Investigating the Use of Autonomic Cloudbursts within the MapReduce Framework

By

SAMPRITA HEGDE

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

May, 2011

ABSTRACT OF THE THESIS

Investigating the Use of Autonomic Cloudbursts within the MapReduce Framework

By SAMPRITA HEGDE

Thesis Director:
Professor Manish Parashar

Today MapReduce framework is increasingly becoming a popular programming paradigm for data intensive computing, especially when there is ad-hoc data to be processed. In MapReduce programming paradigm, computation is done in two stages - a map stage and a reduce stage. The users simply have to provide a 'map' and a 'reduce' function and the underlying framework handles parallelizing and distributing the computation to worker nodes. Currently, the existing MapReduce frameworks work like a batch processing system where the cluster size is assumed to be static. We have developed a new objective-based scheduler which:

1. Provides both deadline and budget based scheduling capability
2. Provides cloudbursting capability where a computation can "burst" out to cloud whenever the existing datacenter is not capable of meeting the objective.

Using these features, it is possible to run any MapReduce application subject to a user objective on any existing cluster by leveraging utility cloud resources. In this thesis, we use the Comet coordination engine and the MapReduce framework which is built on top of Comet Engine. The new autonomic scheduler works with the MapReduce Framework

and manages the cluster as well as cloud in order to meet computation requirements. We have investigated the use of cloudbursting for MapReduce applications. We found that it is possible to run the application subject to both time and budget based objectives and successfully complete a job by efficiently using datacenter as well as cloud infrastructures.

Acknowledgement and Dedication

I would like to thank my advisor, Dr. Manish Parashar, for giving me an opportunity to work on such an interesting problem, for his enthusiasm, inspiration and encouragement during my research at The Applied Software Systems Laboratory (TASSL). I am grateful to my colleagues at TASSL, all the staff members 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 brother and my husband, for their understanding, endless encouragement, and love.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

Introduction

Today, data intensive computing is becoming increasingly prevalent. To meet the computing needs, various parallelization techniques and parallel algorithms are becoming key aspects of the modern computing world. Also, as more and more multi-core processors are emerging it is increasingly becoming necessary to exploit the parallelism inherent in the processor architecture. Most of these algorithms are Single Program Multiple Data algorithms (SPMD).

One of the techniques in which these SPMD programs are implemented is MapReduce. Many computations like processing of web logs, crawled documents, computing inverted indices, various representations of graph structures of web documents are conceptually very straight forward. However, input data for these computations is so large that it has to be distributed across a large number of machines. The issues of parallelization, data distribution, synchronization, failure handling etc make this conceptually simple problem a very complex one with large amount of code to deal with these issues. MapReduce programming framework provides an effective and simple solution to these problems. A user has to simply provide a 'Map' and a 'Reduce' function and the underlying framework takes care of parallelization and most other issues that arise when the computation has to be distributed.

MapReduce framework was made popular by Google [1] to support distributed computing of large data. Map-Reduce abstraction is inspired by the *map* and the *reduce primitives* found in Lisp and many other functional languages. In MapReduce, input data is divided into many logical records. Each record consists of a key-value pair. The Map function, which is provided by a user, takes an input record of key value pair and produces an intermediate set of key-value pairs. MapReduce library takes care of grouping the values belonging to the same key. The Reduce function takes the intermediate key and the set of values associated with it and does the computation to possibly form a smaller set of values. From the user's perspective the problems now become simpler and all other complexities that arise out of parallelization are handled by the MapReduce framework.

1.1 Problem Description and Motivation

In modern computing, MapReduce is widely used to process large volumes of data in parallel. Although MapReduce framework has been inspired by the functional language primitives, the purpose of the framework is to enable large scale parallel processing on commodity machines.

MapReduce libraries are available in Java, C#, Python, C++ etc. One of the popular implementation which is publicly available is Hadoop [9]. The implementation of Hadoop MapReduce framework has been inspired by Google MapReduce framework [1]. It makes extensive use of a distributed file system to store all the input, output and

intermediate data. Although this strategy makes the framework extremely robust, it introduces additional overhead in terms of file read/writes. As such, the system is not very efficient when it comes to small data-sets. At present, all these frameworks essentially function like batch processing systems. A user submits computations in the form of jobs and these jobs are scheduled by the MapReduce library based on different scheduling policies like FCFS (First Come First Serve), Fairshare scheduling etc. But all of the existing frameworks [9] [16] today assume a static and constant cluster size. The static scheduler attempts to schedule tasks so as to reduce data transfer on the distributed file system which is running underneath. It does not attempt to estimate the job completion time. Also, there is no possibility of setting deadline or any kind of user objectives for job completion. If a job needs to be completed within a deadline, it is constrained by the resources available to it. A deadline might be needed for several reasons.

Consider the following use case:

One of the common jobs in web based organizations is periodic processing of web logs. Suppose the web logs and click stream logs needs to be processed every hour no matter how large the data is. Obviously, the size of the data depends on the web traffic at that hour. Hence in order to meet the deadline, it becomes necessary that sufficient resources are made available beforehand even for the highest anticipated web traffic so that these data can be processed within the stipulated time. This is usually done by recording historical data of maximum web traffic and provisioning more than enough resources for processing that data. But this happens only occasionally and often it happens that only

part of the resources is frequently used. Provisioning for these extra computing resources and maintaining them is extra overhead in terms of both money and man-hours.

We noticed that if there was a scheduler that dynamically calculates resource requirements and provisions them for MapReduce computing, then the cost involved in the extra resource provisioning can be reduced significantly. Availability of Cloud Computing infrastructure to use computing resources only when needed suits effectively to these needs.

In this thesis, we have implemented an objective based scheduler for MapReduce framework which estimates the resource requirements and dynamically provisions the resources for a job. The objective might be time based or budget based. We have designed this scheduler to work with a MapReduce framework that is built on the Comet Co-ordination engine [2] [11].

CometCloud is an existing framework that provides decentralized virtual shared space coordination for running distributed applications on large clusters. CometCloud also offers Cloudbursting and Cloudbridging capabilities. Cloudbursting refers to on-demand scale up and scale down and scale out of the resources. Cloudbridging refers to the ability to work with different kinds of resources (public cloud and private datacenter) at the same time. The MapReduce framework is an application built on top of Comet to support MapReduce applications. We have implemented a scheduler to support autonomic cloud bridging and cloudbursting for MapReduce framework in CometCloud. We also evaluate the effectiveness of the scheduler using a real world application for Protein Data Bank

mining developed by Bristol Myers Squib. We demonstrate how cloud bursting can be used to effectively perform large MapReduce computations using the limited data center resources and scaling out to the clouds when necessary, thus saving a lot of monetary investment in data center infrastructure.

1.2 Overview of Comet Based MapReduce Framework and the Autonomic Scheduler

CometCloud [2] is a scalable content based coordination space for distributed environments. It provides a scalable tuple space abstraction for communication, synchronization and distributed processes. The Comet space is constructed from a multi dimensional information space. The application layer provides many programming paradigms and one of them is a master/worker framework. In this programming paradigm, the master generates tasks and places them on the comet virtual shared space. The workers pull such tasks from the space and do the necessary computation. The task can have different attributes which are specified as a task tuple. A task tuple consists of a simple XML string describing various attributes.

This programming paradigm has been used in the MapReduce framework as well. The MapReduce framework built on Comet provides a conceptual architecture model. It provides a *map* and a *reduce* interface, very similar to the Hadoop MapReduce framework [9] which is a publicly available open source implementation of MapReduce framework. Both Hadoop and Comet MapReduce require the user to implement the *map* and the *reduce* functions.

The main interfaces of the Comet MapReduce framework are

- **Input Reader** which is responsible for reading the input data
- **The Mapper** which does the Map computations
- **The Reducer** which does the reduce computations.

The MapReduce Master gets input from the input reader and generates tasks and inserts them to the comet space. Workers "pull' the tasks from such a comet space. The workers then determine the type of the task (Map / Reduce) by reading the task attributes in the task tuple and perform computation accordingly.

1.3 Contribution

The goal of this research is to enable autonomic cloudbursting for Comet based MapReduce framework using an objective based scheduler. Based on a given objective, the scheduler has to perform following jobs:

i. ***Estimation:*** Based on the objective provided by a user, estimate the time required to compute a job. Workers may span across clouds. Hence it is necessary to take into account that different clouds will have nodes with different processing speeds and memory associated with them.
ii. ***Scheduling:*** Based on the estimation, a suitable resource class is selected and the number of workers needed to complete the job in that resource class is decided.

iii. ***Monitoring:*** The progress of the job is continuously monitored and it is compared with the estimated job progress. Its results are sent to the adaptation module.

iv. ***Adaptation:*** Based on the results of the monitoring module, the remaining tasks are re-estimated and the number of workers and the resource class is decided. The remaining tasks are rescheduled on these resource classes.

In this research, we have implemented a unique scheduler which can effectively fulfill the scheduling needs of a MapReduce application. We have demonstrated that this MapReduce application can be seamlessly integrated and run on both public clouds, like Amazon EC2 and private datacenters, like the Rutgers CAC datacenter. We have also demonstrated that a MapReduce application can be completed according to user objectives and without the need for over-provisioning of resources and thus reducing significant infrastructure costs.

