Autonomic Cloudbursting for MapReduce framework using a Deadline based Scheduler

By

Samprita Hegde

A Dissertation 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 DISSERTATION
Autonomic Cloudbursting for MapReduce framework using a Deadline based Scheduler
By Samprita Hegde

Dissertation Director:
Professor Manish Parashar

Today MapReduce framework is increasingly becoming 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 fully takes care of parallelizing and distributing the computation to the worker nodes. Currently, the existing MapReduce frameworks work like a batch processing system. We have a developed a new deadline based scheduler which does two things,

1. Provides deadline based scheduling capability which presently does not exist in any of the existing frameworks.
2. Provides cloudbursting capability where a computation can “burst” out to cloud whenever the existing datacenter is not capable to meeting the deadline

Using these features, it is possible to run any MapReduce application on any existing cluster by leveraging the utility cloud resources.

In this thesis we use the Comet coordination engine and the MapReduce framework which is built on top of the Comet Engine. The new autonomic scheduler works with the MapReduce Framework and manages the cluster as well as cloud in order to meet computation deadline requirements. We have investigated the use of cloudbursting for MapReduce applications. We found that it is possible to run the application subject to deadline and successfully complete the tasks by efficiently using the datacenter as well as cloud infrastructures.

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 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

Table of Figures

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 today's computing world. Also, as more and more multicore 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, the input data is so large that it has to be distributed across 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. The user has to simply provide a 'Map' and a 'Reduce' function and the underlying framework takes care of the parallelization and all other issues that come when the computation has to be distributed.

The MapReduce framework was made popular by Google [1] to support distributed computing of large data. The Map-Reduce abstraction is inspired by the *map* and the

reduce primitives found in Lisp and many other functional languages. In MapReduce, the input data is divided into many logical records. Each record consists of a key-value pair. The Map function which is provided by the user takes an input record of key value pair and produces an intermediate set of key-value pairs. The MapReduce library takes care of grouping the values belonging to the same key. The Reduce function which is also provided by the user takes the intermediate key and a 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 very simple and all other complexities that arise of parallelization are handled by the MapReduce framework.

1.1 Problem Description and Motivation

Today MapReduce is widely used to compute 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.

Today 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 the Google MapReduce framework [1]. It makes an extensive use of the 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. Presently all these frameworks essentially function like batch processing systems. A user submits the 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 the tasks so as to reduce the data transfer on the distributed file system which is running underneath. It does not attempt to estimate the job completion time. Also, there is presently no possibility of setting deadline 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 both money and energy.

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. The availability of Cloud Computing infrastructure to use the computing resources only when needed suits effectively to our needs.

In this thesis, we have implemented deadline based scheduler for MapReduce frameworks which estimates the resource requirements and dynamically provisions the resources for the job. The scheduler also makes sure that job is completed within the deadline. We have designed this scheduler to work with the MapReduce framework that is built on Comet Co-ordination engine [2] [11].
CometCloud is an existing framework that which provides a decentralized virtual shared space coordination framework for running distributed applications on large clusters. CometCloud also offers Cloud bursting 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 comet. We also evaluate the effectiveness of the scheduler using a real world application for Protein Data 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

Comet [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 master/worker framework. In this programming paradigm the master generates task and inserts into the comet virtual shared space. The workers pull the 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 is consists of a simple XML string describing various attributes.

This programming paradigm has been used the MapReduce framework as well. The MapReduce framework built on Comet provides a conceptual architecture model. It provides the *map* and *reduce* interfaces 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 *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 the input from the input reader and generates the tasks and inserts them to the comet space. The workers "pull' the tasks from the comet space. The workers then determine the type of the task (Map / Reduce) by reading the task attributes in the task tuple and perform the computation accordingly.

1.3 Contribution

The goal of this research is to enable autonomic cloudbursting for Comet based MapReduce framework using a deadline based scheduler. The scheduler has to perform two jobs

i. Based on the deadline provided by the user, estimate number of workers needed to compute the job. The workers may belong to different cloud. Hence it is necessary to take into account that different cloud will have nodes with different processing speeds and memory associated with them.
ii. Continuously monitor the progress of the job and dynamically scale-up, scale-down or scale-out the number of nodes to make sure that the deadline in met. The scheduler also needs to seamlessly integrate private and public clouds to meet the computing needs.

In this research we have implemented a unique scheduler which can effectively fulfill the scheduling needs of any MapReduce application. We have demonstrated this MapReduce application and be seamlessly integrated and run on both public cloud like Amazon EC2 and private cloud like Rutgers CAC datacenter. We have also

demonstrated that a MapReduce application can be completed within the deadline and without the need for over provisioning of the resources and thus reducing significant infrastructure costs.

To summarize, the main theme of this thesis includes

i. Understand the need for a deadline 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 within deadline. This scheduler runs within the MapReduce master and continuously monitors the job progress in order to determine the resource needs.
iii. Develop a cloud agent to control different cloud and datacenters.
iv. Evaluate the scheduler by running a MapReduce application subjected to deadline 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 Map reduce the computation is done in terms of key value pairs. The computation takes an input of a set of key value pairs and produces another set of key value pair. As explained in the earlier section the *map* function provided by the user takes an input pair and provides a set of intermediate key/value pairs. The map reduce 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 possible form a smaller set of values.

The most common example sited to illustrate MapReduce is word count. This problem involves counting if each 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 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 the cluster of multiple machines. The MapReduce framework assumes that the 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 large number of nodes.

Figure 1 shows the flow of MapReduce executions.

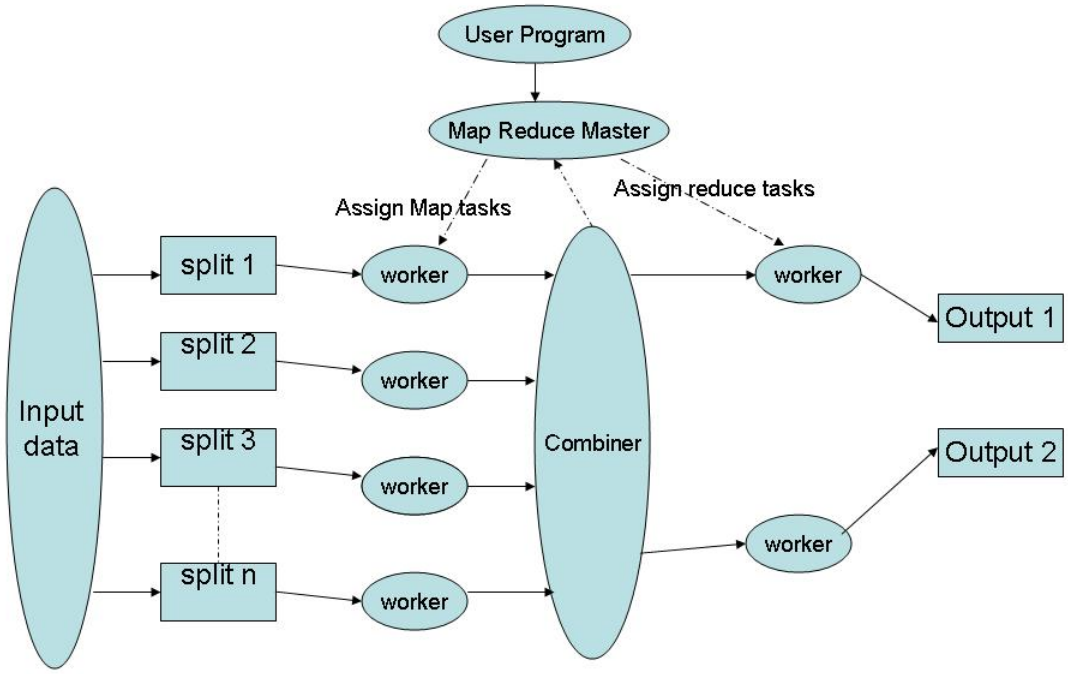

Figure 1 MapReduce execution overview [1]

2.3 CometCloud Architecture

Comet [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 as tuples along with an efficient communication and coordination support. It also provides application framework for master/worker paradigm.

