Dynamic Allocation of Virtual Machines In the Cloud

Author: Eoin McCarthy

Supervisor: Dr. Donal O'Mahony

A dissertation submitted to the University of Dublin, in partial fulfilment of the requirements for the degree of Masters in Computer Science.

Submitted to the University of Dublin, Trinity College, May 2013

Abstract

Dynamic Allocation of Virtual Machines in the Cloud

Author: Eoin McCarthy
University of Dublin, 2013

Supervisor: Dr. Donal O'Mahony

Cloud Computing represents a change in dynamics for the way computers are used. Applications are migrating from local storage to the internet and more and more users are using the internet to access them. New scheduling concerns have come about as a result of decentralising computing power. There are issues such as latency, data locality, monetary cost and environmental impact to consider when choosing a cloud computing solution.

The aim of the research was to select a subset of the new scheduling concerns unique to cloud computing and develop a scheduling algorithm which would focus on this subset. The subset chosen was monetary cost and environmental impact. The system developed allows a user to define the importance of these concerns to them when executing a workload and the system will attempt to minimize either the CO_2 emissions or the monetary cost or both without impacting on the execution time.

Declaration

I, Eoin McCarthy, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University, and that the library may lend or copy any part thereof on request.

Signature:

Date:

Summary

This dissertation aims to provide a design and implementation of a scheduling algorithm which contends with a subset of scheduling concerns which are unique to cloud computing. A state of the art of current cloud computing notions and cloud computing scheduling systems is described. From this research a decision is made to focus on the scheduling concerns of monetary cost and environmental impact from a cloud computing scheduling perspective. A system is designed which utilises Amazon EC2 and HTCondor to implement a scheduling algorithm which uses user input to define priorities for the concerns mentioned. This system is implemented using Python code.

The system is then evaluated under the headings of resource utilisation, execution time, environmental impact and monetary cost. The results are obtained through the use of distributed computing benchmarks being passed through the scheduling system. The results are compared between different use cases and to an unmodified HTCondor scheduled pool of resources. Advantages to using the system are shown in terms of cost savings, carbon emissions without impacting on execution time greatly.

Acknowledgements

I would like to thank my parents for funding my studies and supporting me throughout my life. I would like to thank my friends and family for any and all of their support and assistance to me during the year.

I would like to express my gratitude to my supervisor Donal O'Mahony for his guidance and support over the course of this research undertaking. I would also like to thank the faculty and staff of the Computer Science department for helping me gain the skills I needed in order to complete this Masters course.

Finally, I would like to thank all my classmates throughout my five years of study in Trinity College, I will remember our time together fondly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter will discuss the background to the research undertaken. Then the motivations for the system are discussed. After that, the overall goals of the system are described. Lastly the structure of the dissertation is given.

1.1 Background

Cloud computing has become more and more prevalent in recent years in computer science. It represents a shift in the usage of computers away from computers existing as a single unconnected node, on which all software is installed locally and run, to a node on a network which can utilise compute resources and software from other nodes on the network remotely. This notion of cloud computing includes a multitude of very different services.

With this migration from local to distributed compute resources, a number of changes have come about in the field of scheduling. Traditional OS(Operating System) scheduling deals with scheduling work to a single/multiple processors on a single physical computer. With the advent of cloud computing, work can be scheduled to numerous physical computers which can be located hundreds of miles from one another, each with multiple processors. The computers which perform the computation can also be located hundreds of miles from the submitting machine, and from the data which is being processed. With this knowledge, new scheduling concerns need to be dealt with. These include latency (between computers), data locality, monetary cost and environmental impact. The system detailed in this dissertation deals with monetary cost and environmentally impact.

1.2 Motivation

The primary motivation for pursuing a dissertation in this area was the fact that this mastery is still in its infancy. Cloud computing as mentioned is still a new phenomenon in the area of computer science. With traditional operating system scheduling there are already several widely accepted standards and very little experimentation outside of these methods/algorithms. However within cloud computing scheduling system a single standard has not yet emerged due to the greater number of factors involved. This ensures that there is a lot of research work ongoing, and ensured that this topic was an interesting and worthwhile topic to research and contribute to.

The reason that environmental impact was selected as one of the concerns to deal with is that the environment is a very hot topic nowadays and cloud computing providers are under scrutiny about the amount of energy their DCs (data centres) produce, and the damage to the environment which is caused by these. Cloud service providers can achieve a lot of energy savings with the design and technologies in use in their DCs, but they have little control over the software which runs on their rented resources, meaning that more energy savings could be achieved from the user end, which is what the system described attempts to do.

The reason that monetary cost was selected as one of the concerns is that lowering monetary costs is already one of the reasons why end users and corporations are migrating to cloud computing solutions. To allow for further savings on top of the already substantial savings offered to users of these services would be a great incentive.

1.3 Aims and Objectives

This project aims to develop a cloud computing scheduling system which will allow users to define priorities for the importance of environmental impact and monetary cost to them for the work they are submitting. The system will then minimize the carbon emissions, cost or both over the execution time of this job.

This aim is achieved through the completion of three tasks. Firstly, a state of the art review is given of the current state of cloud computing scheduling. Second a prototype of the system is designed for a specific cloud computing service (Amazon Elastic Cloud

Compute). Finally the system is implemented on this service and evaluated.

1.4 Dissertation Structure

A review of the State of the Art of this area is given in Chapter 2. From this research a system will be conceived. A description of this system is given in Chapters 3 and 4 at a high level and low level respectively. Chapter 5 describes the evaluation methods for the system and the results obtained. Finally, Chapter 6 details the conclusions which can be taken from the results obtained and any further work which can be done on the system.

Chapter 2

State of the Art

A discussion of the current state of the art of cloud computing scheduling, beginning with an introduction to cloud computing. A description of the current service models in practice in cloud computing are given, with a focus on cloud infrastructure services and Amazon EC2. Then a description of current cloud computing scheduling solutions will be given with a focus on HTCondor.

2.1 Cloud Computing

Cloud computing is a new phenomenon in the realm of computing science. Recently a paradigm shift from the use of local compute power to the use of computing power which is served over the internet has come about. The term cloud computing is a relatively new term which came to popularity sometime during the years 2006 and 2007. It is a blanket term referring to a number of different technological services all of which share certain attributes or relationships which enable them to be labelled "cloud computing". The popularity of the term can be attributed to the companies Google and Amazon who launched services (Google Docs and Amazon Web Services) around this time. These services were the first widely used cloud computing services wherein people could access applications remotely over the internet instead of having them installed locally on their computers.

2.1.1 Introduction

The term cloud computing refers to a vast number of different kinds of technological services (Armbrust et al. (2010)), such as Grid computing, cluster computing, web applications and many more. The commonality between these services is that they all involve the usage of compute resources which reside on a network and are accessed remotely by the user in some way. The key is that there is an abstraction between the user and the physical hardware which they are performing computing on. Another key notion in cloud computing is the notion of an infinite pool of resources. While the actual amount of resources is not infinite, to a user the illusion of infinite resources is created as compute resources can be requested and relinquished as required. The cloud computing service models described in this section are taken from Mell & Grance (2011). The different types of cloud computing services offered can be arranged from a low level of abstraction to a high level, similar to Youssef et al. (2008).

2.1.2 Infrastructure as a Service

At the lowest level of abstraction from user to hardware, IaaS (Infrastructure as a Service) exists. These services take the form of users actually requisitioning hardware from a service provider, either in the form of virtual machines or storage. A big incentive to use IaaS services is that rather than purchase physical hardware which may not be used at all times, a user or company can only requisition the hardware virtually from a service provider when it's needed which can lead to lower costs. The user in this case has the highest amount of control over the software running on the hardware, down to the OS that is installed on it in the case of virtual machines. The service provider is only in charge of the maintenance of the physical hardware. Amazon Elastic Cloud Compute is an example of an IaaS cloud service provider which will described in detail as this is the service on which the system designed runs on. Other cloud infrastructure service providers include Dropbox (2013) (Cloud Storage) and Rackspace (2013) (Virtualization).

2.1.3 Amazon EC2 Overview

Amazon EC2 (Amazon (2013)) is a cloud virtualization service offered by Amazon as part of their AWS (Amazon Web Services) suite. In its most basic form users can provision virtual servers from Amazon at a fixed cost per hour from a number of different locations. Amazon has data centres all over the world from which virtual machines can be requested; Dublin, California, Singapore, Sydney, Sao Paolo to name a few. A user logs onto the AWS console and launches an "instance", which is a virtual machine. The machines come in different instance types depending on the compute power required. The instance types come in many different sizes. The standard sizes range from m1.small with 1.7GB RAM and a 1 core CPU to m1.xlarge with 15GB RAM and a 4 core CPU. There are also High Memory, High CPU and even Cluster Compute instance types which are specialised for very high performance computing work. Instance payment comes in two flavours, on demand instances and spot instances. On demand instances cost a set price per hour while spot instance prices fluctuate.

When a user launches an instance on EC2, they select an AMI(Amazon Machine Image) which decides the Operating System and software that will come preconfigured on their instance. Amazon provides numerous basic AMIs for many different Operating systems, and AMIs with premium software can be purchased on Amazon's AMI marketplace. Users can also create their own AMIs defining the OS and software they want to start up on any instances they request.

Spot Instances

Spot instances are typically considerably (up to 90%) cheaper than on demand instances, for example, at the time of writing the spot price for a m1.medium sized instance type in Ireland is \$0.032 compared to the on demand price of \$0.13. Spot instances are offered by Amazon when the demand for ”on-demand” instances in a DC does not meet the supply. If this happens there are unused compute resources in a DC which would represent a waste of energy and resources. In this case Amazon offers these unused resources at a cheaper price than the on demand instances. While Amazon claim that the price for these instances is based on supply and demand (Amazon (2013b)), it is suggested by Ben-Yahuda et al. (2011) that the price is actually calculated through a hidden reserve price.

Spot instances are not requested in the same way as an on demand instance. Instead, users bid on them, setting a maximum price that they are willing to spend per hour on the spot instance. As long as the current spot price does not exceed a user's maximum bid, they will be provisioned the instance and pay the current spot price. If, however, the current spot price increases and exceeds a user's maximum bid, Amazon will take back the instance immediately and without warning, meaning that usually work that requires reliability should not be run on spot instances. If Amazon retake an instance, the user will not pay for the hour during which this occurs. Numerous companies have migrated to using EC2 spot instances over on demand instances in order to save money, for analytics (GoSquared (2013)), image processing (SaltedServices (2013)), scientific research (DNAnexus (2013)) and more.

