SCALABLE PARALLEL COMPUTING ON CLOUDS:

EFFICIENT AND SCALABLE ARCHITECTURES TO PERFORM PLEASINGLY PARALLEL, MAPREDUCE AND ITERATIVE DATA INTENSIVE COMPUTATIONS ON CLOUD ENVIRONMENTS

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
May 2014

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

DOCTORAL COMMITTEE

GEOFFREY FOX, PH.D.

(PRINCIPAL ADVISOR)

BETH PLALE, PH.D.

DAVID LEAKE, PH.D.

JUDY QIU, PH.D.

April 21, 2014

DEDICATION

This thesis is dedicated to my dear parents, my lovely wife Bimalee and my lovely son Kaveen.

ACKNOWLEDGEMENT

First and foremost, I express my sincere gratitude to my advisor, Prof. Geoffrey Fox, for his invaluable advice, knowledge sharing and inspiration for this thesis as well as for my other research endeavors. This thesis would have been impossible without his research insights and guidance. I consider it an honor to have worked under him.

I would also like to thank my research committee members, Prof. Beth Plale, Prof.David Leake and Prof. Judy Qiu for the invaluable support and guidance for my research and for their comments to improve this thesis. I'm very grateful to my previous advisor Prof. Dennis Gannon for giving me the opportunity to start my graduate studies and for the excellent help and advice he gave me. I would also like to thank Prof.Beth Plale one more time for the excellent help she gave me during the transition between the advisors and for co-advising me during that period.

I owe lot of gratitude for Dr. Sanjiva Weerawarana for encouraging me to undertake the challenge of graduate studies and providing his help and guidance throughout the process. Thanks for being an inspiration and a role model for myself as well as for countless number of other Sri Lankan students. I would also like to thank Prof. Arun Chauhan for advising me in my GPU related research and spending his valuable time to help me.

I would like to thank Persistent Systems for providing me with a fellowship to fund the last two years of my PhD. I also would like to thank Microsoft for the Azure compute and storage grants that enabled a major part of our research presented in this thesis.

It has been an absolute pleasure to work with my brilliant colleagues at the SALSA research group and at the CGL lab. I would like to thank past and present SALSA and CGL group members, Tak-Lon Wu, Saliya Ekanayaka, BingJing Zang, Jerome Michel, Ruan Yang, Xiaoming Guo, Jaliya Ekanayaka, Jong Choi,

Seung-Hee Bae, Hui Li, Fei Tang, Yudou Zhou, Scott Beason and Andrew Young for their collaboration and insightful comments to my research.

I would also like to thank Suresh Marru and the past Extreme Lab colleagues for the help and support during first two years of my studies as well as after that. I'm really fortunate to be part of two lab families.

I would also like to thank Milinda Pathirage, Saliya Ekanayaka and the Sri Lankan community in Bloomington for making our time at Bloomington a wonderful experience.

My gratitude goes to my manager Dr.Chris Groer and the Customer Analytics Group at KPMG LLP (formerly Link Analytics) for trusting me to provide me with an exciting employment opportunity and for providing the flexibility to work on the final parts of my thesis.

I would like to thank all the professors at School of Informatics and Computing for the excellent education I received. Also thanks goes to the administrative staff of Pervasive Technology Institute, especially Gary Miksik and Mary Nell Shiflet, and the staff of School of Informatics and Computing for providing continual help during my Ph.D. study. I would like to thank Grace Waitman for all her help on proof reading this thesis.

Last but not least, I would like to thank my loving family for all the support and encouragement they provide me throughout my life. I'm deeply indebted to my dear late parents, without whose love, guidance and encouragement, I would not be where I am today. Thanks to my loving wife Bimalee and my loving son Kaveen for putting up with me for all the missing family times and for providing me with the love and encouragement throughout my research. Words cannot express what both of you have given me!

THILINA GUNARATHNE

SCALABLE PARALLEL COMPUTING ON CLOUDS:
EFFICIENT AND SCALABLE ARCHITECTURES TO PERFORM PLEASINGLY PARALLEL, MAPREDUCE AND ITERATIVE DATA INTENSIVE COMPUTATIONS ON CLOUD ENVIRONMENTS

Over the last decade, three major disruptive trends driven by the software industry altered the scalable parallel computing landscape. These disruptions are the data deluge (i.e., shift to data-intensive from compute-intensive), next generation compute and storage frameworks based on MapReduce, and the utility computing model introduced by cloud computing environments. This thesis focuses on the intersection of these three disruptions and evaluates the feasibility of using cloud computing environments to perform large-scale, data-intensive computations using next-generation programming and execution frameworks. The current key challenges for performing scalable parallel computing in cloud environments include identifying suitable application patterns, identifying efficient and easy-to-use programing abstractions to represent those patterns, performing appropriate task partitioning and task scheduling, identifying suitable data storage and staging architectures, utilizing suitable communication patterns, and identifying appropriate fault tolerance mechanisms.

This thesis will identify three types of application patterns that are well suited for cloud environments. Presented first are pleasingly parallel computations, including pleasingly parallel programming frameworks for cloud environments. Secondly, MapReduce-type applications are explored, including a decentralized architecture and a prototype implementation to develop MapReduce frameworks using cloud infrastructure services. Third and finally, data-intensive iterative applications, which encompass many graph processing algorithms, machine-learning algorithms, and more, are considered. We present the Twister4Azure architecture and runtime as a solution for implementation of data-intensive iterative applications in 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 and 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 distributed computations, which leads to many advantages. We also present the applicability of collective communication operations to the iterative MapReduce computations on cloud and cluster environments, enriching these computations with additional application patterns without sacrificing the desirable properties of the MapReduce model. The addition of collective communication operations enhances the iterative MapReduce model by offering many performance improvements and ease-of-use advantages.

TABLE OF CONTENTS

1. Introduction 1

1.1 Statement of research problem 4

1.2 Research Challenges 5

1.3 Thesis contributions 10

1.4 Thesis outline 12

2. Background 15

2.1 Cloud Environments 15

2.1.1 Microsoft Azure platform 15

2.1.2 Amazon AWS 17

2.2 MapReduce 19

2.2.1 MapReduce execution framework 20

2.3 Iterative MapReduce 21

2.4 Execution Frameworks 23

2.4.1 Apache Hadoop 26

2.4.2 Microsoft Dryad 28

2.4.3 Twister 29

2.4.4 Haloop 30

2.4.5 Spark 31

2.4.6 Microsoft Daytona 31

2.4.7 i-MapReduce and PrIter 32

2.4.8 Google Pregel and Apache Giraph 33

2.4.9 Other related cloud execution frameworks 34

2.5 Application types 34

2.5.1 Pleasingly Parallel Applications 34

2.5.2 MapReduce Type Applications 35

2.5.3 Data Intensive Iterative Applications 35

2.5.4 MPI type applications 36

2.6 Applications 36

2.6.1 Cap3 37

2.6.2 Generative Topographic Mapping Interpolation 37

2.6.3 Blast+ sequence search 38

2.6.4 Sequence alignment using SmithWaterman GOTOH (SWG) 39

2.6.5 KMeansClustering 39

2.6.6 Multi-Dimensional Scaling (MDS) 40

2.6.7 Bio sequence analysis pipeline 41

3. Pleasingly parallel computing on cloud environments 43

3.1 Pleasingly parallel application architecture 44

3.1.1 Classic Cloud processing model 44

3.1.2 Pleasingly parallel processing using MapReduce frameworks 45

3.1.3 Usability of the technologies 47

3.2 Evaluation Methodology 49

3.3 Cap3 50

3.3.1 Performance with different EC2 cloud instance types 51

3.3.2 Scalability study 52

3.4 BLAST 54

3.4.1 Performance with different cloud instance types 55

3.4.2 Scalability 57

3.5 GTM Interpolation 59

3.5.1 Application performance with different cloud instance types 60

3.5.2 GTM Interpolation speedup 60

3.6 Summary 62

4. MapReduce type applications on cloud environments 64

4.1 Challenges for MapReduce in the Clouds 65

4.2 MRRoles4Azure (MapReduce Roles for Azure) 67

4.2.1 Client API and Driver 69

4.2.2 Map Tasks 69

4.2.3 Reduce Tasks 70

4.2.4 Monitoring 71

4.2.5 Fault Tolerance 71

4.2.6 Limitations of MRRoles4Azure 72

4.3 Performance evaluation 72

4.3.1 Methodology 72

4.3.2 Smith-Waterman-GOTOH (SWG) pairwise distance calculation 74

4.3.3 Sequence assembly using Cap3 78

4.4 Summary 80

5. Data intensive iterative computations on cloud environments 81

5.1 Twister4Azure – Iterative MapReduce 83

5.1.1 Twister4Azure Programming model 83

5.1.2 Data Cache 87

5.1.3 Cache Aware Scheduling 89

5.1.4 Data broadcasting 91

5.1.5 Intermediate data communication 93

5.1.6 Fault Tolerance 95

5.1.7 Other features 95

5.1.8 Development and current status 96

5.2 Twister4Azure Scientific Application Case Studies 96

5.2.1 Methodology 96

5.2.2 Multi-Dimensional Scaling - Iterative MapReduce 99

5.2.3 KMeans Clustering 102

5.3 Summary 106

6. Performance Implications for Data Intensive Parallel applications on Cloud Environments ... 108

6.1 Inhomogeneous data 108

6.1.1 SW-G Pairwise Distance Calculation 108

6.1.2 CAP3 112

6.2 Virtualization overhead 114

6.2.1 SW-G Pairwise Distance Calculation 115

6.2.2 CAP3 116

6.3 Sustained performance of clouds 117

6.4 Data Caching on Azure Cloud instances for Iterative MapReduce computations 118

6.4.1 Local Storage Based Data Caching 119

6.4.2 In-Memory Data Caching 119

6.4.3 Memory Mapped File Based Data Cache 121

6.5 Summary 123

7. Collective Communications Primitives for Iterative MapReduce 125

7.1 Collective Communication Primitives 127

7.2 MapReduce-MergeBroadcast 130

7.2.1 API 130

7.2.2 Merge Task 130

7.2.3 Broadcast 131

7.2.4 MapReduceMergeBroadcast Cost Model 132

7.2.5 Current iterative MapReduce Frameworks and MapReduce-MergeBroadcast 133

7.3 Collective Communications Primitives for Iterative MapReduce 133

7.3.1 Requirements 134

7.3.2 Advantages 135

7.3.3 Programming model 136

7.3.4 Implementation considerations 137

7.4 Map-AllGather Collective 138

7.4.1 Model 138

7.4.2 Execution model 138

7.4.3 Data Model 139

7.4.4 Cost Model 140

7.4.5 Fault tolerance 141

7.4.6 Benefits 141

7.5 Map-AllReduce Collective 142

7.5.1 Model 142

7.5.2 Fault Tolerance 146

7.5.3 Benefits 146

7.6 Implementations 147

7.6.1 H-Collectives: Map-Collectives for Apache Hadoop 148

7.6.2 Map-Collectives for Twister4Azure iterative MapReduce ... 149

7.6.3 Implementation Considerations on cloud environments ... 150

7.7 Evaluation ... 152

7.7.1 Multi-Dimensional Scaling (MDS) using Map-AllGather ... 153

7.7.2 K-KMeansClustering using Map-AllReduce ... 158

7.8 Summary ... 165

8. Conclusions and Future Works ... 167

8.1 Summary and Conclusions ... 167

8.2 Solutions to the research challenges ... 172

8.3 Future Work ... 177

8.4 List of publications related to this thesis ... 179

9. References ... 181

Curriculum Vitae ... 188

LIST OF FIGURES

Figure 1 A sample MapReduce execution flow 21

Figure 2 Steps of a typical MapReduce computation 21

Figure 3 Structure of a typical data-intensive iterative application. 22

Figure 4 Multi-Dimensional Scaling application architecture using iterative MapReduce 40

Figure 5 Bio sequence analysis pipeline 42

Figure 6 Classic cloud processing architecture for pleasingly parallel computations 45

Figure 7 Hadoop MapReduce based processing model for pleasingly parallel computations........ 46

Figure 8 Cap3 application execution cost with different EC2 instance types........ 51

Figure 9 : Cap3 applciation compute time with different EC2 instance types 52

Figure 10 Parallel efficiency of Cap3 application using the pleasingly parallel frameworks 53

Figure 11 Cap3 execution time for single file per core using the pleasingly parallel frameworks 53

Figure 12 : Cost to process 64 BLAST query files on different EC2 instance types 55

Figure 13 : Time to process 64 BLAST query files on different EC2 instance types 56

Figure 14 Time to process 8 query files using BLAST application on different Azure instance types........ 57

Figure 15 : BLAST parallel efficiency using the pleasingly parallel frameworks 58

Figure 16 : BLAST average time to process a single query file using the pleasingly parallel frameworks 58

Figure 17 : Cost of using GTM interpolation application with different EC2 instance types 59

Figure 18 : GTM Interpolation compute time with different EC2 instance types 60

Figure 19: GTM Interpolation parallel efficiency using the pleasingly parallel frameworks 60

Figure 20 : GTM Interpolation performance per core using the pleasingly parallel frameworks 61

Figure 21 MapReduceRoles4Azure: Architecture for implementing MapReduce frameworks on Cloud environments using cloud infrastructure services 69

Figure 22 Task decomposition mechanism of SWG pairwise distance calculation MapReduce application 74

Figure 23 SWG MapReduce pure performance 75

Figure 24 SWG MapReduce relative parallel efficiency 76

Figure 25 SWG MapReduce normalized performance 76

Figure 26 SWG MapReduce amortized cost for clouds 77

Figure 27 Cap3 MapReduce scaling performance 78

Figure 28 Cap3 MapReduce parallel efficiency 79

Figure 29 Cap3 MapReduce computational cost in cloud infrastructures 79

Figure 30 Twister4Azure iterative MapReduce programming model 85

Figure 31 Cache Aware Hybrid Scheduling 90

Figure 32 Twister4Azure tree based broadcast over TCP with Azure Blob storage as the persistent backup 93

Figure 33 MDS weak scaling. Workload per core is constant. Ideal is a straight horizontal line 99

Figure 34 MDS data size scaling using 128 Azure small instances/cores, 20 iterations 100

Figure 35 Twister4Azure Map Task histogram for MDS 100

Figure 36 Number of executing Map Tasks in the cluster at a given moment. 101

Figure 37 KMeans Clustering Scalability. Relative parallel efficiency of strong scaling. 103

Figure 38 KMeansClustering Scalability. Weak scaling.. 103

Figure 39 Twister4Azure Map Task execution time histogram for KMeans Clustering. 105

Figure 40 Twister4Azure number of executing Map Tasks in the cluster at a given moment 105

Figure 41 Performance of SW-G for randomly distributed inhomogeneous data. 110

Figure 42 Performances of SW-G for skewed distributed inhomogeneous data. 111

Figure 43 Performance of Cap3 for random distributed inhomogeneous data. 113

Figure 44 Performance of Cap3 for skewed distributed inhomogeneous data 114

Figure 45 Virtualization overhead of Hadoop SW-G on Xen virtual machines 115

Figure 46 Virtualization overhead of Hadoop Cap3 on Xen virtual machines 116

Figure 47 Sustained performance of cloud environments for MapReduce type of applications 118

Figure 48 Execution traces of Twister4Azure MDS using in-memory caching. 123

Figure 49 Execution traces of Twister4Azure MDS using Memory-Mapped file based caching 123

Figure 50 MapReduce-MergeBroadcast computation flow 131

Figure 51 Map-Collective primitives 134

Figure 52 Map-AllGather Collective 139

Figure 53 Map-AllReduce collective 143

Figure 54 Example Map-AllReduce with Sum operation 144

Figure 55 MDS Hadoop using only the BC Calculation MapReduce job per iteration 154

Figure 56 MDS application implemented using Twister4Azure 155

Figure 57 Hadoop MapReduce MDS-BCCalc histogram 157

Figure 58 H-Collectives AllGather MDS-BCCalc histogram 157

Figure 59 H-Collectives AllGather MDS-BCCalc histogram without speculative scheduling 157

Figure 60 Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce Weak scaling. 161

Figure 61 Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce Strong scaling 161

Figure 62 Twister4Azure K-means weak scaling with Map-AllReduce 162

Figure 63 Twister4Azure K-means Clustering strong scaling. 163

Figure 64 HDInsight KMeans Clustering compared with Twister4Azure and Hadoop 164

LIST OF TABLES

Table 1 Windows Azure instance types (as of July 2013) 15

Table 2 Sample of Amazon Web Services EC2 on-demand instance types 17

Table 3 Summary of MapReduce frameworks 24

Table 4: Summary of pleasingly parallel cloud framework features 47

Table 5 : Cost Comparison of Cap3 execution among different cloud environments 54

Table 6 Evaluation cluster configurations 98

Table 7 The execution time analysis of a MDS computation with different data caching mechanisms 120

Table 8 Collective communications support in MPI, Hadoop, H-Collectives and Twister4Azure 129

Table 9 Summary of Map-Collectives patterns 137

1. INTRODUCTION

Traditionally, High performance parallel computations using technologies like MPI have been the dominant force of large scale distributed parallel computing. These technologies have been used to implement a plethora of large scale scientific computations very successfully, and they have been driven mainly by the academics. Examples of applications implemented using these technologies include many fluid dynamics computations such as weather predictions and molecular dynamic simulations. Applications implemented using these technologies are typically compute and/or communication bound, as opposed to disk bound, and they have very long execution times. Usually these types of computations are performed by physicists, meteorologists, astro-physists, etc. These technologies typically require specialized interconnects for optimal performance.

However, over the last decade, three interconnected disruptions have happened in the large scale distributed parallel computing landscape, mainly driven by the software industry. These are the emergence of data intensive computing (aka big data), the emergence of the utility computing model introduced by Cloud computing offerings and the emergence of new generation of storage, programming and execution frameworks such as MapReduce.

A. Big Data

The emergence of "Big Data" is fueled by the massive amount of data now flowing through virtually every field of science as well the technology industry. These massive data include the results of massive experiments such as the Large Hadron Collider (LHC), the rapid data produced by equipment such as new generation of sequencing machines, the data generated by the overabundance of sensors, the ever increasing data set of the World Wide Web and many more. Scientists as well as those in the technology industry are reliant more than ever on large scale data and its analysis to uncover valuable information and observations. Jim Gray has noted that, increasingly, scientific breakthroughs will be powered by computing capabilities that support the ability of researchers to analyze massive data sets. Aptly, he dubbed data intensive scientific discovery "the fourth scientific paradigm of discovery [1]." While the users of these data crave more power and greater ease of use to store and process these large data volumes, preprocessing, processing and analyzing these large amounts of data present unique and challenging problems. Often, traditional HPC is not the optimal choice to implement data intensive computations.

B. Cloud Computing

Cloud computing introduces a utility computing model combined with a rich set of cloud infrastructure services offering a very viable environment in which to perform data intensive computations. Cloud computing offerings by major commercial players provide on-demand computational services over the Web, which can be purchased within a matter of minutes by simply using a credit card. The utility computing model of these cloud computing offerings opens up exciting new opportunities for users to perform their data intensive parallel computations. An interesting feature of Cloud computing is the ability to increase the throughput of the computations by horizontally scaling computing resources without incurring any additional overhead costs. This is facilitated by the virtually unlimited resource availability of cloud computing infrastructures, which are backed by the world's largest data centers owned by major commercial players such as Amazon, Google and Microsoft. We expect that the economies of scale enjoyed by cloud providers will translate into lower costs for users. Cloud computing platforms also offer a rich set of distributed cloud infrastructure services including storage, messaging and database services with cloud-specific service guarantees. These services can be leveraged to build and deploy scalable distributed applications on cloud environments. While clouds offer raw computing power combined with cloud infrastructure services offering storage and other services, there is a need for distributed computing frameworks to harness the power of clouds both easily and effectively. At the same time, it should be noted that Clouds offer unique reliability and sustained performance challenges to large scale computations due to the virtualization, multi-tenancy, and non-dedicated commodity connectivity characteristics of the cloud environments. They do not provide the high-speed interconnects needed by high performance frameworks such as MPI. This produces the need for distributed parallel computing frameworks specifically tailored for cloud characteristics to harness the power of clouds both easily and effectively.

C. MapReduce

MapReduce consists of a storage framework, a programming model and an associated execution framework for distributed processing of very large data sets. The MapReduce distributed data analysis framework was originally introduced by Google [2], and it provides an easy-to-use programming model that features fault tolerance, automatic parallelization, scalability and data locality-based optimizations. These features and the simplicity of the programming model allow 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. Due to the 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. MapReduce frameworks are typically not optimized for the best performance or parallel efficiency of small scale applications. The main goals of MapReduce frameworks include framework managed fault tolerance, the ability to run on commodity hardware, the ability to process very large amounts of data and horizontal scalability of compute resources. MapReduce frameworks like Hadoop trade off costs such as large startup overheads, task scheduling overheads and intermediate data persistence overheads for better scalability and reliability. Though introduced by the industry and used mainly in the information retrieval community, it is shown [3-5] that MapReduce frameworks are capable of supporting many scientific application use cases as well, making these frameworks good choices for scientists to easily build large, data-intensive applications that need to be executed within cloud infrastructures. Apache Hadoop [6] and Microsoft DryadLINQ [7] are two such distributed parallel data processing frameworks that support MapReduce type computations.

The work of this thesis focuses on the intersection of the above three disruptions and evaluates the feasibility of Cloud Computing environments to perform large scale data intensive computations using new generation programming and execution frameworks such as MapReduce.

In the following subsections, we present a detailed discussion of some of the challenges to performing scalable parallel computing on clouds and the approaches we propose to solve them. We focus mostly on scientific use cases as the technology industry leads the way with exploring other use cases such as web searching, various recommender systems, and targeted marketing, etc. However, the solutions we propose are equally applicable for scientific use cases as well as for industry use cases.

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. Understand the challenges and bottlenecks to perform scalable parallel computing on cloud environments
2. Propose solutions to the challenges and bottlenecks identified in 1.
3. Develop 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 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

In this section, we discuss the challenges to performing scalable parallel computing on Cloud environments. These challenges ideally need to be addressed by the programming frameworks that we design and develop. The solutions we propose to solve these challenges are summarized in section 8.2 of the summary and conclusions chapter.

1. **Programming model**

One of the most important components of a computational framework is the programming model abstraction. The programming abstraction for scalable parallel computing should have the ability to express a sufficiently large and useful subset of large-scale data intensive computations. At the same time, it should be simple and easy-to-use by the end user developers. Extending a familiar existing model, rather than inventing a new complex model, would be preferable from an end-user perspective. Another requirement is that the programming model should be suitable for efficient execution in cloud environments. An ideal programming abstraction to perform scalable parallel computing on clouds should strive for a balance of the above requirements.

2. Data Storage

Typical cloud storage offerings have large bandwidth and latency limitations due to the data that has been stored in off-instance shared storage infrastructures. Overcoming the bandwidth and latency limitations when accessing large input data products from cloud and other storages would be critical to the performance of data intensive computations in clouds as well as in other environments.

Another challenge is to decide where to store and when to store (or whether to store) the output and intermediate data products of the computation. These decisions will have an impact on the fault tolerance and the performance of the computations.

Cloud environments 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 to get the maximum utilization and efficiency from the cloud resources.

3. Task Scheduling

Task scheduling can have a large impact on the performance of distributed computations, especially if the computation consists of thousands of finer grained tasks. In data intensive distributed computations, the tasks should be scheduled efficiently with an awareness of the data locality, which improves the data bandwidth and with an awareness of the data availability in different locations (e.g., cached data), which potentially improves or eliminates the data transfer overheads.

Task scheduling should support the dynamic load balancing of the computations to ensure optimum usage of the compute resources by possibly avoiding the effects of task and compute resource inhomogeneity. Task scheduling should also support the dynamic scaling of the compute resources to take advantage of the dynamic scaling ability of the cloud environments.

4. Data Communication

Cloud infrastructures are known to exhibit inter-node I/O performance fluctuations (due to a shared network, unknown topology), which affect the data communication performance. Frameworks should be designed with consideration 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

Fault tolerance is a very important component of a large scale computation framework. Large scale computations should have the ability to recover from failures to the parts or tasks of the computation without having to re-run the whole computation, preferably with excellent support from the framework. Hence, the framework we design should ensure 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. Large scale computation frameworks should also handle the stragglers (the tail of slow tasks) to optimize the computations.

Node failures are to be expected whenever large numbers of nodes are utilized for computations. These failures become much more prevalent when virtual instances are running on top of non-dedicated hardware in cloud environments. Cloud programming frameworks should be able to recover from node failures and should avoid single point of failures in the presence of node failures.

6. Scalability

Computations should be able to scale well with the 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 scalability of the computations up to hundreds of instances or cores. Also, computations should be able to scale well with increasing input data sizes as well.

7. Efficiency

The framework should facilitate the optimized execution of the applications by 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 need 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 environments are implemented as shared infrastructures operating by using virtual machines. It is possible for the performance to fluctuate based on 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 based on other issues which can be unique to virtual machine environments. The frameworks should contain mechanisms to handle any tasks that take much longer to complete than others; they should also be able to provide appropriate load balancing in the job level.

8. Monitoring, Logging and Metadata storage *

Users should be provided 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. Hence, it is important to select the granularity of logging and the log storage location by trading off the overhead of logging and the durability of the log location.

The 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 accessibility to avoid a single point of failure and performance bottlenecks.

9. Cost effective *

When performing large scale computations on cloud environments, we need to keep the cost of cloud services to an acceptable amount, while extracting the most out of the cloud services.

Choosing suitable instance types is one aspect of optimizing the costs. Clouds offer users several types of instance options, with different configurations and price points. It is important to select the best matching instance type, both in terms of performance as well as money-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.

In this thesis, we do not focus on the research issues involving monitoring, logging and metadata storage*(9), cost effectiveness*(10) and the ease of usage*(11). However, the solutions and frameworks we have developed as part of this thesis research provide and, in some cases, improve the industry standard solutions for each of these issues.

1.3 Thesis contributions

The following comprise a summary of the contributions of this thesis:

- **Architecture, programming model and implementations to perform pleasingly parallel computations on cloud environments utilizing cloud infrastructure services.**
  - ➢ Designed and implemented an architecture and a programming model to perform pleasingly parallel type computations on cloud environments using both cloud infrastructure services as well as using existing MapReduce frameworks. Implemented several large scale pleasingly parallel applications using the above designed framework, and performed a detailed study of performance and cost of the cloud environments to perform pleasingly parallel computations.

