the map tasks directly on to those placeholders in a much finer granularity than in Dryad, as and when the individual map tasks finish. This allows the Hadoop implementation to perform natural global load balancing. In this case it might even be advantageous to have varying task execution times to iron out the effect of any trailing map tasks towards the end of the computation. Dryad implementation pre allocates all the tasks to the compute nodes and does not perform any dynamic scheduling across the nodes. This makes a node which gets a larger work chunk to take considerable longer time than a node which gets a smaller work chunk, making the node with a smaller work chuck to idle while the other nodes finish.

7.1.2 CAP3

Unlike in Smith-Waterman Gotoh (SW-G) implementations, CAP3 program execution time does not directly depend on the file size or the size of the sequences, as it depend mainly on the content of the sequences. This made it hard for us to artificially generate inhomogeneous data sets for the CAP3 program, forcing us to use real data. When generating the data sets, first we calculated the standalone CAP3 execution time for each of the files in our data set. Then based on those timings, we created data sets that have approximately similar mean times while the standard deviation of the standalone running times is different in each data set. We performed the performance testing for randomly distributed as well as skewed distributed (sorted according to individual file running time) data sets similar to the SWG inhomogeneous study. The speedup is taken by dividing the sum of sequential running times of the files in the data set by the parallel implementation running time.

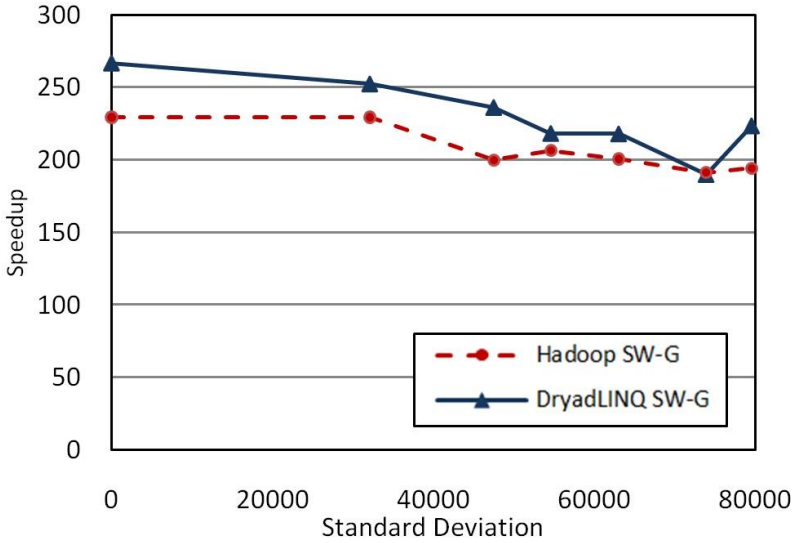


Figure 34 Performance of Cap3 for random distributed inhomogeneous data.

Figure 34 and Figure 35 depict the CAP3 inhomogeneous performance results for Hadoop & Dryad implementations. Hadoop implementation shows satisfactory scaling for both randomly distributed as well as skewed distributed data sets, while the Dryad implementation shows satisfactory scaling in the randomly distributed data set. Once again we notice that the Dryad implementation does not perform well for the skewed distributed inhomogeneous data due to its static non-global scheduling.

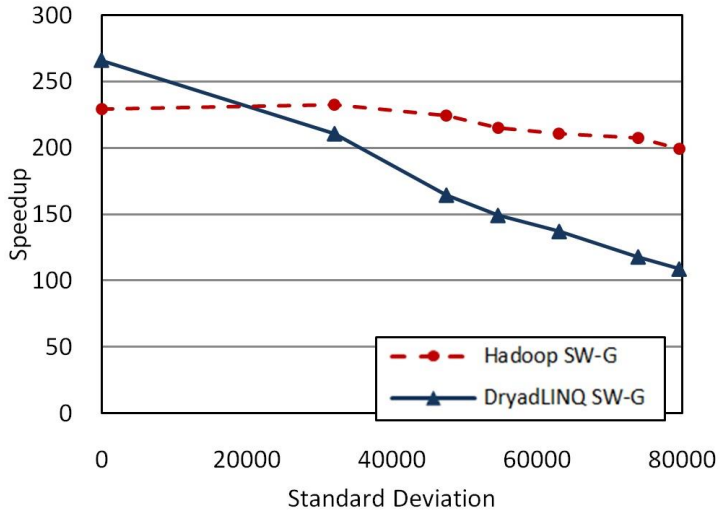


Figure 35 Performance of Cap3 for skewed distributed inhomogeneous data

7.2 Virtualization overhead

With the popularity of the computing clouds, we can notice the data processing frameworks like Hadoop, Map Reduce and DryadLINQ are becoming popular as cloud parallel frameworks. We measured the performance and virtualization overhead of several MPI applications on the virtual environments in an earlier study [54]. Here we present extended performance results of using Apache Hadoop implementations of SW-G and Cap3 in a cloud environment by comparing Hadoop on Linux with Hadoop on Linux on Xen[55] para-virtualised environment.

While the Youseff, Wolski, et al.[56] suggests that the VM's impose very little overheads on MPI application, our previous study indicated that the VM overheads depend mainly on the communications patterns of the applications. Specifically the set of applications that is sensitive to latencies (lower communication to computation ration, large number of smaller messages) experienced higher overheads in virtual environments. Walker[37] presents benchmark results of the HPC application performance on Amazon EC2, compared with a similar bare metal local cluster, where he noticed 40% to 1000% performance degradations on EC2. But since one cannot have complete control and knowledge over EC2 infrastructure, there exists too many unknowns to directly compare these results with the above mentioned results.

7.2.1 SW-G Pairwise Distance Calculation

Figure 36 presents the virtualization overhead of the Hadoop SW-G application comparing the performance of the application on Linux on bare metal and on Linux on Xen virtual machines. The data sets used is the same 10000 real sequence replicated data set used for the scalability study in the section 4.1.1. The number of blocks is kept constant across the test, resulting in larger blocks for larger data sets. According to the results, the performance degradation for the Hadoop SWG application on

virtual environment ranges from 25% to 15%. We can notice the performance degradation gets reduced with the increase of the problem size.

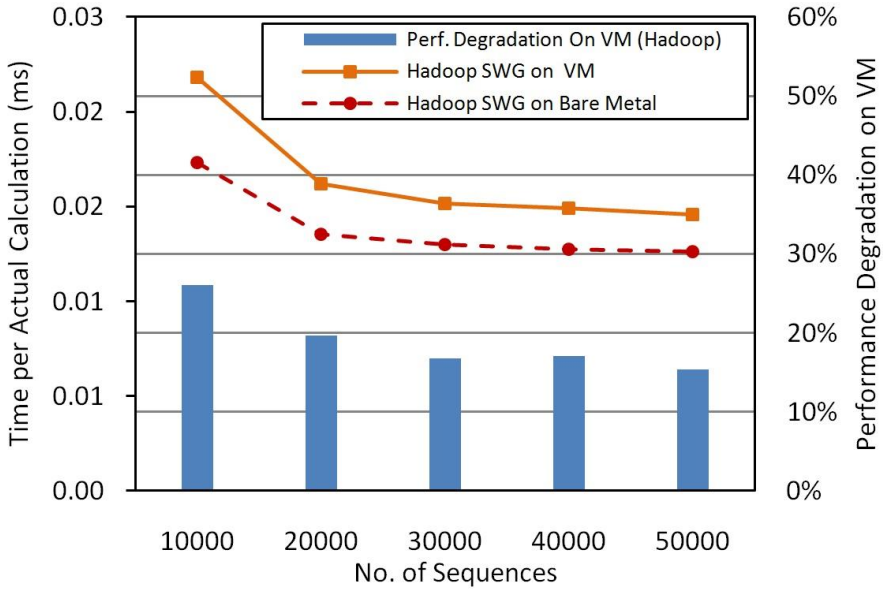


Figure 36 Virtualization overhead of Hadoop SW-G on Xen virtual machines

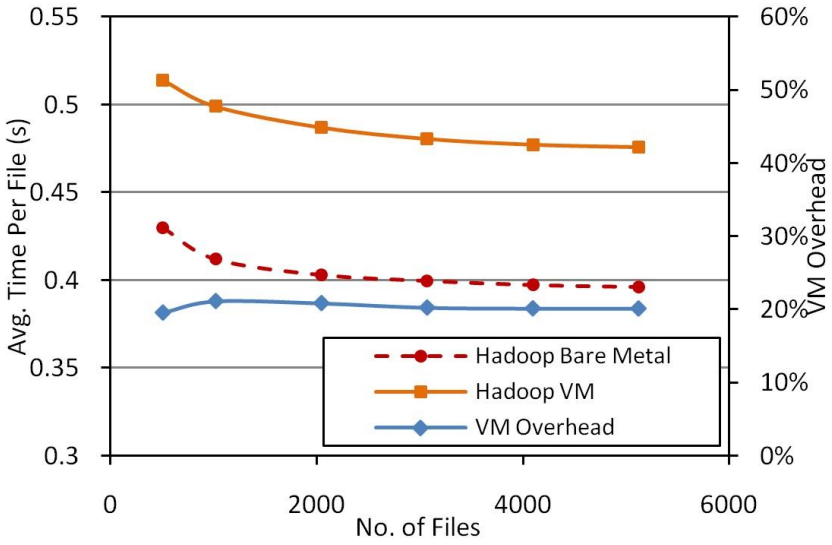


Figure 37 Virtualization overhead of Hadoop Cap3 on Xen virtual machines

In the xen para-virtualization architecture, each guest OS (running in domU) perform their I/O transfers through Xen (dom0). This process adds startup costs to I/O as it involves startup overheads such as communication with dom0 and scheduling of I/O operations in dom0. Xen architecture uses shared memory buffers to transfer data between domU's and dom0, thus reducing the operational overheads when performing the actual I/O. We can notice the same behavior in the Xen memory management, where page table operations needs to go through Xen, while simple memory accesses can be performed by the guest Oss without Xen involvement. According to the above points, we can notice that doing few coarser grained I/O and memory operations would incur relatively low overheads than doing the same work using many finer grained operations. We can conclude this as the possible reason behind the decrease of performance degradation with the increase of data size, as large data sizes increase the granularity of the computational blocks.

7.2.2 CAP3

Figure 37 presents the virtualization overhead of the Hadoop CAP3 application. We used the scalability data set we used in section 4.1.2 for this analysis. The performance degradation in this application remains constant - near 20% for all the data sets. CAP3 application does not show the decrease of VM overhead with the increase of problem size as we noticed in the SWG application. Unlike in SWG, the I/O and memory behavior of the CAP3 program does not change based on the data set size, as irrespective of the data set size the granularity of the processing (single file) remains same. Hence the VM overheads do not get changed even with the increase of workload.

7.3 Sustained performance of clouds

When discussing about cloud performance, the sustained performance of the clouds is often questioned. This is a valid question, since clouds are often implemented using a multi-tenant shared

VM-based architecture. We performed an experiment by running the SWG EMR and SWG MRRoles4Azure using the same workload throughout different times of the week. In these tests, 32 cores were used to align 4000 sequences. The results of this experiment are given in Figure 5. Each of these tests was performed at +/- 2 hours 12AM/PM. Figure 5 also includes normalized performance for MRRoles4Azure, calculated using the EMR as the baseline. We are happy to report that the performance variations we observed were very minor, with standard deviations of 1.56% for EMR and 2.25% for MRRoles4Azure. Additionally, we did not notice any noticeable trends in performance fluctuation.