2.1.4 Platform as a Service

The next level of abstraction up from IaaS is known as PaaS (Platform as a service). In PaaS the service provider controls a platform for a user to develop, test and host software on. The user is responsible for maintaining and deploying the software which they created, and the service provider is responsible for the physical hardware, and maintaining the platform. The platform usually takes the form of services which allow a user to develop, deploy and host applications without needing to worry about the infrastructure underlying the application. The benefits of PaaS include ease of use, as the developer is free to focus on development rather than setting up development environments and hosting solutions, and scalability, as the platforms usually offer automatic scaling depending on the usage of the software developed. Examples of popular PaaS solutions include Google App Engine (Google (2013b)) and Windows Azure (Microsoft (2013b)).

2.1.5 Software as a Service

The model with the highest level of abstraction between the user and the physical hardware is SaaS (Software as a Service). Users access SaaS applications over the internet rather than installing them locally on their own computers. The users are only responsible for the usage of the applications, and the maintenance of both the application and the physical hardware falls to the service provider. An exception to this is that if the application is actually being hosted on a PaaS service itself meaning that the different models can stack on top of one another, i.e. an application developed using a PaaS service can be made available as a SaaS application. The benefits of SaaS include platform independence, as applications are usually accessed over a web browser meaning that all that is required is an internet connection and browser software, and low cost, as again high powered hardware is not required to access SaaS applications. Popular SaaS applications include Gmail (Google (2013a)) and Office 365 (Microsoft (2013a)).

2.2 Cloud Computing Scheduling

With this migration to cloud computing has come a change in the concerns which need to be contended with in scheduling systems. Traditional scheduling systems only need to schedule onto a single processor, or multiple processors on the same physical machine. Cloud computing scheduling systems have the possibility of scheduling work to different processors which could be distributed all over the world. As a result of this, scheduling work can be a much more complex task as there are more factors to take into consideration. There are currently a multitude of different cloud computing scheduling solutions in existence or development. This section focuses on the HTCondor scheduling system as it is used in the eventual system implementation.

2.2.1 HTCondor Overview

HTCondor is a system designed by the University of Wisconsin-Madison Department of Computer Science. It is a distributed job scheduler for high throughput computing (Tannenbaum et al. (2001)). HTCondor can be considered an opportunistic scheduler. The unique features HTCondor consists of are: Machine Pools, ClassAds and Job Checkpointing.

Machine Pools

In HTCondor, a machine pool consists of the machines which are currently available to execute jobs. A machine pool is a dynamic entity, allowing machines to enter and leave the pool. It consists of a single central manager, one or more execute nodes and one or more submit nodes. A node in this case refers to a machine in the network. The central manager is the controller of the pool. On this node, the daemons for matchmaking and checkpointing run. An execute node advertises that it is available to execute work. A submit node is authorised to submit jobs to the pool. A single node can occupy more than one of these positions i.e. a single node can submit and execute jobs, or the central manager can also submit jobs.

The traditional use of HTCondor is for an opportunistic pool of machines. On a campus, for instance, many machines will be registered as execute nodes and will dynamically enter the pool when they become idle for 15 minutes. This way unused cycles of idle machines can be used to contribute to grid computing jobs.

ClassAds

Another unique aspect of HTCondor are ClassAds. ClassAds are the mechanism by which the HTCondor central manager of a pool matches a job submitted by a submit node to an execute node. A job is submitted in the form of a submit file, which details the requirements for a job along with the name of the executable, and some other parameters to do with transferring files to the execute machine. HTCondor then translates this submit file and its parameters into a ClassAd. This is known as a Job ClassAd. Alongside this is a Machine ClassAd. A Machine ClassAd is created by HTCondor for each execute machine in the pool. The entries in a ClassAd are in the form of a 'name = value' statement.

When a job is submitted, the Central Manager node then compares the statements in the Job ClassAd to all Machine ClassAds in the pool. The job is distributed to a machine if the two match. This process is known as match making in HTCondor and is the process by which it schedules work.

Job Checkpointing

Due to the opportunistic nature of HTCondor's dynamic machine pools, a machine can become unavailable to a pool if the machine ceases being idle. This would mean that a job running on these machines would be interrupted. However, HTCondor allows for checkpointing of the state of a job. If the job needs to be pre-empted, such as if the machine it is running on becomes active again, the job re-enters the queue. Once it is reassigned to a machine, the job continues from the most recent checkpoint which is stored on the Submit node. This checkpointing system fits well the use of Amazon EC2 spot instances.

2.2.2 Other Cloud Computing Scheduling Systems

The previous section focused on HTCondor which is an opportunistic High Throughput cloud scheduling system. However, as mentioned previously there are many other scheduling systems which attempt to focus on other concerns. In this section I will detail some of the other current research concerning these systems.

One of the concerns which the system detailed in this report focuses on is carbon emissions. The incentive to minimise carbon emissions comes from the topicality of climate change currently. Cloud computing service providers are under pressure to release the information about the energy their data centres produce as having multitudes of servers powered on at one place not only costs energy to run them, but also significant amounts of energy in order to cool the DC and distribute power. As a result of this, the pursuit of scheduling systems which can minimize the carbon emissions of workloads in a cloud computing environment is a meaningful one. One solution proposed by Goiri et al. (2012) is to schedule workloads to servers at times when most green energy will be available, predicting the amount of solar energy that will be available. Another method proposed by Doyle et al. (2011) and Hatzoupoulos et al. (2013) is to move the workloads around to data centres that emit less carbon or have more green energy available.

Another concern with cloud computing scheduling is that of data locality. This means that the server which is executing a job on some data may be on another rack, or even in another data centre from the server which stores the data which is being processed. This can cause overhead in the transfer of data which can slow down the execution time of the job. Jin et al. (2011) presents a data locality driven scheduler for a cloud computing system which dynamically alters data locality depending on the current load. Hindman et al. (2010) presents a scheduler which allows multiple scheduling frameworks to operate on a single node, improving resource utilisation and data locality. Malik et al. (2012) presents a solution for a multi-cloud environment which aims to minimize latency by grouping nodes in the cloud together based on their inter node latency.

It can be seen then that there are numerous new factors to contend with in the design of a cloud computing scheduling system. The majority of these scheduling systems focus on a single, or up to two, scheduling concerns leaving the user little option in terms of varying the importance of the concerns. The system detailed in this dissertation offers user choice in terms of the focus of the scheduling algorithm. This leads to a more flexible scheduling system from the user's perspective.

2.3 Conclusion

This section details the current state of cloud computing scheduling, with a focus on the technologies which were used in the system implemented. The project aims to make a contribution to this area by building upon ideas which were described or referenced to in this section. The project uses a cloud IaaS service and a cloud computing scheduling system and builds up a scheduling system on top of these which aims to minimize cost and carbon emissions without impacting on job execution time.

Cloud computing is a new area and many new scheduling systems have emerged focusing on different concerns. From an environmental perspective, provisioning machines depending on geographical parameters can be used to minimize the carbon cost of a job. The system implemented uses geographical parameters to also minimize monetary cost of a job.

Chapter 3

Design

This chapter describes the high level design of a system which serves to contend with some of the issues which were discussed in Chapter 2. Namely the system attempts to offer a user-driven method of specifying certain scheduling concerns with a job and then submitting and executing the job with these concerns in mind. The scheduling concerns the system attempts to address are execution time, carbon emissions and monetary cost. A user specifies which one (or more) of these they want to focus on when submitting a job/jobs, and the system will attempt to minimize that aspect over the course of the execution of the job(s).

3.1 System Architecture

The system consists of three separate modules: a job submitter, a queue manager and a server manager. An architecture diagram of the system can be seen in Figure 3.1.

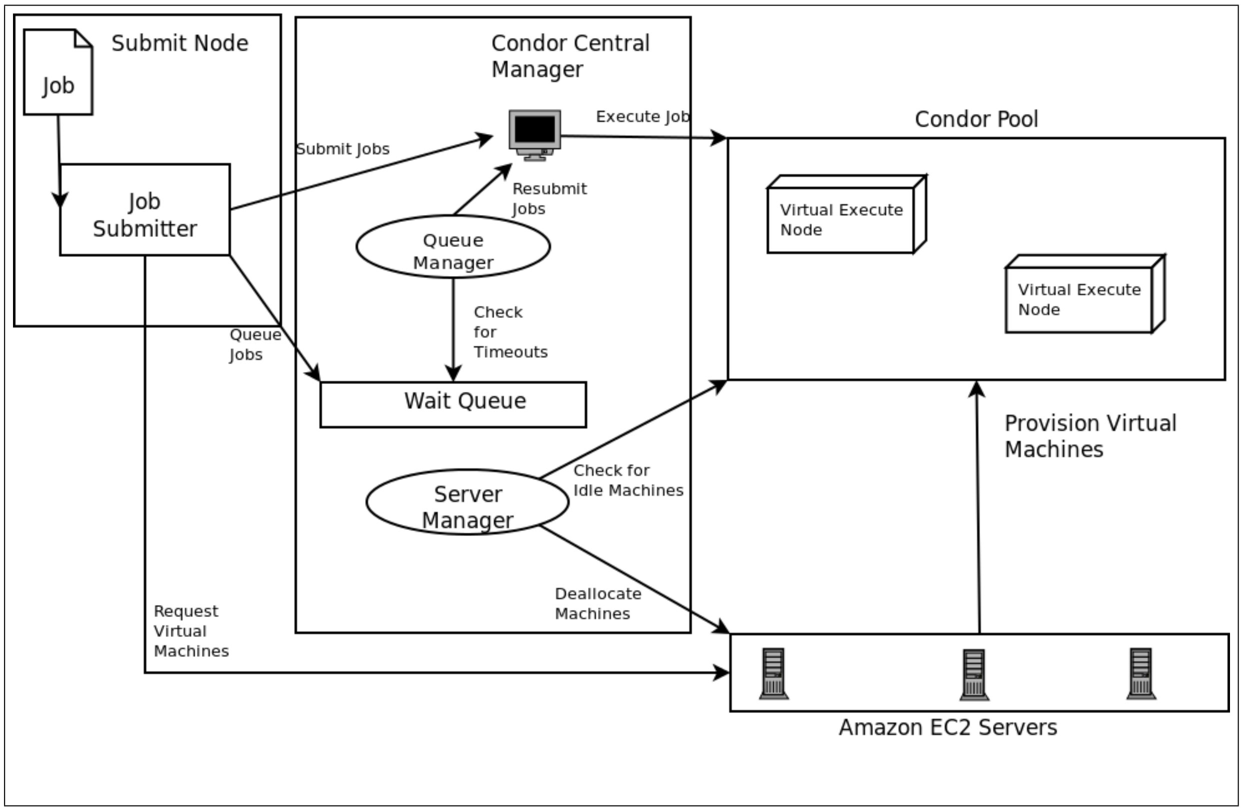

Figure 3.1: *High level architecture diagram of the system.*

In this diagram it can be seen that the two manager modules; queue and server managers, reside on the HTCondor Central Manager node. The job submitter must reside on a Submit node. The job submitter is responsible for communicating with the Central Manager in order to submit jobs to the execute pool. The job submitter also sends jobs, when necessary, to the wait queue which resides on the Central Manager. Lastly the job submitter communicates with Amazon EC2 in order to provision instances when needed.