- **Decentralized architecture and implementation to perform MapReduce computations on cloud environments utilizing cloud infrastructure services.**
    - Designed a decentralized, scalable and fault tolerance architecture to perform MapReduce computations on cloud environments using cloud infrastructure services. Developed a prototype implementation of the framework for Microsoft Windows Azure Cloud.
    - Implemented several large scale MapReduce applications and performed a detailed study of the performance and the 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 implementation to perform iterative MapReduce computations on cloud environments utilizing cloud infrastructure services.**
    - Designed a decentralized, scalable and fault tolerant architecture and programming model to efficiently perform data intensive iterative MapReduce computations on cloud environments using cloud infrastructure services. Developed a prototype implementation for Windows Azure cloud.
    - Introduced a multi-level data caching approach to solve the data bandwidth and latency issues of cloud storage services for iterative MapReduce. Designed a high performance low overhead cache aware task scheduling algorithm for iterative applications on Cloud environments.
    - Utilized hybrid data transfer approaches to improve data communication performance on cloud environments without sacrificing fault tolerance capabilities.

    - Implemented several large scale iterative MapReduce applications and performed a detailed study of the performance and the challenges to perform iterative MapReduce type computations on cloud environments, including the effect of multi-level data caching and scheduling policies on the application performance.

- **Map-Collectives collective communication primitives for iterative MapReduce**
    - Introduced All-to-All type collective communication operations to the iterative MapReduce model, and developed prototype implementations of collective communication primitives for Apache Hadoop MapReduce (for local clusters and clouds) and for Twister4Azure iterative MapReduce (for the Windows Azure cloud).
    - Implemented several large scale applications using the Map-Collectives and performed a detailed study of the performance of the above mentioned collective communication primitives in cluster and cloud environments

1.4 Thesis outline

Chapter 2 of this thesis discusses the related works of this thesis, and provides an introduction to the cloud environments and example applications used in this thesis. Chapter 2 also provides a classification of application that is used throughout 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 mechanisms, scheduling strategies, fault tolerance and several other dimensions.

Chapter 3 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 the implementations and the performance of several pleasingly parallel applications using the frameworks that were introduced in this chapter.

Chapter 4 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 Microsoft Windows Azure cloud infrastructure services. MapReduceRoles4Azure architecture successfully leverages 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 5 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 a 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.

Chapter 6 studies some performance implications for executing data intensive computations on cloud environments. These include the study of the 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 on the performance of MapReduce type applications. We also discuss and analyze how various data caching strategies on the Azure cloud environment affect the performance of data intensive iterative MapReduce applications.

Chapter 7 discusses the applicability of All-to-All collective communication operations to Iterative MapReduce without sacrificing the desirable properties of the MapReduce programming model and execution framework such as fault tolerance, scalability, familiar API's and the 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 results to all the workers. The MapReduce-MergeBroadcast model is presented as a canonical model representative of 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 solutions to the research challenges and future work in chapter 8.

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 a service (PaaS) model; it offers the .net runtime as the platform through two programmable roles called Worker Roles and Web Roles. Azure also supports VM roles (beta), which enables the users to deploy virtual machine instances that can support 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 [8].

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. The Azure Table can contain a virtually unlimited number of entities (*aka* records or rows) where a single entity can be as large as 1MB. Entities contain properties (*aka* cells), that can be as large as64KB. 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 support two types of Blobs, Page blobs that are optimized for random read/write operations and Block blobs that are optimized for streaming. The 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 all parties.

2.1.2 Amazon AWS

Amazon Web Services (AWS) [9] 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 instance, which costs the same as the Extra-Large (XL) instance but offers greater CPU power and less memory than XL instances. Table 1 provides a summary of the EC2 instance types used in this thesis. 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	Actual CPU	Cost per

		units	cores	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 transfers.

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 on the web. Data objects in S3 are access controllable and can be organized into buckets. S3 pricing is based on the size of the stored data, the amount of data transferred and the number of API requests.

2.2 MapReduce

MapReduce consists of a storage framework, a programming model and an associated execution framework for distributed processing of very large data sets. The MapReduce distributed data analysis framework was originally introduced by Google [2], and it provides an easy-to-use programming model that features fault tolerance, automatic parallelization, scalability and data locality-based optimizations. Due to its 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. The 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 allow 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 partitions the processing of very large input data in to a set of independent tasks. The MapReduce data model consists mainly of key-value pairs. The MapReduce programming model consists of *map(key$_1$, value$_1$)* function and *reduce(key$_2$, list<value$_2$>)* function, borrowed from the functional programming concepts. The map function operates on every element in the input data set and the reduce function combines and aggregates the output of the map function. Each map function invocation is independent of the others; this allows for parallel executions on different data sets. This property also enables richer fault tolerance implementations. However, users should be careful not to have any side effects from their map functions that will violate the independence property. MapReduce programs written as Map and Reduce functions will be parallelized by the framework and will be executed in a distributed manner.

2.2.1 MapReduce execution framework

MapReduce frameworks are typically not optimized for the best performance or parallel efficiency of small scale applications. The main goals of MapReduce frameworks include framework managed fault tolerance, ability run on commodity hardware, the ability to process very large amounts of data and horizontal scalability of compute resources. MapReduce frameworks like Hadoop trade off 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 into partitions, where each partition would be processed by a single Map task. When a Map Reduce computation has more map tasks than Map slots available in the cluster, the tasks will be scheduled in waves. For example, a computation with 100 Map tasks executing in a cluster of 200 Map slots will execute as approximately 5 Map task waves. Figure 1 depicts a sample MapReduce execution flow with multiple Map tasks waves. Tasks in MapReduce frameworks are scheduled dynamically by taking data locality into 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. MapReduce frameworks typically persist the Map output data in the local disks of the Map nodes.

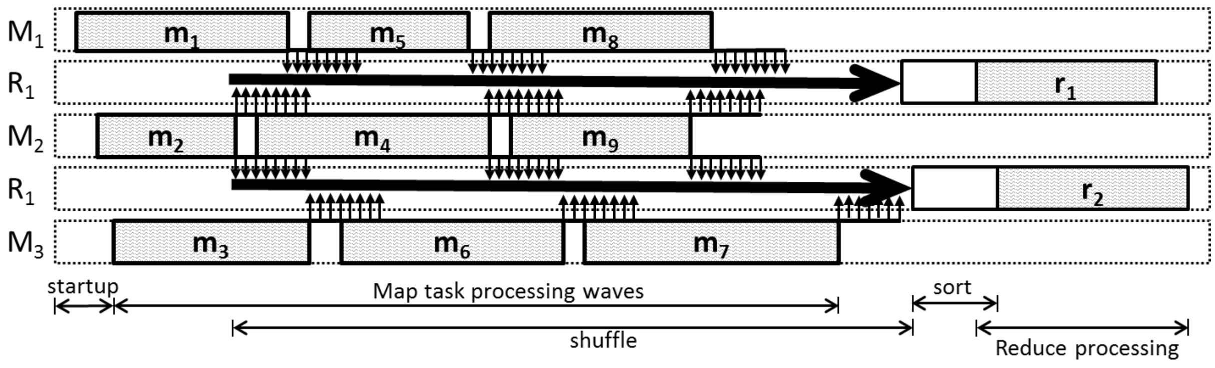


Figure 1 A sample MapReduce execution flow

Reduce tasks fetch the data from the Map nodes and perform an external-merge sort on the data. The fetching of intermediate data starts as soon as the first map task completes the execution. Reduce task starts the reduce function processing after all of the Map tasks are finished and after all the intermediate data are shuffled and sorted. Figure 2 depicts the steps of a typical MapReduce computation.

	Map Task					**Reduce Task**			
Task Scheduling	**Data read**	**Map execution**	**Collect**	**Spill**	**Merge**	**Shuffle**	**Merge**	**Reduce Execution**	**Write output**

Figure 2 Steps of a typical MapReduce computation

2.3 Iterative MapReduce

Many important data-intensive applications and algorithms can be implemented as iterative computation and communication steps, where computations inside an iteration are independent and synchronized at the end of each iteration through reduce and communication steps. Often, each iteration is also amenable to parallelization. Many statistical applications fall into this category, including graph processing, clustering algorithms, data mining applications, machine learning algorithms, data visualization algorithms, and most of the expectation maximization algorithms. Their preeminence is a result of scientists relying on clustering, mining, and dimension reduction to interpret the data. The emergence of computational fields such as bioinformatics and machine learning has also contributed to increased interest in this class of applications.

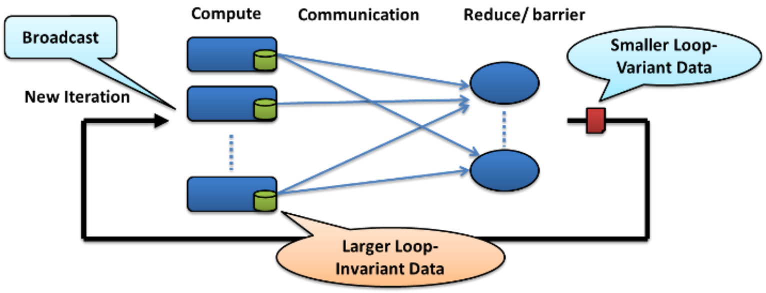


Figure 3 Structure of a typical data-intensive iterative application.

As mentioned in the section above, there exists a significant amount of data analysis, data mining and scientific computation algorithms that rely on iterative computations with which we can easily specify each iterative step as a MapReduce computation. Typical data-intensive iterative computations follow the structure depicted in Figure 3.

We can identify two main types of data in these computations: the very large loop invariant input data and the smaller loop variant delta values. The loop invariant input data would be the set of input data points. Single iterations of these computations are easy to parallelize by processing the data points (or blocks of data points) independently in parallel while performing synchronization between the iterations through communication steps. In a K-means Clustering computation, the loop invariant input data would be the set input data vectors, while in a PageRank calculation, the loop invariant input data would be a representation of the link graph. The loop invariant nature of these input data points gives rise to several optimization possibilities.

Delta values are the result of processing the input data in each iteration. Often, these delta values are needed for the computation of the next iteration. In a K-means Clustering computation, the loop variant delta values are the centroid values. In PageRank calculations, the delta values conform to the page rank vector.

Other general properties of the data-intensive iterative MapReduce calculations include relatively finer-grained tasks resulting in more prominent intermediate I/O overheads and a very large number of tasks due to multiple iterations giving more significance to the scheduling overheads.

Fault Tolerance for iterative MapReduce can be implemented either in the iteration-level or in the task-level. In the case of iteration-level fault tolerance, the check pointing will happen on a per iteration basis, and the frameworks can avoid check pointing the individual task outputs. Due to the finer grained nature of the tasks along with a high number of iterations, some users may opt for higher performance by selecting iteration level fault tolerance. When iteration level fault tolerance is used, the whole iteration would need to be re-executed in case of a task failure. Task-level fault tolerance is similar to the typical MapReduce fault tolerance, and the fault-recovery is performed by execution of failed Map or Reduce tasks.

2.4 Execution Frameworks

Table 3 Summary of MapReduce frameworks

	Google MapReduce[2]	**Apache Hadoop**[6]	**Twister** [10]	**Microsoft Dryad**[11]	**Twister4Azure[12]**
Parallel Model	MapReduce	MapReduce	MapReduce, Iterative MapReduce	DAG execution, Extensible to MapReduce and other patterns	MapReduce, iterative MapReduce
Data Storage	GFS (Google File System)	HDFS (Hadoop Distributed File System)	Local disks	Shared Directories & local disks	Azure Blob Storage
Data Communication	Files	Files over HTTP	Publish/subscribe messaging Framework, TCP, Optimized broadcasts	Files, TCP Pipes, Shared Memory FIFO	Files, Optimized TCP intermediate data transfer and broadcasts, Collective Communication operations

Scheduling	Data/rack Locality	Data/rack Locality, Dynamic task scheduling through global queue	Data Locality, Map task reuse with data caching	Data locality; Topology based, run time graph optimizations	Dynamic task scheduling through global queue, Cache aware scheduling, Collective communication based scheduling
Fault Tolerance	Re-execution of failed tasks; Duplicate execution of slow tasks	Re-execution of failed tasks; Duplicate execution of slow tasks	Re-execution of Iterations	Re-execution of failed tasks; Duplicate execution of slow tasks	Re-execution of failed tasks; Duplicate execution of slow tasks, Re-execution of iterations
Language Support	C++, Sawzall	Java, Hive,Pig Latin, Scalding	Java	C#, DryadLINQ [55]	C#
Runtime Environment	Linux Cluster.	Linux Clusters, Amazon EMR, Azure HDInsights	Linux Cluster	Windows HPCS cluster	Window Azure Compute, Windows Azure Local Development Fabric

2.4.1 Apache Hadoop

Apache Hadoop [6] MapReduce is a widely used open-source implementation of the Google MapReduce [2] distributed data processing framework.

Apache Hadoop MapReduce uses the Hadoop distributed parallel file system (HDFS) [13] 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. The HDFS is designed for deployment on commodity clusters and achieves reliability through replication of data across nodes. The HDFS partitions the data files into coarser grained blocks of 10s or 100s of Megabytes; it stores these blocks on the native file system of the nodes. The block size and the replication factor are configurable as cluster wide as well as for individual data sets. With this HDFS data partitioning and storage strategy, a very large data set or a very large file would effectively get stored in a distributed manner across all or most of the nodes of the cluster, providing very large aggregate read bandwidth when processing the data. The HDFS central NameNode store manages the Meta data of the files stored in HDFS.

When executing Map Reduce programs, Hadoop optimizes data communication by scheduling computations near the data by using the block data locality information provided by the HDFS file system. Hadoop has an architecture consisting of a master node with many client workers, and it 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.

Over the years, Hadoop has grown into a large eco system of projects providing functionalities on top of Hadoop MapReduce and HDFS. These include high level data processing languages such as Apache Hive [14] and Apache Pig [15], Google BigTable [16] like tabular data storage solutions such as Apache HBase [17] and Apache Accumulo [18], large scale machine learning libraries such as Apache Mahout [19], graph processing libraries such as Giraph [20], and data ingesting projects such as Apache Flume [21], Apache Sqoop [22], etc.

2.4.1.1 Amazon Elastic Map Reduce

Amazon Elastic MapReduce (EMR) [23] provides MapReduce as an on-demand service hosted within the Amazon infrastructure. EMR is a hosted Apache Hadoop 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. 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.

2.4.1.2 Azure HDInsight

HDInsight is an Apache Hadoop as service offering hosted within the Microsoft Windows Azure cloud. Similar to Amazon EMR, HDInsight uses Windows Azure instances for computing power and Azure blob storage for long term data storage. Hadoop jobs can be submitted to HDInsights through a web interface or by using the command line of the master node or programmatically through Windows PowerShell. Windows Azure blob storage can be accessed from Hadoop MapReduce computations using a HDFS compatible file system layer (WASBS). Alternatively, users can use the HDFS deployed over the instance storage for data storage as well, but the data in this HDFS would be lost after the termination of the HDInsight cluster.

HDInsight is a community technology preview (beta) as of January 2014.

2.4.2 Microsoft Dryad

Dryad [11] 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 they can be extended to represent many other parallel abstractions as well. Similar to MapReduce frameworks, the Dryad scheduler optimizes the data transfer overheads by scheduling the computations near data and handles failures through the 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 [7] API, which is a high level declarative language layer on top of Dryad. DryadLINQ queries get translated into 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.

2.4.3 Twister

The Twister [10] iterative MapReduce framework is an expansion of the traditional MapReduce programming model, which supports traditional as well as iterative MapReduce data-intensive computations. Twister supports MapReduce in the manner of "configure once, and run many times". Twister configures and loads static data into Map or Reduce tasks during the configuration stage, and then reuses the loaded data through the iterations. In each iteration, the data is first mapped in the compute nodes, and reduced, then combined back to the driver node (control node). Twister supports direct intermediate data communication, using direct TCP as well as using messaging middleware, across the workers without 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 [24]. Fault detection and recovery are supported between the iterations.

Java 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 [25] is used for the current experiments. Twister performs optimized broadcasting operations by using the chain method [26] and uses the minimum spanning tree method [27] 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 [13].

2.4.4 Haloop

Haloop [28] extends Apache Hadoop to support iterative applications and supports the 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 it also uses a separate MapReduce job to do the calculation (called Fix point 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 similar to traditional Hadoop. After that, the master node remembers the association between the data and the 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 a reducer input cache and a mapper input cache. In addition to these two, there is another cache called the reducer output cache, which is specially designed to support Fix point Evaluations. HaLoop can also cache intermediate data (reducer input/output cache) generated by the first iteration.

2.4.5 Spark

Spark [29] is an open source large data analytics framework that supports in-memory caching and interactive querying of data. Spark addressed some of the bottlenecks of the MapReduce model, while retaining the scalability and fault tolerance features of MapReduce. Spark provides better performance than Hadoop for many types of computations. Spark is implemented using Scala and builds on top of Hadoop Distributed File System (HDFS).

Spark introduces an abstraction called Resilient Distributed Datasets (RDDs) [30], which are distributed data sets partitioned across the cluster. RDD's can be created by performing deterministic operations on raw data or on other RDD's. RDD's are a form of intermediate data structures, and can be cached in memory for fast in-memory computations such as iterative computations and interactive queries. RDD's contain lineage information, and can be rebuilt in case a partition is lost due to some reason. Spark uses the RDD lineage information to provide fault tolerance support similar to MapReduce. Spark supports parallel computations on RDD's and provides a set of operators that can be applied on RDD's.

There are several projects that are building functionalities on top of Spark. These include an Apache Hive compatible data processing language layer called Shark[31], a large scale machine learning library named MLib, Spark-Streaming[32] stream data processing project and the GraphX[33] graph processing library.

2.4.6 Microsoft Daytona

Microsoft Daytona [26] 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 which enables data backup and easier failure recovery. Daytona supports the caching of static data between iterations. Daytona combines the output data of the Reducers to form the output of each iteration. Once the application has completed, the output can be retrieved from Azure Blob storage or can be continually processed by using other applications. In addition to the above features, which are similar to Twister4Azure, Daytona also provides automatic environment deployment and data splitting for MapReduce computations; it also claims to support a variety of data broadcast patterns between the iterations. However, as opposed to Twister4Azure, Daytona uses a single master node based controller to drive and manage the computation. This centralized controller substitutes for the 'Merge' step of Twister4Azure, but makes Daytona prone to single point 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 they can remain in the cloud for further processing and/or visualization. Daytona is available as a "Community Technology Preview" for academic and non-commercial usage.

2.4.7 i-MapReduce and PrIter

i-MapReduce [34] is another iterative MapReduce framework built on top of Hadoop. i-MapReduce reduces the startup overhead of creating new tasks in each iteration by supporting persistent Map and Reduce tasks. Persistent Map and Reduce tasks keep running until all the iterations are done. However, this requires that the cluster has enough free task slots to execute all the tasks of the computation at the same time. Similar to Twister and Twister4Azure, i-MapReduce improves the data shuffling by shuffling only the loop-variant data. It also partitions the data in such a way that the Map and Reduce tasks will have a one to one correspondence, and it uses this property to support the asynchronous execution of tasks. PrIter[35] improves i-MapReduce by introducing prioritized iterations, where it can prioritize the computations that provide the most help in terms of the convergence of the iterative algorithm.

2.4.8 Google Pregel and Apache Giraph

Pregel [36] is a large scale graph processing framework developed at Google. The Pregel programming model consists of vertices and edges, and it can be programmed by using iterations. Pregel follows the Bulk Synchronize Parallel (BSP) [37] model of computations, in which each iteration consists of independent computations at the vertices followed by communication and barrier synchronization. In a Pregel iteration, vertices receive messages sent to them in the previous iteration, perform the vertex computation independent of other vertices, and finally, send messages to the other vertices that will be received in the next iteration. The vertex computation can alter the state of that vertex, alter the state of the outgoing edges of that vertex and can change the graph topology as well. Pregel claims scalability up to thousands of computers and claims the ability to process billions of vertices and edges. Pregel achieves fault tolerance by check pointing the vertex and edge states at the beginning of each iteration.

Apache Giraph is an open source implementation of the Pregel model. Apache Giraph is built on top of Hadoop and translates the graph processing to a series of MapReduce computations. However, Giraph supports keeping the graph state in-memory throughout the computation, which improves the performance of the computations.

2.4.9 Other related cloud execution frameworks

CloudMapReduce [38] for Amazon Web Services (AWS) and Google AppEngine MapReduce [39] follow an architecture similar to MRRoles4Azure, in which they utilize the cloud services as the building blocks. Windows Azure HPC scheduler enables the users to launch and manage high-performance computing (HPC) and other parallel applications in the Windows 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 [40] is an implementation of a parallel BLAST on the Azure environment that uses Azure cloud services with an architecture similar to the Classic Cloud model described in section 3. CloudClustering [41] is a prototype KMeansClustering implementation that uses Azure infrastructure services. CloudClustering uses multiple queues (single queue per worker) for job scheduling and supports the caching of loop-invariant data.

2.5 Application types

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

2.5.1 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 and BLAST [42] searches, as well as 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 types of applications can be mapped to the MapReduce programming model as Map only applications or MapReduce applications with trivial reduce phases such as simple aggregation or collection of data.

Chapter 3 of this thesis focuses on providing solutions to executing pleasingly parallel applications on cloud environments and using cloud oriented applications frameworks. The pleasingly parallel applications discussed in this thesis include BLAST sequence searching (section 2.6.3), Cap3 sequence assembly (section 2.6.1) and GTM interpolation (section 2.6.2).

2.5.2 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 [43] and WordCount applications.

Chapter 4 of this thesis focuses on providing solutions to executing the MapReduce type of applications on cloud environments; it also introduces the MapReduceRoles4Azure decentralized cloud MapReduce framework for Azure cloud. MapReduce type applications discussed in this thesis include Smith-Watermann-GOTOH sequence distance calculation (section 2.6.4) application.

2.5.3 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 into 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. The emergence of computational fields, such as bioinformatics, and machine learning, has also contributed to an increased interest in this class of applications.

Chapter 5 of this thesis focuses on providing solutions to executing data intensive iterative applications on cloud environments; it also introduces the Twister4Azure iterative MapReduce framework for Azure cloud. Also chapter 7 of this thesis introduces the collective communication primitives for iterative MapReduce type applications. Data intensive iterative type applications discussed in this thesis includes KMeansClustering (section 2.6.5) and Multi-Dimensional-Scaling (section 2.6.6) applications.

2.5.4 MPI type applications

Applications with more complex inter-process communication and coordination requirements than the data intensive iterative applications fall into 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.

2.6 Applications

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

2.6.1 Cap3

Cap3 [44] 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 need to be assembled. The 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. The Cap3 is less memory intensive than the GTM Interpolation and BLAST applications we discuss below. The size of a typical data input file for the Cap3 program and the result data file range from hundreds of kilobytes to few megabytes. The output files resulting from the input data files can be collected independently and do not need any combining steps.

2.6.2 Generative Topographic Mapping Interpolation

Generative Topographic Mapping **(**GTM) [45] 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 [46, 47] has been developed as an out-of-sample extension to process much larger data points with a 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 it 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.

2.6.3 Blast+ sequence search

NCBI BLAST+ [42] is a very popular bioinformatics application that is used to handle sequence similarity searching. It is the latest version of BLAST [48], 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 into 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 a few bytes to a few Megabytes.

2.6.4 Sequence alignment using SmithWaterman GOTOH (SWG)

The SmithWaterman [49] algorithm with GOTOH [50] (SWG) improvement is used to perform pairwise sequence alignment on two FASTA sequences. We used the SWG application kernel in parallel to calculate the all-pairs dissimilarity of a set of *n* sequences resulting in *n*n* distance matrix. A set of map tasks for a particular job are generated using the blocked decomposition of the 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; this stresses the performance of inter-node communication and output data storage. The SWG can be considered as a memory-intensive application.

We used open source implementations, named JAligner and NAligner[11], of the Smith Waterman – Gotoh algorithm SW-G modified to ensure low start up effects by each thread, processing a large number (above a few hundred) of sequence calculations at a time. The 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 [43].

2.6.5 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 the clustering of several Terabytes of data using KMeansClustering and millions of centroids.

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

2.6.6 Multi-Dimensional Scaling (MDS)

The objective of multi-dimensional scaling (MDS) is to map a data set in a high-dimensional space to a user-defined lower dimensional space with respect to the pairwise proximity of the data points [51]. Dimensional scaling is used mainly in the visualizing of high-dimensional data by mapping them into a 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.

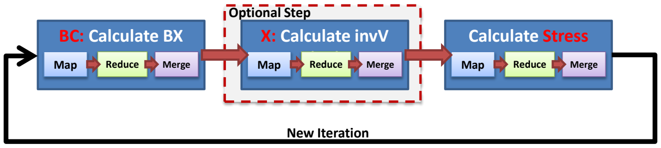


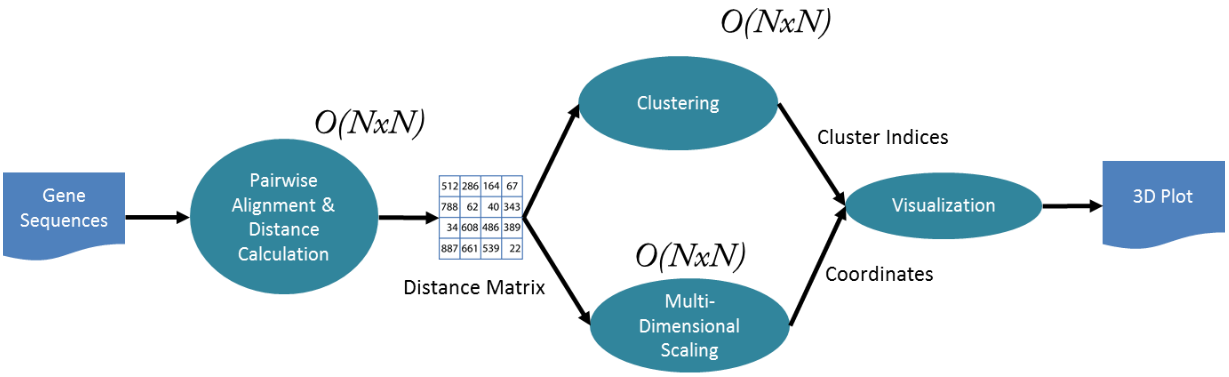
Figure 4 Multi-Dimensional Scaling SMACOF application architecture using iterative MapReduce

For the purposes of this dissertation, we use Scaling by MAjorizing a COmplicated Function (SMACOF) [52], an iterative majorization algorithm. The input for MDS is an *N*N* matrix of pairwise proximity values, where *N* is the number of data points in the high-dimensional space. The resultant lower dimensional mapping in *D* dimensions, called the *X* values, is an *N*D* matrix.

