ADAPTIVE CO-SCHEDULER FOR HIGHLY DYNAMIC RESOURCES

by

Kartik Vedalaveni

A THESIS

Presented to the Faculty of

The Graduate College at the University Of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master Of Science

Major: Computer Science

Under the Supervision of Professor David Swanson

Lincoln, Nebraska

August, 2013

ADAPTIVE CO-SCHEDULER FOR HIGHLY DYNAMIC RESOURCES

Kartik Vedalaveni, M.S

University Of Nebraska, 2013

Adviser: David Swanson

There are many kinds of scientific applications that run on high throughput computational (HTC) grids. HTC may utilize clusters opportunistically, only running on a given cluster when it is otherwise idle. These widely dispersed HTC clusters are heterogeneous in terms of capability and availability, but can provide significant computing power in aggregate. The scientific algorithms run on them also vary greatly. Some scientific algorithms might use high rates of disk I/O and some might need large amounts of RAM. Most schedulers only consider cpu availability, but unchecked demand on these associated resources that aren't managed by resource managers may give rise to several issues on the cluster.

On the grid there could be different schedulers on different sites and we cannot rely upon features of one kind of scheduler to characterize the nature and features of every job. Most state of the art schedulers do not take into account resources like RAM, Disk I/O or Network I/O. This is as true for the local schedulers as much it is for the grid. Often there is a need to extend these schedulers to solve situations arising from new and/or complex use cases either by writing a plugin for existing schedulers or by CoScheduling. A key issue is when resources like RAM, Disk I/O or Network I/O are used in an unchecked manner and performance degrades as a result of it. Further scheduling jobs that claim the degraded resources could overwhelm the resource to an extent that the resource will finally stop responding or the system will crash.

With an increase in the number of entities concurrently using the resource, there is a

need to monitor and schedule concurrent and unmanaged access to any given resource to prevent degradation. These issues that we encounter in real life at the Holland Computing Center are the basis and motivation for tackling this problem and for developing an adaptive approach for scheduling. This co-scheduler must be aware of multi-resource degradation, balance load across multiple sites and run clusters at high efficiency and share resources fluidly. An initial implementation tested at HCC will be evaluated and presented.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Dr. David Swanson. Throughout my research and writing he's been a constant source of support for me. His guidance and teachings have helped not only to shape my thesis but also in all walks of my life. I thank David for identifying this interesting research problem and bringing it to me. His hard work has been a constant source of inspiration. I would like to thank Derek Weitzel for working with me to solve the issues with grid submissions and for helping me with his technical expertise.

I'd like to take this opportunity to thank the entire team at Holland Computing Center for providing me with the required resources, support and continuous advise even after I broke their systems. It has been a wonderful experience working with these folks.

Contents

List of Figures

Chapter 1

Introduction

Ian Foster states in his paper The Anatomy Of the Grid [6] that *Grid computing is about controlled sharing of resources with resource owners enforcing policies on the owned resources.* The resources come in the form of hardware and software that allows us to submit jobs, run the jobs and monitor the jobs on the grid. Universities usually have multiple clusters across their campus and these are usually owned by different departments but stand united under the banner of the university. Campus grids are mini grids where jobs are spanned across multiple clusters based on the need of the user and available resources. There is a provision for jobs to overflow to the national grid infrastructure[17].

Modern schedulers used in clusters provide numerable features for policy making, resource management and scheduling. The problem of cluster performance degradation that occurs when one of the resources is overwhelmed is a problem that hasn't been addressed. This problem occurs when many jobs are scheduled on a single system. Some of the schedulers like maui [8] are sophisticated enough to take into account contention of other resources like RAM but ultimately convert the 2D vector values of CPU and RAM into a single scalar value. In-practice this hard-codes the value or presents these resources in a fixed ratio which makes us question effectiveness of such scheduling mechanisms.

At Holland Computing Center of the University of Nebraska-Lincoln, we're tackling this issue of cluster degradation caused by over exploitation of one or more resources. The proposed solution adaptively schedules such highly dynamic resources on the grid and across multiple sites by adaptively scaling with respect to the performance of a given cluster. We schedule based on minimum turnaround time of the sites which will help increase the throughput of the overall workload for the Co-Scheduler, which also is a good load-balancer in itself.

Existing schedulers depend on the availability of resources and frequent polling of it via resource manager to determine the slots for scheduling. It should be noted that state of the art schedulers like maui/torque, slurm, HTCondor take into account only CPU as a resource and the resources like RAM, Disk I/O or Network I/O are either ignored or their resource equivalent is converted into a scalar value which isn't an effective way of tackling the multiple resource scheduling problem. This might result in scheduling excessive jobs on a single machine. In some schedulers, for example it'll be pre-assigned on every node that 1 CPU-CORE will have 2 GB of memory. This kind of pre-assignment seems like it's addressing the resource degradation of RAM in addition to CPU but suffers from the above mentioned problems. We take a turnaround time approach to measure degradation. We define a resource to be degraded if and when we submit jobs that results in increasing the turnaround time by 25% . It must be noted that there isn't explicit measurement of status of individual resources like RAM, Disk I/O, Network I/O, since there are no resource managers keeping track of these resource allocations. The Co-Scheduler is driven by the fact that turnaround time increases when concurrent jobs accessing these resources reach a threshold value which in-turn causes degradation and this is the basis for this work.

One aspect to Co-Scheduler is degradation detection and management by adapting to the throttling of a resource. Another aspect is to efficiently distribute jobs across

multiple sites which increases the throughput of the workload. The Co-Scheduler adapts to degradation by backing off submission rates and waiting for a period of time (till the detection of a target capacity) for the next incremental submission. It efficiently distributes the jobs and load-balances across multiple sites, thus submitting more jobs to a site with lesser turnaround time and also ensuring to submit lesser jobs to the sites with larger turnaround time. This efficiently load-balances across multiple sites on a grid and improves throughput because the loads may vary dynamically. The capacity of jobs the cluster can efficiently run without degradation needs to be detected and updated again and again after a given period of time.

Finally, since the grid environment is a heterogeneous environment, it's absolutely necessary to use APIs that are available on all the systems across the grid. To implement such a Co-Scheduler we limit ourselves to HTCondor based clusters and utilize libcondorapi. The result is a threaded Co-Scheduler that submits to multiple sites concurrently and is aware of multiple-resource degradation and load-balances efficiently across multiple sites of the grid.

Please note that all the references to the Co-Scheduler in this thesis refers to the adaptive Co-Scheduler designed by us.

Chapter 2

Background

2.1 High-Throughput Computing

High Throughput Computing (HTC) is defined as a computing environment that delivers large amounts of computational power over a long period of time. The important factor being over a long period of time which differentiates HTC from HPC which focuses on getting a large amount of work done in a small amount of time. The workloads that run on HTCondor systems don't have an objective of how fast the job can be completed but how many times can the job be run in the next few months. In another definition of HTC, European Grid Infrastructure defines HTC as a computing paradigm that focuses on the efficient execution of a large number of loosely coupled tasks [1]. One such distributed computing software providing High-Throughput Computing services is HTCondor developed by the team at University of Wisconsin-Madison.

2.2 HTCondor

HTCondor is a distributed system developed by HTCondor team at the University of Wisconsin-Madison. It provides an HTC environment to sites that foster research computing and enables sites to share computing resources on otherwise idle computers at a given site. HTCondor includes a batch queuing system, scheduling policy, priority scheme, and resource classifications for a pool of computers, which is mainly used for compute-intensive jobs. HTCondor runs on both UNIX and Windows based workstations that are all connected by a network, although there are other batch schedulers out there for dedicated machines. The power of HTCondor comes from the fact that the amount of compute power represented by the sum total of all the non-dedicated desktop workstations sitting on people's desks is sometimes far greater than the compute power of a dedicated central resource. There are many unique tools and capabilities in HTCondor which make utilizing resources from non-dedicated systems effective. These capabilities include process checkpoint and migration, remote system calls and ClassAds. HTCondor also includes a powerful resource manager with an efficient match-making mechanism that is implemented via ClassAds, which makes HTCondor understandable when compared with other compute schedulers[1]. With numerous features of HTCondor, it is used for grid computing on the scale of large number of massive computers that are loosely coupled on the open science grid. We architected a bioinformatics program iSG, indel sequence generator which simulates the evolutionary events of highly diverged DNA and protein sequences. iSG is a high memory footprint program, we architected it so as to make it run on the grid and use less amount of peak memory [9]. We retained the order of execution of individual jobs by making extensive use of HTCondor's DAGMAN.