The server manager communicates with the HTCondor execute pool and with Amazon EC2, to deallocate idle machines when appropriate. The queue manager communicates with the wait queue and the Central Manager in order to detect timeouts on waiting jobs and to resubmit jobs when necessary. The HTCondor pool is made up of provisioned virtual EC2 instances which act solely as execute nodes.

This pool of EC2 instances as execute nodes is similar to how HTCondor is implemented on Rocks Clusters (Papadopoulos (2011)). The difference between the system and Rocks Clusters is that to extend a pool on Rocks Clusters a user must manually run and terminate instances. There is no automatic provisioning of instances at job submission or terminating of idle instances. Also on Rocks Clusters the ability to extend a user's pool to EC2 instances is simply to increase the amount of computing power available, and no dynamic location based provisioning is done. Juve et al. (2009) also uses a HTCondor pool of EC2 instances to execute scientific workflows, but again the pool of machines was pre provisioned and static.

3.2 Algorithms

The user inputs to the algorithm which were mentioned previously which control the scheduling of work are Priority, CO_2 Importance and Monetary Cost. The options for Priority are High, Medium or Low. The options for CO_2 Importance are High or Low. For monetary cost, a user can enter a maximum cost per hour that they are willing to spend for the job to complete. A last input that a user needs to specify is the number of processes over which to distribute the job. This parameter is both a user choice and a necessary parameter to a HTCondor job. The necessity of this input is a constraint of the HTCondor matchmaking service which requires this knowledge in order to correctly distribute a job to more than one node.

Pseudo-code versions of the algorithms controlling the system are presented below. An epoch is a period of time chosen to pass between cycles of the manager modules are run. The use of epochs for managing servers is similar to that of Goiri et al. (2012). The epoch selected for the system was 10 minutes. This number could be increased or decreased depending on the preciseness of control a user requires.

```
Every Epoch:


Query Status of Waiting Queue()
for each job in Waiting Queue:
        Query Status of HTCondor Pool()
        if (num(unclaimed machines)>=(machines_required)):
                submit job
        else if (submission_time > (2 hours - epoch) ago)):
                request instances()
                submit job


Query Status of HTCondor Pool()
if (num(unclaimed machines)!=0):
        terminate(unclaimed machines)
```

Figure 3.2: *Pseudo Code listing of the algorithms for the server and queue managers*

The line *Query status of Waiting Queue* is a method which returns a listing of all the jobs which are in the waiting queue along with the information from their submit files. The line *Query status of HTCondor Pool* is a method which returns a list of the machines currently in the pool and their status. The *submit job* line refers to submitting the job to be matchmade by the HTCondor scheduler.

```
At Job Submission:


Calculate optimal instance type
Locate cheapest spot instances
Locate greenest Data centre (DC)


if (job priority == High):
        request instance() in nearest DC
else:
        Query status of HTCondor Pool()
        if (num(unclaimed machines)>=(machines_required)):
                submit job
        else if (job priority == Medium):
                if (CO2 Importance == High):
                        if (Max Cost < on demand instance cost):
                                request spot instances in greenest DC
                                submit job
                        else:
                                request on demand instances in greenest DC
                                submit job
                else:
                        if (Max Cost < on demand instance cost):
                                request spot instances in cheapest DC
                                submit job
                        else:
                                request on demand instances in cheapest DC
                                submit job
        else if (job priority == Low):
                queue job in waiting queue
```

Figure 3.3: *Pseudo Code listing of the algorithm for the job submitter*

As can be observed from Figure 3.2, the queue manager is in charge of ensuring that jobs do not remain in the waiting queue for too long. The timeout value of one hour was arbitrarily selected for testing purposes which are described in Chapter 5. The purpose of the waiting queue is to complete low priority jobs solely by reusing idle instances (unclaimed machines) in the pool instead of provisioning new instances. However without a timeout on the waiting time, jobs could hypothetically end up waiting indefinitely, which would not be ideal. For each job in the waiting queue, the queue manager first checks if there are enough unclaimed machines in the pool to run the job. If not, the manager checks whether the job has been waiting long enough to reach the timeout value before the next check. If so, instances are provisioned and the job is submitted. Otherwise the job remains in the waiting queue for another epoch.

The server manager is in charge of ensuring that idle machines do not remain in the pool for too long after they were provisioned. In order to achieve this, firstly the server manager runs after the queue manager ensuring that any unclaimed machines will be picked up by queued jobs if necessary. Once this phase has passed, any unclaimed machines left in the pool will be released from the pool back to Amazon in order to reduce monetary and carbon costs which would be incurred by maintaining idle powered on machines for no reason.

The job submitter module is the only user facing module. It is responsible for creating submit files to be sent to the HTCondor scheduler, as well as requesting any necessary instances from Amazon EC2. As mentioned previously, a user is also queried for the number of processes needed if their job is a parallel program. The job submitter calculates the optimal instance type to provision based on the max price the user is willing to pay and the number of processes needed. Then the availability zone with the cheapest spot instance prices at that time is located. Finally the "greenest" data centre is located.

3.2.1 Calculating the Optimal Instance Type

Given the number of processes required by a job, and the maximum price per hour, a decision can be made regarding the optimal instance type to provision for a job. Firstly the maximum price is divided by the number of processes which gives the max price per process. Next, values for price per instance process and price per spot instance process are calculated. For example an m1.large instance type provides 2 processes usable by HTCondor. The price of an m1.large on demand instance for an hour is \$0.26. This means that the price per instance process for an m1.large instance is \$0.13. If a user sets their maximum price per hour for an 4 process job at greater than or equal to \$0.52, the optimal instance type returned will be m1.large. The pseudo code listing of the algorithm to calculate this is shown in Figure 3.4. This algorithm ensures that the highest powered instance is used. The check to ensure that the number of processes required divides evenly into the number of processes provided by an instance is to ensure that instance types are used which can cater for all processes required i.e. that m1.large instance types are not provisioned when 3 processes are required for the job, as there would always be either too many or too few nodes available at any one time.

```
price per process = max price / num processes
for each instance type (starting at m1.large and decreasing in size) :
        if (num process % processes provided = 0):
        price per instance process = price per hour / processes provided
        spot price per instance process =
                                spot price per hour / processes provided
        if (job priority == high)
                if (price/process >= price/instance process):
                        optimal instance = current type
                        break
        else:
                if (price/process >= price/instance process):
                        optimal instance = current type
                        break
                if (price/process >= price/spot instance process)
                        optimal instance = current type
                        break
result = optimal instance
```

Figure 3.4: *Pseudo Code listing of the algorithm to decide the optimal instance type*

3.2.2 Locating the Cheapest Spot Instances

Once the instance type to be used has been determined, the next step is to determine which availability zone offers the cheapest spot instance of this type. In the different data centres the price for spot instances fluctuates meaning that, depending on uptake in a particular data centre, the price could be unusually low or high; thus the cheapest availability zone can and does change over time. This process simply requires querying Amazon and retrieving the spot prices for the specified instance type in all the availability zones at the time and then selecting the lowest result and noting this as the cheapest availability zone.

3.2.3 Locating the Greenest Data Centre

Doyle et al. (2011) makes the argument that the carbon emissions caused by powering servers can vary greatly over different locations. This means that by discovering the data centre which uses the most green energy, the carbon produced by a job can be minimized. In order to locate the ”greenest” data centre, the system requires making some assumptions. We assume that all data centres make as much use of green energy as possible, meaning that if there is more energy provided by renewable resources in the location of the data centre, the data centre itself will be more green. So to locate the greenest data centre, the system will attempt to discover the location with most renewable energy available.

The two primary sources of renewable energy are solar power and wind. The sunnier and windier a location is, the more power will be generated through these sources (Sharma et al. (2010)). Thus we can use the wind speed in a location as a proxy for the wind energy produced in that location. Similarly in order to measure solar energy produced it is challenging to quantify ”sunniness”, however the cloud coverage in an area can be used as an inverse proxy for the solar energy produced there i.e. the higher the cloud coverage in an area, the less solar energy. In order to use these values, we assume that all the locations have an equal amount of green and brown electricity generators and the only thing that affects the green energy produced is the availability of wind and sun. This is a naive assumption that suffices as a simple metric for the system as it stands, however the system could easily be modified to use a more sophisticated method of deciding the greenest data centre.

Overall a quantified measure of the greenness of a location at any point in time can be given by $Windspeed/MaxWindspeed + (T) * (100 - CloudCoverage)$, where T is a value between 0 and 1 depending on the time of day. This ensures that at night time, when no solar energy would be acquired, the solar energy value is not used as $T = 0.0$. The MaxWindspeed figure is the most common rated wind speed for commercial wind turbines. This is the wind speed at which common wind turbines will output most power. The value for this is set to be 30mph (Windpower-Program (2013)). Dividing the current windspeed by this, and setting the max result of this division to be 1, removes the conflicting units from this equation. This metric is extremely simple however it appears to give an accurate proxy of carbon intensity when compared to historical data retrieved from Eirgrid (2013). To test the accuracy of the metric real time information for carbon intensity over five days was recorded from Eirgrid. Then, data for the windspeed and cloud coverage for the same days was recorded from Intellicast. With these, the metric of greenness was calculated at these times and compared to the carbon intensity at the time. From these tests it was observed that the higher the value of greenness, the lower the carbon intensity value. This metric also gives a simple numerical value which can be used to compare greenness between availability centres in order to calculate the greenest.

Once the three calculations have been done by the job submitter, it moves onto submission phase. The first input it considers is the job Priority. If the Priority is high, all other inputs are discarded and an instance is immediately provisioned in the nearest data centre and the job is submitted. However if the Priority is not high the job submitter moves into a more complicated submission algorithm. From there, firstly the machine pool is queried. If there are enough unclaimed machines (of any type) to satisfy the job in question, the submitter immediately submits the job. However if there is not, again the submitter must do more work. If then, the job Priority is found to be low, the job is immediately sent to the Wait queue where it will wait for a machine to become idle.