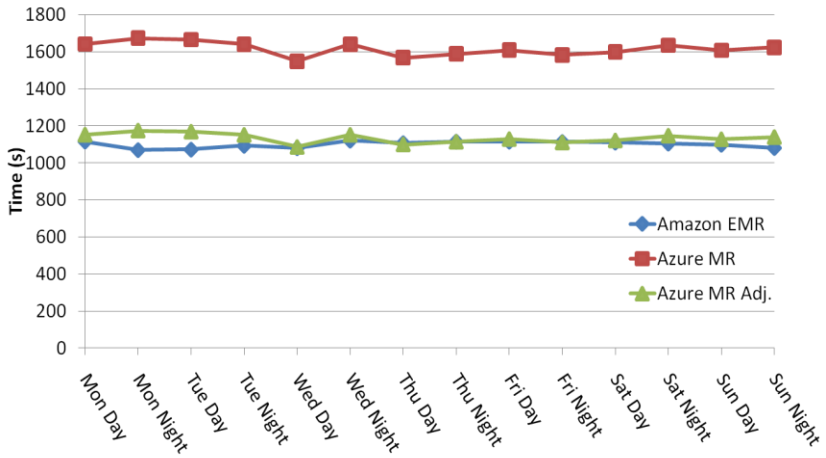


Figure 38 Sustained performance of cloud environments for MapReduce type of applications

7.4 Performance Considerations for Data Caching on Azure for Iterative MapReduce computations

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 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 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.

7.4.1 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.

7.4.2 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 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 6.1.3, Twister4Azure supports cache-aware scheduling for in-memory cached data as well as for local-storage cached data.

Table 7 Execution time analysis of a MDS computation with different data caching mechanisms. (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	6,390	750	40	3,662	131	0
In-Memory	small * 1	2072	4,052	895	140	3,924	877	143

Cache	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

7.4.2.1 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. Figure 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).

7.4.3 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 (row 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 (row 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. Figure 14(b) presents an execution trace of a job that uses Memory Mapped file based caching.

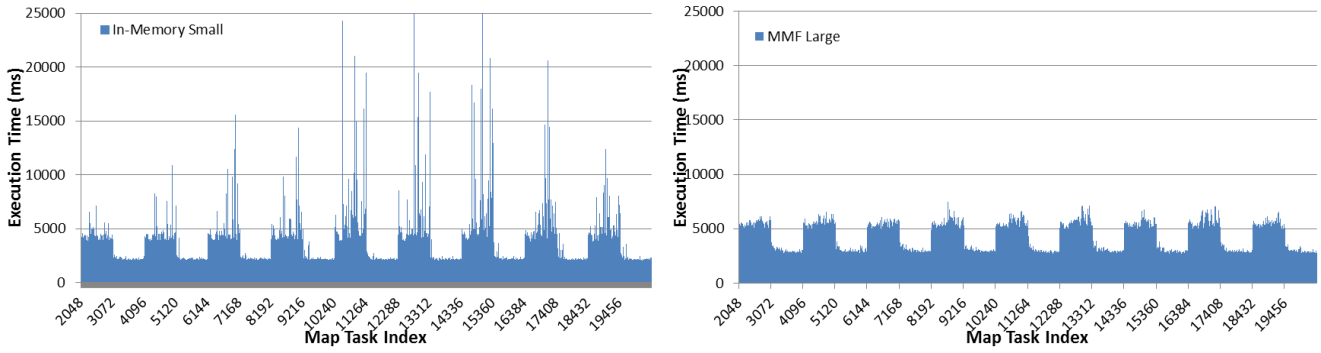


Figure 39 Execution traces of MDS iterative MapReduce computations using Twister4Azure showing the execution time of tasks in each iteration. The taller bars represent the MDSBCCalc computation, while the shorter bars represent the MDSStressCalc comput

7.5 Summary

Many real world data sets and problems are inhomogeneous in nature making it difficult to divide those computations into equally balanced computational parts. But at the same time most of the inhomogeneity of problems are randomly distributed providing a natural load balancing inside the sub tasks of a computation. We observed that the scheduling mechanism employed by both Hadoop (dynamic) and DryadLINQ (static) perform well when randomly distributed inhomogeneous data is used. Also in the above study, we observed that given there are sufficient map tasks, the global queue based dynamic scheduling strategy adopted by Hadoop provide load balancing even in extreme scenarios like skewed distributed inhomogeneous data sets. The static partition based scheduling strategy of DryadLINQ does not have the ability to load balance such extreme scenarios. It is possible, however, for the application developers to randomize such data sets before using them with DryadLINQ, which will

allow such applications to achieve the natural load balancing of randomly distributed inhomogeneous data sets we described above.

We also observed that the fluctuation of MapReduce performance on clouds is minimal over a weeklong period, assuring consistency and predictability of application performance in the cloud environments. We also performed tests using identical hardware for Hadoop on Linux and Hadoop on Linux on Virtual Machines to study the effect of virtualization on the performance of our application. These show that virtual machines give overheads of around 20%.

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.

8. Collective Communications Primitives for Iterative MapReduce

<This section needs to be updated with the content from the CCGRID paper and the final version of collective communications report to MSFT>

In this work, we present the applicability of collective communication operations to Iterative MapReduce without sacrificing the desirable properties of MapReduce programming model and execution framework such as , fault tolerance, scalability, familiar API's and data model, etc. Addition of collective communication operations enriches the iterative MapReduce model by providing many performance and ease of use advantages such as providing efficient data communication operations optimized for the particular execution environment and use case, providing programming models that fit naturally with application patterns and allowing users to avoid overheads by skipping unnecessary steps of the execution flow.

The solutions presented in this paper focus on mapping All-to-All type collective communication operations, AllGather and AllReduce to the MapReduce model as Map-AllGather and Map-AllReduce patterns. Map-AllGather gathers the outputs from all the map tasks and distributes the gathered data to all the workers after a combine operation. Map-AllReduce primitive combines the results of the Map Tasks based on a reduction operation and delivers the result to all the workers. We also present the MapReduceMergeBroadcast as a canonical model representative of the most of the iterative MapReduce frameworks.

We present prototype implementations of Map-AllGather and Map-AllReduce primitives for Twister4Azure and Hadoop (called H-Collectives). We achieved up to 33% improvement for

KMeansClustering and up to 50% improvement with Multi-Dimensional Scaling in addition to the improved user friendliness. In some case, collective communication operations virtually eliminated almost all the overheads of the computations.

In addition to the common characteristics of data-intensive iterative computations, we noticed several common communication and computation patterns among some of the data-intensive iterative MapReduce computations. In this paper, we propose Map-Collectives, a set of collective communications primitives, to support such common patterns. These patterns have the ability to further optimize performance and usability of iterative MapReduce applications while preserving the unique advantages of the MapReduce paradigm. These collective communication primitives provide trifold advantages to the MapReduce users: 1) offer easier more natural programming models, 2) improve performance by allowing the framework to perform environment-aware communication and computation optimizations, 3) reduce overheads by avoiding unnecessary computation and communication steps. These patterns can be implemented on any of the current Iterative MapReduce frameworks as well as on traditional MapReduce frameworks like Hadoop. In this paper we present the Map-AllGather and Map-AllReduce primitives and initial performance results for them. Another primitive that we are working on is the Map-Scatter primitive.

8.1 Collective Communication

When performing distributed computations, often times the data needs to be shared and/or consolidated among the different nodes of the computations. Collective communication primitives are the communication operations that involve a group of nodes simultaneously[34], rather than exchanging data between just a pair of nodes. These powerful operations make it much easier and efficient to perform complex data communications. Some of the collective communication operations also provides synchronization capabilities to the applications as well. Collective communication

operations are used heavily in MPI type of HPC applications. A lot of research[34] had been done to optimize the performance of these collective communication operations, as they have a large impact on the performance of HPC applications. There exist many different implementations of collective communication primitives supporting many different algorithms and topologies to suit the different environments and different use cases. The best collective implementation for a given scenario depends on many factors including message size, number of workers, topology of the system, the computational capabilities/capacity of the nodes and etc [3]. Oftentimes collective communication implementations follow a poly-algorithm approach to automatically select the best algorithm and topology for the given scenario.

There are two main categories of collective communication primitives.

- Data redistribution operations
  - These operations can be used to distribute and share data across the worker processors. Examples of these include broadcast, scatter, gather, and allgather operations.
- Data consolidation operation
  - This type of operations can be used to collect and consolidate data contributions from different worker processors. Examples of these include reduce, reduce-scatter and allreduce.

We can also categorize collective communication primitives based on the communication patterns as well.

- All-to-One: gather, reduce
- One-to-All : broadcast, scatter
- All-to-All : allgather, allreduce, reduce-scatter
- Barrier

MapReduce model supports the All-to-One type communications through the Reduce step. MapReduce-MergeBroadcast model we introduce in section 8.3 further extends this support through

the Merge step. Broadcast operation introduced in MapReduce-MergeBroadcast model serves as an alternative to the One-to-All type collective communication operations. MapReduce model contains a barrier between the Map and Reduce phases and the iterative MapReduce model introduces a barrier between the iterations (or between the MapReduce jobs corresponding to iterations). The solutions presented in this paper focus on introducing All-to-All type collective communication operations to the MapReduce model.

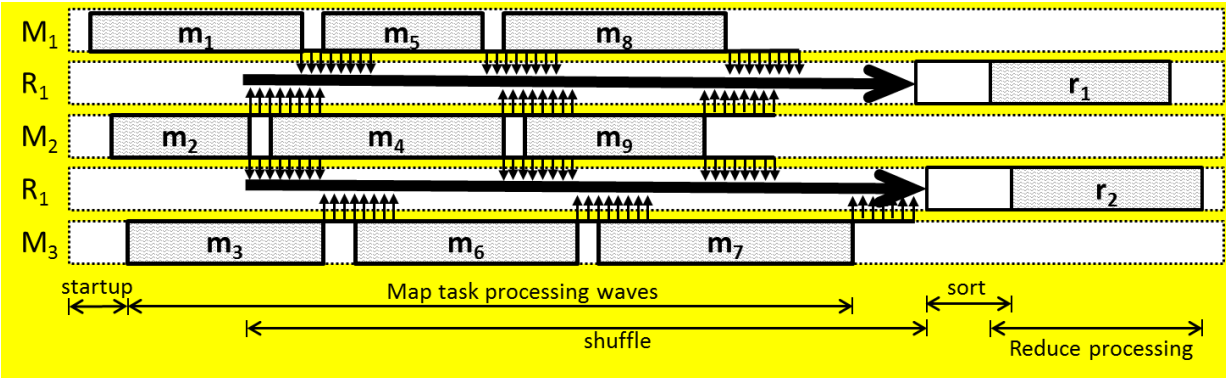
We can implement All-to-All communications using pairs of existing All-to-One and One-to-All type operations present in the MapReduce-MergeBroadcast mode. For an example, AllGather operation can be implemented as Reduce-Merge followed by Broadcast. However, these type of implementations would be inefficient and would be harder to use compared to dedicated optimized implementations of All-to-All operations.

8.2 MapReduce

MapReduce consist of a programming model and an associated execution framework for distributed processing of very large data sets. MapReduce partitions the processing of very large input data in to a set of independent tasks. MapReduce programming model consists of *map(key_1, $value_1$)* function and *reduce(key_2, list<$value_2$>)* function. MapReduce programs written as Map and Reduce functions will be parallelized by the framework and will be executed in a distributed manner. MapReduce framework takes care of data partitioning, task scheduling, fault tolerance, intermediate data communication and many other aspects of MapReduce computations for the users. These features and the simplicity of the programming model allows users with no background or experience in distributed and parallel computing to utilize MapReduce and the distributed infrastructures to easily process large volumes of data.

MapReduce frameworks are typically not optimized for the best performance or parallel efficiency of small scale applications. Main goals of MapReduce frameworks include framework managed fault tolerance, ability run on commodity hardware, ability to process very large amounts of data and horizontal scalability of compute resources. MapReduce frameworks like Hadoop tradeoff costs such as large startup overheads, task scheduling overheads and intermediate data persistence overheads for better scalability and reliability.

When running a computation, MapReduce frameworks first logically split the input data in to partitions, where each partition would be processed by a single Map task. When a Map Reduce computation has more map tasks than the Map slots available in the cluster, the tasks will be scheduled in waves. For an example, a computation with 100 map tasks executing in a cluster of 200 Map slots will execute as approximately 5 map task waves. Tasks in MapReduce frameworks are often times dynamically scheduled taking data locality in to the consideration. Map tasks read the data from the assigned logical data partition and process them as key value pairs using the provided *map* function. The output key-value pairs of a map function are collected, partitioned, merged and transferred to the corresponding reduce tasks. Most of the MapReduce frameworks persists the Map output data in the Map nodes.