The limits of MDS are more bounded by memory size than by CPU power. The main objective of parallelizing MDS is to leverage the distributed memory to support processing of larger data sets. In this thesis, we implement the parallel SMACOF algorithm described by Bae et al [46]. This results in iterating a chain of three MapReduce jobs, as depicted in Figure 4. For the purposes of this dissertation, we performed 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.

2.6.7 Bio sequence analysis pipeline

The bio-informatics genome processing and visualizing pipeline [14] shown in Figure 5 inspired some of the application use cases analyzed in this thesis. This pipeline uses the SmithWatermann-GOTOH application, described in section 2.6.4, or the BLAST+ application, described in section 2.6.3, for sequence alignment, Pairwise clustering for sequence clustering and the Multi-Dimensional Scaling application, described in section 2.6.6, are used 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.



512	286	164	67
788	62	40	343
34	608	486	389
887	661	539	22

Figure 5 Bio sequence analysis pipeline [14]

3. Pleasingly parallel computing on cloud environments

A pleasingly parallel application is an application that can be parallelized, and thus requires 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 [44] sequence assembly, the 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 [9] and Microsoft Windows Azure [53] cloud computing platforms, and Apache Hadoop [6] MapReduce and Microsoft DryadLINQ [7], as the distributed parallel computing frameworks.

The work of this chapter have been presented and published as a workshop paper [54] and as a journal paper [55].

3.1 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

3.1.1 Classic Cloud processing model

Figure 6 depicts the Classic Cloud processing model. Varia [56] and Chappell [57] describe similar architectures that are implemented using the Amazon and Azure processing models, respectively. The Classic Cloud processing model follows a task processing pipeline approach with independent workers. It uses cloud instances (EC2/Azure Compute) for data processing, and it uses Amazon S3/Windows Azure Storage for 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 the cloud instances pick tasks from the scheduling queue. The configurable visibility timeout feature of the Amazon SQS and the Azure Queue services 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.

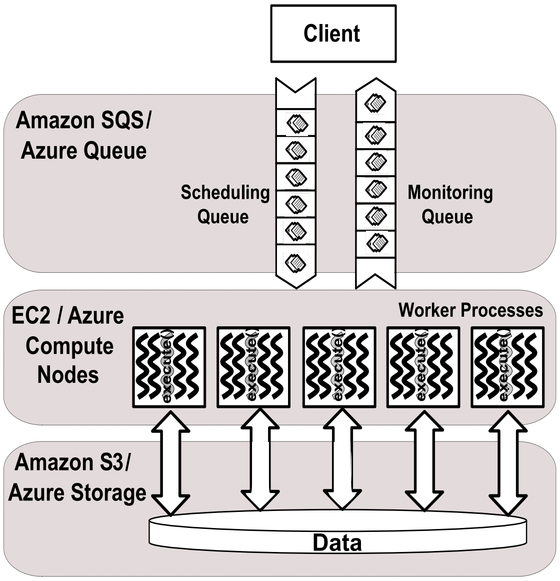


Figure 6 Classic cloud processing architecture for pleasingly parallel computations

For the applications discussed in this chapter, a single task is comprised 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; they will then process them using an executable program before uploading the results back to the cloud storage. In this implementation, the 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 the compute capacity.

3.1.2 Pleasingly parallel processing using MapReduce frameworks

We implemented similar pleasingly parallel processing frameworks using Apache Hadoop [6] and Microsoft DryadLINQ [7].

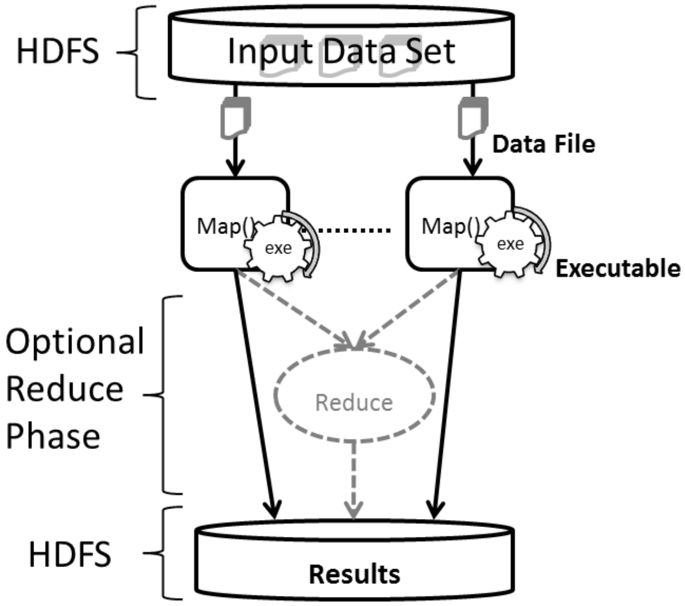


Figure 7 Hadoop MapReduce based processing model for pleasingly parallel computations

As shown in Figure 7, 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 [7] 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 7, 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.

3.1.3 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.

3.2 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:

$$\mathrm{ParallelEfficiency} = \frac{\mathrm{T1}}{\mathrm{pTp}} \quad \text{--- Equation 1[58]}$$

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 chapter, 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 chapter.

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

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 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 section, 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.

3.3 Cap3

We implemented a distributed parallel version of Cap3[44] sequence assembly application for Amazon EC2, Microsoft Azure, DryadLINQ and for Apache Hadoop using the frameworks that were presented in section 3.1. Cap3 program is a pleasingly parallel application that is often used in parallel with lots of input files. More details on Cap3 are given in section 2.6.1.

3.3.1 Performance with different EC2 cloud instance types

Figure 8 and Figure 9 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.

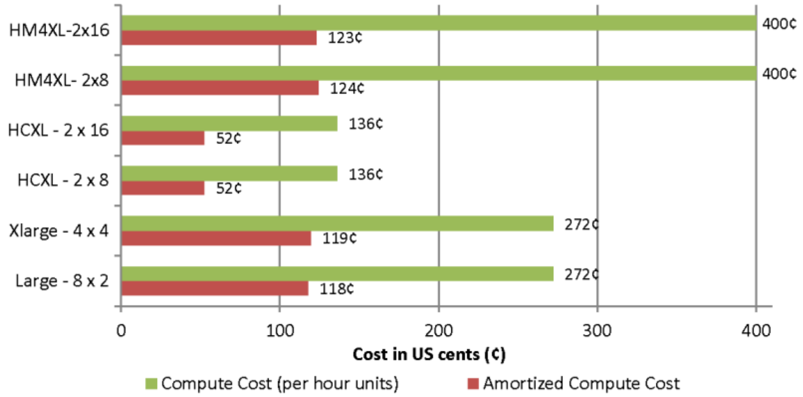


Figure 8 Cap3 application execution cost with different EC2 instance types

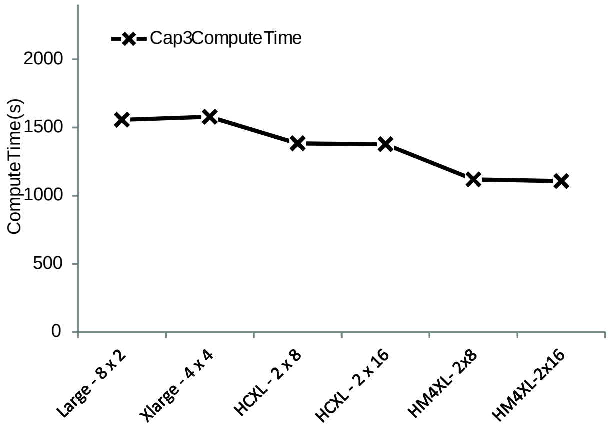


Figure 9 : Cap3 applciation compute time with different EC2 instance types

3.3.2 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 [43] 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 [43]. 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.

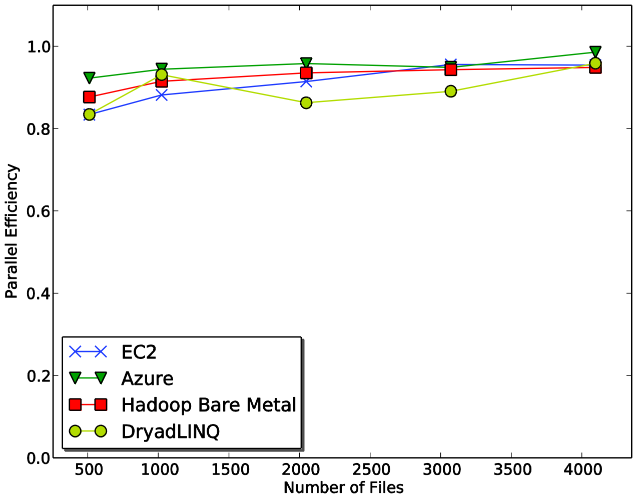


Figure 10 Parallel efficiency of Cap3 application using the pleasingly parallel frameworks

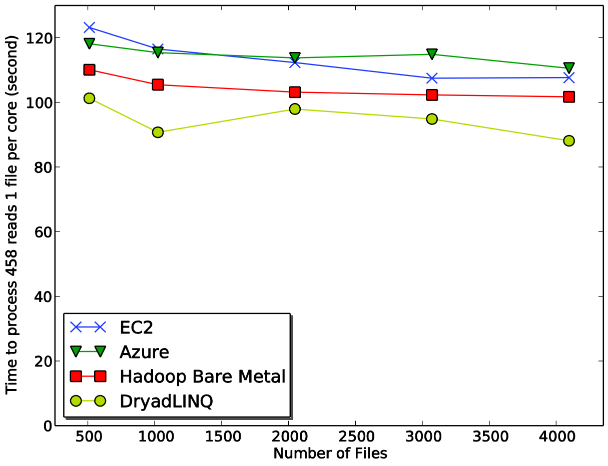


Figure 11 Cap3 execution time for single file per core using the pleasingly parallel frameworks

Based on Figure 10 and Figure 11, we can conclude that all four implementations exhibit comparable parallel efficiency (within 20%) with low parallelization overheads. When interpreting Figure 11, 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.

3.3.2.1 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.

3.4 BLAST

NCBI BLAST+ [42] is a very popular bioinformatics application that is used to handle sequence similarity searching described in section 2.6.3. 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 3.1. 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 chapter do not include the database distribution times.

3.4.1 Performance with different cloud instance types

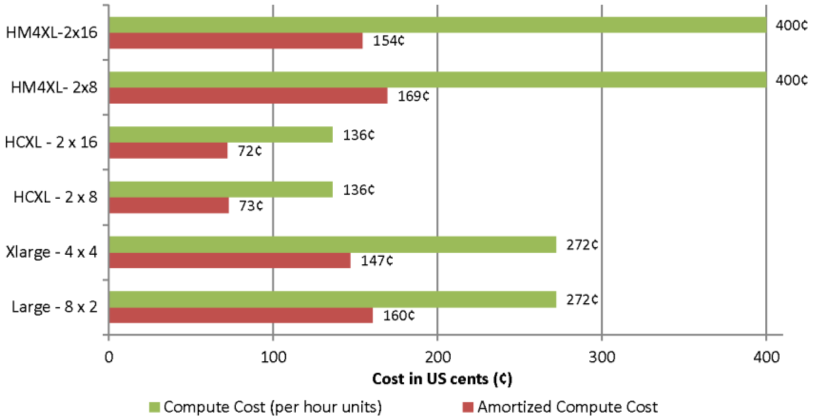


Figure 12 : Cost to process 64 BLAST query files on different EC2 instance types

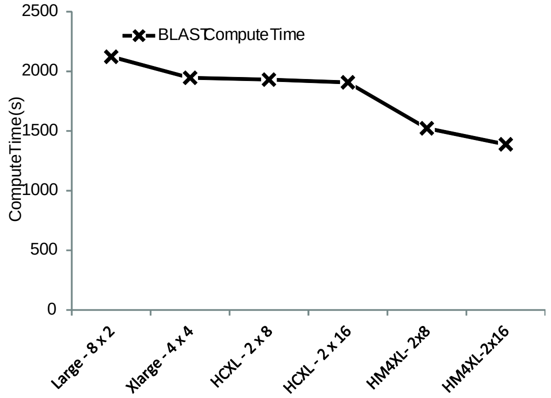


Figure 13 : Time to process 64 BLAST query files on different EC2 instance types

Figure 12 and Figure 13 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 14 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 14 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.

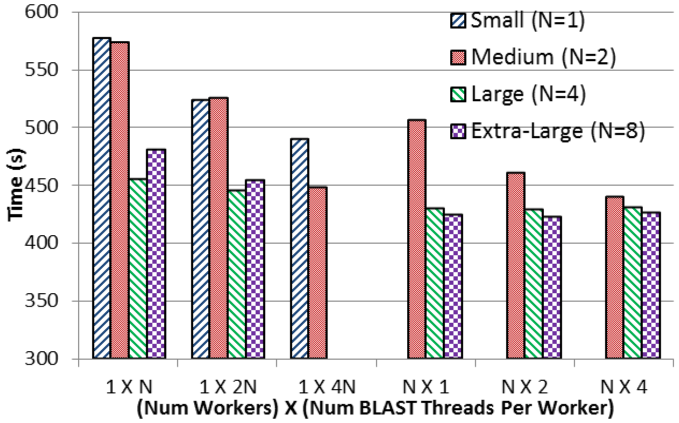


Figure 14 Time to process 8 query files using BLAST application on different Azure instance types

3.4.2 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.

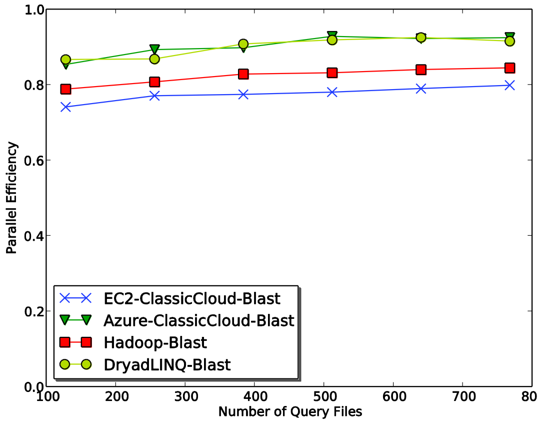


Figure 15 : BLAST parallel efficiency using the pleasingly parallel frameworks

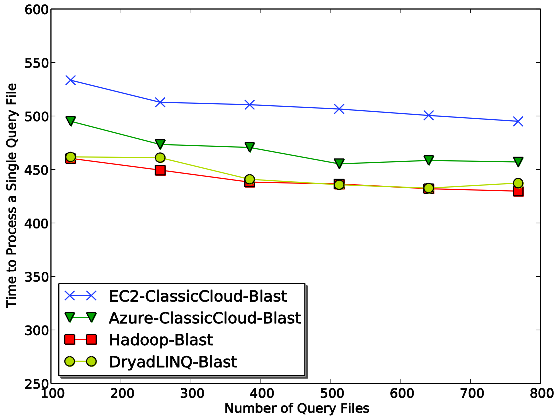


Figure 16 : BLAST average time to process a single query file using the pleasingly parallel frameworks

Figure 15 depicts the absolute parallel efficiency of the distributed BLAST implementations, while Figure 16 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.

3.5 GTM Interpolation

We implemented distributed parallel versions of GTM interpolations application described in section 2.6.2 for Amazon EC2, Microsoft Azure, DryadLINQ and for Apache Hadoop using the frameworks that were presented in section 3.1.

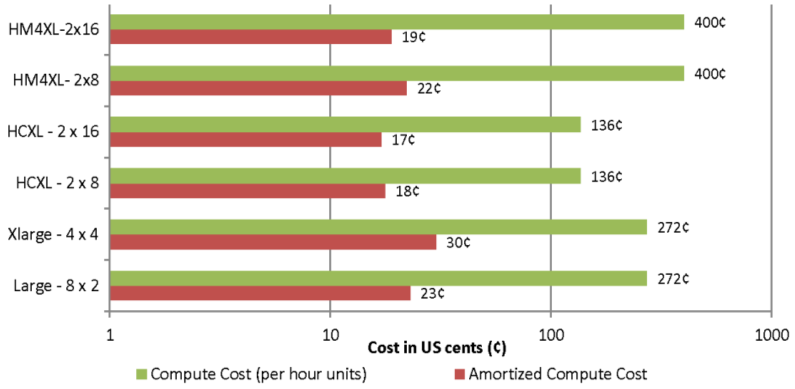


Figure 17 : Cost of using GTM interpolation application with different EC2 instance types

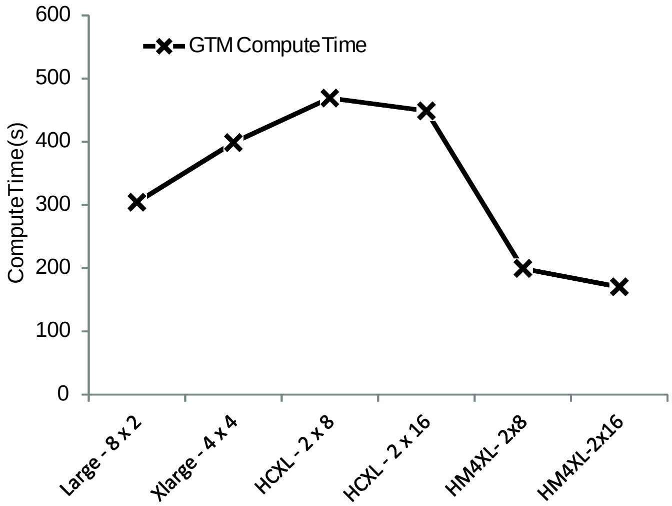


Figure 18 : GTM Interpolation compute time with different EC2 instance types

3.5.1 Application performance with different cloud instance types

According to Figure 18, 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.

3.5.2 GTM Interpolation speedup

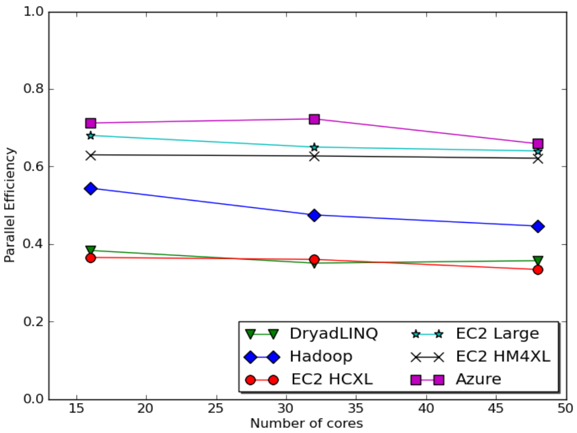


Figure 19: GTM Interpolation parallel efficiency using the pleasingly parallel frameworks

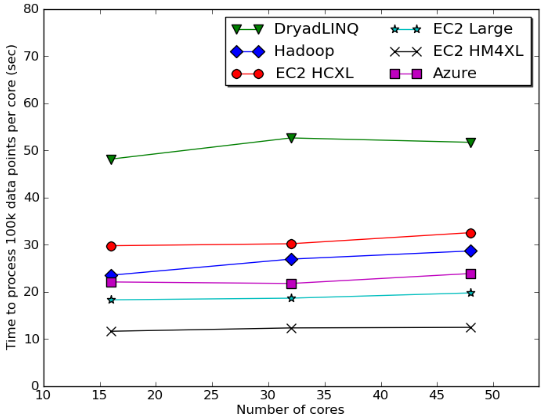


Figure 20 : GTM Interpolation performance per core using the pleasingly parallel frameworks

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 19 and Figure 20 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.

3.6 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 chapter 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 offer 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 both quality of services and operational 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 chapter, in the next couple of chapters, we present a fully-fledged MapReduce framework with iterative-MapReduce support for the Windows Azure Cloud infrastructure by 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.

4. MapReduce type applications on cloud environments

The MapReduce distributed data analysis framework model introduced by Google [2] 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 [3-5] 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 the 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 providers, to deliver a fault tolerant, dynamically scalable runtime with a familiar programming model for users.

Several options exist for executing MapReduce jobs on cloud environments, such as manually setting up a MapReduce (e.g.: Hadoop[6]) cluster on a leased set of computing instances, using an on-demand MapReduce-as-service offering such as Amazon ElasticMapReduce (EMR) [23]or using a cloud MapReduce runtime such as MRRoles4Azure or CloudMapReduce[38]. 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) [49, 50] and Cap3 [59] sequence assembly. We have performed experiments to gain insights about the performance of MapReduce in the clouds for the selected applications, and we compare its performance to MapReduce on traditional clusters. For this study, we use an experimental version of MRRoles4Azure.

Our work was further 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); this experience prompted us to perform the current analyses. In this chapter, we show that MapReduce computations performed in cloud environments, including MRRoles4Azure, have the ability to perform comparably to MapReduce computations on dedicated private clusters.

The work of this chapter has been presented and published as a conference paper [60].

4.1 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 [61], 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 [13]. 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 is 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 6.3).
- **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 is 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.

4.2 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.

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. The MRRoles4Azure implementation 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 [2], Hadoop [62] as well as Twister [63] 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 [43], 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.

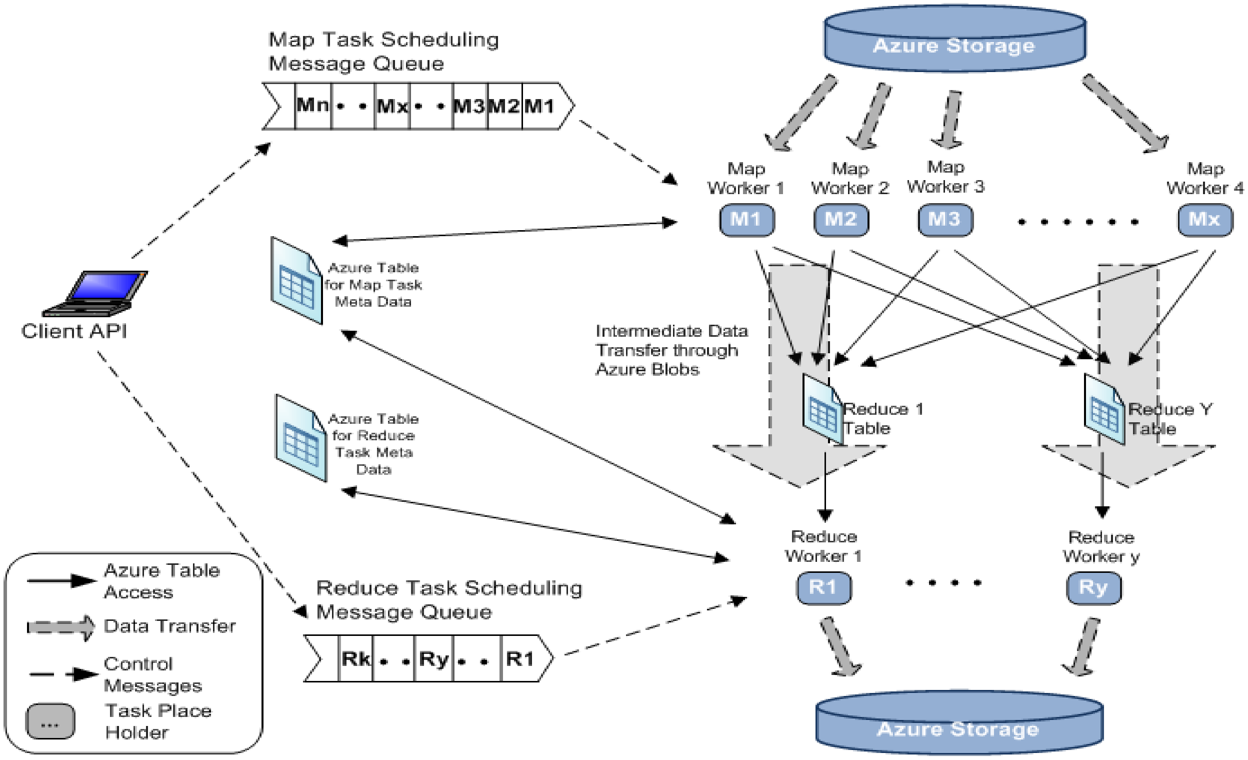


Figure 21 MapReduceRoles4Azure: Architecture for implementing MapReduce frameworks on Cloud environments using cloud infrastructure services

4.2.1 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 is possible to use the client driver to monitor the progress and completion of the MapReduce jobs.

4.2.2 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.

4.2.3 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 is 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.

4.2.4 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.

4.2.5 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 MRRoles4Azure, 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.

4.2.6 Limitations of MRRoles4Azure

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.

4.3 Performance evaluation

4.3.1 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 ($\mathrm{ParallelEfficiency} = \frac{\mathrm{T1}}{\mathrm{pTp}}$ --- Equation 1) and relative parallel efficiency calculations used to present results in this chapter.

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

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. $\alpha = p/p1$, where p1 is the smallest number of CPU cores for the experiment.

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

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

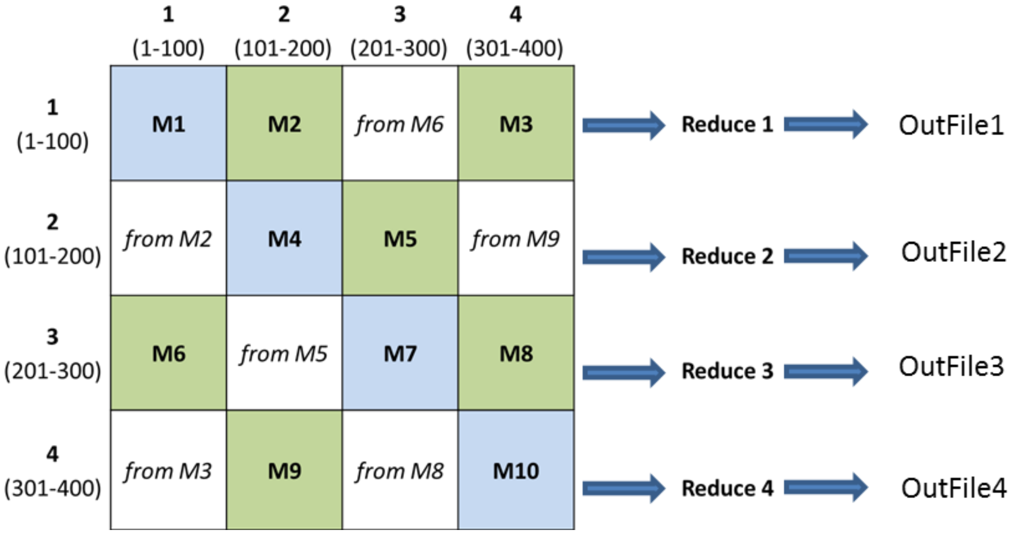


Figure 22 Task decomposition mechanism of SWG pairwise distance calculation MapReduce application

In this application, we use the Smith-Waterman [49] algorithm with GOTOH [50] (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 Figure 22, this property, together with blocked decomposition, 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 [43]. The MRRoles4Azure implementation also follows the same architecture and blocking strategy as in the Hadoop-SWG implementation. Hadoop-SWG uses the open source JAligner [64] as the computational kernel, while MRRoles4Azure SWG uses the C# implementation, NAligner [64] 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.