The virtual shared space 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. The following diagram gives a schematic representation of the Comet engine

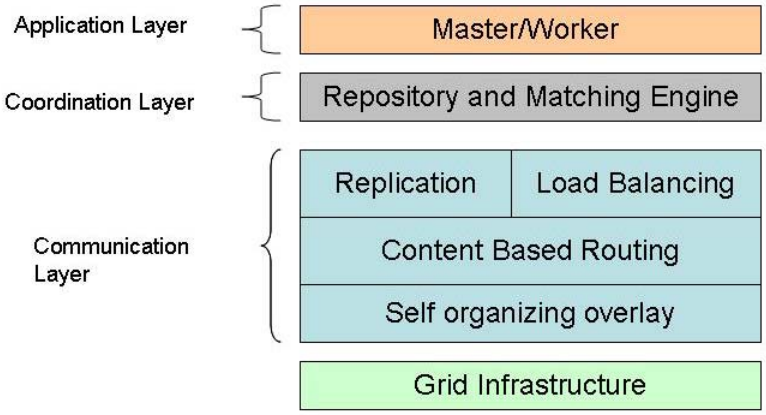

Figure 2: Schematic representation of Comet Infrastructure

In Comet the 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 (kD) 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 primitive for the applications

- *Out(T_s, t)* : It is a non-blocking operation which inserts a tuple t into the space T_s.
- $In(T_s, \bar{t})$) : It is a blocking operation that removes a tuple t matching the template $\bar{t}$ from the space T_s and returns it.
- $Rd(T_s, \bar{t})$: It is 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. Every node keeps a replica of its successor neighbor node's state. This replica is constantly updated whenever there is a state change is 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 the 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 then 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 the 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 the input data has to be read into data records.
ii. **MapReduce Master:** This class extends the Comet Master framework and it s responsible to generating the task tuple, inserting them in to the space. It also monitors that task and regenerates the missing tasks. Initially it generates the map tasks and puts them to space when then consumes by the workers. It collects the 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 the Map results have been collected it generates the reduce tasks and inserts them to the space. It collects the reduce results and writes the final results to the disk.

iii. **MapReduce Worker:** This class implements the Comet Worker framework. The worker nodes continuously queries the space for the available tasks, consumes a task when available, does the required computation and sends the result back to the master. When the worker consumes a amp and reduce task it invokes the appropriate application level mapper and reducer method to do the required computation.

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 the user submits a job to the MapReduce master, the master reads the inputs keys with the help of input reader. It then generates the 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 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.

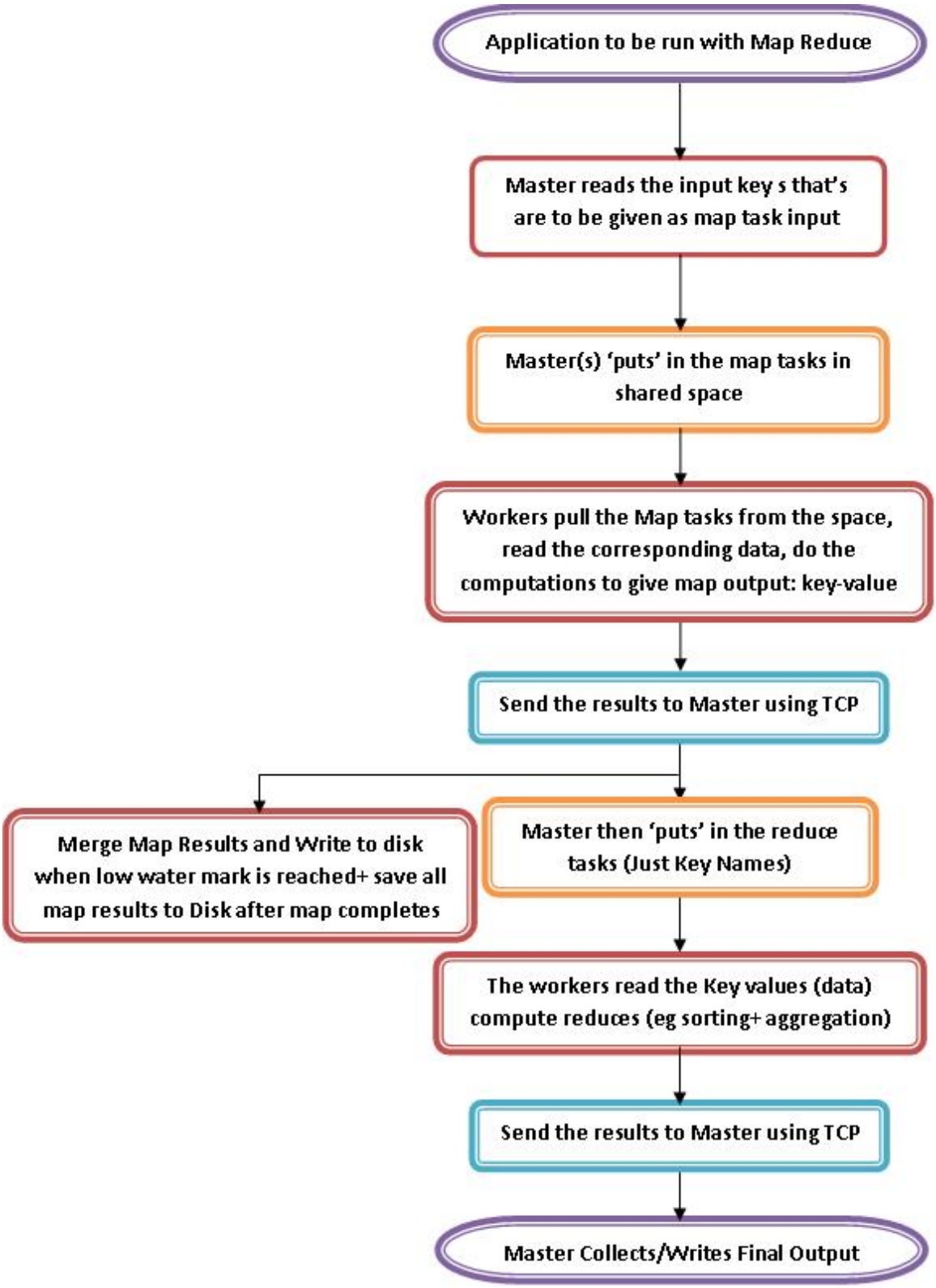

Figure 3 MapReduce data flow on comet [14]

2.1 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 use as when necessary. The payment model of the cloud is essentially “pay as you use” 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 **Cloud bursts.**

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 Cloud bursting in Comet

Today most of the compute intensive and complex applications are run on cluster-based datacenters. As such these datacenters have become ubiquitous in industry and research. But as the application computing requirements grow the infrastructure costs, cooling and their management costs also increase. Hence the typical strategies like over provisioning no longer become feasible. As such autonomic cloud bursts can leverage the utility clouds to provide on demand scale-out and scale-in capabilities. Figure 4 represents how cloud bursting 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 the computation, scheduling, monitoring and collecting the results. The second type of computing resource is the secure workers which have special security credentials for accessing secure data. The secure workers along with the masters form the Comet virtual shared space. The 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 Cloud bursts are primarily used for adding/deleting the unsecured workers as when the computing resource requirements change as they are easy and less expensive to add or delete than the secure comet workers.

