Accelerating Hadoop Map-Reduce for Small/Intermediate Data Sizes using the Comet Coordination Framework

By

Shivangi Chaudhari

A Thesis submitted to the

Graduate School-New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

written under the direction of

Professor Manish Parashar

and approved by

_____

_____

_____

New Brunswick, New Jersey

October, 2009

ABSTRACT OF THE THESIS

Accelerating Hadoop Map-Reduce for Small/Intermediate Data Sizes using the

Comet Coordination Framework

By Shivangi Chaudhari

Thesis Director:
Professor Manish Parashar

MapReduce has been emerging as a popular programming paradigm for data intensive computing in clustered environments. MapReduce as a framework for solving embarrassingly parallel problems has been extensively used on large clusters. These frameworks support ease of computation for petabytes of data mostly through the use of a distributed file system example the Google File System – used by the proprietary 'Google Map-Reduce'.

In the "Map", the master node takes the input, divides it into smaller sub-problems, and distributes those to worker nodes. The worker node processes that smaller problem, and passes the answer back to its master node. In the "Reduce", the master node then takes the answers of the sub-problems and combines them to get the final output after reduces. The advantage of MapReduce is that, it allows for distributed processing of the map and reduction operations, assuming each operation is independent of the other, all can be executed in parallel.

We found that file writes and reads to the distributed file system, have an overhead especially for smaller data sizes of the order of few tens of GB's. Our solution provides the MapReduce framework built over Comet framework utilizing TCP sockets for

communication and coordination and uses in-memory operations for data whenever possible. The objective of this thesis is to

(1) understand the behaviors and limitations of MapReduce in the case of small-moderate datasets

(2) develop coordination and interaction framework to complement MapReduce-Hadoop to address these shortcomings

(3) demonstrate and evaluate using a real world application

In this thesis we use Comet and its services to build a MapReduce infrastructure that address the above requirements - specifically enable pull based scheduling of Map tasks as well as stream based coordination and data exchange. The framework is based on the master-worker concept. Comet is a decentralized (peer-to-peer) computational infrastructure that supports applications having high computational requirement.

Our System's interfaces are similar to the Hadoop MapReduce framework, to make applications built on Hadoop easily portable to Comet-based framework. The details of the implementation and evaluation of an actual pharmaceutical problem, with its results have been described. We found that out solution can be used to accelerate the computations of medium sized data by delaying or avoiding the use of distributed file reads and writes.

Acknowledgement and Dedication

I would like to thank my advisor Professor Manish Parashar for giving me this opportunity to work on something practical and interesting, for his enthusiasm, his inspiration, his encouragement, his sound advice and great efforts during my research in The Applied Software Systems Laboratory (TASSL). I am grateful to my colleagues at TASSL and other friends at Rutgers University for their emotional support and help, which made my study at Rutgers enjoyable and fruitful. I would like to thank the staff at the Center for Autonomic Computing (CAC) and Department of Electrical and Computer Engineering for their assistance and support. I wish to thank my parents and my brothers, for their understanding, endless encouragement, and love. Especially Girish for helping me see the silver lining all the times I was in low spirits and giving that encouragement and support. Finally, I want to thank Tutku for anything and everything, without whose support, company and help, I wouldn't have even got through the 2 years of my Master's.

Contents

List of illustrations

Chapter 1

Introduction

Computation and data intensive scientific data analyses are increasingly prevalent. The use of parallelization techniques and algorithms is the key to achieving better scalability and performance for the software engineering data analyses. Most of these analyses can be thought of as a Single Program Multiple Data (SPMD) algorithm or a collection thereof. These SPMDs can be implemented using different techniques such as threads, MPI, and *MapReduce*

There are several considerations in selecting an appropriate implementation strategy for a given data analysis. These include data volumes, computational requirements, algorithmic synchronization constraints, quality of services, easy of programming and the underlying hardware profile. We are interested in the class of scientific applications where the processing exhibits the composable property. Here, the processing can be split into smaller computations, and the partial-results from these computations merged after some post-processing, to constitute the final result. This is distinct from the tightly coupled parallel applications where the synchronization constraints are typically in the order of microseconds instead of the 50-200 millisecond coupling constraints in composable systems [4]. When the volume of the data is large, even tightly coupled parallel applications can sustain less stringent synchronization constraints. This observation also favors the MapReduce technique since its relaxed synchronization constraints do not impose much of an overhead for large data analysis tasks.

1.1 Motivation & Problem Description

MapReduce is a software framework made popular by Google [3] to support distributed computing on large data sets (terabytes to petabytes of data) on clusters of computers. The framework is inspired by map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as their original forms. MapReduce libraries have now been written in C++, C#, Java, Python, F# and other programming languages. These frameworks use the distributed file system as its underlying means of communication and storage.

We noticed that while the existing frameworks work for smaller datasets (a few gigabytes), they do not provide the same kind of efficiency as they would for lager sized petabytes datasets. A major factor being the extensive use of distributed File System and scarce use of memory for storage. These systems mostly store the intermediate results of the computations on local disks, where the computation tasks are executed, and then informs the appropriate workers to retrieve (pull) them for further processing. Although this strategy of writing intermediate result to the file system makes MapReduce frameworks robust such as Hadoop [9], it introduces an additional step and a considerable communication overhead, which could be a limiting factor for some MapReduce computations. Different strategies, such as writing the data to files after a certain number of iterations or using redundant reduce tasks, may eliminate this overhead and provide better performance to the applications. We noticed that the performance gains for smaller sized data, as in the case of applications in the pharmaceutical domain are improved significantly.

In this thesis we build on an existing framework ‘Comet’ [19], [1] which provides a decentralized shared space coordination framework for running distributed applications over large clusters. We decided to build a MapReduce framework support on Comet, application which relies on TCP sockets for communication and provides a stable coordination system. Comet supports the Master-Worker programming abstraction. In this thesis, we explore the possibility of using the Master Worker abstraction as the basis for building the MapReduce framework. This framework targets applications with lower datasizes of few gigabytes, and bypasses the read write overhead of the file system by using in-memory resources and TCP communication. We also evaluate the performance and stability of our Comet based MapReduce framework. Our results show that our framework accelerates the application from 60% – 300% depending on the resources available and the size of data workload on HPCS infrastructure [18].

1.2 Overview of Comet Based MapReduce Framework

The overall goal of this research is to design, implement and evaluate the MapReduce programming abstraction, based on a coordination infrastructure that enables the communications and interactions of heterogeneous entities in large decentralized distributed Grid environments. The specific objectives include:

(i) design a conceptual model for MapReduce applications, which uses the scalable, resilient, and simple coordination abstractions of Comet;

(ii) develop a MapReduce framework to demonstrate the conceptual model;

(iii) develop/port application systems to illustrate the effectiveness and feasibility of using the infrastructure for supporting MapReduce based Grid applications.

This thesis presents the design and implementation of the MapReduce programming framework over Comet. Comet is a fully decentralized coordination infrastructure for Grid environments. It provides a scalable, decentralized tuple space abstraction to address communication and synchronization of distributed processes and software elements. It provides a global virtual shared-space constructed from the semantic information space used by entities for coordination and communication.

The MapReduce infrastructure implements the conceptual architecture model. The model is built similar to the Hadoop MapReduce framework. It manages the distribution of task and computations using the pull based Master Worker implementation over Comet. Its main interfaces are the Input reader, the Mapper, which does the map computations and the Reducer, which does the reduce computations. It is developed as an application infrastructure layer on Comet, which provides programming frameworks and mechanisms. The MapReduce infrastructure has been deployed and evaluated on a campus network and Microsoft HPCS cluster at Rutgers University.

The developed infrastructure is evaluated using a classical WordCount application and an implementation of a real world pharmaceutical application from Bristol Meyers Squibb. Experimental evaluations using these applications demonstrate the flexibility, scalability

and effectiveness of the infrastructure, as well as its ability to support complex coordination requirements of the applications.

1.3 Contribution

This research investigates the Comet based MapReduce framework as an alternative to the existing MapReduce systems and the possibility of the developed infrastructure being used as an accelerator in coordination with the existing systems for small data sizes. The key contribution of this work is that it lays out a conceptual architecture model and provides a practical implementation of MapReduce programming paradigm based applications, using the Comet coordination infrastructure that facilitates scalable, robust, and efficient interaction and communication for Grid applications. We have used the Comet and its services to build this MapReduce infrastructure that address the above requirements - specifically enable pull based scheduling of Map tasks as well as stream based coordination and data exchange The main components of this research include:

(a) Understand the behaviors and limitations of existing MapReduce frameworks in the case of small to moderate data sets (understand the cross over points).

(b) Design of the MapReduce architecture model over Comet, which provides API's for building applications and running them reliably. It exploits the global virtual shared-space abstraction provided by Comet that can be associatively accessed by all system peers.

(c) Develop application programming systems using Comet based MapReduce infrastructure for solving practical problems.

(d) Deploy and evaluate the framework and a real world application on Rutgers University campus networks. The experiments evaluate the performance of Comet based MapReduce system. The experimental results demonstrate the scalability and efficiency of these systems as well as the feasibility of using Comet to support MapReduce type of applications.

Chapter 2

Background & Related Work

2.1 MapReduce Programming Paradigm basics

2.1.1 Functional Programming Concepts

MapReduce programs are designed to compute large volumes of data in a parallel fashion. This requires dividing the workload across a large number of machines. This model would not scale to large clusters (hundreds or thousands of nodes) if the components were allowed to share data arbitrarily. The communication overhead required to keep the data on the nodes synchronized at all times would prevent the system from performing reliably or efficiently at large scale.

Instead, all data elements in MapReduce are *immutable*, meaning that they cannot be updated. If in a mapping task you change an input (key, value) pair, it does not get reflected back in the input files; communication occurs only by generating new output (key, value) pairs, which are then forwarded by the system the next phase of execution.

2.1.2 List Processing

Conceptually, MapReduce programs transform lists of input data elements into lists of output data elements. A MapReduce program will do this twice, using two different list processing idioms: *map*, and *reduce*. These terms are taken from several list processing languages such as LISP, Scheme, or ML.

2.1.3 Mapping Lists

The first phase of a MapReduce program is called *mapping*. A list of data elements are provided, one at a time, to a function called the *Mapper*, which transforms each element individually to an output data element.

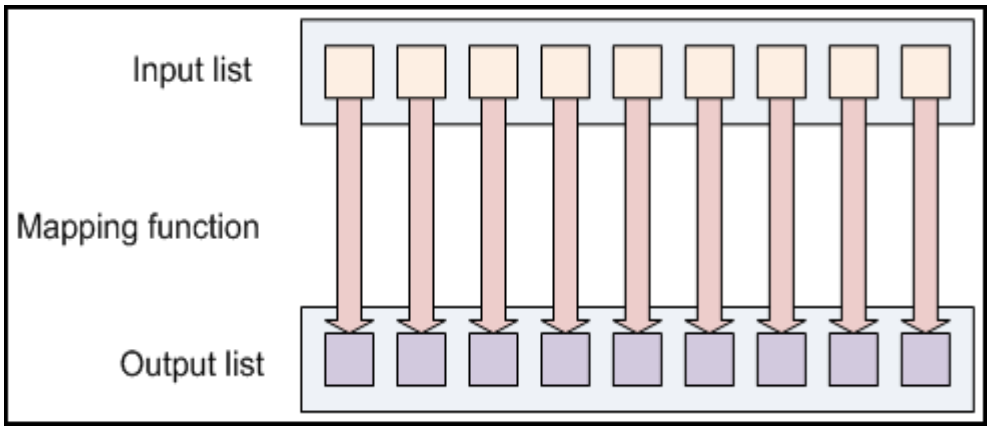

Figure 1: Mapping creates a new output list by applying a function to individual elements of an input list.

As an example of the utility of map: Consider a function toUpper(str) which returns an uppercase version of its input string. You could use this function with map to turn a list of strings into a list of uppercase strings. Note that we are not *modifying* the input string here. Instead we are returning a new string that will form part of a new output list.

2.1.4 Reducing Lists

Reducing lets you aggregate values together. A *reducer* function receives an iterator of input values from an input list. It then combines these values together, returning a single output value.

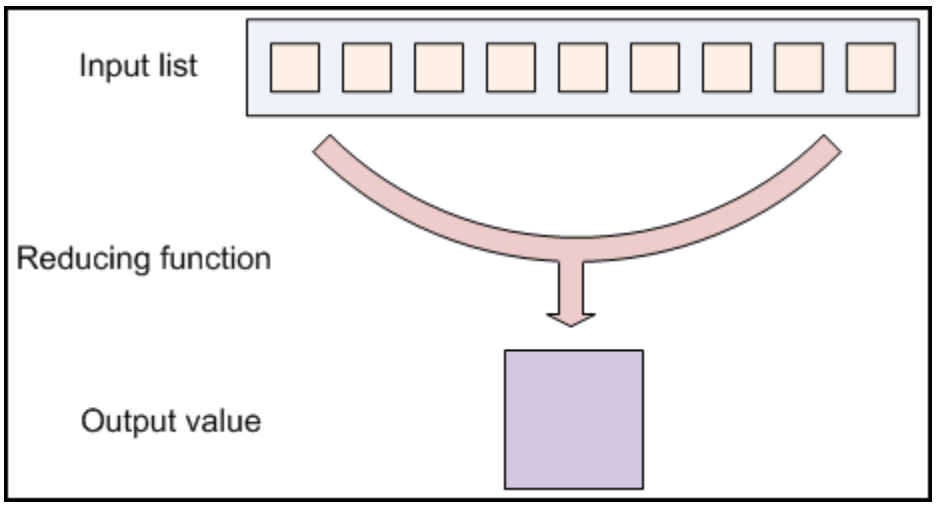

Figure 2: Reducing a list iterates over the input values to produce an aggregate value as output.

