

ARCHITECTURE AND PERFORMANCE
OF RUNTIME ENVIRONMENTS FOR
DATA INTENSIVE SCALABLE
COMPUTING

Jaliya Ekanayake

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
Indiana University
December 2010

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)

Dennis Gannon, Ph.D.

Andrew Lumsdaine, Ph.D.

David Leake, Ph.D.

December 20, 2010

Copyright © 2010
Jaliya Ekanayake
Department of Computer Science
Indiana University
ALL RIGHTS RESERVED

With sincere love, I dedicate this thesis to my parents, my wife Randi, and our daughter Sethmi.

ACKNOWLEDGEMENTS

I have been humbled by the guidance, encouragement, support that I have received from my advisors, colleagues, and my loved ones throughout this work. I owe them a great deal of gratitude and wish to express my sincere thanks.

I owe my deepest gratitude to my advisor, Prof. Geoffrey Fox. It has been his insightful research direction, invaluable guidance, constant encouragement, and generous support that has made everything possible for me to complete this degree. He was there with me for every hurdle and in every dead-end, and he helped me to navigate my path towards the Ph.D. I would like to express my deepest gratitude for all his help.

I would like to thank my entire committee: Dr. Dennis Gannon, Prof. Andrew Lumsdaine, and Prof. David Leake for their help and guidance throughout this research work. I am indebted to them for their valuable feedback and the help they have each given to me from the beginning of my research.

I am grateful to Dr. Sanjiva Weerawarana for encouraging me to undertake this challenging path of life, and also for being an inspiration and a mentor from the days of my undergraduate studies. I would like to express my deepest gratitude for all his help.

I would specially like to thank Dr. Judy Qiu for her guidance and support for my research. She has been a mentor, a colleague, and a friend to me. Together, we worked on several research projects, and her unprecedented support and dedication have served the crucial foundation of support that has made the Twister project a reality.

I am indebted to Dr. Roger Barga and the Cloud Computing Futures team at Microsoft for encouraging me and providing me with time off to work on this thesis while working for Microsoft.

During my stay at Pervasive Technology Institute (PTI), I was fortunate enough to work with many brilliant people. Dr. Shrideep Pallickara mentored me for the research and publications. His impeccable experience in software engineering contributed extensively to help shape the architecture of Twister, and I have benefited immensely from the numerous discussions we have had. I would also like to thank Dr. Marlon Pierce for providing me help and guidance throughout my stay in PTI. Whether it be a problem with user accounts or a research issue, he was always ready and willing to help.

Throughout this research, I was fortunate enough to work with many of my colleagues on research publications, demonstrations, and projects. This journey would not have been the same without their help, critique sessions, and friendship. I would like to express my sincere thanks to my lab mates: Seung-Hee Bea, Jong Choi, Thilina Gunarathne, Li Hui, Bingjing Zhang, Stephen Tak-lon-wu, and my brother Saliya Ekanayake.

None of this work would have been possible without the enormous support and encouragement I received from my loved ones. They deserve much credit for all of my accomplishments. My father, Prof. P.B. Ekanayake has served as an unfailing inspiration to me. What he has achieved amidst enormous hardships has provided me with examples for many situations in my life. The soothing phone calls that my mother, Padmini Ekanayake, have made to me every day since I came to the United States have carried the love and support of my parents to me, and have motivated me to achieve higher goals. Words are not enough to praise my wife Randika in what she has been to me during this endeavor. She is the strength and motivation that has kept me going through one hurdle after another in this five year long journey. Without her support and encouragements, I would not have come this far. I must also mention my daughter Sethmi for refreshing my days with her beautiful smile and sacrificing her valuable "time with Daddy" for

his Ph.D. work. I would also like to thank my brother Saliya and his wife Kalani for their sincere support and encouragements.

I would like to thank Grace Waitman for all her help on proof reading this thesis with her impeccable comments and suggestions on improving the clarity of the write-up.

Finally, I would like to thank the academic and administrative staff of the School of Informatics and Computing, along with the PTI who have taught me or helped me and made the last five years a fun and rewarding experience. Thank you.

ABSTRACT

Architecture and Performance of Runtime Environments for Data Intensive Scalable Computing

By

Jaliya Ekanayake

Doctor of Philosophy in Computer Science

Indiana University, Bloomington

Prof. Geoffrey C. Fox, Chair

In a world of data deluge, considerable computational power is necessary to derive knowledge from the mountains of raw data which surround us. This trend mandates the use of various parallelization techniques and runtimes to perform such analyses in a meaningful period of time. The information retrieval community has introduced a programming model and associated runtime architecture under the name of MapReduce, and it has demonstrated its applicability to several major operations performed by and within this community. Our initial research demonstrated that, although the applicability of MapReduce is limited to applications with fairly simple parallel topologies, with a careful set of extensions, the programming model can be extended to support more classes of parallel applications; in particular, this holds true for the class of Composable Applications.

This thesis presents our experiences in identifying a set of extensions for the MapReduce programming model, which expands its applicability to more classes of applications, including the iterative MapReduce computations; we have also developed an efficient runtime architecture, named Twister, that supports this new programming model. The thesis also includes a detailed discussion about mapping applications and their algorithms to MapReduce and its extensions, as

well as performance analyses of those applications which compare different MapReduce runtimes. The discussions of applications demonstrates the applicability of the Twister runtime for large scale data analyses, while the empirical evaluations prove the scalability and the performance advantages one can gain from using Twister.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	V
ABSTRACT.....	VIII
LIST OF FIGURES	XVI
LIST OF TABLES	XVIII
CHAPTER 1. INTRODUCTION	1
1.1. Introduction	1
1.2. The MapReduce Programming Model.....	4
1.3. Motivation.....	6
1.4. Problem Definition.....	11
1.5. Contributions	12
1.6. Thesis Outline	13
CHAPTER 2. PARALLEL RUNTIMES & PROGRAMMING MODELS.....	15
2.1. Taxonomy of Parallel/Distributed Runtimes	16
2.2. Cloud and Cloud Technologies.....	17
2.3. Existing MapReduce Architectures	17
2.3.1. Handling Input and Output Data.....	17
2.3.2. GFS and HDFS.....	18
2.3.3. Sector	20
2.3.4. DryadLINQ and the Concept of Partitioned Table.....	21
2.3.5. Handling Intermediate Data.....	22

2.3.6.	Scheduling Tasks.....	23
2.3.7.	Fault Tolerance	24
2.4.	Batch Queues	24
2.5.	Cycle Harvesting	26
2.6.	Work Flow	27
2.7.	Parallel Languages	28
2.7.1.	Sawzall.....	29
2.7.2.	DryadLINQ.....	30
2.7.3.	PigLatin	30
2.8.	Message Passing.....	31
2.9.	Threads	33
2.10.	Cloud	34
2.11.	Summary of Features Supported by Different Runtimes.....	36
CHAPTER 3. APPLICATION CLASSES		37
3.1.	Map-only Applications.....	40
3.2.	MapReduce Applications.....	41
3.3.	Iterative MapReduce Applications	42
CHAPTER 4. A PROGRAMMING MODEL FOR ITERATIVE MAPREDUCE COMPUTATIONS		45
4.1.	Static vs. Variable Data.....	46
4.2.	Long Running Map/Reduce Tasks.....	47
4.3.	Granularity of Tasks	48
4.4.	Side-effect-free Programming.....	48
4.5.	Combine Operation.....	49

4.6.	Programming Extensions	50
CHAPTER 5.	TWISTER ARCHITECTURE.....	51
5.1.	Handling Input and Output Data	53
5.2.	Handling Intermediate Data.....	55
5.3.	Use of Pub/Sub Messaging	57
5.4.	Scheduling Tasks.....	57
5.5.	Fault Tolerance	57
5.6.	Twister Implementation.....	60
5.6.1.	Software Requirements	60
5.6.2.	Twister Daemon.....	60
5.6.3.	Twister Driver.....	61
5.6.4.	Pub-sub Brokers	62
5.6.5.	File Manipulation Tool.....	63
5.7.	Twister API	64
CHAPTER 6.	APPLICATIONS AND THEIR PERFORMANCES	66
6.1.	Performance Measures and Calculations.....	67
6.1.1.	Performance and Scalability	67
6.1.2.	Speedup.....	68
6.1.3.	Parallel Overhead.....	68
6.1.4.	Parallel Efficiency.....	69
6.2.	Hardware Software Environments.....	70
6.3.	CAP3 Data Analysis.....	72
6.3.1.	Hadoop Implementation.....	72
6.3.2.	DryadLINQ Implementation.....	73
6.3.3.	Twister Implementation.....	75

6.3.4.	Performance Evaluation.....	75
6.3.5.	Discussion	76
6.4.	High Energy Physics (HEP) Data Analysis	77
6.4.1.	Hadoop Implementation.....	78
6.4.2.	DryadLINQ Implementation.....	79
6.4.3.	Twister Implementation.....	80
6.4.4.	Performance Evaluation.....	81
6.4.5.	Discussion	81
6.5.	Pairwise Similarity Calculation.....	82
6.5.1.	Introduction to Smith-Waterman-Gotoh (SWG).....	82
6.5.2.	Hadoop Implementation.....	84
6.5.3.	DryadLINQ Implementation.....	84
6.5.4.	Twister Implementation.....	85
6.5.5.	Performance Evaluations	85
6.5.6.	Discussion	86
6.6.	K-Means Clustering.....	87
6.6.1.	Hadoop Implementation.....	89
6.6.2.	DryadLINQ Implementation.....	89
6.6.3.	Twister Implementation.....	90
6.6.4.	MPI Implementation.....	90
6.6.5.	Performance Evaluation.....	91
6.6.6.	Discussion	91
6.7.	PageRank.....	92
6.7.1.	Hadoop Implementation.....	93
6.7.2.	Twister Implementation.....	94
6.7.3.	Performance Evaluation.....	94
6.7.4.	Discussion	95

6.8.	Multi-Dimensional Scaling (MDS) Application	96
6.8.1.	Twister Implementation.....	96
6.8.2.	Performance Analysis.....	98
6.8.3.	Discussion	98
6.9.	Matrix Multiplication.....	99
6.9.1.	Row-Column Decomposition Approach	99
6.9.2.	Fox Algorithm for Matrix Multiplication.....	101
6.9.3.	Fox Algorithm using Twister's Extended MapReduce	102
6.9.4.	Performance Evaluation.....	105
6.9.5.	Discussion	109
6.10.	Twister Benchmark Application and Micro Benchmarks	110
6.10.1.	Structure of the Benchmark Application.....	110
6.10.2.	Micro Benchmarks	111
6.11.	Conclusion	114
CHAPTER 7. RELATED WORK.....		116
7.1.	Apache Mahout	116
7.2.	Pregel	120
7.3.	Other Runtimes	123
CHAPTER 8. CONCLUSIONS AND FUTURE WORK.....		125
8.1.	Summary of Work.....	125
8.2.	Conclusions.....	126
8.2.1.	Applicability	128
8.2.2.	Performance and Scalability	129
8.3.	Contributions	131
8.4.	Future Work.....	133

8.5. List of Publications Related to This Thesis	135
REFERENCES...	137
VITA.....	144

LIST OF FIGURES

Figure 1. Data Flow in MapReduce programming model.....	5
Figure 2. Composable applications in a workflow.	39
Figure 3. The iterative MapReduce programming model supported by Twister.	46
Figure 4. Architecture of Twister MapReduce runtime.	52
Figure 5. Two approaches used by the Twister to transfer intermediate data.	56
Figure 6. Components of Twister Daemon.....	60
Figure 7. Processing data as files using DryadLINQ.....	74
Figure 8. Speedups of different implementations of CAP3 application measured using 256 CPU cores of Cluster-III (Hadoop and Twister) and Cluster-IV (DryadLINQ).....	75
Figure 9. Scalability of different implementations of CAP3 application measured using 256 CPU cores of Cluster-III (Hadoop and Twister) and Cluster-IV (DryadLINQ).....	76
Figure 10. MapReduce for the HEP data analysis.....	78
Figure 11. Simulating MapReduce using DryadLINQ.....	79
Figure 12. Performance of different implementations of HEP data analysis applications measured using 256 CPU cores of Cluster-III (Hadoop and Twister) and Cluster-IV (DryadLINQ).....	81
Figure 13. MapReduce algorithm for SW-G distance calculation program	84
Figure 14. Parallel Efficiency of the different parallel runtimes for the SW-G program (Using 744 CPU cores in Cluster-I).	86
Figure 15. Performance of different implementations of K-Means clustering algorithm performed using 256 CPU cores of Cluster-III (Hadoop, Twister, and MPI) and Cluster-IV (DryadLINQ)	91
Figure 16. Elapsed time for 16 iterations of the Hadoop and Twister PageRank implementations (Using 256 CPU cores in Cluster-II).	95

Figure 17. Efficiency of the MDS application (in Cluster-II).....	98
Figure 18. Matrix multiplication Row-Column decomposition (top). Twister MapReduce implementation (bottom).	100
Figure 19. 2D block decomposition in Fox algorithm and the process mesh.....	101
Figure 20. Virtual topology of map and reduce tasks arranged in a square matrix of size qxq	103
Figure 21. Communication pattern of the second iteration of the Fox - MapReduce algorithm shown using 3x3 processes mesh. Thick arrows between map and reduce tasks show the broadcast operation while dash arrows show the shift operation.....	104
Figure 22. Sequential time for performing one block of matrix multiplication - Java vs. C++	106
Figure 23. Performance of Row-Column and Fox algorithm based implementations by using 64 CPU cores of Cluster II. (Note: In this evaluation, we have not used Twister's TCP based direct data transfers as explained in section 5.2).....	107
Figure 24. Performance of Twister and MPI matrix multiplication (Fox Algorithm) implementations by using 256 CPU cores of Cluster II. The figure also shows the projected compute-only time for both Java and C++.	108
Figure 25. Parallel overhead of three matrix multiplication implementations by using 256 CPU cores of Cluster II.....	109
Figure 26. The structure of the micro benchmark program.....	111
Figure 27. Time to send a single data item from the main program to all <i>map</i> tasks against the size of the data item.	112
Figure 28. Time to scatter a set of data items to <i>map/reduce</i> tasks against scatter message size. ...	112
Figure 29. Total time for the <i>map</i> to <i>reduce</i> data transfer against $\langle Key, Value \rangle$ message size.	113

LIST OF TABLES

Table 1. Comparison of features supported by different parallel programming runtimes	36
Table 2. Application classification.....	38
Table 3. Classes of MapReduce applications	44
Table 4. Commands supported by the Twister’s file manipulation tool.	63
Table 5. The Application program interface of Twister.	64
Table 6. Details of the computation clusters used.....	71
Table 7. Characteristics of data sets (B = Billions, AM = Adjacency Matrix)	94
Table 8. Breakdown of the amount of communication in various stages of the Row-column based matrix multiplication algorithm.	100
Table 9. Breakdown of the amount of communication in various stages of the Fox algorithm.	102
Table 10. Breakdown of the amount of communication in various stages of the Twister MapReduce version of Fox algorithm.	105

1.1. Introduction

With the advancements that have been made in relation to scientific instruments and various sensor networks, the spread of the World Wide Web, and the widespread use of digital media a data deluge has been created in many domains. In some domains such as astronomy, particle physics, and information retrieval, the volumes of data are already in the peta-scale. For example, High Energy Physics (HEP) experiments such as CMS and Atlas in the Large Hadron Collider (LHC) are expected to produce tens of Petabytes of data annually even after trimming the data via multiple layers of filtrations. In astronomy, the Large Synoptic Survey Telescope produces data at a nightly rate of about 20 Terabytes. Although not in the same range as particle physics or

astronomy, instruments in biology, especially those related to genes and genomes, produce millions of gene data ensuring biology a place among data intensive domains.

The increase in the volume of data also increases the amount of computing power necessary to transform the raw data into meaningful information. Although the relationship between the size of the data and the amount of computation can vary drastically depending on the type of the analysis performed, most data analysis functions with asymptotic time complexities beyond the simplest $O(n)$ can require considerable processing power. In many such situations, the required processing power far exceeds the processing capabilities of individual computers, and this reality mandates the use of efficient parallel and distributed computing strategies to meet the scalability and performance requirements inherent in such data analyses.

A careful analysis on applications performed on these large data sets revealed that most such applications are composed of pipelines of data/compute intensive stages or filters. For some applications such as converting a large collection of documents to another format, a single filter stage is sufficient, whereas an application such as pagerank [1] require iterative application of the pagerank computation stage until a convergence in results is obtained. Most of these applications can be parallelized by applying a high level parallel construct such as MapReduce[2] to the above mentioned stages. As the volume of data increases or the amount of computation performed at each such stage increases, the overhead in applying a higher level parallel constructs to these individual stages diminishes making the overall computation parallelizable using higher level parallel constructs. We named such applications “composable” and make the prime focus in this thesis.

The advancements made in computing and communication technology of the last decade also favor parallel and distributed processing. Multi-core and many-core computer chips are becoming the norm after the classical mechanism of increasing the performance of computer

processors by increasing the clock frequency has met its peak governed by the quantum physics. These powerful multi-core chips allow many programs otherwise executed in sequential fashion to exploit the benefits of parallelism and pack thousands of CPU cores into computation clusters and millions of cores into data centers[3]. Similarly, the advancements in communication technology have reduced the latencies involved in data transfers, which also favor distributed processing.

To support data intensive scalable computing, the information retrieval industry has introduced several new distributed runtimes and associated programming models to the spectrum of parallel and distributed processing runtimes. MapReduce[2] and Dryad[4] are two such prominent technologies. As many data analyses become more and more data intensive, the ratio of CPU instructions to I/O instruction becomes reduced. According to [5], in many of these applications, the CPU: I/O ratios are well below 10000:1. The above runtimes have adopted a more data centered approach: they support moving computation to data favoring local data reads, simple programming models, and various quality of services. Initial results from the information retrieval industry show that they can be deployed in large computation infrastructures built using commodity hardware and that they provide high-throughput computing capabilities amidst various types of failures in computation units.

Although the above technologies have shown promising results in information retrieval, their applicability to a wide variety of parallel computing has not been studied well. This thesis focuses on MapReduce technologies and its related programming model. Here, we try to understand the following foci: the applicability of the MapReduce programming model to different classes of parallel applications, especially for the composable applications; how the existing MapReduce runtimes support these applications; and how the programming model could be extended to design an efficient MapReduce runtime to support more classes of parallel

applications by incorporating knowledge and experience from classical approaches to parallel processing such as MPI[6, 7]. The thesis also presents a detailed performance analysis of different MapReduce runtimes in terms of performance, scalability and quality of services.

1.2. The MapReduce Programming Model

MapReduce is a distributed programming technique proposed by Google for large-scale data processing in distributed computing environments. Jeffrey Dean and Sanjay Ghemawat describe the MapReduce programming model as follows:

- The computation takes a set of input (key,value) pairs, and produces a set of output (key,value) pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce.
- Map, written by the user, takes an input pair and produces a set of intermediate (key,value) pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.
- The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically, just zero or one output value is produced per Reduce invocation[2].

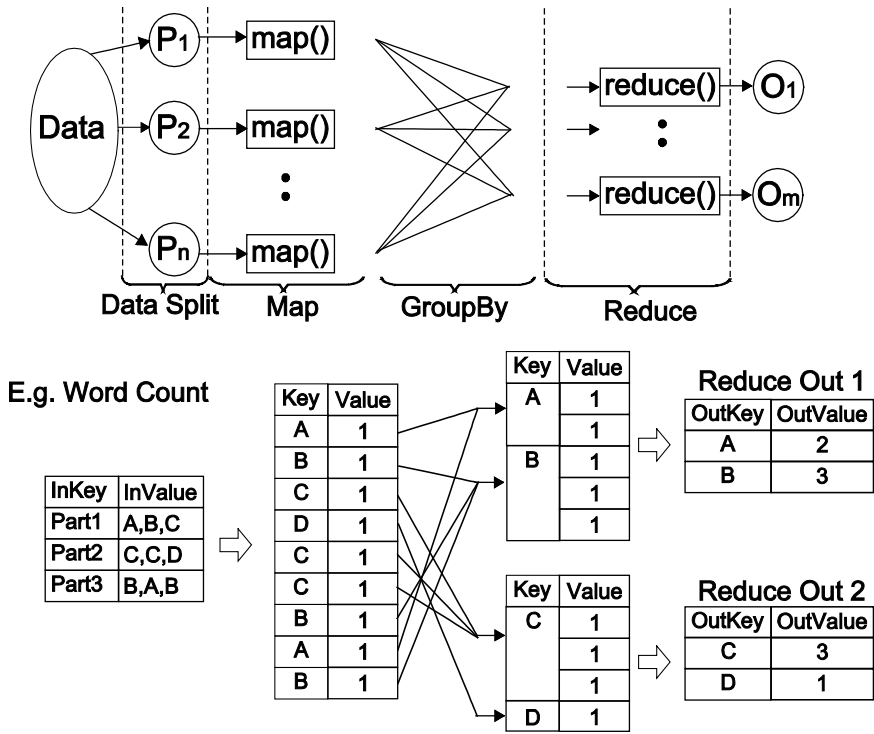


Figure 1. Data Flow in MapReduce programming model

Furthermore, because of its functional programming inheritance, MapReduce requires both map and reduce tasks to be “side-effect-free”. Typically, the map tasks start with a data partition and the reduce task performs such operations as “aggregation” or “summation”. To support these, MapReduce also requires that the operations being performed by the reduce task to be both “associative” and “commutative”. These are common requirements for general reductions. For example, in MPI the default operations or user defined operations in MPI_Reduce or MPI_Allreduce are also required to be associative and may be commutative.

Counting word occurrences within a large document collection is a typical example used to illustrate the MapReduce technique. The data set is split into smaller segments and the map function is executed on each of these data segments. The map function produces a (key, value) pair for every word it encounters. Here, the “word” is the key and the value is 1. The framework groups together all the pairs, which have the same key (“word”), and invokes the reduce function

by passing the list of values for a given key. The reduce function adds up all the values and produces a count for a particular key, which in this case is the number of occurrences of a particular word in the document set. Figure 1 shows the data flow and different phases of the MapReduce programming model.

1.3. Motivation

The increasing amount of data and the vast computing power they require to be transformed into knowledge have created a diverse set of problems in many domains. As a response to this growing need the information retrieval community has come up with new programming models and runtimes such as MapReduce and Dryad, that adopt more data centered approaches to parallel processing and provide simpler programming models. On the other hand, the parallel runtimes such as MPI and PVM used by the High Performance Computing (HPC) communities have accumulated years of experience of HPC communities into their programming models; thus their efficient runtimes are applicable to more classes of applications. This thesis is motivated by the following hypothesis:

With the diversity of the parallel applications and the need to process large volumes of data, we argue that, by extending simpler, data centered programming model of MapReduce using the proven architectures and programming models in HPC world, we will expand its usability to more classes of parallel applications.

The MapReduce programming model has attracted a great deal of enthusiasm because of its simplicity, as well as the improved quality of services it can provide. In classic job execution infrastructures, the scheduling decisions are influenced mainly by the availability of computer resources. Further, in many classic parallel runtimes, the movement of data to the individual parallel processes is typically handled via shared file systems or a master process sending data to slave processes. As many data analyses become more and more data intensive, the ratio of CPU

instructions to I/O instruction reduces. Amdahl's IO law states that the programs complete one I/O per 50,000 instructions[8], and Jim Gray and Prashant Shenoy claim that, in many scientific applications, this rate drops well below one I/O per 10000 instructions[9]. As we build infrastructures to handle the data deluge, the above observations suggest that using new programming models such as MapReduce based on the concept of "moving computation to data" could be more efficient. Although the MapReduce was originated from the information retrieval industry, our initial evaluations show it can be applied to many Single Program Multiple Data (SPMD)[10] style problems in various scientific domains as well.

Classic parallel applications that were developed by using message passing runtimes such as MPI[11] and PVM[12] utilize a rich set of communication and synchronization constructs offered by those runtimes to create diverse communication topologies within the parallel applications. Further, the parallel algorithms targeted for these runtimes assume the availability of a diverse set of communication constructs in them as well. For example, a matrix multiplication application which implements the Fox algorithm[13] in MPI utilizes the processes mesh construct available in MPI. By contrast, MapReduce and similar high-level programming models support simple communication topologies and synchronization constructs. Although this limits how they can be applied to the diverse classes of parallel algorithms, our initial analyses revealed that many data/compute intensive applications can be implemented by using these high level programming models as well. When the volume of the data is large, algorithms based on simple communication topologies may produce performances comparable to the algorithms that utilize complex communication topologies which have fine grain synchronization constructs. These observations also favor MapReduce, since its relaxed synchronization constraints do not impose much of an overhead for large data analysis tasks. Furthermore, the simplicity and robustness of these programming models supersede the additional overheads.

The emerging trend of performing data analysis on Cloud infrastructures also favors the MapReduce style of simple programming models. Cloud infrastructures are comprised of thousands of computers organized into data centers, that provide both data storage and computation services to the users. Most of these infrastructures are built using commodity hardware, and hence, the typical network bandwidth available between computation units is well below the typical network bandwidth available in high performance computation clusters. Apart from the above conditions, most cloud infrastructures utilize virtual machine technologies to maximize their resource utilization, and also, to isolate the user level applications (including the operating system) from the bare-metal systems. These characteristics introduce latencies into the communication medium, and are significantly higher than those of computation clusters. Programming models such as MapReduce rely on relaxed synchronization constraints; thus they operate with higher task granularities that have less susceptibility to latencies. These features make them an ideal match to operating on Clouds.

Various data analysis applications show different data and compute intensity characteristics. Further, they also exhibit diverse communication topologies. MapReduce is typically applied to large scale data parallel applications with simple communication topologies. However, as more and more applications have become data intensive, we have noticed that the MapReduce programming model can be used as a parallelization construct in many other types of applications as well. Applications with simple iterative computations represent an important class that expands across domains such as data clustering, machine learning, and computer vision. In many of these algorithms, the computations performed inside the iterations can be represented as one or more MapReduce computations to exploit parallelism. This idea is also shown by Cheng Tao et al., in their paper demonstrating how MapReduce can be applied to iterative machine learning algorithms in multi-core computers[14]. However, we noticed that, to

support such algorithms and expand the usability envelope of MapReduce effectively, we need several extensions to its programming model as well as an efficient implementation.

There are some existing implementations of MapReduce such as Hadoop[15] and Sphere[16], most of which adopt the initial programming model and the architecture presented by Google. These architectures focus on providing maximum throughput for single step MapReduce computations (computations that involve only one application of MapReduce) with better fault tolerance. To support the above goals, they incorporate various measures which can be justified for some of the large scale data analysis but which introduce considerable performance overheads for many other applications for which the MapReduce programming model can prove applicable. For example, in Hadoop, the intermediate data produced at the *map* tasks are first stored in the local disks of the compute node where the *map* task is executed. Later, *reduce* tasks download this output of the map tasks to the local disks where the *reduce* tasks are being executed. This approach greatly simplifies the fault handling mechanism of Hadoop, as the output of each parallel task exists in some form of file system throughout the computation. Hadoop also utilizes a dynamic task scheduling mechanism for its *map/reduce* tasks to improve the overall utilization of the compute resources. Although this approach allowed Hadoop to support features such as “dynamic flexibility” – which is a feature that allows the runtime in using a dynamic set of compute resources, this option can also induce higher overheads, for applications that execute tasks repetitively. Furthermore, in these runtimes, the repetitive execution of MapReduce results in new *map/reduce* tasks for each iteration which loads or accesses any static data repetitively. Although these features can be justified for single step MapReduce computations, they introduce considerable performance overheads for many iterative applications.

In contrast to the above characteristics of MapReduce, we noticed that the parallel processes in High Performance Computing (HPC) applications based on MPI, have long lifespans in the applications; most of the time, they utilize high performance communication infrastructures to communicate fine grain messages with higher efficiencies for computation dominated workloads. The use of long-running processes enables the MPI processes to store any static or dynamic data (state information) necessary for the computation throughout the application life cycle. On the other hand the use of stateful processes makes it harder to support fault tolerance. The use of low latency communication infrastructures makes the inter-process communications highly efficient, as compared to the “disk->wire->disk” approach to data transfer adopted by the MapReduce runtimes such as Hadoop. Furthermore, by supporting low-level communication constructs, MPI and similar parallel runtimes allow the user to create parallel algorithms with a wide variety of communication topologies. K. Asanovic et al. presents seven dwarfs capturing various computation and communication patterns of parallel applications into equivalence classes [17].

Although there are analyses like the one described above for the HPC applications, we could not find a detailed analysis which compares high level abstractions such as MapReduce and Dryad and the mapping of parallel applications to these abstractions. This knowledge would be crucial, and would be required in order to map parallel applications to the new programming runtimes so as to obtain the best possible performance and scalability for the applications. This is one of the areas that we will explore in this research.

The above observations support our hypothesis of extending the MapReduce programming model by incorporating some of the features that are present in the HPC runtimes; in doing so we can provide an efficient implementation so that MapReduce can be used with more classes of parallel applications.

1.4. Problem Definition

In a world deluged by data, high-level programming models and associated runtimes that adopt data centered approaches have shown promising results for data intensive computing. MapReduce is one such key technology that has been introduced to tackle large scale data analyses from the information retrieval community. However, we noticed the following gap in the current research concerning these high-level programming models and their architectures; this research strives to minimize this gap.