At this stage if the job Priority is found to be medium, the other inputs come into effect. Firstly the CO_2 importance input is checked. If this is High then from here any instances requested will be in the greenest data centre as calculated previously. From here the max cost input is checked against the on demand cost of the instance type which was decided at the beginning of the algorithm. If the max cost is lower than the on demand cost, spot instances are requested in the greenest data centre, otherwise on demand instances are used. The only difference if the CO_2 importance is Low is instead of the instances being requested in the greenest data centre, they are requested in the cheapest data centre, which was also decided at the beginning of the algorithm.

This algorithm achieves the aim of designing a user driven system which allows jobs to be completed while reducing one or more different cloud computing scheduling concerns.

Chapter 4

Implementation

This chapter discusses the low level implementation which was performed in order to implement the system. The system consists of 3 components managing a HTCondor pool of Amazon EC2 instances. Also, each pre-existing element of this architecture(machine pool and EC2 instances) required some level of modification in order to operate as a single system.

4.1 HTCondor and Amazon EC2 Instances

The first task which was implemented was the creation of an Amazon Machine Image(AMI) which would allow newly launched instances to automatically join a HTCondor pool. Also necessary to implement on these instances were some non-standard attributes to the Machine ClassAd which would allow the system to more accurately match jobs to the correct machines which were launched to execute them.

To do this an Amazon instance was configured to connect to the system's HTCondor pool. Once that was completed a start up script was created which automatically invoked the HTCondor daemons on instance launch. This script also adds attributes to the Machine ClassAd for the availability zone of the instance, the Amazon set instance ID, the instance type and the JobID, if specified. These attributes are used by the matchmaker to ensure that the correct jobs match to the correct machines. Once this had been done, an AMI was created from this instance. This AMI was copied to all availability zones.

4.2 Job Submitter

The Job Submitter is the most complex component in the system. It has five different phases during its execution. The first phase is reading in the user's input. The second phase is calculating the three global variables required, which are described in Section 3.4. The third phase is building the HTCondor submit file for submission to the Central Manager. The fourth phase is requesting Amazon EC2 instances. The final phase is either submitting or queuing the job as required. The job submitter consists of roughly 400 lines of Python code.

Reading the User Input

The program firstly asks the user a series of questions to do with their requirements for their jobs. The first question is whether the job is a parallel job or a single process job. Next the program asks for the different user inputs: Time Priority (High/Med/Low), Green Priority(High/Low) and Max Cost. The final question posed to a user is the number of processes required for the job.

4.2.1 Calculating the Optimal Instance Type

When calculating the Optimal instance type for use for the job being submitted, the Priority, Cost and process count requirements are used. A text file is stored in the directory of the job submitted which contains a listing of the instance types and their corresponding on demand prices and processes offered. Also required is the current spot price for each instance type. This can be retrieved directly from Amazon. Amazon offers a web service interface for retrieving information about current running instances and requesting new instances or terminating running instances. It can return the current spot price for an instance type.

boto

The library used for interfacing with the Amazon EC2 web services is a python external library called boto (2013). This library takes care of formatting web service requests and responses and serves them in a concise and easy to process manner. When using the library, the first action is the creation of a connection object to an availability zone. To connect to an availability zone, an availability zone object is needed. Availability zone objects can be returned through the boto call: *regions = boto.ec2.regions()*. This method returns a list of all of the online availability zones. By checking *region.name* for the entries of this list, the location of these list entries can be discovered. With this information, a connection object to any availability zone can be created.

Once a connection object has been created, the next step was to return the spot price for each instance type. Boto offers a web service called *get_spot_price_history()* which takes in an instance type, start time and end time, and returns a list of the recent spot price values between the start and end times requested, the last entry in the list being the most recent. In order to retrieve a current spot price, a request for the spot price history for the previous hour is made. Once the instance type list has been fully populated, the algorithm which was described in Figure 3.4 is performed resulting in a string value for the optimal instance.

4.2.2 Locating the Cheapest Spot Instances

In order to locate the availability zone with the cheapest spot instances, the spot instances for each region for the optimal instance type are retrieved and the minimum is selected.

4.2.3 Locating the Greenest Data Centre

As described in Section 3.2 the method for deciding which data centre is the greenest data centre involves comparing a combination of the Windspeed and Cloud Coverage values for each location and finding the maximum. The method to do this involves three separate methods: one method retrieves the Windspeed for a location, one method retrieves the Cloud Coverage for a location and the last compares the value for each location and returns an answer. Values for windspeed and cloud coverage are available from the website *www.intellicast.com*. The local weather for a location is stored at a URL in the form *http://www.intellicast.com/Local/Weather.aspx?location=XXXXXX* where XXXXXX is replaced by a code for the location. The location codes for each Amazon data centre are known to the system.

With this information each method scrapes the appropriate data from the Intellicast website using a library called Beautiful Soup. With the wind speed and cloud coverage values returned and saved for each location, the values can be combined into a single "greenness" value and compared. This value is calculated using the formula described in Section 3.2.

4.2.4 Building the Submit File

With the three global variables calculated the next task was to build the submit file for the HTCondor scheduler. A submit file consists of the requirements to be sent to the Job ClassAd as well as some instructions to the HTCondor Central manager to do with transferring files and outputting/logging information. A submit file created by the program can be seen in Figure 4.1.

```
Executable      =
Universe        = Parallel
Arguments       =
Log             = job6362442.log
Output          = job6362442.out
error           = job6362442.err
machine_count = 2
transfer_input_files =
should_transfer_files = yes
when_to_transfer_output = on_exit
Requirements    = EC2InstType == "m1.small" && EC2JobID == "6362442"
+Priority = "M"
+Green_Priority = "H"
+Submitted = "25-03-2013T00:19:57"
Queue
```

Figure 4.1: *Sample submit file*

In this example the new attributes which are used by the system can be seen. The number (6362442) is the *JobID* which is a number used to ensure matching of jobs to the correct machine during submission. This JobID is only necessary in High Priority jobs, and Medium Priority jobs when new instances are required to be launched. The requirements parameter is an attribute for a Job ClassAd which tells the scheduler that only a machine with the exact same statements in its Machine ClassAd can be matched to this job. In the example provided, the job will only match to a m1.small sized instance that also has the JobID 6362442. The final three parameters begin with a +. This + sign means that the parameters afterwards are not requirements and are just extra information to be stored in the Job ClassAd.

A JobID is only required when the job requires new instances to be launched. If there are idle instances in the HTCondor pool, and if the job isn't High Priority, the system will match the job to idle instances. If the submit file has a JobID, it will not match to a pre existing machine in the pool as the JobIDs in the ClassAds will differ. So, before adding the JobID parameter to a submit file, the system checks whether there are enough idle instances in the pool to execute the job. Once the submit file string has been built by the program, it is written out to a file in the submit directory.

4.2.5 Requesting Instances

The next method in the program is responsible with requesting Amazon EC2 instances. If, according to the algorithm shown in Figure 3.3, instances are required to be requested, the system uses the boto library methods to request the correct amount of instances in the appropriate locations depending on the user's input. If spot instances are required according to the user's max price per hour, then the boto library method call for creating a spot instance request is used which is similar to requesting on demand instances excepting for a max bid parameter. If the job time priority is Low then no new machines are required to be launched.

4.2.6 Submitting Jobs/Queuing Jobs

The last action the job submitter performs is submitting or queuing a job. To submit a job, the system uses the HTCondor submission process with the submit file previously created by the program.

In the case where the job is a Low priority job, it must be placed in a wait queue which is maintained on the Central Manager node. The wait queue on this manager takes the form of a folder on which the Queue manager is run every epoch. To submit a job to the wait queue, the submit file, executable file and any input files must be transferred securely to this folder on the Central Manager node over the SSH protocol. The executable file can be identified through the *Executable* parameter of the submit file, and any input files can be identified through the *transfer_input_files* parameter of the submit file. Once this phase is complete the job submitter completes execution.

4.3 Server Manager

The purpose of the server manager is to ensure that idle machines don't remain in the pool for too long, incurring both economic and green costs. The server manager first identifies unclaimed machines and then terminates them. To terminate an instance the instance-id is required which is stored in the Machine ClassAd. This can be retrieved by querying the status of the HTCondor pool. Once a list of instance IDs for idle machines has been compiled, next the server manager must use the boto interface to terminate these instances. The boto call for terminating an instance is *conn.terminate_instances(instance_id).* The program simply connects to every EC2 availability zone and iterates through the list of Instance IDs attempting to terminate them.

4.4 Queue Manager

The Queue Manager matches waiting jobs to idle instances and ensures that no job is in the wait queue for too long. The Queue manager first reads in the submit files of all of the files in the WaitQueue directory on the Central Manager. For every job, the Manager counts the number of unclaimed instances of the type required by that job in the pool. If there are enough idle instances, the submit file is submitted to the Central Manager.

If there aren't enough idle instances then the time that the job was submitted must be checked. If this time is over the timeout length minus the epoch length (110 minutes in the implementation described) from the current time then the job has been waiting for too long. In this case, the appropriate number and type of instances are requested and the job is submitted to the Central Manager.

When a job leaves the Wait Queue, the submit file is copied into a SubmitFile directory outside the WaitingQueue directory and the original file is deleted from this directory, ensuring that on the next round the same job won't be checked again. If neither of the conditions for submitting a job are met, then the job remains in the wait queue for another epoch.

Chapter 5

Evaluation and Testing

This chapter discusses the methodology used in evaluating the system, and presents the results gathered. The system was evaluated under a number of different headings. These are resource utilisation, execution time, green energy usage and monetary cost. The system is aimed to reduce monetary cost and increase green energy usage without significantly increasing overall execution time over a traditional cloud scheduling system. Finally the resource utilisation aspect aims to ensure that all available compute resources are being utilised at any point in time, otherwise they should be quickly released so that Amazon can redistribute this computing power, minimizing idle time for resources. Having idle machines in the pool is inefficient from a cost and environmental stand point as a user would be paying for the privilege and the machines would be wasting energy.

5.1 Methodology

In order to evaluate the system, test workloads were submitted through the system to a HTCondor pool managed by the system. The same workloads were also be submitted to both a small size static HTCondor pool of Amazon EC2 instances, and a large static pool of Amazon EC2 instances, showing the advantages the system offers. The hypothesis is that the system offers significantly improved execution time over a small pool, improved resource utilisation and reduced monetary cost over a large pool and increases green energy utilisation over both static pools.