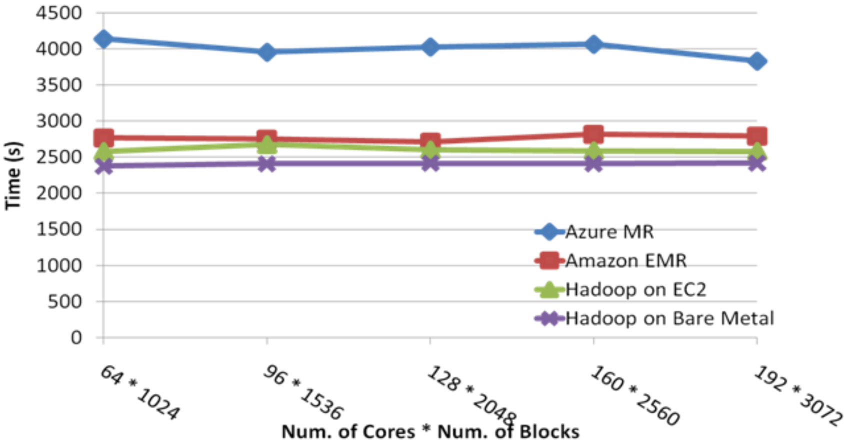


Figure 23 SWG MapReduce pure performance

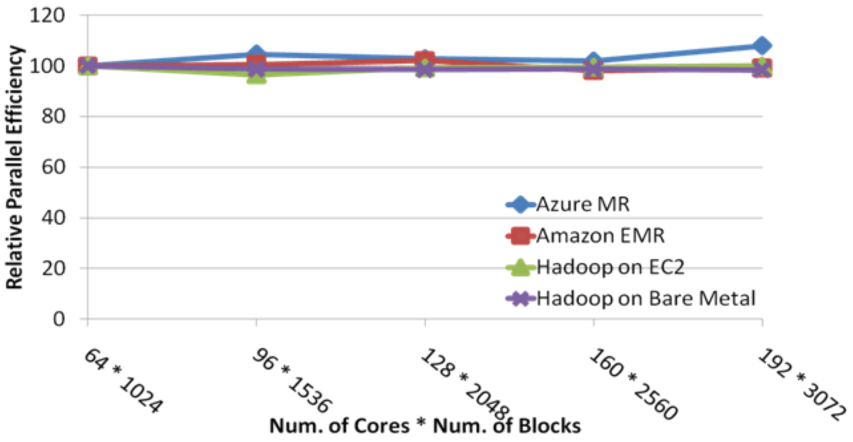


Figure 24 SWG MapReduce relative parallel efficiency

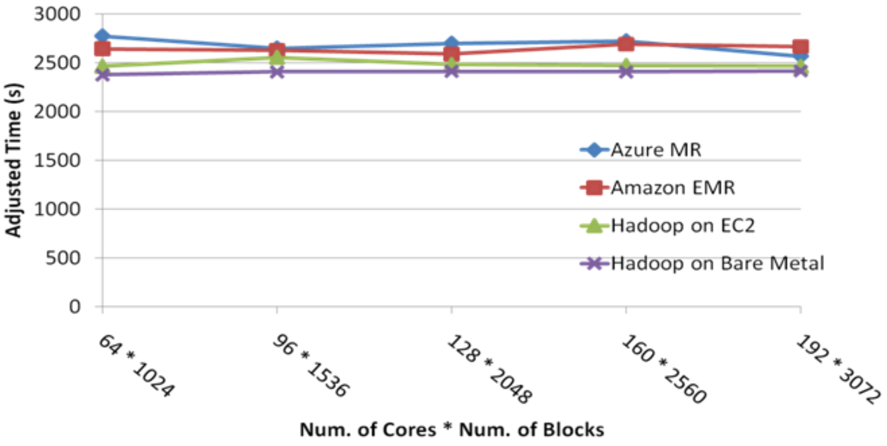


Figure 25 SWG MapReduce normalized performance

Due to the all-pairs nature and the block-based task decomposition of the SWG MapReduce implementations, it is 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 23. 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 23 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.

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 is 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 25, where we can observe that all four environments show comparable performance and good scalability for the SWG application. Figure 24 depicts the relative parallel efficiency of SWG MapReduce implementations using the 64 core, 1024 block test case as *p1* (see section V-A).

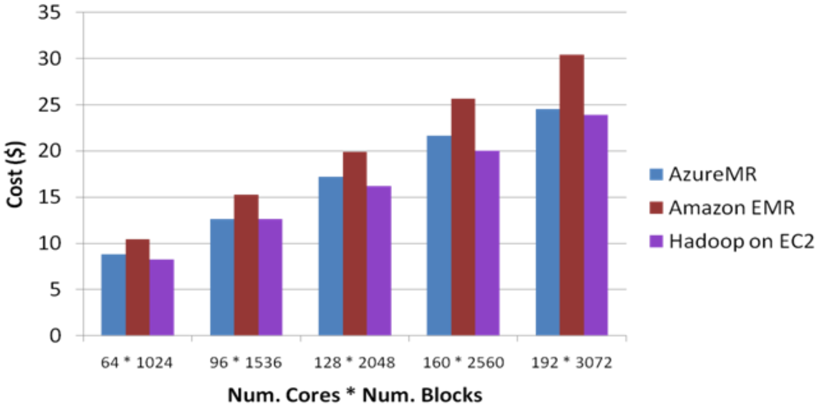


Figure 26 SWG MapReduce amortized cost for clouds

In Figure 26 we present the approximate computational costs for the experiments performed using cloud infrastructures. Even though the cloud instances are hourly billed, costs presented in Figure 26 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.

4.3.3 Sequence assembly using Cap3

Cap3 [59] 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 2.6.1 and more details about Cap3 Hadoop implementation are given in section 3.3.

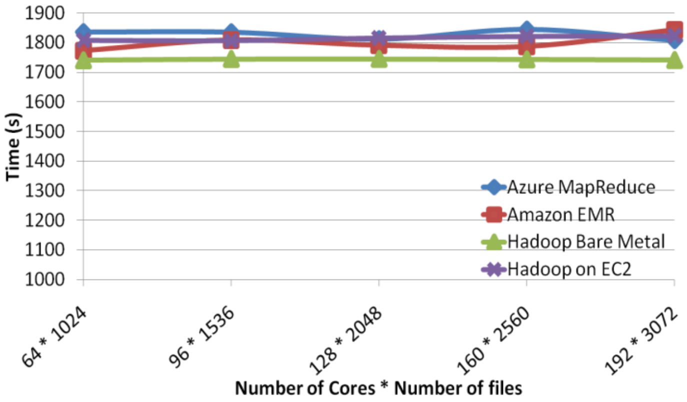


Figure 27 Cap3 MapReduce scaling performance

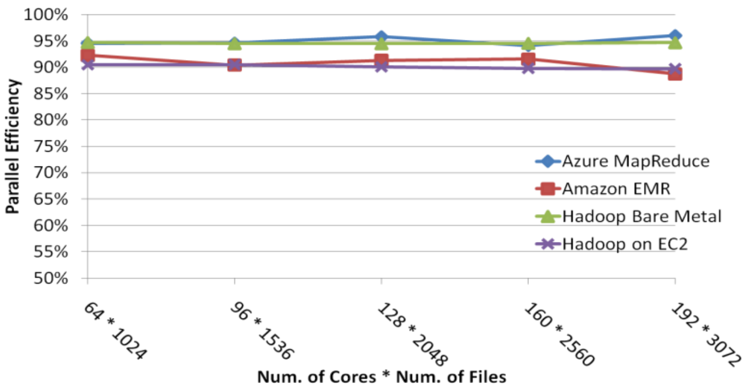


Figure 28 Cap3 MapReduce parallel efficiency

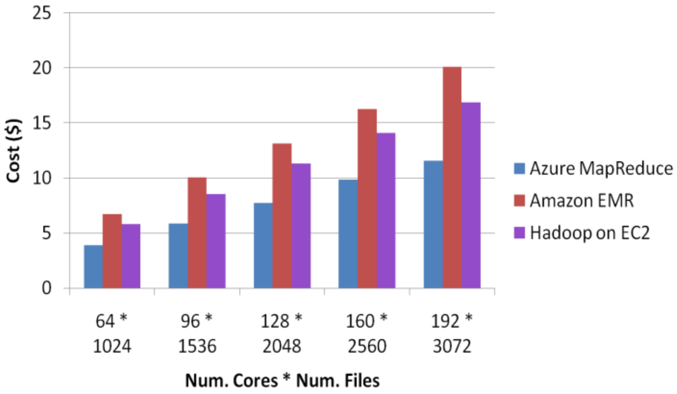


Figure 29 Cap3 MapReduce computational cost in cloud infrastructures

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 Bare Metal and Hadoop-EC2 experiments, while they were stored in Amazon S3 and Azure Blob storage for EMR and MRRoles4Azure experiments respectively. Figure 27 presents the pure performance of the Cap3 MapReduce applications, while Figure 28 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 29 depicts the approximate amortized computing cost for the Cloud MapReduce applications, with MRRoles4Azure showing an advantage.

4.4 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 by 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 comparably to the other MapReduce implementations; thus, these results prove the feasibility of the MRRoles4Azure architecture. We also explored the challenges presented by cloud environments to execute MapReduce computations and we discussed how we overcame them by using 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 and no upfront costs, coupled together with the features of on demand availability, horizontal scalability and the easy to use programming model make using MapReduce in cloud environments a very viable option and an enabler for 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.

5. 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; this makes 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 in 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 the 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; this thus enables a wide array of large-scale iterative data analysis and scientific applications to utilize the Azure platform easily and efficiently in a fault-tolerant manner. Twister4Azure effectively utilizes the eventually consistent, high-latency Azure cloud services to deliver performance that is comparable to traditional MapReduce runtimes for non-iterative MapReduce, while outperforming traditional MapReduce runtimes for iterative MapReduce computations. Twister4Azure has minimal management and maintenance overheads, and it 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.

The applications of Twister4Azure can be categorized according to three classes of application patterns. The first of these are the Map only applications, described in section 2.5.1, which are also called pleasingly (or embarrassingly) parallel applications. Examples of this type of application include Monte Carlo simulations, BLAST+ sequence searches, parametric studies and most of the data cleansing and pre-processing applications. Twister4Azure contains sample implementations of the BLAST+ and Cap3 as pleasingly parallel applications.

The second type of applications includes the traditional MapReduce type applications, described in section 2.5.2, which utilize the reduction phase and other features of MapReduce. Twister4Azure contains sample implementations of the SmithWatermann-GOTOH (SWG) [43] pairwise sequence alignment and Word Count as traditional MapReduce type applications.

The third and most important type of applications Twister4Azure supports is the iterative MapReduce type of 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 5.2.2 and **Error! Reference source not found.** present detailed analyses of Multi-Dimensional Scaling and KMeans Clustering iterative MapReduce implementations.

The work of this chapter has been presented and published as a conference paper [65] and as a journal paper [12].

5.1 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.

5.1.1 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.

5.1.1.1 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 Figure 3. We can identify two main types of data in these computations, the loop invariant input data and the loop variant delta values. Loop variant delta values are the result, or a representation of the result, of processing the input data in each iteration. Computations of an iteration 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 **Error! Reference source not found.**). 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. Section 2.3 provides more information on iterative MapReduce and iterative data intensive computations.

Code 1 Typical data-intensive iterative computation

```
k ← 0;
MAX ← maximum iterations
$\delta^{[0]}$ ← initial delta value
while ( k< MAX_ITER || f($\delta^{[k]}$, $\delta^{[k-1]}$) )
    foreach datum in data
        β[datum] ← process (datum, $\delta^{[k]}$)
    end foreach
    $\delta^{[k+1]}$ ← combine(β[])
    k ← k+1
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 30 presents the flow of the Twister4Azure programming model.

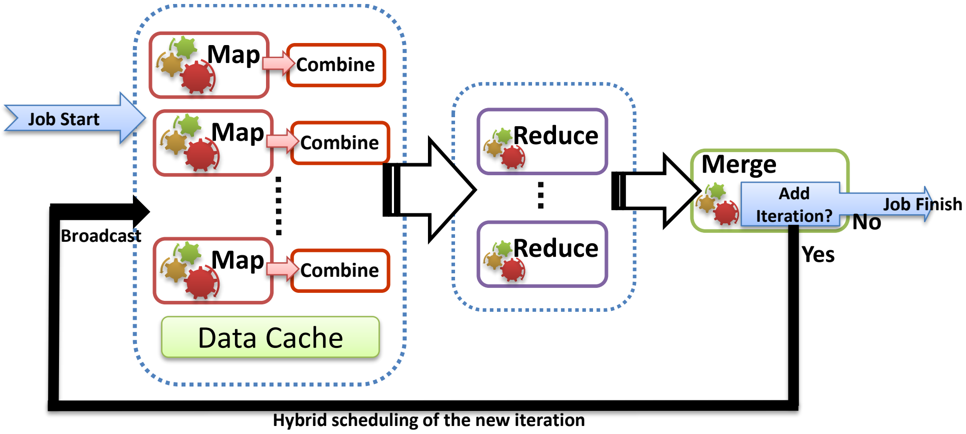


Figure 30 Twister4Azure iterative MapReduce programming model

Code 2 Data-intensive iterative computation using Twister4Azure programming model

```
k ← 0;
MAX ← maximum iterations
$\delta^{[0]}$ ← initial delta value
$\alpha$ ← true

while ( k< MAX_ITER || $\alpha$)
    distribute datablocks
    broadcast $\delta^{[k]}$
    map (datablock, $\delta^{[k]}$)
            foreach datum in datablock
                $\beta$[datum] ← process (datum, $\delta^{[k]}$)
            end foreach
            emit ($\beta$)

    reduce (list of $\beta$)
            $\beta'$ ← combine (list of $\beta$)
            emit ($\beta'$)

    merge (list of $\beta'$, $\delta^{[k]}$)
            $\delta^{[k+1]}$ ← combine (list of $\beta$)
            $\alpha$ ← f($\delta^{[k]}$, $\delta^{[k-1]}$)
            emit ($\alpha$, $\delta^{[k+1]}$)

    k ←k+1
end while
```

5.1.1.2 Map and Reduce API

Twister4Azure extends the *map* and *reduce* functions of traditional MapReduce to include the loop variant delta values as an input parameter. This additional input parameter is a list of key, value pairs. This parameter can be used to provide an additional input through a broadcast operation or through a scatter operation. Having this extra input allows the MapReduce programs to perform Map side joins, avoiding the significant data transfer and performance costs of reduce side joins[27], and avoiding the often unnecessary MapReduce jobs to perform reduce side joins. The PageRank computation presented by Bu, Howe, et.al. [28] 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>)

5.1.1.3 Merge

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

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

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

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

5.1.2 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
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 6.4 presents a more detailed discussion about the performance trade-offs and implementation strategies of the different caching mechanisms.

5.1.3 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 Figure 31, 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.

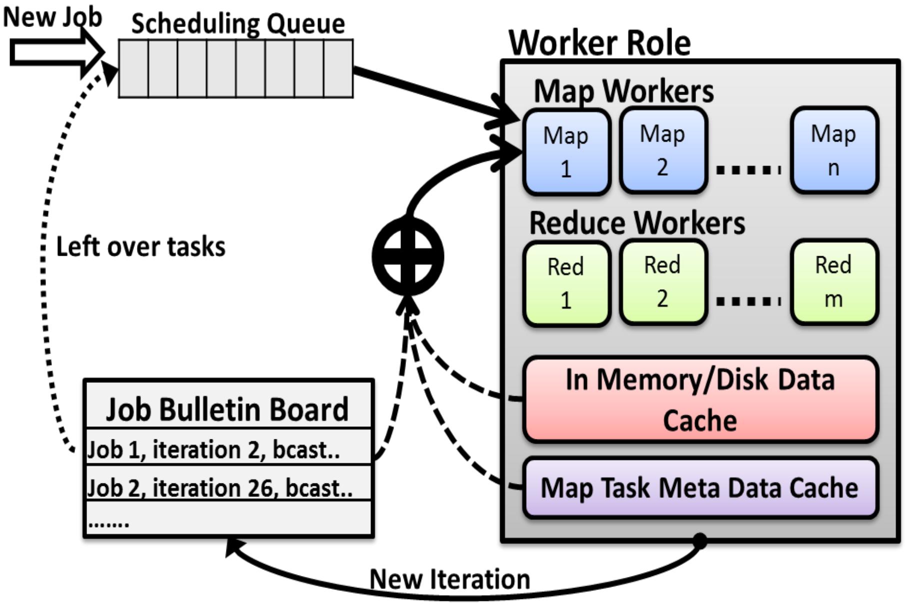


Figure 31 Cache Aware Hybrid Scheduling

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
```

5.1.4 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 Figure 3. This method supports a configurable number of parallel outgoing TCP transfers per instance (three parallel transfers in Figure 3) , 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, as shown by N_3 in the Figure 3. 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.

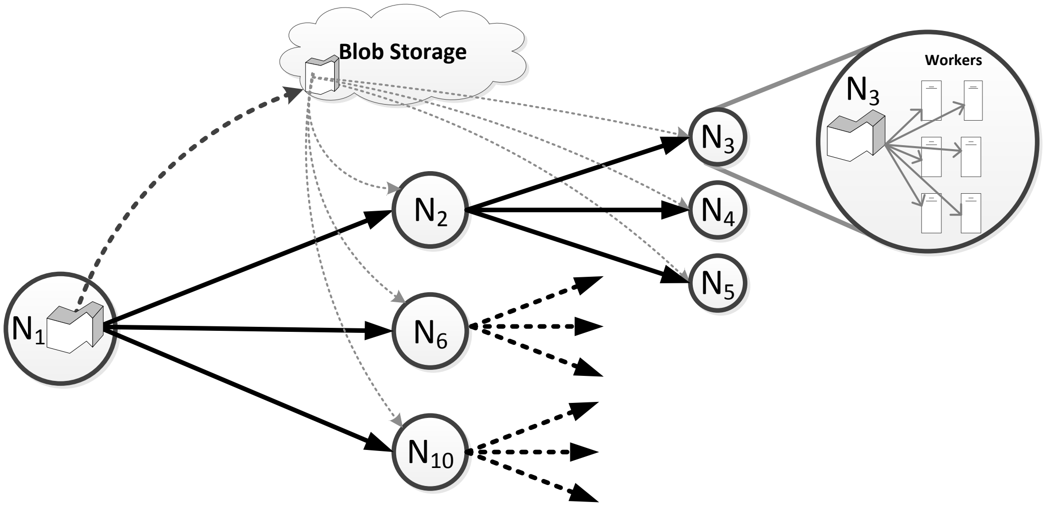


Figure 32 Twister4Azure tree based broadcast over TCP with Azure Blob storage as the persistent backup.

5.1.5 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 [66] 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.

5.1.6 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.

5.1.7 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 2.6.6 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 Worker Role project.

5.1.8 Development and current status

Developing Twister4Azure was an incremental process, which began with the development of pleasingly parallel cloud programming frameworks [55] (section 3) for bioinformatics applications utilizing cloud infrastructure services. MRRoles4Azure [60] (section 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.

5.2 Twister4Azure Scientific Application Case Studies

5.2.1 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 [10] 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 [13] 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 6) 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.

5.2.2 Multi-Dimensional Scaling - Iterative MapReduce

We implemented the SMACOF algorithm for Multi-Dimensional Scaling (MDS) described in section 2.6.6 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 Figure 4. For the purposes of this chapter, we perform an unweighted mapping that 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.

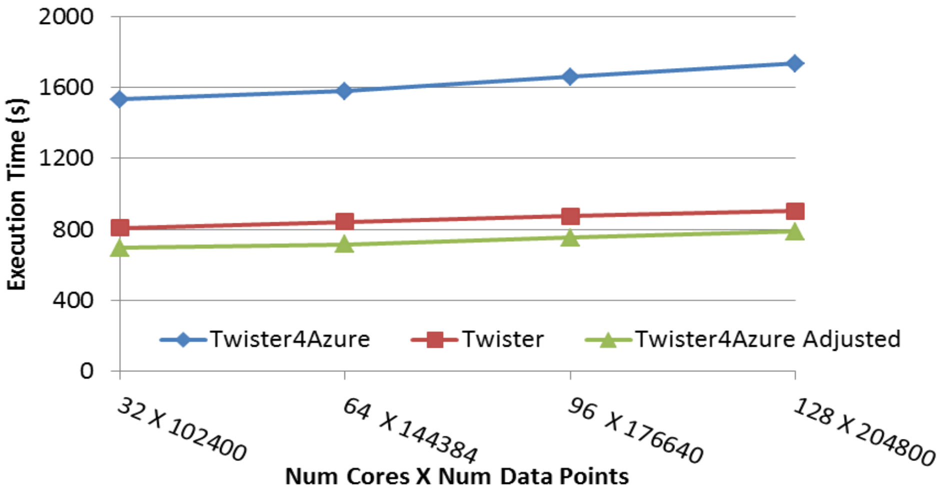


Figure 33 MDS weak scaling. Workload per core is constant. Ideal is a straight horizontal line

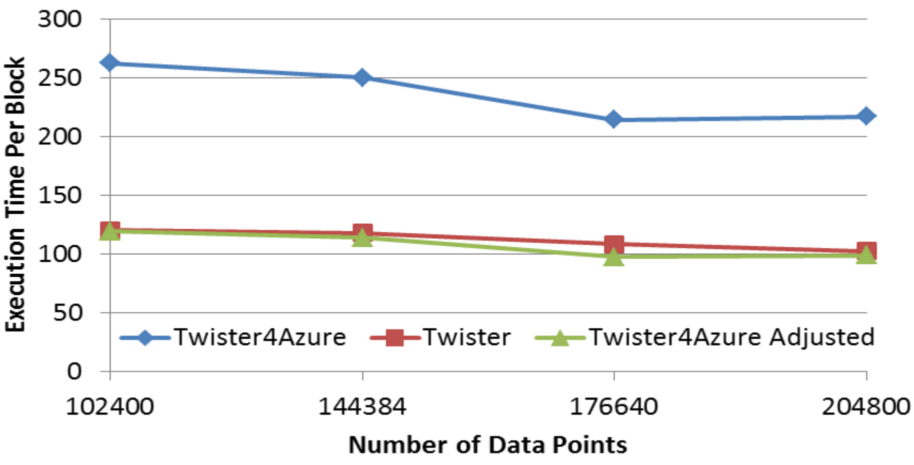


Figure 34 MDS Data size scaling using 128 Azure small instances/cores, 20 iterations

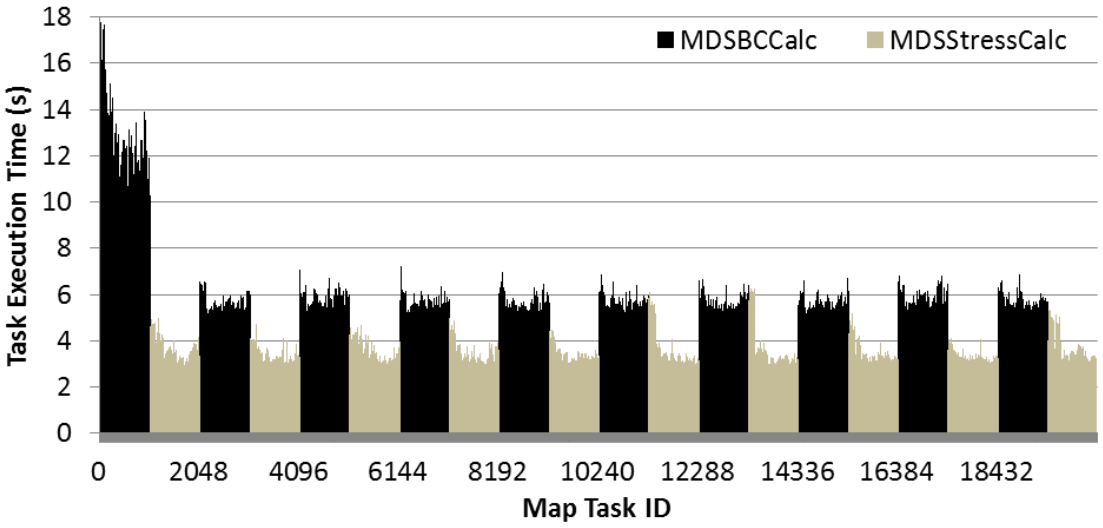


Figure 35 Twister4Azure Map Task histogram for MDS of 204800 data points on 32 Azure Large Instances (graphed only 10 iterations out of 20). Two adjoining bars represent an iteration (2048 tasks per iteration), where each bar represent the different applications inside the iteration.

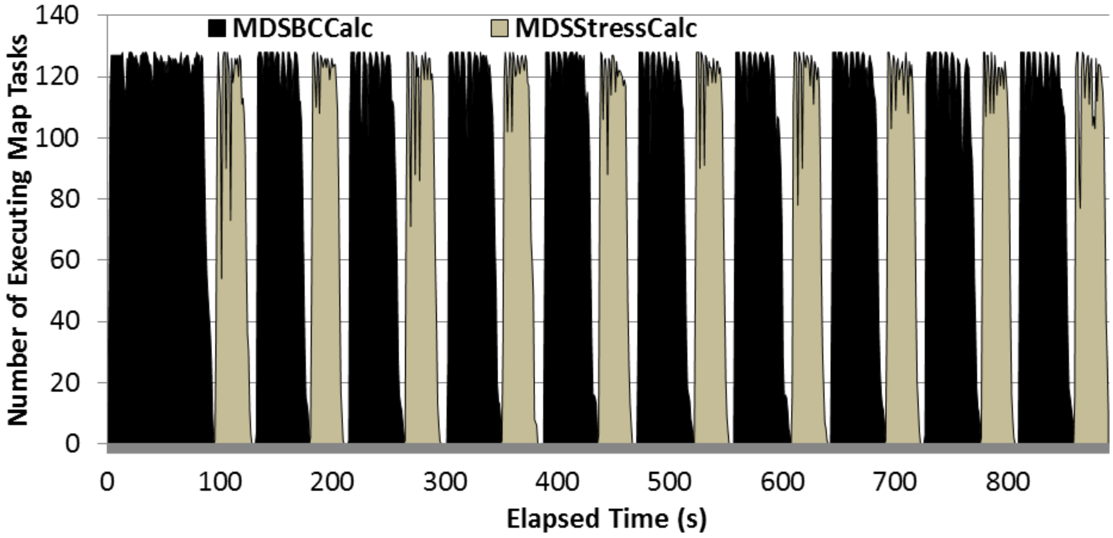


Figure 36 Number of executing Map Tasks in the cluster at a given moment. Two adjoining bars represent an iteration.