Reducing is often used to produce "summary" data, turning a large volume of data into a smaller summary of itself. For example, "+" can be used as a reducing function, to return the sum of a list of input values.

2.1.5 Putting Them Together in MapReduce:

The MapReduce framework takes these concepts and uses them to process large volumes of information. A MapReduce program has two components: one that implements the mapper, and another that implements the reducer. The Mapper and Reducer idioms described above are extended slightly to work in this environment, but the basic principles are the same.

Keys and values: In MapReduce, no value stands on its own. Every value has a key associated with it. Keys identify related values. For example, a log of time-coded speedometer readings from multiple cars could be keyed by license-plate number; as follows:

AAA-123 65mph, 12:00pm

ZZZ-789 50mph, 12:02pm

AAA-123 40mph, 12:05pm

CCC-456 25mph, 12:15pm

...

The mapping and reducing functions receive not just values, but (key, value) pairs. The output of each of these functions is the same: both a key and a value must be emitted to the next list in the data flow.

MapReduce is also less strict than other languages about how the Mapper and Reducer work. In more formal functional mapping and reducing settings, a mapper must produce exactly one output element for each input element, and a reducer must produce exactly one output element for each input list. In MapReduce, an arbitrary number of values can be output from each phase; a mapper may map one input into zero, one, or many outputs. A reducer may compute over an input list and emit one or several different outputs.

Keys divide the reduce space: A reducing function turns a large list of values into one (or a few) output values. In MapReduce, all of the output values are not usually reduced together. All of the values with the same key are to be presented to a single reducer together. This is performed independently of any reduce operations occurring on other lists of values, with different keys attached.

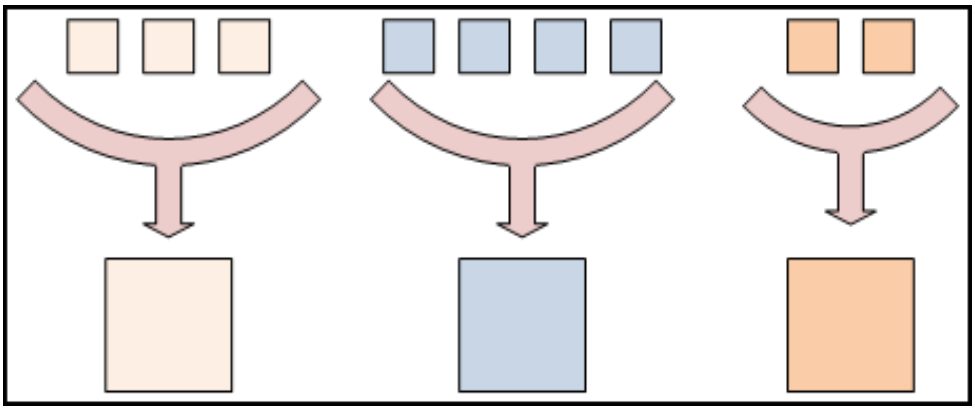

Figure 3: Different colors represent different keys. All values with the same key are presented to a single reduce task.

2.2 Dataflow

The frozen part of the MapReduce framework is a large distributed sort. The Key components, that a MapReduce application defines, are:

- an *input reader*
- a *Map* function
- a *partition* function
- a *compare* function
- a *Reduce* function
- an *output writer*

2.2.1 Input reader

The *input reader* divides the input into 16MB to 128MB splits and the framework assigns one split to each *Map* function. The *input reader* reads data from stable storage (typically a distributed file system like Google File System) and generates key/value pairs.

A common example will read a directory full of text files and return each line as a record.

2.2.2 Map function

Each *Map* function takes a series of key/value pairs, processes each, and generates zero or more output key/value pairs. The input and output types of the map can be (and often are) different from each other.

If the application is doing a word count, the map function would break the line into words and output the word as the key and "1" as the value.

2.2.3 Partition function

The output of all of the maps is allocated to particular *reducer* by the application's *partition* function. The *partition* function is given the key and the number of reduces and returns the index of the desired *reduce*.

A typical default is to hash the key and modulo the number of *reduces*.

2.2.4 Comparison function

The input for each *reduce* is pulled from the machine where the *map* ran and sorted using the application's *comparison* function.

2.2.5 Reduce function

The framework calls the application's *reduce* function once for each unique key in the sorted order. The *reduce* can iterate through the values that are associated with that key and output 0 or more values.

In the word count example, the *reduce* function takes the input values, sums them and generates a single output of the word and the final sum.

2.2.6 Output writer

The *Output Writer* writes the output of the reduce to stable storage, usually a distributed file system, such as Google File System.

2.3 Existing Systems

2.3.1 Google MapReduce

MapReduce is a parallel programming technique derived from the functional programming concepts and is proposed by Google for large-scale data processing in a distributed computing environment [3], [10]. The Google MapReduce framework is not open source. However this section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet. In their environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used. Typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

2.3.2 Apache MapReduce (Hadoop)

Hadoop [9], [11] is an open source framework for running applications on large clusters built of commodity hardware from Apache. The Hadoop framework transparently provides applications both reliability and data motion. Hadoop implements Map/Reduce, where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. In addition, it provides the Hadoop distributed file system (HDFS) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. Both Map/Reduce and the distributed file system are designed so that node failures are automatically handled by the framework.

2.3.3 CGL MapReduce

CGL-MapReduce [7], [4] is another MapReduce runtime from Indiana University that uses streaming for all the communications, which eliminates the overheads associated with communicating via a file system. The use of streaming enables the CGL-MapReduce to send the intermediate results directly from its producers to its consumers. Currently, they have not integrated a distributed file system such as HDFS with CGL-

MapReduce, and hence the data should be available in all computing nodes or in a typical distributed file system such as NFS.

2.4 MapReduce Applications

MapReduce is useful in a wide range of applications, including: "distributed grep, distributed sort, web link-graph reversal, term-vector per host, web access log stats, inverted index construction, document clustering, machine learning, statistical machine translation, etc" Most significantly, when MapReduce was finished, it was used to completely regenerate Google's index of the World Wide Web, and replaced the old *ad - hoc* programs that updated the index and ran the various analyses.

MapReduce's stable inputs and outputs are usually stored in a distributed file system. The transient data is usually stored on local disk and fetched remotely by the reduces.

2.5 Comet

Comet is a decentralized (peer-to-peer) computational infrastructure that extends Desktop Grid environments to support applications that have high computational requirements along with non linear communication requirements. It provides a decentralized and scalable tuple space, efficient communication and coordination support, and application-level abstractions that can be used to implement Grid applications based on the master-worker/BOT paradigm.

The tuple space is essentially a global virtual shared-space constructed from the semantic information space used by entities for coordination and communication. This information space is deterministically mapped, using a locality-preserving mapping, onto the dynamic

set of peer nodes in the Grid system. The resulting structure is a locality preserving semantic distributed hash table (DHT) [1] built on top of a self-organizing structured overlay.

2.5.1 Architecture

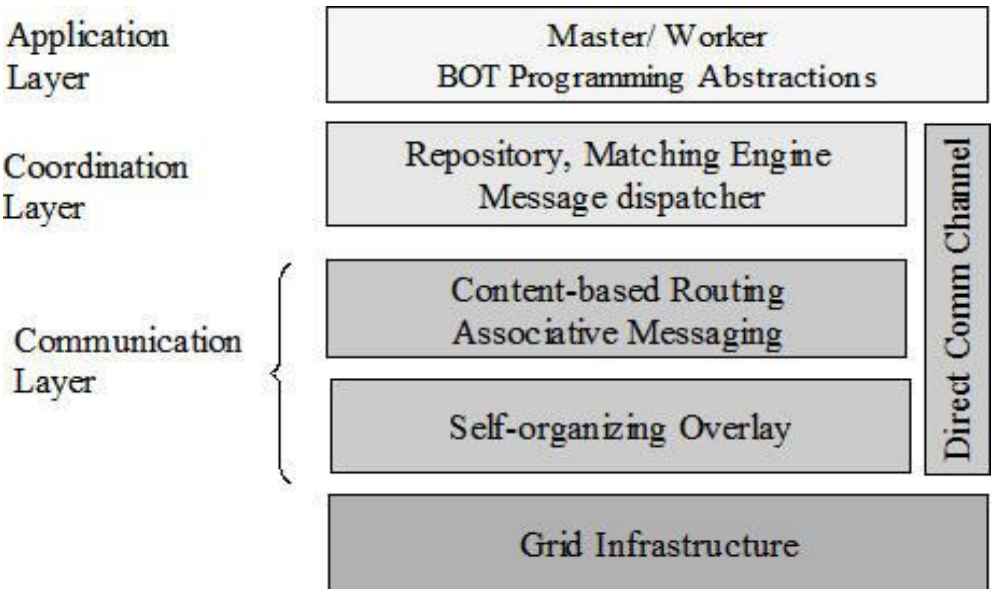

Figure 4: A schematic overview of the CometG system architecture.

The communication layer provides an associative communication service and guarantees that content-based messages, specified using flexible content descriptors, are served with bounded cost. This layer also provides a direct communication channel to efficiently support large volume data transfers between peer nodes. The communication channel is implemented using a thread pool mechanism and TCP/IP sockets.

The coordination layer provides the Linda-like shared-space coordination interfaces: (i) Out(ts, t): a non-blocking operation that inserts tuple t into space ts. (ii) In(ts,t, timeout):a

blocking operation that removes a tuple t matching template t from the space ts and returns it. If no matching tuple is found, the calling process blocks until a matching tuple is inserted or the specified timeout expires. In the latter case, null is returned. (iii) Rd(ts, t, timeout): a blocking operation that returns a tuple t matching template t from the space ts. If no matching tuple is found, the calling process blocks until a matching tuple is inserted or the specified timeout expires. In the latter case, null is returned. This method performs exactly like the 'In' operation except that the tuple is not removed from the space.

Replication:

Comet provides programming layer, service layer, and infrastructure layer. In infrastructure layer, we use Chord overlay for self-organizing layer and Squid information discovery scheme for content-based routing. This layer now provides **replication** and load balancing to **support dynamic join and leave** as well as node failure. Every node keeps the replica of its successor node's state, and it does not only reflect changes to the replica whenever its successor notifies state changes, but also notifies its every change to its predecessor. If a node fails, the predecessor node merges the replica into its state and then makes a new replica of the new successor. If a new node joins, its newly defined predecessor changes the replica to reflect the new node's state and its newly defined successor gives its state information to it. For load balancing, load should be redistributed to nodes whenever a node joins and leaves. Here, load means the number of tasks stored on a node. If there are more nodes on the overlay, then a node keeps the less number of tasks on its storage.

Scheduling and Monitoring:

Scheduling and monitoring of tasks are provided under the application framework. **Task consistency** check is done for lost tasks. Even though replication is provided in infrastructure layer, some tasks could be lost because of network congestion. In this case because failure has occurred, replication cannot treat the lost tasks. Hence, the master waits for the result of each task for some pre-defined time duration and if it does not receive the result back, then it **regenerates the lost task**. If the master receives duplicate results of a task, it ignores the later results.

This feature is also helpful in **case of failure of nodes** during a run. The tasks on the node that would be lost will be regenerated by the master after a predefined timeout.

Programming Abstraction

There are Api's made available to build applications to utilize the Comet framework with its features. Sample applications show code snippets that can be used to exploit the distributed infrastructure.

2.5.2 Master Worker Paradigm

The Comet provides coordination space abstractions and programming modules to support master-worker/Bag-Of-Task (BOT) parallel formulations of asynchronous computations, where the individual tasks are independent and do not require inter-task communications.

The Master Worker paradigm in Comet differs from the existing models, in the way that in Comet we use the pull model – than the push models that exist. Here the Master need not push the tasks to the workers, as workers query for the tasks and pull it out of the shared space. The master can just pour in the tasks in tuple space and resume doing other computations and monitoring tasks.

Chapter 3

Hadoop MapReduce Evaluation and Observations

MapReduce is the key algorithm that the Hadoop MapReduce engine uses to distribute work around a cluster

3.1 The Map

A map transform is provided to transform an input data row of key and value to an output key/value:

- map(key1,value) -> list<key2,value2>

That is, for an input it returns a list containing zero or more (key, value) pairs:

- The output can be a different key from the input
- The output can have multiple entries with the same key

3.2 The Reduce

A reduce transform is provided to take all values for a specific key, and generate a new list of the *reduced* output.

- reduce(key2, list<value2>) -> list<value3>

3.3 The MapReduce Engine

The key aspect of the MapReduce algorithm is that if every Map and Reduce is independent of all other ongoing Maps and Reduces. As a result the operation can be run

in parallel on different keys and lists of data. On a large cluster of machines, you can go one step further, and run the Map operations on servers where the data lives. Rather than copy the data over the network to the program, you push out the program to the machines. The output list can then be saved to the distributed filesystem, and the reducers run to merge the results. Again, it may be possible to run these in parallel, each reducing different keys.

A distributed filesystem spreads multiple copies of the data across different machines. This not only offers reliability without the need for RAID-controlled disks, it offers multiple locations to run the mapping. If a machine with one copy of the data is busy or offline, another machine can be used.

A job scheduler (in Hadoop, the JobTracker), keeps track of which MR jobs are executing, schedules individual Maps, Reduces or intermediate merging operations to specific machines, monitors the success and failures of these individual *Tasks*, and works to complete the entire batch job.