5.1.1 Condor Pools

The small static pool used consisted of 12 Virtual EC2 Instances. A selection of instance types from three different availability zones were used: EU West (Dublin), US East (North Virginia) and US West (California). These three availability zones were used constantly throughout the evaluation, however the system can use all availability zones. The selection of instance types used from each availability zone is shown in the table below. The selection of instance types only includes the standard types, and does not include specialised high power instance types such as ones with extra RAM or processors.

Instances Types	t1.micro	m1.small	m2.medium	m3.large
EU West	1	1	1	1
US East	1	1	1	1
US West	1	1	1	1

Table 5.1: *A table showing the selection of instances types for a small static HTCondor pool.*

This selection gives the small pool an acceptable spread of computing power over the availability zones selected without over specialising into one specific zone or instance type. It allows the test scenarios to complete within an acceptable time constraint.

The large static pool used consists of 48 Virtual EC2 Instances. Similar to the small pool which was described above it makes use of a selection of instance types from three availability zones. This pool was much larger, allowing for much more compute power, however it led to higher monetary and environmental costs. The selection can be seen below.

Instances Types	t1.micro	m1.small	m2.medium	m3.large
EU West	4	4	4	4
US East	4	4	4	4
US West	4	4	4	4

Table 5.2: *A table showing the selection of instances types for a large static HTCondor pool.*

This selection allows the test scenarios to complete considerably quicker than with the small static pool however at a much higher economic and carbon cost.

Finally the results obtained from these two pools are compared to the results of the test scenarios running through a HTCondor pool managed by the system. The system was constrained to only request instances/ spot instances in the three availability zones mentioned previously, and also only to launch instances of the types which were available to the static pools. The dynamic nature of this pool allows for an acceptable execution time while saving on monetary cost over the large static pool (due in part to the use of the spot instances), and executing significantly faster than the small static pool. And finally the dynamic nature of the Instance spawning should allow for an overall proportional carbon emission decrease over both static pools.

5.1.2 Workloads

This section will describe the different types of MPI(Message Passing Interface) Distributed Computing benchmarks which were used to evaluate the system. These benchmarks aim to simulate real world applications running on a cloud computing operating system. The benchmarks used are SuperLU, FFTW and NPB. A brief description of what each of these benchmarks do before describing the schedule of each test scenario will now be given.

SuperLU

SuperLU (Li (2005)) is a library for C (or Fortran) for the solution of sparse linear systems of equations of types AX = B, where A, X and B are large square matrices. It is available in three different types: Sequential, Multithreaded or Distributed. The sequential and multithreaded flavours are aimed at a single computer, but the distributed flavour is aimed at parallel processors, such as those in a HTCondor pool. We will be using the Distributed SuperLU flavour. This flavour uses MPI for interprocess communication which is ideal as support for MPI applications has been implemented in the system, and the static HTCondor pools also support MPI.

The application takes in two matrices A and B as files with double precision real numbers populating the matrix. The solution X, overwrites B in memory. The inputs A and B can either be replicated globally or distributed over the processors. In the implementation used, the inputs will be replicated globally as HTCondor easily allows transferring input files to every worker node in the pool using the *transfer_input_files=* parameter to a job submit file. There are also parameters to the SuperLU_Dist application which specify the amount of processors used and how the process group grid is arranged, i.e. *pddrive_ABglobal -r 2 -c 2* will create a process grid of 4 processors arranged in a 2x2 grid.

NASA Advanced Supercomputing Parallel Benchmarks (NPB)

NPB (NASA Advanced Supercomputing Division (2013)) is a suite of parallel benchmarks released by NASA to test the performance of parallel supercomputers. There are multiple different benchmarks within the suite, all derived from complex fluid dynamics applications. The benchmarks are also broken up into different classes of complexity. Class S and W are tiny sized problems, Classes A through C are standard size (where each letter corresponds to a 4x increase in size from one letter to the next) and Classes D through F are large tests (where each letter corresponds to a 16x increase in size). All of the benchmarks are available in either C or Fortran and use MPI or OpenMP to communicate between parallel processes.

The benchmark used is the IS (Integer Sort) kernel. This benchmark is available up to Class D size of complexity and is the only C only application. This benchmark performs a bucket sort algorithm on small integers. As the class size of the benchmark increases, so too does the number of keys and the maximum key value. As with SuperLU above, the number of processors that the benchmark should be run on needs to be known in advance, in this case the benchmark must be compiled with the parameters of *NPROCS=* (number of processes) and *CLASS=* (problem size).

FFTW

FFTW (Frigo & Johnson (2005)), "Fastest Fourier Transform in the West", is a C library for calculating the discrete Fourier Transform in arbitrary dimensions. The library uses the fast Fourier Transform algorithm in its computation. FFTW, like SuperLU, comes in multiple flavours: Serial, Multithreaded and Distributed. As with SuperLU we will use the Distributed flavour, which uses MPI for interprocess communication. The input data for a distributed FFTW benchmark does not need to be replicated globally, as each process only requires a slice of the input to operate on. However, as the simplest method of transferring input data is to transfer it all using the *transfer_input_files* parameter in a submit file, the data is replicated. Then on execution, when initializing the data, only the data that that process requires from the transferred file is initialized.

5.2 Scenarios

Now that the different benchmarks which made up the test workloads which were run through the three pools have been described, now the different scenarios which were tested on the system will be described. As discussed in previous chapters, there are three user definable inputs to the system: Priority, Importance of Carbon and Price. Each of these corresponds to one scenario in which that input was satisfied as much as possible. A fourth scenario where a mixture of these inputs was satisfied was also tested, which is a scenario more representative of every day use of the system.

What is meant by "satisfy these inputs"; for example, when trying to satisfy the priority input, the system will attempt to minimize the execution time by specifying "High" priority for each job. Similarly for when trying to satisfy the Carbon, carbon emissions will be minimized by specifying "High" carbon importance for each job in the workload, thus ensuring all instances are launched in what is considered the greenest data centre. To satisfy cost, jobs will be specified with low cost thresholds, and then instances should be launched in the availability zone with the cheapest spot instance price at that instant. Finally, in the mixed input scenario, work will be submitted with a mixture of High/Medium/Low priorities, High/Low carbon and High/Low cost thresholds. These scenarios will be compared against each other, and against the results gathered from the static pools.

The total workload used consists of a combination of large and small problem size versions of the benchmarks described previously. For SuperLU, the small problem size is an input file of a real matrix of 4096x4096 dimensions which came included in the SuperLU Examples folder. The large problem size is an input file of a real symmetric matrix Norris/fv1 (dimensions: 9604x9604) which was retrieved from the University of Florida Sparse Matrix Collection (Tim Davis and Yifan Hu (2011)). From now on these benchmarks will be referred to as SuperLU_Small and SuperLU_Large. For the NPB Integer Sort benchmark, the small problem size is a Class B size problem, which is 2^{25} keys and a maximum key value of 2^{21}. The large problem size is a Class D size problem, which is 2^{31} keys and a maximum key value of 2^{27}. These will be referred to as IS_Small and IS_Large from now on. For FFTW, the small problem size is an input file containing a 2D array of floating point numbers of 256*256 dimensions. The large problem size consists of a similar input file of 1024*1024 dimensions. These problem sizes are will be referred to as FFTW_Small and FFTW_Large.

Each workload will consist of thirty jobs altogether, ten of each different benchmark. Within this ten seven small size problems and three large sized problems will be submitted. The reason for this size of workload is to ensure that test takes a substantial amount of time. The allocation of jobs can be seen in the table below.

Problem Sizes	Small	Large
SuperLU	7	3
FFTW	7	3
NPB IS	7	3
Total	21	9

Table 5.3: *A table showing the allocation of problem sizes and benchmarks in a scenario workload.*

These jobs were then split up into five groups of six. All jobs in a group were submitted simultaneously, and all groups were submitted separately over the course of the test period at thirty minute intervals. The reason the groups were submitted separately was to firstly, more accurately represent real life usage of the system, and secondly to ensure that the test doesn't simply execute every job at the beginning of the test. The group allocations can be seen tabulated below. These allocations were created to try to balance the amount of small and large problem sizes in each group, and also to spread the different benchmarks over the test period.

Jobs	Small	Large
	1xSuperLU_Small	1xSuperLU_Large
Group 1	2xFFTW_Small	
	1xIS_Small	1xIS_Large
	2xSuperLU_Small	
Group 2	1xFFTW_Small	1xFFTW_Large
	1xIS_Small	1xIS_Large
	1xSuperLU_Small	1xSuperLU_Large
Group 3	1xFFTW_Small	1xFFTW_Large
	2xIS_Small	
	2xSuperLU_Small	
Group 4	1xFFTW_Small	1xFFTW_Large
	1xIS_Small	1xIS_Large
	1xSuperLU_Small	1xSuperLU_Large
Group 5	2xFFTW_Small	
	2xIS_Small	

Table 5.4: *A table showing the grouping of jobs.*

Submit Files

Lastly, an example of the resulting submit files which will be submitted is given below. In a Priority scenario all jobs will be given "High" Priority. An example of a submit file for an IS_Large High Priority Job can be seen below.

```
Executable      = mp2script

Universe        = Parallel

Arguments       = is.D.4

Log             = job8844654.log

Output          = job8844654.out

error           = job8844654.err

machine_count = 4

transfer_input_files = is.D.4

should_transfer_files = yes

when_to_transfer_output = on_exit

Requirements    = EC2InstType == "m1.large" && EC2JobID == "8844654"

+Priority = "H"

+Green_Priority = "-"

+Submitted = "17-04-2013T21:35:01"

Queue
```

Figure 5.1: *Sample Experiment submit file*

In this submit file one can clearly see the ”Priority” parameter is set to High, the ”Green Priority” is set to don’t care and the job name can be seen under the ”Arguments” parameter: IS (Integer sort) class D for 4 processors. In a green scenario the ”Priority” would be set to ”M” or ”L” signifying Medium or Low and the ”Green Priority” would be set to High. A submit file for a static HTCondor pool would look almost identical except without the added lines for ”Requirements”, ”Priority”,”Green Priority” and ”Submitted”.