We compared the Twister4Azure MDS performance with Java HPC Twister MDS implementation. The Java HPC Twister experiment was performed in the HighMem cluster (Table 6). The Twister4Azure tests were performed on Azure Large instances using the Memory-Mapped file based (section 6.4.3) data caching. Java HPC Twister results do not include the initial data distribution time. Figure 33 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 34 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 35 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 36 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.

5.2.3 KMeans Clustering

The K-Means Clustering [67] algorithm described in section 2.6.5 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.

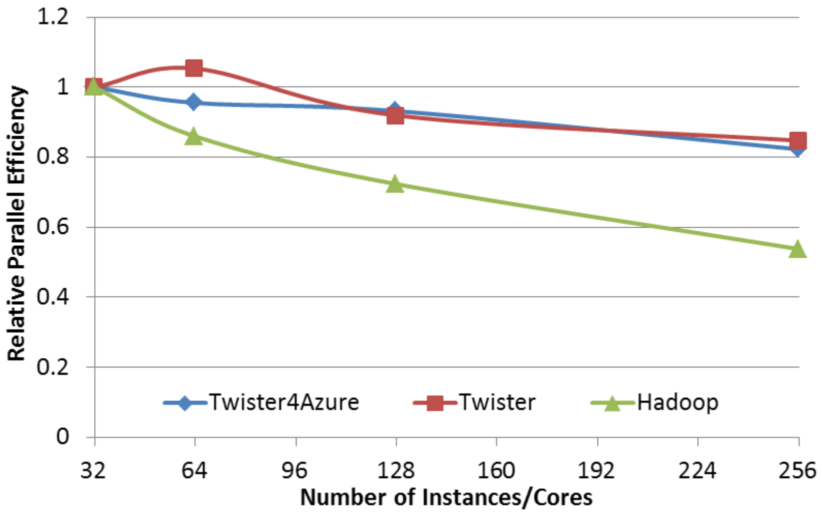


Figure 37 KMeans Clustering Scalability. Relative parallel efficiency of strong scaling using 128 million data points.

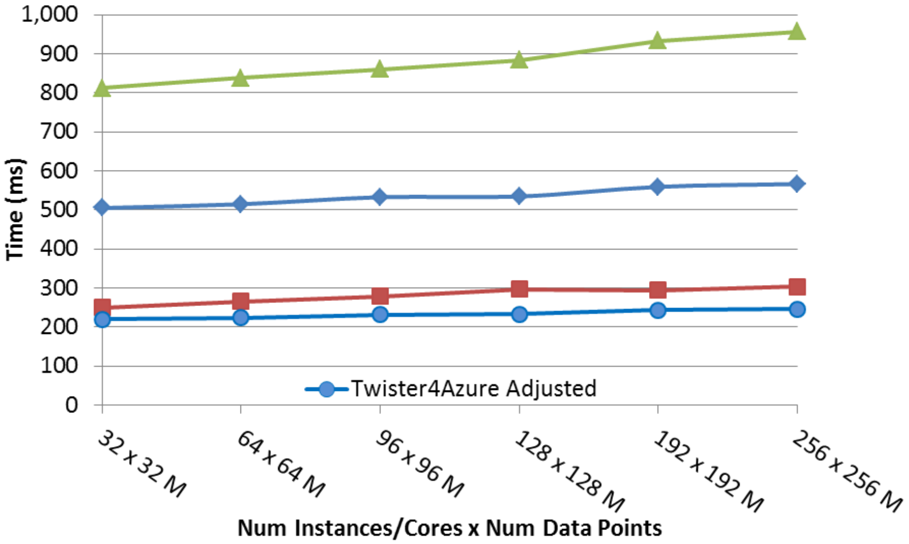


Figure 38 KMeansClustering Scalability. Weak scaling. Workload per core is kept constant (ideal is a straight horizontal line).

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 6). 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 37 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 38 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 Figure 37, 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 38 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.

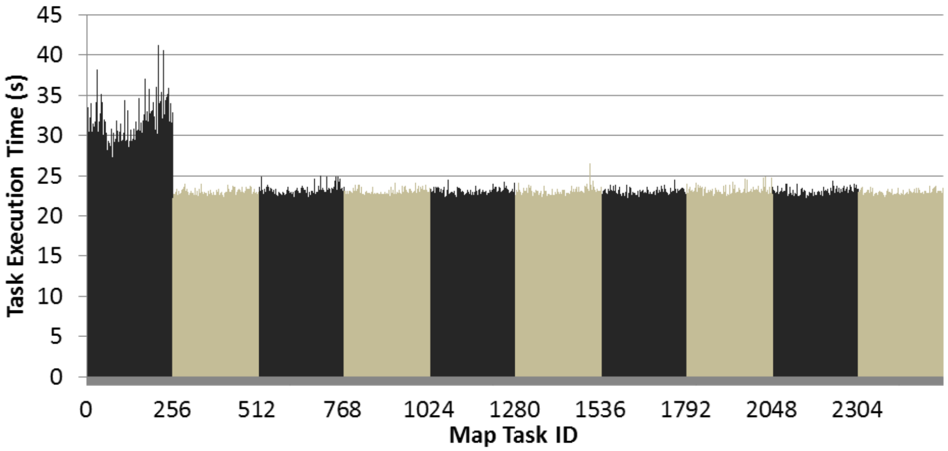


Figure 39 Twister4Azure Map Task execution time histogram for KMeans Clustering 128 million data points on 128 Azure small instances.

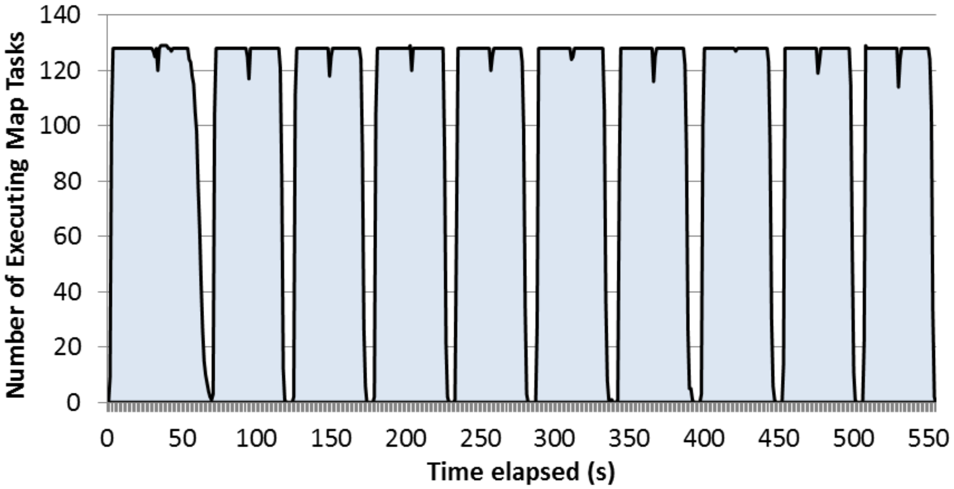


Figure 40 Twister4Azure number of executing Map Tasks in the cluster at a given moment

Figure 39 depicts the execution time of Map Tasks across the whole job. The higher execution time of the tasks in the first iteration is due to the overhead of initial data downloading, parsing and loading, which is an indication of the performance improvement we get in subsequent iterations due to the data caching. Figure 40 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.

5.3 Summary

We presented Twister4Azure, a novel iterative MapReduce distributed computing runtime for Windows Azure Cloud. Twiser4Azure enables users to perform large-scale data intensive parallel computations efficiently on the Windows Azure Cloud, by hiding the complexity of scalability and fault tolerance when using Clouds. The key features of Twiser4Azure presented in this chapter 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 a 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 the Java HPC Twister and the 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 performance results 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 results comparable to the Apache Hadoop on a dedicated local cluster.

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

In this section, we explore several aspects of the applications and the environments that would affect the performance of executing data intensive parallel applications on cloud environments. These include the investigations of the load balancing effects of inhomogeneous data, the effects of the virtualization overhead in virtualized environments, the performance variation with time in cloud environments and the different mechanisms to cache data in cloud instances.

6.1 Inhomogeneous data

Next generation parallel data processing frameworks such as Hadoop and DryadLINQ are designed to perform optimally when a given job can be divided into 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 into 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 and Dryad use 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.

6.1.1 SW-G Pairwise Distance Calculation

The inhomogeneity of data applies for the gene sequence sets, too, where individual sequence lengths and their contents can vary greatly. 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 the Smith-Waterman-Gotoh algorithm is approximately proportional to the product of the lengths of two sequences (O(mn)). All of the above described distributed implementations of the Smith-Waterman similarity calculation mechanisms rely on block decomposition to break down the larger problem space into sub-problems that can be solved using the distributed components. Each block is assigned two sub-sets of sequences, where the Smith-Waterman pairwise distance similarity calculation needs to be performed for all of 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 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.

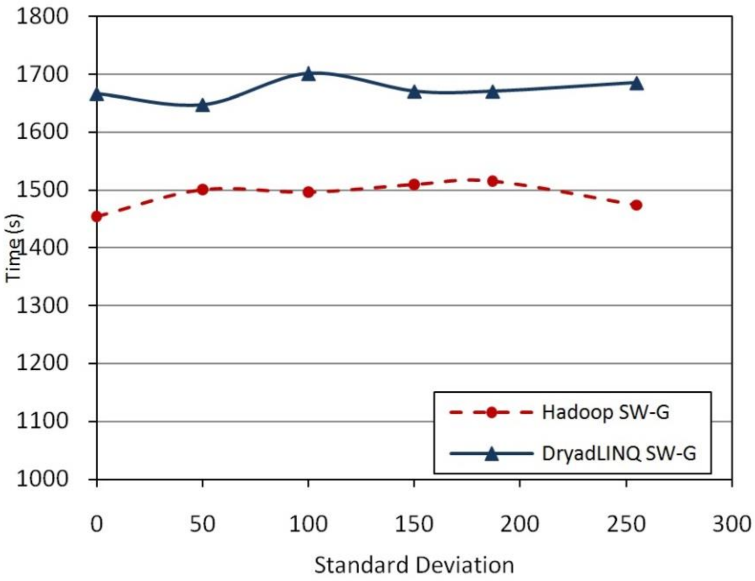


Figure 41 Performance of SW-G for randomly distributed inhomogeneous data with '400' mean sequence length.

For the inhomogeneous data study, we decided to use controlled inhomogeneous input sequence sets with the same average length and varying standard deviation of lengths. It is 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 content of the sequences. This property of the computation makes it possible for us to ignore the content 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 41 presents the execution time taken for the randomly distributed inhomogeneous data sets with the same mean length, by the two different implementations, while Figure 42 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 Figure 41, 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.

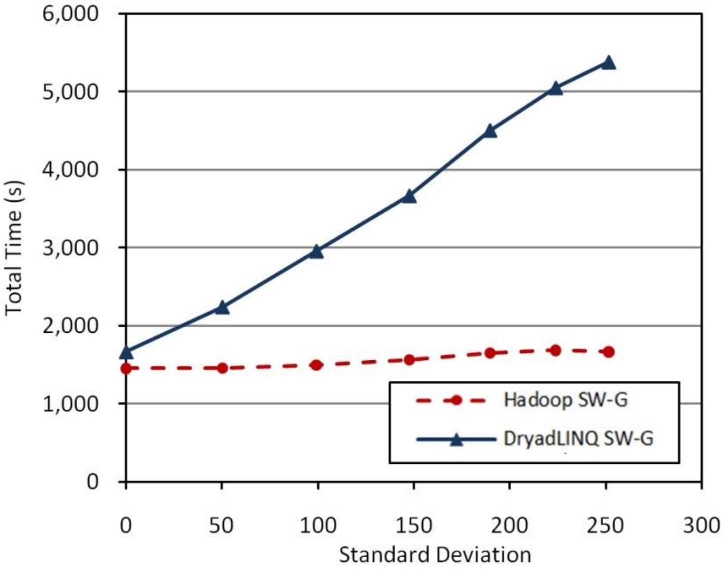


Figure 42 Performances of SW-G for skewed distributed inhomogeneous data with '400' mean sequence length.

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. The 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 onto 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. The Dryad implementation pre-allocates all the tasks to the compute nodes and does not perform any dynamic scheduling across the nodes. This makes any node which gets a larger work chunk take considerably longer than does a node which gets a smaller work chunk; this phenomenon causes the node with a smaller work chuck to idle while the other nodes finish.

6.1.2 CAP3

Unlike in Smith-Waterman Gotoh (SW-G) implementations, the CAP3 program execution time does not directly depend on the file size or the size of the sequences, as it depends 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 stand-alone 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 stand-alone 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.

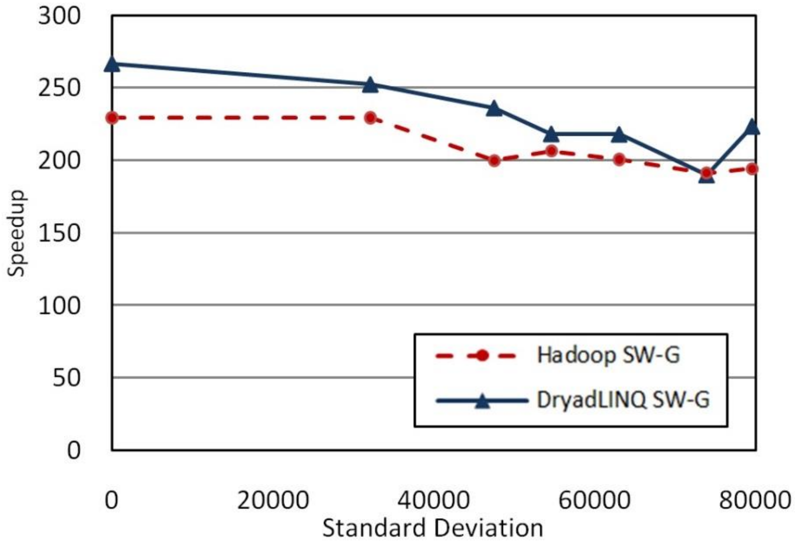


Figure 43 Performance of Cap3 for random distributed inhomogeneous data.

Figure 43 and Figure 44 depict the CAP3 inhomogeneous performance results for the Hadoop and Dryad implementations. The 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.

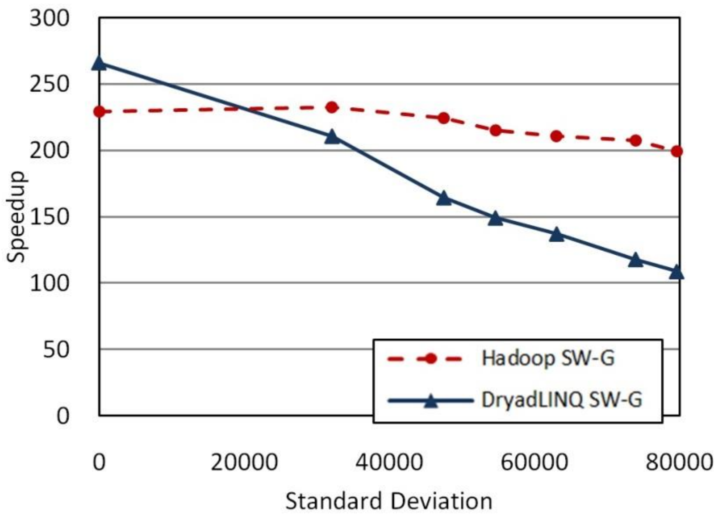


Figure 44 Performance of Cap3 for skewed distributed inhomogeneous data

6.2 Virtualization overhead

With the popularity of the computing clouds, we can notice that 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 [68]. Here, we present the 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 [69] para-virtualized environment.

While the Youseff, Wolski, et al. [70] suggest that the VM's impose very little overheads on the 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 (a lower communication to computation ration, with a large number of smaller messages) experienced higher overheads in virtual environments. Walker [71] presents the 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 over and knowledge of the EC2 infrastructure, there exist too many unknowns to directly compare these results with the above mentioned results.

6.2.1 SW-G Pairwise Distance Calculation

Figure 45 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 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 a virtual environment ranges from 25% to 15%. We can notice the performance degradation gets reduced with the increase of the problem size.

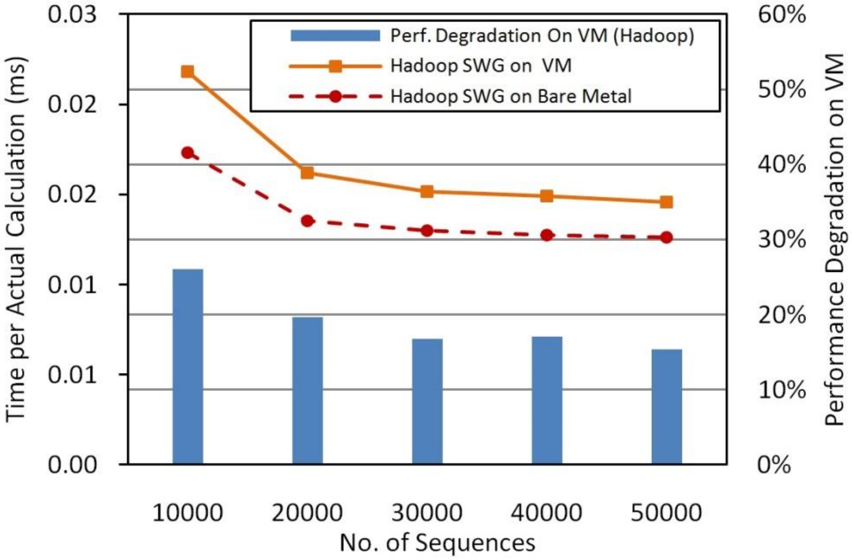


Figure 45 Virtualization overhead of Hadoop SW-G on Xen virtual machines

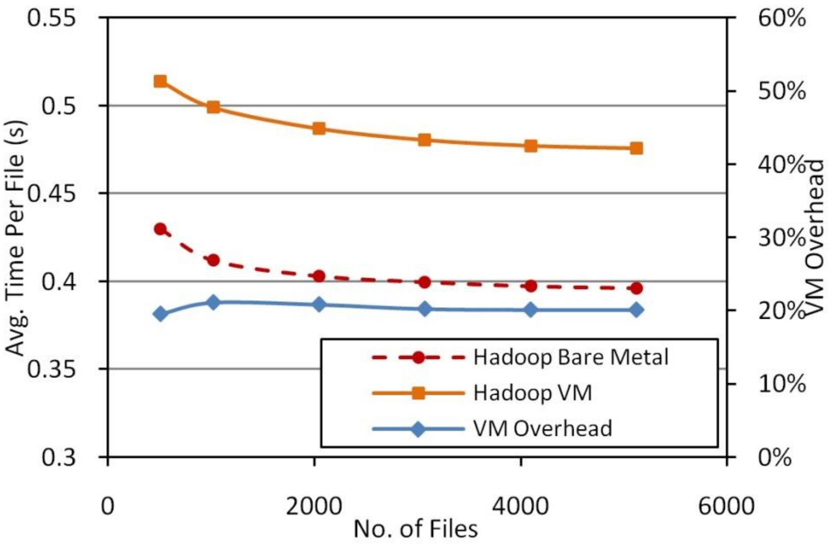


Figure 46 Virtualization overhead of Hadoop Cap3 on Xen virtual machines

In the Xen para-virtualization architecture, each guest OS (running in domU) performs its I/O transfers through Xen (dom0). This process adds startup costs to the I/O, as it involves startup overheads such as communication with dom0 and the 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 need 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 the performance degradation with the increase of data size, as large data sizes increase the granularity of the computational blocks.

6.2.2 CAP3

Figure 46 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. The CAP3 application does not show the decrease of the VM overhead with the increase of the 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 the workload.

6.3 Sustained performance of clouds

When discussing 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 the 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.

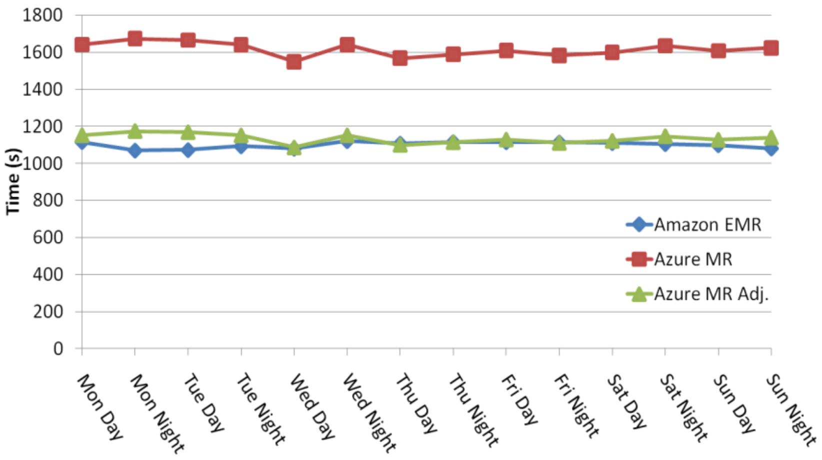


Figure 47 Sustained performance of cloud environments for MapReduce type of applications

6.4 Data Caching on Azure Cloud instances 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 5.2.2. 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 acquisition 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.

6.4.1 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.

6.4.2 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, and 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 5.1.3, Twister4Azure supports cache-aware scheduling for in-memory cached data as well as for local-storage cached data.

Table 7 The execution time analysis of a MDS computation with different data caching mechanisms. (204800 * 204800 input data matrix, 128 total cores, and 20 iterations. 20480 BCCalc map tasks)

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

6.4.2.1 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 48 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 a smaller standard deviation in the Map F^n time in row 1 of Table 7). 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 7).

6.4.3 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. The .NET framework version 4 introduces first class support for memory-mapped files to the .NET world. The .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 these 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 7).

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 7). 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 7). 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 7, 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 49 presents an execution trace of a job that uses Memory Mapped file based caching. The taller bars represent the MDSBCCalc computation, while the shorter bars represent the MDSStressCalc computation. A pair of BCCalc and StressCalc bars represents an iteration.

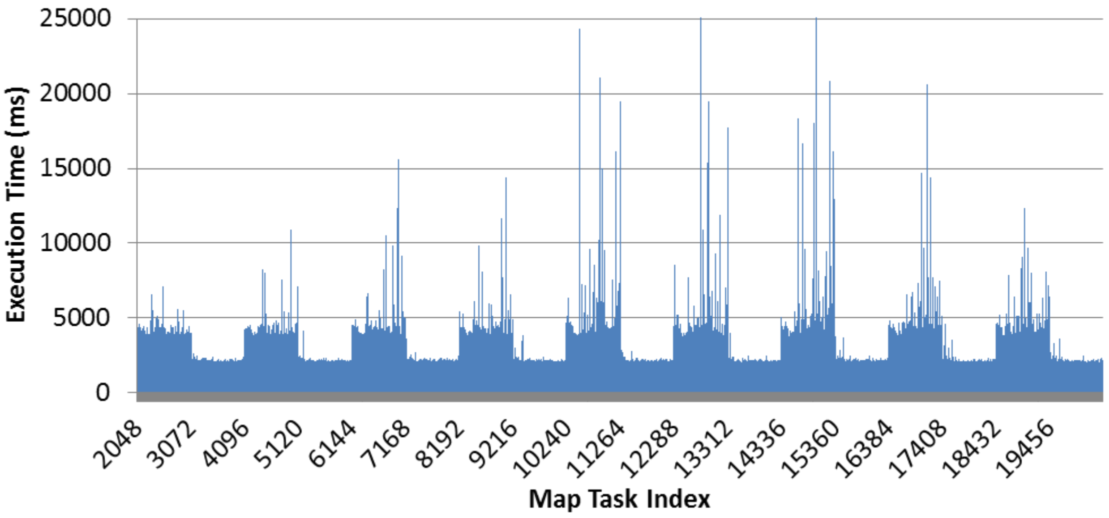


Figure 48 Execution traces of Twister4Azure MDS using in-memory caching on small instances. The taller bars represent the MDSBCCalc computation, while the shorter bars represent the MDSStressCalc computation, and together they represent an iteration.

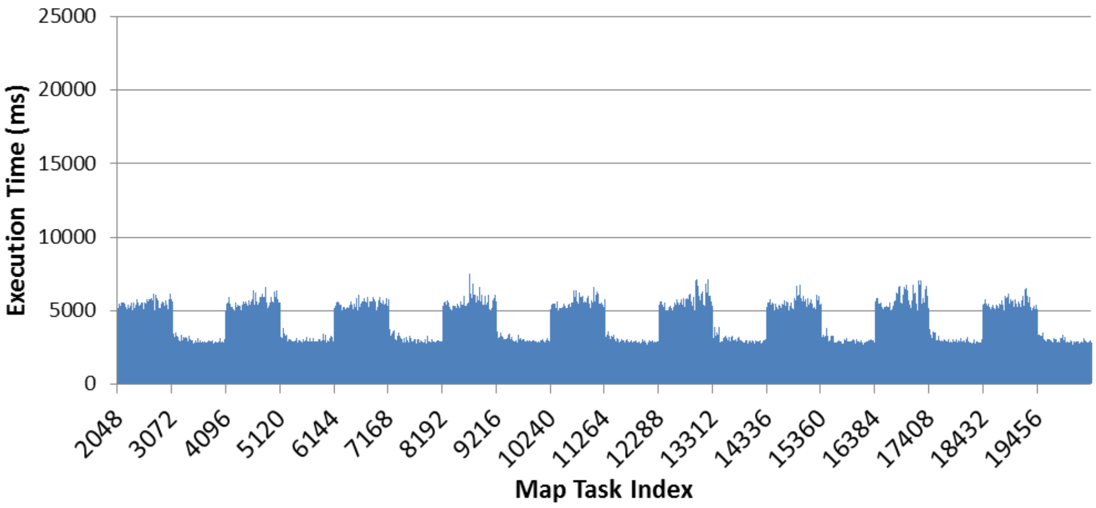


Figure 49 Execution traces of Twister4Azure MDS using Memory-Mapped file based caching on Large instances.

6.5 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 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) performs 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 provides 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 data sets such as these before using them with DryadLINQ, which will allow these applications to achieve the natural load balancing of the randomly distributed inhomogeneous data sets we described above.