The filesystem and Job scheduler can somehow be accessed by the people and programs that wish to read and write data, and to submit and monitor MR jobs.

Apache Hadoop is such a MapReduce engine. It provides its own distributed filesystem and runs [HadoopMapReduce] jobs on servers near the data stored on the filesystem -or any other supported filesystem, of which there is more than one.

3.4 Experimental test runs on Hadoop MapReduce

We did simple small test runs over Hadoop, using the classical example for MapReduce programs, the word count example. We also had some application data from Bristol Meyers Squibb, which is used to identify bio-active poses in the Protein molecule structures. This computation is a tremendous challenge in drug discovery in pharmaceutical industries.

We observed that the extensive use of the distributed filesystem in Hadoop induces an overhead for smaller sizes of data. For large data sizes the filesystem overhead is offset by the massive size of data.

The graph below led us to look into alternate methods to handle applications with small data sizes that need to use a MapReduce framework.

	Computation secs	Overhead secs
Good Case -WC	1.74	0.52347
Bad Case -PDB	0.00113912	4.27212

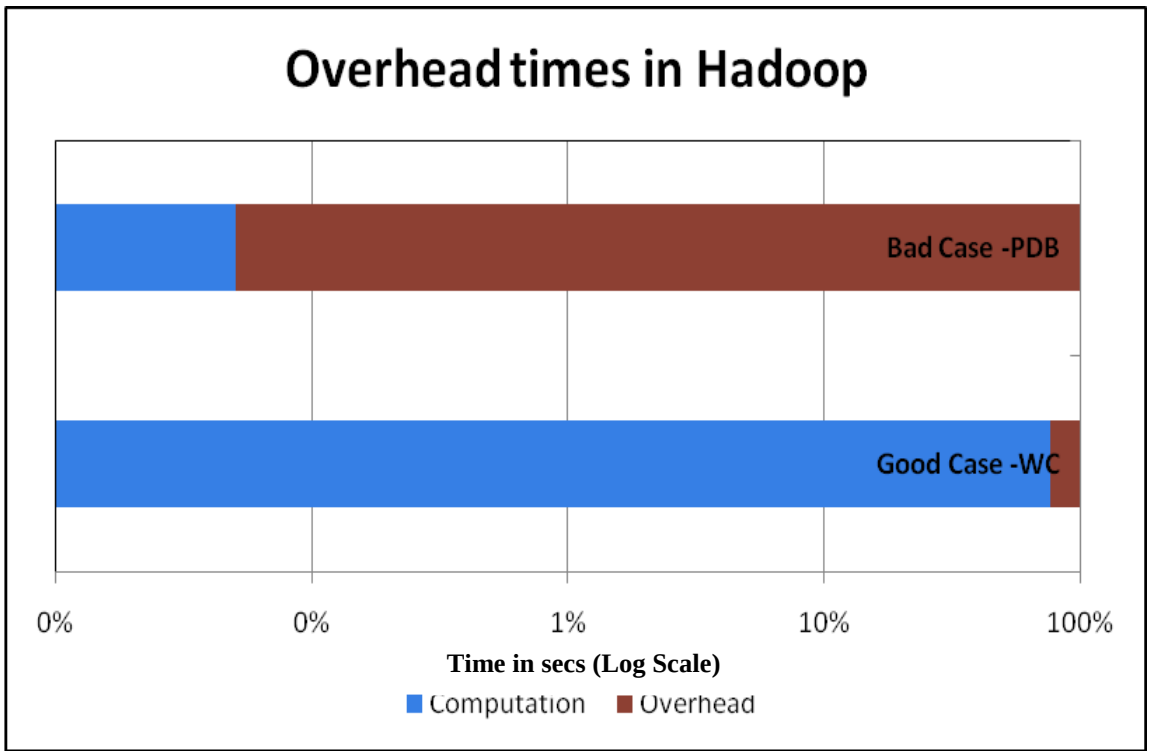

Figure 5: Performance of Hadoop over different application types.

From the Figure 5 we can see that the incase of the word count example the computation time is larger than the I/O and reporting overhead, which is a good case however the Protein Data Bank example is a bad case where the computation time is way lower than the overhead. So this indicates that for cases like the latter use of more in-memory computation and storage can potentially accelerate application execution.

We observed that the read write speeds in the Hadoop distributed file system are higher as compared to the Linux NFS filesystem and a shared folder on the Microsoft HPCS platform.

Platform ->	Hadoop DFS	Linux NFS	Microsoft Shared Folder
Read Speed	1.249 MBPS	1288.65979 MBPS	9560.84MBPS
Write Speed	125.256KBPS	56.999 MBPS	6486.07MBPS

This prompted us to look for alternatives for accelerating cases in which the data size varies from small to medium. We observed that for smaller data sizes any master –worker framework along with NFS could be used for a similar MapReduce engine. So we decided to use Comet Coordination framework which already supported Master – Worker paradigm based applications. We built a MapReduce programming abstraction utilizing Comet API's exposed through the Master Worker framework. The abstraction delays the use of file system by performing most computations in memory and only resorting to disk reads and writes when the application free memory goes below the specified threshold. We also observed the reads and writes to local disks is faster than over the network and hence can be exploited for handling local data for each task.

Chapter 4

MapReduce Abstraction over Comet

4.1 Architecture

This chapter presents the conceptual architecture model and implementation of MapReduce over the Comet distributed and decentralized coordination infrastructure. The Comet conceptual architecture model is based on a global virtual shared-space constructed from a semantic information space that is used by entities for coordination and communication. Comet adapts the Squid information discovery scheme to deterministically map the information space onto the dynamic set of peer nodes in the Grid system. The resulting structure is a locality preserving semantic distributed hash table (DHT) on top of a self-organizing structured overlay. The decentralized tuple space maintains content locality and guarantees that content-based tuple queries, using flexible content descriptors in the form of keywords, partial keywords and wildcards, are delivered with bounded costs. All system peers can associatively access the Comet space without requiring the location information of tuples and host identifiers. The Master Worker / (BOT) paradigm has been implemented over Comet which utilizes the Comet API's mainly – 'out', 'in' and 'read'. Due to the space-based nature of Comet infrastructure, the master worker type of applications display a pull based mechanism rather than the more prevalent push based models. The Master thus, can insert tasks in the space and the workers can pull the task by querying for tasks with keywords, wildcards or a combination of both. This very Master Worker abstraction is used to build the MapReduce programming abstraction. As seen in the figure below the MapReduce

abstraction sits between the Master Worker abstraction layer and the application layer.

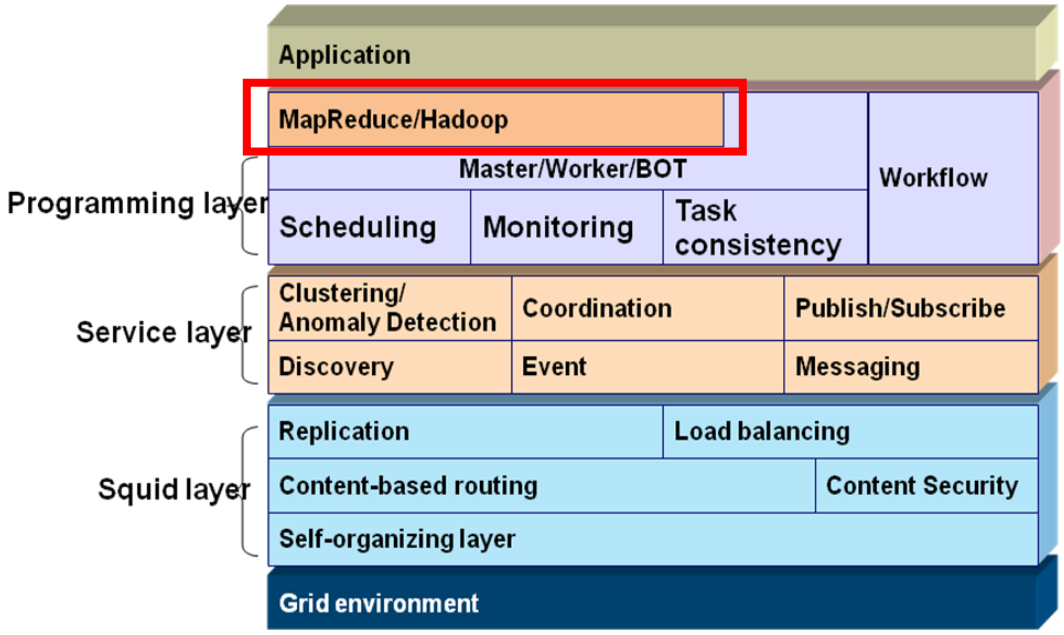

Figure 6: A schematic overview of the Comet system with MapReduce abstraction layer.

4.1.1 The MapReduce Data Flow

The flowchart below gives the data and functional flow of the MapReduce application run on Comet.

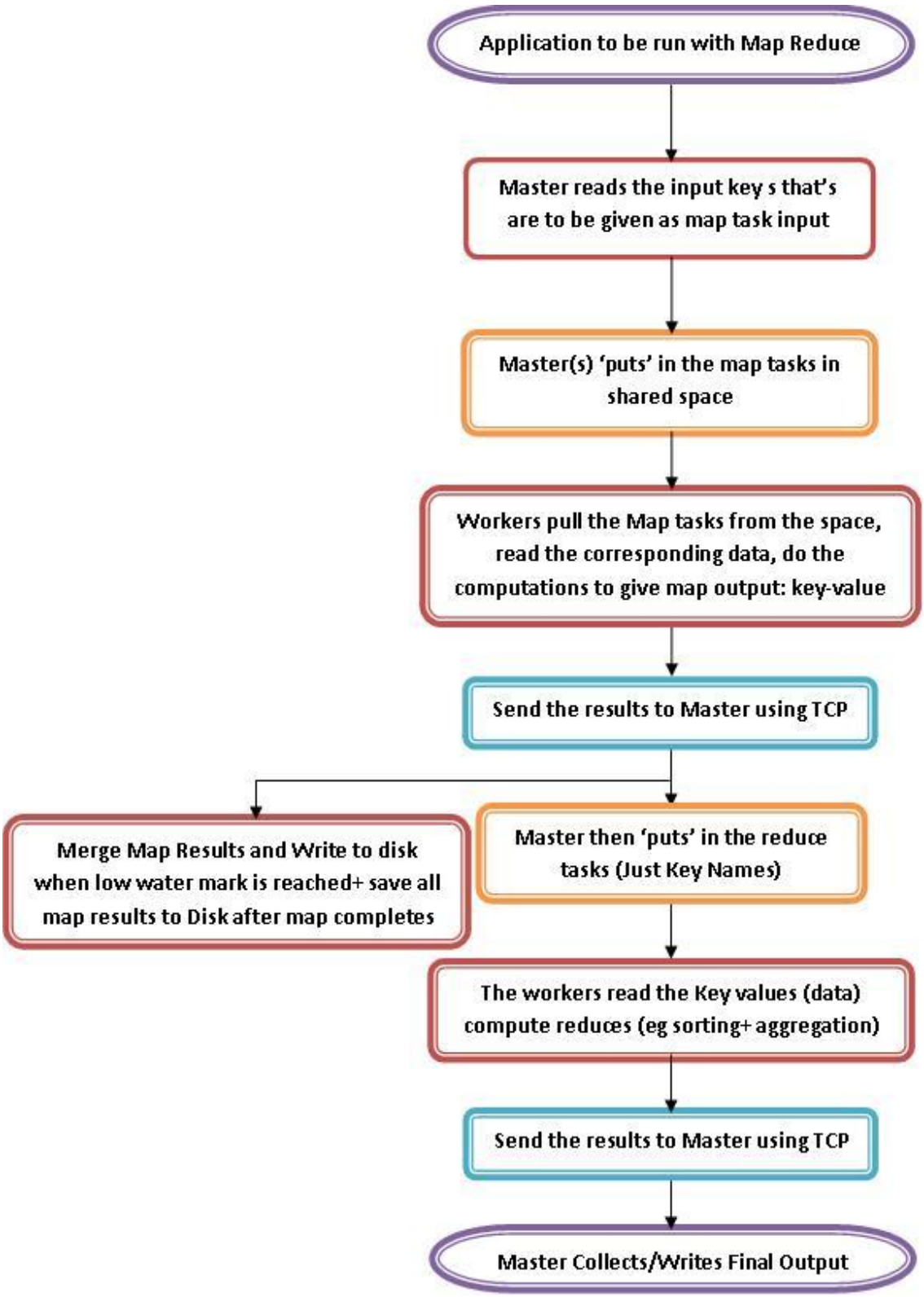

Figure 7: Flowchart for the data and execution flow in Comet based MapReduce.

1. **Comet Master**:
    a. Reads the input using the implementation given by the user to read data. Then creates Map tasks and puts them in the shared space using the ‘out’ primitive of Comet.
2. **Comet Worker – Mapper**
    a. Picks up map jobs from the space initially and executes the user supplied mapper implementation to get Key, Value pair from a given input.
    b. The map result, which is a Key and a vector of Values, is sent to the master.
3. **Output Collector in Master**
    a. The master periodically merges the different map results (partial aggregation and sorting is done)
    b. Once all Map tasks are done the reduce tasks are put into space (which are basically the values for a given key to do the final computation for the reduce)
4. **Comet Worker - Reducer**
    a. Picks up reduce jobs from the space and run the user supplied reducer implementation to get final aggregation of Key, Value.