5.3 Results

As mentioned in the introduction to this chapter the system will be evaluated under a number of different headings: Execution time, carbon efficiency, monetary cost and resource utilisation. Under each heading the results obtained for the four scenarios described previously will be compared and contrasted with each other and with the static HTCondor pools. From these results, conclusions will be extrapolated regarding the performance of the system. Note: The word ”workload” in this chapter refers to the total schedule of 30 jobs which are run through each system which constitutes a test.

5.3.1 Execution Time

The execution time taken for running the workload through each scenario is shown tabulated below.

	Total Execution Time
Small Static Pool	10 hrs 31 minutes
Large Static Pool	4 hrs 43 minutes
Time Priority Scenario	4 hrs 58 minutes
Green Priority Scenario	5 hrs 33 minutes
Low Cost Scenario	6 hrs 5 minutes
Mixed Scenario	5 hrs 21 minutes

Table 5.5: *A table showing the total execution time of each scenario.*

Here one can firstly see the clear decrease in execution time between the small and large static pools. Despite there being 4x as many instances available at any one time with the large static pool it is only a little better than twice as fast overall. This is attributed to overhead within the HTCondor matchmaking and scheduling system and the fact that at times there wasn't enough work to have all the machines in the large static pool active. As expected the Time Priority scenario executed fastest. The Low Cost scenario being the slowest of the dynamic pools is also unsurprising as in this scenario there is some overhead involved between the time when a bid for a spot instance is made, and the honouring of the bid and launching of the instance by Amazon. The Mixed scenario coming in around the average execution time for the dynamic pools is explained by the fact that it is made up of a mixture of priorities, pricing needs and green requirements thus taking in the strengths and weaknesses of the other scenarios and levelling out. Lastly the Green Priority scenario was, as expected slower than high priority scenario but faster than the cost scenario. The average execution time for the dynamic scenarios was 5 hours and 29 minutes. This is under an hour slower than the large expensive static pool and considerably faster than the small static pool, as per the hypothesis presented previously.

5.3.2 Resource Utilisation

In this section the comparison between the machine allocations over time in each of the scenarios with each other and with the static pools will be presented. Firstly a time series graph of the number of machines allocated to the pool over execution time for the Priority scenario can be seen in Figure 5.2, overlaying the machine's allocations for the two static pools.

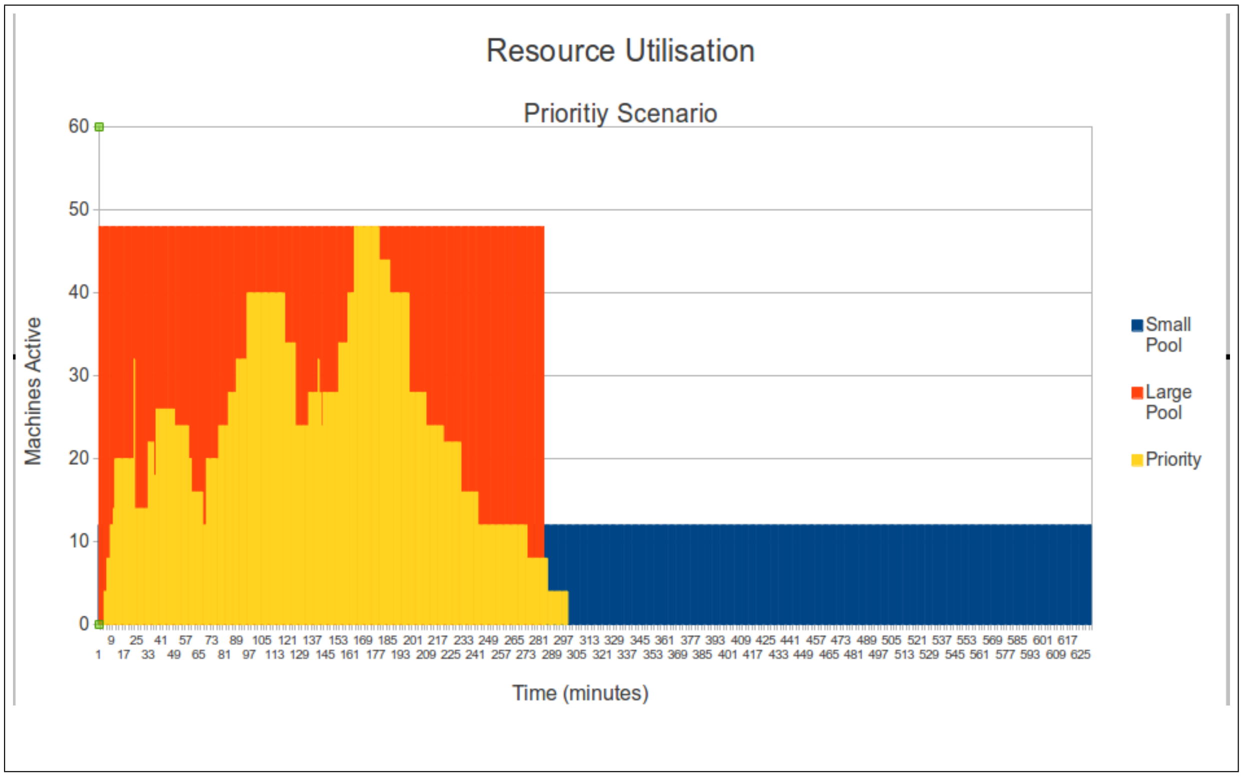

Figure 5.2: *Time Series Graph Depicting Resource Utilisation over a Time Priority Scenario*

As can be seen in this graph, the number of machines allocated to the pool over the course of the Priority scenario peaked at the same number of machines as were in the large static pool. This shows that the large static pool contained more machines than were required at most times of the workload, however at peak times 100% of the machines in the pool were active. The average number of machines allocated over the course of the Priority scenario was 24.5 meaning that on average a little over half the number of machines in the large pool were required to be allocated at once in order to complete the workload within 15 minutes of having 48 machines allocated the entire time. This shows a clear advantage of having a dynamic managed pool over a static pool when all jobs are required to be completed at a high priority.

The next two graphs (Figure 5.3 and Figure 5.4) presented show the resource utilisation of the dynamic pool overlayed on the static pools for the scenarios for Green and Cost.

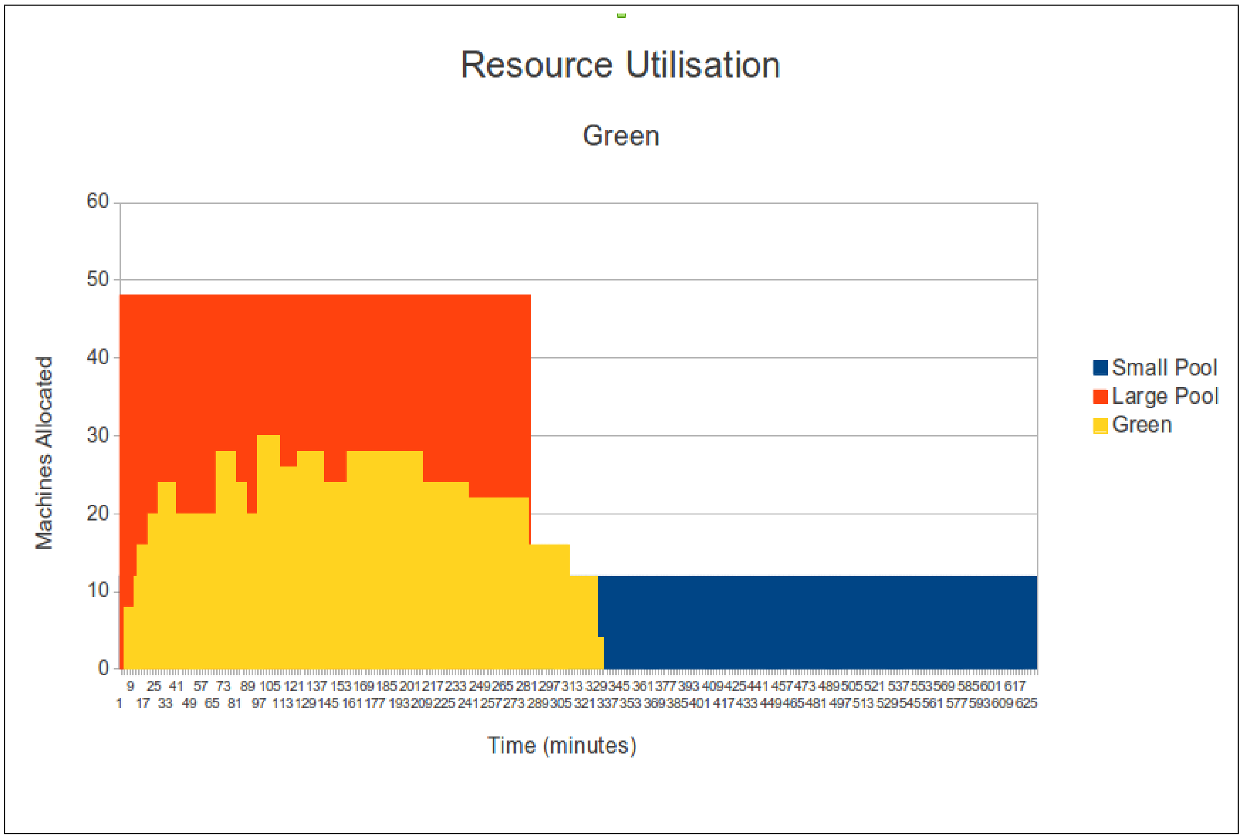

Figure 5.3: *Time Series Graph Depicting Resource Utilisation over a Green Priority Scenario*