To summarize, the main theme of this thesis are:

i. Understand the need for an objective based scheduler for MapReduce applications and also understand the challenges involved in developing such a scheduler.

ii. Design and develop a generic scheduler for completing a MapReduce job according to a given user objective. This scheduler runs within the MapReduce master and continuously monitors the job progress in order to determine resource needs.

iii. Develop a cloud agent to control different cloud and datacenters.

iv. Evaluate the scheduler by running a MapReduce application subjected to a user objective and see how efficiently the resources are allocated and computation is performed.

v. Evaluate various scheduling policies which determine how the different clouds and private datacenters are provisioned during the course of the computation.

Chapter 2

Background and Related Work

2.1 MapReduce Basics

MapReduce [12] is a programming framework that enables automatic parallelization of large scale data processing. It provides efficient parallelization for large clusters of commodity machines. In MapReduce, computation is done in terms of key value pairs. The *map* function provided by a user takes an input pair and provides a set of intermediate key/value pairs. The MapReduce library groups together all the values associated with a single key and then passes them to the *reduce* function which is also provided by the user. The reduce function accepts an intermediate key and all the values associated with it. It merges these values to possibly form a smaller set of values.

The most common example sited to illustrate MapReduce is word-count. This problem involves counting occurrence of each word in a large collection of documents. The problem is essentially straight forward, but parallelization required to process the large data makes it a complex problem. This problem can be solved as a MapReduce application in the following manner.

The Map function might look like as shown below:

```
map ( String Key, String Value )
// key: document file name
```

```
//value: documents contents
     for each word w in value
          emitIntermediate( w, 1 )
```

The reduce function might look like this:

```
reduce (String key, Iterator values)
//key: a word
//value: a list of counts
     int sum = 0;
     for each v in values
          sum += v
     emit (result)
```

2.2 MapReduce Execution

The map and reduce invocations are distributed across a cluster of multiple machines. The MapReduce framework assumes that data is immutable. In essence, the input data cannot be changed in the map or reduce function. This makes the framework efficient and scalable to a large number of nodes.

Figure 1 shows the flow of MapReduce executions.

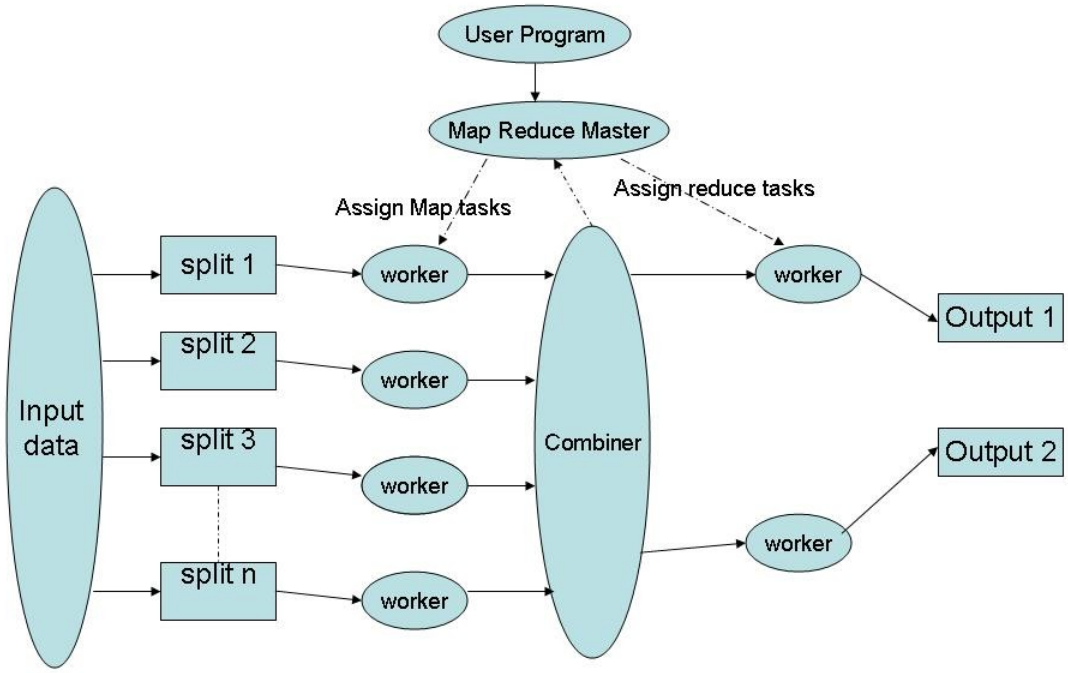

Figure 1 MapReduce execution overview [1]

2.3 CometCloud Architecture

CometCloud [2] is a decentralized (peer to peer) coordination engine that supports applications with high computing requirements. It provides a decentralized virtual shared space which can store entities, called tuples, along with an efficient communication and coordination support. It also provides application framework for master/worker paradigm.

The virtual shared space is constructed from the semantic information space used by participating nodes for communication and coordination. The space is deterministically

mapped, using a locality preserving mapping technique to a dynamic set to peer nodes. Figure 2 gives a schematic representation of the Comet engine.

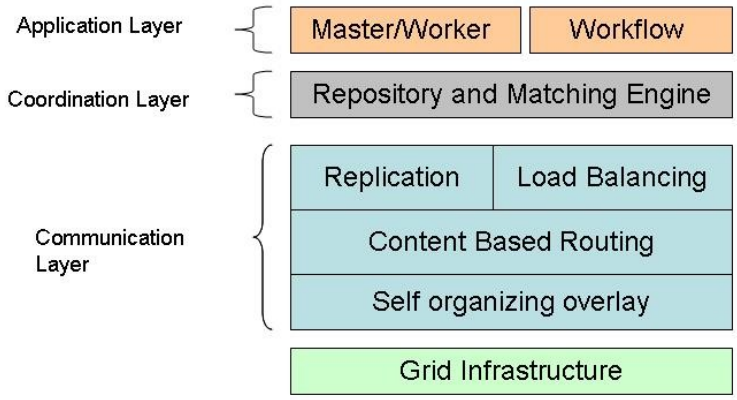

Figure 2: Schematic representation of Comet Infrastructure

In Comet, data is associated with a tuple which is a simple XML string representing the information relevant to the application. Comet employs the Hilbert Space-Filling Curve (SFC) [13] to map tuples from a semantic information space to the linear node index. Each tuple is associated with '*k*' keywords selected from its tag and names. They are defined as the keys of the tuple in the *k*-dimensional (*k*D) information space. If the keys of a tuple only include complete keywords, the tuple is mapped as a point in the information space and located on at most one node. If its keys consist of partial keywords, wildcards, or ranges, the tuple identifies a region in the information space,

corresponding to a set of points in the index space. Each node stores the keys that map to the segment of the curve between itself and the predecessor node.

Comet provides following functional primitives:

- *Out(T_s, t)* - A non-blocking operation which inserts a tuple *t* into space T_s.
- $In(T_s, \bar{t})$ - A blocking operation that removes a tuple *t* matching the template $\bar{t}$ from the space T_s and returns it.
- $Rd(T_s, \bar{t})$ - A blocking operation that returns a tuple *t* matching the template $\bar{t}$ from the space T_s and returns it.

Replication:

As seen from Figure 2 comet provides application layer, coordination layer and communication layer. The Chord [3] overlay service is used to provide a self organizing overlay. This layer also provides replication as well as load balancing. Each node maintains the state of its successor node. Successor node will always be its nearest neighbor. This replica is constantly updated whenever there is a state change in the successor neighboring node. If the neighboring node undergoes failure then its state is merged with the nodes where its replica is maintained. The chord layer also provides load balancing, i.e., whenever a new node joins the overlay, number of task tuples stored in each node is redistributed accordingly.

Task Monitoring:

The programming/application layer which supports the master/worker paradigm provides task monitoring. When a master generates tasks and inserts them into the comet space, the task monitoring services periodically queries the space and detects if any tasks are missing. A task can be considered as missing if it has been consumed but the master has not got the result for a pre-specified time. When the task monitor determines a certain task is missing, it regenerates the task and inserts it into the space. If the Master receives the result multiple times then it ignores the later results. Tasks might go missing due to various reasons like multiple node failures, network issues etc. In such cases the communication layer cannot replicate the lost tasks. Thus Task monitor provides application level resilience towards nodes/task failures.

2.4 MapReduce on Comet

The MapReduce abstraction [8] has been built on top of the Master/Worker programming paradigm that is explained in the previous subsection. Figure 3 gives the complete execution flow of the MapReduce framework in Comet. The Comet MapReduce framework does most of its processing in-memory and hence provides better acceleration of small to medium data-set when compared to other MapReduce implementations like Hadoop MapReduce [9]. The Comet map reduce has following components:

i. **Input Reader-** This is an interface that the user has to implement to determine how input data has to be read into data records.
ii. **MapReduce Master:** This class extends the Comet Master framework and is responsible for generating the task tuples, inserting them into the space. It also monitors the tasks and regenerates any missing tasks. Initially, the master generates map tasks and puts them into the comet space then, the tasks are consumed by the workers. It

collects results from the workers and merges the results belonging to the same key. It saves the map results to the disk. If the master's memory is insufficient to merge the map results, it uses the disk cache. Once all Map results have been collected, it generates reduce tasks and inserts them into the shared space. Once reduce results are available, they are saved to disk.

iii. **MapReduce Worker:** This class implements the Comet Worker framework. Worker nodes continuously query the shared space for available tasks. When a task becomes available, a worker consumes that task, performs the required computation and sends the result back to the master. When a worker consumes a task, it invokes the appropriate application level *mapper* and *reducer* methods.

iv. **Mapper:** This is the interface that the user implements to define the map function.

v. **Reducer:** This is the interface that the user implements to define the reduce function.

vi. **Output collector:** This interface is used to collect the outputs of map and reduce tasks. This interface is also implemented by the user.