- First, the applicability of MapReduce to the diverse field of parallel computing is not well studied. We try to fill this gap by applying MapReduce to a selected set of applications to represent various classes of applications that MapReduce could be applied to; we intend to demonstrate the mapping of parallel algorithms to the MapReduce domain while comparing and contrasting the characteristics of few existing high level programming models.
- Second, from our preliminary research, we have identified that, although the MapReduce programming model is a simple yet powerful programming construct that can be used in many parallel algorithms, current MapReduce runtimes are inefficient for many applications that require repetitive application of MapReduce or applications that require low latency communication. On the other hand, some features of the HPC runtimes make them highly desirable for some of these applications, even though the programming models of the HPC runtimes are not simple nor are they an ideal match for the data deluge. By incorporating the lessons learnt from the HPC community to MapReduce, we wish to address the challenge of devising a new programming model and supporting it on an efficient runtime.

- Finally, a detailed performance analysis of high level programming runtimes, which identify their strengths and weaknesses for different classes of applications while comparing them with solutions from the HPC world, has not yet been performed. Furthermore, identifying the behaviors of these runtimes in virtualized environments such as Clouds certainly exist as a goal worthy of further study and research.

1.5. Contributions

We envision the following contributions could emerge from this research:

- The architecture and the programming model of an efficient and scalable MapReduce runtime which could expand applicability of the MapReduce programming model to more classes of data intensive computations, especially for iterative MapReduce computations.
- A prototype implementation of the proposed architecture and the programming model that minimizes the overheads suffered by typical MapReduce runtimes.
- A classification of the problems that can be handled by MapReduce and algorithms for mapping these to the MapReduce model while minimizing overheads, followed by a detailed discussion of several scientific applications that could be developed using different runtimes including the proposed runtime.
- A detailed performance analysis comprised of application level performance characteristics to micro benchmarks, which can evaluate the performance, scalability, and overhead of the proposed runtime against other relevant runtimes.

1.6. Thesis Outline

This thesis is organized as follows:-

We present the current state of the parallel programming models and runtimes related to this thesis in Chapter 2. Here, we focus on several aspects of parallel computing such as scalability, throughput vs. efficiency, level of abstraction, fault tolerance, and support for handling data. The chapter also introduces some of the technologies that we used for our performance comparisons with the proposed runtime.

After classifying the MapReduce applications to several distinct classes in Chapter 3, we introduce the extended MapReduce programming model that we propose in this thesis in Chapter 4, along with an explanation of how it can be used to support different classes of applications. Furthermore, in this chapter, we compare the proposed programming model with some of the other parallel programming techniques that can be used to implement such applications, and we try to conclude with a set of equivalence classes of applications.

This is followed by a detailed explanation of the architecture and the implementation of the proposed runtime “Twister” in Chapter 5. Here we discuss the various architectural designs we used in the Twister runtime and compare and contrast it with other MapReduce and HPC runtimes; we also discuss the feasibility of our proposed architecture.

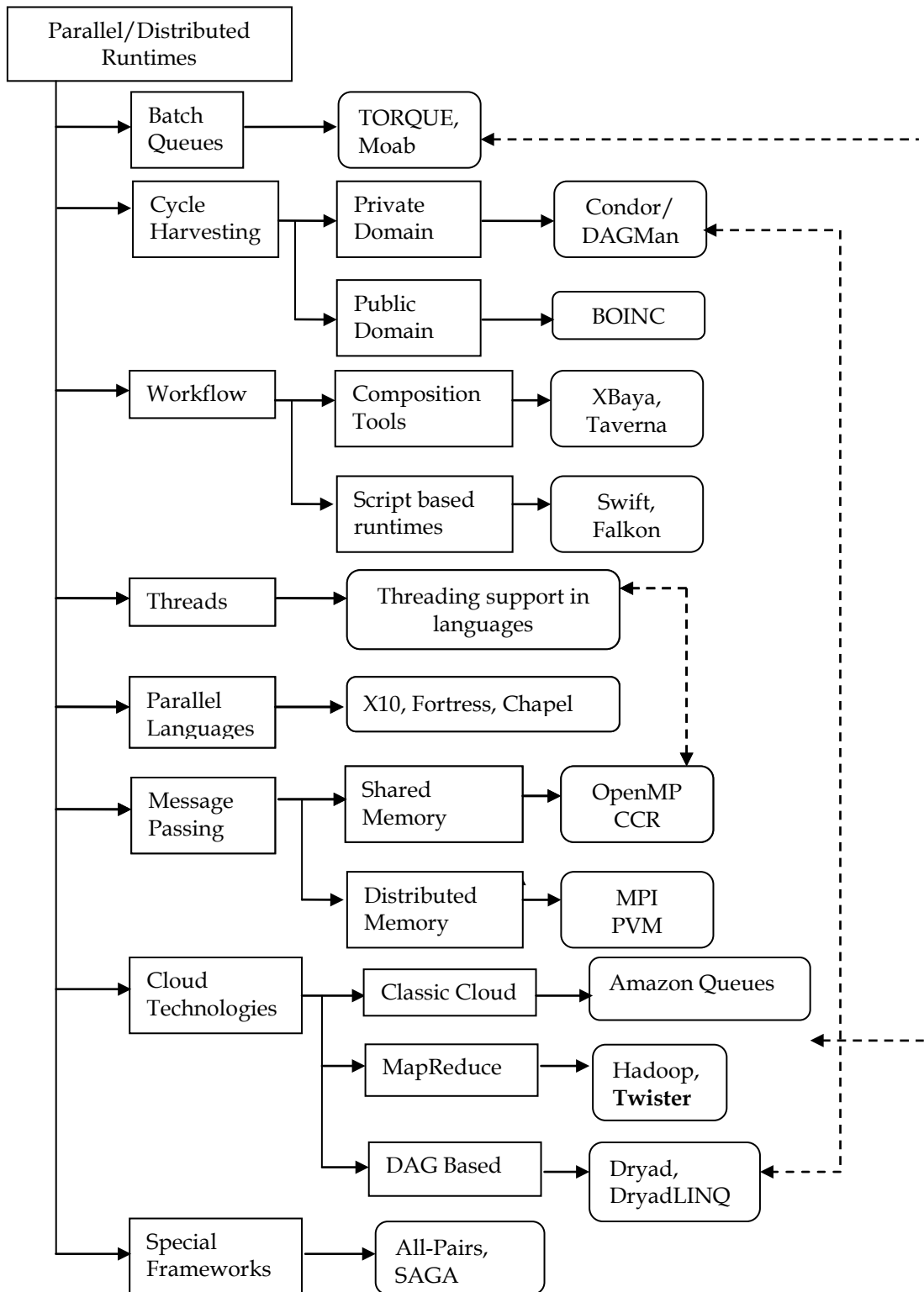
In Chapter 6, we present the data analysis applications we have implemented using various parallel runtimes in order to understand their benefits and compare them with the proposed architecture. Under each application we describe the type of benchmarks we have conducted the results obtained followed by a discussion on results. This chapter serves as the proof of our hypothesis presented in the Section 1.3 of this thesis.

Finally we present our conclusion and outline the direction of future work in Chapter 8 after following a related work to this thesis in Chapter 7.

Chapter 2. **Parallel Runtimes & Programming Models**

The discussions found in this thesis extend into many areas in parallel and distributed processing runtimes and their programming models. Moreover, our work is built on top of the results of much previous research. In this section, we discuss the state of the art in the above areas. First, we will present a taxonomy of parallel and distributed processing runtimes relevant to our research, by showing the cosmic view of the technologies and the position of the proposed runtime in this list of technologies. As our work is mainly centered on MapReduce technologies, we will next discuss the exiting MapReduce technologies and compare the other relevant technologies with MapReduce using six dimensions: (i) data handling, (ii) communication, (iii) synchronization, (iv) task granularities, (v) scalability, and (vi) Quality of Service (QoS). We also extend the discussions to the cloud computing paradigm and its related technologies as well.

2.1. Taxonomy of Parallel/Distributed Runtimes



2.2. Cloud and Cloud Technologies

Cloud and Cloud Technologies are two distinct terms that we use throughout this thesis; hence, the term deserves a clear definition. By “Cloud,” we refer to a collection of infrastructure services such as the Infrastructure-as-a-service (IaaS) and the Platform-as-a-Service (PaaS), etc., provided by various organizations where virtualization plays a key role. By “Cloud Technologies,” we refer to various technologies associated with clouds such as storage services like the S3[18], communication queues like the Simple Queue Service (SQS) in Amazon and the Azure Queues in Windows Azure[19]; most importantly, we also focus on the high level runtimes such as Hadoop[15] and Dryad [20].

2.3. Existing MapReduce Architectures

Along with the MapReduce programming model, Jeffrey Dean and Sanjay Ghemawat describe in their paper the architecture that they adopted at Google. Most of their decisions are based on the scale of the problems that they solved using MapReduce and the characteristics of the large computing infrastructure in which these applications were deployed. Apache Hadoop and several other MapReduce runtimes such as Disco [21] and Sector/Sphere also adopted most of these architectural decisions. Below, we will list some of the most important characteristics of these runtime, as it will be useful to explain and compare them with the architectural decisions we made in this thesis later.

2.3.1. Handling Input and Output Data

The key motivation behind the MapReduce programming model is to support large scale computations that show “pleasingly parallel” characteristics in terms of data. As applications become more and more data intensive, their performances are greatly determined by the bandwidth of the medium used to access data. In this respect, moving data to the available

computing resources before processing as done by many classic distributed and parallel computing infrastructures is not feasible. To eliminate this costly (in terms of performance) data movement, MapReduce architectures have introduced the concept of the “data-compute node”, which represents a computer that is used as both a data storage device and a computation unit. (Note: Please note that following this point, we will simply use the term “node” to refer to a “data-compute node” when we discuss MapReduce related technologies).

The above approach allows MapReduce runtimes to utilize larger disk bandwidth produced by the local disks of the nodes. However, to manage data in these local disks and to obtain meta-data to move computation to data, a higher level data management infrastructure is necessary. To support these features, most MapReduce runtimes use distributed storage infrastructures built using local disks to read input data and store final output data. Both Google and Hadoop utilize distributed fault-tolerance file systems – GFS[22] and HDFS[15] in their MapReduce runtimes. Sphere MapReduce runtime utilizes a distributed file system named Sector that uses slightly different architecture than GFS or HDFS. Microsoft DryadLINQ[23] on the other hand, uses a simple meta-data construct named a “partitioned file” to process data from local disks of the compute nodes that are organized as Windows shared directories. With these features, most MapReduce runtimes use distributed storage infrastructures to read input data and store final output data. In the remainder of this section, we will discuss some of the above storage architectures.

2.3.2. GFS and HDFS

The Google File System (GFS) has been developed to provide a distributed fault tolerance file system built by using a large number of commodity machines. Many design decisions of the GFS have been influenced by the type of operations they performed on large data sets as well as the typical applications they use. For example, they noticed that most common file access patterns for

large data sets is either the initial file creation, file read, or file appends. Random updates or writes on large data files are rare. We also noticed similar characteristics on large scientific data products. Typically, most of these data sets are read from different applications (algorithms) for inferences and rarely modified. In both Google and Hadoop MapReduce runtimes, the distributed file system is used to read input data and store output data. This further simplifies the type of operations performed on the file system and makes it almost similar to the “write-once-read-many” access model.

The GFS architecture has two main components: (i) the GFS master and (ii) the GFS chunk server. The Hadoop Distributed File System (HDFS) is much closer to the GFS in design and in the HDFS; these entities are called (i) the Name Node and (ii) the Data Node respectively. The GFS master keeps track of all the meta-data including the file system namespace, while the GFS chunk servers store data chunks assigned to them by the GFS master. Both the GFS and the HDFS store data as fixed size chunks or blocks within the distributed file system, and they use replications to recover from failures. Data which is read and is written directly to and from clients goes to the chunk servers (data nodes), which are located using the meta-data served by the master. Both file systems provide an interface with common file system operations to clients although they do not implement a standard file system API such as POSIX.

The use of fixed sized blocks simplifies the design of the GFS and the HDFS since the blocks which belong to a file can be calculated using the record ranges. Furthermore, these file systems use fairly large blocks, typically megabytes in size, as compared with the classic distributed file systems. This feature reduces the number of blocks at the chunk servers and blocks related meta-data that need to be stored at the master. Also, reading large files is simplified by reading blocks. However, we noticed that the matching block boundaries and data parallelism for various data types is not straightforward. For example, most scientific data is typically stored as files and the

boundaries for parallel processing typically exist at the file level. If the files are fairly equal in size, one can group several files into a block (if the files are comparatively smaller than the typical best performance block sizes), or they can select an appropriately sized block to match the size of the files. Still, the blocks may need to be padded to match the differences in file sizes and the block boundaries. If the files are not similar in size, the padding will increase. Breaking a file into multiple blocks is an option, when the data in files are represented as records and the data parallelism exist at record level. For example, in text data (web pages, text documents etc..) a record can be a sentence, a line of text or a paragraph, and many operations performed on text data collections can be parallelized at this records level.

2.3.3. Sector

Sector is introduced as a storage cloud[16]. Similar to the GFS and the HDFS architectures, Sector also uses a master to hold meta-data while a set of worker nodes store files. The authors claimed that it can be deployed across wide area networks with high speed network connections and can support better upload and download capabilities. The main distinction between Sector and the GFS and the HDFS is that it does not store large data sets into chunks or blocks, and instead, it expects the user to handle data partitioning. Sector stores these files (data partitions) as is, in the local disks of the storage nodes and it supports replications. The main advantage of this approach is that, a computation infrastructure built on top of Sector (Sphere is such a computation infrastructure), can access files directly as native files instead of accessing them via an API provided by Sector. This is highly beneficial when legacy applications need to be used as data processing functions (as executables) in the MapReduce style processing runtimes. For example, a gene assembly program named CAP3 [24] that we will discuss later expects input data to be passed as files using command line arguments. To execute such an application using Hadoop, the Hadoop application first needs to copy the data from HDFS to the local machine's file system and

invoke CAP3 executable passing input file names in the command line. The data copying from the HDFS to the local file system is required, since the HDFS does not provide a standard file system interface such as POSIX. In contrast, the same application with Sector can directly execute the CAP3 program passing input files as command line arguments, since they exist as files (not as blocks) in the local file system. However, this approach leaves the task of partitioning data to the user, which we think could be supported by providing a tool to perform data partitioning using custom partitioning schemes.

2.3.4. DryadLINQ and the Concept of Partitioned Table

The academic release of Microsoft DryadLINQ [25] uses Windows shared directories to read input data and to store output data. Instead of providing a file system to keep track of the data partitions and their replications, DryadLINQ expects the user to provide a special file named the "Partitioned File" to the runtime, which contains the meta-data regarding the data partitions and their locations among the collection of local hard disks of the computation nodes. With this information, DryadLINQ tries to schedule data processing tasks on the nodes on which the data is available. It also supports replicas of data partitions so that, in the case of failure of a given node, the tasks could be rescheduled to run on a different node. DryadLINQ provides programming constructs to partition data based on the "hash" and "range" values. However, it does not provide a tool for the user to distribute data partitions across computation nodes, or to collect the results to a shared location, which we think, is very important for these types of distributed runtimes.

Overall, we also think that a distributed file system that keeps track of data partitions and their replications is an ideal candidate for distributed runtimes based on MapReduce. The GFS, the HDFS, and the Sector file systems handle fault tolerance by supporting data replications, and they actively maintain a given number of replications of data blocks amidst node failures, which

makes the overall distributed runtime more robust. However, with the capability of using directly accessible files as data partitions, the approach adopted by Sector is more flexible.

2.3.5. Handling Intermediate Data

In most MapReduce runtimes, the intermediate data produced after the map stage of the computation is handled using the following steps.

1. The *map* outputs are first buffered in memory and continuously pushed to a file(s) in the local disk of the nodes the map tasks are executed. The meta-data regarding these outputs are sent to the master process.
2. The master process assigns *map* outputs to appropriate reduce tasks based on some form of a “key selector” and it notifies the reducers, which then retrieve data via some communication protocol such as HTTP and store them in the local disks where they are being executed.
3. Once all the map outputs are received for a particular reduce task, the runtime performs a sorting operation on the reduce inputs (map outputs) based on the “key” and invoke the reduce function.

This scheme of handling intermediate data is both scalable and robust. Since the intermediate data is handled in files, the volume of intermediate data is limited only by the amount of local disk space available in all compute nodes. It is robust because it makes the fault tolerance functionality of the runtime simpler and straightforward. For example, if the *map* tasks do not store their outputs in the local disks first, a failed reduce task will require a re-execution of all the *map* tasks to get its portion of *reduce* inputs. With the above scheme, a failed *reduce* task only needs to collect data from the nodes where the *map* tasks stored their outputs.

Although the above approach is robust and scalable, it adds a considerable latency to the data transfer between the *map* and the *reduce* tasks, especially with workloads with equal work load

distributions. Since the data transfer can start immediately after a map task is completed, the effect of latency is less significant for MapReduce computation where the work load distribution at the map tasks is not uniform. In these type of computations, the data transfer continues along with the *map* stage, and, at the end of the *map* stage of the computation, the reduce tasks need to wait till the data is retrieved from the slow *map* tasks. However, for workloads with equal load distribution, all the data transfers start in a close time interval and the reduce tasks need to wait till all the data is transferred via disk->wire->disk transfer approach.

Classic parallel runtimes such as MPI uses in-memory communication mechanisms to transfer data between parallel processes, and hence, they operate with minimum latencies. On the other hand, the performance gain results in highly complex fault tolerance mechanisms in MPI. In our design, we try to incorporate the in-memory data communication approach with MapReduce.

2.3.6. Scheduling Tasks

Google's MapReduce and Hadoop use a dynamic scheduling mechanism. In this approach, the master assigns map/reduce tasks to the available computation resources at the runtime. DryadLINQ, on the other hand, uses a static scheduling approach in which the parallel tasks in a particular stage of the DAG (Note: DryadLINQ uses a DAG as the execution flow) are assigned to nodes at the beginning of the computation. Both approaches have their own pros and cons. With the dynamic scheduler in Hadoop, it can utilize compute resources when they become available, which yield a higher utilization. It also makes the re-scheduling of tasks, in the case of a failure, more straightforward for the master process. Furthermore, when the workload is skewed and there are more tasks than there are available computation resources (CPU cores or threads), this approach can effectively taper out the skewness of the task distribution. In contrast, a static scheduling with the capability of re-scheduling in the event of failures will produce minimum scheduling overhead. To load balance a skewed work load with this approach, one can use

randomization in task assignment so that tasks with different skewness are assigned to a given processing element.

2.3.7. Fault Tolerance

Failures are common in distributed runtimes that operates on thousands of computers, especially when the computation infrastructure is built using commodity hardware equipments. Although this is different from the experience we have in using high end computation clusters with better networking equipments, and also in using leased resources (virtual machines) from Cloud providers, we also identify the need for producing distributed runtimes with fault tolerance capabilities.

Handling failures is one of the key considerations of Google's MapReduce architecture, and similarly, this is also the case in Hadoop as well. In both Google and Hadoop MapReduce, the distributed file systems handle the failures of the disks or nodes using data replication. Therefore, applications can process input data amidst node failures, provided that the number of replicas of data and the replica placement can effectively handle failures. Further, their approach of writing intermediate data products to persistent storage simplifies the failure handling logic.

In both Hadoop and the Google's MapReduce, failures of *map* tasks are handled by rerunning them, while a failure of *reduce* tasks requires downloading the outputs of *map* tasks and re-execution of the *reduce* task. The master process that handles the scheduling and keeps track of the overall computation is assumed to run on a node that is less susceptible to failures. A failure in this node requires a total restart of the overall runtime.

2.4. Batch Queues

Batch queues provide an interface to schedule jobs using computation infrastructures ranging from single clusters to computation grids[26]. The jobs for these schedulers could be as simple as

an application running on a single computer, or as complex as a parallel application that runs on thousands of computers. The main task of these schedulers is to allocate resources to this wide variety of requirements in a fair manner while maximizing the resource utilization. Many parallel applications that resemble “embarrassingly parallel” characteristics, either as data parallel or task parallel, can be scheduled as a collection of independent tasks using job queues. These independent tasks typically do not require any form of inter task communication, and hence, they fit best with such a scheduling mechanism.

There are several ways one can access input data in the above types of applications, including network file system in a cluster, shared file systems, or even the option of moving data dynamically to the local disks using scripts. Typically, the data is moved to computation resources during the execution time. MapReduce programming model reduces to a “map-only” mode when no *reduce* phase is used in the computation which resembles an embarrassingly parallel application executed as a collection of *map* tasks. However, unlike the batch queues where the input data for applications are typically moved to computation resources, the data centered approach adopted in MapReduce allows for better data and computation affinity by minimizing the data movement costs. Several have suggested[27, 28] locality aware scheduling to minimize the data movement, as this is especially effective for data intensive applications.

The task granularities of such applications vary with the type of application but coarse grain tasks would yield lower overheads due to lower scheduling overheads. With correct task granularities, similar parallel applications scale almost linearly with the data and computing resources. When MapReduce is used as a “map-only” computation to execute a collection of independent tasks, the distinction between the Batch Queues and MapReduce becomes blurred except for the differences in the data handling. Batch queues provide mechanisms to monitor the status of jobs, and mechanisms to guarantee fault tolerance, which are based on checkpoints and

the re-execution of task[29, 30]. Scheduling a large number of related parallel tasks using job queues can be required by some parallel applications to which runtimes such as SWARM[31] can be applied.

Thilina et al. showed that the same concept of batch queues can be used in Clouds to schedule parallel applications that are embarrassingly parallel[32]. In these settings, the schedulers are replaced by a user developed applications that simply monitors a message queue for task descriptions to execute on cloud resources.

2.5. Cycle Harvesting

Cycle harvesting techniques such as the Berkeley Open Infrastructure for Network Computing (BOINC)[33] achieve massive computational power by aggregating the compute time donated by the voluntary participants around the globe. Condor is a workload management system that was first developed as cycle harvesting technique, and later, it evolved into a fully fledged distributed processing infrastructure. Condor creates a pool of computational resources, to which the users can submit jobs. The resources in the Condor pool can be either cycle sharing or dedicated. Typical data access patterns in both these approaches (public and private domains) involve moving data to the computation resources and hence, they are more suitable for applications with higher computation to data access ratios. Condor provides two problem solvers (i) Master-Worker, and (ii) Directed Acyclic Graph Manager (DAGMan) [34]- which could be used to execute various parallel computation tasks. The master-worker approach can be used to execute a set of parallel computation tasks such as parameter searches where the program performs the same computation on slightly different inputs. These types of computations are *embarrassingly parallel* in nature and work well with this type of infrastructures. DAGMan allows the user to specify the computation task as a Directed Acyclic Graph (DAG), in which the vertices represent the computation tasks and the edges represent the data flow between two computation tasks

(vertices). Although the DAG approach expands the applicability of Condor to more complex parallel processing algorithms, still, the expressiveness of a DAG is limited compared to a scripting language or a fully-fledged programming language that could be used to describe the problem. In addition, representing iterative computations is hard in DAG based systems. Condor supports task-level check pointing so that a task executing in a compute node can be stopped and migrated to another node for further execution in the case of the failure of the current hardware node.

2.6. Work Flow

Workflows schedule a set of related tasks to be tied up as an execution graph (workflow) on distributed computational resources where the tasks could be simple executables or fully-fledged parallel programs that use hundreds of processes. With Grid computing [26], the scientist use workflows to execute large scale scientific applications using many heterogeneous systems across the Grid. G. Fox and D. Gannon define the workflow in the context of Grid computing as follows.

“The automation of the processes, which involves the orchestration of a set of Grid services, agents and actors that must be combined together to solve a problem or to define a new service” [35]

Workflow runtimes such as Pegasus[36] extend the resource allocation to the emerging cloud environments as well. Typical workflow scheduling is mainly determined by the availability of the resources and hence, the workflow runtimes schedule data movement jobs along with the tasks in the workflows. Data locality aware scheduling is used for data intensive applications. The granularity of the services used in the workflows is coarser than the individual computations in MapReduce domain. However, for many embarrassingly parallel applications that are composed of independent tasks, this distinction becomes blurred. G. Fox and D. Gannon further classify the workflows into four distinct classes depending on the complexity of the workflow

graphs: (i) linear workflows, (ii) acyclic graphs, (iii) cyclic graphs, and (iv) complex – too large and complex to effectively “program” as a graph.

The composition of workflows can range from simple scripts to widely used service composition languages to graphical interfaces such as those discussed here [37]. Supporting the quality of services such as fault tolerance and monitoring is an important consideration in workflow runtimes. J. Workflow runtimes support fault tolerance in two granularities : (i) task-level, and (ii) workflow-level. At the task level, individual services or tasks are supported with fault tolerance, whereas in the workflow level, the entire workflow graph is supported with fault tolerance. Yu and R. Buyya in their paper[38] present a list of major workflow runtimes and their support for fault tolerance. Workflow composition tools such as Xbia[39] provides graphical interface to compose workflows as well as to real time monitoring of running workflows.

2.7. Parallel Languages

Parallel languages try to minimize the complexity of programming parallel applications. There are a wide variety of parallel languages targeted for different styles of programming[40] However, as the applications become more and more data intensive, languages that supports coarse the grained SPMD style programming models on distributed memory architectures are proven to be more beneficial. In this respect, language extensions provided as libraries such as MPI (in various languages such as C, C++, and FORTRAN), PVM, Charm[41] provide more flexibility to program parallel algorithms on distributed memory infrastructures (We will discuss them in more detail in the next section). In contrast, as hardware resources become more and more multi/many core oriented, the applications can use parallel languages to exploit the fine grain parallelism available in programs.

Three high level languages that came into the forefront with the emergence of MapReduce technologies are Sawzall[42], DryadLINQ, and PigLatin[43]; all of which provide high level

abstractions to develop MapReduce style programs and other extensions using basic MapReduce constructs. These languages are discussed in the rest of this section.

2.7.1. Sawzall

Sawzall is an interpreted programming language for developing MapReduce like programs using Google's distributed infrastructure services such as GFS and MapReduce. R. Pike et al. present its semantics and its usability in their paper[42]. The language supports computations with two distinct phases. First, a “query” phase performs an analysis on a single data record and emits an output to an aggregator operation specified in the program. Unlike MapReduce, each parallel task in the query phase emits a single value and the aggregator, written in typical sequential languages (C++), collects the emitted results and produces another set of outputs. Finally, the outputs of the aggregators are collected to a single file and stored. The programming model bears a strong resemblance to MapReduce and, according to R. Pike et al., both phases of the Sawzall programs are executed using the MapReduce infrastructure itself. Sawzall provides high level abstractions to define the operations in the query phase and the program flow using predefined aggregators. The Sawzall interpreter schedules the computations across distributed computing infrastructure with the use of the MapReduce infrastructure.

Since Sawzall utilizes both the MapReduce and the GFS infrastructures, the applications developed using Sawzall also get the benefits of data compute affinity and the robustness of MapReduce. The language simplifies the development of some MapReduce applications, but the extra simplifications and its coupling with text processing may limit its usage for general MapReduce applications.

2.7.2. DryadLINQ

DryadLINQ stems from the Language Integrated Query (LINQ)[44] extensions provided by Microsoft. LINQ provides a query interface to many data structures such as arrays and lists that typical programs operate on. DryadLINQ extends this concept to a distributed tables and lists using the underlying Microsoft Dryad[4] infrastructure. With DryadLINQ, the user can execute programming functions, executables on partitions of data defined via a construct named a “partitioned table”. The typical operation involves query operations such as “Select” and “SelectMany” and the aggregate operations such as “GroupBy” and “Apply”. However, unlike Sawzall which maps computations to MapReduce, the applications developed using DryadLINQ are compiled to Dryad executable DAGs, which provides more flexibility in expressing complex applications. Furthermore, with its coupling to PLINQ (Parallel LINQ), DryadLINQ applications can exploit parallelism at the machine level by using typical LINQ queries as well. For example, a collection of records that are assigned to a single machine in a distributed computation can be processed by using LINQ (underneath using PLINQ) in parallel. This is especially beneficial in multi-core computers.

2.7.3. PigLatin

PigLatin is a high-level language developed to simplify query style operations on large data sets using Apache Hadoop. It generates MapReduce programs necessary for query operations expressed using its syntaxes and it executes them using Hadoop. The data is typically consumed as “tuples” comprised of many “fields” and the query operations are defined on tuples. Similar to DryadLINQ, PigLatin also supports user defined functions for various query operations as well. However, unlike DryadLINQ the PigLatin queries and sequential programs cannot be interspersed by limiting its use for complex operations.

2.8. Message Passing

MPI[6], the de-facto standard for parallel programming, is a language-independent communications protocol that uses a message-passing paradigm to share data and state among a set of cooperative processes. MPI specification defines a set of routines to support various parallel programming models such as point-to-point communication, collective communication, derived data types, and parallel I/O operations. There were many parallel programming efforts based on the general principle of message passing such as Chimp[45] and PVM[12] before the wide acceptance of MPI, and it captures the knowledge gained from most of its predecessors. The use of fully-fledged programming languages allows MPI programs to express both the parallel tasks and the overall program logic using all of the available language constructs without being restricted to a particular subset such as a graph language or a script. MPI runtimes are available for many programming languages such as C++, Fortran, Java and C# and hence, they have become the de-facto standard for parallel processing

Typical MPI deployments involve computation clusters with high-speed network connections between nodes. MPI processes have a direct mapping to the available processors or to the processor cores in the case of multi-core systems yielding a static scheduling. Applications can utilize these static sets of processes in various topologies such as 2D or 3D grids, graphs, and even no topologies using the MPI communication constructs in addition to dynamic processes groups.

MPI communication constructs can consist of two forms: (i) individual process to process communication and (ii) collective communication. Two cooperating processes use “send” and “receive” constructs to perform inter-process communication which can be manifested in three modes: (i) standard – a message is delivered when the receive is posted; (ii) ready – the corresponding receive should be posted before the send operation; and (iii) synchronous – the

send does not return until the matching receive is executed. These three modes are then coupled with two client side versions - blocking and non-blocking, and these various combinations provide the user with six configurations. The collective communication constructs such as broadcast, scatter, and gather all have two forms: (i) one-to-all and (ii) all-to-all; these options offer another six modes of communication between processes. These flexible communication routines allow programs to utilize various topologies as opposed to the limited programming topologies supported by the higher level programming models such as MapReduce and Dryad in which virtually no direct process-to-process communication is supported.