Reduce tasks fetch the data from the Map nodes and performs an external-merge sort on the data. Fetching of intermediate data starts as soon as the first map task completes the execution. Reduce task starts the reduce function processing after all the Map tasks are finished and after all the intermediate data are shuffled and sorted.

8.2.1 MapReduce Cost Model

There exist several models that are frequently used by the message passing community to model to data communication performance[57]. We use the Hockney model[57, 58] for the simplicity. Hockney model assumes the time to send a data set with n data items among two nodes is α+nβ, where α is the latency and β is the transmission time per data item (1/bandwidth). Our model of the I/O cost of MapReduce computations is as follows[59]. Hockney model cannot model the network congestion.

δ – Disk time read/write a data item

W – Number of map waves. W= $\left\lceil \frac{m}{num\ map\ slots} \right\rceil$

m, r – Number of map and reduce tasks

p - Number of worker nodes

n_{in} , n_v , n_m, n_r - number of total input, loop variant, map output and reduce output data items. ($n_{in} = \sum_{i=0}^{m} input(map_i),\ \ n_m = \sum_{i=0}^{m} output(map_i),\ \ n_r = \sum_{i=0}^{m} output(reduce_i)$)

	Map Task					Reduce Task			
Task Scheduling	Data read	Map execution	Collect	Spill	Merge	Shuffle	Merge	Reduce Execution	Write output
λ	$\frac{W\delta n_{in}}{m}$	$\frac{W}{m} f(n_{in})$	$\frac{Wn_m}{m}\left(\delta + \log\left(\frac{Wn_m}{m}\right)\right)$			$\frac{m}{W}\alpha + \frac{n_m}{Wr}\beta$	$\frac{n_m}{r}(1+\delta)$	$\frac{f(n_m)}{r}$	$\frac{n_r\delta}{r}$

$$T_{MR} = \boldsymbol{\lambda} + \frac{W}{m}\left(\delta n_{in} + f(n_{in}) + n_m\left(\delta + \log\left(\frac{Wn_m}{m}\right)\right)\right) + \frac{m}{W}\alpha + \frac{n_m}{Wr}\beta + \frac{1}{r}(n_m(\delta+1) + f(n_m) + n_r\delta)$$

$$T_{MR} = \boldsymbol{\lambda} + \frac{W}{m}\left(f(n_{in}) + n_m \log\left(\frac{Wn_m}{m}\right)\right) + \frac{f(n_m)+n_m}{r} + \left(\frac{Wn_{in}+Wn_m}{m} + \frac{n_m+n_r}{r}\right)\delta + \frac{1}{W}\left(m\alpha + \frac{n_m}{r}\beta\right)$$

8.2.1.1 Assumptions

- When W map task waves are present, we assume all the output data of the first W-1 map tasks waves are completely shuffled by the time the first task of the W^{th} wave completes the execution. This is possible when the average execution time of a map task is larger than the average time to shuffle the output data of a map task to the reduce tasks. In the case where the average execution time of a map task is smaller than the average time to shuffle the output data of a map task to the reduce tasks, then the cost would be average execution time of a single map wave plus the time to transfer all the data.
- Each map task wave outputs approximately the same amount of data.
- Map output data will get equally partitioned to the reduce tasks, each reduce task getting n_m/r data.
- Reduce tasks will get scheduled in a single wave. (r < num. reduce slots)
- Map tasks are homogeneous and every task finishes at the same time. Hence n_m/Wr represent the intermediate data headed for a reduce task simultaneously. In reality, the map tasks have variations in the run times and the intermediate data shuffling takes advantage of this by starting the shuffle of finished tasks while the other tasks are still executing. This allows the applications to more efficiently utilize the bandwidth and to improve the data shuffling performance and overhead.
- Assume no network congestion and conflicts.

8.3 MapReduce-MergeBroadcast

In this section we introduce MapReduce-MergeBroadcast as a generic programming model for the data-intensive iterative MapReduce applications. Programming model of most of the current iterative MapReduce frameworks can be specified as MapReduce-MergeBroadcast.

8.3.1 API

MapReduce-MergeBroadcast programming model extends the *map* and *reduce* functions of traditional MapReduce to include the loop variant delta values as an input parameter. MapReduce-

MergeBroadcast provides the loop variant data (*dynamicData*) to the *Map* and *Reduce* tasks as a list of key-value pairs using this additional input parameter.

Map(<key>, <value>, list_of <key,value> dynamicData)

Reduce(<key>, list_of <value>, list_of <key,value> dynamicData)

This additional input can be used to provide the broadcast data to the Map and Reduce tasks. As we show in the later sections of this paper, this additional input parameters can used to provide the loop variant data distributed using other mechanisms to the map tasks. This extra input parameter can also be used to implement additional functionalities such as performing map side joins.

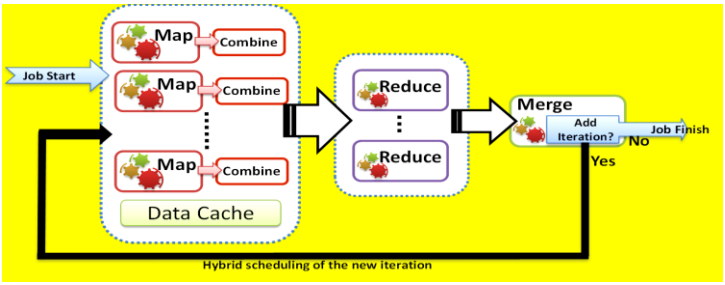


Figure 1. Twiter4Azure MapReduce-MergeBroadcast compute/data flow

8.3.2 Merge Task

We define *Merge* as a new step to the MapReduce programming model to support iterative applications. It is a single task, or the convergence point, that executes after the *Reduce* step.

It can be used to perform summarization or aggregation of the results of a single MapReduce iteration. The *Merge* step can also serve as the "loop-test" that evaluates the loops condition in the iterative MapReduce programming model. Merge tasks can be used to add a new iteration, finish the job, or schedule a new MapReduce job. These decisions can be made based on the number of iterations or by comparison of the results from previous and current iterations, such as the k-value difference

between iterations for K-means Clustering. Users can use the results of the current iteration and the broadcast data to make these decisions. Oftentimes the output of the merge task needs to be broadcasted to tasks of the next iteration.

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 flow would appear as follows:

Figure 2. MapReduce-MergeBroadcast computation flow

The programming APIs of the Merge task can be where the "reduceOutputs" are the outputs of the reduce tasks and the "broadcastData" is the loop variant broadcast data for the current iteration.

Merge(list_of <key,list_of<value>> reduceOutputs, list_of <key,value> dynamicData)

8.3.3 Broadcast

Boradcast operation broadcasts the loop variant data to all the tasks in iteration. In typical data-intensive iterative computations, the loop-variant data is orders of magnitude smaller than the loop-invariant data. In the MapReduce-MergeBroadcast model, the broadcast operation typically broadcasts the output data of the Merge tasks to the tasks of the next iteration. Broadcast operation of MapReduce-MergeBroadcast can also be thought of as executing at the beginning of the iterative MapReduce computation. This would make the model Broadcast-MapReduce-Merge, which is essentially similar to the MapReduce-Merge-Broadcast when iterations are present.

…$MapReduce_n$-> $Merge_n$-> $Broadcast_n$-> $MapReduce_{n+1}$-> $Merge_{n+1}$-> $Broadcast_{n+1}$-> $MapReduce_{n+2}$-> Merge...

Broadcast can be implemented efficiently based on the environment as well as the data sizes. Well known algorithms for data broadcasting include flat-tree, minimum spanning tree (MST), pipeline and

chaining[57]. It's possible to share broadcast data between multiple tasks executing on a single node, as MapReduce computations typically have more than one map/reduce/merge worker per worker-node.

8.3.4 MapReduceMergeBroadcast Cost Model

Merge is a single task that receives the outputs of all the reduce tasks. The cost of this transfer would be $r\alpha + n_r\beta$. The execution time of the Merge task would be relatively small, as typically the merge would be performing a computationally trivial task such as aggregation or summarization. The output of the Merge task would need to be broadcasted to all the workers of the next iteration. A minimal spanning tree based broadcast would cost,

$$T_{MST-BCast} = (\alpha + \beta n_v)\log(p)$$

Based on these costs, the total cost of a MapReduce-MergeBroadcast can be approximated as follows. The broadcast needs to done only once per worker node as the map tasks executing in a single worker node can share the broadcasted data among the tasks.

$$T_{MR-MB} = T_{MR} + r\alpha + n_r\beta + f(n_r) + (\alpha + \beta n_v)\log(p)$$

8.3.5 Current iterative MapReduce Frameworks and MapReduce-MergeBroadcast

Twister4Azure supports the MapReduce-MergeBroadcast natively. In Twister, the combine step is part of the driver program and is executed after the MapReduce computation of every iteration. Twister is a MapReduce-Combine model, where the Combine step is similar to the Merge step. Twister MapReduce computations broadcast the loop variant data products at the beginning of each iteration, effectively making the model Broadcast-MapReduce-Combine. This is semantically similar to the

MapReduce-MergeBroadcast, as a broadcast at the beginning of iteration is similar to a broadcast at end of the previous iteration.

HaLoop performs an additional MapReduce computation to do the fixed point evaluation for each iteration, effectively making this MapReduce computation equivalent to the Merge task. Data broadcast is achieved through a MapReduce computation to perform a join operation on the loop variant and loop invariant data.

All the above models can be generalized as Map->Reduce->Merge->Broadcast.

8.4 Motivation

Why this is not guaranteed,

- Has to fit with the data model
- Has to fit with the computational model which is more non-deterministic and has multiple waves, large overheads and inhomogeneous tasks.
- Has to retain scalability and framework managed fault tolerance of MapReduce
- Has to keep the programming model simple and easy to understand

8.4.1 Motivation

While implementing some of the iterative MapReduce applications, we started to notice several common execution flow patterns. Another point of interest is that some of these applications have very trivial Reduce and Merge tasks when implemented using the MapReduce-MergeBroadcast model, while other applications needed extra effort to map to the MapReduce-MergeBroadcast model. This is owing to the execution patterns being slightly different than the iterative MapReduce pattern. In order to solve such issues, we introduce the Map-Collectives communications primitives, inspired by the MPI collective communications primitives, to the iterative MapReduce programming model.

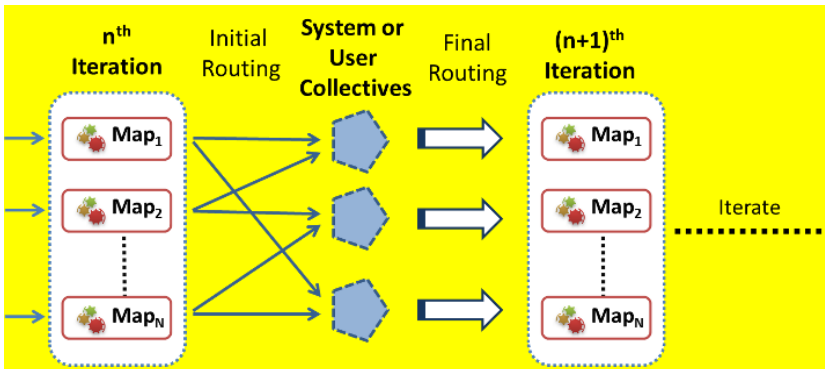


Figure 3. Collective Communication primitives

These primitives support higher-level communication patterns that occur frequently in data-intensive iterative applications, also in addition to substituting certain steps of the computation. These collective primitives can be thought of as a Map phase followed by a series of framework-defined communication and computation operations leading to the next iteration.

In this paper we propose three collective communication primitive implementations: Map-AllGather, Map-AllReduce and Map-Scatter. You can also identify MapReduce-MergeBroadcast as another collective communication primitive as well. Map-AllGather, Map-Allreduce and Map-Scatter also exist as special cases of the MapReduce-MergeBroadcast. These Map-Collective primitives provide many improvements over the traditional MapReduce-MergeBroadcast model. They are designed to maintain the same type of framework-managed excellent fault tolerance supported by the MapReduce frameworks.