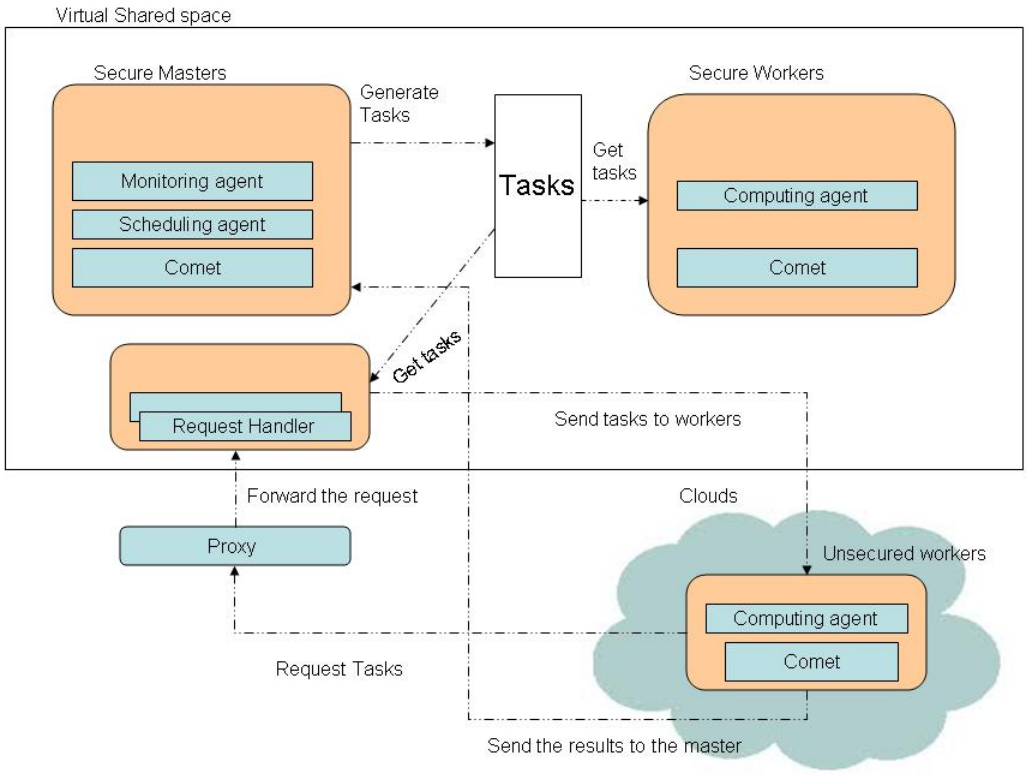

Figure 4 Autonomic Cloud bursts in Comet MapReduce[8]

3.2 Autonomic Scheduler for Comet based MapReduce

The 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.

Scheduling Algorithm for Homogenous tasks:

Designing a scheduler for application which has homogenous it fairly straight forward. Once a task's computing time and numbers of tasks are known it is fairly easy to compute the total runtime and thus determine the number of nodes required to complete the application in a given amount of time.

The variables that are considered to determine number of nodes that should be provisioned for each cloud is explained below:

i. Number of Clouds → M
ii. Number Tasks to be completed → N
iii. The time taken to complete a task by Cloud C_i → t_i
iv. Maximum number of available nodes in Cloud C_i → m_i
v. Deadline for the job completion → D

The algorithm can be explained as shown in the flow chart in figure 5

The following steps explain how the scheduler works

i. Send a runtime test task to a node of each cloud and get the time required to complete the task.
ii. Starting with cloud C_i, find out how many nodes are needed to finish the N tasks within the given deadline.

iii. If the number of nodes needed exceed the maximum available nodes in that cloud, determine how many tasks can be completed using the maximum available nodes.

iv. Now, for remaining tasks repeat the step *ii* and *iii* for cloud C_{i+1}.

Now of course, when not enough resources are available then the scheduler informs that not enough resources are available and does its best effort to complete the job as soon as possible.

The algorithm is described as a flowchart in figure 5.

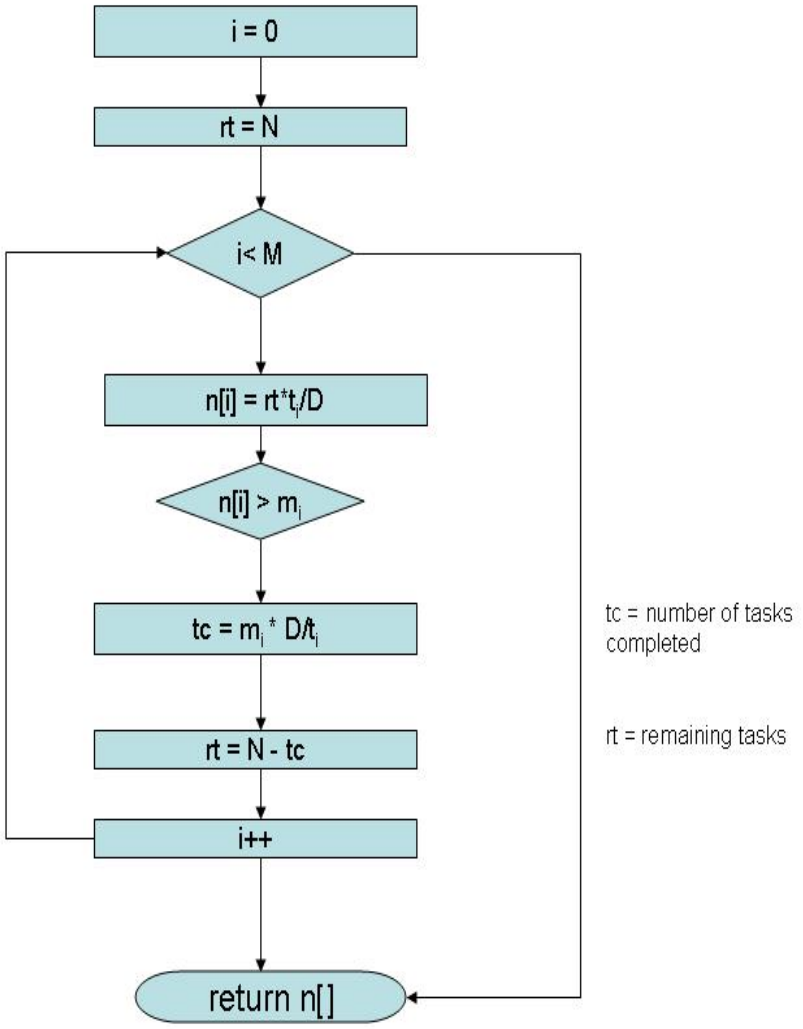

Figure 5 Algorithm to provision number of nodes in each cloud for homogeneous tasks

Scheduling algorithm for Heterogeneous tasks:

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 the 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. But 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.

ii. For each cloud C_i, find out how many nodes are needed to finish the N remaining tasks within the given deadline or the remaining time. Number of nodes needed to complete the 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. If the number of nodes needed exceed the maximum available nodes in that cloud, determine how many tasks can be completed using the maximum available nodes. This can be calculated using the following expression

$$NumCompletedTasks = \frac{\text{Re}\, mainingTime * NumNodesInCloud_i}{(AvgTaskTime + communicationOverhead)}$$

iv. Now, for remaining tasks repeat the step *ii* and *iii* for cloud C_{i+1}.

v. As the scheduling may not be very accurate periodically repeat steps *ii* and *iii* with remaining number of tasks.

vi. Launch/delete the number of nodes in each cloud as determined from the steps *ii* to *iv*.

As we seen from the algorithm description, it is evident that the order in which each Cloud is selected has a significant impact on the scheduling decisions. If suppose $Cloud_1$ is the datacenter and $Cloud_2$ is a public cloud, then the tasks are scheduled on the public cloud only when all resources in the datacenters are exhausted. But if we reverse the order of the clouds then the tasks are first scheduled on the public cloud and then on the datacenter.

The algorithm can be represented in a flow chart as shown in Figure 7.