Fine-grained sub computations and small messages are characteristics common for typical MPI programs. By contrast, MapReduce uses coarse grained computations and messages. In MPI, the messages are routed in a highly efficiently manner by using the low latency communication channels between the computation nodes, whereas in MapReduce, the messages typically go through a high latency path of local disks->wire->local disk which is essential in providing robust runtimes.

Accessing input data via shared file systems is a common approach to accessing data in MPI. An interface to high performance parallel I/O was introduced in MPI2 (MPI standard 2). Its implementations, such as ROMIO[46], minimize the effect of non-contiguous fine grained data accesses by accessing data in large blocks and transferring them by using the MPI interconnect network. However, the data centered approach adopted by MapReduce and similar runtimes, provides a different set of capabilities to the applications - specifically, the possibility of moving computation to data. Most MapReduce runtimes schedule tasks depending on data locality; this is acquired based on the distributed file system which serves as their foundation, and on top of which they are built.

The rich set of communication constructs available in MPI makes it highly desirable for implementing parallel applications. However, this feature also makes it harder to support fault tolerance in MPI as well. As the processes and messages both store state of the overall computation, complex fault tolerance strategies need to be incorporated to achieve a high degree of robustness. W. Gropp and E. Lusk in their survey paper[47] on fault tolerance in MPI suggest several approaches of establishing MPI programs fault tolerance. There are many ongoing research projects such as OpenMPI[48], FT-MPI[49] and MPICH-V[50], which address the fault tolerance in MPI as well.

2.9. Threads

Various thread libraries are used to exploit the parallelism in shared memory hardware, ranging from graphics processors to large scale SMP (Symmetric Multiprocessing) machines. Threads support fine grained task distributions and provide the first level of parallelism to programs in many applications. Implementations of the POSIX threads[51], boost[52], OpenMP[53], TPL[54] , and Intel TBB [55] are examples of these types of libraries. Furthermore, most languages support some form of threading support as well. Libraries such as CCR[56] provide more sophisticated parallelism based on message passing concepts while PLINQ[57] provides parallel querying capabilities to the .NET languages.

Threads are used to support parallelism at the machine level by various runtimes. For example, OpenMP and MPI can be used in a hybrid approach to produce distributed parallel applications, Similarly, MapReduce runtimes utilize threading to handle parallelism at the machine level and, in DryadLINQ, PLINQ handles the parallelism at the node level while Dryad manages tasks across nodes. In most of these cases, threads are used to execute parallel tasks which utilize multiple processing elements of the underlying hardware platform, although the efficiency of these approaches depends mainly on the characteristics of the application. For example, when a

MapReduce runtime uses threads to execute a set of tasks, the performance depends on the data/compute intensive characteristics of the tasks. If they are highly compute intensive, then the threads will produce near linear performance with the number of processing elements. On the other hand, if the tasks are I/O bound, as in many cases of data intensive computing, the performance depends greatly on the memory and disk bandwidths of the underlying hardware platform.

The paper presented by Cheng-Tao et al. discusses their experience in developing a MapReduce implementation for multi-core machines[14]. Although their work is one of the key motivation of our research, our preliminary research revealed that the performance of such a runtime is lower than a solution developed using pure threads. However, it can provide a simple programming model for the user.

2.10. Cloud

Among many definitions of Cloud computing, the most prominent features include: (i) providing infrastructure, software, and platform as services accessible over the web; (ii) use virtualization for many of its benefits, including isolation of operating systems from bare-hardware and better utilization of resources; and (iii) exploiting economies of scale to deliver these services using massive scale data centers. This trend has created large scale cloud deployments in many commercial infrastructures such as Amazon EC2, Microsoft Azure[19], GoGrid[58], and ElasticHosts[59]. Furthermore, the availability of open source cloud infrastructure software such as Nimbus[60] and Eucalyptus[61], and the open source virtualization software stacks such as Xen Hypervisor[62], allows organizations to build private clouds to improve the resource utilization of the available computation facilities. This option can provide most of the benefits from the commercial clouds except the virtually infinite view of the resources.

The introduction of commercial cloud infrastructure services has allowed users to provision compute clusters fairly easily and quickly, by paying a monetary value for the duration of their usage of the resources. The provisioning of resources happens in minutes, as opposed to the hours and days required in the case of traditional queue-based job scheduling systems. In addition, the use of such virtualized resources allows the user to completely customize the Virtual Machine (VM) images, and use them with root/administrative privileges; this is another feature that is hard to achieve with traditional infrastructures. The possibility of dynamically provisioning additional resources by leasing them from commercial cloud infrastructures makes the use of private clouds more promising.

However, cloud infrastructures provide more services than renting virtual machines to the users. For example, Amazon cloud offerings include storage mechanisms such as Simple Storage Service (S3), Blob Storages – binary large objects stored in persistent manner, messaging services such as Simple Queue Service (SQS), and also computation runtimes such as Elastic MapReduce. Microsoft Azure provides a platform as services by offering basic requirements to develop applications on data centers as services. These services include blobs, queues, databases and also workers that are named as “roles”. All these services differ from the traditional view of computing in individual machines or computation clusters where the applications have access to a local disk, shared file systems, and fast network communications.

The relevance of Cloud services to the data intensive computing is two folds. First, Cloud services represent an alternative to acquiring the computation power (and storage) necessary to do large scale analyses. Second, the various services provided by the Clouds can be used to develop new paradigms for data analyses. For example, one can develop MapReduce-like applications by using cloud services such as blobs, queues, and tables, which are inherently fault tolerance and reliable without using any runtimes such as Hadoop.

2.11. Summary of Features Supported by Different Runtimes

The following table highlights the features supported by three Cloud Technologies and MPI.

Table 1. Comparison of features supported by different parallel programming runtimes

Feature	Hadoop	Dryad/DryadLINQ	Sphere/Sector	MPI
Programming Model	MapReduce and its variations such as “map-only”	DAG based execution flows (MapReduce is a specific DAG)	User defined functions (UDF) executed in stages. MapReduce can be simulated using UDFs	Message Passing (Variety of topologies constructed using the rich set of parallel constructs)
Input/Output data access	HDFS	Partitioned File (Shared directories across compute nodes)	Sector file system	Shared file systems
Intermediate Data Communication	Local disks and Point-to-point via HTTP	Files/TCP pipes/ Shared memory FIFO	Via Sector file system	Low latency communication channels
Scheduling	Supports data locality and rack aware scheduling	Supports data locality and network topology based run time graph optimizations	Data locality aware scheduling	Based on the availability of the computation resources
Failure Handling	Persistence via HDFS Re-execution of failed or slow map and reduce tasks	Re-execution of failed vertices, data duplication	Re-execution of failed tasks, data duplication in Sector file system	Program level Check pointing (OpenMPI[63], FT MPI[49])
Monitoring	Provides monitoring for HDFS and MapReduce	Monitoring support for execution graphs	Monitoring support for Sector file system	XMPI [64], Real Time Monitoring MPI [65]
Language Support	Implemented using Java. Other languages are supported via Hadoop Streaming	Programmable via C# DryadLINQ provides LINQ programming API for Dryad	C++	C, C++, Fortran, Java, C#

The applicability of a parallel runtime to a problem at hand is mainly determined by the parallel topology of the application and whether the runtime can be effectively used to support such a topology. Moreover, the development of a parallel algorithm to a problem is also determined by the parallel constructs supported by the runtime used to implement such an algorithm. For example, an algorithm that expects direct communication between parallel processes is a better match for runtimes with message passing capabilities; such an algorithm may not be a suitable candidate for runtimes such as MapReduce. Therefore it is important to understand the various classes of applications that a particular runtime can support; in our context, it is the MapReduce runtimes in which we are interested.

Parallel applications can be categorized according to their mapping to hardware and software systems. A broader classification based on Flynn’s taxonomy[66] uses Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD) categories to classify parallel applications. Fox defined five classes of applications as follows[67].

Table 2. Application classification

1	Synchronous	The problem can be implemented with instruction level Lockstep Operation as in SIMD architectures
2	Loosely Synchronous	These problems exhibit iterative Compute-Communication stages with independent compute (map) operations for each CPU that are synchronized with a communication step. This problem class covers many successful MPI applications including partial differential equation solution and particle dynamics applications.
3	Asynchronous	Compute Chess and Integer Programming; Combinatorial Search often supported by dynamic threads. This is rarely important in scientific computing but it stands at the heart of operating systems and concurrency in consumer applications such as Microsoft Word.
4	Pleasingly Parallel	Each component is independent. In 1988, Fox estimated this at 20% of the total number of applications but that percentage has grown with the use of Grids and data analysis applications as seen here. For example, this phenomenon can be seen in the LHC analysis for particle physics [68].
5	Metaproblems	These are coarse grain (asynchronous or dataflow) combinations of classes 1)-4). This area has also grown in importance and is well supported by

		Grids and is described by workflow.
--	--	-------------------------------------

The composable applications we discussed earlier contain features from classes 2, 4, and 5. Here, the applications are composed of one or more individually parallelizable stages. Parallel runtimes as Google MapReduce, Hadoop, and Dryad can be used to parallelize stages that perform “pleasingly parallel” operations such as calculating Smith Waterman or calculating histogram of events. As we will show in this thesis, with extended MapReduce capabilities, some stages that require MPI style parallel constructs can also be implemented using MapReduce. Multiple such applications can then be executed in workflows to achieve complex processing. With applications involving only the simple parallel operations such as applying a computation function to a collection of data files, the distinction between composability by using a runtime such as MapReduce vs. workflows gets blurred. Figure 2 shows the structure of composable applications.

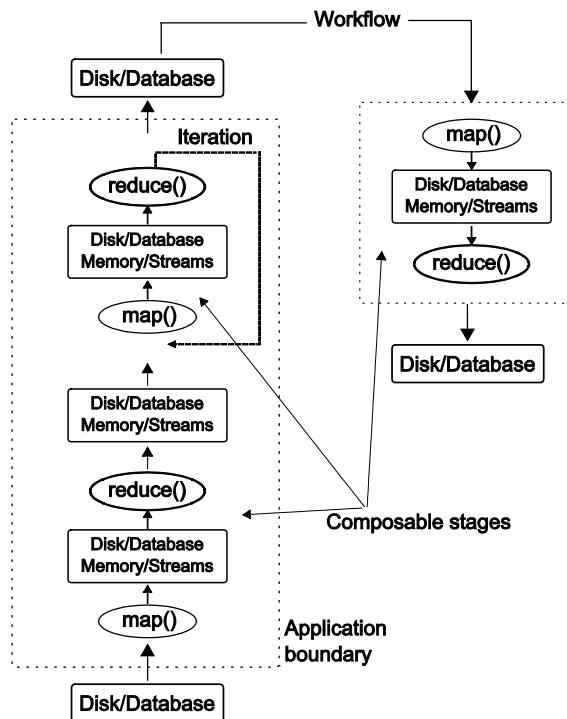


Figure 2. Composable applications in a workflow.

One can classify MapReduce with its support for large scale data processing as a new addition to the above list of categorizations. The MapReduce programming model can support applications in classes 4 and 5, and with the extended programming model and the efficient runtime that we will propose in this thesis it can support some of the applications in class 2 as well. Therefore, the MapReduce class subsumes aspects of classes 2, 4, and 5.

The types of applications that can be supported (some problems in 2, 4, and 5 above) in MapReduce can be categorized into four distinct sub classes: (i) map-only; (ii) map-reduce; (iii) iterative-map-reduce; and (iv) extended MapReduce applications. Complex applications can be built by combining the first three basic execution units under the MapReduce programming model and additional algorithms can be supported with further extensions. We will defer the discussion on further extensions to a later section (Chapter 6 section 6.9) and focus on the first three types of basic MapReduce classes in this chapter. Although this categorization applies to the MapReduce model, we can use the same sub categories to classify applications that can be supported by the high-level runtimes such as Microsoft Dryad as well.

3.1. Map-only Applications

The embarrassingly parallel class of applications represents the simplest form of parallel computations with minimum inter-task dependencies. Converting a collection of documents to a different form, parametric sweeps, and brute force searches in cryptography are all examples of this category of applications. In the MapReduce programming model, the tasks that are being executed at a given phase have similar executables and similar input and output operations. With zero *reduce* tasks, the MapReduce model reduces to a map-only model which can be applied to many embarrassingly parallel applications. Similarly, the DAG based execution flow of Dryad will also reduce to the collection of independent vertices in this category. Furthermore, other runtimes such as Sphere, that uses user-defined functions, and the software systems such as

batch queues, Condor[34], Falcon [69], classic cloud services such as Amazon Queues, and SWARM [31], all provide similar functionality by scheduling large numbers of individual maps/jobs.

3.2. MapReduce Applications

In MapReduce, the *map* and *reduce* phases perform the computations in parallel while the combination of intermediate keys and the shuffling strategies can be used to create different parallel topologies according to the parallel algorithm. For example, consider a word sorting application implemented in MapReduce. Here, the *map* tasks simply perform a scatter operation on the input words. The intermediate keys are the words themselves, and the shuffling is done so that the words that start with a given letter end up in the same *reduce* tasks. The *reduce* tasks then sort their inputs, and one can create the complete sorted output by taking the reduce outputs in the order of the letters assigned to the reduce tasks. In this application, the intermediate keys and the shuffling mechanisms are used to simulate a bucket sort[70] algorithm. Although not completely independent of the above, the runtime parameters such as (i) input data partitions, (ii) the number of maps, (ii) the number of reducer tasks can be used to fine tune the parallelism of MapReduce applications.

The applications described in the Google paper mainly use the *map* stage to distribute the intermediate <key,value> pairs putting less weight on the *map* stage of the computation while the *reduce* tasks perform significant amount of the computations. However, this approach produces large amounts of intermediate data transfers. To minimize this, the authors introduce a local reduction operation, which can perform a local reduction operation on the *map* outputs produced in a given machine. From our experience in mapping various applications to the MapReduce model, we argue that, by making map tasks coarser grained, one can gain better performance. For example, in the word count program, instead of inputting a single word to a *map* task, one can

write a map task that takes a set of lines or a whole file as the input and which can then produce partial word counts. This will make map tasks coarser and reduce the amount of intermediate communications, as the map tasks can perform a local reduction by itself by accumulating counts of words it encounters. However, we also note that these characteristics are highly application dependent. For example, in an application that we will discuss in more detail later, the *map* tasks perform most of the computation and the *reduce* tasks simply combine results. On the other hand, in the matrix multiplication (“Fox algorithm using extended MapReduce”), we use *map* tasks to distribute matrix blocks while the *reduce* tasks perform the matrix multiplication operations.

Selecting an appropriate key selector function is also an important aspect that one should consider in mapping applications to MapReduce domain. A creative use of key selectors will produce elegant MapReduce algorithms. Applications that can be implemented using MPI collective operations can be implemented using MapReduce, but still this does not capture all the low-level messaging constructs offered by the runtimes, such as MPI.

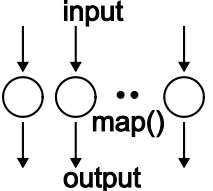
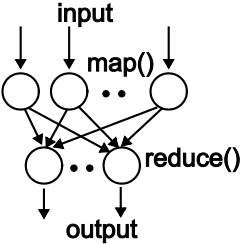
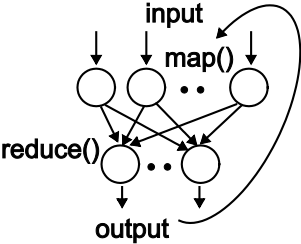
3.3. Iterative MapReduce Applications

Clustering, classification, pattern mining, and dimension reduction are some of the areas where many of the iterative algorithms are used. For example, K-Means[71], Deterministic Annealing Clustering[72], pagerank[73], and dimension reduction algorithms such as SMACOF[74] are all examples of such algorithms. Most of these types of algorithms can be parallelized by applying the SPMD style to the main computations that are executed inside the iterations. Depending on the algorithm, there can be one or more SPMD steps inside the main iterative construct. Once such an algorithm is developed, applying MapReduce to parallelize the SPMD sections is a fairly straightforward. Cheng Tao et al. described this idea by giving ten such machine learning algorithms in their paper[14].

However, the “side effect free”-nature of the MapReduce programming model does not fit well for iterative MapReduce computations in which each *map* and *reduce* tasks are considered as atomic execution units with no state shared in between executions. In parallel runtimes such as those of the MPI, the parallel execution units live throughout the entire life of the program; hence, the state of a parallel execution unit can be shared across invocations. On the other hand, the side effect free nature of MapReduce is one of the key features that makes it easier to support fault tolerance. We propose two strategies to extend the MapReduce programming model to suit this class of applications: (i) an intermediate approach where the *map/reduce* tasks are still considered side effect-free, but the runtime allows for the configuring and the re-usage of the *map/reduce* tasks. Once configured, the runtime caches the *map/reduce* tasks. In this way, both the *map* and the *reduce* tasks can keep the static data in memory, and can be called iteratively without loading the static data repeatedly; (ii) allow *map/reduce* tasks to hold states (support computations with side effects) and adopt different fault tolerance strategies that suit iterative computations. These extensions are discussed in more detail in the next section.

Table 3 shows the data/computation flow of these three MapReduce patterns, along with examples.

Table 3. Classes of MapReduce applications

Map-only	Map-reduce	Iterative map-reduce
		
<ul style="list-style-type: none"> • Converting a collection of documents to different formats • Processing a collection of medical images, • Brute force searches in cryptography • Parametric sweeps 	<ul style="list-style-type: none"> • HEP data analysis (more details will follow) • <i>Histogramming</i> operations, • distributed search, and distributed sorting • Information retrieval 	<ul style="list-style-type: none"> • Clustering • Classification/Regression • Dimension Reduction • Matrix Multiplication • Pagerank

Chapter 4. **A Programming Model for Iterative MapReduce Computations**

As discussed in the previous section, there are many iterative algorithms that can be parallelized by applying the SPMD model to the main computations that are executed inside the iterations. Once such algorithms are identified MapReduce can be used as a parallelization construct to implement the SPMD portions of the algorithms resulting iterative MapReduce computations. In this section, we will discuss an extended MapReduce programming model which can be used to support most such computations efficiently.

Further analysis of some of these algorithms revealed a set of common characteristics such as: most such algorithms utilize data products that remain static throughout the computation as well

as data products that change during the computations; many of them use iterations until convergence; many require the *reduce* output as a whole to make the decision to continue or stop iterations; and we also discovered that the iteration boundaries can be used as good fault tolerance checkpoints. To support such algorithms, we need an extended MapReduce programming model and an efficient runtime implementation, which we try to provide in Twister. (Note: Twister is the name given to the MapReduce runtime we developed as part of this research; hence, we will use it hereafter to refer to the new MapReduce runtime). Twister adopts a programming model that can support the above features of iterative algorithms. A high level view of the programming model is shown in Figure 3 followed by a detailed discussion.

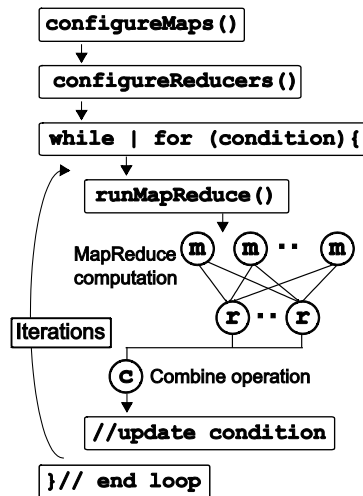


Figure 3. The iterative MapReduce programming model supported by Twister.

4.1. Static vs. Variable Data

Many iterative applications we analyzed display the common characteristic of operating on two types of data products that we called static and variable data. Static data (most of the time the largest of the two) is used in each iteration and remain fixed throughout the computation, whereas variable data consists of the computed results in each iteration which become consumed in the next iteration, in many expectation maximization (EM) type algorithms. For example, if we

consider the K-means clustering algorithm[71], during the n^{th} iteration the program uses the input data set and the cluster centers computed during the $(n-1)^{\text{th}}$ iteration to compute the next set of cluster centers. Similarly, in each iteration, the pagerank algorithm[73] accesses the static web graph and the current pageranks computed during the previous step. To support *map/reduce* tasks operating with these two types of data items, we introduced a “configure” phase for *map* and *reduce* tasks, which can be used to load (read) any static data to the *map* and *reduce* tasks. With this improvement, a typical *map* phase of the computation then consumes the variable data specified as *(key, value)* pairs as well as the static data loaded during the configuration phase. This is different from other MapReduce runtimes where only one input data set is accessible at the *map* phase of the computation. At the same time, typical MapReduce applications (applications without iterative MapReduce computations), when developed using the Twister runtime, require the use of the configure phase to access any input data. For example, in a word-count application, the input data partitions are assigned to *map* tasks during the *configure* phase of the computation, whereas the actual word count operation happens during the *map* phase of the computation.

4.2. Long Running Map/Reduce Tasks

The above programming extension adds capabilities of handling both static and variable data in *map/reduce* tasks. However, reading static data in each execution of the MapReduce computation is highly inefficient. Although some of the typical MapReduce computations such as information retrieval consume very large data sets, many iterative applications we encounter operate on moderately sized data sets that can fit into the distributed memory of the computation infrastructures. This observation led us to explore the idea of using long-running *map/reduce* tasks similar to the parallel processes in many MPI applications that last throughout the life of the computation. The long running (cacheable) *map/reduce* tasks eliminate the necessity of reloading static data in each iteration. Current MapReduce implementations such as Hadoop and

DryadLINQ do not support this behavior, and hence, they initiate new *map/reduce* tasks and load static data in each iteration, which introduce considerable performance overheads for iterative MapReduce computations. Although rare among iterative applications, one can use Twister with extremely large data sets that cannot be fit into the distributed memory of the computation infrastructure by reading data directly from the disks without loading them to memory.

4.3. Granularity of Tasks

The applications presented in Google's MapReduce paper[2] used fine grained *map* tasks. For example, in the word count application, the *map* tasks simply produce $(word, 1)$ pairs for each word they encounter. However, we noticed that, by increasing the granularity of the *map* tasks, one can reduce the volume of the intermediate data that needs to be transferred between *maps* and *reduce* tasks. In the above example, instead of sending $(word, 1)$ for every word, the *map* task can produce partial sums such as $(word, n)$. With the option of configurable *map* tasks, the *map* task can access large blocks of data/or files. In Twister, we adopted this approach in many of our data analysis applications to minimize the intermediate data volumes and to allocate more computation weight to the *map* stage of the computation. Hadoop uses an intermediate combiner operation just after the *map* stage of the computation to support similar behavior. Our approach requires some of the functionality of the *reduce* tasks to be coded in the *map* stage of the computation. However, this coding effort pays off, since the overall execution happens as a single task which results in higher efficiency.

4.4. Side-effect-free Programming

At first glance, the concept of long-running *map/reduce* tasks seems to violate the "side-effect-free" nature of MapReduce by enabling users to store state information in *map/reduce* tasks. However, since the configure operation supports only the static data, the users can still develop "side-effect-

free” MapReduce computations using Twister. Furthermore, the current fault tolerance mechanism in Twister only guarantees the restoring of static data that can be reloaded by using a data partition or any static parameters shared from the main program. This also encourages side-effect-free computations. However, some applications can benefit from the capability of the storing state in *map/reduce* tasks which give rise to a new model of MapReduce computations. We will discuss this approach in section 6.9. Therefore, the users of the Twister runtime can chose to use the fault tolerance capabilities of Twister by storing only static configurations in long running *map/reduce* tasks, or by using the long running tasks to develop MapReduce applications with transient states stored in them (i.e. with side effects), but without the fault tolerance capabilities.

4.5. Combine Operation

In Google’s MapReduce architecture, the outputs of the *reduce* tasks are stored in the distributed file system (GFS) as a collection files. A similar architecture is adopted in Hadoop as well. However, most iterative MapReduce computations require accessing the “combined” output of the *reduce* tasks to determine whether to proceed with another iteration or not. With Twister, we have introduced a new phase to MapReduce named “Combine” that acts as another level of reduction (Note: this is different from the local combine operation that runs just after the *map* tasks in Hadoop). One can use the *combine* operation to produce a collective output from all the *reduce* outputs. In Twister MapReduce, the combine operation is executed by the main program, which contains the iterative construct that enables it to access the reduce output as a whole. However, since this option requires the reduce output to be transferred to the main program, it is only feasible with applications which produce comparatively smaller reduce outputs, or applications which only require meta-data about the reduce output in order to proceed with iterations. For most iterative applications we analyzed, a significant reduction in data volume

occurs when the computation transitions from *map* to *reduce*. This is not the case in typical non-iterative MapReduce computations such as sorting, where all the input is available as the reduce output. However, most such applications do not require iterative computations and they can simply ignore the combine phase of Twister.

4.6. Programming Extensions

We have also incorporated a set of programming extensions to MapReduce in Twister. One of the most useful extensions is **mapReduceBCast(Value value)**. As the name implies, this extension facilitates the process of sending a single Value (Note: MapReduce uses *(key,value)* pairs) to all *map* tasks. For example, the “Value” can be a set of parameters, a resource (file or executable) name, or even a block of data. Apart from the above options, the “configure” option described in section 4.1 is supported in Twister in multiple ways. *Map* tasks can be configured using a “partition-file” – a file containing the meta-data about data partitions and their locations. In addition, one can configure *map/reduce* tasks from a set of values. For example **configureMaps(Value[]values)** and **configureReduce(Value[]values)** are two programming extensions that Twister provides. Twister also provides broadcast style operation between the *map* and *reduce* phases using which a map task can send a single *(key,value)* pair to multiple *reduce* tasks, allowing it to support complex parallel algorithms. We will discuss how these extensions are supported in the coming section.

Twister is a distributed in-memory MapReduce runtime optimized for iterative MapReduce computations. It reads data from local disks of the worker nodes and handles the intermediate data in distributed memory of the worker nodes. Twister utilizes a publish-subscribe (pub-sub) messaging infrastructure for communication and data. In this section, we will explain the architecture of the Twister MapReduce runtime.

The Twister architecture consists of three main entities: (i) client side driver (Twister Driver) that drives the MapReduce computation; (ii) Twister Daemon running on every worker node; and (iii) the broker network (Note: we will simply use the term “broker network” to refer to the messaging infrastructure throughout the discussion). Figure 4 shows the architecture of the Twister runtime.

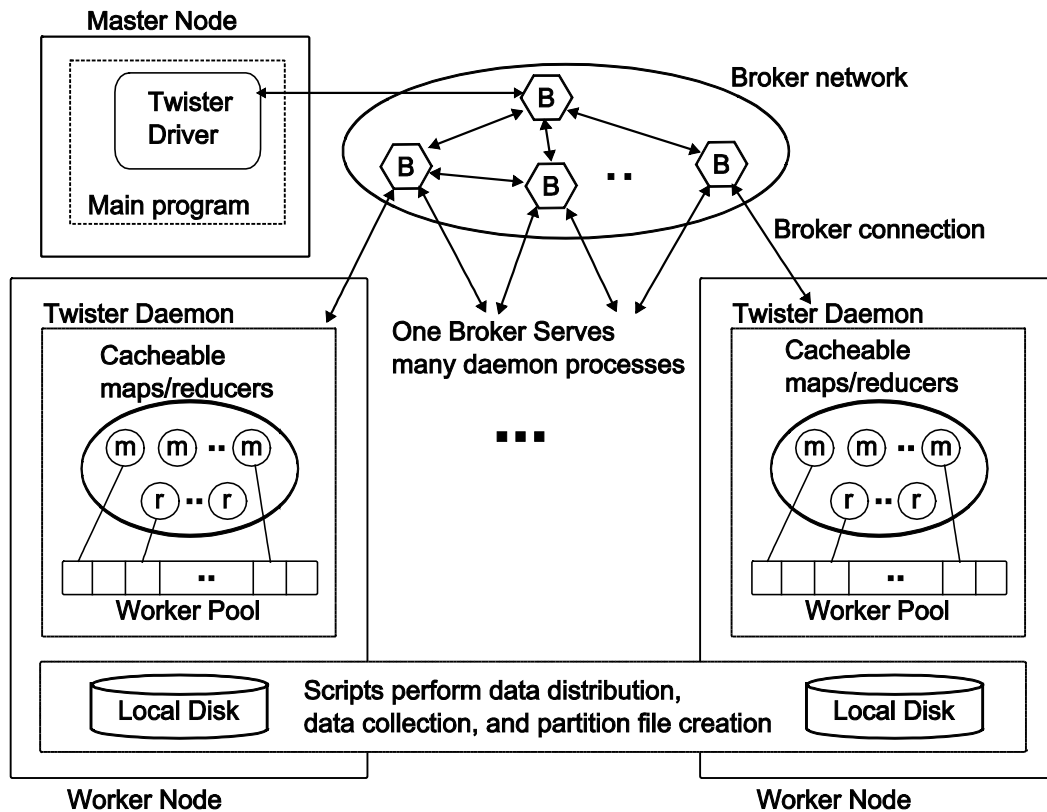


Figure 4. Architecture of Twister MapReduce runtime.

A Twister Daemon runs on every compute node of the computation infrastructure and acts on the commands issued by the Twister Driver. During the initialization of the runtime, Twister starts a daemon process in each worker node, which then establishes a connection with the broker network to receive commands and data. The daemon is responsible for executing map/reduce tasks assigned to it, maintaining a worker pool (thread pool) to execute map and reduce tasks, notifying status to the Twister Driver, and finally responding to control events.