2.3 Grid Computing

As the cost of computers and the network connecting them is decreasing, there is movement towards a paradigm shift in computing with the clustering of geographically distributed resources. The goal of this is to provide a service oriented infrastructure. Organizations like the Grid community and Global Grid Forum are constantly investing effort in making Grid a platform with standard protocols, to provide seamless and secure discovery and access to infrastructure and interactions among the resources and services.

With the advent of parallel programming and distributed systems it became obvious that loosely coupled computers can be used for computing purposes and this network of workstations gave rise to the notion of distributed computing. The Globus project began in 1996 and Argonne National Laboratory was responsible for a process and middleware communication system called Nexus which provides remote service requests across heterogeneous machines. The goal of Globus was to build a global Nexus that would provide support for resource discovery, data access, authentication and authorization.

At this point "Grid" replaced the use of "meta-computer" and *researchers from a collaboration of universities termed Grid as integrated resources with integration of many computational visualization and information resources into a coherent infrastructure.*

The following are goals and visions of Grid Computing:

Seamless Aggregation of Resources and Services Aggregation involves three aspects,

1. Aggregation of geographically distributed resources
2. Aggregation of capacity
3. Aggregation of capability

The key factors to enable such aggregation includes protocols and mechanisms to secure discovery, access to and aggregation of resources for the realization of virtual

organization and applications that can exploit such an environment.

Ubiquitous Service-Oriented Architecture This is the ability of the grid environment to do secure and scalable resource and data discovery along with scheduling and management based on a wide variety of application domains and styles of computing.

Autonomic Behaviors The dynamism, heterogeneity and complexity of the grid has made many researchers rethink their systems. This new trend aims at system configuration and maintenance of the grid with the least human effort and has led to numerous projects like autonomic grids, cognitive grids and semantic grids.

A key computational grid in the USA is the Open Science Grid, described further in section 2.5. At the heart of the grid computing tools provided by the open science grid lies the globus toolkit, that provides features and interfaces that enable grid computing.

2.4 Globus

The Globus project was intended to accelerate the then meta-computing that was built on distributed and parallel software technologies. The term meta-computing was used before the term grid was actually coined which refers to a networked virtual supercomputer, constructed dynamically from geographically distributed resources linked by high speed networks [5]. Scheduling and managing a large group of heterogeneous resources is a challenging and daunting task on the grid. The Globus toolkit provides a framework for managing and scheduling of grid resources across heterogeneous environments.

The Globus toolkit consists of a set of modules, each module provides an interface for the provision of an implementation of low level functionalities of the toolkit:

- Resource location and allocation

- Communications
- Unified resource information service
- Authentication interface
- Process creation
- Data Access

There are multiple Virtual Organizations across Open Science Grid. One such VO is Nebraska's HCC VO at Holland Computing Center. A VO provides researchers with computing resources and enables sharing of resources across other VO's. It additionally it signs the user's certificate so that a user is identified with a given VO.

We delegate our identification on the grid. *voms-proxy-init* is used to generate a voms proxy at the submit host, we can specify voms as our virtual organization, hcc:/hcc in this case, and hours as the number of hours the proxy would be active/valid. It must be ensured that the proxy period is approximately greater than the length of period of runtime of jobs for successful running of all the jobs. A proxy provides a secure way to access grid resources, by creating a proxy and delegating our identity to it for a short period of time it enables us to sandbox the theft of the proxy credentials and limit the damage to the short lived proxy and keep our original set of credentials safe.

HTCondor-G extends HTCondor to the grid environment. Jobs are sent across the grid universe using Globus software. The Globus toolkit provides support for building grid systems but submitting, managing and executing jobs have the same capabilities in both the HTCondor and the HTCondor-G world. Globus provides fault tolerant features to HTCondor-G jobs. GRAM is the Grid Resource Allocation and Management protocol, which supports remote submission of computational request.

gt2 is an initial GRAM protocol which is used in Globus Toolkit version 1 and 2. gt2 is also referred to as the pre-web services GRAM or GRAM2. Similarly, gt5 is the latest GRAM protocol, which is an extension of GRAM2 and is intended to be more scalable and robust, referred to as GRAM5.

2.5 Open Science Grid

The Open Science Grid(OSG), provides service and support for resource providers and scientific institutions using a distributed fabric of high throughout computational services. OSG was created to facilitate data analysis from the Large Hadron Collider[13]. OSG doesn't own resources but provides software and services to users and enables opportunistic usage and sharing of resources among resource providers. The main goal of OSG is to advance science through open distributed computing. The OSG provides a multi-disciplinary partnership to federate local, regional, community and national cyber-infrastructures to meet the needs of research and academic communities at all scales.

OSG provides resources and directions to Virtual Organizations(VO's) for the purposes of LHC experiments and HTC in general.

Building an OSG site requires analysis of the requirements and careful planning. The major components of an OSG site include a Storage Element and a Compute Element. Storage elements (SE) manage physical systems, disk caches and hierarchical mass storage systems. A SE is an interface for grid jobs to the Storage Resource Management protocol (SRM) and the Globus GridFTP protocol and others. A storage element requires an underlying storage system like hadoop, xrootd and a GridFTP server and an SRM interface.

A Compute Element(CE) allows grid users to run jobs on your site. It provides a large number of services when run on the gatekeeper. The basic components include the GRAM and GridFTP on the same CE host to successfully enable file transfer mechanisms of HTCondor-G.

Chapter 3

Related Work

3.1 Comparison of existing mechanisms

There are situations where we can have multiple HTCondor pools exist and some of the pools have many idle slots that are available for utilization. To efficiently utilize the resources across pools condor provides mechanisms like HTCondor flocking and HTCondor job router. These mechanisms provide similar functionality to the Co-Scheduler designed here. In the following sections, we compare and evaluate these mechanisms with the design and functionality of Co-Scheduler.

3.1.1 HTCondor Flocking

Flocking refers to a mechanism where a job that cannot run on its own HTCondor pool due to a lack of resources runs in another HTCondor pool where resources are available. HTCondor flocking enables load sharing between pools of computers. As pointed out by Campus Grids Thesis[17] flocking helps balance large workflows across different pools because of the scavenging and greedy nature of the HTCondor scheduler.

To accomplish flocking HTCondor uses multiple components. `condor_schedd` adver-

tises that it has idle jobs to the remote `condor_collector`. During the next phase of negotiation, if it's found that there are computers available then they are allotted to the jobs in the matchmaking phase and the jobs are then run on the remote pools. It so appears, and the local job queue's maintained as if the jobs are running locally.

Although HTCondor flocking has workflow balancing features across multiple HTCondor pools, it isn't aware of the slower and faster sites/pools. The adaptive Co-Scheduler keeps tabs on turnaround time and is aware of which site is faster or slower. If we look at the idea that scavenging idle resources increases throughput, it does. However, if we look at the case where jobs are greater than the available slots across all the pools, HTCondor flocking stops working here as deeper understanding of sites would be required here to push more jobs at faster sites and importantly push less jobs at slower sites. Flocking's role is to scavenge idle computing slots across multiple pools.

Another aspect where HTCondor flocking isn't designed to perform well is when jobs are contending for the same resource (CPU, RAM, Disk I/O & Network I/O). Even though we'll be having idle slots on remote pools, HTCondor flocking might not be able to use those slots as they'd be degraded because of the contention. In this case again Co-Scheduler comes in handy. Based on the turnaround time, Co-Scheduler waits for a random amount of time for degradation to clear and doesn't put excessive job load on the degraded resource. It diverts the jobs elsewhere to perhaps another site, which in-turn increases the throughput.

To conclude, we can say that HTCondor flocking provides features of balancing large workflows and doesn't include features that would detect degradation in the cluster. It also doesn't keep track of information of site performance, which might be exploited to increase the overall throughput of the system.

3.1.2 HTCondor Job Router

The HTCondor manual defines the functions of job router to be the following:

> The HTCondor Job Router is an add-on to the `condor_schedd` that transforms jobs from one type into another according to a configurable policy[1]. This process of transforming the jobs is called job routing.

HTCondor Job Router can transform vanilla universe jobs to grid universe jobs and as it submits to multiple sites, the rate at which it starts submitting equals the rate at which the sites execute them. This provides a platform to balance large workflows across multiple grid sites and replenish the jobs at faster sites once they get done. Job router sends more jobs to a site if the jobs submitted are not idle and stops submitting jobs if the submitted jobs sit idle on the remote cluster. Job router is not aware about which site is faster of slower.