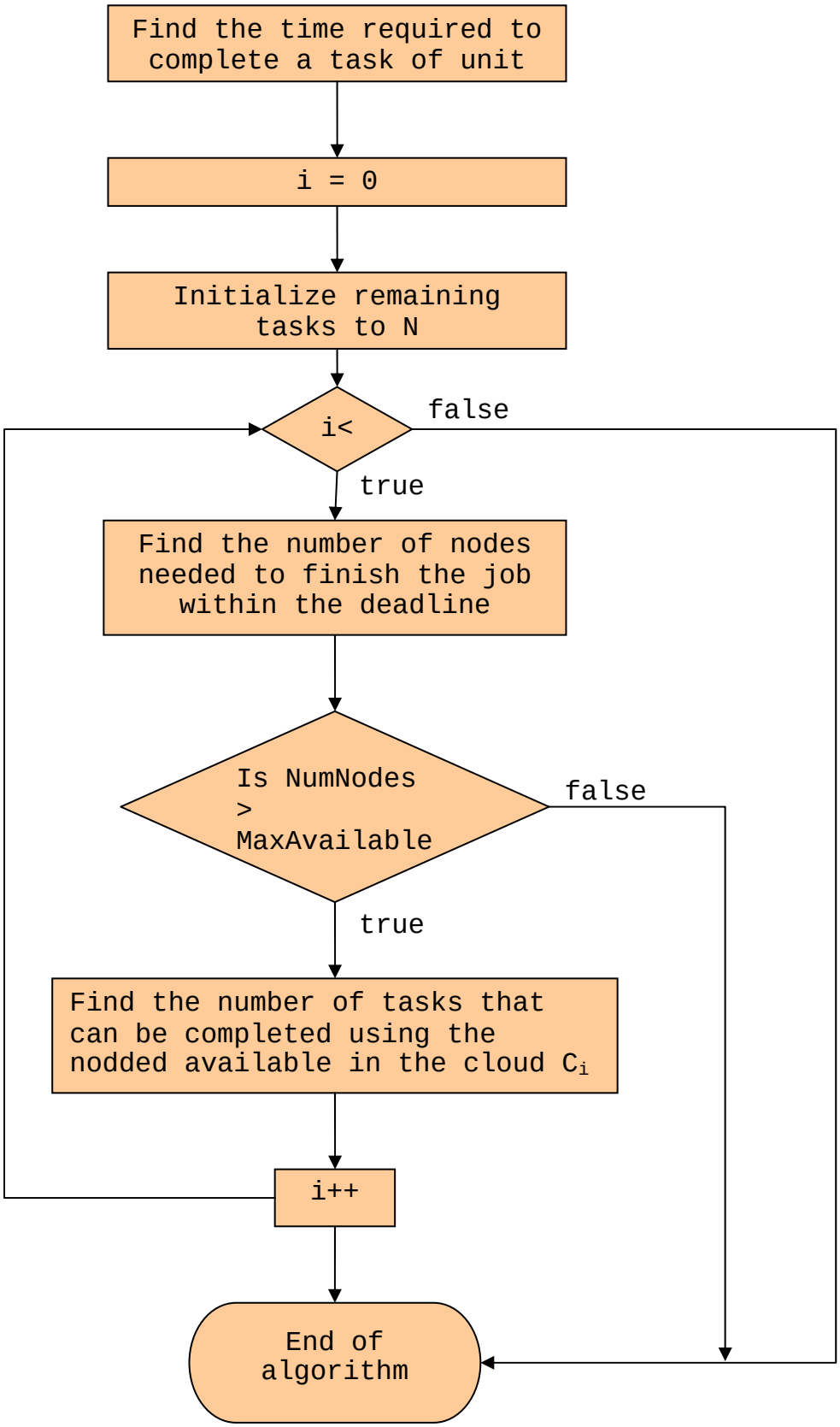

Figure 6 Algorithm for scheduling heterogeneous Map reduce tasks

3.3 Scheduler Implementation

The autonomic scheduler for MapReduce has been implemented on the Comet's application framework. The scheduler object in instantiated within the MapReduce master and then it forks as a separate thread. One other 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 ratio of time required to complete Map and Reduce stages for the application that is being run. This information is obtained based on initial test runs without the scheduler. The autonomic scheduler has basically 3 components.

i. **The scheduling agent**: This determines the number of nodes to be provisioned and the algorithm used is as explained in the previous section.
ii. **The monitoring agent:** This component maintains 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.
iii. **The cloudburst manager:** This component uses the information given by the scheduling agent add/deletes the nodes in each of the clouds.

A schematic representation of the scheduler and its components is shown in Figure 8

Execution

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. The scheduler runs in two phases: Runtime estimation phase and Application scheduling phase.

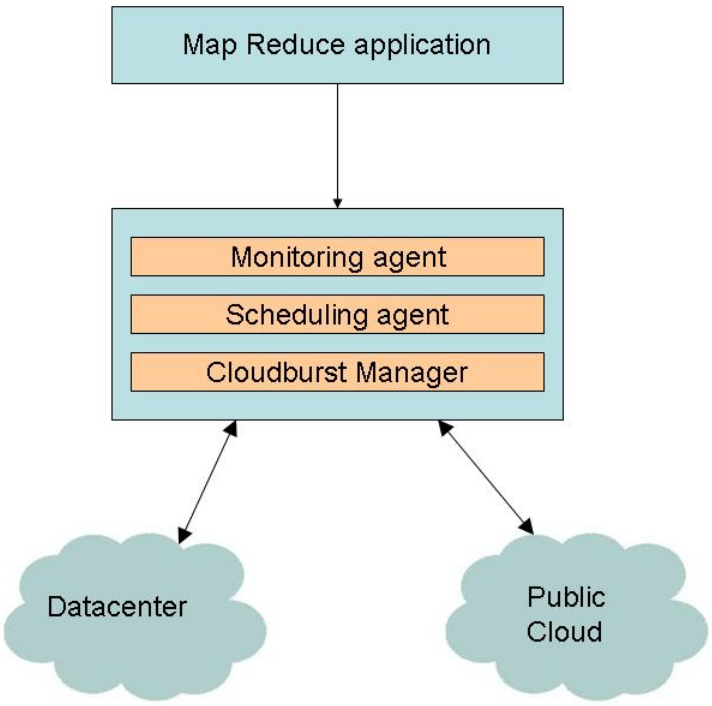

Figure 7 Schematic representation of the scheduler with its components

Runtime Estimation 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 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 within the specified deadline. 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 the 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 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 and the Map reduce framework has been implemented using java which makes it platform independent and hence it can run on both windows and Linux. Following classes are used for Autonomic scheduling and cloud bursting for map reduce framework.

3.1 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 method 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.

3.2 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 5

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 the 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 if for mining Protein Data Bank (PDB) structures for distance in formation.

5.1 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 like, possible 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 in expensive 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 the Map Distance application that has 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 the 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 run it as a MapReduce application.

The file size distribution of the PDB database is as shown in the figure 9

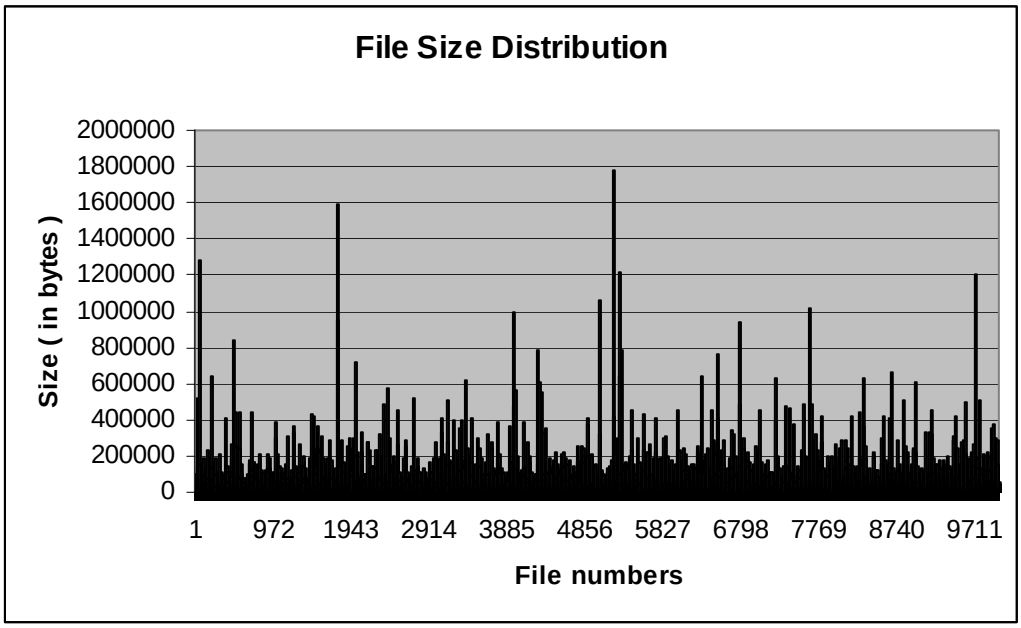

Figure 8 PDB File size distribution

As seen from Figure 9, the PDB database consists of a large number of files of different length with an average size of a few kilobytes. For our experiments we had a total of 10,000 files. As explained earlier processing large number of small files is very efficient in Comet Map reduce framework.