The client side driver provides the programming API to the user, and converts these Twister API calls to control commands and input data messages sent to the daemons running on worker nodes via the broker network. It also handles the recovery of MapReduce computations in an event of a failure.

Twister uses a publish/subscribe messaging infrastructure to handle four types of communication needs : (i) the sending/receiving control events; (ii) sending data from the client side driver to the Twister daemons; (iii) handling intermediate data transfer between the *map* and *reduce* tasks; and (iv) sending the outputs of the reduce tasks back to the client side driver to invoke the combine operation. Currently, it supports the NaradaBrokering[75] and the ActiveMQ[76] messaging infrastructures. However, the Twister architecture clearly separates the communication functionalities from the implementation of the other components so that it becomes very straightforward to use other messaging infrastructures, such as those are based on persistent queues.

5.1. Handling Input and Output Data

Twister provides two mechanisms to access input data for map tasks: (i) reading data from the local disks of worker nodes; and (ii) receiving data directly via the broker network. The first option allows Twister to start the MapReduce computations by using large data sets spread across the worker nodes of the computing infrastructure. Twister assumes that the data read from the local disks are maintained as files, and hence, it supports file based input format, which simplifies the implementation of the runtime. The use of the native files allows Twister to pass data directly to any executable (for example a script or compiled program running as a *map* or *reduce* computation) as command line arguments; this feature is not possible with file systems such as HDFS.

Both Sector and DryadLINQ adopted the same file based input data partitioning strategy as well. A possible disadvantage of this approach is that it does require the user to break up large data sets into multiple files. However, our experience is that it is better to leave the input data partitioning to the user, rather than providing a standard block based approach. For example, in some applications, the input data may already be stored as a collection of files; this situation does

not require any partitioning. In some applications such as BLAST, the data is available in databases which require specific partitioning strategies be adopted. Furthermore, the fixed size block based approach adopted by both Google and Hadoop imposes another restriction as well; the block size is fixed in the runtime but not per application. Therefore, if one needs to run applications with different input data partition sizes, the runtime needs to be reconfigured to achieve optimal performance. For example, in Hadoop, if the block size defined for the HDFS is 64MB, a file inserted to it which is only a few kilobytes in size would require an entire 64MB block. When data is accessed it will retrieve this entire block as well. This optimization issue is not present in the file based approach used in Twister.

In Twister, the meta-data regarding the input file distribution across the worker nodes is read from a file called "partition-file". Currently, the partition file contains a list of tuples consisting of (*file_id*, *node_id*, *file_path*, *replication_no*) fields in them. The concept of the partition-file in Twister is inspired by the DryadLINQ's partitioned-file mechanism. Twister provides a tool which can perform typical file system operations across the worker nodes such as: (i) creating directories; (ii) deleting directories; (iii) distributing input files across worker nodes; (iv) copying a set of resources/input files to all worker nodes; (v) collecting output files from the worker nodes to a given location; and (vi) creating a partition-file for a given set of data that is distributed across the worker nodes. Although these features do not provide the full capabilities that one can achieve via a distributed file system such as GFS or HDFS, the above features try to capture the key requirements of running MapReduce computations using the data read from local disks to support the concept of "moving computation to data". Integrating a distributed file system such as HDFS or Sector with Twister will serve as interesting possibilities for future research work.

Twister also supports sending input data for *map* task directly from the main program via the broker network as well. It will be inefficient to send large volumes of input data via the broker

network for map tasks. However, this approach is very useful for sending small variable data (Note: please refer to the discussion of static vs. variable data in section 4.1) to *map* tasks. For example, a set of parameters, a set of rows of a matrix, or a set of cluster centers represents such data items.

5.2. Handling Intermediate Data

To achieve better performance, Twister handles the intermediate data in the distributed memory of the worker nodes. The results of the *map* tasks are directly sent to the appropriate *reduce* tasks where they get buffered until the execution of the *reduce* computation. Therefore, Twister assumes that the intermediate data produced after the *map* stage of the computation will fit in to the distributed memory of the computation infrastructure. This is generally the case for many iterative MapReduce computations. As we have mentioned earlier, this option is not feasible for computations that produce intermediate data that is larger than the available total memory of the computation nodes. To support such applications, one can extend the Twister runtime to store the *reduce* inputs in local disks instead of buffering in memory and provides an iterator construct which reads data from disk.

The use of memory->wire->memory data transfer in Twister gives it a considerable performance gain compared to disk->wire->disk approach adopted by many other MapReduce runtimes. Twister uses two mechanisms for transferring intermediate data: (i) via the broker network; and (ii) via direct node to node TCP links. As explained earlier, each Twister daemon maintains a connection with one of the pub-sub brokers from the broker network. In addition to above, each daemon also starts a TCP sever as well. When an intermediate data item (a *(key,value)* pair) is produced at a Twister daemon, it identifies the pub-sub topic to which the data item needs to be sent based on the key of the *(key,value)* pair. Then if the size of the data is smaller than 1KB the daemon publishes a message containing the data item to a topic in the broker network. A

daemon subscribed to that topic directly receives the message containing the data item via the broker network. If the data item is larger in size, the first daemon simply adds the data item to an internal cache, maintained by the daemon, and publishes a message containing a key to locate the data item in this daemon's cache. The message also carries the hosting daemon's IP and port information as well. Once the message is received by a daemon that are subscribed that topic, it contact the TCP server of the first daemon and retrieves the data item. This approach eliminates the overloading of the broker network by large data transfers and also adds a considerable parallelism to the intermediate data transfer phase of MapReduce. Twister uses this mechanism for *map-to -reduce* data transfer as well as to reduce-to-combine data transfer phases of the MapReduce computation. Figure 5 these two approaches.

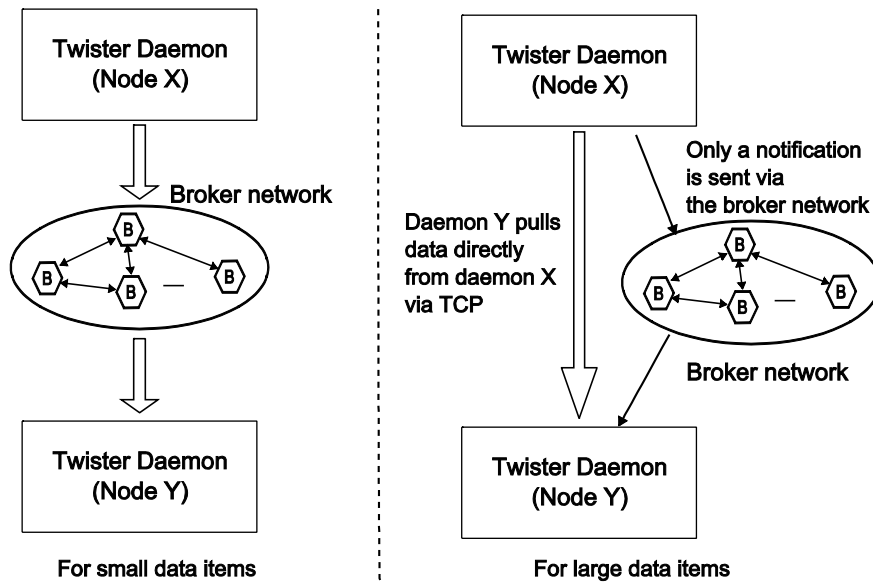


Figure 5. Two approaches used by the Twister to transfer intermediate data.

Note: Please not that the TCP based direct data transfer mechanism described above is added to the Twister to at later phase of this research work, and hence, it is used in the benchmarks discussed in section 6.9 and 6.10 only.

5.3. Use of Pub/Sub Messaging

The use of the publish-subscribe messaging infrastructure improves the efficiency of the Twister runtime. However, to make the runtime scalable, the communication infrastructure should also be scalable. Both the NaradaBrokering and the ActiveMQ pub-sub messaging infrastructures we used in Twister can be configured as broker networks (as shown in Figure 4), so that the Twister daemons can connect to different brokers in the network reducing the load on a given broker. This is especially useful when the application uses `mapReduceBcast()` with large data sets. A benchmark performed using 624 Twister daemons revealed that by using 5 brokers (connected hierarchically with 1 root broker and 4 leaf brokers) rather than 1 broker, the broadcast time can improve by 4 folds for 20MB broadcast messages.

5.4. Scheduling Tasks

The cacheable *map/reduce* tasks used in Twister are only beneficial if the cached locations remain fixed. Therefore, Twister schedules *map/reduce* tasks statically. However, in an event of a failure of worker nodes, it will reschedule the computation on different set of nodes. The static scheduling may lead to un-optimized resource utilization with skewed input data or execution times of the *map* tasks. However, one can minimize this effect by randomizing the input data assignment to the *map* tasks. Ideally, Twister should support multiple scheduling strategies to support various classes of applications, but these improvements are left as future work in the current implementation.

5.5. Fault Tolerance

Twister supports fault tolerance for iterative MapReduce computations. Our approach is designed to save the application state of the computation between iterations so that, in the case of a failure, the entire computation can be rolled back to the previous iteration. Supporting

individual *map* or *reduce* failures in each iteration requires adopting an architecture similar to Google, which will eliminate most of the efficiencies that we have gained using Twister for iterative MapReduce computations. Therefore, we decided to provide fault tolerance support only for iterative MapReduce computations in the current implementation of Twister, based on the following three assumptions: (i) Similar to Google and Hadoop implementations, we also assume that the master node (where the Twister Driver would be executed) failures are rare; and (ii) the communication infrastructure can be made fault tolerance independent of the Twister runtime; and (iii) the data is replicated among the nodes of the computation infrastructure. Based on these assumptions, we try to handle failures of *map/reduce* tasks, daemons, and worker nodes failures.

The combine operation is an implicit global barrier in iterative MapReduce computations. This feature simplifies the amount of state Twister needs to remember in case of a failure to recover. To enable fault tolerance, Twister saves the configurations (information about the static data) used to configure *map/reduce* tasks before starting the MapReduce iterations. In most MapReduce computations, these simply mean only the partition file which contains the meta-data regarding the data distribution. Then it also saves the input data (if any) that is sent directly from the main program to the *map* tasks. In the case of a failure Twister Driver executes the following sequence of actions to recover from the failure:

- (i) Discards the existing *map/reduce* tasks in the given iteration and instructs them to terminate.
- (ii) Polls the Twister Daemons to identify the available (running) Twister Daemons
- (iii) Reconfigures the *map/reduce* tasks and assigns them to the available daemons according to the data locality. In this respect, Twister groups tasks depending on the

available data replications to the available daemons as evenly as possible while maintaining data locality.

- (iv) Executes the current iteration.

As this description demonstrates, the current fault tolerance strategy does not support the recovery of individual tasks. This implies that any state stored in *map/reduce* tasks will be lost in the case of a failure under the current implementation. As we have mentioned above, for side-effect-free MapReduce computations, this does not impose a limitation because the tasks are naturally considered as stateless. Furthermore, when considering the number of iterations executed in an iterative MapReduce computations, (typically hundreds of iterations) re-executing a few failed iterations do not impose a considerable overhead.

Although we left as a future work, one can implement the following fault tolerance strategy with Twister by incorporating a distributed fault tolerance file system such as HDFS. Each Twister daemon can store the state of the *map/reduce* tasks under its control in each n^{th} iteration into the distributed file system where n is specified by the user depending on the application ($1 \leq n \leq \text{max iterations}$). We need a distributed file system to recover from hardware crashes such as disk failures or machine failures, but for software failures such as daemon failures, the saving can be done on local disks. This approach will allow tasks with states to be recovered in the case of a failure. However, still to recover individual task without rolling back entire iterations all the outputs (*map/reduce*) need to be saved to some form of persistent storage as in Google or Hadoop.

5.6. Twister Implementation

5.6.1. Software Requirements

Twister is implemented in Java programming language. For starting, stopping, and file manipulation operations, Twister uses a set of shell scripts which internally invoke remote commands by using secure shell (SSH) protocol. To communicate between nodes, these scripts expect certificate based login between the nodes of the computation infrastructure, which is a common requirement in many distributed runtimes such as MPI and Hadoop. The Java language supports execution of any compiled program as a separate executable, and hence, in addition to pure java functions the user can invoke any script or executables inside the *map* or *reduce* functions using Twister. The use of shell scripts and the SSH limits the usage of Twister to Unix like operating systems.

5.6.2. Twister Daemon

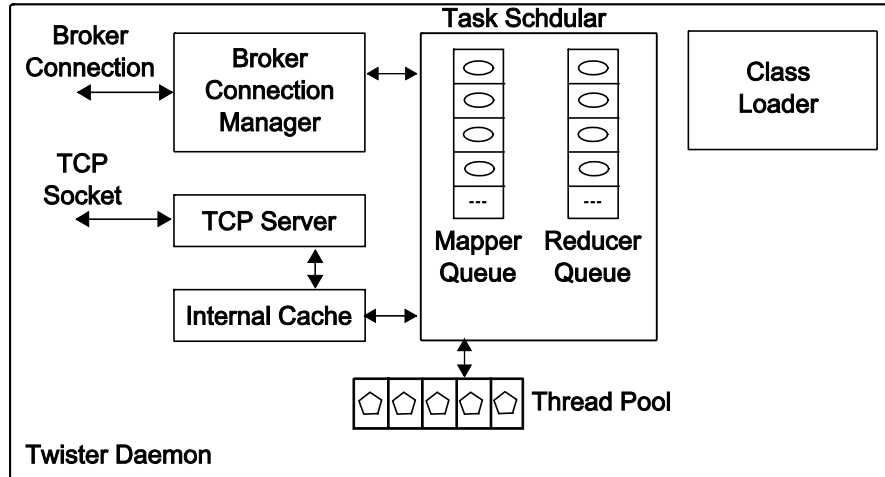


Figure 6. Components of Twister Daemon

Twister daemon is a standalone Java program that is comprised of several software components such as: a thread pool; a TCP server, a task scheduler, internal cache, a class loader, and a broker connection manager. Figure 6 shows these components.

The broker connection manager maintains a connection with a particular broker in the broker network. When Twister is used with ActiveMQ, it establishes a connection to one of the available brokers and it maintains an active connection to one of the available brokers in the network. Since the TCPserver is the bulk data transfer point of the daemon, it is implemented with the support of concurrent connections. Task scheduler maintains a mapper queue and a reducer queue. A mapper holds a single *map* task, and executes it passing the input data and collects any output data produced by the *map* function. It then sends this data to the appropriate reducer. The reducer is a wrapper for a single *reduce* task. It collects intermediate data targeted for that *reduce* task and executes it when the Twister Driver instruct it via the broker network. The reducer also sends the *reduce* outputs to the *combine* function. Task scheduler assigns *map/reduce* tasks to the thread pool for execution. Twister Driver assigns individual tasks to Twister Daemons (processes) while internally the daemons use threads to execute individual *map/reduce* tasks. Therefore, the Twister runtime supports a hybrid task scheduling approach, which is especially effective for computation nodes with multiple processor cores.

Twister does not require the user defined *map/reduce* functions to be available during the deployment of the runtime. The user can upload jar files containing the implementations of MapReduce computations at any time of the Twister's life cycle. It loads these jars dynamically using a custom class loader which is initiated per each job by the runtime. This enables the user to develop MapReduce applications incrementally without needing to restart the Twister runtime.

5.6.3. Twister Driver

Twister Driver is a library that needs to be used in the main program of the MapReduce computation. The user program invokes Twister Driver passing it a job description. The driver then assigns *map/reduce* tasks to Twister daemons and waits for their responses. When all *map*

tasks are completed, the Twister Driver notifies all reducers to invoke their corresponding *reduce* tasks. When reducers send *reduce* outputs, the driver collects them and invoke the *combine* function. Although the *reduce* tasks are executed once all *map* tasks are completed, the data transfer between *map* and *reduce* tasks happens immediately after the completion of the individual *map* tasks. Therefore, in typical MapReduce computations, one can expect the data transfer to be interspersed with computation.

5.6.4. Pub-sub Brokers

Current Twister implementation supports the NaradaBrokering and the ActiveMQ pub-sub brokers. The deployment of the broker network is left to the users to manage as it is highly specific to the individual broker network used. Information regarding the broker network is passed to Twister via a configuration file.

5.6.5. File Manipulation Tool

Twister provides a simple tool to manipulate files and executable programs across computation nodes. The interface is provided in a shell script, which supports the following set of commands.

Table 4. Commands supported by the Twister's file manipulation tool.

Command	Parameters	Description
initdir	[Directory to create - complete path to the directory]	Create a directory in all compute nodes (let's call this <i>data_dir</i>).
mkdir	[sub directory to create - relative to <i>data_dir</i> specified]	Create a sub directory inside data directory in all compute nodes.
rmdir	[sub directory to delete - - relative to <i>data_dir</i>]	Remove a sub directory inside data directory in all compute nodes.
put	[src directory(local)] [destination directory (remote) - relative to <i>data_dir</i>] [file filter pattern] [number of duplicates (optional)]	Distribute input data across compute nodes. This command evenly distributes the available files in the input directory to all compute nodes. It utilizes multiple threads to speed up the process.
putall	[input data directory (local)] [destination directory (remote) - relative to <i>data_dir</i>]	Copy data or any resources in the input directory to all compute nodes.
cpj	[resource to copy to the apps directory]	Copy any user defined application jar files to all compute nodes.
ls	[-a][directory sub directory relative to <i>data_dir</i>]	List files/directories inside the <i>data_dir</i> .
create_partition_file	[common directory - relative to <i>data_dir</i>] [file filter pattern][partition file name]	Creates a partition file containing all data files available in a particular directory of all compute nodes.

5.7. Twister API

Twister Application Program Interface allows users to develop MapReduce computations that can be executed using Twister. Following table lists the individual API construct along with a small description.

Table 5. The Application program interface of Twister.

Type of function	Application Program Interface
Map	<pre> /* Configure the map task */ void configure(JobConf jobConf, MapperConf mapConf) /* The map function */ void map(MapOutputCollector collector, Key key, Value val) /* Any clean up necessary to map task */ void close() </pre>
Reduce	<pre> /* Configure the reduce task */ void configure(JobConf jobConf, ReducerConf reducerConf) /* The reduce function */ void reduce(ReduceOutputCollector collector, Key key, List<Value> values) /* Any clean up necessary to reduce task */ void close() </pre>
Combine	<pre> /* Configure the combine task */ void configure(JobConf jobConf) /* The combine function */ void combine(Map<Key, Value> keyValues) /* Any clean up necessary to reduce task */ void close() </pre>

Configure Maps	<pre> /* Configure map tasks using a partition file */ void configureMaps(String partitionFile) /* Configure map tasks using a set of Value objects */ void configureMaps(Value[] values) </pre>
Configure Reduce	<pre> /* Configure reduce tasks using a set of Value objects */ void configureMaps(Value[] values) </pre>
Run MapReduce	<pre> /* Execute MapReduce using already configured map and reduce tasks */ TwisterMonitor runMapReduce() /* Execute MapReduce using a set of (key,value) pairs sent from the main program */ TwisterMonitor runMapReduce(List<KeyValuePair> pairs) /* Execute MapReduce passing one Value sent from the main program. Each map will get a Key is similar to its map task number. */ TwisterMonitor runMapReduceBCast(Value val) </pre>

In the next section we will discuss some of the applications that we have implemented in Twister and how these architectural features enable better efficiencies in them.

One motivation of this research is to understand the applicability of different parallel runtimes to various classes of applications and analyze their performance characteristics. To achieve this goal, we have implemented a series of data analysis programs using several available parallel runtimes such as Apache Hadoop, Microsoft DryadLINQ, MPI, and Twister. We selected these applications to represent the different classes of applications that are described in Chapter 3. This way, by analyzing their parallel implementations, we can predict the suitability of the above runtimes to applications of similar nature. Some of these are implemented using all the above runtimes whereas some are implemented using few runtimes depending on the suitability.

The easiness in implementing a parallel algorithm using a given runtime and its parallel constructs, gives us an idea about the suitability of the runtime to the class of applications that we

are interested. Also, the selection of real applications to implement gives us an in-depth understanding of various challenges that one may face in developing data analysis programs using the above runtimes. This analysis also serves as a proof to the programming extensions that we have introduced in Twister. Rest of this section is organized as follows. First, it gives a brief introduction to different performance measures that we have used and the calculations performed to understand the various parallelization characteristics. Next, it will give detailed information regarding the hardware and software environments we used in our evaluations. Finally, the section moves into describing the applications and their different implementations. We present performance evaluations under each application followed by a discussion.

6.1. Performance Measures and Calculations

Performance, scalability, overhead, and efficiency are the most common measurements we used in this analysis. Bellow we will give a brief introduction to these measures.

6.1.1. Performance and Scalability

Performance of an application is typically measured as the average execution time, and it is a key measurement that one can use to compare different algorithms and implementations at a higher level. In parallel and distributed applications, this is a collective measure of the performance of the computation units (CPU), the memory, the input/output subsystems, the network, and even the efficiency of the parallel algorithm. Furthermore, the execution time of an algorithm also varies with the size of the problem, i.e. the amount of data, used for the computation. Therefore, in most of our applications, we measure the average execution time by varying the amount of data to understand its effect on performance. For an efficient parallel algorithm, the performance should vary with data according to the computation complexity of the underlying algorithm. For example, the performance of a matrix multiplication typically has a quadratic relation to the

input data. However, it can deviate from the expected relationship due to the effects of parallel overhead, the cache effects, and other runtime overheads.

Scalability in parallel applications is measured by using two approaches. The “strong scalability” is the measure of execution times by keeping the total problem size fixed while increasing the amount of processors. The “weak scalability” is the measure of execution times by keeping the problem size per processor fixed while increasing the number of processors. In most of our performance measures we use the first approach.

6.1.2. Speedup

Speedup is used to understand how well a parallel application performs compared to a sequential version of the same problem, and is calculated using the following formula.

$$Speedup = \frac{T(1)}{T(p)} \quad (1)$$

In this formula $T(1)$ is the execution time of a sequential program whereas $T(p)$ denotes the execution time of the parallel program when p processors are used. When $T(1)$ is measured using a sequential program, the formula gives the absolute speedup. Also, a variation of this formula can be used with parallel applications, in which $T(1)$ is obtained by running the parallel application itself on a single processor. The speedup is governed by the Amdahl law[77] and therefore most programs produce sub linear speedups. However, in some applications, one can notice super linear speedups due to cache effects.

6.1.3. Parallel Overhead

Parallel overhead of an application is a measure of the extra work that the program performs compared to its sequential counterpart. This comprises of, the overheads introduced by the communication and synchronization. Also, in some applications, the duplicate work performed

in parallel tasks may add up to this value. The overhead is calculated using the following formula in which p is the number of parallel processors used, $T(p)$ is the execution time when p processors are used, and $T(1)$ is the sequential execution time.

$$\text{Parallel Overhead} = \frac{p \cdot T(p) - T(1)}{p \cdot T(1)} \quad (2)$$

In this thesis, we use the overhead calculation extensively to compare the parallelization characteristics of different implementations of a given algorithm. In such situations, for $T(1)$ we use sequential execution time of the best performing implementation. In some applications, we use estimations for the sequential time when evaluating sequential time is not possible due to extremely long execution times.

6.1.4. Parallel Efficiency

Parallel efficiency is another measure we use for evaluating the performance of parallel applications. It identifies the efficiency in which the computation resources are used by the parallel programs and is calculated using the formula shown below. Here, p is the number of parallel units, $T(p)$ is the running time with p parallel units, and $T(1)$ is the sequential execution time.

$$\text{Parallel Efficiency} = \frac{T(1)}{p \cdot T(p)} \quad (3)$$

Estimating serial execution times for some applications is not straightforward and, hence, we calculated parallel efficiency using the formula (4) below in which $\alpha = p_1/p_2$ where p_2 is the smallest number parallel units (CPU cores) used for the experiment, so $\alpha \geq 1$. This calculates the parallel efficiency with respect to the minimum number of parallel units used for the experiment.

$$\text{Parallel Efficiency} = \frac{T(p_2)}{\alpha \cdot T(p_1)} \quad (4)$$

6.2. Hardware Software Environments

We have carried out a series of performance evaluations for the different applications by using several computation clusters. This is mainly to facilitate the operating system requirements imposed by the different parallel runtimes. For example, DryadLINQ runs only on Windows Server 2008 operating system whereas Apache Hadoop and Twister run only on Linux operating systems. In most cases, the cluster are installed with one operating system, therefore we had to use multiple computation clusters to obtain the measurements. Although we use different hardware resources, for most of the evaluations, we make sure that the different runtimes run on earthier on the same hardware (with different operating systems depending the runtime requirements) or at least in nearly similar hardware environments.

Apart from hardware differences, depending on the runtime used, the applications are implemented with the best suitable features available in the selected runtime. For example, in DryadLINQ the files are accessed from shared directories while Hadoop uses HDFS. These variations add different performance characteristics to the underlying application, and therefore, the performance comparisons will not be identical in every situation. However, our motivation in this research is to develop and deploy applications using the best strategy for each runtime and analyze their performances to see what benefits one can gain from these different technologies and runtimes. For performance analysis, we used several computation clusters as follows.

Table 6. Details of the computation clusters used.

Cluster ID	Cluster-I	Cluster-II	Cluster-III	Cluster-IV
# nodes	32	230	32	32
# CPUs in each node	6	2	2	2
# Cores in each CPU	8	4	4	4
Total CPU cores	768	1840	256	256
CPU	Intel(R) Xeon(R) E7450 2.40GHz	Intel(R) Xeon(R) E5410 2.33GHz	Intel(R) Xeon(R) L5420 2.50GHz	Intel(R) Xeon(R) L5420 2.50GHz
Memory Per Node	48GB	16GB	32GB	16GB
Network	Gigabit Infiniband	Gigabit	Gigabit	Gigabit
Operating Systems	Red Hat Enterprise Linux Server release 5.4 -64 bit Windows Server 2008 Enterprise - 64 bit	Red Hat Enterprise Linux Server release 5.4 -64 bit	Red Hat Enterprise Linux Server release 5.3 -64 bit	Windows Server 2008 Enterprise (Service Pack 1) - 64 bit

We use the academic release of DryadLINQ, Apache Hadoop version 0.20.2, MPI.NET, OpenMPI, and Twister for our performance comparisons. Both Twister and Hadoop use JDK (64 bit) version 1.6.0_18, while DryadLINQ and MPI.NET uses Microsoft .NET version 3.5. We use OpenMPI version 1.0.

6.3. CAP3 Data Analysis

An EST (Expressed Sequence Tag) corresponds to messenger RNAs (mRNAs) transcribed from the genes residing on chromosomes. Each individual EST sequence represents a fragment of mRNA, and the EST assembly aims to re-construct full-length mRNA sequences for each expressed gene. Because ESTs correspond to the gene regions of a genome, EST sequencing has become a standard practice for gene discovery, especially for the genomes of many organisms that may be too complex for whole-genome sequencing. EST is addressed by the software CAP3 which is a DNA sequence assembly program developed by Huang and Madan[24]. CAP3 performs several major assembly steps including computation of overlaps, construction of contigs, construction of multiple sequence alignments, and generation of consensus sequences to a given set of gene sequences. The program reads a collection of gene sequences from an input file (FASTA file format) and writes its output, including the standard output, to several output files. During an analysis, the CAP3 program is invoked repeatedly to process a large collection of input FASTA files.

Input.fasta -> Cap3.exe -> Stdout + Other output files

Processing a collection of input files using CAP3 can easily be parallelized by performing each invocation in a different processing unit. Since there is no inter task dependencies, it resembles a typical embarrassingly parallel application, i.e. according to our classification; it represents the “map-only” class of applications. We have developed parallel CAP3 programs by using Hadoop, DryadLINQ, and Twister runtimes. The details are presented below.

6.3.1. Hadoop Implementation

The Hadoop version of CAP3 is implemented by developing a map task that executes the CAP3 program as a separate process on a given input FASTA file. The CAP3 application is implemented in C and it expects the inputs as native files. However, the Hadoop file system only

provides an interface accessible via its APIs. This forced us to use a network file system available in the computation cluster to store and access input data for the CAP3 program. This limitation arises in Hadoop when a parallel application uses an executable, most probably a legacy application that needs to access input as files via the command line arguments, as *map* or *reduce* tasks. Since HDFS stores data as blocks on the host file system, the stored data can only be accessed via an APIs provided by the HDFS. Currently HDFS provides both Java and C++ APIs to access data. However, to use these APIs one need to change the existing programs, which is not possible when the source code of an application is not available. One alternative approach is to store input files in HDFS and alter the map task to move the input file from the HDFS to the local file system before invoking the executable program. Since Hadoop schedules tasks based on the data locality, the data movement between HDFS and the local file system typically requires only a local data copying operation and hence may not incur network overheads. However, for large data sets, this approach produces considerable overheads and large temporary storage space.

6.3.2. DryadLINQ Implementation

To implement a parallel CAP3 application using DryadLINQ, we adopted the following approach: (i) the input files are partitioned among the nodes of the cluster so that each node stores roughly the same number of input files; (ii) a “data-partition” (A text file for this application) is created in each node containing the names of the input files available in that node; (iii) a DryadLINQ “partitioned-file” (a meta-data file understood by DryadLINQ) is created to point to the individual data-partitions located in the nodes of the cluster. These three steps enable DryadLINQ programs to execute queries against all the input data files. After these steps, the DryadLINQ program which performs the parallel CAP3 execution becomes just a single line program contacting a “Select” query which select each input file name from the list of file names

and executes a user defined function on that. In our case, the user defined function calls the CAP3 program passing the input file name as program arguments. The function also captures the standard output of the CAP3 program and saves it to a file. Then it moves all the output files generated by CAP3 to a predefined location. Figure 7 shows the method we have adopted to process input as files using DryadLINQ.