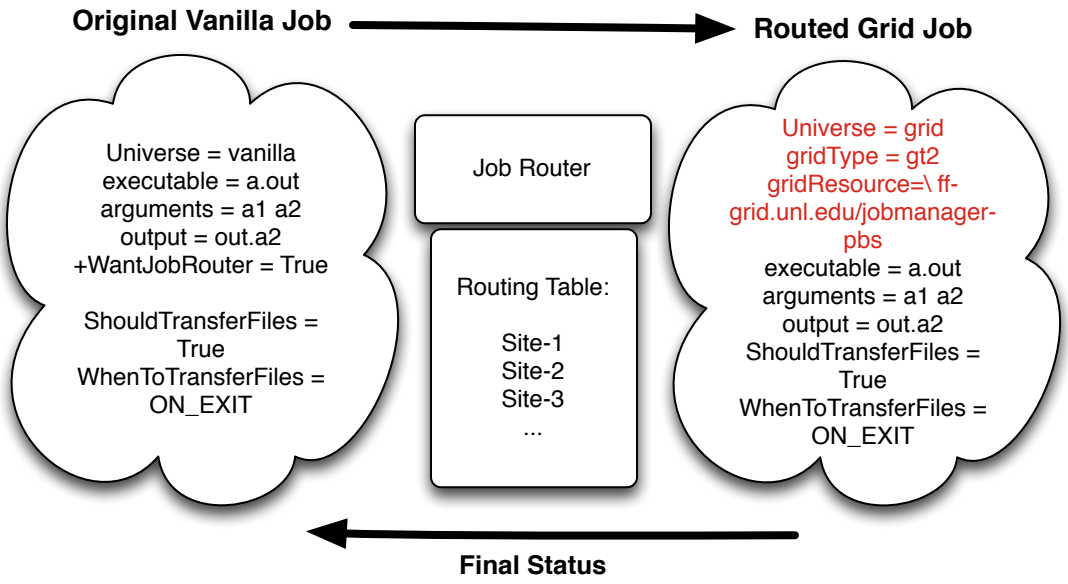

Figure 3.1: JobRouter: Transformation of Jobs

```
  # Now we define each of the routes to send jobs on
JOB_ROUTER_ENTRIES = \
   [ GridResource = "gt5 ff-grid.unl.edu/jobmanager-pbs"; \
     name = "Firefly"; \
   ] \
   [ GridResource = "gt5 tusker-gw1.unl.edu/jobmanager-pbs"; \
     name = "Tusker"; \
   ] \
   [ GridResource = "gt5 pf-grid.unl.edu/jobmanager-condor"; \
     name = "Prairiefire"; \
   ]\
```

Figure 3.2: Configuration to implement Job Routing

A job is transformed to the grid universe by making a copy of the original job ClassAd and modifying some attributes of the job. This copy is called the routed copy and this routed copy shows up in the job queue with a new job id[1].

HTCondor job router utilizes a routing table which contains the listing of sites the job must be submitted to and the name of the potential grid resources. Routing is processed via a HTCondor config file which is defined by the new ClassAds.

HTCondor Job Router appears to be a step up from HTCondor Flocking in terms of scavenging resources and sending the extra jobs to another HTCondor pool. HTCondor Job Router also maintains the submission rate of the jobs on submit hosts equal to that of remote clusters. But the job router does not keep track of how fast each HTCondor cluster is. Thus HTCondor job router does not optimize the displacement of jobs from a slower cluster. Since it maintains the rate of submission of jobs, we can be sure that if a job is completed at a faster site, it's immediately replaced by the next one. The same is true for a job at a slower site. This might result in increasing the degradation if there exists some. For example, if the job router kept track of slower sites, it can send less jobs to slower sites and thereby increase the overall throughput of the given workflow.

A preliminary examination shows that HTCondor job router does seem to have degradation detection features, since it does not submit to a pool that already has idle jobs. However, close examination reveals that even though it submits to a pool with non-idle jobs, it isn't aware of the fact that these jobs in the queue would have undergone degradation due to contention on same resource and thus isn't a viable mechanism for degradation detection.

3.1.3 OSG Match Maker

OSG Match maker is a tool that was created to be distributed among small to medium sized VOs to be used as a powerful submit interface to OSG. It is designed to retrieve VO specific site information and also to verify and maintain the sites by regularly submitting verification jobs to make sure a site can continue receiving jobs. OSG Match maker is another tool that provides information about the grid sites to the HTCondor scheduler. HTCondor scheduler does all the matchmaking with the ClassAds attributes. OSG Match Maker is no longer supported on OSG.

Chapter 4

Adaptive Co-Scheduler For Highly Dynamic Resources

4.1 Introduction

Due to the heterogeneous nature of the grid there could be problems with resource sharing on the grid. We propose a solution that primarily solves the problem of degradation due to contention among jobs for any given resource like RAM, I/O and the network. The contention among jobs sometimes give rise to degradation depending upon the availability of the resource that results in reduced performance of the cluster or crashes the cluster altogether. Efficient and high throughput distribution of jobs across multiple cluster is another challenging problem that we have solved in this particular Co-Scheduler Solution.

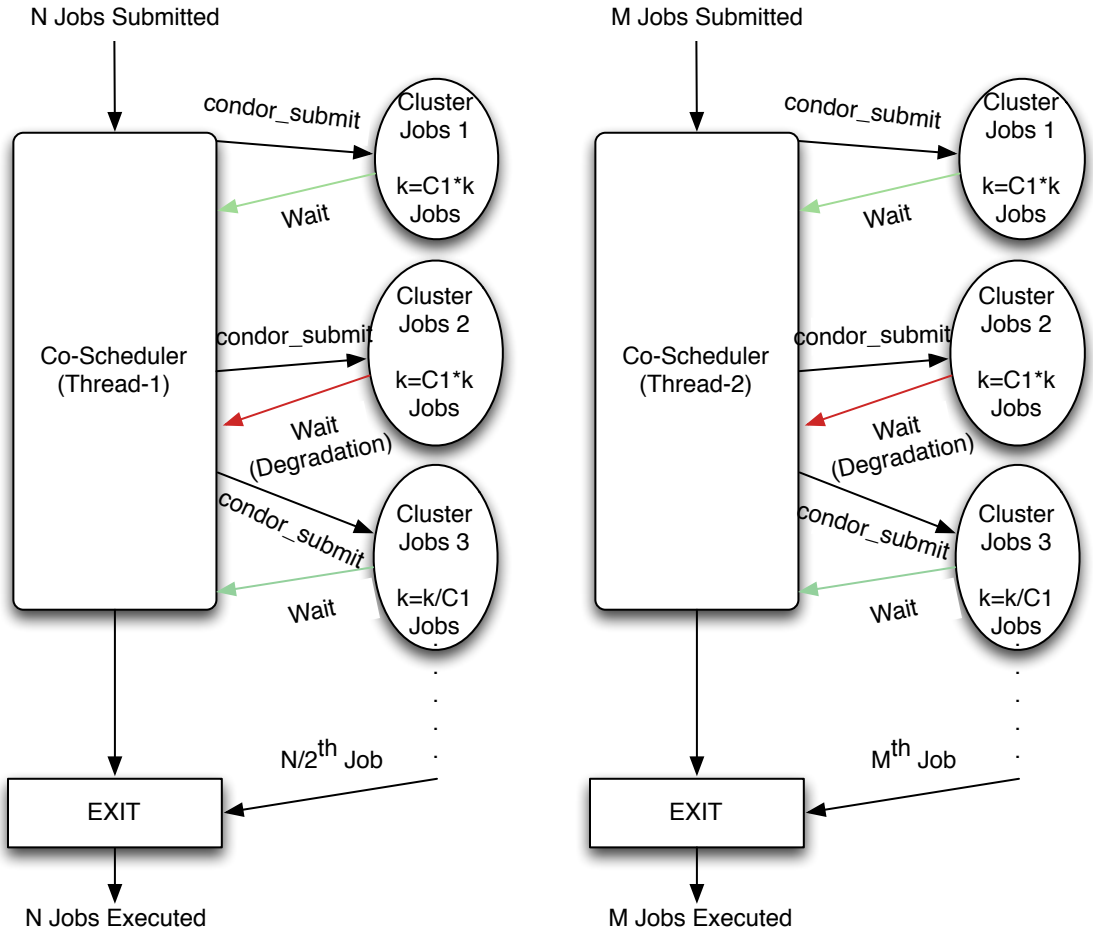

Figure 4.1: Degradation occurrence: Co-Scheduling

In figure 4.1, with C1 called job propagation factor and k the initial number of jobs submitted we can see that on each iteration we're adding $k = C1 * k$ jobs, and on the next iteration, if we have a degraded system, then we submit $k = k/C1$ amount of jobs by backing off on the amount of jobs submitted.