It is not our objective to find the most optimal implementations for each of the environments, rather to present a sufficiently optimal implementation for each of the primitive and for each of the environments, to prove the performance efficiencies that can be gained through even using a modest implementation of these operations. We leave finding of the most optimized methods for each environment as a future work. Also we should note that finding most optimal implementations

for cloud environments might end up being a moving target as cloud environments evolve very rapidly and the cloud providers release new cloud services and other features very frequently.

8.4.1.1 Performance

These primitives allow us to skip or overlap certain steps of the typical iterative MapReduce computational flow. Having patterns that are more natural avoids unnecessary steps in traditional MR and iterative MR.

Another advantage is the ability of the frameworks to optimize these operations transparently for the users, even affording the possibility of different optimizations (poly-algorithm) for different use cases and environments. For an example, a communication algorithm that's best for smaller data sizes might not be the best for larger data sizes. In such cases, the collective communication operations can opt to implement multiple algorithm implementations to be used for different data sizes.

Finding the most optimal communication pattern implementations for a cloud environment is harder for the outsiders as due to the black box nature of cloud environments where we don't have any information about the actual topology or the interconnects of public clouds. This presents an interesting opportunity for cloud providers to provide optimized implementations of these primitives as cloud infrastructure services, that can be used by the framework developers.

These primitives also have the capability to make the applications more efficient by overlapping communication with computation, Frameworks can start the execution of collectives as soon as the first results are produced from the Map tasks. For example, in the Map-AllGather primitive, partial Map results are broadcasted to all the nodes as soon as they become available. It is also possible to perform some of the computations in the data transfer layer, likein the Map-AllReduce primitive. The data reduction can be performed hierarchically using a reduction tree.

8.4.1.2 Ease of use

These primitive operations make life easier for the application developers by presenting them with patterns and APIs that fit more naturally with their applications. This simplifies the case when porting new applications to the iterative MapReduce model.

In addition, by using the Map-Collective operations, the developers can avoid manually implementing the logic of these operations (e.g. Reduce and Merge tasks) for each application and can rely on optimized operations provided by the framework.

8.4.1.3 Scheduling with iterative primitives

Iterative primitives also give us the ability to propagate the scheduling information for the next iteration. These primitives can schedule the tasks of a new iteration or application through the primitives by taking advantage of the primitives to deliver information about the new tasks. This can reduce scheduling overheads for the tasks of the new iteration.Twister4Azure successfully employs this strategy, together with the caching of task metadata, to schedule new iterations with minimal overhead.

8.4.1.4 Programming model

Iterative MapReduce collective communication primitives can be specified as an outside configuration option without changing the MapReduce programming model. This permits the Map-Collectives to be compatible with frameworks that don't support the collectives. This also makes it easier to use collectives by developers who are already familiar with MapReduce programming.

8.4.1.5 Implementations

Map-Collectives can be add-on improvements to MR frameworks. The simplest implementation would be applying these primitives using the current MapReduce model on the user level, thenproviding them as a library. This will achieve ease of use for the users and in addition help with the performance.

More optimized implementations can implement these primitives as part of the MapReduce framework as well as providing the ability to optimize the data transfers based on environment and use case.

Table 8 Summary of patterns

Pattern	Execution and communication flow	Frameworks	Sample applications
MapReduce	Map->Combine->Shuffle->Sort->Reduce	Hadoop, Twister, Twister4Azure	WordCount, Grep, etc.
MapReduce-MergeBroadcast	Map->Combine->Shuffle->Sort->Reduce->Merge->Broadcast	Twister, Haloop,Twister4Azure	KMeansClustering, PageRank,
Map-AllGather	Map->AllGather Communication->AllGather Combine	H-Collectives, Twister4Azure	MDS-BCCalc
Map-AllReduce	Map->AllReduce (communication + computation)	H-Collectives, Twister4Azure	KMeansClustering, MDS-StressCalc
Map-ReduceScatter	Map->Scatter (communication + computation)	H-Collectives, Twister4Azure	PageRank, Belief Propagation

8.5 Map-AllGather Collective

AllGather is an all-to-all collective communication operation that gathers data from all the workers and distribute the gathered data to all the workers [34]. We can notice the AllGather pattern in data-intensive iterative applications where the "reduce" step is a simple aggregation operation that simply aligns the outputs of the Map Tasks together in order, followed by "merge" and broadcast steps that transmit the assembled output to all the workers. An example would be a MapReduce computation that generates a matrix as the loop variant delta, where each map task outputs several rows of the resultant matrix. In this computation we would use the Reduce and Merge tasks to assemble the matrix together and then broadcast the assembled matrix.

Example data-intensive iterative applications that have the AllGather pattern include MultiDimensionalScaling.

8.5.1 Model

We developed a Map-AllGather iterative MapReduce primitive similar to the MPI AllGather[34] collective communication primitive. Our intention was to support the above-mentioned scenario in a more efficient manner.

8.5.1.1 Execution model

Map-AllGather primitive broadcasts the Map Task outputs to all computational nodes (all-to-all communication) of the current computation, then assembles them together in the recipient nodes. Each Map worker will deliver its result to all other workers of the computation once the Map task finishes. In the Matrix example mentioned above, each Map task broadcasts the rows of the matrix it generates to all the other workers. The resultant matrix would be assembled at each worker after the receipt of all the Map outputs.

The computation and communication pattern of a Map-AllGather computation is Map->AllGather communication (all-to-all) -> AllGather combine. As we can notice, this model substitute the shuffle->sort->reduce->merge->broadcast steps of the MapReduce-MergeBroadcast with all-to-all broadcast and Allgather combine.

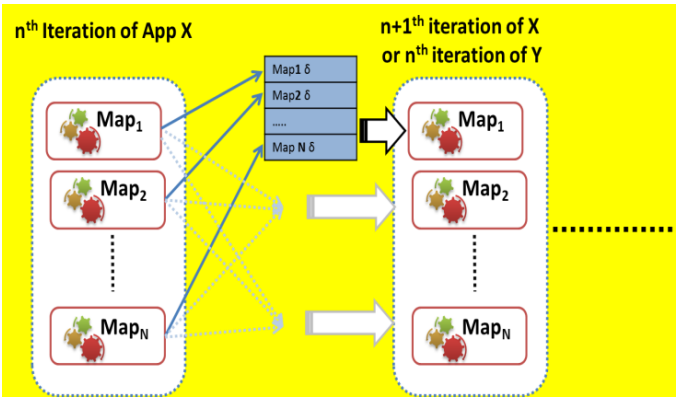


Figure 4. Map-AllGather Collective

8.5.1.2 Data Model

Users can use their Mapper implementations as it is with Map-AllGather primitive. User can just specify the pattern and then the shuffle-> reduce of MapReduce would be replaced by AllGather communication and computations. The final assembly of AllGather data can be performed by implementing a custom combiner or using a default AllGather-combine. The result of AllGather-Combine will be provided to the Map tasks of the next iteration as the loop variant values using the API's and mechanisms suggested in section 2.2.1.

<diagram comparing MapReduce code with AllReduce Code>

The default combiner should work for most of the use cases as the combining of AllGather data is a oftentimes a trivial process. The default combiner expect the Map outputs to be in <int, double[]> format. The map output key represents the location of the corresponding value in the assembled value. In the above mentioned Matrix example the key would represent the row index of the output matrix

and the value would contain the corresponding row vector. Map outputs with duplicate keys (same key for multiple output values) are not supported and ignored in the default Map-Allgather combine step.

The custom AllGather-Combine interface would be as follows.

<AllGather-Combine> API

8.5.1.3 Cost Model

An optimized implementation of AllGather, such as a by-directional exchange based implementation[34], will reduce the cost of the AllGather component to,

$$T_{AllGather} = \log(m)\,\alpha + \frac{m-1}{m} n_v \beta$$

It's also possible to further reduce this cost by performing local aggregation in the Map worker nodes. In the case of AllGather, summation of size of all map output would be approximately equal to the loop variant data size of the next iteration ($n_m \approx n_v$). The variation of Map task completion times will also help to avoid the network congestion in these implementations.

$$T_{AllGather} = \log(p)\,\alpha + \frac{p-1}{p} n_v \beta$$

Following provides an estimation of the cost of per iteration of Map-AllGather computation.

$$T_{Map-AllGather} = \boldsymbol{\lambda}' + \frac{W}{m}\left(\delta n_{in} + f(n_{in}, n_v)\right) + \log(p)\,\alpha + \frac{p-1}{p} n_v \beta$$

As we can compare with T_{MR} and $T_{MR\text{-}MB}$ in sections xx and xx, Map-Allgather substitute the Map output processing (collect, spill, merge), Reduce task (shuffle, merge, execute, write), Merge task (shuffle, execute) and broadcast overheads with a less costly AllGather operation. The MapReduce job startup overhead can also be significantly reduced by utilizing the information contained in the

AllGather transfers to aid in scheduling the tasks of the next iteration. Hence Map-AllReduce per iteration overhead is significantly reduced than the traditional MapReduce job startup overhead ($\lambda' \ll \lambda$) as well.

8.5.2 Fault tolerance

8.5.2.1 Correctness

When task level fault tolerance (typical mapreduce fault tolerance) is used and some of the All-Gather data products are missing due to communication and other failures, it's possible for the workers to read map output data from the persistent storage (eg:HDFS) to perform the All-Gather computation.

The fault tolerance model and the speculative execution model of MapReduce make it possible to have duplicate execution of tasks. Duplicate detection can be performed before the final assembly of the data at the recipient nodes.

8.5.3 Benefits

Use of the AllGather primitive in an iterative MapReduce computation eliminates the need for reduce, merge and the broadcasting steps in that particular computation. Additionally the smaller sized multiple broadcasts of our Map-AllGather primitive would be able to use the network more effectively than a single larger broadcast originating from a single point.

Implementations of AllGather primitive can start broadcasting the map task result values as soon as the first map task is completed. In a typical MapReduce computations, often times the Map task execution times are inhomogeneous making. This mechanism ensures that almost all the data is broadcasted by the time the last map task completes its execution, resulting in overlap of computations with communication. This benefit will be more significant when we have multiple waves of map tasks.

In addition to improving the performance, this primitive also enhances the system usability as it eliminates the overhead of implementing reduce and/or merge functions. Map-AllGather can be used to schedule the next iteration or the next application of the computational flow as well.

8.5.4 Implementations

In this section we present two implementations of the Map-AllGather primitive. These implementations are proof of concept to show the advantages achieved by using the Map-AllGather primitive. It's possible to further optimize them using more advanced algorithms, based on the environment they will be executing, the scale of the computations, and the data sizes as shown in MPI collective communications literature[34]. One of the main advantages of these primitives is the ability for improvement without the need to change the user application implementations, leaving us open for optimization of these implementations in the future.

8.5.4.1 Twister4Azure Map-AllGather

The Map-AllGather primitive is implemented in Twister4Azure using the Windows Communication Foundation (WCF)-based Azure TCP inter-role communication mechanism of Azure platform, with the Azure table storage as a persistent backup. It performs simple TCP-based broadcasts for each Map task output, which is an all-to-all linear implementation. Workers don't wait for the other workers to be completed and start transmitting the data as soon as a task is completed taking advantage of the inhomogeneous finish times of Map tasks to avoid the communication scheduling and network congestion. More sophisticated implementations can be a tree based algorithm or a pairwise exchange based algorithm, however, the performance of these optimized algorithms may be hindered by the fact that all data will not be available at the same time.

8.5.4.2 H-Collectives Map-AllGather

H-Collectives Map-AllGather primitive is implemented using the Netty NIO library on top of Apache Hadoop. Performs simple TCP-based best effort broadcasts for each Map task output. If a data product is not received through the TCP broadcasts, then it would be fetched from the HDFS. The tasks for the next iteration are already scheduled and waiting to start as soon as all the AllGather data is received, getting rid of most of the MapReduce application startup and task scheduling overheads.

8.6 Map-AllReduce Collective

AllReduce is a collective pattern which combines a set of values emitted by all the workers based on a specified operation and makes the results available to all the workers[34]. This pattern can be seen in many iterative data mining and graph processing algorithms. Example data-intensive iterative applications that have the Map-AllReduce pattern include KMeansClustering, Multi-dimensional Scaling Stresscalc computation and PageRank using out links matrix.