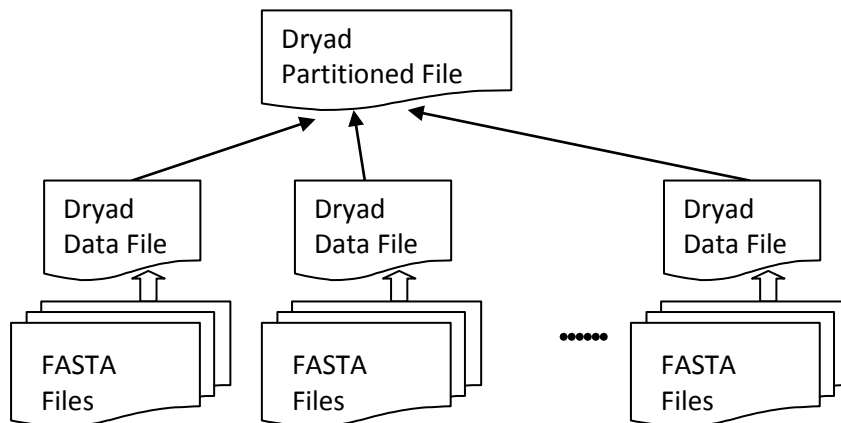


Figure 7. Processing data as files using DryadLINQ.

Although we use this program specifically for the CAP3 application, the same pattern can be used to execute other programs, scripts, or analysis functions written using frameworks such as R and Matlab, on a collection of data files. (Note: In this application, we rely on DryadLINQ to process the input data on the same compute nodes where they are located. If the nodes containing the data are free during the execution of the program, the DryadLINQ runtime will schedule the parallel tasks to the appropriate nodes to ensure co-location of process and data; otherwise, the data will be accessed via the shared directories.) Unlike in Hadoop, in DryadLINQ the user is expected to handle the data partitioning. Also, it stores input data on the local disks of the compute nodes directly as files. Therefore, the locally stored data files are directly accessible to the computation vertices that are schedule to run on that particular computation node.

6.3.3. Twister Implementation

Similar to Hadoop, in Twister, the user can create a map-only application by not specifying a particular *reduce* task. Twister adopts DryadLINQ's approach to input data storage where it stores input files on the local disks of the compute nodes. The input FASTA files are distributed using the data distribution tool provided by Twister before executing the application. The meta-data regarding the file distribution is stored in a *partition file*, which is used by the Twister for scheduling computation tasks.

6.3.4. Performance Evaluation

We measured both the performance and the scalability of the three implementations of CAP3 by using a data set containing FASTA files each with roughly 460 short sequences. The results of these benchmarks are shown in Figure 8 and Figure 9.

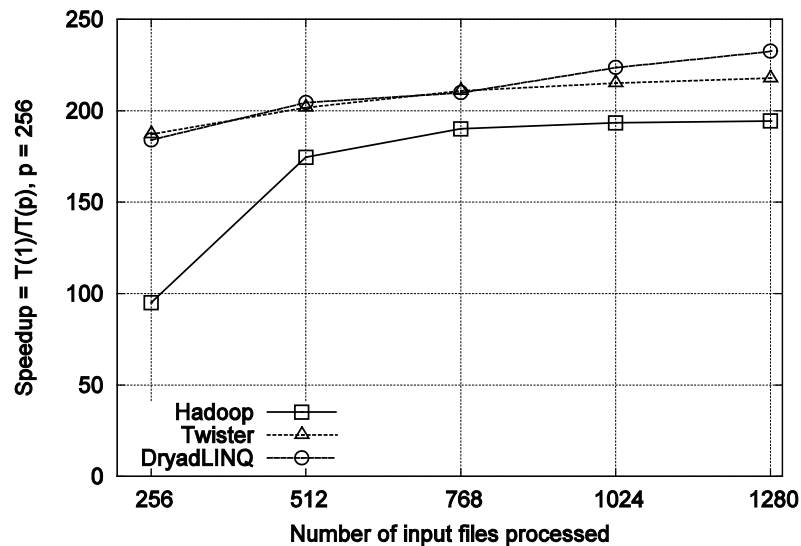


Figure 8. Speedups of different implementations of CAP3 application measured using 256 CPU cores of Cluster-III (Hadoop and Twister) and Cluster-IV (DryadLINQ).

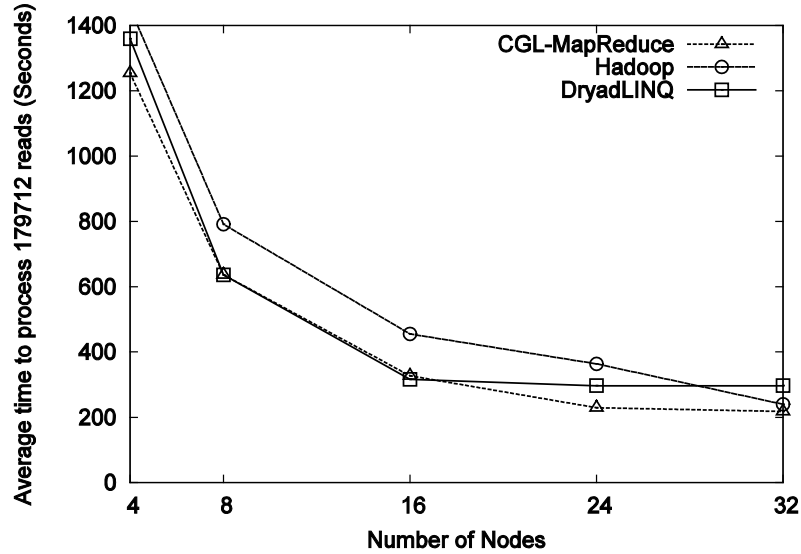


Figure 9. Scalability of different implementations of CAP3 application measured using 256 CPU cores of Cluster-III (Hadoop and Twister) and Cluster-IV (DryadLINQ).

6.3.5. Discussion

Although the main processing section of the CAP3 represents a “map-only” operation, all the above implementations used the corresponding parallel runtime to collect the outputs generated to a single computer. This reduces the overall speedup achievable for the parallel applications. However, the speedups in Figure 8 show that all three runtimes works equally well for CAP3. We also expect them to behave in the same way for similar applications with simple parallel topologies. Figure 9 indicates that for the data set we selected, the different implementations scale up to 16 computation nodes before producing diminishing returns. We expect better scalability characteristics for larger problem sizes. As we have explained in section 6.3.1 in the Hadoop implementation, we stored the input files in a network file system shared across the computation nodes rather than the HDFS. This prevented Hadoop implementation from exploiting data locality in scheduling computation task, and hence, produced lower speedups compared to the other two runtimes. This behavior may prevail in parallel applications developed using Hadoop

that uses legacy applications or unmodifiable executables in *map/reduce* tasks. Integrating Hadoop with a distributed file system that implements POSIX standard file system interfaces with the capability of providing data locality information would solve this issue.

6.4. High Energy Physics (HEP) Data Analysis

Most experiments in high energy physics produce large volumes of data. With the advancements in particle accelerators and detectors the physicist are getting closer and closer to uncovering some of the unsolved mysteries in the universe. Hedron colliders such as LHC (Large Hedron Collider) and its several detectors are expected to produce peta-bytes of data per year. Most of the data (the events generated as results of particle collisions) that exit from the initial stages of online filtration needs to be analyzed using some rigorous analysis functions to extract the information hidden in them. High Energy Physics group at Caltech provided us with such an analysis application with a large data set so that we can explore the new programming technologies to implement a parallel version of their application.

The input data for this application consists of a large number of binary files each taking roughly 33MB of disk space. The program is composed of two stages. First, the input files are processed by complex analysis function written using a language named ROOT [78], which is an interpreted C++ like language that can be used for rapid prototyping. The output of this stage of the computation is a histogram of identified features per given input file. During the next step, all the histograms produced in the previous stage is merged to form a single histogram representing the entire data analysis. This clearly resembles a perfect MapReduce application in which the *map* task perform the first stage of the computation while a set of reduce tasks can be used to merge the partial histograms to form next level of partial histograms. Finally, these partial histograms can be merged to produce the final histogram representing the entire analysis. This process is shown in Figure 10

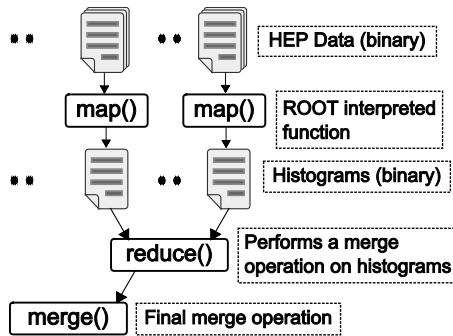


Figure 10. MapReduce for the HEP data analysis.

Although the characteristics of the application fits directly with the MapReduce model, its dependency to the ROOT analysis framework makes it challenging to implement using existing MapReduce runtimes. We were able to solve some of these challenges and were able to implement parallel applications for the above problem using Hadoop, DryadLINQ and the Twister runtimes.

6.4.1. Hadoop Implementation

In Hadoop implementation, the *map* and *reduce* tasks invoke ROOT interpretable analysis programs directly as executables. However, as shown in the CAP3 analysis, since the executables are expecting input data as files via the command line arguments, we could not use the HDFS effectively to store data. Therefore, we stored the input data in a high performance parallel file system (Lustre) and programmed the *map* tasks to download input files to local disk of the compute nodes during the runtime. In this approach, each *map* task first, copy an input file from the shared parallel file system to the local disk where it has been scheduled to run, and then executes the first stage of the computation that produces a histogram in the local file system. Next, the *map* task reads the resulting histogram back to the memory and “emits” (produces an output to the runtime) it to the Hadoop runtime so that Hadoop can send this data to an appropriate reduce task. The *reduce* tasks save the data objects received from the *map* tasks to files in the local file system and then execute another ROOT interpreted script which merges

histograms. Finally, the main program collects the partial histograms produced at the end of the each *reduce* task and combines them together to form a final output of the computation. As we will show in Figure 12 the online data movement introduced a significant performance penalty for the Hadoop implementation of this application.

6.4.2. DryadLINQ Implementation

MapReduce is a subset of the DAG based execution model that the Dryad/DryadLINQ support. We can simulate the three main phases of MapReduce; *map*, *shuffle and group by*, and *reduce*, using three queries in DryadLINQ. According to the authors of DryadLINQ MapReduce can be simulated in DryadLINQ as follows.

```
public static MapReduce( // returns set of Rs
    source, // set of Ts
    mapper, // function from T → Ms
    keySelector, // function from M → K
    reducer // function from (K,Ms) → Rs
) {
    var mapped = source.SelectMany(mapper);
    var groups = mapped.GroupBy(keySelector);
    return groups.SelectMany(reducer);
}
```

Figure 11. Simulating MapReduce using DryadLINQ.

In the above code segment, the first “SelectMany” query applies a function - *mapper* - to all inputs which produces one or more (*key,value*) pairs at each invocation. These pairs are then grouped according to their *keys* by a *GroupBy* operation which accepts a *KeySelector* function that determines the grouping behavior. Finally, the *reduce* function is applied to each group using another “SelectMany” query. According to the MapReduce model, the *reduce* function produces only a single output per each group of values corresponding to a particular *key*. Therefore, the last query in the above code segment can be replaced by a “Select” operation as well. (Note: In DryadLINQ, the “Select” operation applies a user defined function to an input and returns a

single output whereas “SelectMany” performs the same functionality but can return one or more outputs at each invocation).

For DryadLINQ implementation, we first distribute input data to local disks of the compute nodes and created a partitioned-file comprising of meta-data about the data distribution using a similar approach described in section 6.3.2. The first phase of the program produces a collection of vertices in DryadLINQ’s DAG based execution flow. Each of these vertices executes a ROOT interpreted function as a separate executable, which processes the input data files passed in as command line arguments. Next, each vertex reads the output histogram and returns it to the DryadLINQ runtime as an object with an “id” field generated using a random number that lies in the range of zero and the number of *reduce* tasks specified by the user. During the next step of the program, a “GroupBy” operation is performed on these data objects grouping them according to their id field. A “Select” operation is used next to combine the grouped histograms together to produce new partial histograms. Finally, the main program collects these partial histograms and combines them together to produce the final histogram.

6.4.3. Twister Implementation

The Twister implementation of the HEP analysis uses *map* and *reduce* tasks similar to that of Hadoop implementation. However, there are few notable differences in the two implementations. Unlike in Hadoop, in the Twister version, the input data is distributed to the local disks of the compute nodes and the map tasks directly access them as files. This eliminates the requirements of using a separate parallel file system to hold input data or to use two levels of indirections as in DryadLINQ. Furthermore, in Twister the histograms produced after the map stage of the computation is directly transferred to the reduce tasks via the publish/subscribe messaging infrastructure where as in both Hadoop and DryadLINQA the data transfer happens via file systems. In Hadoop, reduce outputs are written to the distributed file system (HDFS) and a

combine operation on these outputs needs to be done manually. Twister’s *combine* phase represents another level of reduce operation which can be used to combine the results of the reduce stage of the computation. Therefore the final merging of histograms is handled in the *combine* stage of Twister.

6.4.4. Performance Evaluation

We measure the performance of this application using two computation clusters each with 256 CPU cores. The performance is measured by increasing the input sizes up to 1TB, and the results are shown in Figure 12.

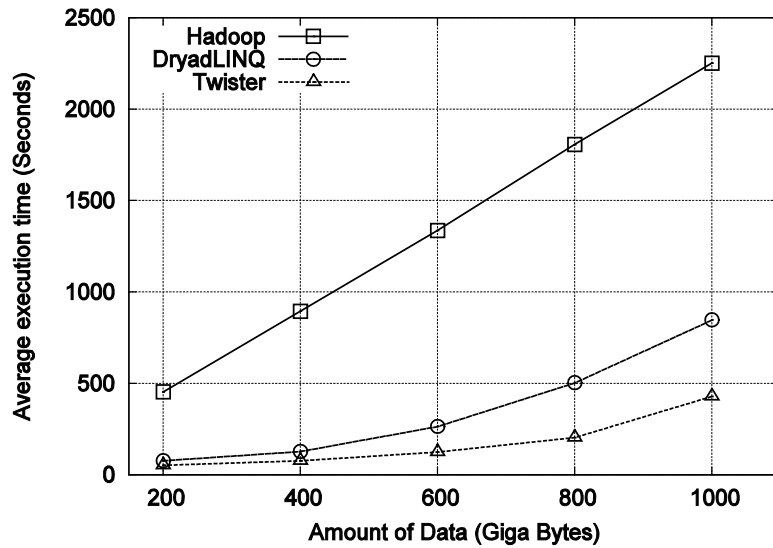


Figure 12. Performance of different implementations of HEP data analysis applications measured using 256 CPU cores of Cluster-III (Hadoop and Twister) and Cluster-IV (DryadLINQ).

6.4.5. Discussion

The results in Figure 12 highlight that the Hadoop implementation has a considerable overhead compared to the DryadLINQ and the Twister implementations. This is mainly due to the differences in storage mechanisms used in these frameworks. DryadLINQ and Twister access the

input from local disks where the data is partitioned and distributed before the computation. As we have explained, the Hadoop implementation moves input data from IU Data Capacitor – a high performance parallel file system based on Lustre file system – to the local disks during the execution time. The dynamic data movement in the Hadoop implementation incurred a considerable overhead to the overall computation. In contrast, the ability to read input from the local disks gives significant performance improvements to both DryadLINQ and Twister implementations.

Histogramming is a natural match to the MapReduce programming model and such applications can be implemented in similar fashion as above. The moving-computation-data support in MapReduce improves performance of many data intensive applications. We expect Hadoop to show similar performance characteristics to applications where the data can be utilized from its distributed file system –HDFS.

6.5. Pairwise Similarity Calculation

Calculating similarity or dissimilarity between each element of a data set with each element in another data set is a common problem and is generally known as an All-pairs[79] problem. This section discusses one such application in gene sequencing. The application we have selected calculates the Smith Waterman Gotoh(SW-G)[80] distance (say δ_{ij} –distance between sequence i and sequence j) between each pair of sequences in a given gene sequence collection.

6.5.1. Introduction to Smith-Waterman-Gotoh (SWG)

Smith-Waterman [81] is a widely used local sequence alignment algorithm for determining similar regions between two DNA or protein sequences. In our studies we use Smith-Waterman algorithm with Gotoh's improvement for Alu sequencing. The Alu clustering problem [82] is one of the most challenging problems for sequencing clustering because Alus represent the largest

repeat families in human genome. As in metagenomics, this problem scales like $O(N^2)$ as given a set of sequences we need to compute the similarity between all possible pairs of sequences.

We adopted the following algorithm to map this application to the MapReduce domain. To clarify our algorithm, let's consider an example where N gene sequences produces a pairwise distance matrix of size $N \times N$. We decompose the overall computation by considering the resultant matrix and group the overall computation into a block matrix of size $D \times D$. Due to the similarities of distances δ_{ij} and δ_{ji} we only calculate the distances in the blocks of the upper triangle of the block matrix as shown in Figure 13. Each of these blocks is assigned to a *map* task which calculates SW-G distances for each pair of sequences within that block. Moreover, each *map* task that calculates a non-diagonal block produces a transpose block of the calculated distances as well. This allows us to construct the full $N \times N$ distance matrix by computing only the half of the actual distances. The row number of a given block is used as the input key for the *reduce* tasks, which simply collect the data blocks corresponding to a row and write to output files after organizing them in their correct order. At the end of the computation all the blocks corresponding to a single row block will be written to a file by the *reduce* tasks.

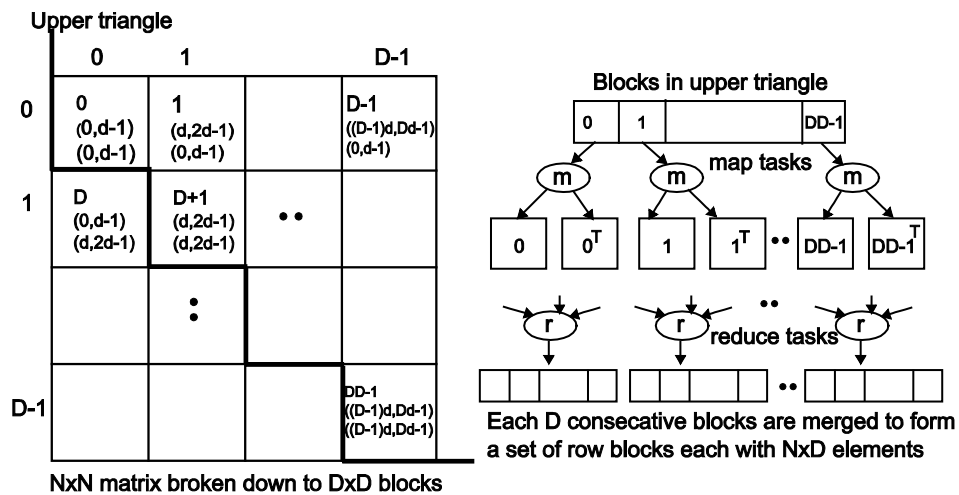


Figure 13. MapReduce algorithm for SW-G distance calculation program

6.5.2. Hadoop Implementation

We used JAligner[83] library to calculate SW-G distances in the Hadoop implementation of the above algorithm, where it is used in the *map* task to compute a block of distances. The block size (D) can be specified via an argument to the program. Also, it needs to be specified in such a way that there will be much more *map* tasks than the number of processing elements in the computation cluster. This way the Apache Hadoop can schedule *map* tasks as a pipeline, which results a global load balancing of the application. The input data is distributed to the worker nodes through the Hadoop distributed cache, which makes them available in the local disk of each compute node.

6.5.3. DryadLINQ Implementation

The DryadLINQ version of the above application is developed by simulating MapReduce programming model using DryadLINQ queries as explained in the HEP data analysis application. For this implementation we have used NAligner library (C# version of the JAligner library) to calculate Smith Waterman distances. First, the main program computes the block boundaries and assigns them to vertices. DryadLINQ replicates the input sequence file to every vertex automatically. This approach is possible, because the size of the sequence file, even with large number of sequences, is not considerably large when compared to the size of the output matrix. Each vertex computes Smith Waterman distances in a given block and produces two blocks as output. The output blocks are indexed based on their corresponding row numbers where they will fit in to the resultant distance matrix. Next a “*GroupBy*” operation is performed on all the output blocks to group them according to their row numbers, followed by another stage of “*Select*” operation to combine blocks in a particular row to a single output file. The

overall computation produces D (refer to Figure 13) output files corresponding to the D row blocks in the resultant matrix.

6.5.4. Twister Implementation

We also developed a Twister counterpart of the above algorithm by adopting a similar approach to the Hadoop implementation. Apart from minor differences in the programs, the *map* and *reduce* functions for the Twister implementation are very similar to that of Hadoop. In all three implementations we expect the input sequence file to be available in all the compute nodes while we use block indices and block boundaries as input (*key,value*) for map tasks. In Hadoop implementation the (*key,value*s) are assigned to *maps* by writing them to individual files in HDFS, whereas in Twister the main program can directly send them to *map* tasks using the *runMapReduce(KeyValue[])* API call. Twister also uses JAligner as the alignment engine.

6.5.5. Performance Evaluations

We identified samples of human and Chimpanzee Alu gene sequences using Repeatmasker[84] with Rebase Update [85] and produced a data set of 50000 sequences by replicating a random sample of 10000 sequences from the original data. We used this data to measure parallel performance of DryadLINQ, Hadoop, and Twister runtimes. Figure 14 shows the parallel efficiency (η) of each runtime under varying data sizes.

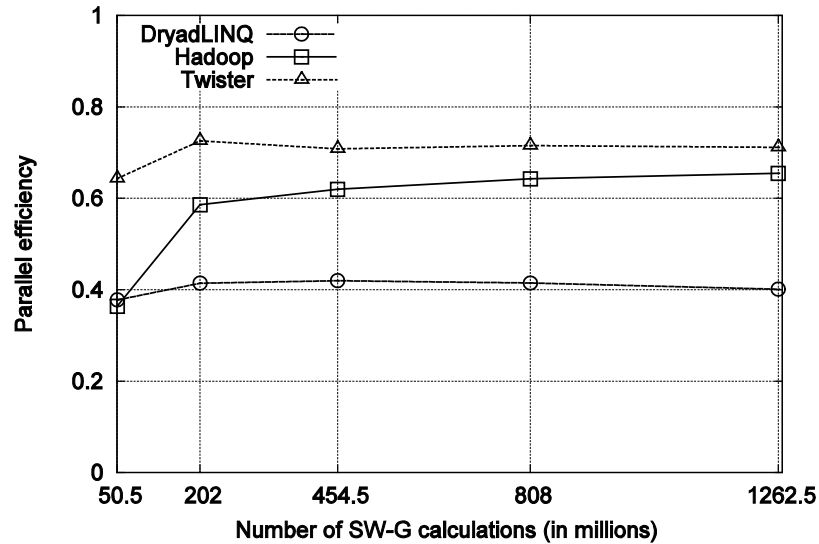


Figure 14. Parallel Efficiency of the different parallel runtimes for the SW-G program (Using 744 CPU cores in Cluster-I).

6.5.6. Discussion

Due to the sheer volume of SW-G comparisons, we did not measure the sequential execution times of the above programs. Instead, we estimated serial running time by simply summing up the times spent on each *map* and *reduce* tasks. The results clearly show that all three runtimes achieve maximum efficiencies and maintains them with the increase of data. Although the absolute efficiency is not correctly reflected by the estimated serial time, it provides a valuable base point for our comparisons. Since this is a typical MapReduce computation, we expect all runtimes to achieve higher absolute efficiencies. Twister outperforms Hadoop, because of its faster data communication mechanism, and the lower overhead in the static task scheduling. Moreover, in Hadoop each *map/reduce* task is executed as a separate process (a Java Virtual Machine - JVM) whereas in Twister they are executed using threads. The Lower efficiency in DryadLINQ was mainly due to an inefficient task scheduling mechanism used in the initial academic release[86].

To evaluate the scalability of the Twister runtime further, we performed another benchmark using 1632 CPU cores of Cluster-II. In this evaluation, the Twister runtime is configured to use a daemon in each CPU core simulating a cluster of 1632 single core nodes. The efficiencies calculated for this evolution shows a value of 79% indicating that the runtime is scalable to such number of nodes. These results also prove that the Twister is capable of running typical MapReduce computations although we have added enhancements focusing on iterative MapReduce computations.

6.6. K-Means Clustering

K-Means clustering [71] is a well-known data clustering algorithm that performs an iterative computation to find a given number of cluster centers in a given input data set starting from a random set of cluster centers. In each iteration, the algorithm computes the distance (typically Euclidean distance) between the current cluster centers and all the input data points, and assigns each data point to a nearest cluster center. Then, it computes the new cluster centers by calculating the average distances of points assigned to a given cluster center. To check the convergence, the algorithm performs a comparison between the cluster centers produced during the n^{th} iteration and the cluster centers produced from $(n-1)^{\text{th}}$ iteration. If this difference is greater than a given threshold the iterations will continue.

In this algorithm, the major computation is the calculation of distances between the cluster centers and the data points. Therefore, a parallel algorithm can be developed by performing this computation in parallel. A MapReduce version of the above algorithm is shown below.

K-means Clustering Algorithm for MapReduce

Do

Broadcast C_n

[Perform in parallel] –the map() operation

for each V_i

for each $C_{n,j}$

$$D_{ij} \leq \text{Euclidian}(V_i, C_{n,j})$$

Assign point V_i to $C_{n,j}$ with minimum D_{ij}

for each $C_{n,j}$

$$C_{n,j} \leq \sum_i^{K_j} (V_i)$$

Emit ($j, [C_{n,j}, K_j]$)

[Perform Sequentially] –the reduce() operation

Collect all $[C_{n,j}, K_j]$

for each $C_{n,j}$

$$C_{n,j} \leq \sum_i^m (C_{n,j,m})$$

$$K_j \leq \sum_i^m (K_{j,m})$$

$$C_{n,j} = C_{n,j}/K_j$$

Calculate new cluster centers C_{n+1}

$$\text{Diff} \leq \text{Euclidian}(C_n, C_{n+1})$$

while ($\text{Diff} < \text{THRESHOLD}$)

Assume that the input is already partitioned and available in the compute nodes. In this algorithm, V_i refers to the i^{th} vector, $C_{n,j}$ refers to the j^{th} cluster center in n^{th} iteration, D_{ij} refers to the Euclidian distance between i^{th} vector and j^{th} cluster center, and K is the number of cluster centers. The number of *map* tasks is defined by m . We implement the above algorithm using four parallel runtimes, Hadoop, DryadLINQ, Twister, and MPI.

6.6.1. Hadoop Implementation

In Hadoop implementation, depending on the number of *map* tasks to be used, the main program partitions the input data into a collection of files and stores them in HDFS. Each *map* task reads a data partition from HDFS and the current cluster centers from Hadoop's distributed cache. Then the *map* task assigns points to cluster centers and calculates a sum of points for each cluster center. Finally, it "emits" these partial sums along with the number of points in each cluster center to *reduce* tasks. The *reduce* task collects this information, combines the partial cluster centers, produces the new cluster centers, and writes them to a file in HDFS. The main program reads the new cluster centers from HDFS and calculates the difference between the new cluster centers and the previous cluster centers, and determines whether to proceed to with new iteration.

As we have discussed in Chapter 2, Hadoop considers each iteration as a new MapReduce computation, and hence, the input data partitions are read from files in every iteration. Also the communication between *map* and *reduce* tasks, and the *reduce* task and the main program happen via some form of file system. These communication paths add higher overheads compared to an approach of sending them directly via network connections.

6.6.2. DryadLINQ Implementation

DryadLINQ implementation uses an *Apply* operation to perform the map phase of the computation, in which the data vectors are assigned to cluster centers. The apply operation works on a collection of inputs (in this case, a set of input vectors) and produces a single output. Similar to a *map* task in Hadoop implementation, each vertex outputs an object comprising of all the partial cluster centers and the number of points assigned to each cluster center. Another *Apply* operation, which runs sequentially, calculates the new cluster centers for the n^{th} iteration. Finally, the main program calculates the distance between the previous cluster centers and the

new cluster centers using a *Join* operation to compute the Euclidian distance between the corresponding cluster centers. DryadLINQ supports a feature known as “loop unrolling” which can be used to build a single DryadLINQ query to represent multiple iterations of some set of query operations. Deferred query evaluation is a feature of LINQ, whereby a query is not evaluated until the program accesses the query results. Thus, in the K-means program, we accumulate the computations performed in several iterations (we used 4 as our unrolling factor) into one query and only “materialize” the value of the new cluster centers in every 4th iteration.

6.6.3. Twister Implementation

Twister runtime is optimized to handle iterative MapReduce computations. As in Hadoop, the input data is first partitioned into a collection of files and then distributed to the local disks of the compute nodes. Following this, a “partitioned file” is created containing the meta-data of the file partitions. Twister provides a tool to support these operations. *Map* tasks are then configured using “configureMaps()” method passing the above partitioned file. Twister also supports a broadcast style operation to start MapReduce computations - “mapReduceBCast()”. The program uses this method to send the current cluster centers to all map tasks in each iteration. Unlike Hadoop, Twister’s *combine* phase collects the outputs produce after the *reduce* stage to a single location accessible to the main program and the data transfer between phases happens via TCP based connections. This approach minimizes the overhead in overall data transfers, which starts from the main program and return back to it after following *map* and *reduce* phases.

6.6.4. MPI Implementation

We implemented the above parallel K-Means algorithm using MPI as well. This is mainly to evaluate the performance of the MapReduce runtimes comparing to MPI. We use a data portioning scheme similar to MapReduce but kept all the data partitions in a network file system

of the computation cluster. MPI processes are inherently long running, so the input data is loaded only once during the execution.