We also observed that the fluctuation of MapReduce performance on clouds is minimal over a week-long 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.

7. Collective Communications Primitives for Iterative MapReduce

When performing distributed computations, often 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 [61, 72]. Collective communication operations facilitate the optimized communication and coordination between groups of nodes of a distributed computations; this leads to many advantages and makes it much easier and efficient to perform complex data communications inside the distributed parallel applications. Collective communication primitives are very popular in the HPC community and are used heavily in the MPI type of HPC applications. There has been much research [72] conducted to optimize the performance of these collective communication operations, as they have a significant 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.

In addition to the common characteristics of data-intensive iterative computations that we mentioned in section 6, we noticed several common communication and computation patterns among some of the data-intensive iterative MapReduce computations. This section highlights several Map-Collective communication primitives to support and optimize common computation and communication patterns in both MapReduce and iterative MapReduce computations. We present the applicability of Map-Collective operations to enhance (Iterative) MapReduce without sacrificing desirable MapReduce properties such as fault tolerance, scalability, familiar APIs and data model. The addition of collective communication operations enriches the MapReduce model by providing many performance and ease of use advantages. This includes providing efficient data communication operations optimized for particular execution environments and use cases, enabling programming models that fit naturally with application patterns, and allowing users to avoid overhead by skipping unnecessary steps of the execution flow.

We present these patterns as high level constructs that can be adopted by any MapReduce or iterative MapReduce runtime. We also offer proof-of-concept implementations of the primitives on Hadoop and Twister4Azure, and we envision a future where all the MapReduce and iterative MapReduce runtimes support a common set of Map-Collective primitives.

Our work focuses on mapping the All-to-All communication type of 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 MapReduceMergeBroadcast as an important collective in all (iterative) MapReduce frameworks.

This chapter presents 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 cases, collective communication operations virtually eliminated almost all the overheads of the computations.

The work of this chapter has been accepted for publication as a conference paper [73].

7.1 Collective Communication Primitives

Collective communication operations [61] facilitate optimized communication and coordination between groups of nodes of a distributed computation and are used heavily in the MPI type of HPC applications. These powerful operations make it much easier and efficient to perform complex data communications and coordination inside the distributed parallel applications. Collective communication also implicitly provides some form of synchronization across the participating tasks. There exist many different implementations of HPC collective communication primitives supporting numerous algorithms and topologies suited to different environments and use cases. The best 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, etc. 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 movement (aka data redistribution) communication primitives

  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 (aka collective operations) communication primitives

  This type of operations can be used to collect and consolidate data contributions from different worker processes. Examples of these include reduce, reduce-scatter and allreduce.

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

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

MapReduce model supports the All-to-One type communications through the Reduce step. MapReduce-MergeBroadcast model we introduce in section 7.2 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. Table 8 presents a summary of the support for above collective communication primitives in Hadoop, H-Collectives (section 7.6.1) and Twister4Azure.

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 types of implementations would be inefficient and would be harder to use compared to dedicated optimized implementations of All-to-All operations.

Table 8 Collective communications support in MPI, Hadoop, H-Collectives and Twister4Azure

	MPI	**Hadoop**	**H-Collectives**	**Twister4Azure**	Description [74]
All-to-One	**Gather**	shuffle-reduce*	shuffle-reduce*	shuffle-reduce-merge [section 7.2.2]	Gathers together values from a group of processes.
	Reduce	shuffle-reduce*	shuffle-reduce*	shuffle-reduce-merge [section 7.2.2]	Reduces values on all processes to a single value.
One-to-All	**Broadcast**	shuffle-reduce-distributedcache	shuffle-reduce-distributedcache	merge-broadcast [section 7.2.3]	Broadcasts a message to all other processes.
	Scatter	shuffle-reduce-distributedcache**	shuffle-reduce-distributedcache**	merge-broadcast **	Scatters data from one process to all other processes.
All-to-All	**AllGather**		**Map-AllGather**	**Map-AllGather**	Gathers data from all processes and distribute the result to all processes.
	AllReduce		**Map-AllReduce**	**Map-AllReduce**	Combines values from all processes and distribute the result back to all processes.
	Reduce-Scatter		**Map-ReduceScatter** (future work)	**Map-ReduceScatter** (future works)	Combines values from all the processes and scatters the result to all the processes.
Synchronization	**Barrier**	Barrier between Map & Reduce	Barrier between Map & Reduce and between iterations	Barrier between Map, Reduce, Merge and between iterations	Blocks until all process have reached the barrier.

* Use single Reduce task or by having a post processing step that will combine the output from multiple Reduce tasks.

** Workaround using Broadcast, where all the data is sent to all the processes rather than scattering the data.

7.2 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.

7.2.1 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 chapter, 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.

7.2.2 Merge Task

We define *Merge [12]* 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:

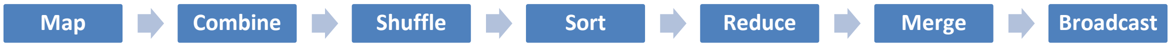


Figure 50 MapReduce-MergeBroadcast computation flow

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)

7.2.3 Broadcast

Broadcast 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 [75]. It is possible to share broadcast data between multiple Map and/or Reduce tasks executing on the same node, as MapReduce computations typically have more than one map/reduce/merge worker per worker-node.

7.2.4 MapReduceMergeBroadcast Cost Model

There exist several models that are frequently used by the message passing community to model to data communication performance [75]. We use the Hockney model [75, 76] 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). Hockney model cannot model the network congestion.

Merge is a single task that receives the outputs of all the reduce tasks. The cost of this transfer would be$r\alpha + n_r\beta$, where n$_r$ is the total number of reduce outputs and r is the number of reduce tasks. 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 cost [72] can be modeled as following, where n$_v$ is the total number of merge outputs (broadcast data).

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

Based on these costs, the total cost of a MapReduce-MergeBroadcast can be approximated as follows. T_{MR} is the cost of the MapReduce computation and $r\alpha + n_r\beta + f(n_r)$ presents the cost of the Merge task. 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.

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

7.2.5 Current iterative MapReduce Frameworks and MapReduce-MergeBroadcast

Twister4Azure [12] 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 [63] 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, which is semantically similar to the MapReduce-MergeBroadcast.

HaLoop [28] 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.

7.3 Collective Communications Primitives for Iterative MapReduce

While implementing iterative MapReduce applications using the MR-MB model, we started to notice several common execution flow patterns across the different applications. Some of these applications had very trivial Reduce and Merge tasks while other applications needed extra effort to map to the MR-MB model owing to the execution patterns being slightly different than the iterative MapReduce pattern. In order to solve such issues, we introduce Map-Collective primitives to the iterative MapReduce programming model, inspired by the MPI collective communications primitives [61].

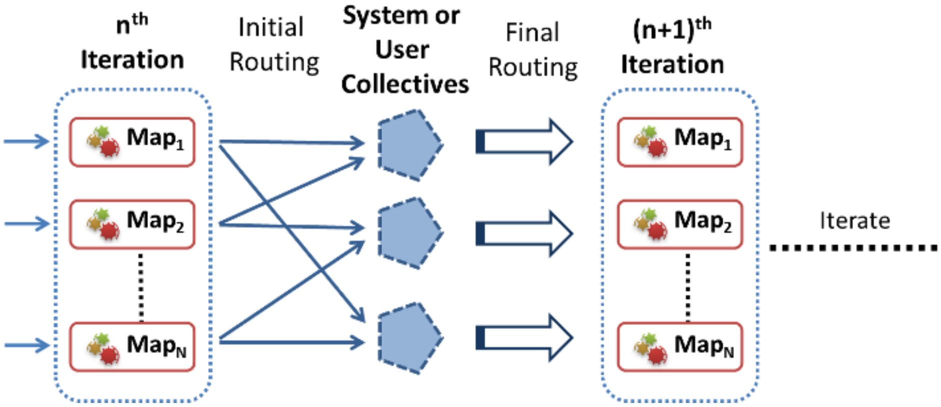


Figure 51 Map-Collective primitives

These primitives support higher-level communication patterns that occur frequently in data-intensive iterative applications by substituting certain steps of the MR-MB computation. As depicted in Figure 51, these Map-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 chapter we propose two collective communication primitive implementations: Map-AllGather and Map-AllReduce. You can also identify MR-MB as another collective communication primitive as well.

7.3.1 Requirements

When designing Map-collective primitives for iterative MapReduce, we should make sure they fit with the MapReduce data model and the MapReduce computational model, which support multiple Map task waves, large overheads, significant execution variations and inhomogeneous tasks. Also the primitives should retain scalability while keeping the programming model simple and easy to understand. These primitives should maintain the same type of framework-managed excellent fault tolerance supported by MapReduce.

7.3.2 Advantages

7.3.2.1 Performance improvement

Introduction of Map-Collective primitives provides 3 types of performance improvements to the iterative MapReduce applications. Map-Collectives can reduce the overheads of the computations by skipping or overlapping certain steps (e.g. shuffle, reduce, merge) of the iterative MapReduce computational flow. Map-Collective patterns also fit more naturally with the application patterns, avoiding the need for unnecessary steps.

Another advantage is the ability for the frameworks to optimize these operations transparently for the users, even allowing the possibility of different optimizations (poly-algorithm) for different use cases and environments. For example, a communication algorithm that’s best for smaller data sizes may not be the best for larger ones. In such cases, the Map-Collective operations can opt to have multiple algorithm implementations to be used for different data sizes.

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, presented in section 4, 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, like the hierarchical reduction in Map-AllReduce primitive.

7.3.2.2 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.

7.3.2.3 Scheduling with iterative primitives

In addition to providing synchronization between the iterations, Map-Collective primitives also give us the ability to propagate the scheduling information for the next iteration to the worker nodes along with the collective communication data. This allows the frameworks to synchronize and schedule the tasks of a new iteration or application with minimal overheads.

For example, as mentioned in section 8.6, Twister4Azure successfully employs this strategy to schedule new iterations with minimal overhead, while H-Collectives use this strategy to perform speculative scheduling of tasks.

7.3.3 Programming model

Map-Collective primitives can be specified as an outside configuration option without changing the MapReduce programming model. This permits the applications developed with Map-Collectives to be backward compatible with frameworks that don’t support them. This also makes it easy for developers who are already familiar with MapReduce programming to use Map-Collectives. For an example, a KMeans Clustering MapReduce implementation with Map, Reduce and Merge tasks can be used with

Map-AllReduce or vice versa without doing any changes to the Map, Reduce or Merge function implementations.

7.3.4 Implementation considerations

Map-Collectives can be add-on improvements to MapReduce frameworks. The simplest implementation would be implementing the primitives using the current MapReduce API and communication model on the user level, then providing the implementation as a library. This will achieve ease of use for the users by providing a unified programming model that better matches application patterns.

More optimized implementations can present these primitives as part of the MapReduce framework (or as a separate library) with the ability to optimize the data transfers based on environment and use case, using optimized group communication algorithms in the background.

Table 9 Summary of Map-Collectives 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 (matrix X matrix), PageRank (matrix X vector)
Map-AllReduce	Map→AllReduce (communication &	H-Collectives,	KMeansClustering, MDS-

	computation)	Twister4Azure	StressCalc

7.4 Map-AllGather Collective

AllGather is an all-to-all collective communication operation that gathers data from all the workers and distributes the gathered data to all the workers [72]. 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 Matrix-vector multiplication, where each map task outputs part of the resultant vector. In this computation we would use the Reduce and Merge tasks to assemble the vector together and then broadcast the assembled vector to workers.

Data-intensive iterative applications that have the AllGather pattern include MultiDimensionalScaling (matrix-matrix multiplication) [51] and PageRank using inlinks matrix (matrix-vector multiplication).

7.4.1 Model

We developed a Map-AllGather iterative MapReduce primitive similar to the MPI AllGather [72] collective communication primitive. Our intention was to support applications with communication patterns similar to the above in a more efficient manner.

7.4.2 Execution model

Map-AllGather primitive broadcasts the Map Task outputs to all computational nodes (all-to-all communication) of the current computation, and then assembles them together in the recipient nodes as depicted in Figure 52. Each Map worker will deliver its result to all other workers of the computation once the Map task is completed.

The computation and communication pattern of a Map-AllGather computation is Map phase followed by AllGather communication (all-to-all) followed by the AllGather combine phase. 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.

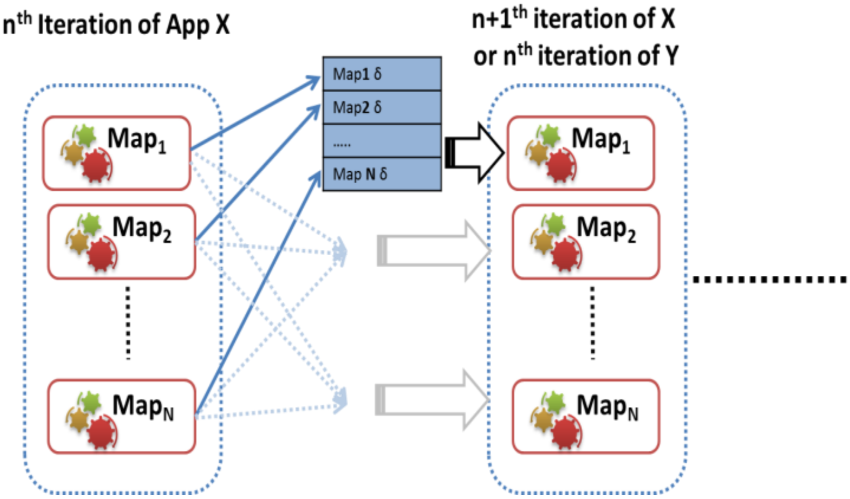


Figure 52 Map-AllGather Collective

7.4.3 Data Model

For Map-AllGather, the map output key should be an integer specifying the location of the output value in the resultant gathered data product. Map output values can be vectors, sets of vectors (partial matrix) or single values. Final output value of the Map-AllGather operation is an assembled array of Map output values in the order of their corresponding keys. The result of AllGather-Combine will be provided to the Map tasks of the next iteration as the loop variant data using the APIs and mechanisms suggested in Section 8.2.2.1.

The final assembly of AllGather data can be performed by implementing a custom combiner or using the default combiner of AllGather-combine. Custom combiner allows the user to specify a custom assembling function. In this case, the input to the assembling function is a list of Map outputs key-value pairs, ordered by the key. This assembling function gets executed in each worker node after all the data is received.

The default combiner should work for most of the use cases, as the combining of AllGather data is oftentimes a trivial process. The default combiner expect the Map outputs to be in <int, double[]> format. In a 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 therefore ignored.

Users can deploy their Mapper implementations as is with Map-AllGather primitive. They need to specify only the collective operation, after which the shuffle and reduce phases of MapReduce would be substituted by the Map-AllGather communication and computations.

7.4.4 Cost Model

An optimized implementation of AllGather, such as a by-directional exchange based implementation [72], we can estimate the cost of the AllGather component as following, where *m* is the number of map tasks.

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

We present the above cost model to demonstrate the communication cost improvements achievable using a highly optimized implementation of this primitive and not as an effort to model the performance of the current implementation. This model won't be used to performance modeling of the current implementations.

It is 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.

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 as well.

7.4.5 Fault tolerance

All-Gather partial data product transfers from Map to workers can fail due to communication mishaps and other breakdowns. When task level fault tolerance (typical MapReduce fault tolerance) is enabled, it is possible for the workers to read the missing map output data from the persistent storage (e.g.HDFS) to successfully perform the All-Gather computation.

The fault tolerance and the speculative execution of MapReduce enable duplicate execution of tasks. Map-AllGather can perform the duplicate data detection before the final assembly of the data at the recipient nodes to handle any duplicate executions.

7.4.6 Benefits

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

Oftentimes the Map task execution times are inhomogeneous [43] in typical MapReduce computations. Implementations of Map-AllGather primitive can start broadcasting the map task result values as soon as the first map task is completed. 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 even more significant when we have multiple waves of map tasks.

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

7.5 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 [72]. 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.

7.5.1 Model

We propose Map-AllReduce iterative MapReduce primitive similar to the MPI AllReduce [72] collective communication operation, to efficiently aggregate and reduce the results of the Map Tasks.

7.5.1.1 Execution Model

The computation and communication pattern of a Map-AllReduce computation is Map phase followed by the AllReduce communication and computation (reduction), as depicted in Figure 53. As we can notice, this model allows us to substitute the shuffle->sort->reduce->merge->broadcast steps of the MapReduce-MergeBroadcast with AllReduce communication in the communication layer. The AllReduce phase can be implemented efficiently using algorithms such as bidirectional exchange (BDE) [72] or hierarchical tree based reduction.

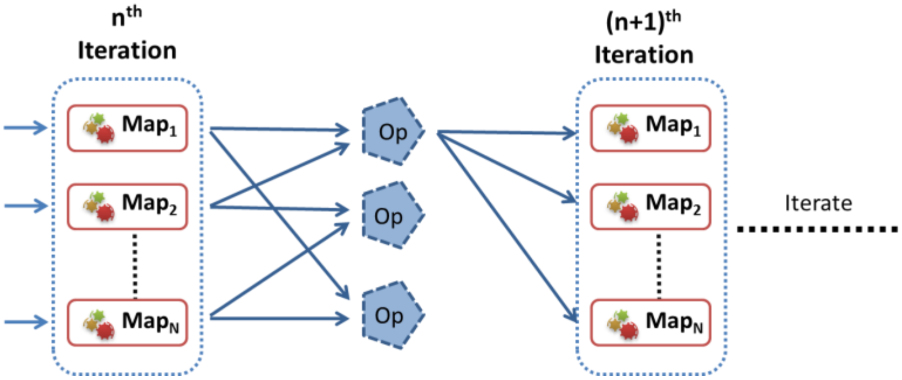


Figure 53 Map-AllReduce collective

Map-AllReduce allows the implementations to perform local aggregation on the worker nodes across multiple map tasks and to perform hierarchical reduction of the Map Task outputs while delivering them to all the workers. Map-AllReduce performs the final reduction in the recipient worker nodes.

7.5.1.2 Data Model

For Map-AllReduce, the map output values should be vectors or single values of numbers. The values belonging to each distinct map output key are processed as a separate data reduction operation. Output of the Map-AllReduce operation is a list of key/value pairs where each key corresponds to a map output key and the value is the combined value of the map output values that were associated with that map output key. As shown in Figure 54, the number of records in the Map-AllReduce output is equal to the number of unique map output keys. For example, 10 distinct Map output keys would result in 10 combined vectors or values. Map output value type should be a number.

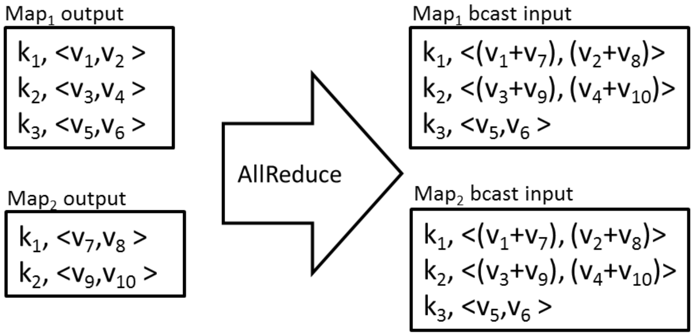


Figure 54 Example Map-AllReduce with Sum operation

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 the Sum operation together with an additional element (dimension) to count the number of data products. Due to the associative and commutative nature of the operations, Map-AllReduce has the ability to start combining the values as soon as the first map task completion. It also allows the Map-AllReduce implementations to use reduction trees or bidirectional exchanges to optimize the operation.

It is also possible to allow users to specify a post process function that executes after the AllReduce communication. This function can be used to perform a simple operation on the Map-AllReduce result or to check for the iteration termination condition. It would be executed in each worker node after all the Map-AllReduce data has been received.

```
list<Key, IOpRedValue>  PostOpRedProcess(list<Key, IOpRedValue> opRedResult);
```

7.5.1.3 Cost Model

An optimized implementation of AllReduce, such as a by-directional exchange based implementation [72], will reduce the cost of the AllReduce component to,

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

We present the above cost model to demonstrate the communication cost improvements achievable using a highly optimized implementation of this primitive and not as an effort to model the performance of the current implementation. This model won't be used to performance modeling of the current implementations.

It is also possible to further reduce this cost by performing local aggregation and reduction in the Map worker nodes as the compute cost of AllReduce is very small. $\frac{m}{p}$ gives the average number of Map tasks per computation that executes in a given worker node, where p is the number of worker nodes. 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.

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

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.

Other efficient algorithms to implement AllReduce communication include flat-tree/linear, pipeline, binomial tree, binary tree, and k-chain trees [75].

7.5.2 Fault Tolerance

If the AllReduce communication step fails for some reason, it is possible for the workers to read the 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. The most trivial fault tolerance model for Map-AllReduce would be a best-effort mechanism, where Map-AllReduce would fall back to using the Map output results from the persistent storage (e.g. HDFS) in case duplicate results are detected. Duplicate detection can be done by maintaining a set of map IDs with each combined data product. It is possible for the frameworks to implement richer fault tolerance mechanisms, such as identifying the duplicated values in localized areas of the reduction tree.

7.5.3 Benefits

Map-AllReduce reduces 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 optimize the computation by performing hierarchical reductions, reducing the number and the size of intermediate data communications.

Hierarchical reduction can be performed in as many levels as needed based on the size of the computation and the scale of the environment. For example, first level in mappers, second level in the node and nth level in rack level, etc. The mapper level would be similar to the "combine" operation of vanilla MapReduce. 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.

7.6 Implementations

In this section we present two implementations of Map-Collectives on Hadoop MapReduce and Twister4Azure iterative MapReduce.

These implementations are proofs of concept presenting sufficiently optimal implementations for each of the primitives and the environments to show the performance efficiencies that can be gained through using even a modest implementation of these operations. It is possible to further optimize these implementation using more advanced communication 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[72]. One of the main advantages of these primitives is the flexibility to improve primitive implementations without the need to change the user application, making it possible to optimize these implementations in the future as future work.

It is not our objective to find the most optimal implementations for each of the environments, especially for clouds where the most optimal implementation might end up being a moving target due to the rapidly evolving nature and the black box nature of cloud environments. This presents an interesting opportunity for cloud providers to develop optimized implementations of these primitives as cloud infrastructure services that can be utilized by the framework developers.

7.6.1 H-Collectives: Map-Collectives for Apache Hadoop

H-Collectives is a Map-Collectives implementation for Apache Hadoop that can be used as a drop in library with the Hadoop distributions. H-Collectives uses the Netty NIO library, node-level data aggregations and caching to efficiently implement the collective communications and computations. Existing Hadoop Mapper implementations can be used with these primitives with only very minimal changes. These primitives work seamlessly with Hadoop dynamic scheduling of tasks, support for multiple map task waves and other desirable features of Hadoop, while supporting the typical Hadoop fault tolerance and speculative executions as well.

A single Hadoop node may run several Map workers and many more map tasks belonging to a single computation. The H-Collectives implementation maintains a single node-level cache to store and serve the collective results to all the tasks executing in a worker node.

H-Collectives speculatively schedules the tasks for the next iteration and the tasks are waiting to start as soon as all the AllGather data is received, getting rid of most of the Hadoop job startup/cleanup and task scheduling overheads. Speculative scheduling cannot be used easily with pure Hadoop MapReduce as we need to add the loop variant data (only available after the previous iteration is finished) to the Hadoop DistributedCache before scheduling the job.

7.6.1.1 H-Collectives Map-AllGather

This implementation performs simple TCP-based best effort broadcasts for each Map task output. Task output data are transmitted as soon as a task is completed, taking advantage of the inhomogeneous Map task completion times. Final aggregation of these data products are done at the destination nodes only once per node. If an AllGather data product is not received through the TCP broadcasts, then it will be fetched from the HDFS.

7.6.1.2 H-Collectives Map-AllReduce

H-Collectives Map-AllReduce use n'ary tree-based hierarchical reductions, where Map task level and node level reductions would be followed by broadcasting of the locally aggregated values to the other worker nodes. 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.

7.6.2 Map-Collectives for Twister4Azure iterative MapReduce

Twister4Azure Map-Collectives are implemented using the Windows Communication Foundation (WCF)-based Azure TCP inter-role communication mechanism, while using the Azure table storage as a persistent backup.

Twister4Azure primitive implementations maintain a worker node-level cache to store and serve the primitive result values to multiple Map workers and map tasks running on a single server. Twister4Azure utilizes the collectives to perform synchronization at the end of each iteration and also to aid in the decentralize scheduling of the tasks of the next iteration by using the collective operations to communicate the new iteration information to the workers.

7.6.2.1 Map-AllGather

Map-AllGather performs simple TCP-based broadcasts for each Map task output, which is an all-to-all linear implementation. Workers start transmitting the data as soon as a task is completed. The final aggregation of the data is performed in the destination nodes and is done only once per node.

7.6.2.2 Map-AllReduce

Map-AllReduce 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. The iteration check happens in the destination nodes and can be specified as a custom function or as a limit on the number of iterations.

7.6.3 Implementation Considerations on cloud environments

As mentioned above, our goal of the above prototype implementations is to demonstrate the viability of Collective Communication primitives for iterative MapReduce. However, here we discuss the differences and challenges one would encounter when implementing highly optimized collective communications primitives for clouds as oppose to for local clusters.

Cloud environments are shared virtualized environments that are known to be relatively less reliable than the local cluster counterparts. The failures can be whole instance failures as well as individual communication operation failures. One of the most important considerations when implementing Map-Collectives to the cloud environments should be to ensure the fault tolerance of the communications as well as of the whole computation. One option is to explore the ability to utilize the fault tolerant high available cloud services to perform the optimized communication for the Map-Collectives. Another option is to make sure the data is persisted and available, to use in case the framework has to retry a communication operation or to facilitate the use of an alternate method of communication in case the main algorithm fails due to some reason. Twister4Azure Map-Collectives implementation takes this approach where the data is persisted in the background to a Cloud storage while the optimized communication is performed. It is also important to keep in mind that any data stored to cloud instance storage would get lost in case of an instance failure or a decommission of an instance.

Communications in Cloud instances have higher latencies than their bare metal counterparts due to the virtualization overhead in the network layer and also due to the shared and commodity nature of the network interconnect. Bandwidth available for the data communication can also be lesser due to the shared nature and due to the usage of commodity interconnects. A balance needs to be achieved in terms of the data communication parallelism and the number and size of the individual data messages. Data communications should be sufficiently parallel to avoid bandwidth bottlenecks in any particular path that may result from the shared nature of the cloud environments. At the same time, the data transfers should result in a relatively smaller number of coarser grained messages, to avoid the high latencies and to reduce the management and fault recovery cost of messages.