4.2 Degradation detection and Capacity based scheduling

In the first part of the solution we target the basic problem of degradation. To give an example, suppose we have a file server that can serve 100 MBps of data which is available for concurrent access and a user has submitted 200 jobs each consuming 10 MBps of data I/O. If 100 CPU slots are available, a typical state of the art Scheduler schedules these 200 jobs without actually looking at the I/O load. This results in a degraded system, over time if more jobs are Scheduled to utilize this file server. This file serve might even crash. We would need a degradation handling mechanism that would adaptively scale with the load and exponentially back off during a high contention period. Thus we need an intervention in the form of a Co-Scheduler that intelligently handles degradation. We start the capacity algorithm by submitting jobs to the cluster and measure the turnaround time of each iteration. If we find that the current turnaround time of an iteration is 25% greater than the previous iteration we term it as a degradation. We now try to find the best capacity of jobs between the current and the previous iteration by exponentially backing off between these two iteration. We keep finding the optimal capacity dynamically for each iteration as the contention may change and optimal capacity keeps varying. Once the optimal capacity is found then it is retained for job submissions for a given amount of time limited to a maximum of ten iterations of jobs, thus solving the problem of performance degradation.

4.3 Multi-site load distribution based scheduling

Since we're already tracking site performance, the other problem that we tackled was that of efficient distribution of work load to multiple sites of the grid, refer to the 4.2. Some of these sites could be error prone, some of them could be faster and some can be slower.

```
C1 -> Job propagation constant
multiSite -> list storing turnaround times of multiple sites.
low ->  min(multiSite)
high -> max(multiSite)
avg -> average(multiSite)
```

$$f(C1) = \begin{cases} C1 * 4 & multiSite_i = low \\ C1/2 & multiSite_i = high \\ C1 * 2 & multiSite < avg \end{cases}$$

Figure 4.2: Multi-site Job distribution function

We need to take an approach that improves the overall throughput of the workload, thus we keep track of average turnaround time for each batch of jobs submitted to the cluster. We take the average turnaround time of the batch of jobs that is submitted across multiple clusters. We group turnaround times of these batches of jobs based on the value of the turnaround time to be greater than, less than, or equal to the average turnaround time of all batch of jobs. This gives us a classification of sites that run faster, we exploit this information in our scheduler. To implement multi-site job submission and load balancing we start submitting one job to each cluster and measure the turnaround time once the job returns. Now we have the information of turnaround time of a particular resource of all the clusters. Continuing to submit jobs concurrently in a similar manner would result in the completion of the workload in a non efficient way. Out of the list of all the turnaround times of different sites we take the sites that have lower than average turnaround time of all batches of jobs and increase the job propagation factor for these jobs. This in-turn increases the number of jobs submitted to that site and helps us efficiently distribute the workload to faster clusters (submitting to a faster site would enable us to submit more jobs to the particular site. As the turnaround time is lower we would finish jobs quickly and make room for more jobs if we submit less to slower sites. These sites can still help us

improve the throughput to a greater extent by enabling us to prevent further degradation at those sites.) Error handling mechanism is gracefully handled by HTCondor and by this approach we have an efficient system with increased throughput and with proper distribution of load across multiple clusters.

The co-scheduler requires the presence of a HTCondor installation and libcondorapi. It is written in the C++ language and has extensively made use of pthreads for synchronization and multithreading. The co-scheduler has two components to it,the first one is a capacity detection algorithm which is found by measuring degradation and the other aspect to it is multi-site workload distribution algorithm .The following paragraphs detail the engineering aspects of the design. The program begins by taking the number of jobs, site's information and submit script information as its input. The number of jobs are the ones submitted across multiple sites. The next input file contains information on the list of sites that can be used for load balancing in the GridResource format of the HTCondor ClassAds API, e.g. *tusker-gw1.unl.edu/jobmanager-pbs*. It is assumed that all the sites listed in the sites input file are working and do not have any misconfiguration issues. The third and the final option to the scheduler is the submit description file. All the job ClassAd information and requirements can be written in this section and all of these will be applicable on the grid when a particular job is scheduled. The two main important attributes required in the submit description file is *universe* which needs to be grid all the time as we are submitting to grid sites, and the other most important thing is the *grid-proxy*. As we know that `condor_wait` [1]can be used to wait for a certain job or number of jobs to complete, it watches the log file and sees if the completion entry of the job is made in the log file that is generated by the *log* command in the HTCondor submit description file. There are two options we can specify to `condor_wait` one is `-num`, `number-of-jobs` that waits till the `number-of-jobs` are completed and the other option is to specify `wait`, seconds that waits for seconds amount of time. If nothing is specified

`condor_wait` waits indefinitely till the job(s) are complete.

4.4 Implementation of Capacity based scheduling

This algorithm detects degradation and once we detect degradation we find the optimal capacity:

Algorithm 1 Algorithm for determining optimal capacity by detecting degradation

```
c2 ← 1.25 {c2: Degradation factor}
while true do
  if T2 < c2*T1 then
    jobSubmission(k) {Submit k jobs}
    T1 = (T1 + T2)/2
  end if
  if T2 > c2*T1 then
    degradation_high ← k
    degradation_low ← k/2
    optimalCapacity(degradation_high, degradation_low)
  end if
end while
```

Algorithm 2 Algorithm for determining optimal capacity by detecting degradation

```
mid ← (high + low)/2
k ← mid
jobSubmission(k) {Submit k jobs}
if T2 < c2 * T1 then
  T1 ← (T1 + T2)/2
  optimalCapacity(mid,high);
end if
if T2 > c2 * T1 then
  optimalCapacity(low,mid);
end if
return mid
```

Algorithm 1 sets up the stage for degradation detection, it submits more jobs if the T2 is within permissible limits else the algorithm calls the optimal capacity algorithm that detects optimal capacity.

Algorithm 2 is passed two parameters high and low. These represent the upper bound where degradation has taken place and lower bound where degradation hasn't taken place. The algorithm works like a modified binary search and takes O(log n) iterations to find the optimal set of jobs.

4.5 Implementation of Multi-site load distribution based scheduling

In figure 4.3. Multi-site load distribution based scheduling uses the turnaround time at each site for sending future jobs, its a threaded algorithm, the following is extracted function per thread:

Algorithm 3 Algorithm for distribution of workflow load across multiple sites on the grid

$c1 \leftarrow 2$ {c1: Job propagation factor} {multiSite is a list storing turnaround times of multiple sites.}
$low = min(multiSite)$
$high = max(multiSite)$
average = $\sum_i multiSite_i$ / Sizeof($multiSite_i$)
if turnaroundTime_Thread == low **then**
  $c1 \leftarrow c1 * 4$
end if
if turnaroundTime_Thread == high **then**
  $c1 \leftarrow c1/2 > 1?(c1/2 : 1)$
end if
if turnaroundTime_Thread < average **then**
  $c1 \leftarrow c1 * 2$
end if

Algorithm 3 classifies different sites into faster and slower sites based on the performance of the sites bound by the turnaround time for the purpose of multi-site load distribution algorithm.

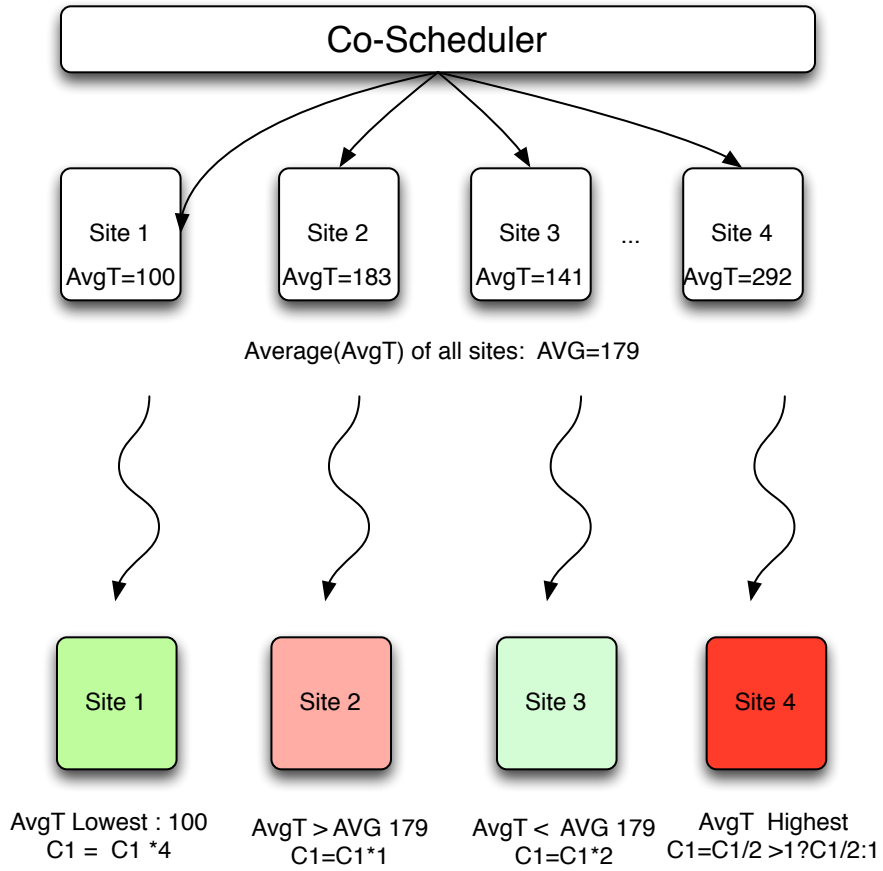

Figure 4.3: Multi-site scheduling algorithm overview, classification of sites into slower and faster

4.6 Programming APIs

4.6.1 HTCondor Log Reader and User API

HTCondor provides Job Log Reader API[1] that polls logs for job events by giving us API access to the events and outcomes. The following is the constructor for initializing a ReadUserLog object.