4.2 Implementation

The MapReduce framework has been implemented in the application layer of the Comet. The code is platform independent and has been tested on both Linux based, Microsoft based clusters. Few primitive runs have also been done on Amazon Ec2 cloud. All the settings for a particular platform can be changed in the mapreduce.properties file. Depending on the configuration settings the master decides to launch workers on a

private network or an unsecure network like the Amazon Cloud. Also the disk storage cache is accordingly implemented, whose value can be set in the mapreduce properties file depending on the platform the application is to be run on.

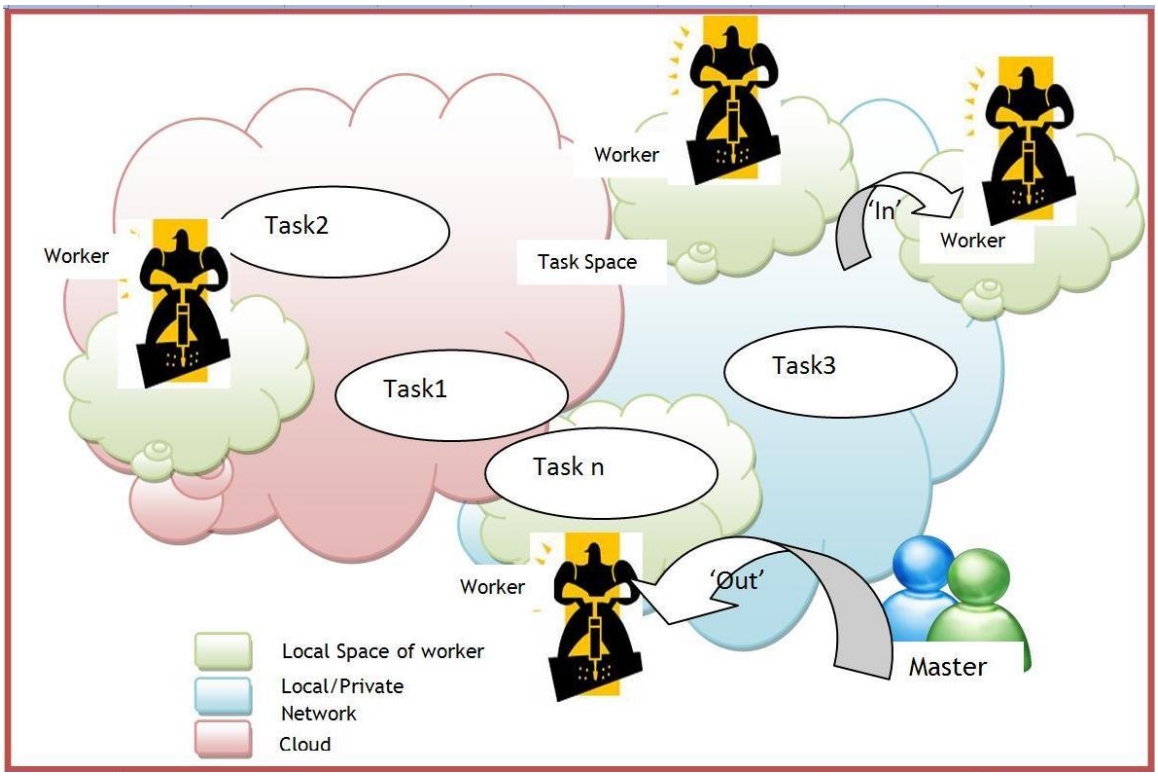

Figure 8: Different tuple spaces of Comet and their interaction.

As mentioned above the MapReduce abstraction is in the application layer of Comet, and is derived from the broad class of Master-Worker type of programming framework. The MapReduce master is responsible for scheduling, managing the input data, the reading, splitting and the distribution of it. The workers get the data and do the processing returning the results to the master. The master does the result tracking, aggregation and monitoring.

The MapReduce abstraction thus developed over Comet mainly consists of the following Java classes/interfaces.

4.2.1 Input Reader

This is an interface provided in the Comet MapReduce Abstraction for reading the input data. The user can implement this interface to design how the input should be divided and read in for the application.

It contains methods mentioned below:

public Object readFile( ) : This method can be used to implement the way the input file needs to be read for a given application.

public boolean isSplittable( ): This method can be used to decide that a given input file can be split in smaller chunks if required.

public boolean isNextChunkAvailable( ): This method can be used to check if there are any chunks of a given splittable file available to read.

public Object readNextChunk( ): This method would implement the code to read the chunks of a file that is split during input read

public void setInputFile(String fileName): This method is used to set the name of the file to be read currently to be processed.

4.2.2 MapReduce Master

The MapReduce Master in turn implements the Comet Master Framework and hence is based on the Master – Worker programming abstraction of Comet. The master is the one node which puts in tasks in the space, schedules and keeps a record of the tasks done, remaining tasks, and monitors tasks and coordination between the workers.

Some of the important methods, functions and classes included in the MapReduce Master are described below

When the Application starts and the Master is launched, it runs some initial routines to finish the pre-processing of setting up the Comet environment, reading the various properties file. It also reads the application level classes to be instantiated for the input reader, mapper and the reducer implementations.

public class TaskGenerateThread **implements** Runnable {}: This class as the name suggests generates the tasks by reading the input and converts each data input Comet MapReduce task tuples of map type data, which are put into the space using the 'out' primitive. It also consists of functions which handle the given list of files in the input directory

public void setResult(**int** id, Object data, String Sender): This method in the master is what is called by the TCPHandler class through the MapReduce Worker, when the worker finishes the computations and sends the result to master using a direct send TCP connection. This method does the tracking of the number of tasks which are finished and updates the task status data structure. Once all the map tasks are computed and all results

are obtained in the master, the reduce tasks are inserted in the tuplespace. Once the results for these reduce tasks are got back then the final result is written into files.

protected void mergeMapOutputs(): This method is called to merge all the map results and every time the free memory of the application goes below a certain fixed limit. This can be considered to be similar to the combiner function which does partial aggregation of the key value pairs. If the available free memory goes lower than a fixed limit (128M) then the master resorts to disk writes. Thus the data structure handling the merger of the map results can be in memory alone or both in memory and on disk depending on the size of the data processed. To handle the disk reads and writes a small utility function is written which syncs the data structure which has some key value pairs in memory and the rest on disk.

public void saveMapOnDisk(String cache): This is used to write the map results onto the disk. Each file in the map cache folder corresponds to each key and the data in the file is the value corresponding to that key.

class MapResultCache **extends** LinkedHashMap{}: This class has all the implementations for using disk as cache during the processing and computation. The details about this are described in the later part section that explains Disk reads and writes.

4.2.3 MapReduce Worker

The MapReduce Worker in turn implements the Comet Worker Framework and hence is based on the Master – Worker programming abstraction of Comet. The Worker is the one node which queries for tasks in the space, does the required computation and sends the results back to the master. The workers also have a capacity to interact with other workers; however this feature is not exploited in the current implementation.

Some of the important methods, functions and classes included in the MapReduce Worker are described below

The worker implements Runnable, and the worker thread continuously queries for tasks in the tuplespace. Whenever it gets tasks, its data is handled but computeTask method.

public Object computeTask(Object obj): This method is the most important method in the worker. It checks for the task type whether map or reduce or a Kill task and then accordingly calls the mapper, reducer or overlay leave methods. This method also takes into consideration, the EC2 nodes / third party involved which then obtain data through TCP connections from the master or the File Server.

public void sendResultToMaster(**int** taskid, Object data, String masterName): This method is used to send result to the master using direct sends via TCP connection.

4.2.4 Mapper interface

The Mapper interface is where the user can implement the mapper function for the application. In general, the worker depending on the whether it gets a map task, calls this interface. The method in this interface is

public void Map(String key, Object value, MapReduceOutputCollector collector): This is the main map method which would have the implementation of the mapping function to be carried out as a computation by the worker. It takes in the key and the corresponding value, and after the result is collected by the Output collector interface (explained further below in section 4.2.6).

4.2.5 Reducer interface

The Reducer interface is where the user can implement the reducer function for the application. In general, the worker depending on the whether it gets a reduce task, calls this interface. The method in this interface is

public void Reduce(String key, Iterator values , MapReduceOutputCollector collector): This is the main reduce method which would have the implementation of the reducer function to be executed out as a computation by the worker. It takes in the key and the corresponding value, and after the result is collected by the Output collector interface (explained further below)

4.2.6 Output Collector Interface

The Output collector interface as the name suggests is used for the purpose of collecting the outputs of the map and the reduce tasks. The map outputs are actually the intermediate results in majority of the applications; however they could be the final results in some application. The reduce tasks are generally the final outputs, however in some applications there could be a need of multiple iterative reduces.

The output collector consists of an abstract collect method as shown below. In the default implementation in the MapReduce worker it is mainly a hashmap of key – value pairs

```
public abstract void collect(Object key,Object value)
```

4.2.7 Disk Read/Write Manager

The MapReduce Paradigm is best exploited for very large data. As the size of data increases, the free memory of the application decreases. In order to avoid failures due to insufficient memory and still be able to handle large amount of data for processing, we resort to disk writes to store intermediate results and read through the written data for further processing if any. So at a given time unless there is an exclusive saving of results on the disk and in process data resides in-memory and on disk depending on the memory usage of the application at a given point.

Comet currently supports mainly a coordination system rather than a system for data exchange. Before deciding upon disk based caching of data we did a few experiments to

measure the Comet ‘out’ and ‘in’ times for a data size of 50MB and the time required to read the same data written to and read from disk in serialized form. We used local disks and shared NFS disks to read/write intermediate results.

From the graph below which compares Comet data transfer to Local disk data transfer we see that the disk read and write times are negligible in the order of 10 ms as compared to the Comet ‘out’ and ‘in’ primitives used to exploit the tuplespace as data cache. We also noticed that as the number of nodes in the overlay increases the log-n based routing of messages across the DHT causes a significant overhead.

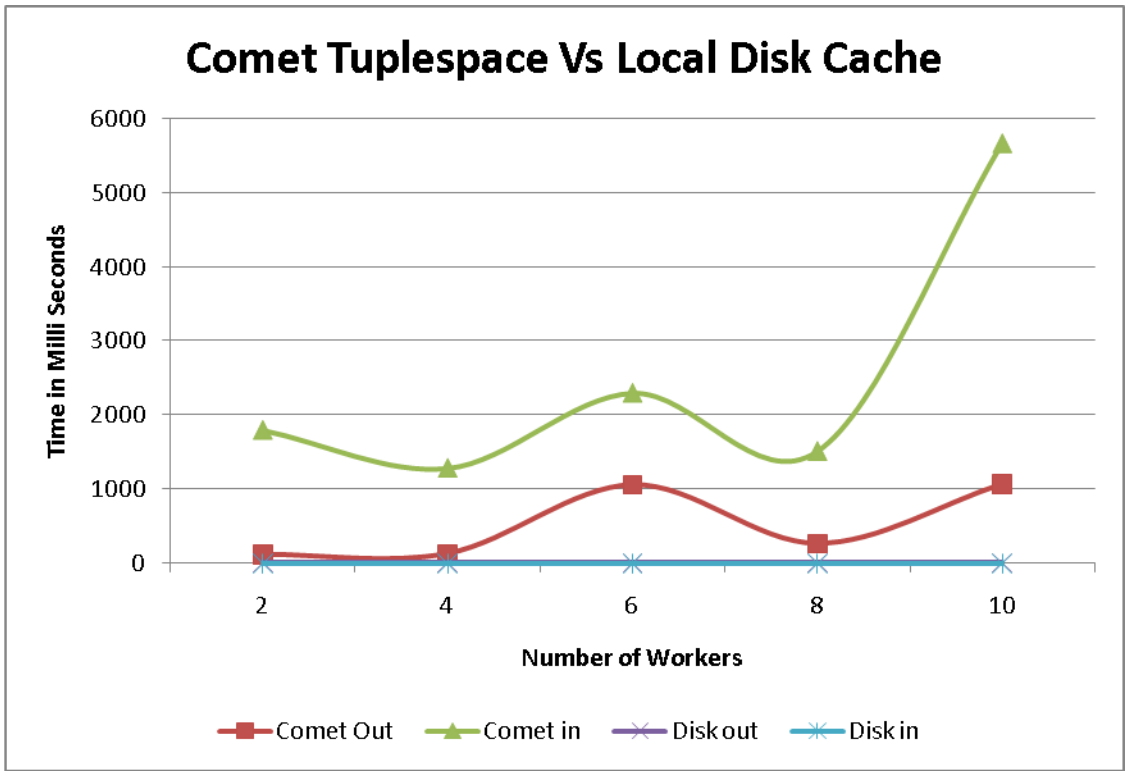

Figure 9: Time graph for Comet Tuplespace Vs Local Disk reads and writes.

From the graph below which compares Comet data transfer to a shared NFS disk data transfer we see that the combined disk read and write times are lower in the order of

1second as compared to the Comet ‘out’ and ‘in’ primitives that requires around 7 seconds for the data size of 50 megabytes.