Some of our experimental results are as shown below:

All our experiments were conducted the Rutgers CAC datacenter and Amazon EC2 Small Instance which is a popular cloud computing offering. Henceforth Amazon EC2 refers to EC2 small instance cloud offering.

In order to compare the computing capacity of the Rutgers CAC datacenter and Amazon EC2, we ran the PDB application separately on both Rutgers cluster and EC2 using fixed number of nodes. The results are as shown in figure 10

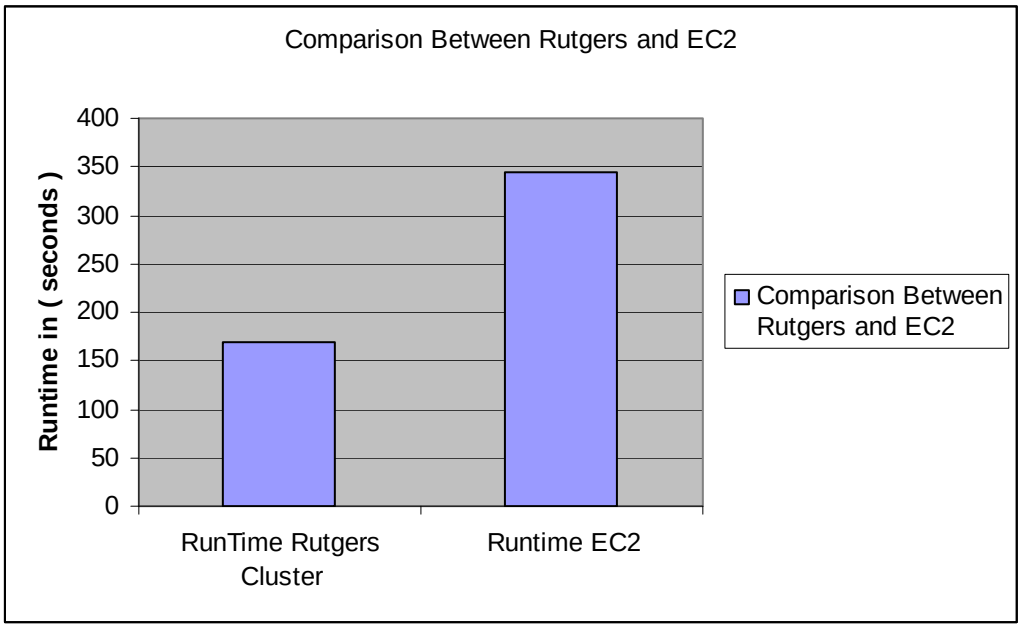

Figure 9: Comparison between Rutgers Cluster and EC2

From Figure 10 we know that a Rutgers cluster node is almost twice as fast as an EC2 node.

Cost Analysis: The cost for the Rutgers Data center includes the hardware investments, software, electricity, cooling infrastructures etc. It was estimated as per the discussion in [15] which says that a datacenter has a life cycle of 10 years and it costs $120/Lifecycle per rack. Using this analysis, the cost of the Rutgers cluster was estimated to be $1.37 per hour. For EC2, the computing cost for a small instance is $0.10 per hour. The data transfer costs $10 per GB.

Using this information, the computing costs for the EC2 and Rutgers cluster for the above described computation on 20 machines is as shown

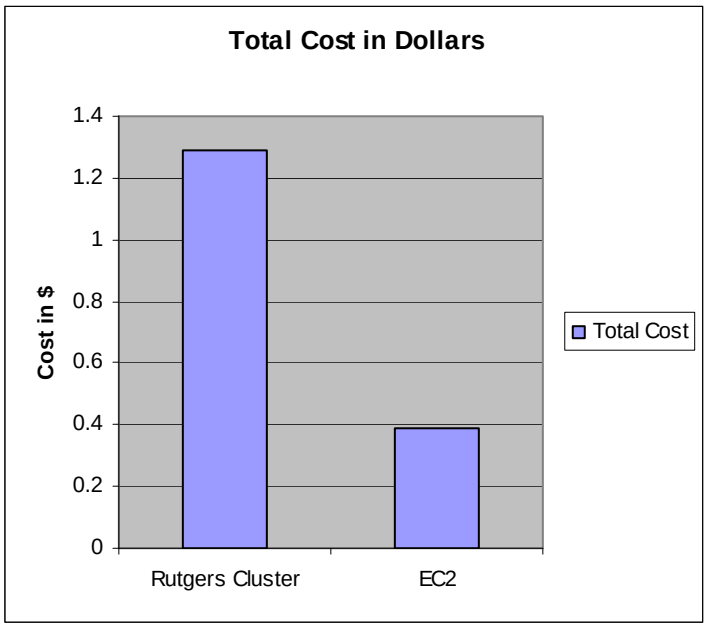

Figure 10 Comparing the Costs involved for EC2 small and Rutgers cluster

As seen from figure 11, it is evident that the Rutgers Cluster is more expensive than EC2.

Experimental set up for cloud bursting

For out experiments we used a mix of Rutgers clusters and EC2 nodes. Our virtual compute cloud consisted of **6 Rutgers Datacenter machines**, **30 Amazon EC2 machines**. The deadline we set was 15 min. For the scheduler to work effectively, the user needs to give the ratio of total Map Time to Total Reduce time. For the PDB application we found that Map phase was approximately 3 times longer than the reduce phase. Hence, the Map deadline was 11.25 min while the Reduce deadline was 3.75 min.

Cloudbursting using the Rutgers Nodes First:

This experiment involves performing cloud bursting where Rutgers nodes are scheduled first. Only when all the nodes are exhausted then the EC2 nodes are

scheduled. This applies to the case where one wants to use the available datacenter resource and only when they run out, the computation "bursts out" to the cloud. The Scheduled number of nodes and the cumulative budget are as shown in Figures 10, 11 respectively.

As we can see from figure 10, in the Map phase the scheduler initially scheduled all the 6 Rutgers datacenter nodes and 8 EC2 nodes. But eventually the EC2 nodes were steadily decreased all the map tasks were completed by the available datacenter nodes. In the reduce phase the scheduler initially scheduled 6 Rutgers datacenter nodes and 15 EC2 nodes. In the middle of the reduce phase the scheduler cut down the EC2 nodes and the job was completed using the only the Rutgers Datacenter nodes.

In figure 13, the cumulative cost follows closely with the Rutgers Datacenter cost as the datacenter nodes and always scheduled during the entire course of the job.