8.6.1 Model

We propose Map-AllReduce iterative MapReduce primitive similar to the MPI AllReduce[34] collective communication operation, that will aggregate and reduce the results of the Map Tasks.

8.6.1.1 Execution Model

The computation and communication pattern of a Map-AllReduce computation is Map->AllReduce communication and computation (reduction) -> AllGather combine. As we can notice, this model substitute the shuffle->sort->reduce->merge->broadcast steps of the MapReduce-MergeBroadcast with hierarchical reduction.

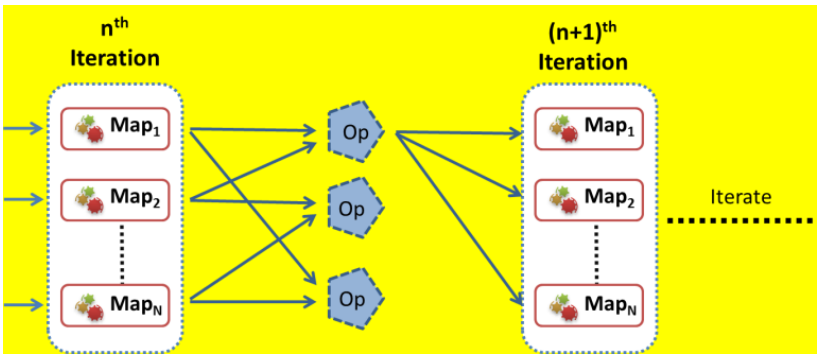


Figure 5. Map-AllReduce collective

Map-AllReduce allows the implementations to perform local aggregation, hierarchical reduction of the Map Task outputs while delivering them to all the workers and to perform the final reduction in the recipient worker nodes.

8.6.1.2 Data Model

In Map-AllReduce, we consider each map output key as a separate data product and the combine operation would be applied for all the values of that particular key. The values can be vectors of single values. For example, if the Map tasks output 10 keys, then the Map-AllReduce result would have 10 combined vectors.

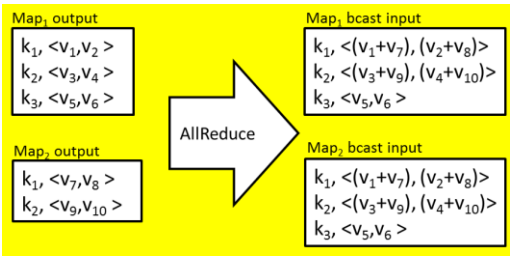


Figure 6. Example Map-AllReduce with Sum operation

In addition to the summation, any commutative and associative operation can be performed using this primitive. Example operations include sum, max, min, count, and product operations. Operations such as average can be performed by using Sum operation together adding an additional element (dimension) to the vector to count the number of data products. Due to the associative and

commutative nature of the operations, AllReduce has the ability to start combining the values as soon as at least one mapper completes the execution. It also allows the AllReduce implementations to use reduction trees to optimize the AllReduce computation.

8.6.1.3 Cost Model

An optimized implementation of AllReduce, such as a by-directional exchange based implementation[34], will reduce the cost of the AllReduce component to,

$$T_{AllReduce} = \log(m)\ (\alpha + n_v\beta + f(n_v))$$

It's also possible to further reduce this cost by performing local aggregation and reduction in the Map worker nodes as the compute cost of AllReduce ($f(n_v)$) is very small. $\frac{m}{p}$ gives the average number of Map tasks per computation that executes in a given worker node. In the case of AllReduce, the average size of each map output would be approximately equal to the loop variant data size of the next iteration ($\frac{n_m}{m} \approx n_v$). The variation of Map task completion times will also help to avoid the network congestion in these implementations.

$$T_{AR} = \log(p)\ (\alpha + n_v\beta + f(n_v)) + \frac{m}{p} f(n_v)$$

Following provides an estimation of the cost of per iteration of Map-AllReduce computation.

$$T_{M-AR} = \lambda' + \frac{W}{m}\left(\delta n_{in} + f(n_{in}, n_v)\right) + \log(p)\ (\alpha + n_v\beta + f(n_v)) + \frac{m}{p} f(n_v)$$

As we can compare with T_{MR} and $T_{MR\text{-}MB}$ in sections xx and xx, Map-AllReduce substitute the Map output processing (collect, spill, merge), Reduce task (shuffle, merge, execute, write), Merge task (shuffle, execute) and broadcast overheads with a less costly AllReduce operation. The MapReduce job

startup overhead can also be reduced by utilizing the information contained in the AllReduce transfers to aid in scheduling the tasks of the next iteration resulting in $\lambda' \ll \lambda$ as well.

8.6.2 Fault Tolerance

When task level fault tolerance (typical mapreduce fault tolerance) is used and the All-Reduce communication step fails for whatever reason, it's possible for the workers to read map output data from the persistent storage to perform the All-Reduce computation.

The fault tolerance model and the speculative execution model of MapReduce make it possible to have duplicate execution of tasks. Duplicate executions can result in incorrect Map-AllReduce results due to the possibility of aggregating the output of the same task twice as Map-AllReduce starts the data reduction as soon as the first Map task output is present. For an example, if a mapper is re-executed or duplicate-executed, then it's possible for the All-Reduce to combine duplicate values emitted from such mappers. The most trivial fault tolerance model for AllReduce would be a best-effort mechanism, where the AllReduce implementation would fall back to using the Map output results from the persistent storage (eg: HDFS) in case duplicate results are detected. Duplicate detection can be performed by maintaining a set of map ID's with each combined data product. It's possible for the frameworks to implement richer fault tolerance mechanisms, such as identifying the duplicated values in local areas of the reduction tree.

8.6.3 Benefits

This primitive will reduce the work each user has to perform in implementing Reduce and Merge tasks. It also removes the overhead of Reduce and Merge tasks from the computations and allows the framework to perform the combine operation in the communication layer itself.

Map-AllReduce semantics allow the implementations to perform hierarchical reductions, reducing the amount of intermediate data and optimizing the computation. The hierarchical reduction can be

performed in as many levels as needed for the size of the computation and the scale of the environment. For example, first level in mappers, second level in the node and n^{th} level in rack level, etc. The mapper level would be similar to the "combine" operation of vanilla map reduce. The local node aggregation can combine the values emitted by multiple mappers running in a single physical node. All-Reduce combine processing can be performed in real time when the data is received.

8.6.4 Implementations

In this section we present two implementations of the Map-AllReduce primitive.

These implementations are proofs of concept to show the advantages achieved by using the Map-Reduce primitive. It's possible to further optimize these implementations using more advanced algorithms, based on the environment they will be executing, the scale of the computations, and the data sizes as shown in MPI collective communications literature [2]. One of the main advantages of these primitives is the ability to improve the implementations without any need to change the user application implementations, leaving us the possibility of optimizing these implementations in the future.

Current implementations use n'ary tree based hierarchical reductions. Other algorithms to implement AllReduce include flat-tree/linear, pipeline, binomial tree, binary tree, and k-chain trees[57].

8.6.4.1 Twister4Azure Map-AllReduce

Current implementation uses a hierarchical processing approach where the results are first aggregated in the local node and then final assembly is performed in the destination nodes. A single Azure worker node may run several Map workers and many more map tasks belonging to the computation. Hence the Twister4Azure Map-AllReduce implementation maintains a worker node-level cache of the AllReduce result values that would be available to any map task executing on that node.

The iteration check happens in the destination nodes and can be specified as a custom function or as a maximum number of iterations.

8.6.4.2 H-Collectives Map-AllReduce

H-Collectives Map-AllReduce primitive is implemented on top of Apache Hadoop using node-level local aggregation and using the Netty NIO library to broadcast the locally aggregated values to the other worker nodes of the computation. A single worker node may run several Map workers and many more map tasks belonging to the computation. Hence the Hadoop Map-AllReduce implementation maintains a node-level cache of the AllReduce result values. The final reduce combine operation is performed in each of the worker nodes and is done after all the Map tasks are completed and the data is transferred.

8.7 Evaluation

Table 9 Execution environments

IU Alamo cluster : Dual Intel Xeon X5550 (8 total cores) per node, 12GB Ram per node, 1Gbps network
Azure Extra Large : 8 cores, 14GB Ram per instance

8.7.1 Multi-Dimensional Scaling using Map-AllGather

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[24]. Dimensional scaling is used mainly in the visualizing of high-dimensional data by mapping them onto

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 and identify relationships.

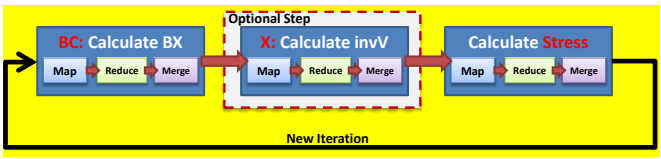


Figure 7. Twister4Azure Multi-Dimensional Scaling

In this paper, we use Scaling by MAjorizing a COmplicated Function (SMACOF)[25], 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. In this paper, we implement the parallel SMACOF algorithm described by Bae et al[20]. This results in iterating a chain of three MapReduce jobs, as depicted in Figure 6. For the purposes of this paper, we perform an unweighted mapping that results in two MapReduce job steps per iteration, BCCalc and StressCalc. MDS is challenging for Twister4Azure due to its relatively finer-grained task sizes and multiple MapReduce applications per iteration.

Each BCCalc Map task generates a portion of the total X matrix. The reduce step of MDS BCCalc computation is a simple aggregation operation, where the reduction simply assembles the outputs of the Map tasks together in order. This X value matrix then needs to be broadcasted in order to be used in the StressCalc step of the current iterations as well as in the BCCalc step of the next iteration. The compute and communication flow of MDS BCCalc computation matches very well with the Map-AllGather primitive. Usage of the Map-AllGather primitive in MDS BCCalc computation eliminates the need for reduce, merge and the broadcasting steps in that particular computation.

8.7.1.1 MDS BCCalculation Step Cost

For the simplicity, in this section we assume each MDS iteration contains only the BCCaculation step and analyze the cost of MDS computation.

Map compute cost can be approximated for large *n* to $d*n^2$, where *n* is the number of data points and *d* is the dimensionality of the lower dimensional space. Input data points in MDS are *n* dimensional (*n*n* matrix). The total input data size for all the map tasks would be n^2 and the loop invariant data size would be *n*d*.

MDS approximate compute and communications cost when using the AllGather primitive,

$$T_{MDS-BCCalc-AllGather} = \lambda' + \frac{Wn^2}{m}(\delta + d) + \log(p)\,\alpha + \frac{p-1}{p} nd\beta$$

In MDS, the number of computations per *l* bytes of the input data are in the range of *k*l*d*, where *k* is a constant and *d* is typically 3. Hence MDS has larger data loading and memory overheads compared to the number of computations.

8.7.1.2 Twister4Azure MDS-AllGather

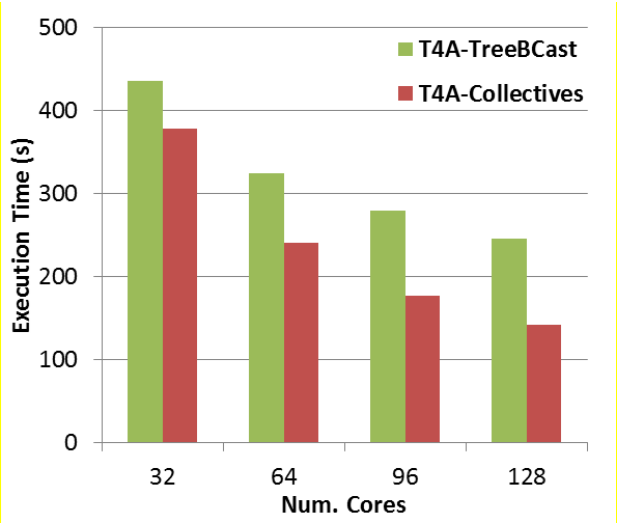


Figure 8. MDS application implemented using Twister4Azure. 20 iterations. 51200 data points (~5GB).