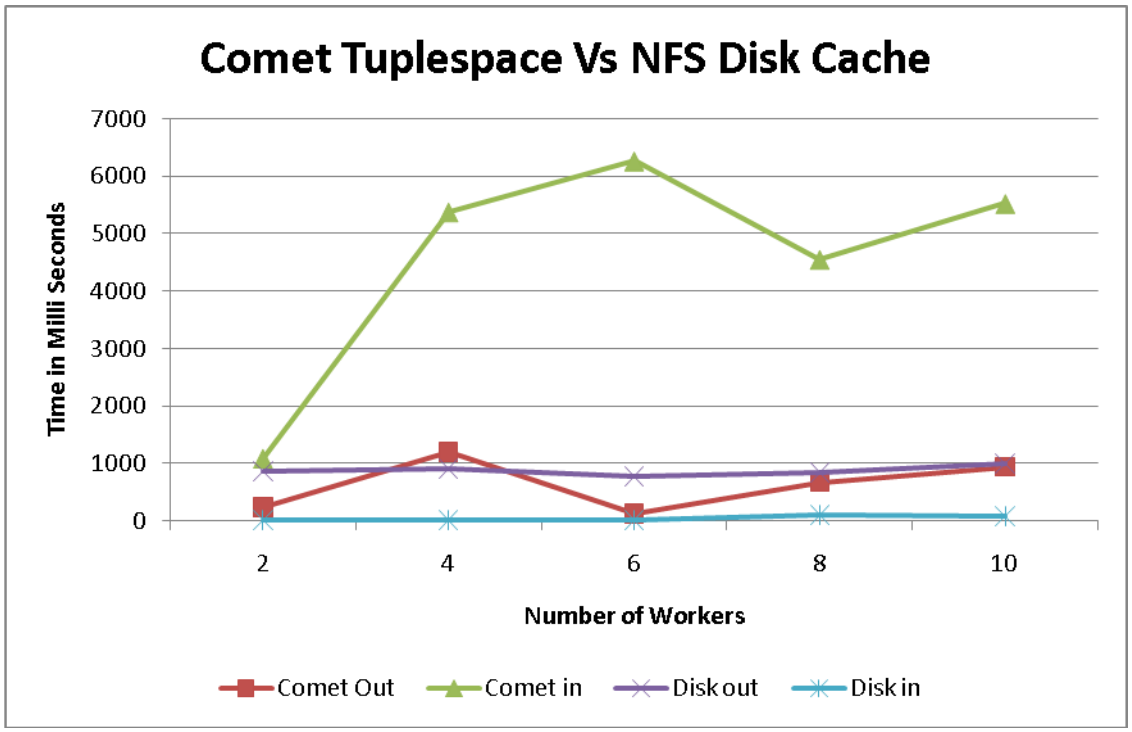

Figure 10 Time graph for Comet Tuplespace Vs NSF Disk reads and writes

The disk is used as cache and this cache is extended from LinkedHashMap. Using this, if a data structure grows very big in size it is distributed between the application memory and the disk. There is a special DiskIterator which extends Iterator class, which helps in reading through the data in the Linked Hash Map or the cache. For all the in-memory data, the Disk iterator inherits the base class ‘Iterator’ functions and for all the data on disk the Disk iterator does read/write/delete of files on the disk depending on the functions called.

The diagrams below show the flowchart explaining the logic for the disk reads and writes done by the master during an application run. This implementation of using the disk for

storing intermediate results enables the application to have a controlled and consistent use of memory without running into out of memory irrespective of the size of the data.

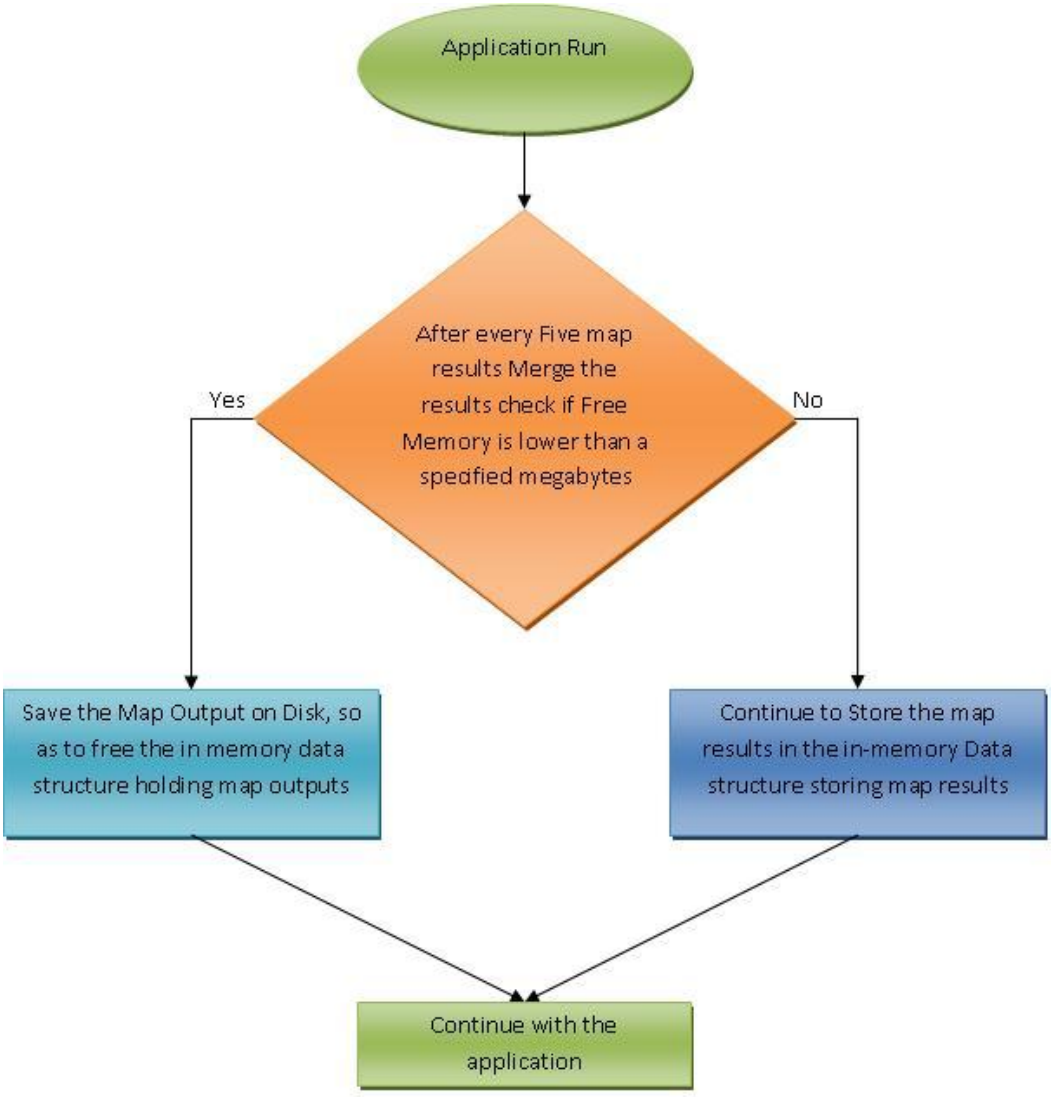

Figure 11: Flowchart explaining Disk write decision depending on free memory value.

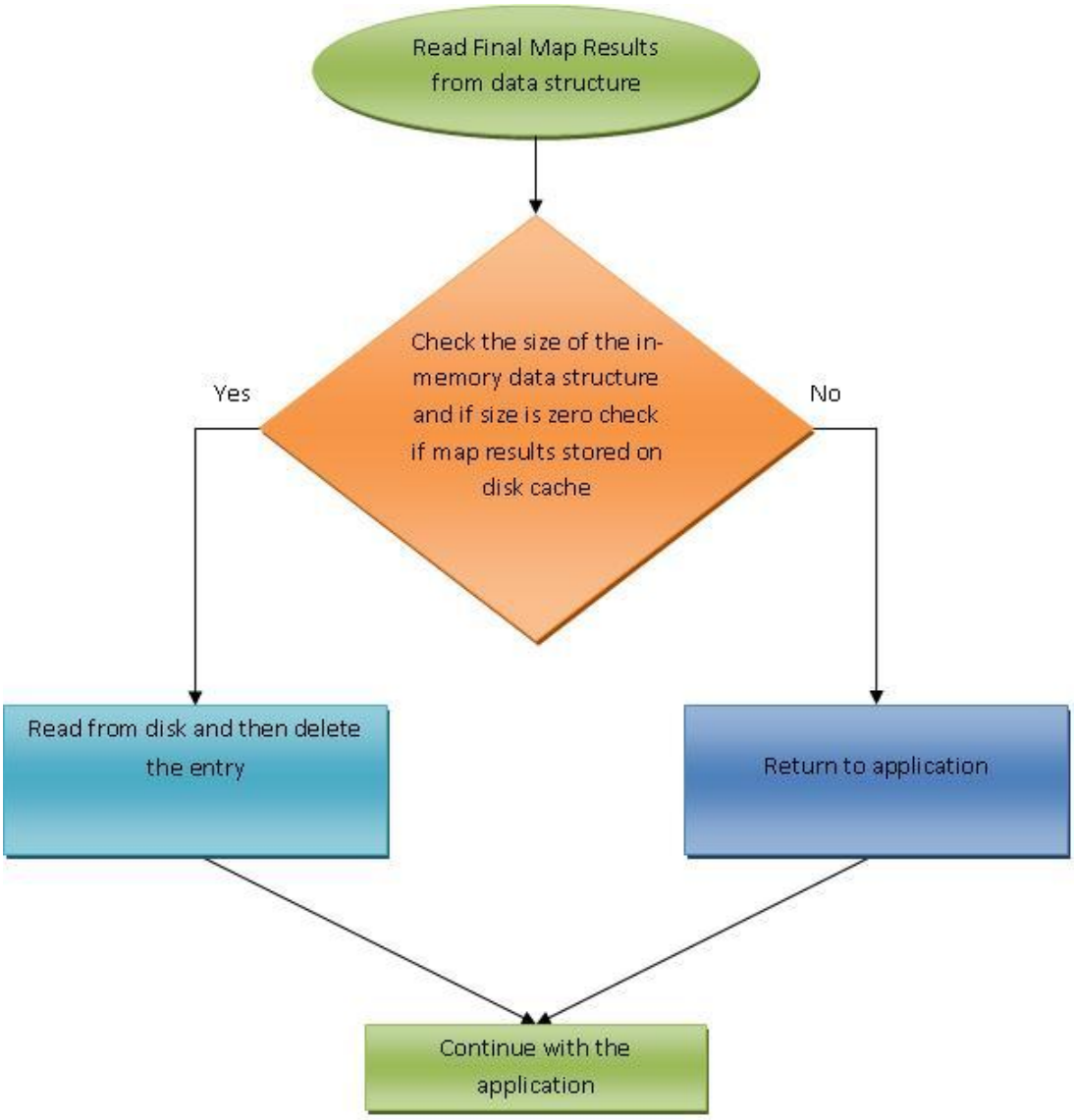

Figure 12: Flowchart explaining Disk read decision depending on free memory value.

4.3 Architectural Comparison of Comet MapReduce with Hadoop MapReduce

The Comet implementation of MapReduce Paradigm although in the nascent stage, it has been built trying to keep the different implementations as far as possible similar to the Hadoop MapReduce interfaces. There cannot be a direct comparison between the Comet and Hadoop Api's, as Hadoop API's are definitely more extensive and exhaustive in their support for different types of applications, different types of file systems, and for the more holistic reporting system. As compared to this Comet-MapReduce is still in the early stages of evolution.

In Comet we have abstracted the most basic and bare minimum implementations to support MapReduce type of applications.

Some of the API's currently supported by Comet and their counterparts in Hadoop are described below:

4.1.1 API's in Comet

1. Mapper Interface
    a. public void Map(String key, Object value, MapReduceOutputCollector collector)
2. Reducer Interface
    a. public void Reduce(String key, Iterator values , MapReduceOutputCollector collector)
3. Input reader Interface

a. public Object readFile( )

b. public boolean isSplittable( )

c. public boolean isNextChunkAvailable( )

d. public Object readNextChunk( )

e. public void setInputFile(String fileName)

4. MapReduce properties file

a. Sets the different configuration including the mapper/reduce/combiner class etc.

4.1.2 API's in Hadoop

1. Mapper Interface

a. void map(K1 key, V1 value, OutputCollector<K2, V2> output, Reporter reporter)

2. Reducer Interface

a. void reduce(K2 key, Iterator<V2> values, OutputCollector<K3, V3> output, Reporter reporter)

3. Input split

a. Has the InputSplit and Record Reader interface which do the job of splitting the input and reading.

4. Job Conf

a. Sets the different configuration including the mapper/reduce/combiner class etc.