Constructor: ReadUserLog reader(fp,false,false);

`ULogEventOutcome (defined in condor_event.h):`

```
Status events for job detection:


ULOG_OK: Event is valid
ULOG_NO_EVENT: No event occurred (like EOF)
ULOG_RD_ERROR: Error reading log file
ULOG_MISSED_EVENT: Missed event
ULOG_UNK_ERROR: Unknown Error
```

All the entries in the log file end with . . . . All the job log entries are named as events and these events could range from being ULOG_OK where the event has taken place and is valid to ULOG_UNK_ERROR where an error has taken place.

The following pseudo-code is extracted from the logReader module that first detects all the valid events and then based on the data-structure of the event object detects if the event kind is ULOG_EXECUTE, meaning the job has begun executing, via eventNumber data member. Finally, we detect the ULOG_JOB_TERMINATED event where the job has successfully terminated. We also cast the more general event object into JobTerminatedEvent to access data members of the JobTerminatedEvent. The comments in the listed pseudo-code provides more details on this extracted code.

Job Submission Pseudo-Code:

```
1 void logReader(string hostFile, args *data, int nSites) {
2         FILE *fp;
3         ReadUserLog reader(fp,false,false);
4         ULogEvent *event = NULL;
5              while(reader.readEvent(event)==ULOG_OK)    {
6             if((*event).eventNumber==ULOG_EXECUTE ) {
```

```
7               //Cast into Execute Event
8                   ExecuteEvent *exec
9                   = static_cast<ExecuteEvent*>(event);
10                  //condor_wait -num K, where K is the
11                  //amount of jobs completed till the wait.
12                      char tmp[100];
13                      sprintf(tmp,"condor_wait -num
14                      \%d \%s",count,hostFile.c_str());
15              }
16              if((*event).eventNumber==ULOG_JOB_TERMINATED)   {
17              //Cast into Job Terminated Event
18                  JobTerminatedEvent *term
19                  = static_cast<JobTerminatedEvent*>(event);
20
21                  if(term->normal)    {
22                      //on Normal termination,
23                      //works only for local jobs,
24                      //find the CPU time of local jobs
25                  }
26              }
27          }
28  }
```

4.6.2 Synchronization Co-Scheduler Code

There are critical sections in the code where synchronization becomes absolutely necessary. One such variable is number-of-jobs. The number of jobs executed across the sites should remain fixed and the value must match the value that has been given as input. In a threaded system where each thread is executing jobs on a different cluster it becomes necessary to define `N`, the number of jobs executed or executing as a critical section. Here we define a pthread mutex and lock it for all write accesses to the `N` , we serialize the access to `N` and make conditional checks during job submission, so as not to allow job submissions when `N` is greater than the input value. By serializing the access across multiple threads the total jobs executed remains equal to the input number of jobs. The following chunk of code block demonstrates the use of mutexes for serialization of `N` among the threads.

```
1 pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
2 pthread_mutex_lock (&mymutex);
3 sumOfK+=k;
4 pthread_mutex_unlock (&mymutex);
```

Figure 4.4: Mutex on Number Of Jobs, sumOfK variable

Another section of code where synchronization becomes important is while distributing the jobs to multiple sites. When measuring which cluster is faster, we need information of turnaround time from all the sites before proceeding, this means all the threads need to be executing the same line of code before proceeding with the further program. Thus the absolute need for synchronization. To handle this problem we make use of conditional pthread variable. The following code demonstrates the use of conditional wait and conditional signal variable that clears the block on all the threads waiting based on the given condition.

```
pthread_mutex_t syncMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t synchronize_cv = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&syncMutex);
multiSite.push_back(stats[thread_id].T1);
tid_multiSite.insert(std::pair<int,int>
(data->tid,stats[thread_id].T1)
);

if ( multiSite.size() < numOfSites )
{
  pthread_cond_wait(&synchronize_cv, &syncMutex);
}
else
{
pthread_cond_broadcast(&synchronize_cv);
}
pthread_mutex_unlock(&syncMutex);
```

Figure 4.5: Synchronization of threads after reading turnaround time

We need to have the turnaround time of the first job, which enables us to measure how fast each cluster is. To do so, we need to wait for each job to complete on the first submission thereafter the jobs are submitted asynchronously and the average turnaround time is updated in the list. In this pseudocode the threads beginning first add the turnaround time to the list and conditionally wait if the size of the list is less than the number of sites. The last thread comes in and the same condition is voided and executes a conditional broadcast to unblock all the waiting threads and the scheduling proceeds further to asynchronously schedule further jobs. In this example of synchronization code there is a possibility that some of the threads wait continuously if the availability of the sites is very low. Currently we assume the existing availability of the cluster for the above code to take effect and reserve the function of deadlock prevention of some threads dude to bad sites for future work.

Chapter 5

Evaluation

5.1 Introduction

We evaluate the degradation detection algorithm by the distribution of turnaround time. The turnaround time of all the jobs when run using co-scheduler and the turnaround time of all the jobs when run directly on the cluster using the HTCondor-G grid interface were both investigated.

If a NFS server has the capacity to serve 50 clients without degradation it implies the turnaround time of all the jobs on the 50 clients are within the limits that would not be a degraded turnaround time. If the same NFS server is forced to serve 200 clients then we would see a degradation and would see an increase in the turnaround time of the jobs that are now contending for the I/O. This would result in degraded turnaround time and the same jobs would have to take lot more time to complete which is the usual scenario that occurs at Holland Computing Center when large amount of I/O bound jobs are submitted to the cluster and there is no check on the usage and the capacity of the I/O resource.

In this evaluation, we've submitted 375 jobs to a cluster tusker using bulk HTCondor-G submission and then submitting it through the co-scheduler so that we can keep track of degradation, detect it and prevent it. The submit scripts for the co-scheduler are generated dynamically and the grid universes are populated from the input sites file. The machine that is used for jobs submission is a HTCondor developmental virtual machine. That had to be configures for the grid submissions before its use for the experiment. 375 Jobs were run on the lustre filesystem and the results for this run in figure 5.1 is of the production level tusker. We were able to see degradation when we submitted jobs in bulk using HTCondor-G interface but we couldn't be sure if the degradation was the result of just our I/O based test job. Thus separate individual NFS servers were setup on both the clusters sandhills and tusker and tests were run again. The results of the tests are projected in the figure 5.5 and figure 5.7.

As stated above we're making use of histogram of binned turnaround time and good put graphs to study the effect of degradation. Ideally a histogram that projects degradation has many jobs that have higher turnaround times compared to a non degraded graph. A histogram that doesn't project degradation will have most of its jobs with the permissible limits of turnaround time and wouldn't have large variance in terms of the turnaround time. We also make use of goodput graphs that plot an area graph w.r.t the sorted list of turnaround time. It provides a good way to visualize the turnaround time. Ideally the area under the curve must be minimum in case of a non degraded graph but in a graph that projects degradation the area under the curve is larger than its non-degraded counterpart.

5.2 Evaluation of Co-Scheduler on Tusker cluster

Figure 5.1 shows how there exists a large variance in turnaround time by having a larger turnaround time than that present in figure 5.3. The former was the result of bulk submission whereas the later is the result of co-scheduler run. The associated goodput graphs in figure 5.2 and figure 5.4 show their respective areas with degraded run taking up larger area. When we look at the turnaround time distribution on independent NFS servers we can inter many things, that the degradation is lot significant in the figure 5.5 and the co-scheduler resolves degradation to a larger extent as noted in figure 5.7.