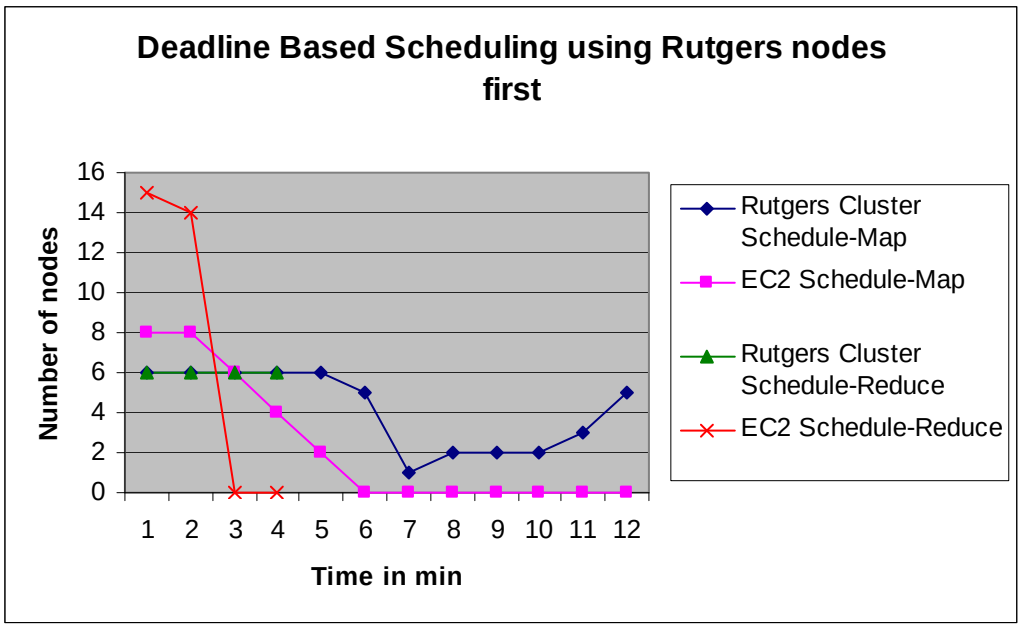

Figure 11 Deadline based scheduling when Rutgers nodes are scheduled first

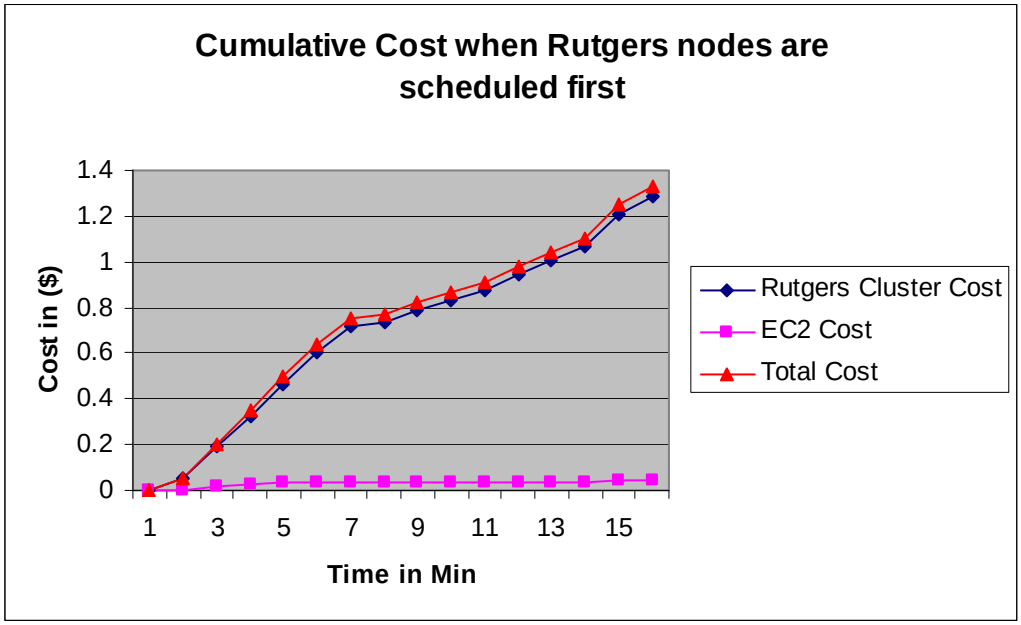

Figure 12 Cumulative Cost when Rutgers nodes are scheduled first

Cloudbursting using the EC2 Nodes First:

This experiment involves performing cloud bursting where EC2 nodes are scheduled first. Only when all the nodes are exhausted then the Rutgers nodes are scheduled. This applies to the case where one wants to use the available datacenter resource and only when they run out, the computation "bursts out" to the cloud.

The Scheduled number of nodes and the cumulative budget are as shown in Figures 14 and 15 respectively. In figure 15 we can see that the cost closely follows that of EC2.

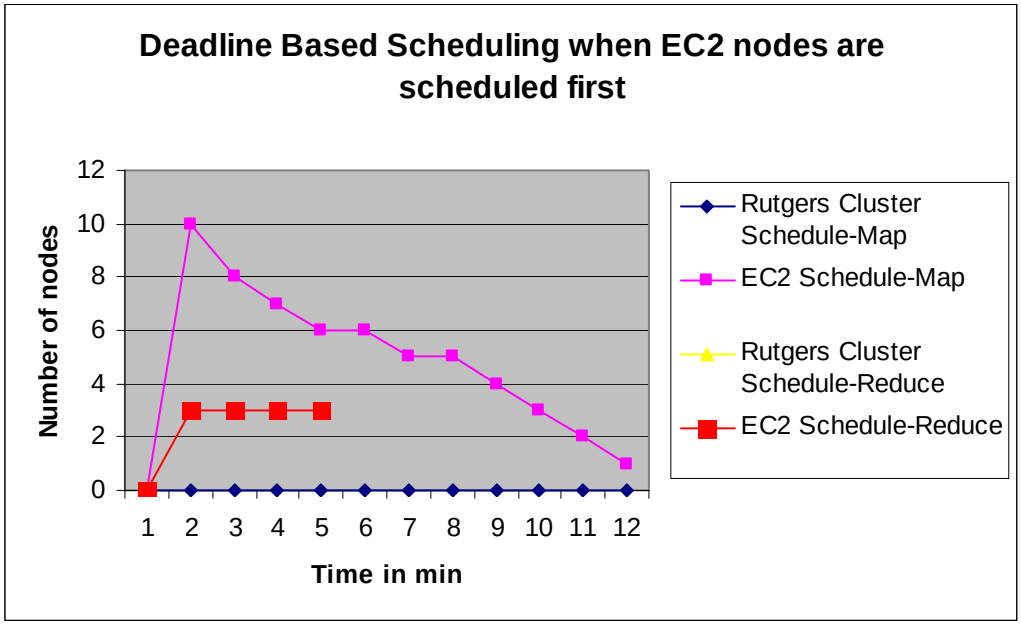

Figure 13 Deadline based scheduling when EC2 nodes are scheduled first

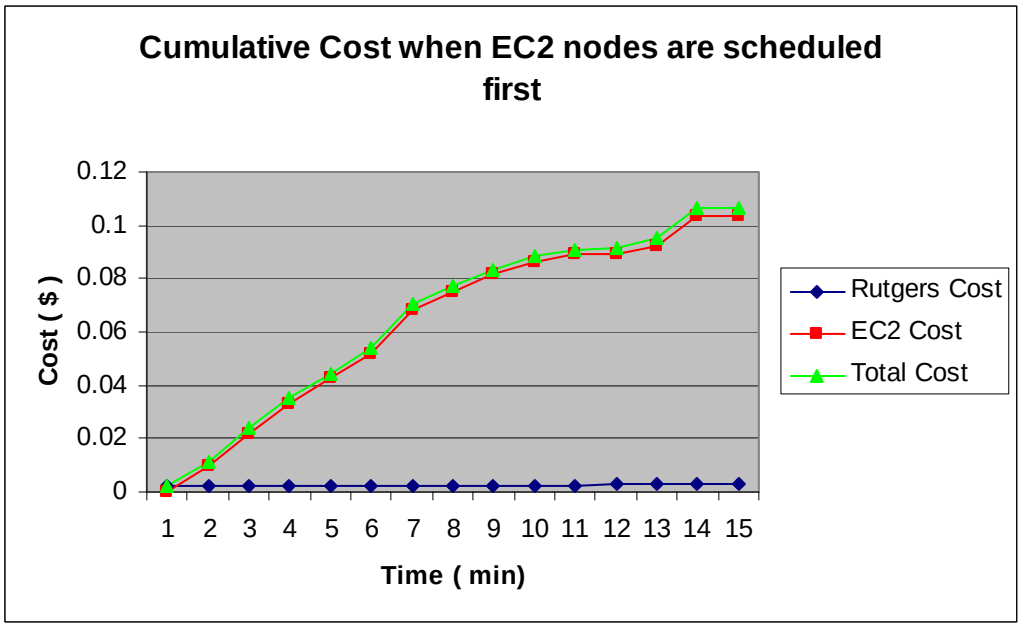

Figure 14 Cumulative costs when EC2 nodes are scheduled first

Comparing figures 13 and 15 we can see that scheduling on EC2 nodes first for the same job resulted in a Total cost $0.1066 whereas scheduling on Rutgers clusters cost $1.33. Hence we can see that scheduling on EC2 first more economical than scheduling on Rutgers cluster first.