4.4 Mining PDB Structures for Distance Information

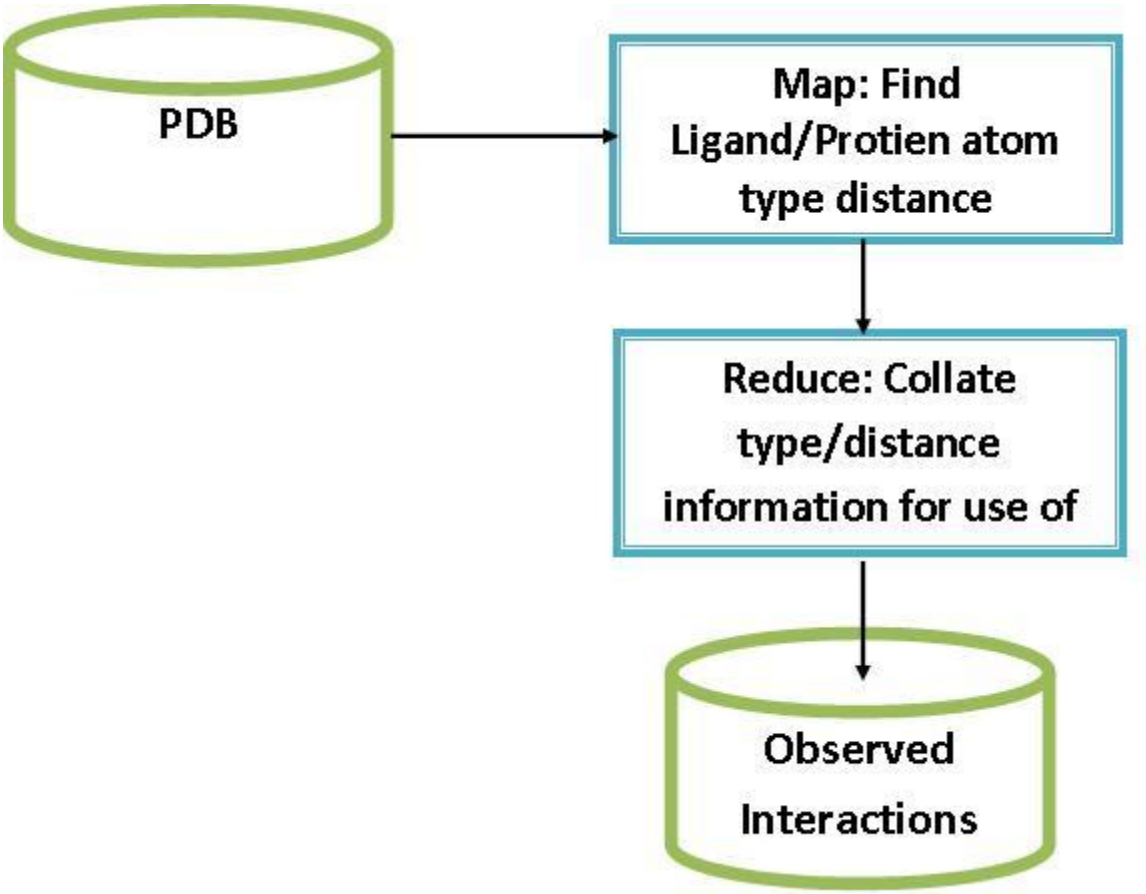

Figure 13: Process in calculations of the Protein Molecule Distances.

Consider the problem of protein-ligand binding. This is the notion that a small molecule (a drug, aka. the ligand) binds to a receptor or protein in the body. This binding event evokes a biological response, possibly the reduction of inflammation, pain relief, etc. Typically, there are a limited number of poses or configurations that this protein-ligand complex can assume (or possibly only one). Identifying this bio-active pose is a

tremendous challenge in drug discovery. Frequently, it is thought to be the lowest energy pose for either the protein or the ligand, but that is typically not the case. The complex can stabilize or make up for a higher energy conformation of the ligand, etc. Both the protein and ligand are three dimensional and flexible and therefore are constantly changing shape. This is really a multi-step problem. Starting with the ligand, one has to identify the bioactive 3D conformation of the ligand. Moving on then to the protein, the bioactive conformation is an even bigger challenge partially because the molecule is so much bigger and there are more possibilities. Lastly, if one could identify both the bioactive conformation of the ligand and the protein, then one is challenged to place the ligand in the correct location and orientation within the protein to produce the desired activity.

There are many ways to generate these poses, as well as many ways to try to determine which ones are (or may be) correct. Some of these calculations are computationally inexpensive, while others may be extraordinarily expensive. In theory, the more computationally expensive methods should yield results that are either more accurate (correct), or provide a higher confidence that they are at least reasonable. Unfortunately, this is not always true. One approach to this problem is to generate a large number of potential poses using a fairly inexpensive method and follow that up with a more expensive calculation to rank them in order of likelihood of being the bio-active pose. However, it is still easy to generate many more potential poses than one can afford to apply an expensive method to.

The idea behind the mapDistances code is to simply filter out some of these potential poses before trying a more expensive method. The Protein Data Bank (PDB) is a

database of known crystal structures and Nuclear Magnetic Resonance (NMR) structures, many of which are protein-ligand complexes. By mining the information contained in these structures, we are generating a scoring function based on known protein-ligand interactions. That is why we are processing through the entire PDB to extract out the interactions between small molecules and proteins. The output of the reduce code is the set of observed interactions after applying a distance bin technique. The distance bins simplify the comparison of a potential interaction to the actual observed interactions. This is just taking a set of observed distances and clumping them together. In order to include some of the atomic environment information, atom types are used rather than simply using the atomic element. This differentiates between aromatic and aliphatic carbons, nitrogens that are in an amide bond versus a primary amine, etc.

Once the counts of the observations are tallied, one can transform them into percentages of the time that a given interaction is observed. Some interactions are never observed and are therefore somewhat unlikely.

The application of the extracted scoring function is to take a set of potential poses and either filter or rank them in order to decide which ones to apply the more expensive method to. Basically, by running the same mapDistances code, one can build up the list of observed interactions and then compare these to the previously extracted interactions from the complete pdb. If the interaction was never observed in the pdb, a penalty is applied. If it was observed, then the percentage of the time it is observed in that distance bin then it gets positive credit. In its more simplistic form, one could imagine summing over the percentage of time an interaction in the theoretical pose was observed in the pdb.

Chapter 5

Experiments & Results

During the course of development of our system we observed several interesting points which lead us to believe that the MapReduce programming abstraction over Comet can be potentially used for medium sized data. We did extensive experiments on Microsoft platform. We did an evaluation of the results based on the time required, memory consumption and also tried a few comparisons, between Comet based MapReduce system developed by us and the Hadoop MapReduce distributed system with the same data set.

We observed that our system was could be effectively used to run MapReduce based applications. Some experimental results are explained below.

5.1 Scalability and Performance of Comet-MapReduce vs. Hadoop MapReduce

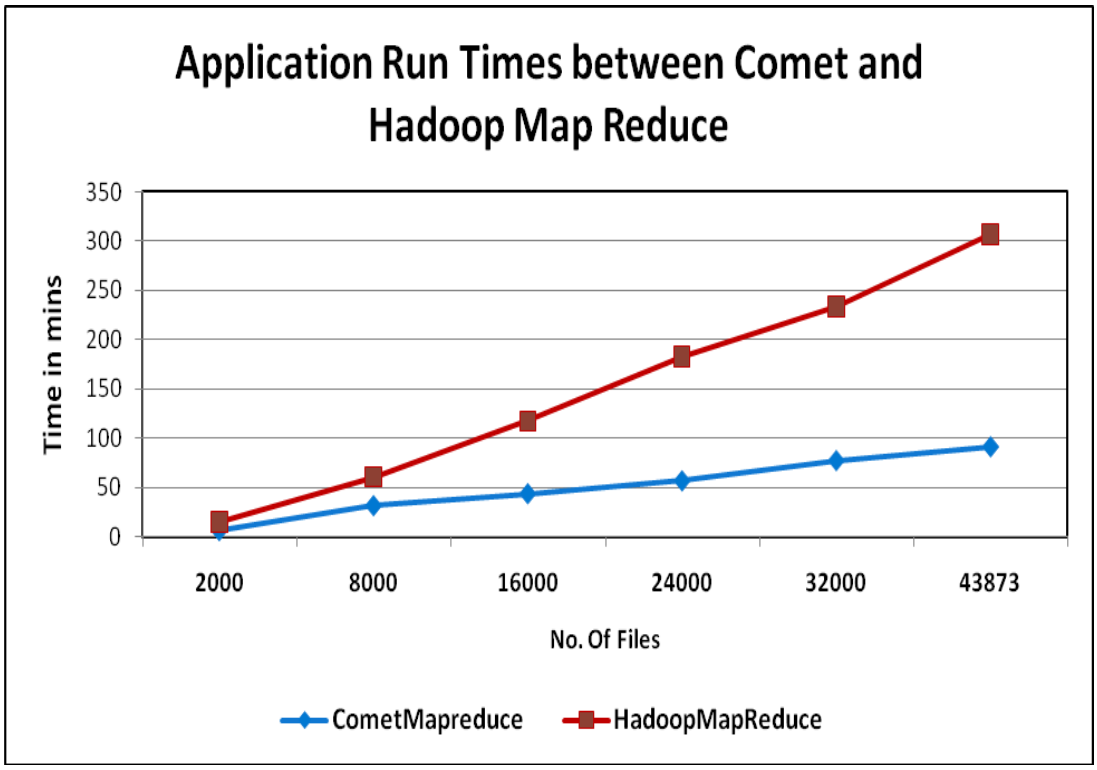

Figure 14: Total Application runtime for MapReduce over Hadoop Vs. MapReduce over Comet.

The above Figure 14 shows the total running time of the application for different set of files of the Protein distance extractor application. We can see that MapReduce on Comet out performs Hadoop MapReduce by taking just one third the time required by Hadoop to run over the entire data set. Thus a 3X improvement in the time is observed. We can also see that comet scales well for larger data. The size of the dataset was 6.5 GB. The tests were run with 2 workers per physical node. Hadoop was run with the same configurations as already been used by BMS. Comet was run with default system properties. Since the Comet MapReduce is built around in-memory operations the maximum allowed memory

for both the systems were configured to be the same. This was done through the **mapred.child.java.opts** property of Hadoop and using the command line option for the Java VM while running Comet.

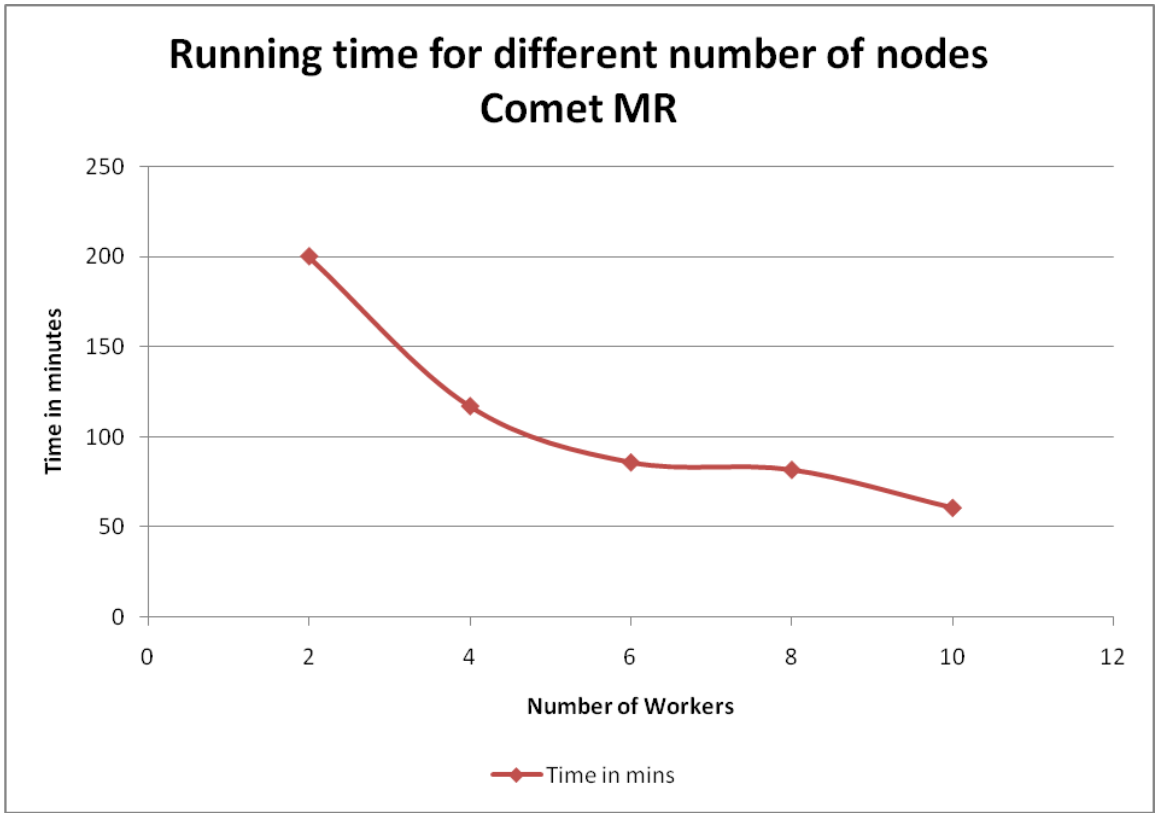

Figure 15: Application running time for a fixed load and varying number of nodes.

The above Figure 15 shows the application run time for fixed size of data. Here we ran 24k files by varying the number of workers which in turn is obtained by varying the number of nodes. This is the classical graph expected in case of parallelization of an application where as the number of machines increase for the given constant size data the computation is distributed across the nodes and hence the running time decreases. From the graph we see that for a data size of 24k files, with 2 workers the time taken for the

application run is over 3 hours (211 minutes) and the run with 10 workers finishes the same processing in about an hour.