6.6.5. Performance Evaluation

To evaluate the performance of the K-means clustering applications, we selected a data set comprising of 2D vector points distributed around a known set of cluster centers. Then we use the K-Means clustering implementations to identify those clusters and verify the results using the earlier known cluster centers. To compare performances, we used only a fixed number of iterations in each implementation. Figure 15 below shows the performance of four implementations of K-Means clustering algorithm.

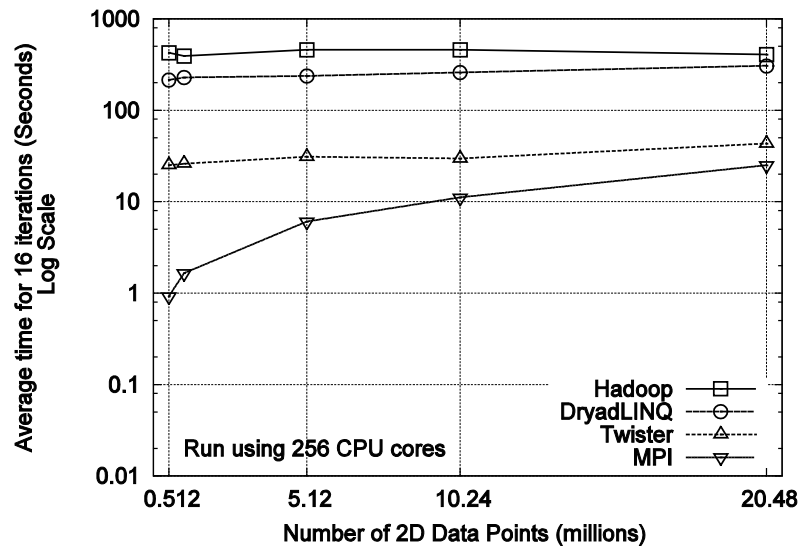


Figure 15. Performance of different implementations of K-Means clustering algorithm performed using 256 CPU cores of Cluster-III (Hadoop, Twister, and MPI) and Cluster-IV (DryadLINQ)

6.6.6. Discussion

Although we used a fixed number of iterations, we changed the number of data points from 500 thousand to 20 millions. In K-means clustering, increase in the number of data points increases

the amount of computation. However, it was not sufficient to ameliorate the overheads introduced by Hadoop and DryadLINQ runtimes. As a result, the graph in Figure 15 mainly shows the overhead of the different runtimes. The use of file system based communication mechanisms and the loading of static input data at each iteration in Hadoop and in each unrolled loop in DryadLINQ resulted considerably higher overheads in these runtimes compared to Twister and MPI. Iterative applications that perform more complex computations or access larger volumes of data may produce better results for Hadoop and DryadLINQ, as the higher overheads induced by these runtimes becomes relatively less significant. However, the aforementioned inefficiencies of these runtimes produce considerable overheads making them less useful for this class of applications.

A straightforward way to implement the above algorithm in MapReduce and DryadLINQ is by using the *map* phase to assign points to cluster centers and send (*cluster center, point*) pairs to the *reduce* stage. This approach results intermediate data transfers in the orders of input data and produce considerable overheads when every iteration performs the same data transfer. One can use a “combine” operation in Hadoop – a reduce operation that runs locally just after the *map* tasks to accumulate results before sending them to *reduce* tasks. Similarly in DryadLINQ one can re-partition data after an assignment and performs a local *combine* operation using “Apply” query in DryadLINQ. In all our implementations, we merged this *combine* stage to the *map* task so that it outputs only the partial cluster centers and their counts as the output after operating on a collection of points. This minimizes the additional overhead in data transferring and scheduling of tasks by the runtimes.

6.7. PageRank

PageRank algorithm calculates numerical value to each web page in World Wide Web, which reflects the probability that the random surfer will access that page. The process of PageRank can

be understood as a Markov Chain which needs recursive calculation to converge. An iteration of the algorithm calculates the new access probability for each web page based on values calculated in the previous iteration. The iterating will stop when the difference (δ) is less than a predefined threshold, where δ is the vector distance between the page access probabilities in N^{th} iteration and those in $(N-1)^{\text{th}}$ iteration.

There already exist many published work optimizing PageRank algorithm, like some of them accelerate computation by exploring the block structure of hyperlinks[87, 88]. In this research, we did not create any new PageRank algorithm, but implemented the most general PageRank algorithm [1] in MapReduce programming model. The web graph is stored as an adjacency matrix (AM) and is partitioned to use as static data in *map* tasks. The variable input for a *map* task is the initial page rank score. The output of *reduce* tasks is the current PageRanks which will be used by the *map* tasks in the next iteration.

6.7.1. Hadoop Implementation

We implemented the above algorithm by using Hadoop similar to the approach we used in K-means clustering. The adjacency matrix is partitioned and stored as a collection of files in HDFS, so that each file is processed by a *map* task. The current PageRank scores are transferred using Hadoop's distributed cache. The *map* tasks update the ranks in its data partition using the current PageRank and send the updated ranks to a collection of reduce tasks, which then compute partial set of PageRanks. Finally the main program calculates the next set of page ranks for the input. The iterations continue until the difference between the current PageRanks and the previous PageRanks reach a certain threshold.

6.7.2. Twister Implementation

By leveraging the features of Twister, we were able to implement PageRank in an efficient manner. Some of these improvements are: (i) the partial adjacency matrices are only loaded once per map task because they can be configured as static data in Twister; (ii) current PageRanks are directly sent to the map tasks using the broadcast feature. Further optimizations that are independent of Twister include: (i) increasing the *map* task granularity by wrapping certain number of URLs entries together and (ii) merging all the tangling nodes as one node to save the communication and computation cost.

6.7.3. Performance Evaluation

We investigated the performance of Hadoop and Twister implementations of the PageRank using ClueWeb data set [89] collected in January 2009. We built the adjacency matrix using this data set and tested the page rank application using 32 computer nodes of Cluster-II. Table 7 summarizes the characteristic of three ClueWeb data sets we used in our tests.

Table 7. Characteristics of data sets (B = Billions, AM = Adjacency Matrix)

ClueWeb data set	CWDS1	CWDS3	CWDS5
Number of AM partitions	4000	2400	800
Number of web pages	49.5M	31.2M	11.7M
Number of links	1.40B	0.83B	0.27B
Average out-degree	28.3	26.8	22.9

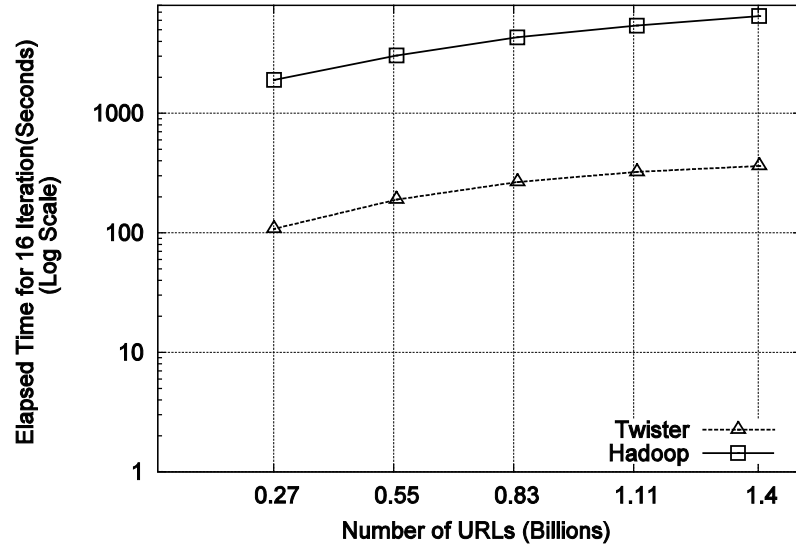


Figure 16. Elapsed time for 16 iterations of the Hadoop and Twister PageRank implementations (Using 256 CPU cores in Cluster-II).

6.7.4. Discussion

Figure 16 shows the performance of PageRank applications under different data sizes. Both Hadoop and Twister show similar performance characteristics with the increasing data sizes. However, it also reveals that Hadoop takes considerably longer time than Twister (Notice the log scale in y axis). We also calculated the efficiency of the PageRank application using formula (4) above with p_1 and p_2 times taken from runs on 128 and 256 CPU cores respectively for the CWDS3 data set. The results revealed that the Twister version of the application can maintain above 80% efficiency at 256 CPU cores. As we have mentioned above, Twister's support for long running *map/reduce* tasks gives it a considerable performance advantage over Hadoop, as the static data is only loaded once for the computation. Furthermore, the direct transfer of intermediate data and the current PageRanks enables it to perform better as well. Although Twister broadcasts current PageRanks to all *map* tasks using the publish/subscribe broker network, the actual data movement only occurs from the broker network to each Twister daemon

(typically one per compute node), which internally shares this data among the map tasks it owns without copying. This approach improves the overall performance of similar algorithms significantly.

6.8. Multi-Dimensional Scaling (MDS) Application

Multidimensional scaling (MDS) is a general term used for the techniques to configure low dimensional mappings of given high-dimensional data with respect to the pairwise proximity information, while the pairwise Euclidean distance within the target dimension of each pair is approximated to the corresponding original proximity value. In other words, it is a non-linear optimization problem to find low-dimensional configuration which minimizes the objective function, called STRESS[90] or SSTRESS [91].

Among many MDS solutions, we are using a well-known expectation maximization (EM) like method called SMACOF (Scaling by Majorizing of COmplicated Function)[74]. SMACOF is based on iterative majorization approach and is calculated by iterative matrix multiplication. For the stop condition, SMACOF algorithm measures the STRESS value of current mapping and compares it to the STRESS value of the previous mapping result. If the difference of STRESS values between previous one and the current one is smaller than threshold value, then it stops iteration. For details of the SMACOF algorithm, please refer to[92].

6.8.1. Twister Implementation

At a very high level the computation performed in SMACOF algorithm can be viewed as a set of matrix and vector multiplications. More precisely, in the n^{th} iteration the current lower dimensional mapping X_n is derived using the formula:

$$X_n = D \times B \times X_{n-1}$$

In this formula D is the distance matrix given as the input to the algorithm, B represents a derived matrix similar in size to D , and X_{n-1} is the mapping to the lower dimension found in the previous iteration. To determine the condition for proceeding with iterations, the algorithm also computes a STRESS value based on the X_n . In the above equation, the D and B are square matrices while the X_n is a vector (2D or 3D depending on the dimension the high dimensional data is reduced). This feature can be used to convert the above equation to a matrix-vector multiplication instead of matrix-matrix multiplication that minimizes the computational complexity of the entire algorithm. Therefore, the Twister implementation calculates X_n using two matrix-vector multiplications. The following pseudo code segment shows the Twister version of the MDS program.

Pseudo Code: Multi Dimensional Scaling using MapReduce

```
//Load static data to map tasks
configureBXMaps()
configureDCMaps()
configureSTRESSMaps()

//Start main iteration
while(diff<THREASHOLD){
    Cn=calculateBXMapReduce(Xn-1)
    Xn=calculateDCMapReduce(Cn)
    STRESSn=calculateSRESSMapReduce(Xn)
    diff=STRESSn-STRESSn-1
}
```

In the above pseudo code, “calculateBXMapReduce(X_{n-1})” calculates the vector C_n resulting from matrix-vector multiplication $B \times X_{n-1}$ in the above formula. The “calculateDCMapReduce(C_n)” calculates the matrix-vector multiplication $D \times C_n$. This application

demonstrates the programming model we envisioned in Twister in which the MapReduce is used as a programming construct to parallelize sections of iterative applications. Furthermore, it also shows how complex iterative applications can be developed using Twister runtime as well.

6.8.2. Performance Analysis

To evaluate the performance of our implementation, we used a data set comprising of 35339 gene sequences that produce a 1.24 billion pair-wise distances in matrix D. Estimating the serial running time for MDS application is not straightforward and hence we calculated the parallel efficiency using the formula (4). The outcome of this benchmark is shown in Figure 17.

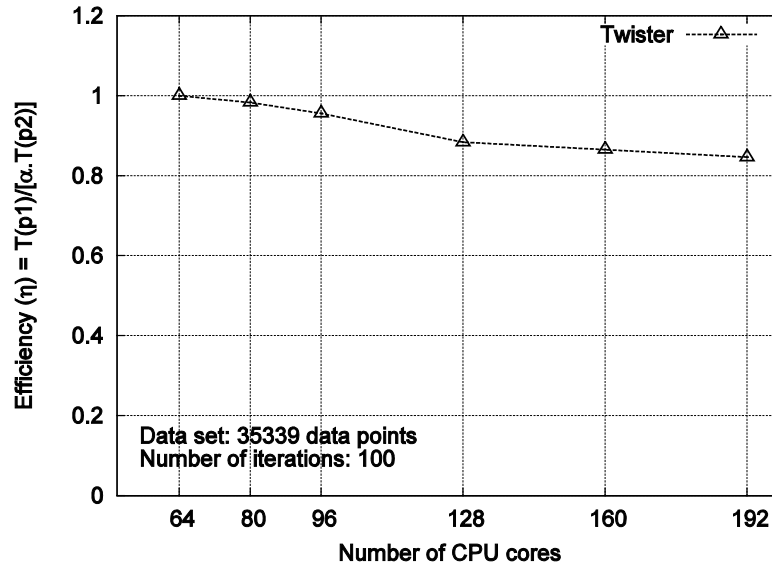


Figure 17. Efficiency of the MDS application (in Cluster-II).

6.8.3. Discussion

For the selected data set, Twister maintains higher efficiencies (>80%) for considerable number of CPU cores. With large data, we expect it to maintain similar efficiencies for even higher number of CPU cores. As we have shown in [86, 93-95] both Hadoop and DryadLINQ showed extremely high overheads for iterative applications such as K-Means clustering or matrix multiplication.

The MDS uses three MapReduce computations in a single iteration involving two matrix- vector multiplications and one STRESS calculation. Thus we expect both Hadoop and DryadLINQ to be highly inefficient for this application and hence did not implement MDS using those runtimes.

6.9. Matrix Multiplication

In this section, we discuss two parallel matrix multiplication algorithms that can be used with Twister. For simplicity of the explanation, we assume that the matrices have square dimensions. Let's consider a matrix multiplication where A and B matrices produce a result matrix of C. We also assume that the multiplication uses n parallel processes.

6.9.1. Row-Column Decomposition Approach

In this algorithm, the first matrix (A) is partitioned into a collection of row blocks. The height of a row block is determined by the dimension of the matrix (N) and the number of iterations (r). The second matrix (B) is partitioned to a set of column blocks. In MapReduce implementation, each *map* task holds a column block of matrix B, and in each iteration it receives a row block of matrix A from the main program. During the i^{th} iteration, j^{th} *map* task calculates the $(i,j)^{th}$ block of matrix C, while the reduce task collects these output blocks and merges them to form a row block of matrix C. For this algorithm, we used the Twister's long running *map/reduce* tasks with configure option, so that the column blocks of matrix B is loaded only once for the entire computation. Figure 18 illustrates this approach.

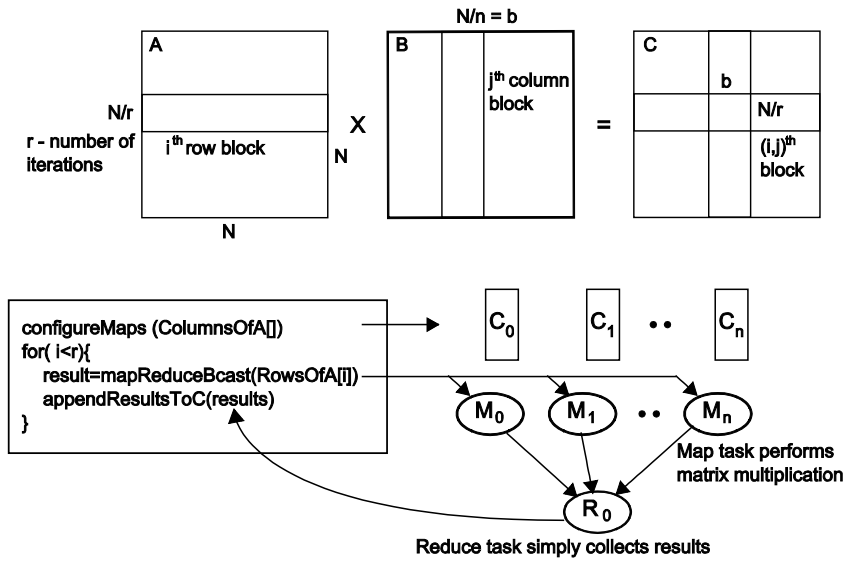


Figure 18. Matrix multiplication Row-Column decomposition (top). Twister MapReduce implementation (bottom).

Since the amount of communication determines the scalability characteristics of an algorithm, in the following table we list the amount of communication performed in each step of the above algorithm.

Table 8. Breakdown of the amount of communication in various stages of the Row-column based matrix multiplication algorithm.

Operation	Amount of communication	Total for r iterations
<code>ConfigureMaps()</code>	$(N*b)*n = N^2$	N^2
<code>mapReduceBcast ()</code>	$(N/r)*N*n = N^2n/r$	N^2n
In between map and reduce	$(b*N/r)*n = N^2/r$	N^2
Collecting results	$(b*N)*n=N^2$	N^2

Total communication = $N^2n+3N^2 = O(N^2n)$

6.9.2. Fox Algorithm for Matrix Multiplication

The Fox algorithm[13] uses 2D block based approach with a square processes mesh. Similar to the above analysis let's also assume that the total number of processes available is n . This leads to a processes mesh of $q \times q$ where $q = \sqrt{n}$. Although the process mesh is a logical arrangement, parallel runtimes such as MPI provide optimized communication constructs for processes arranged in 2D meshes.

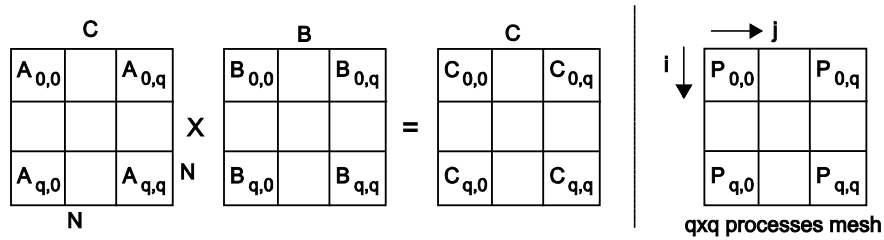


Figure 19. 2D block decomposition in Fox algorithm and the process mesh.

In the Fox algorithm, each process holds a block of matrix A and a block of matrix B and computes a block of matrix C. In k^{th} iteration every process executes the following communication and computation operations.

1. The process that holds $A(i, (i+k) \bmod q)$ broadcasts it to all the process in the row i
2. All the processes in row i receive the above element (say D)
3. Every process calculates $C(i,j) = C(i,j) + D \times B(i,j)$
4. Every process sends the block $B(i,j)$ to the process that holds $B((i+1) \bmod q, j)$

In the previous algorithm, each iteration completes a one row block of the resultant matrix C. In contrast, in this algorithm, each process keeps accumulating the final value for a block of matrix C throughout the computation until it terminates in q iterations.

Similar to the row-column approach, we can also calculate the amount of communication the Fox algorithm performs as follows. To make the analysis similar to the previous algorithm, we use $q = \sqrt{n}$ property in the equations.

Table 9. Breakdown of the amount of communication in various stages of the Fox algorithm.

Operation	Amount of communication	Total for $q = \sqrt{n}$ iterations
Initial data distribution	$2 \cdot N^2$	$2 \cdot N^2$
Broadcast	$(N/q) \cdot (N/q) \cdot q \cdot q = N^2$	$N^2 \sqrt{n}$
Shift operation	$(N/q) \cdot (N/q) \cdot q \cdot q = N^2$	$N^2 \sqrt{n}$
Collecting results	N^2	N^2

$$\text{Total communication} = 2N^2\sqrt{n} + 3N^2 = O(N^2\sqrt{n})$$

The above analysis shows that the Fox algorithm performs far less communication than the row-column based decomposition approach discussed earlier. Further, in the Fox algorithm, each process only requires memory to hold three blocks of matrices while the previous approach requires memory for more than two row blocks at a time. Therefore it is interesting to see if one can implement Fox algorithm using MapReduce.

6.9.3. Fox Algorithm using Twister's Extended MapReduce

We have come up with a MapReduce algorithm that can simulate Fox matrix multiplication. Unlike MPI which supports mesh configuration of processes, MapReduce provides only *map* followed by *reduce* communication pattern. However, we can simulate a square arrangement of processes using MapReduce as follows.

Let's assume that we use n *map* tasks and n *reduce* tasks and each type is arranged to form a square mesh with dimension $q = \sqrt{n}$ as shown in Figure 20. Typically, in MapReduce the keys generated as the *map* outputs are matched to different *reduce* tasks using a hash function. Here, we use an identify function as the "Key Shuffler". Also we use integer keys between 1 and n as *map* output keys, so that a *map* output can be send to a particular *reduce* task depending on the

output key. For example, if *map* task 2 needs to send a message to *reduce* task 5 it can do so by producing a $(key,value)$ pair with key equal to 5 and the message as the value.

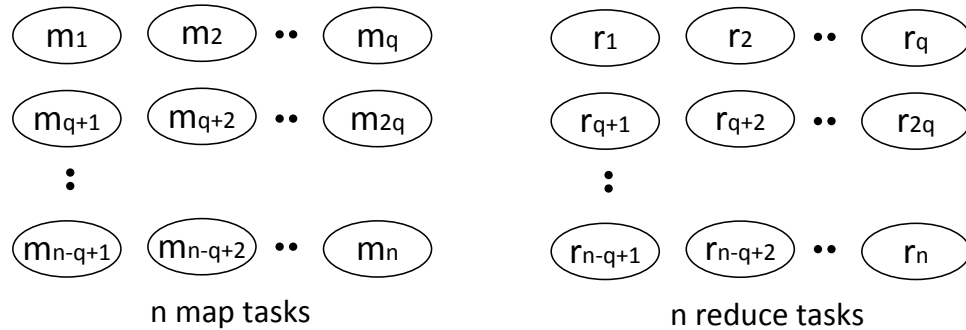


Figure 20. Virtual topology of map and reduce tasks arranged in a square matrix of size $q \times q$

As explained before, in Fox matrix multiplication, at some point of execution each process needs to send its block of matrix A to all the processes (row wise broadcast) in the same row and sends its block of matrix B to the process right above in the process mesh (shift). For this algorithm, we assume n parallel processes executed as *map* and *reduce* computations in two phases of the MapReduce computation. Although we use $2n$ tasks, at a given time, only one set of tasks (map or reduce) will be executed, therefore we can safely assume that there are only n processes. Each *map* task holds a block of matrix A and a block of matrix B while each *reduce* task computes a block of matrix C. In each iteration, the main program, *map*, and *reduce* tasks performs the following operations.

1. Main program sends the iteration number k to all map tasks
2. The map tasks that meet the following condition send its A block (say A_b) to a set of reduce tasks
 - a. Condition for map $\Rightarrow ((\text{mapNo} \div q) + k) \bmod q == \text{mapNo} \bmod q$
 - b. Selected reduce tasks $\Rightarrow ((\text{mapNo} \div q) * q) \text{ to } ((\text{mapNo} \div q) * q + q)$
3. Each map task sends its B block (say B_b) to a reduce task that satisfy the following condition
 - a. Reduce key $\Rightarrow ((q-k)*q + \text{mapNo}) \bmod (q*q)$
4. Each reduce task performs the following computation
 - a. $C_i = C_i + A_b \times B_i \quad (0 < i < n)$
 - b. If (last iteration) send C_i to the main program

The communication pattern for the **second iteration** of this algorithm is shown in Figure 21 using a 3x3 processes mesh.

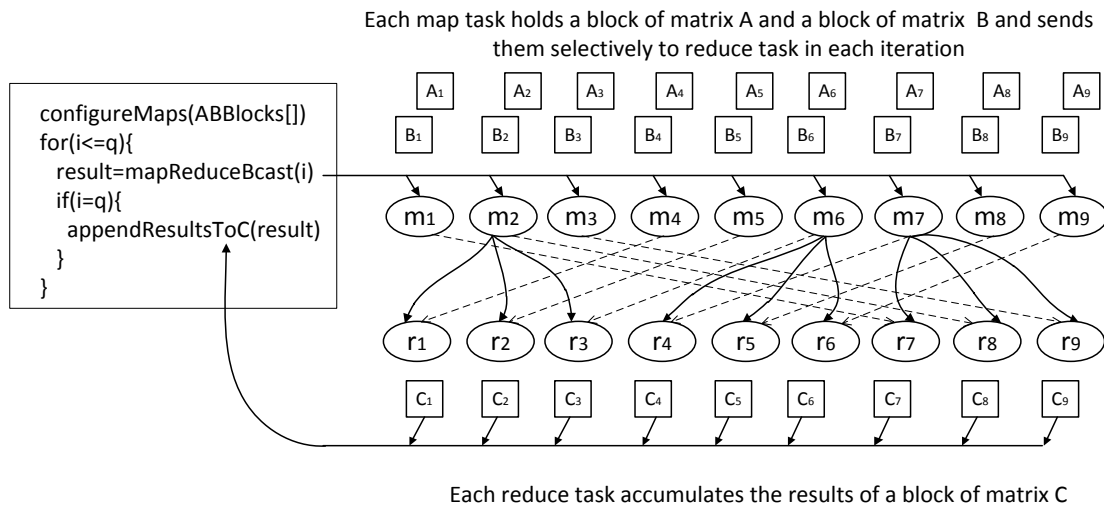


Figure 21. Communication pattern of the second iteration of the Fox - MapReduce algorithm shown using 3x3 processes mesh. Thick arrows between map and reduce tasks show the broadcast operation while dash arrows show the shift operation.

In this algorithm, we assume that the reduce tasks are long running so that they can accumulate the result of a blocks of matrix C, which makes the *reduce* tasks no longer side effect free. That is, we will not be able to recover these states with the current fault tolerance strategy of Twister. However, it is possible to recover the state of these types of computations by saving state of each *reduce* task to a distributed file system in every X number of iterations, where X defines the

number of roll-back iterations necessary in an event of a failure. We will discuss this hybrid approach to fault tolerance in the future work section of this thesis. Apart from the above, the row-wise broadcast is implemented as a selective broadcast operation using Twister’s row-wise broadcast option for a logical mesh of *reduce* tasks, which utilizes the underneath pub-sub infrastructure to handle the broadcast operation. Furthermore, for larger blocks Twister automatically uses direct TCP channels between daemons, which eliminates the loading of the broker network with large data transfers. Table 10 highlights the amount of communication performed in each step of the above algorithm.

Table 10. Breakdown of the amount of communication in various stages of the Twister MapReduce version of Fox algorithm.

Operation	Amount of communication	Total for $q = \sqrt{n}$ iterations
configureMaps	$2 * N^2$	$2 * N^2$
Selective Broadcast	$(N/q) * (N/q) * q * q = N^2$	$N^2 \sqrt{n}$
Shift operation	N^2	$N^2 \sqrt{n}$
Collecting results	N^2	N^2

$$\text{Total communication} = 2N^2\sqrt{n} + 3N^2 = O(N^2\sqrt{n})$$

6.9.4. Performance Evaluation

According to the analysis, the amount of communication in both implementations of the Fox algorithm is the same, and it is lower than that of the row-column approach. To compare these implementations, we performed a set of matrix multiplications using each implementation on the same set of hardware nodes. We used 256 CPU cores of Cluster II and evaluated each algorithm using different sizes of matrices. We also evaluated the performance of an OpenMPI based implementation of the Fox Matrix Multiplication algorithm on the same hardware setting.

Since Twister programs are implemented in Java and the MPI program is implemented in C++, first, we evaluate two sequential programs written in Java and C++ and compare their performances. Figure 22 shows these performances.

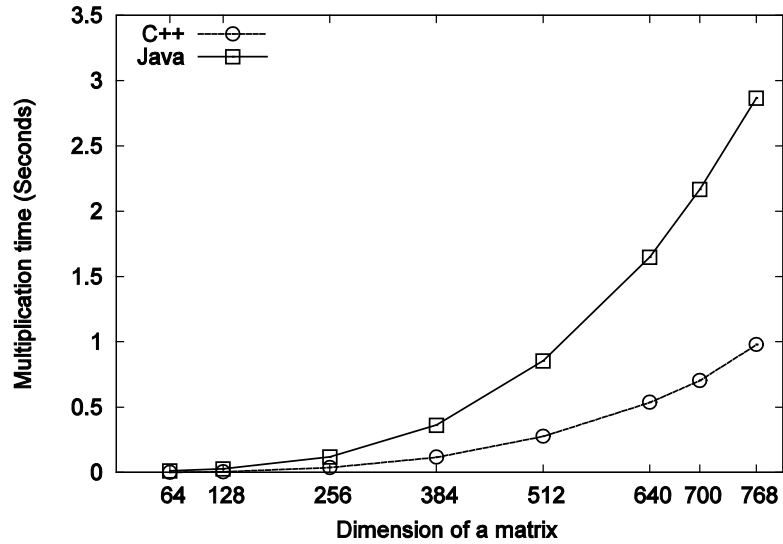


Figure 22. Sequential time for performing one block of matrix multiplication - Java vs. C++

The above measurements are made by using almost identical Java and C++ programs on the same hardware. Here we used matrices similar in size to the blocks that are assigned to individual processing cores in a parallel version of the program. The results show that there is a significant performance difference between Java and C++ for the matrix multiplication operation.