Also it is important to select proper cloud instances for the computations. If a relatively large number of cores are needed for the computation, it is better to use the largest available instance type that does not result in a cost overhead. In almost all the cloud environments, the usage of the largest available instance would ensure exclusive access to a physical node without sharing it with another user. Usage of larger instances enables meaningful local aggregations for the collective communications across the multiple workers running in a single instance and to use a cache to share collective results data with all the workers executing inside a single instance. We employ both local aggregations and results cache in our prototype implementations. Larger instances would alleviate some of concerns of a shared environment as well.

The black box nature of the network architecture of cloud environments rules out any topology specific communication algorithms and any algorithm that requires exclusive use of the network. Communication algorithms in cloud environments for iterative MapReduce collective communications should take advantage of the inhomogeneous nature of the tasks in a computation to reduce and even out the load in the network interconnects. Also they should be able to take advantage of the computation and communication overlap ability provided by the multiple waves of map tasks.

7.7 Evaluation

In this section we evaluate and compare the performance of Map-Collectives with plain MapReduce using two real world applications, Multi-Dimensional-Scaling and K-means clustering. The performance results are presented by breaking down the total execution time in to the different phases of the MapReduce or the Map-Collectives computations. This provides an idea of the performance model and provides a better view of various overheads of MapReduce and the optimizations provided by Map-Collectives to reduce some of those overheads.

In the following figures, 'Scheduling' is the per iteration (per MapReduce job) startup and task scheduling time. 'Cleanup' is the per iteration overhead from reduce task execution completion to the iteration end. 'Map overhead' is the start and cleanup overhead for each map task. 'Map variation' is the overhead due to variation of data load, compute and map overhead times. 'Comm+Red+Merge' is the time for map to reduce data shuffle, reduce execution, merge and broadcast. 'Compute' and 'Data load' times are calculated using the average compute only and data load times across all the tasks of the computation. The common components (data load, compute) are plotted at the bottom of the graphs to highlight variable components.

Hadoop and H-Collectives experiments were conducted in the FutureGrid Alamo cluster, which has Dual Intel Xeon X5550 (8 total cores) per node, 12 GB RAM per node and a 1Gbps network. Twister4Azure tests were performed in Windows Azure cloud, using Azure extra-large instances. Azure extra-large instances provide 8 compute cores and 14 GB RAM per instance.

7.7.1 Multi-Dimensional Scaling (MDS) using Map-AllGather

The objective of MDS, described in detail in section 2.6.6, is to map a dataset in high-dimensional space to a lower dimensional space, with respect to the pairwise proximity of the data points [51]. In this chapter, we use parallel SMACOF [46, 52] MDS, which is an iterative majorization algorithm. The input for MDS is an N*N matrix of pairwise proximity values. The resultant lower dimensional mapping in D dimensions, called the X values, is an N*D matrix.

Unweighted MDS results in two MapReduce jobs per iteration, BCCalc and StressCalc. Each BCCalc Map task generates a portion of the total X matrix. The reduce step of MDS BCCalc computation is an aggregation operation, which simply assembles the outputs of the Map tasks together in order. This X value matrix is then broadcasted to be used by the StressCalc step of the current iterations, as well as by the BCCalc step of the next iteration. MDS performs relatively smaller amount of computations for a unit of input data. Hence MDS has larger data loading and memory overhead. Usage of the Map-AllGather primitive in MDS BCCalc computation eliminates the need for reduce, merge and broadcasting steps in that particular computation.

7.7.1.1 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*.

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.

7.7.1.2 H-Collectives MDS Map-AllGather

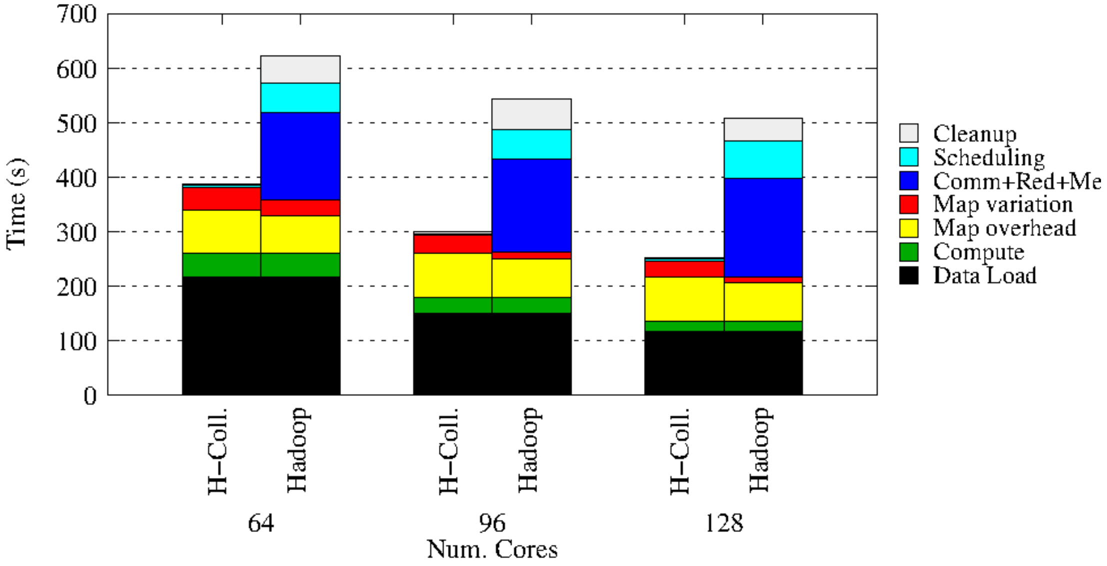


Figure 55 MDS Hadoop using only the BC Calculation MapReduce job per iteration to highlight the overhead. 20 iterations, 51200 data points

We implemented the MDS for Hadoop using vanilla MapReduce and H-Collectives Map-AllGather primitive. Vanilla MapReduce implementation uses the Hadoop DistributedCache to broadcast loop variant data to the Map tasks. Figure 55 shows the MDS strong scaling performance results highlighting the overhead of different phases on the computation. We used only the BC Calculation step of the MDS in each iteration and skipped the stress calculation step to further highlight the AllGather component. This test case scales a 51200*51200 matrix into a 51200*3 matrix. 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.

As we can notice in the Figure 55, the H-Collectives implementation gets rid of the communication, reduce, merge, task scheduling and job cleanup overhead of the vanilla MapReduce computation. However, we notice a slight increase of Map task overhead and Map variation in the case H-Collectives Map-AllReduce-based implementation. We believe these increases are due to the rapid scheduling of Map tasks across successive iterations in H-Collectives, whereas in the case of vanilla MapReduce the map tasks of successive iterations have few seconds between the scheduling do perform housekeeping tasks.

7.7.1.3 Twister4Azure MDS-AllGather

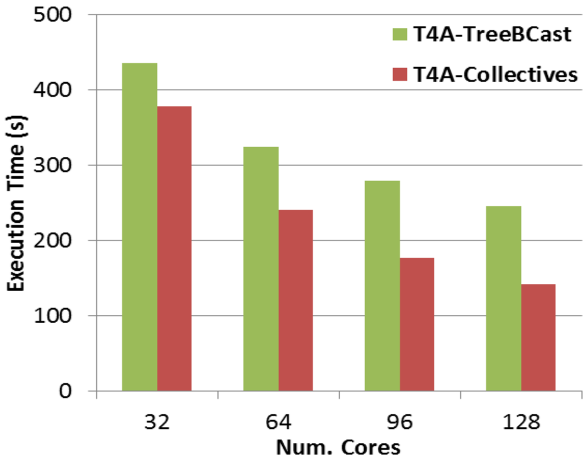


Figure 56 MDS application implemented using Twister4Azure. 20 iterations. 51200 data points (~5GB).

We implemented MDS for Twister4Azure using Map-AllGather primitive and MR-MB with optimized broadcasting. Twister4Azure optimized broadcast is an improvement over simple MR-MB as it uses an optimized tree-based algorithm to perform TCP broadcasts of in-memory data. Figure 56 shows the MDS (with both BCCalc and StressCalc steps) strong scaling performance results comparing the Map-AllGather based implementation with the MR-MB implementation. 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 by 13%-42% over MapReduce with optimized broadcast in the current test cases.

7.7.1.4 Detailed analysis of overheads

In this section we perform detailed analysis of overheads of the Hadoop MDS BCCalc 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.

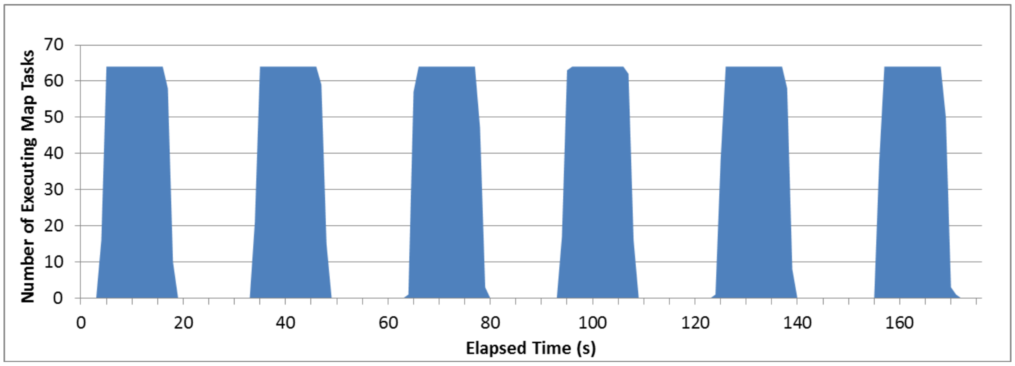


Figure 57 Hadoop MapReduce MDS-BCCalc histogram

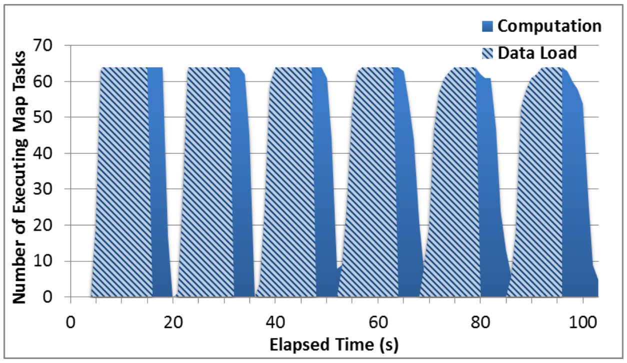


Figure 58 H-Collectives AllGather MDS-BCCalc histogram

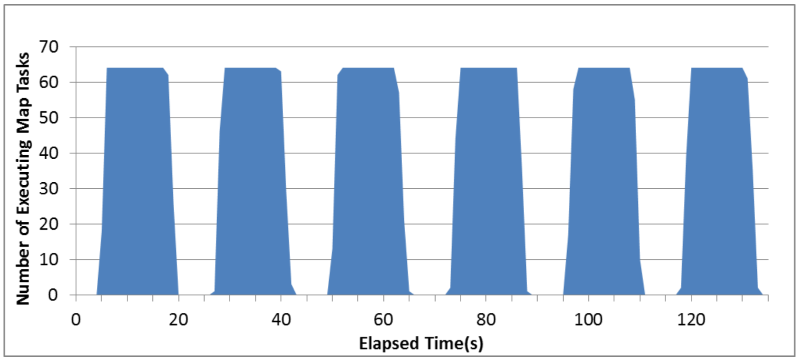


Figure 59 H-Collectives AllGather MDS-BCCalc histogram without speculative scheduling

Figure 58 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 59 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 58 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.

7.7.1.5 Twister4Azure vs Hadoop

Twister4Azure is already optimized for iterative MapReduce [12] 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 58. Twister4Azure avoids this cost by using data caching and cache aware scheduling.

7.7.2 K-KMeansClustering using Map-AllReduce

The K-means Clustering [67] algorithm, described in detail in section 2.6.5, 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.

7.7.2.1 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*.

KMeansClustering 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.

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.

The compute cost difference between KMeansClustering MapReduce-MergeBroadcast and Map-AllReduce implementations is equal or slightly in favor of the MapReduce due to the hierarchical reduction performed in the AllReduce implementation. However, typically the compute cost of the reduction is almost negligible. All the other overheads including the startup overhead, disk overhead and communication overhead favors the AllReduce based implementation.

7.7.2.2 H-Collectives KMeansClustering-AllReduce

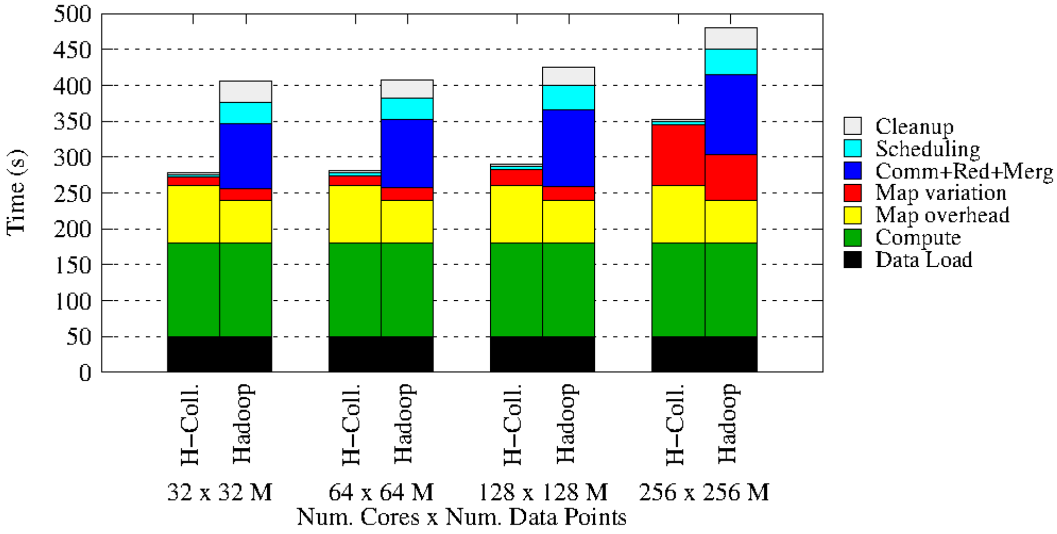


Figure 60 Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce Weak scaling. 500 Centroids (clusters). 20 Dimensions. 10 iterations.

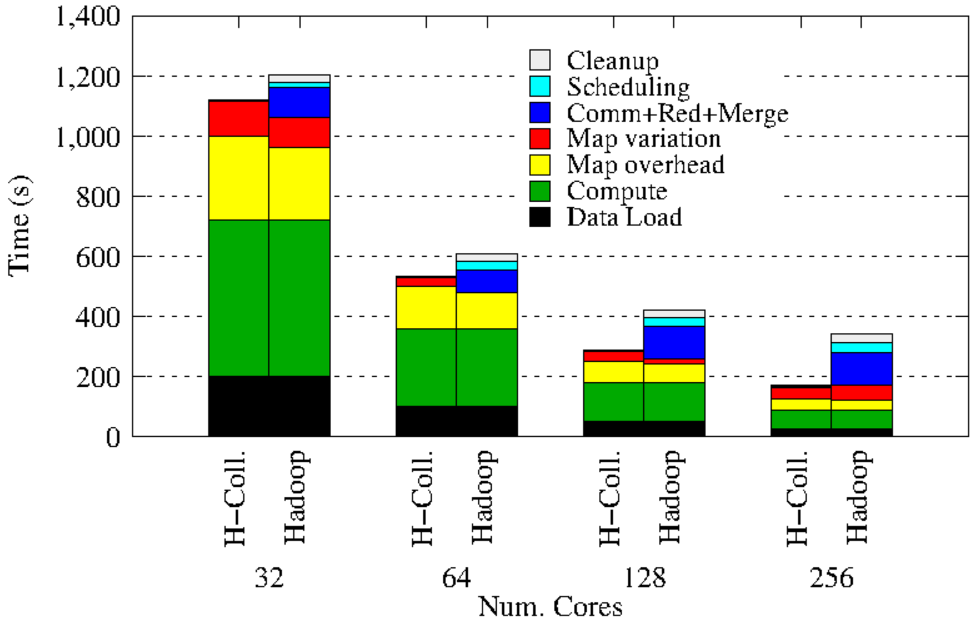


Figure 61 Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce Strong scaling. 500 Centroids (clusters). 20 Dimensions. 10 iterations.

We implemented the K-means Clustering application for Hadoop using the Map-AllReduce and plain MapReduce. The MapReduce implementation uses in-map combiners to perform aggregation of the values to minimize the size of map-to-reduce intermediate data transfers.

Figure 60 illustrates the K-means Clustering weak scaling performance where we scaled the computation while keeping the workload per core constant. Figure 61 presents the K-means Clustering strong scaling performance where we scaled the computation while keeping the data size constant. Strong scaling test cases with smaller number of nodes use more map task waves optimizing the intermediate data communication, resulting in relatively smaller overhead for the computation

As we can see, the H-Collectives implementation gets rid of the communication, reduce, merge, task scheduling and job cleanup overhead of the vanilla MapReduce computation. A slight increase of Map task overhead and Map variation can be noticed in the case of Map-AllReduce based implementation, similar to the behavior observed and explained in above MDS section.

7.7.2.3 Twister4Azure KMeansClustering-AllReduce

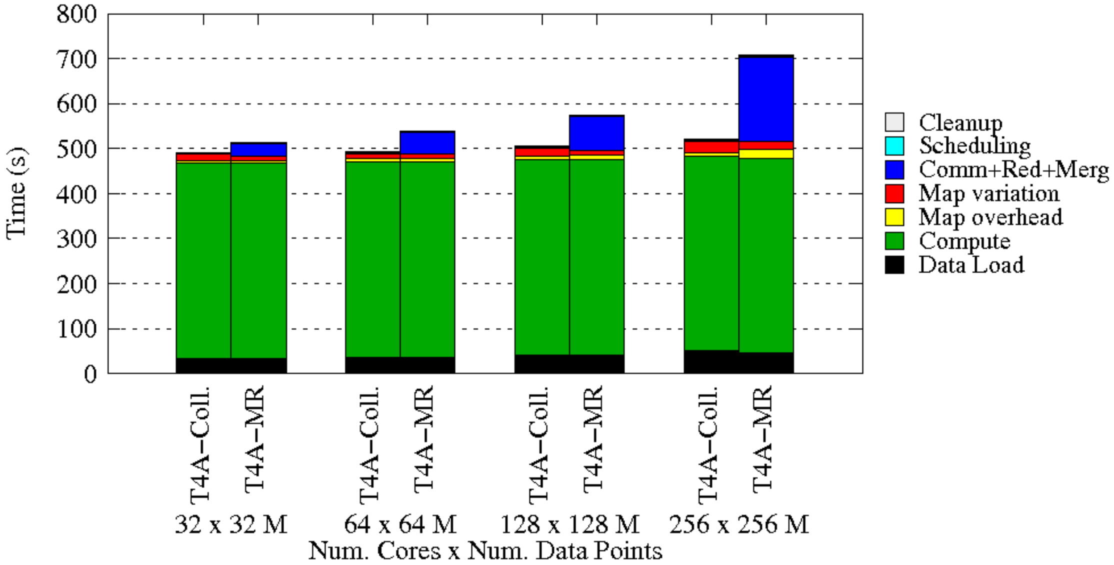


Figure 62 Twister4Azure K-means weak scaling with Map-AllReduce. 500 Centroids, 20 Dimensions. 10 iterations. 32 to 256 Million data points.

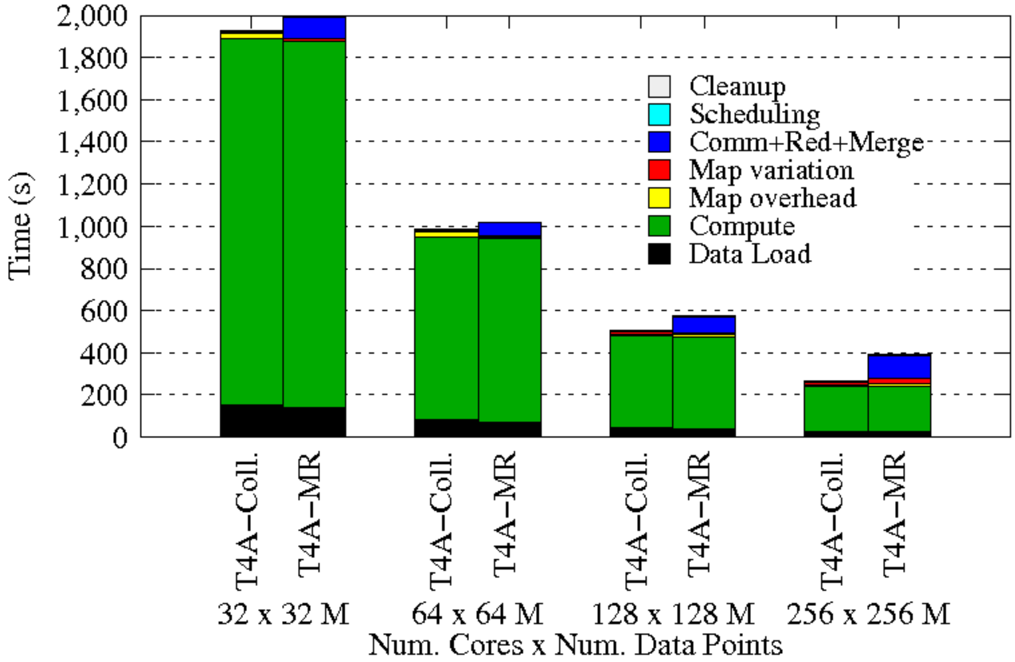


Figure 63 Twister4Azure K-means Clustering strong scaling. 500 Centroids, 20 Dimensions, 10 iterations. 128Million data points.

We implemented the K-means Clustering application for Twister4Azure using the Map-AllReduce primitive and MapReduce-MergeBroadcast. MR-MB implementation uses in-map combiners to perform local aggregation of the output values to minimize the size of map-to-reduce data transfers. Figure 62 shows the K-means Clustering weak scaling performance results, where we scale the computations while keeping the workload per core constant. Figure 63 presents the K-means Clustering strong scaling performance, where we scaled the number of cores while keeping the data size constant. As can be seen in these figures, the Map-AllReduce implementation gets rid of the communication, reduce and merge overheads of the MR-MB computation.

7.7.2.4 Twister4Azure vs Hadoop vs HDInsight

KMeans performs more computation per data load than MDS and the compute time dominates the run time. The pure compute time in of C#.net based application in Azure is much slower than the java based application executing in a Linux environment. Twister4Azure is still able to avoid lot of overheads and improves the performance of the computations, but the significant lower compute time results in lower running times for the Hadoop applications.

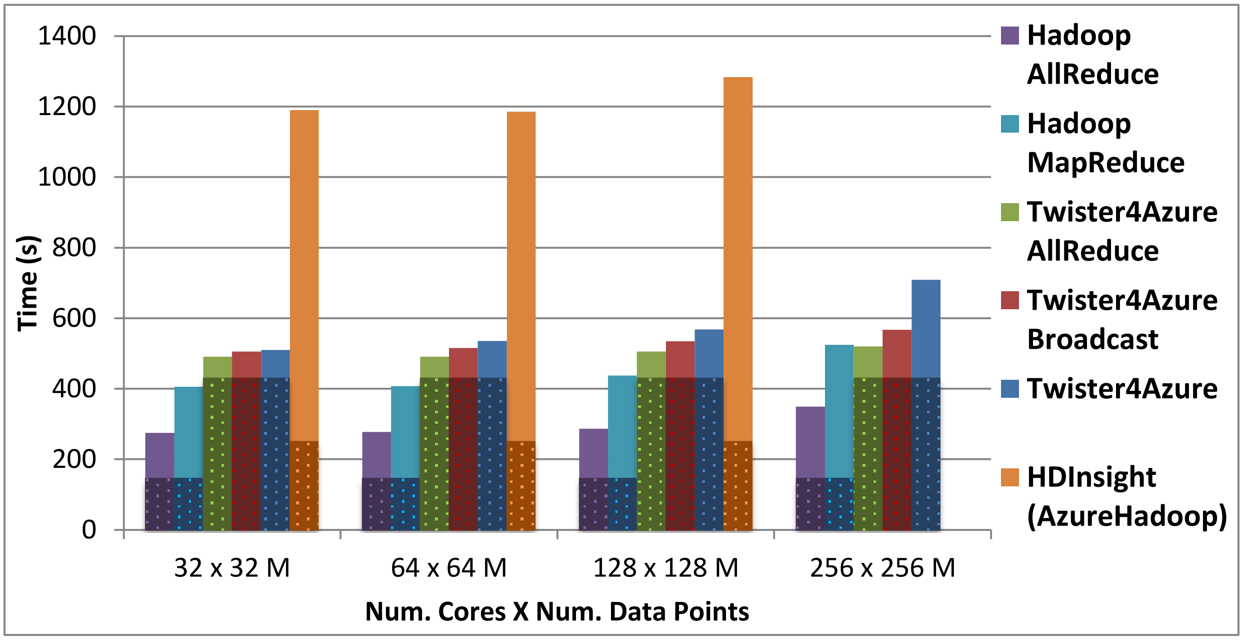


Figure 64 HDInsight KMeans Clustering compared with Twister4Azure and Hadoop

HDInsight offers hosted Hadoop as a service on the Windows Azure cloud. Figure 64 presents the KMeansClustering performance on the HDInsight service using Windows Azure large instances. We executed the same Hadoop MapReduce based KMeansClustering implementation used in section 7.2.3 on HDInsight. HDInsight currently limits the number of cores to 170, which doesn't allow us to perform the 256 core test on it.

Input data for the HDInsight computation were stored in Azure Blob Storage and were accessed through ASV (Azure storage vault), which provides a HDFS file system interface for the Azure blob storage. Input data for the Twister4Azure computation were also stored in Azure blob storage and were cached in memory using the Twister4Azure caching feature.

The darker areas of the bars represent the approximated compute only time for the computation based on the average map task compute only time. Rest of the area in each bars represent the overheads, which would be the time taken for task scheduling, data loading, shuffle, sort, reduce, merge and broadcast. The overheads are particularly high for HDInsight due to the data download from the Azure Blob storage for each iteration. The variation of the time to download data from the Azure Blob storage adds significant variation to the map task execution times affecting the whole iteration execution time.

Twister4Azure computation is significantly faster than HDInsight due to the data caching and other improvements such as hybrid TCP based data shuffling, cache aware scheduling etc. , even though the compute only time (darker areas) is much higher in Twister4Azure (C# vs Java) than in HDInsight.