Reason for Performance Differences:

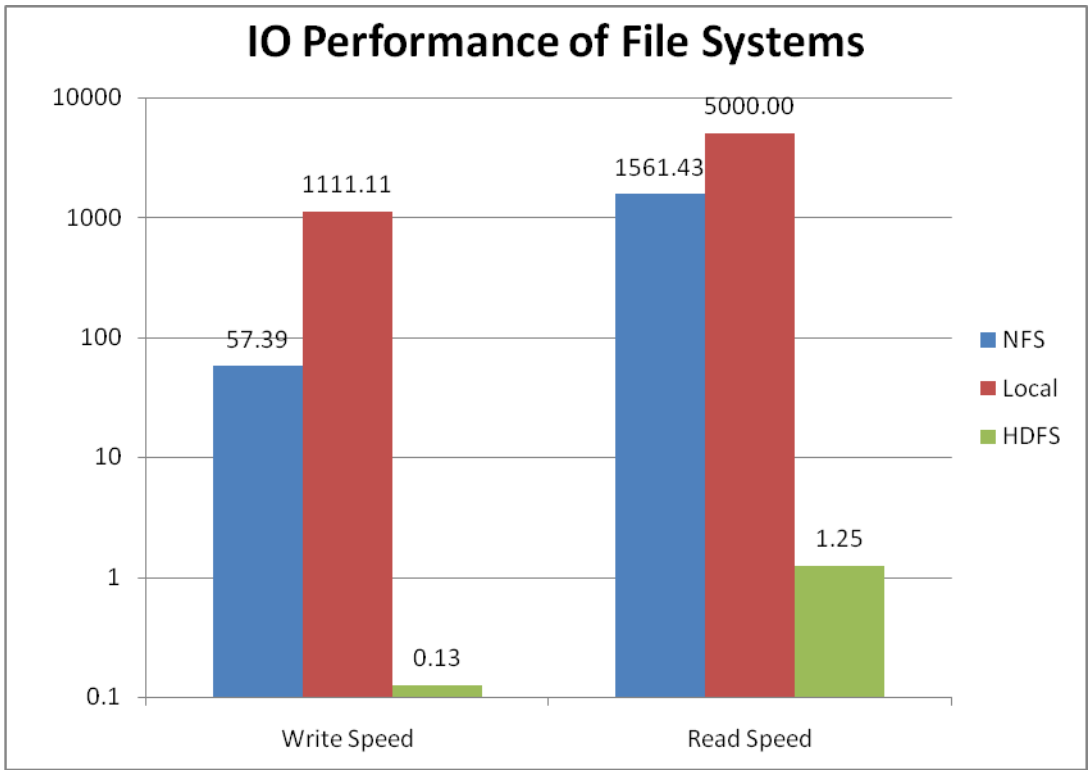

Figure 16: IO Performance of the File systems.

From the Figure 16 we can see that a combination of local and NFS file systems outperforms HDFS IO. This is one of the primary reasons for Comet to perform better than Hadoop MR.

We ran some simple tests to write and read 10 – 50 MB of data from files (about 1000 times) on the different platforms and took the time stamps. The graphs above have the average values of the 1000 reads and writes.

5.2 Memory Metrics on Comet MapReduce

5.2.1 Master Memory Metrics

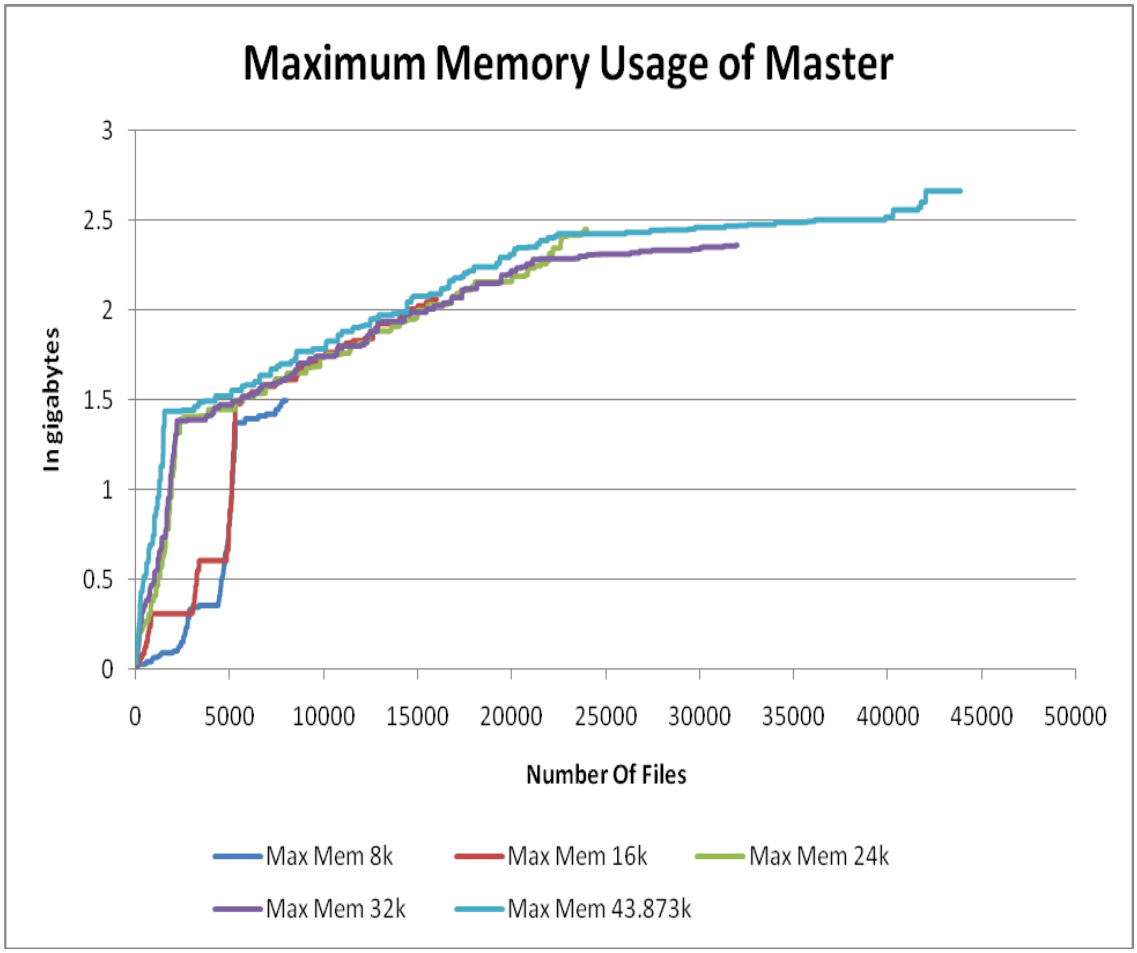

Figure 17: Memory usage trends of the Master node for varying data size.

The Figure 17 above shows the memory consumption in a master node which is also running 2 worker peers simultaneously. We can see that the as the data size increases the memory consumption curve flattens around 2.5GB. The curve flattening is due to the implementation of the disk based hash map described in section 3.2.7 which tries to maintain a specified threshold amount of free memory. The tests for the above graph were run with 3 GB of virtual memory and the low watermark for the memory was

256MB. Thus we can see that Comet based MapReduce is reliably scalable over larger data sizes.

5.2.2 Worker Memory Metrics

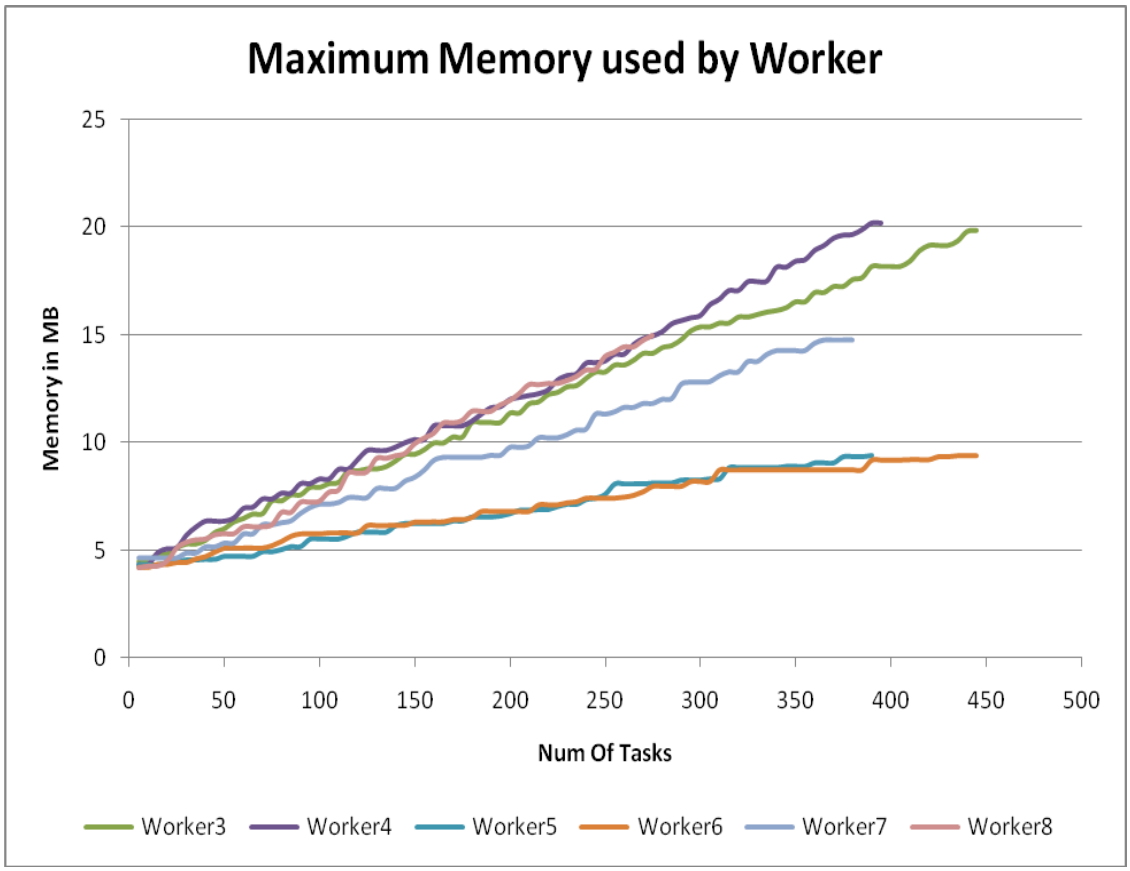

Figure 18: Memory usage trends of the Worker nodes for a given data size

The Figure 18 above shows the memory consumption of workers on different machine for and experiment run on the entire data set. We ran 2 workers per node, hence from the graph you can see that there are sets of 2 with similar reading which indicates that those two workers were on the same node. From the above graph we can see that for the

workers consume very little memory compared to the master. It starts from around 5MB and goes to a maximum memory of about 20MB.

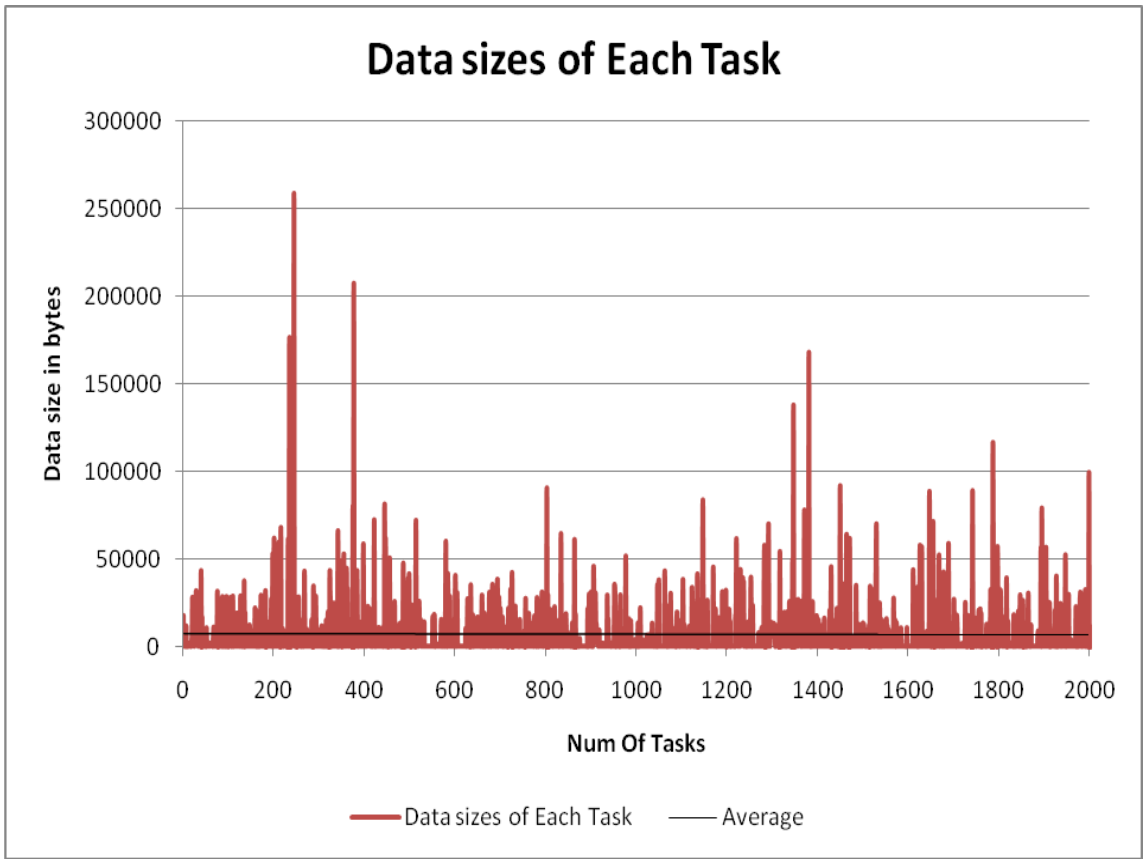

Figure 19: The average data size of each task (map/reduce) for the Protein Data Bank dataset.

Tthe above graph in Figure 19 shows that we can see that the data size handled by each individual map and reduce task is in the order of a few tens of Kilobytes. Hence the memory usage for each worker will be lower than the master and hence the worker nodes can be run on commodity PCs with low RAM. This also shows that as the map ouputs are small in size they need not be written to disk upfront, but can be held in memory for further computations.