We implemented the Multi-Dimensional Scaling application for Twister4Azure using Map-AllGather primitive and MapReduce-MergeBroadcast with optimized broadcasting. Twister4Azure optimized broadcast version is an improvement over vanilla MapReduce where it uses an optimized tree-based algorithm to perform TCP broadcasts of in-memory data. Figure 10 shows the MDS strong scaling performance results comparing the Twister4Azure Map-AllGather based implementation with the MapReduce-MergeBroadcast implementation. This test case scales a 51200*51200 matrix and in to a 51200*3 matrix. The test is performed on Windows Azure cloud using Azure extra-large instances[table 1]. The number of map tasks per computation is equal to the number of total cores of the computation. The Map-AllGather based implementation improves the performance of Twister4Azure MDS over MapReduce with optimized broadcast by 13% up to 42% for the current test cases.

8.7.1.3 H-Collectives MDS-AllGather

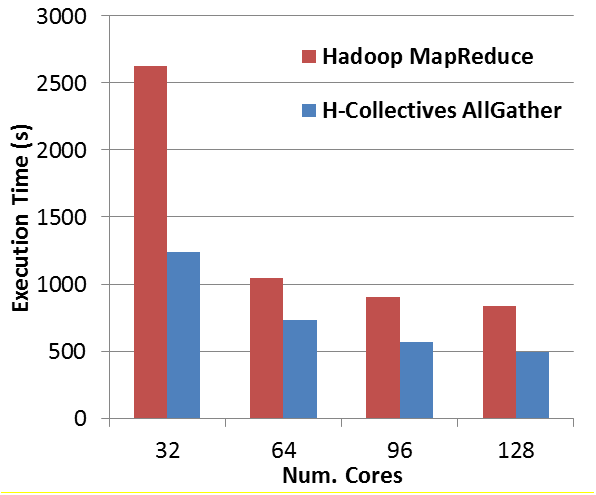


Figure 9. MDS application implemented using Hadoop. 20 iterations. 51200 data points (~5GB).

We implemented the Multi-Dimensional Scaling application for Hadoop using vanilla MapReduce and using H-Collectives Map-AllGather primitive. Vanilla MapReduce implementation uses the Hadoop DistributedCache to broadcast loop variant data to the Map tasks. Figure 11 shows the MDS strong scaling performance results comparing Map-AllGather based implementation with the MapReduce

implementation. This test case scales a 51200*51200 matrix and in to a 51200*3 matrix. The test is performed on the IU Alamo cluster. The number of map tasks per computation is equal to the number of total cores of the computation. The Map-AllGather based implementation improves the performance of MDS over MapReduce by 30% up to 50% for the current test cases.

1.1.1.1.1 Detailed analysis of overheads

In this section we perform detailed analysis of overheads of the Hadoop MDS BCalc calculation using a histogram of executing Map Tasks. In this test, we use only the BCCalc MapReduce job and removed the StressCalc step to show the overheads. MDS computations depicted in the graphs of this section use 51200 *51200 data points, 6 Iterations on 64 cores using 64 Map tasks per iteration. The total AllGather data size of this computation is 51200*3 data points. Average data load time is 10.61 seconds per map task. Average actual MDS BCCalc compute time is 1.5 seconds per map task.

These graphs plot the total number of executing Map tasks at a given moment of the computation. Number of an executing Map tasks approximately represent the amount of useful work done in the cluster at that given moment. The resultant graphs comprise of blue bars that represent an iteration of the computation. The width of each blue bar represents the time spent by Map tasks in that particular iteration. This includes the time spent loading Map input data, Map calculation time and time to process and store Map output data. The space between the blue bars represents the overheads of the computation.

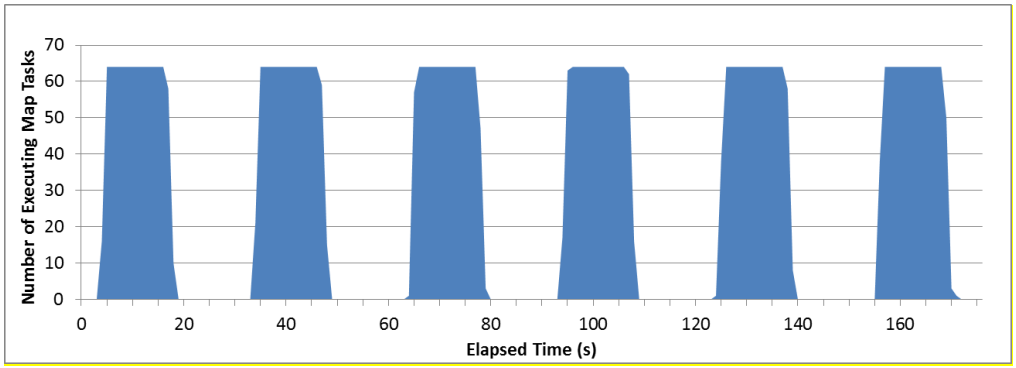


Figure 40 Hadoop MapReduce MDS-BCCalc histogram

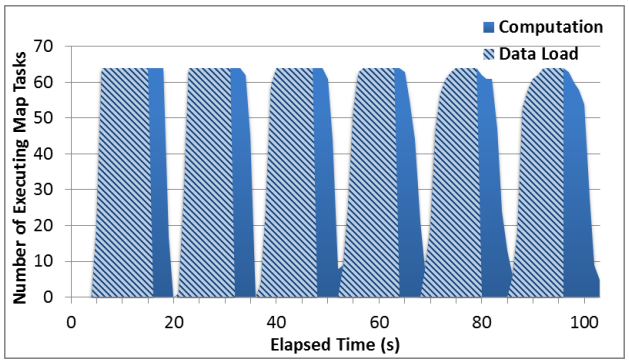


Figure 41 H-Collectives AllGather MDS-BCCalc histogram

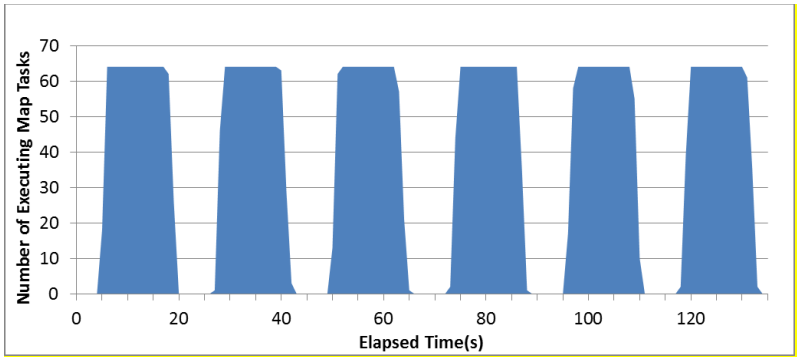


Figure 42 H-Collectives AllGather MDS-BCCalc histogram without speculative scheduling

Figure 40 presents the histogram of MDS using Hadoop MapReduce. Hadoop MapReduce driver program performs the iterations by scheduling a new MapReduce job per iteration. We used Hadoop distributed cache to broadcast the loop variant data. Overheads of this computation include shuffle,

reduce, task scheduling and data broadcast using Hadoop DistributedCache. The average overhead per iteration in this computation is approximately 14 seconds.

Figure 41 presents MDS using H-Collectives AllGather implementation. Hadoop driver program performs speculative (overlap) scheduling of iterations by scheduling the tasks for the next iteration while the previous iteration is still executing and the scheduled tasks wait for the AllGather data to start the actual execution. Blue bars represent the map task time of each iteration, while the stripped section on each blue bar represent the data loading time (time it takes to read input data from HDFS). Overheads of this computation include Allgather communication and task scheduling. MapReduce job for the next iteration is scheduled while the previous iteration is executing and the scheduled tasks wait for the AllGather data to start the execution. As we can notice, the overheads between the iterations virtually disappear with the use of AllGather primitive.

Figure 42 presents MDS using H-Collectives AllGather implementation without the speculative (overlap) scheduling. In this graph, the MapReduce job for the next iteration is scheduled after the previous iteration is finished. This figure compared to Figure 41 shows the gains that can be achieved by enabling optimized task scheduling with the help from the information from collective communication operations. Hadoop MapReduce implementation can't overlap the iterations as we need to add the loop variant data (only available after the previous iteration is finished) to the Hadoop DistributedCache when scheduling the Job.

8.7.1.4 Twister4Azure vs Hadoop

Twister4Azure is already optimized for iterative MapReduce[60] and contains very low scheduling, data loading and data communication overheads compared to Hadoop. Hence, the overhead reduction we achieve by using collective communication is comparatively less in Twister4Azure compared to

Hadoop. Also a major component of Hadoop MDS Map task cost is due to the data loading, as you can notice in Figure 41. Twister4Azure avoids this cost by using data caching and cache aware scheduling.

8.7.2 K-meansClustering using Map-AllReduce

The K-means Clustering[61] algorithm has been widely used in many scientific and industrial application areas due to its simplicity and applicability to large datasets. We are currently working on a scientific project that requires clustering of several TeraBytes of data using K-means Clustering 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 a typical MapReduce implementation, the 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 K-means Clustering, as each Map Task outputs data equivalent to the size of the centroids in each iteration.

K-means Clustering centroid update step is an AllReduce computation. In this step all the values (data points assigned to a certain centroid) belonging to each key (centroid) needs to be combined independently and the resultant key-value pairs (new centroids) are distributed to all the Map tasks of the next iteration.

8.7.2.1 KMeansClustering Cost

KMeans centroid assignment step (Map tasks) cost can be approximated for large n to *n*c*d*, where *n* is the number of data points, *d* is the dimensionality of the data and *c* is the number of centroids. The total input data size for all the map tasks would be *n*d* and the loop invariant data size would be *c*d*.

KMeansClsutering approximate compute and communications cost when using the AllReduce primitive is as follows. The cost of the computation component of AllReduce is *k*c*d,* where k is the number of data sets reduced at that particular step.

$$T_{KM-AR} = \boldsymbol{\lambda}^{'} + \frac{W}{m}(\delta nd + ncd) + \log(p)\,(\alpha + cd\beta + cd) + \frac{m}{p}cd$$

In KMeansClustering, the number of computations per *l* bytes of the input data are in the range of *k*l*c*, where *k* is a constant and *c* is the number of centroids. Hence for non-trivial number of centroids, KMeansClustering has relatively smaller data loading and memory overheads vs the number of computations compared to the MDS application discussed above.

Following is the cost of KMeansClustering using the MapReduce-MergeBroadcast model. KMeansClustering does not need the sorting step and we avoid the cost of sorting for simplicity.

$$T_{KM-MR} = \boldsymbol{\lambda} + \frac{W}{m}(\delta nd + ncd + mcd\delta) + \frac{m}{W}\alpha + \frac{mcd}{Wr}\beta + \frac{1}{r}(mcd\,\delta + mcd + cd\delta) + r\alpha + cd\beta + \mathrm{c} + (\alpha + \beta cd)\log(p)$$

Following is the cost difference between KMeansClustering MapReduce-MergeBroadcast and Map-AllReduce ($T_{KM\text{-}MR} - T_{KM\text{-}AR}$) implementations. The compute cost difference is equal or slightly in favor of the MapReduce due to the hierarchical reduction performed in the AllReduce implementation. All the other overheads including the startup overhead ($\boldsymbol{\lambda}' \ll \boldsymbol{\lambda}$), disk overhead ($\delta$) and communication overhead(α *and* β)favors the AllReduce based implementation.

$$T_{KM-diff} = \{\boldsymbol{\lambda} - \boldsymbol{\lambda}'\} + \left(1 + m\left(\frac{1}{r} - \frac{1}{p}\right) - \log(p)\right)cd + \left(\frac{m}{r} + W + 1\right)cd\delta + \left(\frac{m}{w} + r\right)\alpha + \left(\frac{m}{Wr} + 1\right)cd\beta$$

8.7.2.2 Twister4Azure KmeansClustering-AllReduce

We implemented the K-means Clustering application for Twister4Azure using the Map-AllReduce primitive, vanilla MapReduce-MergeBroadcast and optimized broadcasting. Twister4Azure tree broadcast version is also an improvement over vanilla MapReduce where it uses an optimized tree-based algorithm to perform TCP broadcasts of in-memory data. The vanilla MapReduce implementation and optimized broadcast implementation uses in-map combiners to perform local aggregation of the values to minimize the size of map-to-reduce data transfers. Figure 10's left shows the K-means Clustering weak scaling performance results, where we scale the computations while keeping the workload per core constant. Figure 10's right side presents the K-means Clustering strong scaling performance, where we scaled the number of cores while keeping the data size constant. Results compared different implementations of Twister4Azure.