In all these experiments, the computation was finished a little before deadline. The total computation times in both these cases are shown below in Figure 16. This is because of the fact that in the scheduling algorithm there is little over estimation for allocating resources. This is to make sure that deadline is reached even when there is a failure of nodes or network problems etc.

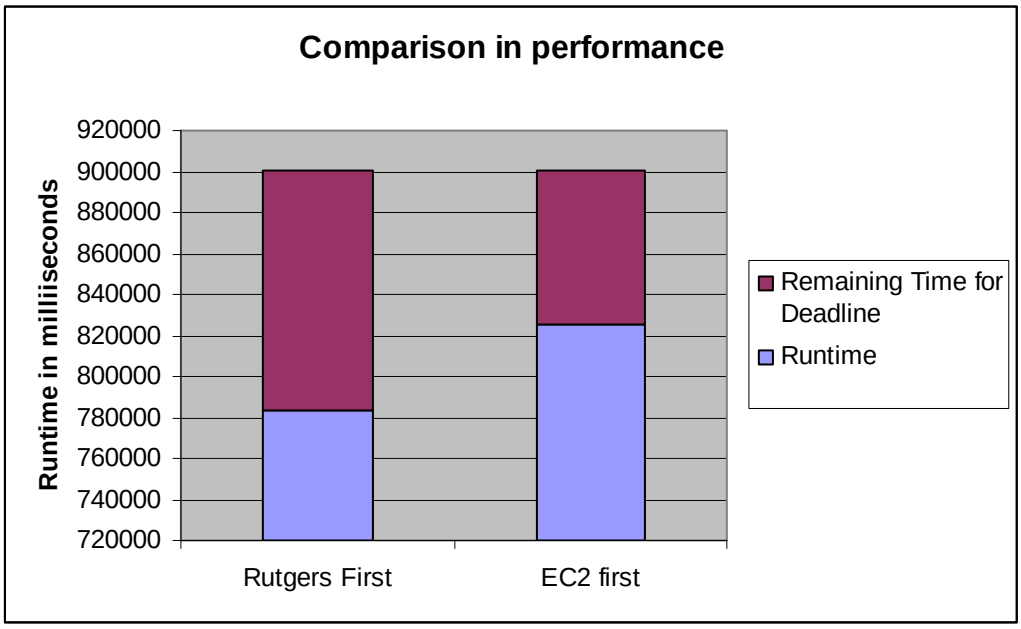

Figure 15 Comparison in performance

Cloudbursting Experiments when communication overhead is not taken into consideration

To study the significance of communication overhead, we conducted experiments where scheduling is done by taking into consideration the computation time only.

Scheduling without communication overhead with Rutgers nodes first

We found that when communication overhead was not taken into consideration, the computation especially in the Map phase missed the deadline by a significant margin. The Number nodes scheduled over time and the cumulative costs are as shown below in figures 17 and 18.

In figure 17 we see that the scheduling for Map phase is for about 14 minutes when the deadline is 11.25 minutes. Thus it has missed the deadline by a few minutes. And also initially very few number of nodes are scheduled and during the last 2 minutes the number of nodes suddenly shoots up for both EC2 and Rutgers nodes. We see that the total number of nodes scheduled on both the clouds in figure 17 is much lower than in figure 112. As a result of this, the total cost incurred is also lesser. This can be seen when we compare cumulative cost is as shown in figure 18 and figure 13.

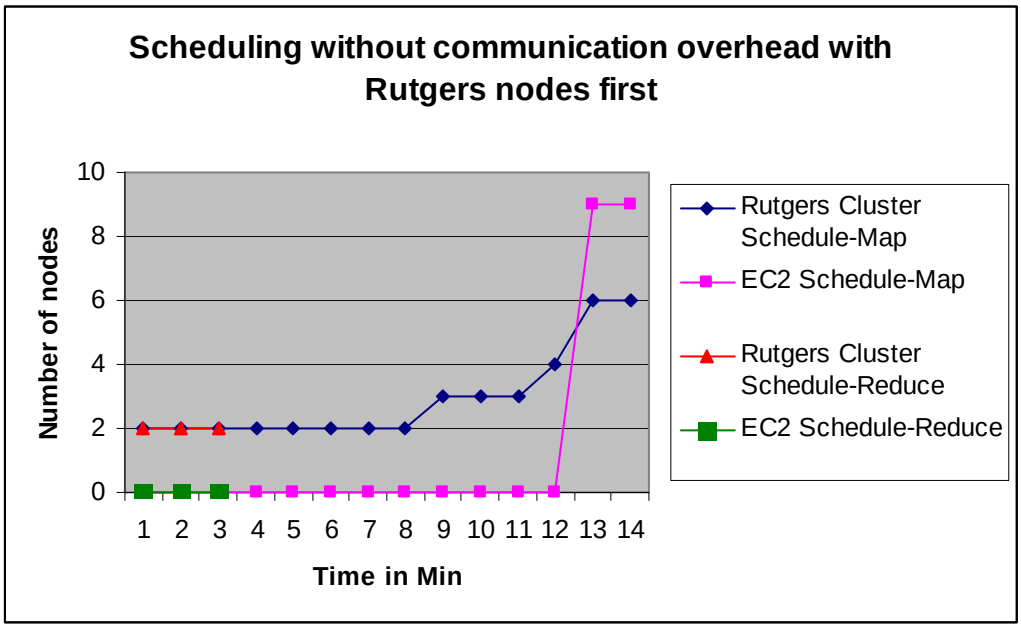

Figure 16 Scheduling without communication overhead with Rutgers nodes first

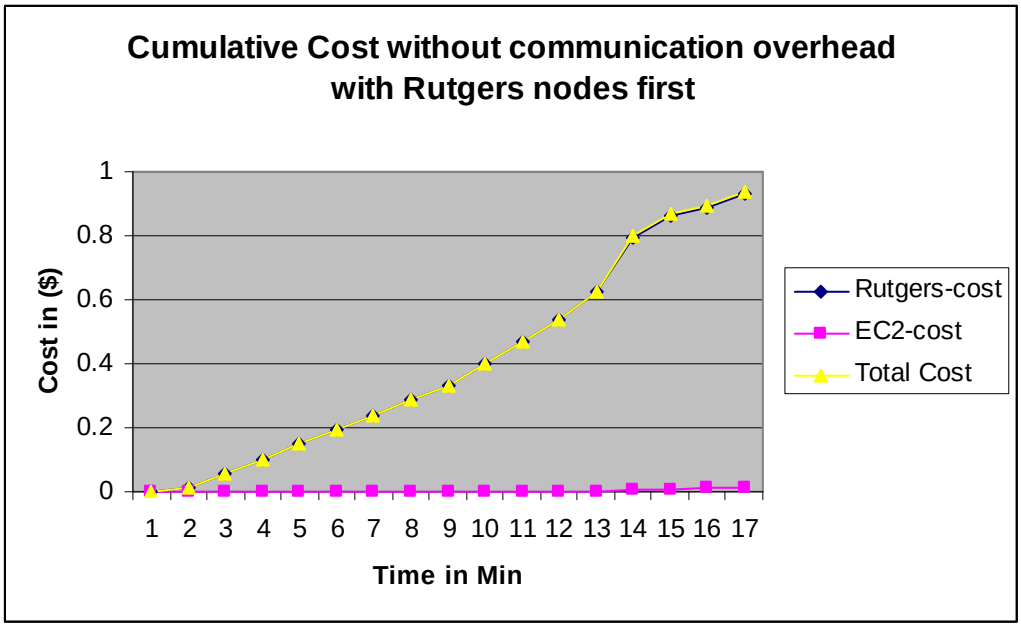

Figure 17 Cumulative Cost without communication overhead when Rutgers nodes are scheduled first

Scheduling without communication overhead with EC2 nodes first

These experiments were conducted to investigate the scheduler behavior when the scheduling algorithm is run without considering the communication overhead and EC2 nodes are scheduled first. The results are as shown in figure 19 and 20. Just like in figure 17, even in figure 19 we can see that the deadline has been missed in the Map stage.