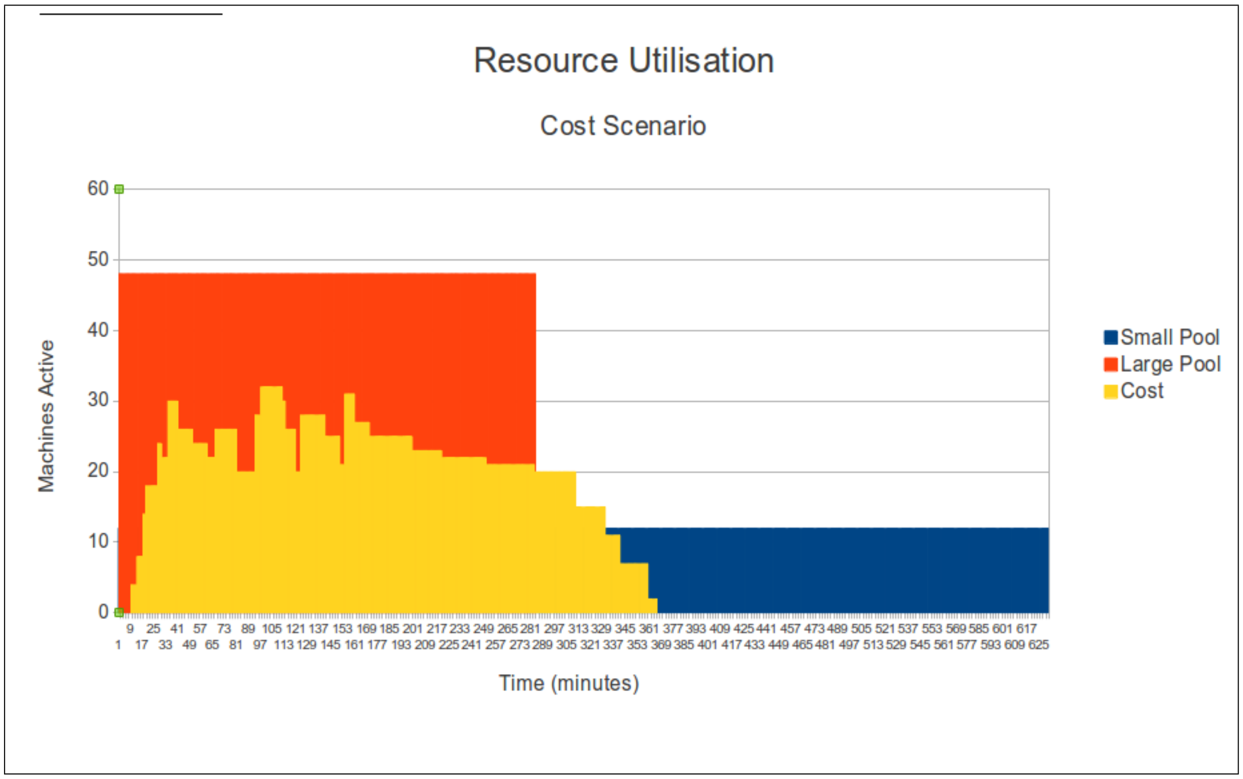

Figure 5.4: *Time Series Graph Depicting Resource Utilisation over a Low Cost Scenario*

These results are presented together as they consist of similar results. When compared with the static pools, the pools for both dynamic scenarios never reach the heights of the large static pool, while executing considerably quicker than the small static pool. More interesting is to compare these results with those of the Priority scenario presented in Figure 5.2. In the Priority scenario machines are allocated as needed with no waiting for unclaimed machines or low spot prices. This leads to the high peaks and low troughs in the graph indicating big allocations followed by big releases of machines. In the graphs for Green and Cost scenarios fewer and smaller peaks can be seen, as well as a distinct lack of troughs. Instead, a steady core of 20 machines is maintained throughout the scenario. This is attributable to the fact that when the priority input is not set to High, the system will always reuse an idle machine instead of requesting a new one. The average machines active in the Green and Cost scenarios are also lower at 20 and 21 machines respectively. Overall these results show that the execution time does not increase drastically when attempting to completely focus on minimizing Cost or Green impact over just attempting to complete the workload as fast as possible.

Figure 5.5 shows the graph of machine allocation in a Mixed scenario.

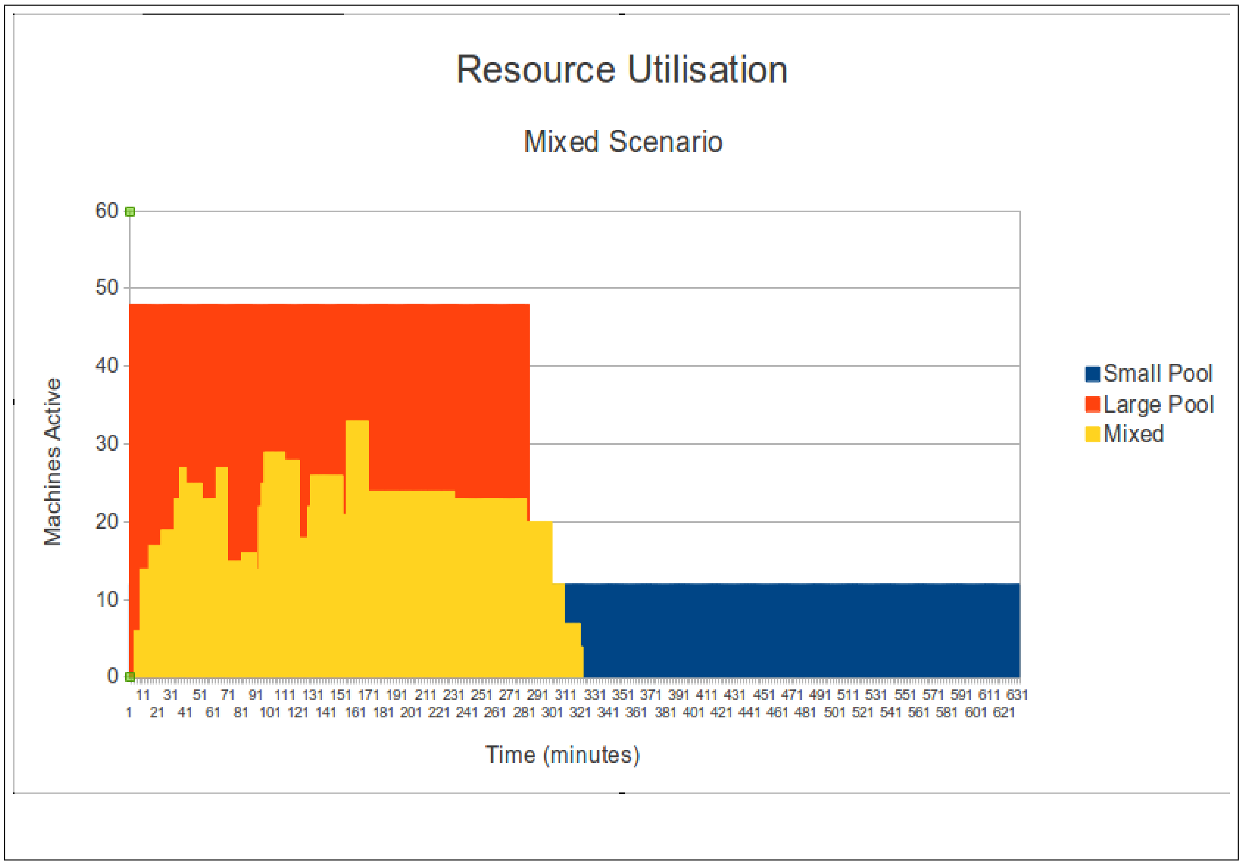

Figure 5.5: *Time Series Graph Depicting Resource Utilisation over a Mixed Scenario*

This scenario attempts to best emulate real world usage of the system, with jobs of mixed priorities, green priorities and costs submitted. As such one would expect this graph to be a mix of the previous graphs, and it is. High peaks and low troughs can be seen as machines are allocated and relinquished due to high priority jobs being submitted, while there remains a steady core of machines being used and reused by green and cost focused submissions. The average number of machines active in this scenario is 22. This is lower than in the Priority scenario while executing only 23 minutes slower. This seems like an adequate trade off in order to allow some jobs to be completed in a low cost or low carbon manner.

5.3.3 Carbon Emissions

Table 5.6 shows the Carbon emission results retrieved from the scenarios and static pools.

	Mean % Of Machines in Greenest Data Centre	Total Power Consumed (Watts)	g CO_2 Emitted
Small Static Pool	33%	29.7KW	13.5Kg
Large Static Pool	33%	54KW	24.5Kg
Time Priority Scenario	24%	32.8KW	16.3Kg
Green Priority Scenario	81%	28.6KW	11.1Kg
Low Cost Scenario	0%	29.1KW	15.9Kg
Mixed Scenario	57%	29.4KW	12.8Kg

Table 5.6: *A table showing the Carbon Results.*

The first column in Table 5.6 measures the average proportion of machines that were active in the Greenest data centre at any instant during the test. In the static pools, as the Greenest data centre can only be one of three and each one has an equal number of machines active in each throughout, the proportion is static at 33%. Within the Time Priority scenario, machines are provisioned in the nearest data centre from the submission, so the proportion of machines in the Greenest data centre is completely dependant on whether the nearest data centre to the point of submission is the Greenest data centre. In the case of the observed results, the nearest data centre to the point of submission was the EU West (Ireland) data centre. In the observed results the EU West data centre was the greenest data centre roughly a quarter of the time, which led to the resulting average of 24% of the Priority scenario machines being allocated in the greenest data centre. This also implies over the course of the test period, the greenest data centre changed.

In the Green Priority scenario, theoretically 100% of the machines would be allocated in the Greenest data centre, as at submission time, every job will attempt to allocate the necessary machines in the Greenest data centre. However in practice, there are a number of factors which prevent this. The first is that when the greenest data centre changes, all machines will not immediately shift to the new data centre. However any new machines allocated will be in the new greenest data centre. The second is that in the

Green scenario jobs will also always try to reuse an idle machine already in the pool. This would also prevent the pool from migrating to the new Greenest data centre by preventing new machines from being requisitioned. It is these factors which lead to the 81% observed average. However this average is still considerably higher than the static pools and other scenarios.

The Low Cost scenario shows the Environmental costs of attempting to complete the workload as cheaply as possible. The reason that 0% of the machines were active in the Greenest DC over the course of this job is that the cheapest spot instances were never available in what was at the time the Greenest DC. This significantly increased the CO_2 emitted over the other scenarios, except for the Time Priority scenario.

In the Mixed scenario, as expected, a combination of the strengths and weaknesses described in the Priority and Green scenarios is observed. All the Green submitted jobs in the scenario should follow the Green scenario and all the High priority jobs should follow the High Scenario. A 57% average seems like an acceptable trade off between the execution time of the High Priority scenario and the green emissions savings of the Green scenario.