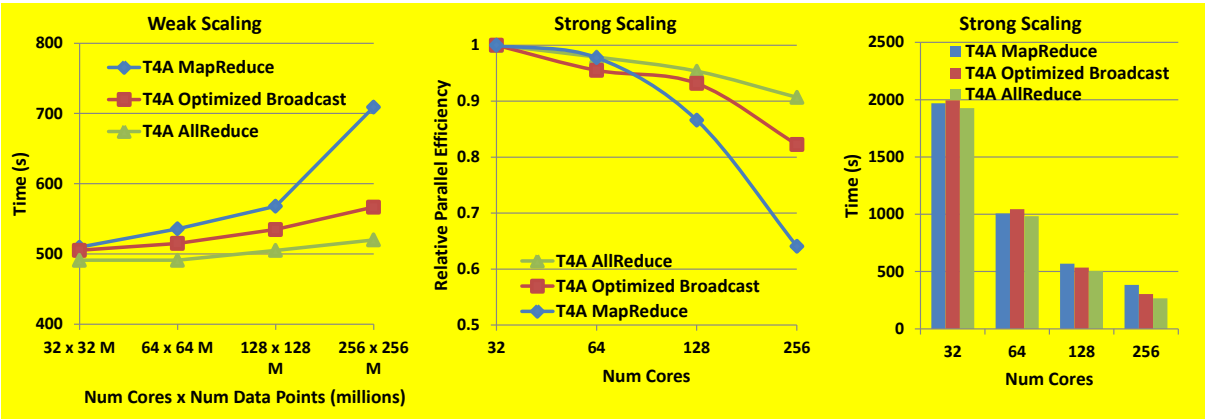


Figure 10. Twister4Azure K-means Clustering comparison with Map-AllReduce. 500 Centroids (clusters). 20 Dimensions. 10 iterations. Left: Weak scaling 32 to 256 Million data points. Right: 128 Million data points. Parallel efficiency relative to the 32 core run time.

8.7.2.3 H-Collectives KMeansClustering-AllReduce

We implemented the K-means Clustering application for Hadoop using the Map-AllReduce primitive and using vanilla MapReduce. The vanilla MapReduce implementation uses in-map combiners to perform aggregation of the values to minimize the size of map-to-reduce data transfers. Figure 11's left side illustrates the Hadoop K-means Clustering weak scaling performance results in which we scale the

computations while keeping the workload per core constant. Figure 11's right side is the Hadoop K-means Clustering strong scaling performance with the number of cores scaled while keeping the data size constant. Strong scaling smaller test cases use more map task waves optimizing the intermediate communication, resulting in relatively smaller overheads for the computation. (HDFS replication factor of 6 increasing data locality)

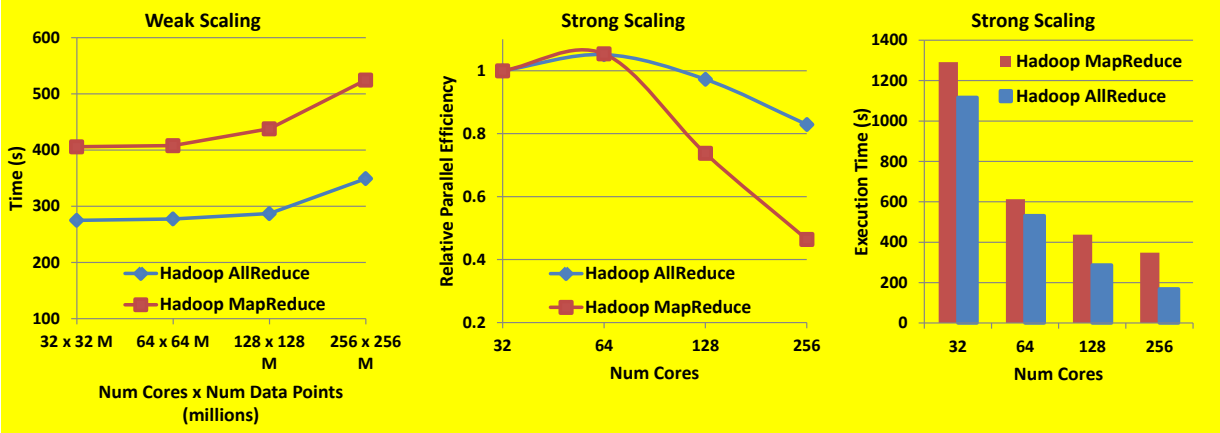


Figure 11. Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce. 500 Centroids (clusters). 20 Dimensions. 10 iterations. Left: Weak scaling 32 to 256 Million data points. Right: 128 Million data points. Parallel efficiency relative to the 32 core run time.

1.1.1.1.2 Detailed analysis of overheads

In this section we perform detailed analysis of overheads of the Hadoop KMeansClustering calculation using a histogram of executing Map Tasks. MDS computations depicted in the graphs of this section use 64 million data points, 20 dimensions, 500 centroids (clusters), 10 Iterations on 64 cores using 128 Map tasks (2 map waves) and 8 reduce tasks (in the case of MapReduce) per iteration. The total Allreduce data size of this computation is 500*20*128 data points. Average data load time is 2 seconds per map task. Average actual KMeans Map compute time is 6.3 seconds per map task.

1.1.1.1.3 Detailed analysis of overheads

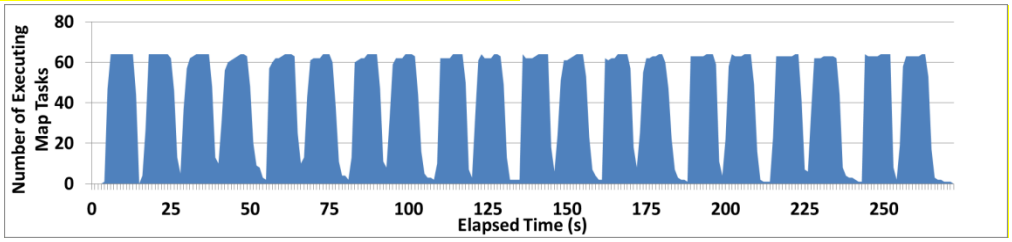


Figure 12. Hadoop MapReduce KMeans histogram

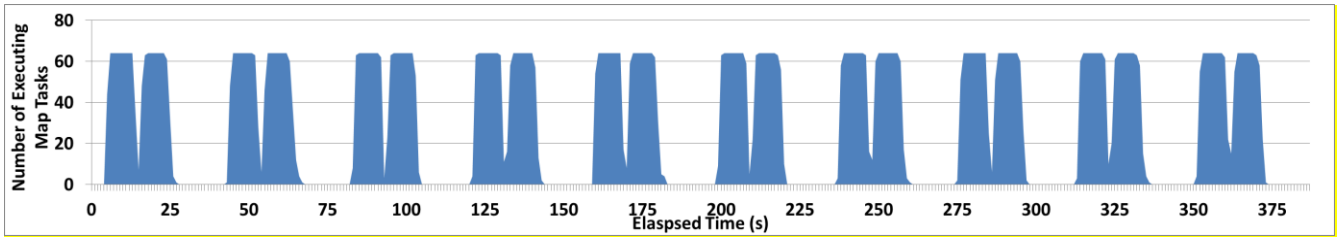


Figure 13. H-Collectives AllReduce KMeans 64 cores.

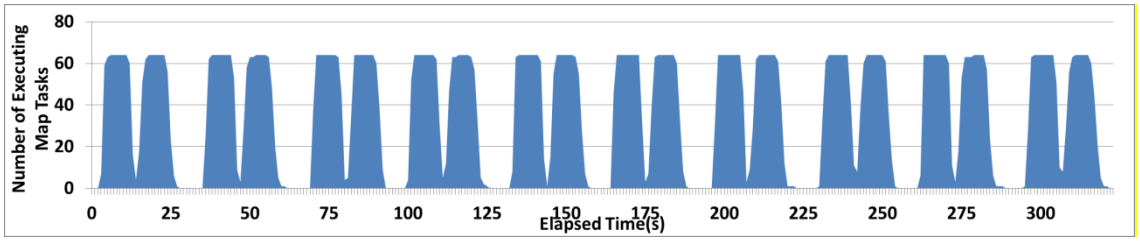


Figure 14. H-Collectives AllReduce KMeans with no speculative scheduling

8.7.2.4 Twister4Azure vs Hadoop

KMeans performs more computation per data load than MDS.

9. CONCLUSION AND FUTURE WORKS

9.1 Summary

Summary of chapters..

Programming Model	1. MapReduce programming model extended to support iterative applications. Supports Pleasingly parallel, MapReduce and iterative MapReduce type applications. 2. Iterative MapReduce Collective Communications primitives (eg: AllGather, SumReduce)
Storage	1. Multi-level caching of data to overcome latencies and bandwidth issues of Cloud Storages 2. Hybrid Storage of intermediate data on different cloud storages based on the size of data.
Task Scheduling	1. Global queue based dynamic scheduling 2. Cache aware execution history based scheduling 3. Communication primitive based scheduling
Data Communication	1. Hybrid data transfers using either or a combination of Azure Blob Storage, Azure Tables and direct TCP communication. 2. Data reuse across applications, reducing the amount of data transfers 3. Collective communication primitives to improve the communication of iterative MapReduce applications.
Fault tolerance	1. Task level fault-tolerance. Re-execute the failed tasks 2. Decentralized control avoiding single point of failures 3. Re-execution of slow tasks 4. Hybrid data communication utilizing a combination of faster non-persistent and slower persistent mediums
Scalability	1. Hybrid data transfers to overcome Azure service scalability issues 2. Hybrid scheduling to reduce scheduling overhead with increasing amount of tasks and compute resources. 3. Primitives optimize the inter-process data communication and coordination. 4. Decentralized architecture facilitates dynamic scalability and avoids single point bottlenecks.
Efficiency	1. Execution history based scheduling to reduce scheduling overheads 2. Multi-level data caching to reduce the data staging overheads 3. Direct TCP data transfers to increase data transfer performance 4. Support for multiple waves of map tasks improving load balancing as well as allows the overlapping communication with computation.
Monitoring, Logging and	1. Web based monitoring console supporting task and job monitoring 2. Azure Table based persistent meta-data storage

Metadata storage *	3. CPU/memory usage monitoring
Cost Effective*	1. Ensuring good utilization of the instances 2. Supporting all the instance types, so that the users can choose the appropriate instances for the use case 3. Support opportunistic environments, such as Amazon EC2 spot instances.
Ease of use*	1. Extends the easy-to-use familiar map reduce programming model 2. Supports local debugging and testing of applications 3. Framework managed fault-tolerance 4. Collective operations allowing users to more naturally translate applications to the iterative MapReduce programming model. Collective operations also free the users from the burden of implementing these operations manually.

* We did not pursue monitoring, logging & metadata storage, cost effectiveness and ease of usage as research issues in our current research. However, the solutions and frameworks that are mentioned in the table that we developed provide and in some cases improve the industry standard solutions for each issue.

9.2 Contributions

- Architecture and programming model to performing pleasingly parallel computations on cloud using cloud infrastructure services.
- Decentralized architecture and programming model to performing MapReduce and iterative map reduce computations on cloud using cloud infrastructure services.
- Prototype implementation for Azure cloud
- Introducing collective communications to MapReduce model. Implementation on Twister4Azure and Hadoop.
- Multi-level caching strategy to improve cloud performance.
- Cache aware scheduling
- Hybrid data transfers

- Applications performance studies using various run time environments including our prototype implementations.
- Inhomogeneous data study.

To be expanded…….

9.3 Future Work

To be expanded…….

9.4 Conclusions

Pleasingly parallel

MapReduce

Inhomogeneous

Iterative

Collective communications

To be expanded…….

9.5 List of publications related to this thesis

Journal, conference and workshop papers

[1] **T. Gunarathne**, T.-L. Wu, J. Y. Choi, S.-H. Bae, and J. Qiu, "Cloud computing paradigms for pleasingly parallel biomedical applications," *Concurrency and Computation: Practice and Experience,* 2011.