7.8 Summary

In this section, we introduced Map-Collectives, collective communication operations for MapReduce inspired by MPI collectives, as a set of high level primitives that encapsulate some of the common iterative MapReduce application patterns. Map-Collectives improve the communication and computation performance of the applications by enabling highly optimized group communication across the workers, by getting rid of unnecessary/redundant steps and by enabling the frameworks to use a poly-algorithm approach based on the use case. Map-Collectives also improve the usability of the MapReduce frameworks by providing abstractions that closely resemble the natural application patterns and reduce the implementation burden of the developers by providing optimized substitutions to certain steps of the MapReduce model. We envision a future where many MapReduce and iterative MapReduce frameworks support a common set of portable Map-Collectives, and we consider this work as a step towards that.

We defined Map-AllGather and Map-AllReduce Map-Collectives and implemented Multi-Dimensional Scaling and K-means Clustering applications using these operations. We also presented the H-Collectives library for Hadoop, which is a drop-in Map-Collectives library that can be used with existing MapReduce applications with only minimal modification. We also presented a Map-Collectives implementations for the Twister4Azure iterative MapReduce framework as well. MDS and K-means applications were used to evaluate the performance of Map-Collectives on Hadoop and on Twister4Azure depicting up to 33% and 50% speedups over the non-collectives implementations by getting rid of the communication and coordination overheads.

8. Conclusions and Future Works

8.1 Summary and Conclusions

In this thesis, we have investigated the applicability of cloud computing environments and related application frameworks to be able to perform large-scale data intensive parallel computations efficiently with good scalability, fault-tolerance and ease-of-use. Over the course of our work, we have acquired greater understanding about the challenges and bottlenecks involved in performing scalable data-intensive parallel computing on cloud environments; we have proposed solutions to overcome these potential obstacles. We selected pleasingly parallel computations, MapReduce type computations and iterative MapReduce type computations as the types of computations that are better suited for execution in cloud environments. We developed scalable parallel programming and computing frameworks specifically designed for cloud environments to support efficient, reliable and user friendly execution of the above three types of computations on cloud environments. Further, we developed data intensive applications using those frameworks, and demonstrated that these applications can be executed on cloud environments in an efficient scalable manner.

In Chapter 3, we introduced a set of frameworks that have been constructed using cloud-oriented programming models to perform pleasingly parallel computations on cloud and cluster environments. Using these frameworks, we demonstrated the feasibility of Cloud infrastructures for the implementation of pleasingly parallel applications such as the Cap3 sequence assembly, the BLAST sequence search and Generative Topographic Mapping (GTM) Interpolation. We analyzed and compared each of these frameworks by performing a comparative study among them based on performance, cost and usability. For the applications we considered, we developed frameworks on top of high latency, eventually consistent cloud infrastructure services that relied on off-the-instance cloud storage; these frameworks were able to exhibit performance efficiencies and scalability comparable to the MapReduce based frameworks with local disk-based storage. In Chapter 3, we also analyzed the variations in cost among the different platform choices (e.g., EC2 instance types), by highlighting the importance of selecting an appropriate platform based on the nature of the computation. We used Amazon Web Services [9] and Microsoft Windows Azure [52] cloud computing platforms, in addition to Apache Hadoop [6] MapReduce and Microsoft DryadLINQ [7] as the distributed parallel computing frameworks.

While models like Classic-Cloud, which we introduced in Chapter 3, 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 we saw in Chapter 3, we developed a fully-fledged MapReduce framework with iterative-MapReduce support for the Windows Azure Cloud infrastructure using Azure infrastructure services as building blocks which provided users the best of both worlds.

In Chapter 4, we introduced a novel decentralized controlled cloud infrastructure services-based MapReduce architecture for cloud environments as well as an implementation of that architecture for the Windows Azure cloud environment, called MRRoles4Azure. MRRoles4Azure fulfilled the much-needed requirement of a distributed programming framework for Windows Azure cloud users. MRRoles4Azure was built using Azure cloud infrastructure services that take advantage of the quality of service guarantees provided by the Azure cloud. Even though cloud services have higher latencies than their traditional counterparts, scientific applications implemented using MRRoles4Azure were able to perform comparatively with the other MapReduce implementations, thus proving the feasibility of the MRRoles4Azure architecture. We also explored the challenges presented by cloud environments to execute MapReduce computations and we discussed how we overcame them in the MRRoles4Azure architecture. We also implemented and analyzed the performance of two MapReduce applications on two popular cloud infrastructures. In our experiments, the MapReduce applications executed in the cloud infrastructures exhibited performance and efficiency characteristics comparable to the MapReduce applications that were executed using traditional clusters. We demonstrated that using MapReduce in cloud environments is a very viable option, as it exhibited performance results comparable to in house clusters, because of its on demand availability, horizontal scalability and its' easy to use programming model; in addition, it poses no upfront costs. This option is also 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.

In Chapter 5, we presented Twister4Azure, a novel iterative MapReduce distributed computing runtime for Windows Azure Cloud. Twiser4Azure enables the users to perform large-scale data intensive iterative computations efficiently on Windows Azure Cloud, by hiding the complexity of scalability and fault tolerance typically present when using Clouds. The key features of Twiser4Azure 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 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. Twister4Azure contains MRRoles4Azure MapReduce capabilities and the Classic-Cloud pleasingly parallel framework capabilities

We presented four real world data intensive applications which were implemented using Twister4Azure and compared the performance of those applications with that of the Twister (Java) and the 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 Twister (Java) 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 exhibited performance comparable to that of Twister (Java) 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.

In Chapter 6, we discussed some of the performance implications of performing scalable parallel computing on cloud environments. These include data inhomogeneity, virtualization overhead, performance variations in clouds infrastructures and various data caching options. Many real world data sets and problems are inhomogeneous in nature, a characteristic that makes it difficult to divide those computations into equally balanced computational parts. But often, the inhomogeneity of problems is randomly distributed, and this provides a natural load balancing inside the sub tasks of a computation. We observed that the scheduling mechanism employed by both dynamic scheduling (Hadoop) and static scheduling (DryadLINQ) performs well when randomly distributed inhomogeneous data is used. Also, in the above study, we observed that, when there are sufficient map tasks, the global queue based dynamic scheduling strategy adopted by Hadoop (and also by MRRoles4Azure and Twister4Azure) provides load balancing even in extreme scenarios like skewed distributed inhomogeneous data sets.

We also observed that the fluctuation of MapReduce performance on clouds is minimal over a week-long period, assuring consistency and predictability of application performance in the cloud environments. We also performed experiments 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 test results showed that the average virtual machines overhead is 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 these performance anomalies.

Finally, in Chapter 7, we introduced Map-Collectives, collective communication operations for MapReduce inspired by MPI collectives, as a set of high level primitives that encapsulate some of the common iterative MapReduce application patterns. Map-Collectives improve the communication and computation performance of the applications by enabling highly optimized group communication across the workers, by getting rid of unnecessary/redundant steps and by enabling the frameworks to use a poly-algorithm approach based on the use case. Map-Collectives also improve the usability of the MapReduce frameworks by providing abstractions that closely resemble the natural application patterns and which reduces the implementation burden of the developers by providing optimized substitutions to certain steps of the MapReduce model. We envision a future where many MapReduce and iterative MapReduce frameworks support a common set of portable Map-Collectives, and we consider this work as a step towards that fulfilling that goal. We defined Map-AllGather and Map-AllReduce Map-Collectives and implemented Multi-Dimensional Scaling and K-means Clustering applications using these operations. We also presented the H-Collectives library for Hadoop, which is a drop-in Map-Collectives library that can be used with existing MapReduce applications with only minimal modifications. We also presented a Map-Collectives implementation for the Twister4Azure iterative MapReduce framework as well. MDS and K-means applications were used to evaluate the performance of Map-Collectives on Hadoop and on Twister4Azure, depicting up to 33% and 50% speedups over the non-collectives implementations by getting rid of the communication and coordination overheads.

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 thesis has shown that it is possible to build efficient, low overhead applications utilizing them. Cloud infrastructure service-based framework prototypes that we developed offered good parallel efficiencies in almost all of the cases we considered. The cost effectiveness of cloud data centers, combined with the comparable performance reported here, suggests that large scale data intensive applications will be increasingly implemented on clouds, and that using MapReduce frameworks will offer convenient user interfaces with little overhead.

8.2 Solutions to the research challenges

In section 1.2, we identified a set of challenges that we faced in performing scalable parallel computing in the Cloud environments. The following section summarizes the solutions that we have proposed to those challenges as part of this thesis.

1. **Programming model**

   We selected the MapReduce programming model extended to support iterative applications as the programming model abstraction to perform large scale computations on cloud environments. The iterative MapReduce model supports pleasingly parallel, MapReduce and iterative MapReduce type applications; this gives us the ability to express a large and a useful subset of large-scale data intensive computations. Iterative MapReduce is simple and easy-to-use by the end user developers who are already familiar with the MapReduce model. As mentioned in section 2.3, the loop variant

& loop invariant data properties and the ability to easily parallelize individual iterations make data intensive iterative computations suitable for efficient execution in cloud environments.

We further extend the iterative MapReduce programming model by introducing the Map-Collectives collective communications primitives (eg: Map-AllGather, Map-AllReduce). As mentioned in section 7.3.2, Map-Collectives improve the usability of the iterative MapReduce model.

2. Data Storage

As presented in sections 5.1.2 and 6.4, we introduced multi-level data caching to overcome the latencies and bandwidth limitations of Cloud Storages so as to improve the performance of data intensive computations in cloud as well as in other environments. The iterative MapReduce architecture and the implementations we have presented in section 5 have the capability to store intermediate data on different cloud storages, based on the size of the data, the access patterns of the data and the performance of different cloud storage options. Also, the frameworks presented in section 5, enable the users to configure the fault tolerance granularity to avoid finer grained check pointing of each task output; this allows them to select a tradeoff between a finer grained fault tolerance vs the performance of the computations.

3. Task Scheduling

Architectures and implementations presented in this thesis use a global queue based dynamic scheduling approach to schedule the tasks, ensuring natural load balancing and dynamic scalability of the system. As presented in section 5.1.3, we introduced a data cache aware task scheduling algorithm to improve the aggregate data bandwidth by eliminating the data transfer overheads.

Further, as mentioned in section 7.3.2.3, the Map-Collectives facilitates communication primitive based task scheduling to remove some of the overheads of the task scheduling.

4. Data Communication

The iterative MapReduce and MapReduce architectures presented in this thesis utilize hybrid data transfers using either a combination of cloud Blob Storage, cloud Tables or direct TCP communication to improve the data communication performance. Also, the frameworks proposed in this thesis enable data reuse across applications which reduce the data transfer requirements.

Map-Collectives identify several commonly used communications patterns of data intensive applications and provide communication and computation abstractions for them. In addition, Map-Collectives improve the communication and computation performance of iterative MapReduce applications by utilizing all-to-all group communications and hierarchical reductions. Map-Collectives also reduce the size of data communication, overlap the communication with computation and enable the possibility of platform specific Map-Collectives implementations suited for the particular cloud environment.

5. Fault tolerance

The MapReduce and iterative MapReduce architectures presented in this thesis support framework managed fault tolerance by ensuring the ability to recover from failure of the parts or tasks of a large scale computation without having to re-run the whole computation. As mentioned in section 5.1.6, the proposed iterative MapReduce architecture supports finer grained task level fault tolerance as well as coarser grained iteration level fault tolerance. Also, we proposed the usage of hybrid data communication mechanisms by utilizing a combination of faster non-persistent mediums and slower persistent mediums, a combination which can enable check-pointing of the computations in the background.

The architectures introduced in this thesis avoid single point of failures through the use of decentralized architectures. This ensures that a single cloud instance failure won't cause a total computation failure. Further, the frameworks handle the stragglers (tail of slow tasks) by scheduling duplicate executions of the slow tasks.

6. Scalability

The iterative MapReduce frameworks introduced in this thesis extend the MapReduce programming model and inherit most of the scalability properties of MapReduce.

The decentralized architectures presented in this thesis facilitate dynamic scalability and avoid single point bottlenecks. They also support hybrid data transfers to overcome cloud service scalability issues by utilizing multiple services that act in parallel to transfer the data. The hybrid scheduling, utilizing a combination of cloud queue storage and cloud table storage, reduces the scheduling overhead even with the increase of the amount of tasks and compute resources of the computations.

7. Efficiency

The iterative MapReduce architecture proposed in this thesis uses multi-level data caching to improve efficiency by reducing the data transfer and staging overheads. Also, it uses direct TCP data transfers to increase data transfer performance, and performs execution history based scheduling to reduce the scheduling overheads. The frameworks proposed in this thesis support multiple waves of map tasks per computation and iteration, which improves the load balancing and the utilizations of the cluster. Frameworks also improve performance and efficiency by overlapping the intermediate data communication with computations, and by performing duplicate executions of straggling tasks.

8. Monitoring, Logging and Metadata storage *

The Twister4Azure iterative MapReduce framework presented in section 5 contains a Web based monitoring console for task and job monitoring, including the monitoring of CPU and memory usages by the cloud instances. It also uses cloud tables for persistent meta-data and log storage.

9. Cost effective *

The frameworks presented in this thesis ensure cost effectiveness by ensuring near optimum utilization of the cloud instances. Also, these frameworks support all the cloud instance types by allowing users to choose the appropriate instances for their use case. The architectures presented in this thesis can also be used with opportunistic environments, such as Amazon EC2 spot instances.

10. Ease of usage *

The frameworks presented in this thesis extend the easy-to-use familiar MapReduce programming model and provide framework-managed fault-tolerance. The Twister4Azure implementation presented in Chapter 5 and the Map-Collectives for Twister4Azure presented in Chapter 7 support local debugging and testing of applications through the Azure local development fabric.

The Map-Collective operations presented in Chapter 7 allow 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.

Note: In this thesis, we do not focus on the research issues involving monitoring, logging and metadata storage*(9), cost effectiveness*(10) and the ease of usage*(11). However, the solutions and frameworks we have developed as part of this thesis research provide and, in some cases, improve the industry standard solutions for each of these issues.

8.3 Future Work

In this thesis, we presented Twister4Azure iterative MapReduce and Map-Collectives as solutions to perform scalable parallel computing in cloud environments. Several areas and directions exist through which we could build on f the foundation established by these frameworks.

One such area is the extension of the Twister4Azure data caching capabilities to a more general distributed caching framework. The first step towards this would be the coordination of the data caches across the different instances; this would allow the programs to share the cached data across the instances. Cache sharing allows a program running in another instance to fetch the data from the caches of other instances, if present, rather than downloading them from the cloud storage. Some use cases for this capability include the re-executions of failed tasks and the duplicate executions of straggling tasks. Further, we can expose a general API to the data caching layer, allowing applications other than Twister4Azure also to utilize the data caching layer. One option is to model this API as a distributed file system.

Another interesting research direction would be to design a domain specific language layer for iterative MapReduce. One option is to extend one of the existing MapReduce language layers such as Hive and Pig to support iterative MapReduce computations. We can also develop workflow layers on top of iterative MapReduce to better compose the applications together; this would allow for a more richer and efficient data sharing (eg: share data stored in cache) process to occur between the applications.

In our work, we presented Map-AllGather and Map-AllReduce implementations as part of the Map-Collectives concept. Another possible Map-Collectives pattern is the Map-ReduceScatter. There are iterative MapReduce applications where only a small subset of loop invariant data product is needed to process the subset of input data in a Map task. In such cases, it is inefficient to make all the loop invariant data available to such computations. In some of these applications, the size of the loop variant data is too large to fit into the memory and it can introduce communication and scalability bottlenecks as well. An example of such a computation is PageRank. The Map-ReduceScatter primitive can be modeled after MPI ReduceScatter to support such use cases in an optimized manner.

Our Map-Collectives work focused mostly on the execution patterns and the programming API's. Another dimension would be to explore the ideal data models for the Map-Collectives model.

Our work mainly focused on the pleasingly parallel, MapReduce and iterative MapReduce type applications. Another pre-dominant type of large-scale applications in the scientific community is the MPI type of applications, which has complex inter-process communication and coordination requirements. Another interesting research direction would be to explore the development of cloud specific programming models to support some of the MPI type application types.

Given the relative novelty of the Big Data movement, cloud infrastructures and the MapReduce frameworks, there exist many more exciting topics to explore for future research work. Some of these examples include the large scale real time stream processing in cloud environments and large scale graph processing in cloud environments.

8.4 List of publications related to this thesis

[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, 23: 2338–2354. doi: 10.1002/cpe.1780.*

[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**, J. Qiu, and D.Gannon, “Towards a Collective Layer in the Big Data Stack”, 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2014). Chicago, USA. May 2014.

[5] **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, Australia. Dec 2011.

[6] **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.

[7] **T. Gunarathne**, T.-L. Wu, J. Qiu, and G. Fox, "Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications," In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10) - ECMLS workshop. ACM, 460-469. DOI=10.1145/1851476.1851544

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

[9] 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.

[10] 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.

[11] **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.

[12] **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.

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.

9. REFERENCES

[1] T. Hey, S. Tansley, and K. Tolle, *The Fourth Paradigm: Data-Intensive Scientific Discovery*, 1 ed.: Microsoft Research, 978-0982544204, 2009.

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

[3] J. Ekanayake, S. Pallickara, and G. Fox, “MapReduce for Data Intensive Scientific Analyses,” in Fourth IEEE International Conference on eScience, 2008, pp. 277-284.

[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] Qiu X., Ekanayake J., Gunarathne T., Bae S.H., Choi J.Y., Beason S. *et al.*, “Using MapReduce Technologies in Bioinformatics and Medical Informatics,” in Using Clouds for Parallel Computations in Systems Biology workshop at SuperComputing (SC09), Portland, Oregon, 2009.

[6] ASF, "Apache Hadoop" [Online], Available: http://hadoop.apache.org/core/. (Retrieved March 5, 2014)

[7] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda *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.

[8] "Windows Azure Compute" [Online], Available: http://www.microsoft.com/windowsazure/features/compute/. (Retrieved July 25th 2011,

[9] *Amazon Web Services*, vol. 2010, Retrieved Mar. 20, 2011, from Amazon: http://aws.amazon.com/.

[10] J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.Bae, J.Qiu *et al.*, “Twister: A Runtime for iterative MapReduce,” in First International Workshop on MapReduce and its Applications of 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010), Chicago, Illinois, 2010.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in ACM SIGOPS Operating Systems Review, 2007, pp. 59-72.

[12] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu, “Scalable parallel computing on clouds using Twister4Azure iterative MapReduce,” *Future Generation Computer Systems,* vol. 29, no. 4, pp. 1035-1048, June 2013, 2013.

[13] ASF, "Hadoop Distributed File System (HDFS)" [Online], Available: wiki.apache.org/hadoop/HDFS. (Retrieved March 5, 2014)

[14] Apache, "Hive" [Online], Available: http://hive.apache.org/. (Retrieved January 6, 2014)

[15] Apache, "Pig " [Online], Available: http://pig.apache.org/. (Retrieved January 6, 2014)

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows *et al.*, “Bigtable: A Distributed Storage System for Structured Data,” *ACM Trans. Comput. Syst.,* vol. 26, no. 2, pp. 1-26, 2008.

[17] Apache, "Hbase" [Online], Available: http://hbase.apache.org/. (Retrieved January 6, 2014)

[18] Apache, "Accumalo" [Online], Available: http://accumulo.apache.org/. (Retrieved January 6, 2014)

[19] Apache, "Mahout" [Online], Available: http://mahout.apache.org/. (Retrieved January 6, 2014)

[20] Apache, "Giraph" [Online], Available: http://giraph.apache.org/. (Retrieved January 6, 2014)

[21] Apache, "Flume" [Online], Available: http://flume.apache.org/. (Retrieved January 6, 2014)

[22] Apache, "Sqoop" [Online], Available: http://sqoop.apache.org/. (Retrieved January 6, 2014)

[23] Amazon, "Amazon Web Services" [Online], Available: http://aws.amazon.com/. (Retrieved March 5, 2014)

[24] Z. Bingjing, R. Yang, W. Tak-Lon, J. Qiu, A. Hughes, and G. Fox, “Applying Twister to Scientific Applications,” in Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, 2010, pp. 25-32.

[25] ASF, "Apache ActiveMQ" [Online], Available: http://activemq.apache.org/. (Retrieved March 5, 2014)

[26] Microsoft, "Microsoft Daytona" [Online], Available: http://research.microsoft.com/en-us/projects/daytona/. (Retrieved March 5, 2014)

[27] J. Lin, and C. Dyer, “Data-Intensive Text Processing with MapReduce,” *Synthesis Lectures on Human Language Technologies,* vol. 3, no. 1, pp. 1-177, 2010/01/01, 2010.

[28] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, “HaLoop: Efficient Iterative Data Processing on Large Clusters,” in The 36th International Conference on Very Large Data Bases, Singapore, 2010.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” in 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10), Boston, 2010.

[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley *et al.*, “Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing,” in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, San Jose, CA, 2012.

[31] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: SQL and rich analytics at scale,” in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, New York, USA, 2013, pp. 13-24.

[32] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, “Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters,” in Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing, Boston, MA, 2012, pp. 10-10.

[33] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, “GraphX: a resilient distributed graph system on Spark,” in First International Workshop on Graph Data Management Experiences and Systems, New York, New York, 2013, pp. 1-6.

[34] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “iMapReduce: A Distributed Computing Framework for Iterative Computation,” in Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, 2011, pp. 1112-1121.

[35] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “PrIter: a distributed framework for prioritized iterative computations,” in Proceedings of the 2nd ACM Symposium on Cloud Computing, Cascais, Portugal, 2011, pp. 1-14.

[36] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser *et al.*, “Pregel: a system for large-scale graph processing,” in Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, Indianapolis, Indiana, USA, 2010, pp. 135-146.

[37] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM,* vol. 33, no. 8, pp. 103-111, 1990.

[38] L. Huan, and D. Orban, “Cloud MapReduce: A MapReduce Implementation on Top of a Cloud Operating System,” in Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on, 2011, pp. 464-474.

[39] Google, "AppEngine MapReduce" [Online], Available: http://code.google.com/p/appengine-mapreduce. (Retrieved March 5, 2014)

[40] 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.

[41] A. Dave, W. Lu, J. Jackson, and R. Barga, “CloudClustering: Toward an iterative data processing pattern on the cloud,” in First International Workshop on Data Intensive Computing in the Clouds, Anchorage, Alaska, 2011.

[42] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. Papadopoulos, K. Bealer *et al.*, “BLAST+: architecture and applications,” *BMC Bioinformatics 2009, 10:421*, 2009.

[43] 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.

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

[45] C. M. Bishop, M. Svensén, and C. K. I. Williams, “GTM: The generative topographic mapping,” *Neural computation,* vol. 10, pp. 215--234, 1998.

[46] S.-H. Bae, J. Y. Choi, J. Qiu, and G. C. Fox, “Dimension reduction and visualization of large high-dimensional data via interpolation,” in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Chicago, Illinois, 2010, pp. 203-214.

[47] Jong Youl Choi, Judy Qiu, Marlon Pierce, and G. Fox, “Generative Topographic Mapping by Deterministic Annealing,” in Proceedings of the 10th International conference on Computational Science and Engineering (ICCS 2010), Amsterdam, The Netherlands, 2010.

[48] NCBI, *BLAST*, Retrieved Sep. 20, 2010, from NIH: http://blast.ncbi.nlm.nih.gov.

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

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

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

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

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

[54] Thilina Gunarathne, Tak-Lon Wu, Judy Qiu, and G. Fox, “Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications,” in Proceedings of the Emerging Computational Methods for the Life Sciences Workshop of 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010), Chicago, Illinois, 2010.

[55] 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,* vol. 23, no. 17, pp. 2338–2354, 2011.

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

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

[58] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta, *Introduction to Parallel Computing*: Addison Wesley (Second Edition), 978-0201648652, 2003.

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

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

[61] MPI-Forum, "MPI: A Message-Passing Interface Standard, Version 3.0" [Online], Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. (Retrieved Sep, 2012)

[62] 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.

[63] J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.Bae, J.Qiu *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.

[64] "JAligner." [Online], Available: http://jaligner.sourceforge.net. (Retrieved December, 2009)

[65] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu, “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.

[66] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, “MapReduce online,” in Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation, San Jose, California, 2010, pp. 21-21.

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

[68] Ekanayake J, Qiu X, Gunarathne T, Beason S, and F. G, "High Performance Parallel Computing with Clouds and Cloud Technologies," Cloud Computing and Software Services, I. M. Ahson S, ed.: CRC Press,ISBN: 1439803153, 2009.

[69] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho *et al.*, “Xen and the art of virtualization,” in Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, 2003, pp. 164-177.

[70] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "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 (VTDC '06), Washington, DC, USA,

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

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

[73] T. Gunarathne, J. Qiu, and D. Gannon, “Towards a Collective Layer in the Big Data Stack,” in 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Chicago, USA, 2014.

[74] USF, "MPI Tutorial : Collective Communication" [Online], Available: http://www.rc.usf.edu/tutorials/classes/tutorial/mpi/chapter8.html. (Retrieved March, 2014)

[75] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, “Performance analysis of MPI collective operations,” in Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, 2005.

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

Curriculum Vitae

Name of Author:	Nanayakkara Kuruppuge Thilina Gunarathne
Date of Birth:	June 26, 1982
Place of Birth:	Kalutara, Sri Lanka
Education:	
May, 2010	**Master of Science, Computer Science** Indiana University, Bloomington, Indiana
Mar, 2004	**Bachelor of Science, Computer Science and Engineering** University of Moratuwa, Sri Lanka
Experience:	
Aug, 2013 – Present	Senior Data Scientist, Customer Analytics, KPMG LLP /Link Analytics LLC, Knoxville, TN, USA
Jun, 2010 - Oct, 2010	Research Intern, IBM Research, Almaden Research Center, San Jose, CA, USA
May, 2008 – Aug, 2008	Research Intern, Microsoft Research, Microsoft Corporation, Redmond, WA, USA
Aug, 2007 – Aug, 2013	Research Assistant, SALSA Lab / Extreme Computing Lab, Indiana University, Bloomington, IN, USA
Jul, 2006 – Jul, 2007	Senior Software Engineer, WSO2 Inc, Colombo Sri Lanka
Honors/Affiliations:	• Persistent Systems Graduate Fellowship (2012-2013) • Graduate Student Scholarship – Indiana University, Bloomington (2007 – 2011) • Committer and Project Management Committee member for Apache Axis2, Apache Web Services and Apache Airavata.(2005 –Present) • Google Summer of Code (2005)