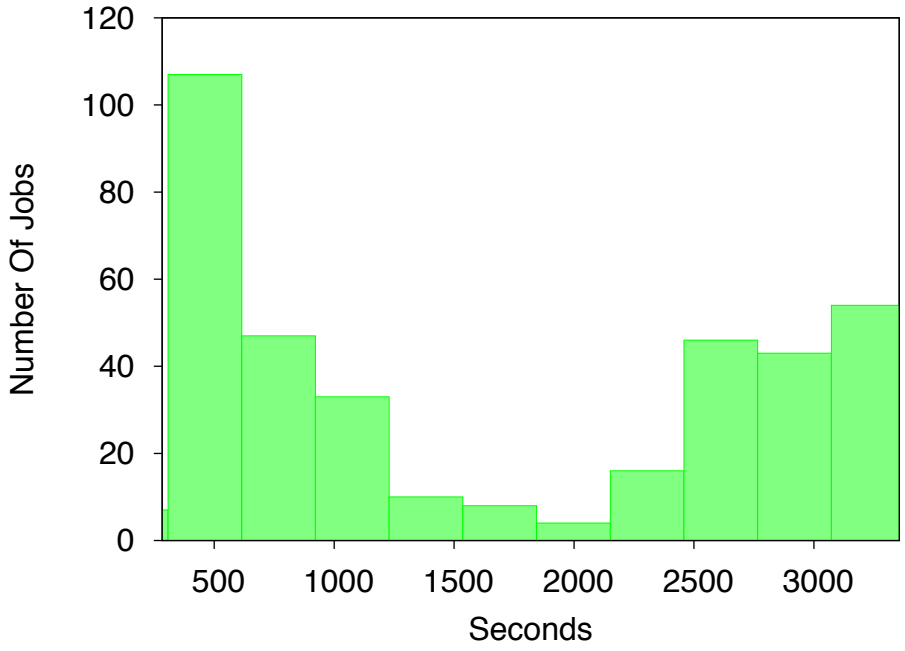

Figure 5.1: Histogram showing turnaround time distribution of 375 Jobs, when run on Tusker cluster

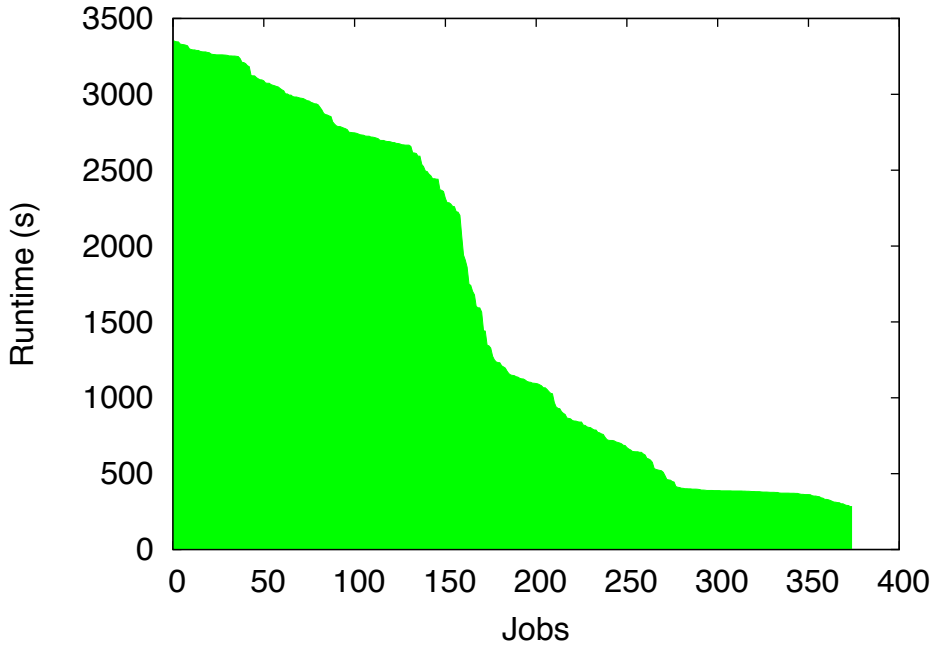

Figure 5.2: Goodput graph showing the turnaround time distribution of all the jobs submitted

5.3 Evaluation of Co-Scheduler on independent NFS server

The test I/O job used in this experiment continuously does disk write operations for a given period of time, the average time taken by the program being 300-500 seconds when run on different resources and generated approximately 3 GB of data file in that period. It was necessary to design our own test I/O program because of the need to change the path to the NFS servers where we'd be doing our benchmark to test the servers. The test I/O job takes two parameters one is the size N and the other is the path where the I/O must take place. There are two clusters under consideration, Sandhills and tusker. 375 jobs are run on each of the clusters and then a separate set of 375 jobs are scheduled using Co-Scheduler. We measure the extent of degradation on each cluster and quantitatively determine how degradation is handled by the Co-Scheduler.

In figure 5.1 considerable jobs take more than 2500 seconds to complete. When the resource is not degraded the test Job takes approximately 300-500 seconds of time.Tusker is currently running Lustre filesystem. If the capacity of the filesystem permitted to serve 375 jobs(or the number of jobs running concurrently) then most jobs would be completed within 500 seconds of time but that isn't the case with the tusker and hence we can conclude the resultant graph is the cause of degradation of disk I/O resource but there needs to be clarification about the source of the degradation. It could either be caused by the test I/O jobs that were submitted or can be caused by a set of jobs that are already running on the cluster. To clarify this issue and get a clear picture on whats causing the degradation the result from the figure 5.5 sheds more information.

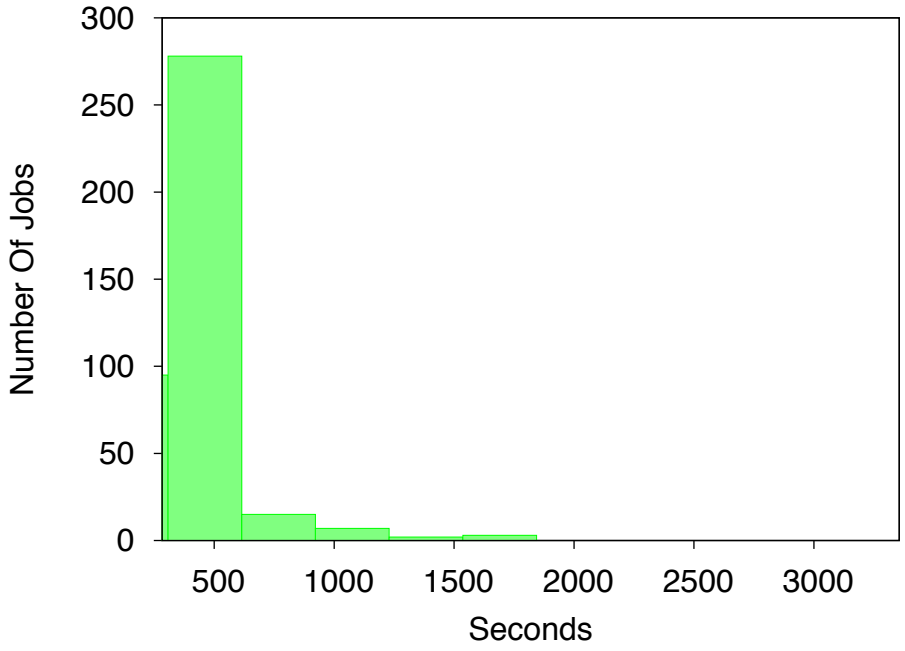

Figure 5.3: Histogram showing turnaround time distribution of 375 Jobs, when run on Tusker cluster