The MapReduce execution and dataflow is as shown in Figure 3. When a user submits a job to the MapReduce master, the master reads the inputs keys with the help of input reader. It then generates map tasks corresponding to the keys and inserts them into the comet space. The workers "pull" the tasks from the space and do the required computation by using the *map* provided by the user. Once the computation is done the result is sent to the master. In case of map tasks, the master runs a "Map Merger" where the results with the same key are merged. After the entire map results are merged, reduce tasks are generated and inserted to the space. These tasks are picked up by the workers and the *reduce* function is applied on the tasks. The results are sent back to the master.

When the master collects all the reduce task results. It writes final output in the given output path.

Map reduce data flow on Comet is as shown in Figure 3

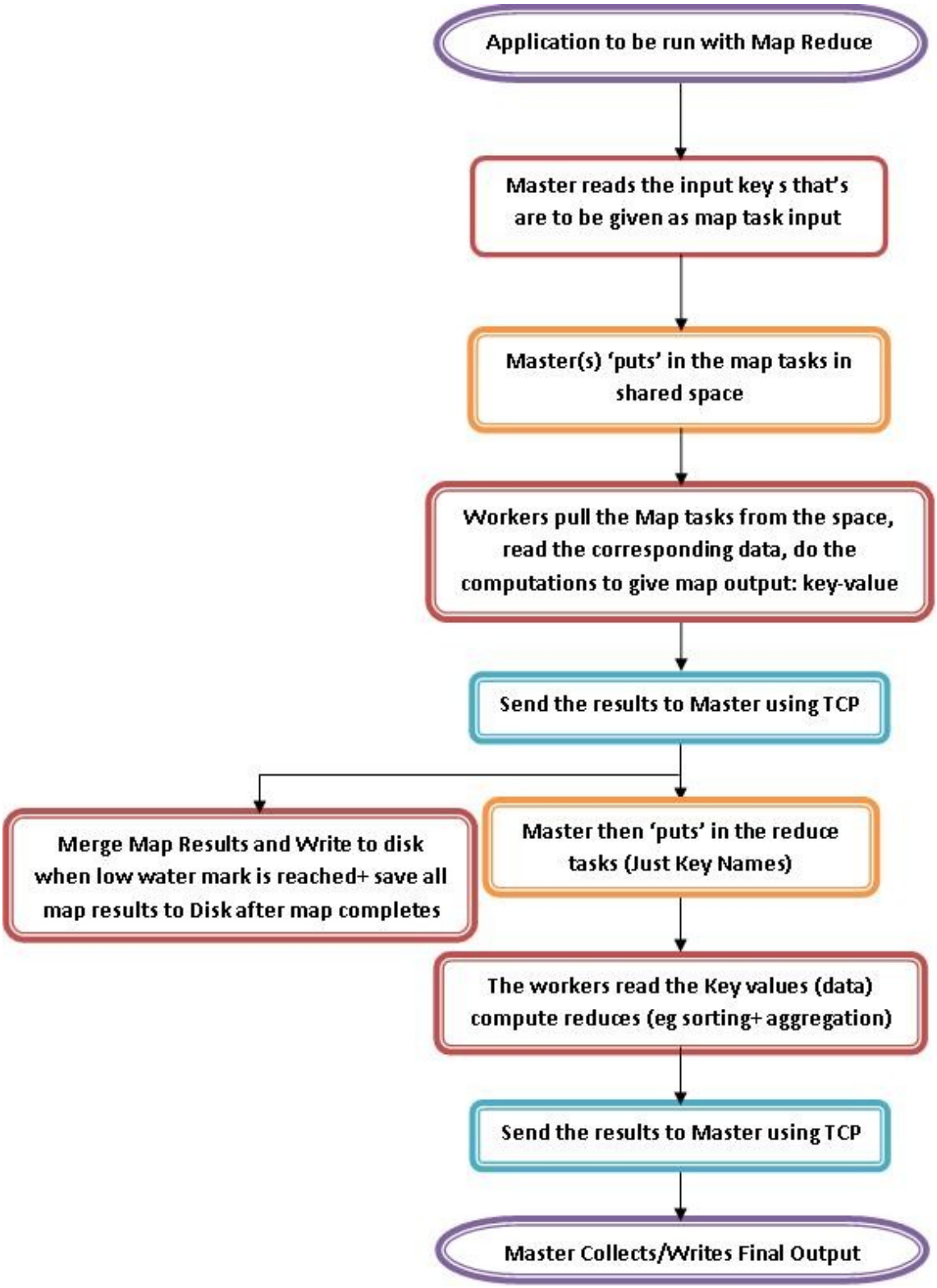

Figure 3 MapReduce data flow on comet [14]

2.5 Cloudbursting and Cloud Bridging

Cloudbursting[8],[9], as the word suggests is reaching out for the cloud computing resources when the computing need of an application exceeds the capacity of the existing datacenter. Today, cloud computing provides a new computing paradigm of on-demand computing access. It provides an abstraction of unlimited computing capacity available to be used as when necessary. The payment model of the cloud is essentially "pay-as-you-go" which means users can now rent computing resources just as they rent utilities like electricity, water etc. When the cloud computing resources can be integrated with the existing grid/private datacenter it opens up new opportunity of on-demand scale up and scale down of computing capacities which is known as **Cloudbursts.**

Today there are several different cloud computing services available in public. Some of the popular cloud computing platforms are Amazon EC2 [4], Microsoft Azure [5], Google App Engine [6], Go Grid [7]. Each of these platforms offers various Service Level Agreements, quality of Service as well as pricing policy. One of the limitation of today's cloud computing platforms is that it is not easy to integrate the cloud computing services of different vendors due to the differences in the computing services offered by them. Hence the users are compelled to select a particular type of service which is suited to run their application. Autonomic cloud bursting aims at integrating these different cloud services with the traditional grid and datacenters and on the fly. **Cloud bridging** aims at "bridging" different types of clouds so that the services offered by each cloud can be exploited to efficiently run an application. Cloud bursting provides the application an

abstraction of resizable computing capacity and the right mix of datacenter and cloud resources can be driven by the user defined high level policies.

Chapter 3

The Autonomic MapReduce Scheduler

3.1 Overview of Autonomic Cloudbursting in Comet

Today, most of the computing intensive applications are run on cluster-based datacenters. These datacenters have become ubiquitous in industry and research alike. But as computing requirements grow, infrastructure costs, cooling and their management costs also increase. Hence typical strategies, like over-provisioning, no longer become feasible. As such, autonomic cloudbursts can leverage utility clouds to provide on-demand scale-out and scale-in capabilities. Figure 4 represents how cloudbursting can be used with CometCloud. As seen in Figure 4, there are basically three types of computing resources. The most secure and robust cloud/computing infrastructure is the one where the secure masters run. The masters are responsible for initiating a computation, scheduling, monitoring and collecting the results. Second type of computing resource is secure workers which have special security credentials for accessing secure data. The secure workers along with the masters form the Comet virtual shared space. Third type of computing resource consists of unsecured workers which are not part of the comet space, but they can request for tasks through a proxy and get tasks for computation. Autonomic Cloudbursts are primarily used for adding/deleting the unsecured workers when the computing resource requirements change as they are easy and less expensive to add or delete than the secure comet workers.

A schematic representation of Autonomic cloudbursts is as shown in Figure 4.

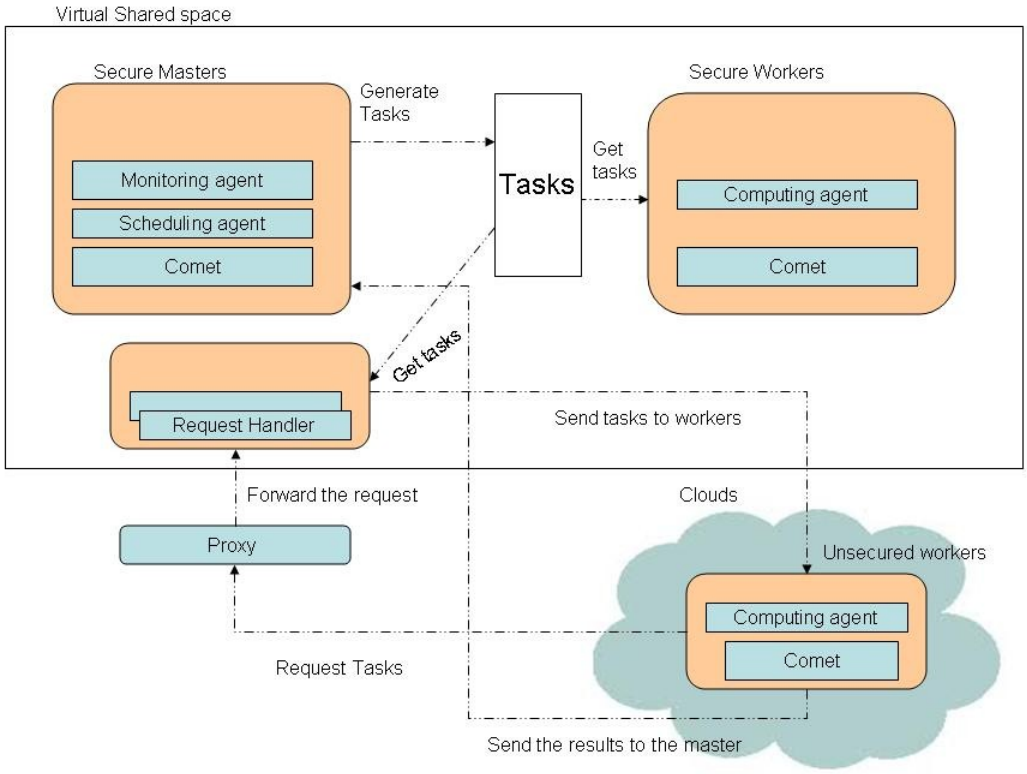

Figure 4 Autonomic Cloudbursts in Comet MapReduce [8]