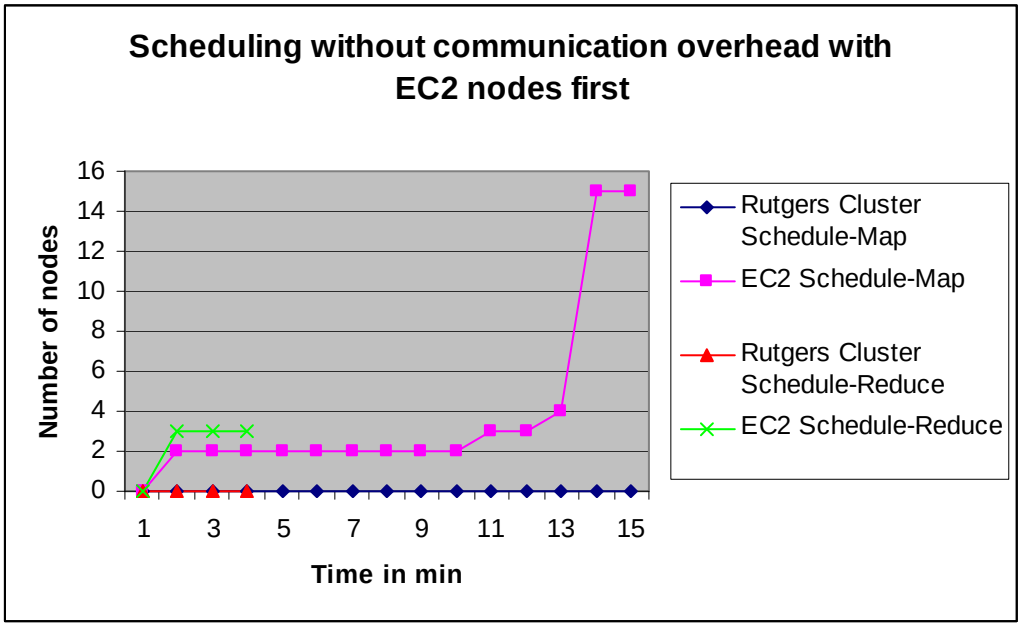

Figure 18 Scheduling without communication overhead with EC2 nodes first

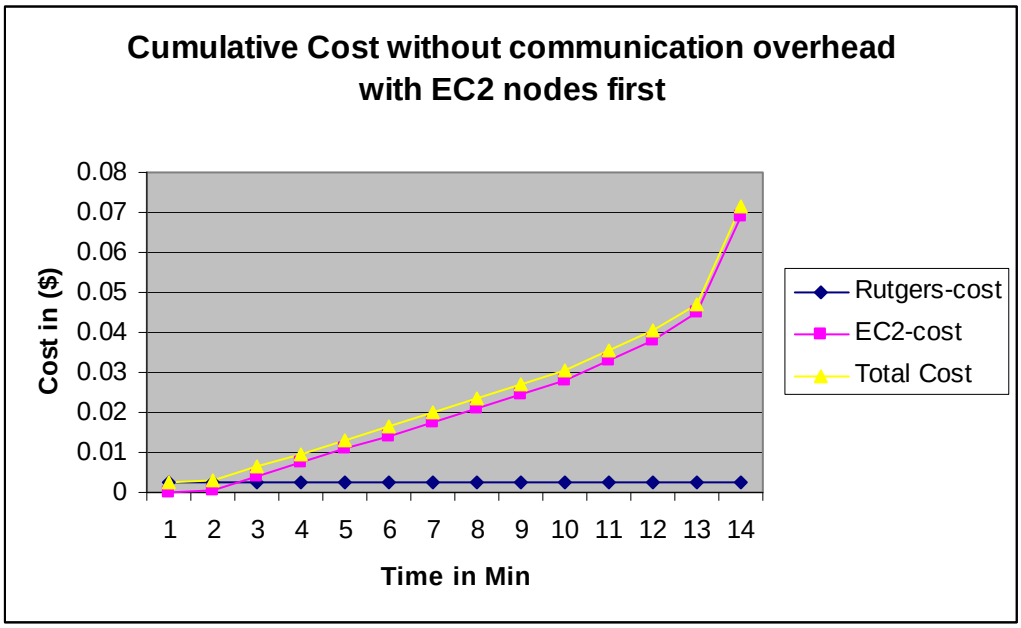

Figure 19 Cumulative Cost without communication overhead with EC2 nodes first

From figures 18 and 20 we can see that the cost of the computation is significantly lower when communication overhead is not considered. The runtime performance when communication overhead is not considered is as shown in figure 21. These experiments show that communication overhead is a significant component that needs to be considered when the nodes are being scheduled.

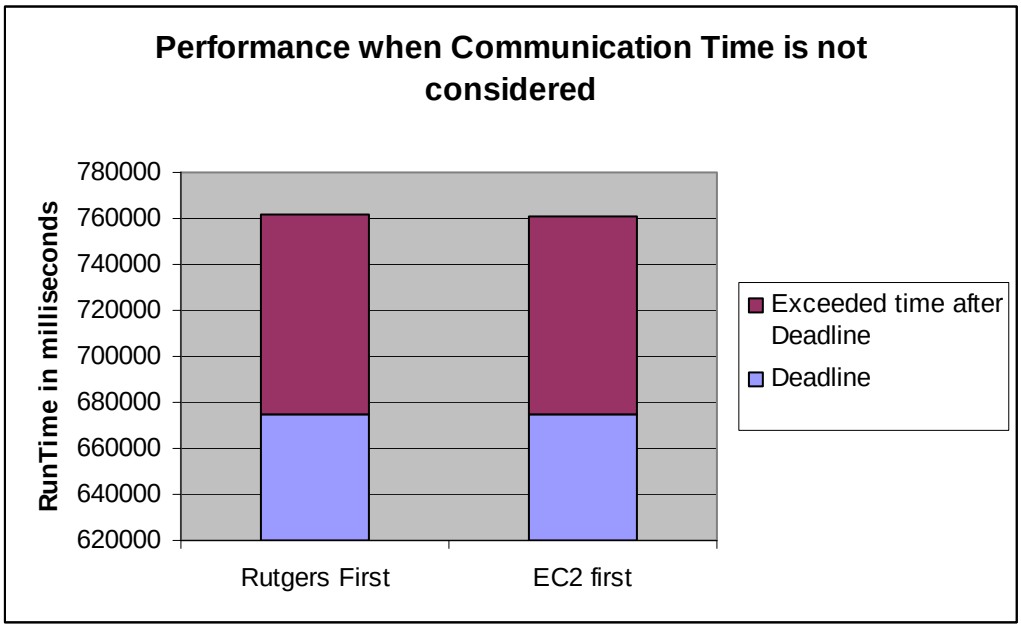

Figure 20 Performance when communication time is not considered

Chapter 6

Summary, Conclusion and Future work

6.1 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 work is the new infrastructure for running MapReduce application subjected to a deadline 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 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. With the 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.

6.2 Conclusion

In this research work we have investigated the use of autonomic cloud bursting for map reduce framework. Cloud bursting gives an abstraction of single virtual compute cloud that integrates both public and private clouds on the fly. The deadline 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 different priority for each cloud. The results have demonstrated the effective use of cloud bursting for MapReduce and have proved that deadline 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.

3.3 Future Work

The concept of deadline based scheduling for MapReduce framework is new. Hence there is a lot of 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 application which 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 and Amazon EC2, but there are many other cloud/grids available today. Hence, it will be an interesting 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.

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. *IEEE Network Computing, Special issue on Information Dissemination 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. *IEEE Transactions on Knowledge and Data 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

.