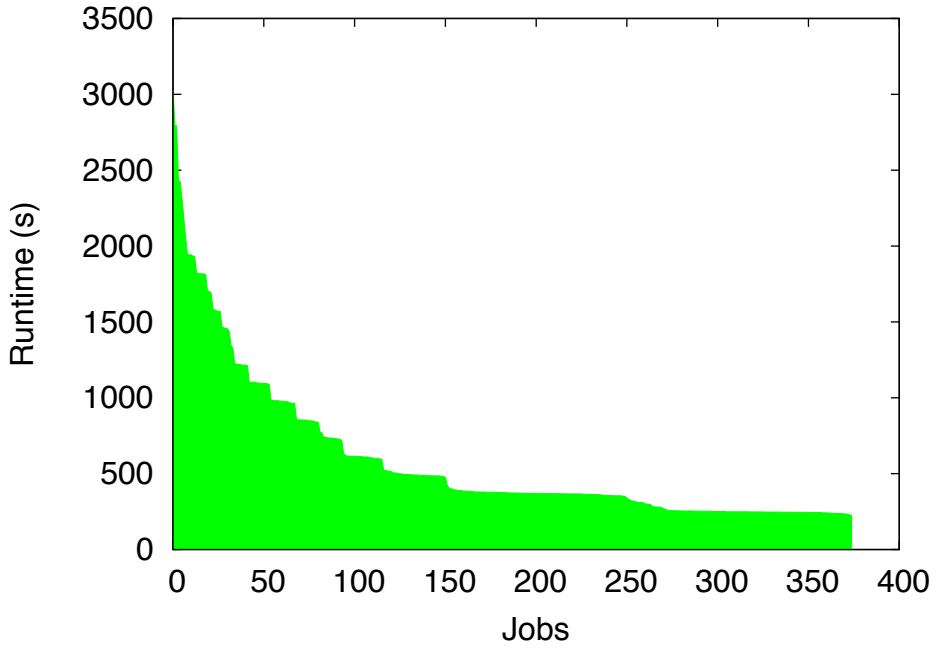

Figure 5.4: Goodput graph showing the turnaround time distribution of all the jobs submitted

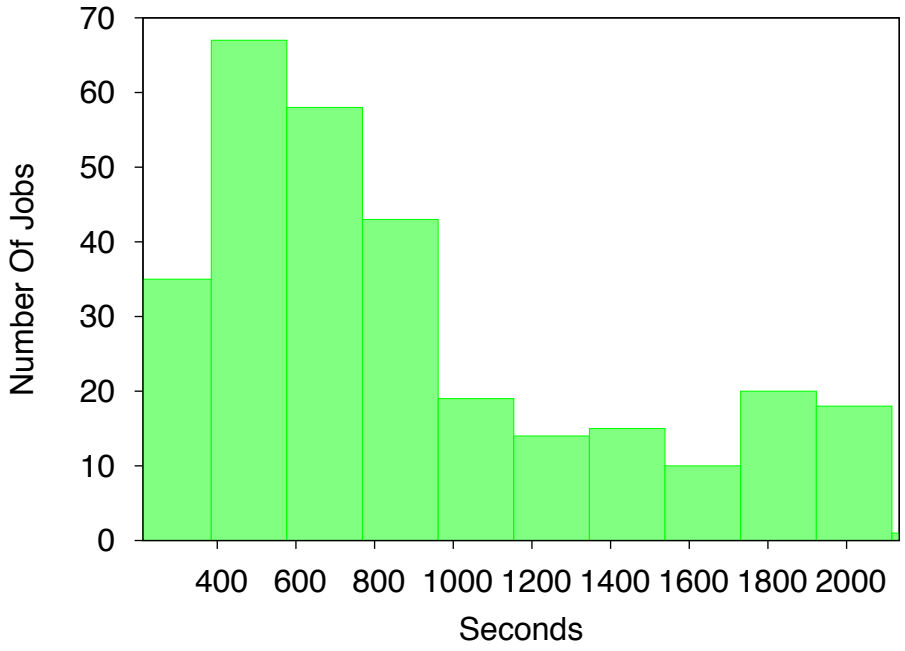

Figure 5.5: Histogram showing turnaround time distribution of 300 Jobs, when run using a coscheduler on a custom small scale independent NFS server

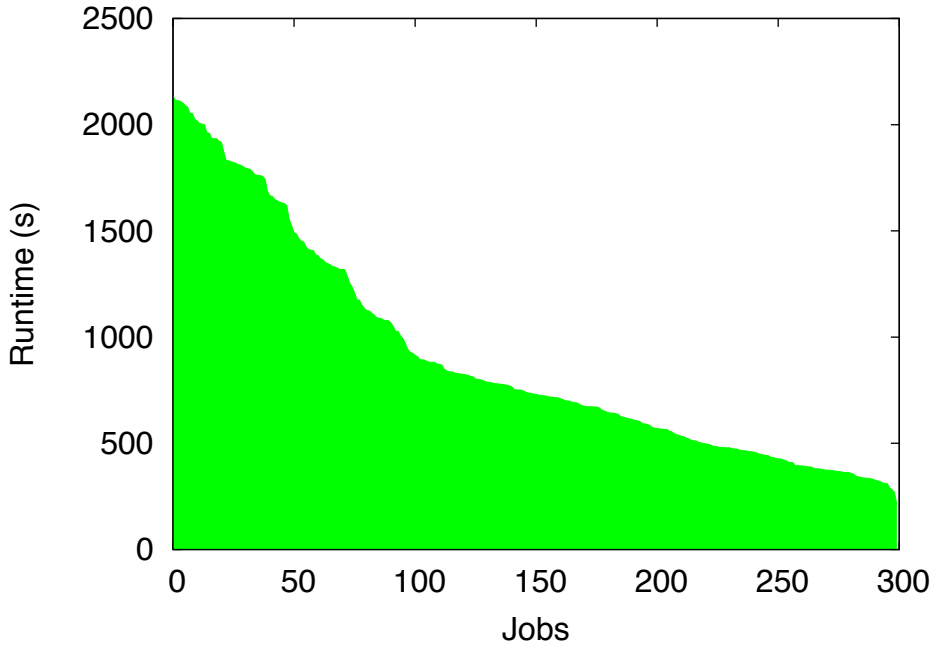

Figure 5.6: Goodput graph showing the turnaround time distribution of all the jobs submitted

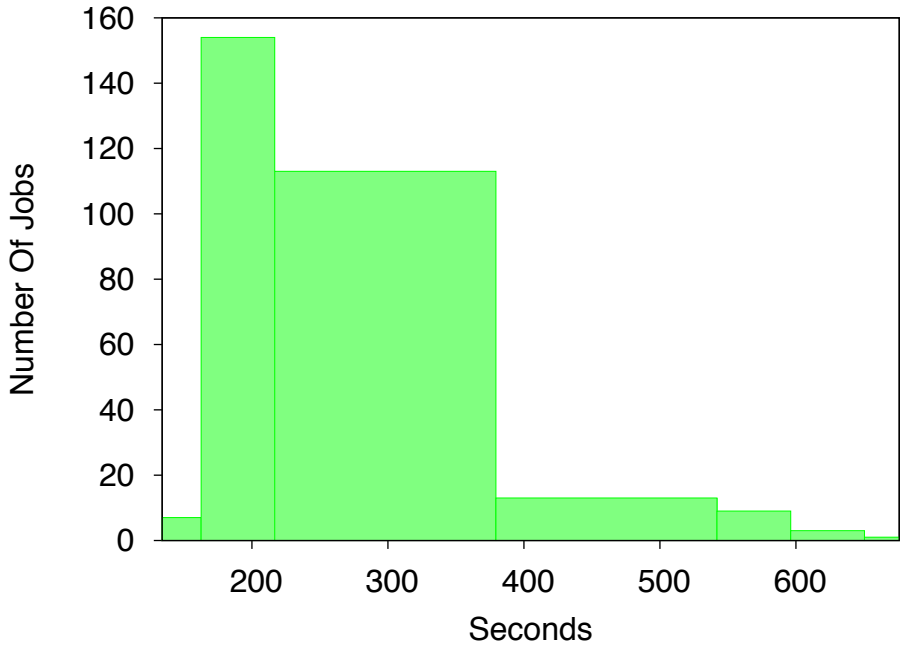

Figure 5.7: Histogram showing turnaround time distribution of 375 Jobs, when run using a coscheduler on a custom small scale independent NFS server