3.2 Autonomic Scheduler for Comet based MapReduce

Autonomic scheduler for Comet based MapReduce framework aims to be a generic scheduler capable of running any MapReduce application. However, designing such a scheduler is not an easy task. There are certain challenges. One of the main challenges is the fact that different applications have different computing requirements. For example computation can have tasks that are linear, logarithmic etc, with respect to space and time requirements. Hence without the knowledge of the application, it is difficult for the scheduler to schedule at a fine granular level. The other challenge in a MapReduce application is that the tasks can be homogenous i.e., each task is of equal size and hence

has same computational needs or can be heterogeneous where the tasks are of different size and hence require different computing needs.

We have developed a scheduler which can schedule the number of nodes required in each of the available resource class based on user objectives. In this thesis we have considered the following user objectives.

3.2.1. Deadline Based Scheduling

In deadline based scheduling, the objective is to complete a job within the given deadline. In order to meet the deadline, the fastest resource class of all the available resource classes is chosen and based on the initial estimation the number of nodes required for that resource class is decided.

The tasks that need to be scheduled can be homogenous or heterogeneous. Scheduling heterogeneous task is much harder than scheduling homogenous tasks. This is because different tasks have different computation requirements which are not known a priori. Also when the computation model of the tasks is not known, the scheduling strategy might not be very accurate.

To design a scheduling algorithm which can schedule each and every task granularly and very accurately is beyond the scope of this thesis. Hence to simplify things we have assumed that computational model is linear. This means that the time required to complete a task is proportional to the size of the data processed by the task. If the relationship between the data size and the computation time required is known, then the

algorithm that we have developed can be easily extended for any type of task. As this algorithm only provides an approximate estimation of the resource requirements, the scheduler periodically monitors the job's progress and always keeps an updated record of number of tasks completed and the remaining time.

The algorithm works like this:

i. Send a runtime test task to a node of each cloud and get the time required to complete the task. Find out the time required to process unit sized data. For example determine the task time per byte if 1 byte is considered as the unit data size. This step is required only when an application is run for the first time.

ii. Based on the task time per unit data, find out how many nodes are needed to finish the N remaining tasks within the given deadline or the remaining time. If this number exceeds the available number of datacenter nodes, then proceed to step (iii). Or else proceed to step (v). Number of nodes needed to complete a job is given by the following expression:

$$NumNodesInCloud_i = \frac{\{(AverageTaskTime * \mathrm{Re}\, mainingTasks) + (CommunicationOverhead * \mathrm{Re}\, mainingTasks)\}}{\mathrm{Re}\, mainingTime}$$

The communication overhead is also considered because each cloud may have different physical location and thus may take different time for communication.

If T_i is the total time elapsed between the instant Master sends a test task to a worker in Cloud C_i and gets back the result then,

The communication can be approximately calculated in the following manner

$$CommunicationOverheadPerUnitData_i = \frac{(T_i - TaskComputeTime)}{(InputDataSize + OutputDataSize)}$$

This basically gives the communication overhead per unit data. When this is multiplied with the average task size, we get the average communication overhead.

iii. Determine how many tasks can be completed using the maximum available datacenter nodes. This can be calculated using the following expression

$$NumCompletedTasks = \frac{RemainingTime * NumNodesInCloud_i}{(AvgTaskTime + communicatonOverhead)}$$

iv. For each cloud resource class C_i, find out how many nodes are need to complete the remaining tasks. This can be computed using the expression given in the step (ii). Select the resource class which is least expensive.

v. Launch/delete the number of nodes in each cloud and datacenter as determined from the steps ii to iv.

The algorithm can be represented in a flow chart as shown in Figure 5.

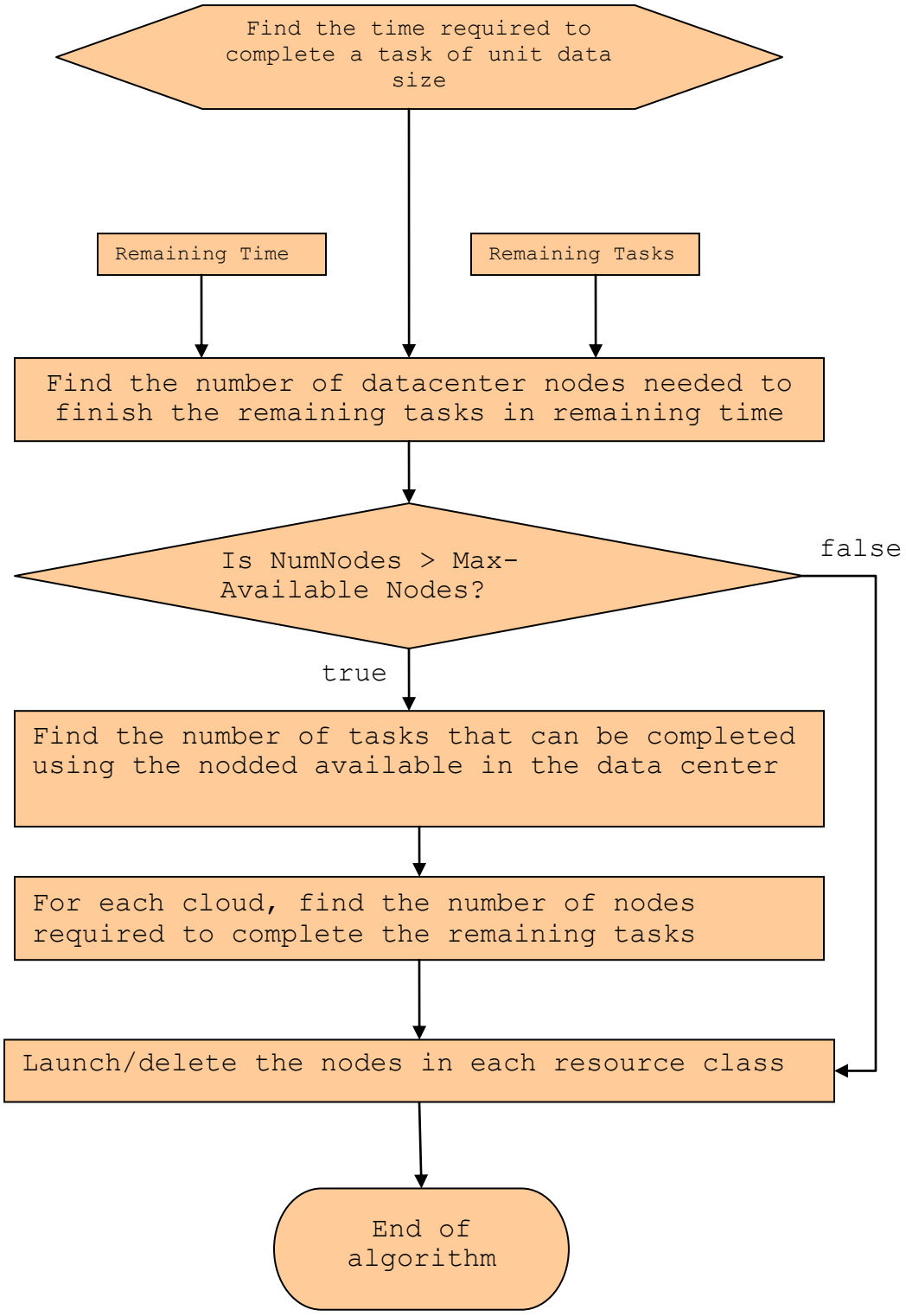

Figure 5 Deadline based scheduling

3.2.2. Budget Based Scheduling

In budget based scheduling, the number of nodes allocated in each resource is restricted by the given budget. If the monitor determines the budget is violated then the next cheapest resource class is scheduled so that the budget limit is met.

The budget based algorithm is as described below:

i. Send a runtime test task to a node of each cloud and get the time required to complete the task. Find out the time required to process unit sized data.
ii. For each of the cloud resource class, determine the number of nodes that can be allocated for the given budget.
iii. Based on the estimation in step (*i*) and number of nodes in step (*ii*), determine the runtime for each resource class. Select the fastest resource class.
iv. Launch the datacenter nodes as well as the cloud based on the result in step (*iii*).