5.3 Load Balancing

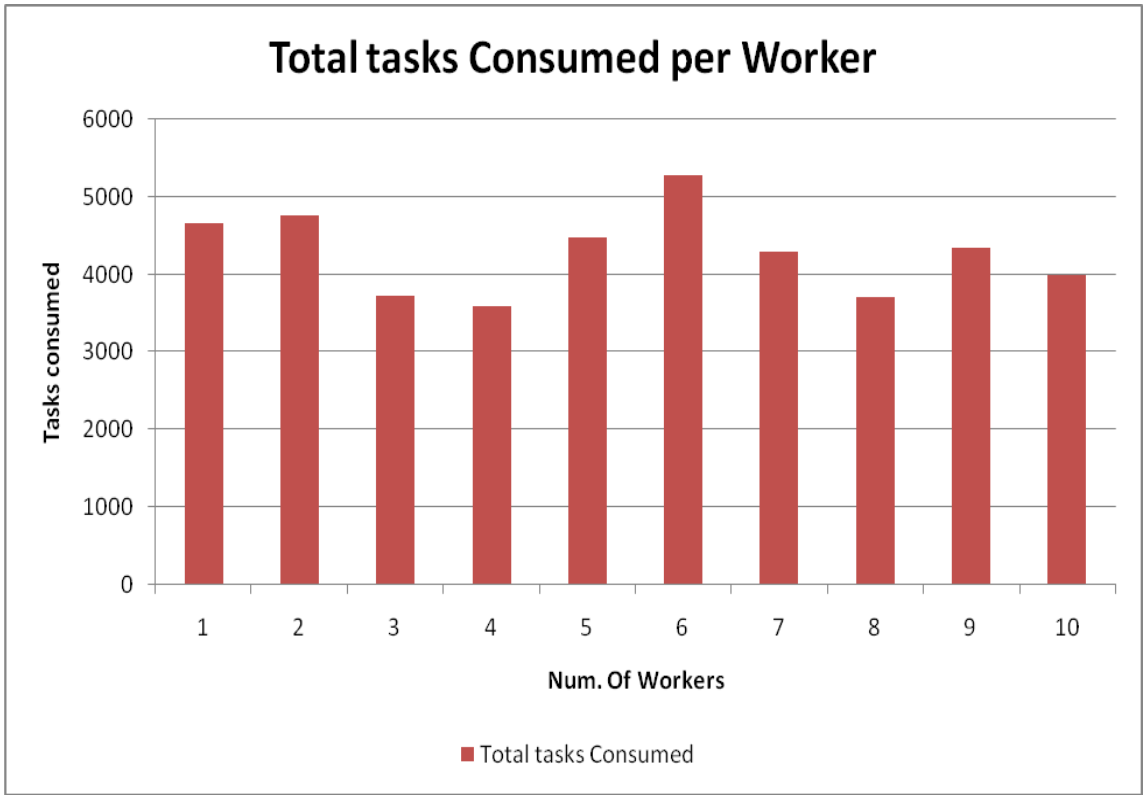

Figure 20: The distribution of tasks per worker - showing the load balancing provided by Comet.

The above graph gives the distribution of tasks across all workers for a full data set run. This shows data for 10 workers running on 5 machines. The load distribution can be evened out by giving a correct number for property chord.ID_BITS which is used for routing of the queries and the tasks distribution. We can see the load balancing if fairly equal with an average of 4350 tasks per worker for a total of approximately 44000 tasks. The tasks taken in by each worker are not the same as this depends on the size of data related to each task. So if some files have larger processing time, then the workers processing such files will consume lesser number of tasks. So if there is another worker

which has already finished its computation then it can pick up the next available task in space.

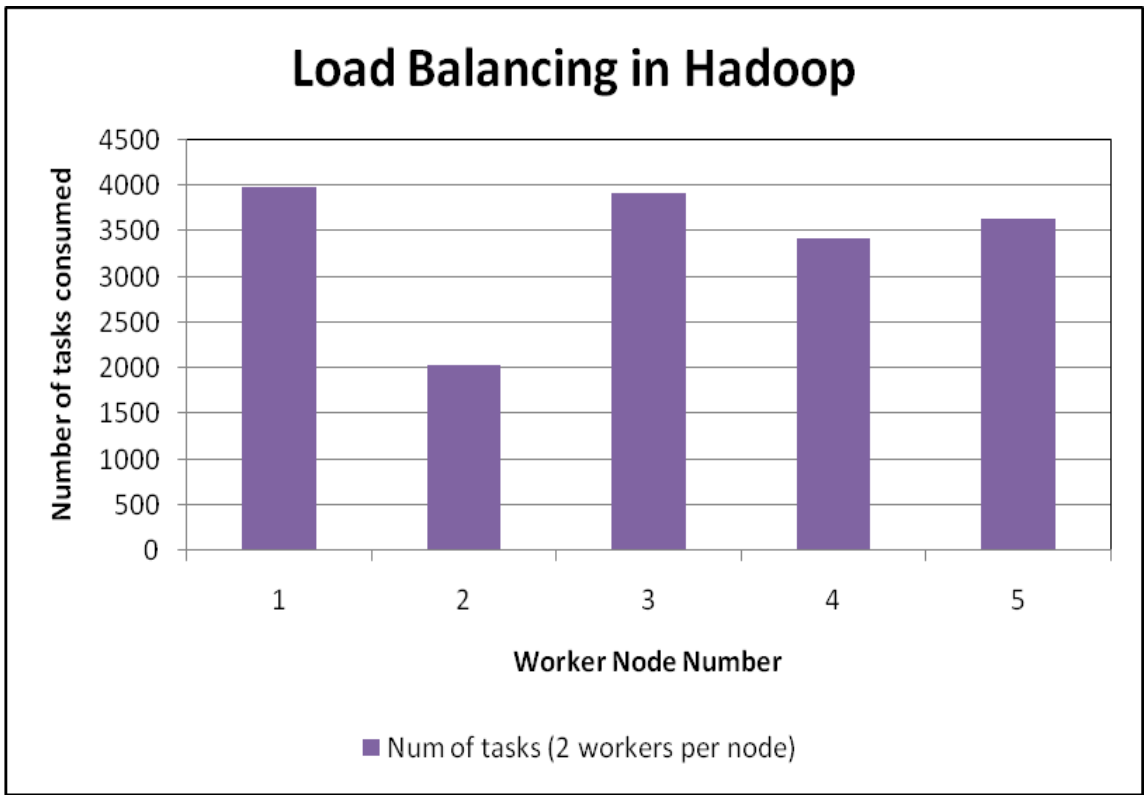

Figure 21: Task distribution per node in Hadoop each running 2 map tasks is equivalent to 2 workers per node.

From the graph in Figure 21 we see that in case of the Hadoop run the load distribution per node is not completely balanced and there is one node that is underutilized. We observed that the data stored by the 'hdfs' on that node was significantly low than the other nodes and as Hadoop tries to localize computation closer to data storage, we see the imbalance.

Thus we see that the pull based MapReduce model on Comet is efficient by avoiding data storage or different sizes of data to be bottlenecks in computation. At any given point there is an optimum utilization of resources.

Chapter 6

Summary, Conclusion & Future Work

6.1 Summary

The primary objective of the research presented in this thesis was to investigate and study behaviors and limitations of MapReduce infrastructure in the case of small to moderate data sets and develop a coordination and interaction framework over Comet to complement MapReduce/Hadoop. Another objective was to prove the feasibility of the application by developing a model application from a real world scenario.

The key contribution of this thesis is the MapReduce conceptual architecture model and implementation infrastructure programming abstractions for supporting applications that a can be executed on the basis of the MapReduce programming paradigm. The framework is built on Comet which employs fully decentralized architecture and provides a global virtual shared-space abstraction that can be associatively accessed by all peer nodes in the system. Thus the research enabled to exploit a pull based scheduling of tasks as well as stream based coordination and data exchange.

A prototype system was developed using a real world application. Bristol Meyer Squibb (BMS) presented a problem statement with the application that they used in their protein distance analysis and we proposed an efficient implementation over Comet and the

Master Worker programming paradigm already existed in Comet . After studying their existing application run over Hadoop, we built a similar interface over Comet. We developed it on the Master Worker abstraction utilizing the tuplespace for coordination. We have got impressive results for the dataset given by BMS, which also proved that there could be more applications based on the same concept ported over to Comet MapReduce. The overall time efficiency showed atleast a 50 % improvement with the Comet MapReduce run as compared to the same implementation over Hadoop. The Memory consumptions have been significantly under control due to the use of the disk cache in Comet. At the workers the memory usage flattens at around 20 MB. Results also showed efficient load balancing of tasks which further enhances the performance of the pull based implementation of Comet infrastructure

6.2 Conclusion

MapReduce programs are designed to compute large volumes of data in a parallel fashion. All data elements in MapReduce are split into key-value pairs which are independent of each other and hence can be processed independently. The efficiency and performance can be improved by running these tasks in an embarrassingly parallel environment. There have been quite a few such frameworks existent but they mainly rely on the distributed file system for their processing and storage of intermediate and final results. This read and write into the distributed file system adds an overhead which becomes significant in case of smaller data sizes of few gigabytes. This research presented that a similar programming paradigm which was developed over the Comet

infrastructure. The pull based model of querying for work by the worker nodes and use of the distributed hash table and space filling curves for routing and load balancing improve the efficiency of the framework. The solution can be used to accelerate the computations of medium sized data by delaying or avoiding the use of distributed file reads and writes.

Our System's interfaces are similar to the Hadoop MapReduce framework, to make applications built on Hadoop easily portable to Comet-based framework. The details of the implementation and evaluation of an actual pharmaceutical problem, with its results have been described.

From the experiments and results seen in the Chapter 5, it can be seen that the MapReduce programming paradigm implemented on Comet has been successfully evaluated with an implementation of the real world application (application with pharmaceutical computations from Bristol-Meyers Squibb).

6.3 Future Work

The current MapReduce abstraction over Comet is very naïve. It has just the basic interfaces that are most essential for such programming abstraction; mainly the Mapper, the Reducer, the Input Reader and the Output collector. All these interfaces can be made more flexible and exhaustive.

More functionality for scheduling and reporting and job monitoring can be extended into the framework. Currently the Disk Manager is a simple implementation to write and read data from the disks; interfacing of the distributed file system provided by Hadoop into Comet MapReduce would give a more stable distributed read and writes to the application and also to be able to support really large data sets of terabytes.

Currently all the tasks are poured into the tuple space by the master. With the added functionality in Comet where the workers can also ‘put’ in tasks, the efficiency and performance of Comet based MapReduce can be improved. The intermediate map outputs which are currently sent back to the master and stored in the master can then be kept in the worker. Once all workers get a notification for map tasks completed, the workers can then put in the intermediate map results as reduce tasks and key based queries should be given to the space so as all the values for a given key are obtained by a single worker to further complete the reduce computations. The final result can then be sent back to the master.

References

[1] Cristina Schmidt and Manish Parashar. Enabling flexible queries with guarantees in p2p systems. *IEE*E *Networ*k *Computing*, *Specia*l *issu*e *o*n *Informatio*n *Disseminatio*n *o*n *th*e *Web*, (3):19–26, June 2004.

[2] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEE*E *Transaction*s *o*n *Knowledg*e *an*d *Dat*a *Engineering*, 13(1):124–141, 2001.

[3] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters OSDI 2004

[4] Jaliya Ekanayake and Shrideep Pallickara, MapReduce for Data Intensive Scientific Analysis, Fourth IEEE International Conference on eScience, 2008, pp.277-284.

[5] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, D. Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. Proc. ACM SIGMOD International Conference on Management of data (SIGMOD 2007), ACM Press, 2007, pp. 1029-1040.

[6] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Sch¨olkopf, J. Platt, and T. Hoffman, editors, Advances in Neural Information Processing Systems 19, pages 281–288. MIT Press, Cambridge, MA, 2007.

[7] CGL MapReduce - http://www.cs.indiana.edu/~jekanaya/cglmr.html

[8] Lammal, Ralf. Google's MapReduce Programming Model Revisited. http://www.cs.vu.nl/~ralf/MapReduce/paper.pdf

[9] Open Source MapReduce: http://lucene.apache.org/hadoop/

[10] Google MapReduce Introduction : http://code.google.com/edu/parallel/mapreduce-tutorial.html

[11]Hadoop documentation: http://hadoop.apache.org/common/docs/current/api/overview-summary.html

[12] Hadoop Wiki : http://wiki.apache.org/hadoop/

[13] Hadoop Tutorial: Downloaded from the official Hadoop Distribution cd

[14] G. L. Steel, Jr. “Parallelism in Lisp,” SIGPLAN Lisp Pointers, vol. VIII(2),1995, pp.1-14.

[15] Amazon EC2 : http://aws.amazon.com

[16] Microsoft HPC : http://www.microsoft.com/hpc/en/us/default.aspx

[17] Running Hadoop by Michael Noll: http://www.michael-noll.com/wiki/Running_Hadoop_On_Ubuntu_Linux_(Multi-Node_Cluster)

[18] Hadoop over Windows : http://v-lad.org/Tutorials/Hadoop/20%20-%20upload%20data.html
http://hayesdavis.net/2008/06/14/running-hadoop-on-windows/

[19] Z. Li and M. Parashar, "A Computational Infrastructure for Grid-based Asynchronous Parallel Applications," Proceedings of the 16th International Symposium on High-Performance Distributed Computing (HPDC), Monterey, CA, USA, pp. 229, June 2007.