[2] **T. Gunarathne**, T.-L. Wu, B. Zhang and J. Qiu, "Scalable Parallel Scientific Computing Using Twister4Azure". Future Generation Computer Systems(FGCS), 2013 Volume 29, Issue 4, pp. 1035-1048.

[3] J. Ekanayake, **T. Gunarathne**, and J. Qiu, "Cloud Technologies for Bioinformatics Applications" *Parallel and Distributed Systems, IEEE Transactions on,* vol. 22, pp. 998-1011, 2011.

[4] **T. Gunarathne**, T.-L. Wu, B. Zhang and J. Qiu, "Portable Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure". 4th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2011). Melbourne, Austrailia. Dec 2011.

[5] **T. Gunarathne**, T. L. Wu, J. Qiu, and G. C. Fox, "MapReduce in the Clouds for Science," presented at the 2nd International Conference on Cloud Computing, Indianapolis, Dec 2010.

[6] **T. Gunarathne**, T.-L. Wu, J. Qiu, and G. Fox, "Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications," presented at the Proceedings of the Emerging Computational Methods for the Life Sciences Workshop of ACM HPDC 2010 conference, Chicago, Illinois, June 2010.

[7] **T. Gunarathne** (Advisor: G. C. Fox). "Scalable Parallel Computing on Clouds". Doctoral Research Showcase at SC11. Seattle. Nov 2011.

[8] J.Ekanayake, H.Li, B.Zhang, **T.Gunarathne**, S.Bae, J.Qiu, and G.Fox., "Twister: A Runtime for iterative MapReduce," presented at the Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010, Chicago, Illinois, 2010.

[9] J. Ekanayake, A. S. Balkir, **T. Gunarathne**, G. Fox, C. Poulain, N. Araujo, and R. Barga, "DryadLINQ for Scientific Analyses," in Fifth IEEE International Conference on eScience: 2009, Oxford, 2009.

[10] **T. Gunarathne**, B. Salpitikorala, A. Chauhan. And G. C. Fox, "Iterative Statistical Kernels on Contemporary GPUs". Int. J. of Computational Science and Engineering (IJCSE), 2013 Vol.8, No.1, pp.58 - 77.

[11] **T. Gunarathne**, B. Salpitikorala, A. Chauhan. And G. C. Fox, “Optimizing OpenCL Kernels for Iterative Statistical Applications on GPUs”. 2nd International Workshop on GPUs and Scientific Applications, Oct 2011.

Planned and/or under review papers

[12] **T. Gunarathne** and J. Qui, "Collective Communication Primitives for Iterative MapReduce," Technical report, July 2013.

Other (book chapters, posters, presentations)

[13] **T. Gunarathne**, J. Qui, and G. Fox, "Iterative MapReduce for Azure Cloud," presented at the Cloud Computing and Its Applications, ANL, Chicago, IL, Apr 2011.

[14] J. Ekanayake, X. Qiu, **T. Gunarathne**, S. Beason and G.C. Fox. “High Performance Parallel Computing with Clouds and Cloud Technologies”. Book chapter in Cloud Computing and Software Services: Theory and Techniques, CRC Press (Taylor and Francis), ISBN-10: 1439803153.

[15] J. Qiu and **T. Gunarathne** "Twister4Azure: Parallel Data Analytics on Azure" presented at the Cloud Futures Workshop 2012, Berkeley, CA, Apr 2012.

10. REFERENCES

[1] J. Dean, and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM,* vol. 51, no. 1, pp. 107-113, 2008.

[2] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for Data Intensive Scientific Analyses." pp. 277-284.

[3] Qiu X., Ekanayake J., Gunarathne T. *et al.*, "Using MapReduce Technologies in Bioinformatics and Medical Informatics."

[4] C. Evangelinos, and C. N. Hill, "Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2.," in Cloud computing and it's applications (CCA-08), Chicago, IL, 2008.

[5] *Apache Hadoop*, Retrieved April 20, 2010, from ASF: http://hadoop.apache.org/core/.

[6] Y. Yu, M. Isard, D. Fetterly *et al.*, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in Symposium on Operating System Design and Implementation (OSDI), San Diego, CA, 2008.

[7] J.Ekanayake, H.Li, B.Zhang *et al.*, "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, Chicago, Illinois, 2010.

[8] Y. Bu, B. Howe, M. Balazinska *et al.*, "HaLoop: Efficient Iterative Data Processing on Large Clusters," in The 36th International Conference on Very Large Data Bases, Singapore, 2010.

[9] T. Gunarathne, B. Zhang, T.-L. Wu *et al.*, "Portable Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure," in 2011 Fourth IEEE International Conference on Utility and Cloud Computing (UCC), Melbourne, Australia, 2011.

[10] *Microsoft Daytona*, Retrieved Feb 1, 2012, from : http://research.microsoft.com/en-us/projects/daytona/.

[11] M. Zaharia, M. Chowdhury, M. J. Franklin *et al.*, "Spark: Cluster Computing with Working Sets," in 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10), Boston, 2010.

[12] "Windows Azure Compute," July 25th 2011; http://www.microsoft.com/windowsazure/features/compute/.

[13] *Amazon Web Services*, vol. 2010, Retrieved April 20, 2010, from Amazon: http://aws.amazon.com/.

[14] Amazon, *Amazon Web Services*.

[15] Y. Gu, Grossman, R, "Sector and Sphere: The Design and Implementation of a High Performance Data Cloud," *Crossing boundaries: computational science, e-Science and global e-Infrastructure I. Selected papers from the UK e-Science All Hands Meeting 2008 Phil. Trans. R. Soc. A,* vol. 367, pp. 2429-2445, 2009.

[16] G. C. Christiam Camacho, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer and Thomas L Madden, "BLAST+: architecture and applications," *BMC Bioinformatics 2009, 10:421*, 2009.

[17] J. Ekanayake, T. Gunarathne, and J. Qiu, "Cloud Technologies for Bioinformatics Applications," *Parallel and Distributed Systems, IEEE Transactions on,* vol. 22, no. 6, pp. 998-1011, 2011.

[18] X. Huang, and A. Madan, "CAP3: A DNA sequence assembly program.," *Genome Res,* vol. *9*, no. *9*, pp. 868-77, 1999.

[19] J. Y. Choi, *Deterministic Annealing for Generative Topographic Mapping GTM*, 2009.

[20] S.-H. Bae, J. Y. Choi, J. Qiu *et al.*, "Dimension reduction and visualization of large high-dimensional data via interpolation," in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Chicago, Illinois, 2010, pp. 203-214.

[21] NCBI, "NCBI Toolkit."

[22] T. F. Smith, and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology,* vol. *147*, no. *1*, pp. 195-197, 1981.

[23] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology,* vol. 162, pp. 705-708, 1982.

[24] J. B. Kruskal, and M. Wish, *Multidimensional Scaling*: Sage Publications Inc., 1978.

[25] J. de Leeuw, "Convergence of the majorization method for multidimensional scaling," *Journal of Classification,* vol. 5, pp. 163-180, 1988.

[26] *cloudmapreduce*, vol. 2010, Retrieved April 20, 2010: http://code.google.com/p/cloudmapreduce/.

[27] "AppEngine MapReduce," July 25th 2011; http://code.google.com/p/appengine-mapreduce.

[28] Wei Lu, Jared Jackson, and 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), Chicago, IL, 2010.

[29] A. Dave, W. Lu, J. Jackson *et al.*, "CloudClustering: Toward an iterative data processing pattern on the cloud."

[30] "*Hadoop Distributed File System HDFS*," December, 2009; http://hadoop.apache.org/hdfs/.

[31] J.Ekanayake, H.Li, B.Zhang *et al.*, “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, Chicago, Illinois, 2010.

[32] Bingjing Zhang, Yang Ruan, Tak-Lon Wu *et al.*, “Applying Twister to Scientific Applications,” in CloudCom 2010, IUPUI Conference Center Indianapolis, 2010.

[33] Apache, "ActiveMQ," http://activemq.apache.org/, 2009].

[34] E. Chan, M. Heimlich, A. Purkayastha *et al.*, “Collective communication: theory, practice, and experience,” *Concurrency and Computation: Practice and Experience,* vol. 19, no. 13, pp. 1749-1783, 2007.

[35] M. Isard, M. Budiu, Y. Yu *et al.*, “Dryad: Distributed data-parallel programs from sequential building blocks,” in ACM SIGOPS Operating Systems Review, 2007, pp. 59-72.

[36] J. Lin, and M. Schatz, “Design patterns for efficient graph algorithms in MapReduce,” in Proceedings of the Eighth Workshop on Mining and Learning with Graphs, Washington, D.C., 2010, pp. 78-85.

[37] E. Walker, “Benchmarking Amazon EC2 for high-performance scientific computing,”*;login: The USENIX Magazine,* vol. 33, no. 5.

[38] J. Ekanayake, and G. Fox, "High Performance Parallel Computing with Clouds and Cloud Technologies."

[39] J. Ekanayake, T. Gunarathne, and J. Qiu, *Cloud Technologies for Bioinformatics Applications*, Indiana University, 2010.

[40] M. C. Schatz, “CloudBurst: highly sensitive read mapping with MapReduce,” *Bioinformatics,* vol. 25, no. 11, pp. 1363-1369, 2009.

[41] A. Matsunaga, M. Tsugawa, and J. Fortes, "CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications ".

[42] W. Edward, “The Real Cost of a CPU Hour,” vol. 42, pp. 35-41, 2009.

[43] J. Wilkening, A. Wilke, N. Desai *et al.*, "Using clouds for metagenomics: A case study." pp. 1-6.

[44] *Windows Azure Platform*, Retrieved April 20, 2010, from Microsoft: http://www.microsoft.com/windowsazure/.

[45] J. Varia, *Cloud Architectures*, Amazon Web Services. Retrieved April 20, 2010 : http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf.

[46] D. Chappell, *Introducing Windows Azure*, December, 2009: http://go.microsoft.com/?linkid=9682907.

[47] Ananth Grama, George Karypis, Vipin Kumar *et al.*, *Introduction to Parallel Computing*: Addison Wesley (Second Edition), 978-0201648652, 2003.

[48] Gustavo Alonso, Fabio Casati, Harumi Kuno *et al.*, *Web Services: Concepts, Architectures and Applications (Data-Centric Systems and Applications)*: Springer Verlag, 978-3642078880, 2010.

[49] "Apache ActiveMQ open source messaging system," http://activemq.apache.org/.

[50] X. Huang, and A. Madan, "CAP3: A DNA sequence assembly program," *Genome Res,* vol. 9, no. 9, pp. 868-77, 1999.

[51] "MPI," *Message Passing Interface*, http://www-unix.mcs.anl.gov/mpi/, 2009].

[52] J. Ekanayake, T. Gunarathne, J. Qiu *et al.*, "Cloud Technologies for Bioinformatics Applications," *Accepted for publication in Journal of IEEE Transactions on Parallel and Distributed Systems*, 2010.

[53] "JAligner.," December, 2009; http://jaligner.sourceforge.net.

[54] Ekanayake J, Qiu X, Gunarathne T *et al.*, "High Performance Parallel Computing with Clouds and Cloud Technologies," *Cloud Computing and Software Services: Edited by Ahson S, Ilyas M,* vol. CRC Press, 2009.

[55] P. Barham, B. Dragovic, K. Fraser *et al.*, "Xen and the art of virtualization." pp. 164-177.

[56] L. Youseff, R. Wolski, B. Gorda *et al.*, "Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems," in Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing, 2006, pp. 1.

[57] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca *et al.*, "Performance analysis of MPI collective operations." p. 8 pp.

[58] R. W. Hockney, "The communication challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Computing,* vol. 20, no. 3, pp. 389-398, 3//, 1994.

[59] H. Herodotou, "Hadoop Performance Models," http://www.cs.duke.edu/starfish/files/hadoop-models.pdf, 2011].

[60] T. Gunarathne, B. Zhang, T.-L. Wu *et al.*, "Scalable parallel computing on clouds using Twister4Azure iterative MapReduce," *Future Generation Computer Systems,* vol. 29, no. 4, pp. 1035-1048, 6//, 2013.

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