The monitor periodically monitors the job progress. If it finds that the budget limit is being violated then the scheduler decides to replace the existing resource with the next cheaper resource class in order to bring the computation cost within the budget limit.

The algorithm can be schematically represented in a flow chart as shown in Figure 6.

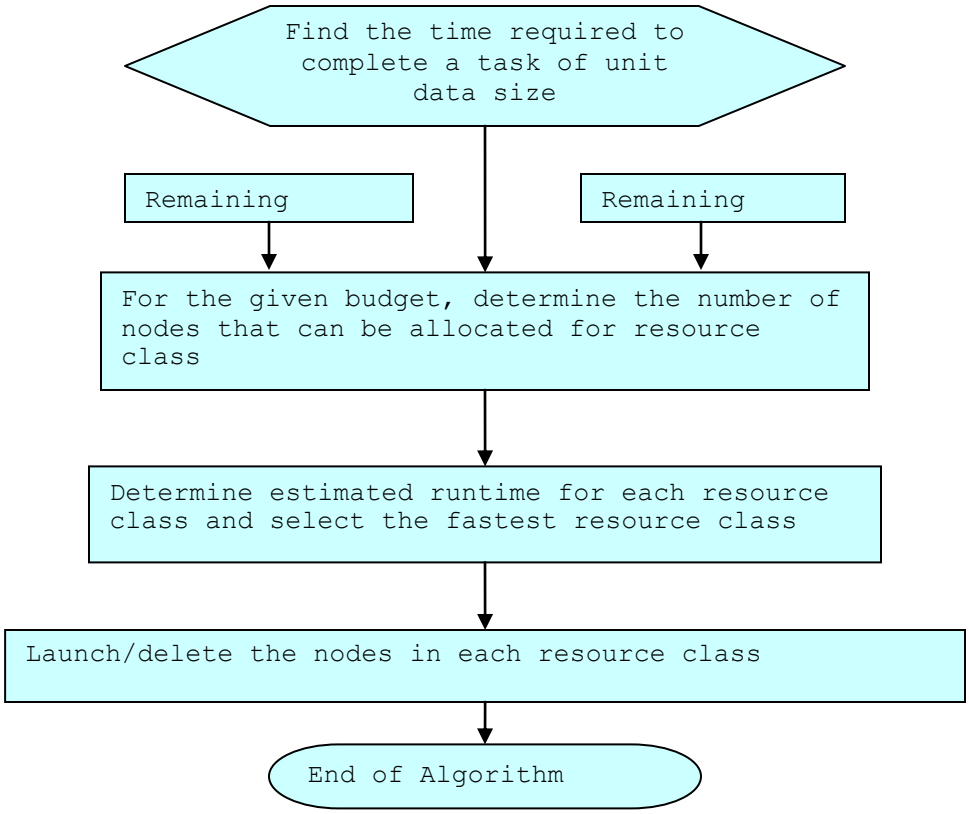

Figure 6 Budget Based Scheduling

3.3 Scheduler Implementation

The autonomic scheduler for MapReduce has been implemented on the CometCloud's application framework. The scheduler object is initially instantiated within the MapReduce master and later it forks as a separate thread. Another challenge for a MapReduce scheduler is that the computation in any MapReduce application takes place in two stages. The map stage and the reduce stage. The scheduler runs its scheduling algorithm separately for two stages. The deadline for each stage is decided on the overall deadline specified by the user and the proportion of time required to complete Map and

Reduce stages for the application that is being run. This information can obtained based on initial test runs without the scheduler or else it is provided by the user. The autonomic scheduler has basically 3 components.

i. **Estimator:** This module is responsible for the initial task time estimation. It sends a test task and measures the time required to complete the task for each of the available resource class. This result is passed on to the scheduling agent.
ii. **The scheduling agent**: This determines the number of nodes to be provisioned based on the algorithm explained in the previous section.
iii. **The monitoring agent:** This component maintains a counter for the number of tasks that has been completed and also it keeps track of how much input data has been processed so far.
iv. **The cloudburst manager:** This component uses the information given by the scheduling agent to add/delete the nodes in each of the clouds.

A schematic representation of the scheduler and its components is shown in Figure 7.

3.3.1. Scheduler Execution Flow

As mentioned earlier, the scheduler is instantiated as an instance in the MapReduce master class. The Map-Reduce master after running some initial routine for preprocessing and setting up the comet environment instantiates the scheduler object.

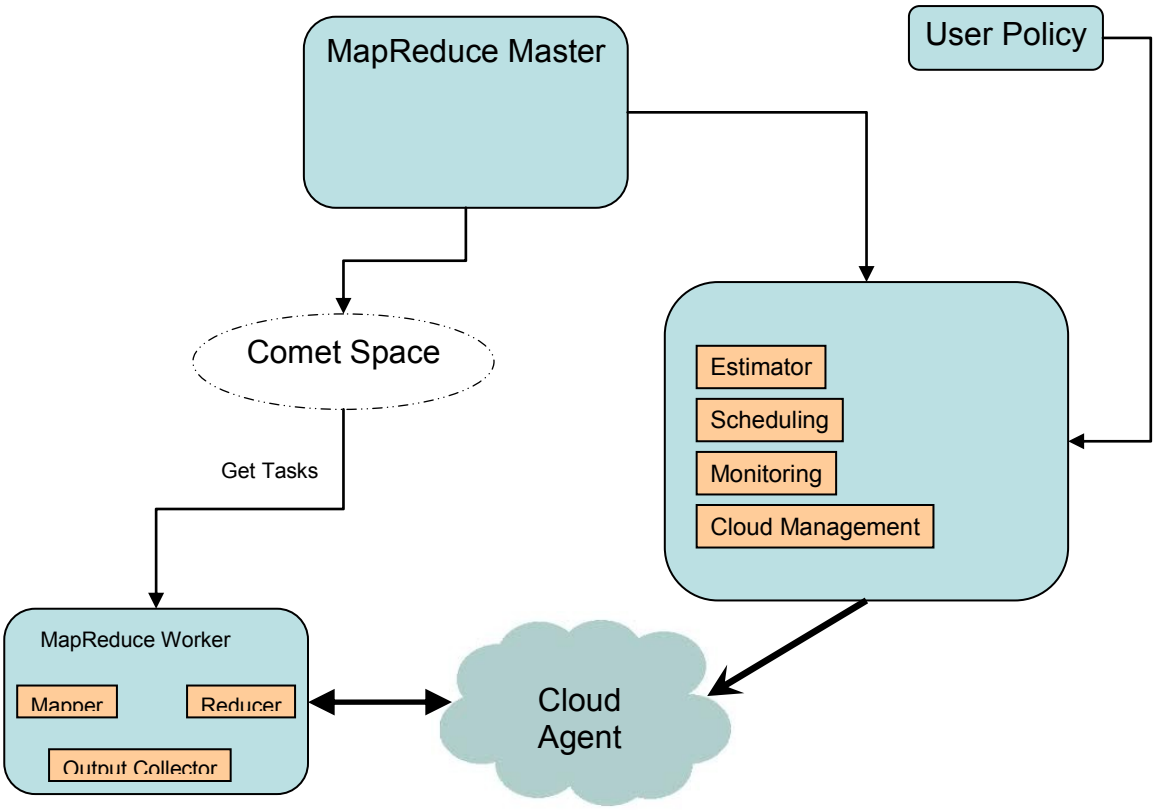

Figure 7 Schematic representation of the scheduler with its components

The scheduler runs in two phases: Runtime estimation phase and Application scheduling phase. In runtime estimation phase the scheduler starts up a single node in each cloud and inserts a test task for each of these nodes. The worker nodes then pick up the task and after doing appropriate computations, they send the result to the master along with information like time required to do the actual computation. Using this information the scheduler determines the computation time for each cloud and it also determines the communication overhead for each of these clouds. It then runs the scheduling algorithm to determine the number of nodes needed in each of these clouds in-order to complete the job according to the user objective. Once the number of nodes is decided, the scheduler then enters the application scheduling phase and the cloudburst manager starts the appropriate number of nodes in each cloud. Every time the master receives the result

from its workers, it updates the counters in the scheduler. The scheduler runs as an independent thread and it continuously monitors the progress of the job and determines if the nodes needs to be added or deleted.

Adding a node is fairly a simple task, as it involves just starting up a new node and running appropriate java process on the node which is essentially a copy of the program. But deleting a node is not that simple. This is because, when the scheduler decides to delete a particular node, it might be computing a task and abruptly deleting a node may result in a lost task. To avoid this situation, the Comet framework has a special task tuple called as "poison pill". When a node "consumes a poison pill", it completes all the computations that it was doing, sends the results to the master and then kills itself. To coordinate these operations, the scheduler sends control messages to the RequestHandler in the CometCloud. The RequestHandler is responsible to delegating the tasks to the workers.

Comet MapReduce framework has been implemented using java which makes it platform independent and hence it can be run on both windows and Linux. Following classes are used for Autonomic scheduling and cloud bursting for map reduce framework.