The third column gives an estimate of the actual grams of Carbon emitted by each scenario over the course of the test workload. This estimate was calculated based on carbon intensity(g CO_2 per KWh) values for the locations of the data centres (retrieved from Eirgrid (2013) for Ireland and EPA (2007) for the US) and an estimate of the power consumption of the different virtual servers. The power consumption estimate is calculated by using an approximate power consumption for a single ECU (Elastic Compute Unit). An ECU has a direct correlation to the tier of instance that is being used. According to Amazon (2013a), a single ECU provides roughly the compute power of a 1.2Ghz Intel Xeon processor. An estimate of one hundred watts per ECU per hour is put forward. This estimate comes from a typical server having peak power of 213W given by Xiabo Fan, Wolf-Dietrich Weber and Luiz Andr Barroso (2007). Roughly 2 ECUs worth of virtual machines could be run on a server of this size. This means that a single m1.small instance will use 100W/hr, and an m2.medium will use 200W/hr. This estimate allows calculation of the total power consumed over the course of the workload for each scenario, which is shown in column 2 of Table 5.6. This estimate for power consumption per hour is not perfect as the hardware running the instances is not homogeneous and also the machines in the pool are virtualized which makes it difficult to pinpoint the exact amount of power consumed, however by applying the same estimate to each scenario it should suffice to give a clear picture.

Given the total amount of power consumed over time, and the carbon intensity for each data centre, a value for total grams of carbon emitted can be calculated. First the power consumed over time is reduced to power consumed per hour. This power consumed per hour is then divided down to a power consumed per hour per data centre. For each data centre the KW/H is then multiplied by the carbon intensity value at that location. Finally all the values are summed up to give a total carbon emitted value for each scenario. It can be seen that the large static pool performs by far the worst from a Green perspective. The small static pool performs far better in this respect as it requires only a quarter of the instances for just a little over twice the time leading to a near halving of the carbon emitted. The lowest carbon emissions result from, as hypothesised, the Green scenario. This scenario demonstrates a 54% reduction in CO_2 emissions versus the large static pool, and even a 17% reduction versus the small static pool, despite executing considerably faster. The Priority scenario offers an acceptable trade between reducing the carbon emissions from the large pool and a speed increase over the Green scenario. Finally, the Mixed scenario offers the most compelling trade by offering the second lowest carbon emissions, while offering an execution time which competes with the Priority scenario.

5.3.4 Cost

Table 5.7 shows the relevant results obtained for monetary cost from the scenarios.

	\$ Cost
Small Static Pool	\$ 11.17
Large Static Pool	\$ 27.90
Time Priority Scenario	\$ 20.80
Green Priority Scenario	\$ 16.25
Low Cost Scenario	\$ 1.65
Mixed Scenario	\$ 10.94

Table 5.7: *A table showing the Cost Results.*

As expected, the Large static pool was the most costly to maintain by a significant margin. Second to this is the Priority scenario. The reason this scenario is expensive is, as mentioned previously, machines are requisitioned as needed in order to speed up the execution. This means that money which could be saved by performed multiple jobs on the same machine is spent requesting new machines. This scenario does not make use of any spot instances which is another way in which money can be saved. The small static pool performs in the middle as regards to cost; the 50% saving over the Priority scenario being traded for over 50% longer execution time. The most important result in this table is the result for the Cost scenario. In this scenario all machines requested are spot instances, and are the cheapest spot instances available. This leads to price cuts of up 90% over on demand instances. This comes with multiple trade offs. Firstly the fact that they are spot instances means that they can be reallocated at will by Amazon, meaning they are less reliable than on demand instances. Secondly, the execution time is increased significantly as can be seen in Section 5.4.1. However this result shows the significant savings that can be achieved by focusing on solely reducing monetary cost. Lastly the Mixed scenario demonstrates, again, some of the advantages of the Cost scenario. In this case a 47% saving over the Priority scenario is observed, however this scenario remains 84% more expensive than focusing on Cost alone by merit of having a selection of High Priority and High Green Priority jobs in the mix. Overall however, the savings offered by the Mixed scenario demonstrate that allowing for some unreliability and increased execution time can result in extremely significant monetary savings.

Chapter 6

Future Work and Conclusion

This section will detail any future work which could be performed regarding the system implemented and the conclusions which were obtained from the research undertaking.

6.1 Future Work

Due to the prototype nature of the system implemented there is a lot of potential future work involved in extending and expanding the system. The system could be extended to work with other cloud infrastructure services such as Rackspace or Rocks Clusters, as it stands now it only functions with Amazon EC2. The system only supports the MPI distributed computing standard, however support for other frameworks such as OpenMP or Hadoop could be implemented. Lastly the system only supports Linux-based Operating Systems, support for Windows could be implemented, especially considering EC2 instances can use Windows and HTCondor has a windows version. The selection of monetary and carbon cost as the scheduling concerns to focus on could be extended to include data locality given the location-based solution implemented.

6.2 Conclusion

This dissertation focused on studying different methods of scheduling in the cloud. The solution selected as the best was to dynamically allocate virtual machines in different data centres depending on user preferences regarding cost and carbon priority. The inclusion of user preferences allows for a more flexible scheduler as the user can choose what the scheduler should focus on. A series of test cases were carried out on the system which showed that in the case of each user decision the scheduler would demonstrate saving appropriate to that selection. The selection to perform geography based allocation of virtual machines means that the solution is open ended in terms of extending to other scheduling concerns such as data locality or latency. The system implemented appears to satisfy the aim of the dissertation; to design and implement a meaningful and worthwhile contribution to the area of cloud computing scheduling.

References

Amazon. (2013a). *Amazon EC2 Instance Types.* Available from `http://aws.amazon.com/ec2/instance-types/`

Amazon. (2013b). *Amazon EC2 spot instances.* Available from `http://aws.amazon.com/ec2/spot-instances/`

Amazon. (2013). *Elastic Cloud Compute.* Available from `http://aws.amazon.com/ec2`

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., et al. (2010). A view of cloud computing. , *53*(4), 50-58. Communications of ACM.

Ben-Yahuda, O. A., Ben-Yahuda, M., Schuster, A., & Tsafrir, D. (2011, November). Deconstructing Amazon EC2 Spot Instance Pricing. , 304 - 311. Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference.

boto. (2013). *boto: A Python Interface to Amazon Web Services.* Available from `http://boto.readthedocs.org/en/latest/`

DNAnexus. (2013). *Dnanexus.* Available from `http://dnanexus.com`

Doyle, J., O'Mahony, D., & Shorten, R. (2011, August). *Server Selection for Carbon Emission Control.* ACM SIGCOMM Workshop on Green Networking.

Dropbox. (2013). *Dropbox.* Available from `www.dropbox.com`

Eirgrid. (2013). *Eirgrid Real Time CO2 Intensity.* Available from `http://eirgrid.com/operations/systemperformancedata/co2intensity/`

EPA. (2007). *eGRID2006 Version 2.1 (April 2007).* Available from `http://epa.gov/cleanenergy/documents/egridzips/eGRID2006V2_1_Summary_Tables.pdf`

Frigo, M., & Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, *93*(2), 216–231. (Special issue on "Program Generation, Optimization, and Platform Adaptation")

Goiri, I., Le, K., Nguyen, T. D., Guitart, J., Torres, J., & Bianchini, R. (2012, April). GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks. *Proceeding of the 7th ACM European Conference on Computer Systems*, 57-70.

Google. (2013a). *Gmail.* Available from `https://mail.google.com`

Google. (2013b). *Google App Engine.* Available from `https://developers.google.com/appengine/`

GoSquared. (2013). *Gosquared.* Available from `http://www.gosquared.com`

Hatzoupoulos, D., Koutsopoulos, I., Koutitas, G., & Heddeghem, W. V. (2013). Dynamic Virtual Machine Allocation in Cloud Server Facility Systems with Renewable Energy Sources.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., et al. (2010). *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Centre* (Tech. Rep.). University of California, Berkeley.

Jin, J., Luo, J., Song, A., Dong, F., & Xiong, R. (2011, May). BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing. , 295 - 304. In Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on.

Juve, G., Deelman, E., Vahi, K., Mehta, G., Berriman, B., Berman, B. P., et al. (2009, December). Scientific Workflow Applications on Amazon EC2. , 59 - 66. E-Science Workshops, 2009 5th IEEE International Conference on.

Li, X. S. (2005, September). An overview of SuperLU: Algorithms, implementation, and user interface. , *31*(3), 302-325.

Malik, S., Huet, F., & Caromel, D. (2012). Latency based dynamic grouping aware cloud scheduling. In *Advanced information networking and applications workshops (waina), 2012 26th international conference on* (pp. 1190–1195).

Mell, P., & Grance, T. (2011). The NIST definition of cloud computing (draft). NIST Special Publication.

Microsoft. (2013a). *Microsoft Office 365.* Available from `http://office.microsoft.com`

Microsoft. (2013b). *Windows Azure.* Available from `http://www.windowsazure.com/en-us/`

NASA Advanced Supercomputing Division. (2013). *NAS Parallel Benchmarks.* Available from `http://www.nas.nasa.gov/publications/npb.html`

Papadopoulos, P. (2011). Extending clusters to Amazon EC2 using the Rocks toolkit. *International Journal of High Performance Computing Applications*, *25*(3), 317–327.

Rackspace. (2013). *rackspace, the open cloud company.* Available from `www.rackspace.com`

SaltedServices. (2013). *Litmus.* Available from `http://litmus.com`

Sharma, N., Gummeson, J., Irwin, D., & Shenoy, P. (2010, June). Cloudy Computing: Leveraging Weather Forecasts in Energy Harvesting Sensor Systems. *SECON*.

Tannenbaum, T., Wright, D., Miller, K., & Livny, M. (2001, October). Condor – a distributed job scheduler. In T. Sterling (Ed.), *Beowulf cluster computing with Linux.* MIT Press.

Tim Davis and Yifan Hu. (2011). The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Sofrware*, *38*(1), 1–25. Available from `http://www.cise.ufl.edu/research/sparse/matrices`

Windpower-Program. (2013). *Wind turbine output variation with steady wind speed.* Available from `www.wind-power-program.com/turbine_characteristics.htm`

Xiabo Fan, Wolf-Dietrich Weber and Luiz Andr Barroso. (2007, June). Power Provisioning for a Warehouse Sized Computer. *Proceedings of the ACM International Symposium on Computer Architecture*.

Youssef, L., Butrico, M., & Silva, D. D. (2008). *Toward a Unified Ontology of Cloud Computing* (Tech. Rep.). University of California, Santa Barbara.

Appendix - Abbreviations

AMI - Amazon Machine Image

AWS - Amazon Web Services

CO_2 - Carbon Dioxide

DC - Data centre

EC2 - Elastic Cloud Compute

ECU - Elastic Computer Unit

FFTW - Fastest Fourier Transform in the West

IaaS - Infrastructure as a Service

MPI - Message Passing Interface

NPB - NASA Advanced Supercomputing Parallel Benchmarks

OS - Operating System

PaaS - Platform as a Service

SaaS - Software as a Service

SSH - Secure Shell

VM - Virtual Machine