Next we compared the performance of the two Twister implementations against increasing matrix dimensions. In this evaluation, both Twister implementations use pub-sub brokers for intermediate data transfers (Note: We have not used the TCP based direct data transfers as explained in section 5.2). The following figure shows these performance characteristics.

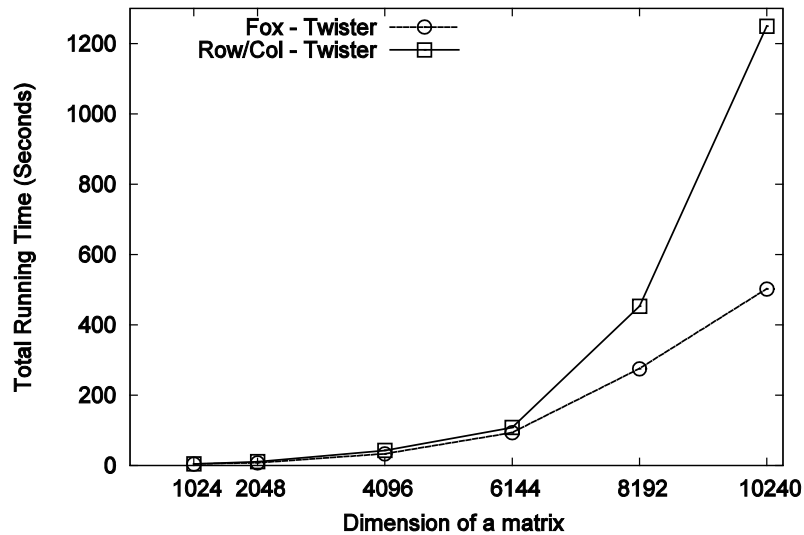


Figure 23. Performance of Row-Column and Fox algorithm based implementations by using 64 CPU cores of Cluster II. (Note: In this evaluation, we have not used Twister’s TCP based direct data transfers as explained in section 5.2).

Next, we evaluated the Twister and the MPI implementations of the Fox algorithm. In this evaluation, we used Twister’s TCP based direct data transfer mechanism. The following graph shows the overall matrix multiplication time of both Twister (Java) and OpenMPI (C++) applications. We measured only the time for matrix multiplication iterations (Initial data distribution and final data collection is ignored). The graph also shows the ideal compute times for each runtime based on the graph above as follows:

$$\text{Compute time} = \text{time per one block} * \text{number of iterations}$$

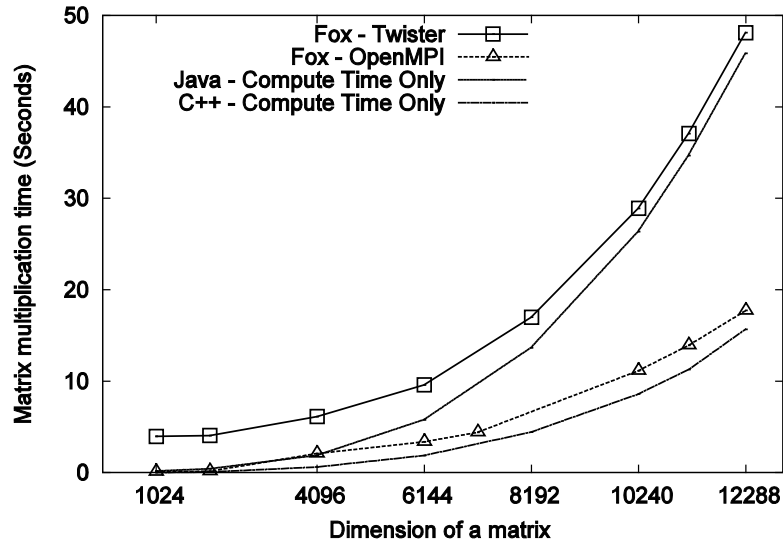


Figure 24. Performance of Twister and MPI matrix multiplication (Fox Algorithm) implementations by using 256 CPU cores of Cluster II. The figure also shows the projected compute-only time for both Java and C++.

According to Figure 24, Twister version of the matrix multiplication application is about three times slower than the OpenMPI counterpart. However, as shown in Figure 22, a multiplication of a block of matrix in Java is roughly three times slower than a C++ implementation. Therefore, if we normalize for these differences, both Twister and MPI would have similar performance characteristics. To understand this better, we performed an overhead calculation for the two implementations. Since there is a considerable difference between the sequential running times of Java and C++, when calculating overheads we used Java sequential times for Twister and C++ sequential times for MPI.

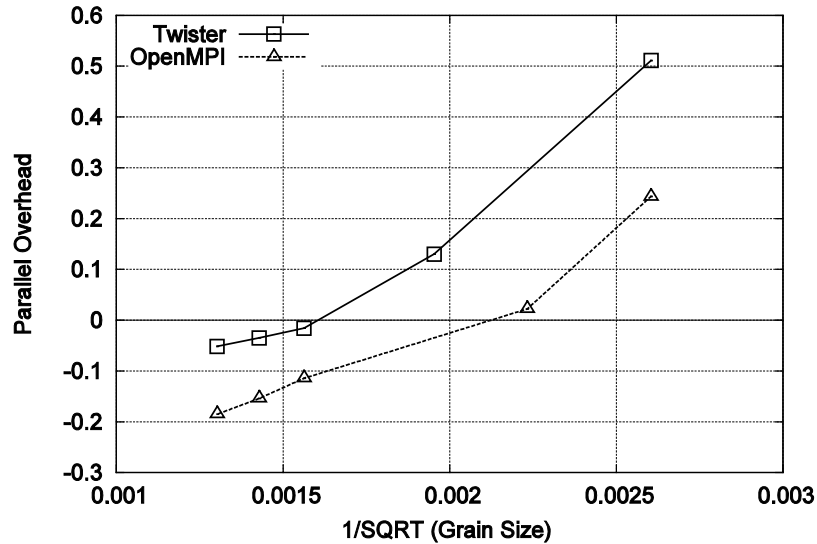


Figure 25. Parallel overhead of three matrix multiplication implementations by using 256 CPU cores of Cluster II

6.9.5. Discussion

The results in Figure 23 clearly indicate that the Twister versions of the Fox algorithm perform much better than the row-column decomposition approach. As we have explained, the Row-Column algorithm performs more data transfer than the Fox algorithm. Also, the broadcast operation in row-column algorithm depends heavily on the performance of the pub-sub brokers. These factors contribute to the higher running time of the row-column approach.

The parallel overheads shown in Figure 25 indicate that both MPI and Twister shows similar overhead characteristics and they both show negative overheads due better utilization of cache in the parallel application than the sequential application. For larger matrices i.e. smaller $1/\text{SQRT}(\text{grain size})$, both Twister and MPI implementations give highly desirable overhead characteristics. The Fox matrix multiplication is has a complex communication pattern compared to typical MapReduce applications. Twister runtime enables the development of such

algorithms in MapReduce programming model and as we have seen in matrix multiplication we expect it to produce better performance characteristics for large problem sizes.

Twister supports long running *map/reduce* tasks to improve the performance of many iterative MapReduce computations. It also allows these tasks to be configured with static data. Since the tasks only store static data, the *map* and *reduce* functions in Twister can still be considered “side effect free”, a feature that simplifies the fault handling mechanism of the runtime. With side-effect free tasks, Twister runtime can restart a failed iteration by simply re-configuring the tasks with the static data and re-executing the failed iteration. Similar to its MPI counterpart, in the Twister implementation of the Fox algorithm, we use *reduce* tasks to accumulate results of blocks of matrix C introducing side-effects. Therefore, under the current Twister implementation, this computation will not tolerate failures. To support fault tolerance with stateful *map* and *reduce* tasks, the runtime needs to save the state of individual tasks to a fault tolerance file system such as HDFS. This is an interesting future work.

6.10. Twister Benchmark Application and Micro Benchmarks

So far in this chapter, we discussed the real data analysis applications and their performances. To understand performance of the Twister runtime better we developed a MapReduce application that can simulate various application patterns. This section discusses some of the micro benchmarks that we performed and our findings.

6.10.1. Structure of the Benchmark Application

In the MapReduce model, the main communication occurs between *map* and *reduce* stages. Apart from this, Twister’s extended programming model supports broadcasting data to all map tasks as well as configuring *map* and *reduce* tasks by sending data directly from the main program (a scatter operation). The benchmark application supports variable message sizes to be used in all

the above three communication phases and variable sleep times, simulating computation times, at the *map* and *reduce* phases. This behavior allows us to use the benchmark application to evaluate Twister runtime under different communication loads or computation loads. The structure of this application is shown in Figure 26 below.

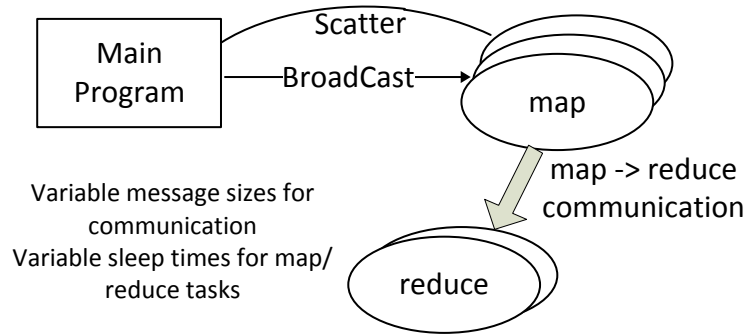


Figure 26. The structure of the micro benchmark program.

6.10.2. Micro Benchmarks

We evaluated the Twister runtime for the above three communication phases using 32 nodes of Cluster II. With each of these tests we used two broker settings; (i) single broker and (ii) 5 brokers connected in mesh configuration. To evaluate the effect of different message brokers we used both NaradaBrokering and ActiveMQ separately for each test setting. However, we could not test NaradaBrokering in full mesh configuration with multiple brokers due to a problem in NaradaBrokering. Therefore, for tests with NaradaBrokering we used a tree configuration for the broker network. Following set of figures shows the performance characteristics of these benchmarks.

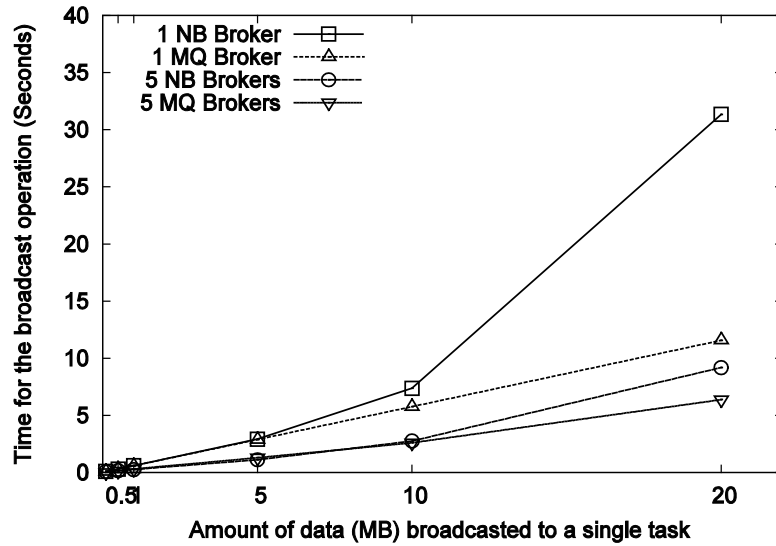


Figure 27. Time to send a single data item from the main program to all *map* tasks against the size of the data item.

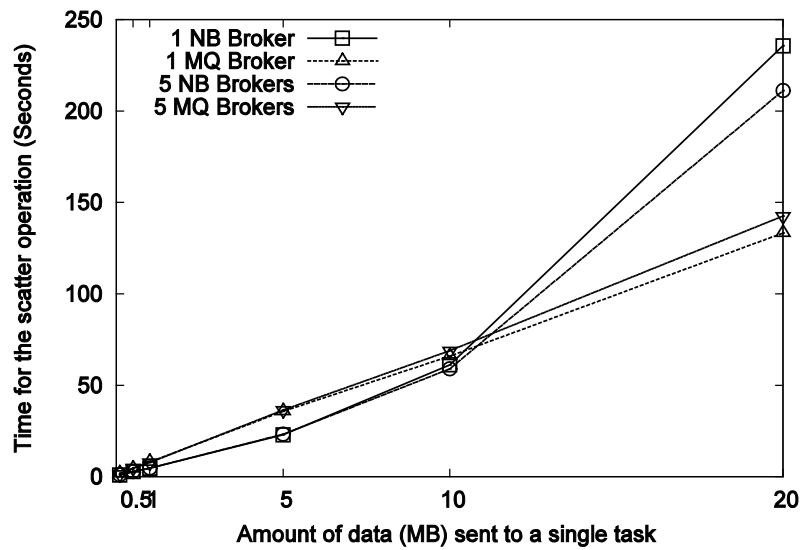


Figure 28. Time to scatter a set of data items to *map/reduce* tasks against scatter message size.

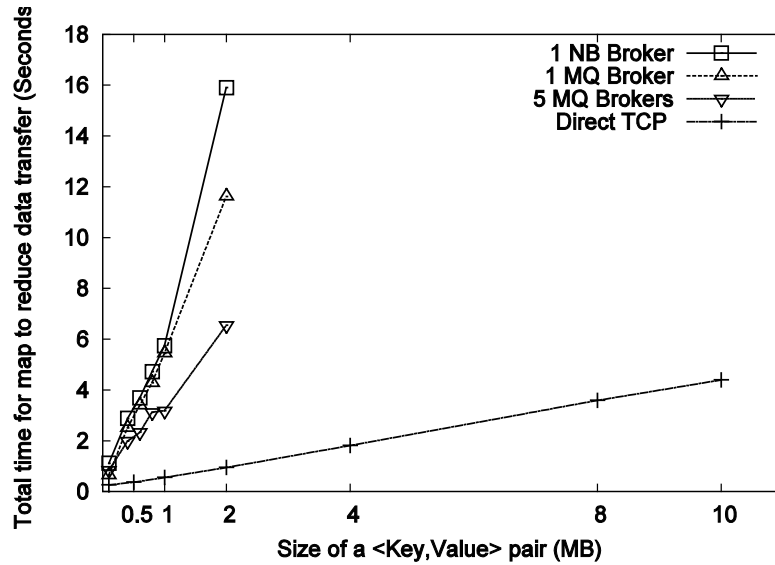


Figure 29. Total time for the *map to reduce* data transfer against <Key,Value> message size.

According to Figure 27, ActiveMQ broker performs the broadcast operation faster than the NaradaBrokering and more brokers in both types speedup the broadcast operation. In Twister, the broadcast operation is used when the main program (Twister Driver) sends some data directly to all *map* tasks. For example, in K-Means clustering, the main program sends the current cluster centers to all *map* tasks in each iteration. In the above benchmark, we used broadcast messages up to 20 megabytes in size which can represent roughly 2.5 million double values in each message proving that the brokered approach we adopted in Twister is capable of such large broadcasts. However, in real applications the broadcasts are typically used to send smaller data items such as parameters to all *map* tasks.

We performed a similar benchmark for the operation that configures *map* and *reduce* tasks. In this operation the data originates from the main program and is scattered to individual *map* and *reduce* tasks. However, unlike the broadcast operation where the broker handles the actual broadcasting, in this operation each piece of data needs to travel from the originator to the destination via the

broker network. That is, all the data needs to go through the connection between the main program and the first broker. This implies that the multiple brokers will not provide much benefit for this operation as observed in the performance results in Figure 28. For message sizes up to about 10 megabytes NaradaBrokering is faster than the ActiveMQ broker. However, for larger messages ActiveMQ handles the scatter operation better than NaradaBrokering.

Finally, we performed a benchmark to evaluate the performance of *map* to *reduce* data transfer. This is a crucial operation in MapReduce programming model. The amount of data transfer from *map* to *reduce* stages varies depending on the application. For example, in applications such as data clustering only the cluster centers need to be sent from *map* task to the *reduce* tasks. In contrast, a sorting operation transfers the entire data set through the map reduce pipeline. Since brokers are optimized for dispersing large number of small messages (or events), sending data via the broker network is acceptable only when the individual messages are considerably small. Larger messages cause considerable delays when sending via a broker network. As it can be seen in Figure 29, the TCP based direct data transfer mechanism solves this issue and it scales well with larger message sizes as well.

6.11. Conclusion

In this section, we discussed a series of data analysis applications that we have developed using Twister. Also, we discussed their respective performance by comparing them with the performance of several other implementations which performs the same algorithms. In each application, we discussed the mapping of the problem to the MapReduce programming model and its different implementations, by explaining the different ways that one can utilize these runtimes, especially the Twister runtime. We believe that the selected set of applications fairly represents the three application classes: (i) the map-only, (ii) the map-reduce, and (iii) the iterative map-reduce that we discussed earlier; therefore, the techniques we applied to these

applications can be reused to support other similar problems in these categories. We also discussed the applicability of Twister to the Fox matrix multiplication algorithm.

We also performed an extensive set of performance analyses to identify the performance characteristics of Twister under different problem categories. For all applications we tested, implementations based on Twister demonstrate the best performances compared to the DryadLINQ and the Hadoop implementations. The use of long running tasks and the faster communication mechanism utilized in Twister make it highly efficient for iterative MapReduce applications for which both DryadLINQ and Hadoop show considerable overheads. Although MPI outperforms Twister in iterative applications, our results indicate that the performance gap between MPI and Twister becomes reduced for large problem sizes. We performed several non-iterative applications such as CAP3 (section 6.3), HEP (section 6.4), and SW-G (section 6.5) which demonstrate the applicability of Twister to typical MapReduce applications. However, unlike Hadoop and DryadLINQ, in the current Twister implementation, we have not implemented fault tolerance support for non-iterative applications. We will discuss some of the possible approaches to make Twister fault tolerant for typical MapReduce applications in the section which discusses potential future research avenues.

The research work related to this thesis can be divided into two categories. First, this work could be classified with the broadly relevant parallel processing runtimes, including several MapReduce implementations that we discussed in Chapter 2, and second, the work could be categorized with the other parallel runtimes, which support some forms of parallel iterative algorithms. In this section we will discuss some of the approaches adopted by others for the second category.

7.1. Apache Mahout

Apache Mahout is a sub-project of the Hadoop, which provides scalable machine learning libraries based on Hadoop's MapReduce programming model. Currently, they support several categories of machine learning applications such as clustering, classification, and

recommendation mining. Although most of these algorithms perform iterative MapReduce computations, since they are based on Hadoop, each iteration is executed as a new MapReduce computation by Hadoop. For example, consider the following pseudo code extracted from the K-Means clustering implementation of the latest Apache Mahout (version 0.3) implementation.

K-Means Clustering - Apache Mahout

[Perform sequentially] the main program

```
1  while(! converged && (iteration<MAX_ITERATIONS))
2      JobConf jobConf = new JobCon()
3      jobConf.setXXX()
4      ..
5      JobClient.runJob(jobConf)
6      converged=isConverged()
7      iteration++
8  end while
```

[Perform in parallel] -the map(Key = id, Value = point)

```
9  Cluster = findNearestClusterForTheInputPoint(point)
10 Emit(cluster,point)
```

[Perform in parallel] - the local combine (Key=cluster,List<Value =point>)

```
11 for each point  $P_i$ 
12     count++
13     sum= sum +  $P_i$ 
14 end for
15 Emit(cluster, [count,sum])
```

[Perform Sequentially] -the reduce(Key=cluster, List<Value=[count,sum])

```
16 for each value  $V_i$ 
17     totalCount= totalCount + $V_i$ .count
18     totalSum= totalSum +  $V_i$ .sum
19 end for
```

```

20 newCluster = calculateCluster(totalCount,totalSum)
21 WriteToFileSystem (clusterId,newCluster)

```

According to this implementation, the main program creates a new Hadoop MapReduce job per each iteration (line number 2) and executes these jobs until convergence of cluster centers or the maximum number of iterations has been reached. During an iteration, at each *map* task, a worker in Hadoop reads the input data and calls the user defined *map* function (line number 9), by passing it point by point, which then calculates the nearest cluster center for the input data point and emits a (*cluster, point*) pair. The local *combine* function calculates the count and the sum for a group of points assigned to a single cluster center and produces a (*cluster, (count, sum)*) key-value pairs as output. The *reduce* function performs a similar operation to the *combine* function. It calculates the new cluster centers from a collection of *combine* outputs. To analyze the overhead of this, we can write the total running time of this computation for *n* iterations using P processors as follows:- (Note: for this analysis we assume that there are a large enough number of *map* and *reduce* tasks to process on P processors).

Time for K-Means on P processors =

$$T(P) = n * [T(\text{job submission}) + T(\text{read input}) + T(\text{map}()) + T(\text{map to reduce data transfer}) + T(\text{reduce}()) + T(\text{write output})]$$

In the above formula, the $T(\text{read-input})$ and $T(\text{write-output})$ represent times for reading an input data partition and writing a reduce output, respectively. If we assume uniform *map* and *reduce* running times, we can estimate the sequential running time of the K-Means clustering program as follows:-

Time for K-Means on 1 processor =

$$T(1) = P * T(\text{read input}) + n * P * T(\text{map}()) + n * P * T(\text{reduce}()) + P * T(\text{write output})$$

If we calculate the overhead of the MapReduce implementation using formula (2) of section Chapter 6, it will be as follows:

$$Overhead = \frac{P(T(P) - T(1))}{T(1)}$$

$$Overhead = \frac{P * n [T(job sub:) + T(read input) + T(map()) + T(data transfer) + T(reduce()) + T(write output)] - P * T(read input) + n * P * T(map()) + n * P * T(reduce()) + P * T(write output)}{P * T(read input) + n * P * T(map()) + n * P * T(reduce()) + P * T(write output)}$$

$$Overhead = \frac{n[T(job sub:) + T(data transfer)] + (n - 1)[T(read input) + T(write output)]}{T(read input) + T(write output) + n[T(map()) + T(reduce())]}$$

With the support for long running tasks, as in Twister, the above overhead reduces to:

$$Overhead = \frac{T(job sub:) + nT(data transfer)}{T(read input) + T(write output) + n[T(map()) + T(reduce())]}$$

This is much smaller than that of Apache Mahout, because it does not include the additional job submission, data reading, and writing times per each iteration. Furthermore, as we have explained in section 2.3, the data transfer in Hadoop goes through the file system twice, a step which incurs considerable data transfer overhead. The overhead of reading input data multiple times increases dramatically in runtimes that read data from remote locations such as Cloud MapReduce[96].

Apache Mahout provides implementations for a set of commonly used machine learning algorithms. However, as we have shown above, irrespective of the algorithm, the implementations incur considerable overheads when they are executed on the Hadoop runtime.

In contrast, most these algorithms will experience minimum overheads on Twister due to the enhanced architecture and the programming model we used.

7.2. Pregel

Pregel[97] is a runtime developed for processing large graphs in which the programs are expressed as a sequence of iterations. A user defined function is evaluated at each vertex of the graph during an iteration, and between iterations, the vertices send messages to each other. The authors describe the programming model as follows:

“Pregel computations consist of a sequence of iterations, called supersteps. During a superstep the framework invokes a user defined function for each vertex, conceptually in parallel. The function specifies behavior at a single vertex V and a single superstep S . It can read messages sent to V in superstep $S + 1$, send messages to other vertices that will be received at superstep $S + 1$, and modify the state of V and its outgoing edges. Messages are typically sent along outgoing edges, but a message may be sent to any vertex whose identifier is known.”

Although the programming model of Pregel is different than Twister’s MapReduce based programming model, there exist some similarities between the two runtimes. Most notably, unlike other MapReduce runtimes such as Hadoop and Dryad, both Twister and Pregel use long running tasks. A Vertex in Pregel holds a user defined value corresponding to the node of the graph that it represents, and it keeps changing this value depending on the computation performed by the user defined function, which is executed at each vertex. Twister also uses long running *map/reduce* tasks, and it supports configuring them with any static data once per computation; this possibility allows for the elimination of the need to re-load static data in each iteration. Although the functional view of MapReduce does not encourage the use of stateful *map/reduce* tasks, as we have shown in the Fox matrix multiplication described in section 6.9, one can use stateful *map/reduce* tasks in Twister to implement complex applications.

The possibility in supporting fault tolerance easily is one of the key benefits of the MapReduce programming model. However, with the use of stateful tasks, this possibility will no longer be in effect, because the tasks cannot be re-executed without losing their current state. The runtime needs to be able to preserve the state of every task in order to recover from failures. Furthermore, the runtime cannot simply save the current state of tasks to the local disks of the computers where they are executed, because a disk failure could result in a complete re-execution of the entire program. In typical MapReduce, a disk failure could result in the re-execution of the failed tasks in order to produce the missing intermediate data, however with stateful tasks this proves impossible. Therefore, the task state must be preserved in a fault tolerant distributed file system such as GFS or HDFS. From the Pregel paper, it is not clear which mechanism the system uses to save the state of the vertices in every super-step. However, it could most likely be stored in the Google File System so as to support fault tolerance. Serializing the entire graph to a distributed file system in each iteration is a costly checkpointing mechanism; therefore, we believe that a checkpoint at every few iterations will be a more practical approach. Currently, we do not support fault tolerance for stateful *map/reduce* tasks in Twister, as it is not coupled with a distributed file system such as HDFS or GFS. The development of this type of failure handling mechanism should emerge in interesting future research.

Under the MapReduce model, there is no direct communication path from the *reduce* stage back to the *map* stage of the computation. However, such a communication can be simulated by writing the output of the *reduce* stage to a distributed file system, and then reading the output back in *map* tasks during the next iteration. To illustrate this approach, let's consider a MapReduce implementation of a PageRank algorithm. For this analysis, we assume that the link graph is presented as an adjacency matrix in the format $\langle\langle page_1, \langle link_1, \dots, link_m_1 \rangle \rangle, \langle\langle page_2, \langle link_1, \dots, link_m_2 \rangle, \dots, \langle\langle page_n, \langle link_1, \dots, link_m_n \rangle \rangle$. The following algorithm shows a possible approach in implementing PageRank in MapReduce.

Pagerank Algorithm for MapReduce

```
1  do
    [Perform in parallel] -the map() operation
2  for each page  $P_i$ 
3       $PR(P_i) = \text{ReadPageRankFromFileSystem}(P_i)$ 
4       $r = PR(P_i) / \text{num\_out\_links}$ 
5      for each link  $L_j$ 
6           $\text{Emit}(L_j, P_i, r)$ 

    [Perform Sequentially] -the reduce() operation
7  Collect all  $(L_j, P_i, r)$ 
8  for each  $L_j$ 
9      for each page  $P_i$ 
10          $PR(L_i) = PR(L_i) + r$ 
11          $\text{WriteOutPutToFileSystem}(L_i, r)$ 

12 while ( $\text{num\_iterations} < \text{MAX\_ITERATIONS}$ )
```

As can be seen in the above algorithm, steps 3 and 12 use a distributed file system to share the current PageRank values between the *reduce* and the *map* stages of the computations. In Twister, we used the combine operation to collect these current PageRank vector to the main program. Then we broadcasted it to all map tasks again. However, in both these implementations, the above steps are responsible for the majority of the running time of the PageRank computation. In Pregel, the above steps are represented by direct messages transferred between super steps. Further, the communication between vertices does not introduce additional overheads. Therefore, the messaging-based approach adopted by Pregel provides a natural programming model for graph based algorithms.

7.3. Other Runtimes

Hoefler et al. discuss an efficient MapReduce implementation using MPI [98]. Their approach take advantage of the built in collective communication routines such as `MPI_Scatter` and `MPI_Reduce` to implement *map* and *reduce* operations, respectively. The use of the `MPI_Reduce` pushes the reduce operation to the MPI library itself and this process involves some limitations: for example, the number of intermediate keys needs to be known beforehand by all processing elements. Furthermore, this approach requires every *map* task to send a message for every key, irrespective of whether it has any data to send for that key. As we have demonstrated in Chapter 6, in MapReduce, the intermediate keys play the role of defining the communication topology between the *map* and *reduce* tasks. In addition, there are no limitations to the number of intermediate `<key, value>` pairs a map task can generate in a given iteration as well. While their approach can be used to simulate MapReduce on MPI, it will be highly efficient for some of the applications; however, it does not cover some of the key issues that MapReduce solves such as moving computation data, distributed input reading, and fault tolerance.

One of our key motivations in this research is to develop an efficient architecture and a programming model for MapReduce by incorporating best practices in terms of the HPC runtimes to MapReduce, but while still keeping the benefits of MapReduce intact. In this respect, the two specific features we have incorporated into Twister include: (i) long running tasks; and (ii) a faster communication mechanism. (With the improvement discussed in section 5.2, the bottleneck of brokers is also eliminated.) The results obtained from several benchmark applications indicate that we have successfully achieved the above objective with Twister. One of the key insights we have demonstrated is that, when the amount of data increases, a runtime with coarser grained tasks yet which utilizes sub optimal data transfer constructs, can achieve efficiencies in the same order as many HPC runtimes.

Ying Yu Bu et al. present LaHoop[99] runtime that extends the Apache Hadoop for iterative MapReduce computations. They also adopt long running tasks and allow tasks to retain static data across iterations. Furthermore, they optimize Hadoop's scheduler to assign tasks to the same location so as to support the process of reusing configured tasks. These optimizations are very similar to what we proposed concerning Twister in our initial paper [95] a few years ago; therefore, we are glad to see others adopting our recommended strategies about supporting iterative MapReduce applications.

The paper presented by Cheng-Tao et al. discusses their experience in developing a MapReduce implementation for multi-core machines[14]. They used the MapReduce runtime to implement several machine learning algorithms, and they demonstrate that MapReduce is especially effective for many algorithms that can be expressible in certain "summation form". Phoenix runtime, presented by Colby Ranger et al., is a MapReduce implementation for multi-core systems and multiprocessor systems [100]. The evaluations used by Ranger et al. is comprised of typical use cases found in Google's MapReduce paper such as word count, reverse index and also iterative computations such as Kmeans. Some of our design decisions in Twister were inspired by the benefits obtained in these shared memory runtimes. For example, in the above runtimes, the data transfer simply requires sharing memory references; in Twister, we use distributed memory transfers. Sending some data values to all map tasks is a trivial operation with shared memory, in Twister we introduced **mapReduceBcast()** to handle such requirements.