MapReduceScheduler.java:

This class is responsible for scheduling and it makes the decision to expand or shrink the clouds. It extends the `java.lang.Thread` class and overrides its `run()` method. One of the main methods of this class is:

- `private int[] allocateResourceByDeadline( HashMap<Integer, Double> taskTimePerByte, int numTasks, long dataSize, double deadline)`

This method allocates the resource by running the scheduling algorithm for the remaining number of tasks as passed by the parameter. The method is invoked periodically by the `run()` method which continuously monitors the task and periodically checks if the cluster size needs to be expanded or shrinked.

- `public int[]allocateResourceByBudget(HashMap<Integer, Double> taskTimePerByte,int numTasks, long dataSize, double budget )`

This method allocates the resource based on the given budget. When the user has selected the budget based policy this method is continuously called by the scheduler thread and based on the job progress, the node adaptation is determined. The method is also invoked periodically by the `run()` method which continuously monitors the task and periodically checks if the cluster size needs to be expanded or shrinked.

CustomizedMapReduceTaskSelection.java:

This class is instantiated in the Comet class RequestHandler.java which picks the tasks and sends it to the unsecured workers in the cloud. This class implements the *CustomizedTaskSelection.java* interface from the Comet framework. The *CustomizedMapReduceTaskSelection.java* is responsible for handling the MapReduce specific control messages that are exchanged between the scheduler and the Request Handler. The communication takes place using TCP sockets. Initially when the scheduler enters the runtime phase it sends a control message to the Request Handler saying that the runtime phase has started. This message is passed in to the

CustomizedMapReduceTaskSelection class and it then generates suitable task templates to pick up the runtime tasks appropriate for each cloud. When the runtime phase ends, the scheduler informs the Request Handler of the same and it then starts querying for the actual computation tasks. Whenever the scheduler decides to delete the nodes in a cloud, it sends a control message to the Request Handler informing the cloud-id and the number of nodes to be deleted. This class then generates the required number of "poison pills" and sends them to the nodes of that particular cloud.

Chapter 4

Experiments and Results

Comet MapReduce framework is specifically suited for applications which have a medium data size. In [14] it has been shown that Comet based MapReduce out performs Hadoop when the number of files are large, but the size of each file is very small. In such cases it has been found that file read/write overhead for Hadoop is much higher than the computation time. But comet uses local file system and NFS file system and for small files does most of the processing in-memory. This makes Comet MapReduce framework perform much better than the Hadoop MapReduce framework.

One such application that has been deployed on Comet MapReduce framework is for mining Protein Data Bank (PDB) structures for distance information.

4.1 Mining PDB Structures

The Protein Databank is a database of known crystal structures (crystallographic database) obtained by crystallography or NMR (Nuclear Magnetic Resonance) spectroscopy. Many of these structures are protein-ligand complexes. By mining the information generated in this database we generate a scoring function which will ultimately tell us how well a molecule can bind to a receptor or protein in our body.

When a molecule/ligand binds to a protein or receptor in our body, it evokes a biological response possibly resulting in pain relief, inflammation reduction etc. Typically there are a limited number of configurations or poses that a protein-ligand complex can assume.

Finding these poses is of immense importance in new drug discovery. Both proteins and ligands are 3 dimensional structures and are constantly changing shape and it is a multi-step problem. The goal is to first identify the bioactive conformation of both the ligand and the protein and then place the ligand in the correct orientation with the protein to produce the desired results.

There are many ways to do this. Some are expensive and hence possibly more accurate and some are fairly inexpensive methods. One approach is to generate large number of potential poses by using the inexpensive method and then use expensive calculation to rank them in the order of likelihood of being a bio active pose.

This is exactly the idea behind Map Distance application that has been developed as a MapReduce code. It extracts the potential poses and then it ranks them to decide which ones to apply the more expensive methods to. As going through the database involves processing large number of files independent of each other, it can be easily made to an embarrassingly parallel application. Hence it is also suited to be run as a MapReduce application.

The file size distribution of the PDB database is as shown in the Figure 8

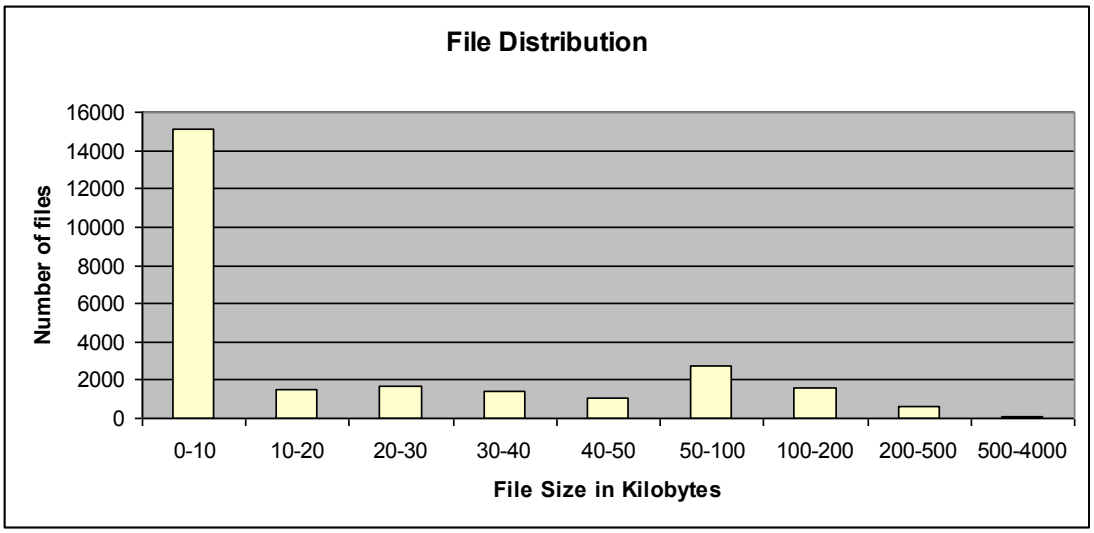

Figure 8 PDB File size distribution

As seen from Figure 8, the PDB database consists of a large number of files of different length with an average size of a few kilobytes. As each file is of a different size, the resulting tasks are heterogeneous in nature. Hence, this application becomes a suitable candidate to evaluate the Autonomic Scheduler. For our experiments we had a total of 25,914 files. As explained earlier, processing large number of small files is very efficient in Comet MapReduce framework.

4.2 Experimental set up

All our experiments were conducted on the Rutgers CAC Dell datacenter and Amazon EC2.

We have a total of 32 machines at Rutgers CAC datacenter. Each machine has 8 cores, 6 GB of RAM and 146 GB disk space. We included 5 instance types for the Amazon EC2. The specification of each of these instance type are as described in the Table 1.

EC2 Instance Type	Mem (GB)	ECU	Virtual Cores	Storage (GB)	Platform (bit)	Cost ($ per hour)
m1.small	1.7	1	1	160	32	0.085
m1.large	7.5	4	2	850	64	0.34
m1.xlarge	15	8	4	1690	64	0.68
c1.medium	1.7	5	2	350	32	0.17
c1.xlarge	7	20	8	1690	64	0.68

Table 1 EC2 instance Type Specifications

4.3 PDB Application Baseline

In order to compare the computing capacity of the Rutgers CAC Dell datacenter and Amazon EC2, we ran the PDB application separately on both Rutgers cluster and EC2 where only one worker of each resource class was used in each experiment. The results are as shown in Figure 9.

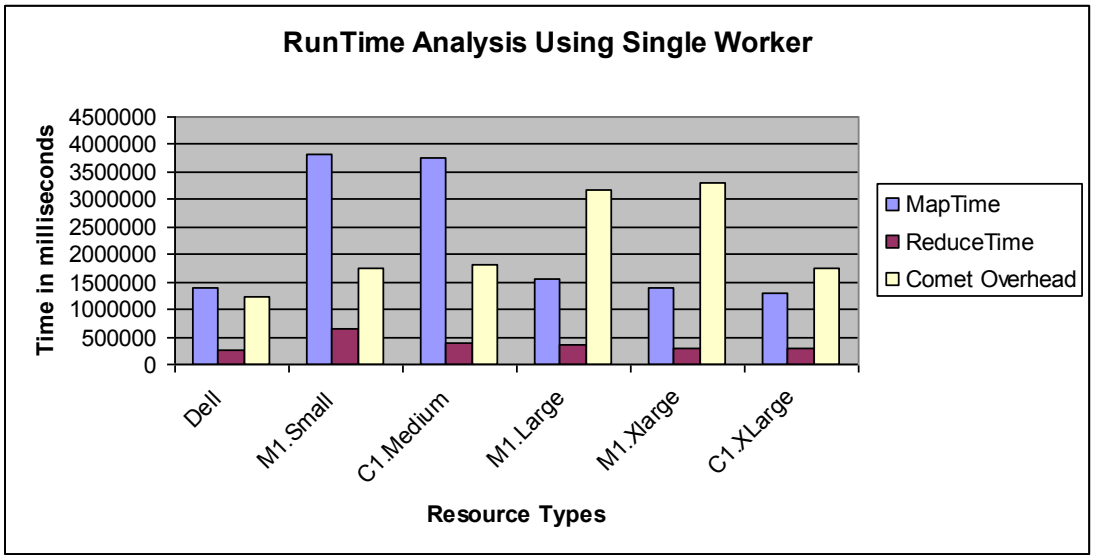

Figure 9: Comparison between Rutgers CAC Cluster and EC2

Cost Analysis: The cost for computation for EC2 workers is as shown in Figure 10. For data transfer, EC2 charges $0.15/GB for all the data transferred out of EC2 and $0.10/GB for all the data transferred in to the EC2 network. The computation is cost is calculated based on the total runtime. Even though Amazon EC2 charges are on hourly basis, for simplicity we have calculated the costs on per second basis.