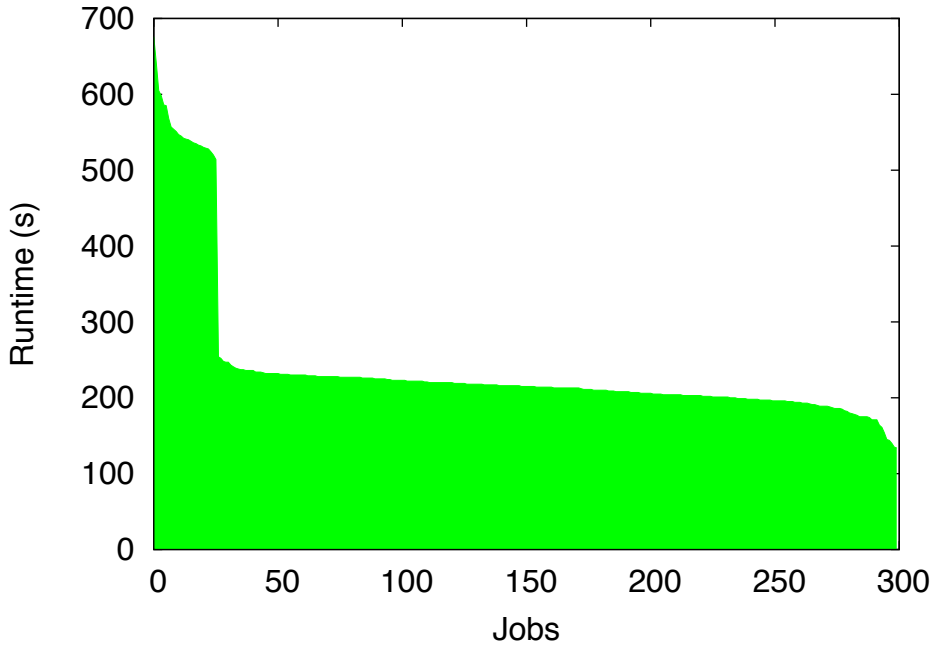

Figure 5.8: Goodput graph showing the turnaround time distribution of all the jobs submitted

In figure we can see most of the jobs take about 500 seconds of time to complete. We've prevented excessive degradation here and the I/O resources serve the concurrent jobs without a degraded turnaround time. This reduces the total time taken for the execution of this batch of jobs making way for the use of cluster for other jobs that might need computing.

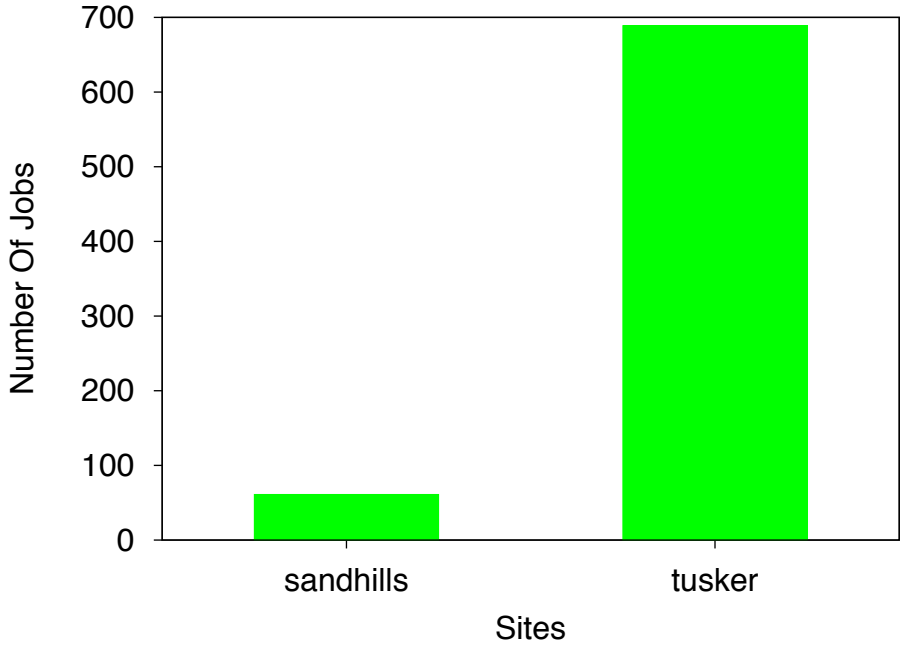

Figure 5.9: Bar plot showing load distribution

In figure 5.9, multi-Site load distribution algorithm distributes the job based on the turnaround time and load on the cluster, the following was the distribution of jobs when run on Co-Scheduler with 750 Jobs. The cluster with lower turnaround time will get more amount of jobs to run. The average turnaround time for the cluster tusker is 764 seconds and the average turnaround time for the cluster sandhills is about 413. The lower turnaround time of sandhills is because of the lower availability which can be noted by the throughput information. For tusker the throughput was 47.2 CPU time hours/elapsed time hours and for sandhills it was 3.12 CPU time hours/elapsed time hours.

5.4 Scalability Analysis of the Co-Scheduler

The Co-Scheduler is architected in a way so that each thread is responsible for dispatching jobs on a given site. The Co-Scheduler can potentially spawn thousands of threads and serve them efficiently but there are only hundreds of OSG sites to run on. Essentially it can scale to all the sites of the OSG.

Co-Scheduler can take large number of jobs as input, of the order of hundreds of thousands of jobs and dispatch the jobs efficiently with the load-balancing algorithm and ensures to run these jobs without incurring degradation.

Co-Scheduler is a robust program written in C++ that schedules based on turnaround time to detect degradation and schedules based on (queue wait time + run-time) for the sake of load balancing purpose.

Chapter 6

Conclusion

Co-Scheduling on the Grid is challenging with the presence of dynamically varying resources like Network, RAM, compute resources, all of which are available opportunistically. The current Co-Scheduler detects degradation and taxes degraded resources less in terms of scheduling jobs to the degraded resources. The capacity of the cluster is a varying quantity that determines the amount of concurrent jobs that may run without degradation. Co-Scheduler successfully finds the capacity of the cluster and maintains the capacity up-to 10 successive iterations of the scheduler before recalculating the capacity.

With the assumption of the availability of cluster(s)(which means further queue wait time is absent), the Co-Scheduler improves throughput of the job set which handles the case of degradation by virtue of a multi-site job distribution algorithm. Because Co-Scheduler directly effects the load on the resource, Co-Scheduler ensures the average overall turnaround time per job is lower.

Resource management is completely abstracted. We do not need to consider the allocation and management of disk I/O or RAM or Network I/O. The Co-Scheduler is driven by the fact that whenever a resource is overloaded it results in degradation and subsequently does not respond optimally or crashes. A simple and efficient way of

tracking degradation is implemented and further monitoring the load of resources on the cluster. As much as this is a positive aspect of the Co-Scheduler, the downside is there isn't a way to tell which of the resource is the cause of the degradation. Because the Co-Scheduler is used with large workloads its usually known based on the workload the resource that is under stress.

While Co-Scheduling offers multiple enhancements for the users of large workload there are some of the downsides of it. Co-Scheduler submits the jobs stepwise, in the sense that it submits a smaller set of jobs and waits for its outcome to decide upon the quantity of jobs on the next iteration. This nature of the Co-Scheduler can accumulate lot of queue wait time if the availability of the compute resources are lower. In the multi-site load distribution algorithm if one of the sites has very low availability the Co-Scheduler would accumulate the queue wait time of that particular site before it can complete its execution. The pre-emption of jobs and rescheduling to enhance such behavior is thus part of the future work.

Bibliography

[1] Condor manual, section 5.6. 2.1, 2.2, 3.1.2, 3.1.2, 4.3, 4.6.1

[2] Monika Choudhary and Sateesh Kumar Peddoju. Turnaround time based job scheduling algorithm in dynamic grid computing environment. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, pages 490–493, New York, NY, USA, 2012. ACM.

[3] David Cieslak Douglas Thain and Nitesh Chawla. Condor log analyzer. http://condorlog.cse.nd.edu, 2009.

[4] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12(1):53–65, 1996.

[5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115, 1997. 2.4

[6] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, August 2001. 1

[7] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[8] Cluster Resources Inc. Cluster resources :: Products - TORQUE Resource Manager:, Jaunary 2011. http://www.clusterresources.com/pages/products/torque-resource-manager.php. 1

[9] Ashu Guru David Swanson Kartik Vedalaveni, Cory Stroupe. Architecting indel-sequence-generator for htcondor. TeraGrid, September 2011. 2.2

[10] Miron Livny and Rajesh Raman. High-throughput resource management. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[11] LLNL. Slurm reference.

[12] Holland Computing Center. Holland computing center, January 2011. http://hcc.unl.edu/.

[13] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, et al. The Open Science Grid. In *Journal of Physics: Conference Series*, volume 78, page 012057. IOP Publishing, 2007. 2.5

[14] I. Sfiligoi. glideinWMS–a generic pilot-based workload management system. In *Journal of Physics: Conference Series*, volume 119, page 062044. IOP Publishing, 2008.

[15] J.V. Sumanth, D.R. Swanson, and Hong Jiang. Adaptive load balancing for long-range md simulations in a distributed environment. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 135–146, 2006.

[16] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[17] Derek Weitzel. Campus grids: A framework to facilitate resource sharing, 2012. 1, 3.1.1

[18] M. Zvada, D. Benjamin, and I. Sfiligoi. CDF GlideinWMS usage in Grid computing of high energy physics. In *Journal of Physics: Conference Series*, volume 219, page 062031. IOP Publishing, 2010.