8.1. Summary of Work

In this dissertation, we presented the architecture and the programming model of an efficient parallel programming runtime, named Twister which is based on MapReduce that can be applied to many data intensive applications. We identified the composable class of applications to which the MapReduce can be effectively applied. We analyzed the domain of MapReduce applications and categorized them into several prominent classes including: (i) map-only; (ii) map-reduce; (iii) iterative map-reduce; and (iv) complex map-reduce; and through this process, we discussed the mapping of the algorithms to the parallel runtime constructs in order to demonstrate how different parallel runtimes, including Twister, could be used to parallelize these applications. We presented a detailed performance analysis of Twister and compared it to other runtimes using a

series real of data analysis applications to demonstrate how it could be used to achieve considerable efficiencies in comparison with typical MapReduce runtimes. Finally, we discussed research related to this thesis, along with a discussion about the current state of the art.

8.2. Conclusions

Large scale data analyses are becoming the norm in many areas of research and in numerous industries, a development that mandates the use of parallel and distribute processing. MapReduce extends the *map-fold* semantics offered in many functional languages to the distributed processing world; it also adds the support of *moving computation to data* by the use of distributed file systems. The simplified programming model of MapReduce allows the underlying runtimes to better support fault tolerance. However, this simplicity also limits its applicability to algorithms with fairly simple communication topologies. We proposed several extensions to the programming model which can potentially improve its overall applicability to more classes of applications.

The programming model proposed in this thesis uses three user defined functions and a main program: (i) *map*; (ii) *reduce*; (iii) *combine*; as well as (iv) a main program containing one or more MapReduce invocations, or, most importantly, an iterative construct (e.g. while or for loop) which invokes one or more MapReduce computations. It uses long running *map* and *reduce* tasks, inspired by classical parallel runtimes such as MPI. Furthermore, the programming model distinguishes between the static and variable data consumed by the *map* and *reduce* tasks which yield a behavior of *configure once and invoke many times*; this feature greatly simplifies the programming logic of many iterative MapReduce applications, and also reduces the overhead of loading static data in each iteration. We also introduced an additional phase called *combine* after the *reduce* phase of MapReduce, so as to collect all the *reduce outputs* to a single location for decision making. Sending a set of data items to individual *map* tasks (a scatter type operation)

and sending one data item to all *map* tasks (a broadcast type operation) prove to be very useful programming constructs that we support in the extended programming model as well.

The architecture presented in the introductory paper of MapReduce[2] exhibits a considerable coupling of the runtime to the infrastructure used in Google and to the type of operations that they perform. One of their recent papers [97], mentioned that preemption is one of the main reasons that they need fault tolerance rather than the hardware failures. These characteristics motivate them to develop their runtime with tight fault tolerance capabilities in which every piece of data produced was retained in some form of file system. We architected Twister to support iterative applications with a relaxed fault tolerance mechanism which achieved considerably higher efficiencies, especially in comparison with runtimes such as Hadoop that share a similar architecture to Google. Performance and efficiency are especially beneficial when running applications on the infrastructures acquired from Cloud on a pay per use basis or from resources allocated via job queues as well.

Our architecture comprises of three main entities: (i) A *MRDriver*, which is used as a library in the main program mentioned above; (ii) a daemon process that manages invocations of *map* and *reduce* tasks in a given computing node; and (iii) a publish/subscribe broker network. *MRDriver* manages a MapReduce computation during its life cycle while the daemons keep invoking *map/reduce* tasks, depending on the instructions relayed by the *MRDriver*. The architecture uses the broker network for transferring data as well as events related to the runtime. Depending on the size of data produced, the daemons either use the broker network or direct TCP links to send intermediate data between to one other. This is highly efficient compared to the *disk->network->disk* based communication mechanisms adopted by other MapReduce runtimes. The runtime also supports computation units with multi-core processors by using configurable thread pools

and employing a process which directly transfers data via memory for tasks residing in the same computation unit.

We implemented a series of data analysis applications representing different classes of MapReduce computations and discussed their algorithms. Most of these applications are implemented using several parallel runtimes such as Hadoop, DryadLINQ, and, in some cases, in MPI, to compare the performance of these runtimes with Twister. To evaluate performance, scalability, and efficiency, we performed a series of benchmarks using these applications; we also used a micro benchmark developed to simulate various application scenarios. These evaluations allowed us to derive the following conclusions regarding the applicability, performance and scalability of the proposed architecture and the programming model.

8.2.1. Applicability

In Chapter 6 of this thesis, we demonstrated the applicability of the proposed programming model to various classes of applications. Although it mainly focused on iterative applications, it does not lose the capability of supporting map-only or map-reduce classes of computations, as we have shown in sections 6.3, 6.4, and 6.5.

The *map* tasks in Twister can be programmed to access data loaded to memory via the *configure* option, or they can also access data directly from files in local disks. If the first option is used, the total amount of data that can be processed is limited to the total memory available in the computation infrastructure. This is typically enough for many iterative MapReduce computations. However, the latter option can be used to process large volumes of data and remains limited only by the total hard disk space available in the computation infrastructure, as we have shown in section 6.4.

Twister sends intermediate data directly from map tasks to reduce tasks either via pub-sub brokers or via TCP links. This does not impose a restriction to the volume intermediate data

transfer. However, since Twister stores the *reduce* inputs in memory, with the current implementation of Twister, the total intermediate data transfer (in a single iteration) should be limited to the total memory available in the computation infrastructure. In most MapReduce computations, a significant reduction of data volume occurs after the *map* phase of the computation; therefore, we expect that for most applications, this will not impose a restriction.

Twister provides fault tolerance to iterative computations by automatically unrolling and re-executing failed iterations. It does not support fault tolerance at the individual map and reduce tasks as in Hadoop, and therefore, it does not provide fault tolerance to typical MapReduce computations. As our experience indicates, the failures are less common in computation resources in academic environments and also, in resources leased from infrastructure services such as Amazon EC2. Therefore, we expect Twister to be an alternate for typical MapReduce computations in these environments as well.

The partition-file based data partitioning mechanism used in Twister allows users to access data directly as files in the local file system. This makes the use of executables or legacy applications in *map* or *reduce* functions fairly easy compared to the block based data partitioning strategies adopted in other runtimes.

8.2.2. Performance and Scalability

Twister significantly outperforms both DryadLINQ and Hadoop for all the iterative MapReduce computations that we have evaluated. For example, in both K-Means clustering and PageRank (section 6.7) computations, Twister performs ten times faster than its closest competitor. In matrix multiplication (section 6.9), Twister shows negative overheads due to super linear speedups, a characteristic that we have seen before only in MPI based runtimes. After normalizing for the performance differences in C++ versus Java, the performance of Twister become very close with the performance of MPI for this application. In the MDS application (section 6.8) we noticed

efficiencies above 80% even though the algorithm performs three MapReduce invocations in each iteration. In map-only and map-reduce applications (sections 6.3, 6.4, and 6.5) the performance of Twister and the other runtimes are in the same order of magnitude; however, in many of them Twister out performs the others.

In Chapter 6, we applied Twister to various MapReduce applications and ran it on moderately sized computation resources. We have performed SW-G (section) on 1629 CPU cores using 1629 Twister daemons and it showed a linearly scalability. Similarly, for many other evaluations, it showed desirable scalability characteristics as well. As we have explored in section 6.10, with micro benchmarks, there are several operations we can consider that can effectively evaluate the scalability of Twister. These include the following: (i) broadcast from main program; (ii) scatter from main program; (iii) intermediate data transfer; and (iv) collection of output to combine operation. In the first operation, the data is broadcasted to all daemons via a broker network and therefore the scalability of twister is governed by the scalability of the broker network used. As we have shown, more brokers can reduce the load on a single broker for this type of operation.

In the second operation, every piece of data goes from the *MRDriver* to a particular broker and then to the target daemon. This operation is not performed in parallel to minimize the load on the initial broker and may hinder the scalability of the application. However, typically, it is not advisable to use this operation for larger data items.

Intermediate data transfer is a significant factor in deciding the scalability of Twister. When intermediate data is transferred only using the broker network, the scalability suffers due to the loading of the brokers. Therefore, Twister adopts the TCP based direct communication for large intermediate data transfers. As discussed in section 6.10.2, this mechanism can transfer arbitrary large data items in parallel without suffering from scalability issues.

Finally, having the *combine* operation to collect the reduce output is a sequential activity, as there is only one combine function execution per iteration. Twister uses the TCP based data transfer strategy for large data items here as well. In addition, the downloading of data happens using multiple threads. For large deployments, transferring significant data to the combiner and processing them at the main program will hinder the scalability of the runtime. Furthermore, depending on the algorithm used, any computation performed in the main program will be sequential and will contribute to scalability degradation.

8.3. Contributions

In section 1.5, we proposed the contribution of this thesis. Here, we will discuss how we achieved them.

- **Architecture and the programming model of an efficient and scalable MapReduce runtime that extends the applicability of MapReduce programming model to more classes of data intensive computing. This proves effective especially, for the iterative MapReduce computations.**

We introduced a MapReduce programming model based on long running tasks with cacheable static data. We also introduced a *combine* operation and its semantics to MapReduce. As we have discussed above, these changes extends MapReduce to iterative and complex classes of applications.

- **A prototype implementation of the proposed architecture and the programming model that minimizes the overheads suffered by typical MapReduce runtimes.**

After realizing the benefits of the novel MapReduce runtime, we developed a release version of the software including a cluster deployment mechanism and a set of scripts to manipulate data across the local nodes of the computations clusters. We released Twister as an open source project under Indiana University's Academic License to the public via

www.iterativemapreduce.org. We also developed a detailed user guide and examples demonstrating the use of this runtime along with the source codes. The work in this thesis was showcased at the doctoral symposium of the annual Super Computing conference in 2009 (SC-09 in Portland) and a lengthy tutorial of Twister was given during the National Center for Supercomputing Applications (NCSA)'s Virtual School Summer Courses in September 2010.

- **The classification of problems that can be handled by MapReduce and algorithms for mapping these to the MapReduce model while minimizing overheads, followed by a detail discussion on different implementations using the proposed runtime as well as two other relevant runtimes.**

We classify the MapReduce domain into four classes and discuss their characteristics, as well as how to map algorithms in different categories to the MapReduce programming model while incurring minimum overheads. Chapter 6 of this thesis discusses these aspects in greater detail.

- **A detailed performance analysis comprised of application level performance characteristics to micro benchmarks and which evaluates the performance, scalability, and overhead of the proposed runtime against relevant runtimes.**

We performed a series of benchmarks using large data sets and considerable processing infrastructures on different versions of applications we have developed using DryadLINQ, Hadoop, and Twister, and in several cases, in MPI. Although we used these evaluations to understand the performance characteristics of Twister, they also served as comparisons of Twister with other runtimes; further, this highlights their strengths and weaknesses. Furthermore, we used a set of micro benchmarks to simulate various application scenarios which demonstrated the performance characteristics of Twister under such scenarios.

Overall, we have successfully extended the MapReduce programming model to iterative class of applications and we have demonstrated how to get the most benefits possible from it for more complex applications as well. The prototype we developed, Twister, was released as an open source project so that others could explore and benefit from and improve the scalable software architecture we discussed in this thesis. The related discussion concerning relevant research and the current state of the art, in combination with the extensive set of performance analyses we have included in this thesis, may help readers to gain an overall understanding of the different MapReduce runtimes and their applicability.

8.4. Future Work

In this research, we focused on an efficient runtime to support iterative MapReduce computations. We extended MapReduce programming model and introduces several improvements to its architecture. However, there are several areas which future work could do to build on this model:

Fault tolerance is one of the key features in MapReduce. Current Twister implementation supports fault tolerance for iterative MapReduce computations by re-executing failed iterations entirely. It does not support fault tolerance for typical MapReduce computations, which require saving the intermediate data in some form of file system that hinders the performance of iterative applications. One can research these issues and devise a methodology to checkpoint applications after a certain number (say n) of iterations. Then for typical MapReduce applications, we can use the same strategy with n as one.

Although we can store intermediate outputs in local disks of the compute nodes to achieve fault tolerance for typical and iterative MapReduce computations, this process does not solve the fault tolerance requirements of complex applications such as the Fox matrix multiplication in which

the *map* and *reduce* tasks accumulate state of the computation. A fault tolerance distributed file system is required to support such applications.

As in other MapReduce runtimes, Twister also assumes that the failures of the Master node is rare, i.e. if the Master node, where the Twister Driver runs fails, then the entire computation fails in Twister as well. There can be multiple ways to support Master failures: (i) a check pointing mechanism that can save the state of the Twister Driver and the “main program” which uses the Twister Driver; (ii) a duplicate master based approach; or even a (iii) master election based approach. These potential research avenues will prove fascinating for future developments which could prove common to many runtimes

Running multiple MapReduce applications in a workflow fashion is another common usage of MapReduce. In many such cases, one MapReduce application consumes the output of one or more previous MapReduce applications. In the current Twister runtime, the output of reduce tasks are stored in the local disk of the compute nodes, a process which does not guarantee fault tolerance with disk failures. A distributed file system or a simple data replication mechanism with a meta-data catalog needs to be integrated with Twister to support fault tolerance to such applications.

Incorporating the above type of file systems with Twister and understanding the effects in relation to the overall architecture and performance proves to be another interesting area for future research. There are multiple choices to adopt in this regard: (i) a block based file system such as HDFS, (ii) a distributed file storage such as Sector, and (iii) a distributed high performance file system such as Lustre. All these options provide different capabilities and could be well suited to different applications.

Current Twister implementation uses a static set of hardware nodes that are configured during the initialization of the runtime. Dynamic scaling of processing resources is a very important

property that a runtime should support. This is especially useful when applications are executed using Cloud resources. For example, an application may have several phases of computations in which only a few of these phases have higher processing requirements. The ability of a runtime to dynamically scale its processing units will save money when used in such scenarios. To support dynamic scalability with Twister, one needs to improve its task scheduling mechanism as well as add capabilities to dynamically move input data between processing units. For example, the addition of processing units requires Twister to re-distribute data and start computations on the newly added nodes.

Current Twister architecture uses a pubsub broker network as well as direct TCP links for data communication. The use of publish/subscribe infrastructure enables the runtime to connect data producers and consumers using virtualized topics, whereas direct TCP links are used to avoid the broker network from getting flooded with large data transfers. A separate communication layer that provides both these functionalities would be a definite improvement to the architecture of Twister.

The programming extensions we have introduced in Twister enable it to be used with iterative MapReduce applications. One can introduce more programming extensions by analyzing more classes of applications, especially complex applications, so as to extend its applicability further.

8.5. List of Publications Related to This Thesis

Following is a list of publications directly related to this thesis:

- Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, Geoffrey Fox, [Twister: A Runtime for Iterative MapReduce](#)," The First International Workshop on MapReduce and its Applications (MAPREDUCE'10) - HPDC2010

- Jaliya Ekanayake, (Advisor: Geoffrey Fox) [Architecture and Performance of Runtime Environments for Data Intensive Scalable Computing](#), Doctoral Showcase, SuperComputing2009. ([Presentation](#))
- Jaliya Ekanayake, Atilla Soner Balkir, Thilina Gunarathne, Geoffrey Fox, Christophe Poulain, Nelson Araujo, Roger Barga, [DryadLINQ for Scientific Analyses](#), Fifth IEEE International Conference on e-Science (eScience2009), Oxford, UK.
- Jaliya Ekanayake, Geoffrey Fox, [High Performance Parallel Computing with Clouds and Cloud Technologies](#), First International Conference on Cloud Computing (CloudComp09) Munich, Germany, 2009.
- Geoffrey Fox, Seung-Hee Bae, Jaliya Ekanayake, Xiaohong Qiu, and Huapeng Yuan, [Parallel Data Mining from Multicore to Cloudy Grids](#), High Performance Computing and Grids workshop, 2008.
- Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox [MapReduce for Data Intensive Scientific Analysis](#), Fourth IEEE International Conference on eScience, 2008, pp.277-284.

REFERENCES

- [1] *The Power Method*. Available: http://en.wikipedia.org/wiki/Pagerank#Power_Method
- [2] J. D. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *ACM Commun*, vol. 51, pp. 107-113, January, 2008.
- [3] *Datacenter*. Available: http://en.wikipedia.org/wiki/Data_center
- [4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," presented at the Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, Lisbon, Portugal, 2007.
- [5] G. Bell, J. Gray, and A. Szalay, "Petascale Computational Systems:Balanced CyberInfrastructure in a Data-Centric World," *IEEE Computer*, vol. 39, pp. 110-112, 2006.
- [6] C. T. M. Forum, "MPI: a message passing interface," presented at the Proceedings of the 1993 ACM/IEEE conference on Supercomputing, Portland, Oregon, United States, 1993.
- [7] (2009, December). *MPI*. Available: <http://www-unix.mcs.anl.gov/mpi/>
- [8] J. L. Hennessy and D. A. Patterson, *Computer Architecture, a Quantitative Approach*: Morgan Kaufman, 1990.
- [9] J. Gray and P. Shenoy, "Rules of thumb in data engineering," in *16th International Conference on Data Engineering*, San Diego, CA , USA 2000, pp. 3-10.
- [10] "single program multiple data," in *Algorithms and Theory of Computation Handbook*, P. E. Black, Ed., ed: CRC Press LLC, 1999.
- [11] *MPI (Message Passing Interface)*. Available: <http://www-unix.mcs.anl.gov/mpi/>
- [12] *PVM (Parallel Virtual Machine)*. Available: <http://www.csm.ornl.gov/pvm/>
- [13] G. Fox and S. Otto, "Matrix Algorithms on the Hypercube," California Institute of Technology 1985.
- [14] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," in *NIPS*, ed: MIT Press, 2006, pp. 281-288.
- [15] A. S. Foundation. (2009, *Apache Hadoop*). Available: <http://hadoop.apache.org/core>
- [16] Y. G. Gu, "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.

- [17] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley UCB/EECS-2006-183, December 18 2006.
- [18] *Amazon Simple Storage Service (Amazon S3)*. Available: <http://aws.amazon.com/s3/>
- [19] *Windows Azure Platform*. Available: <http://www.microsoft.com/windowsazure/>
- [20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 59-72, 2007.
- [21] *Disco project*. Available: <http://discoproject.org/>
- [22] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29-43, 2003.
- [23] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in *OSDI*, R. Draves and R. v. Renesse, Eds., ed: USENIX Association, 2008, pp. 1-14.
- [24] X. Huang, & Madan, A., "CAP3: A DNA sequence assembly program.," *Genome Res*, vol. 9, pp. 868-77, 1999.
- [25] M. Research. (2009, *Dryad and DryadLINQ Academic Release*. Available: <http://research.microsoft.com/en-us/downloads/03960cab-bb92-4c5c-be23-ce51aee0792c/default.aspx>
- [26] I. T. Foster, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," presented at the Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, 2001.
- [27] K. Ranganathan and I. Foster, "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications," presented at the Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, 2002.
- [28] G. Khanna, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz, "A Data Locality Aware Online Scheduling Approach for I/O-Intensive Jobs with File Sharing," presented at the 12th International Workshop on Job Scheduling Strategies for Parallel Processing, France, 2006.
- [29] (2010, *LSF Batch Concepts*. Available: http://people.ee.ethz.ch/~ballisti/computer_topics/lfs/admin/01-conce.htm
- [30] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience: Research Articles," *Concurrency and Computation Practice and Experience*, vol. 17, pp. 323-356, 2005.

- [31] S. L. Pallickara and M. Pierce, "SWARM: Scheduling Large-Scale Jobs over the Loosely-Coupled HPC Clusters," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, 2008, pp. 285-292.
- [32] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox, "Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications," presented at the Emerging Computational Methods for the Life Sciences Workshop of ACM HPDC 2010 conference,, 2010.
- [33] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," presented at the Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, 2004.
- [34] J. Frey. *Condor DAGMan: Handling Inter-Job Dependencies*. Available: <http://cs.wisc.edu/condor/dagman/>
- [35] G. C. Fox and D. Gannon, "Special Issue: Workflow in Grid Systems: Editorials," *Concurr. Comput. : Pract. Exper.*, vol. 18, pp. 1009-1019, 2006.
- [36] *Pegasus Project*. Available: <http://pegasus.isi.edu/>
- [37] S. Dustdar and W. Schreiner, "A survey on web services composition," *Int. J. Web Grid Serv.*, vol. 1, pp. 1-30, 2005.
- [38] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," *SIGMOD Rec*, vol. 3, pp. 44-49, 2005.
- [39] S. Shirasuna, "A Dynamic Scientific Workflow System for Web services Architecture," Ph.D., Department of Computer Science, Indiana University Bloomington, Bloomington, 2007.
- [40] I. Foster, "Languages for Parallel Processing," in *Handbook on Parallel and Distributed Processing*, J. Blazewicz, et al., Eds., ed, 2000.
- [41] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds., ed: MIT Press, 1996, pp. 75-213.
- [42] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Sci. Program.*, vol. 13, pp. 277-298, 2005.
- [43] *Apache Pig*. Available: <http://pig.apache.org>
- [44] (2009, December). *LINQ Language-Integrated Query*. Available: <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>
- [45] R. A. A. Bruce, S. Chapple, N. B. MacDonalds, A. S. T. and, and S. Trewin, Edinburgh Parallel Computing Center, University of EdinburghNovember, 1999 CHIMP and PUL: Support for portable parallel computing.

- [46] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," presented at the Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation, 1999.
- [47] W. G. a. E. Lusk, "Fault Tolerance in Message Passing Interface Programs," *International Journal of High Performance Computing Applications*, vol. 18, pp. 363-372, 2004.
- [48] *Open MPI:Open Source High Performance Computing*. Available: <http://www.openmpi.org/>
- [49] G. E. Fagg and J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World," presented at the Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2000.
- [50] *MPICH-V*. Available: <http://mpich-v.lri.fr/>
- [51] *POSIX Thread*. Available: http://en.wikipedia.org/wiki/POSIX_Threads
- [52] *Boost Library*. Available: <http://www.boost.org/>
- [53] *OpenMP*. Available: <http://openmp.org/wp/>
- [54] *Task Parallel Library*. Available: <http://msdn.microsoft.com/en-us/library/dd460717.aspx>
- [55] *Threading Building Blocks*. Available: <http://www.threadingbuildingblocks.org/>
- [56] *Microsoft Robotics Developer Studio*. Available: <http://msdn.microsoft.com/en-us/library/bb648752.aspx>
- [57] J. Duffy and E. Essey. (2007, Parallel LINQ: Running Queries On Multi-Core Processors. Available: <http://msdn.microsoft.com/en-us/magazine/cc163329.aspx>
- [58] ServePath. (2009, *GoGrid Cloud Hosting*. Available: <http://www.gogrid.com/>
- [59] ElasticHosts. (2009, *Cloud Hosting*. Available: <http://www.elastichosts.com/>
- [60] K. Keahey, I. Foster, T. Freeman, and X. Zhang, "Virtual workspaces: Achieving quality of service and quality of life in the Grid," *Sci. Program.*, vol. 13, pp. 265-275, 2005.
- [61] D. Nurmi, R. Wolski, C. Grzegorzcyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-Source Cloud-Computing System," in *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, 2009, pp. 124-131.
- [62] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," presented at the Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, 2003.

- [63] T. I. M. Joshua Hursey, Andrew Lumsdaine, "Interconnect agnostic checkpoint/restart in Open MPI," *Proceedings of the 18th ACM international symposium on High Performance Distributed Computing*, pp. 49-58, 2009.
- [64] *XMPI - A Run/Debug GUI for MPI*. Available: <http://www.lam-mpi.org/software/xmpi/>
- [65] S. H. Russ, R. Jean-Baptiste, T. S. K. Kumar, and M. G. Harmon, "Transparent Real-Time Monitoring in MPI," presented at the Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, 1999.
- [66] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.*, vol. C-21, p. 948, 1972.
- [67] G. C. Fox, R. D. Williams, and P. C. Messina, *Parallel Computing Works! : Morgan Kaufmann* 1994.
- [68] *Enabling Grids for E-science (EGEE)*. Available: <http://www.eu-egee.org/>
- [69] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Lightweight task execution framework," presented at the Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Reno, Nevada, 2007.
- [70] E. Corwin and A. Logar, "Sorting in linear time - variations on the bucket sort," *J. Comput. Small Coll.*, vol. 20, pp. 197-202, 2004.
- [71] J. B. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. vol. 1, L. M. L. Cam and J. Neyman, Eds., ed: University of California Press, 1967.
- [72] K. Rose, E. Gurewitz, and G. Fox, "A deterministic annealing approach to clustering," *Pattern Recogn. Lett.*, vol. 11, pp. 589-594, 1990.
- [73] S. Brin and L. Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. Available: <http://infolab.stanford.edu/~backrub/google.html>
- [74] J. de Leeuw, "Applications of convex analysis to multidimensional scaling," *Recent Developments in Statistics*, pp. 133-145, 1977.
- [75] S. Pallickara and G. Fox, "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids," presented at the Middleware 2003, 2003.
- [76] *ActiveMQ*. Available: <http://activemq.apache.org/>
- [77] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," in *American Federation of Information Processing Societies*, pp. 483-485.

- [78] (2009, December). *ROOT, Data Analysis Framework*. Available: <http://root.cern.ch/>
- [79] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain, "All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids," in *IEEE Transactions on Parallel and Distributed Systems*, 2010, pp. 33-46.
- [80] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology* vol. 162, pp. 705-708, 1982.
- [81] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, pp. 195-197, March 25 1981.
- [82] A. L. Price, E. Eskin, and P. A. Pevzner, "Whole-genome analysis of Alu repeat elements reveals complex evolutionary history," *Genome Res*, vol. 14, pp. 2245-2252, 2004.
- [83] (2009, December). *JAligner*. Available: <http://jaligner.sourceforge.net>
- [84] A. F. A. Smit, R. Hubley, and P. Green. (2004, *Repeatmasker*. Available: <http://www.repeatmasker.org>
- [85] J. Jurka, "Rebase Update:a database and an electronic journal of repetitive elements," *Trends in Genetics*, vol. 6, pp. 418-420, 2000.
- [86] J. Ekanayake, A. Balkir, T. Gunarathne, G. Fox, C. Poulain, N. Araujo, and R. Barga, "DryadLINQ for Scientific Analyses," presented at the 5th IEEE International Conference on e-Science, Oxford UK, 2009.
- [87] Y. Zhu, S. Ye, and X. Li, "Distributed PageRank computation based on iterative aggregation-disaggregation methods," presented at the Proceedings of the 14th ACM international conference on Information and knowledge management, Bremen, Germany, 2005.
- [88] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub, "Exploiting the Block Structure of the Web for Computing PageRank," Stanford InfoLab, Technical Report2003.
- [89] (2009, *The ClueWeb09 Dataset*. Available: <http://boston.lti.cs.cmu.edu/Data/clueweb09/>
- [90] J. Kruskal, "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis," *Psychometrika*, vol. 29, pp. 1-27, 1964.
- [91] Y. Takane, Young, F. W., & de Leeuw, J., "Nonmetric individual differences multidimensional scaling: an alternating least squares method with optimal scaling features," *Psychometrika*, vol. 42, pp. 7-67, 1977.
- [92] I. Borg, & Groenen, P. J., *Modern Multidimensional Scaling: Theory and Applications*: Springer, 2005.
- [93] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, and G. Fox, "High Performance Parallel Computing with Clouds and Cloud Technologies," in *Cloud Computing and Software Services: Theory and Techniques*, ed: CRC Press (Taylor and Francis).

- [94] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," presented at the Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Chicago, Illinois, 2010.
- [95] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for Data Intensive Scientific Analyses," presented at the Proceedings of the 2008 Fourth IEEE International Conference on eScience, 2008.
- [96] H. Liu and D. Orban, "Cloud MapReduce: a MapReduce Implementation on top of a Cloud Operating System," Accenture Technology Labs.
- [97] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," presented at the Proceedings of the 2010 international conference on Management of data, Indianapolis, Indiana, USA, 2010.
- [98] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards Efficient MapReduce Using MPI," presented at the Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Espoo, Finland, 2009.
- [99] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "HaLoop: Efficient Iterative Data Processing On Large Clusters."
- [100] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *13th International Symposium on High-Performance Computer Architecture*, 2007, pp. 13-24.

Vita

Name: Jaliya Ekanayake

Date of Birth: January 14, 1980

Place of Birth: Matara, Sri Lanka

Education:

January, 2007

Master of Science, Computer Science - 2007

Indiana University, Bloomington, Indiana

March, 2004

Bachelor of Science, Computer Science and Engineering - 2004

University of Moratuwa, Sri Lanka

Experience:

May, 2010 - Present

Research Software Development Engineer, Microsoft Research
Microsoft Corporation, Redmond, Washington

June, 2009 - September, 2009

Summer Intern, Microsoft Research
Microsoft Corporation, Redmond, Washington

August, 2005 - May, 2010

Research Assistant
Community Grids Lab, Indiana University
Bloomington, Indiana

February, 2005 - August, 2005

Research Engineer
Lanka Software Foundation, Colombo Sri Lanka

Honors/Affiliations:

Graduate Student Scholarship - Indiana University,
Bloomington (2005 - 2010)

Lanka Software Foundation Fellowship (2004-2005)

Committer for Apache Sandesha, Apache Axis2, and a member
of the Project Management Committee for Apache Sandesha
(2004 -2008)