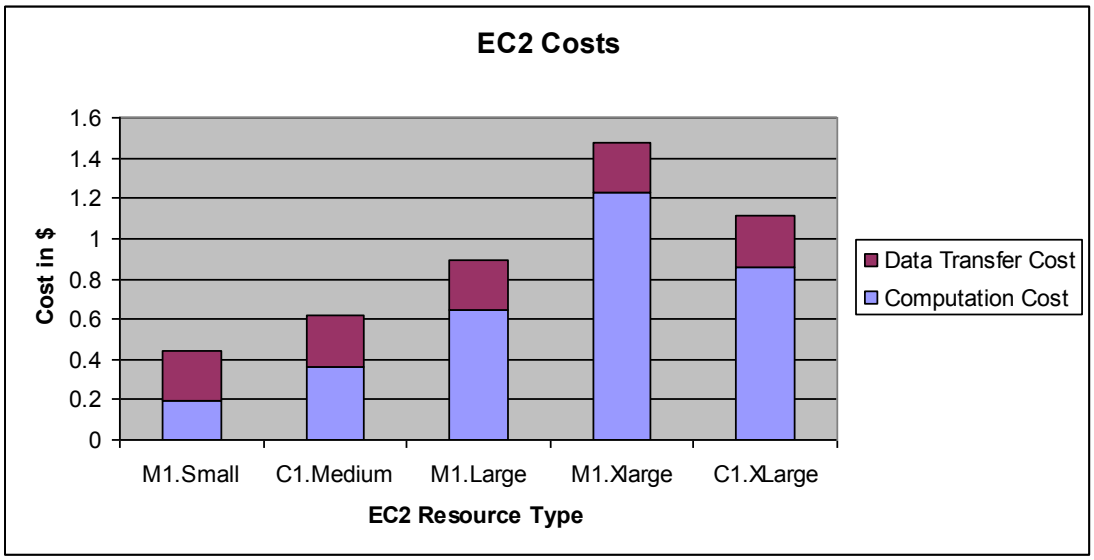

Figure 10 Comparing the Costs involved for EC2

We also compared the performance of the Hadoop Map Reduce framework and Comet MapReduce framework. For this experiment, we used the Rutgers Dell datacenter. We used 1 master and 31 workers. The results are as shown in Figure 11. From Figure 11, it is evident that for applications such as PDB mining, Comet MapReduce is very efficient compared to Hadoop Map Reduce when it comes to Runtime and disk usage.

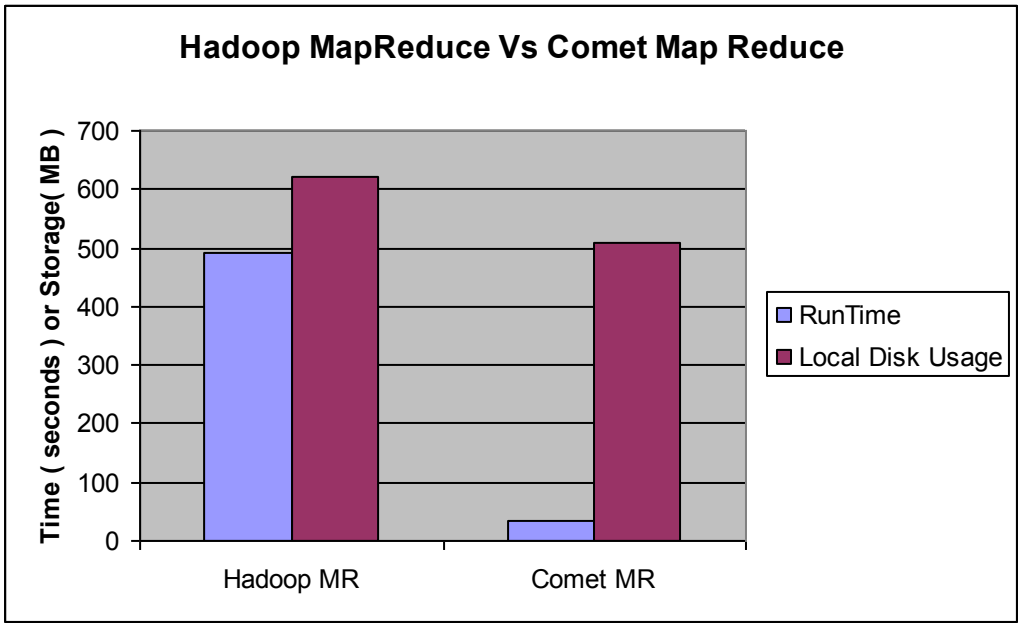

Figure 11 Hadoop MapReduce and Comet MapReduce

4.4 Objective Driven Scheduling

4.4.1. Deadline based scheduling

For our experiments, we used a mix of Rutgers CAC cluster nodes and EC2 nodes.

From the baseline experiment where we ran the PDB application on a single Rutgers node, we found that the runtime is around 4200 seconds (see Figure 9). Using this information we decided to set the deadline from a range of 18 minutes to 10 minutes, where we reduced the deadline by 2 minutes for each of our experiments. We limited the number of nodes in the Rutgers Datacenter to 5 in order to demonstrate cloud bursting.

Figure 12 shows the runtime for different deadlines. We can see that the user objectives were successfully met by cloudbursts. However, by decreasing the deadline, more EC2

needs to be launched which results in increasing data transfer and hence, increasing network overhead.

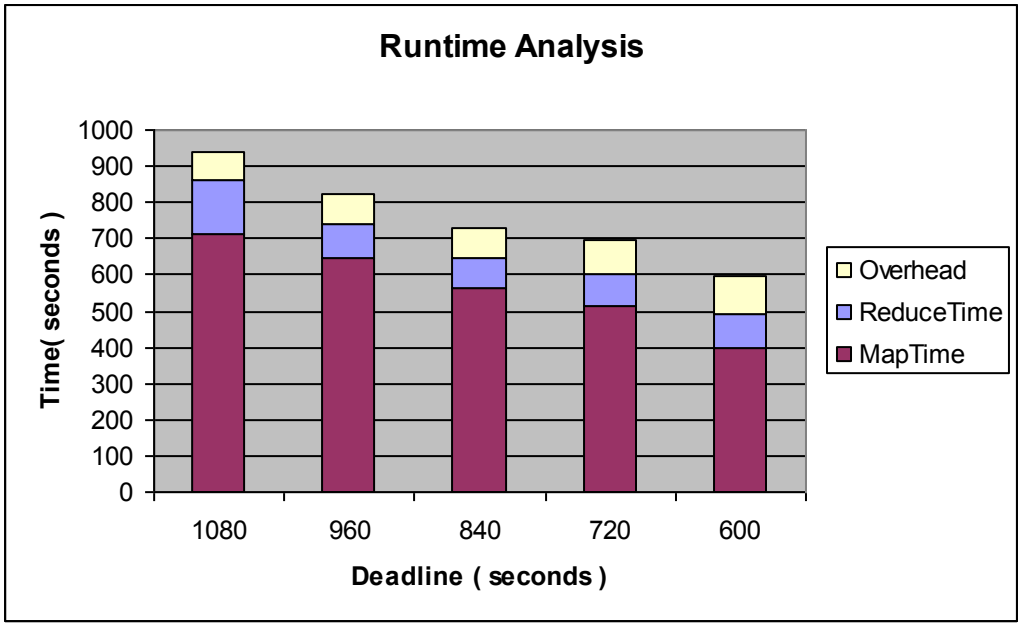

Figure 12 Runtime for Deadline based scheduling

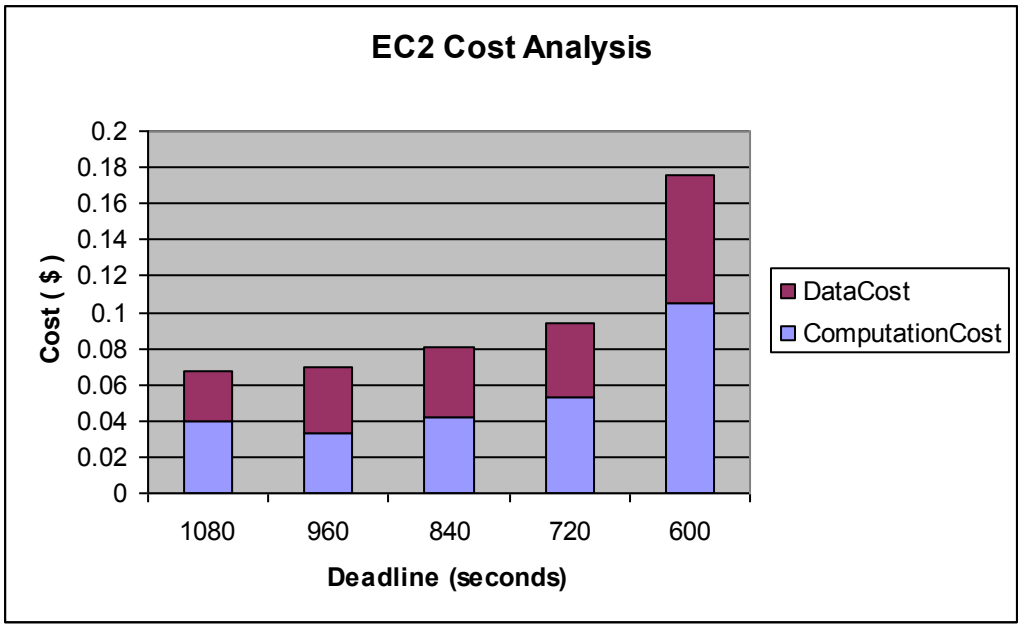

Figure 13 EC2 Cost for different deadlines

Figure 13 shows the EC2 for different deadlines. As we can see from the figure, the EC2 cost increases as the deadline is reduced. The data transfer cost increases with decreased deadline.

We configured scheduler to monitor the job progress at every minute. The detailed scheduling decision at every scheduling stage is as shown in Table 2. As seen in the table, the number of workers is decided to launched/deleted on the fly by the cloud manager.

Deadline (Seconds)		Time in minutes																	
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	Rutgers Nodes	5	5	5	5	5	4	4	2	2	1	1	1	1	1	5	5	5	0
	EC2 Nodes	2	2	2	1	0	0	0	0	0	0	0	0	0	0	8	7	3	0
1080	EC2-Instance Type	c1.medium														m1.small			
	Rutgers Nodes	5	5	5	5	5	5	4	3	2	1	1	1	5	5	0	0		
	EC2 Nodes	5	4	3	2	1	0	0	0	0	0	0	0	7	4	0	0		
960	EC2-Instance Type	m1.small												m1.small					
	Rutgers Nodes	5	5	5	5	5	5	4	2	1	1	1	5	5					
	EC2 Nodes	5	4	3	2	1	0	0	0	0	0	0	4	5					
840	EC2 Instance Type	m1.small										c1.medium							
	Rutgers Nodes	5	5	5	5	5	5	5	2	1	1	5	6						
	EC2 Nodes	6	5	4	3	2	1	0	0	0	0	4	5						
720	EC2 Instance Type	m1.small								c1.medium									
	Rutgers Nodes	5	5	5	5	5	5	2	1	5	5								
	EC2 Nodes	14	10	8	6	2	0	0	0	7	10								
600	EC2 Instance Type	m1.small								c1.medium									

Table 2 Allocated workers at every scheduling period

4.4.2. Budget Based Scheduling

For budget based scheduling we used budgets $0.5, $1, $2, $4 and $8. As the application for mining PDB data finished within 1 hour, we did not reschedule the workers based on budget limit. The run obtained when different budgets were used is as shown in the Figure 14. We can see that the number of EC2 workers used in each of the experiment increases almost exponentially. However, the speed up achieved is not significant. This is due to the fact that many of the operation in Comet MapReduce like task insertion, Map Results merging are carried in the Master and as a result these operations are sequential. Also, as more and more EC2 workers are used there is a significant network over head involved in data transfer. The number of EC2 nodes and their type is as shown in Table 3.

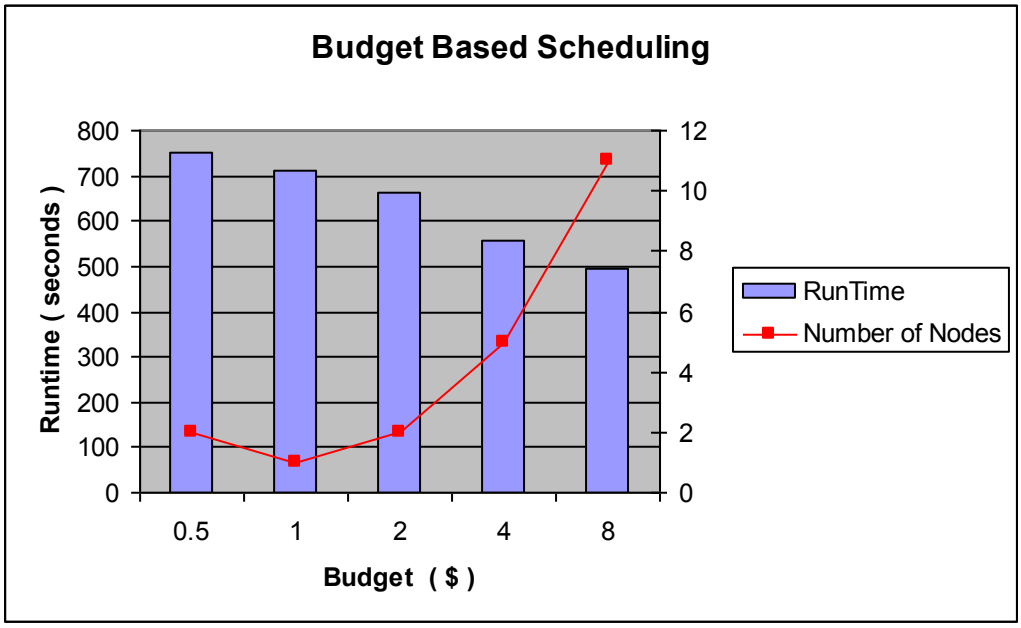

Figure 14 Runtime for Budget based scheduling

Budget ($)	Number of Workers	EC2 instance Type
0.5	2	m1.small
1	1	m1.xlarge
2	2	m1.xlarge
4	5	m1.xlarge
8	11	m1.xlarge

Table 3 Allocation of workers for Budget Based scheduling

The results we have obtained so far clearly indicate that the user objectives can be successfully met using the Autonomic cloudbursts.

Chapter 5

Summary, Conclusion and Future work

5.1 Summary

The primary object of this research presented in this thesis is to develop a deadline based scheduler which enables Cloudbursting and Cloudbridging for MapReduce applications.

A key contribution of this thesis is the new infrastructure for running MapReduce application subjected to a user policy which leverages the available public clouds to meet a sudden increase in demand in computing requirements. This research has opened up a new approach for using MapReduce framework for deadline or budget based applications. Its feasibility has been investigated using a real world pharmaceutical application and it has been proved that indeed MapReduce need not be only for a static cluster size but it can be expanded and shrinked on the fly and effectively meet any computing demand based on a high level user policy. With a generic scheduler and periodic monitoring, any computation that can be expressed using *map* and *reduce* functions can now leverage cloudbursting and thus effectively use both existing datacenter as well as other available cloud resources.

5.2 Conclusion

In this research work we have investigated the use of autonomic cloud bursting for MapReduce framework. Cloudbursting gives an abstraction of single virtual compute cloud that integrates both public and private clouds on the fly. The policy driven scheduler provides an innovative approach for running MapReduce applications.

We have deployed a real world application using a combination of private datacenter as well as public cloud at Rutgers University and Amazon EC2 and have presented the experimental results with different combination of the nodes in the cloud and using different user level policies The results have demonstrated the effective use of cloudbursting for MapReduce and have proved that objective based scheduling is possible.

Since the interfaces provided by Comet Map-Reduce is very similar to Hadoop MapReduce, it is very easy to port the existing applications in the Hadoop MapReduce and use the deadline based scheduling for those applications.

5.1 Future Work

The concept of deadline based scheduling for MapReduce framework is new. Hence there is scope for future work in this direction. Some of them are listed below.

i. The use of this approach needs to be studied in detail for different classes of MapReduce applications like those that are mainly computational in nature, applications which involve a lot of File I/O etc.

ii. The behavior of cloud bursting needs to be studied for different types of clouds and data centers. Currently we have extensively tested on the Rutgers Datacenter as well as Amazon EC2, but there are many other cloud/grids available today. Hence, it will be interesting to study the behavior in different types of computing resources.

iii. In situations where data itself is distributed over multiple clouds, then the Map reduce framework along with the scheduler can be extended so that each cloud has an agent which is responsible for generating the map tasks. Once the results are obtained, the data can then be merged in a single cloud and reduce tasks can be sent out to space. This way, both the master and scheduler can be decentralized.

iv. Investigating if this approach will work for Hadoop map Reduce framework as well would be an interesting research work in itself. Hadoop MapReduce works very well for large datasets. If cloudbursting can be leveraged in Hadoop, then it can significantly help reducing the datacenter costs.

References

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

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

[3] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[4] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[5] Azure service platform. http://www.microsoft.com/azure/.

[6] Google app engine. http://code.google.com/appengine/.

[7] Gogrid. http://www.gogrid.com.

[8] H.Kim, M.Parashar, D.Foran,L.Yang. Investigating the Use of Autonomic cloudbursts for High-Throughput Medical Image Registration. The 10th IEE/ACM International Conference on Grid Computing (Grid 2009) , Banff, Canada, Oct 2009

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

[10] H. Kim, Y. el Khamra, S. Jha, and M. Parashar, "An autonomic approach to integrated hpc grid and cloud usage," in e-Science, 2009. e-Science '09. Fifth IEEE International Conference on, Dec. 2009, pp. 366–373

[11] 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

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

[13] 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

[14] “Accelerating Hadoop Map-Reduce for Small/Intermediate Data Sizes using the Comet Coordination Framework”, *Master’s Thesis* submitted by S. Chaudhari, Rutgers University.

[15] APC, “Determining total cost of ownership for data center and network room infrastructure,” White paper, 2002.

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

.